

Community Experience Distilled

Learning Android Forensics

A hands-on guide to Android forensics, from setting up the forensic workstation to analyzing key forensic artifacts

Rohit Tamma Donnie Tindall

[PACKT] PUBLISHING open source community experience distilled

Learning Android Forensics

A hands-on guide to Android forensics, from setting up the forensic workstation to analyzing key forensic artifacts

Rohit Tamma

Donnie Tindall

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Android Forensics

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2015

Production reference: 2280415

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78217-457-8

www.packtpub.com

Credits

Authors

Rohit Tamma

Donnie Tindall

Reviewers

Tom Anderson

Manish Chasta

Heather Mahalik

Gudipaty Laxmikant Pratap

Pujan P Shah

Vijay Kumar Velu

Commissioning Editor

Julian Ursell

Acquisition Editor

Rebecca Youé

Content Development Editor

Amey Varangaonkar

Technical Editor

Anushree Arun Tendulkar

Copy Editors

Karuna Narayanan

Vikrant Phadke

Aarti Saldanha

Project Coordinator

Suzanne Coutinho

Proofreaders

Safis Editing

Paul Hindle

Indexer

Tejal Soni

Graphics

Disha Haria

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Authors

Rohit Tamma is a security consultant working for a Fortune 500 company. With over 6 years of experience in the field of security, he is experienced in performing vulnerability assessments and penetration testing for web and mobile applications. He is currently focusing on mobile forensics on the Android platform. Rohit has also coauthored *Practical Mobile Forensics*, *Packt Publishing*. You can contact him at `tamma.rohit5@gmail.com` or on Twitter at `@RohitTamma`.

I would like to dedicate this book to my parents, my friends, and the countless number of people on whose shoulders we stand today.

Donnie Tindall is a digital forensics engineer at Dagger Networks, where he evaluates smartphone application security for various use cases and provides unique solutions to challenging forensic issues. Previously, he worked for Basis Technology, where he provided on-site mobile device forensics support for the U.S. government, including the development and teaching of mobile forensics courses to government and military users. Prior to that, he worked as a consultant for the FBI Terrorist Explosive Device Analytical Center, where he was responsible for handling mobile device forensics on media associated with improvised explosive devices. Donnie has performed thousands of mobile device extractions on Nokia, BlackBerry, Android, iPhone, and other devices. He is also an IACIS Certified Forensic Computer Examiner and instructor of FOR585, SANS Institute's smartphone forensics course. Donnie can be reached at `MobileForensicsResearch@gmail.com`.

First, I need to thank my wife, Amber, for putting up with me locking up myself in the office for hours at a time while writing this book. Also, thank you to my son, Dominic, for allowing me to use the computer long enough to get things done (without complaining—most of the time). And of course, thanks to my parents for helping me get where I am today.

A huge thanks goes to Heather "Hank" Mahalik and Lee "Leroy/Crogs" Crognale for thinking of me when this book was proposed. They have helped me immeasurably in my career. Much of what I wrote in my chapters would be wrong if it wasn't for the help of Dustin Fraze—bowing before your superior technical expertise is always a good idea, so thanks for trying to make me a little smarter. Jim Connor, thank you for the test devices that I could not buy myself and for answering the questions that were so dumb that Dustin ignored them. Finally, thanks to James Nuttall for his help with the mem RAM analysis tool in Chapter 5 and for showing me a few Android tricks along the way.

About the Reviewers

Tom Anderson is a software developer and forensic researcher with over 10 years' experience in mobile forensics. He currently works at NowSecure as the technical product manager for a forensics product and is also involved in security research.

I would like to thank my wife and my two beautiful daughters for their continued support and understanding. I would also like to thank my team at NowSecure for building an great forensics product.

Manish Chasta is a security researcher, analyst, author, speaker, and enthusiast, working with Indusface as the manager of managed security services. With an impressive career of over 9 years in mobile security, web application security, and cyber forensics, he has also spoken at prestigious security-related events and conferences.

Manish's vast experience includes auditing numerous mobile and web applications across the internet banking, core banking, finance, healthcare, CRM, telecom and e-commerce sectors. He has even authored numerous security-based articles. Over the years, many clients have benefitted from his training and workshop sessions on digital forensics, application security, and ethical hacking.

I would like to thank my beloved wife, Archana, my beautiful daughter, Mishi, my family, and teammates at Indusface for their encouragement and support that made it possible for me to review this book.

Heather Mahalik is a project manager and leads the forensic effort at Oceans Edge. She is the course leader for the FOR585 SANS Smartphone Forensics course and coauthors the Advanced Smartphone Forensics and Macintosh Forensics courses. With over 12 years of experience in digital forensics, she currently focuses her efforts on mobile device exploitation, forensic course development, instruction, and research on smartphone forensics.

Prior to joining Oceans Edge, Heather was the mobile exploitation team leader at Basis Technology, aiding the U.S. government. Previously, she worked at Stroz Friedberg and as a contractor for the U.S. Department of State, Computer Investigations and Forensics Lab. Heather earned her bachelor's degree from West Virginia University. She coauthored *Practical Mobile Forensics*, *Packt Publishing*, and has authored white papers and forensic course material. She has taught hundreds of courses worldwide to law enforcement, government, IT, e-discovery, and other forensic professionals, focusing on mobile devices and digital forensics.

My work on this book is dedicated to my husband and son. You are the two greatest men and the most important people in my life. Thank you for supporting me in my career and crazy endeavors!

Gudipaty Laxmikant Pratap is a digital forensics analyst, incident handler, cybercrime investigator, and smartphone forensics examiner. He has a master's degree (MS) in digital forensics and information assurance. He has expertise in imaging hard drives, flash drives, mobile devices, laptops, and desktops using hardware and software currently recognized and approved in the forensics field. He is proficient in automating forensic analysis using a wide array of tools currently recognized and approved by the court of law.

Laxmikant specializes in smartphone and BYOD forensics on the latest devices and mobile platforms, such as Android, Apple iOS, Windows, Symbian, and so on. He has conducted training sessions on cybercrime investigations, smartphone forensics, and incident response for investigating officials of law enforcement agencies (Anti Corruption Bureau, Central Detective Training School, Central Crime Station, Forensic Sciences Laboratory, Crime Investigation Department, the income tax investigation department, and a state police department) and corporate entities across India.

Laxmikant is an avid and passionate researcher on new artifacts. He likes developing automated solutions for real-world forensic challenges.

I would like to express my special gratitude to Packt Publishing, who gave me the golden opportunity to be a part of a book on a topic that is very dear to me. Reviewing this book gave me immense pleasure. It also helped me learn so many things that were otherwise not known to me. My heartfelt thanks to my fellow reviewers, authors, and the entire panel associated with this book. I would like to acknowledge the people who mean the world to me: mom, dad, Ganesh, and Chaitanya. I consider myself the luckiest person in the world to have such a supportive family, standing by me with their love and support.

Pujan P Shah has been active in the field of information technology since 2007. During his learning, he acquired knowledge and experience of working on Windows, Macintosh, Unix, networking, programming, malware analysis, cloud security, digital forensics, and incident response. He has done his master's degree in digital forensics and information assurance. While pursuing this degree, he completed his training from the computer forensics division at the Directorate of Forensic Science, Gujarat, India.

Currently, Pujan is working as a digital forensics researcher and security analyst with companies giving training and services on cyber security and forensics to law enforcement and corporate agencies. In the past, he worked on some major research projects related to malware analysis, cloud forensics, cryptography, steganography, network security, and forensics of the virtual environment. He also has experience of over 5 years in using programming languages such as C, Shell scripts, Python, Java, and VB.Net. He has published an article called *A Digital Forensics Case Study Using Autopsy* in the *eforensics* magazine.

A person can achieve his dreams only when there is support of his or her family. I am lucky to have a family that supported and facilitated my pursuit of academic activities, overlooking other responsibilities. Many thanks to all of them: my mom, dad, brother, and teachers. My father's suggestion made me rethink, change, and remold my career. I acknowledge his valuable contribution. I owe my accomplishment to my family. Finally, I would like to thank my mother for being my constant source of motivation and my friends who had always helped and assisted me with the programming.

Vijay Kumar Velu is a passionate information security practitioner currently working as technical manager in KPMG Global Services, based in India. He has more than 8 years of IT industry experience, is a licensed penetration tester, and specializes in providing technical solutions to a variety of cyber problems. He holds multiple security qualifications, including certified ethical hacker, EC-council-certified security analyst, and computer hacking forensics investigator.

Vijay has helped clients assess threats and vulnerabilities through penetration testing, web application security assessments, social engineering, technical security diagnostic reviews, network and application architecture reviews, and gap assessments with regulatory standards across the banking, telecom, retail, government, services, and insurance sectors. Vijay has also lead security investigations associated with sophisticated cyber intrusions. He has helped clients detect, investigate, and respond to attacks believed to be orchestrated by transnational criminal enterprises and state-sponsored hackers.

Vijay was invited to be a speaker at the Open Cloud Conference held in Bangalore, and he has also delivered multiple guest lectures/training on the importance of information security at various business schools in India.

I would like to thank my family, friends (HACKERZ), my mentor, Lokesh Gowda, and my team at the workplace.

---

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface **ix**

Chapter 1: Introducing Android Forensics **1**

Mobile forensics **3**

The mobile forensics approach **4**

Investigation Preparation 4

Seizure and Isolation 4

Acquisition 8

Examination and Analysis 9

Reporting 9

Challenges in mobile forensics **10**

The Android architecture **12**

The Linux kernel 13

Libraries 13

Dalvik virtual machine 14

The application framework 15

The applications layer 16

Android security **16**

Security at OS level through Linux kernel 17

Permission model 18

Application sandboxing 19

SELinux in Android 20

Application Signing 21

Secure interprocess communication 22

Android hardware components **23**

Core components 23

Central processing unit 23

Baseband processor 24

Memory 24

SD Card 25

Display 25
Battery 26
Android boot process 28
Boot ROM code execution 28
The boot loader 29
The Linux kernel 30
The init process 31
Zygote and Dalvik 32
System server 34
Summary 35
Chapter 2: Setting Up an Android Forensic Environment 37
Android forensic setup 37
The Android SDK 38
Installing the Android SDK 38
Android Virtual Device 41
Connecting and accessing an Android device from the workstation 44
Identifying the device cable 45
Installing device drivers 45
Accessing the device 46
Android Debug Bridge 49
Using adb to access the device 50
Detecting a connected device 50
Directing commands to a specific device 51
Issuing shell commands 51
Basic Linux commands 52
Installing an application 55
Pulling data from the device 55
Pushing data to the device 56
Restarting the adb server 56
Viewing log data 57
Rooting Android 59
What is rooting? 59
Why root? 60
Recovery and fastboot 61
Recovery mode 61
Fastboot mode 64
Locked and unlocked boot loaders 64
How to root? 65
Rooting an unlocked boot loader 66
Rooting a locked boot loader 68
ADB on a rooted device 68
Summary 70

Chapter 3: Understanding Data Storage on Android Devices 71
Android partition layout 71
Common partitions in Android 71
boot loader 72
boot 72
recovery 72
userdata 72
system 72
cache 72
radio 72
Identifying partition layout 73
Android file hierarchy 74
An overview of directories 75
acct 75
cache 75
d 75
data 76
dev 78
init 78
mnt 78
proc 78
root 79
sbin 79
misc 79
sdcard 80
system 81
ueventd.goldfish.rc and ueventd.rc 84
Application data storage on the device 85
Shared preferences 86
Internal storage 87
External storage 90
SQLite database 90
Network 91
Android File system overview 91
Viewing filesystems on an Android device 92
Common Android filesystems 93
Flash memory filesystems 93
Media-based filesystems 94
Pseudo filesystems 95
Summary 98
Chapter 4: Extracting Data Logically from Android Devices 99
Logical extraction overview 99
What data can be recovered logically? 100
Root access 100

Manual ADB data extraction **101**
USB debugging 101
Using ADB shell to determine if a device is rooted 103
ADB pull 104
Recovery mode 107
Fastboot mode 113
Determining bootloader status 113
Booting to a custom recovery image 115
ADB backup extractions **117**
Extracting a backup over ADB 117
Parsing ADB backups 119
Data locations within ADB backups 120
ADB Dumpsys **123**
Dumpsys batterystats 124
Dumpsys procstats 125
Dumpsys user 125
Dumpsys App Ops 126
Dumpsys Wi-Fi 127
Dumpsys notification 127
Dumpsys conclusions 128
Bypassing Android lock screens **128**
Lock screen types 128
None/Slide lock screens 129
Pattern lock screens 129
Password/PIN lock screens 129
Smart Locks 129
General bypass information 130
Cracking an Android pattern lock **132**
Cracking an Android PIN/Password 136
Android SIM card extractions **137**
Acquiring SIM card data 138
SIM Security 140
SIM cloning 141
Issues and opportunities with Android Lollipop **141**
Summary **143**
Chapter 5: Extracting Data Physically from Android Devices **145**
Physical extraction overview **145**
What data can be acquired physically? 146
Root access 146

Extracting data physically with dd **147**
Determining what to image 148
Writing to an SD card 150
Writing directly to an examiner's computer with netcat 151
Installing netcat on the device 151
Using netcat 152
Extracting data physically with nanddump **153**
Verifying a full physical image 154
Analyzing a full physical image **154**
Autopsy 155
Issues with analyzing physical dumps 158
Imaging and analyzing Android RAM **159**
What can be found in RAM? 160
Imaging RAM with LiME 160
Imaging RAM with mem 162
Output from mem 164
Acquiring Android SD cards **164**
What can be found on an SD card? 165
SD card security 166
Advanced forensic methods **166**
JTAG 166
Chip-off 168
Bypassing Android full-disk encryption 169
Summary **170**
Chapter 6: Recovering Deleted Data from an Android Device **173**
An overview of data recovery **173**
How can deleted files be recovered? 174
Recovering data deleted from an SD card **175**
Recovering data deleted from internal memory **180**
Recovering deleted data by parsing SQLite files 181
Recovering deleted data through file carving techniques 183
Analyzing backups **186**
Summary **189**
Chapter 7: Forensic Analysis of Android Applications **191**
Application analysis **192**
Why do app analysis? 193
The layout of this chapter 193
Determining what apps are installed **194**
Understanding Linux epoch time 194

Wi-Fi analysis **196**
Contacts/call analysis **196**
SMS/MMS analysis **198**
User dictionary analysis **200**
Gmail analysis **201**
Google Chrome analysis **202**
Decoding the WebKit time format 206
Google Maps analysis **207**
Google Hangouts analysis **208**
Google Keep analysis **210**
Converting a Julian date 211
Google Plus analysis **211**
Facebook analysis **213**
Facebook Messenger analysis **217**
Skype analysis **220**
Recovering video messages from Skype 223
Snapchat analysis **224**
Viber analysis **225**
Tango analysis **227**
Decoding Tango messages 230
WhatsApp analysis **234**
Decrypting WhatsApp backups 236
Kik analysis **237**
WeChat analysis **238**
Decrypting the WeChat EnMicroMsg.db database 240
Application reverse engineering **242**
Obtaining the application's APK file 242
Disassembling an APK file 244
Determining an application's permissions 246
Viewing the application's code 246
Summary **247**

Chapter 8: Android Forensic Tools Overview **249**

ViaExtract **249**
Backup extraction with ViaExtract 252
Logical extraction with ViaExtract 254
Examining data in ViaExtract 258
Other tools within ViaExtract 260

Autopsy **261**
Creating a case in Autopsy 261
Analyzing data in Autopsy 266
ViaLab Community Edition **272**
Setting up the emulator in ViaLab 273
Installing an application on the emulator 277
Analyzing data with ViaLab 280
Summary **282**
Conclusion **282**
Index **283**

Preface

Learning Android Forensics uses free open source tools to show you how to forensically recover data from Android devices. All of you, from beginners to experts, are encouraged to follow along with step-by-step directions to learn how to acquire and examine evidence and gain a deeper understanding of the Android forensic process. Commercial forensic tools typically give an examiner a button to press (commonly called the Find Evidence button). This book goes behind the scenes and shows what many of these tools are actually doing, giving you much deeper knowledge of how they work. Commercial forensic tools also frequently fail to recover data from third-party applications; there are simply too many apps available to write a tool that covers all of them. This book shows you how to manually analyze over a dozen popular applications. It teaches techniques and procedures for understanding data that can be carried over to analyzing almost any other application.

What this book covers

Chapter 1, *Introducing Android Forensics*, introduces mobile forensics, the general approach, and the challenges faced. This chapter also provides an overview of the Android architecture, security features, boot process, and so on.

Chapter 2, *Setting Up an Android Forensic Environment*, covers the steps to perform to get an established forensic setup to examine Android devices. This chapter also explains the use of ADB commands on the Android device.

Chapter 3, *Understanding Data Storage on Android Devices*, provides a detailed explanation of what kind of data is stored in the device, where it is stored, how it is stored, and details of the filesystems in which it is stored.

Chapter 4, *Extracting Data Logically from Android Devices*, covers various logical data extraction techniques using free and open source tools. The logical methods covered include ADB pull, ADB backup, ADB dumpsys information, and SIM card extractions. Bypassing device lock screens is also covered.

Chapter 5, *Extracting Data Physically from Android Devices*, demonstrates various physical data extraction techniques. Physical methods include dd and nanddump, as well as using netcat to write data to the examiner's computer. RAM and SD card imaging is also covered.

Chapter 6, *Recovering Deleted Data from an Android Device*, provides an overview on recovering data deleted from an Android device. This chapter explains procedures to recover data deleted from an SD card and also from a phone's internal storage.

Chapter 7, *Forensic Analysis of Android Applications*, covers forensic analysis of Android applications, data obfuscation methods used by popular applications, reverse engineering of Android applications, and the methods required for it.

Chapter 8, *Android Forensic Tools Overview*, explains various open source and commercial tools that are helpful during forensic analysis of Android devices.

What you need for this book

This book covers various forensic approaches and techniques on Android devices. The content is organized in a manner that allows any user to examine an Android device and perform forensic investigation. No prerequisite knowledge is needed because all the topics are explained, from basic to in-depth. Knowledge of mobile platforms, especially Android, will definitely be an advantage. Wherever possible, the steps required to perform various forensic activities using tools are explained in detail.

Who this book is for

This book is intended for forensic examiners with little or basic experience in mobile forensics on the Android platform. It will also be useful to computer security professionals, researchers, and anyone seeking a deeper understanding of Android mobile internals. Finally, this book will come in handy for those who are trying to recover accidentally deleted data (photos, contacts, SMS, and more) from an Android device.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Until Android 4.4, all apps present under `/system` were treated equally."

A block of code is set as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.rohit">

  <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

Any command-line input or output is written as follows:

```
shell@android:/ $ cat default.prop
cat default.prop
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
ro.allow.mock.location=0
ro.debuggable=0
persist.sys.usb.config=mtp
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "In Android devices, this option is usually found by navigating to **Settings** | **Developer options**."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

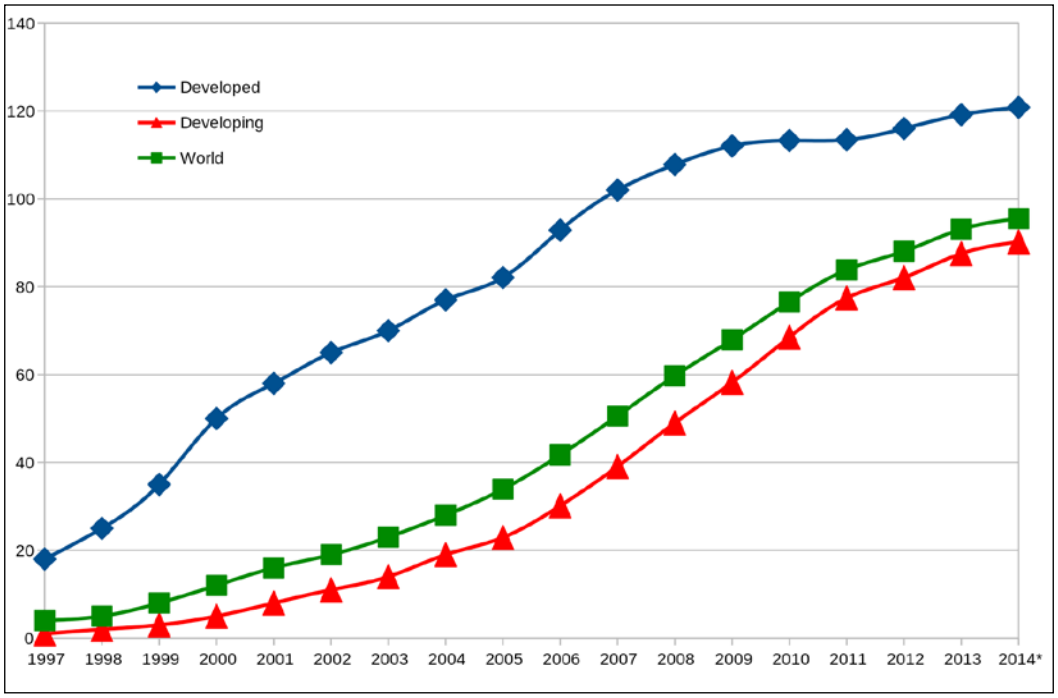
1
Introducing Android Forensics

Mobile forensics is a branch of digital forensics which is evolving in today's digital era. Android forensics deals with extracting, recovering and analyzing the data present on an Android device through various techniques. However, it is important to have a clear understanding of the platform and other fundamentals before we dive in and find out how to extract data. In this chapter, we will cover the following topics:

- Mobile forensics
- Mobile forensics approach
- Android architecture
- Android security
- Android hardware components
- Android boot process

The world today is experiencing technological innovation like never before. This growth is almost exponential in the field of mobile devices. Gartner, a technology research and advisory firm, in their forecasts published in June 2014, predicted that mobile phone shipments are soon set to break 2.4 billion units. This statistic alone reflects the unprecedented growth of mobile devices. Mobile phones have not only increased in number but also have become more sophisticated in terms of functionality.

The following screenshot referenced from `http://en.wikipedia.org/wiki/File:Mobile_phone_subscribers_1997-2014_ITU.svg` shows the increase in graph of mobile phone subscribers per 100 inhabitants from 1997 to 2014:



Mobile phone subscribers per 100 inhabitants from 1997-2014

Within mobile phones, smart phones are very much becoming the norm. Improvements in the computing power and data storage of these devices enable us to perform a wide range of activities. We are increasingly becoming dependent on these mobile devices for most of our activities. Apart from performing routine tasks such as making calls, sending messages, and so on, these devices also support other activities such as sending e-mails, surfing the Internet, recording videos, creating and storing documents, identifying locations with **Global Positioning System** (**GPS**) services, managing business tasks, and much more. In other words, mobile devices are now a repository of sensitive personal information, containing a wealth of user data. Quite often, the data sitting on a device is more valuable than the device itself. For instance, calls made from a device could be valuable information for law enforcement agencies. The fact that mobile forensics played a crucial role in solving high-profile cases, such as the 2010 Times Square car bombing attempt and the Boston marathon bombings, reaffirms the increasing role of mobile forensics in many government and law enforcement cases.

Mobile forensics

Mobile device forensics is a branch of digital forensics which deals with extracting, recovering and analyzing digital evidence or data from a mobile device under forensically sound conditions. Simply put, it deals with accessing the data stored on devices which includes SMS, contacts, call records, photos, videos, documents, application files, browsing history and so on, and also recovering data deleted from devices using various forensic techniques. It is important that the process of recovering or accessing details from a device is forensically sound, if it has to be admitted in a court of law and to maintain the integrity of the evidence. If the evidence has to be admitted in a court of law, it is important that the original device is not tampered with.



The term *forensically sound* is often used in the digital forensics community to clarify the correct use of a particular forensic technology or methodology. Mobile forensics, especially Android forensics, is evolving fast, owing to the fact that it has a market share of 84 percent (as per market research firm IDC).

As explained by Eoghan Casey in his book *Digital Forensics and Investigation*, forensic soundness is not just about keeping original evidence unaltered. Even the routine task of acquiring data from a hard drive using a hardware write-blocker may cause alterations (for example, making a hidden area of the hard drive accessible) on the drive. One of the keys to forensic soundness is documentation. Documenting how the device is handled from the beginning is very important. Hence, an investigation can be considered forensically sound if the acquisition process preserves the original data and its authenticity and integrity can be validated. Evidence integrity checks ensure that the evidence has not been tampered with from the time it was collected. Integrity checks are done by comparing the digital fingerprint of the evidence taken at the time of collection with the digital fingerprint of the evidence in current state.

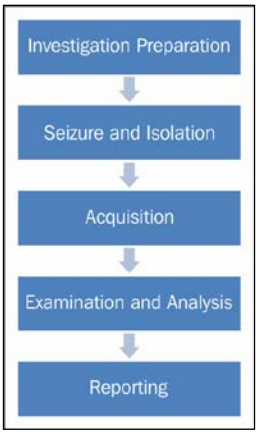
There is a growing need for mobile forensics due to several reasons. Some of the prominent reasons are:

- Use of mobile phones to store personal information
- Increased use of mobile phones to perform online activity
- Use of mobile phones in several crimes

Mobile forensics on a particular device is primarily dependent on the underlying operating systems. Thus we have different fields such as Android forensics, iOS forensics, Blackberry forensics, and so on.

The mobile forensics approach

Once the data is extracted from a device, different methods of analysis are used based on the underlying case. As each investigation is distinct, it is not possible to have a single definitive procedure for all cases. However, the overall process can be broken into five phases as shown in the following diagram:



Phases in mobile forensics

The following section discusses each phase in detail:

Investigation Preparation

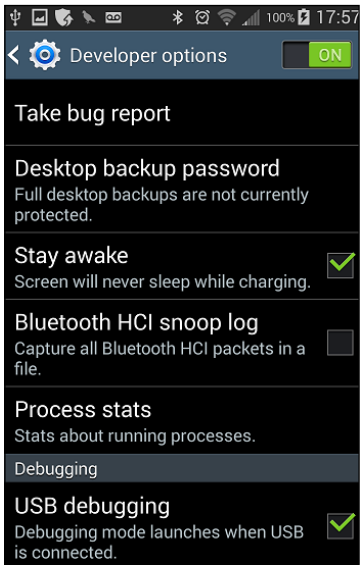
This phase begins when a request for examination is received. It involves preparing all of the paperwork and forms required to document the chain of custody, ownership information, the device model, its purpose, the information that the requestor is seeking, and so on. The chain of custody refers to the chronological documentation or paper trail, showing the seizure, custody, control, transfer, analysis, and disposition of physical or electronic evidence. From the details submitted by the requestor, it's important to have a clear understanding of the objective for each examination.

Seizure and Isolation

Handling the device during seizure is one of the important steps while performing forensic analysis. The evidence is usually transported using anti-static bags which are designed to protect electronic components against damages produced by static electricity. As soon as the device is seized, care should be taken to make sure that our actions don't result in any data modification on the device. At the same time, any opportunity that can aid the investigation should also not be missed.

Following are some of the points that need to be considered while handling an Android device during this phase:

- With increasing user awareness on security and privacy, most of the devices now have screen lock enabled. During the time of seizure, if there is a chance to do so, disable the passcode. Some devices do not ask the user to re-enter the passcode while disabling the lock screen option.
- If the device is unlocked, try to change the settings of the device to allow greater access to the device. Some of the settings that can be considered to achieve this are as follows:
    - **Enable USB debugging**: Enabling this option gives greater access to the device through **Android debug bridge** (**adb**) connection. We are going to cover adb connection in detail in *Chapter 2*, *Setting Up Android Forensic Environment*. This will greatly aid the forensic investigator during the data extraction process. In Android devices, this option is usually found under **Settings** | **Developer options**, as shown in the following screenshot. In later Android versions starting from 4.2, the developer options are hidden by default. To enable them, navigate to **Settings** | **About Phone** and tap on **Build number** 7 times.
    - **Enable stay awake setting**: Enabling this option and charging the device will make the device stay awake which means that, it doesn't get locked. In Android devices, this option is usually found under **Settings** | **Developer options**, as shown in the following screenshot:

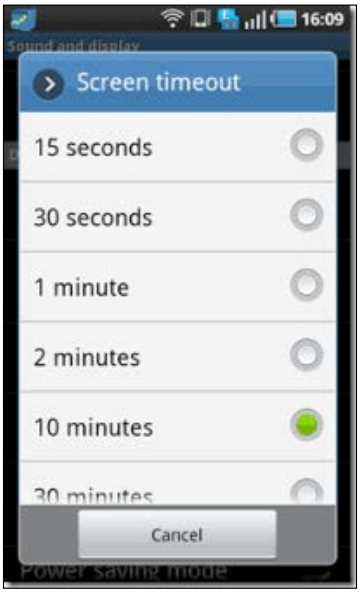


Stay awake and USB debugging options

- **Increase Screen timeout**: This is the time for which the device will be active once it is unlocked. Depending on the device model, this time can be set up to 30 minutes. In most devices, it can be accessed under **Settings** | **Display** | **Screen timeout**, as shown in the following screenshot:



Please note that the location to access this item changes across different versions and models of Android phones.



Screen timeout option on an Android device

In mobile forensics, it is of critical importance to protect the seized device so that our interaction with the evidence (or for that matter, an attacker's attempt to remotely interact with the device) does not change the evidence. In computer forensics, we have software and hardware write blockers that can perform this function. But in mobile forensics, since we need to interact with the device to pull the data, these write blockers are not of any use. Another important aspect is that we also need to prevent the device from interacting with wireless radio networks. As mentioned earlier, there is a high probability that an attacker can issue remote wipe commands to delete all data, including e-mails, applications, photos, contacts, and other files on the device.

The **Android Device Manager** (**ADM**) and several other third-party apps allow the phone to be remotely wiped or locked. This can be done by signing into the Google account that is configured on the mobile device. Using this software, an attacker can also locate the device, which could pose a security risk. For all these reasons, isolating the device from all communication sources is very important.



Have you thought about remote wipe options that do not require internet access? **Mobile Device Management** (**MDM**) software provides a remote wipe feature just by sending an SMS. Isolating the device from all communication options is crucial.

To isolate the device from a network, we can put the device in **Airplane mode** if there is access to the device. Airplane mode disables a device's wireless transmission functions, such as cellular radio, Wi-Fi, and Bluetooth. However, this may not always be possible because most of the devices are screen-locked. Also, as Wi-Fi is now available in airplanes, some devices now allow Wi-Fi access in Airplane mode. Hence, an alternate solution would be to use a Faraday bag or RF isolation box, as both effectively block signals to and from the mobile phone. But, one concern with these isolation methods however, is that once they're employed, it is difficult to work with the phone because you cannot see through them to use the touch screen or keypad. For this reason, Faraday tents and rooms exist, as shown in the following screenshot (taken from `http://www.technicalprotection.co.uk/`), but are very expensive.



Pyramid-shaped Faraday tent

Even after taking all these precautions, certain automatic functions, such as alarms can trigger. If such a situation is encountered, it must be properly documented.

Acquisition

The acquisition phase refers to the extraction of data from the device. Due to the inherent security features of mobile devices, extracting data is not always straight forward. Depending on the operating system, make, and model of the device, the extraction method is decided. The following types of acquisition methods can be used to extract data from a device:

- **Manual acquisition**: This is the simplest of all acquisition methods. The examiner uses the user interface of the phone to browse and investigate. No special tools or techniques are required here, but the limitation is that only those files and data that are visible through a normal user interface can be extracted. Data extracted through other methods can also be verified using this.
- **Logical acquisition**: This is also called logical extraction. This generally refers to extracting the files that are present on a logical store such as a filesystem partition. This involves obtaining data types, such as text messages, call history, pictures and so on, from a phone. The logical extraction technique works by using the original equipment manufacturer's APIs for synchronizing the phone's contents with a computer. This technique usually involves extracting the following evidence:
    - Call Logs
    - SMS
    - MMS
    - Browser history
    - People
    - Contact methods
    - Contacts extensions
    - Contacts groups
    - Contacts phones
    - Contacts setting
    - External image media (metadata)
    - External image thumbnail media (metadata)
    - External media, audio, and misc. (metadata)
    - External videos (meta data)

    - MMSParts (includes full images sent via MMS)
    - Location details (GPS data)
    - Internet activity
    - Organizations
    - List of all applications installed, along with their version
    - Social networking apps data such as WhatsApp, Skype, Facebook, and so on.

- **Filesystem acquisition**: This is a logical procedure and generally refers to the extraction of a full file system from a mobile device. File system acquisition can sometimes help in recovering deleted contents (stored in SQLite files) that are deleted from the device.
- **Physical acquisition**: This involves making a bit-by-bit copy of the entire flash memory. The data extracted using this method is usually in the form of raw data (as a hexadecimal dump), which can then be further parsed to obtain file system information or human readable data. Since all investigations are performed on this image, this process also ensures that original evidence is not altered.

Examination and Analysis

In this phase, different software tools are used to extract the data from the memory image. In addition to these tools, an investigator would also need the help of a hex editor, as tools do not always extract all the data. There is no single tool that can be used in all cases. Hence, examination and analysis requires a sound knowledge of various file systems, file headers, and so on.

Reporting

Documentation of the examination should be done throughout the process, noting down what was done in each phase. The following points might be documented by an examiner:

- Date and time the examination started
- Physical condition of the phone
- The status of the phone when received (ON/OFF)
- Make, model, and operating system of the phone
- Pictures of the phone and individual components
- Tools used during the investigation
- Data documented during the examination

The data extracted from the mobile device should be clearly presented to the recipient so that it can be imported into other software for further analysis. In the case of civil or criminal cases, wherever possible, pictures of data, as it existed on the cellular phone, should be collected, as they can be visually compelling to a jury.

Challenges in mobile forensics

With the increased usage of Android devices and the wider array of communication platforms that they support, demand for forensic examination has automatically grown. While working with mobile devices, forensic analysts face a number of challenges. The following points shed light on some of the mobile forensics challenges faced today:

- **Preventing data alteration on the device**: One of the fundamental rules to remember in forensics is to not modify the evidence. In other words, the forensic techniques that are applied to a device to extract any information, should not alter the data present on the device. But this is not practical with respect to mobile forensics because simply switching ON a device might also change certain state variables that are present on the device. With mobile devices, background processes always run and a sudden transition from one state to another can result in the loss or modification of data. Therefore, there is a chance that data may be altered either intentionally or unintentionally by the forensic analyst. In addition to this, there is a high possibility that an attacker can remotely change or delete the content present on the device. As mobile phones use different communication channels (cellular, Wi-Fi, Bluetooth, infrared, and so on) the possibility of communicating through them should be eliminated. Features such as remote data wiping would enable an attacker to remotely wipe the entire device just by sending an SMS or by simply pressing a button that sends a wipe request to the Android device. Unlike computer forensics, mobile device forensics requires more than just isolating the device from the network.

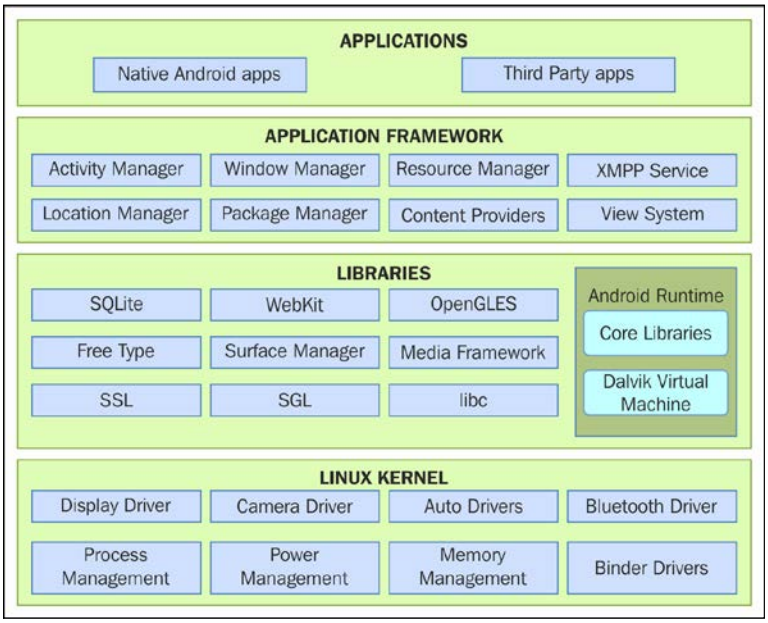
- **Wide range of operating systems and device models**: The wide range of mobile operating systems available in the market makes the life of a forensic analyst more difficult. Although Android is the most dominant operating system in the mobile world, there are mobile devices which run on other operating systems, including iOS, Blackberry, Windows, and so on, which are often encountered during investigations. Also for a given operating system, there are millions of mobile devices available that differ in OS versions, hardware, and various other features. For example, within the Android operating system, there are around 10 versions, and for each version, there are different customizations made by different manufacturers. Based on the manufacturer, the approach to acquiring forensic artifacts changes. To remain competitive, manufacturers release new models and updates so rapidly that it's hard to keep track of all of them. Sometimes within the same operating system the data storage options and file structures also change, making it even more difficult. There is no single tool that can work on all the available types of mobile operating systems. Therefore, it is crucial for forensic analysts to remain updated on all the latest changes and techniques.
- **Inherent security features**: As the concept of "privacy" is increasingly gaining importance, mobile manufacturers are moving towards implementing robust security controls on devices, which complicates the process of gaining access to the data. For example, if the device is passcode protected, the forensic investigator has to first find a way to bypass the passcode. Similarly, full disk encryption mechanisms that are implemented on some of the latest devices prevent law enforcement agencies and forensic analysts from accessing the information on the device. Apple's iPhone encrypts all the data present on the device by default, using hardware keys built into the device. It is very difficult for an examiner to break these encryption mechanisms using techniques such as brute force.
- **Legal issues**: Mobile devices can be involved in crimes that span across the globe and can cross geographical boundaries. In order to tackle these multijurisdictional issues, the forensic examiner needs to be aware of the nature of the crime and also regional laws.

The Android architecture

Before we proceed with the internals of Android forensics, this section introduces you to Android as an operating system and covers various fundamental concepts that need to be understood to gain experience in the area of forensics.

Any operating system (desktop or mobile) takes responsibility for managing the resources of the system and provides a way for applications to talk to hardware or physical components in order to accomplish certain tasks. The Android operating system is no different. It powers mobile phones, manages memory and processes, enforces security, takes care of networking issues, and so on. Android is open source and most of the code is released under the Apache 2.0 license. Practically, this means that mobile phone device manufacturers can access it freely, modify it, and use the software according to the requirements of any device. This is one of the primary reasons for its popularity.

The Android operating system consists of a stack of layers running on top of each other. Android architecture can be best understood by taking a look at what these layers are and what they do. The following diagram referenced from `http://elinux.org/images/c/c2/Android-system-architecture.jpg`, shows the various layers involved in the Android software stack:



Android architecture

Android architecture is in the form of a software stack comprising kernel, libraries, runtime environment, applications, middleware, and services. Each layer of the stack (and also elements within each layer) is integrated in a way that provides an optimal execution environment for mobile devices. The following sections focus on the different layers of the Android stack, starting at the bottom with the Linux kernel.

The Linux kernel

Android OS is built on top of the Linux kernel with some architectural changes made by Google. Linux was chosen as it is a portable platform that can be compiled easily on different hardware. The Linux kernel is positioned at the bottom of the software stack and provides a level of abstraction between the device hardware and the upper layers. It also acts as an abstraction layer between the software and hardware present on the device. To understand this better, consider the case of a camera click. What actually happens when you click a photo using the camera button on your mobile device? At some point, the hardware instruction, such as *pressing a button*, has to be converted to a software instruction such as *to take a picture and store it in the gallery*. The kernel contains drivers which can facilitate this process. When the camera button click is detected, the instruction goes to the corresponding driver in the kernel, which sends the necessary commands to the camera hardware, similar to what occurs when a key is pressed on a keyboard. In simple terms, the drivers in the kernel control the underlying hardware. As shown in the preceding figure, the kernel contains drivers related to Wi-Fi, Bluetooth, USB, audio, display, and so on.

All the core functionalities of Android, such as process management, memory management, security, and networking, are managed by Linux kernel. Linux is a proven platform when it comes to security and process management. Android has taken leverage of the existing Linux open source OS to build a solid foundation for its ecosystem. Each version of Android has a different version of the underlying Linux kernel. As of September 2014, the current Android version 4.2 is built upon Linux kernel 3.4 or newer, but the specific kernel version depends on the actual Android device and chipset.

Libraries

On top of Linux kernel are Android's native libraries. It is with the help of these libraries that the device handles different types of data. For example, the media framework library supports the recording and playback of audio, video and picture formats. These libraries are written in the C or C++ programming languages and are specific to a particular hardware. Surface Manager, Media framework, SQLite, WebKit, OpenGL, and so on are some of the most important native libraries.

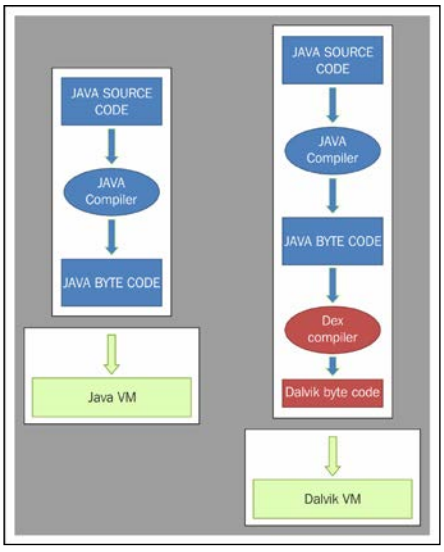
Dalvik virtual machine

Android applications are programmed using the Java programming language. The main reason for choosing Java is because it's a well-known language and has a massive developer base. Android wanted to take advantage of this existing developer community, rather than coming up with a new language.



This later prompted Oracle to file a case in court against Google claiming that its copyrights and patents were violated. But the jury finally declared that Google did not infringe on Oracle's patents, and the trial judge ruled that the structure of the Java APIs used by Google was not copyrightable.

When a Java program is compiled, we get byte code. A **Java virtual machine** (**JVM**) (a virtual machine is an application that acts as an operating system) can execute this byte code. In the case of Android, this Java byte code is further converted to **Dalvik** byte code by the dex compiler. This Dalvik byte code is then fed into **Dalvik virtual machine** (**DVM**) which can read and use the code. Thus, the `.class` files from the Java compiler are converted to `.dex` files using the dx tool. Dalvik byte code is an optimized byte code suitable for low-memory and low-processing environments. Also, note that JVM's byte code consists of one or more `.class` files, depending on the number of Java files that are present in an application, but Dalvik byte code is composed of only one `.dex` file. Each Android application runs its own instance of the DVM. The following diagram shows the difference between the program compilation of a Java application and an Android application.



JVM vs DVM

Since Android 5.0, Dalvik has been replaced by **Android Run Time** (**ART**) as the platform default. The ART was introduced in Android 4.4 on an experimental basis. Dalvik uses **just-in-time** (**JIT**) compilation which compiles the byte code every time an application is launched. However ART uses **ahead-of-time** (**AOT**) compilation by performing it upon the installation of an application. This greatly reduces the mobile device's processor usage, as the overall compilation during the operation of an application is reduced.

The application framework

Android applications are run and managed with the help of an Android application framework. It is responsible for performing many crucial functions such as resource management, handling calls, and so on. The Android framework includes the following key services, referenced from `http://fp.edu.gva.es/av/pluginfile.php/745396/mod_imscp/content/2/1_overview_of_the_android_architecture.html`:

- **Activity manager**: This service controls all aspects of the application lifecycle and activity stack.
- **Content providers**: This service allows applications to publish and share data with other applications.
- **Resource manager**: This service provides access to non-code embedded resources such as strings, color settings, and user interface layouts.
- **Notifications manager**: This service allows applications to display alerts and notifications to the user.
- **View system**: This service provides an extensible set of views used to create application user interfaces.
- **Package manager**: The system by which applications are able to find out information about other applications currently installed on the device.
- **Telephony manager**: This service provides information to the application about the telephony services available on the device such as status and subscriber information.
- **Location manager**: This service provides access to the location services allowing an application to receive updates about location changes.

The applications layer

The topmost layer in the Android stack consists of applications (called **apps**) which are programs that users directly interact with. There are two kinds of apps discussed as follows:

- **System apps**: These are applications that are pre-installed on the phone and are shipped along with the phone. Applications such as default browser, e-mail client, contacts, and so on, are examples for system apps. These cannot be uninstalled or changed by the user as they are read-only on production devices. These are usually present mounted in the `/system` directory. Until Android 4.4, all apps present under `/system` were treated equally. But from Android 4.4 onward, apps installed in `/system/priv-app/` are treated as privileged applications and are granted permissions with protection level signatureOrSystem to only privileged apps.
- **User-installed apps**: These are the applications that are downloaded and installed by the user from various distribution platforms such as Google Play. Google Play is the official app store for the Android operating system, where users can browse and download the applications. Based on December 2014 statistics from AppBrain, there are around 1,418,453 Android apps in the Play Store. These apps are presently found in the `/data` directory. More information about how security is enforced between them is discussed in the following sections.

Android security

Android as a platform has certain features built into its architecture that ensure the security of users, applications, and data. Although they help in protecting the data, these security features sometimes prevent investigators from gaining access to necessary data. Hence, from a forensic perspective, it is first important to understand the inherent security features so that a clear idea is established about what can or cannot be accessed under normal circumstances. The security features and offerings that are incorporated aim to achieve three things:

- To protect user data
- To protect system resources
- To make sure that one application cannot access the data of another application

The next sections provide an overview of the key security features in the Android operating system.

Security at OS level through Linux kernel

The Android operating system is built on top of the Linux kernel. Over the past few years, Linux has evolved into a secure operating system trusted by many corporations across the world for its security. Today, most of the mission critical systems and servers run on Linux because of its security. By having the Linux kernel at the heart of its platform, Android tries to ensure security at the OS level. Also, Android has built a lot of specific code into Linux to include certain features related to mobile environment. With each Android release the kernel version also has changed. The following table shows Android versions and the corresponding Linux kernel version:

Android Version	Linux Kernel Version
1.0	2.6.25
1.5	2.6.27
1.6	2.6.29
2.2	2.6.32
2.3	2.6.35
3.0	2.6.36
4.0	3.0.1
4.1	3.0.31
4.2	3.4.0
4.3	3.4.39
4.4	3.8

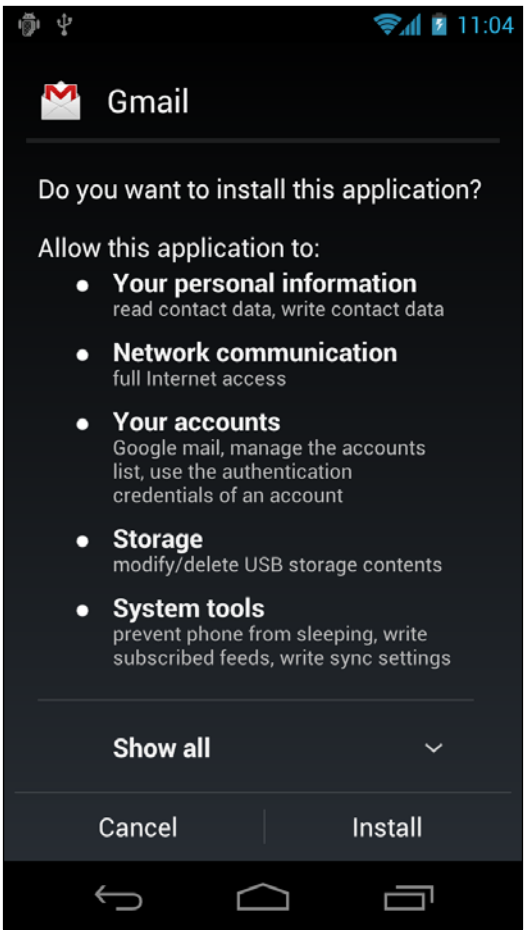
Linux kernel used in various Android versions

The Linux kernel provides Android with the below key security features:

- A user-based permissions model
- Process isolation
- Extensible mechanism for secure IPC

Permission model

Android implements a permission model for individual apps. Applications must declare which permissions (in the manifest file) they require. When the application is installed, as shown in the following screenshot, Android will present the list to the user so that they can view the list to allow installation or not:



Sample permission model in Android

Unlike a desktop environment, this provides an opportunity for the user to know in advance which resources the application is seeking access to. In other words, user permission is a must to access any kind of critical resource on the device. By looking at the requested permission, the user is more aware of the risks involved in installing the application. But most users do not read these and just give away a lot of permissions, exposing the device to malicious activities.



It is not possible to install an Android app with a few or reduced permissions. You can either install the app with all the permissions or decline it.

As mentioned earlier, developers have to mention permissions in a file named `AndroidManifest.xml`. For example, if the application needs to access the Internet, the permission `INTERNET` is specified using the following code in the `AndroidManifest.xml` file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.example.rohit">

  <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

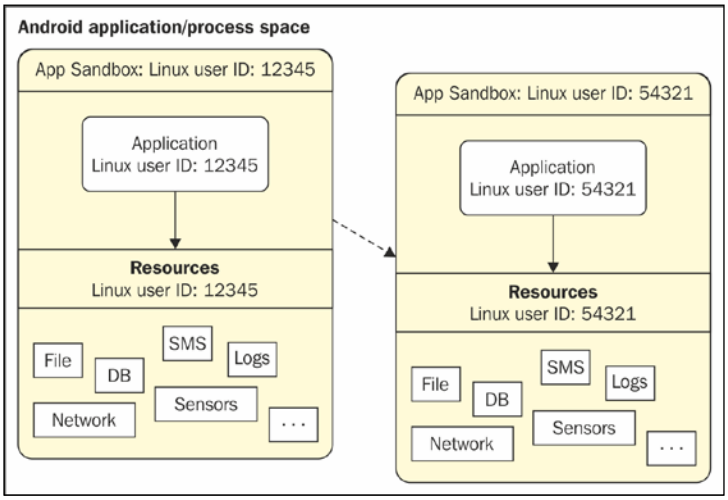
Android permissions are categorized into four levels which are as follows:

Permission Type	Description
Normal	This is the default value. These are low risk permissions and do not pose a risk to other applications, system or user. This permission is automatically granted to the app without asking for user approval during installation.
Dangerous	These are the permissions that can cause harm to the system and other applications. Hence, user approval is necessary during installation.
Signature	These are automatically granted to a requesting app if that app is signed by the same certificate as the one that declared/created the permission. This level is designed to allow apps that are part of a suite, or otherwise related, to share data.
Signature/System	A permission that the system grants only to the applications that are in the Android system image, or that are signed with the same certificate as the application that declared the permission.

Application sandboxing

In order to isolate applications from each other, Android takes advantage of the Linux user-based protection model. In Linux systems, each user is assigned a unique **user ID** (**UID**) and users are segregated so that one user does not access the data of another user. All resources under a particular user are run with the same privileges. Similarly, each Android application is assigned a UID and is run as a separate process. What this means is that even if an installed application tries to do something malicious, it can do it only within its context and with the permissions it has.

This application sandboxing is done at the kernel level. The security between applications and the system at the process level is ensured through standard Linux facilities such as user and group IDs that are assigned to applications. This is shown in the following screenshot, referenced from `http://www.ibm.com/developerworks/library/x-androidsecurity/`.



Two applications on different processes on with different UID's

By default, applications cannot read or access the data of other applications and have limited access to the operating system. If application A tries to read application B's data, for example, then the operating system protects against this because application A does not have the appropriate privileges. Since the application sandbox mechanism is implemented at the kernel level, it applies to both native applications and OS applications. Thus the operating system libraries, application framework, application runtime, and all applications run within the Application Sandbox. Bypassing this sandbox mechanism would require compromising the security of Linux kernel.

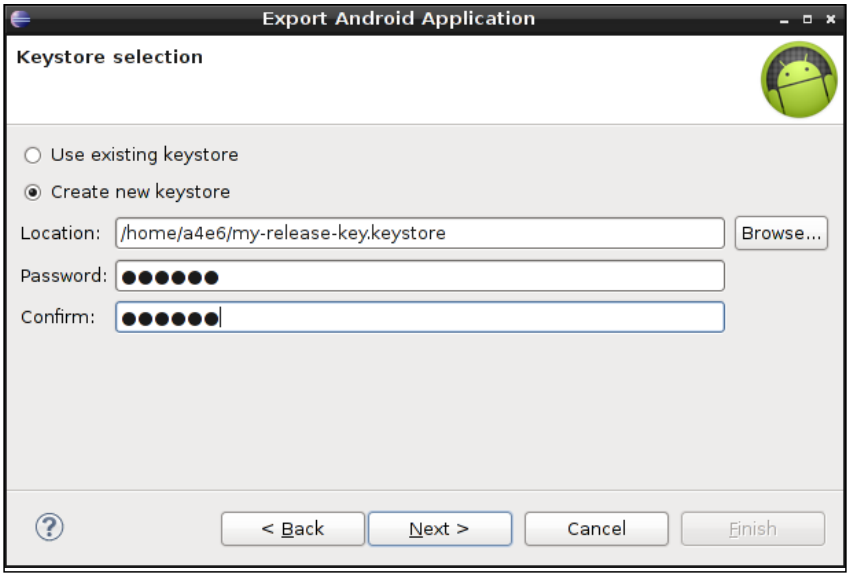
SELinux in Android

Starting with Android 4.3, **Security-Enhanced Linux** (**SELinux**) is supported by the Android security model. Android security is based on discretionary access control, which means that applications can ask for permissions, and users can grant or deny those permissions. Thus, malware can create havoc on phones by gaining permissions. But SE Android uses **mandatory access control** (**MAC**) which ensures that applications work in isolated environments. Hence, even if a user installs a malware app, the malware cannot access the OS and corrupt the device. SELinux is used to enforce MAC over all processes, including the ones running with root privileges.

SELinux operates on the principle of *default denial*. Anything that is not explicitly allowed is denied. SELinux can operate in one of two global modes: **permissive mode**, in which permission denials are logged but not enforced, and **enforcing mode**, in which denials are both logged and enforced. As per Google's documentation, in the Android 5.0 Lollipop release, Android moves to full enforcement of SELinux. This builds upon the permissive release of 4.3 and the partial enforcement of 4.4. In short, Android is shifting from enforcement on a limited set of crucial domains (`installd`, `netd`, `vold` and `zygote`) to everything (more than 60 domains).

Application Signing

All Android apps need to be digitally signed with a certificate before they can be installed on a device. The main purpose of using certificates is to identify the author of an app. These certificates do not need to be signed by a certificate authority and Android apps often use self-signed certificates. The app developer holds the certificate's private key. Using the same private key, the developer can provide updates to his applications and share data between applications. In debug mode, developers can sign the app with a debug certificate generated by the Android SDK tools. You can run and debug an app signed in debug mode but the app cannot be distributed. To distribute an app, the app needs to be signed with your own certificate. The key store and the private key which are used during this process need to be secured by the developer as they are essential to push updates. The following screenshot shows the key store selection option that is displayed while exporting the application:

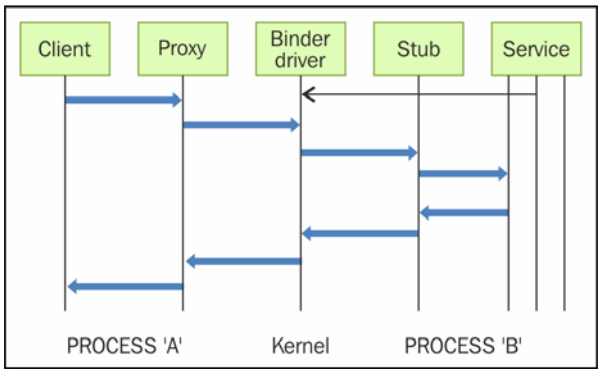


Keystore selection while exporting Android app

Secure interprocess communication

As discussed in the above sections, sandboxing of the apps is achieved by running apps in different processes with different Linux identities. System services run in separate processes and have more privileges. Thus, in order to organize data and signals between these processes, an **interprocess communication** (**IPC**) framework is needed. In Android, this is achieved with the use of the **Binder** mechanism.

The Binder framework in Android provides the capabilities required to organize all types of communication between various processes. Android application components, such as intents and content providers, are also built on top of this Binder framework. Using this framework, it is possible to perform a variety of actions such as invoking methods on remote objects as if they were local, synchronous and asynchronous method invocation, sending file descriptors across processes, and so on. Let us suppose the application in **Process 'A'** wants to use certain behavior exposed by a service which runs in **Process 'B'**. In this case, **Process 'A'** is the client and **Process 'B'** is the service. The communication model using Binder is shown in the following diagram:



Binder Communication Model

All communication between the processes using the Binder framework occurs through the `/dev/binder` Linux kernel driver. The permissions to this device driver are set to world readable and writable. Hence, any application may write to and read from this device driver. All communications between the client and server happen through **proxies** on the client side and **stubs** on the server side. The proxies and the stubs are responsible for sending and receiving the data, and the commands, sent over the Binder driver.

Each service (also called a Binder service) exposed using the Binder mechanism is assigned with a **token**. This token is a 32-bit value and is unique across all processes in the system. A client can start interacting with the service after discovering this value. This is possible with the help of Binder's **context manager**. Basically, the context manager acts as a name service, providing the handle of a service using the name of this service. In order to get this process working, each service must be registered with the context manager. Thus, a client needs to know only the name of a service to communicate. The name is resolved by the context manager and the client receives the token that is later used for communicating with the service. The Binder driver adds the UID and the PID value of the sender process to each transaction. As discussed earlier, each application in the system has its own UID and this value is used to identify the calling party. The receiver of the call may check the obtained values and decide if the transaction should be completed. Thus, the security is enforced, with the Binder token acting as a security token as it is unique across all processes.

Android hardware components

Android is compatible with a wide range of hardware components. Having a Linux kernel made this easy, as Linux supports large variety of hardware. This gives manufacturers a lot of flexibility as they can design based on their requirement, without worrying about compatibility. This poses a significant challenge for forensic analysts during investigations. Thus, understanding the hardware components and device types would greatly help in understanding Android forensics.

Core components

The components present in a device change from one manufacturer to another and also from one model to another. However, there are some components which are found in most mobile devices. The following sections provide an overview of the commonly-found components of an Android device.

Central processing unit

The central processing unit (**CPU**), also known as processor, is responsible for executing everything that happens on a mobile device. It tells the device what to do and how to do it. Its performance is measured based on the number of tasks it can complete per second, known as a **cycle**. For example, 1 GHz processor can process one billion cycles per second. The higher the capacity of the processor, the smoother the performance of the phone will be.

When dealing with smart phones, we come across the following terminologies—ARM, x86 (Intel), MIPS, Cortex, A5, A7, or A9. ARM is the name of a company that licenses their architectures (branded Cortex) with different models coming up each year such as the aforementioned A series (A5, A7, and A9). Based on these architectures, chip makers release their own series of chipsets (Snapdragon, Exynos, and so on) which are used in mobile devices. The latest smartphones are powered by dual core, quad core and even octa core processors.

Baseband processor

Smartphones today support a variety of cellular protocols, including GSM, 3G, 4G and 4G LTE. These protocols are complicated and require a large amount of CPU power to process data, generate packets and transmit them to the network provider. To handle this process, smartphones now use a baseband modem which is a separate chip included in smartphones that communicates with the main processor. These baseband modems have their own processor called the baseband processor and run their own operating system. The baseband processor manages several radio control functions such as signal generation, modulation, encoding, as well as frequency shifting. It can also manage the transmission of signals.

The baseband processor is generally located on the same circuit board as the CPU but consists of a separate radio electronics component.

Memory

Android phones, just like normal computers, use two primary types of memory–**random access memory** (**RAM**) and **read-only memory** (**ROM**). Although most users are familiar with these concepts, there is some confusion when it comes to mobile devices.

RAM is volatile, which means its contents are erased when the power is removed. RAM is very fast to access and is used primarily for the runtime memory of software applications (including the device's operating system and any applications). In other words, it is used by the system to load and execute the OS and other applications. So the number of applications and processes that can be run simultaneously depends on this RAM size.

ROM (commonly referred to as Android ROM) is non-volatile, which means it retains the contents even when the power is off. Android ROM contains the boot loader, OS, all downloaded applications and their data, settings and, so on.

Note that the part of memory that is used for the boot loader is normally locked and can only be changed through a firmware upgrade. The remaining part of the memory is termed by some manufacturers as user memory. The data of each application stored here will not be accessible to other applications. Once this memory gets filled up, the device slows down. Both RAM and Android ROM are manufactured into a single component called as **Multichip Package** (**MCP**).

SD Card

The SD card has great significance with respect to mobile forensics because, quite often, data that is stored in these can be vital evidence. Many Android devices have a removable memory card commonly referred to as their **Secure Digital** (**SD**) card. This is in contrast to Apple's iPhone which does *not* have any provision for SD cards. SD cards are non-volatile, which means data is stored in it even when it is powered off. SD cards use flash memory, a type of **Electrically Erasable Programmable Read-Only Memory** (**EEPROM**) that is erased and written in large blocks instead of individual bytes. Most of the multimedia data and large files are stored by the apps in SD card. In order to interoperate with other devices, SD cards implement certain communication protocols and specifications.

In some mobile devices, although an SD card interface is present, some portion of the on-board NAND memory (non-volatile) is carved out for creating an emulated SD card. This essentially means the SD card is not removable. Hence, forensic analysts need to check whether they are dealing with the actual SD card or an emulated SD card. SD memory cards come in several different sizes. Mini-SD card and micro-SD card contain the same underlying technology as the original SD memory card, but are smaller in size.

Display

Mobile phone screens have progressed dramatically over the last few years. Below is a brief description of some of the widely used types of mobile screens as described at `http://www.in.techradar.com/news/phone-and-communications/mobile-phones/Best-phone-screen-display-tech-explained/articleshow/38997644.cms`.

The **thin film transistor liquid crystal display** (**TFT LCD**) is the most common type of screen found in mobile phones. These screens have a light underneath them which shines through the pixels to make them visible.

The **active-matrix organic light-emitting diode** (**AMOLED**) is a technology based on organic compounds and known for its best image quality while consuming low power. Unlike LCD screens, AMOLED displays don't need a backlight; each pixel produces its own light, so phones using them can potentially be thinner.

Battery

Battery is the lifeblood of a mobile phone and is one of the major concerns with modern smartphones. The more you use the device and its components, the more battery is consumed. The following different types of batteries are used in mobile phones:

- **Lithium Ion (Li-ion)**: These batteries are the most popular batteries used in cell phones as they are light and portable. They are well known for their high energy density and low maintenance. However, they are expensive to manufacture compared to other battery types.
- **Lithium Polymer (Li-Polymer)**: These batteries have all the attributes of a Lithium Ion battery but with ultra slim geometry and simplified packaging. They are the very latest and found only in few mobile devices.
- **Nickel Cadmium (NiCd)**: These batteries are old technology batteries and suffer from memory effect. As a result, the overall capacity and the lifespan of the battery are reduced. In addition to this, NiCd batteries are made from toxic materials that are not environment-friendly.
- **Nickel Metal Hydrid (NiMH)**: These batteries are same as the NiCd batteries, but can contain higher energy and run longer, between 30 and 40 percent. They still suffer from memory effect, but comparatively less than the NiCd batteries. They are widely used in mobile phones and are affordable too.

The battery type can be found by looking at the details present on its body. For example, the following is an image of a Li-ion battery:



Lithium-ion battery

Most SD cards are located behind the battery. During forensic analysis, accessing an SD card would require removing the battery which would power off the device. This can have certain implications which will be discussed in detail in later chapters.

Apart from the components described above, here are some of the other components that are well known:

- Global Positioning System
- Wi-Fi
- Near field communication
- Bluetooth
- Camera
- Keypad
- USB
- Accelerometer and gyroscope
- Speaker
- Microphone

Android boot process

Understanding the boot process of an Android device will help us to understand other forensic techniques which involve interacting with the device at various levels. When an Android device is first powered on, there is a sequence of steps that are executed, helping the device to load necessary firmware, OS, application data, and so on into memory. The following information is compiled from the original post published at `http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html`.

The sequence of steps involved in Android boot process is as follows:

1. Boot ROM code execution
2. The boot loader
3. The Linux kernel
4. The init process
5. Zygote and Dalvik
6. The system server

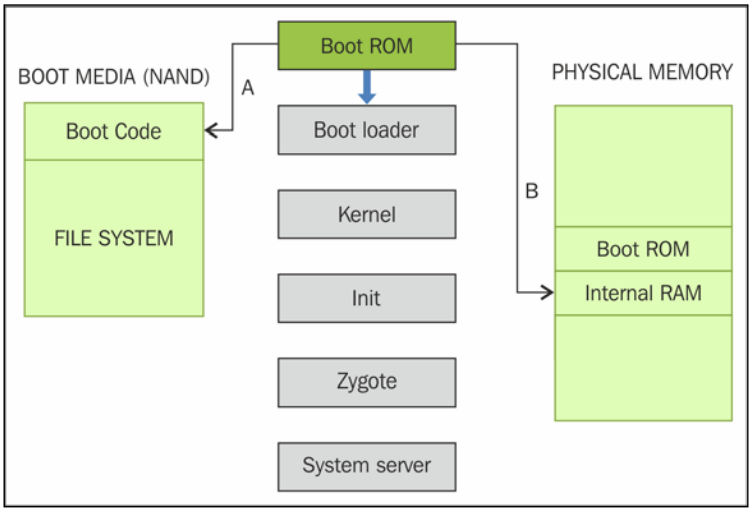
We will examine each of these steps in detail.

Boot ROM code execution

Before the device is powered on, the device CPU will be in a state where no initializations will have taken place. Once the Android device is powered on, execution starts with the boot ROM code. This boot ROM code is specific to the CPU the device is using. As shown in the the following diagram, this phase includes two steps:

1. When, boot ROM code is executed, it initializes the device hardware and tries to detect the boot media. Thus, the boot ROM code scans till it finds the boot media. This is similar to the BIOS function in the boot process of a computer.

2. Once the boot sequence is established, the initial boot loader is copied to the internal RAM. After this, the execution shifts to the code loaded into the RAM.



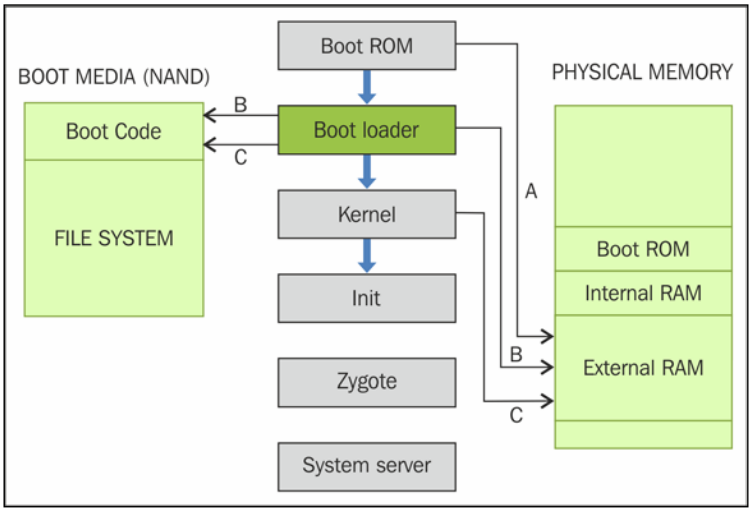
Android boot process: Boot ROM code execution

The boot loader

The boot loader is a piece of program that is executed before the operating system starts to function. Boot loaders are present in desktop computers, laptops and mobile devices as well. In an Android boot loader, there are two stages—**initial program load** (**IPL**) and **second program load** (**SPL**). As shown in the following diagram, this involves three steps explained as follows:

1. IPL deals with detecting and setting up external RAM.
2. Once external RAM is available, the SPL is copied into the RAM and execution is transferred to it. The SPL is responsible for loading the Android operating system. It also provides access to other boot modes such as fastboot, recovery, and so on. It initiates several hardware components such as console, display, keyboard and file systems, virtual memory, and other features.

3. After this, the SPL tries to look for the Linux kernel. It will load this from the boot media and copy it to the RAM. Once the boot loader is done with this process, it transfers the execution to the kernel.



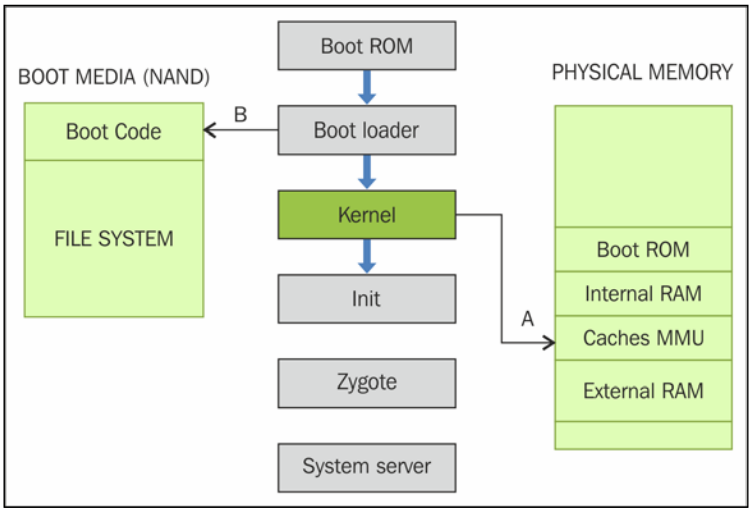
Android boot process: The boot loader

The Linux kernel

The Linux kernel is the heart of the Android operating system and is responsible for process management, memory management, and enforcing security on the device. After the kernel is loaded, it mounts the **root file system** (**rootfs**) and provides access to system and user data, as described in the following steps:

1. When the memory management units and caches have been initialized, the system can use virtual memory and launch user space processes.

2. The kernel will look in the rootfs for the init process and launch it as the initial user space process.



Android boot process: The kernel

The init process

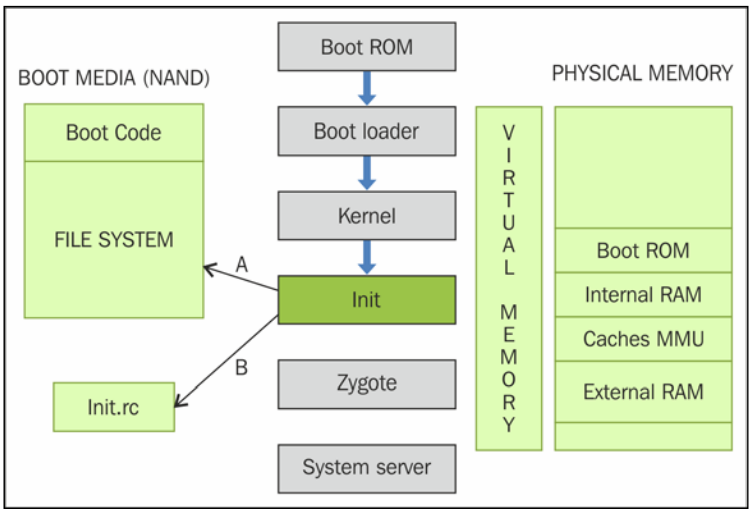
The init is the very first process that starts and is the root process of all other processes.

1. The init process will look for a script named `init.rc` that describes the system services, file system, and any other parameters that need to be set up.
    - The `init` process can be found at:`<android source>/system/core/init`..
    - The `init.rc` file can be found in source tree at `<android source>/system/core/rootdir/init.rc`.



More details about the Android file hierarchy will be covered in *Chapter 3, Understanding Data Storage on Android Devices*.

2. The `init` process will parse the `init.rc` script and launch the system service processes. At this stage, you will see the Android logo on the device screen.



Android boot process: The init process

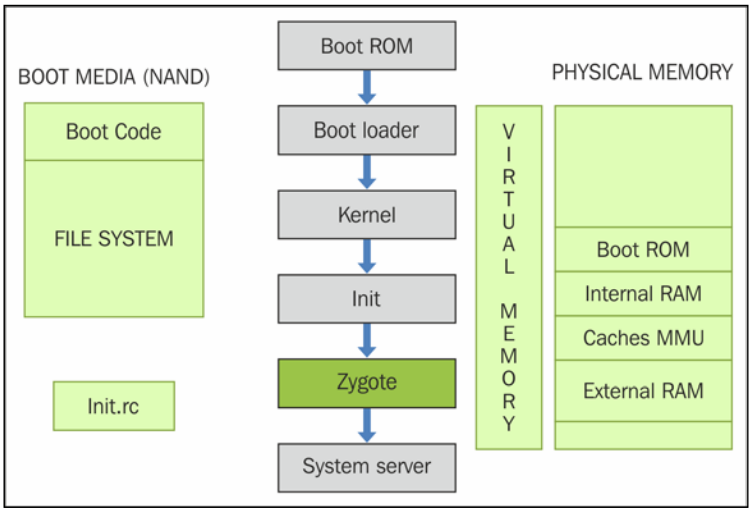
Zygote and Dalvik

Zygote is one of the first init processes created after the device boots. It initializes the Dalvik virtual machine and tries to create multiple instances to support each android process. As discussed in earlier sections, the Dalvik virtual machine is the virtual machine which executes Android applications written in Java.

Zygote facilitates using a shared code across the VM, thus helping to save the memory and reduce the burden on the system. After this, applications can run by requesting new Dalvik virtual machines that each one runs in. Zygote registers a server socket for zygote connections, and also preloads certain classes and resources. This Zygote loading process has been more clearly explained at `http://www.kpbird.com/2012/11/in-depth-android-boot-sequence-process.html`. This is also explained as follows:

- `Load ZygoteInitclass`: This class loads the `ZygoteInit` class. Source Code: `<Android Source>/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java`.

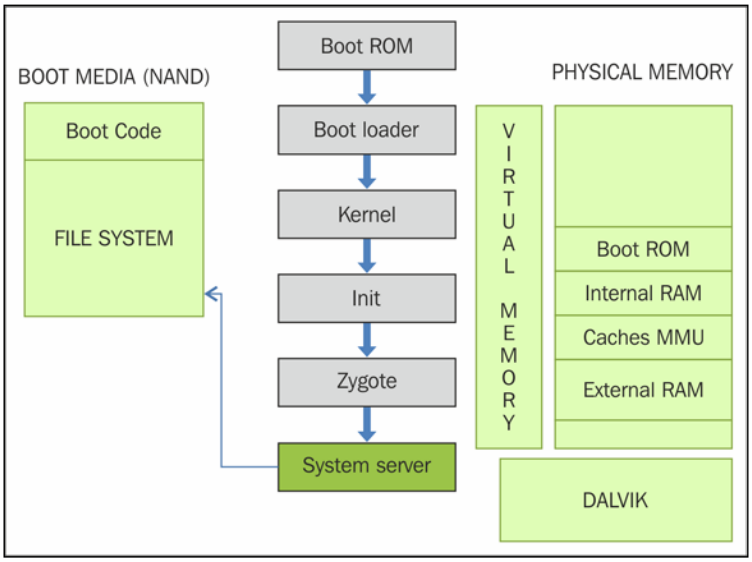
- `registerZygoteSocket()`: This registers a server socket for zygote command connections.
- `preloadClasses()`: This is a simple text file containing the list of classes that need to be preloaded will be executed here. This file can be seen at `<Android Source>/frameworks/base`.
- `preloadResources()`: This deals with native themes and layouts. Everything that includes the `android.R` file will be loaded using this method.



Android boot process: The Zygote

System server

All the core features of the device such as telephony, network, and other important functions, are started by the system server, as shown in the following diagram:



Android boot process: System server

The following core services are started in this process:

- Start Power Manager
- Create Activity Manager
- Start Telephony Registry
- Start Package Manager
- Set Activity Manager Service as System Process
- Start Context Manager
- Start System Context Providers
- Start Battery Service
- Start Alarm Manager
- Start Sensor Service
- Start Window Manager
- Start Bluetooth Service
- Start Mount Service

The system sends a broadcast action called `ACTION_BOOT_COMPLETED` which informs all the dependent processes that the boot process is complete. After this, the device displays the home screen and is ready to interact with the user. The Android system is now fully operational and is ready to interact with the user.

As explained earlier, several manufacturers use the Android operating system on their devices. Most of these device manufacturers customize the OS based on their hardware and other requirements. Hence, when a new version of Android is released, these device manufacturers have to port their custom software and tweaks to the latest version.

Summary

Understanding Android architecture and its security model is crucial to having a proper understanding of Android forensics. The inherent security features in Android OS, such as application sandboxing, permission model, and so on, safeguard Android devices from various threats and also act as an obstacle for forensic experts during investigation. Having gained this knowledge of Android internals, we will discuss more about what type of data is stored on the device and how it is stored, in the next chapter.

2
Setting Up an Android Forensic Environment

It is crucial to have an established forensic environment set up before the start of any forensic examination. The forensic analyst needs to be in total control of the workstation at all times. This chapter will take you through everything that is necessary to have an established forensic set up to examine Android devices.
In this chapter, we will cover the following topics:

- Installation of necessary software on the workstation
- Connecting and accessing an Android device from the workstation
- Using ADB commands on the device
- Rooting Android devices

The Android forensic setup

Setting up a sound and well-controlled forensic environment is crucial before the start of any investigation. Start with a fresh and **forensically sterile** computer.
A forensically sterile computer is one that prevents the potential of cross contamination, does not introduce unwanted data, and is free from viruses and other malware. This is to ensure that the software present on the machine does not interfere with the current investigation. Install basic software, such as the following ones; they are necessary to connect to the device and perform analysis:

- Android SDK
- Mobile drivers
- MS Office packages
- Tools used for analysis

The Android SDK

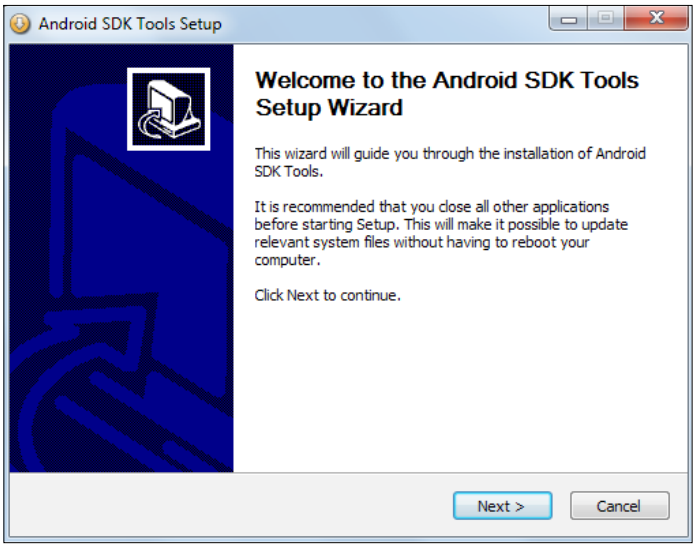
It is important that we begin the discussion with the Android SDK. The Android **Software Development Kit** (**SDK**) helps developers build, test, and debug applications to run on Android. It includes software libraries, APIs, emulator, reference material, and many other tools. These tools not only help create Android applications but also provide documentation and utilities that help significantly in forensic analysis of Android devices. Having sound knowledge of the Android SDK can help you understand the particulars of a device. This, in turn, will help you during an investigation.

During forensic examination, the SDK helps us connect the device and access the data present on the device. The SDK is supported in most of environments, including Windows, Linux, and OS X. It can be downloaded for free from `http://developer.android.com/sdk/index.html`.

Installing the Android SDK

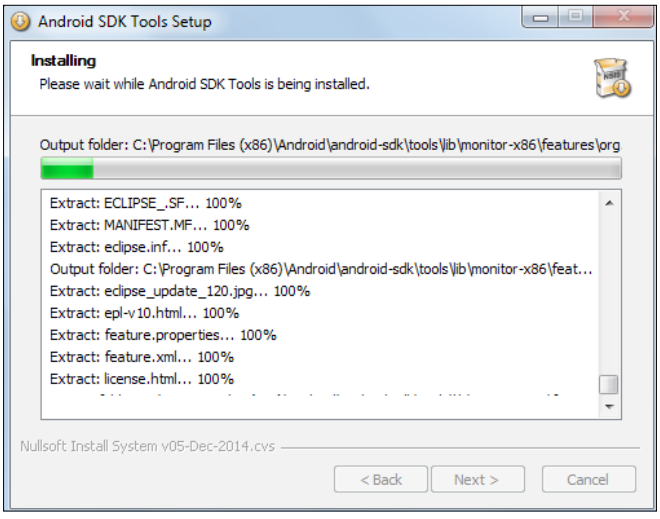
Google now offers Android Studio and SDK tools only as download options. Android studio contains the Android IDE, SDK tools, Android 5.0 (Lollipop) platform, Android 5.0 system image with Google APIs, and other newly introduced features. However, for a forensic lab setup, downloading the SDK tools package alone would be sufficient. The following is a step-by-step procedure to install the Android SDK on a Windows 8 machine:

1. Before starting the Android SDK installation, make sure that the system has the **Java Development Kit** (**JDK**) installed, because the Android SDK is dependent on the Java SE Development Kit. JDK can be downloaded from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. Select the correct download based on your operating system.
2. Download the latest version of the SDK tools package from `http://developer.android.com/sdk/index.html`. The `.exe` version of the package is recommended for download.
3. Run the installer file downloaded in step 2. A wizard window will appear, as shown in the following screenshot. Then, click on **Next**.



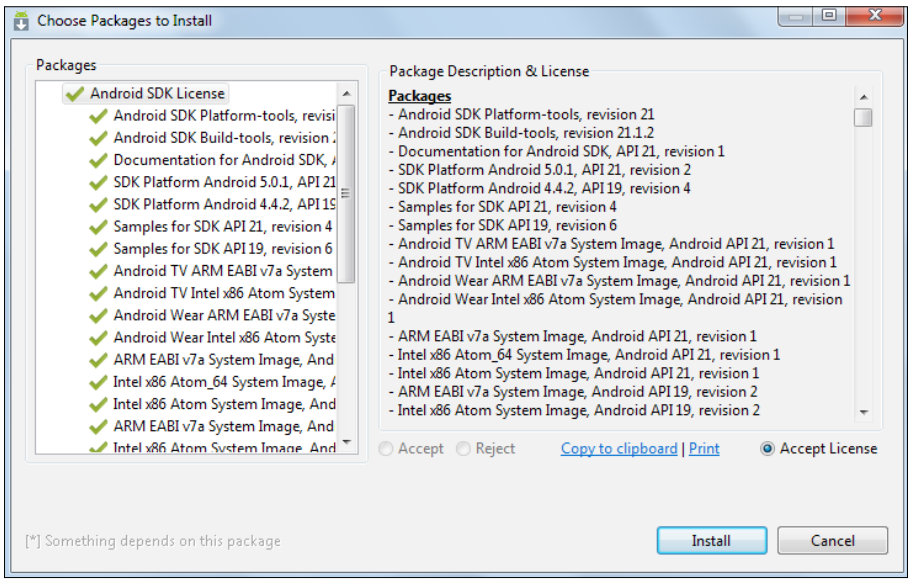
The Android SDK setup wizard

4. The setup will automatically detect whether Java is installed on the system and select the path where Java is installed.
5. Choose the installation location and remember it for future use. In this example, we will install it in `C:\Program Files (x86)`. In the case of a 32-bit operating system, the default location would be `C:\Program Files`. All the necessary files will be extracted to this location, as shown in the following screenshot:



The Android SDK tools installation

6. Once the installation is complete, open the `C:\Program Files (x86)\Android\android-sdk` directory and double-click on `SDK Manager.exe`. Make sure that you select the Android SDK Platform tools and any one release platform version of Android, as shown in the following screenshot. Some of the items are automatically selected. For instance, Google USB Driver is necessary to work with Android devices on Windows and is selected by default. Accept the license terms and then proceed to install it by clicking the **Install** button:



Android packages installation

The last step in the preceding process takes some time to complete. Once it is done, the Android SDK installation is complete. You can now update the system's environment variables (path) by pointing to the executable files.



The minimal ADB and fastboot tool, which is only 2 MB in size and available freely on XDA forums (`http://forum.xda-developers.com/showthread.php?t=2317790`), can be used without installing the complete Android SDK. This tool is a Windows installer that will install the latest version of ADB and fastboot quickly and easily.

Android Virtual Device

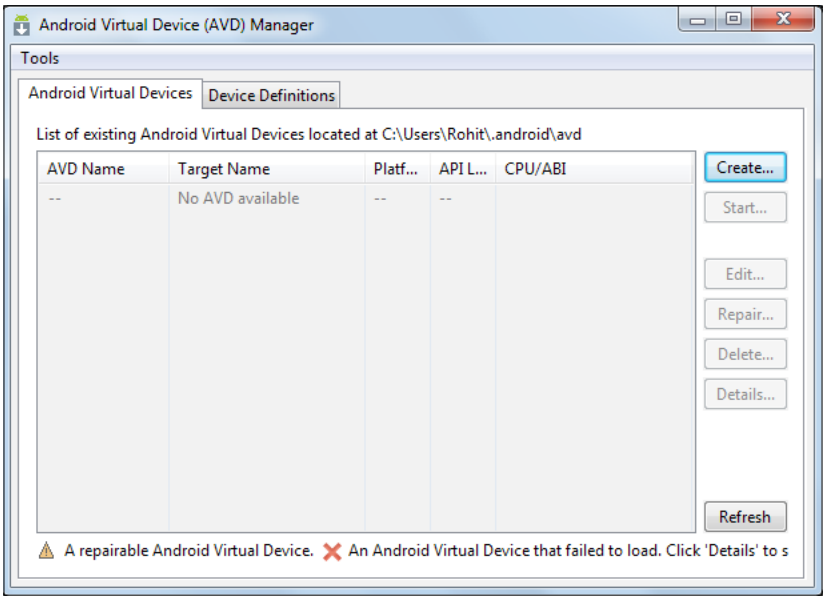
With the Android SDK installed, you can create an **Android Virtual Device** (**AVD**), which is an **emulator** that runs on the workstation. An emulator is often used by developers when creating new applications. However, an emulator is also considered helpful during forensic investigations. It allows the investigator to understand how certain applications behave and how the installation of an application affects the device. Another advantage is that you can design an emulator with the desired version. This is especially helpful when working with devices running on older versions of Android. Also, AVD comes with root as default.

The following steps will guide you to create an AVD on the workstation:

1. Open command prompt (`cmd.exe`). To start AVD manager from the command line, navigate to the path where SDK is installed and call the android tool with the `avd` option as shown here:

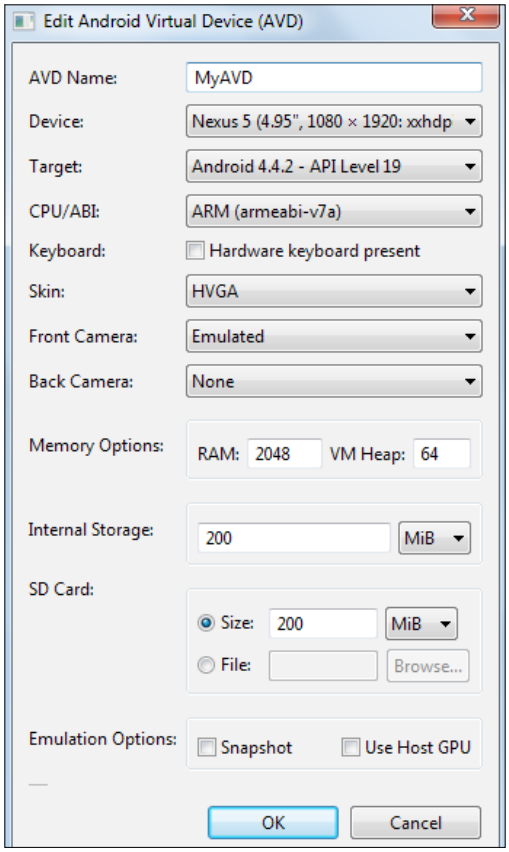
   ```
   C:\Program Files (x86)\Android\android-sdk\tools>android avd
   ```

   This will automatically open AVD manager, as shown in the following screenshot. AVD manager can also be started using the graphical AVD manager. To start it, navigate to the location where the SDK is installed and double-click on `AVD Manager`.



The Android Virtual Device Manager

2. Click on **Create** in the **AVD Manager** window to create a new virtual device. Click on **Edit** to change the configuration of an existing virtual device, as shown in the following screenshot:

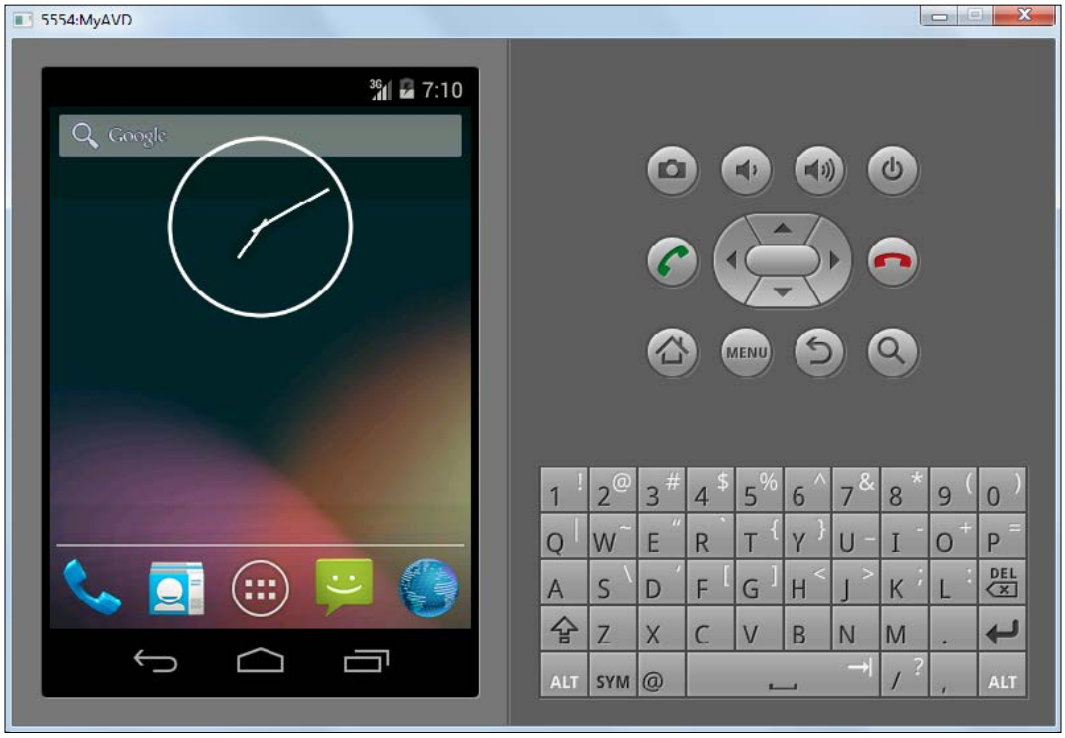


AVD details

3. Enter the necessary details based on the following information:
    - **AVD Name**: Provide any name for the virtual device, for example, `MyAVD`.
    - **Device**: Select any device from the available options based on the screen size.
    - **Target**: This option helps you select the platform of the device. Note that only those versions that were selected and installed during the SDK installation will be shown here for you to select. The platform version can be selected based on the OS of the seized device. For our example, the Android 4.4.2 platform is selected.

- **Hardware**: You can select hardware features to customize the emulator, for example, the size of internal storage memory, SD card, and so on. Once again, details such as screen resolution, hardware, and so on can be selected based on the details corresponding to the seized device.

4. Once this is done, the **AVD Manager** screen appears with the newly created AVD listed under the **Android Virtual Devices** tab. Select the AVD and click on **Start**. Then, click on **Launch**.
5. The emulator will be automatically launched. This could take several minutes, depending on the workstation's CPU and RAM. Here is a screenshot of an AVD after its successful launch:



The Android emulator

An emulator can be used to configure e-mail accounts, install applications, surf the Internet, send text messages, and so on. Forensic analysts and security engineers can learn a great deal about Android and how it operates by leveraging the emulator and examining the network, filesystem, and data artifacts. The data created when working on an emulator is stored in your home directory, in a folder named .android. For instance, in our example, the details about `MyAVD` emulator that we created earlier are stored in `C:\Users\Rohit\.android\avd\MyAVD.avd`. There are several files present under this directory, and here are some files of interest for a forensic analyst:

- `cache.img`: This is the disk image of the `/cache` partition.
- `sdcard.img`: This is the disk image of the SD card partition.
- `Userdata-qemu.img`: This is the disk image of `/data` partition. The `/data` partition contains valuable information about the device user.
- `config.ini`: This is the configuration file that stores hardware options in AVD's local directory.
- `emulator-user.ini`: This file contains values that can reset the position of the window.
- `Androidtool.cfg`: This file can be used to manually set proxy settings for the Android SDK.

Connecting and accessing an Android device from the workstation

In order to extract information from an Android device, it first needs to be connected to the workstation. As mentioned earlier, care should be taken to make sure that the workstation is forensically sterile and used only for the purpose of investigation. A forensically sterile workstation is one that has a proper build and is free from viruses and other malware. When a device is connected to the computer, changes can be made to the device. Hence, it is crucial that the forensic examiner maintains control over the device at all times. In the world of mobile forensics, using write-protection mechanisms may not be of great help, as they prevent successful acquisition of the device. This is because during acquisition, certain commands need to be pushed to the device to extract the necessary data.

Identifying the device cable

An Android device can be connected to the workstation using the physical USB interface of the device. This physical USB interface allows the device to connect, share data, and recharge from a computer. USB interfaces might change from manufacturer to manufacturer and also from device to device. There are different types, such as mini-USB, micro-USB, and other proprietary formats. Here is a brief description of the most widely used connector types:

Connector type	Description
Mini—A USB	It is approximately 7 x 3 mm in size, with two of the corners on one long side lifted out.
Micro—B USB	It is approximately 6 x 1.5 mm in size, with two corners cut off to form a trapezoid.
Co-axial	It has a circular hole with a pin sticking up in the middle. There are different sizes, varying from 2 to 5 mm in diameter. This type is widely used with Nokia models.
D Sub-miniature	It has a rectangular shape with two rounded corners. The length of the rectangle varies, but the height is always 1.5 to 2 mm. This type is used mostly by Samsung and LG devices.

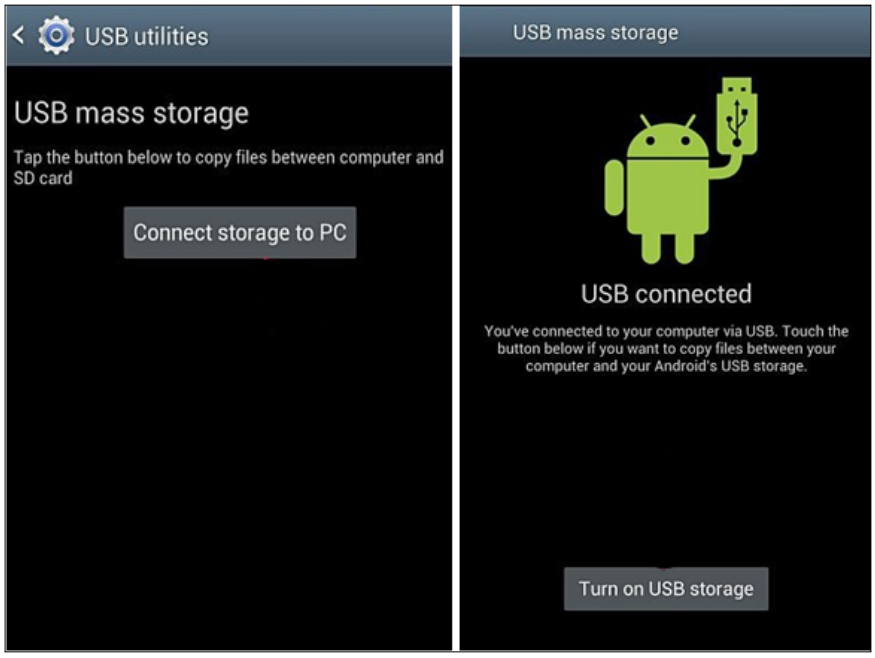
Hence, the first step in acquisition is to identify what kind of device cable is required.

Installing device drivers

A mobile device can communicate with the computer only when the necessary device drivers are installed on the computer. Without the necessary drivers, the computer may not be able to identify and work with the connected device. Since Android may be modified and customized by the manufacturers, there is no single generic driver that would work for all the Android devices. Each manufacturer has their own proprietary drivers and distributes them along with the phone. So, it is important to identify the specific device driver that needs to be installed. Of course, some of the Android forensic toolkits come with some generic drivers or a set of most used drivers. They may not work with all models of Android phones. Some Windows operating systems are able to auto detect and install the drivers once the device is plugged in but, more often than not, Windows fails. The device drivers for each manufacturer can be found on their respective websites.

Accessing the device

After installing the necessary device drivers, connect the Android device to the computer, directly using the USB cable in order to access it. It is important to use genuine manufacturer-specific cables, because universal cables may not work properly with certain devices. Also, the investigator may encounter certain driver issues. Some of the devices may not be USB 3.0 compatible, which may lead to failed driver installations. In this case, it is recommended that you try switching to USB 2.0 ports. Once the device is connected, it will appear as a new drive, and you can access the files on the external storage. Some older Android devices may not be accessible unless the **Connect storage to PC** option (navigate to **Settings** | **USB utilities**) is enabled on the device. In this case, after connecting the device through a USB, the **Turn on USB storage** option needs to be selected, as shown in the following screenshot:



USB mass storage connection

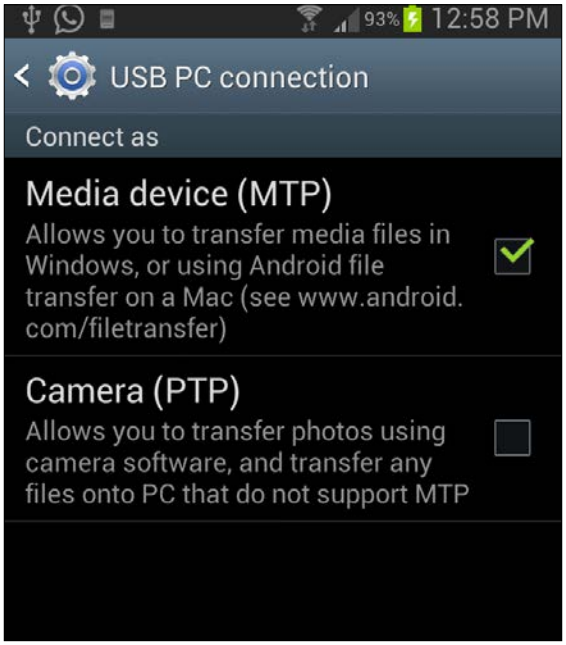
This is because older Android devices required USB mass storage mode to transfer files between a computer and the device. Latest Android devices use the **Media Transfer Protocol** (**MTP**) or the **Picture Transfer Protocol** (**PTP**), as there were some issues with the USB mass storage protocol. With USB mass storage, the drive makes itself completely available to the computer, just as if it were an internal drive.

However, the problem is that the device that is accessing the storage needs exclusive access to it. In other words, when the device drive is connected to the computer, it has to be disconnected from the Android operating system running on the device in order to work. So, any files or apps stored on the SD card or USB storage will be unavailable when it is connected to the computer. In MTP, the Android device doesn't expose its entire storage to Windows. Instead, when you connect a device to your computer, the computer queries the device, and the device responds with a list of files and directories it offers. If the computer has to download a file, it would send a request to the file from the device, and the device would send the file over the connection. PTP is also similar to MTP and is commonly used by digital cameras. In this mode, the Android device will work with digital camera applications that support PTP but not MTP. On the latest devices, you can select either MTP or PTP options by going to **Settings** | **Storage** | **USB computer connection**.



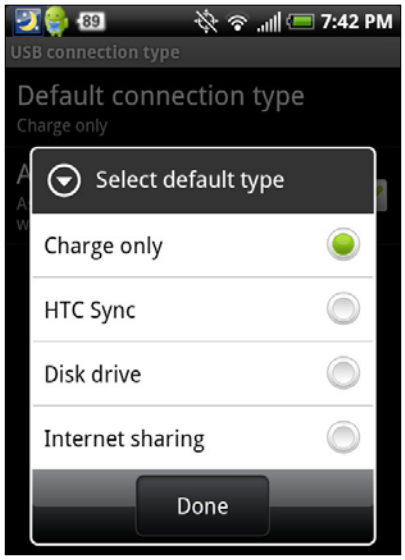
On some Android devices, the option to select MTP and PTP protocols is provided only after connecting the device to the computer. After the device is connected, watch out for the Notifications bar at the top of your screen, and you will see a USB symbol appear. Pull down the notifications bar, and you will find an option to switch between MTP and PTP.

As shown in the following screenshot, the **MTP** and **PTP** options are shown only after connecting the device to a computer and pulling down the notifications bar:



USB PC connection in an Android device

In the case of certain Android devices (especially with HTC), the device may expose more than one functionality when connected with a USB cable. For instance, as shown in the following screenshot, when an HTC device is connected to the workstation, it presents a menu with four options:



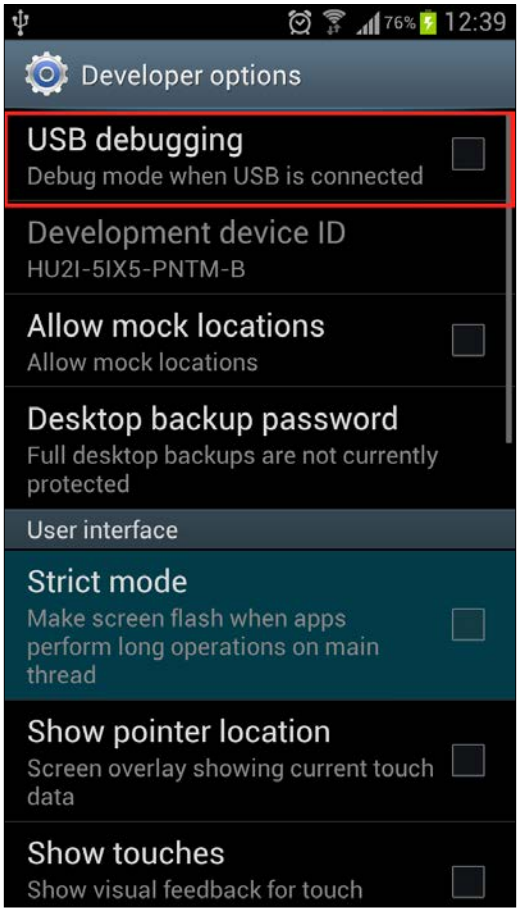
Disk drive option on a HTC device

The default selection is **Charge only**. When the **Disk drive** option is selected, it is mounted as a disk drive. When the device is mounted as a disk drive, you will be able to access the SD card present on the device.

From a forensic point of view, the SD card has a significant value, as it may contain files that are important for an investigation. Most of the images and large files related to multimedia are stored in this external storage. SD cards are commonly formatted with the FAT16 filesystem, but you might also encounter some SD cards that have FAT32 and other filesystems. As discussed in *Chapter 1*, *Introducing Android Forensics*, note that most of the recent devices have the emulated SD card feature that uses the device's NAND flash to create a non-removable SD card. Thus, all the sensitive files present on the external storage can be accessed in this way. However, the core application data stored under `/data/data` will remain on the device and cannot be accessed in this way.

Android Debug Bridge

In Android forensics, **Android Debug Bridge** (**ADB**) plays a very crucial role. It is present at `<sdk_path>/platform-tools`. In order to work with ADB, the **USB-debugging** option needs to be enabled. On a Samsung phone, you can access this by going to **Settings** | **Developer** options; as shown in the following screenshot:



The USB debugging option in Android

However, this may not be the case with all the devices, as different devices have different environments and configuration features. Sometimes, the examiner might have to use certain techniques to access the developer options on a few devices. These techniques are device specific and need to be researched and determined by the forensic analyst, based on the device type and model.



On some devices, the **Developer options** menu is hidden and can be turned on by tapping on the **Build Number** field (navigate to **Settings** | **About Device**) *seven* times.

Once the **USB debugging** option is selected, the device will run the **adb daemon** (**adbd**) in the background and will continuously look for a USB connection. The daemon usually runs under a non-privileged shell user account and thus does not provide access to internal application data. However, on rooted phones, adbd will run under the root account and thus provide access to the entire data. On the workstation (where the Android SDK) is installed, adbd will run as a background process. Also, on the same workstation, a client program will run that can be invoked from a shell by issuing the `adb` command. We are going to see this in the following sections. When the adb client is started, it first checks whether the adbd is already running. If it isn't, it initiates a new process to start the abdb. The daemons communicate over their local host on ports 5555 through 5585. The even port communicates with the device's console, while the odd port is for adb connections. The adb client program communicates with the local adbd over port 5037.

Using adb to access the device

As stated earlier, adb is a powerful tool that allows you to communicate with the Android device. We will now look at how to use adb and access certain parts of the device that cannot be accessed normally. It is important to note that the collection of data through adb may or may not be accepted as evidence in court. This will depend on the laws of respective countries. The following sections list some of the commonly used adb commands, their meanings, and usage in a logical sequence.

Detecting a connected device

After connecting the device to the workstation and before issuing other adb commands, it is helpful to know whether the Android device is properly connected to the adb server. This can be done using the `adb.exe` devices command, which lists out all the devices that are connected to the computer, as shown in the following command. This would also list the emulator if it is running at the time of issuing the command:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe devices
List of devices attached
4df16ac5115e4e04        device
```



Remember that if the necessary drivers are not installed, then the preceding command would show a blank message. If you encounter this situation, download the necessary drivers from the manufacturer and install them.

As seen in the preceding commands, the output contains the serial number of the device, followed by the connection state. The serial number is a unique string used by ADB to identify each Android device. The possible values of the connection state and their meaning is explained in the following lines:

- `offline`: The instance is not connected to adb or is not responding.
- `device`: The instance is connected to the adb server.
- `no device`: There is no device connected.

Directing commands to a specific device

If more than one device is connected to the system, you must specify the target device while issuing the commands. For example, consider the following case:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe devices
List of devices attached
4df16ac5115e4e04        device
7f1c864544456o6e    device
```

As shown in the preceding command-line output, there are two devices attached to the workstation. In this case, `adb` needs to be used along with the `-s` option to issue commands to the device of your choice:

```
adb shell -s4df16ac5115e4e04
```

Similarly, the `-d` command can be used to direct an `adb` command to the only attached USB device, and the `-e` command can be used to direct an `adb` command to the only running emulator instance.

Issuing shell commands

As mentioned in *Chapter 1*, *Introducing Android Forensics*, Android runs on a Linux kernel and provides a way to access the shell. Using ADB, you can access a shell to run several commands on an Android device. For those who are not familiar with the Linux environment, the Linux shell refers to a special program that allows you to interact with it by entering certain commands from the keyboard. The shell will execute the commands and display their output.

More details about how things work on the Linux environment have been provided under the *Rooting Android device* section in this chapter. The `adb` shell command can be used to enter into a remote shell, as shown in the following command-line output. Once you enter the shell, you can execute most of the Linux commands:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe shell
shell@android:/ $
```

After executing the command, observe that the shell prompt is displayed to the user. In this shell prompt, commands can be executed on the device. For instance, as shown in the following command line, the `ls` command can be used to view all the files within a directory:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe shell
shell@android:/ $ ls
ls
acct
cache
config
d
data
default.prop
dev
efs
etc
factory
fstab.smdk4x12
```

The following section explains some of the widely used Linux commands that are very helpful while interacting with an Android device.

Basic Linux commands

We will now take a look at some of the Linux commands and their usage with respect to an Android device:

- `ls`: The ls command (with no option) lists files and directories present in the current directory. With the `-l` option, it also shows their size, modified date and time, owner of file and it's permission, and so on as shown in the following command-line output:

  ```
  shell@android:/ $ ls -l
  ```

```
ls -l
drwxr-xr-x root     root              2015-01-17 10:13 acct
drwxrwx--- system   cache             2014-05-31 14:55 cache
dr-x------ root     root              2015-01-17 10:13 config
lrwxrwxrwx root     root              2015-01-17 10:13 d -> /sys/
kernel/debug
drwxrwx--x system   system            2015-01-17 10:13 data
-rw-r--r-- root     root          116 1970-01-01 05:30 default.
prop
drwxr-xr-x root     root              2015-01-17 10:13 dev
drwxrwx--x radio    system            2013-08-13 09:34 efs
lrwxrwxrwx root     root              2015-01-17 10:13 etc -> /
system/etc
```

Similarly, here are a few options that can be used along with the `ls` command. Depending on the requirement, one or more of these options can be used by the investigator to view the details:

Option	Description
`a`	Lists hidden files
`c`	Displays files by timestamp
`d`	Displays only directories
`n`	Displays the long format listing, with GID and UID numbers
`R`	Displays subdirectories as well
`t`	Displays files based on timestamp
`u`	Displays the file access time

- `cat`: The cat command reads one or more files and prints them to standard output, as shown in the following command lines:

```
shell@android:/ $ cat default.prop
cat default.prop
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
```

```
ro.allow.mock.location=0
ro.debuggable=0
persist.sys.usb.config=mtp
```

The `>` operator can be used to combine multiple files into one.
The `>>` operator can be used to append to an existing file.

- `cd`: The `cd` command is used to change from one directory to another. This is used while navigating from one folder to another. The following example shows commands used to change to the system folder:

```
shell@android:/ $ cd /system
cd /system
shell@android:/system $
```

- `cp`: The `cp` command can be used to copy a file from one location to another. The syntax for this command is as follows:

```
$ cp [options] <source><destination>
```

- `chmod`: The `chmod` command is used to change the access permissions to filesystem objects (files and directories). It may also alter special mode flags. The syntax for this command is as follows:

```
$ chmod [option] mode files
```

For example, `chmod 777` on a file gives permission to everyone to read, write, and execute it.

- `dd`: The `dd` command is used to copy a file, converting and formatting according to the operands. With Android, the `dd` command can be used to create a bit-by-bit image of the Android device. More details about the imaging are covered in *Chapter 5*, *Extracting Data Physically from Android Devices*. Here is the syntax that needs to be used with this command:

```
dd if=/test/file of=/sdcard/sample.image
```

- `rm`: The `rm` command can be used to delete files or directories. Here is the syntax for this command:

```
rm file_name
```

- `grep`: The `grep` command is used to search files or output for a particular pattern. The following example shows searching a `default.prop` file for the word `secure`:

```
 shell@android:/ # cat default.prop | grep secure
ro.secure=1
```

- `pwd`: The `pwd` command displays the current working directory. For example, the following command-line output shows that the current working directory is `/system`:

```
shell@android:/system $ pwd
pwd
/system
```

- `mkdir`: The `mkdir` command is used to create a new directory. The syntax for this command is as follows:

```
mkdir [options] directories
```

- `exit`: The `exit` command can be used to exit the shell you are in. Just type `exit` in the shell to exit from it.

Installing an application

During forensic analysis, there might be cases where you need to install a few applications on the device in order to extract some data. To do so, you can use the `adb.exe install` command. Along with this command, as shown in the following command-line output, you need to specify the path to the `.apk` file that you want to install:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe install
C:\rohit\test.apk
4311 KB/s (13855934 bytes in 3.138s)
    pkg: /data/local/tmp/test.apk
Success
```

However, it is important to note that installing third-party apps may not be accepted in a court of law. Hence, a forensic investigator needs to be cautious before installing any third-party app on the device.

Pulling data from the device

You can use the `adb pull` command to pull the files present on the Android device to the local workstation. Here is the syntax to use this command:

```
adb pull <remote><local>
```

Here, `<remote>` refers to path of the file on the Android device, and `<local>` refers to the location on the local workstation where the file needs to be stored. For instance, the following command-line output shows a `Sample.png` file being pulled from the Android device to a `temp` folder on computer:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe pull /
sdcard/Pictures/MyFolder/Sample.png C:\temp
1475 KB/s (145039 bytes in 0.096s)
```

However, on a normal Android phone, you will not be able to download all the files using the `adb pull` command, because of the inherent security features enforced by the operating system. For example, files present under the `/data/data` folder cannot be accessed in this manner on an Android device that is not rooted. More details about this topic have been covered in *Chapter 4*, *Extracting Data Logically from Android Devices*.

Pushing data to the device

You can use the `adb push` command to copy files from the local workstation to the Android device. Here is the syntax to use this command:

```
adb push <local><remote>
```

Here, `<local>` refers to location of the file on the local workstation, and `<remote>` refers to the path on the Android device where the file needs to be stored. For instance, the following command-line output shows a `test.png` file copied from the computer to the `Pictures` folder of an Android device:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe push
C:\temp\test.png /sdcard/Pictures
2950 KB/s (145039 bytes in 0.048s)
```

You can only push the files to the folders for which the user account has privileges.

Restarting the adb server

In some cases, you might need to terminate the adb server process and then restart it. For example, if adb does not respond to a command. This may resolve the problem.

To stop the adb server, use the `kill-server` command. You can then restart the server by issuing any other adb command.

Viewing log data

In Android, the `logcat` command provides a way to view the system debug output. Logs from various applications and portions of the system are collected in a series of circular buffers which then can be viewed and filtered by this command:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe logcat
--------- beginning of /dev/log/main

I/InputReader( 2841): Touch event's action is 0x0 (deviceType=0) [pCnt=1,
s=0.40234 ]

I/InputDispatcher( 2841): Delivering touch to current input target:
action: 0x0

I/InputDispatcher( 2841): Delivering touch to current input target:
action: 0x0

I/InputDispatcher( 2841): Delivering touch to current input target:
action: 0x0
...
I/SecCamera-JNI-Java( 2841): stopPreview

V/SecCamera-JNI-Cpp( 2841): release camera

V/SecCamera-JNI-Cpp( 2841): release
...
D/STATUSBAR-BatteryController( 3162): onReceive() - ACTION_BATTERY_
CHANGED

D/STATUSBAR-BatteryController( 3162): onReceive() - level:48

D/STATUSBAR-BatteryController( 3162): onReceive() - plugged:2

D/STATUSBAR-BatteryController( 3162): onReceive() - BATTERY_STATUS_
CHARGING:
```

The log message shown here is just a sample message. During investigation, logs need to be carefully analyzed to gather information on location details, data/time information, application details, and so on. Each log begins with a message type indicator, as described in the following table:

Message Type	Description
`V`	Verbose
`D`	Debug
`I`	Information
`W`	Warning
`E`	Error
`F`	Fatal
`S`	Silent

The `logcat` command can also be used to view full cellular radio debugging, as shown in the following command-line output:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe shell
logcat -b radio -v time


03-22 17:06:22.155 E/RIL     (12513): RX: 01
03-22 17:06:22.155 D/RILJ    ( 2815): [UNSL]< UNSOL_RESPONSE_VOICE_
NETWORK_STATE_CHANGED
03-22 17:06:22.155 D/RILJ    ( 2815): [7100]> OPERATOR
03-22 17:06:22.155 D/RILJ    ( 2815): [7101]> DATA_REGISTRATION_STATE
03-22 17:06:22.155 E/RIL     (12513): TX: Time: 1095039892 / 164875824
03-22 17:06:22.155 E/RIL     (12513): TX: M:IPC_NET_CMD S:IPC_NET_
SERVING_NETWORK T:IPC_CMD_GET
 l:7 m:5e a:0
03-22 17:06:22.160 D/RILJ    ( 2815): [7102]> VOICE_REGISTRATION_STATE
03-22 17:06:22.160 D/RILJ    ( 2815): [7103]> QUERY_NETWORK_SELECTION_
MODE
03-22 17:06:22.160 E/RIL     (12513): RX: Time: 1095039894 / 164875826
03-22 17:06:22.160 E/RIL     (12513): RX: M:IPC_NET_CMD S:IPC_NET_
SERVING_NETWORK T:IPC_CMD_RES
P l:12 m:ff a:5e
03-22 17:06:22.160 E/RIL     (12513): RX: 02 02 04 34 30 34 34 39 23 19
79
```

```
03-22 17:06:22.160 D/RILJ    ( 2815): [7100]< OPERATOR {Airtel, Airtel,
40449}
03-22 17:06:22.170 E/RIL     (12513): TX: Time: 1095039906 / 164875839
03-22 17:06:22.170 E/RIL     (12513): TX: M:IPC_NET_CMD S:IPC_NET_REGIST
T:IPC_CMD_GET l:9 m:5f
 a:0
03-22 17:06:22.170 E/RIL     (12513): TX: FF 03
03-22 17:06:22.175 E/RIL     (12513): RX: Time: 1095039909 / 164875841
03-22 17:06:22.175 E/RIL     (12513): RX: M:IPC_NET_CMD S:IPC_NET_REGIST
T:IPC_CMD_RESP l:12 m:
ff a:5f
03-22 17:06:22.175 E/RIL     (12513): RX: 04 03 02 0B 19 79 E1 4A 2E 01
00
03-22 17:06:22.175 E/RIL     (12513): TX: Time: 1095039909 / 164875841
03-22 17:06:22.175 E/RIL     (12513): TX: M:IPC_NET_CMD S:IPC_NET_REGIST
T:IPC_CMD_GET l:9 m:60...
```

Rooting Android

"Rooting" is a word that is very often heard with respect to Android devices. As a forensic examiner, it is essential to understand this in detail. This will help you gain the knowledge that is required to understand the internals of the device. It will also help you gain expertise on several issues that are encountered during an investigation. Rooting Android phones has become a common phenomenon and rooted phones are very often encountered during investigations. Also, depending on the situation and data to be extracted, the examiner himself has to root the device in order to extract certain data. The following sections talk about rooting an Android device and other related concepts.

What is rooting?

To understand rooting, it is essential to understand how Unix-like systems work. The original Unix operating system on which Linux and other Unix-like systems are based was designed from the very beginning as a multiuser system. This is primarily because personal computers did not yet exist, and hence, it was necessary to have a mechanism to separate and protect the resources of the individual users while allowing them to use the system simultaneously. However, in order to perform privileged tasks, such as granting and revoking powers for ordinary users, accessing critical system files to repair or upgrade the system, and so on, it was necessary to have a system administrator account that has superuser access. So we have two types of accounts: normal user accounts, which have fewer privileges, and a superuser or root account, which has all the privileges.

Hence, root is the user name or account that, by default, has access to all commands and files on a Linux or other Unix-like operating system. It is also referred to as the root account, root user, and the superuser. So, in Linux, the root user has the power to start or stop any system service, edit or delete any file, change the privileges of other users, and so on. You have learned that Android uses the Linux kernel, and hence, most of the concepts present in Linux are applicable to Android as well. However, when you buy an Android phone, normally, it does not let you log in as a root user. Rooting an Android phone is all about gaining this root access on the device to perform actions that are not normally allowed on the device.

It is also important to understand the difference between rooting and **jailbreaking**, as they are often wrongly assumed to be the same. Jailbreaking a device that runs Apple iOS allows you to remove certain restrictions and limitations put in place by Apple. For instance, Apple does not allow us to sideload an unsigned application on the device. So, by jailbreaking, you can install applications that are not approved by Apple. In contrast, Android allows sideloading of applications. Jailbreaking a phone involves bypassing several security restrictions simultaneously. Thus, gaining root access on the device is only one of the aspects of jailbreaking a device.

Why root?

Rooting is often performed by many people with the goal of overcoming limitations that carriers and hardware manufacturers put on Android devices. By rooting an Android device, you can alter or replace system applications and settings, run specialized apps that require administrator-level permissions, or perform operations that are otherwise inaccessible to a normal Android user. These actions include uninstalling the default apps (especially the bloatware) that come along with the phone. Rooting is also done for extreme customization; for instance, new customized ROMs could be downloaded and installed. However, from a forensic analysis point of view, the main reason for rooting is to gain access to those parts of the system that are normally not accessible. Most of the public root tools will result in a permanent root, where the changes persist even after rebooting the device. In the case of a temporary root, the changes are lost once the device reboots. Temporary roots should always be preferred in forensic cases.

As explained in *Chapter 1*, *Introducing Android Forensics*, in Linux systems, each user is assigned a **unique user ID** (**UID**), and users are segregated so that one user does not access the data of another user. Similarly, in Android, each application is assigned a UID and is run as a separate process. App UIDs are assigned usually in the order they are installed, starting from 10001. These IDs are stored in the `packages.xml` file in `/data/system`. This file, in addition to storing UIDs, stores the Android permissions of each program as described in its manifest file.

The private data of each application is stored in the `/data/data` location and is accessible only to that application. Hence, during the course of investigation, the data present at this location cannot be accessed. However, rooting a phone will allow you to access the data present in any location. It is important to keep in mind that rooting a phone has several implications, as described here:

- **Security risk**: Rooting a phone might expose the device to security risks. For instance, imagine a malicious app that has access to the entire operating system and to the data of all the other apps installed on the device.
- **Bricking of your device**: If rooting is not done in a proper manner, it might result in the bricking of your device. "Bricking" is a word commonly used with phones that are dead or cannot be turned on in any way.
- **Voiding your warranty**: Depending on the manufacturer and carrier, rooting a device may void your warranty, since it exposes the device to several threats.
- **Forensic implications**: Rooting an Android device will allow an investigator to access a larger set of data, but it involves the alteration of certain portions of the device. Hence, the device should be rooted only when it is absolutely necessary.

Recovery and fastboot

Before dealing with the process of rooting, it is necessary to understand boot loader, recovery, and fastboot modes in Android. The following sections explain these in detail.

Recovery mode

An Android phone can be seen as a device having three main partitions: boot loader, Android ROM, and recovery. The boot loader is present in the first partition and is the first program that runs when the phone is powered on. The primary job of this boot loader is to take care of low-level hardware initialization and boot the device into other partitions. It usually loads the Android partition, commonly referred to as Android ROM, by default. Android ROM contains all the operating system files that are necessary to run the device. The recovery partition, commonly referred to as stock recovery, is the one that is used to delete all user data and files or to perform system updates.

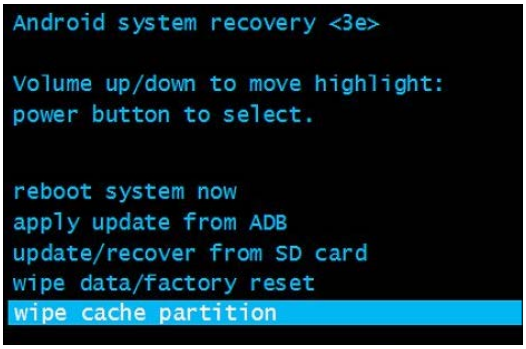
Both of these operations can be started from the running Android system or by manually booting into the recovery mode. For example, when you do a factory reset on your phone, recovery boots up and erases the files and data. Likewise, with updates, the phone boots into the recovery mode to install the latest updates that are written directly to the Android ROM partition. Hence, the recovery mode is the screen that you see when you install any official update on the device.

Accessing the recovery mode

The recovery image is stored on the recovery partition, and it consists of a Linux image with a simple user interface controlled by hardware buttons. The recovery mode can be accessed in two ways:

- By pressing certain combinations of keys when booting the device (usually, by holding volume +, volume -, and power buttons during the bootup)
- By issuing the adb reboot recovery command to a booted Android system

Here is the screenshot of the stock recovery mode on an Android device:



Android stock recovery

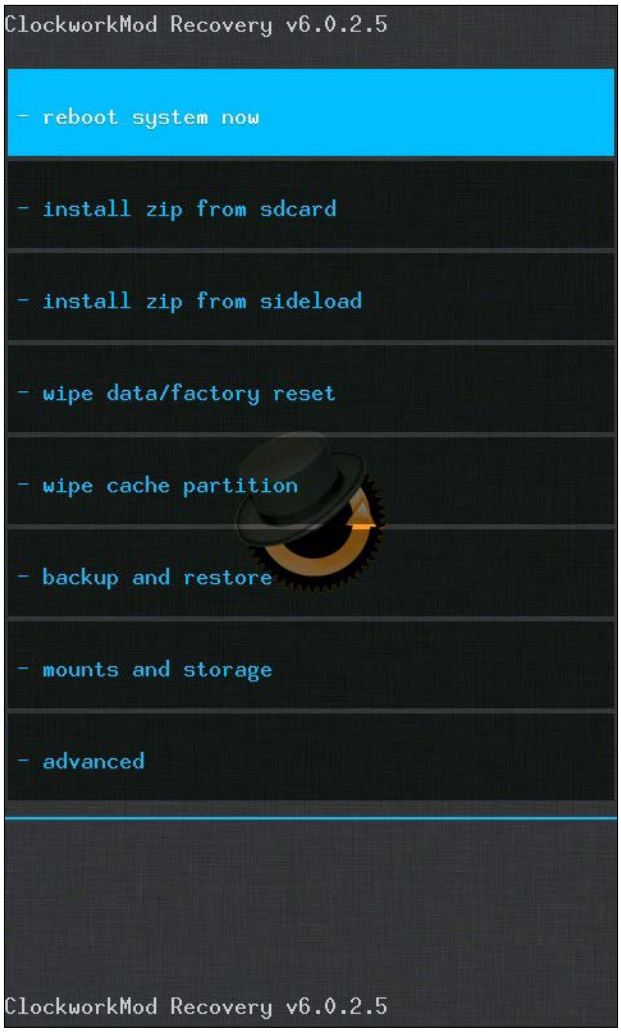
The stock Android recovery is intentionally very limited in functionality. It has the options to reboot the system, apply updates from adb and SD card, factory reset, and so on. However, custom recovery offers many more options.

Custom recovery

Custom recovery is a third-party recovery environment. Flashing this recovery environment onto your device replaces the default stock recovery environment with a third-party, customized recovery environment. These are the most common features that are included in custom recovery:

- Full backup and restore functionality (such as NANDroid)
- Allow unsigned update packages or allow signed packages with custom keys

- Selectively mounts device partitions and SD card
- Provide USB mass storage access to SD card or data partitions
- Provide full ADB access, with the ADB daemon running as root
- Fully featured BusyBox binary (Busybox is a collection of powerful command-line tools in a single binary executable)
- There are several custom recovery images available in the market today, such as ClockworkMod Recovery, TeamWin Recovery Project, and so on. The following screenshot shows the options available with ClockworkMod Recovery v6.0.2.5:



ClockworkMod Recovery

Fastboot mode

Fastboot is a protocol that can be used to reflash partitions on your device. It is one of the tools that comes along with the Android SDK. It is an alternative to the recovery mode to do installations and updates and also to unlock the boot loader in some cases. While in fastboot, you can modify the filesystem images from a computer over a USB connection. Hence, it is one of the ways to install the recovery images and just boot in some cases. Once the phone is booted into fastboot, you can flash image files in the internal memory. For example, the custom recovery images, such as ClockworkMod recovery, discussed earlier can be flashed in this manner. One of the easiest ways to flash the ClockworkMod recovery is through the ROM Manager app. Once this app is installed on a rooted Android device, as shown in the following screenshot, the app provides a **Flash ClockworkMod Recovery** option to install the recovery:



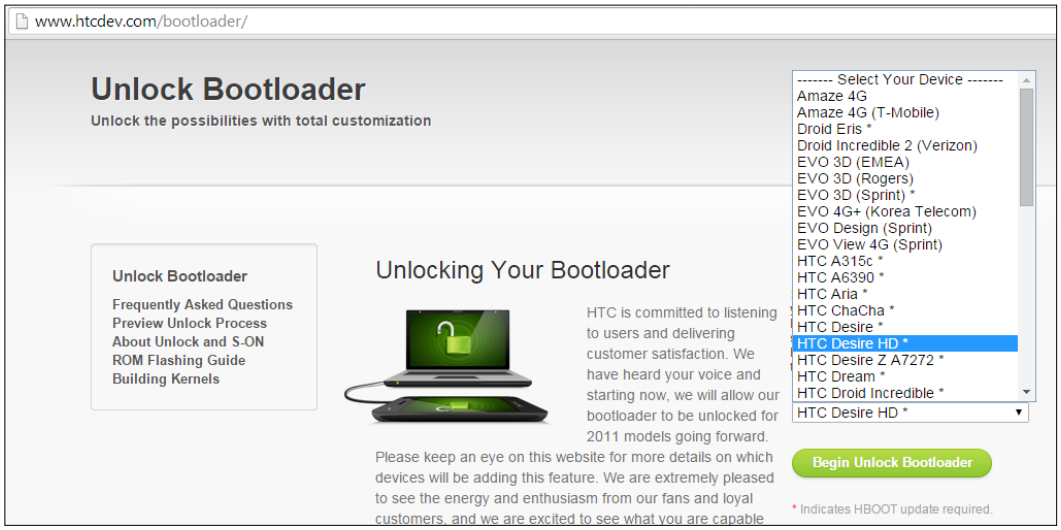
Flashing ClockworkMod Recovery from the ROM Manager app

Locked and unlocked boot loaders

Boot loaders may be locked or unlocked. Locked boot loaders do not allow you to perform modifications to the device's firmware by implementing restrictions at the boot loader level. This is usually done through cryptographic signature verification. Hence, unsigned code cannot be flashed to the device. In other words, in order to run any recovery image or your own operating system, the boot loader needs to be unlocked first. Unlocking the boot loader could result in 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. Thus, the attacker has full access to the data contained on the device. As a result of this, a factory data reset is performed on the phone when unlocking a locked boot loader so that all the data is erased. Hence, it is important to perform this only when it is absolutely necessary. Some devices have ways to unlock them officially. For these devices, boot loader can be unlocked by putting the device into the fastboot mode and running the `fastboot oem unlock` command. This will unlock the boot loader and do a complete wipe of the Android device.

Some other manufacturers provide unlocking through different means, for instance, through their websites and so on. The following screenshot shows the HTC website providing support to unlock HTC devices:



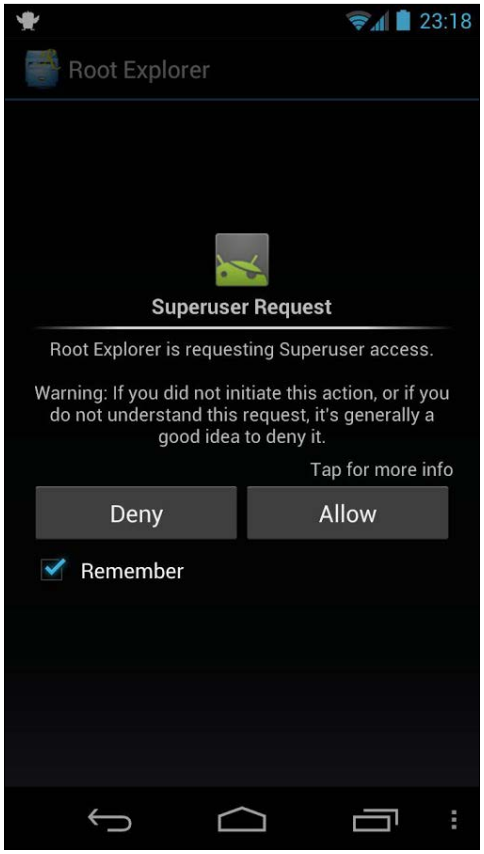
The HTC website providing support to unlock boot loader

How to root

This section talks about how to deal with both a locked and an unlocked boot loader. Gaining root access on a device with an unlocked boot loader is very easy, while gaining root access on a device with a locked boot loader is not so straightforward. The following sections explain this in detail.

Rooting an unlocked boot loader

In Unix-like systems, superuser is a special user account used for system administration and has privileges to access and modify all the files in an operating system. The process of rooting mainly involves copying the **superuser** (**su**) binary to a location in the current process's path (`/system/xbin/su`) and granting it executable permissions with the `chmod` command. Hence, the first step here is to unlock the boot loader. As explained in the *Locked and unlocked boot loaders* section, depending on the device in question, unlocking a boot loader can be done either through the fastboot mode or through following vendor-specific boot loader unlock procedure. The su binary is usually accompanied by an Android application, such as Superuser, that provides a graphical prompt each time an application requests root access, as shown in the following screenshot:



Superuser request

Once the boot loader is unlocked, you can make all the desired changes to the device. Hence, copying the su binary and granting it executable permissions can be done in many ways. The most common method is to boot a custom recovery image. This allows us to copy the su binary into the system partition and set the appropriate permissions through a custom update package.

On an unlocked boot loader device, follow these steps to root the device:

1. Download custom recovery image from `http://www.clockworkmod.com/rommanager` and su update package from `http://superuserdownload.com/`. The custom recovery image can be anything as long as it supports your device. Similarly, the su update package can be SuperSU, SuperUser, or any other package of your choice.
2. Copy both custom recovery image and the su update package to the SD card of the Android device.
3. Next, put the device into fastboot mode.
4. Open the command prompt, and enter the following command:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>
fastboot boot recovery.img
```

5. In the preceding command, `recovery.img` is the recovery image you downloaded.
6. From the `recovery` menu, select the `To apply an update zip file` option and browse to the location on your device where the su binary update package is present.



Since Android 4.1 version, a new feature called the sideload mode has been introduced. This feature allows us to apply an update zip over ADB without copying it to the device beforehand. To sideload an update, run the `adb sideload su-package.zip` command, where `su-package.zip` is the filename of the update package on your computer.

Alternately, you can also modify a factory image to add a su binary. This can be done by unpacking an ext4 formatted system image, adding a su binary, and repacking it. If this image is flashed, it will contain the su binary, and the device will be rooted.



Rooting is a highly device-specific process. Hence, a forensic investigator needs to be cautious before applying these techniques on any Android device.

Rooting a locked boot loader

When the boot loader is locked and cannot be unlocked through any available means, rooting the device requires us to find a security flaw that can be exploited. However, before that, it is important to identify the type of boot loader lock. It can vary depending on the manufacturer and the software version. With some mobiles, fastboot access may not be allowed, but you can still flash using the manufacturer's proprietary flashing protocol, such as Samsung ODIN. Some devices enforce signature verification on selected partitions only, such as boot and recovery. Hence, it may not be possible to boot into custom recovery. However, you can still modify the factory image to include the su binary, as explained in the preceding section.

If the boot loader cannot be unlocked through any means, then the only option is to find some vulnerability on the device that allows us to exploit and add the su binary. The vulnerability can be in an Android kernel, in a process running as root, or any other issue. It is device specific and needs to be researched extensively before trying it on any device. Here are some of the common exploits used in rooting an Android device:

- psneuter
- asroot
- Exploid
- GingerBreak
- RageAgainstTheCage
- Volez
- Levitator
- zergRush
- mempodroid
- Razr blade

ADB on a rooted device

We have already seen how the ADB tool can be used to interact with the device and execute certain commands on the device. However, on a normal Android phone, certain locations, such as `/data/data`, cannot be accessed. For example, the following the command-line output appears when you try to access `/data/data` on a normal device:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe shell
shell@android:/ $ cd /data/data
```

```
cd /data/data
shell@android:/data/data $ ls
ls
opendir failed, Permission denied
```

This is because the private data of all the applications is stored in this folder. Thus, the security is enforced by Android. Only the root user has access to this location. Hence, on a rooted device, you will be able to see all the data under this location, as shown in the following commands:

```
C:\Program Files (x86)\Android\android-sdk\platform-tools>adb.exe shell
shell@android:/ # ls /data/data
ls /data/data
android.googleSearch.googleSearchWidget
com.android.MtpApplication
com.android.Preconfig
com.android.apps.tag
com.android.backupconfirm
com.android.bluetooth
com.android.browser
com.android.calendar
com.android.certinstaller
com.android.chrome
com.android.clipboardsaveservice
com.android.contacts
com.android.defcontainer
com.android.email
com.android.exchange
com.android.facelock
com.android.htmlviewer
com.android.inputdevices
com.android.keychain
com.android.mms
```

As shown in the preceding commands, the private data of all the applications can now be seen easily by navigating to the respective folders. Hence, the ADB tool on a rooted device is very powerful and allows an examiner to access all the data of applications installed on the device. This is possible provided the device is not pattern or PIN protected or registered to the machine with an RSA key.



Sometimes, even on a rooted phone, you will see the permission-denied message. In such cases, after executing the adb shell command, try entering the superuser mode by typing `su`. If root is enabled, you will see # without asking for password.

Summary

Setting up a proper forensic environment is crucial prior to conducting investigation on an Android device. The Android SDK installation is necessary to use tools such as ADB that come along with it. Using ADB, an examiner can communicate with the device, view folders on the device, and pull data and copy data to the device. However, not all folders can be accessed on a normal phone in this manner. This is because the device's security enforcements prevent an examiner from viewing the locations that contain private data. Rooting a device solves this issue, as it provides unlimited access to all the data present on the device. Rooting a device with an unlocked boot loader is straightforward, while rooting a device with a locked boot loader involves exploiting some security bug.

With this knowledge about accessing the device, you will now learn how data is organized on an Android device and many other details in *Chapter 3*, *Understanding Data Storage on Android Devices*.

3
Understanding Data Storage on Android Devices

The primary motive of forensic analysis is to extract necessary data from the device. Hence, for effective forensic analysis, it is imperative to know what kind of data is stored on the device, where it is stored, how it is stored, and the details of the filesystems on which the data is stored. This knowledge is very important to a forensic analyst to take an informed decision about where to look for data and the techniques that can be used to extract the data. In this chapter, we will cover the following topics:

- Android partition layout and file hierarchy
- Application data storage on the device
- An overview of the Android filesystem

Android partition layout

Partitions are logical storage units made inside the device's persistent storage memory. Partitioning allows you to logically divide the available space into sections that can be accessed independently of each other.

Common partitions in Android

The partition layout varies between vendors and versions. However, a few partitions are present in all the Android devices. The following sections explain some of the common partitions found in most of the Android devices.

boot loader

This partition stores the phone's boot loader program. This program takes care of initializing the low-level hardware when the phone boots. Thus, it is responsible for booting the Android kernel and booting into other boot modes, such as the recovery mode, download mode, and so on.

boot

As the name suggests, this partition has the information and files required for the phone to boot. It contains the kernel and RAM disk. So, without this partition, the phone cannot start its processes.

recovery

Recovery partition allows the device to boot into the recovery console through which activities such as phone updates and other maintenance operations are performed. For this purpose, a minimal Android boot image is stored. This boot image serves as a failsafe.

userdata

This partition is usually called the data partition and is the device's internal storage for application data. A bulk of user data is stored here, and this is where most of our forensic evidence will reside. It stores all app data and standard communications as well.

system

All the major components other than kernel and RAM disk are present here. The Android system image here contains the Android framework, libraries, system binaries, and preinstalled applications. Without this partition, the device cannot boot into normal mode.

cache

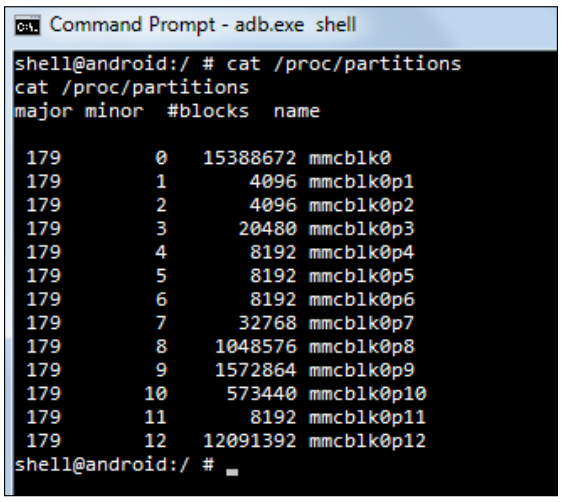
This partition is used to store frequently accessed data and various other files, such as recovery logs and update packages downloaded over the cellular network.

radio

Devices with telephony capabilities have a baseband image stored in this partition that takes care of various telephony activities.

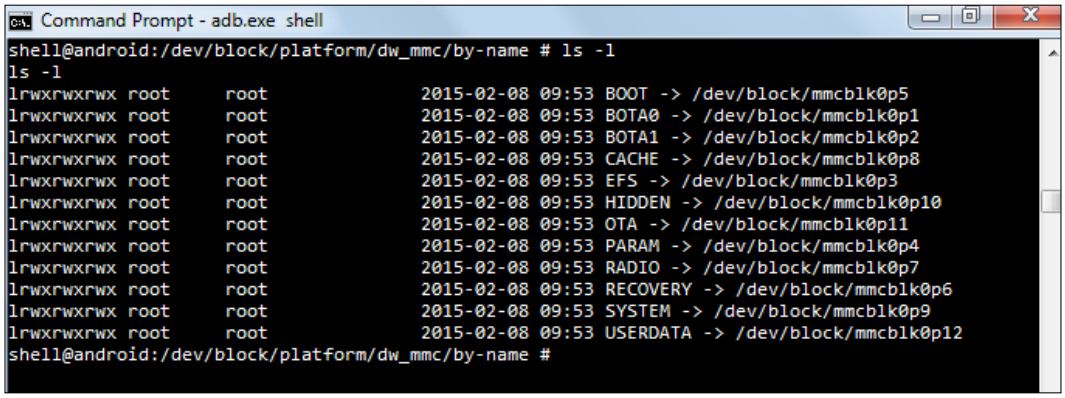
Identifying partition layout

For a given Android device, partition layout can be determined in a number of ways. The `partitions` file under `/proc` gives us details about all the partitions available on the device. The following screenshot shows the contents of the `partitions` file:



Partitions file in Android

The entries in the preceding screenshot show only the block names. To get a mapping of these blocks to their logical functions, check the contents of the `by-name` directory present under `/dev/block/platform/dw_mmc`. The following screenshot shows the contents of this directory:

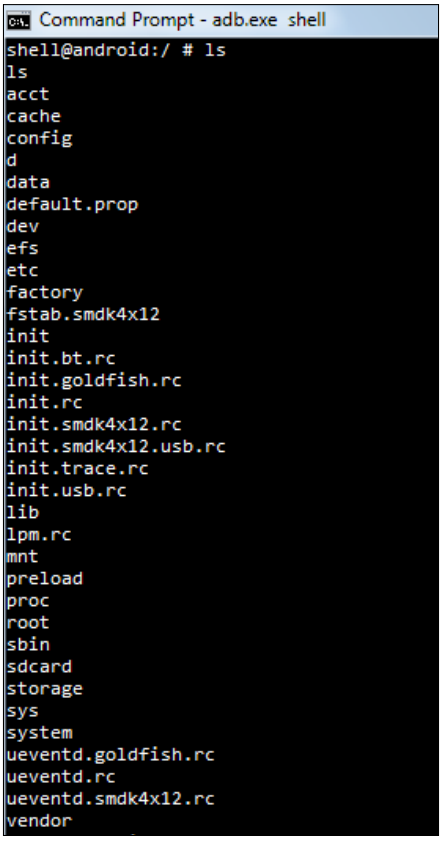


Mapping of blocks to their logical functions

As you can see in the preceding output, various partitions such as system, user data, and so on are present in the partition layout.

Android file hierarchy

In order to perform forensic analysis on any system (desktop or mobile), it's important to understand the underlying file hierarchy. A basic understanding of how Android organizes its data in files and folders helps a forensic analyst narrow down his research to specific locations. If you are familiar with Unix-like systems, you will understand the file hierarchy in Android very well. In Linux, the file hierarchy is a single tree, with the top of the tree being denoted as `/`. This is called the **root**. This is different from the concept of organizing files in drives (as with Windows). Whether the filesystem is local or remote, it will be present under the root. Android file hierarchy is a customized version of this existing Linux hierarchy. Based on the device manufacturer and the underlying Linux version, the structure of this hierarchy may have a few insignificant changes. To see the complete file hierarchy, you need to have root access. The following screenshot shows the file hierarchy on an Android device:



Folders present under / (root) in Android

An overview of directories

The following sections provide an overview of the directories present in the file hierarchy of an Android device.

acct

This is the mount point for the acct cgroup (control group) that provides for user accounting.

cache

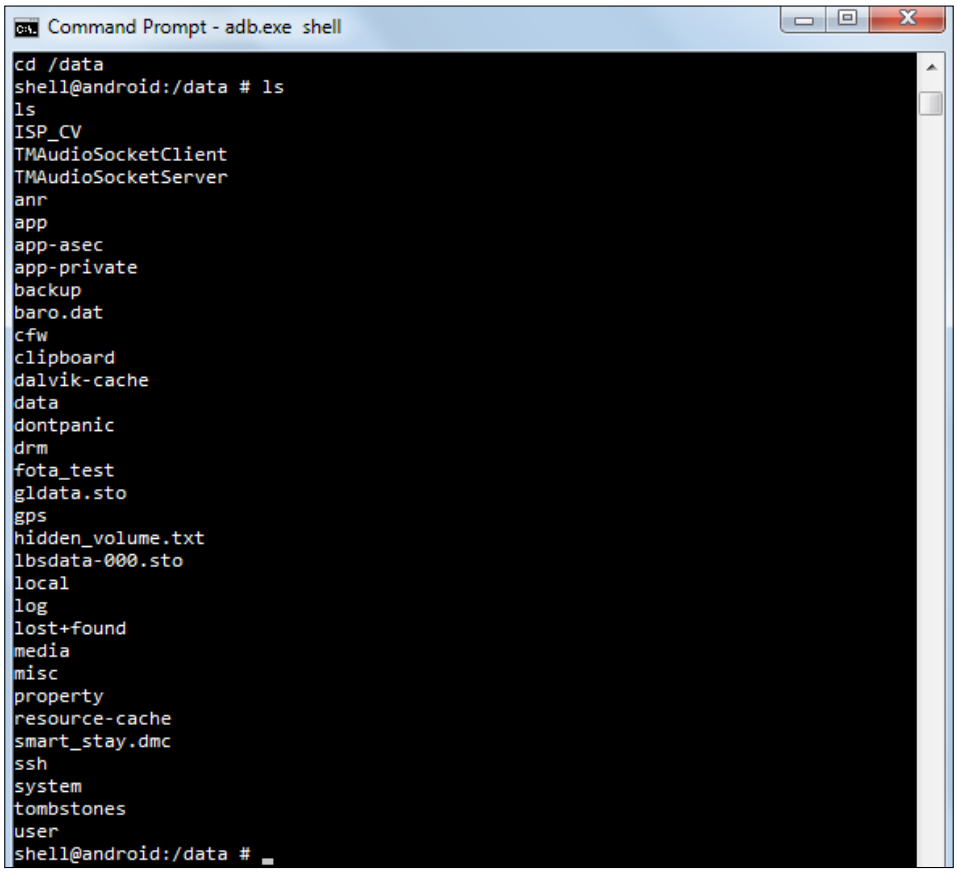
This is the directory (`/cache`) where Android stores frequently accessed data and app components. Wiping the cache doesn't affect your personal data, but simply deletes the existing data there. There is also another directory in this folder called `lost+found`. This directory holds recovered files (if any) in the event of filesystem corruption, such as incorrectly removing the SD card without unmounting it and so on. The cache may contain forensically relevant artifacts, such as images, browsing history, and other app data.

d

This is a symbolic link to `/sys/kernel/debug`. This folder is used to mount the debugfs filesystem and to debug kernel.

data

This is the partition that contains the data of each application. Most of the data belonging to a user, such as the contacts, SMS, dialed numbers, and so on, is stored in this folder. This folder has significant importance from a forensic point of view as it holds valuable data. The following screenshot shows the folders present in this partition:



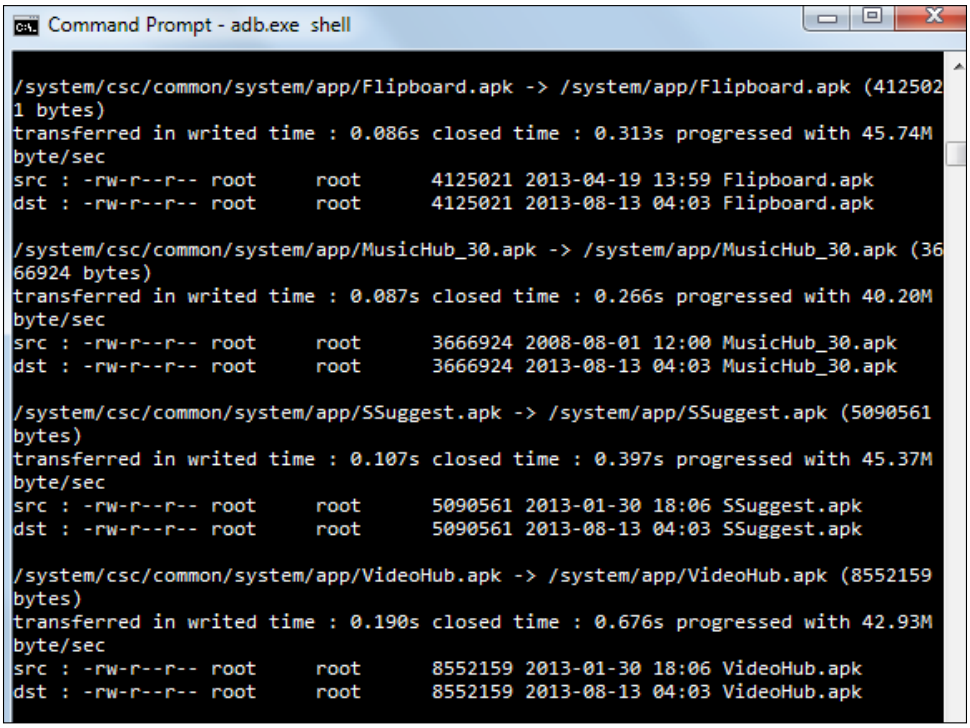
Contents of data partition of an Android device

The following sections provide a brief explanation of other important subdirectories present under the `data` folder.

dalvik-cache

As discussed in *Chapter 1*, *Introducing Android Forensics*, Android applications contain `.dex` files that are optimized versions of Java bytecode. When an application is installed on an Android device, some modifications are performed on the corresponding `.dex` file, and a resultant file called `.odex` file (optimized `,dex` file) is created. It is then cached in the `/data/dalvik-cache` directory so that it doesn't have to perform the optimization process every time it loads `application.log`.

This folder contains several logs that might be useful during examination, depending on the underlying requirements. For example, the following screenshot shows one of the log files `recovery_log.txt`, which gives details about the recovery log:



The recovery_log.txt file output

data

The `/data/data` partition contains the private data of all the applications. Most of the data belonging to the user is stored in this folder. This folder has significant importance from a forensic point of view as it holds valuable data. This partition is covered in detail in the *Internal Storage* section.

dev

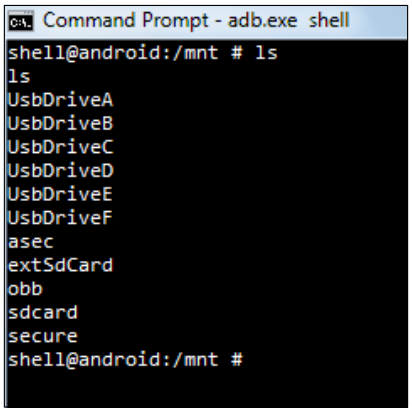
This directory contains special device files for all the devices. This is the mount point for the tempfs filesystem. This filesystem defines the devices available to the applications.

init

As discussed in *Chapter 1*, *Introducing Android Forensics*, when booting the Android kernel, the init program is executed. This program present under this folder.

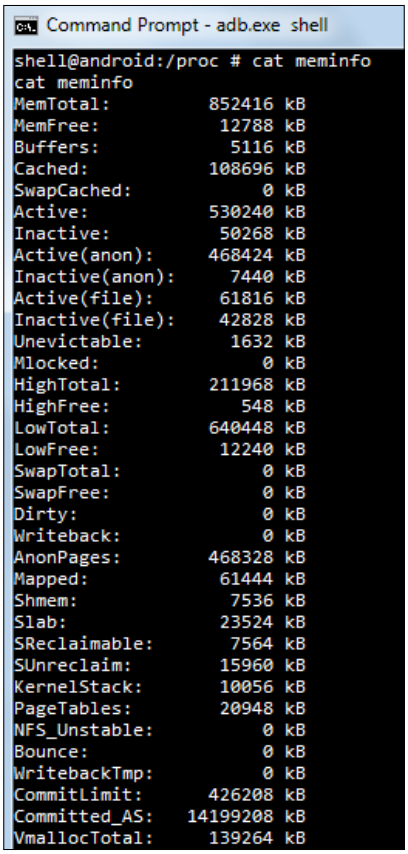
mnt

This directory serves as a mount point for all the filesystems, internal and external SD cards, and so on. The following screenshot shows the mount points present in this directory:



proc

This is the mount point for the procfs filesystem that provides access to the kernel data structures. Several programs use `/proc` as the source for their information. It contains files that have useful information about the processes. For instance, as shown in the following screenshot, `meminfo` present under `/proc` gives information about the memory allocation:



meminfo file under proc folder in Android

root

This is the home directory for the root account. This folder can be accessed only if the device is rooted.

sbin

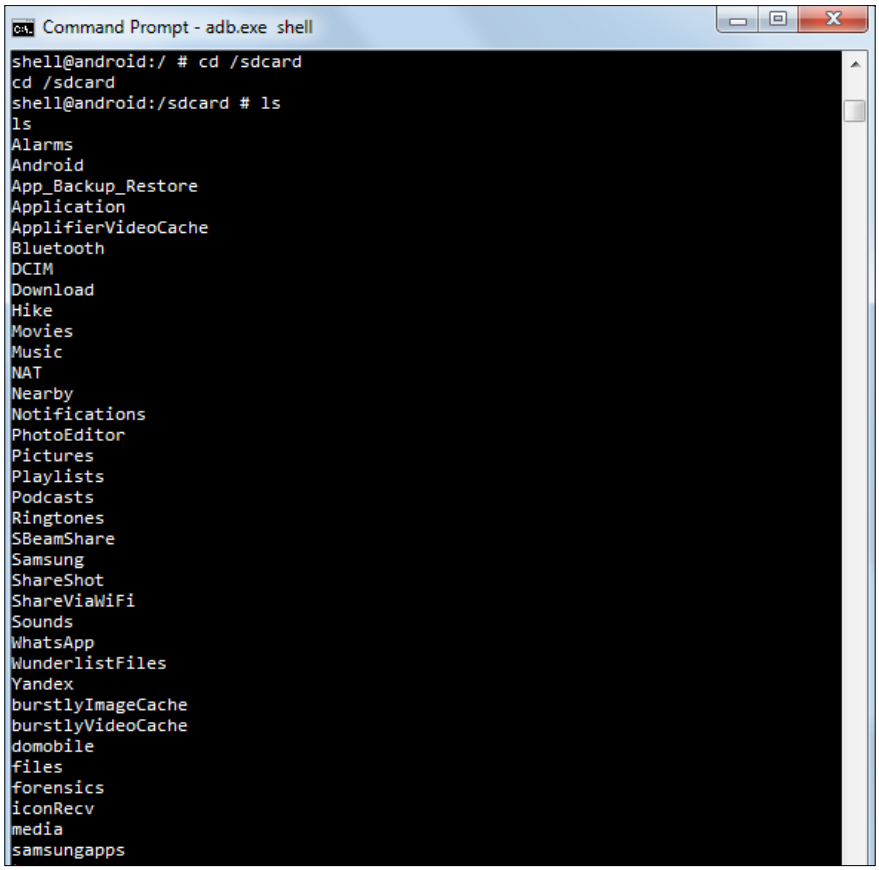
This contains binaries for several important daemons. This is not of much significance from a forensic perspective.

misc

As the name suggests, this folder contains information about miscellaneous settings. These settings mostly define the state, that is, ON/OFF. Information about hardware settings, USB settings, and so on can be accessed from this folder.

sdcard

This is the partition that contains the data present on the SD card of the device. Note that this SD card can be either removable storage or non-removable storage. Any app on your phone with the `WRITE_EXTERNAL_STORAGE` permission may create files or folders in this location. There are some default folders, such as `android_secure`, `Android`, `DCIM`, `media`, and so on, present in most of the mobiles. The following screenshot shows the contents of `/sdcard` location:



Contents of the sdcard partition of an Android device

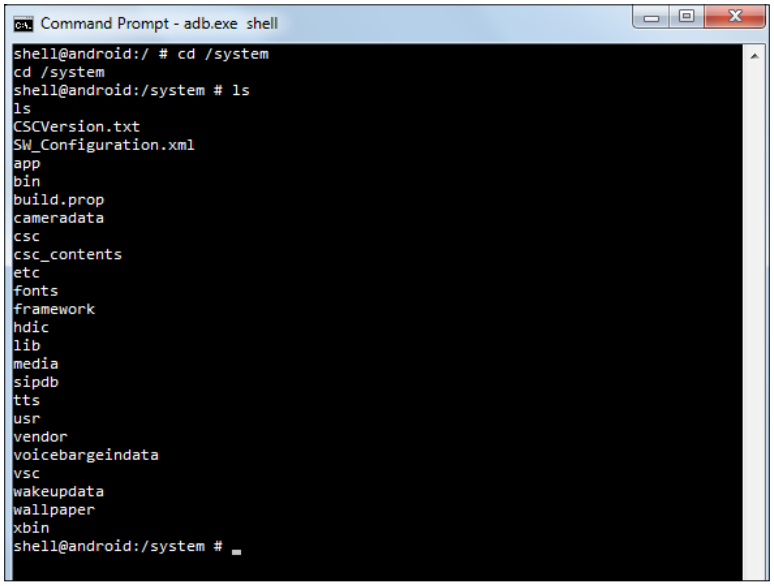
Digital Camera Images (**DCIM**) is the default directory structure for digital cameras, smartphones, tablets, and related solid-state devices. Some tablets have a `photos` folder that points to the same location. Within `DCIM`, you will find photos you have taken, videos, and thumbnails (cache) files. Photos are stored in `/DCIM/Camera`.

Android developer's reference explains that there are certain public storage directories that are not specifically tied to a specific program. Here is a quick overview of these folders:

- `Music`: Media scanner classifies all media found here as user music.
- `Podcasts`: Media scanner classifies all media found here as a podcast.
- `Ringtones`: Media files present here are classified as ringtones.
- `Alarms`: Media files present here are classified as alarms.
- `Notifications`: Media files under this location are used for notification sounds.
- `Pictures`: All photos, except the ones taken with a camera, are stored in this folder.
- `Movies`: All movies, except the ones taken with a camera, are stored in this folder.
- `Download`: Miscellaneous downloads are stored in this folder.

system

This directory contains libraries, system binaries, and other system-related files. The pre-installed applications that come along with the phone are also present in this partition. The following screenshot shows the files present in the system partition on an Android device:

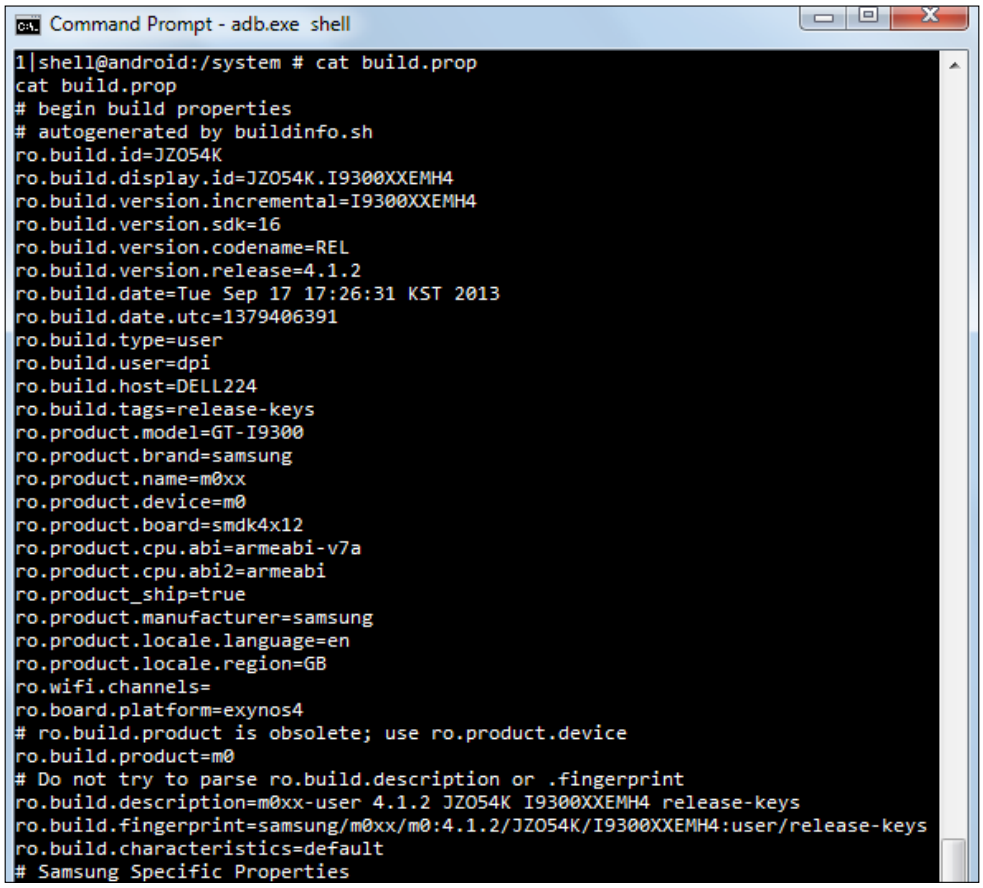


Contents of the system partition of an Android device

Here are some of the interesting files and folders present in the `/system` partition that are of interest to a forensic investigator.

build.prop

This file contains all the build properties and settings for a given device. For a forensic analyst, this file gives an overview about the device model, manufacturer, Android version, and many other details. Contents of this file can be viewed by issuing a `cat` command, as shown in the following screenshot:

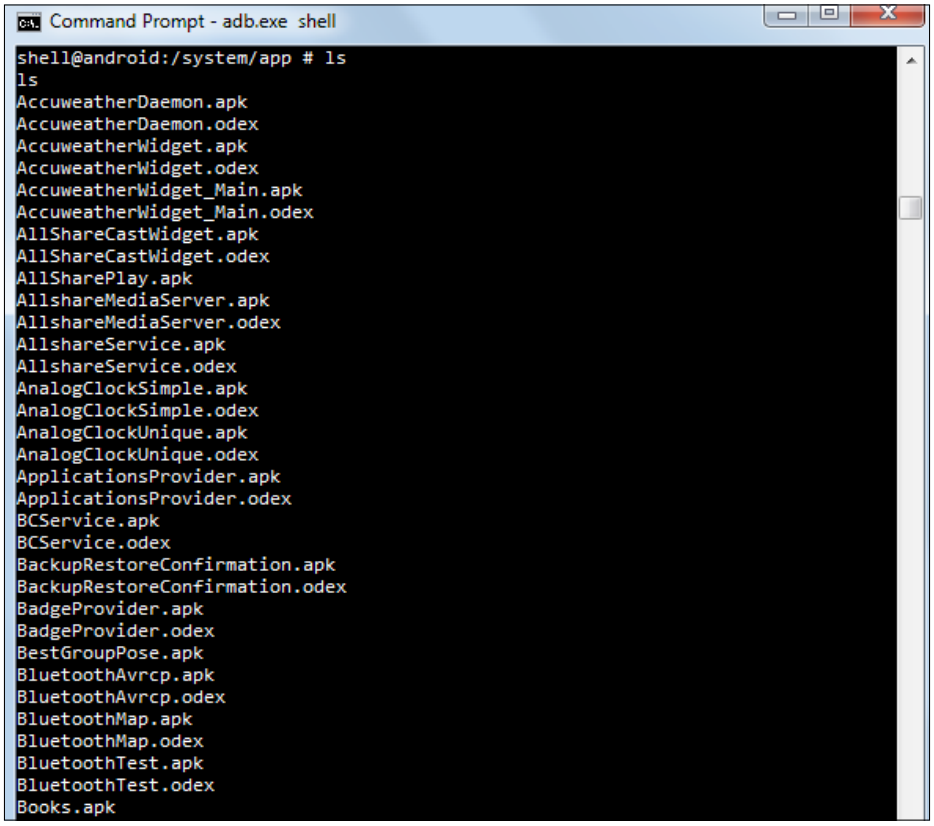


The build.prop file output

As shown in the preceding output, you can find out the product model, CPU details, and Android version by viewing this file content. On a rooted device, tweaking the `build.prop` file could lead to a change in several system settings.

app

This folder contains system apps and preinstalled apps. This is mounted as read only to prevent any changes. The following screenshot shows various system-related apps that are present in this folder:



System apps present under the /system/app partition

Along with the APK files, you might have also noticed `.odex` files in the preceding output. In Android, applications come in packages, with the `.apk` extension. These APKs contain `.odex` files whose supposed function is to save space. The `.odex` files are collection of certain parts of an application that are optimized before booting.

framework

This folder contains the sources for the Android framework. In this partition, you can find the implementation of key services, such as the system server with the package and activity managers. A lot of the mapping between the Java application APIs and the native libraries is also done here.

ueventd.goldfish.rc and ueventd.rc

These files contain configuration rules for the `/dev` directory.

To sum up, here is a screenshot of the Android file tree reference from `http://wiki.robotz.com/index.php/Android_File_System`:

```
device
|
+---acct
+---cache
+---config
+---d
+---data
|   +---app
+---dev
+---efs
+---etc
+---factory
+---lib
+---mnt
|   +---asec
|   +---extSdCard
|   +---obb
|   +---sdcard
|   |   +DCIM
|   +---secure
|   +---UsbDriveA
|   +---UsbDriveB
|   +---UsbDriveC
|   +---UsbDriveD
|   +---UsbDriveE
|   +---UsbDriveF
+---preload
+---proc
+---root
+---sbin
+---sdcard
+---storage
+---sys
+---system
|   +---app
|   +---bin
|   +---cameradata
|   +---csc
|   +---etc
|   +---fonts
|   +---framework
|   +---hdic
|   +---lib
|   +---media
|   +---T9DB
|   +---tts
|   +---usr
|   +---vendor
|   +---vsc
|   +---wallpaper
|   +---xbin
+---vendor
```

Android file tree

Application data storage on the device

Android devices store a lot of sensitive data through the use of apps. Although we have earlier categorized apps as system and user-installed apps, here is a more detailed split:

- Apps that come along with Android
- Apps installed by the manufacturer
- Apps installed by a wireless carrier
- Apps installed by the user

All of these store different types of data on the device. Application data often contains a wealth of information that is relevant to the investigation. Here is a sample list of possible data that can be found on an Android device:

- SMS
- MMS
- Chat messages
- Backups
- E-mails
- Call logs
- Contacts
- Pictures
- Videos
- Browser history
- GPS data
- Files or documents downloaded
- Data that belongs to installed apps (Facebook, Twitter, and other social media apps)
- Calendar appointments

Data belonging to different applications can be stored either internally or externally. In the case of external storage (SD card), data can be stored in any location. However, in the case of internal storage, the location is predefined. To be specific, internal data of all apps present on the device (either system apps or user-installed apps) is automatically saved in the `/data/data` subdirectory, named after the package name. For example, the default Android e-mail app has a package named `com.android.email`, and the internal data is stored in `/data/data/com.android.email`. We will discuss this in detail in the upcoming sections, but for now, this knowledge will be sufficient to understand the following details.

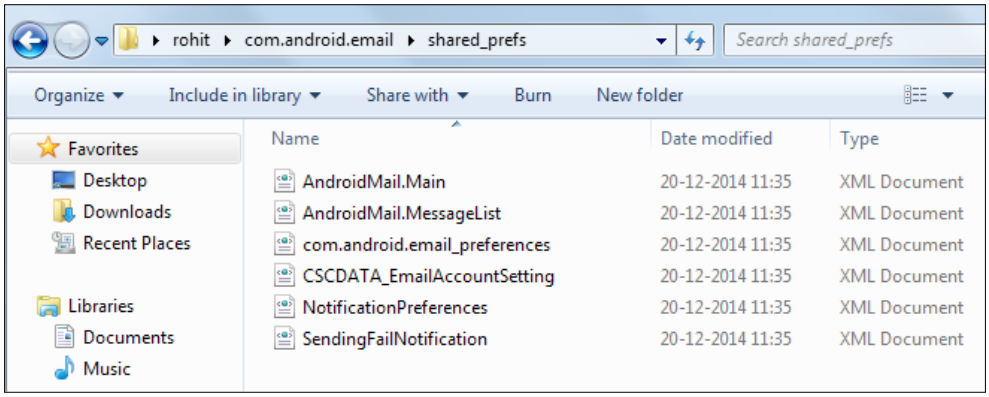
Android provides developers with certain options to store data to the device. The option that can be used depends on the underlying data that is to be stored. Data that belongs to applications can be stored in one of the following locations:

- Shared preferences
- Internal storage
- External storage
- SQLite database
- Network

The following sections provide a clear explanation about each of these options.

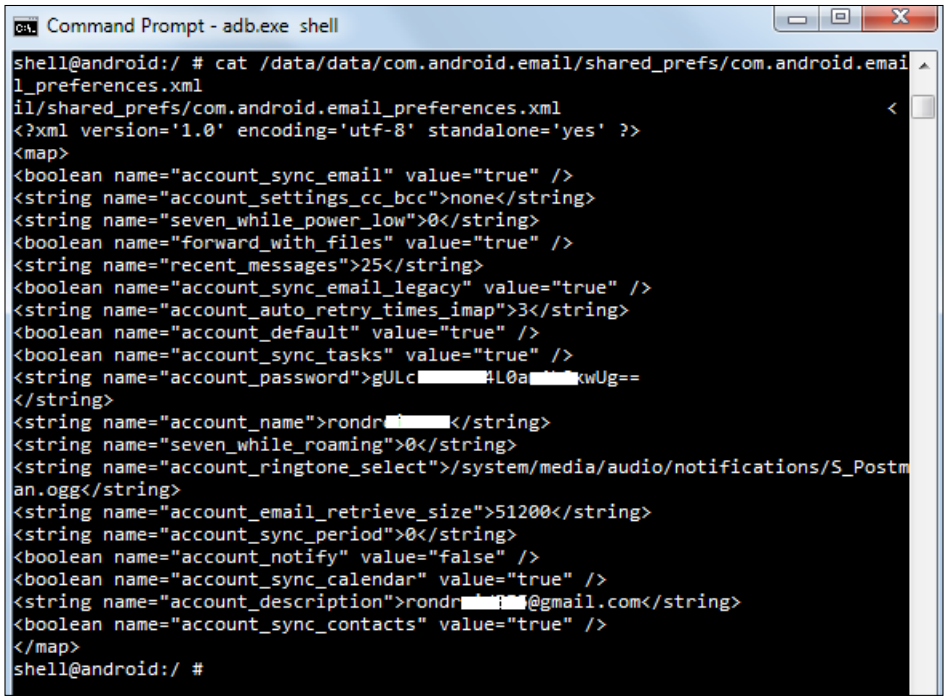
Shared preferences

This location provides a framework to store key-value pairs of primitive data types in the `.xml` format. Primitive data types include boolean, float, int, long, and string. Strings are stored in the **Universal Character Set Transformation Format-8** (**UTF-8**) format. These files are typically stored in the application's `/data/data/<package_name>/shared_prefs` path. For instance, the `shared_prefs` folder for the Android e-mail app contains less than six `.xml` files, as shown in the following screenshot:



Contents of the `shared_prefs` folder of the Android e-mail app

As explained in *Chapter 2, Setting Up an Android Forensic Environment*, contents of these files can be viewed using the `cat` command. The following screenshot shows the contents of the `com.android.email_preferences.xml` file:



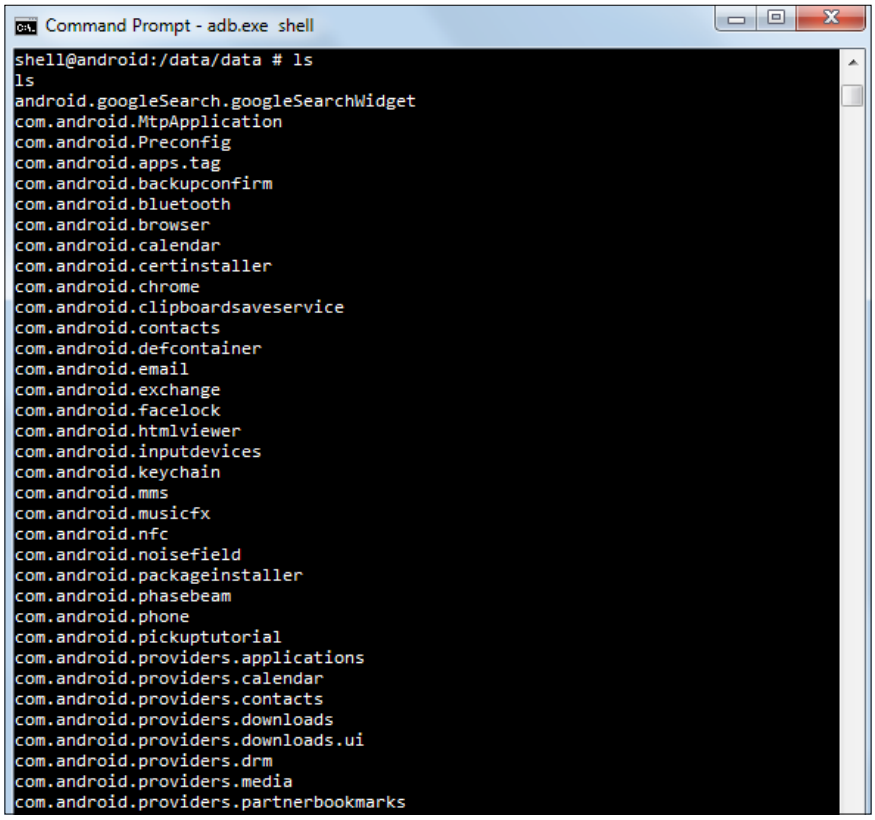
Android e-mail app's shared preferences file content

As seen in the preceding screenshot, the data is stored in name-value pairs. Perhaps in the preceding `.xml` file, `account_name`, `account_password`, `recent_messages` are some of the interesting parameters from a forensic point of view. Many applications use shared peferences to store sensitive data, because it is light weight. Thus, they can be a key source of information during a forensic investigation.

Internal storage

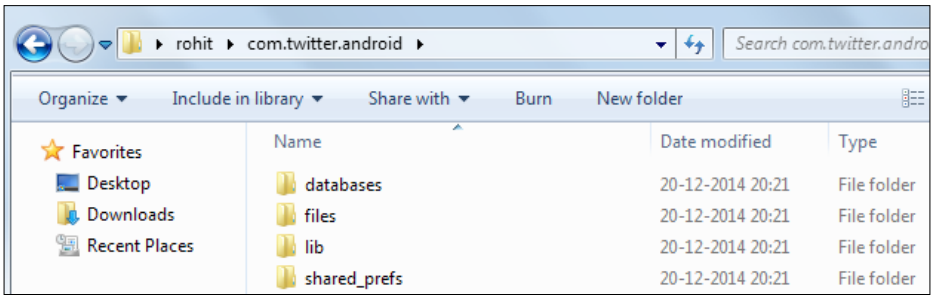
The files here are stored in the internal storage. These files are located typically in the application's `/data/data` subdirectory. Data stored here is private and cannot be accessed by other applications. Even the device owner is prevented from viewing the files (unless they have root access). However, based on the requirement, the developer can allow other processes to modify and update these files.

The following screenshot shows the details of the apps stored with their package name in the `/data/data` directory:



Contents of the `/data/data` folder in Android

Internal data of each app is stored in their respective folders. For instance, the following screenshot shows the internal storage that belongs to the Twitter app on an Android device:

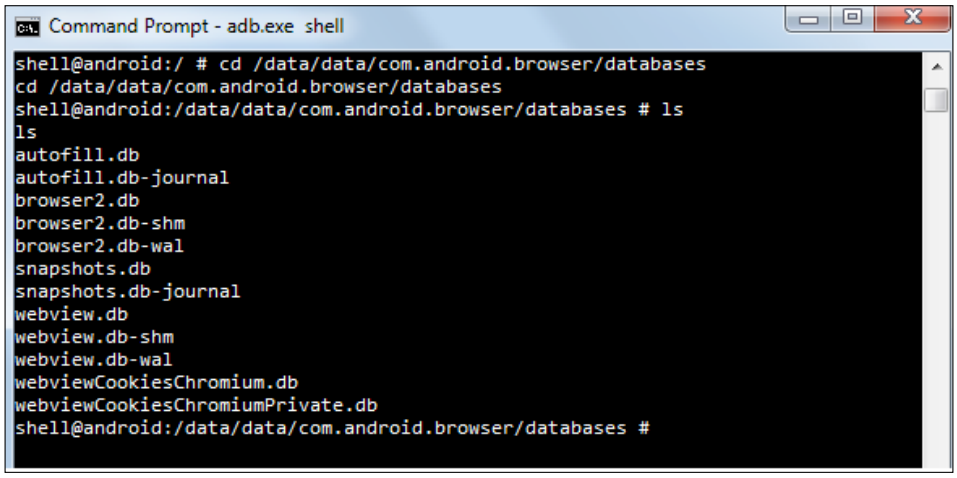


Internal storage of the Android Twitter app

Usually, the `databases`, `lib`, `shared_pref`, `cache` folders are created for most of the applications. The following table provides a brief description of these folders:

Sub directory	**Description**
shared_prefs	XML file of shared preferences
lib	Custom library files required by app
files	Developer-saved files
cache	Files cached by app
databases	SQLite and journal files

Folders other than these are custom folders created by the app developer. The `databases` folder is the one that contains crucial data that helps in forensic investigations. As shown in the following screenshot, data in this folder is stored in SQLite files:



SQLite files present under the databases folder of the Android browser app

This data can be viewed using tools such as SQLite Browser. More details about how to extract data is covered in detail in *Chapter 4*, *Extracting Data Logically from Android Devices*.

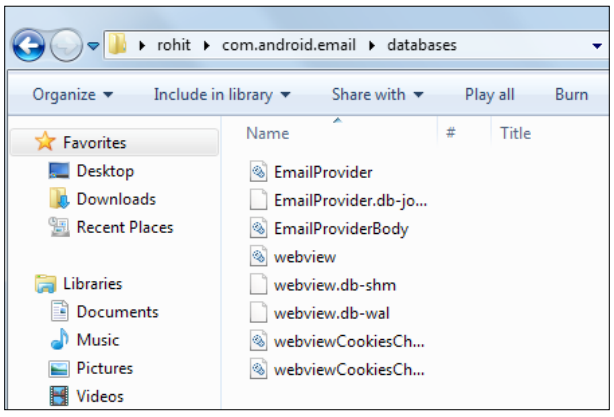
External storage

Files can also be stored by the apps in external storage. External storage can be a removable media, such as an SD card or non-removable storage that comes with the phone. In the case of a removable SD card, data can be used on other devices just by removing the SD card and inserting it in any other device. SD cards are usually formatted with the FAT32 filesystem, but other filesystems, such as EXT3 and EXT4, are also being used increasingly. Unlike internal storage, external storage does not have strict security enforcements. In other words, data stored here is public and can be accessed by other applications, provided the requesting apps have the necessary permissions.

For example, the Twitter app discussed earlier also stores certain files on the SD card in the `/Android/data` location. Large files, such as images and videos, loaded by the apps are usually stored in the external storage for faster retrieval.

SQLite database

SQLite is a popular database format present in many mobile systems and is used for structured data storage. SQLite is open source, and unlike many other databases, it is compact and offers lot of functionality. Android supports SQLite through dedicated APIs, and hence, developers can take advantage of it. SQLite databases are a rich source of forensic data. The SQLite files used by the apps are generally stored at `/data/data/<ApplicationPackageName>/databases`. For example, in the case of the Android e-mail app, the following screenshot shows the SQLite files present in its `databases` folder. We will examine these files in more detail in the upcoming sections. From a forensic point of view, they are highly valuable since they often store a lot of important data handled by the application. The contents of the `databases` folder can be seen in the following screenshot:



The databases folder of the Android e-mail app

Network

You can use the network to store and retrieve data on your own web-based services. To do network operations, the classes in the `java.net.*` and `android.net.*` packages can be used. These packages provide developers with the low-level APIs that are necessary to interact with the network, web servers, and so on.

Android filesystem overview

Understanding the filesystem is very important in Android forensics, as it helps us gain knowledge of how the data is stored and retrieved. This knowledge about properties and the structure of a filesystem will prove to be useful during forensic analysis. Filesystem refers to the way data is stored, organized, and retrieved from a volume. A basic installation may be based on one volume split into several partitions; here, each partition can be managed by a different filesystem. Microsoft Windows users are mostly familiar with the FAT32 or NTFS filesystem, whereas Linux users are more familiar with the EXT2 or EXT4 filesystem. As is true in Linux, Android also utilizes mount points and not drives (that is `C:` or `E:`). Each filesystem defines its own rules to manage the files on the volume. Depending on these rules, each filesystem offers a different speed for file retrieval, security, size, and so on. Linux uses several filesystems and so does Android. From a forensic point of view, it's important to understand what filesystems are used by Android and to identify the filesystems that are of significance to the investigation. For example, the filesystem that stores the user's data is of primary concern to us, as opposed to a filesystem used to boot the device.

As mentioned earlier, Linux is known to support a large number of filesystems. These filesystems used by the system are not accessed by drive names, but instead are combined into a single hierarchical tree structure that represents these filesystems as a single entity. Each new filesystem is added into this single filesystem tree when it is mounted.



In Linux, mounting is an act of attaching an additional filesystem to the currently accessible filesystem of a computer.

Thus, the filesystems are mounted on to a directory, and files present in this filesystem are now the contents of that directory. This directory is called a **mount point**. It makes no difference whether the filesystem exists on the local device or on a remote device. Everything is integrated into a single file hierarchy that begins with root. Each filesystem has a separate kernel module that registers the operations that it supports with something called **virtual file system** (**VFS**). VFS allows different applications to access different filesystems in a uniform way. By separating the implementation from the abstraction, adding a new filesystem becomes a matter of writing another kernel module. These modules are either part of the kernel or are dynamically loaded on demand. The Android kernel comes with a subset of a vast collection of filesystems that range from the **Journal File System** (**JFS**) to the Amiga filesystem. All the background work is handled by the kernel when a filesystem is mounted.



The preceding information is referenced from `http://trikandroid.hol.es/page/100/`.

Viewing filesystems on an Android device

The filesystems supported by the Android kernel can be determined by checking the contents of the `filesystems` file that are present in the `proc` folder. The content of this file can be viewed using the following command:

```
shell@Android:/ $ cat /proc/filesystems
cat /proc/filesystems
nodev sysfs
nodev rootfs
nodev bdev
nodev proc
nodev cgroup
nodev tmpfs
nodev binfmt_misc
nodev debugfs
nodev sockfs
nodev usbfs
nodev pipefs
nodev anon_inodefs
nodev devpts
ext2
ext3
```

```
ext4
nodev ramfs
vfat
msdos
nodev ecryptfs
nodev fuse
fuseblk
nodev fusectl
exfat
```

In the preceding output, the filesystems preceded by the `nodev` property are not mounted on the device.

Common Android filesystems

The filesystems present in Android can be divided into three main categories, which are as follows:

- Flash memory filesystems
- Media-based filesystems
- Pseudo filesystems

Flash memory filesystems

Flash memory is a type of constantly-powered non-volatile memory that can be erased and reprogrammed in units of memory called blocks. Due to the particular characteristics of flash memories, special filesystems are needed to write over the media and deal with the long erase times of certain blocks. While the supported filesystems vary on different Android devices, the common flash memory filesystems are as follows:

- **Extended File Allocation Table** (**exFAT**): This type of filesystem is a Microsoft proprietary filesystem optimized for flash drives. As a result of the license requirements, it is not part of the standard Linux kernel. However, a few manufacturers provide support for this filesystem.
- **Flash Friendly File System** (**F2FS**): This type of filesystem is introduced by Samsung as an open source filesystem. The basic intention was to build a filesystem that takes into account the characteristics of the storage devices based on the NAND flash memory.

- **Journal Flash File System version 2** (**JFFS2**): This type of filesystem is a log-structured filesystem used in Android. JFFS2 is the default flash filesystem for **Android Open Source Project** (**AOSP**) since the Ice Cream Sandwich version. Filesystems such as LogFS, UBIFS, YAFFS, and so on have been developed as a replacement for JFFS2.
- **Yet Another Flash File System version 2** (**YAFFS2**): This type of filesystem is an open source, single-threaded filesystem that was released in 2002. It is mainly designed to be fast when dealing with NAND flash. YAFFS2 utilizes OOB. This is often not captured or decoded correctly during forensic acquisition, which makes analysis difficult. YAFFS2 was the most popular release at one point and is still widely used in Android devices. YAFFS2 is a log-structured filesystem. Data integrity is guaranteed even in the case of sudden power outage. In 2010, there was an announcement stating that in releases after Gingerbread, devices were going to move from YAFFS2 to EXT4. Currently, YAFFS2 is not supported in newer kernel versions, but certain mobile manufacturers might still continue to support it.
- **Robust File System** (**RFS**): This type of filesystem supports the NAND flash memory on Samsung devices. RFS can be summarized as a FAT16 (or FAT32) filesystem where journaling is enabled through a transaction log. Many users complain that Samsung should stick to EXT4. RFS has been known to have lag times that slow down the features of Android.

Media-based filesystems

Besides the flash memory filesystems discussed earlier, Android devices typically support the following media-based filesystems:

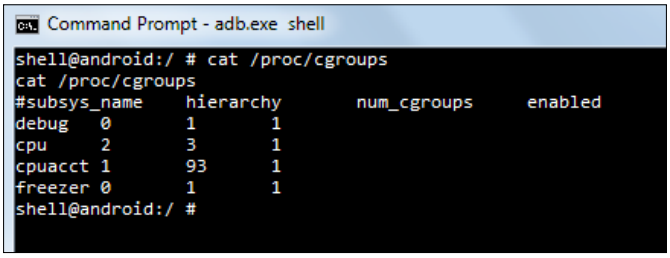
- **EXTended file system** (**EXT2/EXT3/EXT4**): This filesystem was introduced in 1992 specifically for the Linux kernel. This was one of the first filesystems and used the virtual filesystem. EXT2, EXT3, and EXT4 are the subsequent versions. Journaling is the main advantage of EXT3 over EXT2. With EXT3, in the case of an unexpected shutdown, there is no need to verify the filesystem. The EXT4 filesystem, the fourth extended filesystem, has gained significance with mobile devices that implement dual-core processors. The YAFFS2 filesystem is known to have a bottleneck on dual-core systems. With the Gingerbread version of Android, the YAFFS filesystem was swapped for EXT4.
- **File Allocation Table** (**FAT**): These filesystems, such as FAT12, FAT16, and FAT32, are supported by the MSDOS driver.

- **Virtual File Allocation Table** (**VFAT**): This filesystem is an extension of the FAT16 and FAT32 filesystems. Microsoft's FAT32 filesystem is supported by most Android devices. It is supported by almost all the major operating systems, including Windows, Linux, and Mac OS. This enables these systems to easily read, modify, and delete the files present on the FAT32 portion of the Android device. Most of the external SD cards are formatted using the FAT32 filesystem.

Pseudo filesystems

In addition to these, there are pseudo filesystems that can be thought of as logical groupings of files. Here are some of the important pseudo filesystems found in an Android device:

- **control group** (**cgroup**): This type of pseudo filesystem provides a way to access and define several kernel parameters. There are a number of different process-control groups present. As shown in the following command-line output, the list of groups can be seen in the `/proc/cgroups` file:



The cgroups file output

Android devices use this filesystem to track their jobs. They are responsible for aggregating the tasks and keeping track of them.

- **rootfs**: This type of filesystem is one of the main components of Android and contains all the information required to boot the device. When the device starts the boot process, it needs access to many core files and, thus, mounts the root file system. This filesystem is mounted at `/` (root folder). Hence, this is the filesystem on which all the other filesystems are slowly mounted. If this filesystem is corrupt, the device cannot be booted.

- **procfs**: This type of filesystem contains information about kernel data structures, processes, and other system-related information in the `/proc` directory. For instance, the `/proc/filesystems` file displays the list of available filesystems on the device. The following command shows all the information about the CPU of the device:

```
shell@Android:/ $ cat /proc/cpuinfo
cat /proc/cpuinfo
Processor : ARMv7 Processor rev 0 (v7l)
processor : 0
BogoMIPS : 1592.52
processor : 3
BogoMIPS : 2786.91
Features : swp half thumb fastmult vfp edsp neon vfpv3 tls
CPU implementer : 0x41
CPU architecture: 7
CPU variant : 0x3
CPU part : 0xc09
CPU revision : 0
Chip revision : 0011
Hardware : SMDK4x12
Revision : 000c
Serial : ****************
```

- **sysfs**: This type of filesystem mounts the `/sys` folder, which contains information about the configuration of the device. The following output shows various folders in the `sys` directory in an Android device:

```
shell@Android:/ $ cd /sys
cd /sys
shell@Android:/sys $ ls
ls
block
bus
class
dev
devices
firmware
fs
```

```
kernel
module
power
```

Since the data present in these folders is mostly related to configuration, this is not usually of much significance to a forensic investigator. However, there could be some circumstances where we might want to check whether a particular setting was enabled on the phone. Analyzing this folder could be useful under such conditions. Note that each folder consists of a large number of files. Capturing this data through forensic acquisition is the best method to ensure that this data is not changed during examination.

- **tmpfs**: This type of filesystem is a temporary storage facility on the device that stores the files in RAM (volatile memory). This is often mounted on the `/dev` directory. The main advantage of using RAM is its faster access and retrieval. However, once the device is restarted or switched off, this data will not be accessible anymore. Hence, it's important for a forensic investigator to either examine the data in RAM before a device reboot happens or extract the data.

You can use the `mount` command to see different partitions and their filesystems available on the device, as follows:

```
shell@Android:/sdcard $ mount
mount
rootfs / rootfs rw 0 0
tmpfs /dev tmpfs rw,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,relatime,mode=600,ptmxmode=000 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
tmpfs /mnt/asec tmpfs rw,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,relatime,mode=755,gid=1000 0 0
/dev/block/nandd /system ext4 rw,nodev,noatime,user_
xattr,barrier=0,data=ordered 0 0
/dev/block/nande /data ext4 rw,nosuid,nodev,noatime,user_
xattr,barrier=0,journal_checksum,data=or dered,noauto_da_alloc 0 0
/dev/block/nandh /cache ext4 rw,nosuid,nodev,noatime,user_
xattr,barrier=0,journal_checksum,data=or dered,noauto_da_alloc 0 0
/dev/block/vold/93:64 /mnt/sdcard vfat rw,dirsync,nosuid,nodev,noexec,rel
atime,uid=1000,gid=1015,fmask=0702, dmask=0702,allow_utime=0020,codepage=
cp437,iocharset=ascii,shortname= mixed,utf8,errors=remount-ro 0 0
```

```
/dev/block/vold/93:64 /mnt/secure/asec vfat rw,dirsync,nosuid,nodev,noexe
c,relatime,uid=1000,gid=1015,fmask=0702, dmask=0702,allow_utime=0020,code
page=cp437,iocharset=ascii,shortname= mixed,utf8,errors=remount-ro 0 0
tmpfs /mnt/sdcard/.Android_secure tmpfs ro,relatime,size=0k,mode=000 0 0
/dev/block/dm-0 /mnt/asec/com.kiloo.subwaysurf-1 vfat ro,dirsy
nc,nosuid,nodev,relatime,uid=1000,fmask=0222,dmask=0222,codep
age=cp437,iocharset=ascii, shortname=mixed,utf8,errors=remount-ro 0 0
```

As seen in the preceding command-line output, different partitions have different filesystems, and they are mounted accordingly.

Summary

Having sound knowledge of Android's partition layout, filesystems, and important locations will help the forensic investigator during the process of extracting data from the device. The user data location on the Android device contains a bulk of user information that can be crucial for any forensic investigation. However, most of these files may be accessed only on a rooted phone (especially, files present in the `/data/data` location). You have also learned about Android data-storage options, various filesystems used by Android, and their significance.

With this knowledge, you will now learn how to logically and physically extract the data from an Android device in the upcoming chapters.

4
Extracting Data Logically from Android Devices

This chapter will be covering logical data extraction by using free and open source tools wherever possible. The majority of the material covered in this chapter will use the ADB methods previously discussed in *Chapter 2*, *Setting Up an Android Forensic Environment*.

By the end of this chapter, the reader should be familiar with the following:

- What logical extraction means
- What data to expect from logical extractions
- What data is available with and without root
- Manual ADB data extractions
- ADB Backup extractions
- ADB dumpsys information
- How to bypass Android lock screens
- SIM card extractions

Logical extraction overview

In digital forensics, the term logical extraction is typically used to refer to extractions that do not recover deleted data, or do not include a full bit-by-bit copy of the evidence. However, a more correct definition of logical extraction, also defined in *Chapter 1*, *Introducing Android Forensics*, is any method that requires communication with the base operating system. Because of this interaction with the operating system, a forensic examiner cannot be sure that they have recovered all of the data possible; the operating system is choosing which data it allows the examiner to access.

In traditional computer forensics, logical extraction is analogous to copying and pasting a folder in order to extract data from a system; this process will only copy files that the user can access and see. If any hidden or deleted files are present in the folder being copied, they will not be in the pasted version of the folder.

As you will see, however, the line between logical and physical extractions in mobile forensics is somewhat blurrier than in traditional computer forensics. For example, deleted data can routinely be recovered from logical extractions on mobile devices, due to the prevalence of SQLite databases being used to store data. Furthermore, almost every mobile extraction will require some form of interaction with the Android operating system; there is no simple equivalent to pulling a hard drive and imaging it without booting the drive. For our purposes, we will define a logical extraction as the process that obtains data visible to the user, and may include data that has been marked for deletion.

What data can be recovered logically?

For the most part, any and all user data may be recovered logically:

- Contacts
- Call logs
- SMS/MMS
- Application data
- System logs and information

The bulk of this data is stored in SQLite databases, so it is even possible to recover large amounts of deleted data through a logical extraction.

Root access

When forensically analyzing an Android device, the limiting factor is often not the type of data being sought, but rather whether or not the examiner has the ability to access the data. Root access has been covered extensively in *Chapter 2*, *Setting Up an Android Forensic Environment*, but it is important enough to warrant repetition. All of the data listed above, when stored on the internal flash memory, is protected and requires root access to read. The exception to this is application data that is stored on the SD card, which will be discussed later in this book.

Without root access, a forensic examiner cannot simply copy information from the data partition. The examiner will have to find some method of escalating their privileges in order to gain access to the contacts, call logs, SMS/MMS, and application data. These methods often carry many risks, such as the potential to destroy or "brick" the device (making it unable to boot), and may alter data on the device in order to gain permanence. The methods commonly vary from device to device, and there is no universal, one-click method to gain root access to every device. Commercial mobile forensic tools such as **MicroSystemation XRY** and **Cellebrite UFED** have built-in capabilities to temporarily and safely root many devices, but do not cover the wide range of all Android devices.

Throughout this chapter, we will make note of situations where root is required for each technique demonstrated.



The decision to root a device should be made in accordance with your local operating procedures and court opinions in your jurisdiction. The legal acceptance of evidence obtained by rooting varies by jurisdiction.

Manual ADB data extraction

The ADB `pull` command can be used to pull single files or entire directories directly from the device on to the forensic examiner's computer. This method is especially useful for small, targeted examinations. For example, in an investigation strictly involving SMS messages, the examiner can choose to pull just the relevant files.

USB debugging

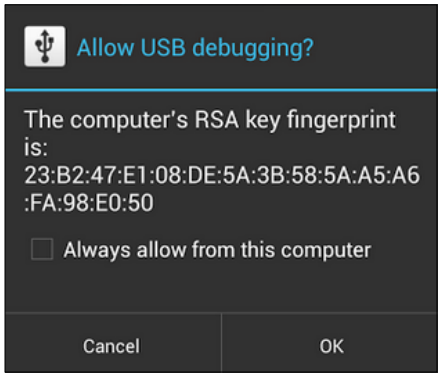
Setting up the ADB environment has been previously discussed in this book. However, the device under examination must also be configured properly. USB debugging is the actual method through which the examiner's computer will communicate with the device. The **USB debugging** option is found under the **Developer options** in the **Settings** menu. However, as of Android 4.2, the Developer Options menu is hidden; to reveal it, a user has to go to **Settings** | **About Phone**, and then tap the **Build Number** field *seven* times. An on-screen dialog will appear that says **You are now a developer!** At this point, **Developer options** is available in the **Settings** menu; simply open this menu and select **Enable USB debugging**.

In addition to USB debugging, the correct drivers must be installed on the examiner's computer. Generally they can be found online, either from the manufacturer's website or at `www.xda-developers.com`. If commercial forensic tools are installed on the machine, the appropriate drivers may already be installed.



Another excellent resource is the Universal ADB Driver that can be downloaded for free at `http://drivers.softpedia.com/get/MOBILES/Clockworkmod/Clockworkmod-Universal-Android-ADB-Driver.shtml`.

Prior to Android 4.2.2, enabling **USB debugging** was the only requirement to communicate with the device over ADB. In Android 4.2.2, Google added **Secure USB debugging** option. The **Secure USB debugging** option adds an additional requirement of selecting to connect to a computer on the device's screen; this prevents ADB access to locked devices from untrusted computers:



RSA fingerprint dialog

If **Always allow from this computer** is selected, the device will store the computer's RSA key and the prompt will not appear on future connections to that computer, even if the device is locked.



It may be possible, depending on the device and OS version, to circumvent the **Secure USB debugging** protection. Find more information at `https://labs.mwrinfosecurity.com/advisories/2014/07/03/android-4-4-2-secure-usb-debugging-bypass/`.

It is also possible to bypass **Secure USB debugging** by using a computer previously authorized to access the device, which is discussed in the Issues with Android Lollipop section later in this chapter.

Once **USB debugging** has been enabled and the **Secure USB debugging** check passed (depending on Android version), the device is ready for examination. To verify that the device is connected and ready to use ADB, execute the following command:

```
adb devices
```

If no devices are shown, ensure that **USB debugging** is enabled and that the proper device drivers have been installed:

```
C:\Users\Android_Examiner>adb devices
List of devices attached
```

If the device status is `offline` or `unauthorized`, the **Secure USB debugging** prompt needs to be selected on the screen:

```
C:\Users\Android_Examiner>adb devices
adb server is out of date.  killing...
* daemon started successfully *
List of devices attached
T076001020      offline
```

If everything is running correctly, the device status should show `device`:

```
C:\Users\Android_Examiner>adb devices
adb server is out of date.  killing...
* daemon started successfully *
List of devices attached
T076001020      device
```

Using ADB shell to determine if a device is rooted

The simplest method to determine if a device is rooted is to use the ADB shell. This will open a shell on the device that will be accessed on the examiner's computer; this means that any commands run in the shell will be executed on the device. Once **USB debugging** is enabled and **Secure USB debugging** is bypassed (or from recovery mode, as discussed in later in this chapter), open a terminal on the local computer and run:

```
adb shell
```

The shell will appear one of two ways, either with `$` or `#`:

```
C:\Users\Android_Examiner>adb shell
shell@ghost:/ $
```

On Linux systems, the `#` symbol is used to indicate a root user, and the `$` symbol indicates a non-root user. If the shell returns showing `#`, the shell has root access:

```
C:\Users\Android_Examiner>adb shell
~ #
```

One further step may be required on some rooted devices. If the shell returns `$`, try running the `su` command:

```
su
```

If the su binary is installed on the device, which is usually a part of the root process, this will escalate the shell's permissions to root if it did not open with them.



Some older devices automatically ran the shell as root; simply opening the ADB shell may be enough to give an examiner root access.

ADB pull

As discussed in *Chapter 2*, *Setting Up an Android Forensic Environment*, the ADB `pull` command is used to transfer files from the device to the local workstation. The format for the ADB `pull` command is:

```
adb pull [-p] [-a] <remote> [<local>]
```

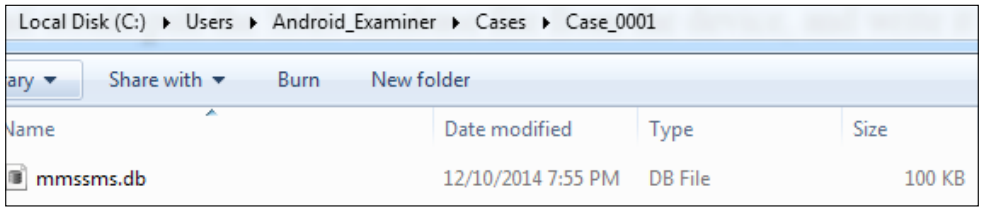
The optional `-p` flag shows the transfer's progress, while the optional `-a` flag will copy the file's timestamp and mode. The `<remote>` parameter is the exact path to the file on the device. The optional `<local>` parameter is the path where the file will be written on the examiner's computer. If no local path is specified, the file will be written to the present working directory. To see what an ADB `pull` command may look like, run the following command:

```
adb pull -p /data/data/com.android.providers.telephony/databases/mmssms.
db C:/Users/Cases/Case_0001
```

This command would pull the SMS database file from the device, and write it to a directory for the case. Again, note that the device must be rooted for this to work; otherwise, the output would simply show that `0` files were pulled. In our case, the following output is obtained:

```
C:\Users\Android_Examiner>adb pull -p /data/data/com.android.providers.telephony
/databases/mmssms.db Cases\Case_0001\mmssms.db
Transferring: 102400/102400 (100%)
1666 KB/s (102400 bytes in 0.060s)
```

The preceding output shows that the file is `1020400` bytes in size. As a result of our command, the `mmssms.db` database now resides in the `Case_0001` folder:



The database pulled from the device, seen in Windows Explorer

The database can now be examined with a SQL Browser or other forensic tools, which will be covered in *Chapter 7*, *Forensic Analysis of Android Applications*.

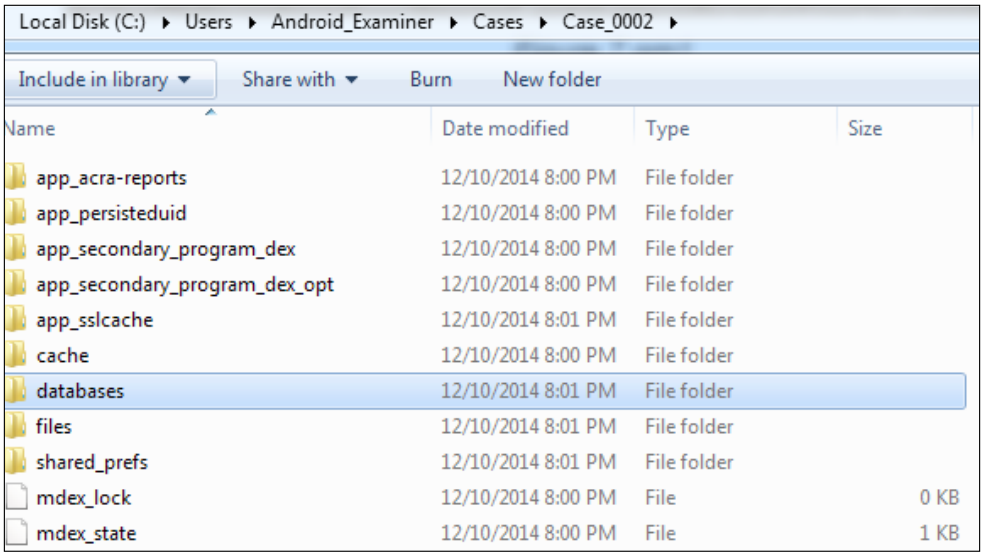
Similarly, if an investigator wishes to pull the files for an entire application, that can be done with ADB pull also:

```
C:\Users\Android_Examiner>adb pull -p /data/data/com.google.android.gm Cases\Cas
e_0002
pull: building file list...
skipping special file 'lib'
pull: /data/data/com.google.android.gm/shared_prefs/Folder-donnietindall@gmail.c
om-^iim.xml -> Cases\Case_0002/shared_prefs/Folder-donnietindall@gmail.com-^iim.
xml
pull: /data/data/com.google.android.gm/shared_prefs/Account-donnietindall@gmail.
com.xml -> Cases\Case_0002/shared_prefs/Account-donnietindall@gmail.com.xml
```

This time, the ADB pull command fetched every file in the `com.google.android.gm` directory, which happens to contain all of the data for Gmail. The output was quite long, as it individually listed all `31` pulled files, so the entire output is not shown in the following figure, in which we see the total size of the transfer is shown as `1233373` bytes:

```
pull: /data/data/com.google.android.gm/app_sslcache/www.google.com.443 -> Cases\
Case_0002/app_sslcache/www.google.com.443
pull: /data/data/com.google.android.gm/app_sslcache/android.clients.google.com.4
43 -> Cases\Case_0002/app_sslcache/android.clients.google.com.443
31 files pulled. 0 files skipped.
1475 KB/s (1233373 bytes in 0.816s)
```

Now, the `Case_0002` directory contains all of the files from the Gmail application, as shown in the following screenshot:



All files pulled from the Gmail directory, seen in Windows Explorer

It is even possible to do the following:

```
adb pull -p /data/data/ \Cases\Case_0003
```

This would pull every logical file available from the `/data/data` directory, and put them in the examiner's `Case_0003` folder. This is not equivalent to a physical image, as certain files are skipped and deleted files will not be copied, but it is a simple method for pulling the vast majority of a user's application data.

Another advantage of the ADB `pull` command is that it is highly useful for scripting purposes. A knowledgeable examiner can maintain a list of paths for common files of interest, and write a script that automatically pulls these files from a device, or even have the script automatically pull the entire `/data/data` directory. A simple example of Python code that will perform this function is:

```
from subprocess import Popen
from os import getcwd

command = "adb pull /data/data " + getcwd() + "\data_from_device"
p = Popen(command)
p.communicate()
```

Note that code is not very refined; it's only purpose is to illustrate the ease with which ADB commands can be scripted. At the very least, properly implementing the code should include the option to specify an output directory and handle any errors. However, the six lines of the preceding code would be sufficient to pull the entire `/data/data` directory logically assuming **USB debugging** is enabled and the device is rooted.

Recovery mode

In order to truly be forensically sound, ADB data extractions should not be used against a phone while it is turned on. While the device is running, timestamps can be modified and applications may be running and updating files in the background. To avoid this, an examiner should place the device into a custom recovery mode, as shown in *Chapter 2*, *Setting Up an Android Forensic Environment*, if possible. ADB access is not available through the stock Android recovery mode. Typically, the first step in the rooting process is to flash a custom recovery mode to allow a method for repairing the device if something goes wrong. Rooted devices are far more likely to contain a custom recovery, but it is possible to flash a custom recovery to a non-rooted device. This method also allows the examiner to avoid the **Secure USB debugging** prompt on newer versions of Android, although our testing shows that this does not work on Android Lollipop. Recovery mode also may not require **USB debugging** to be enabled, which makes it an excellent option for bypassing a locked device.



This method will not work against devices with full disk encryption enabled. Booting into Recovery Mode will *not* decrypt the data partition.

The process to boot into recovery mode will vary for each device. Typically, it involves some combination of powering the device off and holding the volume and power keys. Guides for specific models can be found online.

The stock recovery mode will typically show a picture of an Android being operated on:



Stock Recovery mode

Custom recoveries look like the following screenshots. Also, stock recoveries will not allow ADB communication; running adb devices will simply show no devices.



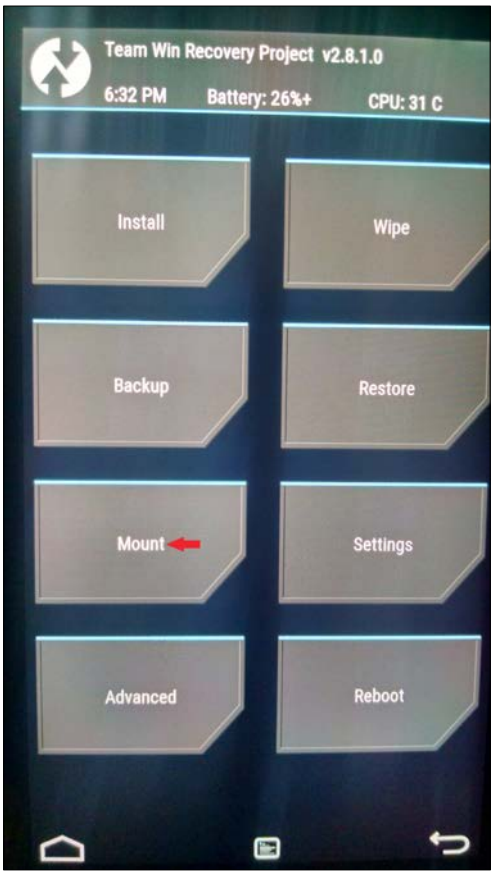
Custom recovery images for many devices can be found at `https://www.clockworkmod.com/rommanager` and `http://teamw.in/project/twrp2`.

If a device is in a custom Recovery Mode and the correct drivers have been installed on the examiner's computer, the device can be accessed via ADB as if it were live. Note that its status using the adb devices command now shows that it is in recovery mode:

```
C:\Users\Android_Examiner>adb devices
adb server is out of date.  killing...
* daemon started successfully *
List of devices attached
FA31RS505163    recovery
```

There is one final step before the examiner can begin extracting data over ADB: the data partition must be mounted in order to access user data. Some custom recoveries may mount this automatically, and others might not. If using either the Clockwork Mod Recovery or **Team Win Recovery Project** (**TWRP**) images from the URLs above, the data partition can be mounted by selecting **Mounts** and then selecting the data partition, as shown in the following screenshots. The recovery menu is generally either navigated by using the volume keys to move up and down and the power button to select, or may be touch-based depending on the custom recovery image used.

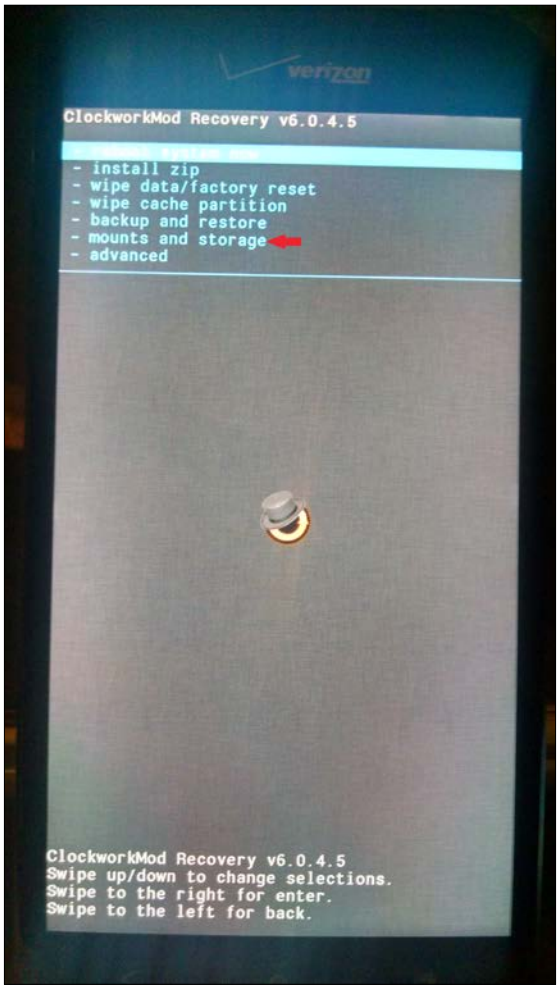
For a TWRP recovery, from the main recovery screen, select **Mount**:



After choosing **Mount**, select the partition(s) to be mounted:



In a Clockwork Mod Recovery, select **mounts and storage**:



Then select the partition(s) to mount:



Once the data partition (and any other partition the examiner wants to investigate) is mounted, the examiner can perform ADB data extractions, as demonstrated earlier in this chapter.

If the device does not have a custom recovery, the following section will show how to boot into one.

Fastboot mode

Fastboot is another protocol utility built into the Android SDK, and is used for interacting directly with a device's bootloader. Essentially, it is a much lower-level version of ADB, and is frequently used to flash new images to a device. How can this be helpful to an examiner?

Fastboot can allow an examiner to boot from a custom recovery image, and temporarily gain root access on a device, thus gaining access to data that would have been unavailable otherwise. Fastboot does not require USB debugging to be enabled or root access. The process of loading a custom bootloader onto a device is commonly used by commercial forensic tools to temporarily root a device, but a skilled examiner can also perform the process manually. Using this method, the recovery image is loaded into RAM; no permanent data on the device is altered in any way.

The most important requirement for using fastboot is an unlocked bootloader; locked bootloaders will not allow a device to boot from code that isn't specifically signed by the manufacturer. Unfortunately for forensic purposes, most devices no longer ship with an unlocked bootloader as it is a serious security risk, and manually unlocking a bootloader typically erases the user data. As such, the number of devices for which this is a feasible method is somewhat limited. But, when it works, it's an absolutely invaluable tool for an examiner to have in their arsenal.



This method will not work against devices with full disk encryption enabled. Booting into recovery mode will *not* decrypt the data partition.

Determining bootloader status

Much like everything involving Android forensics, there is no one guaranteed method to determine if a bootloader is locked, as it varies depending upon the manufacturer. To boot into the bootloader, use the following ADB command:

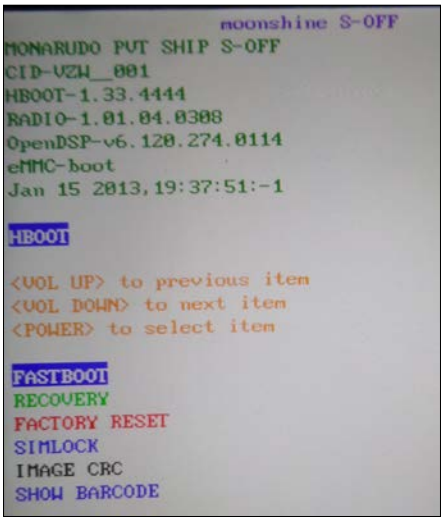
```
adb reboot bootloader
```

The device should boot to a screen that shows information regarding the bootloader. Frequently, this screen will display the bootloader status, as seen in the following screenshot.

Here's a generic, stock fastboot menu from a Nexus 5. Note that the **Lock State** indicates that the bootloader is unlocked:



A standard HTC fastboot screen is as follows:



Following is a standard Samsung **Odin** mode screen; Odin is the Samsung proprietary equivalent to fastboot:



Booting to a custom recovery image

Once the bootloader is determined to be unlocked, an examiner will need a custom recovery image from which to boot. An excellent source of recovery images is either `https://www.clockworkmod.com/rommanager` or `http://teamw.in/twrp_view_all_devices`. Both sites offer coverage of a wide variety of devices, and will provide the same functionality for the purposes of this method.



It is absolutely critical to select the correct recovery image for the device being examined; they are not interchangeable, and booting from the wrong image may brick the device.

Once a recovery image is selected and downloaded, the device needs to be placed into fastboot mode. This can be accomplished in one of two ways:

- ADB
- Physical device buttons

To enter fastboot device over ADB, the device must already have **USB debugging** enabled. The command to enter fastboot mode over ADB is:

```
adb reboot bootloader
```

If USB Debugging cannot be enabled or ADB cannot be used, there is also typically a combination of buttons to press while the device is booting, similar to entering Recovery Mode. The exact combination can be found online for each device specifically.

Once the device is in fastboot mode, running the following command will verify if the device is connected and ready to communicate:

```
fastboot devices
```

The following command will load the custom recovery image into RAM and boot the device into Recovery Mode:

```
fastboot boot 'path to image'
```

```
C:\Users\Android_Examiner>fastboot boot recovery-clockwork-6.0.4.5-dlx.img
downloading 'boot.img'...
OKAY [  0.982s]
booting...
OKAY [  0.004s]
finished. total time: 0.992s
```

The device should now reboot and enter Recovery Mode. As shown in the Recovery Mode section, the /data partition may need to be mounted in order to access user data.

Entering the ADB shell will show that the examiner now has root access. The device will allow root access until it is rebooted.

```
C:\Users\Android_Examiner>adb shell
adb server is out of date.  killing...
* daemon started successfully *
~ #
```

If the fastboot boot command fails, it is a likely indicator that the device's bootloader is locked, as shown in the following screenshot:

```
C:\Users\Android_Examiner>fastboot boot recovery-clockwork-6.0.4.7-falcon.img
downloading 'boot.img'...
OKAY [  0.771s]
booting...
(bootloader) Command restricted
FAILED (remote failure)
finished. total time: 0.876s
```

ADB backup extractions

Google implemented ADB backup functionality, beginning in Android 4.0 Ice Cream Sandwich. This allows users (and forensic examiners) to backup application data to a local computer over ADB. This process does not require root, and is therefore highly useful for forensic purposes. However, it does not acquire every application installed on the device. When a developer makes a new app, it is set to allow backups by default, but this can be changed by the developer. In practice, it seems the vast majority of developers leave the default setting, which means that backups do capture most third-party applications. Unfortunately, most Google applications disable backups; full application data from apps such as Gmail and Google Maps will not be included.



This method will not be useful against a locked device as user interaction with the screen is required.

Extracting a backup over ADB

The format of the ADB backup command is:

```
adb backup [-f <file>] [-apk|-noapk] [-obb|-noobb] [-shared|-noshared]
[-all] [-system|-nosystem] [<packages...>]
```

The flags are as follows:

- `-f`: Names the path for the output file. If not specified, defaults to `backup.ab` in present working directory.
- `[-apk|noapk]`: Choose whether or not to back up the `.apk` file. Defaults to `-noapk`.
- `[-obb|-noobb]`: Choose whether or not to back up `.obb` (APK expansion) files. Defaults to `-noobb`.
- `[-shared|-noshared]`: Choose whether or not to back up data from shared storage and the SD card. Defaults to `-noshared`.

- `[-all]`: Include all applications for which backups are enabled.
- `[-system|-nosystem]`: Choose whether or not to include system applications. Defaults to `-system`.
- `[<packages>]`: Explicitly name application packages to be backed up. Not needed if using `-all` or `-shared`.

An example ADB backup command to capture all possible application data would be:

```
adb backup -f C:/Users/Cases/Case_0001/backup.ab -shared -all
```

Alternatively, an example ADB backup command to capture a specific application's data would be:

```
adb backup -f C:/Users/Cases/Case_0001/facebook.ab com.facebook.katana
```

You should see something like:

```
C:\Users\Android_Examiner>adb backup -shared -all
Now unlock your device and confirm the backup operation.
```

When performing a backup, the user must approve the backup on the device. This means that backups cannot be performed without bypassing screen locks:



Accepting the backup on the device

Depending on the number of applications installed, the backup process may take a significant amount of time.

Parsing ADB backups

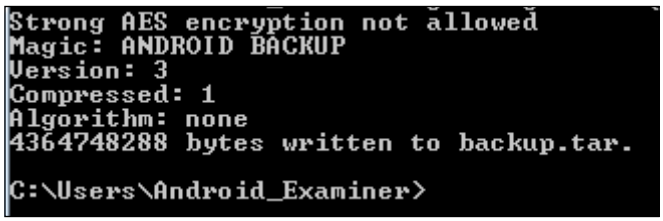
The resulting backup data is stored as a `.ab` file, but is actually a `.tar` file that has been compressed with the **Deflate** algorithm. If a password was entered on the device when the backup was created, the file would also be AES encrypted. It should also be mentioned that these files may exist on a suspect's computer, and can be analyzed using the same methods.

There are many free utilities to turn the `.ab` backup file into a `.tar` that can be viewed. One such utility is the Android Backup Extractor, found at `http://sourceforge.net/projects/adbextractor/`.

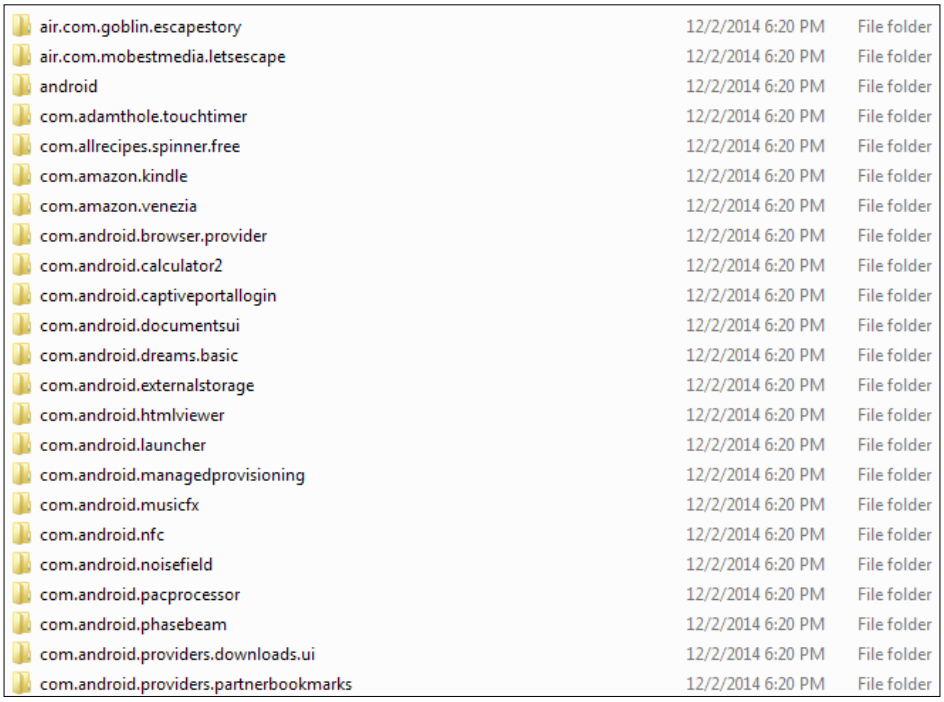
To use the Android Backup Extractor, simply extract its files into the directory with the backup. The command to run the utility is:

```
java -jar abe.jar unpack backup.ab backup.tar
```

If the command runs properly, the command line will display as follows:



The first line of the output informs the examiner that the file was not encrypted. Had it been encrypted, the examiner would have to pass the password as an argument at the end of the command line. As seen in the output, the backup created in the previous section is approximately 4 GB, even though it is still compressed. The `.tar` file will be at the path specified on the command line or the current working directory if no path is specified. Decompressing the `.tar` file may be done manually on a Linux command line or with one of the many Windows archive utilities, such as WinRAR or 7Zip:



Directories within the backup, seen in Windows Explorer

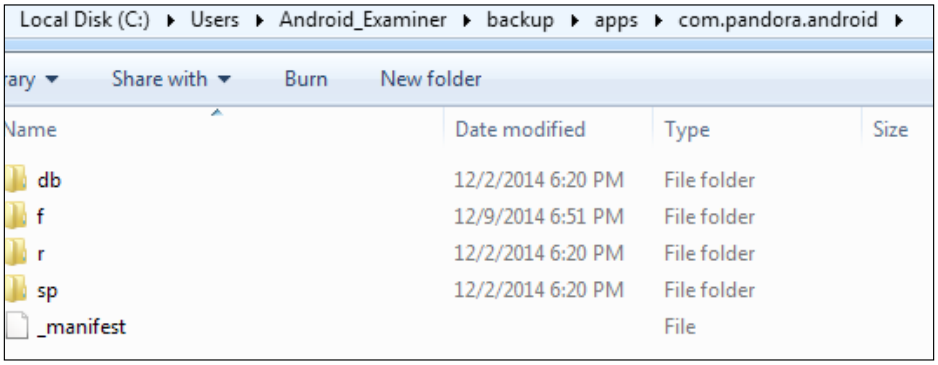
Data locations within ADB backups

Now that the backup has been converted to a `.tar` file and then extracted, the examiner can view the data contained in the backup. In our example, there are two directories found in the root of the backup:

- `apps`: This folder contains data from `/data/data` for applications that were included in the backup.
- `shared`: This folder contains all data from SD card, only present if the –shared argument was passed at the command line.

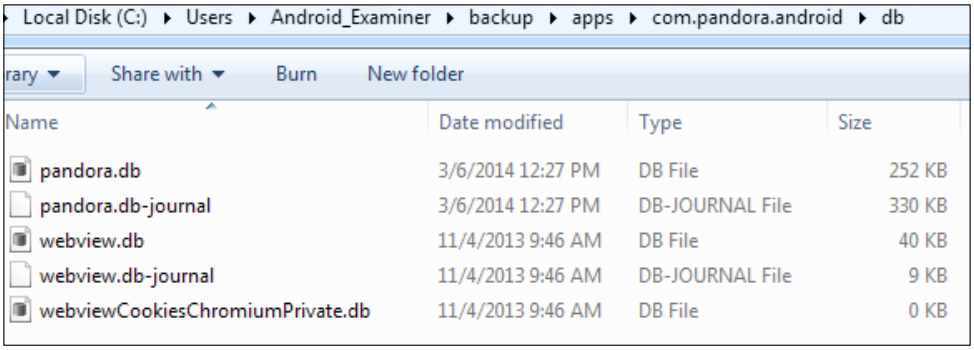
Note that the files within the apps directory are stored in directories by their package name (just as seen in `/data/data` from within the ADB shell), and the shared directory is exactly what the user would see if they accessed the SD card by plugging it into a computer.

For a benign example of user data that was pulled from the backup, the user's **Pandora** activity is shown below. Pandora is a streaming music service with millions of downloads in the Google Play Store. Pandora's application data will be contained in the apps folder of the backup in the folder named `com.pandora.android`.



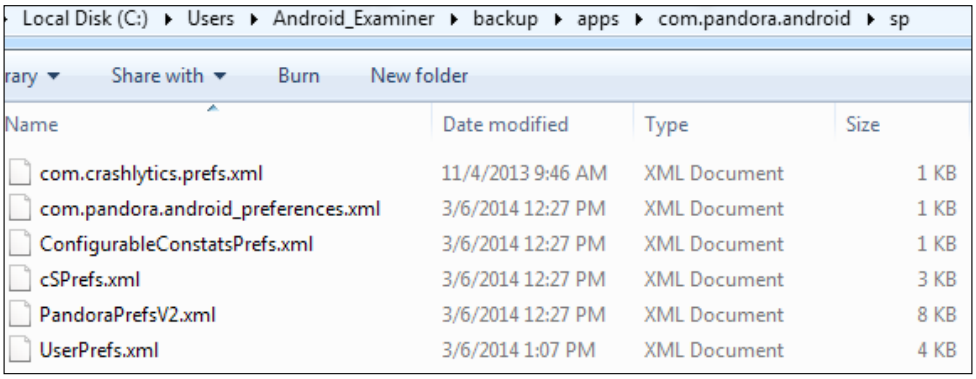
The Pandora directory from the backup

This is a fairly standard layout for an Android application, as discussed in *Chapter 2, Setting Up an Android Forensic Environment*. The application's databases will be in the `db` folder:



Files within the db folder of the Pandora backup

XML configuration settings will be in the `sp` folder:



Files within the sp folder of the Pandora backup

Using a database viewer to view `pandora.db` reveals stations that the user has created, as well as the timestamp for when it was created:

_id	stationToken	stationName	...ick	...har	...Add	...Rer	...De	...sCle	...sVi	ressi	dateCreated
Filter	Filter	Filter	Fi...	Fi...	Fi...	Fi...	Fi...	Fi...	Fi...	Fil...	Filter
1	1729551345663006807	Christmas Radio	0	0	0	1	1	1	1	1	1385999292130
2	270490452007508055	Cold As Ice Radio	0	0	1	1	1	0	0	1	1280610444947
3	211607193408690263	Jason Mraz Radio	0	0	1	1	1	0	0	1	1269795747721
4	210973539703642199	Shuffle	1	0	0	0	0	0	0	1	1269651653575
5	210973496753969239	Jack's Mannequin Radio	0	0	1	1	1	0	0	1	1269651649846

Contents of pandora.db from the backup

Looking in the XML preferences file, the timestamp of the app installation can be found under `firstInstallId`. Note that the exact method for converting the timestamps is shown in *Chapter 7, Forensic Analysis of Android Applications*:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<map>
  <string name="totalForegroundTime">0</string>
  <string name="lastUserInteractionTimestamp">-1</string>
  <string name="lastTransmission">1394126848807</string>
  <string name="lastUserSessionTimestamp">-1</string>
  <string name="firstInstallId">1394126848795</string>
```

Contents of the XML preferences file

If, for some odd reason, the user's Pandora usage was a major question in the investigation, what could an examiner determine from these two seemingly innocuous files?

Firstly, the `lastTransmission` and `firstInstallID` timestamps are within milliseconds of each other, indicating that the application was never used after it was installed. Furthermore, the creation dates of each station precede the installation of the application, in some cases, by years. This would be an indicator that the user has used Pandora on other devices, which may be highly relevant to the investigation.

While Pandora is generally not germane to digital forensic investigations, it is an example of data that can be gleaned from a simple backup over ADB. A more detailed application analysis will be presented in *Chapter 7*, *Forensic Analysis of Android Applications*.

ADB Dumpsys

Dumpsys is a tool built into the Android OS, generally used for development purposes to show the status of services running on the device. However, it can also contain forensically interesting information. Dumpsys does not require root access, but like all ADB commands, it does require USB Debugging to be enabled on the device and Secure USB Debugging to be bypassed.

The exact services that can be viewed differ across devices and Android versions. To view a list of all possible services that can be dumped, run the following command:

```
adb shell service list
```

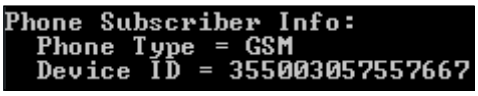
The output of the command will appear as a list, shown as follows:

```
Found 86 services:
0       ismsapp: []
1       AtCmdFwd: [com.qualcomm.atfwd.IAtCmdFwd]
2       partial_display_service: [android.app.IPartialDisplayService]
3       sip: [android.net.sip.ISipService]
4       phone: [com.android.internal.telephony.ITelephony]
5       iphonenvinfo: [com.motorola.android.telephony.IPhoneNVInfo]
6       isms: [com.android.internal.telephony.ISms]
7       iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
8       simphonebook: [com.android.internal.telephony.IIccPhoneBook]
9       nfc: [android.nfc.INfcAdapter]
10      modality: [com.motorola.slpc.IModalityService]
11      media_router: [android.media.IMediaRouterService]
12      print: [android.print.IPrintManager]
```

The service name located before the colon is the argument we will pass to dumpsys. A valid dumpsys command, using service number seven (`iphonesubinfo`) in the preceding screenshot, looks like this:

```
adb shell dumpsys iphonesubinfo
```

In the following screenshot, we see that the output of the `iphonesubinfo` service includes the device IMEI:

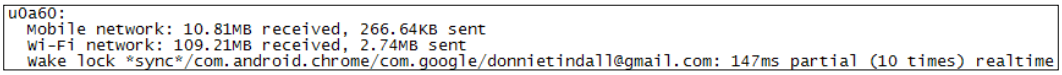


There are many forensically interesting dumpsys services; following are several examples. As the dumpsys services may vary by OS version and device, this list is not all-inclusive and is merely intended to show the usefulness of dumpsys to a forensic examiner:

- iphonesubinfo
- batterystats
- procstats
- user
- appops
- Wi-Fi
- notification

Dumpsys batterystats

Batterystats is used to show the usage of running applications. Its output can be very verbose, depending on the number of applications in use. In the following screenshot , the output was redirected to a file because it did not fit in the Windows command line:



This shows us the network usage of Google Chrome. This information can be used to show that the application had been used recently, and this information will exist even if Chrome was used in Incognito Mode and leaves no forensic evidence elsewhere.



The **wake lock** section can be very useful for detecting malware. A wake lock is a method of keeping the device awake (not entering sleep mode), and is indicative of an application attempting to stay running in the background.

Dumpsys procstats

The **procstats** is a service to display the processor usage by running applications. Similar to batterystats, it is another method that can be used to show that an application was recently used on a device, as shown in the following screen shot:

```
* com.android.chrome / u0a60:
          TOTAL: 7.8% (52MB-84MB-123MB/48MB-73MB-108MB over 44)
            Top: 7.7% (52MB-84MB-123MB/48MB-73MB-108MB over 44)
         Imp Fg: 0.01%
         Imp Bg: 0.00%
        Service: 0.07%
       Receiver: 0.01%
     (Last Act): 8.2% (53MB-62MB-70MB/49MB-57MB-66MB over 29)
       (Cached): 83% (5.2MB-56MB-69MB/4.2MB-52MB-64MB over 65)
```

Dumpsys user

Beginning with Android Jelly Bean, Google added support for multiple users on tablet devices. With the release of Lollipop, Google extended this support to phones. One of the most challenging problems in digital forensics for a long time has been to prove who was using a device when incriminating actions were performed; "Who was behind the keyboard?"

Running dumpsys on the user service will show last login info for all users:

```
Users:
  UserInfo{0:Amber:13} serialNo=0
    Created: <unknown>
    Last logged in: +1h54m10s900ms ago
  UserInfo{10:Donnie:10} serialNo=10
    Created: +4m9s288ms ago
    Last logged in: +4m1s837ms ago
```

As only one user can be logged in at a time, looking at the user with the most recent login will identify the account currently in use on the device.

Dumpsys App Ops

The **App Ops** may be the most interesting dumpsys service. The term App Ops is generally used to refer to permissions accessible by an application. In older versions of Android, it was rumoured that Google would include the ability for users to revoke specific permissions from an application. This has never come to fruition, but this service at least remains, and shows the last time an application used each permission that it can access. Following is another example from Google Chrome:

```
Uid u0a60:
  Package com.android.chrome:
    COARSE_LOCATION: mode=0; duration=0
    FINE_LOCATION: mode=0; time=+8h57m51s355ms ago; duration=0
    VIBRATE: mode=0; time=+1d7h2m45s243ms ago; duration=+12ms
    POST_NOTIFICATION: mode=0; time=+6d7h2m42s380ms ago; duration=0
    READ_CLIPBOARD: mode=0; time=+5d8h12m52s649ms ago; duration=0
    WRITE_CLIPBOARD: mode=0; time=+10d20h49m23s22ms ago; duration=0
    TAKE_MEDIA_BUTTONS: mode=0; time=+176d17h18m19s460ms ago; duration=0
    TAKE_AUDIO_FOCUS: mode=0; time=+1h7m12s279ms ago; duration=0
    AUDIO_RING_VOLUME: mode=0; time=+23h52m52s671ms ago; duration=0
    AUDIO_MEDIA_VOLUME: mode=0; time=+1h31m46s692ms ago; duration=0
    WAKE_LOCK: mode=0; time=+17m43s597ms ago; duration=+55ms
    MONITOR_LOCATION: mode=0; time=+110d8h9m26s749ms ago; duration=+1s219ms
```

In the above output, we can see that approximately 1 hour and 7 minutes before App Ops was dumped with dumpsys, Chrome used the `TAKE_AUDIO_FOCUS` permission, and later used `AUDIO_MEDIA_VOLUME`. This indicates what Chrome was used to listen to and when.

A somewhat more interesting example is the following phone application:

```
Uid 1001:
  Package com.android.phone:
    VIBRATE: mode=0; time=+2h34m31s210ms ago; duration=+1s20ms
    READ_CONTACTS: mode=0; time=+44m2s299ms ago; duration=0
    WRITE_CONTACTS: mode=0; time=+44m2s201ms ago; duration=0
    READ_CALL_LOG: mode=0; time=+4d7h29m35s902ms ago; duration=0
    WRITE_CALL_LOG: mode=0; time=+44m2s6ms ago; duration=0
    POST_NOTIFICATION: mode=0; time=+1d1h31m34s242ms ago; duration=0
    CALL_PHONE: mode=0; time=+1d0h56m59s194ms ago; duration=0
    READ_SMS: mode=0; time=+4d7h29m36s362ms ago; duration=0
    WRITE_SMS: mode=0; time=+3h5m48s341ms ago; duration=0
    WRITE_SETTINGS: mode=0; time=+17m18s147ms ago; duration=0
    SYSTEM_ALERT_WINDOW: mode=0; time=+20h41m26s834ms ago; duration=+4s776ms
    TAKE_AUDIO_FOCUS: mode=0; time=+53m41s785ms ago; duration=0
    WAKE_LOCK: mode=0; time=+1m23s617ms ago; duration=+15ms
```

`44` minutes ago, the user used the phone application and required the `READ_CONTACTS` permission, then immediately used the `WRITE_CALL_LOG` permission. We can surmise that the user made a phone call `44` minutes ago; even if they had deleted the call from the records afterwards.

Dumpsys Wi-Fi

The **Wi-Fi** service will show a list of all SSIDs for which a connection has been saved. This could be useful for showing that a user was at a certain location, for example. More detailed Wi-Fi information is also available on the file system, but requires root to view. Using dumpsys, we can access this data without requiring root:

```
ID: 9 SSID: "FOR585" BSSID: null PRIO: 51
 KeyMgmt: WPA_PSK Protocols: WPA RSN
 AuthAlgorithms:
 PairwiseCiphers: TKIP CCMP
 GroupCiphers: WEP40 WEP104 TKIP CCMP
 PSK: *anonymous_identity NULL
```

Dumpsys notification

The **notification** service will provide information about currently active notifications. This can be useful for recording the state of a device when it is seized, or identifying which application is displaying a specific notification. Each notification can be rather large and contain a lot of information, only some of which may be of use. The following example shows an incoming email from the Gmail application, which includes the subject (`This is a test email`) and body (`To see a test notification`):

```
      }
    NotificationRecord(0x4226a928: pkg=com.google.android.gm user=UserHandle(0)
id=31465589 tag=null score=0: Notification(pri=0 contentView=com.google.android.
gm/0x1090064 vibrate=default sound=content://settings/system/notification_sound
defaults=0x6 flags=0x11 kind=[null] 2 actions))
      uid=10068 userId=0
      icon=0x7f0200df / com.google.android.gm:drawable/ic_notification_mail_24dp

      pri=0 score=0
      contentIntent=PendingIntent(42aae7f8: PendingIntentRecord(42ca7258 com.goo
gle.android.gm startActivity))
      deleteIntent=PendingIntent(42ab3e38: PendingIntentRecord(42d97190 com.goog
le.android.gm startService))
      tickerText=Donnie Tindall
      contentView=android.widget.RemoteViews@42a18b58
      defaults=0x00000006 flags=0x00000011
      sound=content://settings/system/notification_sound
      vibrate=null
      led=0x00000000 onMs=0 offMs=0
      actions={
        [0] "Delete" -> PendingIntent(42913958: PendingIntentRecord(42a2f818 com
.google.android.gm startService))
        [1] "Reply" -> PendingIntent(4290bd48: PendingIntentRecord(420f50b0 com.
google.android.gm startActivity))
      }
      extras={
        android.title=Donnie Tindall
        android.support.actionExtras={0=Bundle[EMPTY_PARCEL], 1=Bundle[EMPTY_PAR
CEL]}
        android.subText=donnietindall@gmail.com
        android.showChronometer=false
        android.icon=2130837727
        android.text=This is a test email
To see a test notification
        android.progress=0
        android.progressMax=0
        android.showWhen=true
        android.people=[Ljava.lang.String;@41fadfb0 {
          mailto:donnietindall@gmail.com
        }
        android.largeIcon=android.graphics.Bitmap@428a3650 (128x128)
        android.infoText=null
        android.wearable.EXTENSIONS=Bundle[mParcelledData.dataSize=1200]
        android.progressIndeterminate=false
        android.scoreModified=false
      }
```

Dumpsys conclusions

Running the dumpsys command with no service name will run dumpsys on all available services. However, the output will be very large, and should be redirected into a text file. On most platforms, the command to do this would be:

```
adb shell dumpsys > dumpsys.txt
```

This would write the output to dumpsys.txt in the current working directory. The output can then be searched, or a parsing script can be run to pull out known relevant fields.

Dumpsys is an extremely powerful tool that can be used to show information that cannot be obtained elsewhere on the device. We recommend running dumpsys on every Android device when it is seized, prior to being shut down. This will save a wide variety of information that may be useful later, and does not require root.

Bypassing Android lock screens

Lock screens are the most challenging aspect of Android forensic examinations. Frequently, the entire investigation depends on the examiner's ability to gain access to a locked device. While there are methods to bypass them, this can be highly dependent on the OS version, device settings, and technical capabilities of the examiner. There is no magical solution that will work every time on every device. Commercial forensics tools such as Cellebrite and XRY have fairly robust bypass capabilities, but are far from infallible. This chapter will show how an examiner can increase their odds of bypassing locked devices with free tools and methods.



An examiner should never attempt to guess a Pattern/PIN/Password on the device. Many manufacturers implement a setting that will wipe the device after a number of failed attempts. Many also allow the user to lower that number.

Lock screen types

There are many methods used to secure a device, and the methods for bypassing each vary:

- None/Slide
- Pattern
- PIN
- Password

- Smart Lock
    - Trusted Face
    - Trusted Location
    - Trusted Device

Other security options may exist; as Android is open source, the possibilities are only limited by the developer's imagination. These are the options that are available in the stock version of Android Lollipop released by Google. Most security options used by vendors generally use one of these stock options as a failsafe, in case a user is unable to log in with their unique options. Versions in which the setting was first used also refer to stock Android; various manufacturers may have implemented them sooner.

None/Slide lock screens

The **Slide to unlock** screen is the default setting of most Android devices. It provides no level of security, and is bypassed by sliding a finger on the screen in the indicated direction.

Pattern lock screens

Pattern lock screens are the iconic Android security method. Frequently referred to as **swipe codes** and similar names, these require the user to trace a pattern on the device with a finger. A common bypass for this lock is the **smudge attack**, looking for patterns left on the screen by the user's finger.

Password/PIN lock screens

Users familiar with Apple's iOS will recognize this option. It requires a user to type a password or PIN in order to unlock the device. These are lumped together because forensically, they are identical: they store their passwords the same way.

Smart Locks

The **Smart Lock** is a term introduced in Android Lollipop, although the Face unlock option was previously available. They require a specific condition to unlock the device: a user's face must be recognized, the user must be in a known location, or a specific other device must be nearby.

Trusted Face

Face unlock works exactly as it sounds: it uses facial recognition to determine if the user has been previously been set up as a trusted user. Older versions of Face locks were easily fooled by pictures of a trusted user, though newer versions may require the user to blink in order to unlock the device.

Trusted Location

Trusted Location is available in Android Lollipop and is also commonly referred to as **geo-fencing**. If a user is in a location that has been marked as trusted (such as home or work), the device will not lock. There is no input required from the user, but the GPS must be enabled.

Trusted Device

Trusted Device is available in Android Lollipop and works via Bluetooth; if a device that has been setup as a trusted device is nearby, the lock screen will be disabled. This may be used with smart watches, vehicles that pair over Bluetooth, Bluetooth headsets, or any other Bluetooth-capable device.



All Smart Lock options require a Pattern/PIN/Password as a backup security method. This means we only have to learn how to bypass Patterns/PINs/Passwords in order to crack all of the security options.

General bypass information

In all cases, bypassing the lock screen will require retrieving a file from the device. Pattern locks are stored as hash values at `/data/system/gesture.key` and PIN/Password locks are stored as hash values at `/data/system/password.key`. Additionally, the password.key hash is salted; the salt value is stored at `/data/data/com.android.providers.settings/databases/settings.db` prior to Android 4.4, and `/data/system/locksettings.db` on devices running Android 4.4 and higher.

If the device is locked, how is an examiner supposed to access these files? Again, there is no magic solution that works every time, but there are some options, which are as follows:

- ADB
    - Requires root
    - Requires **USB debugging**
    - Requires **Secure USB debugging** pairing (depending on OS version)

- Booting into a custom recovery mode
    - Does not require root (root will be given through the recovery image)
    - Does not require **USB debugging** (accomplished via fastboot)
    - Does not require **Secure USB debugging** (this is bypassed entirely)
    - Requires an unlocked bootloader
- JTAG/Chip-off
    - Highly advanced
    - Does not require any specific device settings or options

The files that need to be pulled to crack a PIN/password on devices prior to Android 4.4 are:

- `/data/system/password.key`
- `/data/data/com.android.providers.settings/databases/settings.db`

The files that need to be pulled to crack a PIN/password on devices running Android 4.4 and higher are:

- `/data/system/password.key`
- `/data/system/locksettings.db`

Only one file needs to be pulled to crack a Pattern lock on all versions of Android:

- `/data/system/gesture.key`



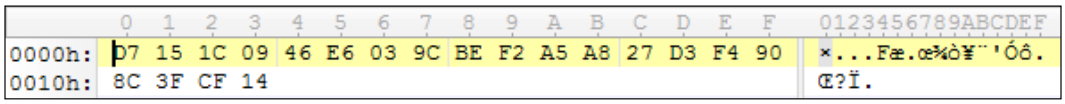
It is not always necessary to actually crack the PIN or Password. They can also be bypassed by simply overwriting or deleting the files. However, this is changing the original evidence and may not be forensically valid in your jurisdiction.

Note that the below cracking sections do NOT apply to Lollipop devices. The pattern locks are no longer unsalted, and as of the time of writing, no information has been published regarding how to recover the salt. However, the lock screen can still be bypassed by deleting the relevant files.

Many tools exist that will bypass lock screens automatically; however, in this chapter, we will show the manual process to explain what these tools are doing in the background. A good tool that is free for law enforcement can be found at `https://andriller.com/`.

Cracking an Android pattern lock

Now that we have `gesture.key`, which contains the pattern lock information, let's take a look at the file contents:



Contents of gesture.key in a hex editor

The hex contents of the file are an unsalted SHA-1 hash of the swipe pattern. The fact that there are a limited number of possible patterns (there is a four digit minimum and a nine digit maximum because each number can only be used once), the simplest method for cracking this hash is a dictionary attack. An examiner can create a dictionary consisting of every possible pattern, but re-inventing the wheel isn't always necessary. CCL Forensics, based in the UK, provides a free Python script to create the hash dictionary. It can be downloaded at `http://www.cclgroupltd.com/product/android-pattern-lock-scripts/`.

The file is `GenerateAndroidGestureRainbowTable.py`. To run it, Python 3 must be installed on the examiner's system. Python 3 can be downloaded at `https://www.python.org/downloads/`. Many forensics tools provide Python support or use it themselves, so an examiner may already have it installed. To execute the file, simply navigate to the directory containing it and run:

```
C:\Users\Android_Examiner>python GenerateAndroidGestureRainbowTable.py
2014-12-06 15:59:15.758673: Building hashes for patterns with length 3
2014-12-06 15:59:15.763673: Building hashes for patterns with length 4
2014-12-06 15:59:15.799676: Building hashes for patterns with length 5
2014-12-06 15:59:16.041689: Building hashes for patterns with length 6
2014-12-06 15:59:17.741787: Building hashes for patterns with length 7
2014-12-06 15:59:29.103436: Building hashes for patterns with length 8
2014-12-06 16:00:43.295680: Building hashes for patterns with length 9

C:\Users\Android_Examiner>
```

The script may take a while to run, possibly between 20 and 30 minutes. Once it completes, there should now be a file called `AndroidLockScreenRainbow.sqlite` in the same directory as the `GenerateAndroidGestureRainbowTable.py` script.

Now that we have a database containing the hash of every possible Android pattern, we simply need to look up the hash we found in the `gesture.key` file. This can be done manually with a SQLite viewer or even SQL commands. However, CCL Forensics also provides `Android_GestureFinder.py`, a script that will look up the hash in the database created previously.

Unfortunately, this free script doesn't quite suit our purposes. It is built to look at a physical dump binary and find a lock screen pattern; we already have the file containing the pattern. In order for the script to work properly, we will need to modify the code. `Android_GestureFinder.py` will need to be opened in some sort of a code-friendly editor; Notepad++, Sublime Text, or the Python IDLE GUI will all work. The following screenshots are from Sublime Text. Make a copy of the file, open the original and find line 85, which says:

```
if regex.match(chunk) is not None:
```

This line needs to be commented out. To do so, just place a `#` sign at the beginning of the line:

```
#if regex.match(chunk) is not None:
```

Due to Python's formatting, we will now undo indents in the lines following the statement we commented out. Lines 86, 89, 91, and 92 need to be moved to the left, so that they are in line with our commented out statement. Finally, line 94 needs to move four backspaces to the left so that it is one tab indented from the line above it. The final code should look like this:

```
85          #if regex.match(chunk) is not None:
86          lookup_hash = hexlify(chunk[:20]).decode()
87
88              # look up hash in database
89          cur.execute("SELECT pattern FROM RainbowTable WHERE hash = ?", (lookup_hash,))
90
91          result = cur.fetchone()
92          if result:
93                  # offset, hash, pattern
94              results.append([f.tell() - CHUNK_SIZE, lookup_hash, result[0]])
```

Final code for Android_GestureFinder.py

Note that line 85 now begins with `#`, lines 86, 89, 91, and 92 are in line with line 85, and line 94 is indented one tab to the right from the other lines (or four backspaces from its original location).

Now the code is ready to run against our file; save the changes to `Android_GestureFinder.py`. Ensure that the `AndroidLockScreenRainbow.sqlite` and `gesture.key` files are in the same directory as `Android_GestureFinder.py`, and run the following script:

```
C:\Users\Android_Examiner>python Android_GestureFinder.py gesture.key
Offset          Hash                                             Pattern
    -2012       d7151c0946e6039cbef2a5a827d3f4908c3fcf14         [0, 4, 8, 7, 6]
```

The output should return very quickly, as it is performing a simple lookup in the hash database. The `Offset` is negative, due to the fact that we used the script against a single file; if pointed at a binary physical dump, it would display the offset of the lock screen hash within the blob. The `Hash` column shows the hash value that was found, and the `Pattern` is the corresponding lock screen pattern:



Lock screen numbering

In this example, the pattern would begin at `0` in the top left corner, pass through **4** in the center, touch **8** in the bottom right corner, then cross the **7** in the bottom middle, and end at the **6** in the bottom left corner. This pattern can now be used on the device to bypass the lock screen.

If the script encountered errors, the file was likely not modified correctly. The following result would likely indicate that the script was not modified properly; perhaps the copy of the file was run instead of the modified version:

```
No lock patterns found in gesture.key.
```

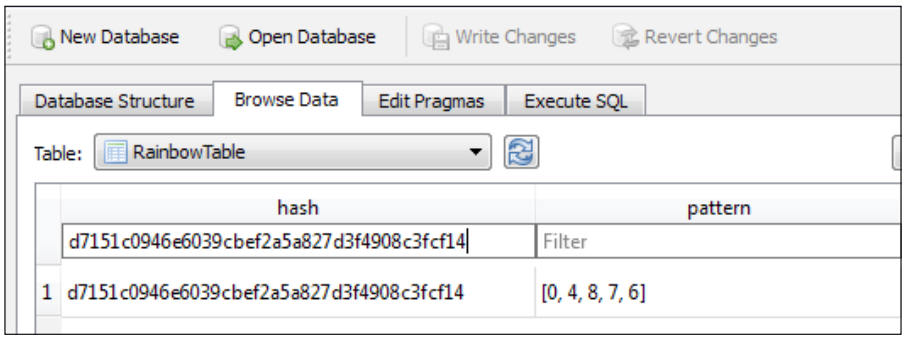
To resolve this error, verify that the script was modified as shown above. The following error is indicative of improper indentation:

```
  File "Android_GestureFinder.py", line 86
    lookup_hash = hexlify(chunk[:20]).decode()
                                               ^
IndentationError: unindent does not match any outer indentation level
```

To resolve this error, navigate to the specified line (86 in the example above), and ensure that the alignment is as shown in the preceding modified code.

If the errors cannot be resolved, or if modifying the script is too daunting to an examiner, the hash value can always be looked up manually in the hash database. An excellent free SQL viewer, DB Browser for SQLite, can be found at `http://sourceforge.net/projects/sqlitebrowser/`.

Open AndroidLockScreenRainbow.sqlite with DB Browser for SQLite, and select the Browse Data tab. Then, simply type the hash value found in gesture.key into the search field in the hash column. Note that the characters in the database are lower case; the search field is case-sensitive:

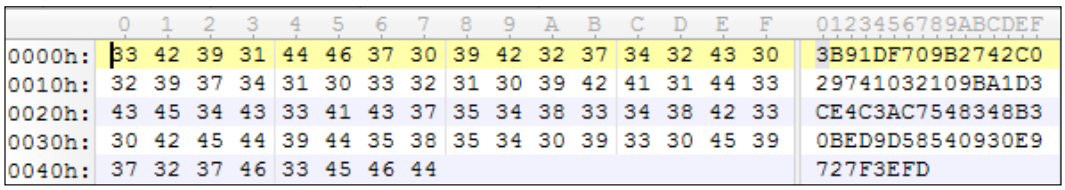


Contents of the AndroidLockScreenRainbow.sqlite file in SQLite Browser

The results are the same as if the script had been run.

Cracking an Android PIN/Password

To crack the PIN/Password lock, we'll need to take a look in the contents of the files pulled earlier. `Password.key` is very similar to `gesture.key`; it contains a hash of the password as shown in following screenshot:

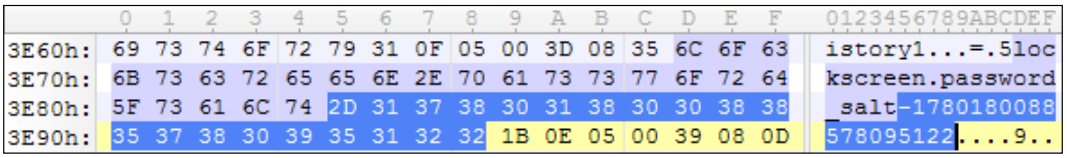


	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	33	42	39	31	44	46	37	30	39	42	32	37	34	32	43	30	3B91DF709B2742C0
0010h:	32	39	37	34	31	30	33	32	31	30	39	42	41	31	44	33	29741032109BA1D3
0020h:	43	45	34	43	33	41	43	37	35	34	38	33	34	38	42	33	CE4C3AC7548348B3
0030h:	30	42	45	44	39	44	35	38	35	34	30	39	33	30	45	39	0BED9D58540930E9
0040h:	37	32	37	46	33	45	46	44									727F3EFD

Contents of password.key in a hex editor

However, this time the hash is salted. To have a chance at cracking it, the salt will have to be recovered. As noted above, its location will be dependent on the version of Android the device is running. If the device is running version 4.3 or lower, it will be located in the `settings.db` file within the `secure` table. In 4.4 or higher, it will be in the `locksettings.db` within the `locksettings` table. The following example shows `locksettings.db`, but the process is identical for both.

Within the database file, we'll need to locate the `lockscreen.password_salt` key. This can be done in a SQL browser, or just by opening the file in a hex editor and searching. The salt value is highlighted as follows:



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
3E60h:	69	73	74	6F	72	79	31	0F	05	00	3D	08	35	6C	6F	63	istory1...=.5loc
3E70h:	6B	73	63	72	65	65	6E	2E	70	61	73	73	77	6F	72	64	kscreen.password
3E80h:	5F	73	61	6C	74	2D	31	37	38	30	31	38	30	30	38	38	_salt-1780180088
3E90h:	35	37	38	30	39	35	31	32	32	1B	0E	05	00	39	08	0D	578095122....9..

Contents of locksettings.db in a hex editor

For the Pattern lock, we were able to use a dictionary attack to quickly break the pattern, because there were a relatively small number of possibilities. With a salted hash, dictionary attacks are infeasible, so instead we will simply brute-force it.

Once again, CCL Forensics provides a useful Python script for this purpose. Other cracking tools, such as hashcat, could also be used. The CCL Forensics PIN/Password tool can be downloaded for free at `http://www.cclgroupltd.com/product/android-pin-password-lock-tool/`. Two files will be downloaded, `BruteForceAndroidPin.py` and `RecoverAndroidPIN.py`. `RecoverAndroidPIN.py` is for locating the necessary files within a physical image; we won't be needing it for our purposes.

The format for `BruteForceAndroidPIN.py` is:

```
python BruteForceAndroidPIN.py <hash> <salt> <max code length (4-16)> t
```

The `t` argument on the end is used to indicate the hash is of a password; it is not needed for cracking a PIN and would simply increase the time it takes to run.

The hash value shown above is from a PIN, so we simply need to fill in the `<hash>`, `<salt>`, and `<max code length>` fields:

```
C:\Users\Android_Examiner>python BruteForceAndroidPin.py 3B91DF709B2742C02974103
2109BA1D3CE4C3AC7548348B30BED9D58540930E9727F3EFD -1780180088578095122 16
Passcode: 2587
```

The PIN from this example was `2587` as indicated in the output. It took less than a second for this PIN to crack, however longer PINs or even short passwords may take significantly longer.

Android SIM card extractions

Traditionally, SIM cards were used for transferring data between devices. In the past, SIM cards were used to store many different types of data, such as:

- User data
    - Contacts
    - SMS messages
    - Dialed calls
- Network data
    - **Integrated Circuit Card Identifier** (**ICCID**): Serial number of the SIM
    - **International Mobile Subscriber Identity** (**IMSI**): Identifier that ties the SIM to a specific user account
    - **MSISDN**: Phone number assigned to the SIM
    - **Location Area Identity** (**LAI**): Identifies the cell that a user is in
    - **Authentication Key** (**Ki**): Used to authenticate to the mobile network
    - Various other network-specific information

With the rise in capacity of device storage, SD cards, and cloud backups, the necessity for storing data on a SIM card has decreased. As such, most modern smartphones typically do not store much, if any, user data on the SIM card. All network data listed above does still reside on the SIM, as a SIM is necessary to connect to all modern (4G) cellular networks.

As with all Android devices, though there is no concrete stipulation that user data can't be stored on a SIM, it simply doesn't happen by default. Individual device manufacturers can easily decide to write user data to the SIM, and individual users can download applications to provide that functionality. This means that a device's SIM card should always be examined during a forensic examination. It is a very quick process, and should never be overlooked.

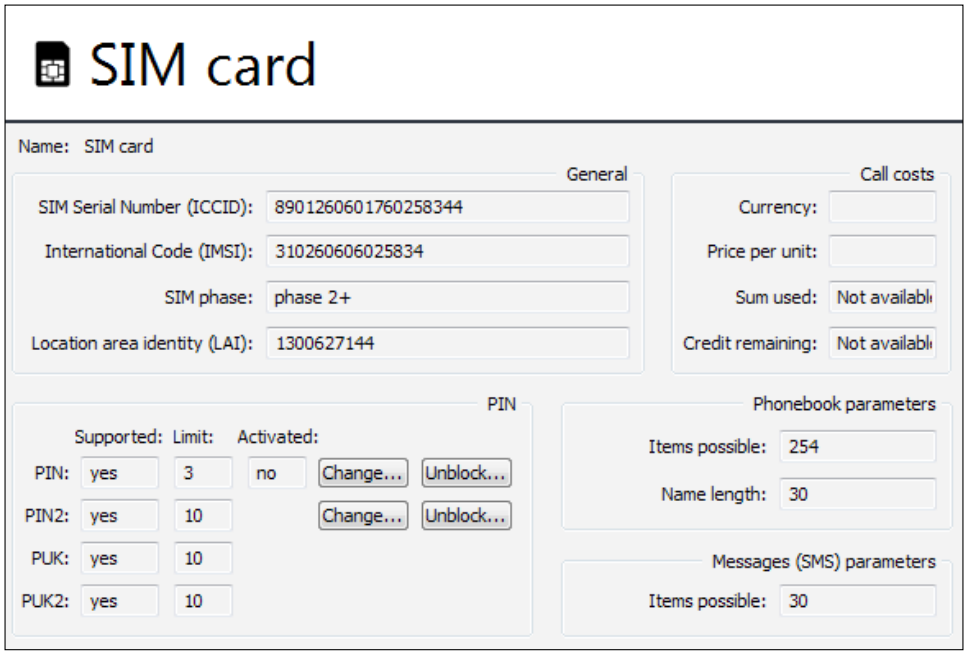
Acquiring SIM card data

The SIM card should always be removed from the device and examined separately. While some tools claim to read the SIM card through the device interface, this may not recover deleted data or all data on the SIM; the only way for an examiner to be certain all data was acquired is to read the SIM through a standalone SIM card reader with a tool that has been tested and verified.

The location of the SIM will vary by device, but is typically either stored beneath the battery or in a tray located on the side of the device. Once the SIM is removed, it should be placed in a SIM card reader. There are hundreds of SIM card readers available in the marketplace, and all major mobile forensics tools come with an included reader that will work with their software. Often, the forensic tools will also support third-party SIM readers as well.

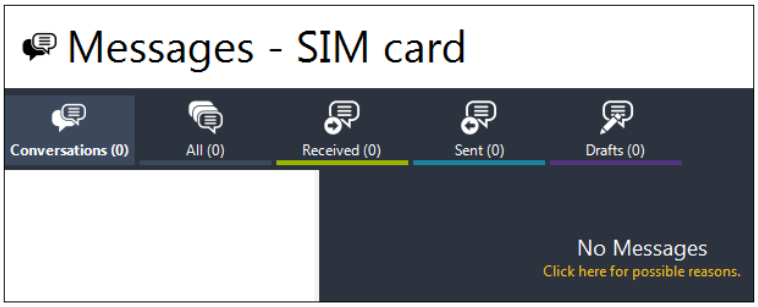
There is a surprising lack of thorough, free SIM card reading softwares. Any software used should always be tested and validated on a SIM card that has been populated with known data prior to being used in an actual forensic investigation. Also, keep in mind that much of the free software available works for older 2G/3G SIMs, but may not work properly on a modern 4G SIM. We used the Mobiledit! Lite, a free version of Mobiledit!, for the following screenshots. It is available at `http://www.mobiledit.com/downloads`.

The following screenshot shows a sample 4G SIM card extraction from an Android phone running version 4.4.4; note that nothing that could be considered user data was acquired, despite the SIM being used actively for over a year, though fields such as the ICCID, IMSI, and MSISDN (own phone number) could be useful for subpoenas/warrants or other aspects of an investigation.



SIM card extraction overview

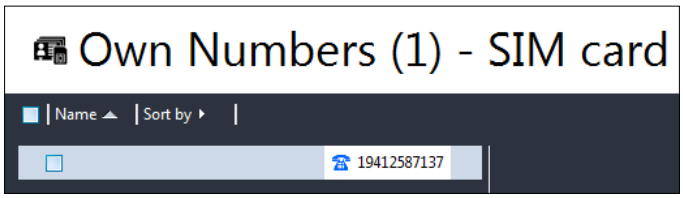
The following screenshot highlights SMS messages on the SIM card:



The following image highlights the phonebook of the SIM card:



The following image highlights the phone number of the SIM card (also called the MSISDN):



SIM security

Due to the fact that SIM cards conform to established international standards, all SIM cards provide the same security functionality: a 4- to 8-digit PIN. Generally, this PIN must be set through a menu on the device. On Android devices, this setting is found at **Settings** | **Security** | **Set up SIM card lock**. The SIM PIN is completely independent of any lock screen security settings, and only has to be entered when the device boots. The SIM PIN only protects user data on the SIM; all network information is still recoverable even if the SIM is PIN locked.

The SIM card will allow three attempts to enter the PIN. If one of these attempts is correct, the counter will reset. On the other hand, if all of these attempts are incorrect the SIM will enter **Personal Unblocking Key** (**PUK**) mode. The PUK is an 8-digit number assigned by the carrier, and is frequently found on documentation when the SIM is purchased. Bypassing a PUK is not possible with any commercial forensic software; because of this, an examiner should never attempt to enter the PIN on the device as the device will not indicate how many attempts remain before the PUK is activated. An examiner could unwittingly PUK-lock the SIM, and be unable to access the device. Forensic tools, however, will show how many attempts remain before the PUK is activated, as seen in the screenshots preceding.



Common carrier defaults for SIM PINs are 0000 and 1234. If 3 tries remain before activating the PUK, an examiner may successfully unlock the SIM with one of these defaults.

Carriers frequently retain PUK keys when a SIM is issued. These may be available through a subpoena or warrant issued to the carrier.

SIM cloning

The SIM PIN itself provides almost no additional security, and can easily be bypassed through SIM cloning. SIM cloning is a feature provided in almost all commercial mobile forensic software, although the term cloning is somewhat misleading. SIM cloning, in the case of mobile forensics, is the process of copying the network data from a locked SIM onto a forensically sterile SIM that does not have the PIN activated. The phone will identify the cloned SIM based on this network data (typically the ICCID and IMSI) and think that it is the same SIM that was inserted previously, but this time there will be no SIM PIN. This cloned SIM will also be unable to access the cellular network, which makes it an effective solution similar to Airplane mode. Therefore, SIM cloning will allow an examiner to access the device, but the user data on the original SIM is still inaccessible as it remains protected by the PIN.

We are unaware of any free software that performs forensic SIM cloning. However, it is supported by almost all commercial mobile forensic kits. These kits will typically include a SIM card reader, software to perform the clone, as well as multiple blank SIM cards for the cloning process.

Issues and opportunities with Android Lollipop

As noted several times in this chapter, the recent unveiling of Android Lollipop Version 5.0 has introduced many strong security features, which of course creates complications for forensic examiners. It was initially announced that Android Lollipop devices would ship with full disk encryption by default, however, Google later retracted this requirement due to performance issues on many devices. Instead of a requirement, Google only strongly suggests that full disk encryption be enabled when the user first creates an account. Google has also hinted that this will be a requirement in future OS versions, more information can be found in section **9.9** at `http://static.googleusercontent.com/media/source.android.com/en/us/compatibility/android-cdd.pdf`.

Devices with full disk encryption enabled make bypassing locked devices all but impossible, because even if the key files could be recovered, they would be encrypted; though surely the commercial tool manufacturers will eventually catch up. At the time of this writing, there is no known method for bypassing a locked, encrypted Lollipop device, unless it has **USB debugging** enabled and previously remembered a computer's RSA key to bypass **Secure USB debugging**. In this case, the `adbkey` and `adbkey.pub` files can be pulled from the suspect's computer and placed on an examination machine; the device will then think it is communicating with a known, approved computer. The `adbkey` and `adbkey.pub` files can be found at `C:\Users\<username>\.android` on Windows computers and `/Users/<username>/.android` on Apple computers.

There is one significant advantage for examiners performing Lollipop forensics: the Smart Locks mentioned earlier in this chapter. Smart Locks allow a user to set conditions that, if met, will leave the device unlocked without the need to enter the password even once. If the examination is performed at a Trusted Location that has been enabled, the examiner will not need to bypass the lock. The same is true if a Trusted Device is nearby and turned on while the device is being examined. However, there is no indication on the device that a trusted device is being used; the device just appears to not have a lock screen. Thus, securing all digital evidence from a scene becomes even more critical. Devices that would have previously been overlooked, such as Bluetooth headsets, may turn out to be the key that gets the examiner past a locked device. It is becoming increasingly common for devices to be paired with vehicles, so an examiner may have to perform an extraction while sitting in the suspect's vehicle! The additional security of Lollipop means that examiners may have to get more creative with the forensic process.

Android Lollipop also brings multi-user support to all devices, which was previously limited to tablets. On a device with multiple accounts, data for all users is still found in the `/data` partition, but resides in a slightly different location. If multiple accounts are setup on the device, the app data directory for each account can be found in `/data/user`. Each user will have a unique number assigned; `0` is the first account setup on the device. The `/data/user/0` directory is actually a symbolic link to `/data/data`. The second account was in the `10` directory, which directly contained all application data for the second user. Each user added is stored in a directory incremented by `10`; i.e. the third user is in `/data/user/20`, the fourth user is in `/data/user/30`, and so on.

Summary

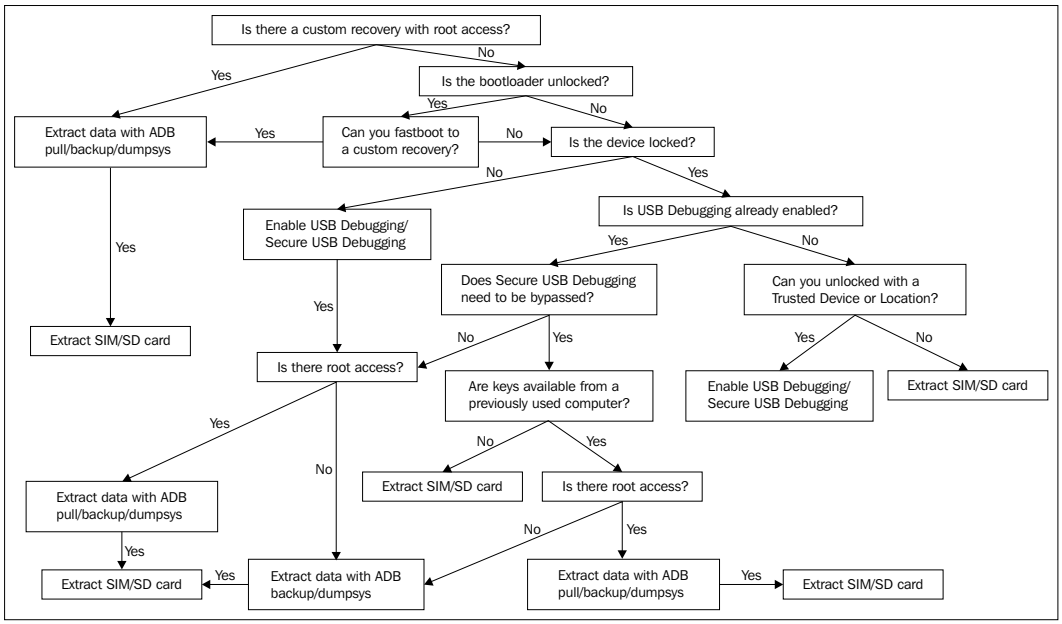
This chapter has covered many topics related to logical extractions of Android devices. As a recap, the various methods and their requirements are as follows:

Method	Requirements
ADB pull	• **USB debugging** enabled • **Secure USB debugging** bypassed on 4.2.2+ • Root access to obtain user data
ADB pull from Recovery Mode	• Must be a custom recovery to enable ADB access • Root access to obtain user data
Fastboot to boot from custom recovery image	• Unlocked bootloader • Boot image for device
ADB backup	• **USB debugging** enabled • **Secure USB debugging** bypassed on 4.2.2+ • Must be done from a running device (not Recovery mode)
ADB dumpsys	• **USB debugging** enabled • **Secure USB debugging** bypassed on 4.2.2+ • Must be done from a running device (not recovery mode)
SIM card extraction	• None, should be done independent of device

Additionally, valuable user data can be recovered from the SD card, which will be covered in *Chapter 5*, *Extracting Data Physically from Android Devices*.

If a screen is locked, an examiner can pull the key files using the methods listed above and crack them in order to bypass it.

There is a lot of data in this chapter. To help simplify it somewhat, a suggested *best practices* flow chart is shown as follows:



Android Forensics Flow Chart

5
Extracting Data Physically from Android Devices

This chapter will be covering physical data extraction using free and open source tools wherever possible. The majority of the material covered in this chapter will use the ADB methods previously discussed in this book. By the end of this chapter, the reader should be familiar with the following concepts:

- What physical extraction means
- What data to expect from physical extractions
- Physical data extractions using the `dd` and `nanddump` commands
- RAM imaging and analysis
- SD card acquisitions
- JTAG and chip-off methods

Physical extraction overview

In digital forensics, a physical extraction is an exact bit-for-bit image of the electronic media, and this definition remains true for mobile devices too. In traditional computer forensics, this typically involves removing the evidence drive from the suspect's computer and imaging it via a write blocker without ever booting the drive, resulting in an image file containing an exact copy of the suspect's drive. The output is frequently referred to as a **raw image**, or simply a **bin** (binary) file. Physical extractions differ from logical, in that, they are an exact copy of the device's memory, and include unallocated space, file slack, volume slack, and so on.

In mobile forensics, the result is the same; an exact bit-for-bit image of the device, but the methods are somewhat different. For example, removing the flash memory from the device to image can be both time-consuming and expensive, and requires a lot of specialized knowledge (though it can be done as discussed in the chip-off section later in the chapter). Furthermore, short of using advanced **Joint Test Action Group** (**JTAG**) or chip-off methods, the device must be booted to some degree (and written to in many cases) in order to access the data. Finally, finding a tool that can even parse the final image file can be very difficult. Hard drive images and file systems have long been documented and studied, while mobile images and file systems change frequently; in some cases mobile file systems are even unique to a specific manufacturer. Knowing what to do with the image after it is acquired can be just as challenging as acquiring the image in the first place!

Many of the techniques discussed in *Chapter 4*, *Extracting Data Logically from Android Devices*, will still apply here. Booting into a custom recovery is still the most forensically sound process; physically acquiring a live device should be avoided if at all possible.

What data can be acquired physically?

The short answer is: *everything*. As a physical acquisition is an exact image of the device, every bit of data on the device is in the image file. As mentioned previously, with a physical extraction, an examiner is usually only limited by their ability to find the relevant data. Generally, this is due to a lack of good image analysis tools in the mobile forensics space. To further compound the matter, applications have been known to encode or otherwise obfuscate user data, so simply browsing through the image in a hex editor will frequently miss valuable evidence. This chapter will show various methods to mount or otherwise view the filesystem of a physical extraction, while *Chapter 7*, *Forensic Analysis of Android Applications*, will focus on analyzing data from specific applications.

Root access

Once again, just as in logical extractions, root access is going to be a critically important aspect of physical extractions. To manually image a device, we are going to have to execute commands on the device from the ADB shell, and these will require root permissions. If root access cannot be obtained, the SD card can generally still be imaged. The only recourse beyond that is JTAG or chip-off methods.

Extracting data physically with dd

The `dd` command should be familiar to any examiner who has done traditional hard drive forensics. The `dd` command is a Linux command-line utility used by definition to convert and copy files, but is frequently used in forensics to create bit-by-bit images of entire drives. Many variations of the `dd` commands also exist and are commonly used, such as `dcfldd`, `dc3dd`, `ddrescue`, and `dd_rescue`. As the `dd` command is built for Linux-based systems, it is frequently included on Android platforms. This means that a method for creating an image of the device often already exists on the device!

The `dd` command has many options that can be set, of which only forensically important options are listed here. The format of the `dd` command is as follows:

```
dd if=/dev/block/mmcblk0 of=/sdcard/blk0.img bs=4096
conv=notrunc,noerror,sync
```

- `if`: This option specifies the path of the input file to read from.
- `of`: This option specifies the path of the output file to write to.
- `bs`: This option specifies the block size. Data is read and written in the size of the block specified, defaults to 512 bytes if not specified.
- `conv`: This option specifies the conversion options as its attributes:
    - `notrunc`: This option does not truncate the output file.
    - `noerror`: This option continues imaging if an error is encountered.
    - `sync`: In conjunction with the `noerror` option, this option writes `\x00` for blocks with an error. This is important for maintaining file offsets within the image.



Do not mix up the `if` and `of` flags, this could result in overwriting the target device!

A full list of command options can be found at `http://man7.org/linux/man-pages/man1/dd.1.html`.

Note that there is an important correlation between the block size and the `noerror` and `sync` flags: if an error is encountered, `\x00` will be written for the entire block that was read (as determined by the block size). Thus, smaller block sizes result in less data being missed in the event of an error. The downside is that, typically, smaller block sizes result in a slower transfer rate. An examiner will have to decide whether a timely or more accurate acquisition is preferred.

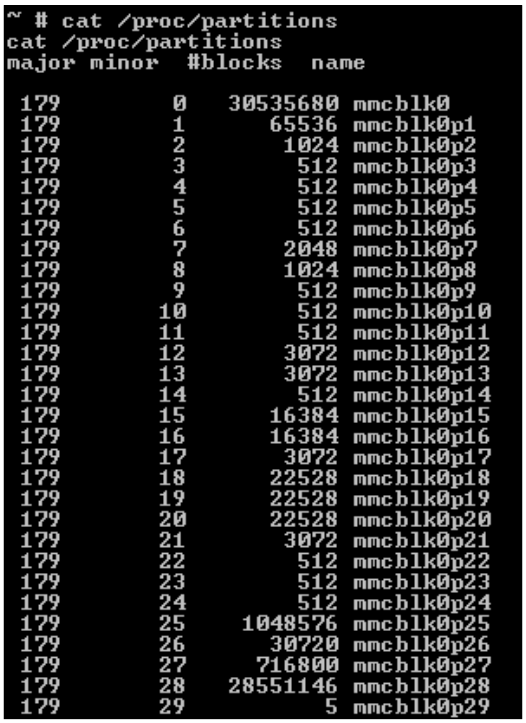
As discussed in the previous chapter, booting into recovery mode for the imaging process is the most forensically sound method.

Determining what to image

When imaging a computer, an examiner must first find what the drive is mounted as; `/dev/sda`, for example. The same is true when imaging an Android device. The first step is to launch the ADB shell and view the `/proc/partitions` file using the following command:

```
cat /proc/partitions
```

The output will show all partitions on the device:



```
~ # cat /proc/partitions
cat /proc/partitions
major minor  #blocks  name

 179        0   30535680 mmcblk0
 179        1      65536 mmcblk0p1
 179        2       1024 mmcblk0p2
 179        3        512 mmcblk0p3
 179        4        512 mmcblk0p4
 179        5        512 mmcblk0p5
 179        6        512 mmcblk0p6
 179        7       2048 mmcblk0p7
 179        8       1024 mmcblk0p8
 179        9        512 mmcblk0p9
 179       10        512 mmcblk0p10
 179       11        512 mmcblk0p11
 179       12       3072 mmcblk0p12
 179       13       3072 mmcblk0p13
 179       14        512 mmcblk0p14
 179       15      16384 mmcblk0p15
 179       16      16384 mmcblk0p16
 179       17       3072 mmcblk0p17
 179       18      22528 mmcblk0p18
 179       19      22528 mmcblk0p19
 179       20      22528 mmcblk0p20
 179       21       3072 mmcblk0p21
 179       22        512 mmcblk0p22
 179       23        512 mmcblk0p23
 179       24        512 mmcblk0p24
 179       25    1048576 mmcblk0p25
 179       26      30720 mmcblk0p26
 179       27     716800 mmcblk0p27
 179       28   28551146 mmcblk0p28
 179       29          5 mmcblk0p29
```

In the output shown in the preceding screenshot, `mmcblk0` is the entirety of the flash memory on the device. To image the entire flash memory, we could use `/dev/blk/mmcblk0` as the input file flag (`if`) for the `dd` command. Everything following it, indicated by `p1`- `p29`, is a partition of the flash memory. The size is shown in blocks, in this case the block size is 1024 bytes for a total internal storage size of approximately 32 GB. To obtain a full image of the device's internal memory, we would run the `dd` command with `mmcblk0` as the input file.

However, we know from previous chapters that most of these partitions are unlikely to be forensically interesting; we're most likely only interested in a few of them. To view the corresponding names for each partition, we can look in the device's `by-name` directory. This does *not* exist on every device, and is sometimes in a different path, but for this device it is found at `/dev/block/msm_sdcc.1/by-name`. By navigating to that directory and running the `ls -al` command, we can see to where each block is symbolically linked as shown in the following screenshot:

```
/dev/block/platform/msm_sdcc.1/by-name # ls -al
ls -al
__bionic_open_tzdata: couldn't find any tzdata when looking for localtime!
__bionic_open_tzdata: couldn't find any tzdata when looking for GMT!
__bionic_open_tzdata: couldn't find any tzdata when looking for posixrules!
drwxr-xr-x    2 root     root           620 Jul  3 20:29 .
drwxr-xr-x    4 root     root           700 Jul  3 20:29 ..
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 DDR -> /dev/block/mmcblk0p24
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 aboot -> /dev/block/mmcblk0p6
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 abootb -> /dev/block/mmcblk0p11
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 boot -> /dev/block/mmcblk0p19
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 cache -> /dev/block/mmcblk0p27
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 crypto -> /dev/block/mmcblk0p26
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 fsc -> /dev/block/mmcblk0p22
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 fsg -> /dev/block/mmcblk0p21
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 grow -> /dev/block/mmcblk0p29
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 imgdata -> /dev/block/mmcblk0p17
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 laf -> /dev/block/mmcblk0p18
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 metadata -> /dev/block/mmcblk0p14
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 misc -> /dev/block/mmcblk0p15
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 modem -> /dev/block/mmcblk0p1
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 modemst1 -> /dev/block/mmcblk0p12
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 modemst2 -> /dev/block/mmcblk0p13
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 pad -> /dev/block/mmcblk0p7
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 persist -> /dev/block/mmcblk0p16
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 recovery -> /dev/block/mmcblk0p20
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 rpm -> /dev/block/mmcblk0p3
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 rpmb -> /dev/block/mmcblk0p10
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 sbl1 -> /dev/block/mmcblk0p2
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 sbl1b -> /dev/block/mmcblk0p8
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 sdi -> /dev/block/mmcblk0p5
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 ssd -> /dev/block/mmcblk0p23
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 system -> /dev/block/mmcblk0p25
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 tz -> /dev/block/mmcblk0p4
lrwxrwxrwx    1 root     root            20 Jul  3 20:29 tzb -> /dev/block/mmcblk0p9
lrwxrwxrwx    1 root     root            21 Jul  3 20:29 userdata -> /dev/block/mmcblk0p28
```

If our investigation was only interested in the `userdata` partition, we now know that it is `mmcblk0p28`, and could use that as the input file to the `dd` command.

If the `by-name` directory does not exist on the device, it may not be possible to identify every partition on the device. However, many of them can still be found by using the `mount` command within the ADB shell. Note that the following screenshot is from a different device that does not contain a `by-name` directory, so the data partition is not `mmcblk0p28`.

```
~ # mount
mount
rootfs on / type rootfs (rw,seclabel)
tmpfs on /dev type tmpfs (rw,seclabel,nosuid,relatime,mode=755)
devpts on /dev/pts type devpts (rw,seclabel,relatime,mode=600)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,seclabel,relatime)
selinuxfs on /sys/fs/selinux type selinuxfs (rw,relatime)
tmpfs on /storage type tmpfs (rw,seclabel,relatime,mode=050,gid=1028)
tmpfs on /mnt/secure type tmpfs (rw,seclabel,relatime,mode=700)
tmpfs on /mnt/fuse type tmpfs (rw,seclabel,relatime,mode=775,gid=1000)
/dev/block/mmcblk0p33 on /cache type ext4 (rw,seclabel,nodev,noatime,nodiratime,data=ordered)
/dev/block/mmcblk0p34 on /data type ext4 (rw,seclabel,nodev,noatime,nodiratime,data=ordered)
/dev/block/mmcblk0p32 on /system type ext4 (rw,seclabel,nodev,noatime,nodiratime,data=ordered)
/dev/block/mmcblk0p24 on /devlog type ext4 (rw,seclabel,nodev,noatime,nodiratime,data=ordered)
```

On this device, the data partition is `mmcblk0p34`. If the `mount` command does not work, the same information can be found using the `cat /proc/mounts` command. Other options to identify partitions depending on the device are the `cat /proc/mtd` or `cat /proc/yaffs` commands; these may work on older devices. Newer devices may include an `fstab` file in the root directory (typically called `fstab.<device>`) that will list mountable partitions.

Writing to an SD card

The output file of the `dd` command can be written to the device's SD card. This should only be done if the suspect SD card can be removed and replaced with a forensically sterile SD to ensure that the `dd` command's output is not overwriting evidence. Obviously, if writing to an SD card, ensure that the SD card is larger than the partition being imaged.



On newer devices, the `/sdcard` partition is actually a symbolic link to `/data/media`. In this case, using the `dd` command to copy the `/data` partition to the SD card won't work, and could corrupt the device because the input file is essentially being written to itself.

To determine where the SD card is symbolically linked to, simply open the ADB shell and run the `ls -al` command. If the SD card partition is not shown, the SD likely needs to be mounted in recovery mode using the steps shown in *Chapter 4*, *Extracting Data Logically from Android Devices*.

In the following example, `/sdcard` is symbolically linked to `/data/media`. This indicates that the `dd` command's output should not be written to the SD card.

```
lrwxrwxrwx    1 root     root            11 Jul  3 20:29 sdcard -> /data/media
```

In the example that follows, the `/sdcard` is not a symbolic link to `/data`, so the `dd` command's output can be used to write the `/data` partition image to the SD card:

```
lrwxrwxrwx root     root              2014-12-12 20:32 sdcard -> /storage/emulated/legacy
```

On older devices, the SD card may not even be symbolically linked.

After determining which block to read and to where the SD card is symbolically linked, image the `/data` partition to the `/sdcard`, using the following command:

```
dd if=/dev/block/mmcblk0p28 of=/sdcard/data.img bs=512
conv=notrunc,noerror,sync
```

```
~ # dd if=/dev/block/mmcblk0p32 of=/sdcard/data.img bs=512 conv=notrunc,noerror,sync
dd if=/dev/block/mmcblk0p32 of=/sdcard/data.img bs=512 conv=notrunc,noerror,
sync
3801086+0 records in
3801086+0 records out
1946156032 bytes (1.8GB) copied, 170.866351 seconds, 10.9MB/s
```

Now, an image of the `/data` partition exists on the SD card. It can be pulled to the examiner's machine with the ADB `pull` command, or simply read from the SD card.

Writing directly to an examiner's computer with netcat

If the image cannot be written to the SD card, an examiner can use **netcat** to write the image directly to their machine. The netcat tool is a Linux-based tool used for transferring data over a network connection. We recommend using a Linux or a Mac computer for using netcat as it is built-in, though Windows versions do exist. The examples below were done on a Mac.

Installing netcat on the device

Very few Android devices, if any, come with netcat installed. To check, simply open the ADB shell and type `nc`. If it returns saying `nc is not found`, netcat will have to be installed manually on the device. Netcat compiled for Android can be found at many places online. We have shared the version we used at `http://sourceforge.net/projects/androidforensics-netcat/files/`.

If we look back at the results from our `mount` command in the previous section, we can see that the `/dev` partition is mounted as `tmpfs`. The Linux term `tmpfs` means that the partition is meant to appear as an actual filesystem on the device, but is truly only stored in RAM. This means we can `push` netcat here without making any permanent changes to the device using the following command on the examiner's computer:

```
adb push nc /dev/Examiner_Folder/nc
```

The command should have created the `Examiner_Folder` in `/dev`, and `nc` should be in it. This can be verified by running the following command in the ADB shell:

```
ls /dev/Examiner_Folder
```

Using netcat

Now that the netcat binary is on the device, we need to give it permission to execute from the ADB shell. This can be done as follows:

```
chomd +x /dev/Examiner_Folder/nc
```

We will need two terminal windows open with the ADB shell open in one of them. The other will be used to listen to the data being sent from the device.

Now we need to enable port forwarding over ADB from the examiner's computer:

```
adb forward tcp:9999 tcp:9999
```

`9999` is the port we chose to use for netcat; it can be any arbitrary port number between 1023 and 65535 on a Linux or Mac system (1023 and below are reserved for system processes, and require root permission to use). Windows will allow *any* port to be assigned.

In the terminal window with ADB shell, run the following command:

```
dd if=/dev/block/mmcblk0p34 bs=512 conv=notrunc,noerror,sync | /dev/
Examiner_Folder/nc -l -p 9999
```



`mmcblk0p34` is the user data partition on this device, however, the entire flash memory or any other partition could also be imaged with this method. In most cases, it is best practice to image the entirety of the flash memory in order to acquire all possible data from the device. Some commercial forensic tools may also require the entire memory image, and may not properly handle an image of a single partition.

In the other terminal window, run:

```
nc 127.0.0.1 9999 > data_partition.img
```

The `data_partition.img` file should now be created in the current directory of the examiner's computer. When the data is finished transferring, netcat in both terminals will terminate and return to the command prompt. The process can take a significant amount of time depending on the size of the image.

Extracting data physically with nanddump

In all of the preceding examples, the partitions were all **Multimedia Card** (**MMC**) blocks, which is typically seen in newer devices. Older devices, however, are far more likely to consist of **Memory Technology Device** (**MTD**) blocks. We have seen cases in the past where the `dd` command was unable to properly image an MTD block, although more often than not, it works fine. If `dd` fails, there is a widely distributed utility called `MTD-Utils` used to read and write from MTD blocks; the `nanddump` command is a part of `MTD-Utils`, and can be used similarly to `dd` in order to read from an MTD block. In those cases where `dd` failed, `nanddump` was always successful.

Versions of `nanddump` compiled for Android can be found in many places online; we used the one found at `https://github.com/jakev/android-binaries/blob/master/nanddump`.

The process to put `nanddump` on the device is the same as the one used previously for netcat:

```
adb push nanddump /dev/Examiner_Folder/nanddump
chmod +x /dev/Examiner_Folder/nanddump
```

Just like `dd`, the `nanddump` command can be used to write either to an SD card or the examiner's computer via netcat. From a terminal window, run the following command:

```
adb forward tcp:9999 tcp:9999
```

From a separate terminal window within the ADB shell, run the following command:

```
/dev/Examiner_Folder/nanddump /dev/block/mmcblk0p34 | /dev/Examiner_
Folder/nc -l -p 9999
```

In the first terminal window where the `adb forward` command was used, run the following command:

```
nc 127.0.0.1 9999 > data_partition.img
```

Verifying a full physical image

Verifying whether an image file is identical to the device is a critical step in traditional digital forensics. On Android devices, it can be a little trickier, if not impossible. The image created can be hashed using whatever tool the examiner typically uses. Verifying the memory on the device can be done through the ADB shell using the following command, where the path given is the block or partition that was imaged:

```
md5sum /dev/block/mmcblk0
```

However, the `md5sum` command is not included on all Android devices. If it is not included, an examiner may be able to find a version compiled for their device online, and push it to the device in a `tmpfs` partition as shown previously with netcat and `nanddump`.

Another issue is that if the image was acquired live, i.e. not in recovery mode as discussed in the previous chapter, it is a virtual certainty that the MD5 hashes will *not* match, as data is constantly changing on the device (even if it is **radio frequency** (**RF**) shielded or in Airplane mode). In this case, an examiner would have to document that the device was live when acquired and explain that the hashes are not expected to match.

Analyzing a full physical image

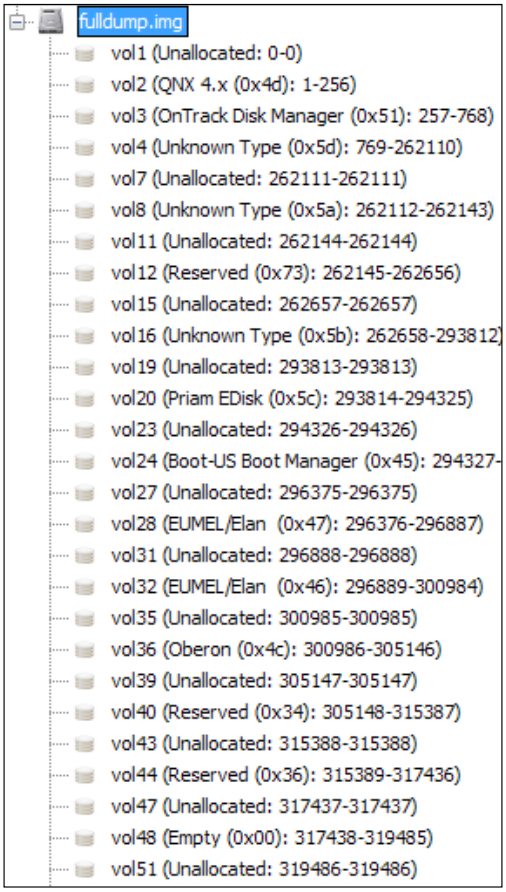
Once an image has been obtained using one of the discussed methods, an examiner could conceivably go through the image manually and extract each partition, but would probably prefer to avoid doing that. Luckily, there is a wide variety of mobile forensic tools that can ingest a physical image, such as Cellebrite, XRY, Mobile Phone Examiner, and many others. Unfortunately, none of these are free or open source. By far the most popular analysis tool that is free and open source is Autopsy by Brian Carrier.

Autopsy

The Sleuth Kit began as a set of Linux-based command line tools for forensics; eventually, a browser-based GUI named Autopsy was added. Recently, Autopsy has been released as a stand-alone platform on Windows, and includes support for analyzing Android images. Version 3.1.1 is shown in the following screenshots. The full process for loading and analyzing an image will be covered in *Chapter 8*, *Forensic Analysis of Android Applications*.

Autopsy can be downloaded at `http://www.sleuthkit.org/`.

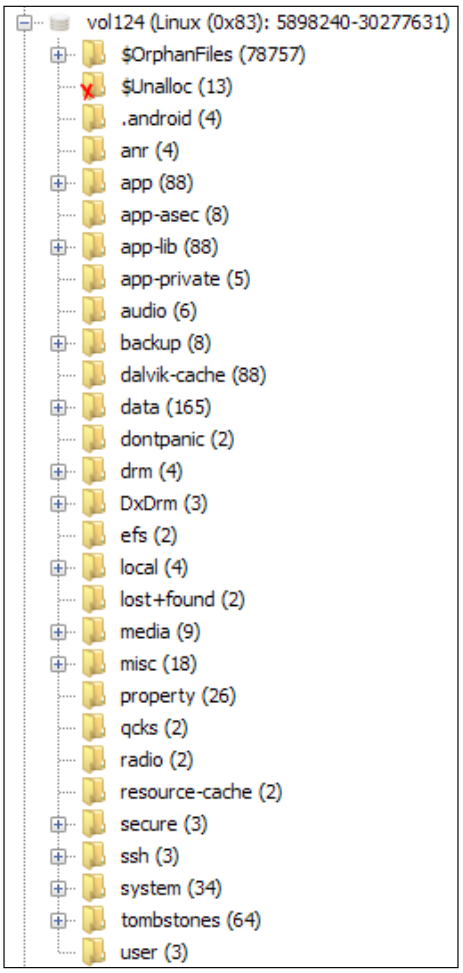
Once the image has been loaded, expanding the image will show all the volumes that Autopsy found, as shown in the following screenshot:



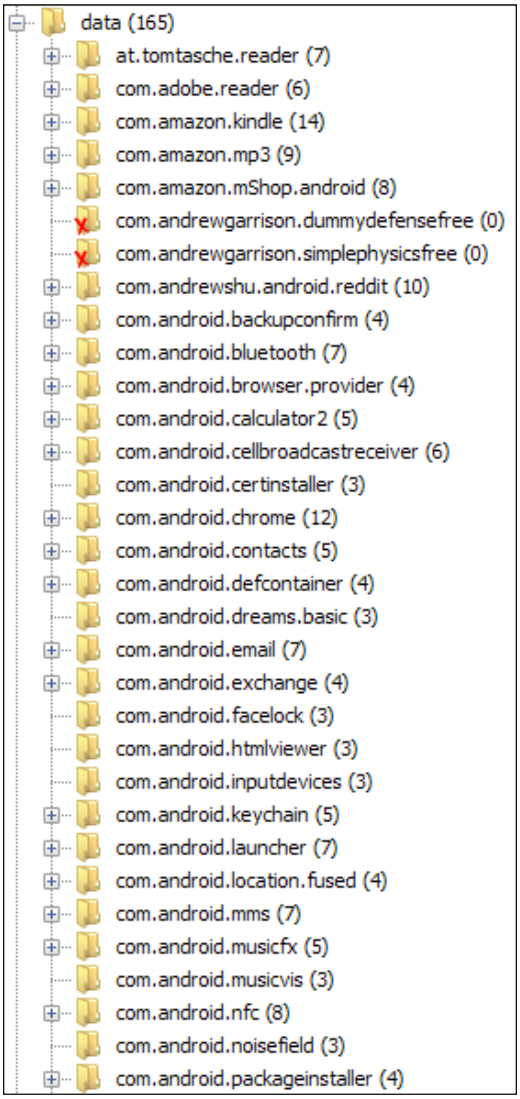


Far more volumes were found than the number of partitions on the device. They may either be false positives created by the tool, or a result of wear-leveling on the device.

One of these volumes will be the data partition, seen as follows:.

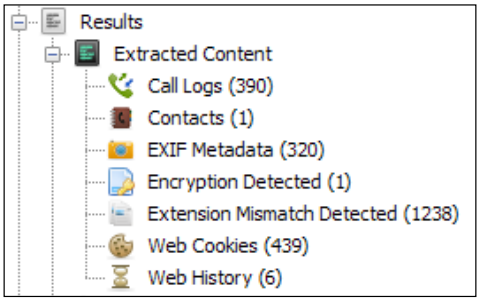


Note that the media directory seen above is the SD card, as it was symbolically linked to the `data` partition. The `data` folder within the `data` partition will contain application data:



As each application is installed, a directory is created for it. The directories shown in the preceding screenshot with a red cross on the folder icon were deleted, indicating that the application was removed from the device.

Finally, Autopsy does a good job pulling out some data automatically for an examiner, but as with all forensic tools, this information should be verified manually, as seen in *Chapter 7, Forensic Analysis of Android Applications*.



Issues with analyzing physical dumps

The most common problem we see on many forensic forums and email lists is examiners obtaining a physical dump and then not being able to load that dump into a tool that claims to support the device. Most of the time, this is because the examiner fails to account for the **Out-of-Band** (**OOB**) area.

The OOB area, sometimes called the spare area, is a small section of the flash memory reserved for metadata. The metadata usually consists of **error correcting code** (**ECC**), information about bad blocks, and in some cases, information about the filesystem. This causes an issue for examiners because most mobile forensic tools do not account for the OOB area; they expect it to not be included in the image. When presenting the tool with an image containing the spare area, the tool frequently does not know what to do and fails to parse the data properly.

The reason that tools fail to account for the OOB area is that it is *not* included in `dd` images, which is what most tools use to create their images. The OOB area may be included when using the `nanddump` command, though depending on the binary used, there may be an option to exclude it. The OOB area is included with chip-off and JTAG images.

To properly load the image into forensic tools, the OOB area will need to be removed first. A general rule of thumb is that the OOB size is based on the page size of the device; for every 512 bytes of page size there will be 16 bytes of OOB area. For example, a device with 2048 byte page sizes would likely have 64 bytes of OOB area at the end of each page. However, this is completely up to the memory manufacturer. Before attempting to remove the OOB area, an examiner should find the datasheet for the specific memory chip to confirm the page and OOB area sizes. This can generally be done by finding the memory chip on the phone's circuit board and searching for the model number of the chip.

The following is some sample code for a Python script that will remove the OOB area from an image. Just as in the last chapter, we don't claim to be Python experts and we're sure there are better, more efficient ways to do this, but this one does work:

```
import sys
file_to_parse = open(sys.argv[1],'rb')
file_after_removal = open('file_out.bin','wb')
while file_to_parse:
  lines_out = file_to_parse.read(2048)
  if lines_out:
    file_after_removal.write(lines_out)
    file_to_parse.seek(64,1)
  if not lines_out:
    break
print 'Done'
file_to_parse.close()
file_after_removal.close()
```

This file, if named `OOB_Remover.py`, would be executed with the following command:

```
python OOB_Remover.py C:\Users\Android_Examiner\physicaldump.bin
```

The output file, with no OOB area, would be named `file_out.bin` in the directory where the script was executed. The original is not edited or modified in anyway.

Note that the code as it is written assumes a page size of 2048 and an OOB size of 64; these two numbers would have to be edited for the specific sizes of the memory chip the image was taken from. The output should then be able to be loaded into commercial mobile forensic tools.

Imaging and analyzing Android RAM

Pulling Android memory is not applicable in a very large number of cases due to the fact that it requires root access. Most public root processes involve rebooting the phone, which erases volatile RAM, meaning that by the time an examiner gains root to image the RAM, it's too late because the RAM has been erased. Because of this and possibly other reasons, there is not great support for Android RAM imaging and analysis in the commercial forensic world. However, there are cases where imaging RAM is applicable, and may prove invaluable. If a device is already rooted when it is seized, imaging the RAM should be a mandatory step in the seizure process. As powering the phone off will erase the RAM, the device should be placed in Airplane mode (any other network connections such as Wi-Fi and Bluetooth disabled), and the RAM should be imaged immediately to avoid the device battery dying before the RAM can be pulled.

The main challenge when it comes to RAM is the analysis. RAM is completely raw, unstructured data; there is no filesystem. When viewed in a hex editor, RAM appears to just be a giant blob of data with very little rhyme or reason to help examiners figure out what they are looking at. This difficulty is compounded by the fact that modern devices commonly have gigabytes worth of RAM. RAM can easily be searched for keywords using traditional forensic tools and methods, but that presumes an examiner knows exactly what they are looking for.

What can be found in RAM?

Any data that is written to the flash memory *must* pass through RAM, there is no other way for the processor to communicate with the flash memory. This means that almost anything done on the device may be found in the contents of a RAM dump. Depending on the amount of device usage, data may remain in RAM indefinitely, until it needs to be overwritten. RAM dumps frequently contain text typed on the device, including usernames and passwords, and application data that is not stored permanently on the device. For example, older versions of the Facebook application stored the contents of a user's News Feed in a database in its application folder; newer versions do not save the user's News Feed, but it exists in RAM.

Imaging RAM with LiME

The most common tool for Android RAM acquisition is the **Linux Memory Extractor** (**LiME**), previously known as DMD. LiME is free and open source, but isn't highly user-friendly as it requires the user to compile it from the source code, which can only be done on a Linux system. The compilation process must also be done for each specific version of Android for each device being examined, which somewhat limits its usability in the field. This is necessary because LiME is not a binary (as were the netcat and nanddump tools we used before); instead it is a kernel module that must be built specifically for each kernel it will be loaded into.

In order to ensure the proper kernel source code is downloaded, we will need to determine the model and software version for a device, which can be done by scrolling through the phone menu to **Setting | About Phone**. Alternatively, this information can be found in the ADB shell by running the following command:

```
cat /system/build.prop
```

The software version in the model should be in the first few lines at the top of the file.

Luckily, most Android manufacturers do release their kernel source code; a quick Google search can usually turn up source code for each model and software version. The following are the open source release sites for a few major manufacturers:

- **Samsung**: `http://opensource.samsung.com/reception.do`
- **Motorola**: `http://sourceforge.net/motorola/`
- **HTC**: `http://www.htcdev.com/devcenter`
- **Google Nexus**: `https://source.android.com/source/building-kernels.html`



The correct model and version source must be used. Using the wrong kernel source to compile LiME will, at the very least, not work on the device. Loading an incompatible kernel module could also crash the device.

To obtain the source code for LiME, navigate to `https://github.com/504ensicsLabs/LiME` and choose the **Download ZIP** option, then extract the zip.

There are many excellent resources online explaining how to compile LiME for a specific kernel, and also how to create a custom `Volatility` plugin to examine the resulting RAM dump, so we won't detail them here. A couple of them are listed here:

- `http://lime-forensics.googlecode.com/files/LiME_Documentation_1.1.pdf`
- `https://code.google.com/p/volatility/wiki/AndroidMemoryForensics`

One point missing from these sources is that the step that uses ADB `pull` to obtain the `/proc/config.gz` file may not work on all devices. If the file does not exist, the correct config file can be found in the source code, usually in the `/arch/arm/configs` folder. It is usually named after the processor model, for example, `apq8064_defconfig`.

Imaging RAM with mem

As described in the preceding section, using LiME is not for the faint-hearted; it is a very daunting process fraught with complications. It seems unlikely that an examiner in the field will download and compile kernel source code. The **mem** tool was developed by James Nuttall to address these issues. Rather than a kernel module that needs to be compiled on a device-specific basis, mem is a binary similar to the netcat and nanddump examples used previously in this chapter. The mem tool can be downloaded at `http://sourceforge.net/projects/androidforensics-mem/files/`.

Mem is an executable binary that needs to be pushed to the device and executed using the exact procedures detailed previously for netcat and nanddump. It may seem counter-intuitive to push something to RAM in order to read the RAM, but this is an accepted necessity in computer forensics. In our opinion, it is better to overwrite a small portion of RAM than to actually push it to the device and overwrite evidentiary user data.

Mem has the capability to read the entire RAM, or to target specific, forensically-interesting processes (applications). Assuming that mem is pushed to the same location on the device used for netcat above, the format for running mem is as follows:

```
/dev/Examiner_Folder/mem <PID>
```

`PID` is the ID of the process to read; if it is set to 0, all of RAM will be imaged. To view the list of processes within the ADB shell, use the following command:

```
ps
```

In the following screenshot, we can see that `PID` is the second column of the output:

```
root@android:/ # ps
ps
USER     PID   PPID  VSIZE  RSS     WCHAN    PC         NAME
root      1     0     568    428   c0149df4 00010594 S /init
root      2     0     0      0     c00a7ed4 00000000 S kthreadd
root      3     2     0      0     c0091408 00000000 S ksoftirqd/0
root      4     2     0      0     c00a3980 00000000 S kworker/0:0
root      5     2     0      0     c00a3980 00000000 S kworker/u:0
root      6     2     0      0     c00db948 00000000 S migration/0
root      16    2     0      0     c00a36f0 00000000 S khelper
```

The output can be quite large, though interesting processes can be found by simply reading through the list:

```
u0_a26    1712  215   731412 48296 ffffffff 402effb8 S com.google.android.gms.wearable
u0_a26    1769  215   880776 71676 ffffffff 402effb8 S com.google.android.gms
u0_a11    2132  215   696672 43124 ffffffff 402effb8 S com.google.android.calendar
u0_a85    2154  215   694340 46468 ffffffff 402effb8 S com.fsck.k9
u0_a136   2210  215   752060 57832 ffffffff 402effb8 S kik.android
u0_a66    2252  215   887568 175500 ffffffff 402effb8 S com.facebook.katana
```

In the above screenshot, we can see that Kik, Facebook, Calendar, and Gmail are all running. An alternative to reading the entire output is to search for known applications using `grep`. For example, to find Facebook in the output we could run the following command:

```
ps | grep facebook
```

The output of which would show only the entry for Facebook as follows:

```
root@android:/ # ps | grep facebook
ps | grep facebook
u0_a66    2252  215   875580 156156 ffffffff 402effb8 S com.facebook.katana
root@android:/ #
```

We can see that the PID of Facebook is `2252`. To avoid overwriting data on the device, mem is written to be used in conjunction with netcat just as shown in the *Writing directly to an examiner's computer with netcat* section of this chapter. So, capturing the RAM used by Facebook requires the following steps:

1. In a terminal on the examiner's computer, run:

   ```
   adb forward tcp:9999 tcp:9999
   ```

2. In a terminal window within ADB shell, run:

   ```
   /dev/Examiner_Folder/mem 2252 | /dev/Examiner_Folder/nc -l -p 9999
   ```

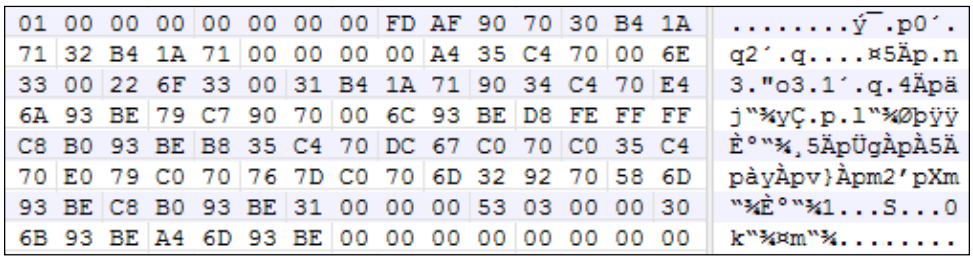
3. In the terminal window on the examiner's computer run:

   ```
   nc 127.0.0.1 9999 > FB_RAM.bin
   ```

4. When mem has finished running, there should be a file called `FB_RAM.bin` in the working directory of the examiner's computer.

Output from mem

As mentioned previously, there aren't many good ways to examine RAM because there is no filesystem; the output is just a blob of data. This is still true for data acquired with mem; the output from the Facebook RAM pulled above is a 550 MB unstructured blob of data. The following screenshot can be viewed as an example:



```
01 00 00 00 00 00 00 00 FD AF 90 70 30 B4 1A  ........ý¯.p0´.
71 32 B4 1A 71 00 00 00 00 A4 35 C4 70 00 6E  q2´.q....¤5Äp.n
33 00 22 6F 33 00 31 B4 1A 71 90 34 C4 70 E4  3."o3.1´.q.4Äpä
6A 93 BE 79 C7 90 70 00 6C 93 BE D8 FE FF FF  j"¾yÇ.p.l"¾Øþÿÿ
C8 B0 93 BE B8 35 C4 70 DC 67 C0 70 C0 35 C4  È°"¾¸5ÄpÜgÀpÀ5Ä
70 E0 79 C0 70 76 7D C0 70 6D 32 92 70 58 6D  pàyÀpv}Àpm2'pXm
93 BE C8 B0 93 BE 31 00 00 00 53 03 00 00 30  "¾È°"¾1...S...0
6B 93 BE A4 6D 93 BE 00 00 00 00 00 00 00 00  k"¾¤m"¾........
```

We suggest using strings or some other search function to narrow down the data to hopefully find useful user data. The file can also be loaded into computer forensic tools like EnCase or FTK in order to search for keywords, and carving tools can be used to locate images.

However, with enough patience and dedication, useful information can be found, such as this post from a user's news feed:

```
xMQ==","label":"Born"}]}}}ranges":[],"text":"My
 favorite kind of walk around the neighborhood
is the kind with a stop for frozen yogurt."},"a
```

Eventually, the mem developers hope to work on a volatility profile to help with analyzing the output.

Acquiring Android SD cards

As discussed above and in previous chapters, the SD card can refer to a physical, external SD card or a partition within the flash memory. A removable external SD card can be imaged separately from the device through a write-blocker with typical computer forensics tools, or using the `dd`/`nanddump` techniques shown in the previous section, although the former is usually faster as it does not need to write data over netcat.

Physically imaging an SD card is very similar to the physical imaging discussed above; in fact, if the SD card is symbolically linked to the `/data` partition, it would be acquired as part of the `/data` partition as seen in the Autopsy screenshots. The only difference in the process is that if the SD card is being imaged, the output file cannot be written to the SD card! This means using the netcat methods covered previously is the best option for physically imaging an internal SD card.

What can be found on an SD card?

By default, the SD card is typically used to store large files, including downloaded items and pictures taken with the device. Many applications will also create their own directory on the SD card for storing data such as images sent or received through chat applications. In some cases, as will be seen in *Chapter 8*, *Android Forensic Tools Overview*, there even are applications that will routinely perform a backup of all their data to the SD card. This is especially useful to forensic examiners because they may not be able to access the internal memory due to security settings or the inability to obtain root, but may be able to access the SD card.

Common SD card locations of interest include, but of course are not limited to, the following locations:

- `/DCIM`: This location includes pictures taken on the device
- `/Pictures/Screenshots`: This location contains screenshots taken on the device
- `/Download`: This location contains downloaded files
- `/Android/data`: This is the storage location for many applications
- `/AppName`: This is the storage location for many applications



The `/Android/data` and `/AppName` folders may persist even if the app has been deleted. Contents of the folders will be deleted, but the folders may remain; which is an indication that the application was previously installed on the device.

These are just common default locations. If a device is rooted, the user could place any data from the internal memory onto the SD card.

SD card security

In older versions of Android, simply plugging a phone into a computer would logically mount the SD card and allow an examiner access to its data. In some version of Android (possibly 3.0) this changed, although the exact version could not be found in the various change logs that we examined. Newer versions of Android will not automatically allow access to the SD card from a computer if a screen lock is in use, meaning the screen lock will have to be bypassed in order to gain access to the SD card. The obvious exception to this is a physical, external SD card can still be removed and analyzed with traditional computer forensic methods.

SD cards can also be encrypted, either through the device full-disk encryption if it is an internal SD card, or through third-party applications if it is external. In some cases, activating the full-disk encryption will leave the SD card unencrypted, though this varies depending upon the device manufacturer.



The full-disk encryption in Android Lollipop also encrypts the SD card.

Advanced forensic methods

In addition to the methods discussed in the previous chapters, there are also more advanced, specialized methods available. JTAG and chip-off methods are both highly useful tools in many common situations, but require advanced training (and a lot of practice before working on live evidence!). The final advanced method, a cold boot attack to recover encryption keys, is far more theoretical.

JTAG

JTAG is a standard developed by the **Institute of Electrical and Electronics Engineers** (**IEEE**). During the device production process, it is used to communicate with the processor through a specialized interface for testing purposes. Luckily for forensic examiners, it also allows them to communicate directly with the processor and retrieve a full physical image of the flash memory.

To perform a JTAG extraction, the device must be taken apart down to the circuit board. The circuit board will contain multiple taps (physical contacts on the device circuit board), though they are commonly unlabelled and there are usually far more taps than required for JTAG. To determine the correct taps, an examiner would have to either find a pin-out online (or included with their tool of choice), or use electronic test equipment to determine what each tap is. The examiner will then have to solder a wire to each tap, or use adapters (sometimes called jigs) that are commercially available, and connect to their JTAG box through a provided adapter as shown in the following image:



HTC Evo before and after being hooked up for JTAG (courtesy of http://lowcostwin4n6.blogspot.com/)

JTAG may sound complicated (perhaps it is), but it serves many useful purposes and two advantages are listed here:

1. It does not require the device to be powered on and so:
    - Can be successful even if the device is damaged
    - There are no RF-shielding concerns
    - Does not require root, ADB, or USB debugging
2. It can be used to recover device PINs/passwords and so:
    - Can image the entire flash memory and recover/crack password files as shown in Chapter 4

Many manufacturers make JTAG tools; many of the common ones used for mobile forensics can be found at `http://teeltech.com/mobile-device-forensic-tools/jtag-equipment/`. The RIFF box listed on the site is probably the most frequently used for mobile forensics, as it comes with support (including pin-outs) for a wide variety of devices.

JTAG is not always successful, or even possible. Though the interface is almost always on the circuit board, the manufacturer can choose to disable it after the device is manufactured.

Chip-off

Chip-off involves heating the device's circuit board until the solder holding the components to the board melts, and then removing the flash memory chip. The memory chip can then be read using commercial tools, resulting in a full physical image. Chip-off techniques, like JTAG, stem from the commercial electronic production process. The process of melting the solder (commonly called reflow or rework) is used to place and remove components from a circuit board, and the readers used to acquire the memory are used to both read and write to memory chips, often in bulk quantities.



A memory chip being removed from a damaged phone (courtesy of www.binaryintel.com/services/jtag-chip-off-forensics/chip-off_forensics/)

Chip-off has the same benefits as JTAG: it does not require the device to power on, and can be used to acquire the PIN/password from a locked device; though acquiring the PIN/password is generally a moot point, chip-off is normally considered to be a destructive process. While the memory chip can be replaced on the device, it is a technically demanding process and requires further training. But, as a last resort, chip-off is an excellent alternative for devices that would otherwise be impossible to examine.

Chip-off is significantly more expensive than JTAG, as a specialized rework station and commercial memory reader is required. There are dozens of rework stations available and they all provide essentially the same functionality. There is also a wide range of memory readers, though we have had great success with this reasonably priced model at `http://www.dataman.com/programmers/universal/dataman-48pro2-super-fast-universal-isp-programmer.html`. A rework station and reader aren't the only costs associated with chip-off; most readers will also require a specific adapter for each model of chip to be read.

Bypassing Android full-disk encryption

Before going any further, please note that this section is *highly* impractical. However, we present it here in the hopes that some aspiring forensic developer will see it and decide it is a worthy enough cause to make it more widely applicable (and also because it's really neat).

Cold boot attacks have been demonstrated and used many times, but until recently could not be used effectively against Android devices. A cold boot attack is based on the idea that RAM is less volatile at lower temperatures (the data remains longer), so freezing a device can allow an examiner to access RAM and find the key needed to decrypt the device. This was recently demonstrated successfully by a team of researchers whose paper and research can be found here: `https://www1.informatik.uni-erlangen.de/frost`.

Again, note that these techniques were only validated against one device (the Galaxy Nexus), and their tool is a loadable kernel module, much like LiME, and relies on a custom recovery image that would have to be created for each device it is used against.

Summary

This chapter discussed several techniques used for physically imaging internal memory or SD cards and some of the common problems associated with them:

Technique	Problems associated
dd	• Usually pre-installed on device • May not work on MTD blocks • Does not obtain Out-of-Band area
nanddump	• Not commonly found on the device, must be pushed to device • Works well with MTD blocks • May obtain Out-of-Band area, based on options in the binary used

Additionally, each imaging technique can be used to either save the image on the device (typically on the SD card), or used with netcat to write the file to the examiner's computer:

Technique	Features
Writing to SD card	• Easy, doesn't require additional binaries to be pushed to the device • Familiar to most examiners • Cannot be used if SD card is symbolically linked to the partition being imaged • Cannot be used if the entire memory is being imaged
Using netcat	• Usually requires yet another binary to be pushed to the device • Somewhat complicated, must follow steps exactly • Works no matter what is being imaged • May be more time consuming than writing to the SD

Multiple tools used for RAM imaging were also demonstrated:

Tool	**Features**
LiME	• Must be compiled for each device being examined • Very complicated process • Known, well-documented procedures for analysis • Output is a dump of all RAM
Mem	• Can be used on any device with no additional steps • New tool, not as widely used and documented • Output is one file for each process running on the device

Finally, we briefly discussed chip-off and JTAG techniques on an introductory level.

The next chapter will demonstrate the recovery of deleted data from physical images like the ones created in this chapter.

6
Recovering Deleted Data from an Android Device

The extraction and acquisition techniques that you have learned so far will help you access various details such as call logs, messages, and so on. However, these techniques do not help us see the data that is deleted from the device. In this chapter, you will learn about data-recovery techniques that will enable you to view the data that is deleted on the device. Deleted data could contain highly sensitive information, and thus, data recovery is a crucial aspect of mobile forensics. In this chapter, we will cover the following topics:

- An overview of data recovery
- Recovering data deleted from an SD card
- Recovering data deleted from a phone's internal storage

An overview of data recovery

Data recovery is a powerful concept within digital forensics. It is the process of retrieving deleted data from a device or SD card when it cannot be accessed normally. Being able to recover data that is deleted by a user could help solve several civil and criminal cases. This is because most of the accused just delete the details on the device, hoping that the evidence will be destroyed. Thus, in most of the criminal cases, deleted data can be crucial, because it may contain information the user wanted to erase from the Android device. For example, consider the scenario where a mobile phone has been seized from a terrorist.

Wouldn't it be of the greatest importance to know which items have been deleted by them? Access to any deleted SMS messages, pictures, dialed numbers, and so on can be of critical importance, as they may reveal a lot of sensitive information. From a normal user's point of view, recovering data that has been deleted would usually refer to the operating system's built-in solutions, such as the Recycle Bin in Windows. While it's true that data can be recovered from these locations, due to an increase in user awareness, these options don't often work. For instance, on a desktop computer, people now use *Shift* + *Delete* whenever they want to delete a file completely from their desktop. Similarly, within mobile environments, users are aware of restore operations provided by apps and so on. In spite of this, data recovery techniques allow a forensic investigator to access the data that is deleted from the device.

With respect to Android, it is possible to recover most of the deleted data, including SMS, pictures, application data, and so on. However, it is important to seize the device in a proper manner and follow certain procedures, without which the data might be deleted permanently. To ensure that the deleted data is not lost forever, it is recommended that you keep the following points in mind:

- Do not use the phone for any activity after seizing it. The deleted data exists on the device until the space is needed by some other incoming data. Hence, the phone must *not* be used for any sort of activity so that the data is not overwritten.
- Even when the phone is not used, without any intervention from our end, data can be overwritten. For instance, an incoming SMS would automatically occupy the space that overwrites the deleted data. Also, remote wipe commands can wipe the content present on the device. To prevent the occurrence of such events, you can consider the option of placing the device in Faraday bags, as explained in *Chapter 1*, *Introducing Android Forensics*. Thus, care should be taken to prevent delivery of any new messages or data through any means of communication.

How can deleted files be recovered?

When a user deletes any data from the device, the data is not actually erased and continues to exist on the device. What gets deleted is the pointer to this data. All filesystems contain metadata that maintains information about the hierarchy of files, file names, and so on. Deletion does not actually erase the data, but instead, it removes the filesystem metadata. Just deleting the metadata increases the performance of operating systems; deleting the pointer and marking the space as available is an extremely fast operation compared to actually erasing all the data. Thus, when text messages or any other files are deleted, they are just made invisible to the user. However, the files are still present on the device as long as they are not overwritten by some other data.

Hence, it is possible to recover them before new data comes in and occupies the space.

Recovering deleted data on Android involves two scenarios:

- Recovering data that is deleted from an SD card, such as pictures, videos, and so on
- Recovering data that is deleted from a device's internal storage, such as SMS, dialed numbers, browsing history, application data, chat logs, and so on

The following sections cover the techniques that can be used to recover deleted data from both the SD card and internal storage of an Android device.

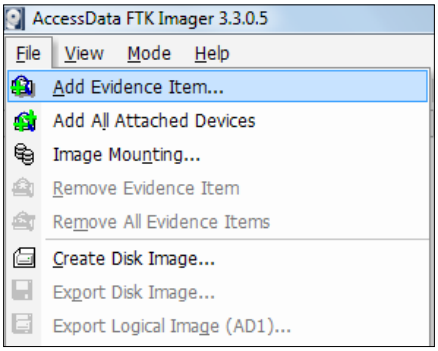
Recovering data deleted from an SD card

Data present on an SD card can reveal a lot of information that is useful during a forensic investigation. The fact that pictures, videos, voice recordings, application data, and so on are stored on the SD card adds weight to this. As mentioned in the previous chapters, Android devices often use the FAT32 filesystem on the SD card. The main reason for this is that the FAT32 filesystem is widely supported in most operating systems, including Windows, Linux, and Mac OS X. The maximum file size on a FAT32-formatted drive is around 4 GB. With increasingly high resolution formats that are now available, this limit is commonly reached. Recovering the data deleted from an external SD can be pretty easy if it can be mounted as a drive.

If the SD card is removable, it can be mounted as a drive by connecting it to a computer using a card reader. Any files can be transferred to the SD card while it's mounted. Some of the older devices that use USB mass storage also mount the device as a drive when connected through a USB cable. As explained earlier, in forensics, in order to make sure that the original evidence is not modified, a physical image of the disk is taken, and all further experimentation is done on the image itself. Similarly, in the case of SD card analysis, an image of the SD card needs to be taken. The process of imaging is similar to the one explained in *Chapter 5*, *Extracting Data Physically from Android Devices*. Once the imaging is done, we get a `dd` image file. In our example, we will use a free tool, FTK Imager. This tool is an imaging utility, and in addition to creating disk images, it can also be used to explore the contents of a disk image.

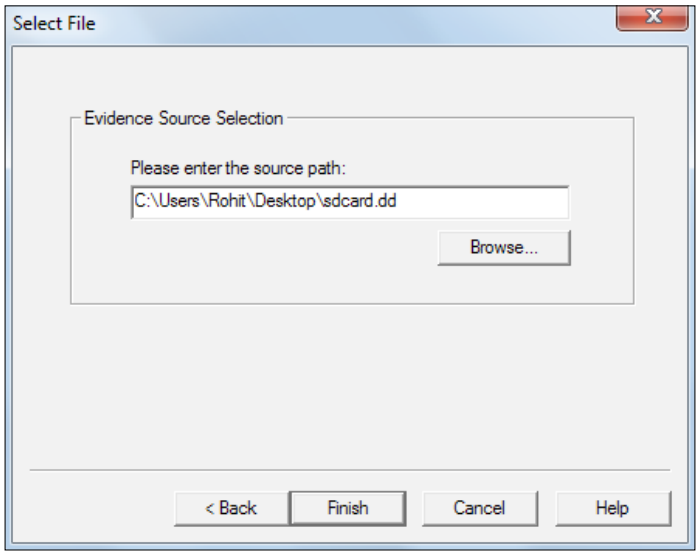
The following steps are required to recover the contents of an SD card using FTK Imager:

1. Start FTK Imager. Click on **File** and then on **Add Evidence Item** in the menu.



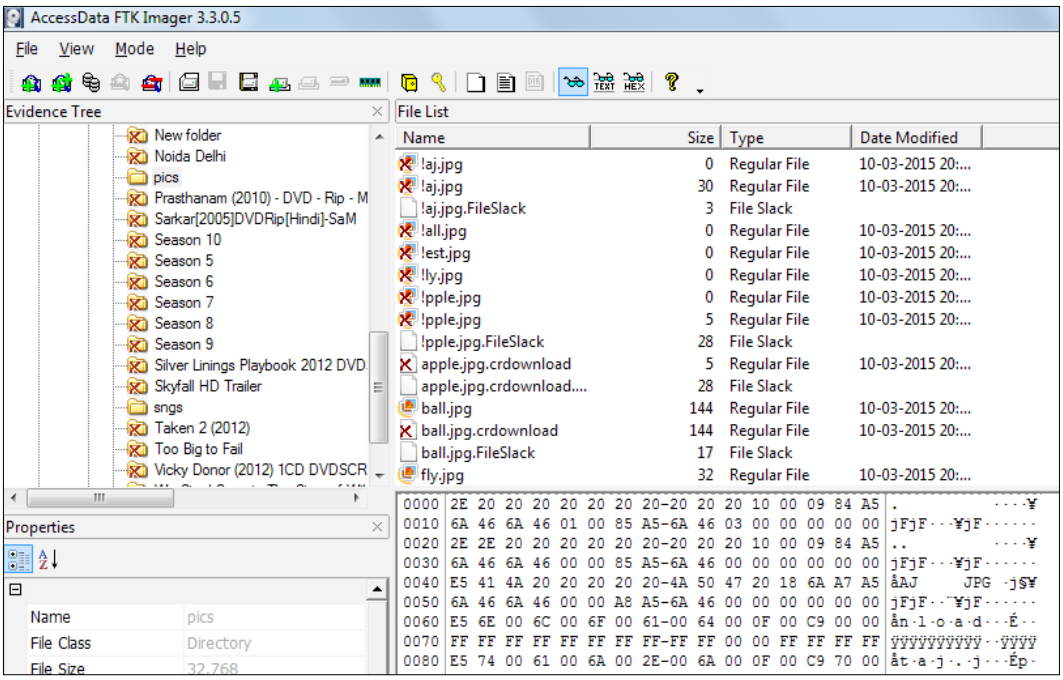
Adding evidence in FTK Imager

2. Select **Image File** in the **Select Source** dialog and click on **Next**.
3. In the **Select File** dialog, browse to the location where you downloaded the `sdcard.dd` file. Select the file and click on **Finish**:



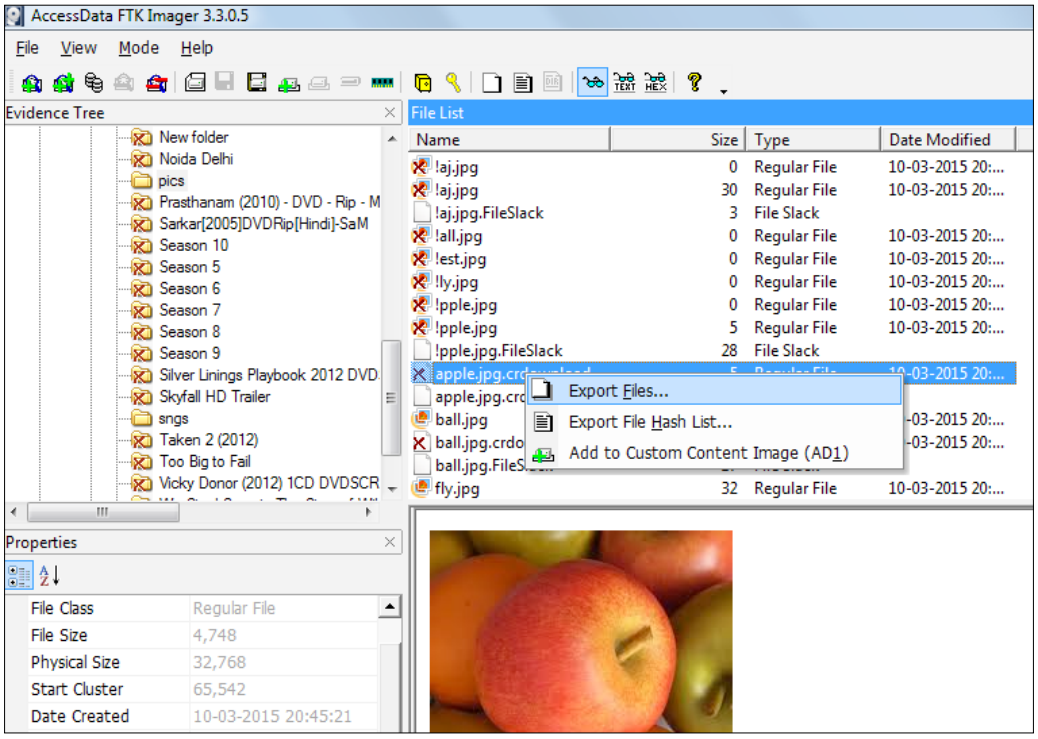
Selecting the image file for analysis in FTK Imager

4. FTK Imager's default display will appear with the contents of the SD card visible in the **View** pane in the lower-right corner. You can also click on the **Properties** tab below the lower left pane to view the properties for the disk image.
5. In the left pane, the drive has opened. You can open folders by clicking on the **+** sign. When you highlight a folder, its contents are shown on the right pane. When a file is selected, its contents can be seen in the bottom pane.
6. As shown in the following screenshot, the deleted files will have a red cross over the icons derived from their file extensions:



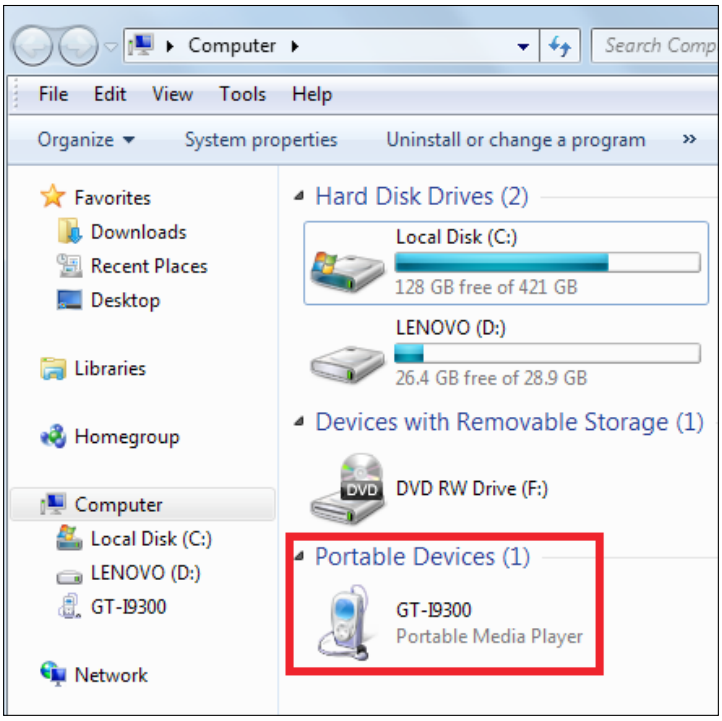
Deleted files shown with red cross over the icons

7. To export the file, right-click on the file that contains the picture and select **Export Files**:



Exporting the deleted files in FTK Imager

Sometimes, only a fragment of the file is recoverable. This file fragment cannot be read or viewed directly. In this case, we need to look through unallocated spaces for more data. Carving can be used to recover from unallocated spaces. Winhex, Scalpel, Foremost, and Photorec are some of the tools that can help you do this. Most of the latest Android devices do not mount as a mass storage. This is because these devices do not support USB mass storage and, instead, use the MTP or PTP protocol. With USB mass storage, the computer would need exclusive access to the storage. In other words, the external storage needs to be completely disconnected from the Android OS when it is connected to a workstation. This leads to several other complications with respect to mobile apps. When an Android device uses MTP, it appears to the computer as a media device and not as removable storage, as shown in the following screenshot:



An Android device shown under Portable Devices in Windows

However, the normal data-recovery tools would need a mount drive in order to perform a scan, though this is not a recommended procedure as it might result in changes being made to the device. Hence, most of the latest devices that use MTP/PTP are not treated as mount drives. So, the traditional data-recovery tools that work for computers do not work on these devices.

For the reasons mentioned earlier, when the device uses MTP/PTP and is not mounted as a drive, the recovery can be done by certain Android-specific data-recovery tools that need the USB debugging option to be turned on. Almost all the Android data-recovery tools on the market need you to enable USB debugging so that your device and the SD card can be recognized before starting Android data recovery. Also, these tools work only on rooted devices. We will now look at recovering data deleted from the internal memory of an Android device.

Recovering data deleted from internal memory

Recovering files deleted from Android's internal memory, such as app data and so on, is not supported by most analytical tools. This is for two main reasons. First, unlike the common filesystems used in SD cards, the filesystems used by internal memory may not be recognized and mounted by forensic tools. Second, the examiner cannot get access to the raw partitions of the internal memory of an Android phone, unless the phone is rooted. The following are some of the other issues the examiner may face when attempting to recover data from the internal memory on Android devices:

- To get access to the internal memory, you can try to root the phone. However, the rooting process might involve writing some data to the `/data` partition. This process could overwrite valuable data on the device.
- Unlike SD cards, the internal filesystem here is not FAT32 (which is widely supported by forensic tools). The internal filesystem could be YAFFS2 (in older devices), EXT3, EXT4, RFS, or something proprietary built to run on Android. Therefore, many of the recovery tools designed for use with Windows filesystems won't work.
- Application data on Android devices is commonly stored in the SQLite format. While most forensic tools provide access to the database files, they may have to be exported and viewed in a native browser. The examiner must examine the raw data to ensure that the deleted data is not overlooked by the forensic tool.

For these reasons, recovering data deleted from the internal memory of an Android device is difficult, but not impossible. The internal memory of Android devices holds the bulk of the user data and the possible keys to your investigation. As mentioned earlier, the device must be rooted in order to access the raw partitions. It's important to note that most of the Android-recovery tools on the market do not highlight the fact that they work only on rooted phones. Hardware-based solutions, such as UFED and XRY, are fully capable of recovering deleted data from internal memory as well as SD cards. Let's now see how we can recover deleted data from an Android phone.

Recovering deleted data by parsing SQLite files

Most of the application data in Android is stored in SQLite files. Data related to text messages, e-mails, and most app data is stored in SQLite files. SQLite databases can store deleted data within the database itself. Files marked for deletion by the user no longer appear in the active SQLite database files. Therefore, it is possible to recover the deleted data, such as text messages, contacts, and more, by analyzing these SQLite files. There are two areas within a SQLite page that can contain deleted data: unallocated blocks and free blocks. Most of the commercial tools that recover deleted data scan the unallocated blocks and free blocks of the SQLite pages. Parsing the deleted data can be done using the available forensic tools such as Oxygen Forensics SQLite Viewer. The trial version of the SQLite Viewer can be used for this purpose. However, there are certain limitations on the amount of data that you can recover.

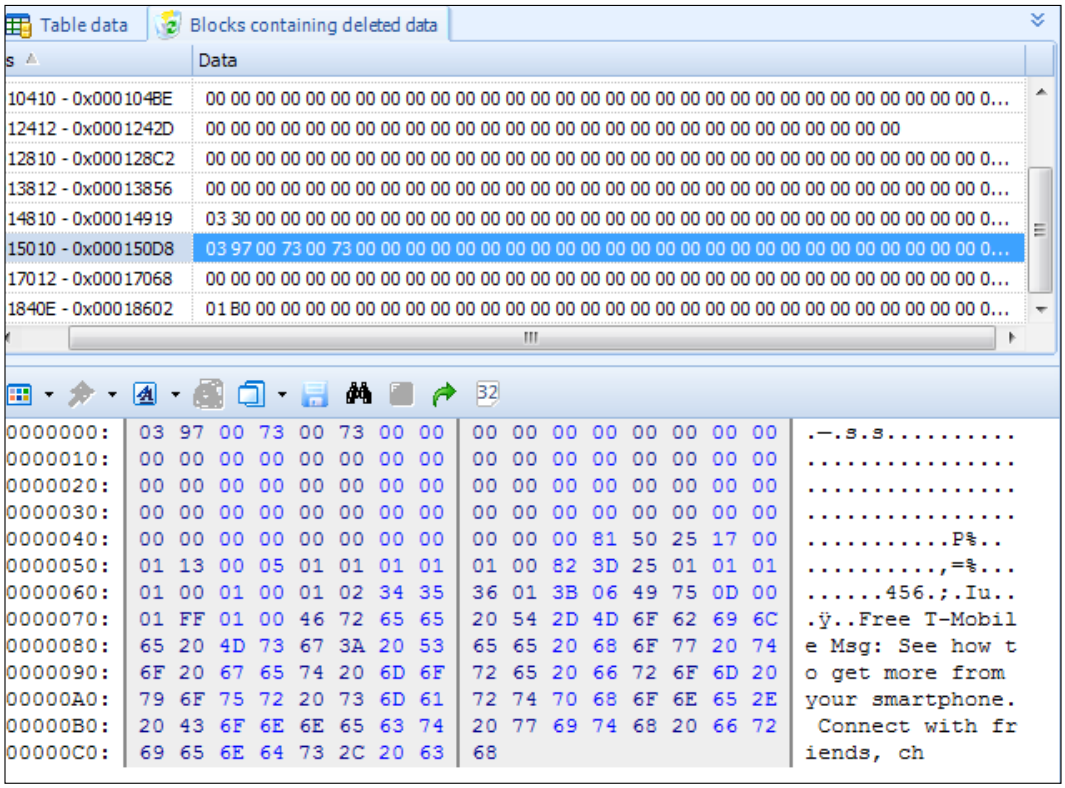
For our example, we will recover the deleted SMS messages from an Android device. Recovering deleted SMS messages from an Android phone is quite often requested as part of forensic analysis on a device, mainly because it's the most popular form of communication. There are different ways to recover deleted text messages on an Android device. However, with respect to recovery through parsing SQLite files, we need to understand where the messages are being stored on the device. In *Chapter 4*, *Extracting Data Logically from Android Devices*, we explained the important locations on the Android device where user data is stored. Here is a quick recap:

- Every application stores its data in the `/data/data` folder (again, this requires root access to acquire data)
- The files at `/data/data/com.android.providers.telephony/databases` contain details about SMS/MMS

Deleted text messages can be recovered by examining an SQLite database file named mmssms.db stored in /data/data/com.android.providers.telephony/databases. Here are the steps required:

- Extract the SMS database SQLite file (`mmssms.db`). This can be done using logical data-extraction techniques, which are covered under *Chapter 4*, *Extracting Data Logically from Android Devices*.

- Once the files are extracted to the local machine, you can use available tools such as Cellebrite. These tools may extract the deleted details automatically. You can also manually check for fragments using a Hex viewer or a SQLite Viewer, such as Oxygen Forensics SQLite Viewer.
- One way to view the deleted data is by clicking on the **Blocks containing deleted data** tab in the Oxygen Forensics SQLite Viewer, as shown in the following screenshot (referenced from `http://az4n6.blogspot.in/2013/02/finding-and-reverse-engineering-deleted_1865.html`):



Deleted data shown in Oxygen Forensics SQLite Viewer

You can also try the available open source Python scripts (`http://az4n6.blogspot.in/2013/11/python-parser-to-recover-deleted-sqlite.html`) that parse the SQLite files for deleted records.

Recovering deleted data through file carving techniques

File carving is an extremely useful method in forensics, because it allows hidden or deleted data to be recovered for analysis. In simple terms, file carving is the process of reassembling files from fragments in the absence of filesystem metadata. In file carving, specified file types are searched for and extracted across the binary data to create a forensic image of a partition or an entire disk. File carving recovers files from the unallocated space in a drive based merely on file structure and content, without any matching filesystem metadata.



Unallocated space refers to the part of the drive for which there are no longer any pointers in file system structures such as file tables .

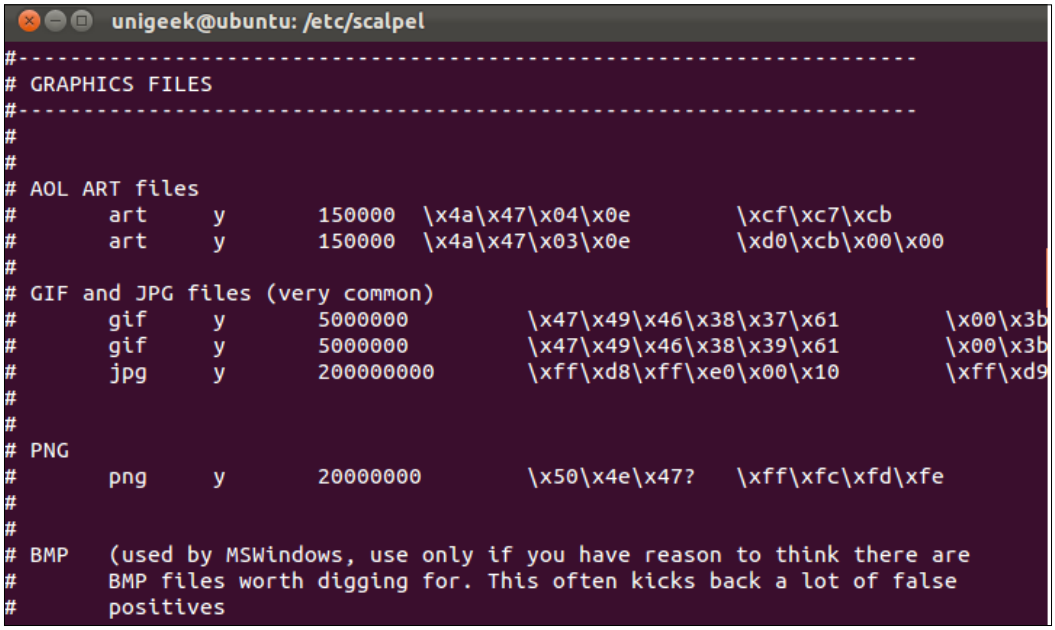
Files can be recovered or reconstructed by scanning the raw bytes of the disk and reassembling them. This can be done by examining the header (the first few bytes) and footer (the last few bytes) of a file.

File-carving methods are categorized based on the underlying technique in use. The header-footer carving method relies on recovering the files based on the header and footer information. For instance, the JPEG files start with 0xffd8 and ends with 0xffd9. The locations of the header and footer are identified, and everything between these two endpoints is carved. Similarly, the file structure carving method is based on the internal layout of a file to reconstruct the file. However, the traditional file-carving techniques, such as the ones we've already explained, may not work if the data is fragmented. To overcome this, new techniques, such as smart carving, use the fragmentation characteristics of several popular filesystems to recover the data.

Once the phone is imaged, it can be analyzed using tools such as **Scalpel**. Scalpel is a powerful open source utility to carve files. This tool analyzes the block database storage and identifies the deleted files and recovers them. Scalpel is filesystem independent and is known to work on various filesystems including, FAT, NTFS, EXT2, EXT3, HFS, and so on. The following steps explain how to recover files using Scalpel on an Ubuntu workstation:

1. Install Scalpel on the Ubuntu workstation using the `sudo apt-get install scalpel` command.

2. The `scalpel.conf` file present in the `/etc/scalpel` directory contains information about the supported file types, as shown in the following screenshot:



Scalpel configuration file

This file needs to be modified in order to include the files that are related to Android. A sample `scalpel.conf` file can be downloaded from `https://asecuritysite.com/scalpel.conf.txt`. You can also uncomment the files and save the `conf` file to select the file types of your choice. Once this is done, replace the original `conf` file with the one that is downloaded.

Scalpel needs to be run along with the preceding configuration file on the `dd` image being examined. You can run the tool using the command shown in the following screenshot, by inputting the configuration file and the `dd` file. Once the command is run, the tool starts to carve the files and build them accordingly.

```
File  Edit  View  Search  Terminal  Help
unigeek@ubuntu:~$ scalpel -c /home/unigeek/Desktop/scalpel-android.conf /home/un
igeek/Desktop/userdata.dd -o /home/unigeek/Desktop/rohit
Scalpel version 1.60
Written by Golden G. Richard III, based on Foremost 0.69.

Opening target "/home/unigeek/Desktop/userdata.dd"

Image file pass 1/2.
/home/unigeek/Desktop/userdata.dd: 100.0% |*************|    3.9 MB    00:00 ETA
Allocating work queues...
Work queues allocation complete. Building carve lists...
Carve lists built.  Workload:
gif with header "\x47\x49\x46\x38\x37\x61" and footer "\x00\x3b" --> 0 files
gif with header "\x47\x49\x46\x38\x39\x61" and footer "\x00\x3b" --> 2 files
jpg with header "\xff\xd8\xff\xe0\x00\x10" and footer "\xff\xd9" --> 71 files
jpg with header "\xff\xd8\xff\xe1" and footer "\x7f\xff\xd9" --> 1 files
png with header "\x50\x4e\x47\x3f" and footer "\xff\xfc\xfd\xfe" --> 0 files
png with header "\x89\x50\x4e\x47" and footer "" --> 71 files
sqlitedb with header "\x53\x51\x4c\x69\x74\x65\x20\x66\x6f\x72\x6d\x61\x74" and
footer "" --> 0 files
email with header "\x46\x72\x6f\x6d\x3a" and footer "" --> 0 files
doc with header "\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1\x00\x00" and footer "\xd0\xcf\
x11\xe0\xa1\xb1\x1a\xe1\x00\x00" --> 0 files
doc with header "\xd0\xcf\x11\xe0\xa1\xb1" and footer "" --> 0 files
htm with header "\x3c\x68\x74\x6d\x6c" and footer "\x3c\x2f\x68\x74\x6d\x6c\x3e"
 --> 1 files
pdf with header "\x25\x50\x44\x46" and footer "\x25\x45\x4f\x46\x0d" --> 0 files
pdf with header "\x25\x50\x44\x46" and footer "\x25\x45\x4f\x46\x0a" --> 0 files
wav with header "\x52\x49\x46\x46\x3f\x3f\x3f\x3f\x57\x41\x56\x45" and footer ""
 --> 0 files
amr with header "\x23\x21\x41\x4d\x52" and footer "" --> 0 files
```

Running the Scalpel tool on the image file

The output folder specified in the preceding command now contains a list of folders based on the file types. Each of these file types contains data based on the folder name. For instance, `jpg 2-0` contains the recovered files related to the `.jpg` extension:



Output of the Scalpel tool

As shown in the preceding screenshot, each folder contains recovered data from the Android device, such as images, `.pdf` files, `.zip` files, and so on. While some pictures are recovered completely, some are not recovered to a full extent, as shown in the following screenshot:

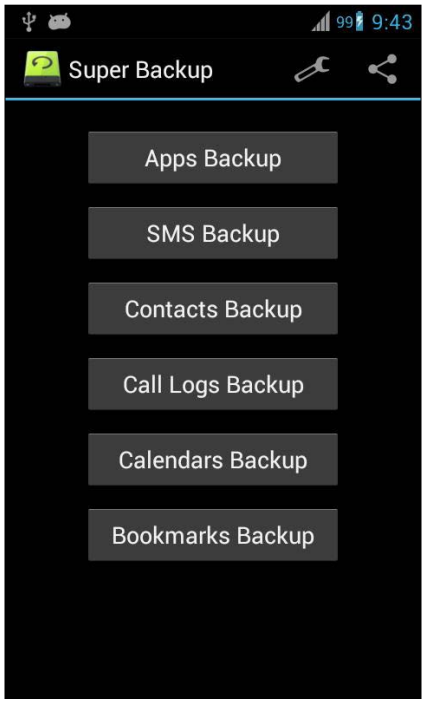


Data recovered using the Scalpel tool

Applications such as DiskDigger can be installed on Android devices to recover different types of files from both the internal memory and SD cards. Applications such as DiskDigger include support for `.jpg` files, `.mp3` and `.wav` audio, `.mp4` and `.3gp` video, raw camera formats, Microsoft Office files (`.doc`, `.xls`, and `.ppt`), and so on. However, as mentioned earlier, the application requires root privileges on the Android device in order to recover the content from the internal memory. Thus, file-carving techniques play a very important role in recovering important deleted files from the device's internal memory.

Analyzing backups

It is also recommended that you check whether the device has any backup applications or files installed. The initial release of Android did not include a mechanism for the users to back up their personal data. Hence, several backup applications have been used extensively by the users. Using these apps, users have the ability to back up their data either to the SD card or to the cloud. For example, the `Super Backup` app contains the options to back up call logs, contacts, SMS, and so on, as shown in the following screenshot:



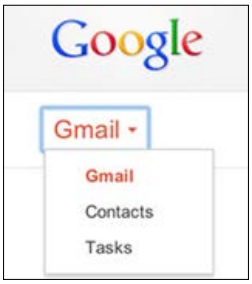
The Super Backup Android app

On detection of a backup application, the forensic examiners must attempt to determine where the data is stored. The data saved in a backup may contain important information, so looking for any third-party backup app on the device could be very helpful.

You can also restore the contacts on the device using the **Restore Contacts** option through the Google account configured on the device. This would work if the user of the device has previously synced their contacts using the **Sync Settings** option available in Android. This option synchronizes the contacts and other details and stores them in the cloud. A forensic examiner with legal authority or proper consent can restore the deleted contacts if they can get access to the Google account configured on the device. Once the account is accessed, perform the following steps to restore the data:

1. Log in to the Gmail account.

2. Click on **Gmail** in the top-left corner and select **Contacts**, as shown in the following screenshot:



The Contacts menu in Gmail

3. Click on **More**, which is present above the contacts list.
4. Click on **Restore Contacts**, and the following screen appears:



The Restore Contacts dialog box

5. Using this technique, you can restore the contact list to the state that it was in at any point within the past 30 days.

Summary

Data recovery is the process of retrieving deleted data from the device. Thus, it is a very important concept in forensics. In this chapter, we saw various techniques to recover deleted data from both an SD card and internal memory. While recovering the data from a removable SD card is easy, recovering data from internal memory involves a few complications. SQLite file-parsing and file-carving techniques aid a forensic analyst in recovering the deleted items that are present in the internal memory of the Android device. Checking for any installed backup apps on the device is recommended as it saves both time and effort.

In the next chapter, we will try to understand the forensic perspective on analysis of Android apps.

7
Forensic Analysis of Android Applications

This chapter will cover application analysis, using free and open source tools. It will focus on analyzing the data that would be recovered using any of the logical or physical techniques detailed in *Chapters 4* and *Chapter 5*. It will also rely heavily on the storage methods discussed in *Chapter 2*. We will see numerous SQLite databases, XML files, and other file types from various locations within the file hierarchy described in the second chapter. By the end of this chapter, you should be familiar with the following topics:

- An overview of application analysis:
  - Contacts/Calls/SMS
  - Wi-Fi
  - User dictionary
- Third-party applications and various methods used by popular applications to store and obfuscate data listed as follows:
  - Plain text
  - Epoch time
  - WebKit time
  - Misnaming file extensions
  - Julian dates
  - Base64 encoding
  - Encryption
  - Basic steganography
  - SQLCipher
- Basic application reverse engineering

Application analysis

Forensically analyzing an application is as much of an art as it is a science. There are myriad ways an application can store or obfuscate its data. Different versions of the same application may even store the same data differently. A developer is really only limited by their imagination (and Android platform restrictions) when it comes to choosing how to store their data. As a result of these factors, application analysis is a constantly shifting target. The methods that an examiner uses one day may be completely irrelevant the next.

The end goal of forensically analyzing an application is consistently the same, to understand what the app was used for and find user data.

In this chapter, we will look at the current version of many common applications. As apps can, and do, change how they store data through updates, nothing in this chapter is a definitive guide for how to analyze that application. Instead, we will look at a broad range of applications to show a variety of different methods used by applications to store their data. For the most part, we will look at very common applications (millions of downloads from Google Play), except for cases where looking at an obscure app can reveal interesting new ways of storing data.



While we made every attempt to be thorough in our usage of each application when populating our test data, it is entirely possible that not every feature of every application was used. The apps analyzed in the following sections are examples of how to examine data from that application, but may not include every possible bit of data that may be recovered.

All of our testing used the default settings of each application, as if the application was downloaded and immediately used. Different settings may affect the data that is stored and the location of the data on the device.

Also, this analysis was done on a Nexus 5 running Android 5.0.1. Certain manufacturers, such as HTC and Samsung, may provide applications that duplicate functionality from these apps (such as a home screen widget that accesses Facebook). These apps may store data in different locations. Some files we analyze may not be present on other versions.

Why do app analysis?

For starters, even standard phone functions, such as contacts, calls, and SMS, are done through applications on Android devices. So, even acquiring basic data requires us to analyze an application. Second, a person's app usage can tell you a lot about them: where they've been (and when they were there), who they've communicated with, and even what they may be planning in the future.

Many phones can come out of the box with more than 20 preinstalled applications. A Yahoo study in 2014 revealed that users have, on average, 95 apps installed on their device. A Nielsen study showed that the average user uses 26 apps per month. An examiner has no real way of knowing which of these apps could contain information useful for an investigation, and therefore, all of them must be analyzed. An examiner may be tempted to skip over certain apps that would appear to have little useful data, such as games. This would be a bad idea, though. Many popular games, such as *Words with Friends* or *Clash of Clans*, have a built-in chat feature that could yield useful information. The following analysis will focus heavily on messaging applications, as our experience shows that these tend to be the most valuable in forensic analysis.

The layout of this chapter

For each application we examine, we will provide a package name, version number if possible, and files of interest. For example:

Package name: `com.android.providers.contacts`

Version: Default version with Android 5.0.1 (not listed within app)

Files of interest:

- `/files/`
    - `photos/`

All apps store their data in the `/data/data` directory by default. Apps could also use the SD card if they ask for this permission when the app is installed. The package name is the name of the directory for the application in the `/data/data` directory. Files of interest are from the root of the package name (that is, `/data/data/com.android.providers.contacts/files/photos` for the preceding example). Paths to data on the SD card are shown beginning with `/sdcard` (that is, `/sdcard/com.facebook.orca`). Do not expect to find data paths beginning with `/sdcard` in the `/data/data` directory of the application!

We will begin by looking at some of Google's applications, because these are preinstalled on the vast majority of devices (though they do not have to be). Then, we will look at third-party applications that can be found on Google Play.

Determining what apps are installed

To see what applications are on the device, an examiner could navigate to `/data/data` and run the `ls` command. However, this doesn't provide well-formatted data that will look good in a forensic report. We suggest that you pull the `/data/system/packages.list` file. This file lists the package name for every app on the device and path to its data (if this file does not exist on the device, the `adb shell pm list packages -f` command is a good alternative). For example, here is an entry for Google Chrome (the full file on our test device contained 120 entries):

```
com.android.chrome 10034 0 /data/data/com.android.chrome default 3003,1028,1015
```



This is the first method of data storage: plain text. Often, we will see apps store data in plain text, including data you wouldn't expect (such as passwords).

Perhaps of greater interest is the `/data/system/package-usage.list` file, which shows the last time a package (or application) was used. It's not perfect; the times shown in the file did not correlate exactly with the last time we used the app. It appears that the app updating or receiving notifications (even if the user does not view them) may affect the time. However, it is good for a general indication of the last apps the user accessed:

```
com.android.chrome 1422206858650
```

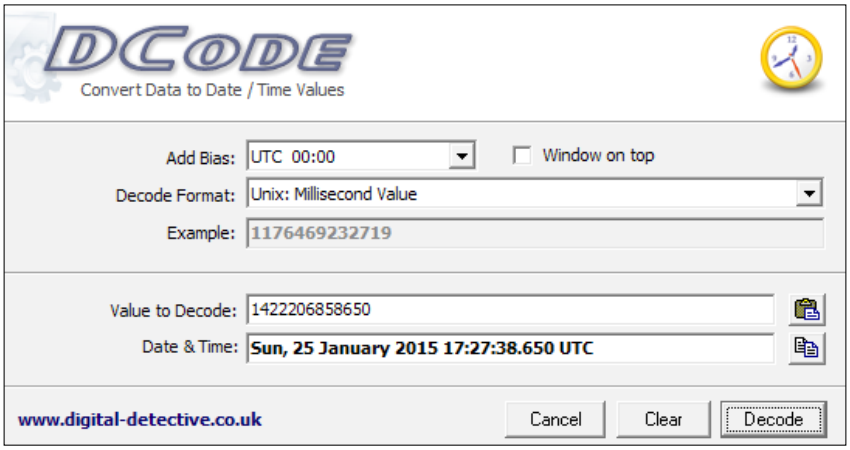
If you're wondering where the time is in the preceding line, it's in a format known as Linux epoch time.

Understanding Linux epoch time

Linux epoch time, also known as Unix time or Posix time, is stored as the number of seconds (or milliseconds) since midnight on 1 January, 1970, UTC. A 10-digit value indicates it is in seconds, while a 13-digit value is indicative of a millisecond value (at least for times likely to be found on a smartphone, as 9-digit second and 12-digit millisecond values haven't occurred since 2001).

In the preceding example, the value is `1422206858650`; Google Chrome was last used 1 billion, 422 million, 206 thousand, 858 seconds, and 650 milliseconds since midnight on 1 January, 1970! Don't worry; we don't know what date/time that is either. There are many scripts and tools available for download to convert this value to a human-readable format. We prefer the free tool **DCode**, which can be found at `http://www.digital-detective.net/digital-forensic-software/free-tools/`.

In DCode, simply select **Unix: Millisecond Value** from the drop-down list, type in the value in the **Value to Decode** field, and click on **Decode**:



The **Add Bias** field can be selected to convert the time to the desired time zone.

Alternatively, there is also a very useful online epoch converter at `http://www.epochconverter.com/`.

Using either method, we can see that Google Chrome was actually last used on January 25, 2015, at 17:27:38.650 UTC. Linux epoch time is frequently used on Android devices to store date/time values and will come up repeatedly in our application analysis.



This is the second data storage method: Linux epoch time.

Wi-Fi analysis

Wi-Fi is not technically an application (as evidenced by the fact that it is not recovered from `/data/data`), but it is an invaluable source of data that should be examined. So, we'll briefly discuss it here. Wi-Fi connection data is found in `/data/misc/wifi/wpa_supplicant.conf`. The `wpa_supplicant.conf` file contains a list of access points that the user has chosen to connect to automatically (this is set by default when a new access point is connected to). Access points that the user has "forgotten" through the device settings will not be shown. If the access point requires a password, that would also be stored in the file in plain text. In the following example, the `NETGEAR60` access point required a password (`ancientshoe601`), while `hhonors` did not:

```
network={
    ssid="NETGEAR60"
    psk="ancientshoe601"
    key_mgmt=WPA-PSK
    priority=22
}

network={
    ssid="hhonors"
    key_mgmt=NONE
    priority=50
}
```



The presence of a **Service Set ID** (**SSID**) in this file does NOT mean that this device connected to that access point. These settings are saved to a user's Google account and added to the device when that account is set up. An examiner can only conclude that the user connected to these access points from some Android device, but not necessarily the device being examined.

Contacts/call analysis

Contacts and call logs are stored in the same database. Contacts do not have to be added explicitly by the user. They may be autofilled when an e-mail is sent through Gmail, or a person is added on Google+, or possibly many other ways.

Package name: `com.android.providers.contacts`

Version: Default version with Android 5.0.1 (not listed within app)

Files of interest:

- /files/
    - photos/
    - profile/
- /databases/
    - contacts2.db

The files directory contains photos for the user's contacts in the photos directory and the user's profile photo in the profile directory.

The contacts2.db database contains all of the information about calls made to and from the device and all contacts in the user's Google account. It contains the following tables:

Table	Description
accounts	This shows the accounts on the device that have access to the contacts list. At least one of the accounts will show the user's Google account e-mail address. This list may include third-party apps installed that have permission to access the contacts list (we will see this in the Tango, Viber, and WhatsApp sections).
calls	This contains information regarding all calls to and from the device. The number column shows the remote user's phone number, whether the call was sent or received. The date column is the date/time of the call, stored in the Linux epoch format. The duration column is the length of the call, in seconds. The type column indicates the type of call: • 1 = incoming • 2 = outgoing • 3 = missed The name column shows the remote user's name, if the number was stored in the contact list. The geocoded_location column shows the location of the phone number based on the area code (for US numbers) or country code.
contacts	This contains partial information for contacts (more data can be found in the raw_contacts table). The name_raw_contact_id value corresponds to the _id value in the raw_contacts table. The photo_file_id value corresponds to the filename found in the /files/photos directory. The times_contacted and last_time_contacted columns show the number of times that contact was called from or made a call to the device, and the time of the last call in the Linux epoch format.

Table	Description
`data`	This table contains all of the information for each contact: e-mail address, phone numbers, and so on. The `raw_contact_id` column is a unique value for each contact that can be correlated with the `_id` value in `raw_contact_id` to identify the contact. Note that each contact may have several rows, as seen by the identical `raw_contact_id` values. There are 15 data columns (`data1` to `data15`) that contain some information about the contact, but there are no discernible patterns. The same column may contain the contact name, an e-mail address, a Google+ profile, and so on. The value in the `data14` column correlates to the filenames of the images in the `/files/profiles` path. The `data15` column contains a thumbnail of the contact's profile photo.
`deleted_contacts`	This contains a `contact_id` value and `deleted_contact_timestamp` in the Linux epoch format. However, this cannot be correlated back to any other tables to identify the name of the contact that was deleted. It may be possible to use the deleted data-recovery techniques in *Chapter 6*, *Recovering Deleted Data from an Android Device*, to recover the contact names, though. The `contact_id` value corresponds to the `contact_id` column in the `raw_contacts` table.
`groups`	This shows groups in the contact list, either automatically generated or created by the user. The title of the group is the name of the group. There does not appear to be a way to identify users in each group.
`raw_contacts`	This contains all information for every contact in the contact list. The `display_name` column shows the contact's name, if it is available. To determine the contact's phone number, e-mail address, or other information, the `_id` column value must be matched back to the `raw_contact_id` value in the data table. The `sync3` column shows a timestamp, but based on our testing, this cannot be assumed to be the time the contact was added. We had contacts several years old that were synced this month. The `times_contacted` and `last_time_contacted` columns only apply for phone calls; sending an e-mail or SMS to a contact did not increment these values. We were unable to identify any means to determine whether a contact was added through the phone interface, added as a friend on Google+, or added through other methods.

SMS/MMS analysis

SMS and MMS messages are stored in the same database. In our experience, this database is also used, regardless of what application is used to send the SMS/MMS (that is, sending an SMS through Google Hangouts will populate this database, not the Hangouts database examined here). However, third-party apps may also record the data in their own databases.

Package name: `com.android.providers.telephony`

Version: Default version with Android 5.0.1 (not listed within app)

Files of interest:

- `/app_parts`
- `/databases/`
    - `mmssms.db`
    - `telephony.db`

The `app_parts` directory contains attachments sent as an MMS, both sent and received.

The `telephony.db` database is small, but contains one potentially useful source of information. The table in telephony.db is described as follows:

Table	Description
`siminfo`	This contains historical data for all SIMs that have been used in the device, including the ICCID, phone number (if it was stored on the SIM), and the **mobile country code** (**MCC**) / **mobile network code** (**MNC**), which can be used to identify the network provider.

The `mmssms.db` database contains all information regarding SMS and MMS messages as described in the following table:

Table	Description
`part`	This contains information about files attached to an MMS. Each message will have at least two parts: an SMIL header and the attachment. This can be seen in the `mid` and `ct` columns, as well as the file type attached. The `_data` column provides the path to find the file on the device.
`pdu`	This contains metadata about each MMS. The `date` column identifies when the message was sent or received, in the Linux epoch format. The `_id` column appears to correspond to the mid value in the part column; correlating these values will show the time a specific image was sent. The `msg_box` column shows the direction of the message (`1` = received and `2` = sent).

Table	Description
sms	This contains metadata about each SMS (it does not include MMS information). The address column shows the phone number of the remote user, regardless of whether it was a sent or received message. The person column contains a value that can be looked up in the contacts2.db database and corresponds with raw_contact_id in the data table. The person column would be blank if it was a sent message or if the remote user is not in the contacts list. The date column shows the timestamp when a message was sent in the Linux epoch format. The type column shows the direction of the message (1 = received, 2 = sent). The body column displays the content of the message. The seen column indicates whether or not the message was read (0 = unread, 1 = read); all sent messages will be marked as unread.
words, words_content, words_segdir	This appears to contain duplicate content of messages; the exact purpose of this table is unclear.

User dictionary analysis

The **user dictionary** is an incredible source of data for an examiner. While it is not necessarily a standalone application, its data is stored in /data/data directory as if it were. The user dictionary is populated any time the user types a word that isn't recognized and chooses to save the word to avoid it being flagged by autocorrect. Interestingly, our test device contained dozens of words that we never typed or saved on the device. This data appears to sync with a user's Google account and persists across multiple devices. Words synced from the account were added in alphabetical order at the top of the database, while words added manually afterwards were populated in the order they were added at the bottom.

Package name: com.android.providers.userdictionary

Version: Default version with Android 5.0.1 (not listed within app)

Files of interest:

- /databases/user_dict.db

The table in the user dictionary is described as follows:

Table	Description
words	The word column contains the word that was added to the dictionary. The frequency column should likely be ignored; it displayed the same value (250) regardless of the number of times we used the word.

Here are sample entries from a user dictionary:

_id	word	frequency	locale	appid	shortcut
33	ok	250	en_US	0	
34	reddit	250	en_US	0	
35	smores	250	en_US	0	

Gmail analysis

Gmail is an e-mail service provided by Google. A Gmail account is often asked for, though is not required, when the device is being set up for the first time.

Package name: `com.google.android.gm`

Version: Default version with Android 5.0.1 (not listed within app)

Files of interest:

- `/cache`
- `/databases/`
    - `mailstore.<username>@gmail.com.db`
    - `databases/suggestions.db`
- `/shared_prefs/`
    - `MailAppProvider.xml`
    - `Gmail.xml`
    - `UnifiedEmail.xml`

The `/cache` directory within the application folder contains recent files that were attached to e-mails, both sent and received. These attachments are saved here even if they are not explicitly downloaded by the user.

The `mailstore.<username>@gmail.com.db` file contains a variety of useful information. Interesting tables within the database include the following:

Table	Description
`attachments`	This contains information about attachments, including their size and file path on the device (the `/cache` directory mentioned earlier). Each row also contains a `messages_conversation` value. This value can be compared with the `conversations` table to correlate an attachment with the e-mail it was included within. The `filename` column identifies the path on the device where the file is located.
`conversations`	In older versions, entire e-mail conversations could be recovered. In the current version, Google no longer stores the entire conversation on the device, possibly assuming that the user will have a data connection to download the full conversation. Instead, only the subject line and a "snippet" can be recovered. The snippet is roughly the amount of text that would appear in the notification bar or inbox screen of the app. The `fromCompact` column identifies the sender and any other recipients.

The `suggestions.db` database contains terms that were searched within the application.

The XML files within the `shared_prefs` directory can confirm the account(s) that were used with the application. `Gmail.xml` contained another account that was linked with our test account, but never used with the application. `UnifiedEmail.xml` contained a partial list of senders who e-mailed the account, but with no discernible rationale. Many senders were on the list, but far from all, and they appeared in no particular order. `Gmail.xml` also contained the last time that the application was synced in the Linux epoch format.

Google Chrome analysis

Google Chrome is a web browser and is the default browser on Nexus and many other devices. Chrome data on the device is somewhat unique, in that, it contains data not just from the device, but from all devices on which the user has logged in to Chrome. This means that it is entirely possible (even very likely) that data from the user browsing on their desktop computer will be found in the databases on their phone. However, this also leads to huge amounts of data for an examiner to sort through, but that's a good problem to have.

Package name: `com.android.chrome`

Version: 40.0.2214.89

Files of interest:

- `/app_chrome/Default/`
    - `Sync Data/SyncData.sqlite3`
    - `Bookmarks`
    - `Cookies`
    - `Google Profile Picture.png`
    - `History`
    - `Login Data`
    - `Preferences`
    - `Top Sites`
    - `Web Data`
- `/app_ChromeDocumentActivity/`

All of the files listed earlier in the `/app_chrome/Default` folder, except for the one `.png` file, Bookmarks, and Preferences, are SQLite databases despite the lack of a file extension.

The `SyncData.sqlite3` database is interesting because it appears to contain a list of data that has been synced from the user's account on the device back to Google's servers. Our database, with a very active Chrome account, contained over 2700 entries and included browsing history, autofill form information, passwords, and bookmarks. As an example, we were able to find a term one of the authors had searched for from 2012, seen in the following screenshot. This is interesting because the user purchased this phone in 2014, but previous data is still synced to the device.

mtime	non_unique_name
Filter	Filter
1331830155697	http://www.google.com/search?sourceid=chrome-mobile&ie=UTF-8&q=sim+card+repair+station

Table	Description
`metas`	There are many columns in the database that contain timestamps, and in our database, they all appear to be within seconds of each other for each entry. It is unclear which time corresponds to the exact time an entry was added, but all of the times roughly correspond with the time of the activity in the user's account. The columns with timestamps are `mtime`, `server_mtime`, `ctime`, `server_ctime`, `base_version`, and `server_version`. The `non_unique_name` and `server_non_unique_name` columns show the content that was synced. For example, one of our entries shows: `autofill_entry | LNAME | Tindall` Other entries in these columns include URLs visited, passwords, and even devices that the account has used.

The `Bookmarks` file is a plain-text file that contains information about the bookmarks synced with the account. It includes the name of each site that is bookmarked, the URL, and the date/time it was bookmarked, stored in a format we have not come across yet: the WebKit format. To decode the values, see the Decoding the WebKit time format section.



This is the third method of data storage: the WebKit time format.

The `Cookies` database stores cookie information for sites visited (depending on the site and Chrome settings), including the name of the site, date the cookie was saved, and the last time the cookie was accessed, in the WebKit time format.

The `Google Profile Picture.PNG` file is the user's profile picture.

The `History` database contains the user's web history stored in the following tables:

Table	Description
`keyword_search_terms`	This contains a list of terms that were searched to use Google within Chrome. The `term` column shows what was searched, while `url_id` can be correlated with the URLs table to see the time of the search.
`segments`	This table contains some URLs that were visited, but not all. It is not clear what causes data to be entered into this table.

Table	Description
`urls`	This contains the browsing history for the Google account across all devices, not just the device the database was pulled from. Our history went back approximately 3 months and contained 494 entries, although the Google account is much older than that, and we have certainly visited more than 494 pages in that time. It is unclear exactly what causes this discrepancy or determines the cut-off date for the history. The `id` column is a unique value for each row in the table. The `url` and `title` columns contain the URL visited and the name of the page. The `visit_count` column appears to be an accurate count of how many times the URL was visited. The value in `typed_count` column is always equal to or lesser than the value in the `visit_count` column, but we do not know exactly what it indicates. For some sites, the discrepancy can be accounted for by factoring in the number of times the site was visited through a bookmark rather than typing the URL, but this does not hold true for all cases. The `last_visit_time` column is the last time the URL was visited, in the WebKit time format.
`visits`	This contains a row for each visit to the URLs in the urls table; the number of entries in this table for a URL corresponds to the value in the `visit_count` column of the `url` table. The `url` column value correlates to the value in the `id` column of the `url` table. The time of each visit can be found in the `visit_time` column, again in the WebKit time format.

The `Login Data` database contains login information saved in Chrome and is synced across all devices that use the Google account.

Table	Description
`logins`	The `origin_url` is the site the user visited initially, and `action_url` is the URL of the login page if the user is redirected to one. If the first page visited is the login page, then both URLs are the same. The `username_value` and `password_value` columns show the username and password stored for that URL in plain text; and no, we're not going to include a screenshot of our database! The `date_created` is the date/time that the login information was first saved, in the WebKit time format. The `date_synced` column is the date/time on which the login data was synced locally to the device, again in the WebKit time format. The `times_used` column shows the number of times that login information was autofilled by Chrome after it was saved (excluding the first login, so some values may be 0).

`Preferences file` is a text file and contains the Google account(s) the user has signed into Chrome with.

The `Top Sites` database contains the sites that are most frequently visited, as these are shown by default when Chrome opens.

The `Web Data` database contains information the user has saved in order to automatically fill in forms on websites.

Table	Description
`autofill`	This contains a list of fields on web-based forms and the value the user typed. The `name` column shows the name of the field that was typed in, while the `value` column shows what the user typed. The `date_created` and `date_last_used` columns are self-explanatory and are stored in the Linux epoch format. Note that while this is potentially very valuable information (for example, our database contained a few usernames not stored elsewhere), there is also very little context available. The URL where the information is not stored may not be determinable.
`autofill_profile_emails`	This contains all values the user has saved to autofill the `e-mail` field on a web form.
`autofill_profile_names`	This contains all values the user has saved to autofill the `First`, `Middle`, `Last`, and `Full Name` fields on a web form.
`autofill_profile_phonwa`	This contains all values the user has saved to autofill the `Phone Number` field on a web form.
`autofill_profiles`	This contains all values the user has saved to autofill address information fields on a web form.

The `/app_ChromeDocumentActivity/` directory contains files with history for recent tabs that were open on the device. URLs can be recovered from these files for sites that were visited.

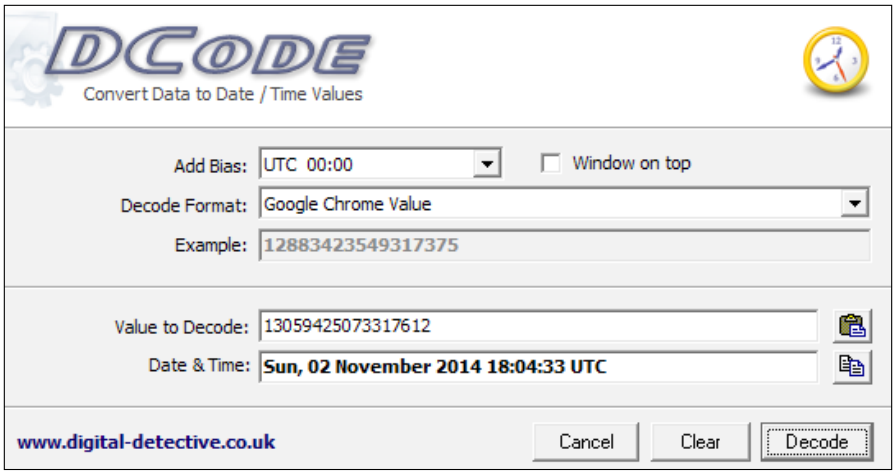
Decoding the WebKit time format

Here is a sample WebKit time value: `13066077007826684`.

At first glance, it appears to be very similar to the Linux epoch time, just slightly longer (perhaps it is storing nanoseconds?). An examiner who attempts to decode this as epoch time will get a date in May 2011, which may seem accurate, but is, in fact, several years off from the correct date!

The WebKit time is an epoch time. It is just based on a different starting point than the Linux epoch time. The WebKit epoch time is the number of microseconds since midnight on January 1, 1601. Yes, we said the year 1601. Once we know where the epoch begins, converting to a recognizable format simply becomes a math problem. However, once again, we'd rather use DCode.

This time, in DCode, choose Google Chrome Value in the **Decode Format** drop-down selection and click on **Decode**:



The actual value of our example is November 2, 2014 at 18:04:33 UTC. This is significantly different from the value we would have come up with if we thought it was a Linux epoch time!

Google Maps analysis

Maps is a map/navigation application provided by Google.

Package name: `com.google.android.apps.maps`

Version: 9.2.0 (#902013124)

Files of interest:

- `/cache/http/`
- `/databases/`
    - `gmm_myplaces.db`
    - `gmm_storage.db`

The `/cache/http` folder contains many files, with `.0` and `.1` file extensions. The `.0` files are web requests for the corresponding `.1` file. The `.1` files are predominantly images and can be viewed by changing their extension appropriately. On our test device, they were either `.jpg` or `.png` files. These files were predominantly locations near the user, not necessarily locations the user specifically searched for.



This is the fourth data storage method: misnamed file extensions.

Always verify the header of a file that can't be opened, or use automated tools, such as EnCase, to detect the mismatched header/file extension. A good resource to verify a file's signature is `http://www.garykessler.net/library/file_sigs.html`.

The `gmm_myplaces.db` database contains locations saved by the user. This file syncs with the user's Google account, so these locations were not necessarily saved using the application.

The `gmm_storage.db` database contains search hits and locations that were navigated to the following table:

Table	Description
`gmm_storage_table`	The `_key_pri` column appears to identify the type of the location. bundled appears to be a hit that came up on a search, while `ArrivedAtPlacemark` identifies locations that were actually navigated to. The `_data` column contains the address for the location.

Google Hangouts analysis

Hangouts is a chat/SMS application provided by Google. Hangouts is the default SMS client on Android devices.

Package name: `com.google.android.gm`

Version: Default version with Android 5.0.1 (not listed within the app)

Files of interest:

- `/cache/volleyCache/`
- `/databases/babel#.db` (our device had babel0.db and babel1.db)
- `/shared_prefs/accounts.xml`

The `cache` directory contains `.0` files, as discussed in the Google Maps example earlier. The files contain a URL to fetch the profile pictures of contacts, as well as a `.jpg` embedded within the file. Visiting the URL or carving the `.jpg` from the file will recover the contact's picture.

The `babel#.db` file contains all messaging data. On our test device, `babel0.db` was blank, and `babel1.db` had all of the data for the active account. There are many tables within this database worth looking at:

Table	Description
`conversations`	This contains conversation data. There is a unique `conversation_id` value for each chat. The `latest_message_timestamp` column is the time of the most recent chat, in the Linux epoch format. The `generated_name` column has a list of all participants in the chat, minus the account on the device. The `snippet_text` column is the content of the most recent message; like Gmail, the entire chat is not stored on the device. The `latest_message_author_full_name` and `latest_message_author_first_name` columns identify the author of the `snippet_text` column. The `inviter_full_name` and `inviter_first_name` columns identify which person initiated the conversation.
`dismissed_contacts`	This has a list of names of former contacts that had been messaged. These are labeled as "Hidden Contacts" within the app.
`messages`	As expected, this contains a detailed message history for each conversation. The `text` column contains the content of the message, and the `timestamp` column is the date/time in the Linux epoch format. The `remote_url` column is, once again, a URL to retrieve images shared in the message. Again, it can be accessed publically. The `author_chat_id` is a value that can be correlated with the participants table to identify the author of each message.
`participants`	This contains a list of people chatted with. It includes full names, profile picture URLs, and a `chat_id` value to identify the person in the messages table.

The `accounts.xml` file has a `phone_verification` field that contains the phone number associated with the Google account when Hangouts is configured to send SMS. This could be highly useful, because it is frequently difficult to obtain the device's phone number as it is often not stored on the device.

Google Keep analysis

Keep is a note-taking application provided by Google. It can also be used to set reminders, either at a certain date/time or when the user is at a specified location.

Package name: `com.google.android.keep`

Version: Default version with Android 5.0.1 (not listed within app)

Files of interest:

- `/databases/keep.db`
- `/files/1/image/original`

The `files/1/image/original` directory contains photos taken using the app. Notes and reminders can both be associated with an image.

The `keep.db` contains all of the information about notes and reminders. There are, once again, several tables of interest:

Table	Description
`alert`	This contains information about location-based reminders. The `reminder_id` column can be correlated with entries in the reminder table. The `reminder_detail` table contains the latitude and longitude set for the reminder. The `scheduled_time` column is the date/time the reminder was set, in the Linux epoch time.
`blob`	This contains metadata about images in the `/files` directory mentioned earlier, including the filename and size. The `blob_id` column can be correlated with the `_id` column in the `blob_node` table.
`blob_node`	This contains the time-created value for the images in the `/files` directory, in the Linux epoch time.
`list_item`	This stores data for each note on the device. The `text` column contains the full text of each note. The `list_parent_id` column is a unique value for each note. If multiple rows have the same value, it means they were created as a list within the same note. The `time_created` and `time_last_updated` columns are the time the note was created and the time it was last synced with the Google servers, in the Linux epoch time.
`reminder`	This contains data about each reminder set within the app. If the reminder is time based, the `julian_date` and `time_of_day` columns will be populated.

Converting a Julian date

Julian dates are similar to the Linux epoch format, simply starting with a different date. The Julian date system counts the number of days since noon on January 1, 4713 BC. The United States Naval Observatory has an excellent Julian date calculator. To obtain the Julian date from the database, simply combine the two columns with a decimal in between. Here is an example:

julian_day	time_of_day
Filter	Filter
2457042	46800000

The preceding date would correspond to the Julian date 2457042.46800000. When this value is input to the website, we can find out that the date the reminder is set for is January 19, 2015 at 23:13:55:2 UTC. The `location_name`, `latitude`, `longitude`, and `location_address` columns would be populated if a reminder is set as location based. Finally, the `time_created` and `time_last_updated` columns are the time the note was created and the time it was last synced with the Google servers, in the Linux epoch time.



The fifth data-storage method is Julian date.

Google Plus analysis

Google Plus is the Google-based social network. It allows us to share text/videos/images, add friends, follow people, and message. Google Plus may also, depending on the user's settings, automatically upload all pictures taken on the user's device.

Package name: `com.google.android.apps.plus`

Version: 4.8.0.81189390

Files of interest:

- `/databases/es0.db`

The `Es0.db` database contains all the information an examiner would expect to find from a social-media account:

Table	Description
`all_photos`	This contains a URL to download images shared by and with the user as well as the creation date/time in the Linux epoch format.
`activites`	This contains data displayed in the user's stream (that is, their news feed). The created and modified time for each post is, once again, stored in the Linux epoch time. The title and comment columns will contain the post title and at least some of the comments from it. The `permalink` column contains a URL that can be followed to view the post, if it was shared publically. If the post is shared privately, the content can still be recovered from the embed table. The `relateds` column contains the hashtags automatically generated for the post by Google; this would also populate even if the post is private.
`activity_contacts`	This contains a list of names for people whose posts are in the activities table.
`all_photos`	This contains a list of *all* photos the user has backed up to Google Plus, whether they were shared or not. The values in the `image_url` column can be used to download any of the user's photos and is publically available. Removing `-d` at the end of the URL will view the image without downloading it. The `timestamp` column is the date/time the image was taken, based on the image metadata. It does not indicate when the image was uploaded.
`all_tiles`	This contains an unknown subset of `all_photos`, but also includes images shared with the user.
`circle_contact`	This contains a list of people the user has added to their circles. It does not include names, but some of the `link_person_id` values include e-mail addresses. The `link_circle_id` value can be correlated with the circles table to identify the name of each circle. The `link_person_id` value can then be correlated with the `contacts` table to identify which user is in which circle.
`circles`	This has all the circles the user has created, as well as a count of the number of users in each one.
`contacts`	This contains a list of all contacts in the user's circles.

Table	Description
`events`	This lists all events the user has been invited to, whether they attended or not. The `name` column is the title of the event. The `creator_gaia_id` column can be correlated with the `gaia_id` column in the contacts table to identify the event creator. The `start_time` and `end_time` columns are the time of the event, in the Linux epoch format. The `event_data` column has the description of the event entered by the creator, as well as information about the location if added. It also lists all the other users who were invited to the event.
`squares`	This contains a list of groups the user has joined.

Facebook analysis

Facebook is a social-media application with more than 1 billion downloads from Google Play.

Package name: `com.facebook.katana`

Version: 25.0.0.19.30

Files of interest:

- `/files/video-cache/`
- `/cache/images/`
- `/databases/`
    - `bookmarks_db2`
    - `contacts_db2`
    - `nearbytiles_db`
    - `newsfeed_db`
    - `notifications_db`
    - `prefs_db`
    - `threads_db2`

The `/files/video-cache` directory contains videos from the user's newsfeed, though there does not appear to be a way to correlate them back to the user who posted them.

The `/cache/images` directory contains images from the user's newsfeed as well as the profile photos of contacts. This directory contains a multitude of other directories (65 on our test phone), and each directory can contain multiple `.cnt` files. The `.cnt` files are typically `.jpg` files or other image formats.

The `bookmarks_db2` database is a list of items that appear on the side of the user's newsfeed, such as groups and applications. Many of these bookmarks are automatically generated by Facebook, but may also be created by the user.

Table	Description
`bookmarks`	This contains all of the info within the database. The `bookmark_name` column is the name of the bookmark displayed to the user. The `bookmark_pic` column has a publically accessible URL to view the `bookmark` icon displayed to the user. The `bookmark_type` column identifies the type of the group. Our testing showed `profile`, `group`, `app`, `friend_list`, `page`, and `interest_list`. Finally, the `bookmark_unread_count` column shows how many messages in the group have not been read by the user.

The `contacts_db2` database predictably contains information about all of the user's contacts stored in the following tables:

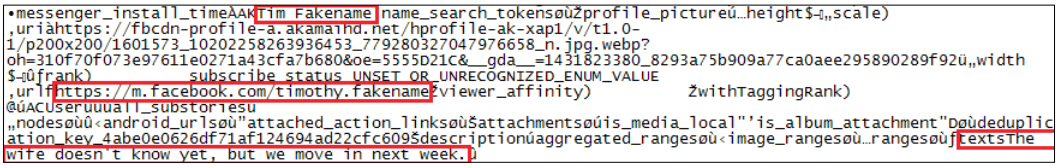
Table	Description
`contacts`	This contains all information about the user's contacts. The `fbid` column is a unique ID that is used to identify the contact in other databases. The `first_name`, `last_name`, and `display_name` columns show the contact's name. The `small_picture_url`, `big_picture_url`, and `huge_picture_url` columns contain public links to the contact's profile picture. The `communication_rank` column appears to be a number identifying how often the contact communicates with the user (taking into account messages, comments, and possibly other factors); a higher number indicates more communication with that contact. The `added_time_ms` column shows the time (in the Linux epoch format) the contact was added as a friend. The `bday_day` and `bday_month` columns show the contact's birth date, but not the year. The `data` column contains a duplicate of all the rest of the data in the database, but also contains the contact's location, which is not found elsewhere in the database.

The `nearbytiles_db` database contains locations near the user that may interest them. This is apparently populated constantly, even if the user does not view the locations. It is interesting because, while it isn't a fine location (most of our tests showed locations within 6–10 miles of our location), it is a rough idea of places a user has been.

Table	Description
`nearby_tiles`	This contains the `latitude` and `longitude` values for locations near the user, as well as the time the location was retrieved from the Facebook servers in the Linux epoch format.

The `newsfeed_db` database contains data shown to the user in their newsfeed. Depending on the usage of the app, it can be a very large file containing the table:

Table	Description
`home_stories`	The `fetched_at` column shows the time the story was pulled from the Facebook servers and likely corresponds closely with the time the user was using the application or saw the story. The `story_data` column contains the story stored as a blob of data. When viewed in a hex or text editor, the username of the person posting the story can be found. The content of the post can also be found in plain text and is often preceded by a tag that says `text`. An example of this is shown in the following screenshot.





Note that the actual content of this one cell in the `story_data` column. It contained over 10,000 bytes of data, though the actual message is only around 50 bytes.

The `notifications_db` database contains notifications sent to the user stored in the following table:

Table	Description
`gql_notifications`	The `seen_state` column shows whether or not the notification has been seen and read. The `updated` column contains the time the notification was updated (that is, sent if it is unread or the time it was read) in the Linux epoch format. The `gql_payload` column contains the content of the notification as well as the sender, similar to the `story_data` column in `newsfeed_db`. The message content again is frequently preceded by the flag `text`. A much smaller amount of data showing the text of the notification can be found in the `summary_graphql_text_with_entities` and `short_summary_graphql_text_with_entities` columns. The `profile_picture_uri` contains a public URL to view the sender's profile picture, and the `icon_url` column has a link to view the icon associated with the notification.

The `prefs_db` database contains application preferences stored as follows:

Table	Description
`preferences`	The `/auth/user_data/fb_username` row shows the user's Facebook username. The `/config/gk/last_fetch_time_ms` value is the timestamp of the app's last communication with Facebook servers, but may not be an exact time of the user's last interaction with the app. The `/fb_android/last_login_time` value shows the last time the user logged in through the app. The database contains many other timestamps. When put together, these timestamps can be used to build a decent profile of the app's usage. The `/auth/user_data/fb_me_user` value contains data about the user, including their name, e-mail address, and phone number.

The `threads_db` database contains messaging information described as follows:

Table	Description
`messages`	Each message has a unique ID in the `msg_id` column. The `text` column contains the message in plain text. The `sender` column identifies the Facebook ID and name of the message sender. The `timestamp_ms` column is the time the message was sent, in the Linux epoch format. The `attachments` column contains a public URL to retrieve attached images. The `coordinates` column would have the sender's latitude and longitude if they have opted to show their location. The `source` column identifies whether the message was sent via the website or app.

Facebook Messenger analysis

Facebook Messenger is a messaging app separate from the main Facebook application. It has over 500,000,000 downloads in the Play Store.

Package name: `com.facebook.orca`

Version: 18.0.0.27.14

Files of interest:

- `/cache/`
    - `audio/`
    - `fb_temp/`
    - `image/`
- `/sdcard/com.facebook.orca`
- `/files/ rti.mqtt.analytics.xml`
- `/databases/`
    - `call_log.sqlite`
    - `contacts_db2`
    - `prefs_db`
    - `threads_db2`

The `/cache/audio` directory contains audio messages sent through the application. The files have a `.cnt` file extension, but are actually `.riff` files that can be played with Windows Media Player, VLC media player, and other programs.

The `/cache/fb_temp` path contains temp files for images and video sent through the application. It is unclear how long these files will remain. In our testing, we sent and received a total of five files, and all five were still in the temp folder one week later.

The `/cache/image` directory contains a multitude of other directories (33 on our test phone), and each directory can contain multiple `.cnt` files. The file header should be verified on each file, as some were video files and some were images. Several of the files from the `fb_temp` folder were found, as well as the profile pictures of some contacts.

The `fb_temp` folder on the SD card contains sent images and video only.

The application also includes an option (disabled by default) to download all the received images/video to the device's gallery. If this option is selected, all received images/video would be found on the SD card.

The `/files/rti.mqtt.analytics.xml` file has the user's Facebook UID.

The `call_log.sqlite` database contains a log of calls made through the application. The `person_summary` table contains the relevant data described as follows:

Table	Description
`person_summary`	The `user_id` column contains the Facebook ID of the remote user. This can be correlated with the `fbid` column in `contacts_db2` to determine the user's name. The `last_call_time` column contains the time of the previous call in the Linux epoch format. This table does not contain information about the direction of the call (sent or received).

The `contacts_db2` file is a SQLite database, despite the lack of a file extension. Useful tables within this database include the following ones:

Table	Description
`contacts`	This table includes the contacts the user has added, as well as the contacts that were scraped from the user's phone book (if the phone book contact uses Facebook Messenger). It contains the first and last names of each contact, as well as that contact's Facebook ID (as discussed in the `call_log.sqlite` table earlier). The `added_time_ms` column shows the time each user was added into the app. This can give some insight into whether the contact was added manually or automatically. A large group of contacts added within milliseconds of each other were likely created automatically when the app was installed. The `small_picture_url`, `big_picture_url`, and `huge_picture_url` columns contain public links to the contact's profile picture. A contact's phone number can be found in the blob of information within the data column. It should be noted that we have no idea where some of the contacts in this database came from. They were not Facebook friends with our account and were not contacts in our device's phone book, but were added at the same time that the phone book was scraped. Our best guess is that some contacts in our phone have phone numbers that Facebook associated with other users.
`favorite_contacts`	The `favorite_contacts` table shows contacts that have been added as favorites by the user. They are identified by the `fbid` column, which can be correlated back to the contacts table.

The `prefs_db` database contains useful metadata about the app and the account:

Table	Description
`preferences`	The `/messenger/first_install_time` value indicates the time the application was installed, in the Linux epoch time. The `/auth/user_data/fb_username` value shows the username associated with the application. The `/config/neue/validated_phonenumber` value shows the phone number associated with the application. The users first and last names can be found in the `/auth/user_data/fb_me_user` value.

Finally, the `threads_db2` database contains data about messages:

Table	Description
`group_clusters`	This shows folders the user has created.
`group_conversations`	This contains the `thread_key` value for each group chat. This can be correlated with the `messages` table.
`messages`	The `thread_key` value is a unique ID generated for each chat session. The `text` column has the contents of each text message sent and received. This also identifies voice calls using the phrases "You called Facebook User.", "Facebook User called you.", and "You missed a call from Facebook User.". The `sender` column identifies which user sent each message (or made each call). The `timestamp_ms` column shows the time each message was sent, in the Linux epoch format. The `attachments` column will show data for each sent or received attachment. The file type is also visible in the data. The `pending_send_media_attachment` column shows the path on the device to recover sent attachments. Finding received attachments directly does not appear possible, although they were recovered in the `/cache/images` directory discussed earlier. There was no way to correlate them with a specific message or sender.

Skype analysis

Skype is a voice-/video-calling app, as well as a messaging app owned by Microsoft. It has over 100,000,000 installs on Google Play.

Package name: `com.skype.raider`

Version: 5.1.0.58677

Files of interest:

- `/cache/skype-4228/DbTemp`
- `/sdcard/Android/data/com.skype.raider/cache/`
- `/files/`
    - `shared.xml`
    - `<username>/thumbnails/`
    - `<username>/main.db`
    - `<username>/chatsync`

The `/cache/skype-4228/DbTemp` directory contained multiple files with no extension. One of these files (`temp-5cu4tRPdDuQ3ckPQG7wQRFgU` on our device) was actually a SQLite database that contained the SSID and **Media Access Control** (**MAC**) of the wireless access points it had been connected to.

The SD card path will contain any images or files received in a chat. If a file is downloaded, it would be in the `Downloads` folder in the root of the SD.

The `shared.xml` file listed the account's username as well as the last IP address that connected to Skype:

```
<LastIP>-1202185837</LastIP>
<LastNetworkIdentity>20e52a088685</LastNetworkIdentity>
<LastProbingFailed>0</LastProbingFailed>
<ListeningPort>1305</ListeningPort>
<NatTracker>
  <ContraProbeResults>184.88.25.147:1305</ContraProbeResults>
```

The `<username>/thumbnails` directory contained the user's profile picture.

The `main.db` database, like it sounds, contains the app usage history. Some important tables to look at are as follows:

Table	Description
`Accounts`	This shows the accounts used on the device and the associated e-mail addresses.
`CallMembers`	This includes call logs from the app. The `duration` table is the duration of the call, and the `start_timestamp` column is the start time in the Linux epoch format; neither of these columns is populated if the call is not answered. The `creation_timestamp` column is the actual beginning of the call. It is populated as soon as the call is initiated within the app, so even unanswered calls are shown in this column. The `ip_address` column shows the IP address of the user for connected calls. The `type` column indicates whether the call was outgoing or incoming (1=incoming, 2=outgoing). The `guid` column also shows the direction of the call, listing each participant from left to right, with the user on the left-hand side being the one who initiated the call. The `call_db_id` column can be correlated with the `Calls` table to find further information about the call.
`Calls`	This is very similar to `CallMembers`, but with less information. It is worth noting that the `begin_timestamp` column in this table is identical to `creation_timestamp` in `CallMembers`. There is an `is_incoming` column to show the direction of the call; `0` indicates outgoing, and `1` indicates incoming. Finally, it should be noted that the duration of some calls did *not* match the `CallMembers` table. One of the durations was a second longer than the other table indicated. It appears that the `CallMembers` table calculates duration based on `start_timestamp`, while the `Calls` table calculates duration based on `begin_timestamp`. The difference in duration is likely caused by the amount of time it took the user to accept the call.
`ChatMembers`	This shows the users in each chat. The `adder` column lists the user that initiated the chat.
`Chats`	This lists each unique chat session. The `timestamp` column is the date/time the conversation began, in the Linux epoch format. The `dialog_partner` column shows users in the chat, excluding the account on the device. The `posters` column shows every user who has made a comment in the chat and includes the account on the device if it has posted. The `participants` column is similar to the `dialog_partner` column, but includes the user's account. Finally, the `dbpath` column contains the name of the chat backup file found in the `<username>/chatsync` directory. This will become important further in this analysis.

Table	Description
Contacts	This is actually a very misleading table. In our test, we added two users to our contact list; the Contacts table has 233 entries! The is_permanent column indicates the status of the users listed in this table; if it is 1, the user would be added as an actual contact within the application. The other 231 entries appear to be names that came up in results when we searched for contacts, but we never communicated with or added them.
Conversations	We have no idea what the difference between Conversations and Chats is. They mostly contain the same information and, in fact, appear to be referencing the same chat sessions.
Messages	This contains every individual message from chats/conversations. The convo_id column has a unique value for each conversation; any messages with the same convo_id value are from the same conversation. The author and from_dispname columns show who wrote each message. The timestamp column, once again, shows the date/time of the message in the Linux epoch format. The type column indicates the type of message that was sent. Here are the values from our testing: • 50: friend request • 51: request accepted • 61: plain text message • 68: file transfer • 30: call begin (voice or video) • 39: call end (voice or video) • 70: video message The body_xml column has the content of the message. For plain-text messages and friend requests, the content is simply what the message said. File transfers show the size and name of the file. Video messages say that they are video messages, but provide no other information. Calls would show the duration if it was connected, and no duration if they were missed/ignored. The identities column shows who sent each message, but may be blank if it was sent by the user account on the device. The reason column appears to be for calls and shows either no_answer or busy to explain why a call was not connected.
Participants	This is similar to ChatMembers. It shows each user involved with a chat/conversation.
SMSes	Our testing did not include SMS messaging. However, each column in this table appears self-explanatory.

Table	Description
`Transfers`	This shows information about files transferred. This includes the filename, size, and path on the device. The `partner_dispname` column identifies which user began the file transfer.
`VideoMessages`	This shows the author and creation timestamp of video messages. Note that video messages are *not* stored on the device. Accessing them will be covered in a separate section later on in the chapter.
`VoiceMails`	Our testing did not include voice mails. However, each column in this table appears self-explanatory.

Recovering video messages from Skype

As noted earlier, video messages are not stored on the device. Luckily, for us, they can be accessed via the Internet. The first step is to verify that a video message was sent by looking in the `Messages` table in the `body_xml` column. Next, note the `convo_id` field for the message shown in the following screenshot.

	convo_id	body_xml
1	257	<videomessage sid="2e384487002d113afcb730f0b6100c5a...

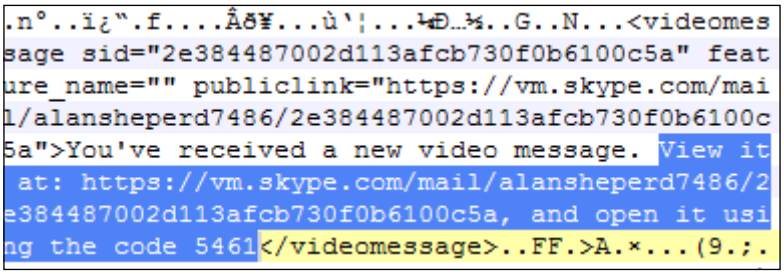
Our video message is in `convo_id` `257`.

Third, look in the `Chats` table for `convo_id` in the `conv_dbid` column and find the `dbpath` value. This will be the name of the conversation's backup file as shown in the following screenshot:

	conv_dbid	dbpath
1	257	282d282b9f5e9be0.dat

To find the backup file, look in `files/<username>/chatsync`. There will be a folder for each conversation; the name of the folder is the first two digits of the backup name. Our backup will be in folder `28`.

Open the backup file in a hex editor, and search for `videomessage`. You should find a URL and a code to access the video:





Actually, accessing the URL may require an additional warrant or legal permission, depending on your local jurisdiction. As this data is not on the device and is private, viewing it without legal guidance could invalidate any evidence found in the video.

Snapchat analysis

Snapchat is an image-sharing and text-messaging service with over 100,000,000 downloads. Its signature feature is that images and videos sent will "self-destruct" after a time limit set by the sender, from 1-10 seconds. Furthermore, if a user takes a screenshot of the image, the sender is notified. Text chats do not have an expiration timer.

Package name: `com.snapchat.android`

Version: 8.1.2

Files of interest:

- `/cache/stories/received/thumbnail/`
- `/sdcard/Android/data/com.snapchat.android/cache/my_media/`
- `/shared_prefs/com.snapchat.android_preferences.xml`
- `/databases/tcspahn.db`

The `/cache/stories/received/thumbnail` path contains thumbnails of pictures taken by the user on the device. The `/sdcard` path contains full-sized images. These remain even after the time limit has expired, and the recipient can no longer access them. The files in both of these locations may not have proper file extensions.

The `com.snapchat.android_preferences.xml` file contains the e-mail address used to create an account and the phone number of the device registered with the account.

The `tcspahn.db` database contains all other information about the app's usage:

Table	Description
`Chat`	This lists all text chats. It shows the sender, recipient, and timestamp in the Linux epoch time and the text of the message.
`ContactsOnSnapchat`	This shows all the users in the user's phonebook who also have Snapchat installed. The `isAddedAsFriend` column would show a `1` value if the user has actually been added as a contact.
`Conversation`	This has information about each open conversation. It includes the sender and recipient and the timestamp of the last sent and received snaps in the Linux epoch format.
`Friends`	This is similar to `ContactsOnSnapchat`, but only includes users who have been added as a friend. It includes the timestamp when each user added the other.
`ReceivedSnaps`	This contains metadata about received images and videos. Once the image/video is viewed, it appears to be removed from this table at some point. It contains a timestamp for each message, a status, information whether or not a snap was screenshot, and the sender.
`SentSnaps`	This contains metadata about sent images and videos. Once the image/video is viewed, it appears to be removed from this table at some point. It contains a timestamp for each message, a status, and the recipient.

Viber analysis

Viber is a messaging and voice-/video-calling app with over 100,000,000 downloads.

Package name: `com.viber.voip`

Version: 5.2.1

Files of interest:

- `/files/preferences/`
    - `activated_sim_serial`
    - `display_name`
    - `reg_viber_phone_num`
- `/sdcard/viber/media/`
    - `/User Photos/`
    - `/Viber Images/`
    - `/Viber Videos/`
- `/databases/`
    - `viber_data`
    - `viber_messages`

The files in `/files/preferences` contain the SIM card's **Integrated Circuit Card ID** (**ICCID**), the name the user displays in the app, and the phone number used to register with the app.

The files in the `/sdcard/viber/media` path are the profile photos of people in the user's contact list who use Viber (regardless of whether they have been added as friends in the app) and all images and video sent through the app.

The `viber_data` file is a database, even though it does not have the `.db` file extension. It contains information about the user's contacts:

Table	**Description**
`calls`	This table did not populate, even though we made calls from within the app.
`phonebookcontact`	This table could be extremely valuable from a forensic standpoint. When Viber is first opened, it scrapes the user's phonebook and adds all the entries it finds to this database. This means it may contain historical data about the user's contacts. If he later deletes an entry from the phone book, it may still be recovered in this database. This table only includes names of contacts in the phonebook.
`phonebookdata`	This is similar to phonebook contact, except that it includes e-mail addresses and phone numbers for contacts in the device's phonebook.

Table	Description
`vibernumbers`	This shows the Viber phone number for each contact in the device's phonebook that uses the app. The value in `actual_photo` corresponds with the filenames in the `/sdcard/viber/media/User Photos` directory.

The `viber_messages` file is a database, even though it does not have the `.db` file extension. It contains information about the app's usage:

Table	Description
`conversations`	This contains a unique ID, the recipient, and date for each unique conversation.
`messages`	This contains each individual message from all conversations. The address is the phone number of the remote party in the conversation. The `date` column is in the Linux epoch format. The `type` column corresponds to incoming or outgoing; `1` is an outgoing message, and `0` is incoming. The `location_lat` and `location_lng` columns will be populated if a location is shared. Shared files can be sent with text to describe them; this is found in the `description` column.
`messages_calls`	This table did not populate, even though we made calls from within the app.
`participants_info`	This has the profile information for each account that has been in a conversation with the user.

Tango analysis

Tango is a voice-/text-/video-messaging application. It has over 100,000,000 downloads in the Play Store.

Package name: `com.sgiggle.production`



This package name is seemingly innocuous and could be overlooked by an examiner thinking it was a game. This is an example of why every application should be analyzed.

Version: 3.13.128111

Files of interest:

- `/sdcard/Android/data/com.sgiggle.production/files/storage/appdata/`
    - `TCStorageManagerMediaCache_v2/`
    - `conv_msg_tab_snapshots/`
- `/files/`
    - `tc.db`
    - `userinfo.xml.db`

The `/TCStorageManagerMediaCache_v2` path on the SD card contains images that were sent and received with the application as well as profile pictures of contacts. However, it also contains many images that were never seen or used in the application. They appear to either be images for ads or stock emoji-type images that can be attached to conversations. The filenames found here can be correlated with `tc.db` to find the exact image that was used in a conversation.

The `conv_msg_tab_snapshots` path on the SD card contains files with a `.dat` extension. When viewed in a hex editor, we were able to find snippets of conversations in plain text, as well as paths and URLs to images sent and received in conversations. It is unclear what causes these files to exist, but it may be possible to retrieve content from these files that may have been deleted in `tc.db`.

The `tc.db` database is what Tango uses to store all message information:

Table	Description
`conversations`	This contains a unique ID in the `conv_id` column for each conversation.
`messages`	This contains messages sent and received through the app. The `msg_id` column is a unique identifier for each message, and the `conv_id` column identifies which conversation the message is from. The `send_time` column identifies the time a message was sent or when it was received, depending on the direction. The `direction` column shows the direction of the message; `1` = sent and `2` = received. The `type` column identifies the type of the message. Based on our testing, they are as follows: • `0` = plain text message • `1` = video message • `2` = audio message • `3` = image • `4` = location/coordinates • `35` = voice call • `36` = attempted voice call (missed by either party) • `58` = attached stock image, such as the emojis found in the TCStorageManagerMediaCache_v2 path Finally, the `payload` column contains the content of the message. The data is Base64 encoded, which will be discussed in detail in the following section.

The `user_info_xml.db` database contains metadata about the account, such as the user's name and phone number. However, its data is entirely Base64 encoded, like the messages in `tc.db`.

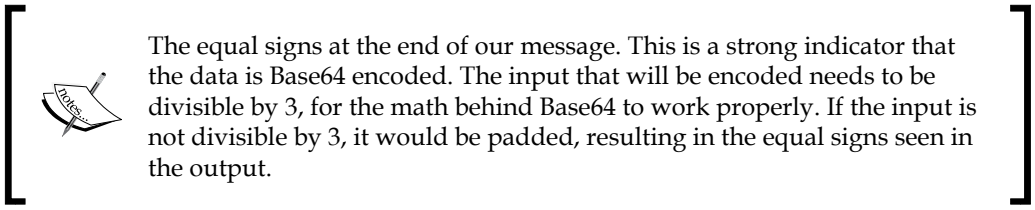


The next data-storage method is Base64.

Decoding Tango messages

Base64 is an encoding scheme that is commonly used for data transport. It is not considered encryption, because it has a known method for decoding and does not require a unique key to decode the data. Base64 contains ASCII-printable characters, but the underlying data is binary (which will make our output somewhat messy!). An example from the `payload` column in the `messages` table of `tc.db` looks like this:

```
EhZtQzVtUFVQWmgxWnNRUDJ6aE44cy1nGAAiQldlbGNvbWUgdG8gVGFuZ28hIEhlcmUn
cyBob3cgdG8gY29ubmVjdCwgZ2V0IHNvY2lhbCwgYW5kIGhhdmUgZnVuIYABAKoBOwoFV
GFuZ28SABoWbUM1bVBVUFpoMVpzUVAyemhOOHMtZyILCgcKABIBMRoAEgAqADD//////
/////8BsAHYioX1rym4AYKAgAjAAQHQAQDoAdC40ELIAgTQAgDqAgc4MDgwODg5yAMA2
AMA2AXTHw==
```



The equal signs at the end of our message. This is a strong indicator that the data is Base64 encoded. The input that will be encoded needs to be divisible by 3, for the math behind Base64 to work properly. If the input is not divisible by 3, it would be padded, resulting in the equal signs seen in the output.

For example, consider the following table:

Input string	Number of characters/ bytes	Output
`Hello,World`	12	`SGVsbG8sIFdvcmxk`
`Hello, World!`	13	`SGVsbG8sIFdvcmxkIQ==`
`Hello, World!!`	14	`SGVsbG8sIFdvcmxkISE=`

You can see that the 12-byte input (divisible by 3) has no padding, while the other two inputs have padding because they are not divisible by 3. This is important because it shows that while the equal signs are a strong indicator of Base64, the lack of an equal sign does not mean it isn't Base64!

Now that we understand a little about Base64 and recognize that our `payload` column is very likely encoded in Base64, we need to decode it. There are websites that will allow the user to paste in encoded data, and it will be decrypted (such as `www.base64decode.org`). However, it is inconvenient for large amount of data as each message must be input individually (and putting evidentiary data on the Internet is also frowned upon in most cases). Likewise, it can be decoded on the command line of Linux-based systems, but is equally inconvenient for large amounts of data.

Our solution was to build a Python script that pulls the Base64 data from the database, decodes it, and writes it back out to a new file:

```
import sqlite3
import base64

conn = sqlite3.connect('tc.db')
c = conn.cursor()
c.execute('SELECT msg_id, payload FROM messages')
message_tuples = c.fetchall()
with open('tcdb_out.txt', 'w') as f:
  for message_id, message in message_tuples:
    f.write(str(message_id) + '\x09')
    f.write(str(base64.b64decode(message)) + '\r\n')
```

To run the code, simply paste this code into a new file named `tcdb.py`, place the script in the same directory as `tc.db`, and on the command line, navigate to that directory and run:

```
python tcdb.py
```

The script will make a file named `tcdb_out.txt` in the same directory. Opening the file in a text editor (or importing it into Excel as a tab-delimited file) will show the `msg_id` value so that the examiner can correlate the message back to the messages table. The decoded payload shows a plain text message (noted as type `0` in the database):

```
16777218        b'\x12\x16mC5mPUPZh1ZsQP2zhN8s-g\x18\x00"Bwelcome to Tango! Here\'s how to connect,
get social, and have fun!\x80\x01\x00\xaa\x01;\n\x05Tango\x12\x00\x1a\x16mC5mPUPZh1ZsQP2zhN8s-
g"\x0b\n\x07\n\x00\x12\x011\x1a\x00\x12\x00*\x000\xff\xff\xff\xff\xff\xff\xff\xff\xff
\x01\xb0\x01\xd8\x8a
\x85\xf5\xaf)\xb8\x01\x82\x80\x80\x08\xc0\x01\x01\xd0\x01\x00\xe8\x01\xd0\xb8\xd0B
\xc8\x02\x04\xd0\x02\x00\xea\x02\x078080889\xc8\x03\x00\xd8\x03\x00\xd8\x05\xd3\x1f'
```



Note that the message content is now visible in plain text and is preceded by the conversation ID. There is also a ton of binary data cluttering up our output; this is likely metadata or other information used by Tango. If the message was received, the user's name will also be in the output (above it is Tango).

There are other message types worth looking at, also. Here is a decoded payload entry for a video message:

```
16777217        b'\x12\x16mC5mPUPZh1ZsQP2zhN8s-g\x18\x01"\x00*K
http://cget.tango.me/contentserver/download/VJTHZwAAoEsMaPAj3tXzwQ/JRToYGJF2h
http://us0501-avmi-vip001.tango.me:8080/contentserver/download/VBEIcQAAAUflGt6uPu4RjA/yY5hPIFc/thumbnail
:\x8c\x01
/storage/emulated/0/Android/data/com.sgiggle.production/files/storage/appdata/TCStorageManagerMediaCache_v2/3
7f52b655d8a03828e5da5bdd7f99b02
@\xd4\xeb\xfe\x01H?R\x16MWGjTGUI75rWt5W5TH_5vw\x80\x01\x01\x8a
\x01\x1bhttp://u.tango.net/qvlqc7g0\x90\x01\x00\x98\x01\x00\xaa\x01;\n\x05Tango\x12\x00\x1a
\x16mC5mPUPZh1ZsQP2zhN8s-g"\x0b\n\x07\n\x00\x12\x011\x1a\x00\x12\x00*\x000\xff\xff\xff\xff\xff\xff\xff\xff
\xff\x01\xb0\x01\xd8\x8a
\x85\xf5\xaf)\xb8\x01\x81\x80\x80\x08\xc0\x01\x01\xd0\x01\x00\xe0\x01\x00\xe8\x01\xe8\xee\x96\x81\xdb
(\xc8\x02\x04\xd0\x02\x00\xea\x02\x0540000\xc8\x03\x00\xd8\x03\x00\xd8\x05\xf7\n'
```

Note that with the video message, we can see two URLs. They are both public, meaning anyone with the link can access them. The URL ending in thumbnail is a thumbnail of the video, while the other URL will download the complete video in the `.mp4` format. The path to the SD card and filename for the image is also shown.

Image and audio messages are stored in a very similar format and contain URLs to either view or download the file. They also contain the path to the file on the SD card.

Here is a sample location message:

```
16777231        b'\x12\x16lkJNty6wjOp-TfbfcTi-wA\x18\x04:
\x91\x01/storage/emulated/0/Android/data/com.sgiggle.production/files/storage/appdata/TCStorageManagerMediaCa
che_v2/7de2c42025cf79bbc029a990506ed287..jpg\x80\x01\x04\xb0\x01\xc6\x9f\xde\xf7\xaf)\xb8\x01\x8f
\x80\x80\x08\xc0\x01\x01\xd0\x01\x01\xe0\x01\x00\xe8\x01\xb1\xa1\xde\xf7\xaf)\x98\x02\xf4\xa3\xde
\xf7\xaf)\xc8\x02\x03\xd0\x02\x01\xb0\x03\x00\xc8\x03\x00\xe2\x04\xab\x018300 North Wickham Road, Melbourne,
FL 32940, USA\nhttps://www.google.com/maps/@28.231424,-80.716292,16z\n(For full experience, upgrade Tango
http://install.tango.net)\x98\x05\x00\xd8\x05\x8b\x94\x9d\xed\x81\xc0\x83\x90!\xe2\x05\x84\x01\tnj\x98T30<@
\x11\x00\x00\xa0\xb9f+T\xc0\x1a\x00"/8300 North Wickham Road, Melbourne, FL 32940,
USA*=https://www.google.com/maps/@28.231424,-80.716292,16z
```

This time, we can see the exact coordinates the user was at as well as the address. Again, a path on the SD card is also present and will show the map view of the location. As with other message types, a received message would also show the sender's name.

Finally, let's take a look at the `userinfo.xml.db` database. Here is what it looks like before being decoded properly:

1	REJfVkVSU0IPTg==	
2	ZGV2aWNIdG9rZW4udGFuZ28=	YzdkMmY0YjdmMWY2YTc3ODA5Y2...
3	cmVsZWFzZV9uYW1I	ZmFsYW5naGluYV9iaWxsaW5nX3Y=
4	M0dfY2FsbHNfYWxsb3dIZA==	MQ==
5	cGVyc2IzdGVudF9jb250YWN0X3ZIcn...	Mw==
6	YWRkcmVzc2Jvb2thY2NIc3M=	MQ==
7	c3dpZnR1c2VybmFtZQ==	MTE1Mzk4M2EzNjEwODc4ZjQwOD...
8	c3dpZnRwYXNzd29yZA==	MDNiMjNjNjY0YWI5ZGI4Yjk1MDFIN...
9	cGFzc3dvcmQ=	MjkxZGVIZDdkNmE2YTFjZDlmYzVl...
10	dXNIcm5hbWU=	NzJiNzFmMjdkM2NkYzY4NWNhZm...
11	dmVyc2Ivbg==	My4xMy4xMjgxMTE=
12	ZGV2aWNIdG9rZW4uZ2Nt	QVBBOTFiR2NIQmgzT29va2MtUGdC...
13	YWRkcmVzc2Jvb2tzdG9yZQ==	MQ==
14	dmFsaWRhdGIvbmNvZGU=	
15	Y291bnRyeWNvZGU=	MQ==
16	Y291bnRyeWNvZGVuYW1I	VW5pdGVkIFN0YXRIcw==

We wrote another script very similar to the first to parse the `userinfo.xml.db` database:

```
import sqlite3
import base64

conn = sqlite3.connect('userinfo.xml.db')
c = conn.cursor()
c.execute('SELECT key, value FROM profiles')
key_tuples = c.fetchall()
with open('userinfo_out.txt', 'w') as f:
  for key, value in key_tuples:
    if value == None:
      value = 'Tm9uZQ=='
    f.write(str(base64.b64decode(key)) + '\x09')
    f.write(str(base64.b64decode(value)) + '\r\n')
```

The only difference in the code is that the filenames, table names, and values changed. This time, both the columns in the database are base64 encoded. Again, the code can be run by placing it in the same location as `userinfo.xml.db` and running it using the following command:

```
python userinfo.py
```

Here is the relevant portion of the resulting output file, showing the personal data the user used to register the account:

```
b'countrycode'  b'1'
b'countrycodename'  b'United States'
b'countryid'    b'1'
b'displayname'  b'None'
b'isocountrycode'   b'US'
b'middlename'   b'None'
b'nameprefix'   b'None'
b'namesuffix'   b'None'
b'phonenumber'  b'(321) 867-5309'
b'user_countrycode_based_on_which_contacts_are_filtered_last_time'  b'1'
b'email'    b'throwaway8675309@gmail.com'
b'firstname'    b'John'
b'lastname' b'Glenn'
```

Further down in the output, there is also a list of all of the user's contacts who use Tango. The output also includes the contacts' names and phone numbers.

WhatsApp analysis

WhatsApp is a popular chat-/video-messaging service with over 500,000,000 downloads in Google Play.

Package name: `com.whatsapp`

Version: 2.11.498

Files of interest:

- `/files/`
    - `Avatars/`
    - `me`
    - `me.jpeg`
- `/shared_prefs/`
    - `RegisterPhone.xml`
    - `VerifySMS.xml`

- /databases/
    - msgstore.db
    - wa.db
- /sdcard/WhatsApp/
    - Media/
    - Databases/

The /files/avatars directory contains thumbnails of the profile pictures of contacts that use the app, and me.jpg is a full-size version of the user's profile picture. The me file contains the phone number associated with the account

The phone number associated with the account can also be recovered in /shared_prefs/RegisterPhone.xml. The /shared_prefs/VerifySMS.xml file shows the time that the account was verified (in the Linux epoch format, of course), indicating when the user first began using the app.

The msgstore.db database, like it sounds, contains messaging data:

Table	Description
chat_list	The key_remote_jid column shows each account the user has communicated with. The value in the table is the remote user's phone number. For example, if the value is 13218675309@s.whatsapp.net, the remote user's number is 1-321-867-5309.
group_participants	This contains metadata about group chats.
messages	This shows all the message data. Once again, the key_remote_jid field identifies the remote sender. The key_from_me value indicates the direction of the message (0 = received, and 1 = sent). The data column contains the text of messages, and timestamp is the sent or received time in the Linux epoch format. For attachments, media_mime_type identifies the file format. The media_size and media_name columns should be self-explanatory. If the attachment had a caption, the text would be shown in the media_caption column. If the attachment was a location, the latitude and longitude columns would be populated appropriately. The thumb_image column has a lot of useless data in it, but also contains the path of the attachment on the device. The raw_data column contains thumbnails for images and videos.

The `wa.db` database is used to store contact information:

Table	Description
`wa_contacts`	Like other apps, WhatsApp scrapes and stores the user's entire phone book and stores the information in its own database. It contains the contacts' names and phone numbers, as well as their statuses if the contacts are WhatsApp users.

The SD card is a treasure trove of WhatsApp data. The `/sdcard/WhatsApp/Media` folder contains a folder for each type of media (Audio, Calls, Images, Video, and Voice Notes), and stores all attachments of that type in the folder. Sent media is stored in a directory called, unimaginatively, `Sent`. Received media is simply stored in the root of the folder.

The `Databases` directory is an even greater source of information. WhatsApp makes a backup of `msgstore.db` nightly and stores the backups here. This allows an examiner to see historical data that may have been deleted. If I delete a chat today, but you look at a backup from yesterday, you would be able to access the data I deleted. The app is even kind enough to put the date in the filename, for example, `msgstore-2015-01-21.1.db`. The only catch is that these backups are encrypted!

Decrypting WhatsApp backups

Luckily, there is a tool available to decrypt the backups. It can be found here, along with detailed installation instructions at `http://forum.xda-developers.com/showthread.php?t=1583021`. Unfortunately, it hasn't been updated in some time and doesn't seem to work on newer versions of the app.

We are not aware of an easy, automated extraction tool for newer versions of WhatsApp. However, with the version of WhatsApp we tested, we had great success using the instructions at `http://forum.xda-developers.com/android/apps-games/how-to-decode-whatsapp-crypt8-db-files-t2975313`. Note that this must be done on a Linux computer. Once the steps have been successfully followed, the result should be a database identical to `msgstore.db` explained earlier.

This is possible because WhatsApp stores the decryption key on the device, in the `/files` directory.



Data storage method 7 is using encrypted files.

Kik analysis

Kik is a messaging app with over 50,000,000 downloads from the Play Store.

Package name: `kik.android`

Version: 7.9.0

Files of interest:

- `/cache/`
  - `chatPicsBig/`
  - `contentpics/`
  - `profPics/`
- `/files/staging/thumbs`
- `/shared_prefs/KikPreferences.xml`
- `/sdcard/Kik/`
- `/databases/kikDatabase.db`

The `chatPicsBig` and `contentpics` directories in `/cache` contain images that were sent and received with the application. The files in `contentpics` contain what appears to be Kik metadata embedded before the image. The `.jpg` has to be carved out of these files. In our testing, all of the files in `contentpics` were also stored in `chatPicsBig`, though this may change with more extensive app usage. The user's profile picture is found in the `/profPics` directory.



Data storage method 8 is using basic steganography, which means, a file is stored within a larger file.

The `/files/staging/thumbs` directory contains thumbnails of images sent and received with the application. Our testing found the same images in this location as the `/cache` directories, but again, it is possible this would vary with more extensive application usage.

The `KikPreferences.xml` file in `/shared_prefs` shows the user's username and e-mail address used with the application. Interestingly, it also contains an unsalted SHA1 hash of the user's password.

The `/sdcard/Kik` directory contains full-sized images that were sent and received in the application. The filenames can be correlated with `messagesTable` in the `kikDatabase.db` database to identify which message contained the image.

The `kikDatabase.db` database contains all of the messaging data from the application stored in the following tables:

Table	Description
`KIKContentTable`	This table contains metadata about sent and received images. Each message is assigned a unique `content_id` value that corresponds to the filenames in the `sdcard/Kik` directory. The preview and icon values for each image correspond to the filenames found at `/files/staging/thumbs`. Each image also contains a `file-URL` value. This is a public URL that can be accessed to view the file.
`KIKcontactsTable`	This table shows the `user_name` and `display_name` values for each contact. The `in_roster` value appears to be set for contacts the user has specifically added (if it is set to `1`). Contacts with an `in_roster` value of `0` appear to be default contacts added automatically. The `jid` column is a unique value for each contact.
`messagesTable`	This table contains all data for messages sent and received with the app. The `body` column shows the text data sent in a message. The `partner_jid` value can be correlated back to the `jid` column in `KIKcontactTable` to identify the remote user. The `was_me` column is used to indicate the direction of the message (`0` = sent, and `1` = received). The `read_state` column shows whether the message has been read or not; `500` = read and `400` = unread. The timestamp, yet again, is in the Linux epoch format. The `content_id` column is populated for message attachments and can be correlated back to `KIKContentTable` for more information.

WeChat analysis

WeChat is a messaging app with over 100,000,000 downloads in the Play Store.

Package name: `com.tencent.mm`

Version: `6.0.2`

Files of interest:



Some of the following paths contain an asterisk (*). This is used to indicate a unique string that will differ for each account. Our device had `7f804fdbf79ba9e34e5359fc5df7f1eb` in place of the asterisk.

- /files/host/*.getdns2
- /shared_prefs/
    - com.tencent.mm_preferences.xml
    - system_config_prefs.xml
- /sdcard/tencent/MicroMsg/
    - diskcache/
    - WeChat/
- /sdcard/tencent/MicroMsg/*/
    - image2/
    - video/
    - voice2/
- /MicroMsg/
    - CompatibleInfo.cfg
    - */EnMicroMsg.db

The *.getdns2 files found in /files/host can be opened as text files or in a hex editor. There is a section called [clientip] that shows the IP address from which the user connected as well as the time of the connection in the Linux epoch format. Our device contained three of these files to show three different connections, though increased application usage may generate more than three of these files.

The com.tencent.mm_preferences.xml file in /shared_prefs records the device's phone number in the login_user_name field. The system_config_prefs.xml file contains the path to the user's profile picture on the device as well as a default_uin value that will be needed later.

The SD card contains a wealth of WeChat data. The /tencent/MicroMsg/diskcache directory contained an image that was never used with the application. We think it was put there when attaching a different image as WeChat loads a view of many images from the device's gallery. The /WeChat directory within /sdcard/tencent/MicroMsg contained images sent from the device.

The /video, /voice, and /voice2 folders within /sdcard/tencent/MicroMsg/* contain exactly what they say: video and voice files sent using the app.

WeChat is fairly unique, in that, it does not utilize a `/databases` directory within the app's directory structure. `MicroMsg` is its equivalent. `CompatibleInfo.cfg` contains the device's IMEI, which will be useful later.

The `*` directory within `/MicroMsg` contains the `EnMicroMsg.db` database. There's only one problem: the database is encrypted using **SQLCipher**! SQLCipher is an open source extension for SQLite that encrypts the entire database. Luckily, like other apps that use encryption that we've seen, the key to decrypting the file is on the device.



Data storage method 9 is using SQLCipher, which involves full-database encryption.

Decrypting the WeChat EnMicroMsg.db database

Fortunately for us, Forensic Focus has an excellent article on doing exactly this at `http://articles.forensicfocus.com/2014/10/01/decrypt-wechat-enmicromsgdb-database/`.

They even provide a Python script to do the work for us at `https://gist.github.com/fauzimd/8cb0ca85ecaa923df828/download#`.

To run the Python script, simply put the EnMicroMsg.db file and the system_config_prefs.xml files in the same directory as the script and, in the command-line, type:

```
python fmd_wechatdecipher.py
```

The script will then prompt you for the **International Mobile Station Equipment Identity** (**IMEI**) of the device. This can be found in the `/MicroMsg/CompatibleInfo.cfg` file, printed somewhere on the device (behind the battery, on the SIM card tray or etched on the back of the device), or by typing `*#06#` in the phone dialer.

The script should run. Place a file called `EnMicroMsg-decrypted.db` in the directory.

Finally, we can now examine the `EnMicroMsg-decrypted.db` database with respect to the following tables stored in it:

Table	Description
`ImgInfo2`	This contains path information for sent and received images. The `bigImgPath` column contains the filename for the image. This can be searched on the SD card to find the picture. Alternatively, images are stored in the `/sdcard/tencent/MicroMsg/*/image2` directory in folders that correspond to the filename. For example, the file named `3b9edb119e04869ecd7d1b21a10aa59f.jpg` can be found in the `image2` directory in the `/3b/9e` path. The folders are broken down by the first 2 bytes of the name and then by the second 2 bytes of the name. The `thumbImgPath` column contains the name of thumbnails for the images.
`message`	This contains all the message information for the app. The `isSend` column indicates the message direction (`0` = received, `1` = sent). The `createTime` table is the timestamp of the message, in the Linux epoch format. The `talker` column contains a unique ID for the remote user. This can be correlated with the `rcontact` table to identify the remote user. The `content` column shows the data of messages sent as text and identifies video calls as "voip_content_voice". The `imgPath` column contains the path to image thumbnails, which can be correlated with the `ImgInfo2` table to locate the full-sized images. It also includes filenames for audio files, which can be searched for or located in the `/sdcard/tencent/MicroMsg/*/voice2` directory.
`rcontact`	This contains a list of contacts and includes many that are added by default by the app. The `username` column can be correlated with the `talker` column in the `message` table. The `nickname` column shows the user's name. The `type` column is an indicator of whether the contact was added manually or automatically (`1` = device user, `3` = added by user, and `33` = added by app). The exception to this is the user "weixin", which is automatically added, but has a type value of `3`.
`userinfo`	This table contains information about the user, including their name and phone number.

Application reverse engineering

The vast majority of Android applications are written in Java. In order to truly reverse engineer Java code, one should generally be able to engineer Java code first. Teaching Java is well beyond the scope of this book. We will, however, show a few useful reversing methods that we think will be useful and can be done by an average mobile forensic examiner. Many hundreds of tutorials and guides have been written online for Android reversing, from the very basic to the highly advance.

Anyone looking for more information on the subject should easily be able to find what they are looking for. As always, `www.xda-developers.com` is an incredibly useful resource, and entire books have been dedicated to the subject. There is also an incredibly detailed, updated list of tools by Ashish Bhatia that can be found at `https://github.com/ashishb/android-security-awesome`.

Obtaining the application's APK file

Applications are installed via `.apk` files. The APK file for an app is stored on the device, even after the application is installed (and is removed when an app is deleted). This APK contains the compiled Java code for the app, the icons and fonts used in the app, and an `AndroidManifest.xml` file that declares the permissions the application needs.

The `APK` file for applications that are installed through Google Play can be found in the `/data/app` directory. Another method to find the APK location is to use the `adb` shell `pm path <package_name>` command. The APK file for preinstalled system applications (that cannot be deleted without root) can be found in the /system/app directory. The APK file itself is stored in a directory named after its package name, followed by a dash and a number. For example, the package name for Kik is `kik.android`, and the APK in `/data/app` is stored as `inkik.android-1`.

Here is the list of APK directories in `/data/app` for the device we tested:

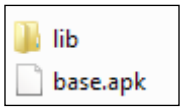


Note that every application we tested has an APK file in this directory, as well as many apps that we did not look at.

Obtaining the APK file is as simple as using the adb pull command. To pull the Kik APK, we will use the following command:

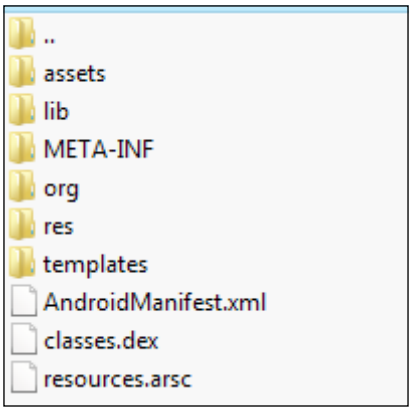
```
adb pull /data/app/kik.android-1
```

This should pull a `lib` directory and a `base.apk` file, which will be in the current directory the command was run from:



Disassembling an APK file

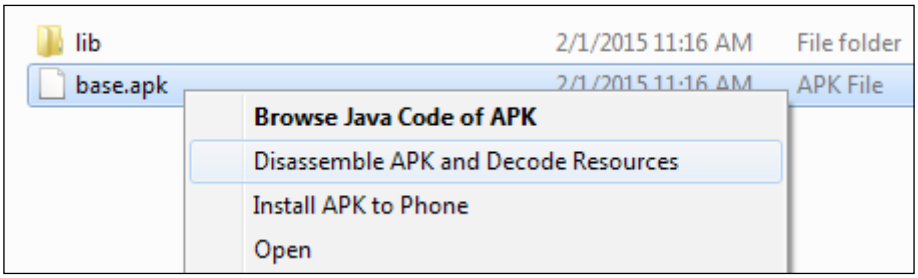
For starters, the APK file is actually just a ZIP compressed file. Renaming the extension to .zip will allow an examiner to open the container and browse the files contained in it:



However, you might not be able to view the AndroidManifest.xml file. There are many tools and methods to fully disassemble the APK, and these can be found in the list we linked to above. Our personal favorite tool, though, is one that allows you to simply right-click on the APK and disassemble it (on Windows only). The APK_OneClick tool can be found at `http://forum.xda-developers.com/showthread.php?t=873466`.

The **Java Runtime Environment** (**JRE**) will have to be installed. It can be found at `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`.

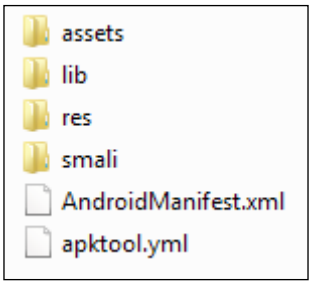
Once the tool and the JRE have been installed, an examiner can simply right-click on the APK and select **Disassemble APK and Decode Resources**:



A pop-up window will appear to show the progress and will disappear if no problems are encountered:

```
I: Baksmaling...
I: Loading resource table...
I: Loaded.
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: C:\Users\Android\apktool\framework\1.apk
I: Loaded.
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Done.
I: Copying assets and libs...
```

If the disassembly ended successfully, there will now be a folder called `base-disasm` in the same directory as the APK. Browsing the directory will show many of the same files and folders we saw when the APK was renamed to a `.zip` file:



Determining an application's permissions

Knowing what an app has permission for can be very useful for an examiner. For starters, it can help narrow down where data is stored. An app without permission to write data to the SD card, for example, won't store any data there. One of the most commonly heard defenses when a suspect is caught with illicit material is that, of course, the suspect has no idea how it got there and it was placed there by a virus. If he says a particular app put that data on his SD card, an examiner can show that the app couldn't have done that because it didn't have permission to write to the SD card. These are just a few basic examples, but again, this is very basic reverse engineering!

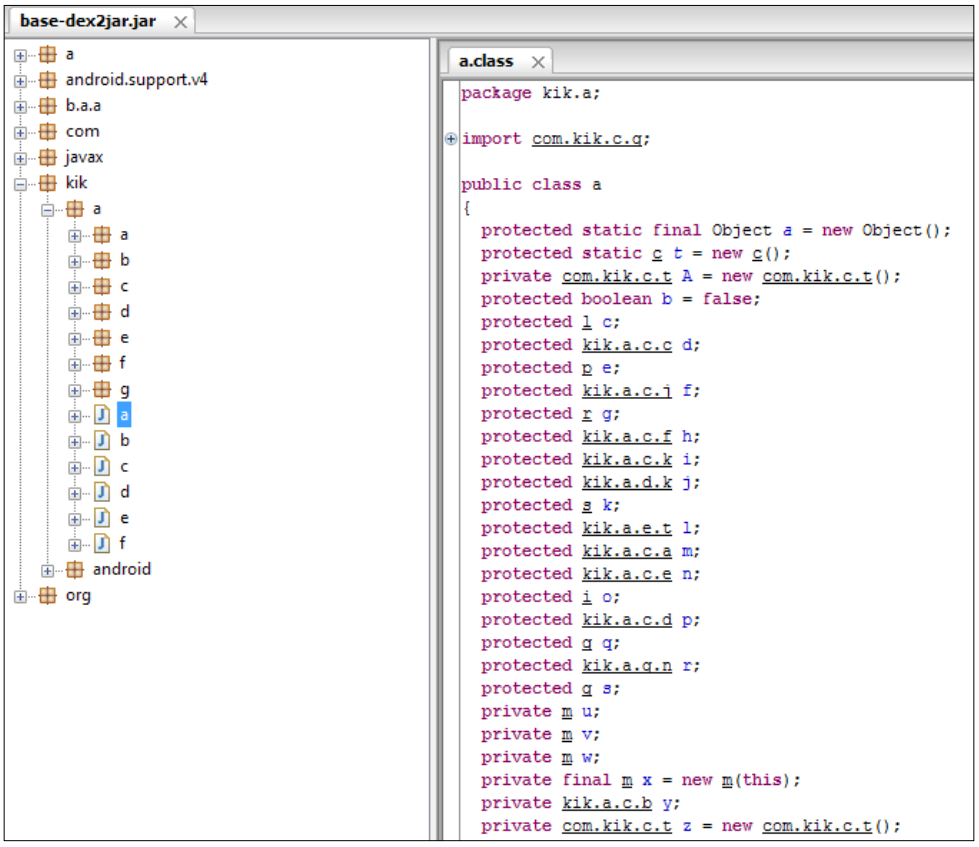
The `AndroidManifest.xml` file from the disassembled APK discussed earlier will contain the app's permissions. These are the equivalent of what the user is shown and has to approve when the app is installed:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.VIBRATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.GET_ACCOUNTS" />
<uses-permission android:name="kik.android.permission.CONTACT" />
<uses-permission android:name="com.android.vending.BILLING" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.CAMERA" />
```

For the specifics of what each permission allows the app to do, Google maintains a list at `http://developer.android.com/reference/android/Manifest.permission.html`.

Viewing the application's code

To view the application's code using the APK_OneClick tool, simply right-click on the APK and select **Browse Java Code of APK**. Again, a window will pop up temporarily showing progress and will disappear if no errors are encountered. Once it completes, a Java Decompiler window will appear, allowing the examiner to browse through the Java code as follows:



Summary

This chapter has been an in-depth study of specific Android applications, and how/where they store their data. We looked at 19 specific applications and discovered 9 different methods of storing and obfuscating data. Knowing that applications store their data in a variety of ways should help an examiner have a better understanding of an app's data that they are examining. This knowledge should, hopefully, push them to look harder when they can't find data they expect the app to have. An examiner has to be able to adapt to the changing world of application analysis. As applications constantly update, an examiner has to be able to update their own methods and abilities in order to keep up.

The next chapter will take a look at several free and open source tools to image and analyze Android devices and reverse engineer applications to discover where their data is stored.

8
Android Forensic Tools Overview

This chapter is an overview of the free and open source Android forensic tools and will show you how to use these tools for common investigative scenarios. By the end of this chapter, the reader should be familiar with the following tools:

- ViaExtract
- Autopsy
- ViaLab

ViaExtract

ViaExtract is a logical and physical extraction tool created by NowSecure (formerly known as ViaForensics). Logical acquisitions (including backups) are available with the free version, while the paid version adds physical extractions. It is freely distributed inside of a virtual machine file (either VMWare or Virtual Box formats) running NowSecure's Santoku Linux distribution. An active Internet connection is required while using the free version. The download and full feature list can be found at `https://www.nowsecure.com/forensics/community/`. Registration is required.

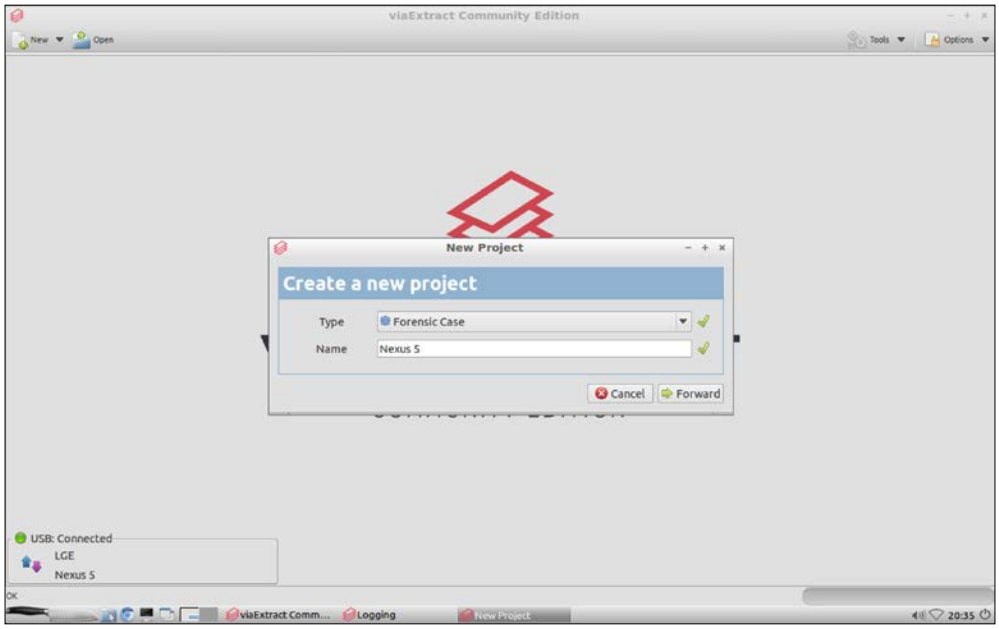
The icons to register the tool and launch ViaExtract can be found on the desktop of the ViaExtract virtual machine:



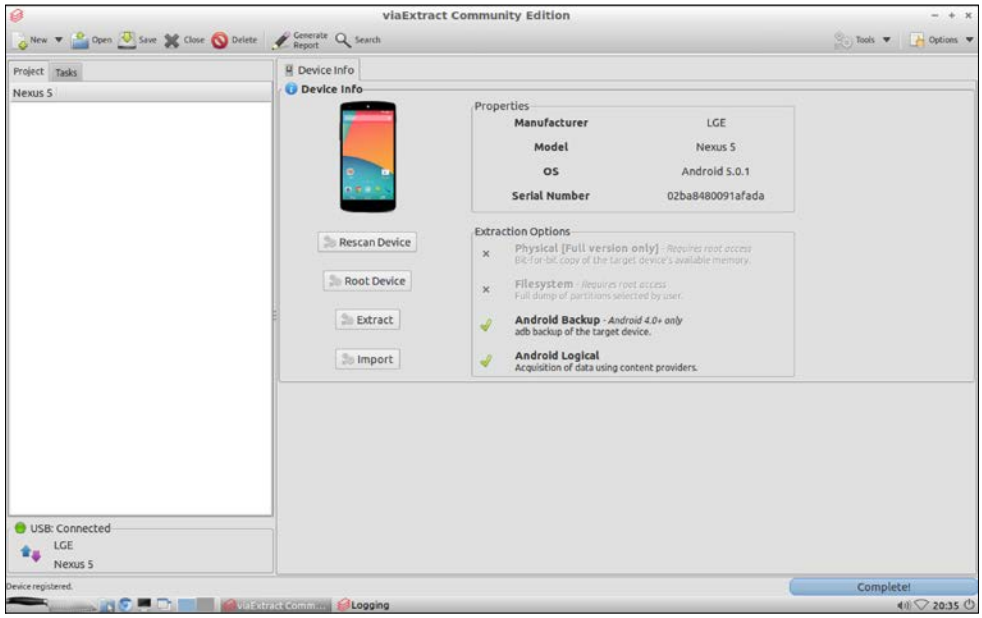
Before launching ViaExtract, ensure that the device to be examined is connected to the computer via a USB. This will ensure that the device is detected. The device will also need to be powered on. Note the appropriate network-isolation measures as discussed in *Chapter 1*, *Introducing Android Forensics*. In the following example, we will examine an LG Nexus 5 running Android Lollipop 5.1. Note that this is shown in the bottom-left corner of the following screenshot.

Then, follow these steps:

1. Clicking on the **New** button in the upper-left corner will bring up the **Create a new project** dialog box:



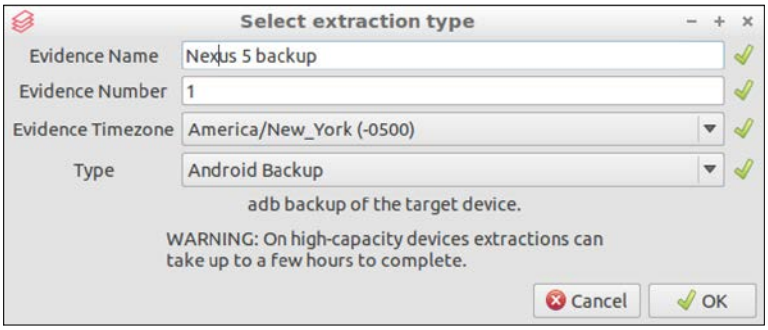
2. Choosing **Forward** will bring up the **Device Info** tab, which is where we will begin our extractions. The list of supported extractions is shown in the following screenshot. In this case, our two options are **Android Backup** or **Android Logical**:



Backup extraction with ViaExtract

To perform backup extraction, follow these steps:

1. Clicking on **Extract** (as seen in the preceding screenshot) will show the **Select extraction type** dialog box. We choose to do a backup extraction first from the **Type** drop-down menu:



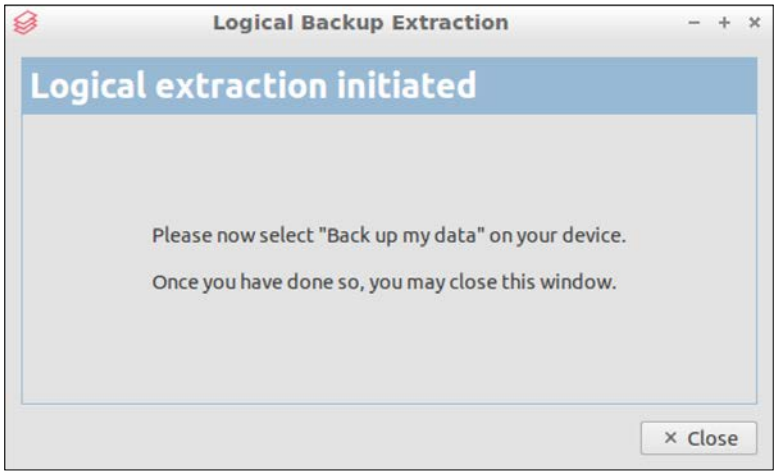
2. Once all of the fields are filled in, the **OK** button will become available. Selecting **OK** will display instructions to accept the process on the device. However, this step will not be available on the device yet:



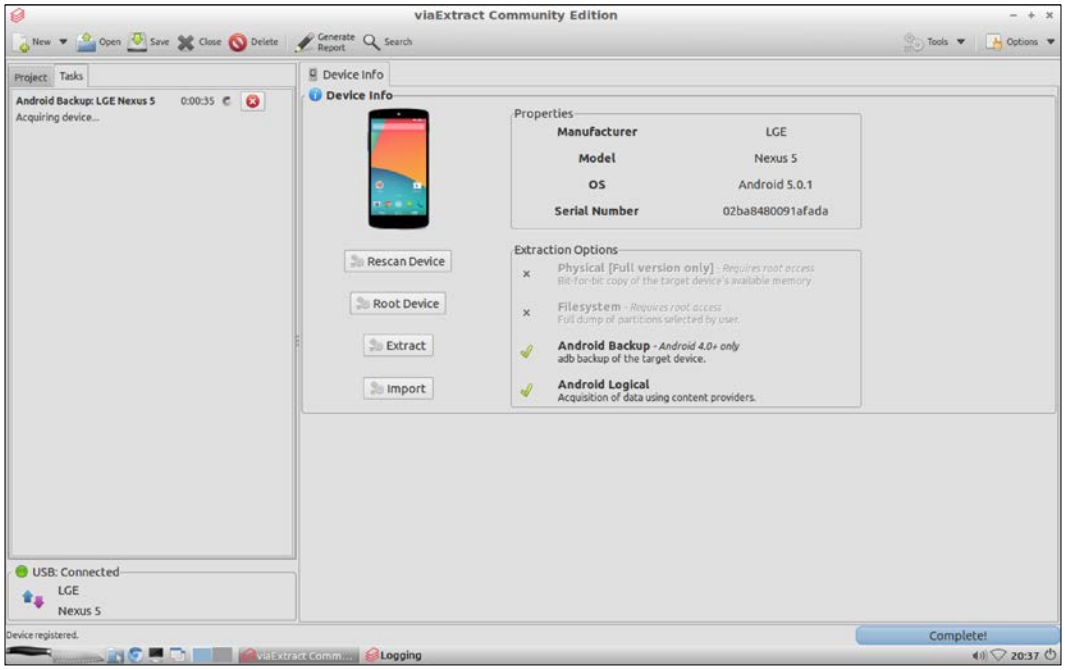
3. After choosing **Forward** on the **Android Backup: Allow backup** screen, ViaExtract will give you the option to attempt the recovery of deleted SQLite data. Though this is only a logical extraction, it is possible, as discussed in detail previously in this book, because of the way SQLite databases store their data.



4. Choosing **Forward on the Android Backup: SQLite recovery** screen will begin the backup. At this point, the backup will have to be accepted on the device, as shown in the preceding **Android:Backup Allow backup** screenshot:



5. After clicking on **Close**, the backup should now be seen progressing in the **Tasks** tab in the upper-left corner as shown in the following screenshot:



Logical extraction with ViaExtract

To begin a logical extraction, follow these steps:

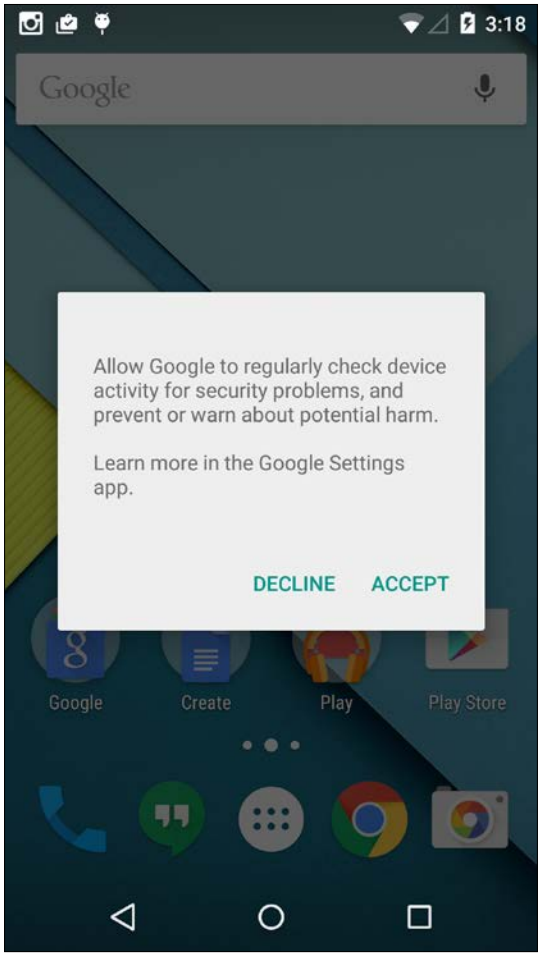
1. Select **Extract** from the **Device Info** tab, just as with the backup extraction we completed earlier. This time, however, choose **Android Logical** from the **Type** drop-down menu. This will launch the logical extraction wizard, which begins by noting that it will require us to install an application on the device:



2. Choosing **Forward** will advance to the **Logical extraction: Options** menu. Here, certain file types can be ignored to speed up the extraction process:



3. At this point, the ViaExtract application will be pushed to the device and further action may be required on the device. We will choose **DECLINE** on the following pop-up message:



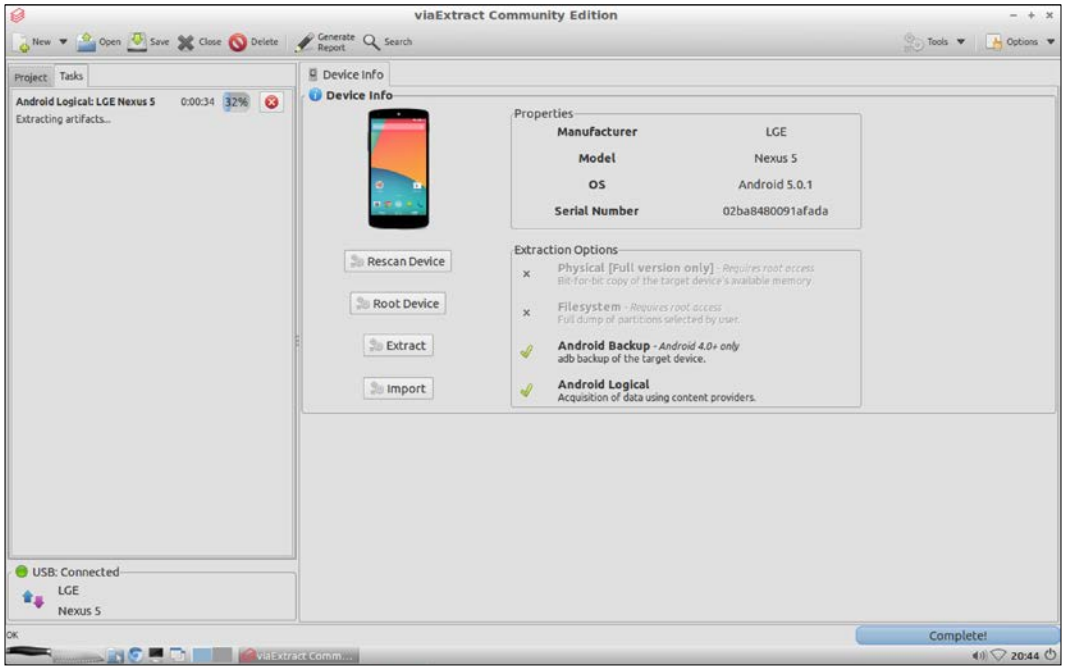
4. After declining the popup, the application can be seen running on the device:



5. Back on the computer, the following message will be displayed:

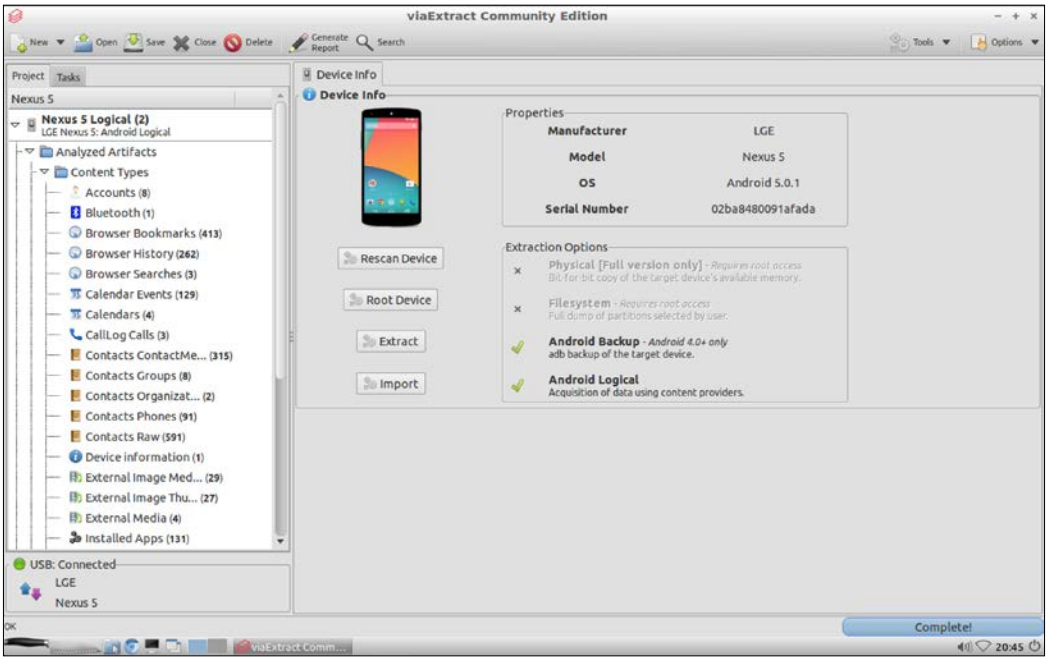


6. Once again, the progress will be visible in the **Tasks** tab in the upper-left corner, shown as follows:

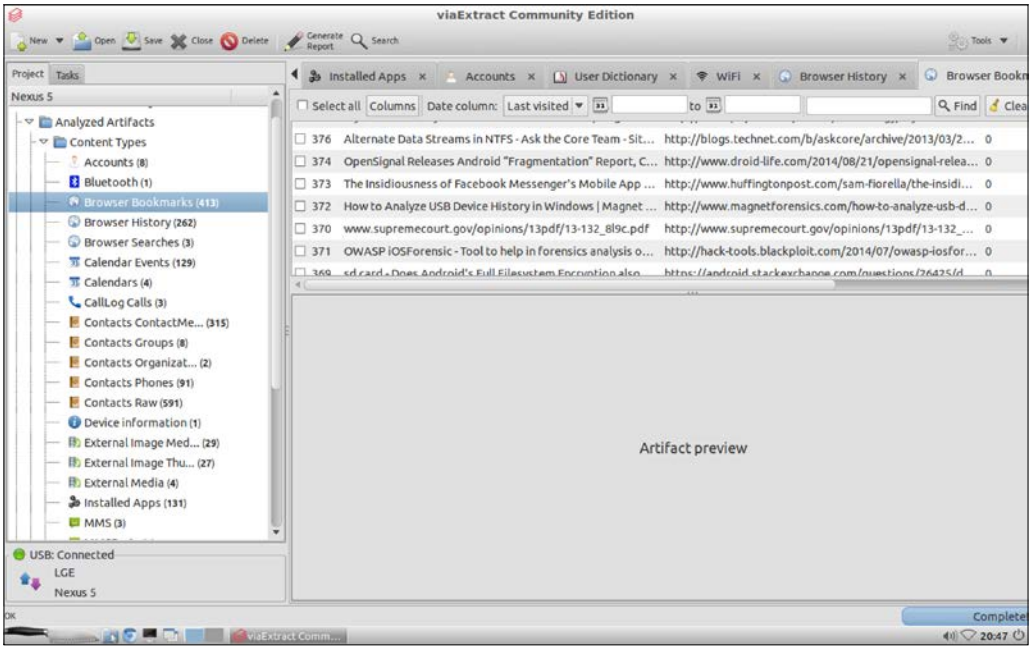


Examining data in ViaExtract

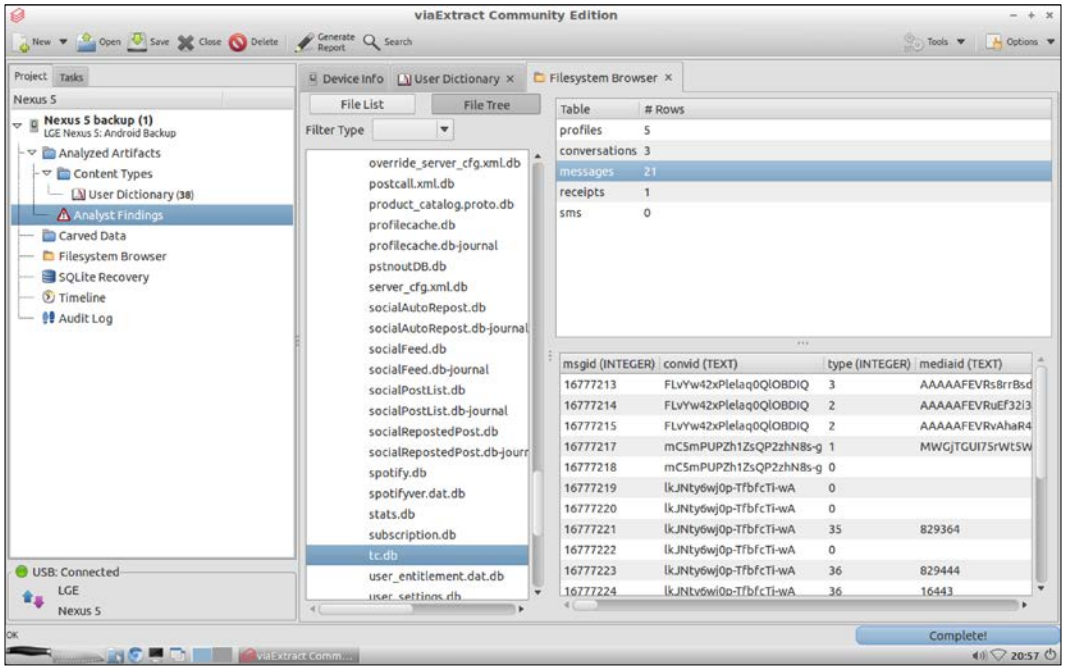
Once an extraction is complete, the data can be viewed in the **Project** tab in the upper-left corner:



To view the content, simply click on it, and it will be displayed on the right-hand side. Here is an example of bookmarks found in a logical extraction:



When examining a backup, the process is the same. Here is an excerpt of the `Tango tc.db` file analyzed in *Chapter 7*, *Forensic Analysis of Android Applications*:



Other tools within ViaExtract

ViaExtract can also attempt to root a device and bypass the passcode. Clicking on the **Root** button from the **Device Info** tab will launch the rooting wizard. Simply follow the pop-up messages to root the device. We did not have any success with this on our test devices, Nexus 5, Moto X (2013 model), and an HTC Droid DNA.

The lock screen bypass wizard is launched from the Tools menu in the upper-right corner. Then, select **Unlock Screen**. This will push an app to the device (which requires USB debugging to be enabled and the RSA authentication to be passed) that will remove the lock screen. This is a useful tool because unlike the manual methods shown in *Chapter 4*, *Extracting Data Logically from Android Devices*, it does not require root access. It was unsuccessful on our Nexus 5 and HTC Droid DNA, but worked perfectly on our Moto X.

Autopsy

Autopsy is a free and open source analysis tool initially developed by Brian Carrier. Autopsy started as a Graphical User Interface for the underlying Linux-based SleuthKit toolset, but the latest release (version 3) is a standalone tool built for Windows. Autopsy can be downloaded at `http://www.sleuthkit.org/autopsy/`.

Autopsy is not intended to perform acquisitions of mobile devices, but can analyze the most common Android filesystems (such as YAFFS and ext). For this example, we will load a full physical image obtained via dd from an HTC Droid DNA, as outlined in *Chapter 5*, *Extracting Data Physically from Android Devices*.

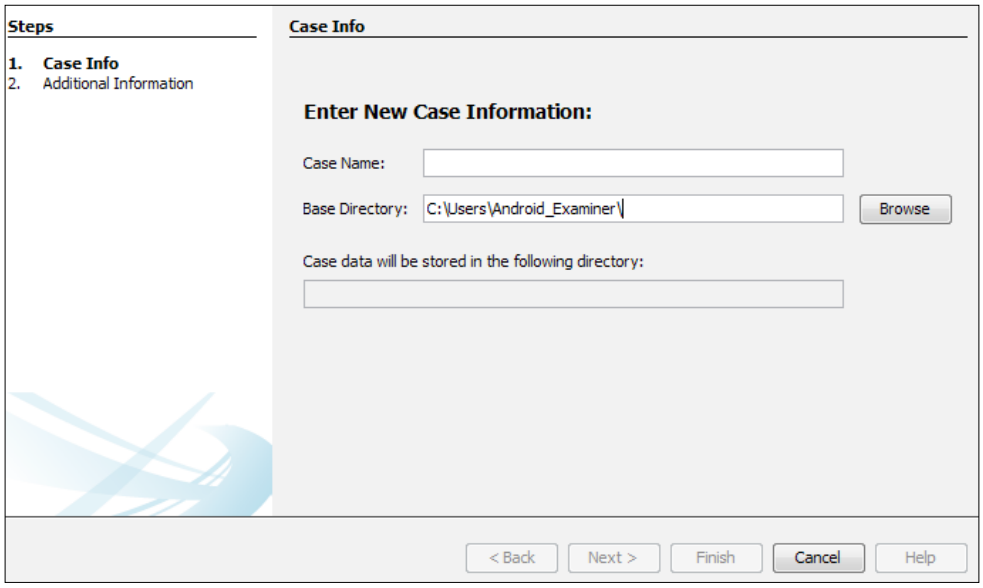
Creating a case in Autopsy

On opening Autopsy, the user will be prompted to choose **Create New Case**, **Open Recent Case**, or **Open Existing Case**:

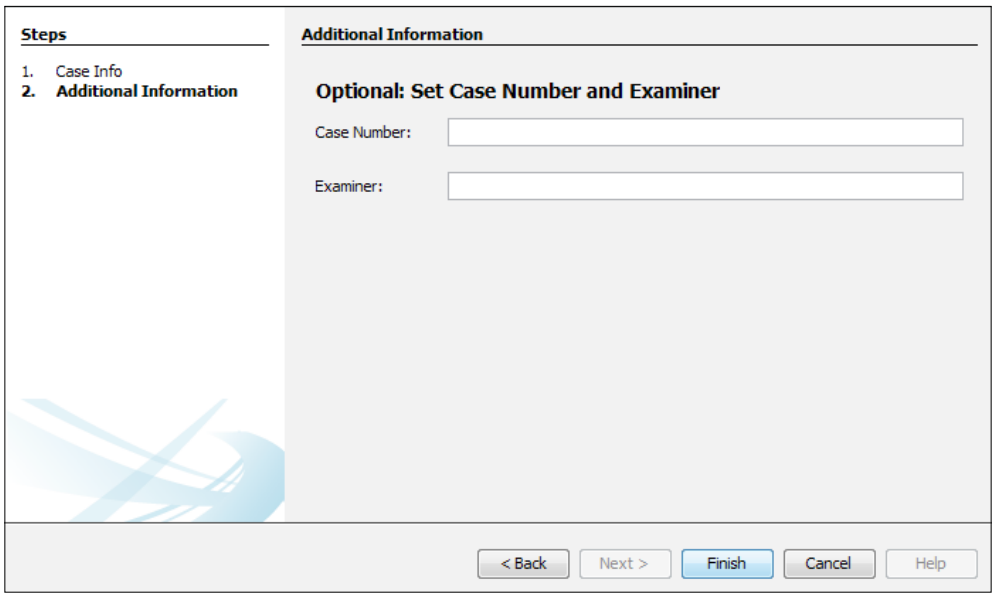


We will create a new case. Follow these steps:

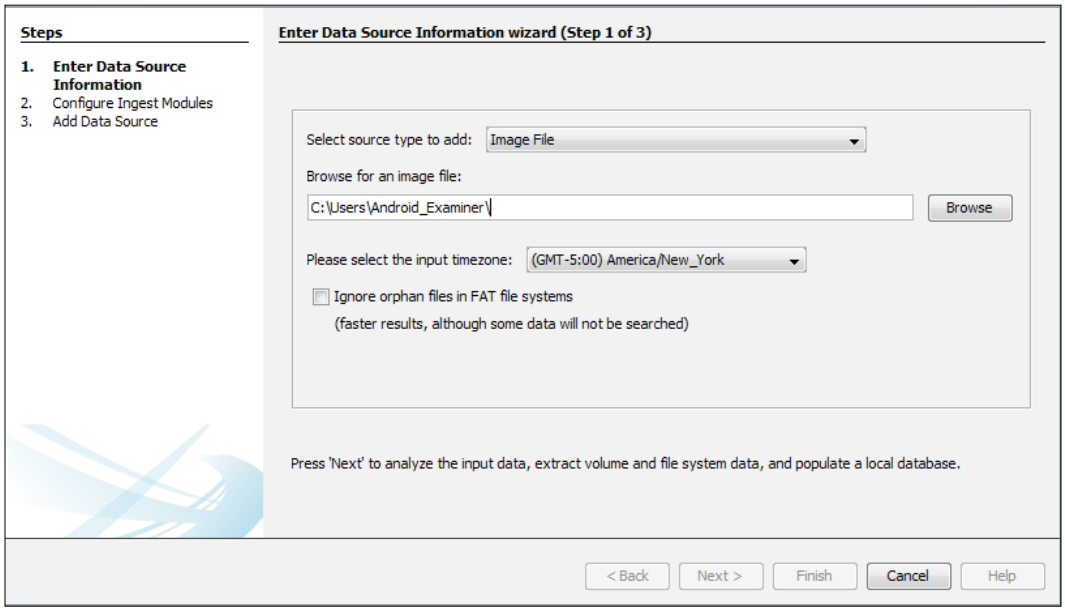
1. After filling in the **Case Name** field, the **Next** button will become available:



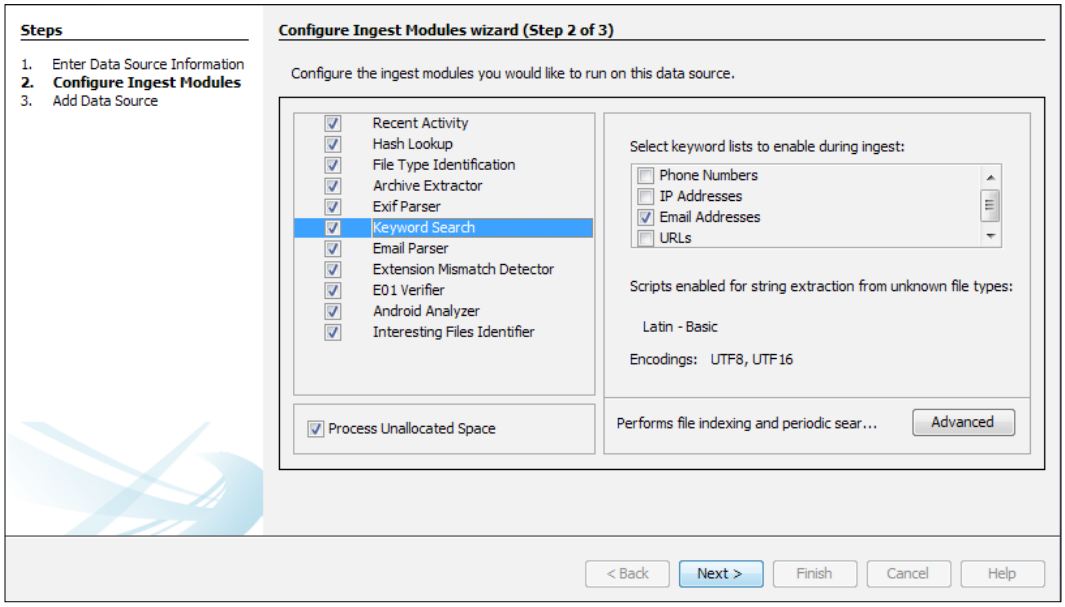
2. On the next screen, an optional **Case Number** and **Examiner** can be entered:



3. Selecting **Finish** will bring up the **Enter Data Source Information** screen. Clicking on **Browse** will allow the user to select an image file to load:



4. After choosing an image file, the **Next** button can be clicked to advance to the **Configure Ingest Modules wizard**:



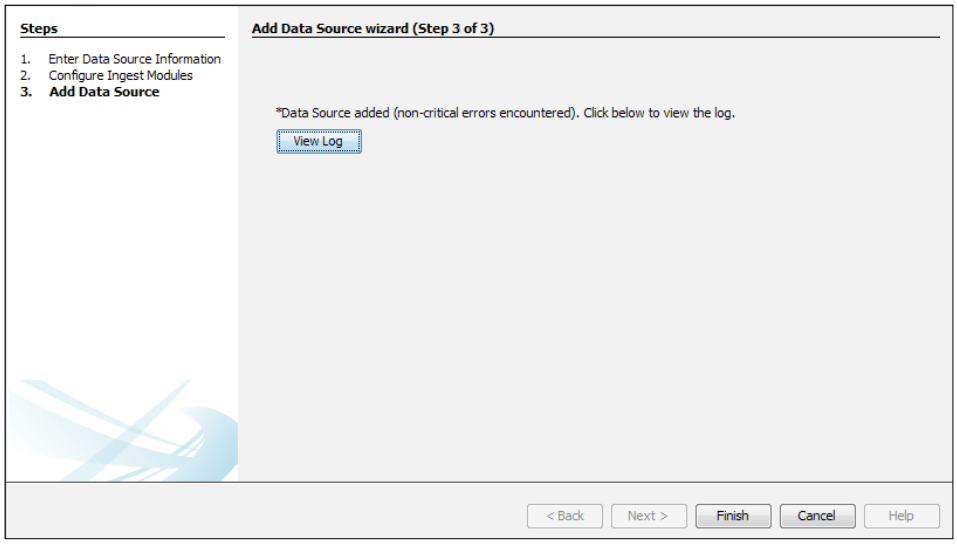
Ingest Modules are tools built into Autopsy that can be run when the case is started or at any point afterwards. The default modules in this version of Autopsy are as follows:

- **Recent Activity**: This extracts recent user activity such as Web browsing, recently used documents, and installed programs.
- **Hash Lookup**: This identifies known and notable files using supplied hash databases, such as a standard NSRL database. It also allows importing custom hash databases.
- **File Type Identification**: This matches file types based on binary signatures.
- **Archive Extractor**: This extracts archive files (`.zip`, `.rar`, `.arj`, `.7z`, `.gzip`, `.bzip2`, `.tar`). It automatically extracts these file types and puts their contents into the directory tree.
- **EXIF Parser**: This ingests JPEG files and retrieves their EXIF metadata.
- **Keyword Search**: This performs file indexing and periodic search using keywords and regular expressions in lists. It allows loading of custom keywords/lists.
- **Email Parser**: This module detects and parses mbox and pst/ost files and populates e-mail artifacts in the blackboard.
- **Extension Mismatch Detector**: These are flag files that have a non-standard extension based on their file types.
- **E01 Verifier**: This validates the integrity of E01 files.
- **Android Analyzer**: This extracts Android system and third-party app data.
- **Interesting Files Identifier**: This identifies interesting items, as defined by interesting item rule sets.

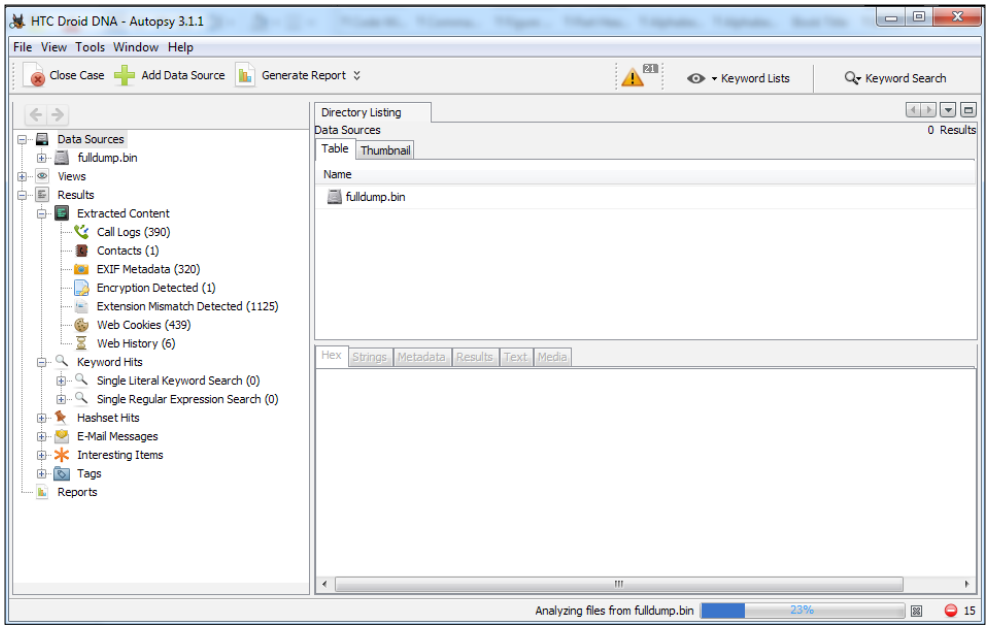


Many of these modules will not be needed for Android devices (E01 Verifier and Email Parser, for example). Only selecting useful modules will speed up the ingest time. Also, note that clicking on a module may bring up more options, as seen in the preceding screenshot.

5. Clicking on **Next** will load the Data Source and begin the Ingest process. Any errors encountered will be noted:

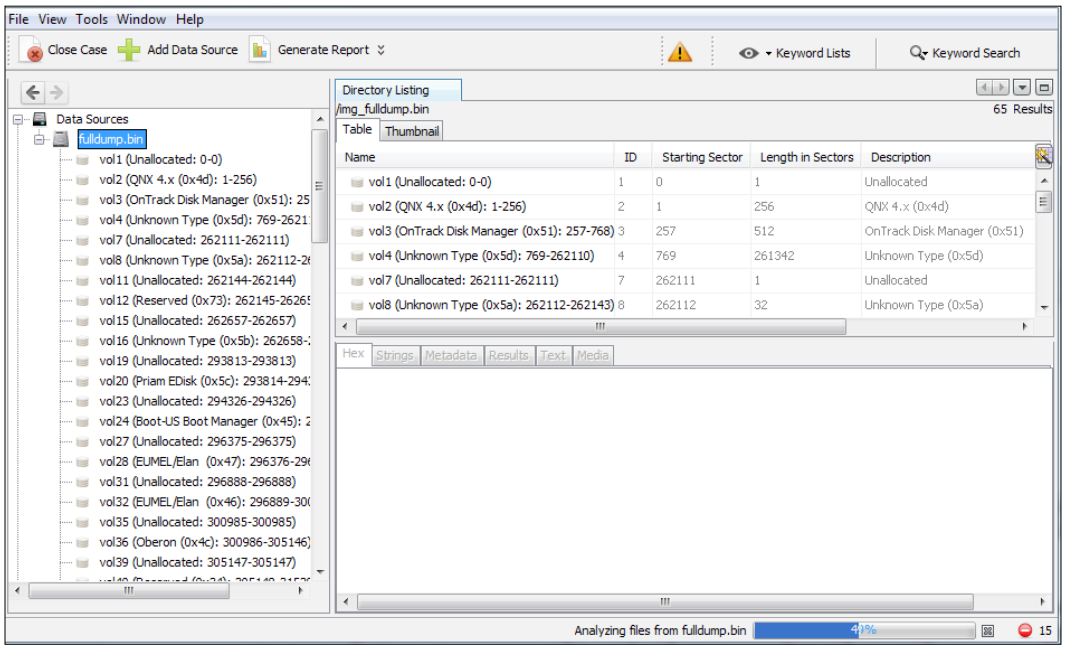


6. Choosing **Finish** will bring the examiner to the main screen for analysis of the ingested case:

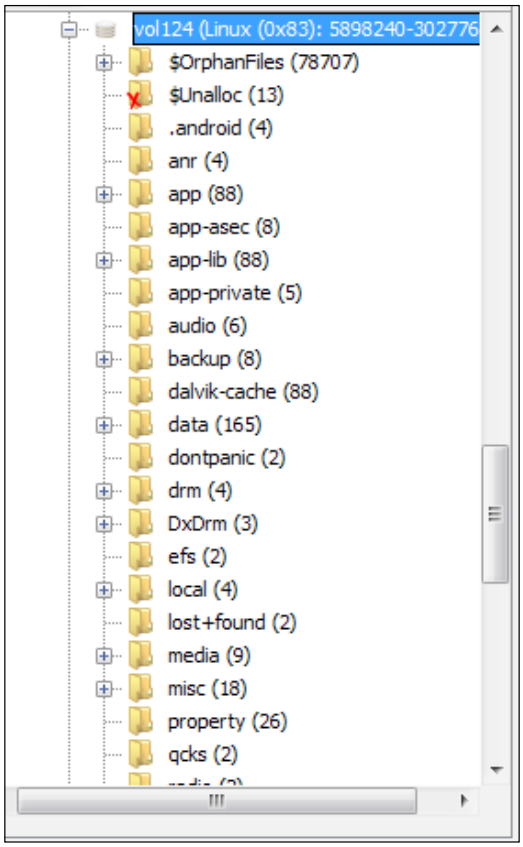


Analyzing data in Autopsy

Even though the case is still being loaded and Ingest Modules are being run (as seen by the progress bar in the bottom-right corner of the previous screenshot), an examiner can begin analyzing the case. Expanding the image file in the upper-left corner will show partitions/volumes identified by Autopsy:

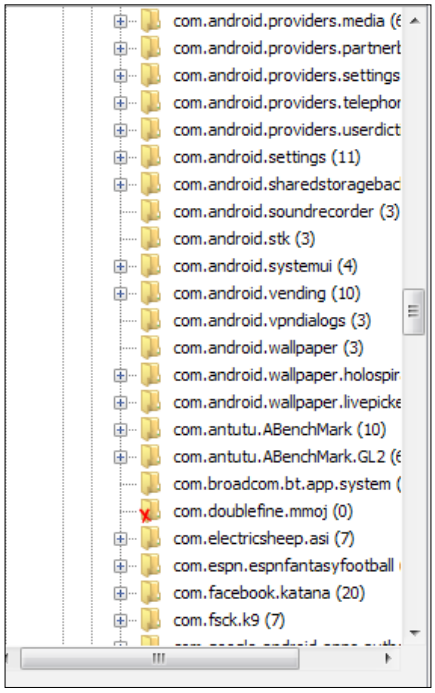


Autopsy identified 65 partitions on our device, the vast majority of which are unallocated. To find the data partition (since we know this is where the vast majority of the data we are interested in is stored), we can simply expand the allocated partitions until we find one that looks like the data partition:

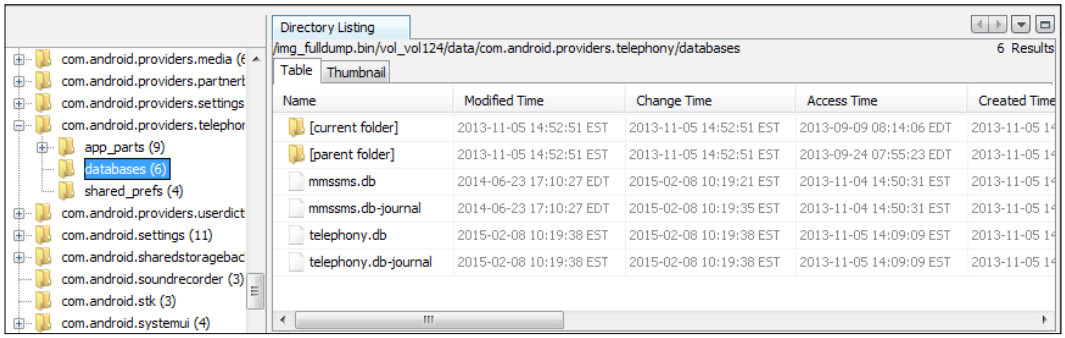


In our image, volume 124 is the data partition. We can see that it has an app directory (where APK files are stored), a data directory (where app data is stored), and a media directory (the symbolically linked location for the SD card).

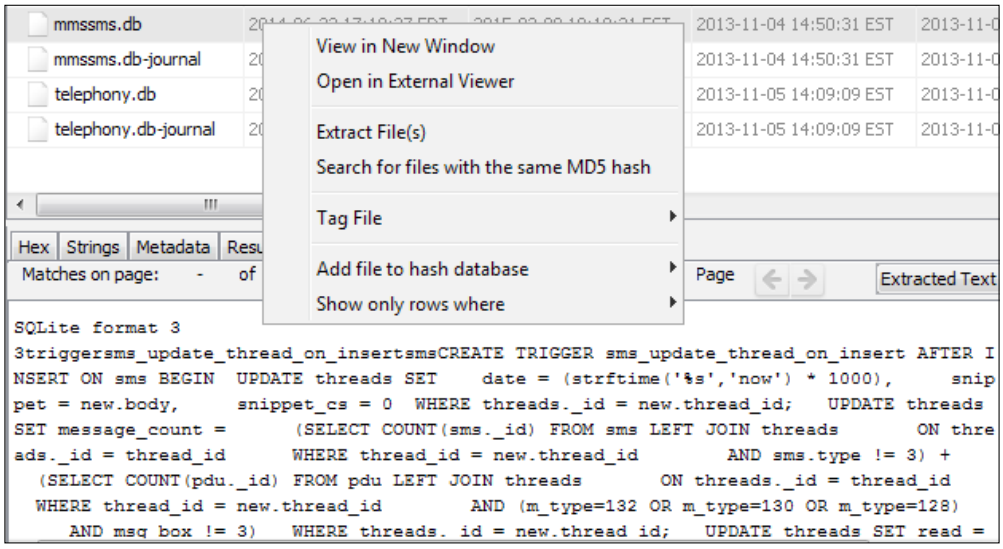
Expanding the data directory will reveal information we should remember from *Chapter 7, Forensic Analysis of Android Applications*. This can also be seen in the following screenshot:



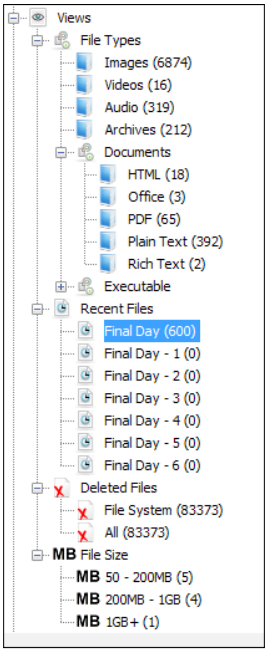
Right away, we can see `com.android.providers.telephony` and `userdictionary` as well as `com.facebook.katana`. How to analyze these applications is covered in *Chapter 7, Forensic Analysis of Android Applications*; this is how to access the relevant files using Autopsy. For example, expanding `com.android.providers.telephony` will show the `mmssms.db` file needed to analyze SMS and MMS data.



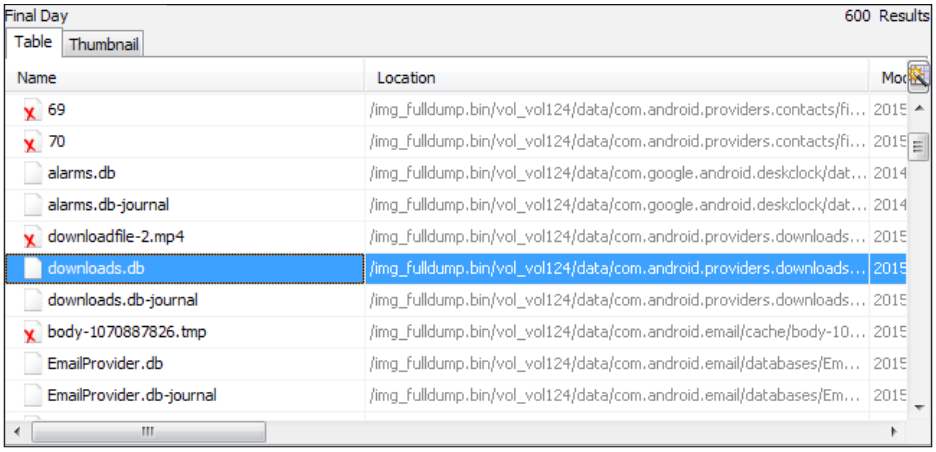
Right-clicking on the file will allow the user to choose **Extract File(s)** or **Open in External Viewer** for further analysis:



Now, let's take a look at the rest of Autopsy's features. Expanding the **Views** section on the left-hand side of the screen will show results from a few of the Ingest Modules used, shown as follows:



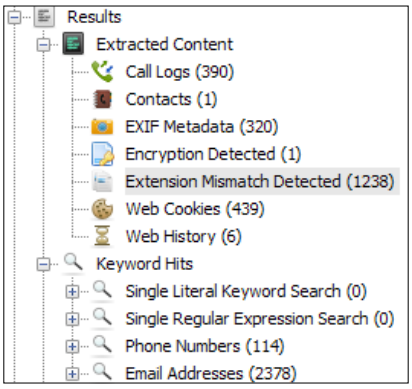
The **File Types** view shows files identified by the `File Type Identification` module. **Recent Files** shows the results from the `Recent Activity` module. In this case, it appears that the device wasn't used for 6 days and then was used on the **Final Day**. Viewing the files identified here can show the user's activity in that time period. Note the red cross, indicating that some of these files were deleted but recovered by Autopsy:



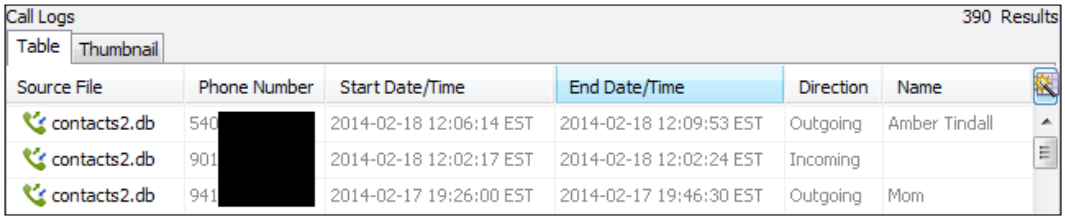
In our case, we can see that the `downloads.db` and `EmailProvider.db` databases were modified. Analyzing these files will show that an e-mail with an attachment was received, and the attachment was then downloaded to the device.

Finally, the **Views** section identifies deleted files (which are very common on mobile devices as a result of wear leveling), as well as large files (which can be useful o quickly find images/video or identify steganography).

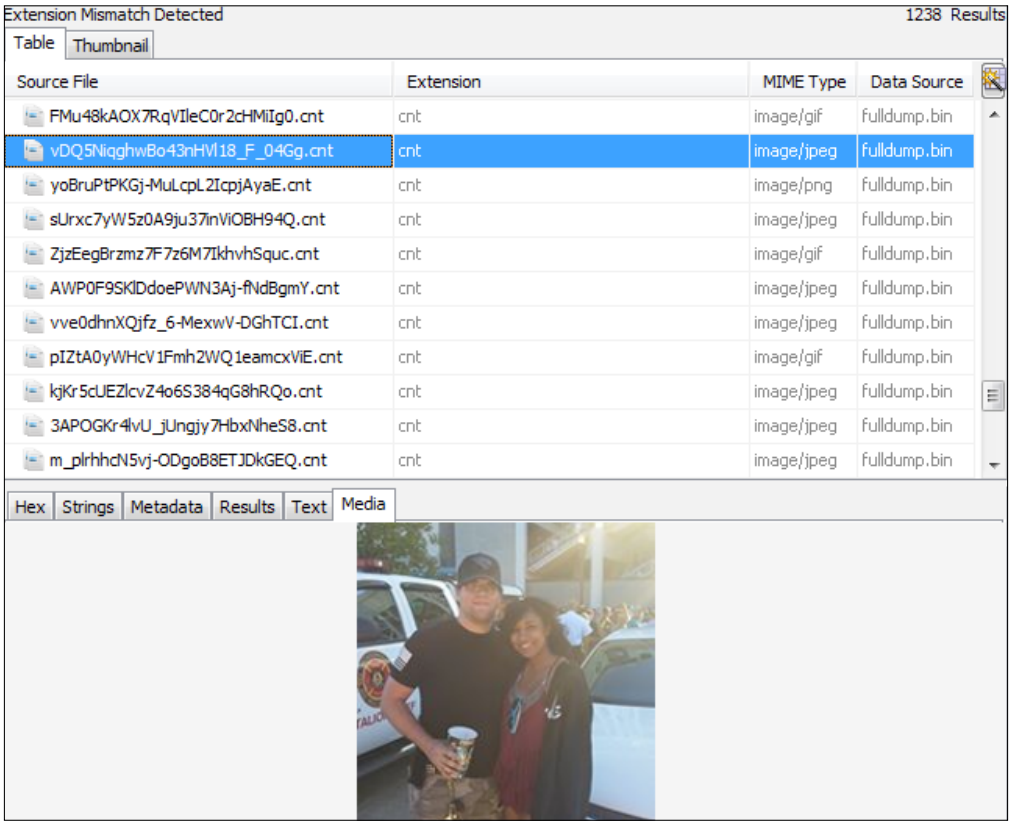
The **Results** section will show the output from the Android Analyzer and Keyword Search modules, as shown in the following screenshot:



The Android Analyzer results seen under **Extracted Content** were mostly as expected. It is worth noting that the **Contacts (1)** section only points to the `contacts.db` file and does not actually parse data from it. For example, **Call Logs** displays data pulled from `contacts2.db`, as described in *Chapter 7*, *Forensic Analysis of Android Applications*:



Extension Mismatch Detected results also show the data we found in *Chapter 7*, *Forensic Analysis of Android Applications*. Several apps were described as having `.cnt` files that were actually JPEG images, and these were appropriately identified by Autopsy, as shown in the following screenshot:



Double-clicking on any of the files seen above will take the user to the location where the file was found in the filesystem.

The **Keyword Hits** section appropriately found many e-mail addresses and phone numbers. However, many of these were found within application files (that is, contact information for the developer of the app) and other places that were not actually stored by the user (this is very common with both mobile and computer forensic tools).

There are many other more advanced features of Autopsy that aren't covered here. To learn more, Basis Technology offers an Autopsy Training course that can be found at `http://www.basistech.com/digital-forensics/autopsy/training/`.

ViaLab Community Edition

ViaLab Community Edition is another free tool developed and released by NowSecure. It is shipped as a standalone virtual machine and can be found at `https://www.nowsecure.com/apptesting/community/` (registration is required). The VM is actually very similar to the Santoku download, which we discussed at the beginning of this chapter, but includes the ViaLab Community Edition tool.



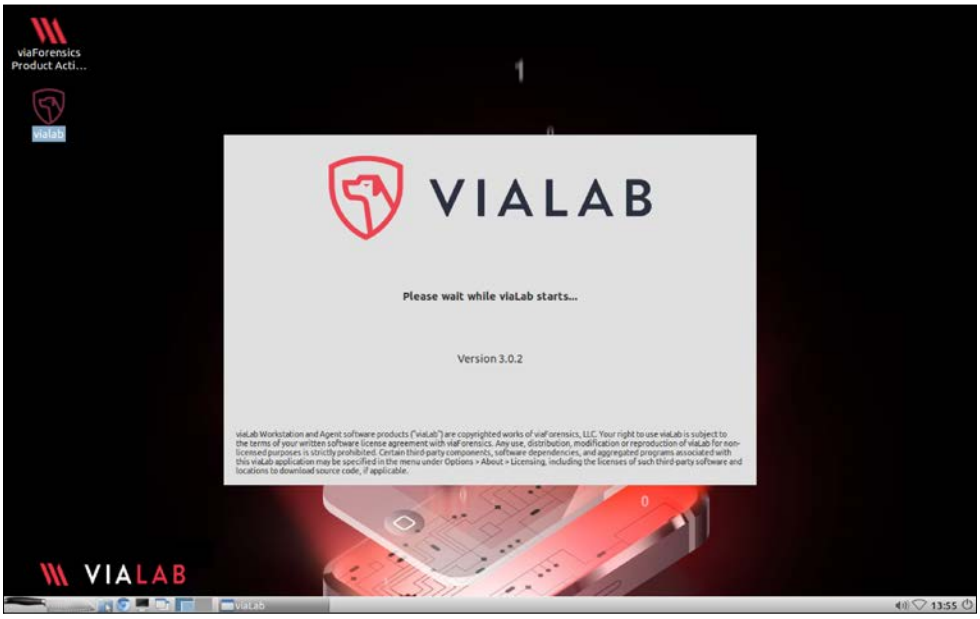
ViaLab requires the examiner's computer to have an Internet connection, in order to use the tool.

The main purpose of ViaLab is to analyze the behavior of an APK, although many of the features to do so are unavailable in the free Community Edition. ViaLab allows you to either manually load an APK file into the Android emulator or run the application on a rooted device. For our example, we manually loaded the APK file for Kik into the Android emulator. We chose Kik because it was analyzed thoroughly in *Chapter 7*, *Forensic Analysis of Android Applications*, so we had a good idea of what to expect and could confirm our previous finding. A good forensic use case for this would be researching an application to learn what data it stores. For example, if a prosecutor is looking for saved videos, an examiner can determine whether the application has that capability and where they are stored.

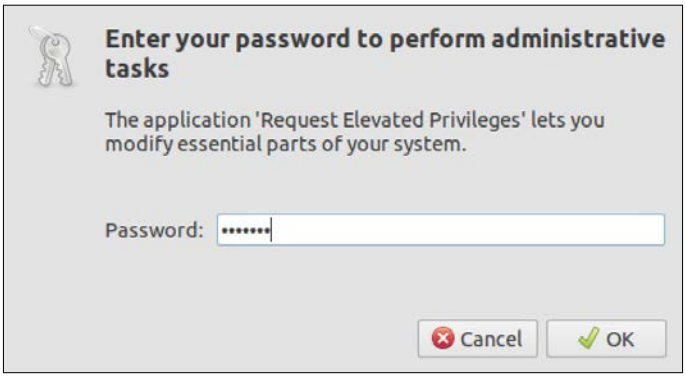
Setting up the emulator in ViaLab

To begin using ViaLab, it must first be activated. This is done through the ViaForensics Product Activation tool found on the desktop. After registering, follow these steps to set up the emulator:

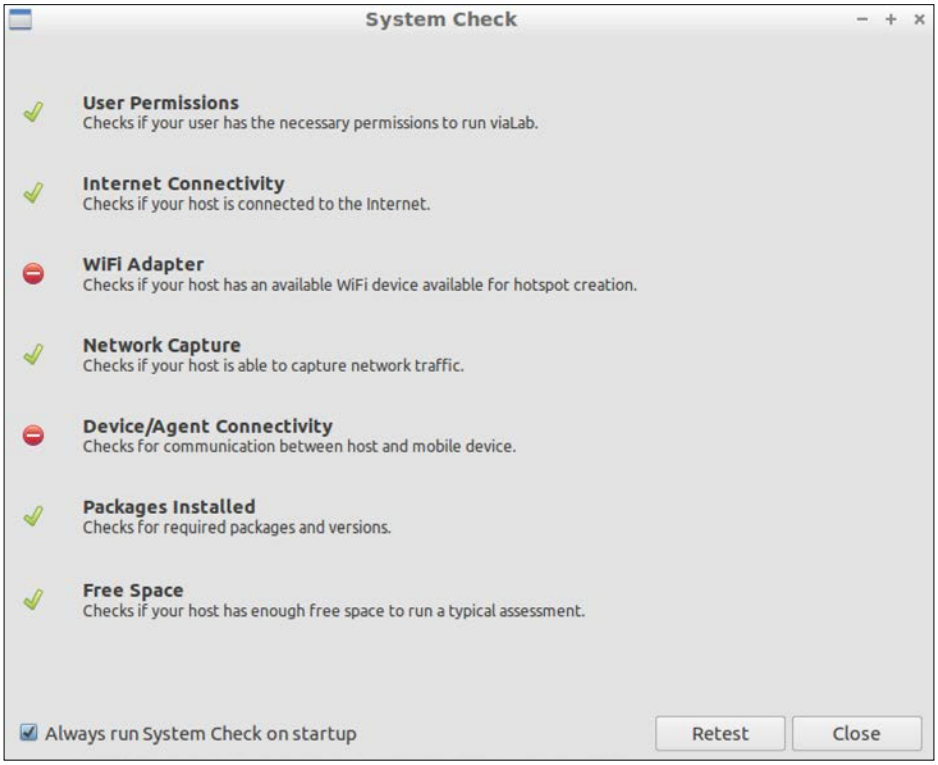
1. Click on the **ViaLab** icon on the desktop to launch the tool:



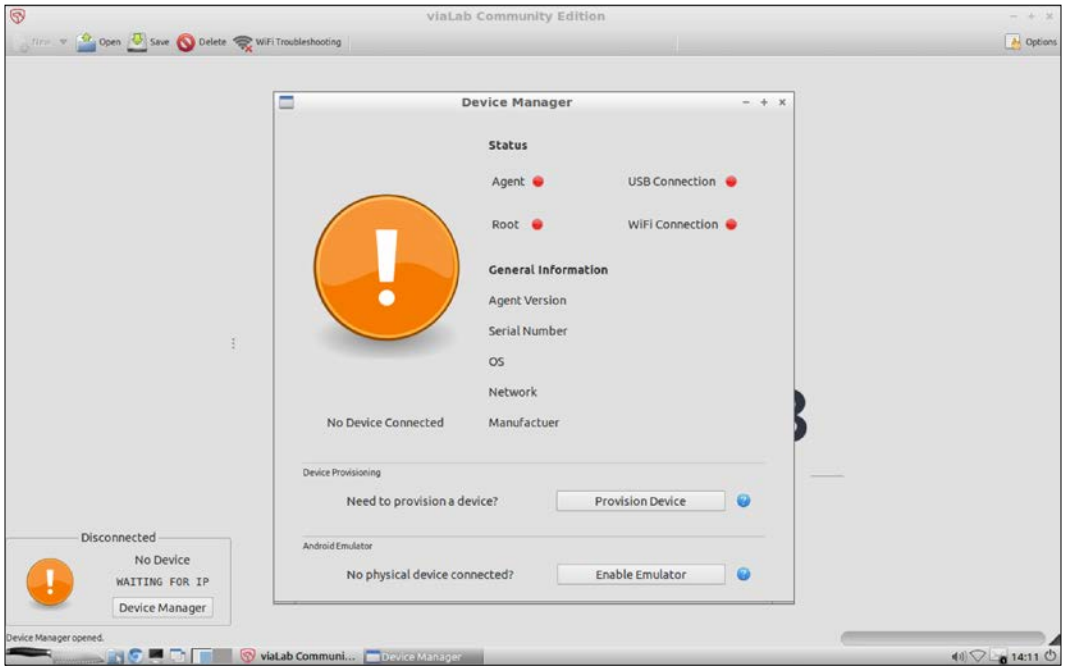
2. The program may prompt for the root password. In the ViaLab VM, the default password is `vialab1`:



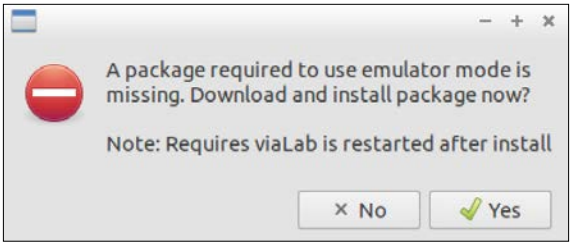
3. After the program launches, a system check will be run. Clicking on **Close** will finish the system check. The **Always run System Check on startup** box can also be deselected to skip this step in the future:



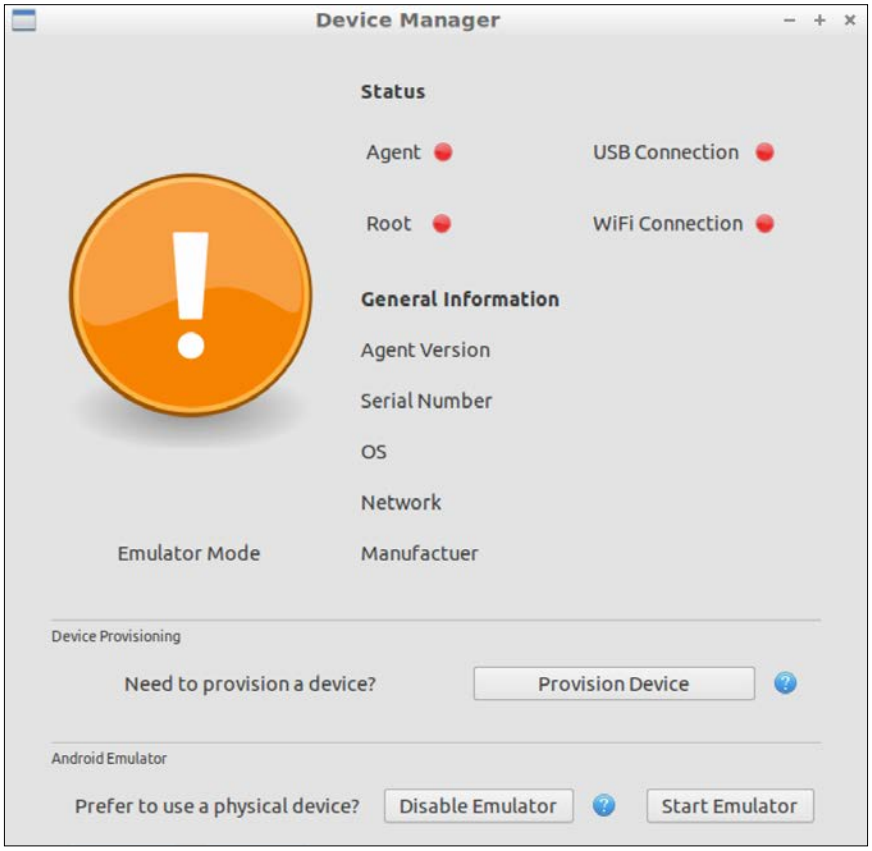
4. To enable the Android emulator (or to configure a real device for ViaLab), select the **Device Manager** icon in the bottom-left corner of the main screen and make the appropriate selection in the window that opens:



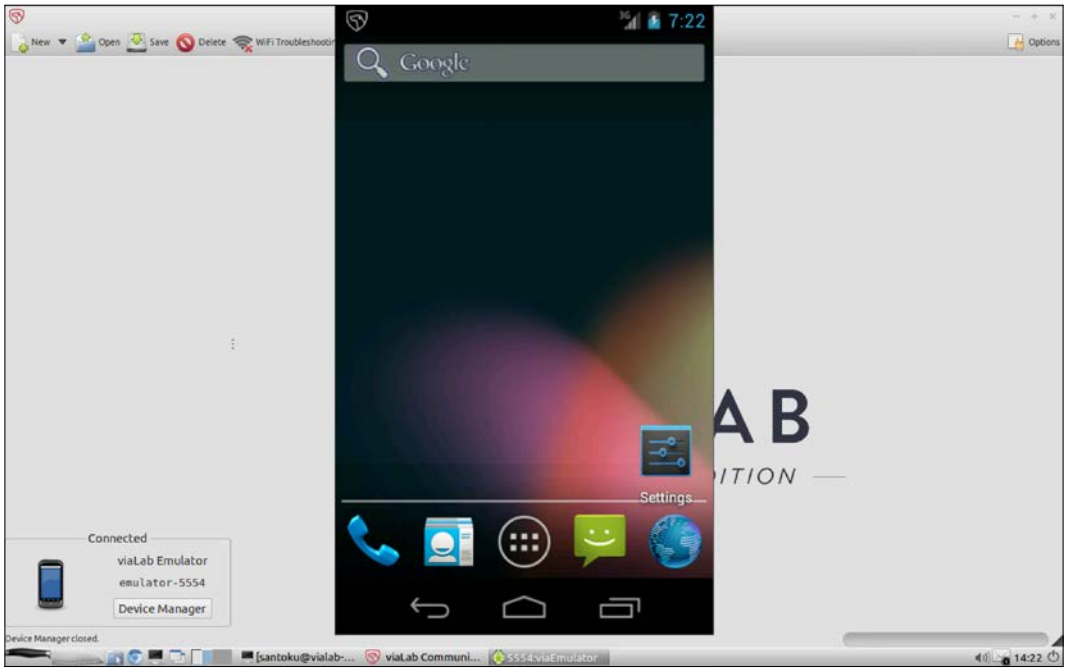
5. The first time the Android emulator is used, additional packages will have to be installed:



6. After the emulator package is installed and ViaLab is restarted, the **Device Manager** will now show a **Start Emulator** option. Choose this to launch the Android emulator:



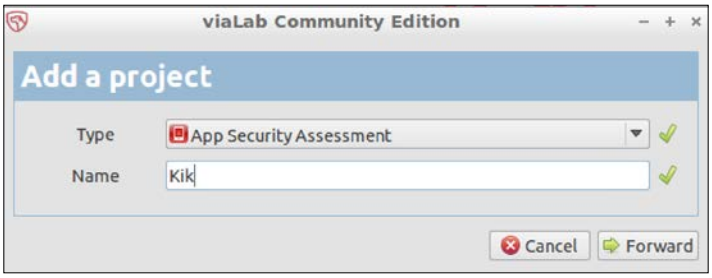
7. The Android emulator will launch and appear in a separate window. The **Device Manager** icon in the bottom-right corner should show that ViaLab is now connected to the emulator:



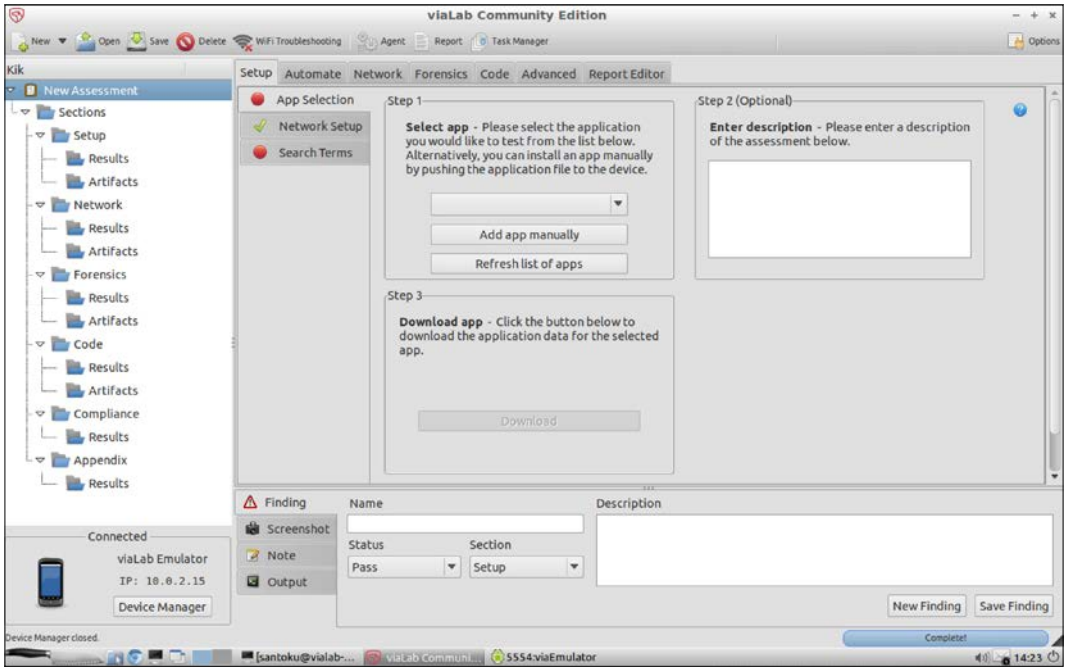
Installing an application on the emulator

To begin the ViaLab analysis, follow these steps:

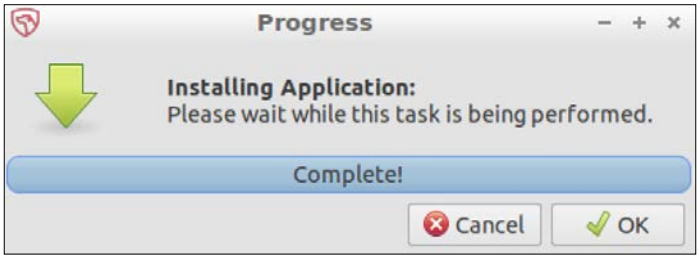
1. Select **New** in the upper-left corner. This will open a window to add a new project:



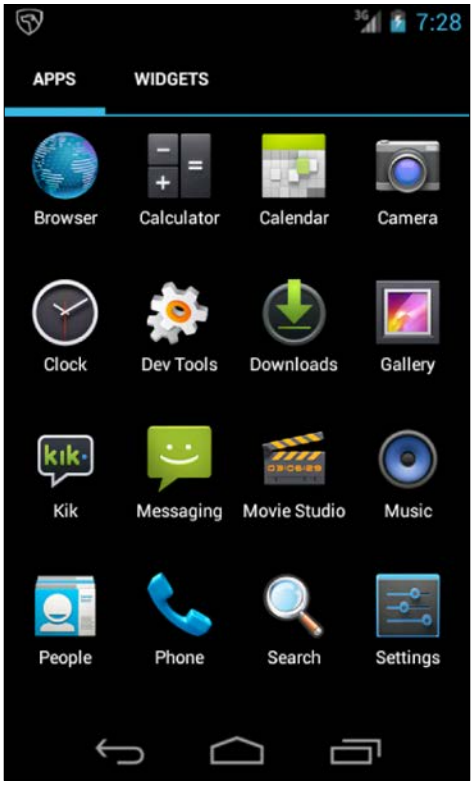
2. Selecting **Forward** from the **Add a project** dialog will bring you to the setup page:



3. To select an APK file to install into the emulator, choose **Add app manually** and select the file. Once the APK file has been selected, the **Download** button will become usable. Select this to push the APK to the emulator, and choose **OK** once it has completed:



4. You can now go to the emulator window and use the application to populate it with test data. Note that performance may be quite slow, as the emulator is a virtual machine running within a virtual machine:

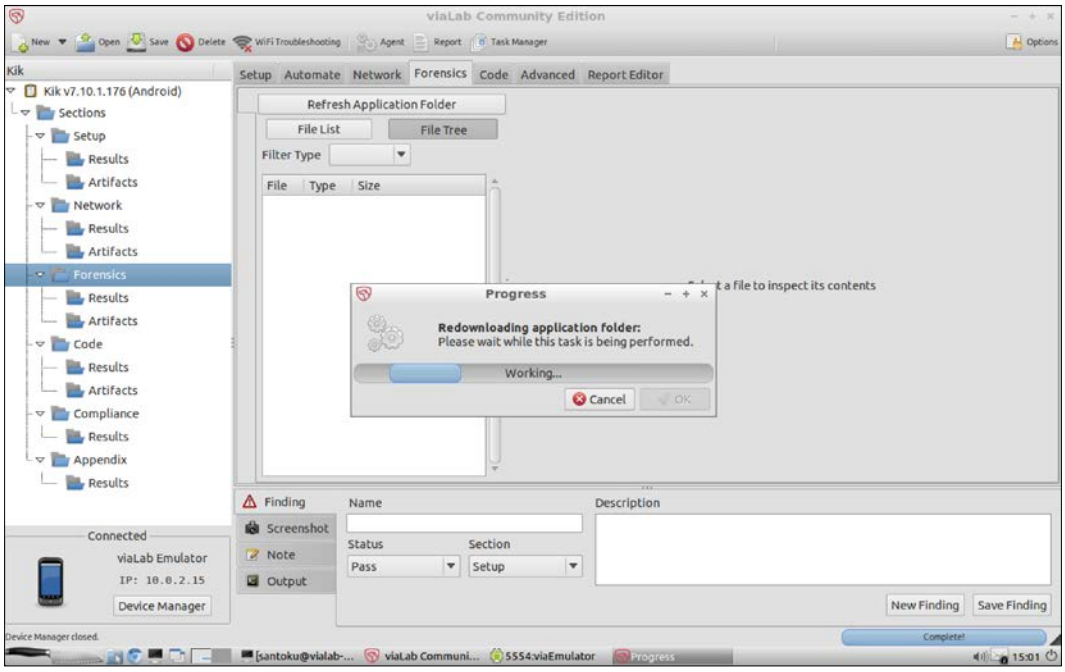


Applications may perform differently when under emulation. For example, Kik required us to solve a Captcha to prove that we were human! Other apps may have reduced functionality (such as anything involving GPS data).

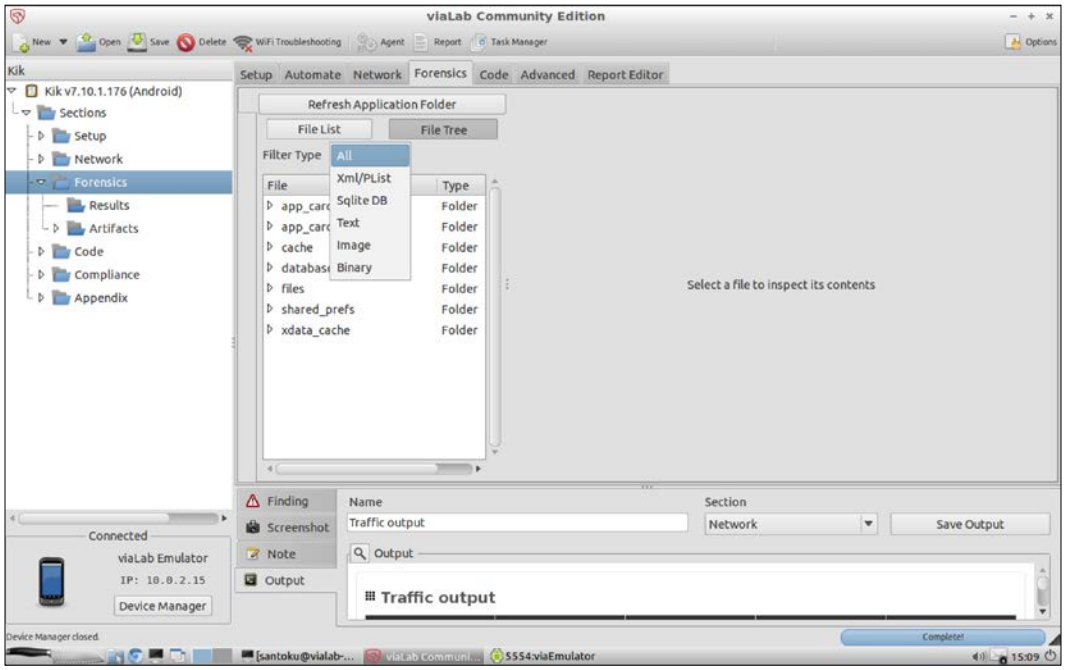
Analyzing data with ViaLab

After populating data within the application, let's analyze it! Follow these steps:

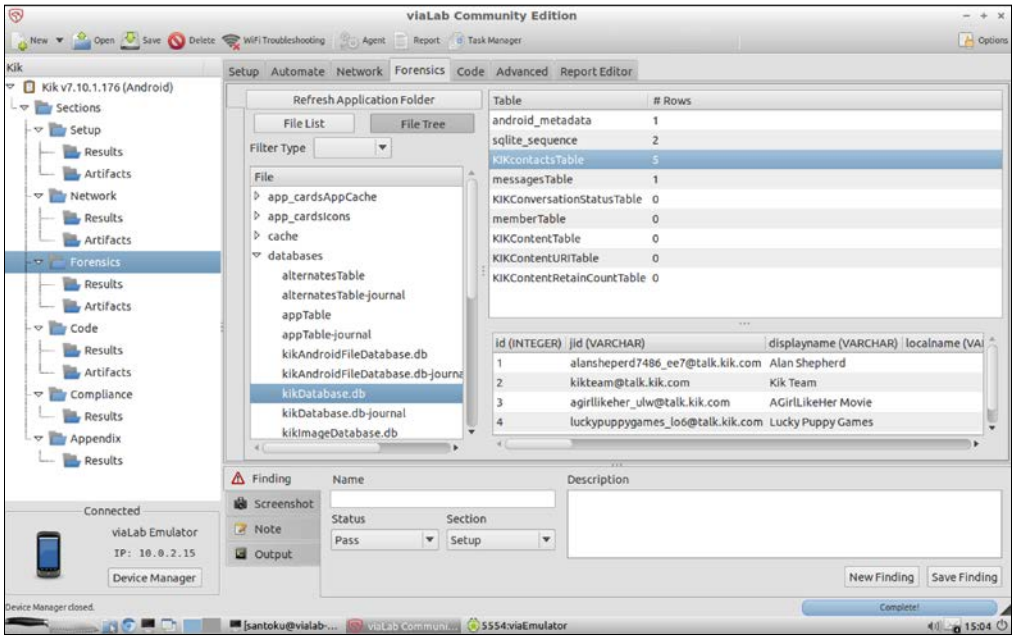
1. Return to ViaLab and choose the **Forensics** tab at the top of the screen. Selecting **Refresh Application Folder** will pull the data from the device:



2. Once the data has been synced, the **File List** button can be selected to filter the files by type:



As an example, here is the contact data we populated within the app, stored in the database we examined in *Chapter 7*, *Forensic Analysis of Android Applications*:



Summary

This chapter was an overview of a few free tools available for Android forensic examiners. These tools are summarized in the following table:

Tool	Features
ViaExtract	• Free, requires registration and an active Internet connection • Logical extractions via an application pushed to the device • Backup extractions • Filesystem extractions if device is rooted • Roots devices • Bypasses screen locks without root by pushing an application to the device
Autopsy	• Free and open source • Used to examine extractions done by other tools • Allows keyword searching, hash lists, and other common forensic methods • Powerful timeline feature • Can recover deleted data from supported filesystems
ViaLab	• Free, requires registration and an active Internet connection • Allows an examiner to run an application from the APK and determine data storage locations • Runs the application in an emulator or on a test device • Valuable tool to show an examiner where data is stored in an app's directory, as well as see the functionality of the application

Conclusion

We'd like to wish all of you the best of luck with your future Android examinations. We sincerely hope that something from this book will help you at some point. Our goal was to make an informative guide to the entire Android forensic process, from beginning to end. We hope you've learned plenty along the way (we certainly did while writing it). Thank you for reading!

Index

A

acct directory 75
acquisition phase, mobile forensics
  about 8
  filesystem acquisition 9
  logical acquisition 8
  manual acquisition 8
  physical acquisition 9
active-matrix organic light-emitting diode (AMOLED) 26
activity manager service 15
ADB
  backup, extracting over 117, 118
ADB backup
  data locations 120-123
  extractions 117
  parsing 119
adb daemon (adbd)
  about 50
  used, for accessing Android device 50
ADB Dumpsys
  about 123, 124
  App Ops 126
  batterystats 124
  conclusions 128
  notification 127
  procstats 125
  user 125
  Wi-Fi 127
adb pull command 55
adb push command 56
adb server
  restarting 56
advanced forensic methods
  about 166
  Chip-off 168, 169
  JTAG 166, 167
ahead of time (aot) compilation 15
Android
  core components 23
  deleted data, recovering on 175
Android Analyzer module 264
Android architecture
  about 12
  application framework 15
  applications layer 16
  Dalvik virtual machine 14, 15
  libraries 13
  Linux kernel 13
Android Backup Extractor
  URL 119
Android boot process
  about 28
  boot loader 29
  boot ROM code execution 28, 29
  Dalvik 32
  init process 31, 32
  Linux kernel 30
  system server 34, 35
  Zygote 32
Android Debug Bridge (ADB)
  about 49, 50
  on rooted device 68-70
Android device
  accessing 46-48
  accessing, adb daemon (adbd) used 50
  accessing, from workstation 44
  application, installing 55
  basic Linux commands 52
  cable, identifying 45
  commands, directing to specific device 51

connected device, detecting 50
connecting 44
data, pulling from 55
data, pushing to 56
drivers, installing 45
filesystems, viewing on 92
log data, viewing 57, 58
rooting 59
shell commands, issuing 51
Android Device Manager (ADM) 7
Android file hierarchy
about 74
directories 75
Android filesystems
about 93
Extended File Allocation Table (exFAT) 93
Flash Friendly File System (F2FS) 93
flash memory filesystems 93, 94
media-based filesystems 94, 95
overview 91
pseudo filesystems 95-97
Android forensics 1
Android full-disk encryption
bypassing 169
Android hardware components 23
Android lock screens
bypassing 128-131
None/Slide lock screens 129
password/PIN lock screens 129
pattern lock screens 129
Smart Locks 129
types 128
Android Lollipop
issues 141, 142
opportunities 141, 142
Android Open Source Project (ASOP) 94
Android partition layout
about 71
identifying 73
Android pattern lock
cracking 132-135
Android PIN/Password
cracking 136, 137
Android RAM
analyzing 159, 160
imaging 159, 160
Android Run Time (ART) 15
Android SD cards
acquiring 164, 165
security 166
Android SDK
about 38
installing 38-40
URL, for downloading 38
Android security
about 16
application sandboxing 19, 20
Application Signing 21
permission model 18, 19
secure interprocess communication 22
security at OS level, through Linux kernel 17
Android SIM card data
acquiring 138-140
Android SIM card extractions 137, 138
Android Virtual Device (AVD)
about 41
creating, on workstation 41-44
application
installing, on emulator 277-279
application data storage, on device
about 85, 86
external storage 90
internal storage 87, 89
network 91
shared preferences 86
SQLite database 90
application framework, Android architecture
about 15
activity manager 15
content providers 15
location manager 15
notifications manager 15
package manager 15
resource manager 15
telephony manager 15
view system 15
application reverse engineering
about 242
APK file, disassembling 244, 245
APK file, obtaining 242-244
code, viewing 246
permissions, determining 246

application sandboxing, Android security 19, 20
Application Signing, Android security 21
applications layer, Android architecture
system apps 16
user installed apps 16
Archive Extractor module 264
Authentication Key (Ki) 137
Autopsy
about 155-158, 261
case, creating in 261-265
data, analyzing in 266-272
URL, for downloading 155
reference link, for training course 272

B

backup
analyzing 186-188
extracting, over ADB 117, 118
extraction, performing with ViaExtract 252, 253
Base64
URL 230
basic Linux commands, Android device
cat 53
cd 54
chmod 54
cp 54
dd 54
exit 55
grep 54
ls 52
mkdir 55
pwd 55
rm 54
batteries, types
Lithium Ion (Li-ion) 26
Lithium Polymer (Li-Polymer) 26
Nickel Cadmium (NiCd) 26
Nickel Metal Hydrid (NiMH) 26
Binder mechanism 22
boot loader, Android boot process 29
boot loader partition 72
boot partition 72
boot ROM code execution, Android boot process 28, 29

C

cache directory 75
cache partition 72
case
creating, in Autopsy 261-265
cat command 53
CCL Forensics
URL 132
CCL Forensics PIN/Password
URL 136
cd command 54
Cellebrite UFED 101
challenges, mobile forensics 10, 11
Chip-off
about 168
references 169
chmod command 54
cold boot attacks 169
component
determining, for imaging 148-150
content providers service 15
context manager 23
control group (cgroup) 95
core components, Android
about 23
baseband processor 24
battery 26, 27
central processing unit (CPU) 23
display 25
memory 24
Secure Digital (SD) card 25
cycle 23

D

Dalvik 14, 32
dalvik-cache 77
Dalvik virtual machine (DVM) 14
data
analyzing, in Autopsy 266-272
analyzing, with ViaLab 280, 281
examining, in ViaExtract 258-260
extracting physically, with dd command 147
extracting physically, with nanddump command 153

pulling, from Android device 55
pushing, to Android device 56
data deleted, from internal memory
recovering 180
data deleted, from SD card
recovering 175-179
data directory
/data/data partition 77
about 76
dalvik-cache 77
data locations, within ADB backups 120-123
data recovery 173, 174
DCode
about 195
URL 195
dd command
about 54
data, extracting physically with 147
reference link, for options 147
dd command, format
bs 147
conv 147
if 147
noerror 147
notrunc 147
of 147
sync 147
d directory 75
Deflate algorithm 119
deleted data
recovering, by parsing SQLite files 181, 182
recovering, on Android 175
recovering, through file carving techniques 183-186
deleted files
recovering 174
dev directory 78
Digital Camera Images (DCIM) 80
digital forensics 145
directories
about 75
acct 75
cache 75
d 75
data 76
dev 78
init 78
misc 79
mnt 78
proc 78
root 79
sbin 79
sdcard 80, 81
system 81
ueventd.goldfish.rc 84
ueventd.rc 84
Dumpsys App Ops 126
Dumpsys batterystats 124
Dumpsys conclusions 128
Dumpsys notification 127
Dumpsys procstats 125
Dumpsys user 125
Dumpsys Wi-Fi 127

E

E01 Verifier module 264
Electrically Erasable Programmable Read-Only Memory (EEPROM) 25
Email Parser module 264
emulator
about 41
application, installing on 277-279
setting up, in ViaLab 273-276
enforcing mode 21
error correcting code (ECC) 158
Examination and Analysis phase, mobile forensics 9
EXIF Parser module 264
exploits, for rooting Android device 68
Extended File Allocation Table (exFAT) 93
EXTended file system (EXT2/EXT3/EXT4) 94
Extension Mismatch Detector module 264

F

Facebook analysis
about 213-216
bookmarks_db2 database 214
contacts_db2 database 214
nearbytiles_db database 215
newsfeed_db database 215
notifications_db database 216
prefs_db database 216
threads_db database 216

Facebook Messenger analysis
about 217
call_log.sqlite database 218
contacts_db2 file 218
prefs_db database 219
threads_db2 database 219
fastboot mode 64
fastboot mode, manual ADB data extraction
about 113
booting, to custom recovery image 115, 116
bootloader status, determining 113-115
File Allocation Table (FAT) 94
file carving techniques
deleted data, recovering from 183-186
filesystems
viewing, on Android device 92
file tree, Android
reference link 84
File Type Identification module 264
Flash Friendly File System (F2FS) 93
flash memory filesystems
about 93
Journal Flash File System version 2 (JFFS2) 94
Robust File System (RFS) 94
Yet Another Flash File System version 2 (YAFFS2) 94
forensic analysis, application
about 192
need for 193
package name 193
full physical image
analyzing 154
verifying 154

G

geo-fencing 130
Global Positioning System (GPS) 2
Gmail analysis
about 201
suggestions.db database 202
Google Chrome analysis
about 202, 203
Cookies database 204
History database 204
Login Data database 205
SyncData.sqlite3 database 203
Top Sites database 206
Web Data database 206
WebKit time format, decoding 206
Google Hangouts analysis
about 208
babel#.db 209
Google Keep analysis
about 210
Julian date, converting 211
keep.db 210
Google Maps analysis
about 207
gmm_myplaces.db database 208
gmm_storage.db database 208
Google Nexus
URL 161
Google Plus analysis
about 211
Es0.db database 212, 213
grep command 54

H

Hangouts 208
Hash Lookup module 264

I

imaging
component, determining for 148-150
init directory 78
initial program load (IPL) 29
init process, Android boot process 31, 32
installed applications
determining 194
Institute of Electrical and Electronics Engineers (IEEE) 166
Integrated Circuit Card ID (ICCID) 226
Interesting Files Identifier module 264
International Mobile Station Equipment Identity (IMEI) 240
International Mobile Subscriber Identity (IMSI) 137
interprocess communication (IPC) 22
Investigation Preparation phase, mobile forensics 4
issues, with analyzing physical dumps 158

J

jailbreaking 60
Java Runtime Environment (JRE) 244
Java virtual machine (JVM) 14
Joint Test Action Group (JTAG) 146, 166
Journal File System (JFS) 92
Journal Flash File System version 2 (JFFS2) 94
JTAG tools
reference link 168
just-in-time (JIT) compilation 15

K

Keyword Search module 264
Kik analysis
about 237
kikDatabase.db database 238

L

libraries, Android architecture 13
LiME (Linux Memory Extractor)
RAM, imaging with 160, 161
URL, for source code 161
Linux epoch time 194, 195
Linux kernel, Android architecture 13
Linux kernel, Android boot process 30
Lithium Ion (Li-ion) batteries 26
Lithium Polymer (Li-Polymer) batteries 26
Location Area Identity (LAI) 137
location manager service 15
locked boot loader
about 64
rooting 68
logical extraction
overview 99-101
performing, with ViaExtract 254-258

M

mandatory access control (MAC) 20
manual ADB data extraction
about 101
ADB pull 104-107
fastboot mode 113
recovery mode 107-112
USB debugging 101-103
media-based filesystems
about 94
EXTended file system (EXT2/EXT3/EXT4) 94
File Allocation Table (FAT) 94
Virtual File Allocation Table (VFAT) 95
Media Transfer Protocol (MTP) 46
mem
output 164
RAM, imaging with 162, 163
reference link 162
Memory Technology Device (MTD) 153
MicroSystemation XRY 101
misc directory 79
mmcblk0p34 152
mnt directory 78
Mobile Device Management (MDM) 7
Mobiledit!
URL 138
mobile forensics
about 1-4, 146
acquisition phase 8, 9
challenges 10, 11
Examination and Analysis phase 9
Investigation Preparation phase 4
need for 3
reporting phase 9, 10
Seizure and Isolation phase 4-8
modules, Autopsy
Android Analyzer 264
Archive Extractor 264
E01 Verifier 264
Email Parser 264
EXIF Parser 264
Extension Mismatch Detector 264
File Type Identification 264
Hash Lookup 264
Interesting Files Identifier 264
Keyword Search 264
Recent Activity 264
mount point 92
MSISDN 137
Multichip Package (MCP) 25
Multimedia Card (MMC) 153

N

nanddump
data, extracting physically with 153
reference link 153
netcat
installing, on device 151
reference link 151
used, for writing image to machine 151
using 152
Nickel Cadmium (NiCd) batteries 26
Nickel Metal Hydrid (NiMH) batteries 26
None/Slide lock screens 129
notifications manager service 15
NowSecure
about 249
URL 249

O

online epoch converter
URL 195
open source Python scripts
reference link 182
open source release sites
references 161
Out-of-Band (OOB) area 158
output file, dd command
writing, to SD card 150, 151
Oxygen Forensics SQLite Viewer 181

P

package manager service 15
partition layout, Android
about 71
identifying 73
partitions, Android
about 71
boot 72
boot loader 72
cache 72
radio 72
recovery 72
system 72
userdata 72
password/PIN lock screens 129
pattern lock screens 129
permission model, Android security 18, 19
permissive mode 21
Personal Unblocking Key (PUK) 140
physical dumps
issues, of analyzing 158, 159
physical extraction
overview 145, 146
root access 146
Picture Transfer Protocol (PTP) 46
proc directory 78
procfs filesystem 96
proxies 22
pseudo filesystems
about 95
control group (cgroup) 95
procfs 96
rootfs 95
sysfs 96
tmpfs 97
pwd command 55
Python 3
URL, for downloading 132

R

radio frequency (RF) 154
radio partition 72
RAM
about 160
imaging, with LiME 160, 161
imaging, with mem 162, 163
raw image 145
Recent Activity module 264
recovery images
URL, for source 115
recovery mode
about 61
accessing 62
custom recovery 62, 63
recovery mode, manual ADB data extraction 107-112
recovery partition 72
reporting phase, mobile forensics 9, 10
resource manager service 15
Robust File System (RFS) 94
root directory 79
rootfs filesystem 95

rooting
about 59
need for 60, 61

S

sbin directory 79
Scalpel 183
scalpel.conf file
URL, for downloading 184
sdcard directory 80, 81
second program load (SPL) 29
Secure Digital (SD) card 25
secure interprocess communication, Android security 22
Secure USB debugging protection
URL 102
Seizure and Isolation phase, mobile forensics
about 4
settings 5-7
SELinux (Security-Enhanced Linux)
about 20
in Android 20
Service Set ID (SSID) 196
SIM cloning 141
SIM security 140
Skype analysis
about 220, 221
main.db database 221, 222
video messages, recovering 223
Smart Locks
about 129
Trusted Device 130
Trusted Face 130
Trusted Location 130
SMS/MMS analysis
about 198, 199
mmssms.db database 199
telephony.db database 199
smudge attack 129
Snapchat analysis
about 224
tcspahn.db database 225
SQLCipher 240
SQLite 90
stubs 22
su update package
URL, for downloading 67
swipe codes 129
sysfs filesystem 96
system apps 16
system directory
about 81
app folder 83
build.prop folder 82
framework folder 83
system partition 72
system server, Android boot process 34, 35

T

Tango analysis
about 227, 228
messages, decoding 230-234
tc.db database 229
Team Win Recovery Project (TWRP) 109
telephony manager service 15
thin film transistor liquid crystal display (TFT LCD) 25
tmpfs filesystem 97
token 23
tools, ViaExtract 260

U

unique user ID (UID) 60
Universal ADB Driver
URL 102
Universal Character Set Transformation Format-8 (UTF-8) format 86
unlocked boot loader
about 65
rooting 66, 67
USB debugging, manual ADB data extraction
about 101-103
ADB shell, used for determining device rooting 103, 104
userdata partition 72
user dictionary analysis 200
user ID (UID) 19
user installed apps 16

V

version, SDK tools package
URL, for downloading 38
ViaExtract
about 249
backup extraction, performing with 252, 253
data, examining in 258-260
launching 250, 251
logical extraction, performing with 254-258
ViaLab
Community Edition 272
data, analyzing with 280, 281
emulator, setting up in 273-276
Viber analysis
about 225, 226
viber_data file 226
viber_messages file 227
view system service 15
Virtual File Allocation Table (VFAT) 95
virtual file system (VFS) 92
Volatility plugin
reference link 161

W

wake lock section 125
WeChat analysis
about 238, 239
EnMicroMsg.db database, decrypting 240
EnMicroMsg-decrypted.db database 241
WhatsApp analysis
about 234-236
backups, decrypting 236
msgstore.db database 235
wa.db database 236
Wi-Fi analysis 196
workstation
Android device, accessing from 44
Android Virtual Device (AVD), creating on 41-44

X

XDA forums
URL 40

Y

Yet Another Flash File System version 2 (YAFFS2) 94

Z

Zygote
about 32
reference link, for loading process 32



Thank you for buying

Learning Android Forensics

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Open Source

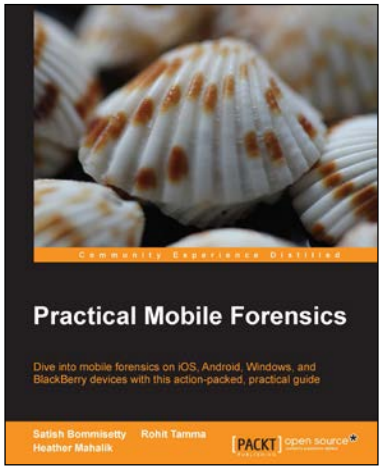
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

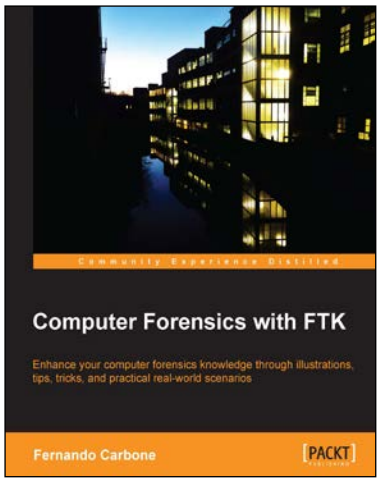


Practical Mobile Forensics

ISBN: 978-1-78328-831-1 Paperback: 328 pages

Dive into mobile forensics on iOS, Android, Windows, and BlackBerry devices with this action-packed, practical guide

1. Clear and concise explanations for forensic examinations of mobile devices.
2. Master the art of extracting data, recovering deleted data, bypassing screen locks, and much more.
3. The first and only guide covering practical mobile forensics on multiple platforms.



Computer Forensics with FTK

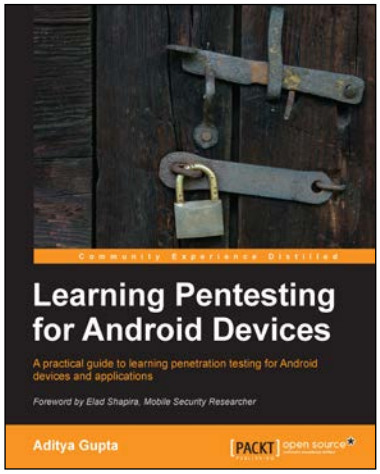
ISBN: 978-1-78355-902-2 Paperback: 110 pages

Enhance your computer forensics knowledge through illustrations, tips, tricks, and practical real-world scenarios

1. Receive step-by-step guidance on conducting computer investigations.
2. Explore the functionality of FTK Imager and learn to use its features effectively.
3. Conduct increasingly challenging and more applicable digital investigations for generating effective evidence using the FTK platform.

Please check **www.PacktPub.com** for information on our titles

[PACKT] open source
community experience distilled
PUBLISHING

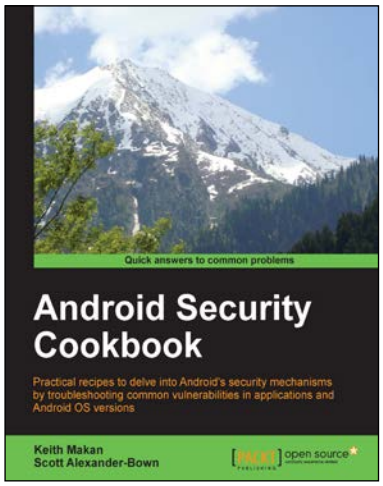


Learning Pentesting for Android Devices

ISBN: 978-1-78328-898-4 Paperback: 154 pages

A practical guide to learning penetration testing for Android devices and applications

1. Explore the security vulnerabilities in Android applications and exploit them.
2. Venture into the world of Android forensics and get control of devices using exploits.
3. Hands-on approach covers security vulnerabilities in Android using methods such as Traffic Analysis, SQLite vulnerabilities, and Content Providers Leakage.



Android Security Cookbook

ISBN: 978-1-78216-716-7 Paperback: 350 pages

Practical recipes to delve into Android's security mechanisms by troubleshooting common vulnerabilities in applications and Android OS versions

1. Analyze the security of Android applications and devices, and exploit common vulnerabilities in applications and Android operating systems.
2. Develop custom vulnerability assessment tools using the Drozer Android Security Assessment Framework.
3. Reverse-engineer Android applications for security vulnerabilities.

Please check **www.PacktPub.com** for information on our titles