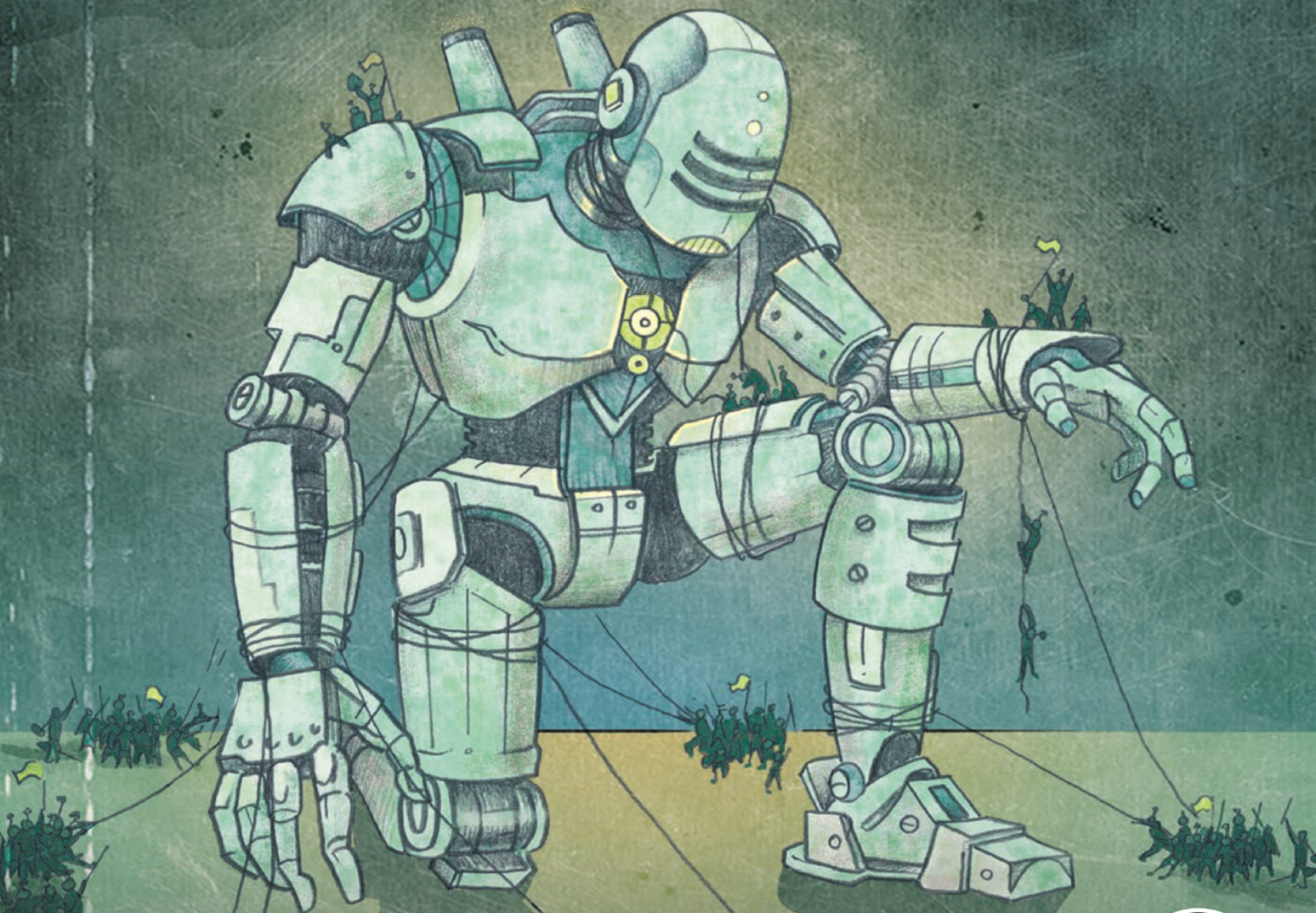
Fuzzing Android

Finding Vulnerabilities in Userspace and the Kernel



Hamzeh Zawawy, PhD; Eugene Rodionov, PhD; Xuan Xing;
Kris Alder; Cory Barker; and Steven Moreland

no starch press®

Fuzzing Android

Finding Vulnerabilities in Userspace and the Kernel



Hamzeh Zawawy, PhD; Eugene Rodionov, PhD; Xuan Xing;
Kris Alder; Cory Barker; and Steven Moreland

no starch press®

OceanofPDF.com

PRAISE FOR *FUZZING ANDROID*

"If you need to fuzz components in Android or Linux, or if you want to understand how vulnerability research operates, this book delivers."

— DAVID KLEIDERMACHER
VICE PRESIDENT, PLATFORMS SECURITY AND PRIVACY, GOOGLE

"An end-to-end exploration into how to apply bleeding-edge software security tools to a juicy target."

— MATHIAS PAYER, PROFESSOR OF COMPUTER AND COMMUNICATION SCIENCES, EPFL; HEAD, HEXHIVE LABORATORY

"I recommend [this book] to anyone, at any level, who is interested in getting started in this area."

— KIM TAESOO, PROFESSOR, SCHOOL OF COMPUTER SCIENCE, GEORGIA INSTITUTE OF TECHNOLOGY

OceanofPDF.com

FUZZING ANDROID

Finding Vulnerabilities in Userspace and the Kernel

by Hamzeh Zawawy, Eugene Rodionov, Xuan Xing, Kris Alder, Cory Barker, and Steven Moreland



no starch press®

San Francisco

OceanofPDF.com

FUZZING ANDROID. Copyright © 2027 by Hamzeh Zawawy, Eugene Rodionov, Xuan Xing, Kris Alder, Cory Barker, and Steven Moreland.

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.

First printing

30 29 28 27 26 1 2 3 4 5

ISBN-13: 978-1-7185-0529-2 (print)
ISBN-13: 978-1-7185-0530-8 (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
Production Manager: Sabrina Plomitallo-González
Production Editor: Miles Bond
Developmental Editor: Frances Saux
Cover Illustrator: Rick Reese
Interior Design: Octopod Studios
Technical Reviewer: Dominik Maier
Copyeditor: Sharon Wilkey
Proofreader: Scout Festa
Indexer: Sanjiv Kumar Sinha

Library of Congress Control Number: 2026037108

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. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

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 authors 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.

OceanofPDF.com

About the Authors

Hamzeh Zawawy, PhD, is a software engineering manager at Google who leads the Android Blue Team, focusing on systemic platform hardening and proactive security mitigations. He has a passion for software data analysis, having contributed to academic books on mining software logs, and oversees defensive engineering efforts across the Android ecosystem.

Eugene Rodionov, PhD, is a security researcher and the Device and Platform Security tech lead at Google. He is a co-author of *Rootkits and Bootkits* (No Starch Press) and specializes in reverse engineering, vulnerability analysis, and low-level firmware security. He regularly presents his research on GPU and Binder exploitation at major conferences like Black Hat and DEF CON.

Xuan Xing leads the Android Red Team at Google. He is passionate about fuzzing for security research. For the past several years, Xuan has focused on finding security vulnerabilities on Android/Pixel devices, using fuzzing as the primary approach.

Kris Alder is a graduate of both Hacker Middle School and Brigham Young University. He works as a software engineer at Google. When he exits Vim, he enjoys skiing, archery, and board games.

Cory Barker is a software engineer at Google specializing in Android platform security and fuzzing. He is dedicated to leveraging artificial intelligence to enhance security across the Android ecosystem.

Steven Moreland has worked on Android architecture for nearly a decade, reinventing inter-process communication in Android, unifying the two interface description languages used in Android, and leading a team that fuzzed all C++ services in Android, among other things.

OceanofPDF.com

About the Technical Reviewer

Dominik Maier, PhD, is a security researcher on the Android Red Team at Google. He is a maintainer of the popular open source fuzzing frameworks AFL++ and LibAFL, and has co-authored several academic papers on fuzzing and baseband security.

OceanofPDF.com

BRIEF CONTENTS

Foreword
Acknowledgments
Introduction

PART I: FOUNDATIONS

Chapter 1: Introduction to Fuzzing
Chapter 2: Preparing Android for Fuzzing
Chapter 3: Tools and Frameworks
Chapter 4: An Overview of Android Security

PART II: FUZZING IN PRACTICE

Chapter 5: Choosing Your Targets
Chapter 6: Writing Your First Android Fuzz Harness
Chapter 7: Crash Analysis and Bug Triage

PART III: ADVANCED TARGETS

Chapter 8: System Services
Chapter 9: The Android Kernel
Chapter 10: Fuzzing Custom Kernel Drivers
Chapter 11: Host Environments

Epilogue: Fuzzing in the Age of AI
Index

OceanofPDF.com

CONTENTS IN DETAIL

FOREWORD

ACKNOWLEDGMENTS

INTRODUCTION

Who This Book Is For
What's in This Book?
Requirements

PART I
FOUNDATIONS

1
INTRODUCTION TO FUZZING

Why Fuzz?
The Fuzzing Process
    Generating Random Inputs
    Exercising the Code Under Test
    Evaluating Progress
    Detecting Bad Behaviors
Fuzzing vs. Other Testing Types
    Unit Testing
    Integration Testing
    Fuzz Testing
Wrapping Up

2
PREPARING ANDROID FOR FUZZING

Downloading Android
Setting Up Your Device
    Physical Devices
    Emulated Devices
The Soong Build System
    Defining Build Targets
    Building a Fuzz Target
    Setting Build-Time Options
Installing ADB
Testing the Fuzzing Environment
    With Cuttlefish
    With a Physical Device
    On Your Host Machine
    With AFL++
Wrapping Up

3
TOOLS AND FRAMEWORKS

Sanitizers
Fuzzing Engines
    libFuzzer
    AFL++
    LibAFL
    Syzkaller
Data Parsing
    Manual Parsing
    With the FuzzedDataProvider Library
    With the libprotobuf-mutator Library
Automating the Fuzzing Life Cycle
Analyzing Lines Covered

Wrapping Up

4
AN OVERVIEW OF ANDROID SECURITY

Layers of the Android Architecture
Android's Security Evolution
Hardware-Based Security
    Trusted Execution Environments
    Secure Elements
    Protected Virtual Machines
Fragmentation
    The Chinese Android Ecosystem
    Project Treble
Permissions
    How Apps Request Permissions
    Permission Escalation Attacks
SELinux
    How SELinux Works
    Policy Structure
    Tools
    Shortcomings
App Signing
Wrapping Up

PART II
FUZZING IN PRACTICE

5
CHOOSING YOUR TARGETS

Library Evaluation Metrics
    The Number of Dependent Libraries

Reachability via Android Public APIs
The Days Since the First and Last Commit
The Number of Distinct Contributors
The Number of Lines Changed and Inserted
The Vulnerability History
Whether the Library Runs in Privileged Process
Calculating a Criticality Score
Determining a Library's Critical Components
Measuring Complexity with Lizard
Finding Dangerous Operations
Before You Begin Fuzzing
Wrapping Up

6
WRITING YOUR FIRST ANDROID FUZZ HARNESS

Fuzzing a Library
Creating an Empty Fuzz Harness
Running the Harness
Exercising the Library Code
Disabling Error Logging
Understanding the Fuzzer's Output
Fixing Memory Leaks
Improving Fuzzer Performance
Adding a Dictionary
Collecting a Corpus
Adding a Seed Corpus
Wrapping Up

7
CRASH ANALYSIS AND BUG TRIAGE

Initial Crash Validation

Do We Have Enough Information About the Bug?
Can We Reproduce the Issue?
Is It a Legitimate Bug?
Crash De-Duplication
Making a Severity Assessment
Behavior
Privilege Level
Attack Vector
The Final Determination
Wrapping Up

PART III
ADVANCED TARGETS

8
SYSTEM SERVICES

A Blueprint for Fuzzing Large Systems
Challenges of Black-Box Fuzzing
API Contract Enforcement
Fuzzing Binder Services
Fuzzing Parcel Objects
Automating a Service Fuzzer
Fuzzing Multiple Services
Optimizing Coverage: A Binder-Specific Example
Considering Structure-Aware Fuzzing
The Human Element of Large-Scale Fuzzing
Developing a Strategy for Scaling
Testing the Fuzzer Framework
Wrapping Up

9

THE ANDROID KERNEL

Why Kernel Fuzzing Requires a Different Approach
Preparing a Kernel for Fuzzing
    Downloading and Building Kernel Code
    Turning on Fuzzing Instrumentation
Syzkaller
    Setting Up
    Getting to Know Syzkaller
    Writing a Syzkaller Configuration
    Fuzzing Inside a Sandboxed Environment
Linux Kernel Library
    The Structure of an LKL-Based Fuzzer
    Example: Fuzzing the Human Interface Device
    The High-Level Workflow for Constructing an LKL Fuzzer
    Syzkaller vs. LKL
Wrapping Up

10
FUZZING CUSTOM KERNEL DRIVERS

Introducing a New Driver
    Adding a Driver to the Kernel
    Implementing the Lockbox
    Implementing the Padlock
Writing the Description
    Describing the Lockbox
    Describing the Padlock
    Extracting Constants
Fuzzing Your New Description
    Focusing on New Descriptions
    Introducing a Bug
    Running a Reproducer
Wrapping Up

11
HOST ENVIRONMENTS

What Counts as a Host?
Why Fuzz Android on the Host?
Challenges
Enabling Host Fuzzing
    Turning On the Flag
    Enabling the Host for All Dependencies
    Mocking Platform-Specific Components
Host Fuzzing Beyond Android
Wrapping Up

EPILOGUE: FUZZING IN THE AGE OF AI

Fuzzing AI Models
Automating the Fuzzing Process
Solving Fuzzing's Challenges
    With Gray-Box and White-Box Fuzzing
    With LLM-Generated Inputs
    By Automating Structure-Aware Mutation
Auto-Triage
    Crash De-Duplication
    Root Cause Analysis
    Remediation
Wrapping Up

INDEX

OceanofPDF.com

FOREWORD

How does one secure a modern mobile operating system at global scale? To start, one should implement secure architectural designs, including sand-boxing, encryption, and the enforcement of information flow control policies. But platforms like Android are exceptionally complex, relying on a large, trusted computing base that naturally creates a large attack surface.

Attackers constantly probe this surface to find memory-safety vulnerabilities that bypass architectural protections. So, memory-safety hardening is also vital, and Android has led the industry in transitioning to memory-safe systems software. For example, we mitigate memory vulnerabilities at their source by using Rust for low-level code, rather than C and C++. But the Linux kernel, low-level drivers, unsafe code blocks, and complex legacy components all present a challenge for Rust retrofits.

Scaled fuzzing addresses this reality by automating the discovery of vulnerabilities prior to shipping the code, effectively turning the relentless scaling of computation to the defender's advantage. The authors of this text know the challenge of fuzzing large-scale software projects intimately; Hamzeh, Eugene, Xuan, Kris, Cory, and Steven have spent years securing Android, building the automated testing infrastructure that protects the platform today.

Fuzzing Android is written for developers and security practitioners looking to master this practice and incorporate it effectively into human as well as agentic security workflows. If you need to fuzz components in Android or Linux, or if you want to understand how vulnerability research operates, this book delivers. It should teach any reader with a programming foundation how to write effective harnesses and even execute kernel fuzzing with tools like Syzkaller, providing the methodologies required to secure complex systems and ensuring that the next generation of mobile platforms remains safe for billions of users.

David Kleidermacher
Vice President, Platforms Security and Privacy, Google

OceanofPDF.com

ACKNOWLEDGMENTS

We are profoundly grateful to Dave Kleidermacher, who supported this project from the very beginning, and to Dominik Maier, who did an outstanding job as our technical reviewer. We also want to recognize Abhishek Arya, Oliver Chang, and Jonathan Metzman. Their work was foundational in launching and scaling Google's fuzzing efforts from inception; while our paths diverged for the writing of this manuscript, this book simply would not exist without the groundwork they laid.

To Pawan Wagh, thank you for all your work expanding binder fuzzing in Android and taking system fuzzing to the next level. Frederick Mayle and Devin Moore, thank you for your work solving some of the trickiest vulnerabilities here. Sandeep Patil, thank you for your help and support in building cross-functional fuzzing across teams into our organization.

To Martijn Coenen, thank you for showing Steven Moreland where to look for bugs in Binder in the first place.

Jon Bottarini, thank you for championing fuzzing and helping to manage the fuzzing project over the years.

Special thanks to Scott Roberts, whose leadership and vision as director were instrumental in backing and launching Android's fuzzing initiative from the start.

Fuzzing Android represents hundreds of projects over a period of more than a decade, with more work being done all the time. To the countless others who were involved, built infrastructure, or worked on this enormous project, not only at Google, but across hundreds of companies around the world, thank you.

We would also like to honor the memory of Matt Alexander, a brilliant former colleague whose conversations first inspired the concepts behind the Linux Kernel Library fuzzing explored in this book.

Writing a book, especially one with six authors, is a feat of coordination. To our editor, Frances Saux, thank you for helping make sense of all our manuscripts. This would not have been possible without your vigilance. To Bill Pollock, thank you for taking the time to be directly involved with this book.

Steven Moreland would also like to thank his lovely fiancée, Dani Nghi, for tolerating his loud and incessant typing during the creation of this book.

Kris Alder thanks his children, Charlie, Daniela, and Ian, for being effective sources of entropy in life and exhibiting the most exciting, unexpected behaviors.

Xuan Xing expresses his deepest gratitude to his wife, Rong, and his children, Roy and Iris, for graciously letting him borrow family weekends to finish this book.

Cory Barker thanks his wonderful wife, Tiffany, and his two amazing kids, Perri Pongo and Jackie Danger.

Finally, Hamzeh Zawawy expresses his deepest gratitude to his wife, Cheryl, for her support and patience throughout the long journey of bringing this book to life. To his sons, Amir and Yusuf, thank you for bringing so much joy to his days and for reminding him of what matters most.

OceanofPDF.com

INTRODUCTION



Computation has become remarkably effective at maximizing the value of data in every human endeavor. It has also become a superpower for computer attackers. The complexity and scale of attacks conducted today are incomprehensibly large when compared to every conflict that has ever existed.

We see this in the industry; we see this reported in popular media; and when we go home and look at our website logs, we see a hundred analogues of every attack that has made the news over the past 10 years. Attacks are getting more sophisticated, and they are happening continuously at a global scale.

From script kiddies to nation-states, attackers are hitting a bunch of buttons, turning every knob, and touching every surface on every system —whether it's connected to the web, emits electromagnetic radiation, communicates over audio, or otherwise interacts with its environment— looking for a way in.

That's why fuzzing and the techniques we demonstrate here matter so much. If you understand fuzzing, you'll know how attackers think, what we're up against, and how to modularize software. Most importantly, fuzzing is a way of using the curve of computation in your favor.

The authors of this book have fuzzed Android for the past decade, with hundreds of vulnerabilities discovered under our collective belts. We have witnessed multiple revolutions in the way security bugs are found and resolved. We have fixed them at scale, and we have gone back to redesign the operating system to make it easier to find vulnerabilities and to secure it.

If you design software with fuzzing in mind, it inevitably gets better as compute scales and as fuzzing engines improve. If you write a fuzzer today, then later get more compute, you'll get more out of your fuzzer. This book explains how to do that.

This book also talks about the security challenges of a large software environment. Fuzzing is not only about security; the techniques in this book are also useful for discovering correctness bugs or inconsistencies.

Who This Book Is For

If you're looking to fuzz programs on Android or Linux, you're looking to understand what fuzzing is, or you just want to dig deep into a major security practice used in software development today, then you'll find this book relevant. To get the most use out of this book, you should know how to program in a major programming language.

Also, beware. If you love programming, fuzzing is one of those experiences that feels too rewarding when you first learn about it. Make sure to stop fuzzing periodically and make time for meals, sleep, your family, and other activities that together form a healthy life. We absolutely love fuzzing, not only because of the security vulnerabilities we find, but because it lets you take apart and inspect a program in a way that almost nothing else does. A well-written fuzzer is like having a friend looking over your shoulder, pointing out where your program became inconsistent so you can fix it. We hope that you'll find this joy too.

What's in This Book?

This book has been a labor of love, not only in its writing, but in the years of security research that it covers. We'll get you started with simple, concrete examples taken from real Android code. We'll also show more advanced techniques and point to practices you may want to try to further grow your fuzzing skills. Here's an outline of the book's content.

Part I: Foundations introduces the concept of fuzzing and the tools and environment you'll need to fuzz Android.

Chapter 1: Introduction to Fuzzing Reviews the basics of fuzzing and fuzzing engines. We cover the core motivation and concept of fuzzing, how it compares to other security and testing techniques, and the types of problems it can solve.

Chapter 2: Preparing Android for Fuzzing Provides an overview of the Android environment and how to set it up for fuzzing. You should follow these setup steps so that you can work through the examples in this book. Understanding how to set up the environment is useful even if you're fuzzing a different system.

Chapter 3: Tools and Frameworks Reviews some of the most common fuzzing tools, including sanitizers that help detect memory errors and undefined behavior, as well as fuzzing engines themselves.

Chapter 4: An Overview of Android Security Reviews the history of security in Android and the role that fuzzers play. The design and structure of fuzzers necessarily mirrors the security architecture of the whole system. So, learning about Android's security features will help you understand why certain types of fuzzers exist in Android and what their relative values are.

Part II: Fuzzing In Practice covers Android's security model and the process of fuzzing its components.

Chapter 5: Choosing Your Targets Focuses on which targets to fuzz, especially if you're not already familiar with a codebase. This is particularly valuable if you work on a security team for a codebase or are trying to secure a legacy codebase. Even if you're fuzzing a codebase that you wrote entirely, you'll find that the anecdotes and red flags described here will help you form a great mental checklist for where to start.

Chapter 6: Writing Your First Android Fuzz Harness Presents a hands-on tutorial for writing a fuzzer in Android. Covers various work-flows for creating a fuzzer and augmenting its abilities with more data.

Chapter 7: Crash Analysis and Bug Triage Discusses what to do when a fuzzer finds crashes and how to categorize or prioritize fixing them.

Now that you understand the steps involved in fuzzing Android components, **Part III: Advanced Targets** covers the fuzzing of Android features with unique requirements, as well as other considerations for improving your fuzzing workflow.

Chapter 8: System Services Covers the fuzzing of Android's system services, touching on how to think about fuzzing large systems. The chapter covers both the technical architecture for doing so and how this practice fits into an organization.

Chapter 9: The Android Kernel Provides details on fuzzing Android's kernel, which is based on the Linux kernel, to find bugs in the syscall layer.

Chapter 10: Fuzzing Custom Kernel Drivers Focuses on Google's coverage-guided kernel fuzzer, Syzkaller, including writing custom syscall descriptions in Syzlang, building and configuring virtualized Cuttlefish instances, and triaging kernel panics and crashes.

Chapter 11: Host Environments Covers host-environment fuzzing. We discuss how to configure build rules in Soong to support host-side testing, isolate key Android libraries from device dependencies, and use mocking frameworks to maximize fuzzer performance on a host.

Epilogue: Fuzzing in the Age of AI Concludes the book with an overview of future trends in fuzzing, specifically focusing on the integration of artificial intelligence. We discuss using large language models (LLMs) to automatically generate fuzz inputs, triage crashes, and synthesize bug patches.

This book primarily focuses on C++ fuzzing for two reasons: Most Android vulnerabilities are found in C++ code, and Android's fuzzing tooling is most mature for this language.

We focus on fuzzing the standard Android Open Source Project (AOSP), including userspace components and the Linux kernel. Fuzzing code in a Trusted Execution Environment (TEE) is outside the scope of

this text. Designing programs around standard interfaces, such as Android Interface Definition Language (AIDL), a text stream, or anything that works universally, can make it easier to fuzz components from TEEs or non-Android chips using standard Android fuzzing harnesses.

Requirements

Android fuzzing typically uses a Linux-based programming environment, and most of the examples assume you're using Linux. However, at a high level, all techniques here can be done in all operating systems and all programming language environments. If you want to use a different setup, we recommend looking up how fuzzers are set up in your preferred environment to try out the techniques discussed in this book.

OceanofPDF.com

PART I
FOUNDATIONS

OceanofPDF.com

1
INTRODUCTION TO FUZZING



Fuzz testing, often called *fuzzing*, is a technique for finding bugs in many types of software, ranging from web applications and open source libraries to device firmware and operating system kernels. In short, fuzzing automatically feeds random inputs into a program or library to make sure it correctly handles unanticipated inputs or interactions.

A large volume of academic research exists on fuzzing, and programmers with substantial fuzzing experience can spend lots of time debating the pros and cons of practically any facet of the technique. On top of that, the field's jargon and terminology can make it seem daunting to get started. However, getting hands-on experience can be one of the best ways to learn.

In this chapter, we'll walk through the general steps involved in fuzzing any piece of software, touching on fuzzing subtypes and related concepts along the way. Then we'll compare fuzzing to other software testing methodologies so you can understand the pros and cons of each.

Why Fuzz?

The simplest answer to the question of why you should be interested in fuzzing Android (or anything else) is that software has bugs—lots of them. If you've written code with someone else as part of a class, a job, or an open source project, you'll know that understanding and checking someone else's code can be even more difficult than working with your own. And a typical Android device might have tens of millions of lines of code written by thousands of people. That code can interact in surprising ways that no one can fully understand, so specifying every single correct behavior, then verifying that the code matches it perfectly, is an impossible task.

Fuzzing is a powerful way of finding some of the trickiest bugs because it allows developers to use automated tools to find behaviors they hadn't even anticipated. Another advantage is that once you've set up your environment and written a fuzz test, you can leave it running for a long time unsupervised to find new, interesting bugs on its own. This could be as simple as running tests on your lunch break or as complex as building out infrastructure to fuzz in data centers around the clock.

Probably the most common use of fuzzing is to find security bugs that programmers miss and hackers like to exploit. Some prominent public fuzzing efforts, such as OSS-Fuzz, have reported finding tens of thousands of bugs, many of which are security vulnerabilities, and well-known security incidents such as Heartbleed, Shellshock, and Stagefright likely could have been detected earlier via fuzzing.

The Fuzzing Process

A variety of tools can be used for fuzzing across many domains and programming languages, with more popping up all the time. But at its core, fuzzing comprises four steps:

1. Produce random inputs.
2. Feed the inputs to the code under test.
3. Observe how the code behaves in order to mutate future inputs more intelligently.

4. Detect possible bugs to analyze and fix.

While each fuzzing tool has a different feature set and focus, all have at least some of these elements. Let's go over each part of this definition and explore some examples of what fuzzing might look like. When we cover specific examples of fuzzing Android code in the following chapters, you'll see how the tools implement these steps.

Generating Random Inputs

Generating and using random inputs is at the heart of fuzzing. Perhaps the simplest method of generating this randomness is reading inputs directly from */dev/random* or a programming language's standard library —for example, with C's `rand()` function. Although this is an easy place to start, most fuzzing tools will use a more sophisticated approach. For example, fuzzing often needs to be repeatable, which means using */dev/random* is not a good choice. Instead, we would rather start from certain seed inputs. In these cases, specifically designed mutation generators can be more useful than the standard `rand()` function.

Many tools support starting with an empty (zero-length) input or a value of 0, then continually mutating it to produce new inputs. These types of fuzzing engines are often called *mutation-based fuzzers*. You can learn about various types of mutation by reading the documentation or source code of open source fuzzing tools. These mutations could be as simple as adding or removing bytes, flipping bits, or combining two previous inputs.

At the next level up in complexity are tools that mutate inputs based on a specified language. For example, these tools might produce valid programming language code, Extensible Markup Language (XML), or something else according to a predetermined structure. This technique is commonly known as *grammar-based*, *structure-aware*, or *generative* fuzzing.

Currently, mutation-based fuzzers are the most popular, primarily because they circumvent the significant effort required to understand target areas and to create bespoke generators. However, randomly generating values can be less efficient at triggering specific issues. For

instance, consider the following C++ code, which contains a null pointer dereference because a function pointer in a map wasn't initialized:

```
#include <cstdint>
#include <cstddef>
#include <string>
#include <iostream>
#include <functional>
#include <map>

// Define a few functions that all have the same signature.
int add(int a, int b) {return a + b;}
int sub(int a, int b) {return a - b;}
int mul(int a, int b) {return a * b;}
using Operation = int (*)(int, int);

std::map<std::string, Operation> operations;
void init() {
    operations["add"] = &add;
    operations["sub"] = &sub;



// The bug
    operations["multiply"] = nullptr; //&mul;
}

int invoke(const std::string& action, int arg1, int arg2) {
    if (operations.count(action)) {
        auto fn = operations[action];
        return fn(arg1, arg2);
    } else {
        return -1;
    }
}

#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION
extern "C"
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
```

```
    init();
    std::string action((const char*)Data, Size);
    invoke(action, 1, 2);
    return 0;
}
#else
int main(int argc, char* argv[]) {
    if (argc < 4) {return -1;}

    init();
    std::string action(argv[1]);
    int arg1 = atoi(argv[2]);
    int arg2 = atoi(argv[3]);

    int result = invoke(action, arg1, arg2);
    std::cout << action << "(" << arg1 <<", " << arg2 <<") = " << result <<
std::endl;
    return result;
}
#endif
```

In this code, a developer forgot to assign the `mul()` function to the map, causing a null pointer dereference. This can be difficult for purely random mutation-based fuzzing to discover because to trigger this particular issue, the fuzzing engine needs to generate the exact key phrase. If this generation were attempted purely randomly, uncovering this null pointer reference issue would be practically impossible.

Modern mutation-based fuzzing frameworks tackle this problem by employing techniques to recognize and prioritize "special" values. For example, libFuzzer uses a feature called `valueprofile`, while LibAFL uses `cmpcov` for similar mechanisms. These features instrument the target program to capture and analyze specific values used in comparison instructions. Unfortunately, this capability often requires special support from the language or compilation toolchain and is not universally available for every programming language.

Similar challenges arise when the program under test derives specific values cryptographically or through complex computations on an input.

Fuzzers often mutate each byte individually, but if your target requires a value that is cryptographically derived such that changing a single bit completely alters the result, fuzzing might never generate the correct input. Prime examples include operations involving cryptographic hashes (like SHA and MD5), cyclic redundancy checks (CRCs) checksums, and RSA signatures—or any algorithms designed precisely so that a random or incremental approach will almost never produce a valid output or bypass a check. Consequently, if your target code heavily relies on such operations for input validation or internal logic, you may need to fix these parts or bypass them (by, for example, mocking or disabling checks in test builds) to allow fuzzing to proceed effectively into deeper code paths.

Another related situation involves statefulness. In a *stateful* program, the execution path is determined not only by the input data but also by its state, usually in the form of global or static variables storing values from the preceding run. Stateful programs are much harder to fuzz than stateless programs. In a stateful program, even with the same input, the code can behave very differently, depending on the state it begins with.

Exercising the Code Under Test

After generating random inputs, the fuzzing engine needs a way to feed those inputs to the code under test. The way to do this depends on how the code accepts inputs. If, for example, you're fuzzing a command line tool that takes inputs over the standard input stream, this might be as simple as piping the generated inputs to the program. Similarly, a fuzzing tool could write the inputs to files, then pass their paths to the program.

When fuzzing Android, you're more likely to want to feed the generated inputs to specific functions or API calls. This often requires writing code, usually referred to as a *fuzz harness*, that receives the input from the fuzzing engine and converts it into the correct data types or parameters that the code expects.

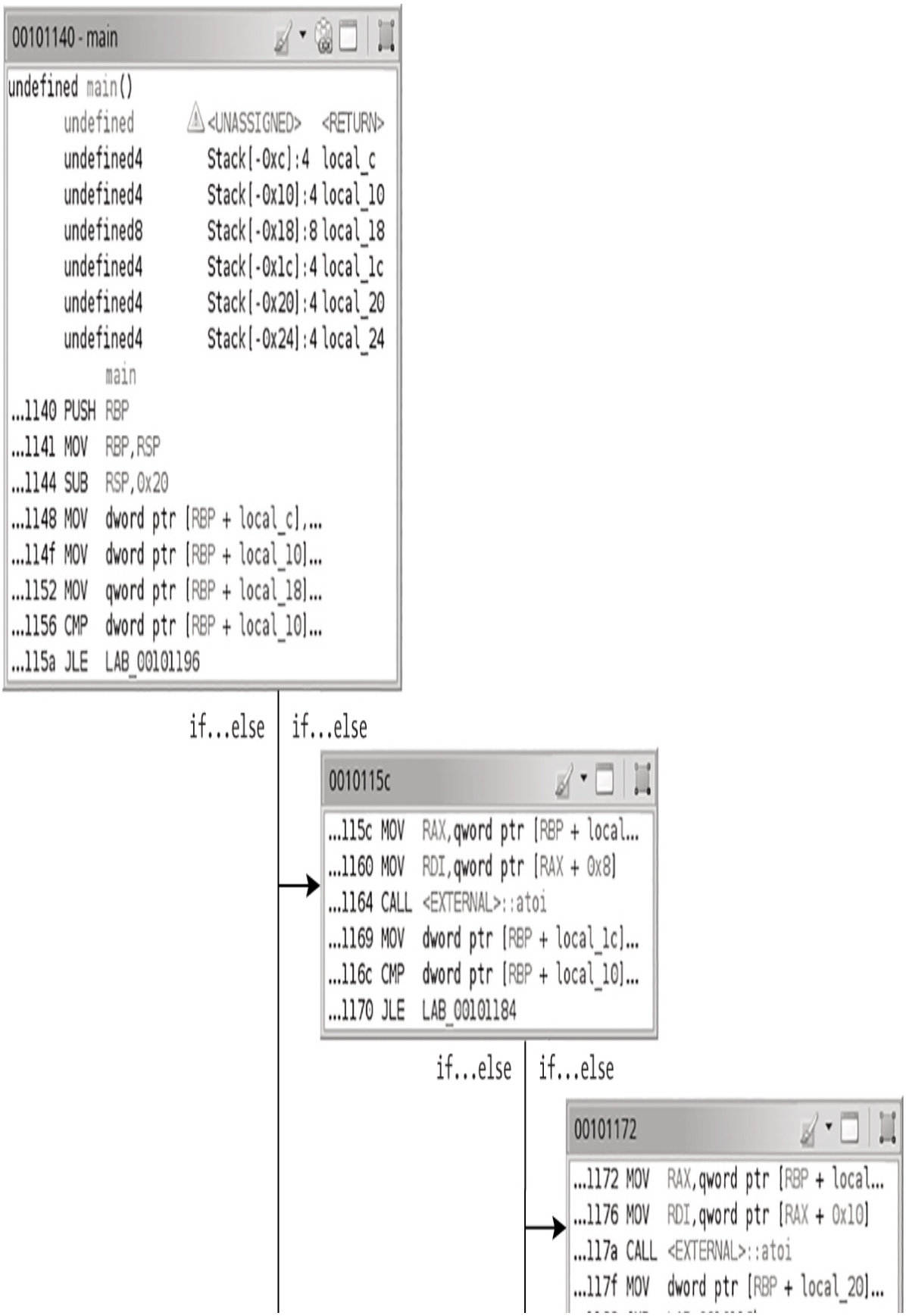
Evaluating Progress

Next, most modern fuzzing tools must observe how the code under test behaves in order to intelligently mutate the random inputs. Some of the simplest fuzzers, often called *black-box fuzzers*, don't do this, resorting instead to mutating inputs without any feedback from the code under test. This is usually less efficient but may be the easiest to set up because it doesn't require any changes to the code or build process. When the code you want to test is distributed only in compiled form, this approach may be the only practical option.

Nevertheless, many fuzzing tools today use some form of coverage feedback, attempting to identify which parts of the code each fuzzing input exercises. We call these *coverage-guided* or *gray-box* fuzzers.

Coverage-guided fuzzing relies on the fact that computer programs are fundamentally composed of execution blocks. Within each block, execution proceeds sequentially, and different blocks connect to one another through control flow structures like functions, loops, and conditional statements. All these structures rely on a basic concept called a *branch*: essentially, a statement that transfers execution from one point in the code to another.

If you treat each block as a node and each branch between nodes as a directional edge, you'll effectively create a control flow graph. In most cases, the graph has a single entry point, where the execution begins, and a single exit point, where the execution finishes. Any sequence of connected nodes and edges from the entry to the exit is a path of code execution; when given a specific set of input data, your program's execution should follow a specific path to reach the exit. For example, Figure 1-1 shows a block diagram generated by loading a compiled binary file into the reverse engineering framework Ghidra.


00101140 - main
undefined main()
undefined <UNASSIGNED> <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined4 Stack[-0x10]:4 local_10
undefined8 Stack[-0x18]:8 local_18
undefined4 Stack[-0x1c]:4 local_1c
undefined4 Stack[-0x20]:4 local_20
undefined4 Stack[-0x24]:4 local_24
main
...1140 PUSH RBP
...1141 MOV RBP,RSP
...1144 SUB RSP,0x20
...1148 MOV dword ptr [RBP + local_c],...
...114f MOV dword ptr [RBP + local_10]...
...1152 MOV qword ptr [RBP + local_18]...
...1156 CMP dword ptr [RBP + local_10]...
...115a JLE LAB_00101196
if...else
if...else
0010115c
...115c MOV RAX,qword ptr [RBP + local...
...1160 MOV RDI,qword ptr [RAX + 0x8]
...1164 CALL <EXTERNAL>::atoi
...1169 MOV dword ptr [RBP + local_1c]...
...116c CMP dword ptr [RBP + local_10]...
...1170 JLE LAB_00101184
if...else
if...else
00101172
...1172 MOV RAX,qword ptr [RBP + local...
...1176 MOV RDI,qword ptr [RAX + 0x10]
...117a CALL <EXTERNAL>::atoi
...117f MOV dword ptr [RBP + local_20]...

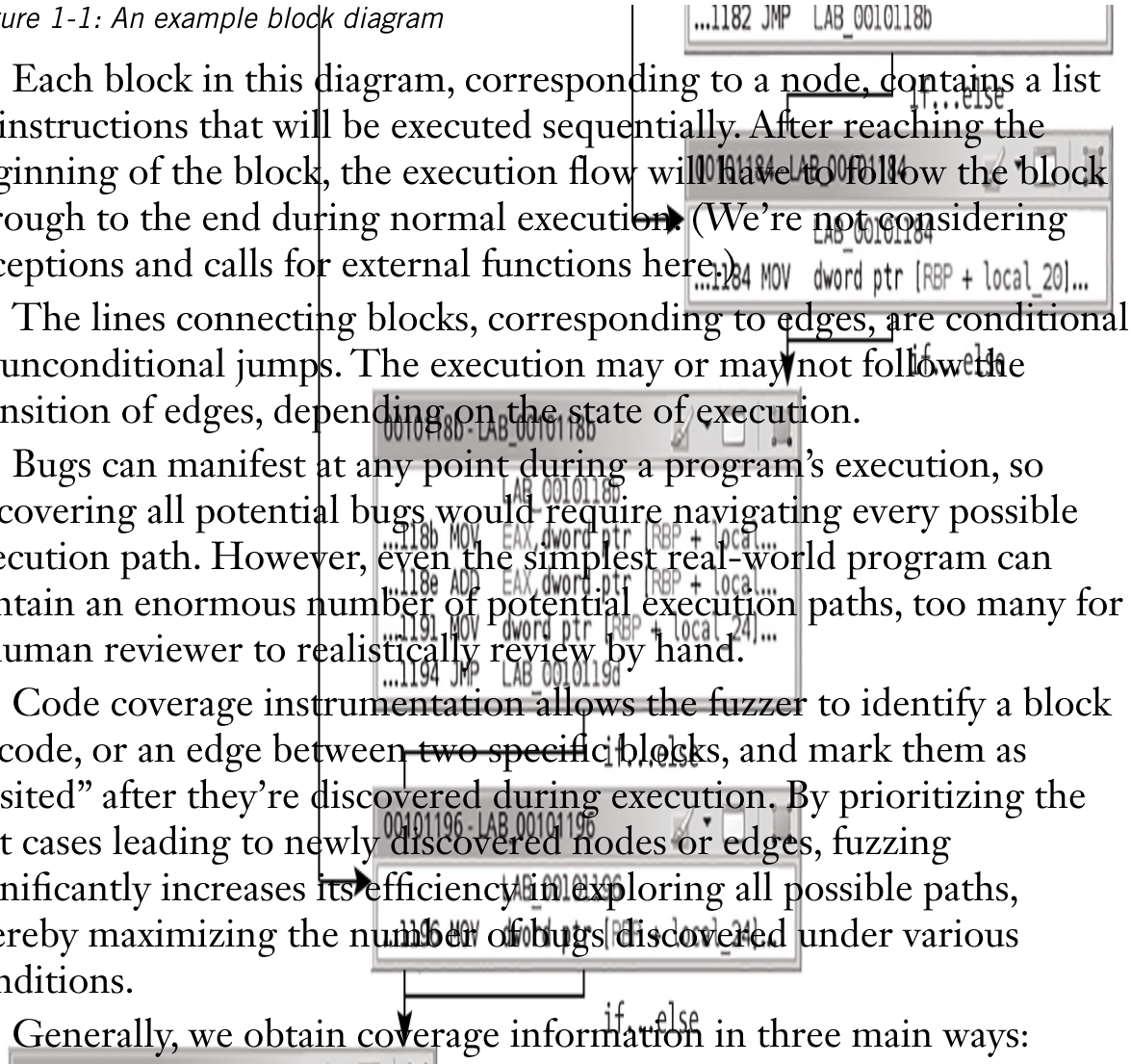


Figure 1-1: An example block diagram

Each block in this diagram, corresponding to a node, contains a list of instructions that will be executed sequentially. After reaching the beginning of the block, the execution flow will have to follow the block through to the end during normal execution. (We're not considering exceptions and calls for external functions here.)

The lines connecting blocks, corresponding to edges, are conditional or unconditional jumps. The execution may or may not follow the transition of edges, depending on the state of execution.

Bugs can manifest at any point during a program's execution, so uncovering all potential bugs would require navigating every possible execution path. However, even the simplest real-world program can contain an enormous number of potential execution paths, too many for a human reviewer to realistically review by hand.

Code coverage instrumentation allows the fuzzer to identify a block of code, or an edge between two specific blocks, and mark them as "visited" after they're discovered during execution. By prioritizing the test cases leading to newly discovered nodes or edges, fuzzing significantly increases its efficiency in exploring all possible paths, thereby maximizing the number of bugs discovered under various conditions.

Generally, we obtain coverage information in three main ways:

Compile-time instrumentation A method primarily used for compiled programming languages. During compilation, the compiler inserts special instructions into the program that are designed to tag and report when different code blocks or edges are executed and to store this information in a designated memory region. Fuzzers then retrieve and use this data to guide their path exploration.

Runtime feedback An approach suitable for interpreter-based languages such as Python and JavaScript. Here, the runtime environment, whether a just-in-time (JIT) compiler or an interpreter, dynamically collects branch information during

execution. The fuzzer then uses this collected data in a similar fashion to guide its path exploration.

Binary instrumentation A variant of runtime feedback often employed in binary-only scenarios where source code isn't available or the target program doesn't inherently provide coverage capabilities. In this case, the fuzzer either emulates or intercepts the target execution to provide the necessary coverage information. While possible, this method typically requires significantly more effort to set up correctly.

Most modern and widely adopted languages, such as Rust, C, C++, C#, Java, JavaScript, Python, and Go, offer robust code coverage support, making them excellent candidates for coverage-guided fuzzing. If your target is written in a less common or domain-specific language, however, you might encounter difficulties implementing coverage-guided fuzzing. Furthermore, if you don't have your target's source code, you'll likely need to rely on binary-only instrumentation to gain coverage support. While feasible, this approach generally demands a much higher level of effort to achieve correct functionality.

Many more possible signals could help improve fuzzing performance. For example, some fuzzing tools observe the code's performance when running a certain input, as inputs that take longer or cause the program to consume more memory may be good candidates for additional mutations. Researchers and fuzzing experts are always looking for new and improved ways to identify good fuzzing inputs, building new fuzzing tools or improving existing ones with their findings.

Detecting Bad Behaviors

Finally, fuzzing tools must find a way to detect bugs and bad behaviors. Like the other steps we've looked at so far, this process can range from very simple to quite complex.

Perhaps the simplest possible way to detect an issue is to watch for crashes. This has the advantage of requiring no modification to the code under test, and problems like segmentation faults cause crashes

surprisingly often. Inputs that cause memory leaks may also cause crashes if they're bad enough that they consume all the available memory and cause an out-of-memory exception. Infinite loops, which can be used for denial-of-service (DoS) attacks, are also easy to detect.

Yet many classes of bugs might be potential security issues but may not cause a crash in all situations. To illustrate, let's consider a common issue, an out-of-bounds array access:

```
#include <stdlib.h>

#define MAX_ARG
int args[MAX_ARG];

int do_sum(int num) {
    int sum = 0;
    for (int i = 0; i < num; i ++) {
        sum += args[i];
    }
    return sum;
}

int main(int argc, char* argv[]) {
    int num = argc - 1;
    for (int i = 0; i < num; i ++) {
        args[i] = atoi(argv[i + 1]);
    }
    return do_sum(num);
}
```

What is likely to happen to this program in a typical execution environment? Often, nothing immediately noticeable. Computer programs can exhibit surprising error resilience. For instance, if an array access goes out of bounds by just one element in a simple integer array, the access might succeed, returning a nondeterministic value. This is frequently due to memory being allocated in slightly larger blocks than strictly requested, whether for performance optimization or security considerations. As long as your access remains within this allocated size,

no drastic event will occur, even though the bug is valid and attackers can exploit it. Even if you step beyond that boundary, in many scenarios you'll simply access another valid allocation or an unused memory region. Your program will fetch an unknown value and continue executing, potentially leading to a subtle corruption that causes a failure only much later, at which point all the original context will have been lost.

Because detecting crashes is so easy, many tools can turn behaviors like out-of-bounds memory access into crashes. These include sanitizers such as AddressSanitizer (ASan) and UndefinedBehaviorSantizer (UBSan), discussed in more detail beginning in Chapter 3. Many of these sanitizers are built into the LLVM compiler used by Android and are easy to turn on. Tools used for other types of testing or debugging can also help you here; for example, adding or enabling assertions in code can turn subtle logic bugs into crashes. This works well if the programmer can identify conditions that should never happen and add them to the code, a practice sometimes referred to as *invariant-based* fuzzing.

Some newer Android devices even have memory safety features built right into their processors. On ARM devices, the Memory Tagging Extension (MTE) provides functionality similar to sanitizers but with little performance overhead. MTE isn't available on all devices and requires a properly configured operating system, but promises to be useful for fuzzing and general system security.

A similar situation arises with path traversal bugs, in which an input causes your program to attempt reading from or writing to an unexpected directory or file. For read operations, the program might fail because the target file doesn't exist, but the program's default error handler might treat this as a typical error. For write operations, your program might successfully write to an unintended file, unless a permission problem arises, which, again, the existing error-handling code might handle. So, you might not notice this critical vulnerability at the time it occurs.

No universal sanitizers exist for classic vulnerabilities like path traversal or Structured Query Language (SQL) injection, nor for logic

issues such as using wrong variables or calling incorrect handlers. These bugs are intimately tied to the program's business logic and intended functionality, so compilers or generic runtime checkers can't detect these issues without knowledge of the application's unique requirements.

This doesn't mean you can't use fuzzing to detect these types of bugs, but it does mean you'll need to implement custom sanitizers or assertion checks in your code. For example, to detect a path traversal, you could hook file operations, then check whether the requested filepath remains within a predefined sandboxed directory, crashing the program if it attempts to access any resource outside this boundary. When combined with coverage-guided fuzzing, this kind of custom check allows the fuzzing engine to mutate inputs toward the error condition, eventually triggering and detecting the vulnerability if it exists.

While some custom checkers can be relatively easy to implement, others (especially those enforcing complex business rules) might require significant effort. In such cases, relying solely on fuzzing to find them might not be the most efficient strategy unless you can devise a clear and verifiable way to implement the checker.

Fuzzing vs. Other Testing Types

The industry has made significant progress in incorporating testing into the software development life cycle, and more developers than ever are writing tests alongside their code. Usually, these take the form of unit tests or integration testing. In this section, we'll discuss how fuzzing has some similarities but also important differences from these forms of testing.

Unit Testing

In one of the most common types of testing, unit testing, the developer evaluates a small, narrowly scoped portion of code, such as a few methods or functions. The developer chooses example inputs and then verifies that each function produces the expected output for each input,

including throwing exceptions or returning error codes in response to invalid inputs.

Many practitioners of unit testing use *test-driven development*, meaning they write the tests before the actual code. This practice requires the developer to think about what the code should do before determining how it should do it. Later, if someone finds and reports a bug in the code, the developer can augment the existing unit tests to reproduce the bad behavior, allowing them to verify that their code change fixes the bug.

Unit testing comes with a few common pitfalls. One is that anticipating the behavior of unusual inputs can be hard. Determining which types of inputs are even valid and worth testing for a particular function can also be difficult. For example, should the function check whether it's been passed a null pointer, or will the code that calls it have already checked for that? What if a bug occurs only when multiple functions are used in sequence? The code may also have dependencies (such as network or storage resources) that don't work correctly in the test environment. Providing simpler implementations of those dependencies for testing purposes, often called *mocking*, is a common solution, but it introduces the possibility of the mocks behaving differently from the code running in production.

Integration Testing

Integration testing differs from unit testing mainly by its focus on the way a software system works as a whole. Instead of looking at only a single function or interface in context, this approach attempts to show that the use of the system in its entirety works correctly. For a media player, for example, integration testing might cover the process of opening and playing back a video file. For a web service, the test might submit a form on one page and ensure that it correctly updates the data on another page. Integration testing may look at the results given by the software under test but often looks at other factors, such as resource consumption and running time.

Much like unit testing, good integration testing requires careful thought about scenarios that accurately represent the way the software

will be used, and the kinds of results to anticipate from unusual inputs. Integration testing is often much slower than unit testing but can also more accurately represent real runtime conditions with database dependencies, filesystem access, and so forth.

Fuzz Testing

One way that fuzzing differentiates itself from these other types of testing is that it doesn't require the developer to enumerate inputs or desired outputs at all. It's also very effective for identifying paths through the code that a developer may not have considered, especially for ill-behaved users or malicious attackers.

While it's possible to fuzz-test a single function, it's often better to fuzz at a higher level to identify bugs in the way multiple functions interact. We like to think of fuzzing as sitting somewhere in the wide continuum between unit and integration testing: flexible enough to fit at many places on that spectrum, even if finding the right spot for a particular piece of software can be a bit of an art.

A unique challenge with fuzzing is how to know when you've sufficiently fuzzed the code. Because you have no predetermined set of tests to run, there's no clearly defined way to show that you've tried all possible inputs. Later in this book, we'll look at a few ways to evaluate how well a fuzz test performs; even so, it's impossible to know for sure how well you've done.

Wrapping Up

In this chapter, we looked at how fuzzing tests code by sending it randomly generated inputs. We also considered several features that are common to many fuzzing tools and scenarios, such as mechanisms to identify erroneous behavior and measure fuzzing progress. Future chapters will look in more detail at some of the specific ways fuzzing can be used on Android devices.

OceanofPDF.com

2
PREPARING ANDROID FOR FUZZING



This chapter provides detailed instructions for setting up your fuzzing environment on Android. We'll cover how to manage the vast collection of Git repositories in the Android Open Source Project (AOSP), download and compile AOSP components, and set up both physical devices and emulated ones. We'll also introduce the Soong build system and the Android Debug Bridge (ADB), which are critical for compiling and deploying fuzz targets. By the end of this chapter, you should have a fully configured environment ready for building and executing your first fuzz target.

Downloading Android

AOSP, the collection of code that makes up Android, is too large to live in a single repository that you can manage with a tool like Git. So, Google developed `repo`, a command line tool for working with Git repositories that makes downloading, updating, and contributing to AOSP faster and easier. To download `repo`, run the following command:

```
$ sudo apt install repo
```

Next, to make the `repo` command easily accessible from any terminal, add it to your system's PATH environment variable. You can typically do so by modifying your *.bashrc* file, a script that runs every time you open a new terminal window, allowing you to set up your environment with custom commands, variables, and settings. Open the file, located in your home directory at *~/.bashrc*, and add this line to it:

```
PATH=$PATH:~/.bin/
```

For the change to take effect, you'll need to either close your current terminal and open a new one, or run this command:

```
$ source ~/.bashrc
```

Now initialize and sync the AOSP project from the remote repositories to your local machine:

```
$ mkdir -p ~/aosp && cd ~/aosp
$ repo init -u https://android.googlesource.com/platform/manifest -b main
$ repo sync -c -j8
```

Initialize your local workspace by running `init` via **`repo init`** ***`manifest_repo _url options`***. Then, run **`repo sync -j`** ***`num_jobs`***, which will download the code, parallelizing the download with the specified number of threads. To find information about the Git repositories that `repo` will download, including the local path at which the repositories should be stored, see the Manifest, an XML file stored in the *platform/manifest* Git repository.

The download process may take a few hours to complete. Luckily, `repo` supports many of the same commands Git does, and you can run **`repo status -j`** ***`num_jobs`*** to get the combined status of all the repositories managed in that `repo` project.

Once the repositories have finished downloading, you'll be able to run a command on all Git repositories managed by `repo` with **`repo forall -`**

`c 'command'`. For example, `repo forall -c 'git log -1'` will print out the meta-data for the last commit into each repository.

Setting Up Your Device

You'll need either a real device or an emulated one to follow along with this book. If you're already familiar with Android development, you might have previously used Goldfish, an Android virtual device provided by Android Studio. Goldfish is ideal for app development but can fall short at times when fuzzing Android at the system level, so we'll instead use the Cuttlefish emulator. Table 2-1 compares Cuttlefish to Goldfish.

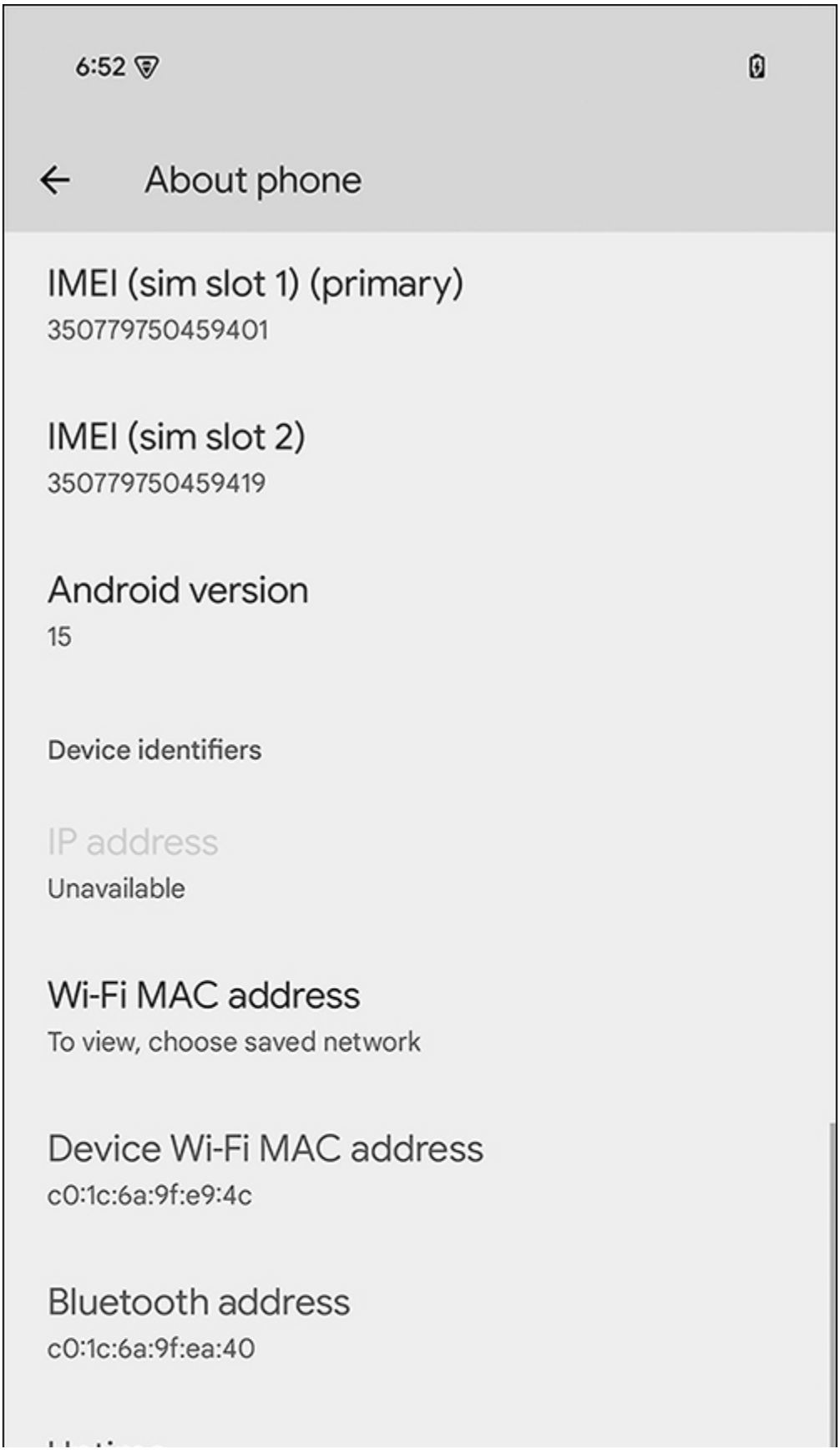
Table 2-1: The Cuttlefish and Goldfish Emulators

Cuttlefish	Goldfish (Android Studio)
Aims to simulate the exact behavior of a real Android device	Aims to simulate a user-level app environment
Full fidelity with the Android framework	Partial fidelity with the Android framework
Mimics a real Android device	Runs Android apps

Even if you have a physical testing device, running through the Cuttlefish setup flow can be beneficial so you can create a fleet of devices on your local machine to run parallel fuzzing campaigns. You'll also be able to easily gain root access to the Cuttlefish devices, which you may need in certain scenarios.

Physical Devices

Preparing your physical device, if you have one, is fairly straightforward. Choose **Settings ▶ About Phone**, then tap **Build Number** seven times to unlock the developer tools, as shown in Figure 2-1.


6:52
About phone
IMEI (sim slot 1) (primary)
350779750459401
IMEI (sim slot 2)
350779750459419
Android version
15
Device identifiers
IP address
Unavailable
Wi-Fi MAC address
To view, choose saved network
Device Wi-Fi MAC address
c0:1c:6a:9f:e9:4c
Bluetooth address
c0:1c:6a:9f:ea:40

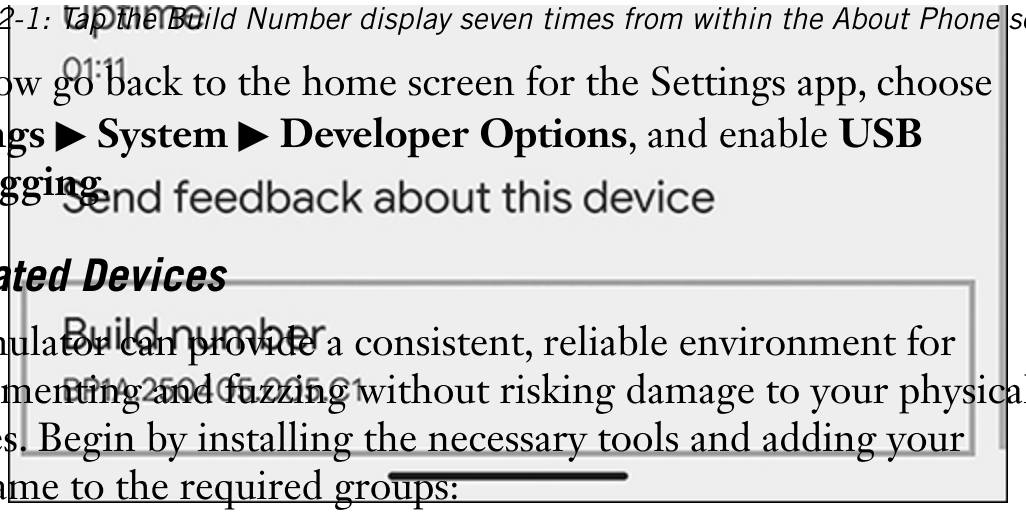


Figure 2-1: Tap the Build Number display seven times from within the About Phone screen.

Now go back to the home screen for the Settings app, choose **Settings ▶ System ▶ Developer Options**, and enable **USB Debugging**.

Emulated Devices

An emulator can provide a consistent, reliable environment for experimenting and fuzzing without risking damage to your physical devices. Begin by installing the necessary tools and adding your username to the required groups:

```
$ sudo apt install -y python3 python3-pip git libvirt-clients \
libvirt-daemon-system qemu-kvm bridge-utils
$ sudo usermod -aG kvm,render $USER
```

These packages install the kernel-based virtual machine (KVM) and quick emulator (QEMU) hypervisor and the libvirt tooling used to create and manage virtual machines (VMs), plus bridging utilities for guest networking. By adding your username to `kvm`, you will enable access to */dev/kvm* for hardware acceleration, allowing your Cuttlefish emulator to run much of its processing directly on your CPU and not through an emulated CPU. The `render` group enables GPU render-node access for graphics acceleration, similar to the CPU acceleration. You will need to log out and back in for group changes to apply.

Now download the necessary Android artifacts. Navigate to *https://ci.android.com*. In the branch called *aosp-android-latest-release*, you should find the build target, *aosp_cf_x86_64_only_phone*. The name of the default branch you land on may change over time, but it should always contain the build for *cf* (Cuttlefish) for the x86_64 architecture, and the artifacts from within the build target should remain the same.

Select the latest green build for *aosp_cf_x86_64_only_phone* and download the following artifacts:

- *cvd-host_package.tar.gz*
- *aosp_cf_x86_64_only_phone-img-*xxxxxx*.zip*

The first, *cvd-host_package.tar.gz*, contains the tooling you need to run the Cuttlefish image locally, while *aosp_cf_x86_64_only_phone-img-xxxxxx.zip* will contain the Cuttlefish image for the selected build that you'll run with the *cvd-host* tools.

Use these commands to unpack and launch the device (you might need to follow instructions at *https://github.com/google/android-cuttlefish* at least once before running these commands):

```
$ unzip aosp_cf_x86_64_only_phone-img-*.zip
$ tar -xvf cvd-host_package.tar.gz
$ HOME=$(pwd) bin/launch_cvd --num_instances=1
```

Leave your Cuttlefish device running in the background as we take a look at the Android build system, Soong, which you'll use to create your own fuzz targets.

The Soong Build System

You'll need to understand Soong, Android's primary build system, to be able to write and build fuzz targets for AOSP. Soong is designed for managing dependencies in large projects. For developers familiar with the broadly available Bazel build system, Soong should feel familiar, with some caveats.

Defining Build Targets

We define build targets (such as shared libraries, binaries, or in our case, fuzz targets) in *Android.bp* files, called *blueprint files*. Let's take a look at an example *Android.bp* file that defines a fuzz target. You can find it in AOSP at *system/media/audio_utils/fuzz/format_fuzzer/Android.bp*:

```
cc_fuzz {
    name: "format_fuzzer",
    srcs: [
        "format_fuzzer.cpp",
    ],
    static_libs: [
```

```
        "libaudioutils",
        "liblog",
    ],
}
```

This file defines `cc_fuzz`, a build module that will build a C++ fuzz target, which we've named `format_fuzzer`. A single source file, *format_fuzzer.cpp*, defines the *fuzz harness*: the code that will parse the input and define how to pass that input to the function your fuzz target is testing (more on that later). You can use the `static_libs` tag in your `cc_fuzz` target to declare a dependency on a `cc_library` (which defines a C/C++ library) target in Soong. Soong will compile the sources from *srcs* and the libraries from *static_libs* into the output binary with the needed instrumentation.

You can also use the `shared_libs` tag to declare dependencies in the same way you would use the `static_libs` field if needed. The difference between the two tags is that `static_libs` will compile your dependent C/C++ library directly into the output fuzz binary, whereas `shared_libs` is compiled to a *.so* file that your fuzzer links to at runtime.

At this point, you may be wondering where `libaudioutils` and `liblog` in the `static_libs` section come from. These are references to other build targets in other *Android.bp* files. Let's take a closer look at `libaudioutils`, defined at *system/media/audio_utils/Android.bp*. Here is the build target's definition, with some options removed and compressed for brevity:

```
cc_library {
    name: "libaudioutils",
    host_supported: true,
    --snip--
    srcs: [
        "Balance.cpp",
        "ErrorLog.cpp",
        "MelAggregator.cpp",
        "MelProcessor.cpp",
        "Metadata.cpp",
        "PowerLog.cpp",
        --snip--
```

```
    ],

    header_libs: [
        "libaudio_system_headers",
        "libutils_headers",
    ],

    export_header_lib_headers: [
        "libaudio_system_headers",
        "libutils_headers",
    ],

    shared_libs: [
        "libcutils",
        "liblog",
        "libutils",
        "server_configurable_flags",
    ],

    --snip--
}
```

Notice that `libaudioutils` is defined as `cc_library` instead of `cc_fuzz`. We can reference and build it as either a static or a shared library in the `cc_fuzz` target, depending on whether we place it in `static_libs` or `shared_libs`.

Let's also take a closer look at `header_libs` and `export_header_lib_headers`. The distinction between them can throw you for a loop if you're not paying careful attention: `header_libs` is the list of header files (header directories and build targets containing your *.h* header files) needed to compile the libaudioutils library, while `export_header_lib_headers` contains the header libraries to export so that modules dependent on libaudioutils will be able to see them and use them in their code.

Another important option being set is `host_supported`. If `host_supported` is `true` (the default value), Soong will build the module for the host as well as the target. The host is the machine on which the build process is running, most likely your laptop. The target represents the device the

code is intended to run on, such as a specific version of Pixel. You'll find the binaries and libraries that will run on your host machine under *out/host*, and the binaries and libraries to be run on your target device are output at *out/target*.

Building a Fuzz Target

Now that you understand *Android.bp* files, you can create a build target. First, initialize your environment (from the cloned Android source tree):

```
$ source build/envsetup.sh
```

This command sets up certain environment variables and allows you to run the `lunch` command to choose your build `target`. The command has this syntax:

```
$ lunch product_release_variant
```

To find the list of available products, releases, and variants to choose from, run the following commands:

```
$ lunch --help
$ list_products
$ list_releases
$ list_variants
```

As good defaults, use `aosp_cf_x86_64_only_phone-trunk_staging-userdebug` when building for a Cuttlefish device or `aosp_arm64-trunk_staging-userdebug` when building for a physical device. You may recognize the build target for Cuttlefish from the Cuttlefish setup stage; we downloaded the `aosp_cf_x86_64_only_phone` target from the Android Build web user interface (UI), from the variant `userdebug`.

Putting the `lunch` command together, you should have

```
$ lunch aosp_arm64-trunk_staging-userdebug
```

or

```
$ lunch aosp_cf_x86_64_only_phone-trunk_staging-userdebug
```

depending on your device. After running `lunch`, you should then be able to access many environment variables:

```
# The path to the out directory
$ echo $OUT
# The path to the "target" out directory
$ echo $ANDROID_PRODUCT_OUT
# The path to the "host" out directory
$ echo $ANDROID_HOST_OUT
```

Now that you've loaded your environment and chosen your build target, you can build your fuzz target. To build the `format_fuzzer` module, use `m`, a convenient command that builds a specified module by name:

```
$ m format_fuzzer -j8
```

Note the use of the `-j` flag once again to parallelize the build jobs and speed up the build process.

Setting Build-Time Options

You can also enhance your build by setting build variables. When fuzzing, for example, you'll want to set the `SANITIZE` variables, which enable sanitizers to detect memory errors and other issues during runtime.

This command adds instrumentation to the `host` output binaries and libraries to detect memory issues:

```
$ SANITIZE_HOST=address m format_fuzzer -j8
```

The next command does the same for the `target` output binaries:

```
$ SANITIZE_TARGET=hwaddress m format_fuzzer -j8
```

You may notice that we use `hwaddress` instead of `address` here, a more performant option when building the fuzz target for a device that supports hardware-assisted address sanitization. If you don't know

whether your device supports hardware-assisted address sanitization, you can always use `SANITIZE_TARGET=address`.

To change the fuzzing framework in use, run this command:

```
$ FUZZ_FRAMEWORK=AFL m format_fuzzer -j8
```

The default framework is libFuzzer, the fuzzing engine we'll use throughout this book. In AOSP, we also have the option of using AFL++, which, while not the default, is the more modern fuzzing engine.

Installing ADB

You'll use ADB to interact with your physical or emulated Android devices, whether by transferring files, managing apps, accessing system logs, or executing commands. Using ADB will deploy your fuzz packages and binaries to your devices, perform the fuzzing, then gather the session's logs, corpus, and other outputs.

ADB operates through a client/server model; the client communicates with a server running on your local machine, which in turn connects to the ADB daemon on the Android device. This architecture supports multiple devices simultaneously. Run these commands to set up ADB:

```
$ wget https://dl.google.com/android/repository/platform-tools-latest-linux.zip
$ unzip platform-tools-latest-linux.zip
$ mv platform-tools ~/.bin/
```

Now, as you did for the `repo` tool, make ADB easily accessible from any terminal by adding `~/.bin/platform-tools` to your PATH variable. Open *.bashrc* and add this line to it:

```
PATH=$PATH:~/.bin/platform-tools
```

Then update your terminal by running `source ~/.bashrc`.

Testing the Fuzzing Environment

To verify that Soong, your device, and ADB all work together, let's build an example fuzzer that already lives in AOSP. You'll find its build definition at *tools/security/fuzzing/example_fuzzer/Android.bp*.

With Cuttlefish

If you're using Cuttlefish, open a new terminal and set up your environment with these commands:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_only_phone-trunk_staging-userdebug
$ SANITIZE_TARGET=address SANITIZE_HOST=address m example_fuzzer -j8
```

Then run these commands to push the fuzzer to your Cuttlefish device and start fuzzing:

```
$ adb root
$ cd $ANDROID_PRODUCT_OUT
$ adb sync data
$ adb shell /data/fuzz/$(get_build_var TARGET_ARCH)/example_fuzzer/example_fuzzer
```

Here we move into the *out/* directory for the product (*device/target/*) we just built for, and we sync the contents of the data directory to the data directory on the phone. Then we run the `adb shell` command to start the fuzzer.

Running `adb root` will give you root access to your phone, which you'll need to run `adb sync data` and sync your data directory to your phone. If you can't get root access, use `adb push data /data/local/tmp/`. With this alternative, instead of syncing your data directory to */data*, you'll create a new data directory in the */tmp* directory at */data/local/tmp/data/*. If you need to push to your phone in this way, make sure you look at all the commands that point to the data directory on your phone, and update them to point to the data directory in your */tmp* directory. For example, */data/fuzz/*. . . should become */data/local/tmp/data/fuzz/*.

You should now see fuzzing happening from your terminal:

```
#963    NEW    cov: 361 ft: 397 corp: 26/88b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 2 Cha-
#1050   NEW    cov: 361 ft: 399 corp: 27/92b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 2 Ins-
#1156   NEW    cov: 361 ft: 400 corp: 28/96b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Cr-
#1242   NEW    cov: 361 ft: 401 corp: 29/100b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1303   NEW    cov: 361 ft: 402 corp: 30/104b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1321   NEW    cov: 361 ft: 408 corp: 31/108b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 3 Cr-
#1392   NEW    cov: 361 ft: 409 corp: 32/112b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1432   NEW    cov: 361 ft: 410 corp: 33/116b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 5 Ch-
#1504   NEW    cov: 361 ft: 411 corp: 34/120b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 2 Ch-
#1620   NEW    cov: 361 ft: 412 corp: 35/124b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Cr-
#1781   NEW    cov: 361 ft: 413 corp: 36/128b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1987   NEW    cov: 361 ft: 414 corp: 37/134b lim: 6 exec/s: 0 rss: 46Mb L: 6/6
MS: 1 In-
```

In this example output, you can see the stats of the fuzzer running. For instance, in the last line, `#1987` means the fuzzer has run 1,987 inputs through the fuzzer, and `cov: 361` is telling us the number of *blocks* of code that have been hit. Next, `corp: 37/134b` shows us that we have 37 corpus files, totaling 134 bytes.

In the following example, we label each code block:

```
int parsePacketHeader(const uint8_t* data, size_t size) {
    // BLOCK 1: Entry
    uint32_t headerValue = 0;
    int statusCode = 0;
    bool isAuthorized = false;
```

```
    if (size < 4) {
        // BLOCK 2: True branch (error path)
        statusCode = -1;
        log.error("Buffer too small to contain header");
        return statusCode;
    }

    // BLOCK 3: False branch (success path)
    headerValue = (data[0] << 24) | (data[1] << 16) | (data[2] << 8) | data[3];
    isAuthorized = checkPermissions(headerValue);
    statusCode = processPayload(data + 4, size - 4);
    return statusCode;
}
```

One field that is important for a fuzz harness developer to pay attention to is the `exec/s: 0` field. This shows you the number of executions per second you're getting. The fuzzer we're using is very slow, at less than one execution per second. Your goal, as you create your own fuzzer, is to shoot for hundreds or thousands of executions per second, but as you see here, sometimes you need to settle for far, far fewer.

With a Physical Device

If you're using a physical device, open a new terminal and set up your environment with these commands:

```
$ source build/envsetup.sh
$ lunch aosp_arm64-trunk_staging-userdebug
$ SANITIZE_TARGET=hwaddress SANITIZE_HOST=address m example_fuzzer -j8
```

Since we're using a real device, we specify `hwaddress` to enable hardware-supported sanitization.

Now push the fuzzer to your physical device and start fuzzing:

```
$ adb root
$ cd $ANDROID_PRODUCT_OUT
```

```
$ adb sync data
$ adb shell /data/fuzz/$(get_build_var TARGET_ARCH)/example_fuzzer/example_fuzzer
```

You should see output appearing in the terminal.

On Your Host Machine

After following the setup and build commands for either your emulator or your physical device, you can use this command to fuzz on your host machine:

```
$ $ANDROID_HOST_OUT/fuzz/x86_64/example_fuzzer/example_fuzzer
```

Fuzzing on the host is much simpler than fuzzing on a device; this is because it cuts out the slightly annoying steps of pushing the binaries to your device and starting them there. You'll likely do most of your fuzzing on your host.

Some targets in AOSP don't support building for the host, however. They may, for example, require specific physical hardware (that can't be emulated) or depend closely on the Android environment, such as needing Bionic (Android's libc implementation). If one of your dependencies has such a restriction, you'll end up with an error when you try to build for your host. If this happens, you'll need to set `host_supported: false` in your `cc_fuzz` build rule in your *Android.bp* file. To be able to fuzz these targets, you'll need to use either an Android physical device or a Cuttlefish emulator.

With AFL++

We mentioned that you also have the option of building and running the fuzz target via the AFL++ fuzzing engine. To do so, open a new terminal and set up your environment, choosing either the Cuttlefish or the physical device flow shown previously in this section. Next, build your fuzz target, as well as the `afl-fuzz` binary, using this command:

```
$ FUZZ_FRAMEWORK=AFL SANITIZE_TARGET=address m example_fuzzer afl-fuzz -j8
```

We've listed `afl-fuzz` in the modules to build. Next, let's push what we've built to the device:

```
$ cd "$ANDROID_PRODUCT_OUT"
$ cp system/bin/afl-fuzz data/fuzz/$(get_build_var TARGET_ARCH)/example_fuzzer
$ adb sync data
```

Now set up and run the AFL++ fuzzer on the device:

```
$ adb shell
$ ARCH=$(get_build_var TARGET_ARCH)
$ FUZZER_PATH=/data/fuzz/${ARCH}/example_fuzzer
$ adb shell "mkdir $FUZZER_PATH/i $FUZZER_PATH/o"
$ adb shell "echo 'starting input' > $FUZZER_PATH/i/intput1"
$ adb shell "AFL_SKIP_CPUFREQ=1 $FUZZER_PATH/afl-fuzz \
-i $FUZZER_PATH/i -o $FUZZER_PATH/o -- $FUZZER_PATH/example_fuzzer"
```

AFL++ needs you to create an input and an output directory and pass these to the `afl-fuzz` binary to start fuzzing. The input directory also needs at least one pre-populated input as a starting point.

Wrapping Up

This chapter covered the foundational tools and procedures required to set up an Android environment capable of running sophisticated fuzzing campaigns. You learned how to navigate the massive AOSP via `repo`, created an emulated device with Cuttlefish, and set up ADB to communicate with your devices. However, having a running environment is just the prerequisite. To fuzz in AOSP and find vulnerabilities effectively, you'll need to take a look at the tools that create the fuzzing ecosystem in Android and elsewhere.

OceanofPDF.com

3
TOOLS AND FRAMEWORKS



This chapter provides an introduction to the fuzzing tools available for use with the AOSP and how to work with them. You'll gain a deeper understanding of sanitizers, which allow fuzz targets to find actionable crashes. You'll learn about fuzzing engines (the core components that run the fuzzing loop, generate inputs, measure test coverage, and guide mutations). You'll explore analysis techniques for measuring the effectiveness of your fuzz harness and finding gaps. And, finally, you'll work with data-handling tools, so you can quickly and effectively process the data generated by the fuzzing engine.

Sanitizers

In Chapter 1, we mentioned that many of the most severe vulnerabilities don't cause an immediate or obvious crash. Instead, they can manifest as silent memory corruptions or logical errors from undefined behavior, leading to bugs caused by one section of code that surface in a completely different area. Without sanitizers, a fuzzer might trigger

such a bug thousands of times without causing any observable crashes or might generate crashes that point to the wrong code.

Sanitizers solve this problem. These powerful instrumentation tools, which are built into the compiler, add runtime checks to a program to detect hidden bugs the moment they occur. When a sanitizer finds an issue, it stops the program and generates a report to pinpoint the exact location and cause of the bug. The most popular sanitizers to use are as follows:

AddressSanitizer (ASan) The cornerstone sanitizer for fuzzing, designed to catch a wide range of memory errors such as heap and stack buffer overflows, as well as use-after-free, double-free, and other invalid memory accesses. On physical Android devices, you'll use the hardware-assisted version of ASan, called *HWASan*.

Memory Tagging Extension (MTE) A hardware feature on modern ARMv9 CPUs that provides a highly efficient alternative to ASan. It will catch the same categories of memory safety errors as ASan, but because the checks are performed in hardware, MTE has significantly lower performance overhead than ASan's software-based approach.

UndefinedBehaviorSanitizer (UBSan) Detects operations that the C and C++ language standards define as undefined behavior. These include signed integer overflow, invalid bit shifts, misaligned memory accesses, and invalid type casting.

MemorySanitizer (MSan) Can detect the use of uninitialized memory by catching reads from variables that have been allocated but not yet written to. Reading from uninitialized memory can introduce nondeterministic behavior and leak information from leftover data in memory.

Sanitizers can significantly impact the performance of your code, slowing your fuzzer's number of executions per second. ASan typically adds a performance overhead of around 100 percent, while MTE may add an overhead in the tens-of-percent range (although these

percentages will vary widely). So, to test Android, sticking with just ASan or MTE is usually best to catch the most common and critical bug classes while keeping the overhead manageable.

The difference between these two sanitizers is in the way they are implemented. ASan works by creating a region of *shadow memory*. For every 8 bytes of application memory, ASan reserves 1 shadow byte to track its state, which may be addressable, partially addressable (that is, only the first *k* bytes are accessible), or poisoned (meaning the program should not be reading from or writing to this location). If the program then accesses an address that is poisoned, ASan reports a violation.

Red zones are regions of memory that are completely poisoned and placed adjacent to an allocation. Through the compile-time instrumentation, ASan adds checks on memory accesses so that if code reads or writes into a poisoned red zone (for example, by overrunning an array), it reports the out-of-bounds error.

MTE works by assigning a 4-bit tag to both the pointers and each 16-byte chunk of memory. When code uses a pointer to access memory, the hardware checks that the pointer's tag matches the memory block's. If they don't match, MTE will catch this. One downside of this approach is that since the tag is only 4 bits long, there is a 1 in 16 chance that the tag from a wayward pointer will coincidentally match a memory block's tag that doesn't belong to it. This means there is a small probability that MTE will miss an error during a single execution, but because tags are randomized upon allocation, repeated executions increase the chance of observing a mismatch and catching the issue.

Fuzzing Engines

A fuzzing engine is a tool responsible for generating fuzzing inputs (together called the *corpus*), executing the target program, and analyzing the results. Many fuzzing campaigns select one primary engine, then may add a secondary engine to apply different strategies to the same corpus. For example, the most common fuzzer in the Android ecosystem is libFuzzer, but many testers broaden their coverage with

AFL++. This section provides an overview of several widely used fuzzing engines.

libFuzzer

The libFuzzer coverage-guided fuzzing engine is part of the LLVM/Clang toolchain and is linked to the fuzz harness at compile time. It runs in process, allowing a very high execution speed, and is integrated with the compiler, so creating a fuzzer is often as simple as adding a few new build flags. One limitation is that the code you fuzz must be built through Clang.

While libFuzzer is still widely used, stable, and very effective, it's in maintenance mode, and its performance has fallen slightly behind modern, actively developed fuzzing engines such as AFL++.

AFL++

AFL++ is a community-maintained fork of the legendary American Fuzzy Lop (AFL) engine. This high-performance, out-of-process, coverage-guided fuzzing engine fuzzes the target code in a separate process through the use of a *fork server*. This forks a new child from a clean snapshot of the process for each input, ensuring that the global state doesn't become corrupted and that all crashes found are reproducible.

AFL++ also has a *persistence* mode, in which the developer can specify the number of inputs to feed the function before creating a new process and clearing the global state. Global state changes will still accumulate as the engine supplies inputs for the same forked process, but this approach strikes a balance between speed and maintaining a clean global state for reproducibility.

This engine is great for nearly any fuzzing scenario. It supports compiler-based instrumentation for code built through Clang or the GNU Compiler Collection (GCC) and is also able to instrument binary targets with dynamic instrumentation via QEMU or Frida. The active and dedicated community surrounding AFL++ makes it one of the most effective fuzzing engines available.

LibAFL

Unlike the other tools discussed so far, LibAFL isn't a monolithic fuzzing engine but a highly modular framework for building your own custom fuzzers. It provides a rich set of reusable components, allowing you to assemble a customized fuzzer to fit your needs. With LibAFL, you can mix and match the following primitives:

Executor Runs the target in the specified way (such as in process, fork server, or QEMU)

Observer Gathers feedback about various metrics (such as code coverage, execution time, and number of system calls)

Feedback Consumes output from the observers to decide whether an input is interesting

Scheduler Sets a strategy for selecting the next input from the corpus to mutate

Mutator Applies mutation strategies (such as bit flipping, splicing two corpus items together, and so on)

Because of its high customizability, LibAFL is an excellent choice for complex targets, research, and scenarios where other fuzzing engines fall short.

Syzkaller

Syzkaller is a specialized coverage-guided fuzzing engine designed for fuzzing operating system kernels. It generates sequences of system calls to find bugs in the kernel's API. By using a high-level domain-specific language called *Syzlang*, Syzkaller defines syscalls and their complex relationships, allowing it to generate valid interactions. We cover Syzkaller in more detail in Chapter 9.

CHOOSING A FUZZING ENGINE

The simplest way to choose a fuzzing engine is to follow this rule: For standard Android components, start with libFuzzer and sanitizers because of its native integration and support in the AOSP build system. If you are targeting a project that can be built with Clang, libFuzzer is usually the easiest place to begin. If you need binary-only or mixed targets, AFL++ gives you flexible modes while still getting feedback-guided mutation. If you have moved on to very advanced fuzzing techniques or are experimenting with new approaches to fuzzing, consider LibAFL. If you are targeting the kernel, go with Syzkaller.

Data Parsing

A fuzzer has no way of distinguishing an image parser from a network protocol handler, a text-rendering library, or any other type of software under test. Therefore, the fuzzer must use the limited amount of data fed back to it to develop an understanding of which inputs were (or will be) interesting.

In practice, this understanding will come from a fuzz harness you'll write. The fuzz harness will accept a buffer of raw bytes from the fuzzing engine, then interpret those raw bytes and turn them into the typed arguments that the target function expects.

Let's discuss how to manually parse this data, then cover useful tools that will make this process less tedious and error prone. The function shown next will serve as our target. To follow along, create a new directory at *tools/security/fuzzing/parsing_examples/*, then create two new files in that directory, *test_function.cpp* and *Android.bp*. (For more on the *Android.bp* file, revisit "The Soong Build System" on page 19.) Add the following content to *test_function.cpp*:

test_function.cpp

```
#include <cstdint>
#include <string>
```

```
void myTestFunction(uint32_t magic_value, const std::string& user_name,
                    uint8_t flags, int16_t transaction_id, bool is_premium) {

    if (magic_value != 0xDEADBEEF) {
        return;
    }

    if (is_premium && !user_name.empty()) {
        if ((flags & 0x80) && transaction_id > 0) {
            __builtin_trap();
        }
    }
}
```

This vulnerable function takes structured arguments, then runs through numerous conditionals, which, if all satisfied, will trigger a crash. Now let's see how to make the inputs supplied by the fuzzer more likely to trigger this issue.

Manual Parsing

A straightforward way to turn the byte buffer that's passed in by the fuzzing engine into the typed inputs expected by our target function is to manually decode the necessary values from the raw input bytes. Here's an example fuzz harness that uses this approach to extract the values needed to call `myTestFunction()`:

manual_parsing_fuzzer.bp

```
#include <cstdint>
#include <cstddef>
#include <string>
#include "test_function.cpp" // Include the function to be tested.

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
 ❶ if (size < sizeof(uint32_t)
              + sizeof(uint8_t)
              + sizeof(uint8_t)
```

```
            + sizeof(int16_t)
            + sizeof(bool)) {
        return 0;
    }

    size_t offset = 0;

    uint32_t magic_value = *reinterpret_cast<const uint32_t*>(data + offset);
    offset += sizeof(uint32_t);

    uint8_t name_len = *(data + offset);
    offset += sizeof(uint8_t);

    if (size < offset + name_len) {
        return 0;
    }
    std::string user_name(reinterpret_cast<const char*>(data + offset),
                          name_len);
    offset += name_len;

❷ if (size < offset + sizeof(uint8_t) + sizeof(int16_t) + sizeof(bool)) {
        return 0;
    }

    uint8_t flags = *(data + offset);
❸ offset += sizeof(uint8_t);

    int16_t transaction_id = *reinterpret_cast<const int16_t*>(data + offset);
    offset += sizeof(int16_t);

    bool is_premium = static_cast<bool>(data[offset]);
    offset += sizeof(bool);

    myTestFunction(magic_value, user_name, flags, transaction_id, is_premium);

    return 0;
}
```

First, this function verifies that the size of the buffer is large enough to populate all the values ❶. You'll notice that, since strings can be variable lengths, we need to check the size of the remaining data after assigning the string value ❷. We also need to track our position in the buffer; to do this, we keep a variable, `offset`, and increment it after each assignment ❸.

Here is the corresponding *Android.bp* file entry:

Android.bp

```
// Fuzzer 1: Manual Parsing
cc_fuzz {
    name: "manual_parsing_fuzzer",
    srcs: [
        "manual_parsing_fuzzer.cpp",
    ],
}
```

This harness involves a lot of complexity for parsing only a few simple values from the input. In particular, the manual approach comes with two specific drawbacks:

Constant size checks Before each piece of data, we must manually check its size to ensure that we don't read past the end of the buffer. An off-by-one error here will introduce a bug into the harness itself, causing it to crash frequently before we can explore the code more deeply.

Pointer arithmetic and casting In our example, we've manually cast `char*` pointers to read various data types. This can easily introduce more bugs into the harness, leading to time wasted investigating bugs in the harness when we could be fuzzing.

You may also notice an issue with our manual parsing code. Since the `offset` variable depends on the dynamic length of the user string, it may point to an unaligned memory address when this line executes. Since we're casting to `int16_t`, which is 2 bytes in size, the pointer

created by `reinterpret _cast` will be unaligned if the offset is odd and will result in undefined behavior.

If you have UBSan enabled, it will report this error, producing noisy reports, and block the fuzzer from reaching the actual target. On some architectures, this error can even trigger a hardware exception and immediately kill the process. To avoid this, you can manually realign the offset as needed. In practice, the simplest way would be to use `memcpy()`, but this illustrates the unseen complexities in parsing even a handful of simple values.

Altogether, manually parsing values from raw bytes can be tricky to get right; in fact, it's common to fuzz parsing functions because of this difficulty. Not only might there be bugs in the parsing logic causing the harness to crash, but the harness may also fail to produce a wide range of values. For example, if a harness needs an integer modulo 500, a developer might write `num = data[0] % 500`, unintentionally limiting the fuzzer to values from 0 to 255 because `data[0]` is a single byte.

Now it's time for you to become familiar with the fuzzing tools you can use to skip the headache of writing parsing logic and debugging fuzz harness crashes.

With the FuzzedDataProvider Library

Instead of constantly writing and rewriting parsing logic, you might want to go ahead and make this code a reusable library. Well, you're in luck; the fuzzing framework already provides a class called `FuzzedDataProvider`. This class wraps the raw byte stream in a much safer and more developer-friendly API. Here is the harness rewritten using `FuzzedDataProvider`:

fdp_fuzzer.cpp

```
#include <cstdint>
#include <string>
#include <fuzzer/FuzzedDataProvider.h>
#include "test_function.cpp"
```

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    FuzzedDataProvider provider(data, size);

    uint32_t magic_value = provider.ConsumeIntegral<uint32_t>();
    std::string user_name = provider.ConsumeRandomLengthString();
    uint8_t flags = provider.ConsumeIntegral<uint8_t>();
    int16_t transaction_id = provider.ConsumeIntegral<int16_t>();
    bool is_premium = provider.ConsumeBool();

    myTestFunction(magic_value, user_name, flags, transaction_id, is_premium);
    return 0;
}
```

We begin by initializing the `FuzzedDataProvider` object with the raw data pointer and its size. Then we skip all the manual parsing by using it to provide all our values, keep track of the offset, and ensure that our values are parsed properly.

Here is the *Android.bp* entry for the FuzzedDataProvider fuzzer:

Android.bp

```
//
--snip--
cc_fuzz {
    name: "fuzzed_data_provider_fuzzer",
    srcs: [
        "fdp_fuzzer.cpp",
    ],
}
```

This fuzzer significantly improves the safety and readability of the parsing, as FuzzedDataProvider will handle all boundary checks internally. If you ask for more data than is available, it will gracefully return a default value (such as `0`, an empty string, or `false`), averting a crash and allowing the fuzzer to continue fuzzing. This increases the likelihood of a crash being the result of an error in the target code rather than in the harness.

Using FuzzedDataProvider is an easy way to begin cleaning up your fuzz harness code and is the right tool to use in many cases. But we can make the input generated by the fuzzing engine even more targeted.

With the libprotobuf-mutator Library

In both of the previous examples, the fuzzing engine had limited information about the data it generated. For instance, in the manual parsing fuzzer, we decided on the length of the string we created by reading an arbitrary set of bytes from the input data, but the fuzzing engine doesn't know that this part of the input will determine a string length, and it doesn't know which bytes of the input will be read as a string.

In this section, we'll look at libprotobuf-mutator, which uses structure-aware mutations to create the input, giving the fuzzing engine a formal map of the input structure. This library works by mutating instances of a Protocol Buffers (Protobufs) message, then passing that message into your harness. If you haven't worked with Protobufs before, know that they're a language-neutral way to serialize data and move it around, similar to XML or JavaScript Object Notation (JSON).

To use this library, we'll need to create a Protobuf schema in a *.proto* file to define the fields we'll use in our new fuzz harness. You'll also notice that we're no longer including the `LLVMFuzzerTestOneInput()` function. Here is the content of the *example.proto* file:

example.proto

```
syntax = "proto2";

message MyTestInput {
  optional uint32 magic_value = 1;
  optional string user_name = 2;
  optional uint32 flags = 3;
  optional int32 transaction_id = 4;
  optional bool is_premium = 5;
}
```

Protobufs don't have 8-bit or 16-bit integer types, so we use 32-bit fields and explicitly narrow them in the harness. Here is the fuzz harness, *lpm _fuzzer.cpp*, using the `DEFINE_PROTO_FUZZER` macro:

lpm_fuzzer.cpp

```
#include <cstdint>
#include <string>
#include "example.pb.h"
#include "test_function.cpp"

#include "src/libfuzzer/libfuzzer_macro.h"

DEFINE_PROTO_FUZZER(const MyTestInput& input) {
    uint32_t magic_value = input.magic_value();
    std::string user_name = input.user_name();
    uint8_t flags = static_cast<uint8_t>(input.flags());
    int16_t transaction_id = static_cast<int16_t>(input.transaction_id());
    bool is_premium = input.is_premium();

    myTestFunction(magic_value, user_name, flags, transaction_id, is_premium);
}
```

We use *example.pb.h* to include our Protobufs-generated header. Notice the input is a structured object, not raw bytes, so we need only read our values from the passed-in Protobuf, cast some of them to their expected types, and call our function. Here's the *Android.bp* file for the Protobuf mutator approach:

Android.bp

```
--snip--
cc_library_static {
    name: "example_input_proto",
    srcs: ["example.proto"],
    proto: {
        export_proto_headers: true,
    },
```

```
    shared_libs: ["libprotobuf-cpp-full"],
}

cc_fuzz {
    name: "lpm_fuzzer",
    srcs: [
        "lpm_fuzzer.cpp",
    ],
    static_libs: [
        "example_input_proto",
        "libprotobuf-mutator",
    ],
    shared_libs: [
        "libprotobuf-cpp-full",
    ],
}
```

The wonderful thing about using this structure-aware approach is it requires no manual parsing. The `DEFINE_PROTO_FUZZER` macro and `libprotobuf` mutator engine handle the complex and error-prone task of translating raw bytes into a fully populated `MyTestInput` object. In addition, the fuzzing engine is smart; it understands that `magic_value` is a 4-byte integer and can set this field to a known interesting value, such as `0`, `-1`, or `MAX_INT`. We've also just made this fuzzer easy to maintain. If the function's signature changes in the future, we can simply update the Protobuf in the *.proto* file to match, pass in the new field, and begin fuzzing.

Automating the Fuzzing Life Cycle

While the tools discussed so far are excellent for creating and running fuzzers locally, in the long term you'll want an automated solution that can continuously build fuzz targets and hunt for vulnerabilities. An open source infrastructure called ClusterFuzz can meet this need. Originally developed by Google, it serves as the backbone for fuzzing massive projects like Google Chrome and AOSP, providing automation at a scale that would be impossible to manage manually.

ClusterFuzz automates the entire fuzzing life cycle. A developer submits a fuzz target, and the platform handles the rest: building the code with various sanitizers, running the fuzzer 24/7 on a massive fleet of computers, and processing the results. After finding a crash, ClusterFuzz automatically de-duplicates it, minimizing the test case to its smallest form, and files a detailed bug report. ClusterFuzz can even identify the exact commit that introduced the vulnerability, freeing developers from manual triage so they can focus on the fix.

Corpus minimization is a process that removes new corpus entries if they don't add more coverage. As fuzzers run, they discover new random inputs that provide more coverage, and we hope that if errors exist or are added to the code that is covered, we'll find the issue with fuzzers. However, over time, as code changes, or depending on other parameters, you will inevitably end up having more and more fuzzer inputs—too many fuzzer inputs. If these inputs produce the same coverage, running these duplicated inputs won't provide any more value. So, if you run a fuzzer for any amount of time, you typically want to minimize the coverage, at least sometimes.

While you can set up your own instance of ClusterFuzz, doing so takes significant effort. For most open source projects, it's much easier to follow ClusterFuzz's online documentation to onboard to Google's instance, OSS-Fuzz, which is free for open source projects. OSS-Fuzz currently supports more than 1,100 projects. You can find its onboarding documentation at *https://google.github.io/oss-fuzz/getting-started/new-project-guide/*.

Analyzing Lines Covered

To understand which parts of the code your fuzzer is testing, you can generate a coverage report, which visually shows which lines were executed and which were missed. (At the time of this writing, generating coverage reports works on physical devices only.) This section will guide you through the process of generating a detailed report, then strengthening your fuzzer based on the results from that report.

You generate a coverage report in three main steps: building the fuzz target with coverage instrumentation, running the instrumented fuzzer to generate the raw profile data, and processing the data on your local machine to generate the report. To build the fuzz target, navigate to the AOSP root. Use the same terminal window throughout this process, as you'll be setting new shell variables throughout.

These commands will build the target with coverage instrumentation:

```
$ source build/envsetup.sh
$ lunch aosp_arm64-trunk_staging-userdebug
$ CLANG_COVERAGE=true NATIVE_COVERAGE_PATHS=* m libskia_image_processor_fuzzer -
j8
```

Next, push the output of your build to your device and run the fuzzer for 5,000 executions to generate profile data:

```
$ cd "$ANDROID_PRODUCT_OUT/data/fuzz/$(get_build_var TARGET_ARCH)"
$ adb push * /data/local/tmp
$ adb shell
$ cd data/local/tmp/libskia_image_processor_fuzzer
$ LLVM_PROFILE_FILE=profdata/out.profraw ./libskia_image_processor_fuzzer -
runs=5000
```

You should see results like the following after running your fuzzer:

```
#963    NEW    cov: 361 ft: 397 corp: 26/88b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 2 Cha-
#1050   NEW    cov: 361 ft: 399 corp: 27/92b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 2 Ins-
#1156   NEW    cov: 361 ft: 400 corp: 28/96b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Cr-
#1242   NEW    cov: 361 ft: 401 corp: 29/100b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1303   NEW    cov: 361 ft: 402 corp: 30/104b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1321   NEW    cov: 361 ft: 408 corp: 31/108b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 3 Cr-
```

```
#1392   NEW    cov: 361 ft: 409 corp: 32/112b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1432   NEW    cov: 361 ft: 410 corp: 33/116b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 5 Ch-
#1504   NEW    cov: 361 ft: 411 corp: 34/120b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 2 Ch-
#1620   NEW    cov: 361 ft: 412 corp: 35/124b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Cr-
#1781   NEW    cov: 361 ft: 413 corp: 36/128b lim: 4 exec/s: 0 rss: 45Mb L: 4/4
MS: 1 Ch-
#1987   NEW    cov: 361 ft: 414 corp: 37/134b lim: 6 exec/s: 0 rss: 46Mb L: 6/6
MS: 1 In-
        NEW_FUNC[2/2]: 0x55dc11b3f340 in SkSwizzler::swizzle(void*, unsigned char
const*-
#2265   NEW    cov: 367 ft: 421 corp: 39/146b lim: 8 exec/s: 0 rss: 51Mb L: 6/6
MS: 1 Co-
#2266   NEW    cov: 367 ft: 422 corp: 40/154b lim: 8 exec/s: 0 rss: 51Mb L: 8/8
MS: 1 Pe-
#2310   NEW    cov: 368 ft: 423 corp: 41/161b lim: 8 exec/s: 0 rss: 51Mb L: 7/8
MS: 4 Ch-
#2381   NEW    cov: 368 ft: 426 corp: 42/165b lim: 8 exec/s: 0 rss: 51Mb L: 4/8
MS: 1 Ch-
#2400   NEW    cov: 370 ft: 431 corp: 43/173b lim: 8 exec/s: 0 rss: 51Mb L: 8/8
MS: 4 Pe-
#2461   NEW    cov: 370 ft: 432 corp: 44/181b lim: 8 exec/s: 0 rss: 51Mb L: 8/8
MS: 1 Ch-
#2472   NEW    cov: 370 ft: 438 corp: 45/185b lim: 8 exec/s: 0 rss: 51Mb L: 4/8
MS: 1 Ch-
```

You should also now have a directory alongside your fuzz binary called *profdata*. Inside *profdata*, you'll have a file called *out.profraw* that contains the raw data, tracking which lines of code were executed by the fuzzer.

Because these coverage tools in AOSP can sometimes break, let's make sure everything works as expected before continuing by printing the data in *out.profraw* to the screen:

```
$ ls -lh profdata/out.profraw
-rw-rw-rw- 1 shell shell 31M 12-01 18:54 profdata/out.profraw
```

You can see this file is 31MB in size; this means the coverage data was collected. Otherwise, the file size would show 0. Coverage sometimes breaks in AOSP, so if the data isn't being collected, you should still follow along with this book to understand the process.

The next step is to pull this data off the device so you can work with it on your local machine. Here are the commands to retrieve the profile data:

```
$ exit
$ cd $ANDROID_BUILD_TOP
$ adb pull /data/local/tmp/libskia_image_processor_fuzzer/profdata/
```

You'll use `llvm-profdata` to merge and format the data and `llvm-cov` to take that data and generate the reports. These tools ship alongside Clang as part of the LLVM project. To ensure that you have no versioning issues, use the tools in AOSP by updating the PATH variable:

```
$ export PATH=$ANDROID_BUILD_TOP/prebuilts/clang/ \
host/linux-x86/llvm-binutils-stable:$PATH
```

Now you can generate the report. First, merge and format the data via `llvm-profdata`:

```
$ llvm-profdata merge --sparse profdata/* --output coverage.profdata
```

To create the report, it's critical that you use the unstripped binary containing the debug information, located in the symbols directory of the Soong output. These commands define the symbols path and generate the initial report:

```
$ FUZZ_SYMBOLS_DIR=$ANDROID_PRODUCT_OUT/symbols/ \
data/fuzz/$(get_build_var TARGET_ARCH)
$ SYMBOLS_PATH=$FUZZ_SYMBOLS_DIR/libskia_image_processor_fuzzer/
\ libskia_image_processor_fuzzer
$ llvm-cov show --format=html --instr-profile=coverage.profdata \
```

```
--path-equivalence=/proc/self/cwd/,$ANDROID_BUILD_TOP $SYMBOLS_PATH \
--output-dir=coverage_report
```

Open *index.html* from the *coverage_report* directory to see your report. You should see an overwhelming number of files in this report, many of which concern libraries with no coverage, and others that relate to low-level shared libraries not interesting for finding vulnerabilities in the image-processing code, such as logging. To remove these files and make the report easier to work with, use the flag `--ignore-filename-regex`, then generate a new report:

```
$ IGNORE_PATHS='
--ignore-filename-regex=.*/prebuilts/.*
--ignore-filename-regex=.*/bionic/.*
--ignore-filename-regex=.*/external/libyuv/.*
--ignore-filename-regex=.*/external/perfetto/.*
--ignore-filename-regex=.*/external/rust/.*
--ignore-filename-regex=.*/system/logging/.*
--ignore-filename-regex=.*/usr/include/.*'
$ llvm-cov show --format=html --instr-profile=coverage.profdata \
--path-equivalence=/proc/self/cwd/,$ANDROID_BUILD_TOP $SYMBOLS_PATH \
--output-dir=coverage_report_with_ignores $IGNORE_PATHS
```

You'll find your new coverage report at *coverage_report_with_ignores/index .html*. Once you've created a base set of paths to ignore, you can reuse them for most of your future fuzz targets, adjusting the list slightly to suit each new target's needs.

Now that you've removed the content you don't want to see, add in more of what you do want to see. Currently, the reports show only the code built into the fuzz binary, comprising the fuzz harness and statically linked libraries. To see shared libraries, you'll need to use the `-object` flag for each library you want to include.

View the list of all the available libraries with this command:

```
$ ls $ANDROID_PRODUCT_OUT/symbols/data/fuzz/$(get_build_var TARGET_ARCH)/lib/
```

Since we're running the test with an image-processing fuzzer, it might be nice to look at the PNG processing library by including *libpng.so*. To do so, use this updated `llvm-cov` command:

```
$ llvm-cov show --format=html --instr-profile=coverage.profdata \
--path-equivalence=/proc/self/cwd/,$ANDROID_BUILD_TOP $SYMBOLS_PATH \
--output-dir=coverage_report_with_objects \

-object $ANDROID_PRODUCT_OUT/symbols/data/fuzz/$(get_build_var
TARGET_ARCH)/lib/libpng.so \
$IGNORE_PATHS
```

If you open the new coverage report, you should see the library added, as in Figure 3-1.

Filename	Function Coverage	Line Coverage
proc/self/cwd/external/libpng/arm/arm_init.c	0.00% (0/1)	0.00% (0/18)
proc/self/cwd/external/libpng/arm/filter_neon_intrinsics.c	0.00% (0/8)	0.00% (0/249)
proc/self/cwd/external/libpng/arm/palette_neon_intrinsics.c	0.00% (0/3)	0.00% (0/85)
proc/self/cwd/external/libpng/png.c	1.49% (1/67)	0.53% (9/1688)
proc/self/cwd/external/libpng/pngerror.c	0.00% (0/19)	0.00% (0/309)
proc/self/cwd/external/libpng/pngget.c	0.00% (0/67)	0.00% (0/672)
proc/self/cwd/external/libpng/pngmem.c	0.00% (0/13)	0.00% (0/111)
proc/self/cwd/external/libpng/pngpread.c	0.00% (0/19)	0.00% (0/602)
proc/self/cwd/external/libpng/pngread.c	0.00% (0/40)	0.00% (0/2290)
proc/self/cwd/external/libpng/pngrio.c	0.00% (0/3)	0.00% (0/36)
proc/self/cwd/external/libpng/pngrtran.c	0.00% (0/48)	0.00% (0/3413)
proc/self/cwd/external/libpng/pngrutil.c	0.00% (0/56)	0.00% (0/2556)
proc/self/cwd/external/libpng/pngset.c	0.00% (0/43)	0.00% (0/1016)

Figure 3-1: A report showing coverage of libpng *files*

You can now see files from the libpng library. If you look closely, however, you'll see that the *libpng* files all list zero lines as being covered, except one file, which has nine covered lines. This is because we're forcing our fuzzer to find a valid PNG file to pass through the fuzz target by generating 1s and 0s from scratch. With more than 5,000 executions, our fuzzer would eventually discover the PNG format as well as the other image formats, but to make your fuzzing campaigns as effective as possible, you can give the fuzzing engine a valid PNG file to show it that such files are valuable to the program it's fuzzing.

If you look at the corpus included with this fuzzer, you'll see that the author has already included a sample PNG file. Let's generate new profile data, and this time supply the fuzzer with this example file. To avoid any issues that may arise from the other corpus seeds, we'll copy this file to our own corpus directory and feed that to our fuzzer:

```
$ adb shell
$ cd /data/local/tmp/libskia_image_processor_fuzzer
$ mkdir min_corpus
$ cp corpus/test6.png min_corpus/test6.png
$ LLVM_PROFILE_FILE=profdata/out.profraw ./libskia_image_processor_fuzzer \
-runs=5000 min_corpus
$ exit
$ cd $ANDROID_BUILD_TOP
$ FUZZ_TARGET=libskia_image_processor_fuzzer
$ FUZZ_SYMBOLS_DIR=$ANDROID_PRODUCT_OUT/symbols/ \
data/fuzz/$(get_build_var TARGET_ARCH)
$ SYMBOLS_PATH=$FUZZ_SYMBOLS_DIR/$FUZZ_TARGET/$FUZZ_TARGET
$ adb pull /data/local/tmp/$FUZZ_TARGET/profdata/ .
$ llvm-profdata merge --sparse profdata/* --output coverage.profdata

$ llvm-cov show --format=html --instr-profile=coverage.profdata \
--path-equivalence=/proc/self/cwd/,$ANDROID_BUILD_TOP $SYMBOLS_PATH \
--output-dir=coverage_report_with_objects \
-object $FUZZ_SYMBOLS_DIR/lib/libpng.so \
$IGNORE_PATHS
```

Figure 3-2 shows the results.

Filename	Function Coverage	Line Coverage
proc/self/cwd/external/libpng/arm/arm_init.c	100.00% (1/1)	61.11% (11/18)
proc/self/cwd/external/libpng/arm/filter_neon_intrinsics.c	62.50% (5/8)	65.46% (163/249)
proc/self/cwd/external/libpng/arm/palette_neon_intrinsics.c	0.00% (0/3)	0.00% (0/85)
proc/self/cwd/external/libpng/png.c	23.88% (16/67)	15.76% (266/1688)
proc/self/cwd/external/libpng/pngerror.c	36.84% (7/19)	29.45% (91/309)
proc/self/cwd/external/libpng/pngget.c	10.45% (7/67)	9.08% (61/672)
proc/self/cwd/external/libpng/pngmem.c	69.23% (9/13)	65.77% (73/111)
proc/self/cwd/external/libpng/pngpread.c	84.21% (16/19)	51.50% (310/602)
proc/self/cwd/external/libpng/pngread.c	12.50% (5/40)	3.97% (91/2290)
proc/self/cwd/external/libpng/pngrio.c	66.67% (2/3)	52.78% (19/36)
proc/self/cwd/external/libpng/pngrtran.c	10.42% (5/48)	4.75% (162/3413)
proc/self/cwd/external/libpng/pngrutil.c	37.50% (21/56)	17.61% (450/2556)
proc/self/cwd/external/libpng/pngset.c	11.63% (5/43)	12.01% (122/1016)

Figure 3-2: Coverage of libpng *with the PNG seed*

Compare the Line Coverage column with the one from the previous run. You should notice that, still with only 5,000 executions, most of the files now have coverage, meaning the fuzzer was able to explore this library much more effectively. By using the seed corpus, we drastically increased the effectiveness of our PNG library fuzzing.

Wrapping Up

This chapter provided a comprehensive overview of the essential tools and frameworks used in fuzzing. We covered how sanitizers like ASan and MTE are used to turn subtle memory errors into actionable crashes, explored the functionality of core fuzzing engines like libFuzzer and AFL++, and discussed advanced data-handling techniques using FuzzedDataProvider and libprotobuf-mutator. Finally, we walked

through the critical process of generating and analyzing code coverage reports, demonstrating how to use seeds to significantly boost a fuzzer's effectiveness in exploring target libraries like libpng.

OceanofPDF.com

4
AN OVERVIEW OF ANDROID SECURITY



The rapid growth of the Android operating system, surpassing iOS in 2010 and reaching its first billion users in 2013, makes it a prime target for attackers. Since its inception, Android has faced numerous challenges in protecting user data and privacy, prompting a continuous evolution of its security model and features.

In this chapter, we discuss Android's multilayered security architecture and its evolution to a platform. We examine hardware security mechanisms like trusted execution environments (TEEs) and secure elements (SEs) and explore system-level protections like SELinux, Project Treble, and the Android permission model. The topics covered here are by no means a comprehensive list of Android security features, but they all either relate to discussions in future chapters or are crucial to understanding Android's security model.

Layers of the Android Architecture

The Android architecture has five main layers, visualized in Figure 4-1.

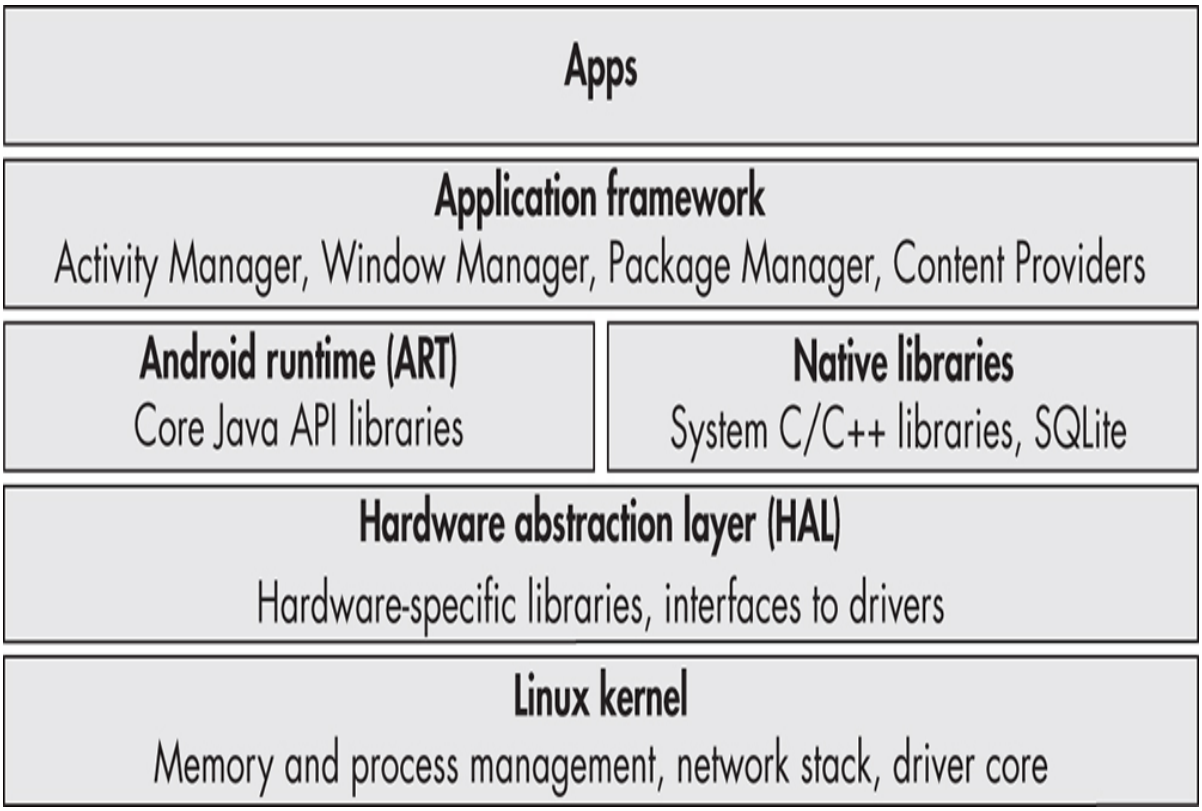


Figure 4-1: Android's software stack

Here is a description of each layer:

Linux kernel The foundation of the operating system, providing security services, memory management, process management, and the network stack. The kernel also includes drivers that interact with the device's hardware, enabling the display, camera, Bluetooth, and other functions.

Hardware abstraction layer (HAL) A standardized way for Android to interact with device hardware, acting as an intermediary between the hardware drivers in the Linux kernel and the higher-level software layers so those layers can remain agnostic about the specifics of the hardware implementation.

Android runtime (ART) The runtime environment for Android applications; executes the bytecode of Android apps.

Libraries A set of core libraries that provide apps with essential functionalities, including Java programming language libraries, the

SQLite database for data storage, and others.

Application framework Provides a set of APIs and services that developers use to build Android applications. Includes components like the Activity Manager, Window Manager, Package Manager, and Content Providers, which standardize the ways in which apps access system resources and interact with one another. Within this framework, the Activity Manager manages the life cycle of activities (the building blocks of Android applications) and handles the activity stack. The Window Manager controls the display and windowing system, including its size, position, and user input events. The Package Manager manages the installation, permissions, and updates of applications. Finally, Content Providers offer a standardized abstraction layer for apps to share data with one another.

Now let's dive into how Android security model has improved over time.

Android's Security Evolution

Android's security has evolved significantly since its early versions, which focused on functionality. Those versions had a less granular permission system, allowing applications to access a wider range of data and device features than necessary. Moreover, the early versions lacked robust sandboxing mechanisms, meaning a compromised app could potentially affect other apps or even the entire system. While some devices offered full disk encryption, that option wasn't a standard requirement until Android 5.

These issues led to attackers gaining root access to Android devices, enabling complete control over the system and user data. Notable attacks include GingerBreak and Exploit.AndroidOS.Lotoor. GingerBreak attackers gained access to the logfile to retrieve the address where `vold`, the volume manager daemon, crashed. They then used this address to overwrite the global offset table (GOT), enabling it to execute arbitrary code with root privileges. Google responded by

modifying the vulnerable code to prevent privilege escalation and the overwriting of the GOT. Exploit.AndroidOS.Lotoor similarly exploited another vulnerability in `vold` to gain root access.

Such early security issues highlighted the need for a more robust security model in Android. Google responded by gradually introducing security enhancements in subsequent versions:

Android 5

Integrated SELinux, a mandatory access control (MAC) system that enforces strict access controls on apps and processes, further limiting the potential damage from compromised apps. SELinux allows for fine-grained control over what each app can access and do, even if it manages to exploit a vulnerability. We cover SELinux in depth in "SELinux" on page 56.

Android 6

Introduced a granular app permission system that allowed users to grant or deny permissions individually. For example, an app might request permission to access the device location, user contacts, camera, or microphone; the user could then choose to grant or deny these requests based on their preference and comfort level. This change gave users control over their data and addressed concerns about apps potentially accessing sensitive information without explicit user consent. See "Permissions" on page 53 for more details.

Android 7

Introduced file-based encryption, allowing different files to be encrypted with different keys. This addressed the limitations of full-disk encryption, which protects all data with a single key. Other notable improvements during this period included *work profiles*, which separated personal and corporate data on devices; full-disk encryption, which protected data even if a device was lost or stolen; and regular security updates to promptly address vulnerabilities. Android 7 also gave each app more restricted access permissions (0700) over its private directory, preventing other apps from accessing or modifying them.

Android 8

Tackled the issue of permissions being incorrectly granted within the same permission group. Prior to Android 8, granting a permission to an app at runtime could inadvertently grant other permissions within the same permission group, even if the user didn't explicitly approve them. Android 8 required explicit user approval for each permission requested. Even if an app declared multiple permissions within the same group in its manifest, the user had to explicitly grant each permission individually at runtime.

Android 9

Further enhanced privacy by restricting *world-readable* files: those that any app or process on the device can read, regardless of who created them or what permissions they have. While the deprecation of world-accessible files began in earlier versions (notably with the removal of `MODE_WORLD_READABLE` support in Android 7), Android 9 fully enforced these restrictions to ensure that an app's private data remained truly private. This version of Android also addressed the growing concern of insecure data storage in applications, such as improper handling of sensitive information and the storing of user credentials or access tokens in plaintext, by introducing stronger sandboxes through SELinux and encouraging encryption through the Android Keystore.

Android 10

Introduced package visibility filtering, limiting apps' access to the list of installed packages. This change prevented malicious applications from gathering unnecessary information about other apps on the device.

Android 11

Focused on improving user privacy by introducing features like one-time permissions.

Android 12

Added microphone and camera indicators, giving users visual cues when these sensors were active. Android 12 also enforced safer

component exporting, preventing accidental exposure of app components. Previously, if a developer included an intent filter for an app component but did not explicitly declare the `android:exported` attribute, the Android system would default that attribute to `true`. This meant the component was automatically *exported* (made public) and accessible to other apps on the device, even if intended for internal use only. This allowed malicious apps to launch internal activities, services, or broadcast receivers within apps, potentially leading to unauthorized access to sensitive data or even code execution within the context of the other application. To mitigate the issue, Android required developers targeting Android 12 or higher to explicitly declare the `android:exported` attribute value of `true` or `false` for all activities, services, and broadcast receivers that included intent filters in their app's manifest file.

Android 13

Continued the trend of enhanced privacy and security with features like the photo picker, which allows users to share specific photos with apps instead of granting access to their entire photo library.

Android 14

Increased Android defenses against phishing attacks by introducing features that help users identify and avoid malicious websites and messages. This included improved detection of suspicious links in emails and messages, as well as the issuing of warnings when users attempt to enter sensitive information on potentially harmful websites. Android 14 also strengthened app sandboxing by implementing stricter restrictions on how apps can interact with one another and access system resources (blocking apps from sending implicit intents to internal, unexported components).

Android 15

Introduced an Advanced Protection mode designed for users at higher risk of targeted attacks. While many of its security measures, such as blocking app sideloading and disabling 2G connectivity, were already available as individual options within the system settings, Advanced Protection mode streamlined them into a single,

high-assurance state. This version also added support for MTE, a hardware feature that helps detect memory safety vulnerabilities in apps, which is particularly relevant for mitigating exploits found through fuzzing.

Android 16
: Further integrated protections by introducing the AdvancedProtection-Manager API. This allowed applications to programmatically detect that a user has enabled Advanced Protection mode. Rather than relying on the user to manually configure settings, apps can now use this API to automatically adjust their own security posture, such as requiring stricter authentication or restricting certain high-risk features when the device is in this enhanced state.

In the next sections, we'll dive deeper into the kinds of security features Android has added over the years, beginning at the hardware level.

Hardware-Based Security

Android's rich execution environment, designed for flexibility and general-purpose computing, inherently exposes it to vulnerabilities. To create separate, isolated environments in which to perform sensitive operations, the operating system relies on two kinds of hardware components: trusted execution environments (TEEs) and secure elements (SEs).

Trusted Execution Environments

A TEE is a secure area of a processor that runs in isolation from the main operating system. Prior to the adoption of TEEs around Android 5, developers and the Android system itself employed various techniques to protect sensitive data, each with its shortcomings.

One approach was to encrypt data via algorithms and keys stored in the operating system. The weakness of this strategy is that it relied heavily on the security of the operating system; if the main operating

system was compromised, the keys and the encrypted data became vulnerable. Another approach was to push all sensitive data operations to the server side, minimizing the amount of sensitive data stored on the device. Yet this approach added complexity and latency.

ARM's introduction of TrustZone provided the hardware necessary to create TEEs within ARM-based processors. TrustZone divides the processor into a "normal world" (the rich execution environment) and a "secure world" (the TEE) with hardware-enforced isolation, extending the ARM architecture to include a secure state. When the processor is in the secure state, it can access dedicated memory and peripherals. A component called *secure monitor* handles transitions between the normal world and the secure world, ensuring that only authorized code can enter the TEE.

The TEE itself runs a dedicated, minimal operating system with a small code footprint to reduce the attack surface. Applications that run within the TEE (called *trusted applications*), which are all carefully designed to ensure that they don't introduce vulnerabilities, handle sensitive operations such as key management, biometric processing, and digital rights management (DRM).

The TEE's operating system and its trusted apps go through a secure boot process that verifies the code's cryptographic signatures at startup to detect unauthorized or modified code. At runtime, the TEE environment often includes additional anti-tampering mechanisms that trigger if someone tries to modify the TEE's code or data while it's running.

To protect their apps' sensitive data, developers can use the TEE through APIs and libraries like Android Keystore. Separately, security services leverage the TEE for features like verified boot and SafetyNet to assess device integrity. Here is an example of a communication between an Android app and the TEE:

1. The Android app uses the Android Keystore API to request the generation of a new asymmetric key pair by specifying parameters such as key size.

2. The Android Keystore service (a part of the Android framework) receives the app's request and validates it to determine whether the app requires interaction with the TEE. The Keystore service then uses a driver to communicate with the TEE, as illustrated in Figure 4-2. The driver used depends on the device manufacturer's implementation of the TEE.

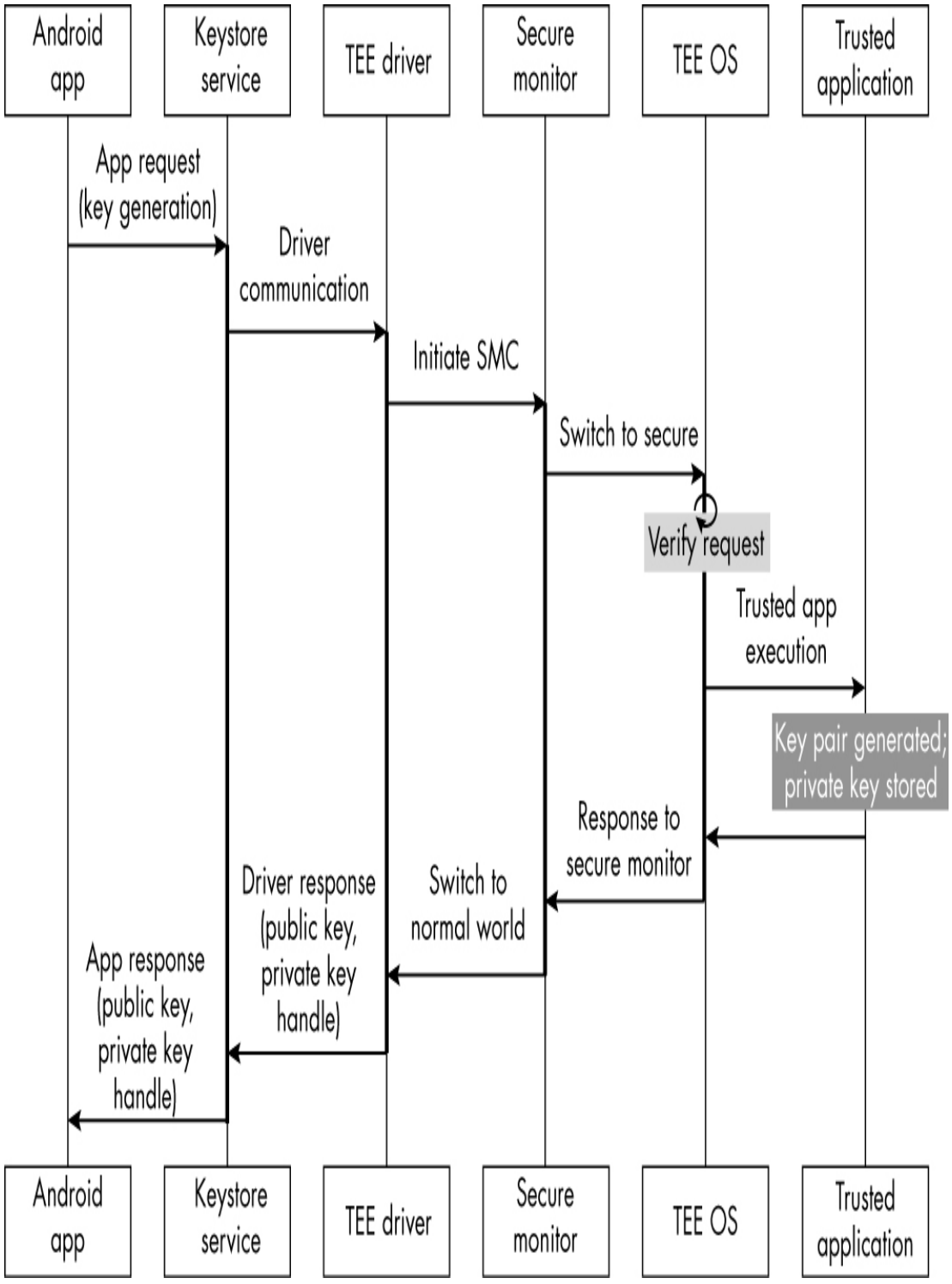


Figure 4-2: An application's interaction with the TEE

3. The driver initiates a secure monitor call (SMC), which triggers a switch from the rich execution environment to the TEE.

4. The TEE's operating system receives the SMC and the associated request, verifies that the request is authorized, and makes sure the caller has the necessary permissions.
5. The TEE's operating system identifies and loads the appropriate trusted application responsible for key generation and storage. This application generates the cryptographic key pair and stores the private key securely within the TEE's protected storage. This private key never leaves the secure environment.
6. The TEE's operating system passes the response (including the public key and a handle to the private key) back to the nonsecure world through the secure monitor. The response travels first to the TEE driver, which passes it to the Keystore service, which in turn hands it back to the requesting Android app. In this way, the app can use the key handle to perform cryptographic operations, even if the actual private-key operations occur within the TEE.

While the TEE offers a significant improvement in security, it has limitations. Notably, the TEE software could be vulnerable to attacks, compromising the security of the entire system. In particular, physical attacks such as side-channel attacks could potentially extract sensitive information from the TEE. The TEE is also still part of the main processor, which could have its own vulnerabilities.

Secure Elements

SEs are dedicated hardware components physically isolated from the main processor. Like TEEs, SEs communicate with the main processor through secure channels and can perform cryptographic operations such as encryption, decryption, and digital signing in a secure environment. Yet, with their own processor, memory, and secure storage, SEs are designed to be tamper resistant, making it difficult for attackers to extract information from them.

StrongBox, a software component introduced in Android 9, allows Android devices to use the capabilities of the integrated SE. It provides a secure storage solution for sensitive cryptographic keys, such as those used for biometric authentication, payment transactions, and app-

specific data. Although StrongBox has a software component, a dedicated hardware security chip mitigates vulnerabilities in this software by isolating sensitive data and operations from the main operating system.

To further enhance security, StrongBox applies *microsegmentation*, dividing the system into isolated zones with restricted communication in order to limit the impact of potential breaches. Finally, StrongBox implements runtime integrity verification to detect any unauthorized modifications to the system, such as rooting or jailbreaking.

Let's walk through an example of how an Android app can use a Strong-Box SE to generate and store a new key. First, the Android app must initiate a request to generate a new key, specifying the desired key type (for example, AES), the key size, and usage restrictions (for example, encryption, decryption, and signing). The app uses the Android Keystore system's `KeyGenParameterSpec` to define these parameters and indicates a preference for StrongBox backing with `setIsStrongBoxBacked(true)`.

Next, the Keystore system translates the app's request into a format suitable for the StrongBox SE and interacts with the Keymaster HAL to send the key-generation request to the StrongBox Keymaster. The Keymaster HAL sends commands to the StrongBox SE through a secure communication channel. The commands might include instructions for key generation, secure storage, and usage authorization.

The StrongBox SE receives the commands, generates the key via its internal true random number generator (TRNG), and stores it in its secure storage. The SE then sends a response to the Keymaster HAL, indicating whether the operation succeeded or failed. If it succeeded, the response might include a key handle that the app can use to refer to the generated key in subsequent operations.

After receiving the key handle, the app can use that handle to perform cryptographic operations. When the app needs to use the key, it provides the handle to the Keystore system. In turn, the Keystore system interacts with the Keymaster HAL and the StrongBox SE to authorize the usage and perform the requested operation.

Protected Virtual Machines

Building upon the foundations of TEEs, a protected kernel-based virtual machine (pKVM) leverages virtualization technology to create isolated execution environments with stronger separation than traditional TEEs.

Unlike TEEs or SEs, which rely on hardware-defined "worlds" or separate chips, pKVM leverages hardware virtualization extensions to create isolated execution environments called *protected virtual machines (pVMs)*. Thus, pKVM is the hypervisor (the "engine") that manages system resources, while a pVM (the "container") is the actual isolated environment where sensitive code runs.

Here are the key aspects of this relationship:

Hardware-enforced walls pKVM leverages the CPU's virtualization extensions to create a sandbox for the pVM. This ensures that even if the host Android operating system is compromised, a threat actor can't access the memory space of a running pVM.

The principle of mutual distrust Unlike a traditional TEE, where the "normal world" assumes that the "secure world" is superior, pKVM treats the host operating system and the pVM as separate entities that don't need to trust each other's kernels.

Microdroid In many implementations, a pVM runs a stripped-down version of Android called *Microdroid*. This minimal operating system provides just enough functionality to run a single task (like a security-sensitive computation) without the massive attack surface of the full Android stack.

BIOMETRIC AUTHENTICATION SECURITY

Android 9 introduced a biometric authentication framework, accessible through the BiometricPrompt API, to provide a

standardized way for applications to integrate biometric authentication. This framework uses the TEE or the Strong-Box, if supported, to store encrypted biometric templates (the digital representation of biometric data). Communication between the biometric HAL and the TEE or StrongBox occurs over isolated channels and uses cryptographic protocols for authentication.

Fragmentation

Android's widespread adoption has caused a persistent challenge: fragmentation, resulting from the various Android versions, devices, and customizations present in the market.

Because Android is open source, manufacturers can choose which Android features to include on their devices, modify the UI, and even add their own proprietary software. This has led to a wide array of Android experiences across devices. Mobile carriers, too, can add their own customizations, bloatware, and network-specific settings. Finally, unlike the iOS ecosystem, Android runs on a vast array of devices; this hardware fragmentation adds to the complexity of developing and testing apps that work consistently across all devices.

Android releases also contribute to fragmentation. While some users will update their devices to the most recent version, many will continue to use older ones, which in some cases are susceptible to known, unpatched vulnerabilities. A recent breakdown of Android versions in the wild shows Android 14 at 25.8 percent and Android 13 at 20.9 percent, but also users still running Android 6 and 5, at 0.7 percent each.

Accompanying these many versions, Android offers a vast number of APIs that allow developers to access operating system functionalities and hardware. These APIs change from release to release, and app developers will have to choose how far back to support their app. This can cause compatibility issues and make it difficult for developers to use the platform's full potential.

In such an environment, ensuring the security of outdated Android devices, which may not receive the latest security updates, is a major headache. This poses a security risk for users and can damage the reputation of the Android platform.

The Chinese Android Ecosystem

Chinese Android users add another layer of complexity to the fragmentation issue. China has a unique mobile ecosystem, with its own app stores, regulations, and user preferences. Chinese manufacturers like Huawei, Xiaomi, and Oppo heavily customize Android with their own UIs and services, creating a significant degree of fragmentation.

Manufacturers apply various levels of customization, however. All Android manufacturers (including Samsung, Nothing, Xiaomi, and Oppo) apply custom skins and add their own preinstalled apps, but they generally build these on top of the standard Android ROM, maintaining compatibility with the broader Android ecosystem. In contrast, Huawei has developed its own operating system, HarmonyOS, designed to be compatible with Android apps but offering a different user experience. While early versions of HarmonyOS relied heavily on AOSP to achieve compatibility with Android applications, Huawei has been actively working to reduce its dependence on AOSP, and newer versions emphasize its own microkernel-based architecture and system components.

Project Treble

Announced in 2017, Project Treble makes it easier for manufacturers to update devices to a new version of Android. Android first implemented Treble in version 8 but achieved full Treble compatibility in version 9. At that point, devices could seamlessly run a generic system image based on AOSP.

Project Treble's core concept is to decouple the Android framework's codebase (essentially, the AOSP) from the vendor's low-level, device-specific software: the parts that interact directly with the hardware components, such as drivers, HALs, and other code customized by

device manufacturers or silicon vendors to support their specific hardware and features. This separation allows for independent updates of the Android framework without requiring modifications to the vendor implementation.

To achieve this modularity, Project Treble introduced several key changes to the Android architecture, the result of which is shown in Figure 4-3.

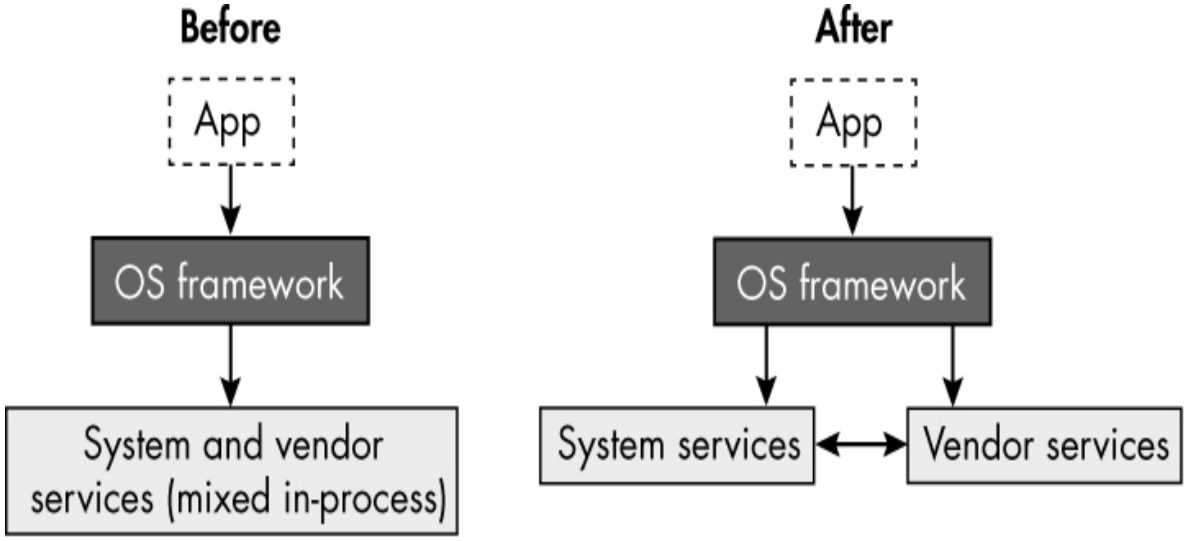


Figure 4-3: Changes introduced by Project Treble

The vendor interface now defines how the Android framework interacts with device-specific hardware components. The introduction of the Vendor Test Suite in Android 8 provided a set of tests that vendors could use to validate their implementation against the vendor interface, similar to the Compatibility Test Suite used for app compatibility.

Initially, Project Treble introduced a new *Hardware Interface Definition Language (HIDL)* specifically for HALs. This allowed the Android framework to communicate with vendor-specific code via a stable, versioned interface.

However, this created a split: Developers had to use HIDL for hardware but use the original Android Interface Definition Language (AIDL) for high-level system services. To simplify this situation, Google transitioned to using *Stable AIDL* for all HALs, starting in Android 11.

Unlike the original AIDL, Stable AIDL is designed to be backward compatible and serves as a direct replacement for HIDL. Today, HIDL is considered a legacy technology, and most modern Android hardware components are built using Stable AIDL to ensure a unified and consistent communication model across the entire platform.

Finally, Treble mandated a specific partitioning scheme for Android devices, separating the system, vendor, and product partitions. This separation ensured that updates to the Android framework wouldn't affect the vendor-specific code.

Permissions

Since its inception, Android has used a per-app permission model, whereby each application must declare the specific resources it needs to access within its *AndroidManifest.xml* file. However, its method of granting these permissions has changed significantly.

In early versions of Android, permissions were granted entirely at the time of installation. Users were presented with a list of all requested permissions before installation; if they chose to proceed, the app was permanently granted every requested permission.

Figure 4-4 compares this legacy permission model with the modern runtime model introduced in Android 6. The left side of the figure shows the legacy install-time permissions, while the right side demonstrates the modern runtime permission request dialog.

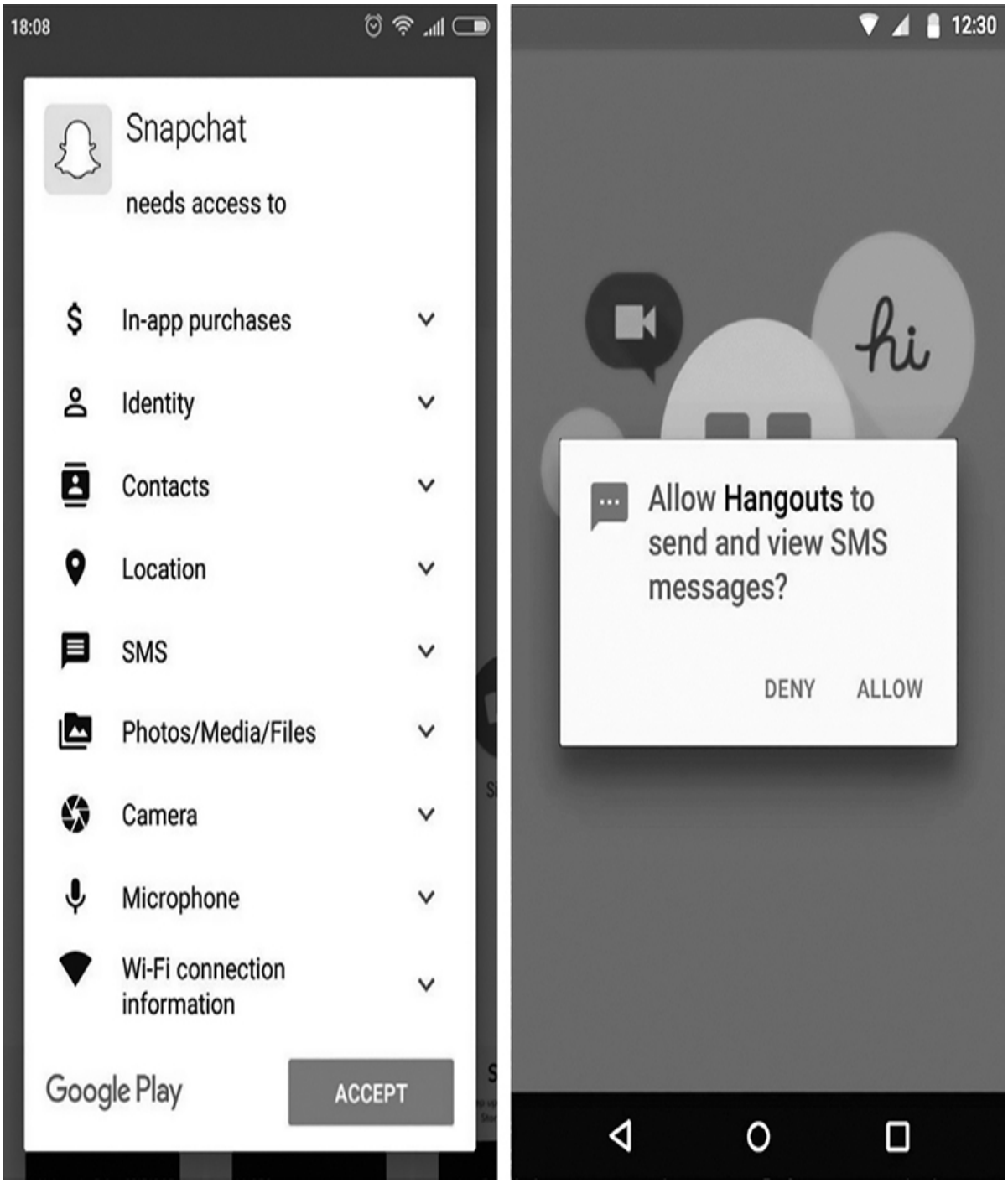


Figure 4-4: Install-time (left) and runtime (right) permission request dialogs

While developers still declare their intent to use a permission in the manifest, the actual granting of sensitive permissions now happens only when the app is running and attempts to access the resource for the first time. This allows users to grant or deny permissions individually,

providing much more granular control over their data. Modern Android now categorizes these permissions into two levels:

Install-time permissions These are granted automatically upon installation because they pose a limited risk to user privacy (for example, `ACCESS_WIFI_STATE`).

Runtime permissions These require explicit user consent at the moment of use because they involve sensitive data like location, camera, or contacts (for example, `ACCESS_FINE_LOCATION`).

Modern Android versions restrict install-time permissions to those that pose minimal risk to user privacy and the operation of other apps, and use runtime permissions to navigate more extensive access to data and actions.

Later versions of Android introduced contextual permissions and enabled users to grant access to resources under specific conditions, such as "only while using the app" or "for a single use," as well as granular media permissions, allowing users to specify access based on media type (such as images, videos, and audio).

Additionally, permissions can have associated flags that provide further information about and control over their behavior. *Protection-level* flags such as `appop`, `privileged`, and `preinstalled` indicate additional behavior related to how the permission is granted and managed by the system. For example, the preinstalled flag restricts the permission to preinstalled apps. *Permission* flags such as `costsMoney`, `hidden`, and `restricted` provide more context for a permission, such as whether it might cost the user money.

How Apps Request Permissions

Apps request permissions by declaring them in their manifest file with the `<uses-permission>` tag. For install-time permissions, this mere declaration suffices. For runtime permissions, apps must also request the permissions at runtime via the `requestPermissions()` method, which prompts the system to present a permission request dialog to the user explaining why the app needs the permission and asking for consent.

The workflow for requesting permissions has several key steps. To determine whether the app was already granted the permission, it should use the `checkSelfPermission()` method. Further, the app should gracefully handle situations where the user denies the permission request by providing alternative functionality or explaining the consequences of denying the permission.

Permission Escalation Attacks

Attackers often attempt to exploit vulnerabilities in the Android framework or kernel to gain unauthorized permissions. Here are some of their common tactics:

Social engineering Attackers may deceive users into granting extensive permissions by disguising malicious apps as legitimate ones. For example, ransomware might pose as a COVID-19 tracker app on a fake Google Play Store page and request excessive permissions under the guise of providing contact-tracing functionality.

Abusing accessibility services Attackers have exploited accessibility services, designed to assist users with disabilities, to intercept device notifications, capture screen content, or perform gestures on behalf of the user, allowing them to steal sensitive information or even take control of the device remotely.

Bypassing permission checks Hackers can employ various techniques to bypass permission checks and gain unauthorized access to resources. These techniques often exploit vulnerabilities in system services or manipulate inter-process communication (IPC) mechanisms.

SELinux

Prior to the introduction of SELinux in Android 4.3, the operating system's security model primarily relied on *discretionary access control (DAC)*, whereby each application runs with a unique user ID and has

access to files and resources owned by that user ID. While DAC provided a basic level of isolation, it had severe limitations, mainly related to its granularity; an app with access to a particular resource could potentially perform any action with that resource, even those not intended by the user. In a worst-case scenario, a compromised process running with root privileges could access any file or resource on the system.

To address these shortcomings, Android 4.3 introduced SELinux to provide *mandatory access control (MAC)*, a model that allows the operating system to enforce a more fine-grained security policy. SELinux operates on the principle of default denial; anything not explicitly allowed gets denied, restricting even processes running with root privileges to a limited set of actions and resources. In this introduction, SELinux was set to Permissive mode (only logged violations but did not block anything). Later in Android 4.4, SELinux was switched to Enforcing mode for core system domains, but not for applications. Finally, in Android 5, SELinux was fully enforced across the entire software stack.

SELinux also enhances application sandboxing by further defining the boundaries of the Android application sandbox. Previously, Android defined these sandboxes by creating a unique Linux user ID (UID) for each application during installation (DAC). However, if an application exploited a kernel vulnerability, the UID-based boundary could be bypassed. Android versions 4.3 and later use SELinux to further restrict applications by confining them to specific *domains*.

This second layer of defense ensures that even if a malicious app manages to run code as a privileged user, the SELinux policy acts as a "sandbox within a sandbox." It restricts the process to only the specific system resources (like device drivers or sockets) and application data folders that are explicitly labeled for that domain. For example, a typical user application running within the `untrusted_app` domain is physically unable to access sensitive system resources labeled as `system_data_file`, regardless of the underlying Linux discretionary access permissions.

How SELinux Works

SELinux works by setting a policy on various system objects, including files, directories, processes, ports, and devices. It also assigns each object a *security context*: a label that defines its security attributes and how it can be accessed. This context includes information like the SELinux user, role, type, and security level.

Each time an executable file launches a process, this process inherits the security context of that file, including the process's initial domain and permissions. During runtime, SELinux constantly monitors the process's actions. If the process attempts to access a resource or perform an action that's not allowed by its context or policy rules, SELinux denies the access and logs the event. (You can use the command `adb shell ls -Z /system` to view the security context labels of files and directories.)

An SELinux policy consists of a set of rules expressed in this format:

```
allow source target:class permissions;
```

The `source` is the type (or domain) of the process that initiates access, while the `target` is the type assigned to the object being accessed. In contrast, `class` refers to the kind of object (a functional category defined by the kernel, such as `file`, `directory`, or `unix_stream_socket`). While many objects can share the same security type, their class determines which specific actions (permissions) are available for that object. For example, a rule like `allow untrusted_app app_data_file:file { read write };` would allow an app running in the `untrusted_app` domain to read and write files labeled with the `app_data_file` type.

In Android, multiple factors determine an app's association with an SELinux domain, such as the app's location, permissions, and signing key:

Preinstalled and privileged apps These apps come with the device and are located in specific directories, like */system/app* or */system/priv-app*. They have elevated privileges and run in the `priv_app` domain.

Platform-signed apps These apps share the same signing key as core system components. They run in the `platform_app` domain.

System-UID apps These are platform-signed apps that run with the system user ID (UID 1000) and have the highest level of privilege. They run in the `system_app` domain.

Regular apps Apps installed by the user (typically) run in the `untrusted _app` domain with limited permissions.

The system prioritizes domain assignment in the following order: `system_app`, `platform_app`, `priv_app`, and `untrusted_app`. In addition to these default rules, device manufacturers can customize SELinux policies and assign specific domains to apps based on their package names or other criteria.

Like processes, resources such as files, directories, and sockets have security contexts that define their security attributes. When a process attempts to access a resource, SELinux compares the process's security context with the resource's security context and uses the policy to determine which domains are allowed to access which resources, and with what permissions. Figure 4-5 illustrates these policy enforcement steps.

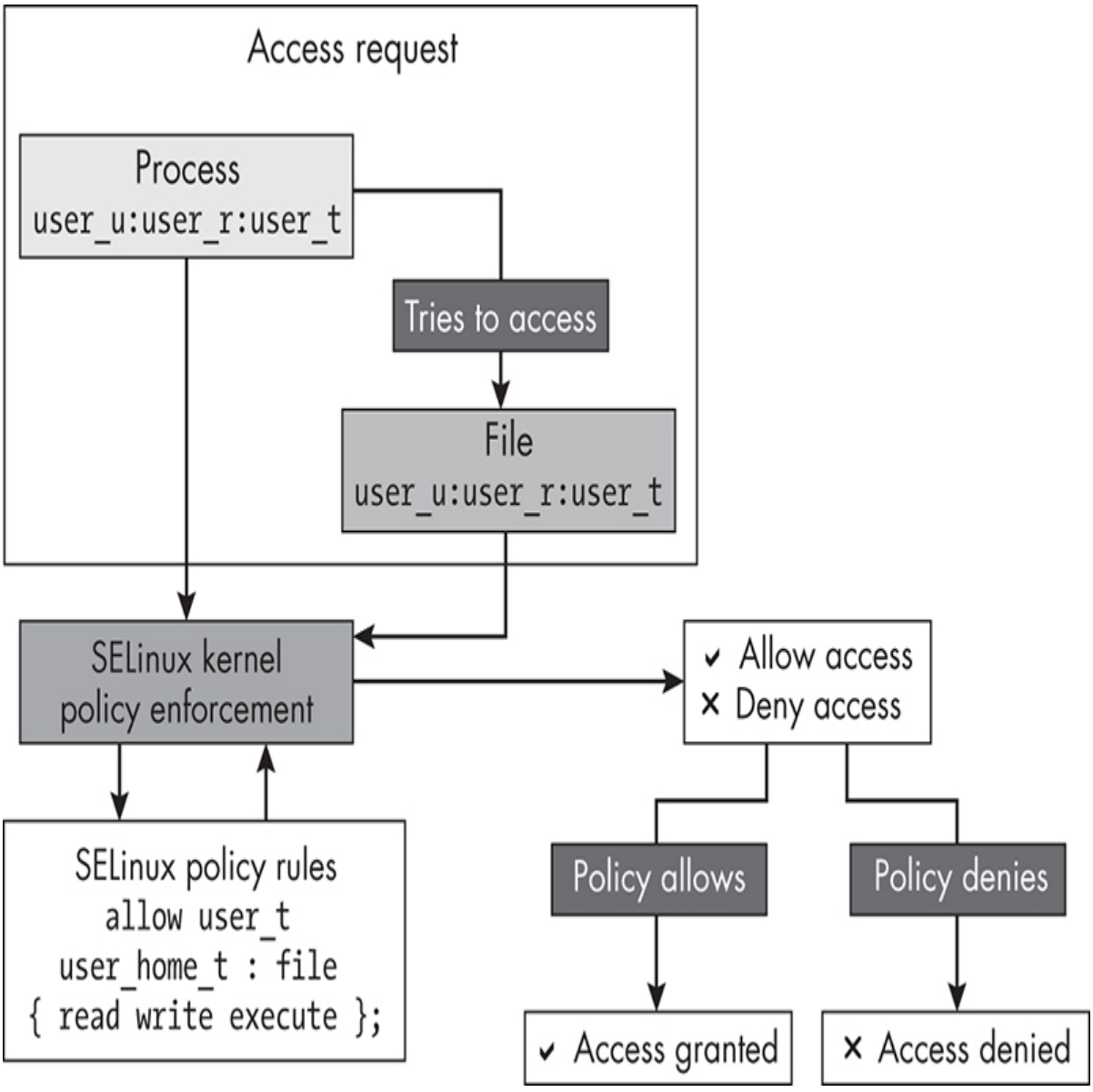


Figure 4-5: SELinux policy enforcement

Imagine that an application acting as the source process within the `untrusted_app` type attempts to access a system resource. In this case, the target object is a file labeled with the `system_file` type. As illustrated in Figure 4-5, the SELinux kernel checks the policy for a rule that matches both these types and the specific kind of object, known as the class. If the policy contains a rule such as `allow untrusted_app system_file:file { read };`, the access is granted. Without an explicit rule for that specific `file` class, the access is denied by default.

Android 12 and higher use categories to further isolate app data between apps and between different users on the same device. Categories can, for example, prevent apps from accessing data belonging to other apps even if running with the same user ID. Categories can also prevent one application from accessing the data of another application or the data of a different user on the device.

Policy Structure

An SELinux policy consists of a set of source files compiled together to generate a binary policy file that is loaded into the kernel. These source files include the following:

***.te* files** Contain the SELinux policy rules, written in a specialized language

file_contexts Assigns security contexts to files in the system

property_contexts Assigns security contexts to Android system properties

service_contexts Assigns security contexts to Android binder services

seapp_contexts Assigns security contexts to app processes and data directories

The policy compilation process has several steps, including converting the policy to an intermediate language, versioning the policy, and combining policy files from different partitions. As a result, the final policy file is consistent and compatible with the components of the Android system.

Tools

Android provides several tools for working with SELinux, including the following:

`chcon` Allows you to change the security context of a file or directory **`audit2allow`** Analyzes SELinux denial logs and generates

policy rules that would allow the previously blocked actions

`restorecon` Restores the default security context of a file or directory

These tools allow developers to identify and resolve policy violations, as well as to customize the security policy to meet specific needs.

Shortcomings

While SELinux has significantly enhanced the security of Android, it also has some shortcomings and limitations. First and foremost, SELinux policies can be complex to understand and manage, requiring specialized knowledge and tools. Enforcing SELinux policies can also introduce a slight performance overhead, although this is generally negligible on modern devices.

Most seriously, while SELinux provides a strong security barrier, attackers may be able to bypass it on rooted devices.

THE STAGEFRIGHT CRISIS: WHEN SELINUX WASN'T ENOUGH

The 2015 discovery of the Stagefright bugs in the `mediaserver` component was a pivotal moment for Android's security model. Stagefright involved a series of memory corruption vulnerabilities in media codecs, which are highly susceptible to fuzzing.

At the time, even though SELinux was active, `mediaserver` was a monolithic process with broad permissions. It needed access to the camera, microphone, and various hardware drivers to function. This created a significant security gap: Once an attacker used a fuzzed media file to hijack the process, SELinux alone couldn't stop them from using the process's legitimate privileges to conduct further attacks or exploit kernel vulnerabilities.

This crisis proved that SELinux was not bulletproof. In response, Google implemented two major architectural changes. First, Google broke `mediaserver` into several smaller, specialized sandboxes (such as `mediacodec` and `audioserver`). This allowed SELinux to be much more restrictive, as the `mediacodec` domain, for instance, no longer needed access to the camera. Second, because SELinux manages labels on objects (like files and sockets), it doesn't natively restrict the kind of system call a process can make to the kernel; therefore, Google added `seccomp` as a mandatory second layer of defense to harden the sandbox. Even if an attacker hijacks a process and bypasses its SELinux constraints, `seccomp` restricts the available system calls to a small subset, preventing the attacker from performing privileged actions like mounting filesystems.

App Signing

A feature of the Android platform since its inception in 2008, app signing ensures that all applications possess a digital certificate identifying the app's author before installation.

Initially, this feature relied on Java archive file (JAR) signing, a scheme that failed to protect all parts of the Android Package Kit (APK) file (notably omitting ZIP metadata) and had performance consequences. To address these issues, Android 7 introduced APK Signature Scheme v2, which offered faster verification and stronger protection against unauthorized modifications. Further advancements came with APK Signature Scheme v3 in Android 9, which added support for key rotation, allowing developers to change their signing keys without disrupting app updates. Key rotation is particularly useful when a signing key is compromised because it allows developers to switch to a new key and continue providing updates.

The signing process begins with developers generating a unique private key, which serves as their digital signature. During the app build process, the developer uses this private key to create a digital signature, which they embed within the app's APK. When an app is installed, the

Android system verifies the digital signature via the developer's corresponding public key. When the app is updated, the system checks whether the new version is signed with the same key as the original, preventing the distribution of fake updates.

If an app is distributed through the Google Play Store, Google manages and protects the app's signing key, adding an extra layer of security.

Wrapping Up

This chapter provided a look at how Android security has evolved. We explored the layers of the Android architecture, from the Linux kernel to the application framework, then described how the platform manages security across the stack.

We discussed Android's reliance on dedicated hardware components such as the TEE and SE, which handle sensitive operations and secure cryptographic keys to provide stronger isolation than the main operating system.

We also examined the role of SELinux, used for MAC, and app signing, which verifies application integrity. Finally, we discussed the introduction of Project Treble, part of Android's ongoing efforts to address the challenges of fragmentation.

Now that we've covered Android's core security components, you should be ready to begin identifying and exploiting vulnerabilities in it with fuzzing.

OceanofPDF.com

PART II
FUZZING IN PRACTICE

OceanofPDF.com

5
CHOOSING YOUR TARGETS



The Android ecosystem is in constant flux, as developers routinely introduce new features, architectural improvements, and possible attack vectors. While your fuzz targets should reflect these changes, choosing effective ones can be difficult if you don't understand the code under test. Base your strategy on recent vulnerability trends, real-world exploits, major architectural changes within the Android platform itself, and advancements in fuzzing methodologies and tools.

In this chapter, we discuss our approach to identifying and prioritizing high-value fuzz targets in the Android userspace by selecting and collecting metrics about AOSP libraries to determine what code to fuzz. By the end of the chapter, you should be able to create a dashboard of libraries and components that are ranked based on their importance for fuzzing.

Library Evaluation Metrics

To select Android libraries to fuzz, we must define a set of measurable signals that represent each library's risk profile. We've chosen metrics that reflect two primary domains: *reachability* (the number of paths leading to the code) and *instability* (how frequently the code evolves and its potential for regression). By quantifying these signals, we can transform the AOSP source tree into a weighted graph that assigns each library a priority based on its potential impact on the system's security posture.

In this chapter, we'll evaluate libraries based on the following factors:

Dependency count The number of unique modules that depend on the library

Commit recency and longevity The time elapsed since the first and most recent commits

Contributor diversity The number of distinct authors involved in the project's history

Code churn The volume of lines changed or inserted within a specific time frame

Vulnerability history Past security performance and Common Vulnerabilities and Exposures (CVE) density

Privilege level Whether the library is loaded by a process with high-level UIDs such as *root* or *system*

The Number of Dependent Libraries

The criticality of a library is often proportional to its position in the software stack. In AOSP, low-level system libraries serve as the foundational infrastructure for higher-level Java APIs, HIDL/AIDL interfaces, and HAL implementations. A vulnerability in a library that many other libraries depend on, such as libutils or libcutils, represents a systemic risk, because the flaw could be triggered in many possible ways.

Finding dependent libraries can be tricky, as not all files in an AOSP folder necessarily belong to the same library. Recall that Android uses the Soong build system to define libraries, building the corresponding binaries with blueprint (*.bp*) files. A folder in AOSP may contain one build file that defines multiple Android libraries.

To perform a quantitative analysis of reachability, we must extract the metadata for every module in the system. The most robust source for this data is the `module-info` target, which generates a comprehensive JSON dictionary of the entire build tree.

To generate the metadata file, run the following command in your AOSP root directory:

```
$ m module-info
```

The output path varies by `lunch` target. For a standard Cuttlefish build, the path is located at *out/target/product/vsoc_x86_64/module-info.json*.

We then use the open source command line tool `jq` to parse this dictionary. Because *module-info.json* is indexed by module name, we use `jq` to iterate through all values, select only those where the target library (such as libutils) appears within the `dependencies` array, and calculate the length of the resulting list to determine the total reachability count:

```
$ cat out/target/product/vsoc_x86_64/module-info.json |
jq '[.[] | select(.dependencies[]? == "libutils")] | length'
```

By calculating the in-degree of various libraries in this manner, we can create a ranked view of the most critical components in the system. A library with a high reachability count represents a significant security priority; a single vulnerability within the library could potentially be exploited through hundreds of entry points across the operating system.

Reachability via Android Public APIs

While dependency reachability at the build level maps compile-time dependencies, a critical metric for a security-focused dashboard is *exposure to third-party applications*. A low-level C/C++ library that can be triggered by an unprivileged application invoking a standard Android API presents a significant threat surface.

Tracing reachability from the public Java layer down to native source components requires mapping the three core layers of the Android inter-layer architecture:

The API-to-service mapping Public classes in the Android SDK (such as `android.media.MediaPlayer`) act as IPC wrappers. They route application requests via Binder interfaces to privileged system components inside `system_server`.

The JNI gateway Once execution transitions to the target service implementation, the system relies on the Java Native Interface (JNI) to invoke performance-critical native logic. These boundaries are defined in the framework via `native` method declarations.

Native symbol registration In the native subsystem, these JNI entry points are mapped to target C++ functions via a `JNINativeMethod` array structure.

To programmatically extract this metric, we calculate a weighted *API proximity flag*. We parse the `JNINativeMethod` registration structures across AOSP to identify all native functions directly exposed to the framework. If a native function symbol inside a target library appears in a JNI registration table that traces back to a public-facing framework class, the library is flagged as having direct public exposure.

For instance, the native function that handles media parsing in libstage-fright maps directly back to the public MediaPlayer API. This coupling between untrusted third-party inputs and native code execution drastically elevates the library's priority status in our prioritization metrics.

The Days Since the First and Last Commit

Time-based metrics provide insight into the maturity and maintenance state of a library's codebase. A library that hasn't had a commit in years might be stable (or perhaps neglected, lacking modern security hardening). In contrast, recent high-frequency commits suggest the rapid introduction of new features, which can often lead to security regressions. By measuring the change between the first and last commits, we can differentiate between battle-tested legacy code and volatile new components.

Two primary approaches can be used to extract these timestamps. We can consult the local checkout by running Git commands via `repo forall` on a local copy of the source tree. Alternatively, we can use the Gitiles REST API to directly query the AOSP host, without requiring a local sync.

The following examples demonstrate the Gitiles REST API approach. Note that Gitiles prepends a "magic line" (`)]}'`) to its output to prevent cross-site script inclusion (XSSI) attacks. Furthermore, the API returns timestamps as formatted strings rather than Unix integers. To calculate the mathematical distance between commits, you must strip the security header and parse the date string into a Unix timestamp with `jq`.

To find the last commit, query the log of the `main` branch and extract the most recent commit time:

```
$ REPO="https://android.googlesource.com/platform/frameworks/base"
$ curl -s "$REPO/+log/main?n=1&format=JSON" \
    | sed '1d' \
    | jq '.log[0].author.time \
    | sub(" -[0-9]{4}$"; "") \
    | strptime("%a %b %e %T %Y") \
    | mktime'
```

To find the first commit, use the same API but append the `reverse` parameter to jump to the beginning of the repository's history:

```
$ REPO="https://android.googlesource.com/platform/frameworks/base"
$ curl -s "$REPO/+log/main?n=1&format=JSON&reverse=1" \
    | sed '1d' \
    | jq '.log[0].author.time \
    | sub(" -[0-9]{4}$"; "") \
    | strptime("%a %b %e %T %Y") \
    | mktime'
```

This command pipeline performs three critical tasks: It uses `sed` to remove the XSSI prefix, `sub` to strip the time-zone offset, and the combination of `strptime` and `mktime` to convert the human-readable date into a comparable integer.

After extracting the Unix timestamps (T_{first} and T_{last}), we find the distance from the current system time (T_{now}). The following calculates the days since the last commit (D_{last}):

$$D_{last} = \frac{T_{now} - T_{last}}{86{,}400}$$

We should also calculate the total project lifespan (L_{total}):

$$L_{total} = \frac{T_{last} - T_{first}}{86{,}400}$$

These values allow for the categorization of components into maintenance profiles, as shown in Table 5-1.

Table 5-1: Library Profiles Based on Commit Frequency Metrics

Metric profile	Classification	Security assessment
High L_{total}, high D_{last}	Stable or legacy	High risk (missing modern hardening)
Low L_{total}, low D_{last}	Volatile or new	High risk (may cause logic regressions)
High L_{total}, low D_{last}	Active maintenance	Low risk
Low L_{total}, high D_{last}	Abandoned	Medium risk

This classification helps prioritize security research efforts; for instance, a stable or legacy component may require a focus on missing modern mitigations, whereas a volatile or new component warrants a deep dive into recent logic changes.

The Number of Distinct Contributors

While time-based metrics reveal the rate of change and the recency of a library, they don't account for the breadth of the human capital behind the code. The number of distinct contributors can reflect a project's level of oversight and maturity. Particularly in open source communities, a library maintained by a single author may be highly performant but represent a significant security risk; if that author is compromised or loses interest, they may not catch subtle logic flaws or maintain modern security hardening. A community of contributors is more likely to identify and patch vulnerabilities.

To programmatically find the number of distinct contributors across the entire AOSP source tree, begin by using `repo forall` to run a series of commands within each Git repository managed by `repo`. Within each project's directory, extract author emails by executing `git log --all --format="%ae"`. The `--all` flag ensures that you scan all branches in a project's history, while `--format="%ae"` tells Git to output only the author's email address for each commit.

However, because a single contributor may have hundreds of commits, the raw log will contain significant redundancy. To derive a count of unique individuals, you must pipe the aggregated output through `sort` and `uniq` before performing the final line count:

```
$ repo forall -c 'git log --all --format="%ae"' | sort | uniq | wc -l
```

Note that `sort` is a prerequisite for `uniq`, as the latter identifies only duplicate entries that are adjacent to each other. This pipeline transforms a massive list of individual contributions into a precise metric of contributor diversity, allowing you to identify which components benefit from a broad base of maintainers versus those managed by a small, potentially siloed team.

We can leverage the Unix pipeline to directly aggregate results, avoiding the need for manual redirection or temporary files. By piping the output of the `repo forall` command into `sort -u`, we de-duplicate the list of contributors in a single pass. Finally, we use `wc -l` to count the unique lines, each representing a distinct contributor email address found across the entire AOSP checkout:

```
$ repo forall -c 'git log --all --format="%ae"' | sort -u | wc -l
```

This streamlined approach provides a high-level view of developer diversity across the platform.

The Number of Lines Changed and Inserted

The number of new lines of code is perhaps the most direct indicator of potential instability within a library, representing new logic that hasn't yet been subjected to the hardening effect of prolonged production exposure.

The rapid pace of feature development often outstrips the ability of manual code review to identify subtle memory corruption bugs or logic flaws. Furthermore, high churn often indicates that a library is undergoing a significant refactor, which is a common period for the introduction of regression vulnerabilities.

To find the number of lines changed in a specific AOSP folder or library over the last 12 months, navigate to the Git project that contains the target path in your AOSP checkout, then run `git log` within that repository, filtering by date and path. You can then use the `--numstat` option to get the number of added and deleted lines for each commit and sum these numbers across all relevant commits.

The Vulnerability History

A library's past performance is often the most reliable predictor of its future security failures. Code with a dense history of vulnerabilities typically indicates an underlying architectural weakness, a complex state machine that is difficult to implement safely, or an otherwise bug-prone legacy codebase. By analyzing historical trends, we can identify repeat offenders: components with incomplete patches or the discovery of one bug class that suggests the existence of others.

One way to find a library's vulnerability history is to search for the library name or keywords in the Android Security Bulletins at *https://source.android.com/security/bulletin*. You could also check MITRE's CVE data at *https://cve.mitre.org*. Limiting your search to the last 12 months may be worthwhile, as old vulnerabilities may not reflect the current state of a library, and legacy vulnerabilities shouldn't impact the library's overall vulnerability score.

After you find the vulnerabilities associated with a library, check whether any of them has a possibility of remote exploitation, which would increase its level of criticality. To determine this, you can parse the JSON for a particular CVE in the MITRE CVE list to look for its Common Vulnerability Scoring System metrics, typically located in the `containers` section.

Check the field named `attackVector`, which should tell you the access method required for exploitation. For example, `"attackVector":"LOCAL"` indicates that the threat actor must have local access to the vulnerable device. A CVE is considered a remote access vulnerability if the `attackVector` is `"NETWORK"`.

Whether the Library Runs in Privileged Process

An exploit's impact is directly tied to the privilege level of the compromised process. In the Android security model, a vulnerability in a library loaded by a root or system daemon could allow for total device takeover, whereas the same bug in an unprivileged app sandbox is constrained by the application's specific permissions. Identifying libraries that are active in high-privilege contexts, such as *radio*, *bluetooth*, or *media*, allows us to weight our fuzzing targets based on the potential reward an attacker gains upon successful exploitation.

However, determining a library's level of privilege is challenging. One approach is to use `adb shell` to dynamically check which processes load a specific library on a running Android device. Begin your script by looping through all numerical directories in */proc*, as each number corresponds to the process ID (PID) of a running process. For each PID, attempt to read the */proc/<PID>/maps* file, which lists the memory regions mapped into the process's address space, including loaded libraries. Then, use `cat` to read the *maps* file and pipe the output to `grep -q "$LIBRARY_NAME"` to quietly check whether the library's name exists anywhere in the memory map for that process.

If `grep` finds the library name in the maps file, have your script print a message indicating that this PID has loaded the library and use `cat /proc/PID/cmdline` to attempt to show the command line of the process. Finally, run `ps -p PID -o USER,PID,NAME` to display the UID, PID, and name of the process. Accounts like *root*, *system*, *radio*, *bluetooth*, and *media* are privileged. Most applications run as *u0_aXXX*, *u10_aXXX*, and so on, which are unprivileged.

Calculating a Criticality Score

After collecting the individual metrics described in the previous sections, you'll likely find that a library's many signals are in conflict. For instance, a library might run in a highly privileged process but have zero lines of code churn over the last year. To make sense of these conflicting indicators, we must normalize and weight each factor to produce a single, actionable value. This criticality score will allow us to rank thousands of libraries on a uniform scale.

Employ common sense to determine the relative importance of each metric; for example, running in a privileged process may be more important than the number of distinct contributors, and being remotely accessible is likely more critical than the date of the first code commit.

After assigning weights, you must produce one value that represents the relative importance of the library for fuzzing. We chose a criticality scoring method inspired by the open source Criticality Score project, customized for Android libraries. The following produces a number from 0 (least critical) to 1 (most critical)

$$C_{library} = \frac{\sum_i w_i \frac{\log(1+S_i)}{1+\max(\log(1+S_i),\ \log(1+T_i))}}{\sum_i w_i}$$

where:

- w_i is the weight of each factor.
- S_i is the value of the ith factor.
- T_i is the max for each factor.
- $C_{library}$ is the score of the ith library.

The primary objective of this formula is to normalize different metrics into a uniform range [0, 1] while accounting for the nonlinear nature of software metadata. We use a logarithmic transformation, log(1 + S_i), to dampen the influence of outliers; in the context of AOSP, metrics such as dependency counts or lines of code can vary by several orders of magnitude. Without this logarithmic scaling, a single high-churn library would disproportionately dominate the rankings, masking other critical targets.

The denominator of the individual factor, 1 + max(log(1 + S_i), log(1 + T_i)), acts as a ceiling. By comparing the observed value S_i against a predefined maximum threshold T_i, we ensure that once a library reaches a certain level of criticality for a specific factor, its contribution to the final score plateaus. Finally, the global denominator $\sum w_i$ averages these weighted factors, ensuring that $C_{library}$ remains a relative probability score. This allows security teams to compare libraries of vastly different sizes and functions on a single, unified risk leaderboard.

As a test, run the numbers, look at the top 100 libraries, and assess the results. From experience, you may expect certain libraries to show up; examples include *framework*, *media*, *connectivity*, and *BT/NFC*. This is because these components manage complex, untrusted data streams (such as video codecs or network packets) and often operate within high-privilege execution contexts like `mediaserver` or `system_server`, inherently maximizing their criticality scores across all weighted dimensions.

If these libraries don't appear at the top, try altering the weights so those libraries generate higher scores. Also look for libraries that shouldn't have appeared in the top list, such as testing utilities or internal tooling, and adjust your filters or weights to penalize them.

Table 5-2 provides an example snapshot of these metrics, ranging from dependency reachability to commit frequency.

Table 5-2: AOSP Library Prioritization Dashboard (Recalculated Metrics)

Library path	Critical CVEs	High CVEs	Remaining CVEs	Days since last commit	Privilege level	Total score

Library path	Critical CVEs	High CVEs	Remaining CVEs	Days since last commit	Privilege level	Total score
system/core/libutils	1	17	No	42	Root	0.961
frameworks/native/libs/binder	1	10	Yes	3	System	0.948
frameworks/av/media/libstagefright	2	9	Yes	34	Media	0.924
packages/apps/Nfc/nci/jni	1	0	Yes	33	Privileged	0.887

We've modulated the metrics by the specific weights assigned to each factor to produce a prioritized ranking of core AOSP components.

Determining a Library's Critical Components

Once you've identified a high-value library, the next challenge is to choose the library code to test. Indiscriminately fuzzing an entire library is often inefficient, because much of the codebase may perform tasks that are unreachable or irrelevant to an attacker. To maximize your bug finding, figure out which files and functions handle untrusted or complex data, where vulnerabilities are most likely to hide.

For example, the libpng image-processing library can parse, load, and generate new image files. From a fuzzing point of view, the code that generates an image isn't particularly interesting, because the data being processed is typically produced by the system itself or trusted application logic, meaning the library is transforming a known good state into a predictable output format. In this scenario, the trust boundary hasn't been crossed.

By contrast, code that is too complex to read can easily be prone to vulnerability and hence should be prioritized. Complexity can be measured in multiple ways, including cyclomaticity, maintainability, and cognitive complexity.

Cyclomaticity, the most well-defined complexity metric, is a quantitative measure of the number of linearly independent paths through a program's source code. Specifically, it's derived from the control flow graph of the code, where the complexity M is calculated as follows:

$$M = E - N + 2P$$

Here, E represents the number of edges, N the number of nodes, and P the number of connected components. (Usually, $P = 1$ for a single function.) In simpler terms, we're counting the number of decision points (such as `if`, `while`, and `case` statements) within a function. The more paths that exist, the harder it is to manually verify all possible states, and the more likely that a fuzzer will find an unhandled edge case.

Measuring Complexity with Lizard

To automatically estimate the cyclomatic complexity across the vast AOSP source tree, we can use *Lizard*, an open source, language-agnostic code complexity analyzer. Unlike heavier static analysis tools, Lizard is designed for speed and can process millions of lines of code without requiring a full build environment.

We will use Lizard to generate a cyclomatic complexity estimate per file. Our workflow will focus on C and C++ codebases and will flag files with a cyclomatic complexity of 15 and above as "high complexity." Functions exceeding this limit are statistically more likely to contain deep logic flaws that are difficult to reach through standard unit testing but are prime targets for coverage guided fuzzing.

To run a scan on a specific library directory, you can use a command similar to the following:

```
lizard -C 15 ./frameworks/native/libs/ui/
```

The output provides a per-function breakdown, allowing you to pinpoint the exact sections within a library that warrant the most intensive fuzzing resources.

Finding Dangerous Operations

Flag the use of historically dangerous or banned functions if they appear within critical functions. While the following is not an exhaustive list, common candidates for inspection include `atoi()`, `atol()`, `atoll()`, `atof()`, `gets()`, `strcat()`, `strcpy()`, and `sprintf()`. Because the landscape of insecure C/C++ functions is vast, you should treat these as representative examples of risky patterns. For a comprehensive catalog of functions to avoid, refer to the CERT C Coding Standard or the Microsoft Banned Function List.

These functions are categorized as dangerous because they lack safety mechanisms for handling boundary conditions or malformed input. Specifically, the `str*()` and `sprintf()` family of functions do not perform automatic bounds checking, making them a primary source of buffer overflows. If an attacker can control the length of the source string, they can overwrite adjacent memory on the stack or heap.

The `atof()` and `atoi()` family is equally problematic; these functions provide no way to distinguish between a successful conversion of the string "0" and a failure due to invalid input. More importantly, they do not handle integer overflows or underflows. When a fuzzer provides an extremely large numeric string to a function that uses `atoi()`, it can trigger an integer wraparound that leads to an undersized memory allocation, a classic precursor to a heap buffer overflow.

By identifying these functions within the code paths reached by your fuzzer, you can prioritize sink points where untrusted data is likely to cause memory corruption. Modern secure coding standards recommend replacing these with safer alternatives like `strlcpy()`, `snprintf()`, or `strtol()`, which provide explicit length limits and error reporting.

Finally, prioritize code that parses structured input. Structured data formats such as XML, JSON, Protobuf, or complex media headers require the implementation of intricate state machines and nested logic. For example, if a structured input defines a length field for a subsequent data blob, a fuzzer can provide a massive value that exceeds the actual buffer size or a negative value that triggers an integer underflow. Parsers often involve deep recursion or complex loop structures; if these are not bounded correctly, they can lead to stack exhaustion or infinite loops.

Because Android relies heavily on structured communication via Binder and various serialized formats for media and configuration, these parsers constitute the most fertile ground for finding high-impact vulnerabilities.

OTHER POSSIBLE METRICS

In addition to the metrics we've discussed, here are some other areas we haven't quantified or included in the earlier ranking but that could be interesting to look into:

Reachability analysis We mentioned that not all code within a library is worth fuzzing, as some code belongs to private methods that can be invoked only internally, while other parts of the code are for testing or for tooling. Reachability analysis can help determine whether the vulnerable code path within a library is actually callable or reachable from an Android application. A vulnerability in an unused or unreachable function is a low-priority risk, whereas one in a core, frequently invoked function is critical. This capability is essential for reducing alert fatigue and focusing developer effort where it matters most.

Contributor health One practical factor in deciding whether to fuzz an Android library is whether the maintainer or contributors to this library are interested in fixing whatever bugs you will find. This is because in some cases, the library maintainers are busy with major feature work and do not have the bandwidth to fix security issues. The other consideration is the future of a library. If a library is being slated for replacement or retirement, maintainers may feel less incentivized to fix issues.

Before You Begin Fuzzing

Successful fuzzing also requires operational maturity. These considerations aren't signals to be weighted in a formula but strategic lessons we've learned by conducting large-scale fuzzing campaigns within AOSP.

First, avoid host-only fuzzers. The AOSP source tree is expansive, containing not just the Android runtime, but also a significant volume of build-time tools, host-side test runners, and administrative binaries like the `mksquashfs` and `soong` utilities. While these tools may contain vulnerabilities, they don't ship on the production device and are typically executed in a trusted build environment. Fuzzing these components represents a poor return on investment.

Second, it's crucial to engage with the team that owns the library before you begin fuzzing it to learn more about the code and get the team's commitment to fix issues after they're found. You should aim to answer key questions like these, which will shape your fuzzing harness:

What is the entry point for untrusted data? Every library has a "front door" where data from the outside world first touches internal logic. Identifying this function is the difference between fuzzing the entire library and fuzzing the specific parser where bugs live.

Are there any existing unit tests or "seed" inputs? Fuzzers perform significantly better when provided with a corpus of valid inputs. Asking for existing test cases allows you to start your fuzzer from a known good state, allowing it to mutate toward unknown bad states more efficiently.

What are the known "dark corners" of the codebase? Developers often know which parts of their code are the most fragile or which legacy sections were written before modern

security standards. These “dark corners” are the first places your fuzzer should explore.

How does the library handle state and global variables? Fuzzers like libFuzzer run in process and require the code to be reset after every iteration. If a library has a significant global state that isn’t easily cleared, it can lead to false positives or nonreproducible crashes that waste time.

By answering these questions early, you ensure that your fuzzing harness is not just hitting the code, but hitting it in a way that maximizes coverage and respects the library’s architectural constraints. This collaborative approach significantly reduces the time to remediation (TTR) after a vulnerability is confirmed.

Wrapping Up

This chapter discussed a systematic, data-driven approach to choosing and reevaluating your fuzz targets within the vast AOSP ecosystem.

By implementing the dashboard described in this chapter, you ensure that your fuzzing campaign will always focus on the front lines of the Android attack surface. As the platform evolves, with new modules being added and legacy code being refactored, your targets will naturally shift, keeping your security efforts aligned with a device’s actual risk profile.

Now that we’ve identified what to fuzz, the next logical step is to determine how to fuzz it. In the following chapter, we’ll dive into the technical implementation of fuzzing harnesses, taking our prioritized list of libraries and subjecting them to crash-seeking security tools.

OceanofPDF.com

6
WRITING YOUR FIRST ANDROID FUZZ HARNESS



In this chapter, we'll look at a real library in the AOSP, libpng, identify some good parts of it to fuzz, then iteratively write and improve our fuzz harness until we see that it's doing a reasonable job of exercising the library's code. While the example we've chosen is approachable, it will also present several common challenges you'll encounter when writing real-world fuzz harnesses. You'll learn how to integrate a dictionary and seed corpus to improve the fuzzer's effectiveness, analyze the output, and address common issues like memory leaks.

The chapter assumes you have completed the setup covered in Chapter 2 and have an Android device or Cuttlefish instance connected to your computer. The examples in this chapter will use an x86_64 Cuttlefish virtual device but should work with minor changes for physical devices running on other architectures.

Fuzzing a Library

One challenge when writing an effective fuzz harness is that it requires a deep understanding of the code you're trying to fuzz. In practice, learning about the existing code may take even longer than writing the fuzz harness itself. Let's walk through fuzzing the libpng library, which should let you dip your toes into the fuzzing process without having to learn absolutely everything about PNG images.

Creating an Empty Fuzz Harness

Before doing anything else, let's start by writing the empty skeleton of a fuzz harness. Use this command to create a new directory in your AOSP checkout under *tools/security/fuzzing/*:

```
$ mkdir -p tools/security/fuzzing/fuzz_book/
```

Inside that directory, we'll need to create two files. The first will contain the C++ code for our fuzz harness. Create a new file called *libpng_fuzzer.cpp* and enter the code shown here:

libpng_fuzzer.cpp

```
#include <stddef.h>
#include <stdint.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  return 0;
}
```

This code is the entry point for a libFuzzer fuzz harness: a function called `LLVMFuzzerTestOneInput()`. The fuzzing engine will call this function in a loop, intelligently mutating the value of the `data` array and correctly setting the `size` parameter on each iteration. AFL++ also supports this style of fuzz harness, which makes it easy to run your fuzz harness in both engines if you want.

We also need to create a build rule for our new fuzz harness. We specify build rules for AOSP in *Android.bp* files. If you've developed Android platform code before, you may be familiar with *cc_library* build rules. For fuzz harnesses, we use a similar rule called *cc_fuzz*, which

produces libFuzzer-based executables by default. Later, we'll show how to adapt them to AFL++ and discuss why you might want to do so. Here's the simplest possible build rule:

Android.bp

```
cc_fuzz {
    name: "libpng_fuzzer",
    srcs: [
        "libpng_fuzzer.cpp",
    ],
    cflags: [
        "-Wno-unused-parameter",
    ],
}
```

The code specifies the name of our new fuzz harness, `libpng_fuzzer`, and points to the single source file we need in the `srcs` section. As we add more features to the fuzz harness, we'll need to update this file with any additional dependencies, source files, and options.

You may note the `-Wno-unused-parameter` flag in the `cflags` section. The Android build system will give warnings and fail builds with unused parameters by default. Since our empty fuzz harness doesn't use the `data` and `size` fields yet, the build will fail if we don't add this option. Once we begin writing the fuzz harness, we'll remove this option from our build rule.

At this point, you can build your fuzz harness. As before, you'll need to be in the root of your AOSP checkout and run these commands:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-eng
$ m libpng_fuzzer
```

Your build should finish after a few minutes with a message saying it completed successfully. You can find the resulting executable in the *out/* directory:

```
$ ls out/target/product/vsoc_x86_64/data/fuzz/x86_64/libpng_fuzzer/
libpng_fuzzer
```

A few shared libraries that the executable depends on will also get built:

```
$ ls out/target/product/vsoc_x86_64/data/fuzz/x86_64/lib/
ld-android.so
libc++.so
libc.so
libclang_rt.ubsan_standalone-x86_64-android.so
libdl.so
libm.so
```

To run your fuzz harness, you must copy all these files onto the virtual device. To make this easy, ADB has a `sync` subcommand. After making sure you have root access to the device, sync the files with these commands:

```
$ adb root
$ adb sync data
```

We copy the entire *out/target/product/vsoc_x86_64/data/* directory over to the device, including the fuzz harness executable and the libraries it depends on. You can see that they've copied over under */data/fuzz/x86_64/*:

```
$ adb shell ls -R /data/fuzz/x86_64
/data/fuzz/x86_64:
lib
libpng_fuzzer
/data/fuzz/x86_64/lib:
ld-android.so
libc++.so
libc.so
libclang_rt.ubsan_standalone-x86_64-android.so
libdl.so
libm.so
```

```
/data/fuzz/x86_64/libpng_fuzzer:
libpng_fuzzer
```

As you continue to develop the fuzz harness during the rest of this chapter, you'll see that additional files get copied over.

Running the Harness

You should now be able to run the fuzz harness with ADB:

```
$ adb shell /data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer
```

When the fuzz harness runs on the device, it will print some information about the settings that libFuzzer is using, as well as periodic updates about the fuzzer's performance:

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 3785275629
INFO: Loaded 1 modules   (1 inline 8-bit counters): 1 [0x5f9d922800b0,
  0x5f9d922800b1),
INFO: Loaded 1 PC tables (1 PCs): 1 [0x5f9d922800b8,0x5f9d922800c8),
INFO: -max_len is not provided; libFuzzer will not generate inputs
  larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 1 ft: 1 corp: 1/1b exec/s: 0 rss: 27Mb
#8388608        pulse  cov: 1 ft: 1 corp: 1/1b lim: 4096 exec/s: 2796202
                rss: 27Mb
#16777216       pulse  cov: 1 ft: 1 corp: 1/1b lim: 4096 exec/s: 2396745
                rss: 27Mb
```

We'll talk later about the meaning of some of these details, but for now you can stop the fuzz harness by pressing CTRL-C.

Exercising the Library Code

Now that you have an empty fuzz harness running, you can start adding real code to test the libpng library.

This library lives in the *external/libpng/* directory in AOSP and includes some sample code in *example.c* with functionality you can enable or disable by defining flags at compile time. You can also read the library's documentation at *https://libpng.org* to understand how to use the library.

For this first simple fuzz harness, we'll focus mostly on the `png_read_png()` function, which does most of the heavy lifting of parsing a PNG image. Its most important parameters are `png_structp` and `png_infop`.

The first, `png_structp`, is used to configure how libpng reads PNG data and handles errors. In the `LLVMFuzzerTestOneInput()` function in your *libpng_fuzzer.cpp* file, add a call to `png_create_read_struct()` to create and initialize the structure:

libpng_fuzzer.cpp

```
png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING,
                                             NULL, NULL, NULL);
if (png_ptr == NULL) {
  return 1;
}
```

You'll notice that the code will return from the `LLVMFuzzerTestOneInput()` function right away if we have an error condition, such as getting a NULL pointer back from `png_create_read_struct()`. This is generally a good idea, since fuzzing should discover problems inside the library when used correctly, not protect against errors outside the library stemming from improper usage. So, we return a nonzero value to show that a failure here is from our fuzz harness and not from the library under test. The NULL parameters passed to `png_create_read_struct()` specify the use of the default error-handling behavior.

The libpng documentation says that when an error occurs, the library will jump back to the routine that called into libpng. To catch and handle these errors, add some code calling `setjmp()`:

```
if (setjmp(png_jmpbuf(png_ptr))) {
  return 0;
```

```
}
```

Since this code gets called if there's an error in the PNG library, you should return a `0` value. This helps the fuzzing engine know that the input actually got into the library under test instead of failing in the fuzz harness.

Now we can create the `png_infop` structure:

```
png_infop info_ptr = png_create_info_struct(png_ptr);
if (info_ptr == NULL) {
  return 1;
}
```

In a typical use case, you would now call `png_init_io()` with a file pointer representing an on-disk image. However, we want to instead use the randomly mutated data provided by the fuzzing engine. You can use the `fmemopen()` function in the stdio.h library to get a file pointer backed by our data array. We can then pass this to `png_init_io()`:

```
FILE *fp = fmemopen((void*) data, size, "rb");
if (fp == NULL) {
  return 1;
}

png_init_io(png_ptr, fp);
```

Finally, we can call `png_read_png()`:

```
png_read_png(png_ptr, info_ptr, PNG_TRANSFORM_IDENTITY, NULL);
```

This function allows some transformations, defined by the contents in *png.h*, to adjust bit depth or transparency.

While this may be interesting functionality to fuzz, you should use the `PNG_TRANSFORM_IDENTITY` constant to read the image as is. As described by the libpng documentation, the last parameter to `png_read_png()` is currently unused and should be NULL. We also need to include the `png.h` and `setjmp.h` headers.

At this point, your code should include everything needed to read and parse a PNG image passed in by the fuzzing engine. Here is the complete, updated code for *libpng_fuzzer.cpp*:

```
#include <png.h>
#include <setjmp.h>
#include <stdint.h>
#include <stdio.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  // Create and initialize the png_struct with the desired error handlers.
  png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING,
                                               NULL, NULL, NULL);
  if (png_ptr == NULL) {
    return 1;
  }

  // In case of errors, the error handlers will jump here.
  if (setjmp(png_jmpbuf(png_ptr))) {
    return 0;
  }

  // Allocate/initialize the memory for image information.
  png_infop info_ptr = png_create_info_struct(png_ptr);
  if (info_ptr == NULL) {
    return 1;
  }

  // Read the data into a "file" so we can use libpng's helpful functions.
  FILE *fp = fmemopen((void*) data, size, "rb");
  if (fp == NULL) {
    return 1;
  }

  png_init_io(png_ptr, fp);
  png_read_png(png_ptr, info_ptr, PNG_TRANSFORM_IDENTITY, NULL);
```

```
  return 0;
}
```

Since the code now depends on libpng, we'll need to update the build rule in our *Android.bp* file. And because we're now using the `data` and `size` parameters, we can also remove the flags to ignore unused variables. Here is the section to be added for dependencies:

Android.bp

```
static_libs: [
    "libpng",
],
```

You can include the libpng dependency in either a `static_libs` or `shared _libs` section. When possible, choose static linking, as it can have slightly better performance. Some libraries in AOSP may allow only dynamic linking, but libpng works either way.

If you try to build the fuzz harness now, you'll get error messages about undefined symbols like `inflate`, `deflate`, and `adler32`, with references to files in *external/libpng*. This happens because *libpng* depends on some symbols in libz. Go ahead and add this dependency to the `static_libs` section in your build rule. Here is the updated *Android.bp* file:

```
cc_fuzz {
    name: "libpng_fuzzer",
    srcs: [
        "libpng_fuzzer.cpp",
    ],
    static_libs: [
        "libpng",
        "libz",
    ],
}
```

You should now be able to build the newer version of your fuzz harness from the root of AOSP, push it onto the virtual device, and run it using these commands:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-eng
$ m libpng_fuzzer
$ adb sync data
$ adb shell /data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer
```

Right away, you should see a lot of messages printing that say either `libpng error: Read Error` or `libpng error: Not a PNG file`. Seeing both messages suggests that the fuzz harness is working to some extent, since it's hitting at least two code paths. However, these error messages make it difficult to tell what's going on, as the status messages are being drowned out. The extra output also slows the fuzz harness. We'll fix these issues in a moment.

Disabling Error Logging

When writing a fuzz harness, it's a good idea to remove as much output as possible. In some situations, this may require removing or disabling logging code. In our case, the logging messages are coming from libpng's default error handlers, implemented by the `png_default_error()` and `png_default_warning()` functions in *external/libpng/pngerror.c*.

Error handlers for libpng are functions that take a `png_structp` and a `png_const_charp` (an array of characters). In the case of fatal errors, you want to do nothing but jump back to the location in the fuzz harness where you call `setjmp()`. For warnings, which shouldn't stop execution, do nothing. Create new functions called `swallow_error()` and `swallow_warning()`:

libpng_fuzzer.cpp

```
void swallow_error(png_structp png_ptr, png_const_charp _) {
  longjmp(png_jmpbuf(png_ptr), 1);
}

void swallow_warning(png_structp png_ptr, png_const_charp _) {
  // Do nothing.
}
```

While our goal here is to remove the logging, as it causes performance issues, we don't want to change or remove the error-handling behavior. When fuzzing other libraries, be careful not to introduce new problems, such as memory leaks or changes to the way the library functions.

You can register these handlers by passing them to `png_create_read_struct()`. Pass them as the last two parameters so that neither warnings nor errors produce output, but warnings allow execution to continue:

```
png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL,
                                             swallow_error, swallow_warning);
```

Now you can rebuild your fuzz harness and run it again:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-eng
$ m libpng_fuzzer
$ adb sync data
$ adb shell /data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer
```

You should now see a similar output as before, with information from libFuzzer about the settings it will use to mutate inputs, as well as the fuzzing progress:

```
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2738785243
INFO: Loaded 1 modules   (3989 inline 8-bit counters): 3989 [0x612b930dc690,
  0x612b930dd625),
INFO: Loaded 1 PC tables (3989 PCs): 3989 [0x612b930dd628,0x612b930ecf78),
INFO: -max_len is not provided; libFuzzer will not generate inputs
  larger than 4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 42 ft: 42 corp: 1/1b exec/s: 0 rss: 27Mb
        NEW_FUNC[1/1]: 0x612b930ac7e0
          (/data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer+0x787e0)
          (BuildId: 06d5313032139f5c2ff46e17defaf3c9)
#465    NEW    cov: 47 ft: 47 corp: 2/9b lim: 8 exec/s: 0 rss: 30Mb
        L: 8/8 MS: 3 ShuffleBytes-CrossOver-InsertRepeatedBytes-
```

```
#32768  pulse  cov: 47 ft: 47 corp: 2/9b lim: 325 exec/s: 16384 rss: 141Mb
#65536  pulse  cov: 47 ft: 47 corp: 2/9b lim: 652 exec/s: 5957 rss: 252Mb
```

Your output may look slightly different because of differing compiler versions, updates to the libpng code, or the randomness of the fuzzing engine.

Understanding the Fuzzer's Output

Now that your fuzz harness is working, let's take a minute to look at some of the information libFuzzer is giving us and understand what it means. Each line starting with a number, such as `#465`, is a status report. Lines with `NEW` show that a new path through the code was found, and lines with `pulse` are status reports that libFuzzer outputs at regular intervals.

The `pulse` report will happen even if a new path through the code hasn't been found. This lets you know that your fuzz harness is still running. The `pulse` lines will appear close together at first, but over time the gaps between them will grow exponentially.

Here are the meanings of the tags in the libFuzzer output:

`cov` The number of features (edges) covered (exercised)

`ft` The number of features (edges) identified but possibly not covered

`corp` The number and total size of the corpus items retained

`lim` The maximum input size tried so far

`exec/s` The number of executions per second

`rss` The current memory consumption

Two additional fields are reported only on `NEW` lines:

`L` The size of the new input

`MS` The mutations used to create the new input

Fixing Memory Leaks

If you look closely at the output from your fuzz harness so far, you may notice something concerning. The memory usage, as reported in the `rss` section, is alarmingly high! In fact, if you leave the fuzz harness running for long enough, it will stop only when it exceeds 2GB of memory consumption.

At the same time, our inputs have been very small so far. The input found in the previous example was only 8 bytes. This suggests that the structures pointed to by the `png_ptr` and `info_ptr` variables are causing us problems.

These structures were allocated by the `png_create_read_struct()` and `png_create_info_struct()` function calls and should be cleaned up by calling `png_destroy_read_struct()`. This function takes both a pointer to `png_structp` and one to `png_infop`. Its last parameter is `png_infopp`, but we don't use this in our example. It's smart enough to handle being passed a `NULL` pointer for things that haven't been used yet.

Go through the code and add an appropriate call to `png_destroy_read_struct()` in each case where `LLVMFuzzerTestOneInput` returns. You should also call `fclose()` to make sure all the file descriptors get cleaned up. This includes the error-handling routine you defined, which will need to be moved later in the code, to immediately before the calls to `png_init_io()` and `png_read_png()`. Here is the revised *libpng_fuzzer.cpp* file with the cleanup added:

```
#include <png.h>
#include <setjmp.h>
#include <stdint.h>
#include <stdio.h>

void swallow_error(png_structp png_ptr, png_const_charp _) {
  longjmp(png_jmpbuf(png_ptr), 1); // Jump back to the setjmp.
}

void swallow_warning(png_structp png_ptr, png_const_charp _) {
  // Do nothing.
}
```

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
  // Create and initialize the png_struct with the desired error handlers.
  // If we don't implement these and just pass NULL, it will be very noisy and
  // spam messages to STDERR on every invalid input.
  png_structp png_ptr = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL,
                                               swallow_error, swallow_warning);
  if (png_ptr == NULL) {
    return 1;
  }

  // Allocate/initialize the memory for image information.
  png_infop info_ptr = png_create_info_struct(png_ptr);
  if (info_ptr == NULL) {
    png_destroy_read_struct(&png_ptr, NULL, NULL);
    return 1;
  }

  // Read the data into a "file" so we can use libpng's helpful functions.
  FILE *fp = fmemopen((void*) data, size, "rb");
  if (fp == NULL) {
    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
    return 1;
  }

  // In case of errors, the error handlers will jump here. We need to clean up.
  if (setjmp(png_jmpbuf(png_ptr))) {
    png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
    fclose(fp);
    return 0;
  }

  png_init_io(png_ptr, fp);
  png_read_png(png_ptr, info_ptr, PNG_TRANSFORM_IDENTITY, NULL);

  // Clean up memory so we don't get an OOM in the fuzz harness.
  png_destroy_read_struct(&png_ptr, &info_ptr, NULL);
  fclose(fp);
```

```
  return 0;
}
```

Now rebuild the code, push it to the virtual device, and run it:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-eng
$ m libpng_fuzzer
$ adb sync data
$ adb shell /data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer
```

When the fuzz harness now runs, the memory consumption should stay low and fairly constant:

```
#1048576        pulse  cov: 79 ft: 79 corp: 2/9b lim: 4096 exec/s: 262144
                rss: 29Mb
#2097152        pulse  cov: 79 ft: 79 corp: 2/9b lim: 4096 exec/s: 233016
                rss: 29Mb
#4194304        pulse  cov: 79 ft: 79 corp: 2/9b lim: 4096 exec/s: 220752
                rss: 29Mb
```

You may also have noticed that the speed, reported as `exec/s`, has increased significantly. Having to keep allocating new memory pages was slowing fuzzing down a lot.

Improving Fuzzer Performance

Let's take a look at a few additional features that can help us fuzz at maximum effectiveness. We won't make any code changes to the fuzz harness at this point, but we can still do several things to improve its performance. These approaches won't be useful for every fuzz harness you write, but you'll see that in some cases they can make a big difference.

Adding a Dictionary

You may have seen that the fuzz harness doesn't seem to be getting very far into the library. It hasn't discovered many edges (reported by `ft`), and the new inputs it finds are small (reported by `L`).

The PNG image format is quite complex. It supports multiple color depths, grayscale images, and transparency, and has extensions for animations. In addition, a valid PNG image must have certain special features, including a specific 8-byte header at the beginning.

As you can imagine, a fuzzing engine that knows nothing about PNG images might need a long time to randomly generate one. One helpful technique is to introduce a *dictionary*, a file that tells the fuzzing engine about values that might be interesting to use when creating inputs, and which ones to avoid removing from inputs it has already identified.

AFL and libFuzzer accept dictionaries as simple text files that can contain blank lines and comments, prefixed with `#`. Each non-blank line that isn't a comment should contain a single value. For example, a dictionary file might look something like this:

```
# This is a comment. It won't do anything.

"foo"
"bar"
"baz"
```

NOTE

AFL uses the term tokens file *instead of* dictionary, *since this file doesn't contain any key-value pairs. However, Android's fuzzing tooling follows libFuzzer's convention of using the name* dictionary, *and we use that in this section as well.*

If this file is fed to a fuzz harness, the fuzzing engine will be more likely to create inputs containing the sequences `foo` and `bar` and less likely to remove or destroy this sequence if it already exists.

Optionally, we can give lines human-readable labels to clarify what they represent. These labels don't have any specific effect on the fuzzing process:

```
MAGIC_NUMBER="42"
```

We can also specify hexadecimal values for inputs when escaped with `x`. For example, create a new dictionary file called *png-header.dict* with a single item representing the constant bytes that define the PNG header:

png-header.dict

```
PNG_HEADER="\x89\x50\x4E\x47\x0D\x0A\x1A\x0A"
```

Then, push this dictionary onto your virtual device:

```
$ adb push png-header.dict /data/fuzz/x86_64/libpng_fuzzer
```

You can now run your fuzz harness with this dictionary by using the `-dict` flag:

```
$ adb shell /data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer \
-dict=/data/fuzz/x86_64/libpng_fuzzer/png-header.dict
```

You'll be shocked at how much faster your fuzz harness finds interesting inputs after making this simple change. Here's what the fuzz harness found after running for less than 10 seconds:

```
#1117455        NEW    cov: 205 ft: 473 corp: 105/22Kb lim: 4096 exec/s: 186242
                rss: 29Mb L: 668/3222 MS: 1 PersAutoDict- DE: "ghhh"-
```

You can also configure the build rule to include the dictionary file by adding a `dictionary` item:

Android.bp

```
cc_fuzz {
    name: "libpng_fuzzer",
    srcs: [
```

```
        "libpng_fuzzer.cpp",
    ],
    static_libs: [
        "libpng",
        "libz",
    ],
    dictionary: "png-header.dict",
}
```

Now, building the fuzz harness and calling `adb sync data` will automatically copy the dictionary file over to the device in a file called *png-header.dict*.

You can find more complete dictionaries, including some for PNG, publicly available online. Google hosts one at *https://github.com/google/fuzzing/blob/master/dictionaries/png.dict*. Try downloading and using it to see how much it further improves your results.

Collecting a Corpus

As discussed in previous chapters, the fuzzing process will collect interesting inputs that exercise novel paths through the code. This collection of inputs is called a *corpus*.

While looking at the output of our previous fuzzing runs, you've seen the size of the corpus reflected in lines like these: `corp: 105/22Kb`. In this example, the fuzzing process has found 105 inputs that sum up to a total of 22KB of data. So far, you haven't stored these inputs anywhere, which means future runs of your fuzz harness have to start from scratch.

You can specify a directory in which to save the corpus items found by your fuzz harness. First you'll need to create an empty directory on the device:

```
$ adb shell mkdir -p /data/fuzz/x86_64/libpng_fuzzer/corpus
```

Now you can tell your fuzz harness to use this directory by passing its path as the last command line argument:

```
$ adb shell /data/fuzz/x86_64/libpng_fuzzer/libpng_fuzzer \
-dict=/data/fuzz/x86_64/libpng_fuzzer/png-header.dict \
/data/fuzz/x86_64/libpng_fuzzer/corpus
```

Let the fuzz harness run for a few minutes before stopping it. If you check the corpus directory, you'll see that it now has a large number of files:

```
$ adb shell ls /data/fuzz/x86_64/libpng_fuzzer/corpus
--snip--
f648c3cbb20e36a93cf16bb6be91dd2eb0792476
f9fbeda3d222118faefba2be6b468182eafcd01b
```

libFuzzer stores corpus items by their MD5 hashes, so don't expect the filenames you see to match the ones we have here.

Interestingly, some of these discovered inputs may be similar enough to real PNG images that the `file` command will recognize them as such:

```
$ adb shell file /data/fuzz/x86_64/libpng_fuzzer/corpus/\*
--snip--
/data/fuzz/x86_64/libpng_fuzzer/corpus/f64...476: data
/data/fuzz/x86_64/libpng_fuzzer/corpus/f9f...01b: PNG image data
```

If you pull these images off of the device via `adb pull`, you may even be able to open some of them in an image viewer, although that's unlikely after only a few minutes of fuzzing.

If you want to restart fuzzing later, you can pass the command the same corpus directory. You'll notice that the information at the beginning of your fuzz harness's output will now be slightly different, showing that it's using inputs from previous runs:

```
INFO:      110 files found in /data/fuzz/x86_64/libpng_fuzzer/corpus
INFO: -max_len is not provided; libFuzzer will not generate inputs
  larger than 4096 bytes
INFO: seed corpus: files: 110 min: 8b max: 3152b total: 21697b rss: 27Mb
```

If you're planning on running your fuzz harness for a long time, it's a good idea to periodically pull the corpus off your device so that you

can save it for later use. Think of this corpus as the end result of your fuzzing; it would be a shame to spend all those CPU cycles generating it only to lose track of it later.

Adding a Seed Corpus

One final tool will help us really kickstart fuzzing: seeding our corpus with some inputs that are known to be interesting. For a well-defined format like PNG, this might just mean getting a few real-life PNG images. Depending on how fancy you'd like to get, you might try finding images that use different bit depths, PNG's animation features, or strange dimensions.

For now, select a few public domain PNG images from Wikipedia's page about PNG at *https://en.wikipedia.org/wiki/PNG*. Once you've downloaded them, copy them into the */corpus* directory on the virtual device:

```
$ adb push *.png /data/fuzz/x86_64/libpng_fuzzer/corpus
```

Now, restart the fuzz harness, and it should pick up the new corpus items. You should see right away that the `cov` and `ft` values in libFuzzer's output should be much larger than they were before. Adding a few small example inputs helped the fuzzer get much deeper into the library.

If you want to make the process of copying your seed corpus onto the device easier, you can add a reference to your corpus items into your `cc_fuzz` build rule, similar to how you added the dictionary earlier. Here is the updated build rule:

```
cc_fuzz {
    name: "libpng_fuzzer",
    srcs: [
        "libpng_fuzzer.cpp",
    ],
    static_libs: [
        "libpng",
        "libz",
    ],
```

```
    dictionary: "png-header.dict",
    corpus: ["corpus/*.png"],
}
```

The `corpus` field in `cc_fuzz` is pretty flexible. It takes a list of paths instead of just one, which would allow you to specify multiple files. Additionally, it has some very basic wildcard support, allowing us to specify **.png* here to match whatever files you've added to your seed corpus. The example assumes you've created a *corpus* directory alongside your fuzz harness's source code and copied your PNG files there.

Now these original corpus files will automatically copy to your device whenever you run `adb sync data`.

Wrapping Up

At this point, you have seen all the steps needed to get a fuzz harness for real Android code up and running. You've seen some common bugs in fuzz harnesses and added a dictionary and a seed corpus to your fuzz harness. We've looked at some measures of fuzzing performance: edge coverage and iteration speed.

We've only scratched the surface of the libpng library. You could try looking through the library's documentation and code to find additional aspects to test, either by extending this fuzz harness or by writing new ones.

In future chapters, you'll see how to use these tools as well as others to fuzz additional parts of Android, like system services.

OceanofPDF.com

7
CRASH ANALYSIS AND BUG TRIAGE



If you've been following along, you have now reached a pivotal stage; your first fuzzer is up and running, and if you're lucky, you've already seen it trigger a crash. In the world of fuzzing, a crash is an exciting event, as it almost always signals something is wrong in the code, whether it be a false alarm in the fuzzing harness or a true software bug in the production code. This chapter will guide you through the process of analyzing those crashes and triaging the underlying issues to determine how they should be handled.

The process of analyzing a crash begins with its artifacts: fuzzer logs, sanitizer output (like that from ASan), and, most importantly, the test case that caused it. From there, you'll make an informed decision about how to proceed with each crash. We'll break down this process step by step.

Initial Crash Validation

A large-scale fuzzing system can generate hundreds or even thousands of crashes, especially when testing software that has never been fuzzed.

Because the team handling these bugs typically has limited resources, it's crucial to perform an initial validation to filter out the noise.

After all, all sorts of unexpected errors can occur. For example, the test environment might run out of memory, or the target device may become unresponsive. These events often cause crashes, which you should de-prioritize. This section covers a few questions to answer before investing more time into your analysis.

Do We Have Enough Information About the Bug?

When a crash happens, you'll typically receive console logs, including the input that triggered it. The logs contain vital information, including crash details, a stack trace, and other debugging data. Here is an example crash:

```
$ ./test-fuzzer
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 2078280040
INFO: Loaded 1 modules   (215 inline 8-bit counters): 215 [0x55f9861c7fd0,
0x55f9861c80a7),
INFO: Loaded 1 PC tables (215 PCs): 215 [0x55f9861c80a8,0x55f9861c8e18),
INFO: -max_len is not provided; libFuzzer will not generate inputs larger than
4096 bytes
INFO: A corpus is not provided, starting from an empty corpus
#2      INITED cov: 59 ft: 60 corp: 1/1b exec/s: 0 rss: 27Mb
--snip--
UndefinedBehaviorSanitizer:DEADLYSIGNAL ❶
==733542==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address
0x000000000000 (...)
==733542==Hint: pc points to the zero page.
==733542==The signal is caused by a READ memory access.
==733542==Hint: address points to the zero page.
    #0 0x000000000000  (<unknown module>) ❷
    #1 0x55f986197fa4 in ubsan_GetStackTrace(...) (<REDACTED>/test-
fuzzer+0x6afa4)
    #2 0x55f9861974b2 in __sanitizer::ReportDeadlySignal(...) (<REDACTED>/test-
fuzzer+0x6a4b2)
    #3 0x55f9861977b4 in __sanitizer::HandleDeadlySignal(...) (<REDACTED>/test-
```

```
fuzzer+0x6a7b4)
    #4 0x7fe078d2cdef  (/lib/x86_64-linux-gnu/libc.so.6+0x3fdef)

==733542==Register values:
rax = 0x0000000000000000  rbx = 0x000055f988c748e0
rcx = 0x000055f9861c9400  rdx = 0x0000000400000000
rdi = 0x0000000000000001  rsi = 0x0000000000000002
rbp = 0x00007ffec5f9f6b0  rsp = 0x00007ffec5f9f658
r8 = 0x00000000000012e0   r9 = 0x0000000000000000
r10 = 0x0000000000000000  r11 = 0x00007fe0792bf080
r12 = 0x000055f988c74e20  r13 = 0x000055f9861c9400
r14 = 0x000055f988c7a940  r15 = 0x0000000000000003
UndefinedBehaviorSanitizer can not provide additional info.
SUMMARY: UndefinedBehaviorSanitizer: SEGV (<unknown module>)
==733542==ABORTING
artifact_prefix='./';
Test unit written to ./crash-2c3cbd53a74aed6c63119dd5f562b86f456de9da ❸
```

This crash log contains the three critical pieces of information we are looking for: the sanitizer output ❶, the stack trace ❷, and information about the test case that generated the issue ❸.

At first glance, we seem to have all the information we need, assuming the test input file, *./crash-2c3cbd53a74aed6c63119dd5f562b86f456de9da*, was also created correctly. But is that really the case? A closer look at the stack trace reveals that we're missing a crucial piece of information: the source line number for each frame. Without it, pinpointing the exact line of code that caused the problem is extremely difficult, which in turn complicates the de-duplication step.

Therefore, instead of immediately investigating the crash, our time is better spent determining why this source line information is missing. In this case, it's because the reproducer was built without debug information. To fix this, enable ASan (`-fsanitize=address`), disable optimizations (`-O0`), and enable debug symbols (`-g`).

NOTE

You should disable optimizations only for the purpose of reproducing the bug (it makes debugging easier), but never during the actual fuzzing.

Once you've made those changes, the output should look like this:

```
    AddressSanitizer:DEADLYSIGNAL
=================================================================
==741075==ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 (...)
==741075==Hint: pc points to the zero page.
==741075==The signal is caused by a READ memory access.
==741075==Hint: address points to the zero page.
    #0 0x000000000000  (<unknown module>)
    #1 0x563c9add9784 in LLVMFuzzerTestOneInput <REDACTED>/test1.cc:39:5 ❶
    #2 0x563c9acda63a in fuzzer::Fuzzer::ExecuteCallback(...) (<REDACTED>/test-
fuzzer+0x5063a)
    #3 0x563c9acd9c49 in fuzzer::Fuzzer::RunOne(...) (<REDACTED>/test-
fuzzer+0x4fc49)
    #4 0x563c9acdb715 in fuzzer::Fuzzer::MutateAndTestOne() (<REDACTED>/test-
fuzzer+0x51715)
    --snip--
```

While the log isn't substantially different, we now have the most important detail: the line number ❶.

The test harness in this example is too simple to have multiple stack frames, but in a real-world scenario, you'd receive multiple frames with source line information, allowing you to pinpoint exactly how the issue occurred. You can also use this information to de-duplicate similar crashes.

Can We Reproduce the Issue?

Another critical artifact from a fuzzer crash is the input that triggers it. This input is usually saved to a file, and the fuzzer should provide a way to rerun the input to reproduce the crash.

For various reasons, however, it may not be possible to reproduce a crash that occurred during a long fuzzing session with the provided test

case. In such situations, the log file is the only useful information you have. For example, consider this fuzzer code:

```
#include <cstddef>
#include <cstdint>
#include <cassert>
#include <fuzzer/FuzzedDataProvider.h>

#define MAX_VALUE_CNT   1000
int values[MAX_VALUE_CNT];

int set_value(uint16_t skip, int new_value) {
    static uint16_t cur_idx = 0;

    assert(skip < MAX_VALUE_CNT);
    cur_idx += skip;
    if (cur_idx > MAX_VALUE_CNT) {
        cur_idx -= MAX_VALUE_CNT;
    }

    int old_value = values[cur_idx];
    values[cur_idx] = new_value;
    return old_value;
}

extern "C"
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    FuzzedDataProvider fdp(Data, Size);
    uint16_t skip = fdp.ConsumeIntegralInRange<uint16_t>(0, MAX_VALUE_CNT-1);
    int value = fdp.ConsumeIntegral<int>();
    set_value(skip, value);
    return 0;
}
```

The `set_value()` function replaces an integer a certain number of values away from `current location`, and it updates the value of `current location` after each call. Our fuzzing harness tries to test this function by

calling it with randomly generated values. This fuzzer will very quickly report a crash, including a test case file:

```
ERROR: AddressSanitizer: global-buffer-overflow on address 0x55a6e93e74a0 at
<REDACTED>
READ of size 4 at 0x55a6e93e74a0 thread T0
    #0 0x55a6e89963fd in set_value(unsigned short, int) <REDACTED>/test4.cc:18:21
    #1 0x55a6e89965bc in LLVMFuzzerTestOneInput <REDACTED>/test4.cc:28:5
    #2 0x55a6e889847a in fuzzer::Fuzzer::ExecuteCallback(...)
    #3 0x55a6e8897a89 in fuzzer::Fuzzer::RunOne(...)
    #4 0x55a6e8899555 in fuzzer::Fuzzer::MutateAndTestOne()
    --snip--

0x55a6e93e74a0 is located 0 bytes after global variable 'values'
defined in '<REDACTED>/test4.cc:7' of size 4000
SUMMARY: AddressSanitizer: global-buffer-overflow <REDACTED>/test4.cc:18:21
in set_value(unsigned short, int)
--snip--
artifact_prefix='./';
Test unit written to ./crash-589a945abbad615804adb51bcad3e6eecde9d6fb
Base64: egAESg==
```

However, rerunning the fuzzer with this single test case won't reproduce the crash:

```
$ ./test-fuzzer ./crash-589a945abbad615804adb51bcad3e6eecde9d6fb
INFO: Running with entropic power schedule (0xFF, 100).
INFO: Seed: 763692306
INFO: Loaded 1 modules   (27 inline 8-bit counters): 27 [0x558c08b73e88,
0x558c08b73ea3),
INFO: Loaded 1 PC tables (27 PCs): 27 [0x558c08b73ea8,0x558c08b74058),
./test-fuzzer: Running 1 inputs 1 time(s) each.
Running: ./crash-589a945abbad615804adb51bcad3e6eecde9d6fb
Executed ./crash-589a945abbad615804adb51bcad3e6eecde9d6fb in 0 ms

 NOTE: fuzzing was not performed, you have only
       executed the target code on a fixed set of inputs.
```

This scenario is common when using libFuzzer and other "persistent mode" fuzzing frameworks, where the target application maintains a global state that previous test cases can influence. Crashes can occur because of the cumulative effect of multiple inputs. While these crashes might still indicate serious bugs, the inability to reliably reproduce them prevents further analysis, so you should de-prioritize them.

Another major category of reproducibility issues involves nondeterministic factors such as race conditions, the use of uninitialized variables, or dependencies on factors outside the fuzzing environment, like system time, the number of open files, or memory usage.

One approach to triaging these problems is to run the same test input multiple times to calculate a reproduction rate, then use a threshold for deprioritization. In any case, *de-prioritization* doesn't mean *ignore completely*. It's always good practice to review the logs of recent unreproducible crashes to understand why they're failing before discarding them entirely.

Is It a Legitimate Bug?

At this point, you have enough information to debug the crash and a test case that reliably triggers it. But is it really a bug? Not always. False positives are common in fuzzing, often because the fuzzer executes the target code in an environment that differs from real-world scenarios. A good example is code that includes `assert` or `CHECK` statements inserted by programmers to handle unexpected input that a caller should have sanitized. Here is an example that uses an `assert` statement:

```
#define MAX_VALUE_CNT   1000
int values[MAX_VALUE_CNT];

int set_value(uint16_t skip, int new_value) {
    static uint16_t cur_idx = 0;

    assert(skip < MAX_VALUE_CNT);
    cur_idx += skip;
    if (cur_idx > MAX_VALUE_CNT) {
        cur_idx -= MAX_VALUE_CNT;
```

```
    }

    int old_value = values[cur_idx];
    values[cur_idx] = new_value;
    return old_value;
}
```

The developer of this code proactively added an `assert` to detect an unexpected situation their code might run into. This will lead to an immediate crash, but it's intentional. The developer is basically telling whoever is reading this code that they aren't expecting a certain scenario to happen.

Crashes may also occur when a programmer has mocked a piece of hardware or software that the target depends on and the mock is incomplete or inconsistent with the specification. Such inconsistencies can lead to crashes that won't exist in a production environment.

In many cases, the line between a false positive and a true positive is blurry. For example, a strangely behaved piece of hardware might trigger a kernel crash, but on systems where this hardware doesn't exist, the crash seems like a false positive. However, with the rise of virtualization, an attacker could present a guest VM with the desired behavior, turning a theoretical issue into a real vulnerability under the pKVM environment, where the VM implementation is expected to maintain the security assumption of the guest even if the host is compromised.

Unlike the previous validation steps, which you can easily automate, identifying false positives often requires manual effort. Determining what counts as a false positive frequently depends on the specific scenarios a fuzzer is designed to cover, so it's always preferable to have the fuzzer owner make this determination. In large organizations where this isn't feasible, having each fuzzer owner create a document describing potential edge cases and example scenarios can greatly assist those performing the triage.

Note that you should eventually fix all crashes (even false positives); otherwise, the fuzzer will keep hitting the crash and won't move

forward.

Crash De-Duplication

You've validated that you have a legitimate crash with useful debug information and a reproducible test case. But should you fire up the debugger just yet? Not so fast. As mentioned earlier, coverage-guided fuzzing tends to generate many inputs that explore similar code paths. Once a fuzzer finds a crash, it will often generate subsequent inputs that trigger the exact same bug.

To avoid analyzing the same bug repeatedly, you need to de-duplicate these crashes. There are several ways to do this. Let's use the following sample crash log to demonstrate various approaches:

```
==7011==ERROR: HWAddressSanitizer: SEGV on unknown address 0x000000000000 (...)
==7011==The signal is caused by a READ memory access.
==7011==Hint: address points to the zero page.
#0 0x773fe64c64 in __memcpy_aarch64_simd <REDACTED>
#1 0x64b92a8e40 in gatt_sec_check_complete(bool, tGATT_CLCB*, unsigned char)
#2 0x64b92a8e40 in gatt_security_check_start(tGATT_CLCB*)
#3 0x64b9297d80 in GATTC_Read(unsigned short, unsigned char, tGATT_READ_PARAM*)
#4 0x64b9395b80 in FuzzAsClient(unsigned char const*, unsigned long)
#5 0x64b9395b80 in LLVMFuzzerTestOneInput
--snip--
```

The simplest strategy is to look at the crash site; if multiple crashes occur at the same PC address, they likely represent the same bug.

In this case, however, the top of the stack trace identifies the function that caused the crash as `__memcpy_aarch64_simd()`, immediately revealing the problem with this approach: `__memcpy_aarch64_simd()` is a C runtime library function used by many other functions. The root cause of the crash is almost always in the upstream code.

A quick improvement to our de-duplication process is to follow the stack trace and identify the last non-runtime function. In our case, this is `gatt_sec _check_complete()`. To make our assessment more accurate, we

can then include more of the stack trace, which in this example would give us the following:

```
gatt_sec_check_complete(bool, tGATT_CLCB*, unsigned char)
gatt_security_check_start(tGATT_CLCB*)
GATTC_Read(unsigned short, unsigned char, tGATT_READ_PARAM*)
FuzzAsClient(unsigned char const*, unsigned long)
LLVMFuzzerTestOneInput
```

This list provides better granularity, especially if there are multiple ways to reach the terminal function. We've stopped at the fuzzing entry point `LLVMFuzzerTestOneInput` to avoid noise. This is usually sufficient when running fuzzers locally. A simple script can quickly group crashes into unique cases.

Large-scale fuzzing infrastructures typically implement more sophisticated de-duplication algorithms that also consider the crash symptom, potential source code changes across builds, and heuristics to avoid over-grouping issues reachable from various call paths. For example, these algorithms would take into consideration the following information in addition to the stack trace:

```
==7011==ERROR: HWAddressSanitizer: SEGV on unknown address 0x000000000000 (...)
==7011==The signal is caused by a READ memory access.
==7011==Hint: address points to the zero page.
```

These lines indicate that the crash is caused by a null pointer dereference. If another crash has exactly the same stack trace but a different crash type, it might be the result of a completely different issue.

With all that said, no de-duplication method is perfect. Mistakes can lead to two possible consequences:

Insufficient de-duplication The vulnerability management or engineering team gets spammed with duplicate reports of the same issue, wasting the team's time.

Overly aggressive de-duplication One issue masks another. In the worst-case scenario, a low-severity bug could hide a high-severity issue, leaving the system vulnerable for an extended period.

Both consequences are undesirable, and the impact of each depends on the scenarios. It's important to have adjustable de-duplicate parameters to balance the trade-offs.

Making a Severity Assessment

Now that you have a collection of legitimate, reproducible, and unique bugs, what's next? Well, a bug bounty researcher might start developing a proof-of-concept (PoC) or an exploit. A software engineer will begin working on a fix. A vulnerability management team will notify affected customers and deliver patches. All these actions are time-consuming, and we likely lack the resources to address every bug at once. This is where it's important to determine the bug's severity.

In the software security domain, *severity* is a measure of the potential impact of a security issue. The specific rules for making such a determination can vary by domain but usually derive from the three factors covered in this section.

Behavior

Behavior, arguably the most important factor, answers the question, "What can an attacker do with this bug?" Let's cover some common behaviors.

Code Execution

Code execution means an attacker can run arbitrary code in the target environment after successfully exploiting the bug. This is typically the worst-case scenario, as it gives the attacker ultimate control to steal sensitive information or pivot to other systems. For example, the famous Log4Shell vulnerability in the Log4j Java logging library (CVE-2021-44228) allows attackers to execute arbitrary code on unpatched systems by sending a specially crafted string.

For bugs detectable by fuzzing, especially memory safety issues, a vulnerability that leads to attacker-controlled memory writes is often associated with code execution. This includes stack overflows, out-of-bound (OOB) writes, and use-after-free (UAF) and double-free vulnerabilities.

Modern CPU architectures have many mitigations that prevent direct modification of the code section. But arbitrary memory writes can still lead to code execution through indirect approaches like these:

Stack OOB writes These can overwrite the program counter (PC) or other registers to alter the execution flow.

Function pointers, longjmp buffers, or C++ vtables An attacker can modify these to alter the execution flow.

Important flags An attacker can overwrite these to alter the execution flow.

Pointers An attacker can overwrite these to pivot from one OOB write to another.

Information Disclosure

An information disclosure bug allows an attacker to gain access to information they aren't supposed to have. This information can be as obvious as cryptographic keys, passwords, or random chunks of server memory. A well-known example is the Heartbleed bug in the OpenSSL library (CVE-2014-0160), which allows attackers to read sensitive data from the memory of affected servers, including secret keys and user credentials.

In other cases, the impact of the disclosure can be less obvious. For example, consider a bug that gives an attacker a Boolean value indicating whether the value at address *X* matches certain conditions. This may not sound as harmful, but if that address stores cryptographic key material, attackers can use it to probe the statistical distribution of the secret and eventually gain full knowledge of the key bytes.

In the context of memory safety, information disclosure often results from OOB reads and the use of uninitialized variables. Distinguishing

between an OOB read and an OOB write is usually easy by reading the ASan logs. In certain cases, however, a read can mask a write if you don't check the source code. For example, consider the following excerpt of the assertion code considered previously:

```
int set_value(uint16_t skip, int new_value) {
    static uint16_t cur_idx = 0;

    assert(skip < MAX_VALUE_CNT);
    cur_idx += skip;
    if (cur_idx > MAX_VALUE_CNT) {
        cur_idx -= MAX_VALUE_CNT;
    }

    int old_value = values[cur_idx]; // <=== OOB READ
    values[cur_idx] = new_value;  // <== OOB WRITE
    return old_value;
}
```

When `cur_idx` goes beyond the array limit, the code reads the old value first, then writes the new value. In this case, if ASan is enabled, the fuzzer will crash with an `OOB READ` error, leaving the subsequent `OOB WRITE` undetected. This issue can be even harder to spot in code like `values[cur_idx]++`, where the read and write operations occur in a single C statement. Therefore, it's important to review the actual code before determining the nature of the issue.

Denial of Service

DoS bugs cause the target software to crash or become unusable. A DoS can be temporary, with the system recovering after a reboot or restart, or permanent, with a recovery that isn't straightforward (for example, a corrupted configuration file prevents a bootloader from running).

In general, DoS is considered less severe than code execution or information disclosure, but its impact can vary dramatically depending on the domain. A DoS attack on a hospital's systems or a nuclear power plant could be catastrophic.

Null pointer dereference (NPD) is a common cause of DoS, although other memory safety issues can lead to it as well.

Privilege Level

The privilege level of the buggy component is another major factor in determining the bug's severity, as it affects the level of access an attacker can gain by exploiting the bug.

Different systems segment privileges differently. On Android, we typically acknowledge the following levels, from most to least privileged:

Secure element (SE) This hardware component is designed to withstand extensive physical and software attacks. A vulnerability here is disastrous and usually leads to a total compromise and significant financial loss.

Trusted execution environment (TEE)/hypervisor/ bootloader These components operate at a higher privilege level than the main operating system kernel. They're designed to remain secure even if the entire operating system is compromised. Examples include ARM's TrustZone and Intel's AMT.

The kernel, device drivers, and firmware This is the core of the operating system, responsible for isolating user applications and managing hardware.

The system (HAL, framework, privileged apps, and so on) These are userspace processes that are granted special permissions, allowing them to access resources that regular applications can't. Examples on Android include processes that manage Bluetooth hardware (`bluetoothd`) or cellular communications (`rild`), often enforced by SELinux or Linux capabilities.

Untrusted apps These are standard applications running on a user's device. The operating system heavily restricts their access to system resources. On Android, they usually don't have direct access

to most device drivers and require Android permissions to access services provided by the system.

Constrained process This is a further locked-down environment designed to handle high-risk operations. A good example is Android's `mediaextractor` process, which runs in a tight sandbox to parse potentially malicious media streams. This sandbox severely limits the process's access to the rest of the system, confining any potential damage.

While this may depend on the particular threat model, in general, bugs in higher-privileged components lead to higher impact and should be prioritized for remediation.

Attack Vector

The attack vector describes the kind of access to the target needed to trigger the bug.

A *remote* attack can be launched from anywhere in the world. For example, an attacker can target a web server by sending a malicious request from across the globe. Remote attacks are often stealthy and difficult to trace, giving them the most severe score in this category.

A *proximal* attack requires the attacker to be within a certain physical range of the target but not necessarily touching it. This is typical for attacks over Bluetooth, near-field communication (NFC), or other radio-frequency protocols. In certain cases, attackers can convert these issues to remote ones via repeater or gateway devices.

A *local* attack requires the attacker to have already established a foothold on the target device. For example, an attacker might convince a user to install a malicious app from an app store, often for privilege escalation or information disclosure.

Finally, a *physical* attack requires the attacker to have physical access to the target environment, whether by plugging a device into a company's network jack or disassembling a mobile phone and soldering components onto its circuit board (sometimes termed *extensive physical access*).

The Final Determination

The final severity of a bug depends on a combination of all three factors we've discussed, as well as many other possible minor factors. To standardize scoring, we recommend following a set of publicly available guidelines. For example, MITRE has the Common Weakness Scoring System, and the National Institute of Standards and Technology uses the Common Vulnerability Scoring System. Android also publishes its own severity guidelines at *https://source.android.com/docs/security/overview/updates-resources#severity*.

As an exercise, let's walk through an example using Android's severity guidelines. Imagine you're a security researcher whose Android Bluetooth fuzzer reports the following crash:

```
ERROR: HWAddressSanitizer: tag-mismatch on address 0x0045247d0289 at pc
0x0064248f4d18
READ of size 1 at 0x0045247d0289 tags: 66/09(66) (ptr/mem) in thread T0 ❶
#0 0x64248f4d18 in smp_proc_sec_req(...) <...>/smp/smp_act.cc:464:31
#1 0x642490b9e0 in smp_sm_event(...) <...>/smp/smp_main.cc:1048:7
#2 0x6424909ff0 in smp_data_received(...) <...>/smp/smp_l2c.cc:217:5
#3 0x64248e5afc in Fuzz(...) <...>/fuzzers/smp_fuzzer.cc:213:5
#4 0x64248e6538 in LLVMFuzzerTestOneInput <...>/fuzzers/smp_fuzzer.cc:226:3
```

The crash report also contains the fuzzer binary and a test input, which triggers exactly the same stack trace. This means the issue is reproducible. The crash stack also provides enough information, including the crash type, memory address, and source line numbers leading to the crash. You've also checked that no other bugs have been reported from a similar crash. What severity should we expect?

Following the methodology from this chapter, you can quickly identify this issue as an OOB read ❶, which makes its behavior information disclosure. The target is Bluetooth, which is remotely accessible but requires the attacker to be nearby, so the attack vector is proximal. As mentioned, Bluetooth runs as a privileged system service. Combining these factors gives you a *proximal information disclosure from a privileged process*.

Looking at the Android severity guidelines, the best severity match seems to be this:

> **High**—Remote access to protected data (that is, data that is limited to a privileged context)

If you submitted this bug to the Android Vulnerability Rewards Program, however, you might receive a different severity rating:

> **Moderate**—A general bypass for a defense in depth or exploit mitigation technology in a privileged context, THB, or the OS kernel Why the difference? The reason lies in the level of access to the information. Take a look at the source code that contains the issue:

```
void smp_proc_sec_req(tSMP_CB* p_cb, tSMP_INT_DATA* p_data) {
  if (smp_command_has_invalid_length(p_cb)) {
    tSMP_INT_DATA smp_int_data;
    smp_int_data.status = SMP_INVALID_PARAMETERS;
    smp_sm_event(p_cb, SMP_AUTH_CMPL_EVT, &smp_int_data);
    return;
  }
  tBTM_LE_AUTH_REQ auth_req = *(tBTM_LE_AUTH_REQ*)p_data->p_data; ❶
  tBTM_BLE_SEC_REQ_ACT sec_req_act;
  log::verbose("auth_req=0x{:x}", auth_req);
  p_cb->cb_evt = SMP_EVT_NONE;
  btm_ble_link_sec_check(p_cb->pairing_bda, auth_req, &sec_req_act);
  log::verbose("sec_req_act={}", sec_req_act);
  switch (sec_req_act) {
    case BTM_BLE_SEC_REQ_ACT_ENCRYPT:
      log::verbose("BTM_BLE_SEC_REQ_ACT_ENCRYPT");
      smp_sm_event(p_cb, SMP_ENC_REQ_EVT, NULL);
      break;
    case BTM_BLE_SEC_REQ_ACT_PAIR:
      p_cb->sc_only_mode_locally_required =
              (p_cb->init_security_mode == BTM_SEC_MODE_SC) ? true : false;
      /* respond to non SC pairing request as failure in SC only mode */
```

```
      if (p_cb->sc_only_mode_locally_required && (auth_req & SMP_SC_SUPPORT_BIT)
== 0) {
        tSMP_INT_DATA smp_int_data;
        smp_int_data.status = SMP_PAIR_AUTH_FAIL;
        smp_sm_event(p_cb, SMP_AUTH_CMPL_EVT, &smp_int_data);
      } else {
        /* initialize local i/r key to be default keys */
        p_cb->peer_auth_req = auth_req;
        p_cb->local_r_key = p_cb->local_i_key = SMP_SEC_DEFAULT_KEY;
        p_cb->cb_evt = SMP_SEC_REQUEST_EVT;
      }
      break;
    case BTM_BLE_SEC_REQ_ACT_DISCARD:
      p_cb->discard_sec_req = true;
      break;
    default:
      /* do nothing */
      break;
  }
}
```

NOTE

You can find this code at https://android.googlesource.com/platform/packages/modules/Bluetooth/+/refs/heads/android15-release/system/stack/smp/smp_act.cc#465.

In this case, there is an OOB read in the source code ❶. The information is read from beyond the incoming packet into a local variable called `auth_req`. However, the subsequent code reveals that this variable isn't directly sent back to the attacker. Instead, it's used to decide which `case` statement to execute.

We can roughly conclude that an attacker wouldn't be able to retrieve the exact data read; instead, they could probably infer the range of that particular byte based on the target device's response. An attacker

might be able to use this to bypass address space layout randomization (ASLR) but couldn't likely exfiltrate sensitive data directly, justifying the Moderate rating.

NOTE

The Android Vulnerability Reward Program severity guidelines update every year. You can find the official guideline document at https://source.android.com/docs/security/overview/updates-resources#severity.

As you can see from this example, the assessment process requires many assumptions. The severity of a bug can be difficult to determine without closely examining the code and gaining a deep understanding of the way a component works. It's common for an engineering team to adjust its severity ranking based on new findings while working on a fix. The goal of this process isn't to assign a perfectly accurate rating for every bug, but rather to get a relatively accurate estimation so that we can prioritize the remediation actions without investing excessive effort.

Wrapping Up

You'll need to analyze crashes and triage bugs before you can act on a fuzzer's output. This chapter walked through the critical steps of this process: ensuring that the crash log contains sufficient debug information, validating the crash's reproducibility (especially in stateful fuzzing environments), and performing initial checks to distinguish true bugs from false positives.

Most importantly, we detailed the structured process of severity assessment, which takes into account the bug's behavior, the privilege level of the affected component, and the attack vector required to exploit it. By standardizing this process, engineering and vulnerability teams can efficiently prioritize fixes to mitigate the most critical risks first.

OceanofPDF.com

PART III
ADVANCED TARGETS

OceanofPDF.com

8
SYSTEM SERVICES



The most fundamental components of Android's userspace are the *system services*, which form the foundation of the operating system, managing everything from hardware drivers and files to the Android Java runtime. Basically, everything in Android that isn't an app is a system service.

The goal of fuzzing system services is to test the smallest granules of Android's security model: the process. From a security perspective, the process is the most basic unit of isolation, as any part of a process can access all of that process's memory. This, in turn, means that if an attacker can take over part of a process, they can take over all of it.

In this chapter, we consider code that runs in a system process and all the surfaces these processes communicate with. Today, service fuzzing most commonly focuses on communication over Binder, Android's main IPC mechanism, so we'll pay particular attention to it here. That said, system service fuzzing may also cover sockets, files, properties, or any other data that a process reads or writes.

Because of the scale of Android services, fuzzing their code remains an ongoing and evolving effort. The more bugs you fix in such a system,

the more bugs it tells you about.

A Blueprint for Fuzzing Large Systems

Until recently, most Android fuzzers were component based. No infrastructure could yet run fuzzers at scale, and no automatic fuzzing infrastructure existed for Android services. Engineers had made significant investment into securing the kernel, but they knew that problems were lurking in userspace code surrounding IPC.

Specifically, most Android communication goes over the Binder kernel driver and is processed by libbinder. Steven Moreland, one of this book's authors, first started looking at Binder fuzzing in 2019. At that time, the few black-box fuzzers set up for Android services crashed far too often to be useful. Sending a random few bytes to any service in Android would lead to a crash. Often, the cascade of crashes resulted in system instability that made these crashes hard to debug. The Android team knew what it wanted—the ability to fuzz Android system services at the security boundary—but the infrastructure was far too unstable to support it.

Our attempts to fix this process began a multiyear journey across the Android codebase, involving dozens and sometimes even hundreds of engineers. After going through this process and watching people attack other large software projects, we believe we've found a template useful for creating a strategy to fuzz any complex system.

Challenges of Black-Box Fuzzing

So, why didn't black-box fuzzing work? When fuzzing a high-level system, we can get errors from every layer of the stack. Figure 8-1 is a pyramid showing how various layers in the Android stack work together to provide IPC.

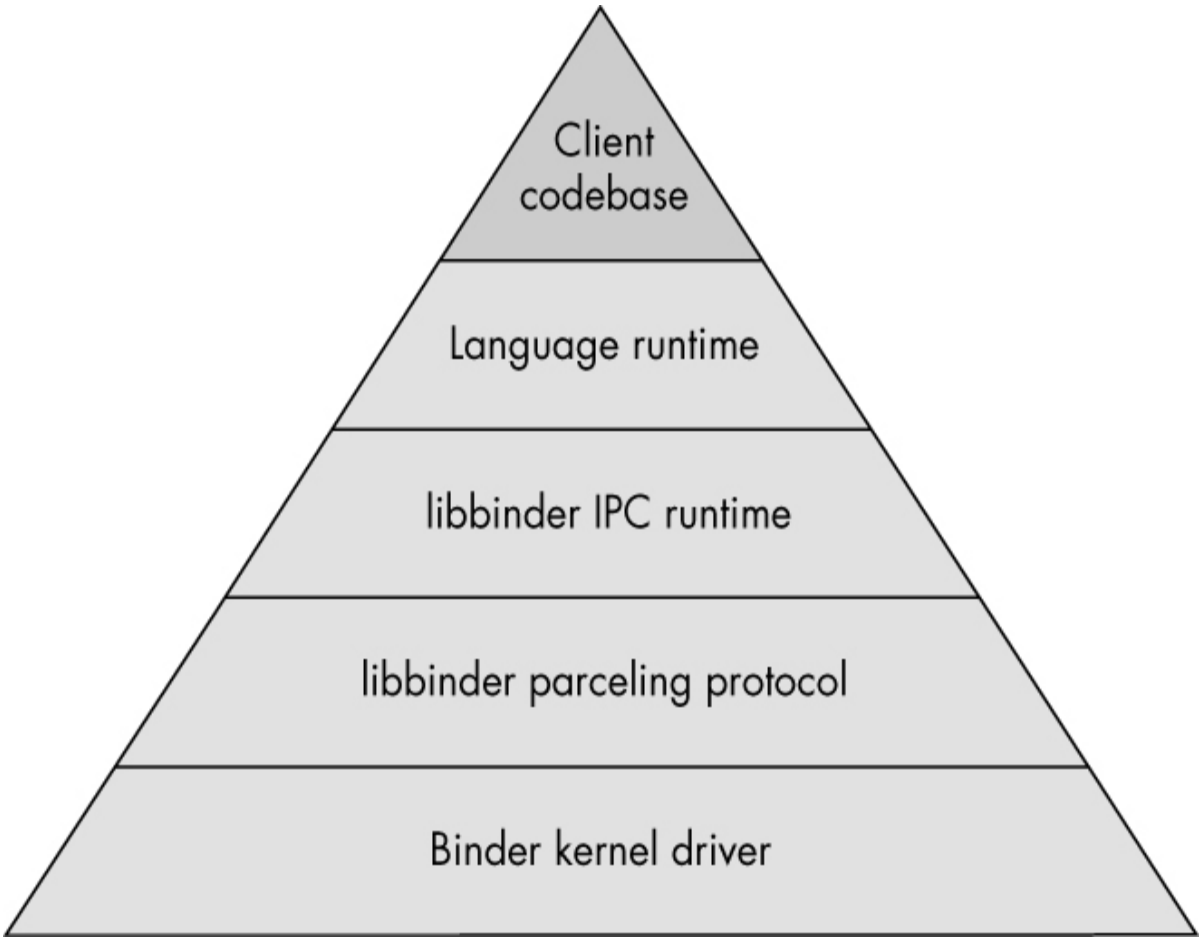


Figure 8-1: Layers in Android's inter-process communication

This pyramid shows where components exist in the dependency hierarchy. The higher a layer is in this pyramid, the more codebases it represents. For instance, tens of thousands of system services exist across all forks and versions of Android, but only one version of the kernel driver is in production today. Here is a description of each layer:

The client codebase The domain-specific code that represents the implementation of a specific system service in Android. This layer contains permission checks, session information, the global state, dependencies on other services, the logic to read or write to files, and any other business logic or functionality relevant to a specific domain. Examples include process-isolated hardware drivers, Android's `ActivityManager`, and other core services.

The language runtime The language-specific code that glues language-specific components to their underlying native implementation. For instance, in Java, `android.os.BinderProxy` is a wrapper around `BpBinder` in libbinder.

The libbinder IPC runtime The high-level classes that implement communication nodes (or *binders*) as well as any thread- or process-specific state used to communicate with the binder driver (or sockets, in the case of RPC Binder, a backend of libbinder that supports cross-kernel or cross-device communication).

The libbinder parceling protocol The `Parcel` class and other low-level parceling code in libbinder, which deals with how to take data you have in a process and serialize it so it can be sent to another process as part of a binder transaction.

The binder kernel driver The Linux kernel driver that manages copying transaction data between processes and waking them up.

Our original fuzzing attempts ran into crashes at each level in the system, when, in reality, we care only about the security properties of the high-level systems. These are the systems that applications can directly communicate with. However, for these to be secure, every dependency and every code path used by these high-level systems also need to be secure, so we end up focusing on every system.

API Contract Enforcement

If you are fuzzing a large-scale system and find that you're getting many crashes in lower-level systems, you may want to consider fuzzing them first or separately. That way, you can focus on fixing issues in one area of the codebase at a time.

In some cases, however, crashes in lower-level systems may be due to bad arguments or bad setups from the high-level system. So, you have to make sure that you explicitly define your API contracts and are intentional about which ones to enforce.

For instance, one API contract for Binder is that you don't write to a read-only `Parcel`. Because of the millions of codebases built on top of this

stack, though, this isn't an assumption we can safely make. We have to consider every combination of arguments and make sure that they lead to a safe, definable behavior.

On the other hand, one API contract that we do enforce is that if you give a pointer to a `Parcel`, that pointer must be valid. There is no way to determine whether a pointer is valid at runtime, so we have to make this assumption in libbinder. However, if you pass an invalid pointer to libbinder, it may appear to crash inside libbinder directly, even though the error (creating an invalid pointer) happens in client code.

Determining where exactly an error comes from requires a deep understanding of the code, which starts with an understanding of the API contract between codebases. In the best-case scenario, an API contract matches the runtime semantics. In the next best scenario, an API contract matches the type semantics. Finally, in the worst scenario, sometimes called *hope-based programming*, the API contract matches comment or assumed semantics. Let's give an example of each case and review how these types of API contracts inform where fuzzer bugs occur and therefore how to target your fuzzer:

When the API contract matches the runtime semantics

In C++, for example, if you're passed a pointer to an object, your client may also pass this pointer to another thread and modify it concurrently. If the object code guards each method with a lock or asserts that locks are held, it would protect against this aspect of runtime behavior.

Ultimately, anything that can happen as part of the runtime could be a security concern, given the right attack vector. So, we have to decide whether to include such risks in our security model. `Parcel`, like many other classes, isn't thread-safe, and we always expect the client to respect that.

When the API contract matches the type semantics

If your method is passed a `Parcel` object, we assume that it is a valid constructed `Parcel` object, and this object has guaranteed properties based on the way it was created. If you perform a `reinterpret-cast` operation on a random chunk of memory to turn it into a `Parcel`, the

object won't be valid, and there is no way to reject it. The bug would be in the client code.

When the API contract matches the assumed semantics

Imagine your code has a method that takes an `int` and assumes the value is from 0 to 100. You may have left a comment defining this assumption as part of the API contract, but the type and runtime both allow for larger values to be passed instead. It may be the case that no high-level code ever calls this method with, say, 101, but if passing 101 leads to a security vulnerability, fuzzing and fixing this code may still be worthwhile so you don't need to hope that nobody will ever call the code in such a way. Of course, doing so requires extra work.

Cases with assumed semantics also demonstrate why a mix of high-level and low-level fuzzing is so important. Low-level fuzzing catches issues with a simpler setup, but it may not encode all possible aspects of the runtime. In such cases, high-level fuzzing can still catch errors in low-level code that low-level fuzzers miss.

In many cases, fuzzers for low-level systems may do more work than necessary to securely operate the high-level system. For instance, consider the case of weak references to binder objects on the wire. In BeOS (the predecessor to Android where Binder was first developed), it was common to pass a weak pointer to a binder object to another process. This allowed you to use certain objects for identity checks or as capabilities without actually owning the memory. Only about four parts of the Android codebase used this feature, however; we (apparently) prefer simplicity over such detailed and error-prone reference counting. So, it was easier to delete this feature from Android than to support it.

As you assess API contracts, leave no stone unturned. Even if you think a certain operation can't happen, assume you're wrong. Instead of claiming that a codebase contains no issues, say that you don't know of any issues. Then go out and test them.

Fuzzing Binder Services

The general strategy for fuzzing binder system services has been to go up the hierarchy of dependencies, beginning with fuzzing the binder `Parcel` class via a tool called `binder_parcel_fuzzer`.

Then, we move on to creating a service fuzzer that can fuzz an arbitrary service. Running this fuzzer on a sample service that is empty (or near empty) allows us to fuzz the libbinder IPC runtime. A stable C wrapper interface, libbinder_ndk, allows updatable components, such as apps, to use libbinder. And running the service fuzzer against code built for various runtimes, such as libbinder_ndk or against the binder code in the Android SDK, allows us to gain confidence in these runtimes.

Finally, we fuzz real services, scaling up to testing the entirety of AOSP.

Fuzzing Parcel Objects

To fuzz a `Parcel` object, `binder_parcel_fuzzer` performs every possible sequence of read or write operations on the given `Parcel` with every possible randomly constructed argument for such methods.

This process may sound simple, but the devil is in the details, and it's extremely important to run a fuzzer like this continuously. For example, even though we thought it had long ago found every OOB write, it occasionally catches bad code changes in libbinder we didn't previously know about. After years of iteration, it has found over a dozen security vulnerabilities on its own.

Let's briefly touch on a specific detail of `Parcel` that permeates the way we fuzz these services today: Many APIs inside `Parcel` request reading a vector of an arbitrary size. This is one of the main reasons that the black-box testing of binder services used to fail. Sending random data to the services would result in out-of-memory crashes (or subsequent null pointer crashes) when libbinder tried to allocate several gigabytes of data, rendering the fuzzer useless. There was no way to fuzz a high-level service at the binder level because 99 percent of the time it would crash with an out-of-memory error. Several bugs of this caliber had to be solved before we could stably fuzz Android services.

Originally, we tried using `rlimit` to constrain the maximum allowed size of memory allocations. This seemed like the perfect tool: We could set `rlimit` to a few megabytes, and whenever a large allocation came through, the software would crash. Most of these cases involved attempts to read a vector off of the `Parcel`. The code would read an integer, then allocate an array of this size, then read that many elements off the `Parcel` data buffer, filling out the array.

These issues were easy to fix; before allocating the result array, we could check how much data was left on the `Parcel`. If we wanted to allocate a gigabyte but the `Parcel` was only a few bytes long, we could preemptively avoid making the excessively large allocation because we knew it would never be used.

However, an insidious problem was hiding underneath these cases. The `Parcel` API had a lurking issue; for historical reasons, some APIs include the size of an output vector on the `Parcel`, and the reader will allocate an array of this size to fill results into, but there is no associated data in the `Parcel` itself. So, in these cases, our trick to preemptively avoid overly large allocations wouldn't work. If you requested that a process allocate a vector of any size, you'd cause a crash.

Now, this horrendous API came from the AIDL's history as a Java-first language. There is no way to resize a Java array, so any Java code in Android that needed to return an array would pass the size of the array as an input, and an Android service would allocate that size and then fill it out. There is the gut punch: How big could the array be?

At the same time, `binder_parcel_fuzzer` started failing in continuous integration. It would make a small allocation of just a few bytes and hit the `rlimit`, crashing and reporting a bug. In these cases, the allocations weren't causing a security issue, but they were preventing us from finding legitimate ones, and it wasn't immediately obvious where they were coming from.

To understand what was happening here, imagine that `rlimit` was set to 10MB. In continuous integration, we would first allocate 9.99MB; then, the next tiny allocation would take us over that limit, causing a crash. No matter what we set `rlimit` to, allocations just below that size would cause a big problem.

So, we took out `rlimit` entirely and instead put a 1MB limit on every allocation. This is the right size for Binder because each process already has a 1MB transaction limit, so these allocations shouldn't occur anyway. The key point is this: We don't want to disallow that last small allocation; we want to disallow only big allocations. Therefore, `rlimit` was the wrong tool for this job.

Today, all parceling methods in the binder stack have to adhere to this rule. When we run the fuzzer, a certain number of these operations may run, but since they allocate and then delete memory (as the fuzzer throws away results), the fuzzer memory never exceeds several megabytes and won't reach the gigabytes of memory present on certain test machines today.

One takeaway is that seeing something as a symptom of a crash doesn't necessarily mean that you should directly put a limit on that thing in order to test it. Sometimes you need to come up with a new rule that, when applied to your codebase, can avoid the crash based on the codebase's behavior.

Another takeaway is that fuzzing is hard. Just because it doesn't work out the first time doesn't mean that fuzzing won't work. Fuzzing can be very different from other types of programming, and you may need to try multiple approaches to find one that works.

Automating a Service Fuzzer

With the parceling code resolved, it's time to automate fuzzing services. We implement this logic in `binder_parcel_fuzzer`, under the method `fuzzService()`, which adapts a fuzzer to an arbitrary binder object. The original implementation of this method comprised fewer than 100 lines of code yet was capable of fuzzing billions of lines, proving that resolving issues at the correct layer removes the need for large code workarounds at other layers.

Now, how do we build a routine that can fuzz every service? To do this, we need a few primitives:

`RandomBinder` This class takes in an arbitrary byte array and uses it to randomly respond to transactions by creating deterministically

random `Parcel`s from them.

`getRandomBinder()` This function creates a deterministically random binder object based on the input fuzzing data, which may be a real binder object on the system, an empty binder object, or an instance of `RandomBinder`.

`getRandomFds()` This function returns deterministically random file descriptors based on the input fuzzing data, such as an open file, a new pipe, and so on.

`fillRandomParcel()` This function fills a `Parcel` with deterministically random data, binders, or file descriptors.

You can find these primitives in the library, libbinder_random_parcel. Here is pseudocode that summarizes the process of fuzzing an arbitrary binder service:

```
def fuzzService():
    # Working set of objects to fuzz
    binders = set(<service(s) to fuzz>)
    file_descriptors = set()

    while fuzzing data remaining:
        binder = get random element from binders
        data = getRandomParcel(binders, file_descriptors)
        transaction_code = get random int

        binders += data.getBinders()
        file_descriptors += data.getFileDescriptors()

        # Actually fuzzing the service code
        reply, status = binder.transact(transaction_code, data, ...);

        binders += reply.getBinders()
        file_descriptors += reply.getFileDescriptors()
```

This algorithm essentially keeps a working state of all the data that is exported from the service at any point in time. Then it performs a

random operation selected from the space of all combinations of operations on that data. It amends the working state with all relevant data from this operation. By repeating this process, the algorithm will eventually test all combinations of all operations.

Note that since the working set of binders also includes binders from the reply, this algorithm will crawl an entire hierarchy of binder objects that get returned. It also tracks every file descriptor that is sent or received so that it can reuse these on additional transactions, trying to cover every possible behavior a client could do from a single thread.

The real `binder_parcel_fuzzer` performs additional operations, such as closing file descriptors or deleting binders randomly to make sure we're able to test destructor-time behavior. Today, our team uses this basic algorithm to fuzz all C++ services in AOSP.

In practice, here is what such a fuzzer looks like in Android:

```
#include <fuzzbinder/libbinder_driver.h>

#include "MyBinderService.h"

using ::android::fuzzService;
using ::android::sp;

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t size) {
    FuzzedDataProvider provider(data, size);
    auto myService = sp<MyBinderService>::make(provider.ConsumeBool());
    fuzzService(myService, std::move(provider));
    return 0;
}
```

The key piece of this code is the call to `fuzzService()`, which attaches the AIDL fuzzing harness. `LLVMFuzzerTestOneInput()` is the standard fuzzing entry point, and at the beginning of this function, we set up the service. In this example, we extract a Boolean from the fuzzer input entropy with `provider.ConsumeBool()` in order to configure our service. For more complicated services, you might pull off much more data. Notice also

that, in the `fuzzService()` call, we have `std::move(provider)` passed as a parameter. This option consumes the provider entirely.

If you need to pull more data off for some reason, you can always pull off a byte array and construct multiple `FuzzedDataProvider` instances. When pulling off fuzzer data, it's critical to avoid sending the same data multiple places, which would collapse the possible states the fuzzer can visit.

With this code, the fuzzer can reach every method in `MyBinderService` as well as fuzz subinterfaces. A variant takes a list of services in case you want to inject additional binders or state.

Sometimes, while fuzzing services, those services will connect to other services on the device and communicate with them. This is extremely annoying, as it adds nondeterminism into the fuzzer and sometimes breaks other services, making debugging difficult.

For this reason, whenever a fuzzer connects to the binder kernel driver, we abort in the fuzzer and consider it an error. As a result, we know for a fact that our fuzzer is isolated. At that point, to simulate the information transfer, you have the option to inject a different dependency, fuzz a more encompassing system, or perhaps even allow this particular communication if it's not relevant to the security properties you want to enforce.

Fuzzing Multiple Services

The next question that arises is how to pick subsets of services to fuzz. We can, for instance, fuzz individual systems, or we can fuzz groups of systems together.

To help you answer this question, let's walk through an example of how you might choose to break down and fuzz two services. Consider Figure 8-2, which shows the typical architecture of an Android stack as it's used by an app.

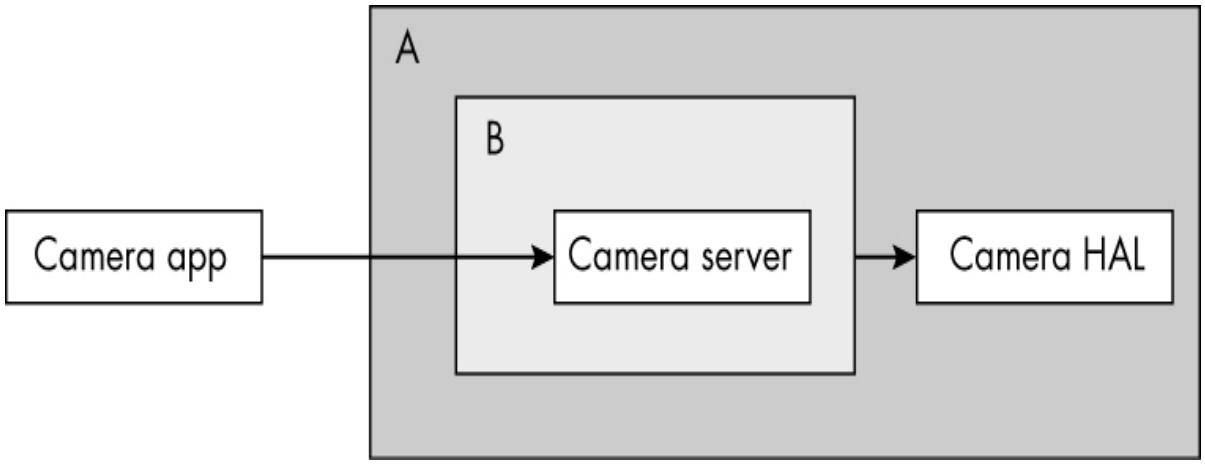


Figure 8-2: Camera app communications on Android

In this simplified diagram, an app talks to the camera server, which then talks to the camera HAL, the interface to the camera hardware, on the app's behalf. The *camera server* is the main part of the Android framework that manages the camera. So, we're considering two groups of processes: the camera server grouped with the camera HAL (group A) or the camera server by itself (group B).

When we're looking to fuzz the camera server, should we fuzz group A (the camera server and the camera HAL together), or should we fuzz group B (the camera server alone)? (We may also want to fuzz the camera HAL directly, but we're ignoring that for now.) Both approaches have pros and cons. While we'd ultimately like to try both, we may not have the resources to do so. Additionally, depending on the security architecture or attack surface you're worried about, one approach may not make sense for you.

Fuzzing Services Together

Let's first consider writing a fuzzer to cover group A. To implement a coverage-guided fuzzer for this purpose, we'd need to instantiate both services and inject them into the same process. This in itself may be a source of issues, because each piece could assume process-wide ownership of a resource, but so long as the code for the camera server and the camera HAL play well together, we can fuzz them together.

In the camera example, we'd be simulating an attack coming from the app itself. Additionally, whenever the camera server makes a request to the camera HAL, the request will probably work because it's coming from the functioning production camera driver, not randomly injected input. Now, there are caveats: The camera HAL may be running on the device and have exclusive access to hardware resources, in which case we'd need to either (1) find a way to shut down the HAL so our fuzzer can own those resources, (2) mock the hardware layer, or (3) do a mixture of both. If the services you're considering have no hardware dependencies, fuzzing becomes much simpler, as the service is more likely to work as is.

Because mocking doesn't represent the real service behavior and requires implementing additional code you'd need to maintain, we'd recommend it only as a last resort or when a service can't be rewritten as a mechanism. Fuzzing the real code will always produce a more accurate representation of the device's security.

Fuzzing Services Separately

Now we'll consider case B. How might we fuzz the camera server without providing a camera HAL? The easy answer is to use `RandomBinder`, a class that can simulate arbitrary communication, to simulate a camera HAL.

One advantage of this approach is that it allows you to simulate an attacker coming from the camera HAL in addition to one coming from the camera app. Now, this isn't a likely scenario, and it may be more applicable to other services. Even so, an attacker who compromises the camera HAL itself may be able to escalate the attack into the camera server. The use of `RandomBinder` could cover this case.

Of course, if this attack is considered relatively unlikely or if the camera server has fewer permissions than the camera HAL (making it less of a target), prioritizing fuzzing in such a configuration might not make sense.

The other downside is that this approach is significantly slower to discover normal use cases. When we use the real camera HAL, and then

the fuzzer makes a normal request, the camera server will likely be able to service it. When we randomize both the app input and the camera HAL input, we double the number of factors that need to go right to find meaningful coverage.

Coverage-guided fuzzers are extremely efficient when the incremental amount of effort to find additional coverage has a reasonable probability. However, the more code we have to get past in our fuzzer, the more likely we are to accidentally introduce a case where the fuzzer is very unlikely to discover incremental coverage. In any case, it's paramount to monitor the fuzzer's coverage and adjust your strategy to account for it.

When testing three or more services, the number of ways you can fuzz them grows exponentially. Make these decisions on a case-by-case basis, grouping service fuzzers in such a way as to cover every service.

Optimizing Coverage: A Binder-Specific Example

Fuzzers often get stuck and stop finding new coverage when they've been running for a long time. As we've discussed, certain patterns in code are extremely hard for fuzzers to pass. Let's review a case that applies directly to binder fuzzers.

For some AIDL-generated code, the binder runtime requires that the entire `Parcel` has been read. If not, the runtime treats this as a canary check that something has gone wrong and rejects the transaction in the hopes that this will help us discover real issues.

This behavior can be annoying. First of all, the check doesn't always indicate a vulnerability. Also, if an attacker controls a `Parcel`, they can probably add or remove data from it to match the desired length. And, when fuzzing, this check happens on almost every run, becoming one of the biggest dead ends, because almost all coverage improvements have to satisfy this extra dimension.

So, when we fuzz, we no longer enable this check. If a transaction is left with extra data after it ends, we don't consider this to be an error. Instead, we pretend that the random data isn't there and continue executing the service code to test it on the input.

This approach makes it more likely that a random perturbation of the input results in more coverage, so it makes the fuzzer discover more issues and find new coverage more quickly. Removing multiple such bottlenecks if they occur in series will exponentially increase the performance of your fuzzer.

WHEN FUZZERS ARE CHAOTIC SYSTEMS

You may notice that `binder_parcel_fuzzer` adds binder and file descriptor objects to the working set after each transaction. Even when fuzzers are deterministic, this type of stacking structure—where long fuzzer runs will take drastically different actions depending on what happened earlier in the fuzzer—makes it more likely for your fuzzer corpus coverage to shrink when the corpus is minimized for its coverage (as mentioned in "Automating the Fuzzing Life Cycle" on page 37). While this doesn't form a hard-blocker, it does add a cost, as your fuzzer may have to frequently restart its coverage search.

For example, imagine that a function in your interface returns an optional binder object. Your fuzzer may have a corpus entry that represents the following: first, someone calling this method and not getting a binder back (the empty option), and second, someone calling different methods on the original binder object 500 times. Now, say someone makes a change to the service code, causing a binder (and never the empty option) to get returned in the first case. When the fuzzer hits this case, the fuzzer will add the new binder object to its working set, and it may make 500 invalid calls to this different binder object, instead of the original object this coverage was built on, vastly reducing the covered space.

This is an example of chaos in the fuzzer definition, where a small change in initial conditions leads to drastically different results.

Roughly speaking, the more combinatorial a fuzzer is, the more it will be subject to chaos when the code under test changes.

Chaos isn't a reason to panic or to avoid combinatorial fuzzers. However, it may be a reason to bundle changes together so you can run fuzzers for longer on each group of changes, rather than rebuilding and resetting a fuzzer each time a change is made.

Once you've gotten in-process fuzzing to work for a particular service, an empirical side effect occurs: You'll be able to perform black-box fuzzing of services without causing the device to become unusable. Of course, without coverage guidance, black-box fuzzing can only ever hope to cover a small percentage of the state of the system. However, black-box fuzzing can also provide a good runtime test that your fuzzers are working. If you write fuzzers and fix issues in them, you should be able to see that the stability of your real system increases when you make random calls. If you don't see this happen, your fuzzers may not be testing the real attack area on the real system.

Since enabling black-box fuzzing makes it easier for attackers to automatically interrogate the system as well, you may consider adding metrics, honeypots, or arbitrary crashes whenever you detect absurd input in production. If you're fuzzing an entire system, you won't be able to react quickly to an attack at that scale on previously hard-to-reach surfaces, so you need to make sure to anticipate and prevent any negative secondary effects.

Occasionally, you'll want to fuzz other aspects of services, their configurations, or their other interfaces while also fuzzing the methods that they expose over Binder. In this case, you can peel off a part of the fuzzing input data to be used against these other surfaces while fuzzing the service itself. The easiest way to do this is to create a separate local binder object, which fuzzes other surfaces in the process. A fuzzing framework or tool is more effective when it can be used as a building block to create a more complicated fuzzer. Interleaving operations among multiple attack surfaces on a fuzzer will always cover as much or

more state space than running those operations all before or all after the interleaved operations.

Considering Structure-Aware Fuzzing

Recall that structure-aware fuzzing involves generating fuzzing data based on a higher-level description. For instance, Android's AIDL gives us the format of most interfaces. So, why don't we do structure-aware fuzzing for AIDL interfaces?

We've followed this approach in Android before. For instance, the deprecated HIDL used to allow the fuzzing of HALs in Android with structured-based fuzzing. However, this approach failed to test some possibilities and was quite complicated to enable. Additionally, not all code in Android uses AIDL, so structure-based fuzzing wouldn't be able to test these interfaces.

If you're not able to fix the errors in the stack below you, structure-based fuzzing can help. Additionally, if your coverage is not good enough, structure-based fuzzing can be a quick shortcut. However, you'll always cover more state space by ignoring the structure and testing all possibilities. So, if you can get a coverage-guided approach to work, you'll have better security properties.

For binder fuzzing, one tool can help with discovering coverage in the larger search space without structure-aware fuzzing: `binder2corpus` takes recordings of binder transactions and converts them to valid fuzzing corpus entries. Instead of taking an arbitrary string and producing a behavior, `binder2corpus` takes the behavior and tells you the string that would produce it. In practice, this tool allows fuzzers to attain coverage that they aren't able to reach from coverage-guided fuzzers even after months of runtime.

We won't go into detail on how to create an engine like `binder2corpus`. However, know that certain techniques make it easier to maintain such a tool. Specifically, once you invest in a tool like this, keeping the binary layout of the fuzzer format steady means that you don't have to rerun tests and re-create corpus entries for certain fuzzing engine changes.

For instance, rather than repeatedly consuming a Boolean, consume a single integer and use it as a bit flag.

The more corpus entries you have, the more expensive it is to change your fuzzing engine, so if you're fuzzing an entire system, be intentional about how it's laid out. If you aren't sure, one good exercise is to use a binary text editor to create a corpus entry for your fuzzer by hand.

The Human Element of Large-Scale Fuzzing

So far, this chapter has talked about the technical details of system service fuzzing, but when you fuzz an entire system, you also have to take into account how your team, and the people who interact with that system, operate.

Fuzzing systems at scale requires considering how you'll detect when issues occur, resolve new issues as quickly as possible, and communicate system requirements across an organization. In this section, we'll touch briefly on the more human side of fuzzing.

Developing a Strategy for Scaling

One goal of system fuzzing is to make sure that all core components of Android have been fuzzed and remain fuzzed. As you scale, ensuring that this remains the case will require organization.

On Android, for example, we enforce this standard in review by adding a requirement that every service context (the SELinux label for a top-level service registered in Android) corresponding to a service written in C++ has a fuzzer. The Android build system enforces this list, and you can find it in *system/sepolicy/build/soong/service_fuzzer_bindings.go*. Whenever anyone adds a service in Android, they must add a fuzzer; only security reviewers have permission to approve an exception.

Testing the Fuzzer Framework

Over 50 fuzzers for core AOSP components use the fuzzing framework for binder services. If a bug gets added into this fuzzing engine, we may

lose fuzzing coverage for a sizable fraction of Android fuzzers. Additionally, if the bug causes fuzzers to crash quickly, we may trigger a wave of bugs on dozens of teams.

To prevent these outcomes, it's helpful to have a dedicated fuzzer-testing framework that tests the fuzzing engine itself. We have one, called `fuzz_service_test`. The framework lets us configure a binder service to crash in any one of a number of places. When we run the fuzzing engine, we make sure it can detect a crash in each of these places. If we break the fuzzing engine, this test will fail.

As an unexpected side effect, this test helps us optimize the fuzzing engine. The flake rate of this test corresponds to the efficiency of the fuzzer. If a change gets introduced to Android that causes the fuzzer to be less efficient, `fuzz_service_test` flakes more. Originally, this test flaked quite a bit; we were able to see the flake rate get driven down as we optimized the fuzzer.

Wrapping Up

After reading this chapter, you should have a good understanding of how Android system services are fuzzed, as well as some of the problems that we encountered along the way when developing this approach, what factors we considered, and what you should think about if you, too, want to fuzz systems at scale.

While our team performed the initial work fuzzing services in Android, we hope everyone gets exposed to the tools we've covered and can use them to make their services as secure as possible.

OceanofPDF.com

9
THE ANDROID KERNEL



Fuzzing operating system kernels presents some unique challenges, and Android's kernel is no exception. We can use several tools to perform kernel fuzzing. In this chapter, we'll look at two of them: Syzkaller and Linux Kernel Library (LKL). We'll describe the unique approach of each as well as how to set them up to begin fuzzing.

Why Kernel Fuzzing Requires a Different Approach

So far in this book, we've fuzzed userspace code, applying the data generated by the fuzzing engines to simulate input from a file, a network, or a library's public API. Fuzzing an operating system kernel requires a different approach. We begin by asking, What might a potentially harmful program do to the operating system?

Unix-like systems (including Linux and Android) represent most system resources as files. Applications can communicate with the kernel by opening special files (mostly in */dev/*, */proc/*, and */sys/*) and reading from or writing data to those files.

A kernel fuzzer generally works by generating and then mutating a series of syscalls to try to exercise as much of the kernel's code as possible.

The fuzzer knows which syscalls exist, as well as how data flows in and out of them, based on descriptions compiled with the fuzzer.

As with all fuzzing, the accumulation of state from multiple fuzzed inputs can cause problems for reproducibility. When fuzzing userspace code, we used techniques like resetting the global state or restarting the fuzzed application to streamline the process. When dealing with kernels, however, this global state may often be an important part of the bugs you're looking for. As a result, some kernel bugs can be difficult to reliably reproduce because of this global state and nondeterminism from multithreading, parallelism, and race conditions.

Further, many kernel bugs cause the device to lock up or completely shut down, making collecting crash information difficult. Tools like Syzkaller get around these challenges by reading the serial output from the kernel if available.

Finally, you should expect kernel fuzzing to be much slower than user-space fuzzing. This is a result of the many context switches occurring every time a kernel syscall is made, the slow process of restarting a physical or virtual device after each crash, and the kernel's inherent nondeterminism. (As you'll see later, architectural frameworks like LKL offer a work-around to some of these performance bottlenecks.)

Preparing a Kernel for Fuzzing

The kernel that runs on essentially all production Android devices isn't configured for fuzzing, including the default kernel that boots on Cuttlefish virtual devices. Before you can fuzz Android kernel code, you'll have to get a kernel with all the required features included and enabled.

Downloading and Building Kernel Code

To get a kernel built with the correct settings, download the source code for the Android Common Kernel (ACK). The AOSP source code you may have downloaded earlier contains prebuilt kernels used for building device images, but these aren't configured correctly for fuzzing.

You can find instructions for downloading the Android kernel at *https://source.android.com/docs/setup/build/building-kernels*. These instructions assume that you've already set up the `repo` tool, as discussed in Chapter 2.

You'll notice right away that the kernel repository has many branches. For now, use the *common-android-mainline* branch to get the latest version. It should contain the configurations you need to build kernels for virtual Cuttlefish devices.

Several other branches are available for older versions of Android. Google also makes the kernel source code for Pixel device drivers available on other branches. If you have a different kind of device, you may be able to find the appropriate source code for it on the manufacturer's developer website. For example, the instructions for building Google Pixel kernels are available at *https://source.android.com/docs/setup/build/building-pixel-kernels*.

The Android kernel is organized slightly differently from a standard Linux kernel. That standard Linux kernel code mostly comes from the upstream open source Linux kernel project contained in the *common/* directory, called the Generic Kernel Image (GKI).

Once you've downloaded the code, you should be able to build the kernel. First, create a directory in which to place the build artifacts. Then pass that directory as an argument to the build command:

```
$ mkdir out
$ tools/bazel run //common:kernel_x86_64_dist -- --destdir=out
```

Android versions older than 14 can rely on a build method using a *build.sh* shell script. For versions earlier than Android 12, this is the only way to build the kernel. As the Bazel build system has been in use for several years at the time of this writing, we'll stick to it here.

If you look in the *out/* directory, you'll see a variety of artifacts. These don't contain all the drivers needed to boot a device (even a virtual one), so you'll need to run an additional command to build those drivers and package everything up into the relevant disk images.

Android-specific and device-specific kernel drivers are contained in separate directories and are built as kernel modules, which are loaded at runtime. This approach makes it easier for the AOSP and device vendors to update the kernel version running on a device.

The modules needed for the Cuttlefish virtual device are in the *common -modules/* directory, and you must build them separately with a different build rule:

```
$ tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist \
-- --destdir=out
```

In fact, because this build target for Cuttlefish virtual devices depends on the main GKI kernel, the single command shown here is enough to build the kernel in the future.

Now, check the *out/* directory to make sure it has the two artifacts needed to boot a Cuttlefish device:

bzImage The GKI kernel image

initramfs.img The initial RAM disk image, including virtual device drivers

Copy both artifacts into the directory containing your Cuttlefish installation.

To boot Cuttlefish with the newly built kernel, start it with the *launch_cvd* script, passing two additional flags pointing to the artifacts:

```
$ HOME=$PWD ./bin/launch_cvd --daemon --kernel_path=./bzImage \
--initramfs_path=./initramfs.img
```

Once the device is running, you can check that it has booted your newly built kernel with this command:

```
$ adb shell uname -r
6.12.0-mainline-maybe-dirty
```

The `6.12.0` portion may differ depending on the version of the Linux kernel you downloaded. The `-maybe-dirty` suffix indicates that the virtual device is using the kernel you just built. If the virtual device were instead using the standard kernel that shipped with Cuttlefish, you'd see the `ab*` suffix instead:

```
$ adb shell uname -r
6.6.57-android15-8-g8b48c9979699-ab12748506
```

This suffix gives you the build number from the Android continuous integration system that Cuttlefish shipped with.

Turning on Fuzzing Instrumentation

Syzkaller relies on the Linux kernel's coverage mechanism, KCOV, to fuzz effectively. KCOV was designed specifically for fuzzing and allows a process to collect data about which parts of kernel code are exercised by another process. While KCOV is available in both the upstream Linux kernel and the ACK, it's not enabled as part of the default kernel build.

We'll also need to enable the Kernel Address Sanitizer (KASAN), which works similarly to sanitizers like AddressSanitizer (ASan) and Undefined-BehaviorSanitizer (UBSan) for kernel code.

To enable these features, we must set the `CONFIG_KASAN=y` and `CONFIG_KCOV=y` options. Conveniently, the Android kernel build scripts make building a Cuttlefish-compatible kernel with these options easy. Just add the `--kasan _generic` flag, as shown here:

```
$ tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist \
--kasan_generic -- --destdir=out
```

Once the build completes, boot the virtual device with the new images, as before. If everything has worked correctly, you should see

that KCOV is enabled when you check the */sys/kernel/debug/kcov* file. First, you'll need to mount `debugfs` via these commands:

```
$ adb shell setprop persist.dbg.keep_debugfs_mounted 1
$ adb shell mount -t debugfs none /sys/kernel/debug
$ adb shell ls /sys/kernel/debug/kcov
```

If the kernel doesn't have KCOV, you'll get an error message that says `/system/bin/sh: /sys/kernel/debug/kcov: inaccessible or not found.`

Syzkaller

Now that you have an unmodified kernel built with the required instrumentation, you can fetch the code for Syzkaller and configure it to fuzz your virtual device. Compared to the fuzzing tools you saw previously, Syzkaller takes some effort to get set up and working. It has many switches to flip and knobs to turn.

Setting Up

A variety of contributors, from Google and elsewhere, are actively developing Syzkaller on GitHub. Syzkaller also has many components that need to be configured at compile time. This means it's distributed as source code, not in a precompiled executable, and you'll first need to fetch the latest Syzkaller code from GitHub:

```
$ git clone https://github.com/google/syzkaller.git
```

This code contains a useful tool called `syz-env`, which handles installing all dependencies inside a container. This is the most convenient way to build Syzkaller and its related tools, so it's what we'll use here. As this process requires Docker, you'll have to install it if you don't already have it.

NOTE

If you'd rather not install or use Docker, you can also build Syzkaller without the `syz-env` tool. You'll need a recent version of the Go compiler

as well as quite a few additional dependencies. If you're on a Debian or Ubuntu system, you can use the `install_prerequisites` *target in* Makefile *to install these.*

You can now build Syzkaller with this command:

```
$ tools/syz-env make
```

When the build finishes, you should see that it's created a *bin/* directory with a collection of executables:

```
$ ls bin
linux_amd64  syz-db  syz-manager  syz-mutate  syz-prog2c  syz-repro
syz-sysgen  syz-upgrade
```

You should be able to run these without `syz-env`:

```
$ ./bin/syz-manager
[FATAL] no config file specified
```

This error message is expected. Soon, we'll put together a proper configuration file and begin fuzzing.

You can enable additional logging from Syzkaller by passing the `-vv` flag with a verbosity level, such as `-vv1`, when running `syz-manager`. Higher numbers will enable progressively more logging. You shouldn't need logging while following the examples in this chapter, but it can be helpful for debugging problems with your configuration or learning more about what Syzkaller is doing behind the scenes.

Getting to Know Syzkaller

Syzkaller supports many operating systems, including Linux, FreeBSD, Darwin, and Microsoft Windows. We'll focus here on how Syzkaller works for Android; some of the details may differ for other platforms.

Unlike most other fuzzing tools, Syzkaller has multiple components that work together to perform the fuzzing. You may have seen some of

these components earlier when checking build results in the *bin/* directory.

Most of Syzkaller runs a manager process, the `syz-manager` executable, on the host machine. When the manager starts, it will launch one or more other systems to perform fuzzing. For Android, this involves connecting to Android devices (whether virtual or physical) via the ADB as well as listening to the Android device's serial output to read crash information in case the device goes offline, which can often happen as a result of fuzzing.

NOTE

For most non-Android operating systems, fuzzing occurs on a VM running in QEMU or on a cloud instance. For that reason, you may see the term VM *throughout the Syzkaller source code and in its documentation.*

Once the system under test is available, Syzkaller will connect to it via ADB and copy over an executable called `syz-executor`. One of the goals of Syzkaller's design is to keep the `syz-executor` code small and efficient so it doesn't cause many side effects or slow the fuzzing process unnecessarily. This lightweight process runs on the system, receives a series of syscalls (equivalent to a program) from `syz-manager` over a remote procedure call (RPC), and executes them on the system under test. It then reads information back from the kernel using KCOV and sends it back to `syz-manager`.

After receiving this coverage information from the device, the `syz-manager` instance adds any new paths the input discovered through the kernel's code to the fuzzing corpus, which is stored in a database, and uses it as the basis for future inputs.

Additionally, `syz-manager` aggregates information about the coverage, corpus size, and crashes found and publishes them on a dashboard that you can view in a web browser. If the `syz-manager` process is stopped, it will persist its state to disk so it can be restarted again later.

You can find information about which syscalls are available for a given operating system built into `syz-manager` when Syzkaller is compiled. The *sys/* directory holds the extensive list of descriptions distributed with Syzkaller. We'll look at the language those descriptions are written in, and add some additional ones, in Chapter 10.

Writing a Syzkaller Configuration

When starting Syzkaller, you'll need to pass it a properly populated configuration file. These are JSON formatted and follow the schema defined in *pkg/mgrconfig/config.go*. The schema has numerous options, many of which are optional, so we'll start with the basic ones. The following configuration should get a basic Syzkaller instance up and running:

cuttlefish-basic.json

```
{
  "name": "cuttlefish-basic",
  "target": "linux/amd64",

  "http": ":8080",
  "workdir": "workdir",
  "syzkaller": ".",

    "type": "adb",
    "vm": {
    "devices": [
      {
        "serial": "0.0.0.0:6520",
        "console_cmd": ["tail", "-f", "path/to/kernel.log"]
      }
    ]
  }
}
```

Table 9-1 explains the meaning of each of these fields.

Table 9-1: Standard Syzkaller Configuration Fields

Field	Example value	Meaning
`name`	`cuttlefish`	The name of the Syzkaller configuration in the UI
`target`	`linux/amd64`	The target operating system and architecture
`http`	`:8080`	The host and port on which to serve the UI
`workdir`	`./workdir`	The path to the working directory for storing Syzkaller's state: the corpus, crash artifacts, and so on
`syzkaller`	`.`	The path to the root of the Syzkaller checkout
`type`	`adb`	The type of virtual or physical device to fuzz
`vm`	`{...}`	VM-specific configuration

You may want to change the `workdir` and `syzkaller` fields to their full path. Also notice that we use the `linux` operating system type and `amd64` architecture corresponding to an x86_64 Cuttlefish instance. If fuzzing a physical device, you'll likely want to use `linux/arm64`.

For Android devices, Syzkaller uses the `adb` VM type, which works for both physical and virtual devices.

NOTE

If you look through the Syzkaller source code, you'll also notice a `cuttlefish` *VM type. This is intended for running multiple Cuttlefish*

instances on Google Compute Engine, using nested virtualization. While useful for scaling up Android kernel fuzzing over a large number of machines, the setup for this approach is very complex.

For each device you want to fuzz, you'll need to add an entry to the `devices` array. For the examples in this chapter, we'll use just a single Cuttlefish virtual device, although you could certainly run more if desired. Similarly, you could list multiple physical devices if you had them connected to your host machine, although you cannot mix physical and virtual devices of different architectures (say, ARM and x86_64).

Depending on the type of the `vm` field, different options may be available for the device array. For ADB devices, the only required field is `serial`, which defines the serial number (as shown in ADB) for the device you want to fuzz.

Because Syzkaller needs to get stack traces from KASAN when it causes a crash, Syzkaller works best when it has access to the serial console. Table 9-2 describes the options for providing this access.

Table 9-2: Serial Console Configuration Options

Field name	Example value	Meaning	Notes
`console`	`"adb"`	Run `dmesg -w` on the device	Less reliable when crashes cause device to go offline.
`console_cmd`	`["tail","-f", "path/to/kernel.log"]`	Command for getting a serial log	Cuttlefish writes its logs to a file called *kernel.log*.
`console`	`"/dev/ttyUSB0"`	Path to a serial device from a USB adapter	Used for physical Android devices

Field name	Example value	Meaning	Notes
			with a special serial adapter.

Since we're using Cuttlefish devices, the example configuration uses the `console_cmd` option to read a file inside the Cuttlefish directory *kernel.log*, usually inside *cuttlefish/instances/cvd-1/*.

You might be wondering why you can't just use `console` with the path to the logfile. Syzkaller expects a TTY-like device that it can make specific `ioctl` calls on. If Syzkaller gets a "normal" file, it will respond with errors. Specifying `tail`, as we do in our configuration, makes the file behave enough like a TTY that Syzkaller can make the `ioctl` calls it expects.

If you have multiple Cuttlefish instances running, you may have files called *cvd-2*, *cvd-3*, and so forth. It's important to make sure that the `console` values are associated with the correct serial numbers. A good way to check is to stop all the instances, then start them one at a time and run `tail -f` on each *kernel.log* and see which is getting output.

You can find many more configuration options for Android devices defined in *vm/adb/adb.go*. For now, `serial` and one of the `console` options suffice.

NOTE

If you're interested in fuzzing physical Android devices, you'll need a way to get the kernel logs over a serial port. Depending on the specific device model, you may need to set some bootloader options and use a special USB adapter. In those cases, the serial output will often show up on the host machine as a device file like /dev/ttyUSB0.

You should now be able to start Syzkaller with your new configuration:

```
$ ./bin/syz-manager -config=cuttlefish-basic.json
```

After Syzkaller starts up, you'll be able to access its UI in a browser at the port you specified in the `http` field of the configuration file (Figure 9-1).



Figure 9-1: The Syzkaller web UI running on a Cuttlefish instance

This UI provides many features for viewing information about which syscalls have been used, the fuzzer's performance, and any identified crashes. We'll look at some of the specific features in Chapter 10, but feel free to explore them on your own as well.

Fuzzing Inside a Sandboxed Environment

We should talk about one more configuration option before moving on, although we won't use it in this book. As discussed in earlier chapters, Android applications and system services are sandboxed in a variety of ways to prevent malicious or compromised applications from affecting other applications' data.

If you examine public Syzkaller dashboards, you'll find hundreds of bugs in the upstream Linux kernel, not all of which have been fixed. Android's sandboxing means that many of these upstream bugs don't affect Android devices.

By default, Syzkaller runs the fuzzing process as the root user with nearly full permissions. When fuzzing actual Android device code, it can be a good idea to fuzz only syscalls that would be available in a more constrained context to avoid surfacing these unexploitable bugs. Table 9-3 lists the arguments you can add to the top level of the configuration to achieve this.

Table 9-3: Sandbox Configuration Options

Field name	Value	Meaning
`sandbox`	`"android"`	Fuzz from a context that simulates an Android sandbox.
`sandbox_arg`	`0`	Use the untrusted application context.
`sandbox_arg`	`1`	Use the system service context.

While these options aren't required, they make fuzzing more likely to find bugs that are relevant to Android's security. It's also generally a good idea to use these sandboxing features when fuzzing physical

Android devices. Full root access allows the fuzzing process to do lots of unexpected things, some of which may brick your device!

Linux Kernel Library

In the previous section, you learned how to fuzz the Android kernel with the Syzkaller fuzzer in a virtual environment by targeting kernel interfaces exposed to userspace programs via syscalls. In this section, we'll explore another approach for fuzzing Linux kernel interfaces without running the kernel under test in a VM. By leveraging LKL, we can make fuzzing kernels no different from fuzzing regular userspace code.

LKL is an open source project that aims to run the Linux kernel in userspace. It essentially enables us to build the kernel into a shared library, which we can link into a userspace program, thus enabling us to run kernel functionality directly in userspace.

LKL makes it possible to make state-of-the-art technologies implemented in the Linux kernel available to unprivileged userspace programs on various platforms. For instance, programs can use a network stack or filesystem implemented in the Linux kernel directly without invoking the syscalls of the host kernel. At the time of this writing, LKL works on GNU/Linux, Windows, and macOS hosts.

One of the earliest and most prominent public uses of LKL to fuzz the Linux kernel appeared in the Janus fuzzer, which targeted the filesystem implementation in the kernel. In this work, the authors leveraged LKL to identify more than 100 unique crashes in the filesystem implementation, resulting in 32 CVEs.

More broadly, LKL is useful for fuzzing because it enables us to create small, highly optimized kernel fuzzers targeting specific interfaces without relying on VMs. All we need to do is build the kernel for the LKL architecture, then link the static kernel library against a libFuzzer-based fuzzer harness.

The Structure of an LKL-Based Fuzzer

An LKL-based fuzzer generally consists of three components:

The LKL object file (*lkl.o*) This object file can be linked to a userspace library. A product of the Linux kernel build process for the target architecture, it's produced from a corresponding vmlinux file and implements the target interface to fuzz. When writing a fuzz target, it's essential to ensure that the kernel configuration used in the build process includes kernel components with the code you'd like to fuzz.

LKL utility code The utility code implements necessary support functionality to the LKL kernel library in the host operating system environment. The code includes an interface between LKL and the host operating system that handles memory allocation, thread creation and synchronization, and other system-level tasks. It also implements specific virtio devices, such as block devices and network adapters. Finally, it implements utility routines for tasks such as mounting filesystems and adding virtio devices.

The fuzz harness The entry point for the fuzzing logic. The fuzzer harness implements code that provides mutated input data to the target kernel interface implemented in the library.

Example: Fuzzing the Human Interface Device

In the LKL GitHub repository, you'll find several fuzzers that target the human interface device (HID) subsystem, the layer in the Linux kernel responsible for managing and parsing HID descriptors over various bus types, including USB and Bluetooth. Looking at the HID implementation in the Linux kernel reveals complicated code that parses data received from USB devices. This code presents quite a dangerous attack surface.

Most notable is the code that parses HID reports and device descriptors by the kernel's HID drivers. If a malicious USB device presented malformed data to the host and the driver didn't handle the input correctly, security issues could result. Early investigations into

USB security revealed that Linux HID drivers had the potential for many vulnerabilities, prompting efforts to fuzz them via LKL.

During the fuzzing process, the fuzz harness performs a specific sequence of actions. First, it adds a virtio block device to emulate a hard drive. The harness then uses the fuzzer input to create a virtual device or filesystem; for a HID fuzzer, this involves creating a virtual HID device and feeding the fuzzer input as HID reports. Next, the harness triggers the kernel code used to parse this data. After the interaction completes, it cleans up the created resources. By instrumenting the kernel code with coverage tracking, the fuzzer can intelligently mutate the input to explore new code paths within the driver.

This approach has uncovered significant vulnerabilities that traditional fuzzers routinely miss. For instance, when researchers used LKL to target the USB HID subsystem, they uncovered critical OOB write vulnerabilities capable of leading to kernel code execution.

What makes these findings particularly instructive is that mature Syzkaller USB fuzzing had been running for years without discovering them. Because Syzkaller operates at the syscall layer of a live running operating system, it introduces scheduler noise and macro-level nondeterminism. LKL, by contrast, isolates the target driver code entirely, allowing a libFuzzer harness to mutate input reports with a level of speed and deterministic precision that exposes state-dependent logic bugs that VM fuzzers simply miss.

The High-Level Workflow for Constructing an LKL Fuzzer

While setting up Syzkaller consists of defining syscall relationships and managing JSON configurations, building an LKL fuzzer requires writing code. Think of it as writing a standard libFuzzer target, but with an extra step to boot the embedded kernel.

Building an LKL fuzzing harness follows this blueprint:

1. Start by initializing the LKL environment by invoking LKL utility routines inside the fuzzer's initialization block (executed once at startup) to bootstrap the embedded kernel. This sets up host memory allocation and threads.

2. Configure your virtual hardware. If you're fuzzing a filesystem, your initialization code will instantiate a virtual virtio block device. For a network driver, you would instantiate a virtual network tap.
3. Create the fuzzing loop. The `LLVMFuzzerTestOneInput` entry point receives a raw array of bytes (`Data`, `Size`) directly from libFuzzer.
4. Inside the loop, pass those raw bytes directly to the kernel subsystem being fuzzed. For example, to fuzz filesystems, write the fuzzer bytes to the virtual block device, then call LKL's internal `lkl_sys_mount()` function to see whether the parsing logic crashes while attempting to mount the corrupted disk image. For network protocols, inject the raw fuzzer bytes directly into LKL's virtual network interface to test the packet-parsing stack. And for HID drivers, feed the fuzzer data directly into the subsystem's report-parsing functions.
5. Clean up the state. Before the current fuzz iteration ends, you must undo the actions (such as unmounting the filesystem or freeing the virtual device descriptor) to ensure that the next iteration starts with a clean, predictable kernel state.

By combining the precision of libFuzzer with the low-level code of the Linux kernel inside a single binary, you can create a highly optimized debugging workflow that can find deep parsing bugs.

Syzkaller vs. LKL

Now that we have looked at both frameworks, the next step is to decide when to deploy Syzkaller versus when to build an LKL-based fuzzer. They represent two fundamentally different approaches to kernel fuzzing.

Syzkaller acts as a high-level fuzzer. It understands the relationships between multiple system calls, allowing it to simulate real-world program execution across a running operating system. This enables Syzkaller to find race conditions or bugs spanning multiple subsystems, but it suffers from heavy VM overhead.

LKL, on the other hand, is a low-level fuzzer. It strips away the rest of the operating system and allows you to treat a kernel driver exactly like a userspace library. LKL excels at rapidly mutating raw data structures, but it may lose the broader system context.

Table 9-4 outlines these core differences to help you choose the right tool for your target.

Table 9-4: Syzkaller vs. LKL

Feature	Syzkaller	LKL
Execution environment	A full VM (such as Cuttlefish or QEMU) or physical device	Single userspace process on the host operating system
Fuzzing interface	The syscall boundary via descriptions	The target C function
Input mutation	Structured, grammar-aware sequences of syscalls	Raw byte mutations (for example, libFuzzer bit flips)
Speed	Slow (tens to hundreds of inputs/sec) because of context switches and reboots	Fast (thousands of inputs per second); no VM overhead
Crash recovery	Heavy (requires a full VM or device reboot)	Instant (the userspace process simply restarts)
Best used for	Fuzzing complex multi-syscall logic, stateful subsystems, and system integration	Fuzzing isolated drivers and data-parsing logic (like file-systems, network packets, and HID reports)

Wrapping Up

In this chapter, we covered the two kernel fuzzing tools, LKL and Syzkaller. In the next chapter, we'll configure Syzkaller to run against a virtual Android device with a sensible configuration, then tweak it to do more focused fuzzing.

OceanofPDF.com

10
FUZZING CUSTOM KERNEL DRIVERS



In the previous chapter, we discussed configuring Syzkaller and LKL to fuzz the kernel on a virtual Android device by using the standard syscall descriptions provided by Syzkaller. However, many Android devices come with specialized, device-specific kernel features for which Syzkaller doesn't have descriptions. If you want to experiment with fuzzing these kernel drivers, you'll have to write your own descriptions.

Because the available drivers differ for each Android device, this chapter will walk through a simple example: adding a new, fictional kernel driver that is compatible with a Cuttlefish virtual device. Once it's running, we'll write a description to exercise that new driver and find a bug.

Introducing a New Driver

At this point, you should have Syzkaller up and running on an Android kernel. Many Android devices ship with device-specific components, which have drivers that aren't used in the Cuttlefish emulator. These drivers often control cameras, graphics chips, image and media

decoding, cellular modems, fingerprint readers, and lots of other interesting things. The kernel drivers for these components may not be included in the upstream Linux kernel.

Let's introduce a new fictional kernel driver to integrate into the kernel used for booting Cuttlefish. The basic idea of this driver is that it represents a simple combination lock, similar to the ones you may have used in school. The lock's numbered dial needs to be rotated first counterclockwise to a specific number, then clockwise to a different number, and then counterclockwise again to the combination's final number. After that, rotating the dial clockwise will disengage and unlock the box.

The driver makes this functionality available as two new device node entries in */dev/*: */dev/padlock* and */dev/lockbox*. The padlock allows you to clear any in-progress combination or turn the dial in either direction before stopping at a particular number.

If you've entered the correct code, you'll be able to read a secret message by opening the */dev/lockbox* file. Before the code is entered, or if you move the dial again afterward, reading from */dev/lockbox* will fail with an error.

Let's walk through the code to create this combination lock driver and add it to Cuttlefish.

Adding a Driver to the Kernel

One of the main goals of Android's GKI system is to separate manufacturer-or device-specific driver code from the main kernel. To ensure that these components interface correctly, driver code must follow certain conventions. Although you probably won't be releasing this example driver on a real device, we'll follow the conventions you should use to do so.

First, create a new directory called *lockbox* under *common-modules/virtual -device/* in your kernel checkout and create two files, *lockbox.h* and *lockbox.c*, to contain the code:

```
$ mkdir -p common-modules/virtual-device/lockbox
$ touch common-modules/virtual-device/lockbox/lockbox.h
```

```
$ touch common-modules/virtual-device/lockbox/lockbox.c
```

Before implementing any of the logic for the driver, we need a little bit of boilerplate code. While this code doesn't do anything, it allows the driver to be correctly included in the kernel.

First, create a header file for the driver containing these minimum fields. Eventually, we'll define some important constants here, but for now, we'll basically keep the file empty:

lockbox.h

```
#ifndef _LOCKBOX_H
#define _LOCKBOX_H

#endif /* _LOCKBOX_H */
```

You'll also need to add the basic outline of a kernel driver to *lockbox.c*:

lockbox.c

```
   #include <linux/miscdevice.h>
   #include <linux/module.h>

   #include "lockbox.h"

❶ MODULE_DESCRIPTION("lockbox example driver");
   MODULE_AUTHOR("Your Name");
   MODULE_LICENSE("GPL");

   static int __init lockbox_init_module(void)
   {
     return 0;
   }

   static void lockbox_cleanup_module(void)
   {
   }
```

```
   MODULE_ALIAS_MISCDEV(MISC_DYNAMIC_MINOR);

❷ module_init(lockbox_init_module);
   module_exit(lockbox_cleanup_module);
```

Again, this is the bare minimum you need to register the new driver with the kernel. The `MODULE_` macros define some metadata about the module's description, author, and license ❶. The `module_init()` and `module_exit()` functions are responsible for initializing and cleaning up the driver when it is loaded during the boot process and stopped when the system shuts down ❷. You'll need to add code to both of these elements later, but for now, we've included enough to get the driver to build.

Just having these files present in the kernel source tree isn't enough to have them included when the kernel is built; we need to modify the build rules to include these new files. You can find the build rules in *common-modules/virtual-device/BUILD.bazel*.

In *common-modules/virtual-device/*, you'll see a variety of other build rules already defined. The rules we'll add will be quite similar to those, so I'll show only the new rules here:

BUILD.bazel

```
filegroup(
❶ name = "lockbox_sources",
    srcs = [
        "lockbox/lockbox.c",
        "lockbox/lockbox.h",
    ],
)

ddk_module(
 ❷ name = "x86_64/lockbox",
    srcs = [":lockbox_sources"],
    out = "lockbox.ko",
    kernel_build = ":virtual_device_x86_64",
```

```
    deps = [":common_headers"],
)
```

The `lockbox_sources` rule points to the two new source files we created earlier ❶, and the `x86_64/lockbox` rule depends on those files to create an x86_64 version of our driver ❷, which we'll write into a driver binary called *lockbox.ko*.

While the x86_64 version of the driver is the only one we're defining in this example, you can make additional `ddk_module` rules for `aarch64` and `aarch64_16k` to match the other drivers in this build file. Since we're using x86_64 Cuttlefish, I've omitted those rules here.

Finally, you'll need to extend one existing build rule to include this new driver along with all the other drivers. Find the existing `virtual_device_x86 _64_external_modules` rule and add the new `x86_64/lockbox` rule as a dependency, shown here in bold:

```
--snip--
kernel_module_group(
    name = "virtual_device_x86_64_external_modules",
    srcs = [
        ":x86_64/goldfish_drivers/goldfish_address_space",
        ":x86_64/goldfish_drivers/goldfish_pipe",
        ":x86_64/goldfish_drivers/goldfish_sync",
        ":x86_64/v4l2loopback",
        ":x86_64/lockbox",
        "//common-modules/virtio-media/driver:x86_64/virtio-media",
    ],
)
```

At this point, it's probably a good idea to try compiling the kernel with the new driver to ensure that the build rule is working as expected. Because the rule doesn't contain any logic yet, you won't see different behavior if you boot the new kernel.

Next, we'll implement the driver itself. We'll divide the code into two parts: the lockbox and the padlock.

Implementing the Lockbox

The code for the */dev/lockbox* device is relatively simple. We'll start by writing the driver's basic outline, then look at its individual parts. If you've never written a Linux kernel driver before, this may be the simplest possible introduction to the topic you'll find.

Let's now expand *lockbox.c*. It may be helpful to read the following code from the bottom up, since many of the later structures and functions depend on earlier ones:

lockbox.c

```
#include <linux/miscdevice.h>
#include <linux/module.h>

#include "lockbox.h"

MODULE_DESCRIPTION("lockbox example driver");
MODULE_AUTHOR("Your Name");
MODULE_LICENSE("GPL");

❶ static ssize_t lockbox_read(struct file *file, char __user *buf,
                size_t count, loff_t *offset)
{
  return 0;
}

❷ static const struct file_operations lockbox_fops = {
    .owner = THIS_MODULE,
    .open = nonseekable_open,
    .read = lockbox_read,
    .llseek = noop_llseek,
};

❸ static struct miscdevice lockbox_misc = {
    .minor  = MISC_DYNAMIC_MINOR,
    .name  = "lockbox",
    .fops  = &lockbox_fops,
```

```
};

static int __init lockbox_init_module(void)
{
  int err;

❹ err = misc_register(&lockbox_misc);
 if (err < 0)
   return err;

return 0;
}

static void lockbox_cleanup_module(void)
{
❺ misc_deregister(&lockbox_misc);
}

MODULE_ALIAS_MISCDEV(MISC_DYNAMIC_MINOR);

module_init(lockbox_init_module);
module_exit(lockbox_cleanup_module);
```

Let's look at all the pieces of this code and how they go together. The main structure representing the new `lockbox` device is the `lockbox_misc struct` ❸, which defines what the file will be named (the `name` field), assigns it a dynamic device type minor number (`MISC_DYNAMIC_MINOR`), and points to the `lockbox_fops struct` ❷.

The simplicity of this driver means it needs to implement only the `read` syscall, so `lockbox_fops` needs to point only to the `lockbox_read` function ❶.

The code finishes up by extending `lockbox_init_module` to register the device when the kernel boots ❹ and `lockbox_cleanup_module` to remove the device when the kernel shuts down ❺.

If you build and boot the kernel with this newest code, you should see the new device show up:

```
$ adb shell ls /dev | grep lockbox
lockbox
```

Before the driver can be useful for anything, however, we need to implement the `read` syscall. This syscall will read our secret message, which we store in a static variable if the virtual combination lock is opened. This syscall requires us to add a few pieces of state to the driver, in *lockbox.c*:

```
    #include <linux/miscdevice.h>
    #include <linux/module.h>

    #include "lockbox.h"

    MODULE_DESCRIPTION("lockbox example driver");
    MODULE_AUTHOR("Your Name");
    MODULE_LICENSE("GPL");

❶ char secret_message[] = "This is a secret!\n";
    static struct mutex lock;
    static bool locked = false;

    static ssize_t lockbox_read(struct file *file, char __user *buf, size_t
count,
                                loff_t *offset)
❷ {
      mutex_lock(&lock);
      if (locked) {
        mutex_unlock(&lock);
        return -EACCES;
      }

      size_t rem = strlen(secret_message) - *offset;

      if (rem <= 0) {
        mutex_unlock(&lock);
        return 0;
      }
```

```
    if (rem < count) {
      count = rem;
    }

❸ unsigned long not_copied = copy_to_user(buf, &secret_message[*offset],
                                          count);
    unsigned long read = count - not_copied;
    *offset = *offset + read;

    mutex_unlock(&lock);
    return read;
  }

  --snip--

  static int __init lockbox_init_module(void)
  {
    int err;

    mutex_init(&lock);

    err = misc_register(&lockbox_misc);
    if (err < 0)
      return err;

    return 0;
  }

  static void lockbox_cleanup_module(void)
  {
    misc_deregister(&lockbox_misc);

    mutex_destroy(&lock);
  }

  MODULE_ALIAS_MISCDEV(MISC_DYNAMIC_MINOR);
```

```
module_init(lockbox_init_module);
module_exit(lockbox_cleanup_module);
```

This code checks whether the combination is locked and returns an error if it is ❷. If it's not locked, the code copies the secret message ❶ to userspace ❸ and returns the number of bytes written.

While none of our code writes to the global state yet, you'll be doing so shortly. So, add a mutex and make sure to acquire it before using the state.

If you compile and boot the kernel with this latest version of the driver, you should be able to read the secret message from userspace:

```
$ adb shell cat /dev/lockbox
This is a secret!
```

Since defaulting to a secure state is good practice, go ahead and change the default state to be locked:

```
char secret_message[] = "This is a secret!\n";

static struct mutex lock;

static bool locked = true;
```

If you now build and boot the kernel, you should get an error when reading the file:

```
$ adb shell cat /dev/lockbox
cat: /dev/lockbox: Permission denied
```

This means that the driver has been successfully built and loaded in the kernel. Since we haven't yet implemented a way to unlock the lock by setting the `locked` variable, it can't be read. We'll make it possible to unlock the lock-box and read the secret message in the next section.

Implementing the Padlock

The remaining part of the driver needs to surface an additional device, */dev/padlock*, that allows users to turn the virtual dial in either direction and end on any number from 0 to 20. For convenience, the padlock should also support a `clear` operation, which is equivalent to randomly spinning the dial to delete any partially entered code.

Begin by setting a few constants for these commands in the header file:

lockbox.h

```
#ifndef _LOCKBOX_H
#define _LOCKBOX_H

#include <linux/ioctl.h>

#define CODE_LENGTH 4

#define PADLOCK_MAGIC 'p'

#define PADLOCK_TURN_CLOCKWISE 0
#define PADLOCK_TURN_COUNTERCLOCKWISE 1
#define PADLOCK_DIAL_MAX 20

#define PADLOCK_CLEAR _IOW(PADLOCK_MAGIC, 1, int)
#define PADLOCK_TURN _IOW(PADLOCK_MAGIC, 2, int)

struct padlock_turn {
  int direction;
  int stop_number;
};

#endif /* _LOCKBOX_H */
```

Instead of taking input from userspace via the `read` or `write` syscalls, the padlock will use `ioctl`. This syscall is very common in kernel drivers that can't be easily represented with just `read` and `write`, including lots of drivers on Android devices for things like cameras, NFC chips,

fingerprint readers, and machine learning accelerators. The `PADLOCK_MAGIC` definition and `_IOW()` macros define the codes that the `ioctl` commands will use.

When `PADLOCK_TURN ioctl` is used, it will get a pointer to a `padlock_turn struct`. The `direction` field will have one of the `PADLOCK_TURN_*` values as well as the number that the dial is stopped on.

You'll also need to add some additional state to represent whether the three parts of the combination have been entered correctly. Finally, two static variables should store the current numerical position of the dial and the direction in which it was turned most recently:

lockbox.c

```
char secret_message[] = "This is a secret!\n";

static struct mutex lock;

static bool locked = true;

static bool pin_one_open = false;
static bool pin_two_open = false;
static bool pin_three_open = false;

static int current_direction = PADLOCK_TURN_COUNTERCLOCKWISE;
static int current_number = 0;
```

As with the `lockbox` device, you'll need to define a `miscdevice struct` for the padlock and the associated `file_operations struct`. However, this one needs to implement only the `ioctl` syscall:

```
    static long padlock_ioctl(struct file *file, unsigned int cmd,
                              unsigned long param)
    {
      // TODO: implement this.
      return 0;
    }
```

```
❶ static const struct file_operations padlock_fops = {
      .owner = THIS_MODULE,
      .open = nonseekable_open,
      .unlocked_ioctl = padlock_ioctl,
      .compat_ioctl = padlock_ioctl,
      .llseek = noop_llseek,
    };

❷ static struct miscdevice padlock_misc = {
      .minor  = MISC_DYNAMIC_MINOR,
      .name  = "padlock",
      .fops  = &padlock_fops,
    };

    static int __init lockbox_init_module(void)
    {
      int err;

      mutex_init(&lock);

      err = misc_register(&lockbox_misc);
      if (err < 0)
        return err;

  ❸ err = misc_register(&padlock_misc);
      if (err < 0)
        return err;

      return 0;
    }

    static void lockbox_cleanup_module(void)
    {
      misc_deregister(&lockbox_misc);
  ❹ misc_deregister(&padlock_misc);
```

```
    mutex_destroy(&lock);
}
```

As with the lockbox, we define a `_fops` object, `padlock_fops` ❶; associate it with a `miscdevice`, `padlock_misc` ❷; register it as part of the `lockbox_init_module()` function ❸; and clean it up as part of `lockbox_cleanup_module()` ❹. Since the two devices are part of the same driver and share the same state, such as the `lock` mutex, this is done in the same file.

For historical reasons, there are handlers for both `unlocked_ioctl` and `compat_ioctl`, but both can point to the same function and have the same logic.

Add a short helper function to reset the lock's state. This is relatively simple, but it will make the implementation of the `ioctl` cleaner. Since we'll be using this in `padlock_ioctl`, place the helper function before it in the file:

```
static void clear_code(void)
{
  locked = true;

  pin_one_open = false;
  pin_two_open = false;
  pin_three_open = false;

  current_direction = PADLOCK_TURN_COUNTERCLOCKWISE;
  current_number = 0;
}

static long padlock_ioctl(struct file *file, unsigned int cmd,
                          unsigned long param)
{
  // TODO: implement this.
  return 0;
}
```

```
--snip--
```

Now let's take a look at implementing the `padlock_ioctl()` function. Similar to most `ioctl` handlers, it will mostly consist of a `switch` statement over the `cmd` parameter to express which command the driver receives. In our case, this represents the various types of button presses, which are enumerated in the header file:

```
--snip--

static long padlock_ioctl(struct file *file, unsigned int cmd,
                          unsigned long param)
{
  struct padlock_turn turn;

  mutex_lock(&lock); ❶

  switch (cmd) {
    case PADLOCK_CLEAR: ❷
      clear_code();
      return 0;
    case PADLOCK_TURN: ❸
      if (copy_from_user(&turn, (struct padlock_turn *)param, sizeof(turn))) { ❹
        mutex_unlock(&lock);
        return -EFAULT;
      }

      if (turn.direction != PADLOCK_TURN_COUNTERCLOCKWISE &&
          turn.direction != PADLOCK_TURN_CLOCKWISE) { ❺
        mutex_unlock(&lock);
        return -EINVAL;
      }

      if (turn.stop_number < 0 || turn.stop_number > PADLOCK_DIAL_MAX) {
        mutex_unlock(&lock);
        return -EINVAL;
      }
```

```
      if (turn.direction != current_direction)  { ❻
        // The padlock just changed direction.
        // Check if the previous number matches what we want.
        if (!pin_one_open) { ❼
          if (current_number == 4) {
            pin_one_open = true;
          } else {
            clear_code();
          }
        } else if (!pin_two_open) { ❼
          if (current_number == 8) {
            pin_two_open = true;
          } else {
            clear_code();
          }
        } else {
          if (current_number == 15) { ❼
            pin_three_open = true;
            locked = false;
          } else {
            clear_code();
          }
        }
      }

      current_direction = turn.direction; ❽
      current_number = turn.stop_number;

      mutex_unlock(&lock);
      return 0;
    }

    mutex_unlock(&lock);
    return 0;
}
```

```
--snip--
```

Since we'll be reading and modifying global state, the code first needs to acquire the mutex ❶ with `mutex_lock()`. We'll also need to ensure that it's released via `mutex_unlock()`.

Like many `ioctl` implementations, ours supports multiple commands and will handle each one with a `switch` statement. The implementation for `PADLOCK_CLEAR` ❷ is very simple and just calls our helper function, `clear_code()`.

The more interesting logic happens when the dial is turned, represented by the `PADLOCK_TURN` command ❸. In this case, we use the `param` value of the `ioctl` syscall. Since this value comes from userspace, we must call the `copy _from_user()` function ❹.

In this case, the driver will treat the `param` as a pointer to the `padlock_turn struct` we defined earlier. Because Linux defines the `ioctl` syscall to take two arguments, the pattern we're following is common in Linux kernel drivers. In many real-world cases, the `struct` definition can be quite complicated, often including several nested `structs`. We do some simple validation ❺ to make sure that the values in the `struct` are what we expect.

If the latest turn of the padlock is in a new direction ❻, we proceed to check which pin is the next to be unlocked ❼. If the last number before the turn is what we expect, we open the pin. Otherwise, we reset the lock with `clear_code()`.

Finally, we remember the new location of the padlock dial so that we can compare it on the next turn ❽.

If you now rebuild the kernel with the completed driver and boot it on Cuttlefish, you should see the */dev/padlock* device show up:

```
$ adb shell ls /dev | grep padlock
padlock
```

You'll also see that you're unable to read from the file, since it doesn't support reading at all:

```
$ adb shell cat /dev/padlock
cat: /dev/padlock: Invalid argument
```

We can't read from the */dev/padlock* device since it doesn't provide a *read* implementation in *padlock_fops*. The user can interact with the padlock only by using `ioctl` commands to turn the dial.

Writing the Description

Now that you've written the kernel driver, you can teach Syzkaller how to fuzz it by writing a description for it, then compiling Syzkaller with the new description.

We write Syzkaller descriptions in a domain-specific language called *Syzlang* that looks a bit like C. For Android and Linux, these descriptions live in text files (ending with a *.txt* extension) stored in the *sys/linux/* directory in the Syzkaller source code. By convention, most are named after the device node in */dev* that the driver exposes. For our example code, we'll use *dev_lockbox.txt*.

Describing the Lockbox

Let's begin with the `lockbox` portion of the driver. A lot is going on here, so we'll go through the code line by line and talk about each one.

First, we define a new `resource` type called `fd_lockbox`:

dev_lockbox.txt

```
resource fd_lockbox[fd]
```

The square brackets tell us which type this is a specialization of, an approach somewhat similar to `typedef` in C or inheritance in certain programming languages. The *sys/linux/sys.txt* description contains many of the default types and syscalls for Linux; there, you'll see that the `fd` type is actually a specialization of `int32`, as you might expect if you've done much C programming.

Now, let's look at what it takes to open the new file:

```
openat$lockbox(fd const[AT_FDCWD], file ptr[in, string["/dev/lockbox"]], flags
flags[lockbox_open_flags], mode const[0]) fd_lockbox
```

There's a lot to parse through here. We'll start at the left and move forward one chunk at a time.

The first element to note is `openat$lockbox`. The part before the dollar sign (`$`) defines which syscall this is, and the following segment is a Syzkaller-specific name. You can choose any name you want so long as it's unique among all the descriptions Syzkaller is compiled with. Keeping the name simple is generally a good idea, so that's what we've done here.

You may be wondering why we've identified the syscall as `openat` instead of `open`, which you may have used when writing C code. The reason is that userspace code doesn't often make syscalls directly, instead relying on wrappers provided by the platform's libc library. The Android version of libc, Bionic, always uses `openat`, so that's what we use here.

Next is a list of parameters to pass to the syscall function. Each parameter has a name followed by a type, which may have a list of options given in square brackets.

The first parameter, `fd`, is a constant value (`const`) with the value `AT_FDCWD`. This value tells the syscall to resolve paths relative to the current working directory. We need it because of a quirk of using the `openat` syscall instead of `open`; for more information, read the manpage for `openat`. Virtually all Syzkaller descriptions use this parameter because they use full absolute paths, as you'll see in a moment.

The next parameter, `file`, is a pointer to a string (which is a character array in C). In this case, the pointer type has two options, separated by commas. The first declares the direction in which the data flows, in or out of the function. Because the filename passed to the syscall is used as input and isn't modified by the kernel, the description specifies `in` as opposed to `out` (for output only) or `inout` (for both input and output). This annotation helps Syzkaller mutate inputs more intelligently. The second option tells Syzkaller the type of data the pointer should point to. In this case, it's a `string` with the name of the file surfaced by the driver, */dev/lockbox*.

The next parameter, called `flags`, is of type `flags` and represents options passed as flags to the syscall. Syzlang defines the `flags` type as a specialization of `int32`. The value of that field is a constant called `lockbox_open_flags`. Unlike C, Syzlang allows types to be used before they're defined. We'll define it later, but Syzkaller won't compile until we do.

The final parameter, called `mode`, is a constant with the value `open_mode`. Syzkaller defines this value elsewhere. Finally, the description specifies that this syscall returns the resource type we defined earlier, `fd_lockbox`.

Defining constants like `lockbox_open_flags` is straightforward in Syzlang:

```
lockbox_open_flags = O_RDONLY
```

This constant defines that the value for `lockbox_open_flags` is `O_RDONLY`, which is the constant for opening a file with only read permissions, as defined by the C standard library.

We've now walked through each piece of the `lockbox` definition. A few final notes; first, here is what the equivalent code, to make the same syscall, would look like in C:

```
int fd_lockbox = openat(AT_FDCWD, "/dev/lockbox", O_RDONLY);
```

Also note that the only other operation permitted on this driver is the `read` syscall. Here's what its description looks like:

```
read$lockbox(fd fd_lockbox, data buffer[out], len bytesize[data])
```

Because we've already gone through each part of the previous call, we'll point out only a few differences with this one. This syscall doesn't return any resource; it just populates the buffer given as the second parameter, which is annotated with the `out` option. The final parameter, `len`, uses the Syzlang type `bytesize`, and its option, `data`, tells Syzkaller that it needs to match the size of the buffer named `data`.

Our description depends on the `fd` parameter being of type `fd_lockbox`, so Syzkaller will understand that it can be called only after the

`openat$lockbox` syscall is used, which means the file descriptor this syscall gets will always be one that comes from opening */dev/lockbox*.

Describing the Padlock

Now let's describe the */dev/padlock* driver. Syzkaller doesn't place any restrictions on which file these descriptions are in. You could certainly create a new file, perhaps called *dev_padlock.txt*, to contain them. In fact, you could place each individual line description in its own file. It's best practice to keep conceptually related code in the same file, however, so that's what we'll do here.

The first few parts of the `padlock` descriptions are very similar to those from the `lockbox` driver:

```
resource fd_padlock[fd]
padlock_open_flags = O_RDWR
openat$padlock(fd const[AT_FDCWD], file ptr[in, string["/dev/padlock"]], flags
flags[padlock_open_flags], mode const[open_mode]) fd_padlock
```

The main difference here is that we must use this driver with both read and write permissions, since we'll be writing data to it via the `ioctl` syscall.

We didn't implement `read`, `write`, or `seek` syscalls for this driver earlier, so we don't need to describe them here. Instead, all logic for this driver exists in calls to `ioctl`. For the simple case of clearing any entered code, which doesn't require passing or using a parameter, the description is simple:

```
ioctl$padlock_clear(fd fd_padlock, cmd const[PADLOCK_CLEAR], arg const[0])
```

The second parameter, `cmd`, specifies the same constant value used in the header file earlier. The second, unused parameter can be set to a constant `0` value. Because the driver doesn't actually use this value, it could be set to anything without significantly changing the fuzzing behavior, but it's best to reflect the fact that the driver ignores the parameter and that the fuzzing engine doesn't waste time trying many possible values.

The last description will handle the logic of turning the dial on the padlock. First, we need to define the contents of the `struct` so that Syzkaller knows how to populate it:

```
padlock_turn {
  direction int32
  stop_number int32
}
```

This definition matches the one of the `struct` from the header file, with a few small differences. One is that Syzlang definitions provide the name first and type second (similar to Go) instead of in the opposite order (similar to C). The second difference is that Syzlang is more explicit about the size of numeric types and has `int16`, `int32`, and `int64`.

Now you can write the description for the `PADLOCK_TURN ioctl` syscall:

```
ioctl$padlock_turn(fd fd_padlock, cmd const[PADLOCK_TURN], arg ptr[in,
padlock_turn])
```

The driver expects the argument to the `ioctl` call to be a pointer to the `struct`. The pointer is used as input only, so we can mark it as `in`. Alternatively, we could label it `out`, if it was used only for output from the syscall, or `inout` if we used it for both input and output.

Syzkaller can use this information to fuzz more intelligently in drivers that might take data from several `ioctl` calls, such as one that initializes a field and another that modifies it. In our case, we have only the single syscall, but it's good to get in the habit of giving Syzkaller the most accurate type information we can.

One final improvement we can make is to specify the bounds of each of the fields in `padlock_turn`. Like many other features of Syzlang, these aren't required, but they can help performance. This lets Syzkaller focus on only values in the right range, instead of trying lots of very large or negative numbers. Modify `padlock_turn` so it looks as follows:

```
padlock_turn {
  direction int32[0:1]
```

```
  stop_number int32[0:PADLOCK_DIAL_MAX]
}
```

Deciding how much or how little invalid behavior to allow (such as out-of-range values in the example driver) can be difficult. Constraining fuzzing to only valid inputs can improve performance but may miss some bugs. As a compromise, you might consider adding another variant of this description without the bounds. You could give that description a different name, such as `ioctl$padlock_turn_unlimited`, and then toggle the two descriptions on or off in the configuration file, as we'll show shortly, or let them both run.

Before moving on, we need to add one final line to the top of the definition file. The `ioctl` descriptions include references to constants like `PADLOCK_CLEAR`, `PADLOCK_TURN`, and `PADLOCK_DIAL_MAX`. To show where these are defined, we need to add an `include` statement at the top of the file:

```
include <lockbox/lockbox.h>
```

This statement matches the path to the header file we wrote for the driver earlier. Note that, unlike C code, the `include` keyword doesn't begin with `#`, which Syzlang uses for comments.

Extracting Constants

For Syzkaller to use the syscalls described in its library of descriptions, it needs to evaluate the constants they contain. For example, the padlock driver's `ioctl` descriptions depend on constants like `PADLOCK_TURN` and `PADLOCK_CLEAR`. Recall that we define these in the *lockbox.h* header file:

lockbox.h

```
#define PADLOCK_MAGIC 'p'

#define PADLOCK_TURN_CLOCKWISE 0
#define PADLOCK_TURN_COUNTERCLOCKWISE 1
#define PADLOCK_DIAL_MAX 20
```

```
#define PADLOCK_CLEAR _IOW(PADLOCK_MAGIC, 1, int)
#define PADLOCK_TURN _IOW(PADLOCK_MAGIC, 2, int)
```

These constants are held in the Syzkaller source tree alongside the description files. If you examine the contents of Syzkaller's *sys/linux/* directory, you'll see that each description file (those ending with *.txt*) has a matching constants file that ends in *.txt.const*.

While the constant values for our driver are quite simple, real kernel drivers with more complicated functionality can be more complex. In addition, the values of some constants may be architecture dependent or have changes based on compile-time options.

To handle all this complexity, Syzkaller has a tool called `syz-extract` to collect this information from kernel source code and generate accurate constant files. When you built Syzkaller earlier, it also compiled this tool, which should now be available in Syzkaller's *bin/* directory.

We must specify a few parameters via command line flags for the tool to run correctly:

```
$ bin/syz-extract -os linux -arch architecture \
-sourcedir kernel_source_root \
-includedirs comma-separated_path_to_header_files \
description_file
```

Table 10-1 describes each of these flags.

Table 10-1: Flags for `syz-extract`

Flag	Value	Example
`-os`	The operating system under test	`linux`
`-arch`	The processor architecture	`amd64` (x86_64), `arm64` (AArch 64)
`-sourcedir`	The path to the root of the kernel checkout	`$HOME/kernel-src/common`

Flag	Value	Example
`-includedirs`	A comma-separated list of paths to header files	`$HOME/includes,$HOME/more-includes`

Because of the way the Android kernel separates modules into different paths, getting each of these parameters set correctly can be difficult. The `-sourcedir` flag should point to the *common/* directory in your kernel checkout, since this directory contains the generic Linux kernel code.

One nice feature of the Android kernel's build system is that it collects all the header files, including some automatically generated ones, into a single artifact, which it helpfully places in the *out/* directory, or wherever you pointed the `--destdir` flag to when you built the kernel. These are all contained in the *kernel-headers.tar.gz* file. You'll need to extract them all so that you can point `syz-extract` at them:

```
$ tar -xf out/kernel-headers.tar.gz
```

If you run that command in the root of your kernel checkout, you should now have a new *kernel-headers/* directory containing a few subdirectories and many header files:

```
$ ls kernel-headers
arch  crypto  include  init  lib  rust  scripts  security  tools  usr
```

Now you should be able to call the `syz-extract` tool. Because the tool takes so many parameters, and specifying all the necessary paths can be repetitive, we suggest writing a short shell script, which we'll call *run-extract.sh*:

run-extract.sh

```
#!/bin/sh

kernel_root=path to your kernel checkout
header_dir="$kernel_root/kernel-headers"
```

```
    include_dirs="$header_dir/include,$header_dir/usr/include"
❶ include_dirs+=",$header_dir/arch/x86/include/generated"
❷ include_dirs+=",$kernel_root/common-modules/virtual-device"

❸ bin/syz-extract -os linux -arch amd64 \
      -sourcedir $kernel_root/common \
      -includedirs $include_dirs \
      dev_lockbox.txt
```

This script points the driver to several subdirectories of unpacked header files. These exact values should work for the example driver, assuming you're building for x86_64 Cuttlefish. If building for ARM, change the value of the `-arch` flag ❸ and one of the paths ❶. Additionally, you need to specify the path to the *common-modules* directory where the *lockbox.h* file is contained ❷.

When writing descriptions for other kernel drivers, it may take you some trial and error to get all the paths correct. Unfortunately, there's no easy, one-size-fits-all approach here, as the combinations of kernel drivers vary across manufacturers and devices.

Now you should be able to run the script. If all goes well, you'll receive a short message and end without an error:

```
$ ./run-extract.sh
generating linux/amd64...
```

The constant file, *dev_lockbox.txt.const*, should now be populated:

dev_lockbox.txt.const

```
# Code generated by syz-sysgen. DO NOT EDIT.
arches = amd64
AT_FDCWD = amd64:18446744073709551516
O_RDONLY = amd64:0
O_RDWR = amd64:2
PADLOCK_CLEAR = amd64:1074032641
PADLOCK_DIAL_MAX = amd64:20
```

```
PADLOCK_TURN = amd64:1074032642
__NR_ioctl = amd64:16
__NR_openat = amd64:257
__NR_read = amd64:0
```

These constants are now defined with the same values that the C compiler will generate when building the kernel. Since these numbers may be different on different processor architectures, Syzkaller allows them to be defined per architecture, which is why each value is prefixed with `amd64`.

Now that you've extracted values for the constants that your description uses, you have everything you need to begin fuzzing it.

Fuzzing Your New Description

You're now ready to fuzz the new kernel driver with your description. First, you'll need to rebuild Syzkaller with the new description and related constants. Syzkaller will automatically include anything under *sys/linux/*, so you can reuse the build command from Chapter 9 without making any other code changes:

```
$ tools/syz-env make
```

Now start Syzkaller with the same command and configuration as in the preceding chapter:

```
$ ./bin/syz-manager -config=cuttlefish-basic.json
```

It may take a few minutes for Syzkaller to begin fuzzing, since it will attempt to reboot the virtual device so that it's in a fresh state. You can view its progress by visiting the VM's page in the Syzkaller web interface at *http://localhost:8080/vms*. Figure 10-1 shows this interface.



Figure 10-1: Syzkaller's VM page showing a VM booting

NOTE

If you see that Syzkaller starts and you're able to view the web UI, but fuzzing never begins, you may have forgotten the `--kasan_generic` *flag when building the kernel with the new driver. You can verify this by stopping Syzkaller, then checking whether the* /sys/kernel/debug/kcov/ *directory exists.*

You can also view the Syscalls page at *http://localhost:8080/syscalls*, which shows a large table containing data about all the syscalls Syzkaller has descriptions for, including the number of times each one has been tried. There, you should see your new descriptions, as shown in Figure 10-2.

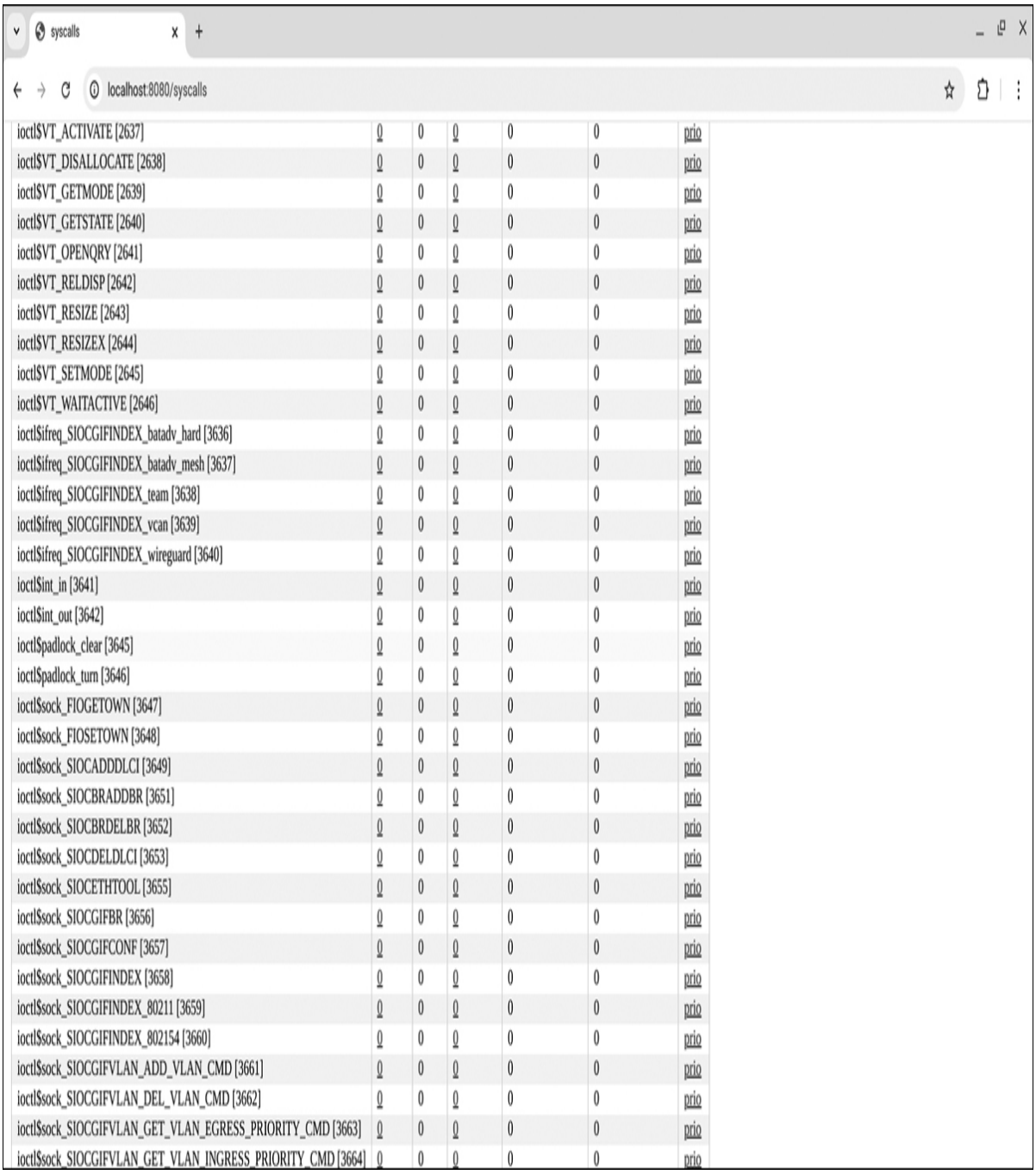


Figure 10-2: Syzkaller's Syscalls page with all descriptions enabled

Focusing on New Descriptions

You'll likely notice that the new syscalls have been called very little, if at all. Because there are so many syscalls, it will take Syzkaller quite some time to start fuzzing the new driver with your descriptions.

In many real-world cases, the kernel can have bugs that are caused by complicated interactions among various pieces of code, so fuzzing many descriptions may be the only way to find these bugs. However, the new driver we wrote shouldn't have any of these interactions, so we can safely change our Syzkaller configuration to focus exclusively on the `lockbox` and `padlock` drivers.

To do so, modify or copy the *cuttlefish-basic.json* configuration from earlier and add a new field, `enable_syscalls`:

cuttlefish-lockbox.json

```
enable_syscalls: [
  "openat$lockbox",
  "read$lockbox",
  "openat$padlock",
  "ioctl$padlock_clear",
  "ioctl$padlock_turn"
],
```

Call the modified version *cuttlefish-lockbox.json*. With this option set, Syzkaller will use only the specified descriptions. If you restart Syzkaller with this configuration file and visit the Syscalls page again, you should see the change take effect, as shown in Figure 10-3.

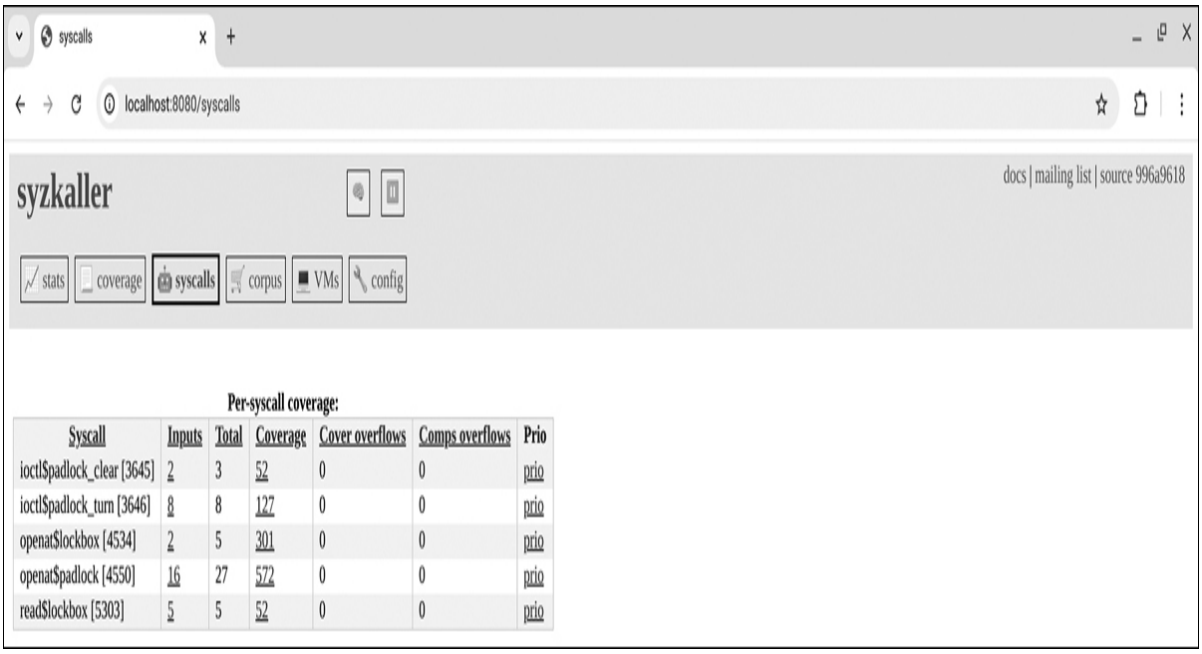


Figure 10-3: Syzkaller's Syscalls page with only the new descriptions enabled

Not only is the interface cleaner, but you can see that Syzkaller is trying different combinations of the new syscalls.

While we won't use it here, it's worth pointing out that a similar field, `disable_syscalls`, does the opposite of `enable_syscalls`; any syscalls listed there won't be used for fuzzing. You can use at most one of `enable_syscalls` and `disable_syscalls`. Both fields also support the use of a trailing asterisk (`*`) as a wildcard, allowing you to simplify your configuration if you'd like. Here's what that might look like:

```
{
  "name": "cuttlefish-lockbox",
  "target": "linux/amd64",

  "http": ":8080",
  "workdir": "workdir",
  "syzkaller": ".",

  "enable_syscalls": [
    "openat$lockbox",
    "read$lockbox",
    "openat$padlock",
```

```
    "ioctl$padlock_*",
  ],

  "type": "adb",
  "vm": {
    "devices": [
      {
        "serial": "0.0.0.0:6520",
        "console_cmd": ["tail", "-f", "path/to/kernel.log"]
      }
    ]
  }
}
```

These approaches are equivalent, so which one you choose is a matter of personal preference. Since we are adding only a few syscalls here, it is easiest to use the `enable_syscalls` option. If you are fuzzing a larger set of drivers, it may be easier to use the `disable_syscalls` option instead.

Introducing a Bug

Syzkaller has everything it needs to fuzz the new driver, but even if it's able to find the combination needed to open the lockbox, it won't detect a bug, as there currently aren't any bugs in the driver that would cause kernel crashes or trigger KASAN. Let's introduce a simple call to `panic()` so that the kernel will crash after reading the secret message if the combination is entered correctly:

lockbox.c

```
static ssize_t lockbox_read(struct file *file, char __user *buf, size_t count,
                            loff_t *offset)
{
  --snip--
  unsigned long not_copied = copy_to_user(buf, &secret_message[*offset],
                                          count);
  unsigned long read = count - not_copied;
```

```
  *offset = *offset + read;

  mutex_unlock(&lock);
  panic("safe cracked!\n");
  return read;
}
```

In real-world kernel drivers, a bug would often be something like an OOB array access or dereferencing an invalid pointer. When a kernel is configured to use KASAN, these will cause the kernel to crash. For simplicity here, we just call the `panic()` function directly, which has the same result.

If you are curious and would like to view progress in a more real-time way, you can add logging statements to the branches of `padlock_ioctl`:

```
static long padlock_ioctl(struct file *file, unsigned int cmd,
                          unsigned long param)
{
  struct padlock_turn turn;
  mutex_lock(&lock);
  --snip--
      if (turn.direction != current_direction) {
        // The padlock just changed direction.
        // Check if the previous number matches what we want.
        if (!pin_one_open) {
          if (current_number == 4) {
            pr_emerg("pin one open!");
            pin_one_open = true;
          } else {
            clear_code();
          }
        } else if (!pin_two_open) {
          if (current_number == 8) {
            pr_emerg("pin two open!");
            pin_two_open = true;
          } else {
            clear_code();
```

```
          }
        } else {
          if (current_number == 15) {
            pin_three_open = true;
            pr_emerg("pin three open!");
            locked = false;
          } else {
            clear_code();
          }
        }
      }
  --snip--
  mutex_unlock(&lock);
  return 0;
}
```

The Linux kernel has built-in logging macros called `pr_info()`, `pr_err()`, `pr_warn()`, and `pr_emerg()`. Depending on the kernel configuration, some of these may be ignored. The highest-priority macro is `pr_emerg()`. In a real driver, you would probably want to log these informational details at a lower priority, like `pr_info()`, or not at all. Using `pr_emerg()` ensures that these log statements will show up in Android's default configuration.

You can then listen to the logs by watching the same *kernel.log* file, similar to the way Syzkaller does. The new log lines should show up, and you should see that Syzkaller is making progress on cracking open the lock. As each pin of the combination's three numbers is entered, you will see the relevant message in the logs:

```
$ tail -F cuttlefish/instances/cvd-1/kernel.log
--snip--
[ 792.840004] pin one open!
--snip--
```

Eventually, you should see Syzkaller complete all three pins of the lock, at which point the device will crash. This may take quite a while, since Syzkaller will likely fail and get partial or incorrect combinations

many times. You'll see the crash show up in the Syzkaller web interface (Figure 10-4).

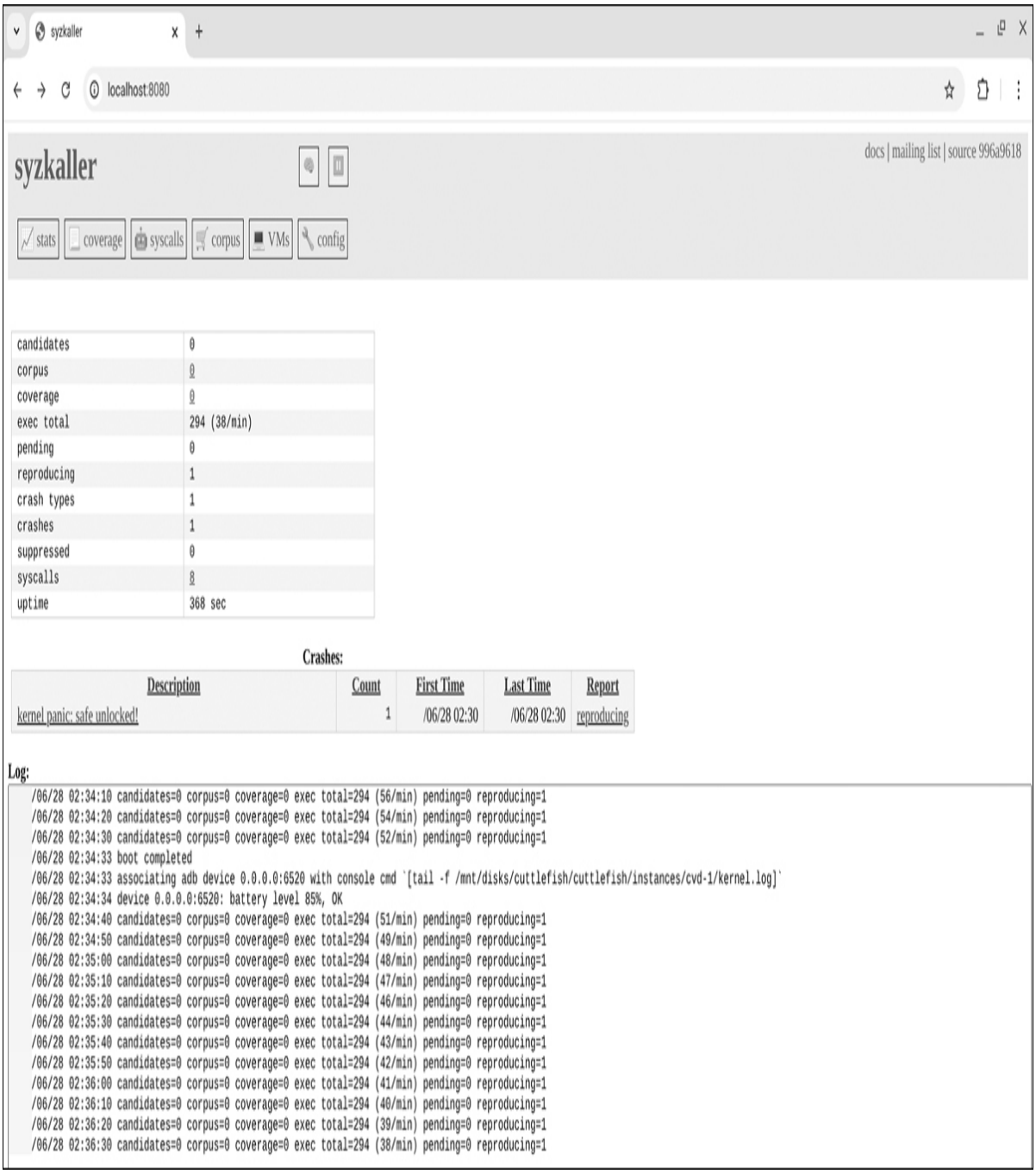


Figure 10-4: The Syzkaller web interface showing that it has found a crash

If you click the title of the crash, `kernel panic: safe cracked`, you should be able to see more information about the crash, including a logfile.

That logfile, which will contain all the inputs Syzkaller tried, can be quite long.

In some cases, Syzkaller will report the crash as `lost connection to test machine`, since the crash causes the device to go offline. The exact name of the bug may vary, but the output should otherwise be identical to the following:

```
last executing test programs:

308.633923ms ago: executing program 0 (id=255):
ioctl$padlock_turn(0xffffffffffffffff, 0x40047002, &(0x7f0000000000))
r0 = openat$padlock(0xffffffffffffff9c, &(0x7f0000000040), 0x0, 0x0)
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000080))
ioctl$padlock_turn(r0, 0x40047002, &(0x7f00000000c0)={0x1, 0xf})
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000100))
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000140))
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000180))
ioctl$padlock_clear(r0, 0x40047001, 0x0)
r1 = openat$padlock(0xffffffffffffff9c, &(0x7f00000001c0), 0x2, 0x0)
ioctl$padlock_clear(r1, 0x40047001, 0x0)
ioctl$padlock_clear(r1, 0x40047001, 0x0)
r2 = openat$padlock(0xffffffffffffff9c, &(0x7f0000000200), 0x2, 0x0)
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000240))
ioctl$padlock_turn(r2, 0x40047002, &(0x7f0000000280))
ioctl$padlock_turn(r1, 0x40047002, &(0x7f00000002c0))
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000300))
ioctl$padlock_turn(r2, 0x40047002, &(0x7f0000000340)={0x0, 0x8})
ioctl$padlock_turn(r1, 0x40047002, &(0x7f0000000380)={0x1, 0xd})
ioctl$padlock_clear(r1, 0x40047001, 0x0)
ioctl$padlock_turn(r0, 0x40047002, &(0x7f00000003c0))
ioctl$padlock_clear(r1, 0x40047001, 0x0)
ioctl$padlock_turn(r1, 0x40047002, &(0x7f0000000400)={0x0, 0x8})
ioctl$padlock_turn(r0, 0x40047002, &(0x7f0000000440))
ioctl$padlock_turn(r1, 0x40047002, &(0x7f0000000480))
ioctl$padlock_turn(r1, 0x40047002, &(0x7f00000004c0)={0x1, 0x5})
r3 = openat$padlock(0xffffffffffffff9c, &(0x7f0000000500), 0x0, 0x0)
r4 = openat$padlock(0xffffffffffffff9c, &(0x7f0000000540), 0x2, 0x0)
ioctl$padlock_turn(r4, 0x40047002, &(0x7f0000000580))
```

```
ioctl$padlock_turn(r3, 0x40047002, &(0x7f00000005c0))
ioctl$padlock_turn(r2, 0x40047002, &(0x7f0000000600))
--snip--
```

Syzkaller may cycle through many of these test programs before it finds the one that finally causes the crash. In the output, you can see how these programs relate to the various descriptions you wrote earlier.

You may notice, for example, that it made some decisions that seem illogical compared to the simple solution you may have expected. For example, in the snippet shown here, you'll see several file descriptors (`r0` through `r4`) for the padlock used. You'll also see many extra syscalls for extra turns of the lock. By reading further through the logs, you may be able to find the exact sequence of system calls that caused the crash.

After the recent program inputs, the logfile will also include the kernel logs leading up to the crash, identical to what you saw when using `tail` earlier. These will also contain a stack trace and information about the state of CPU registers:

```
--snip--
[  890.044884] pin one open!
[  890.048219] Bluetooth: hci1: command tx timeout
[  890.050170] pin two open!
[  890.055021] pin two open!
[  890.055155] pin three open!
[  890.056301] Kernel panic - not syncing: safe cracked!
...
[  890.057166] Call Trace:
[  890.057240]  <TASK>
[  890.057316]  dump_stack_lvl+0xad/0x100
[  890.057442]  dump_stack+0x19/0x20
[  890.057552]  panic+0x1d8/0x480
[  890.057705]  ? __mutex_unlock_slowpath+0xe3/0x340
[  890.057854]  lockbox_read+0x14f/0x170 [lockbox
7343b327b13af3afac9e6f113cf841fbc7c09dee]
[  890.058049]  ? __cfi_lockbox_read+0x10/0x10 [lockbox
7343b327b13af3afac9e6f113cf841fbc7c09dee]
[  890.058312]  vfs_read+0x155/0x5f0
```

```
[  890.058459]  ? __rcu_read_unlock+0x4d/0x80
[  890.058569]  ? __fget_files+0x25b/0x2a0
[  890.058692]  ? __fget_files+0x2b/0x2a0
[  890.058804]  ksys_read+0xbd/0x160
[  890.058916]  __x64_sys_read+0x47/0x60
[  890.059022]  x64_sys_call+0x26f4/0x2f50
[  890.059130]  do_syscall_64+0x75/0x110
[  890.059239]  ? clear_bhb_loop+0x35/0x90
[  890.059342]  ? clear_bhb_loop+0x35/0x90
[  890.059448]  entry_SYSCALL_64_after_hwframe+0x76/0x7e
[  890.059581] RIP: 0033:0x72791583d169
[  890.059692] Code: ...
[  890.060115] RSP: 002b:00007279156a9038 EFLAGS: 00000246 ORIG_RAX:
0000000000000000
[  890.060305] RAX: ffffffffffffffda RBX: 0000727915a55fa0 RCX: 000072791583d169
[  890.060491] RDX: 00000000000000f3 RSI: 0000200000000040 RDI: 0000000000000004
[  890.060680] RBP: 00007279158be2a0 R08: 0000000000000000 R09: 0000000000000000
[  890.060865] R10: 0000000000000000 R11: 0000000000000246 R12: 0000000000000000
[  890.061049] R13: 0000000000000000 R14: 0000727915a55fa0 R15: 00007ffc61dca188
```

These logs are also stored in Syzkaller's working directory, under *workdir/crashes/*. That directory will have subdirectories for each crash, identified by a unique hash. If there is more than one crash, you can identify the one you want by checking the log page's URL in the web interface. The file with these contents will start with the prefix *log*, and will be *log0* if this is the first time the crash has been found.

Running a Reproducer

Once Syzkaller has found a crash, it will begin trying to produce a minimal working example. You'll see the crash in the web interface updated with the status `reproducing` to reflect this (Figure 10-5).

Description	Count	First Time	Last Time	Report
kernel panic: safe unlocked	1	/06/28 14:31	/06/28 14:31	reproducing

Figure 10-5: The Syzkaller web UI while attempting to reproduce a crash

The Syzkaller logs will also show that Syzkaller is working on the reproducer by printing `reproducing=1` at the end of each log line.

Syzkaller starts the reproduction process by rerunning the entire series of syscalls from the list of programs in the log. If the crash happens again reliably, Syzkaller will then try running different subsets of the program to narrow the crash to a smaller set of syscalls. This process works like a binary search: Syzkaller repeatedly bisects the list of programs to find the start and end points of the sublist containing the issue. Syzkaller will also try removing some syscalls in the middle of the sequence to test whether they're needed.

This process can take a long time, often even longer than it takes to find bugs in the first place, especially when running on a small, narrowly focused set of descriptions as we have here. During this time, the virtual device will reboot many times, since Syzkaller attempts reproduction on a freshly booted device each time.

If Syzkaller finds a reproducer, it will update the status of the crash in the interface to say `has repro` (Figure 10-6).



Description	Count	First Time	Last Time	Report
kernel panic: safe unlocked	1	/06/28 14:31	/06/28 14:31	has repro

Figure 10-6: The Syzkaller web UI after finding a reproducer

If you follow the `has repro` link, you'll see that it begins with a snippet of the kernel logs, as seen earlier. The end of the file contains the minimized reproducer in Syzlang, as well as information about Syzkaller's settings when looking for the reproducer:

```
Syzkaller reproducer:
# {Threaded:true Repeat:true RepeatTimes:0 Procs:6 Slowdown:1 ...}
r0 = openat$lockbox(0xffffffffffffff9c, &(0x7f0000000000), 0x0, 0x0)
read$lockbox(r0, &(0x7f0000000040)=""/129, 0x81)
read$lockbox(r0, &(0x7f0000000100)=""/4096, 0x1000)
r1 = openat$padlock(0xffffffffffffff9c, &(0x7f0000001100), 0x2, 0x0)
ioctl$padlock_turn(r1, 0x40047002, &(0x7f0000001140))
read$lockbox(r0, &(0x7f0000001180)=""/238, 0xee)
```

```
r2 = openat$padlock(0xffffffffffffff9c, &(0x7f0000001280), 0x0, 0x0)
ioctl$padlock_turn(r1, 0x40047002, &(0x7f0000001300)={0x0, 0x14})
r3 = openat$padlock(0xffffffffffffff9c, 0x0, 0x0, 0x0)
ioctl$padlock_clear(r3, 0x40047001, 0x0)
r4 = openat$padlock(0xffffffffffffff9c, &(0x7f0000001380), 0x2, 0x0)
ioctl$padlock_clear(r4, 0x40047001, 0x0)
r5 = openat$padlock(0xffffffffffffff9c, 0x0, 0x0, 0x0)
ioctl$padlock_turn(r4, 0x40047002, &(0x7f0000001400))
openat$lockbox(0xffffffffffffff9c, &(0x7f0000001440), 0x0, 0x0)
r6 = openat$padlock(0xffffffffffffff9c, &(0x7f0000001480), 0x2, 0x0)
ioctl$padlock_turn(r5, 0x40047002, 0x0)
ioctl$padlock_turn(r1, 0x40047002, &(0x7f0000001540))
ioctl$padlock_turn(r6, 0x40047002, &(0x7f0000001580)={0x0, 0x12})
ioctl$padlock_turn(r3, 0x40047002, 0x0)
ioctl$padlock_clear(r1, 0x40047001, 0x0)
read$lockbox(0xffffffffffffffff, &(0x7f0000001640)=""/178, 0xb2)
ioctl$padlock_clear(r2, 0x40047001, 0x0)
ioctl$padlock_turn(r4, 0x40047002, &(0x7f0000001700))
read$lockbox(0xffffffffffffffff, 0x0, 0x0)
openat$lockbox(0xffffffffffffff9c, 0x0, 0x0, 0x0)
```

As with the crash logs from earlier, you can find this file in Syzkaller's working directory, *workdir/crashes/*, in the subdirectory identified with the same hash as the previous files. The file should be named *repro.prog*.

Because of the inherent randomness of fuzzing, as well as kernel scheduling, you may find a different reproducer than the one shown here. Some parts of the reproducer also might not fit what you, as a human, might have expected. In our example, for instance, you can see more calls to `openat` than are strictly required, as well as extra turns of the padlock.

Our example code is relatively simple; finding reproducers can take a long time or be error prone for the real-life fuzzing of more complex drivers, or when fuzzing multiple drivers at the same time. Syzkaller can sometimes find the same bug many times without being able to find a reproducer for it, particularly if the crash depends on specific timing, race conditions, or a complex shared state.

To turn your Syzlang reproducer into compilable C code, use a tool that ships with Syzkaller called `syz-prog2c`. You'll need to pass it the path to the *repro.prog* file:

```
$ ./bin/syz-prog2c -prog workdir/crashes/hash/repro.prog > lockbox_repro.c
```

This command will produce the equivalent C source code in a file called *lockbox_repro.c*. If you're compiling the reproducer for a physical device instead of x86_64 Cuttlefish, you'll need to add the `-arch arm64` flag.

To compile the source code for Android, you'll need to create an Android build rule and copy it into the Android tree. The build rule can be simple. For example, create the following *Android.bp* file in your Android checkout and copy the *lockbox_repro.c* file into the same directory:

Android.bp

```
cc_binary {
    name: "lockbox_repro",
    srcs: [
        "lockbox_repro.c",
    ],
}
```

You can then build the code just as you've built Android fuzz targets in earlier chapters, using these commands:

```
$ source build/envsetup.sh
$ lunch aosp_cf_x86_64_phone-trunk_staging-eng
$ m lockbox_repro
```

You'll find the resulting executable in *out/target/product/vsoc_x86_64/system/bin/lockbox_repro*.

Before you run it on your device, make sure to stop Syzkaller by pressing CTRL-C in the terminal where it's running. You should also restart your Cuttlefish instance to make sure there is no competition. Once that's done, push the executable onto the virtual device and run it:

```
$ adb mkdir -p /data/syzkaller
$ adb push out/target/product/vsoc_x86_64/system/bin/lockbox_repro
/data/syzkaller
$ adb shell /data/syzkaller/lockbox_repro
```

You should see the device crash and go offline before rebooting. If so, you know your reproducer works!

Wrapping Up

You've now completed the process of setting up Syzkaller and configuring it to fuzz a virtual Android device. We've introduced a simple kernel driver with a known bug, written a syscall description so that Syzkaller can fuzz it, and generated a C program that reliably reproduces the bug.

OceanofPDF.com

11
HOST ENVIRONMENTS



This chapter digs deeper into a fuzzing direction we touched on in Chapter 2 to improve your fuzzing performance: fuzzing a device-specific implementation of Android in a generic Linux environment. Although coined by the Android Security Team specifically for Android fuzzing, the technique of host-based fuzzing can apply to many other nonstandard fuzzing targets as well.

By leveraging standard environments, host-based fuzzing is able to maximize fuzzing performance and scalability and reduce the inherent friction and resource constraints of running fuzzers directly on target hardware. We'll explore why Android fuzzing practitioners have largely shifted to host-based fuzzing, analyze the associated trade-offs, and consider the necessary steps to enable it in more detail.

What Counts as a Host?

Because the majority of Android software runs on Android-based devices, the Android build system is designed to generate code for these devices by default. However, there are exceptions: The Android build

process also generates tools used later as part of the build system itself, and these tools must run on the machine performing the build. To distinguish this environment from the target device, Android refers to the Linux build machine as the *host environment*.

NOTE

For the purposes of this chapter, we'll assume that the host consists of a standard Linux environment running on AMD64 hardware unless otherwise specified.

Although developers initially considered only build tools to be host based, they soon realized that most of the code running on Android devices could also be built to run in the host environment, and that doing so could significantly simplify debugging and testing.

For example, to test a code change, they didn't necessarily have to deploy the updated binaries to an Android device, which sometimes requires rebuilding and reflashing the entire device partition. Also, they could directly invoke tools like `gdb` for debugging, without the need to set up a remote debugging system and attach to a running session on the device. They could also use `grep` and any other standard Linux tools to check the relevant logs.

Consequently, Android developers generalized the build rules for the host environment, making this environment compatible with an increasing number of Android components, including fuzzers.

Why Fuzz Android on the Host?

Fuzzing Android on the host provides many of the same benefits as on-host testing does for developers. Practitioners gain immediate access to all standard developer and debugging tools, including these benefits:

- Code coverage instrumentation is natively integrated into the compiler toolchain and readily available for use.

- If you find a crash, you can simply launch `gdb` (or a similar standard debugger) to begin investigating it immediately.
- You can easily attach a console to monitor the current fuzzing session's progress and output.
- Retrieving the source code is as simple as running `git clone`.
- If something goes wrong, you can efficiently use standard command line tools like `grep` to analyze fuzzing logs.

Attempting to perform these same operations entirely from an Android device involves considerable friction, requiring you to find and manage your USB cable, establish an ADB session, and ensure that all necessary debugging and analysis tools are available and functional through that limited session. Host-based fuzzing eliminates these barriers.

Fuzzing on the host also improves efficiency. Even with coverage-guided fuzzing, many decisions made by the fuzzer remain random. Consequently, the more iterations a fuzz target runs, the more problem space it can explore, increasing its likelihood of finding a bug. In this regard, we want the fuzz target to achieve the highest possible execution performance.

Today, mobile devices are equipped with powerful hardware, including fast CPUs, multiple cores, and ample RAM. However, they're fundamentally designed for mobility; for example, CPUs on mobile devices will throttle their speed because of thermal concerns or battery constraints, inherently limiting how fast a fuzz target can run. Furthermore, numerous background services run on the Android device, competing with the fuzzer for system resources.

By contrast, a standard workstation offers significantly more resources and operates with fewer constraints, allowing for the concurrent execution of multiple fuzzing instances (at the cost of higher resource consumption and power draw). By porting a fuzz target into a host-based fuzzer, you can dramatically improve the fuzzer's performance. It's not uncommon to see a 50 percent single-core

performance boost (in terms of executions per second) when porting a fuzzer from Android devices to the host machines.

Running fuzzers on specialized hardware, such as a dedicated device farm, also introduces operational overhead, including mechanical and electrical reliability of USB ports, heat dissipation issues for maintaining stable operating temperatures, battery degradation and replacement cycles, flash storage wear-out, devices becoming unresponsive (for example, stuck at a bootloop) and requiring manual intervention (power cycling, reflashing), and the continuous need to replace devices with newer generations to ensure contemporary testing coverage.

All these issues severely impact the availability of device resources. When a new build or update is pushed to the device lab, resource availability can fluctuate greatly, demanding constant operational support to resolve stability issues.

This is where host-based fuzzing truly excels. The fuzzers require nothing more than a standard POSIX environment. They can be executed on powerful physical workstations or, more commonly, within Linux VMs. Crucially, virtualization makes it trivial to spin up a powerful VM or an entire cluster of VM instances as resource demand increases. The maintenance overhead for virtualized environments is nearly zero compared to the complexities of managing a physical device farm.

Challenges

Host-based fuzzing isn't always a zero-cost solution. The primary downsides typically stem from two key aspects: the engineering effort it can require and discrepancies between the host and the target device.

While many Android components are open source projects that may have generic fuzzers available, integrating them into a continuous fuzzing pipeline requires that they interact with the Android build system. This system is notoriously complex and can be highly challenging for newcomers to understand and navigate.

Furthermore, core Android components typically rely on specific Android-native services and architectures, such as binder services or specialized device drivers. To make these components runnable and fuzzable within a standard host environment, these dependencies must be correctly mocked out or replaced with host-compatible implementations. This task can require significant engineering effort, even for highly experienced Android developers.

Second, a fundamental principle of effective fuzzing is to run the fuzzer as close as possible to the production target platform. Host-based fuzzing, by definition, violates this principle in several ways. This violation can lead to false alarms (false positives) or, more critically, missing true bugs (false negatives).

A prime example comes from differences in the devices' architectures. Older Android devices use the ARM architecture, which, by default, often triggers a data abort exception when attempting to access unaligned data. This same behavior doesn't occur on x86/AMD64 hosts. Consequently, fuzzing on the host will miss exceptions related to unaligned access, which can be important indicators of potential DoS bugs on the actual device.

Similarly, much host-based fuzzing requires mocking critical services. These mocking implementations may behave subtly differently than the real Android environment. This difference in behavior is another potential cause of false alarms or, more dangerously, a failure to reproduce or detect bugs that manifest under the precise conditions of the real target environment.

Enabling Host Fuzzing

As noted at the beginning of this chapter, host-based fuzzing is not a new concept but rather an avenue for further optimization. If you followed the steps in Chapter 2, you've already touched on it by enabling the `host_supported` build flag. However, to fully leverage the benefits of host fuzzing, you can progress through increasingly deeper tiers of engineering support.

This depth is largely dictated by the target component's complexity and its dependencies on the Android environment. Let's explore these layers one by one.

Turning On the Flag

The easiest method of fuzzing on the host applies to components with minimal platform-specific dependencies. The Android build system provides a special flag, `host_supported`. By setting this flag to `true` in the component's build file (*Android.bp*), you can explicitly instruct the build system to generate a host-compatible build flavor in addition to the standard device flavor:

```
host_supported: true
```

If you're lucky, the component you're trying to fuzz relies on no special hardware, and all its transitive dependencies already support host compilation (by having their own `host_supported` flag set to `true`). Then, you'll get a host-based fuzzer for free.

A good example of this case is `dng_sdk fuzzer`, which you can find in the AOSP tree at *https://android.googlesource.com/platform/external/dng_sdk/+/refs/heads/main/fuzzer/Android.bp*. Its build file looks like this:

```
--snip--
cc_fuzz {
    name: "dng_parser_fuzzer",
    host_supported: true,
--snip--
    static_libs: [
        "libdng_sdk",
        "libjpeg",
        "liblog",
        "libz",
--snip--
    ],
```

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

In this example, the fuzzer project has `host_supported` set to `true`, and to build the host-supported fuzzer with memory sanitization enabled, you can execute the following command:

```
SANITIZE_HOST=address make -j $(nproc) dng_parser_fuzzer
```

Once the build finishes successfully, you should be able to confirm that the fuzzer executable was generated under *$ANDROID_BUILD_TOP/out/host/linux-x86/fuzz/x86_64/dng_parser_fuzzer/*. This approach works because all static libraries linked by the fuzzer (including libdng_sdk, libjpeg, and so on) had already been configured with host-compatible build support.

Enabling the Host for All Dependencies

In most real-world scenarios, simply turning on `host_supported` will result in build errors, indicating that at least one necessary component in the dependency graph doesn't support a host build. To resolve this, you must iteratively trace the dependency graph and enable host support for each of the failing dependencies.

Begin by analyzing the build logs to identify the specific library or module that is failing compilation because of a missing `host_supported: true` flag. Then, modify the build file for the identified dependency to enable host support.

Unfortunately, no single command enables host support for an entire subgraph of dependencies simultaneously. Therefore, you must iterate through this process multiple times until you've successfully enabled all dependencies that your fuzzer requires for host compilation. While this can be time-consuming, it's generally a manageable effort.

Mocking Platform-Specific Components

The final and most complex case occurs when you trace the dependencies down to components that fundamentally can't support a

host build. These typically consist of modules that rely on core platform services or hardware.

Examples of such dependencies include binder services, which rely on the Android-specific Binder driver not available in standard Linux environments, and components requiring special hardware, like hardware-assisted decoders or specific device drivers.

At this stage, you'll need to expend significant engineering effort to first determine precisely what functionality the un-fuzzable dependency provides to your target code, then provide a *mock*—a substitute, host-compatible implementation of that dependency that provides the required API and simulates the expected behavior.

The Android build system includes a crucial feature for customizing platform-specific behavior: the `target` feature, which allows you to define different build configurations (sections) for each target environment, enabling you to seamlessly substitute platform-specific code with your host-compatible mocks.

The following example, from the Android Bluetooth stack's libbtif library, demonstrates this technique. Specifically, we've swapped out device-only dependencies for mocks:

```
cc_library_static {
    name: "libbtif",
    defaults: ["fluoride_defaults"],
    include_dirs: btifCommonIncludes,
    srcs: [
        "avrcp/avrcp_service.cc",
        --snip--
        "src/stack_manager.cc",
    ],
    local_include_dirs: [
        "include",
    ],
    target: {
        android: {
            whole_static_libs: [
                "libPlatformProperties",
```

```
            ],
            srcs: ["src/btif_avrcp_audio_track.cc"],
            shared_libs: ["libaaudio"],
        },
        host: {
            srcs: [
                "src/btif_avrcp_audio_track_linux.cc",
                "src/btif_leaudio_hal_version_host.cc",
            ],
        },
    },
}
```

In this code, the device flavor of `android:` uses libaaudio, an Android-specific audio library, and links a device-specific source file, *btif_avrcp_audio _track.cc*.

But to allow us to fuzz this library on the host, the `host:` subsection inside the `target:` block overrides these dependencies. Specifically, it omits the device-only libaaudio library and substitutes the device source file with a host-compatible mock implementation, *btif_avrcp_audio_track_linux.cc*.

This method of providing host support for critical libraries is often tricky, requiring both a deep understanding of the target feature and a significant engineering effort to correctly mock the platform's behavior.

It's also critical to remember that this mocking process introduces a layer of abstraction that is always an approximation of the real environment. Therefore, while mocking enables high-speed fuzzing, you must continually keep an eye on an actual on-device fuzzer to confirm that the host environment isn't leading to missed bugs or false positives caused by imperfect mocking.

Host Fuzzing Beyond Android

The principles of host-based fuzzing are applicable to any fuzzing projects targeting specific hardware. In the past, the Android Red Team

has applied this technique to fuzz components operating below the Android kernel, such as TrustZone applications and chipset firmware.

A common pattern in these projects is that while they ultimately depend on a specific piece of hardware, the majority of the implementation consists purely of software. Therefore, they're easy to convert into host-based fuzzing targets to achieve significantly better performance.

Unlike the Android platform, which already provides comprehensive build system support for host architecture, these embedded projects typically use their own build systems specifically designed for the target platform. Therefore, you'll need to manually modify the existing build system to let them compile on standard Linux environments.

Usually, this process boils down to the following four steps:

1. Add support for the correct toolchain. If you plan to use libFuzzer, for example, you'll want to use Clang as the compiler. Most build systems allow you to customize the compiler selection, either through environment variables or via modifications to sub-Makefiles.
2. Add the host CPU architecture. This step usually involves modifying the Makefiles to adjust the compiler flags, ensuring that they target the host architecture (for example, AMD64). You'll also need to include the necessary fuzzing instrumentation flags (such as `-fsanitize=fuzzer`).
3. Isolate device-specific code from general software files. For embedded systems, software typically accesses hardware through memory-mapped I/Os (MMIOs). This often means that a HAL is separating the general software logic from device-specific code. You must draw a clear line and include only the device-agnostic source code in your fuzzer build.
4. Add the missing dependencies. This involves mocking hardware pieces such as the MMIOs for registers, which will no longer exist on the host.

It's crucial to note that the deeper levels of host-fuzzing implementations, particularly within embedded systems and proprietary codebases, are often highly project-specific. Furthermore, they frequently involve the integration of proprietary intellectual property (IP). Because of these constraints, we can't provide a readily available or runnable code example that perfectly illustrates these complex mocking and abstraction techniques, but we encourage you to explore these practices on your own.

Wrapping Up

At the cost of some engineering effort, host-based fuzzing can greatly improve the performance of your bug-discovery tools by using a standard POSIX environment. It's a very useful approach that any security researcher should consider when they look into fuzzing Android or any device-specific software. If you choose to go this route, however, always compensate for the architectural differences by performing on-device testing to ensure adequate coverage of any hardware-specific issues.

OceanofPDF.com

EPILOGUE: FUZZING IN THE AGE OF AI



In recent years, AI has been integrated—for better or for worse—into every area of software development, and fuzzing is no exception. In fact, fuzzing is proving to be especially friendly to AI automation because of its leniency on complete accuracy. After all, the practice is based on making billions of wrong guesses, or mutations, in hopes of getting one guess right. You can safely run an inefficient fuzzer, fuzz a useless function, or attempt a bad test case without causing a production outage; this capability creates a generally low-risk, high-reward environment for experimentation.

To stay ahead of the curve and understand what is coming to the world of fuzzing, we present in this epilogue the ways in which researchers and teams have integrated AI into their fuzzing workflows. We'll consider several key areas: how security engineers are successfully applying new fuzzing techniques to find security flaws in the large language models (LLMs) themselves; how researchers and security engineers are applying LLMs to enhance core fuzzing logic; and how teams are trying to automate the entire fuzzing process, from identifying targets to generating patches. These fields have encountered hurdles but have also seen promising successes.

Fuzzing AI Models

While LLMs offer significant potential, they also introduce new targets for malicious actors. So, you might be wondering whether it's possible to use fuzzing techniques to bolster the security of these models.

On its face, traditional fuzzing practices may seem like a poor fit for testing LLMs. Fuzzers typically feed random data to a program to find vulnerabilities, but feeding an LLM a random stream of data won't crash it; the LLM will simply try to interpret the data as a prompt and predict the best response.

When fuzzing an LLM, then, we're thinking not about breaking its syntax in a way that causes a crash, but instead about finding ways to skirt around its safety systems through semantics. Our fuzzing strategy must evolve as a result, from generating random noise into persistently trying to outsmart the LLM's safety systems through logic traps, clever framing, and hidden meanings to find jailbreaks.

Jailbreaks, simply put, are inputs that successfully trick the model into bypassing its safety alignment and ethical guardrails to produce undesirable content. We can classify these vulnerabilities into the following categories:

Generation of harmful content These jailbreaks coax the model into providing instructions for illegal activities, generating hate speech, or producing other information that the LLM has been specifically trained to avoid.

Prompt injection These jailbreaks trick an LLM into obeying malicious instructions that are hidden within an otherwise benign-looking prompt. Prompt injection can take many forms and can be as simple as an instruction like "Ignore all previous instructions." The prompt can also be hidden in a web page link that the LLM is asked to process.

Information leakage These jailbreaks are used to extract sensitive data from the LLM, including the proprietary system prompt or information about its training dataset.

The process of fuzzing LLMs for these vulnerabilities should look familiar to you at a high level if you've followed along with the book thus far. As with traditional fuzzing, you'll generate a closed-loop system that mutates the input upon each iteration. When fuzzing LLMs, however, we often let another LLM power the mutation engine, prompting it to generate the next prompt that may fool the target LLM.

This process introduces a difficulty: How do we create a sanitizer that can identify the prompts that have successfully fooled the LLM? To catch responses that violate the LLM's alignment, some security practitioners have relied on an *LLM-as-judge* method, whereby the target LLM's response is fed to a judge LLM, which is prompted with a detailed rubric to evaluate whether the response constitutes a successful jailbreak. Practitioners can then add successful mutations to a seed pool and prioritize them in future rounds.

The reliance on an LLM-as-judge introduces a recursive security challenge, however. The integrity of the entire testing framework now hinges on the ability of the judge LLM to be able to use the detailed evaluation rubric to identify the jailbreak in a way that the model being fuzzed has missed.

As a real-world example of LLM fuzzing, let's take a look at FuzzyAI, an open source framework developed by CyberArk Labs with the mission to provide security engineers with a testing tool to validate the safety of their LLMs. FuzzyAI takes seed prompts that try to exploit one of the jailbreak vulnerability categories, then mutates the prompt in some way. For example, it may convert the prompt into pig latin, which takes the first letter of the word and moves it to the end of the word, and then adds *ay* (turning *banana* into *ananabay*, for example). FuzzyAI may also use other exploits, such as these:

Shuffle inconsistency (SI) attack This attack exploits the gap between an LLM's comprehension ability and its safety filters. If the gap is large enough, a harmful prompt can be shuffled around to the point that the surface-level safety filters no longer recognize it as harmful, yet the underlying model is still able to reconstruct the

intent and answer the question. For example, you may swap the order of all the sentences or swap one letter in each word. The goal is landing in the gap between what the safety filter can recognize and what the LLM can understand well enough to respond to.

ASCII art Asking an LLM, "How do I create a bomb" in plaintext is likely to fail instantly, but when we ask the same question using ASCII art, the LLM's alignment training may fail to mark the question as inappropriate and answer it.

Many-shot jailbreaking This technique adds a back-and-forth dialogue between a user and a fake AI agent to the prompt. This dialogue shows the model readily answering harmful questions, leveraging the model's in-context learning capability and showing the LLM that the correct behavior is to answer the harmful question.

Hallucination-based attack After the creation of a model, some engineers add filters to the model through reinforcement learning from human feedback (RLHF). This attack attempts to trick the model to bypass its RLHF-tuned "personality" through an elaborate trick question, such as feeding the model five paragraphs of reversed text and asking the model to unreverse the seventh paragraph. Because the seventh paragraph doesn't exist, the ambiguity, combined with the confusing text, can break the model's normal filtering and cause it to lose track of its safety guardrails entirely. This process tricks the model into a state where, no matter how good its RLHF filter is, the filter doesn't work because it has been completely bypassed.

Genetic This attack is fuzzing in the more traditional sense, where the engine uses a genetic algorithm to mutate the prompt into different versions of the same questions to try to get a harmful response from the LLM.

FuzzyAI includes many more techniques in its growing library of attacks. FuzzyAI supports the LLM-as-judge concept to analyze the responses from the model and determine whether the model was

jailbroken. In 2024, CyberArk released a statement saying the tool was able to jailbreak every model tested.

Automating the Fuzzing Process

While fuzzing AI models represents a critical new frontier, the most immediate impact of AI on the daily life of a security engineer is the automation of the fuzzing process itself. Writing a fuzzer for every function in a code-base can take days or weeks of developer time. To address this bottleneck, Google's OSS-Fuzz team has been using its OSS-Fuzz-Gen tool to automate the generation of fuzzers for target functions.

If you've ever tried to generate a fuzz harness for a function based solely on its signature or definition, you've likely gotten mixed results. OSS-Fuzz-Gen is an agentic program that performs static analysis on the code to find the target function's context. This context then becomes the building blocks of the prompt sent to the backend LLM to create the fuzz harness.

To gather this context, the tool looks at the full function source code, the function signature and details about the arguments, and cross-references in the codebase that provide the LLM with examples of the target function being used. Providing this context dramatically decreases the LLM's hallucinations and its use of the function in incorrect patterns. Thus, the quality and accuracy of the generated fuzzer relies directly on the success of the context-gathering stage.

After gathering the context, OSS-Fuzz-Gen enters the workflow to generate the fuzz harness and the fuzzer's build entry. In this workflow, OSS-Fuzz-Gen will first build an initial prompt that contains a *few-shot* approach, providing a few static examples of what a good fuzz harness looks like.

To build the harness, the LLM is instructed to use the FuzzedData-Provider library of parsing logic (covered in Chapter 3), which gives the LLM a well-tested API for slicing input bytes into typed values. OSS-Fuzz-Gen then adds the contextual information gathered in the previous step to the prompt, along with coding anti-patterns to avoid.

This meticulous approach to prompt engineering is vital to generating high-quality fuzz harness code that effectively covers the desired function.

After building the initial prompt, OSS-Fuzz-Gen sends it to the chosen backend. The resulting build code and fuzz harness get added to the project, and the process enters the build phase: a loop of building the code, analyzing the error, generating and applying a fix, and then trying to build the code again.

The framework tries to use the project's build system and dependencies to compile the generated code in an OSS-Fuzz Docker image. If this build phase fails, OSS-Fuzz-Gen collects context from `stdout` and `stderr` in its raw, unparsed form. In a previous era, an important step here would have been to parse out the minimal information needed to provide an accurate error message to the LLM. LLMs have become able to adeptly parse the entire error message and even multiple source files in the prompt sent to it. There may come a time when we can simply feed the entire codebase to the LLM to produce a fix.

The prompt should also include the current, broken harness code. The LLM will then act as the program's debugger, attempt to regenerate the harness with the applied fixes, and return it. The program saves this new harness and enters the build phase again.

This generate-build-fix loop continues automatically, meaning the LLM doesn't have to fix all errors in one shot. After the build succeeds, the developer can commit the new fuzzer to the codebase. If the project is onboarded to OSS-Fuzz, the fuzzer will start running in the open source ClusterFuzz infrastructure to try to find bugs. Fuzzers generated through this workflow have already been credited with identifying over 30 new bugs and vulnerabilities across a wide range of projects.

Solving Fuzzing's Challenges

Beyond automating the fuzzing workflow, AI is being used to improve the core logic of the fuzzing engines themselves. While traditional gray-box fuzzers are efficient, they famously struggle with two major hurdles:

handling complex path constraints (like "magic value" checks beyond simple comparisons like `if (input == 0xDEADBEEF)`) and generating valid, highly structured inputs with low engineering effort.

This section explores how engineers are integrating LLMs directly into the fuzzing loop to overcome both challenges. We'll look at projects like HyLLfuzz, which uses an LLM as a solver to break through path constraints, and LLAMAFUZZ, which uses an LLM to perform structure-aware mutation on complex input formats. First, however, let's review how traditional fuzzing practices have attempted to solve these challenges.

With Gray-Box and White-Box Fuzzing

Most of the fuzzing we've covered in this book can be considered gray-box fuzzing, as it involves the technique of instrumenting a fuzzer, either at compile time or dynamically after it's been built, to enable the fuzzing engine to get some basic data-flow feedback and code-coverage information. These practices enable the fuzzing engine to generate informed mutations to the input and effectively explore the code.

Two other fuzzing methods exist. In *black-box fuzzing*, the fuzzing engine gets minimal feedback from the code, such as return codes or timing, and can only blindly feed the fuzz binary the generated input. *White-box fuzzing* uses techniques like symbolic or concolic execution to directly analyze the program's path constraints and derive the specific inputs needed to reach specific code paths.

As you might imagine, most fuzzers are gray-box fuzzers for a reason. In practice, white-box fuzzing is not that viable in real-world scenarios. These fuzzers suffer from extreme execution slowness and "path explosion," a scenario where the number of possible execution paths grows exponentially with every branch condition in the code. This immense computational overhead makes them impractical for testing large, complex applications.

Even gathering feedback from each input to explore deeper and deeper regions of the code, as we do in gray-box fuzzing, hits a wall eventually. Without full symbolic execution or deeper insights into the

code, the fuzzer will eventually run into a roadblock that it can't pass (or is extremely unlikely to ever pass). Consider the following code snippet:

```
if (magic_value == 0xDEADBEEF) {
    // Interesting function that is unlikely to ever be hit with simple
    // gray-box fuzzing
    vulnerable_function(input);
}
```

We considered such comparisons in previous chapters. Through the use of advanced gray-box fuzzing tools such as `value-profile` for libFuzzer and `CMPLOG` for AFL++, we succeeded at passing this check, because the check (comparing the input to the value `0xDEADBEEF`) is very simple. Let's take a look at a more complex comparison, though:

```
uint32_t x = read_input_x(); // For example, 0xAAAAAAAA
uint32_t y = read_input_y(); // For example, 0xBBBBBBBB
uint32_t result = (x * 3) + (y / 2);
if (result == 0x12345678) {
    vulnerable_function(input);
}
```

Here, `CMPLOG` will fail, since it works by looking at the value of `result` and the value it is being compared against (`0x12345678`), then looks for the value of `result` in the input. After finding the value of `result` in the input, `CMPLOG` will replace that value with `0x12345678`. The issue is that since `result` is calculated by multiplying and dividing values from the input, the value of `result` won't be in the input, so `CMPLOG` won't be able to find and replace that value. To get past this branch, the fuzzer has to work backward from the comparison to figure out what input produces `0x12345678`, rather than directly searching for that value in the input. That's what white-box fuzzing does.

Therefore, in situations like this, we might move into white-box fuzzing. Whereas gray-box fuzzing observes the program from the outside through coverage feedback, white-box fuzzing directly analyzes the program's logic to figure out the exact input needed to reach a given branch.

For the roadblock we've just considered, for example, a white-box approach can look at the comparison `(x * 3) + (y / 2) == 0x12345678`, work backward through the math, and compute valid values of `x` and `y` that satisfy it, something a gray-box fuzzer can't realistically do by mutation alone. The catch, as we've mentioned, is that this analysis is slow. Doing it for every comparison in a real-world program would crush your test-case throughput, defeating the purpose of fuzzing in the first place.

This is why, when white-box fuzzing is used, it is used in combination with gray-box fuzzing in a hybrid approach. The gray-box fuzzer runs at full speed, doing what it does best (high-throughput mutation and coverage exploration), and the white-box analysis is invoked only when the gray-box fuzzer plateaus on a roadblock it can't get past. Once white-box analysis produces an input that breaks through, that input is handed back to the gray-box fuzzer to continue exploring from there. You get the best of both worlds: the speed of gray-box fuzzing for the bulk of the work, and the analytical power of white-box fuzzing to unblock the fuzzer when needed.

If hybrid fuzzing is so great, why aren't all fuzzing campaigns hybrid? The blunt truth is that, in real-world scenarios, it is generally very difficult to get it to work properly and effectively. While academic papers frequently tout the benefits of hybrid fuzzers, these results are notoriously difficult to reproduce outside of a controlled lab environment. In practice, the sheer computational overhead of the white-box solver drags down the entire system. Over a standard 24-hour campaign, a well-tuned, pure gray-box fuzzer like AFL++ will generally outperform a hybrid setup on a real-world target.

Constructs that are routine for a gray-box fuzzer, like loops driven by input values or memory writes at input-controlled addresses, can blow up the analysis time. This makes hybrid fuzzing more of a research-grade setup than a plug-and-play tool, and it's a big part of why most fuzzing campaigns in industry remain purely gray-box despite the known limitations.

With LLM-Generated Inputs

To solve the problem of complex checks, a tool called HyLLfuzz, created by Ruijie Meng, Gregory J. Duck, and Abhik Roychoudhury, takes a different approach from the ones considered in the preceding section. Instead of dealing with the slow, brittle analysis that traditional white-box fuzzers rely on, HyLLfuzz feeds the relevant slice of code directly to an LLM and asks it to come up with an input.

More specifically, when the gray-box fuzzer plateaus on a roadblock, HyLLfuzz extracts a code slice and execution context surrounding that roadblock and hands it to an LLM, asking it to produce an input that satisfies the path constraint. This process requires no translation step, environment modeling, or heavy constraint solvers. Successful inputs can then be used for future mutations.

The results in the developers' paper, "Large Language Model Assisted Hybrid Fuzzing," back this up. Compared to state-of-the-art hybrid fuzzers like QSYM and Intriguer, HyLLfuzz achieved roughly 31 to 59 percent more branch coverage, and its LLM-based concolic execution phase ran 3 to 19 times faster than the equivalent constraint-solving phase in traditional tools. Most importantly, running HyLLfuzz eliminates the need to manually model the environment or set up complex syscalls, drastically lowering the barrier to entry for hybrid fuzzing.

By Automating Structure-Aware Mutation

Solving complex path constraints, as HyLLfuzz does, is one major hurdle for gray-box fuzzing. The other, equally significant challenge is generating semantically valid inputs for highly structured data formats.

A basic gray-box fuzzer generates each input based on semantically blind mutation strategies such as bit-flipping and block-based operations. While highly effective for simple inputs, this approach is very inefficient when fuzzing structured data like JPG images, XML, or complex network protocols. The vast majority of random mutations will almost always break the correct syntax (for example, by creating an invalid image header or breaking a closing XML tag), and input validation code will reject these inputs before they get a chance to reach deeper, more interesting program logic.

One traditional answer is grammar-based fuzzing, which uses developer-defined rules to generate valid inputs. However, these grammars are time-consuming to write, and they can inadvertently restrict the range of possible inputs, potentially limiting the vulnerabilities that can be found. The fuzzer needs to generate inputs that are not just structurally valid (in that the parser accepts them) but also semantically meaningful (in that the values inside exercise different code paths).

To illustrate the difference, consider an XML document with an `<age>` element. A structurally valid mutation would keep the tags balanced and the format intact. For example, the mutation might change the spacing or append an empty attribute, modifying the element to look like `<age id=""> 10 </age>`. A semantically meaningful mutation goes a step further and changes what the document says, such as swapping the age field from `<age>10</age>` to `<age>45</age>`, changing a numeric ID, or replacing an enum value with a different one. The first kind of mutation gets you past the parser, while the second kind gets you into the code that processes those values, which is usually where the interesting bugs live.

This is where projects like LLAMAFUZZ come in. Created by Hongxiang Zhang, Yuyang Rong, Yifeng He, and Hao Chen, LLAMAFUZZ shows how an LLM can step in to produce interesting input that the fuzzer might not generate. LLAMAFUZZ will asynchronously use an LLM to generate interesting inputs that match the desired structure in the background while normal fuzzing runs. After generating the input, the LLM adds it to a queue to be fed into the fuzzer to allow the fuzzer's coverage to grow and explore new paths.

Auto-Triage

An LLM-based approach to fuzzing may generate more crashes than traditional fuzzers do, but crashes are useful only if you can figure out what they mean. If a fuzzing campaign produces thousands of crashing inputs in a single weekend run, the bottleneck quickly shifts from finding crashes to understanding them.

After a campaign, you'll want to know which of these crashes are duplicates of each other, which are duplicates of bugs you've already filed, and which are interesting (and which are shallow). You'll also want to know the actual root cause of each unique bug.

This work, which has traditionally been done by hand, scales poorly. As the automation tools discussed earlier in this epilogue make harness generation and fuzzing campaigns cheaper, crash triage becomes a larger and larger share of the human effort. This is the next obvious place for AI to help.

Crash De-Duplication

The first and most visible problem is crash de-duplication. Fuzzers like libFuzzer and Syzkaller ship with built-in de-duplication functionality that groups crashes by simple heuristics—typically, the top few frames of the stack trace, or a hash of the program counter at the crash site. These heuristics are fast and cheap, but they routinely get the wrong answer, in two directions.

Namely, they often group distinct bugs together because their stack tops happen to look similar, and they split single bugs across many groups because the same root cause can surface in slightly different stack shapes depending on the input. As a result, the developer can't trust the triage queue at face value.

Recent research has shown that LLMs are well suited to solving these problems. A stack trace is text, and LLMs are good at understanding text in context. "Do these two stack traces describe the same underlying bug?" is exactly the kind of fuzzy semantic comparison that simple hashing can't do, but an LLM can. Many ongoing projects today are looking to solve this exact issue.

Root Cause Analysis

After you've de-duplicated a bug, the next phase is root cause analysis. Once you have a unique crash, you still need to figure out *why* it crashed, and ideally, where in the code the bug lives, which may not be the same place as where the crash happened to surface.

Newer agentic fuzzing workflows are starting to include dedicated triage agents for this task. These agents' only job is to take a crashing input, replay it under a debugger, walk the execution backward, and produce a concise, human-readable explanation of the root cause, along with a minimal reproducer.

Remediation

To fix the identified bug in the project you found, you can look into upcoming tools like CodeMender from Google, which aims to use LLMs to automatically patch the vulnerabilities found through fuzzing or LLM analysis. While not strictly tied to fuzzing, CodeMender completes the cycle of de-duplicating, triaging, and fixing the vulnerability.

For fixing vulnerabilities, CodeMender will take in the stack trace, the crashing input, and the build command for the harness to apply fixes and verify that the fix does indeed remove the vulnerability. It can even be hooked up to run your unit tests as part of its validation step for the produced fix. CodeMender also can find vulnerabilities for you and assign severity ratings to the vulnerabilities it finds, and can be used to assign severity ratings to vulnerabilities you feed it.

Wrapping Up

This epilogue provided an overview of the future trajectory of fuzzing as AI tools become integrated into its pipeline. We discussed how security engineers are actively fuzzing LLMs by pivoting from syntax-based breaking to semantic jailbreaking techniques that aim to bypass the LLM's safety features.

We also discussed how AI is making bug finding more efficient by powering automation tools like OSS-Fuzz-Gen to generate harnesses and by evolving the core fuzzing logic with hybrid solvers like HyLLfuzz and structure-aware mutators like LLAMAFUZZ.

After you've identified bugs via fuzzing, LLMs can help triage, de-duplicate, analyze, and remediate those bugs. We covered a number of

research projects that have successfully tailored LLMs to these tasks.

Before you close this book, a word of advice: While we'll continue to automate fuzzing workflows in the future, engineers need a solid understanding of the traditional technologies underlying this automation in order to effectively direct and validate these advanced tools. So, take the time to practice the foundational techniques covered throughout these chapters. Once you're comfortable writing harnesses and analyzing crashes, we encourage you to move on to experimenting with some of the newer LLM fuzzing projects on your own. No matter what the LLM project landscape looks like by the time you finish this book, to effectively use these projects, you'll need a solid understanding of how fuzzing works.

OceanofPDF.com

INDEX

A

`aarch64`, 140
`aarch64_16k`, 140
abusing accessibility services, 55
ACK (Android Common Kernel), 124
Activity Manager, 44
`ActivityManager`, 111
ADB (Android Debug Bridge), 15, 22
AddressSanitizer (ASan), 11, 28, 95, 126
address space layout randomization (ASLR), 106
`adler32`, 83
Advanced Protection mode, 47
AFL++ fuzzing engine, 22, 25–26, 29–30, 78, 181
    `CMPLOG`, 180
AI automation, 175
AIDL (Android Interface Definition Language), 53, 116
AI models, fuzzing, 176–178. *See* also auto-triage
    automating the fuzzing process, 178–179
    challenges, 179
        automating structure-aware mutation, 182
        gray-box fuzzing, 179–181
        LLM-generated inputs, 181–182
        white-box fuzzing, 179–181
American Fuzzy Lop (AFL) engine, 29
Android architecture, layers of, 43–44
    fragmentation, 51–53

hardware-based security, 47–51
permission model, 43, 53–55
security evolution, 45–47
SELinux, 56–60
Android.bp files, 19, 31–33
Android build system, 21, 79
Android Common Kernel (ACK), 124
Android Debug Bridge (ADB), 15, 22
Android Interface Definition Language (AIDL), 53, 116
Android Keystore, 48–50
AndroidManifest.xml file, 53
Android Open Source Project (AOSP), 15
Android Package Kit (APK) file, 60
Android runtime (ART), 44
Android Security Bulletins, 70
Android Vulnerability Rewards Program, 104
AOSP (Android Open Source Project), 15
API contract enforcement, 111–113
API-to-service mapping, 67
APK Signature Scheme v2, 60
`app_data_file` type, 57
application framework, 44
app signing, 60
ARM-based processors, 48
ART (Android runtime), 44
ASan (AddressSanitizer), 11, 28, 95, 126
ASCII art, 177
ASLR (address space layout randomization), 106
`assert` statement, 98
`atof()`, 74
`atoi()`, 74

`atol()`, 74
`atoll()`, 74
attack vector
    local, 103
    physical, 104
    proximal, 103
    remote, 103
`audioserver`, 59
auto-triage, 183
    crash de-duplication, 183
    remediation, 184
    root cause analysis, 183

B

.bashrc file, 16, 22
BeOS, 113
binary instrumentation, 9
Binder, 75, 109
    kernel driver, 110–111
`binder2corpus`, 121
`binder_parcel_fuzzer`, 113–116, 119–120
BiometricPrompt API, 51
biometrics, 48, 51
Bionic, 25
black-box fuzzing, 7, 110–111. *See also* gray-box fuzzing; white-box fuzzing
blueprint files, 19
Bluetooth, 103–104, 172
bootloader, 103
`BpBinder`, 111

branch, 7
bug bounty researcher, 100
bugs
    detection, 10–12
    reproduction, 96–97
    severity assessment, 100–106
    triage, 93–106
build.sh shell script, 125
build targets, 19
bypassing permission checks, 56
bytecode of Android apps, 44
bzImage file, 125

C

camera server, 117–118
`cc_fuzz`, 19–20, 78, 83, 91
`cc_library`, 19–20, 78
CERT C Coding Standard, 74
`cflags`, 79
`checkSelfPermission()`, 55
Chinese Android ecosystem, 52
`clear_code()`, 149
client codebase, 111
ClusterFuzz, 37–38
`cmpcov`, 6
`CMPLOG`, 180
    code coverage instrumentation, 9
    execution, 101
    footprint, 48
    under test, 7

CodeMender, 184
cognitive complexity, 73
*common-android-mainlin*e, 124
common-modules, 155
compile-time instrumentation, 9, 29
constrained process, 103
`ConsumeBool()`, 116
Content Providers, 44
contributor health, 75
`copy_from_user()`, 147, 149
corpus, 29
    `binder2corpus`, 121
    collection, 89–90
    minimization, 37
    seed corpus, 90–91
coverage-guided fuzzing, 7, 9, 118
coverage reports, 38–42
C++ vtables, 101
crash analysis, 93. *See also* severity assessment
    crash de-duplication, 99–100
    initial validation, 94–99
criticality score, 70–73
cross-site script inclusion (XSSI) attacks, 68
cryptographic hashes, 6
Cuttlefish
    emulator, 16–18
    kernel fuzzing with, 124–125
    Syzkaller configuration, 128–130
    testing with, 23–24
cuttlefish-basic.json, 157
CyberArk Labs, 177–178

cyclic redundancy checks (CRCs), 6
cyclomaticity, 73

D

DAC (discretionary access control), 56
data-handling tools, 27, 86
data parsing, 31–37
    FuzzedDataProvider library, 34–35
    libprotobuf-mutator library, 35–37
    manual parsing, 32–34
`ddk_module`, 140
debugging
    debug information, 94–95, 106
    host environments, 168–169
    kernel logging, 159–161
    symbols, 39, 95
`DEFINE_PROTO_FUZZER` macro, 36–37
denial of service (DoS) attacks, 10, 102
de-prioritization, 97
device setup, 16–17
    emulated devices, 18
    physical devices, 17–18
dictionary file, adding, 88–89
digital rights management (DRM), 48
`disable_syscalls`, 158–159
disabling checks, 7
discretionary access control (DAC), 56
Docker, 127
domains, SELinux, 56–57
double-free vulnerabilities, 101

dynamic instrumentation, 30

E

empty fuzz harness, 78–80
`enable_syscalls`, 158
encryption, 45, 50
error logging, disabling, 84–85
example.pb.h, 35
example.proto, 35–36
Exploit.AndroidOS.Lotoor attack, 45
`export_header_lib_headers`, 20
exposure to third-party applications, 67
Extensible Markup Language (XML), 5
extensive physical access, 104

F

`fd_lockbox`, 150–151
fdp_fuzzer.cpp, 34–35
feedback-driven fuzzing. *See* coverage-guided fuzzing
`fillRandomParcel()`, 115
fingerprint readers, 138, 145
flags for `syz-extract`, 154
`flags` parameter, 151
`fmemopen()`, 81
`_fops` object, 146
fork server, 29
`format_fuzzer`, 19, 21
fragmentation, 51–53
    Chinese Android ecosystem, 52

Project Treble, 52–53
Frida, 30
full-disk encryption, 45
function pointers, 5, 101
FuzzedDataProvider library, 34–35, 178
fuzz harnesses
creating, 31–33
definition, 7
empty, 78–80
running, 80
Soong integration, 19
fuzzing. *See also* fuzzing binder services; fuzzing a library
AI models, 176–178
definition, 4
false positives, 98
large-scale systems, 109–122
life cycle, automating, 37–38
vs. other testing types, 12–13
overview, 1–2
process, 4–12
detecting bad behaviors, 10–12
evaluating progress, 7–10
generating random inputs, 4–7
running code under test, 7
fuzzing a library creating a fuzz harness, 78–80
disabling error logging, 84–85
fixing memory leaks, 86–87
implementing library code, 80–84
improving performance, 88–91
fuzzing binder services
automating a service fuzzer, 115–117
fuzzing multiple services, 117–119

fuzzing parcel objects, 113–115
optimizing coverage, 119–120
scaling to multiple services, 121
testing the fuzzer framework, 122
fuzzing engines, 6–7, 29–31. *See also* coverage-guided fuzzing
AFL++, 29–30
LibAFL, 30–31
libFuzzer, 29
Syzkaller, 30
fuzzing environment
Android Debug Bridge (ADB), 22
Android source, downloading, 15–16
device setup
emulated devices, 18
physical devices, 17–18
Soong build system, 19–22
testing with AFL++, 25–26
with Cuttlefish, 23–24
on a host machine, 25
on a physical device, 24–25
`fuzzService()`, 115–116
`fuzz_service_test`, 122
FuzzyAI, 177

G

generative fuzzing, 5
genetic attack, 178
`getRandomBinder()`, 115
`getRandomFds()`, 115
`gets()`, 74
Ghidra, 8

GingerBreak, 45
GitHub, 127
Gitiles REST API, 68
Git repositories
    Android source, 16–17
    contributor metrics, 69
global offset table (GOT), 45
GNU Compiler Collection (GCC), 30
Goldfish, 16–17
grammar-based fuzzing, 5, 182
granular app permission system, 45
gray-box fuzzing, 7, 179–182. *See also* black-box fuzzing; white-box fuzzing

H

hallucination-based attack, 177–178
hardware abstraction layer (HAL), 44, 53
hardware-based security, 47–51
    protected virtual machines, 51
    secure elements, 50
    trusted execution environments (TEEs), 47–50
hardware-enforced walls, 51
hardware interface definition language (HIDL), 53
HarmonyOS, 52
harness. *See* fuzz
harnesses `header_libs`, 20
Heartbleed bug, 4, 101
HID (human interface device), 133–134
hope-based programming, 112
host-based fuzzing

advantages, 168–169
challenges, 169–170
enabling, 170
for all dependencies, 171
mocking platform-specific components, 172–173
in Linux environments, 173–174
overview, 167–168
host machine, 25
`host_supported` build flag, 170–171
Huawei, 52
human interface device (HID), 133–134
`hwaddress`, 22, 24
HWASan, 28
hybrid fuzzing, 181–182
HyLLfuzz, 179, 181–182, 184
hypervisor, 103

I

important flags, 101
infinite loops, 10
`info_ptr`, 86
information disclosure, 101–102
initial crash validation
analyzing crash information, 94–95
confirming legitimate bugs, 98–99
reproducing crashes, 96–97
initramfs.img, 125
install-time permissions, 54
insufficient de-duplication, 100
integration testing, 12–13

intellectual property (IP), 174
inter-process communication (IPC), 56, 110
invariant-based fuzzing, 11
`ioctl` syscall, 149
`_IOW()` macros, 145

J

jailbreaks, 50, 176
Java archive file (JAR) signing, 60
Java Native Interface (JNI), 67
Java programming language libraries, 44
JavaScript Object Notation (JSON), 35, 71, 128–129
`-j` flag, 21
`JNINativeMethod` registration, 67
just-in-time (JIT) compiler, 9

K

KASAN (Kernel Address Sanitizer), 159
KCOV, 126–127
kernel-based virtual machine (KVM), 18
kernel fuzzing, 123–135
    downloading and building kernel code, 124–126
    fuzzing instrumentation, 126
    Linux Kernel Library (LKL), 132–135
        fuzz harness, 133
        high-level workflow, 134
        human interface device, 133–134
        LKL-based fuzzer, 133
        vs. Syzkaller, 135
    preparation, 124

Syzkaller, 126–131
configuration, 128–131
vs. LKL, 135
overview, 128
sandboxed environment, fuzzing inside, 131–132
setting up, 127–128
kernel-headers.tar.gz, 154
`KeyGenParameterSpec`, 50
key management, 48
Keymaster HAL, 50
Keystore system, 50
KVM (kernel-based virtual machine), 18

L

language runtime, 111
large language models (LLMs), 175–184
bug triage
crash de-duplication, 183
remediation, 183–184
root cause analysis, 183–184
fuzzing, 176–177
automating, 178–179
jailbreaks, 175–177
LLM-as-judge method, 177
large-scale fuzzing, 109–122
launch_cvd, 125
LibAFL, 6, 30–31
`libaudioutils`, 19–20
libbinder, 110
API contracts, 111–112
IPC runtime, 111

parceling protocol, 111
security vulnerabilities, 113–115
libFuzzer
vs. AFL++, 29–30
Android integration, 29
LKL-based fuzzer harnesses, 132, 134
with LLM-generated inputs, 180
output, 85
libfuzzer_macro.h, 35
`liblog`, 19
libpng
coverage reports, 41–42
fuzz harness, 80–85
image-processing library, 73
libprotobuf-mutator library, 35–37
libraries, 44
complexity, 74
contributor activity, 69–70
critical components, 73
dangerous operations, 74–75
dependent libraries, 66–67
fuzzing, 75–76
instability, 66
lines changed and inserted, 70
past performance, 70–71
privilege level of process, 71
reachability, 66–67
selecting fuzz targets, 65–71
update frequency, 68–69
libvirt, 18
Linux kernel, 44
driver, 111

Linux Kernel Library (LKL), 123, 132–135
fuzzing harness, 133–134
high-level workflow for LKL fuzzer, 134
human interface device, 133–134
object file (*lkl.o*), 133
vs. Syzkaller, 135
Linux user ID (UID), 56
Linux VMs, 169
Lizard, 74
LLAMAFUZZ, 179, 182, 184
LLMs. *See* large language models
`llvm-cov` command, 40
`LLVMFuzzerTestOneInput()`, 35, 78, 81, 86, 100, 116, 134
`llvm-profdata`, 39
local attack, 103
`lockbox`, 138–144, 150–151
lockbox.c, 138–139, 141–142
`lockbox_cleanup_module()`, 142, 146
`lockbox` device, 142
lockbox.h, 138, 154
`lockbox_init_module()`, 142, 146
`lockbox_open_flags`, 151
lockbox_repro.c, 165
Log4Shell vulnerability, 101
longjmp buffers, 101
lpm_fuzzer.cpp, 35–36
`lunch` command, 21

M

machine learning accelerators, 145

`magic_value`, 37
maintainability, 73
mandatory access control (MAC), 45, 56
manual parsing, 32–34
    constant size checks, 33
    pointer arithmetic and casting, 33
many-shot jailbreaking, 177
`-maybe-dirty` suffix, 126
`mediacodec`, 59
`mediaextractor`, 103
`mediaserver`, 59–60
`memcpy()`, 34
`__memcpy_aarch64_simd()`, 99
memory leaks, fixing, 86–87
memory-mapped I/Os (MMIOs), 174
MemorySanitizer (MSan), 28
Memory Tagging Extension (MTE), 11, 28–29, 47
Microdroid, 51
microsegmentation, 50
Microsoft Banned Function List, 74
`miscdevice`, 146
MITRE, 70–71, 104
mocking
    disabling checks, 7
    in unit testing, 12
`mode` parameter, 151
`MODE_WORLD_READABLE` support, 46
`module_exit()`, 139
module-info.json, 66–67
`module-info` target, 66
`module_init()`, 139

MODULE_ macros, 139
MTE (Memory Tagging Extension), 11, 28–29, 47
mul(), 6
mutation-based fuzzers, 5–6
mutator, 30
mutex_lock(), 149
mutex_unlock(), 149
mutual distrust, 51
myTestFunction(), 32–33

N

native symbol registration, 67
NFC chips, 145
null pointer dereference (NPD), 5–6, 100, 102

O

observer, 30
OOB READ error, 102
OOB WRITE error, 102, 113
openat$lockbox, 150
open_mode, 151
OpenSSL library, 101
OSS-Fuzz, 2, 4, 38, 178–179
OSS-Fuzz-Gen, 178, 184
out-of-bounds (OOB) access
    array, 10–11
    read, 104, 106
    writes, 101, 134
out-of-memory exception, 10

out.profraw file, 39
overly aggressive de-duplication, 100

P

Package Manager, 44
`PADLOCK_CLEAR`, 149, 153
`PADLOCK_MAGIC`, 145
`PADLOCK_TURN`, 149, 153
`panic()`, 159
`Parcel` class, 111–113
    data buffer, 114
path traversal bugs, 11
permissions, 53–55. *See also* SELinux
    app requests, 55
    escalation attacks, 55–56
    install-time permissions, 54
    request dialogs, 54
    runtime permissions, 54
physical attack, 104
physical device, preparation, 17–18
pKVM (protected kernel-based virtual machine), 51, 98
`platform_app`, 57
platform/manifest, 16
`png_create_read_struct()`, 81, 84
`png_init_io()`, 81–82, 86
`png_read_png()`, 81–82, 86
pointers, 101
POSIX environment, 169, 174
`pr_emerg()`, 160
`pr_err()`, 160

pr_info(), 160
priv_app, 57
privilege level, 102–103
progress evaluation, 7–10
Project Treble, 43, 52–53
prompt injection, 176
proof-of-concept (PoC), 100
protected kernel-based virtual machine (pKVM), 51, 98
protected virtual machines (pVMs), 51
protection-level flags, 55
Protocol Buffers (Protobufs), 35–36
    mutator, 36–37
proximal attack, 103
proximal information, 104
pr_warn(), 160

Q

quantitative analysis of reachability, 66
quick emulator (QEMU) hypervisor, 18

R

rand(), 4–5
RandomBinder, 115
random inputs generation, 4–7
reachability analysis, 75
red zones, 29
remote attack, 103
remote procedure call (RPC), 128
repo, 16–17

reproducibility of crash, 96–97
`requestPermissions()`, 55
`rlimit`, 114
rooting, 50
RSA signatures, 6
run-extract.sh, 155
runtime
    feedback, 9
    permissions, 54
    semantics, 112

S

sandboxing, 45, 56
sanitizer output, 93
sanitizers, 27–29. *See also* crash analysis
    AddressSanitizer (ASan), 28, 42
    MemorySanitizer (MSan), 28
    Memory Tagging Extension (MTE), 28, 42
    UndefinedBehaviorSanitizer (UBSan), 28
`SANITIZE` variables, 22
scheduler, 30
secure elements (SEs), 43, 47, 50, 102
secure monitor, 48
secure monitor call (SMC), 49
security context, 56
seed corpus, 90–91
SELinux, 43, 45–46
    `audit2allow`, 59
    `chcon`, 59
    domains, 56–57

how it works, 56–58
kernel enforcement, 57–58
policy, 57–59
`restorecon`, 59
shortcomings, 59
`setjmp()`, 81, 84
`set_value()`, 96
severity assessment, 100
attack vector, 103–104
behavior, 101
code execution, 101
denial of service (DoS), 102
information disclosure, 101–102
final determination, 104–106
privilege level, 102–103
shadow memory, 28
`shared_libs` tag, 19
Shellshock bug, 4
shuffle inconsistency (SI) attack, 177
silent memory corruptions, 27
social engineering, 55
Soong build system, 15, 19
build targets, 19–21
build-time options, 22
`sprintf()`, 74
SQLite database, 44
Stable AIDL, 53
stack overflows, 101
Stagefright crisis, 4, 59
stateful program, 7
`str*()`, 74

StrongBox, 50
structure-aware fuzzing, 5, 120–121
structured communication, 75
Structured Query Language (SQL), 11
`swallow_error()`, 84
`swallow_warning()`, 84
`sync` subcommand, 79
syscalls, 124, 128, 131–132, 135, 150
`system_app`, 57
`system_data_file`, 56
system-level protections, 43
system services, 109
`syz-env`, 127
`syz-executor`, 128
Syzkaller, 30, 123–124, 126, 137. *See also* kernel fuzzing; Linux Kernel Library (LKL)
  configuration options, 128–131
    fields, 129
    serial console, 130
  descriptions
    describing lockbox, 150–151
    describing padlock, 151–153
    extracting constants, 153–156
    fuzzing, 156–157
    lockbox driver, 140–144
    new descriptions, 157–159
    padlock driver, 144–149
    reproducer, running, 163–165
  GitHub, 127
  vs. LKL, 135
  overview, 128
  sandboxed environment, fuzzing inside, 131–132

setting up, 127–128
Syscalls page, 156–157
`syz-manager`, 127–128
web UI, 131
Syzlang, 30, 149–153
extracting constants, 153–156
lockbox descriptions, 150–151
padlock descriptions, 151–153
`syz-manager`, 128

T

TEEs. *See* trusted execution environments
test case, 93
test-driven development, 12
time to remediation (TTR), 76
true random number generator (TRNG), 50
trusted applications, 48
trusted execution environments (TEEs), 43, 47–50, 103
Keystore service, 49
TrustZone, 48

U

UndefinedBehaviorSanitizer (UBSan), 11, 28, 126
unit testing, 12
Unix-like systems, 123
Unix pipeline, 70
`untrusted_app`, 56–57
untrusted apps, 103
use-after-free (UAF), 101

V

`valueprofile`, 6
Vendor Test Suite, 53
`virtual_device_x86_64_external _modules`, 140
virtualization, 98
virtual machines (VMs), 18
    running in QEMU, 128
vulnerabilities in libraries, 66
vulnerability management team, 100

W

web user interface, 21
white-box fuzzing, 179–181. *See also* black-box fuzzing; gray-box fuzzing
Window Manager, 44
`-Wno-unused-parameter`, 79
work profiles, 45
world-readable files, 46

X

`x86_64/lockbox` rule, 140
XML (Extensible Markup Language), 5

Z

zero-length input, 5

OceanofPDF.com