
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
Tampering and Reverse Engineering	2.4
Cryptography for Mobile Apps	2.5

Android Testing Guide

Platform Overview	3.1
Android Security Testing Basics	3.2
Testing Data Storage on Android	3.3
Testing Cryptography in Android Apps	3.4
Testing Local Authentication in Android Apps	3.5
Testing Network Communication in Android Apps	3.6
Testing Platform Interaction on Android	3.7
Testing Code Quality and Build Settings of Android Apps	3.8
Tampering and Reverse Engineering on Android	3.9
Testing Anti-Reversing Defenses on Android	3.10

iOS Testing Guide

Platform Overview	4.1
iOS Security Testing Basics	4.2
Testing Local Authentication in iOS Apps	4.3

Appendix

Assessing Software Protection Schemes	5.1
Testing Tools	5.2
Suggested Reading	5.3

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 technical processes for verifying the controls listed in the OWASP Mobile Application Security Verification Standard (MASVS). The MSTG is meant to provide a baseline set of test cases for static and dynamic security tests, and to help ensure completeness and consistency of the 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 https://owasp.slack.com/messages/project-mobile_omtg/details/. You can sign up here:

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

Copyright and License



Copyright © 2017 The OWASP Foundation. This document is released under the Creative Commons Attribution ShareAlike 3.0 license. For any reuse or distribution, you must make clear to others the license terms of this work.

Acknowledgements

Note: This table is generated based on the contribution log, which can be found under <https://github.com/OWASP/owasp-mstg/graphs/contributors>. For more details, see the GitHub Repository README under <https://github.com/OWASP/owasp-mstg/blob/master/README.md>. Note that this isn't updated in real time (yet) - we do this manually every few weeks, so don't panic if you're not listed immediately.

Authors

Bernhard Mueller

Bernhard is a cyber security specialist with a talent in hacking all kinds of systems. 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. His pioneering work in mobile security was commended with a BlackHat "Best Research" Pwnie Award.

Sven Schleier

Sven is an experienced penetration tester and security architect who specialized in implementing secure SDLC for web application, iOS 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 on web and mobile app security testing. He has published several security advisories and a 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 spare time to a variety of projects with the goal of advancing the sectors of software and security. He is also teaching at various institutions. He holds CISSP, CSSLP and CEH credentials.

Jeroen Willemse

Jeroen is a full-stack developer specialized in IT security at Xebia with a passion for mobile and risk management. He loves to explain things: starting as a teacher teaching PHP to bachelor students and then move along explaining security, risk management and programming issues to anyone willing to listen and learn.

Top Contributors

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

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

Contributors

Contributors have made a quality contribution with at least 50 additions logged in the GitHub repository.

Jin Kung Ong, Gerhard Wagner, Andreas Happe, Michael Helwig, Denis Pilipchuk, Ryan Teoh, Dharshin De Silva, Anita Diamond, Daniel Ramirez Martin, Claudio André, Enrico Verzegnassi, Prathan Phongthiproek, Tom Welch, Luander Ribeiro, Oguzhan Topgul, Carlos Holguera, David Fern, Pishu Mahtani and Anuruddha.

Reviewers

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

- Anant Shrivastava
- Sjoerd Langkemper

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

The OWASP Mobile Security Testing Guide (MSTG) is an extension of the OWASP Testing Project specifically focusing on the security testing of Android and iOS devices.

The goal of this project is to help people understand the what, why, when, where, and how of testing applications on Android and iOS devices. The project delivers a complete suite of test cases designed to address the OWASP Mobile Top 10, the Mobile App Security Checklist and the Mobile Application Security Verification Standard (MASVS).

Why Does the World Need a Mobile Application Security Testing Guide?

Every new technology introduces new security risks, and mobile computing is no different. Even though modern mobile operating systems like iOS and Android are arguably more secure by design compared to traditional Desktop operating systems, there's still a lot of things that can go wrong when security is not considered during the mobile app development process. Data storage, inter-app communication, proper usage of cryptographic APIs and secure network communication are only some of the aspects that require careful consideration.

Security concerns in the mobile app space differ from traditional desktop software in some important ways. Firstly, while not many people opt to carry a desktop tower around in their pocket, doing this with a mobile device is decidedly more common. As a consequence, mobile devices are more readily lost and stolen, so adversaries are more likely to get physical access to a device and access any of the data stored.

Key Areas in Mobile AppSec

Many mobile app pen testers have a background in network and web app penetration testing, and a lot of their knowledge is useful in mobile app testing. Practically every mobile app talks to some kind of backend service, and those services are prone to the same kinds of attacks we all know and love. On the mobile app side however, there is only little attack surface for injection attacks and similar attacks. Here, the main focus shifts to data protection both on the device itself and on the network. The following are some of the key areas in mobile app security.

Local Data Storage

The protection of sensitive data, such as user credentials and private information, is a key focus in mobile security. Firstly, sensitive data can be unintentionally exposed to other apps running on the same device if operating system mechanisms like IPC are used improperly. Data may also unintentionally leak 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 an adversary gaining physical access is a more likely scenario.

From the view of a mobile app, this extra care has to be taken when storing user data, such as using appropriate key storage APIs and taking advantage of hardware-backed security features when available.

On Android in particular, one has to deal with the problem of fragmentation. Not every Android device offers hardware-backed secure storage. Additionally, a large percentage of devices run outdated versions of Android with older API versions. If those versions are to be supported, apps must restrict themselves to older API versions that may lack important security features. When the choice is between better security and locking out a good percentage of the potential user base, odds are in favor of better security.

Communication with Trusted Endpoints

Mobile devices regularly connect to a variety of networks, including public WiFi networks shared with other (possibly malicious) clients. This creates great opportunities for network-based attacks, from simple packet sniffing to creating a rogue access point and going SSL man-in-the-middle (or even old-school stuff like routing protocol injection - the bad guys aren't picky).

It is crucial to maintain confidentiality and integrity of information exchanged between the mobile app and remote service endpoints. At the very least, a mobile app must set up a secure, encrypted channel for network communication using the TLS protocol with appropriate settings.

Authentication and Authorization

In most cases, user login to a remote service is an integral part of the overall mobile app architecture. Even though most of the authentication and authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. In contrast to web apps, mobile apps often store long-time session tokens that are then unlocked via user-to-device authentication features such as fingerprint scan. While this

allows for a better user experience (nobody likes to enter a complex password every time they start an app), it also introduces additional complexity and the concrete implementation has a lot of room for errors.

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

Interaction with the Mobile Platform

Mobile operating system architectures differ from that of classical Desktop architectures in important ways. For example, all mobile OSes implement app permission systems that regulate access to specific APIs. They also offer more (Android) or less (iOS) rich inter-process communication 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 used incorrectly, sensitive data or functionality might be unintentionally exposed to other apps running on the device.

Code Quality and Exploit Mitigation

"Classical" injection and memory management issues play less of a role on the mobile app side. This is mostly due to the lack of the necessary attack surface: For the most part, mobile apps only interface with the trusted backend service and the UI, so even if a ton of buffer overflow vulnerabilities exist in the app, those vulnerabilities usually don't open up any useful attack vectors. The same can be said for browser exploits such as XSS that are very prevalent in the web world. Of course, there's always exceptions, and XSS is theoretically possible in some cases, but it's very rare to see XSS issues that one can actually exploit for benefit.

All this doesn't mean however that we should let developers get away with writing sloppy code. Following security best practice results in hardened release builds that are resilient against tampering. "Free" security features offered by compilers and mobile SDKs help to increase security and mitigate attacks.

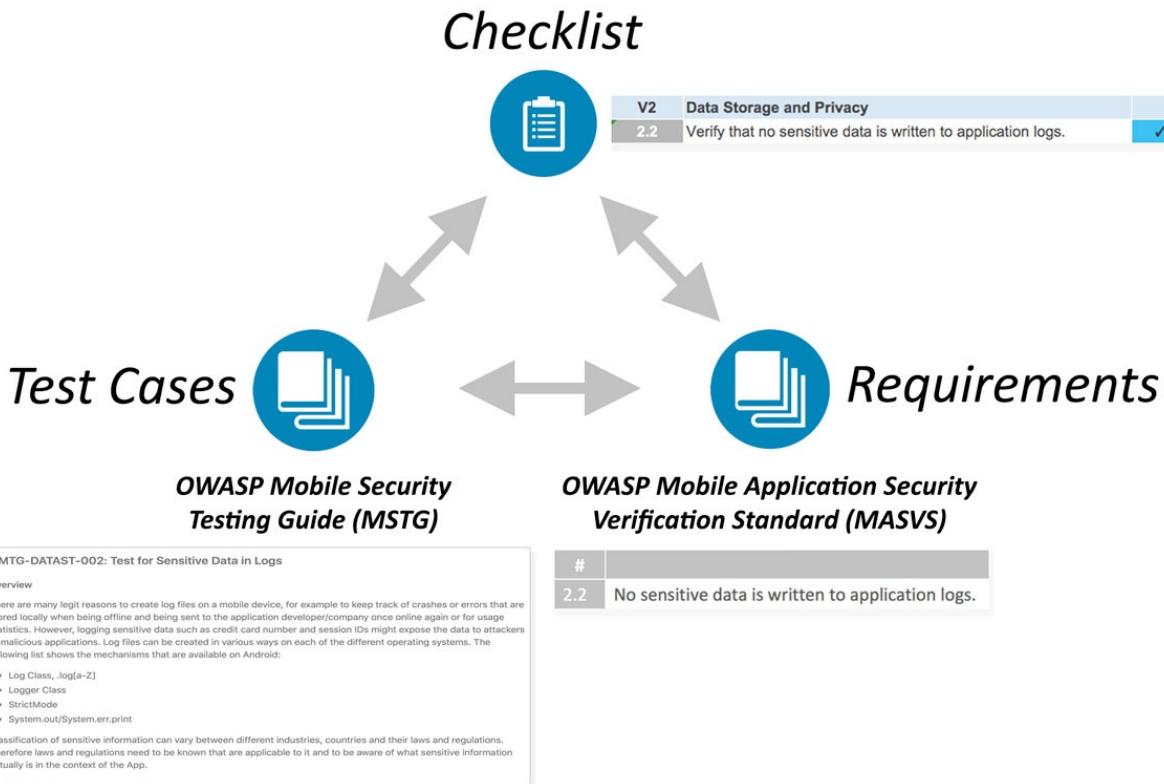
Anti-Tampering and Anti-Reversing

There are three things you should never bring up in date conversations: Religion, politics and code obfuscation. Many security experts dismiss client-side protections outright. However, the fact is that software protection controls are widely used in the mobile app world, so security testers need ways to deal with them. We also think that there is *some* benefit to be had, as long as the protections 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, Checklist and Testing Guide

This guide belongs to a set of three closely related mobile application security documents. All three documents map to the same basic set of security requirements. Depending on the context, they can be used stand-alone or in combination to achieve different objectives:

- The **Mobile Application Security Verification Standard (MASVS)**: A standard that 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 what a secure mobile application is.
- The **Mobile Security Testing Guide (MSTG)**: A manual for testing the security of mobile apps. It provides verification instructions for the requirements defined in the MASVS along with operating-system-specific best practices (currently for Android and iOS). The MSTG helps ensure completeness and consistency of mobile app security testing. It is also useful as a standalone learning resource and reference guide for mobile application security testers.
- The **Mobile App Security Checklist**: A checklist for tracking compliance against the MASVS during practical assessments. The list conveniently links to the MSTG test case for each requirement, making mobile penetration app testing a breeze.



For example, the MASVS requirements could be used in the 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 next chapter, we'll describe how the checklist and guide can be practically applied during a mobile application penetration test.

Organization of the Mobile Security Testing Guide

All requirements specified in the MASVS are described in technical detail in the testing guide. The main sections of the MSTG are explained briefly in this chapter.

The guide is organized as follows:

- In the general testing guide (the following few chapters), we present the mobile app security testing methodology, and talk about general vulnerability analysis techniques as they apply to mobile application security.
- The Android Testing Guide covers the everything specific to the Android platform, including security basics, security test cases, and reverse engineering and tampering techniques and preventions.

- The iOS Testing Guide Testing Guide covers everything specific to iOS, including an overview of the iOS OS, security testing, reverse engineering and anti-reversing.
- The appendix presents 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 self-contained computer programs that are designed to execute on mobile devices. Today, mobile Internet usage has surpassed desktop usage for the first time in history, and mobile apps are the [most widespread kind of applications](#). In this guide, we focus on mobile apps designed to run on the Android and iOS operating systems, which cumulatively take [more than 99% of the mobile OS market share](#). These apps don't necessarily always run only on mobile devices - they are increasingly used on other device types, such as smart watches, TVs, cars, and embedded systems. In this guide, we'll be using the term "app" to refer to any kinds of apps running on popular mobile OSes.

Native App

Most operating systems, including Android and iOS, come with a set of high-level APIs that can be used to develop applications specifically for that system. Such applications are called `native` for the system for which they have been developed. Usually, when discussing a `mobile app`, the assumption is that it is a `native app`, implemented in the standard programming languages for that operating system - either Objective-C or Swift for iOS, and Java or Kotlin for Android.

Native mobile apps can provide fast performance and a high degree of reliability. They usually adhere to platform-specific design principles (e.g. the [Android Design Principles](#), and provide a more consistent UI than `hybrid` and `web` apps. Due to their close integration with the operating system, native apps have access to almost every component of the device (camera, sensors, hardware backed key stores, etc.)

There can be some ambiguity when discussing `native` apps for Android. Android provides two sets of APIs to develop against - the Android SDK and the Android NDK. The SDK (or Software Development Kit) is a Java API and is the default API against which applications are built. The NDK (or Native Development Kit) is a C/C++ based API used for developing application components that require specific optimization, or which can otherwise benefit from access to lower level APIs (such as OpenGL). Normally, you can only distribute apps built with the SDK, which potentially can also consume NDK APIs. Therefore we say that Android `native **apps**` (built with the SDK) can 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. They run in a browser and are usually developed in HTML5. Launcher icons may be created to give starting-up the app a native feel, but these often simply act as browser bookmarks, opening the default web browser and loading the bookmarked web page.

Web apps have limited integration with the general components of the device (usually being sandboxed in the browser), and may have noticeable differences in performance from native apps. Since they typically target multiple platforms, their UIs do not follow some of the design principles users of a specific platform are used to. Their biggest advantage is reduced development and maintenance costs arising from having a single code base, as well as allowing developers to distribute updates without engaging the platform specific app stores (such as by simply changing HTML files on the web server hosting the application).

Hybrid App

Hybrid apps attempt to fill the gap between native and web apps. Namely, hybrid apps are (distributed and executed as) native apps, that have majority of their content implemented on top of web technologies, running in an embedded web browser (web view). As such, hybrid apps inherit some of the pros and cons of both native and web apps.

A web-to-native abstraction layer enables access to device capabilities for hybrid apps that are not accessible to mobile web applications. Depending on the framework used for developing, one code base can result in multiple applications, targeting separate platforms, with a UI closely resembling that of the targeted platform. Nevertheless, usually significant effort is required to exactly match the look and feel of a native app.

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](#)

Mobile App Security Testing

Throughout the guide, use "mobile app security testing" as an catch-all phrase for evaluating the security of mobile apps using static and/or dynamic analysis. In practice you'll find that various terms such as "Mobile App Penetration Testing", "Mobile App Security Review", and others are used somewhat inconsistently in the security industry, but those terms refer to roughly the same thing. Usually, a mobile app security test is this is done as part of a larger security assessment or penetration test that also encompasses the overall client-server architecture, as well as server-side APIs used by the mobile app.

In this guide we cover mobile app security testing in two different contexts. The first one is the "classical" security test done towards the end of the development life cycle. Here, the tester gets access to a near-final or production-ready version of the app, identifies security issues, and writes an (usually devastating) report. The other context is implementing requirements and automating security tests from the beginning of the software development life cycle. In both cases, the same basic requirements and test cases apply, but there's a difference in the high-level methodology and level of interaction with the client.

Security Testing the Old-School Way

The classical approach is to perform all-around security testing of the mobile app and its environment on the final or near-final build of the app. In that case, we recommend using the [Mobile App Security Verification Standard \(MASVS\)](#) and checklist as a reference. A typical security test is structured as follows.

- **Preparation** - defining the scope of security testing, such as which security controls are applicable, what goals the development team/organization have for the testing, and what counts as sensitive data in the context of the test.
- **Intelligence Gathering** - involves analyzing the **environmental** and **architectural** context of the app, to gain a general contextual understanding of the app.
- **Threat Modeling** - consumes information gathered during the earlier phases to determine what threats are the most likely, or the most serious, and therefore which should receive the most attention from a security tester. Produces test cases that may be used during test execution.
- **Vulnerability Analysis** - identifies vulnerabilities using the previously created test cases, including static, dynamic and forensic methodologies.

Preparation

Before conducting a test, an agreement must be reached as to what security level of the to test against. The security requirements should ideally have been decided at the beginning of the SDLC, but this may not always be the case. In addition, different organizations have different security needs, and different amounts of resources to invest in test activity. While the controls in MASVS Level 1 (L1) are applicable to all mobile apps, it is a good idea to walk through the entire checklist of L1 and Level 2 (L2) MASVS controls with technical and business stakeholders to agree an appropriate level of test coverage.

Organizations/applications may have different regulatory and legal obligations in certain territories. Even if an app does not handle sensitive data, it may be important to consider whether some L2 requirements may be relevant due to industry regulations or local laws. For example, 2-factor-authentication (2FA) may be obligatory for a financial app, as enforced by the respective country's central bank and/or financial regulatory authority.

Security goals/controls defined earlier in the SDLC may also be reviewed during the stakeholder discussion. 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 need to agree on the decisions made and on the scope in the checklist, as this will define the baseline for all security testing, regardless if done manually or automatically.

Identifying Sensitive Data

Classification of sensitive information can vary between different industries and countries. Beyond legal and civic obligations, organizations may take a more restrictive view of what counts as sensitive data, and may have a data classification policy that clearly defines what counts as sensitive information.

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

- **At rest** - when the data is sitting in a file or data store
- **In use** - when an application has load the data into its address space
- **In transit** - when data has been sent between consuming process - e.g. during IPC.

The degree of scrutiny to apply to each state may depend on the criticality of the data, and likelihood of access. For example, because the likelihood of malicious actors gaining physical access to mobile devices is greater, data held in application memory may be more at risk of being accessed via core dumps than that on a web-server.

If no data classification policy is available, the following kinds of information are generally considered to be sensitive:

- User authentication information (credentials, PINs etc.).
- Personal Identifiable Information (PII) that can be abused for identity theft: Social security numbers, credit card numbers, bank account numbers, health information.
- Highly sensitive data that would lead to reputational harm and/or financial costs if compromised.
- Any data that must be protected by law or for compliance reasons.
- Finally any technical data, generated by the application or its related systems, that is used to protect other data or the system, should also be considered as sensitive information (e.g. encryption keys).

It may be impossible to detect leakage of sensitive data without a firm definition of what counts as such, so such a definition must be agreed upon in advance of testing.

Intelligence Gathering

Intelligence gathering involves the collection of information about the architecture of the app, the business use cases it serves, and the context in which it operates. Such information may be broadly divided into "environmental" and "architectural".

Environmental Information

Environmental information concerns understanding:

- The goals the organization has for the app. What the app is supposed to do shapes the ways users are likely to interact with it, and may make some surfaces more likely to be targeted than others by attackers.
- The industry in which they operate. Specific industries may have differing risk profiles, and may be more or less exposed to particular attack vectors.
- 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 concerns understanding:

- The mobile app: How the app accesses data and manages it in-process, how it communicates with other resources, manages user sessions, and whether it detects and reacts to running on jailbroken or rooted phones.
- The Operating System: What operating systems and versions does the app run on (e.g. is it restricted to only newer Android or iOS, and do we need to be concerned about vulnerabilities in earlier OS versions), is it expected to run on devices with Mobile Device Management (MDM) controls, and what OS vulnerabilities might be relevant to the app.
- Network: Are secure transport protocols used (e.g. TLS), is network traffic encryption secured with strong keys and cryptographic algorithms (e.g. SHA-2), is certificate pinning used to verify the endpoint, etc.
- Remote Services: What remote services does the app consume? If they were compromised, could the client be compromised?

Threat Modeling

Threat Modeling involves using the results of the information gathering phase to determine what threats are likely or severe, producing test cases that may be executed at later stages. Threat modeling should be a key part of the software development life cycle, and ideally be performed at an earlier stage.

The [threat modeling guidelines defined by OWASP](#) are generally applicable to mobile apps.

Vulnerability Analysis

White-box versus Black-box

As far as the mobile app itself is concerned, there isn't really a difference between white-box and black-box testing. The tester has access to at least the compiled app, and with a good decompiler and disassembler that's pretty much equivalent to having the source code.

Static Analysis

When executing static analysis, the source code of the mobile app is analyzed to ensure sufficient and correct implementation of security controls. In most cases, a hybrid automatic / manual approach is used. Automatic scans catch the low-hanging fruits, while the human tester can explore the code base with specific business and usage contexts in mind, providing enhanced relevance and coverage.

Automatic Code Analysis

Automated analysis tools check the source code for compliance with a predefined set of rules or industry best practices. The tool then typically displays a list of findings or warnings and flags all detected violations. Static analysis tools come in different varieties - some only run against the compiled app, some need to be fed with the original source code, and some run as live-analysis plugins in the Integrated Development Environment (IDE).

While some static code analysis tools do encapsulate a deep knowledge of the underlying rules and semantics required to perform analysis of mobile apps, they can produce a high number of false positives, particularly if the tool is not configured properly for the target environment. The results must therefore always be reviewed by a security professional.

A list of static analysis tools can be found in the chapter "Testing tools".

Manual Code Analysis

In manual code analysis, a human reviewer manually analyzes the source code of the mobile application for security vulnerabilities. Methods range from a basic keyword search with grep to identify usages of potentially vulnerable code patterns, to detailed line-by-line reading of the source code. IDEs often provide basic code review functionality and can be extended through different tools to assist in the reviewing process.

A common approach is to identify key indicators of security vulnerabilities by searching for certain APIs and keywords. For example, database-related method calls like "executeStatement" or "executeQuery" are key indicators which may be of interest. Code locations containing these strings are good starting points for manual analysis.

Compared to automatic code analysis tools, manual code review excels at identifying vulnerabilities in the business logic, standards violations and design flaws, especially in situations where 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 human code reviewer who is proficient in both the language and the frameworks used in the mobile application. As full code review can be time-consuming, slow and tedious for the reviewer; especially for large code bases with many dependencies.

Dynamic Analysis

In dynamic analysis the focus is on testing and evaluating an app by executing it in real-time. The main objective of dynamic analysis is to find security vulnerabilities or weak spots in a program while it is running. Dynamic analysis is conducted both on the mobile platform layer

also be conducted against the backend services and APIs of mobile applications, where its request and response patterns can be analyzed.

Usually, dynamic analysis is performed to check whether there are sufficient security mechanisms in place to prevent disclosure of data in transit, authentication and authorization issues and server configuration errors.

Pros of Dynamic Analysis

- Does not require access to the source code
- Able to identify infrastructure, configuration and patch issues that static analysis tools may miss

Cons of Dynamic Analysis

- Limited scope of coverage because the mobile application must be foot-printed to identify the specific test area
- No access to the actual instructions being executed, as the tool exercises the mobile application and conducts pattern matching on requests and responses

Runtime Analysis

Traffic Analysis

Dynamic analysis of the traffic exchanged between client and server can be performed by launching a Man-in-the-middle (MITM) attack. This can be achieved by using an interception proxy like Burp Suite or OWASP ZAP for HTTP traffic.

- Configuring an Android Device to work with Burp -
<https://support.portswigger.net/customer/portal/articles/1841101-configuring-an-android-device-to-work-with-burp>
- Configuring an iOS Device to work with Burp -
<https://support.portswigger.net/customer/portal/articles/1841108-configuring-an-ios-device-to-work-with-burp>

In case another (proprietary) protocol is used in a mobile app that is not HTTP, the following tools can be used to try to intercept or analyze the traffic:

- Mallory
- Wireshark

Reporting

Avoiding False Positives

A common pitfall for security testers is reporting issues that would be exploitable in a web browser, but aren't relevant in the context of the mobile app. The reason for this is that automated tools used to scan the backend service assume a regular, browser based web application. Issues such as CSRF, missing security headers and others are reported accordingly.

For example, a successful CSRF attack requires the following:

1. It must be possible to entice the logged-in user to open a malicious link in the same web browser used to access the vulnerable site;
2. 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 were used, any malicious link clicked by the user would open in the default browser which has its own, separate cookie store.

Stored cross-site Scripting can be an issue when the app uses Webviews, and potentially even lead to command execution if the app exports JavaScript interfaces. However, reflected cross-site scripting is rarely an issue for the same reasons stated above (even though one could argue that they shouldn't exist either way - escaping output is simply a best practice that should always be followed).

In any case, think about the actual exploit scenarios and impacts of the vulnerability when performing the risk assessment - don't blindly trust the output of your scanning tool.

Tampering and Reverse Engineering

In the context of mobile apps, reverse engineering is the process of analyzing the compiled app to extract knowledge about its inner workings. It is akin to reconstructing the original source code from the bytecode or binary code, even though this doesn't need to happen literally. The main goal in reverse engineering is *comprehending* the code.

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

Reverse engineering and tampering techniques have long belonged to the realm of crackers, modders, malware analysts, and other more exotic professions. For "traditional" security testers and researchers, reverse engineering has been more of a complementary, nice-to-have-type skill that wasn't all that useful in 99% of day-to-day work. But the tides are turning: Mobile app black-box testing increasingly requires testers to disassemble compiled apps, apply patches, and tamper with binary code or even live processes. The fact that many mobile apps implement defenses against unwelcome tampering doesn't make things easier for us.

Mobile security testers should be able to understand basic reverse engineering concepts. It goes without saying that they should also know mobile devices and operating systems inside out: the processor architecture, executable format, programming language intricacies, and so forth.

Reverse engineering is an art, and describing every available facet of it would fill a whole library. The sheer range of techniques and possible specializations is mind-blowing: One can spend years working on a very specific, isolated sub-problem, such as automating malware analysis or developing novel de-obfuscation methods. Security testers are generalists: To be effective reverse engineers, they must be able filter through the vast amount of information to build a workable methodology.

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

Why You Need It

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

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

2. To enhance static analysis in black-box security testing. In a black-box test, static analysis of the app bytecode or binary code is helpful for getting a better understanding of what the app is doing. It also enables you to identify certain flaws, such as credentials hardcoded inside the app.

3. To assess resilience against reverse engineering. Apps that implement the software protection measures listed in MASVS-R should be resilient against reverse engineering to a certain degree. In this case, testing the reverse engineering defenses ("resiliency assessment") is part of the overall security test. In the resilience assessment, the tester assumes the role of the reverse engineer and attempts to bypass the defenses.

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

Ultimately, the reverse engineer always wins.

This is even more true in the mobile world, where the reverse engineer has a natural advantage: The way mobile apps are deployed and sandboxed is more restrictive by design, so it is simply not feasible to include the rootkit-like functionality often found in Windows software (e.g. DRM systems). At least on Android, you have a much higher degree of control over the mobile OS, giving you easy wins in many situations (assuming you know how to use that power). On iOS, you get less control - but defensive options are even more limited.

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

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

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

Basic Tampering Techniques

Binary Patching

Patching means making changes to the compiled app - e.g. changing code in binary executable file(s), modifying Java bytecode, or tampering with resources. The same process is known as *modding* in the mobile game hacking scene. Patches can be applied in any number of ways, from decompiling, editing and re-assembling an app, to editing binary files in a hex editor - anything goes (this rule applies to all of reverse engineering). We'll give some detailed examples for useful patches in later chapters.

One thing to keep in mind is that modern mobile OSes strictly enforce code signing, so running modified apps is not as straightforward as it used to be in traditional Desktop environments. Yep, security experts had a much easier life in the 90s! Fortunately, this is not all that difficult to do if you work on your own device - it simply means that you need to resign the app, or disable the default code signature verification facilities to run modified code.

Code Injection

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

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

However, you can also instrument apps with Substrate by using it to inject Cycript, the programming environment (a.k.a. "Cycript-to-JavaScript" compiler) authored by Saurik of Cydia fame. To complicate things even more, Frida's authors also created a fork of Cycript named "[frida-cycript](#)" that replaces Cycript's runtime with a Frida-based runtime called

Mjølner. This enables Cycript to run on all the platforms and architectures maintained by frida-core (if you are confused now don't worry, it's perfectly OK to be). The release was accompanied by a blog post by Frida's developer Ole titled "Cycript on Steroids" which [Saurik wasn't very fond of](#).

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

Static and Dynamic Binary Analysis

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

Using Disassemblers and Decompilers

Disassemblers and decompilers allow you to translate an app binary code or byte-code back into a more or less understandable format. In the case of native binaries, you'll usually obtain assembler code matching the architecture which the app was compiled for. Android Java apps can be disassembled to Smali, which is an assembler language for the dex format used by dalvik, Android's Java VM. The Smali assembly is also quite easily decompiled back to Java code.

A wide range of tools and frameworks is available: from expensive but convenient GUI tools, to open source disassembling engines and reverse engineering frameworks. Advanced usage instructions for any of these tools often easily fill a book on their own. The best way to get started though is simply picking a tool that fits your needs and budget and buying a well-reviewed user guide along with it. We'll list some of the most popular tools in the OS-specific "Reverse Engineering and Tampering" chapters.

Debugging and Tracing

In the traditional sense, debugging is the process of identifying and isolating problems in a program as part of the software development life cycle. The very same tools used for debugging are of great value to reverse engineers even when identifying bugs is not the

primary goal. Debuggers enable suspending a program at any point during runtime, inspect the internal state of the process, and even modify the content of registers and memory. These abilities make it *much* easier to figure out what a program is actually doing.

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

Advanced Techniques

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

Like always in hacking, the anything-goes-rule applies: Simply use whatever brings you closer to your goal most efficiently. Every binary is different, and every reverse engineer has their own style. Often, the best way to get to the goal is to combine different approaches, such as emulator-based tracing and symbolic execution, to fit the task at hand. To get started, pick a good disassembler and/or reverse engineering framework and start using them to get comfortable with their particular features and extension APIs. Ultimately, the best way to get better is getting hands-on experience.

Dynamic Binary Instrumentation

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

The [Valgrind README](#) contains specific compilation instructions for Android.

Emulation-based Dynamic Analysis

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

Custom Tooling using Reverse Engineering Frameworks

Even though most professional GUI-based disassemblers feature scripting facilities and extensibility, they sometimes simply not well-suited to solving a particular problem. Reverse engineering frameworks allow you perform and automate any kind of reversing task without the dependence for heavy-weight GUI, while also allowing for increased flexibility. Notably, most reversing frameworks are open source and/or available for free. Popular frameworks with support for mobile architectures include [Radare2](#) and [Angr](#).

Example: Program Analysis using Symbolic / Concolic Execution

In the late 2000s, symbolic-execution based testing has gained popularity as a means of identifying security vulnerabilities. Symbolic "execution" actually refers to the process of representing possible paths through a program as formulas in first-order logic, whereby variables are represented by symbolic values, which are actually entire ranges of values. Satisfiability Modulo Theories (SMT) solvers are used to check satisfiability of those formulas and provide a solution, including concrete values for the variables needed to reach a certain point of execution on the path corresponding to the solved formula.

Typically, this approach is used in combination with other techniques such as dynamic execution (hence the name concolic stems from *concrete* and *symbolic*), in order to tone down the path explosion problem specific to classical symbolic execution. This together with improved SMT solvers and current hardware speeds, allow concolic execution to explore paths in medium size software modules (i.e. in the order of 10s KLOC). However, it also comes in handy for supporting de-obfuscation tasks, such as simplifying control flow graphs. For example, Jonathan Salwan and Romain Thomas have [shown how to reverse engineer VM-based software protections using Dynamic Symbolic Execution](#) (i.e., using a mix of actual execution traces, simulation and symbolic execution).

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

Cryptography for Mobile Apps

The following chapter translates the cryptography requirements of the MASVS into technical test cases. Test cases listed in this chapter are based upon generic cryptographic concepts and are not relying on a specific implementation on iOS or Android. This chapter strives to provide recommendations for static testing methods where possible. However, dynamic testing methods are not generally applicable for the problems discussed below and, correspondingly, are not listed here.

Background on cryptography

The primary goal of cryptography is to provide confidentiality, data integrity, and authenticity, even in the face of an attack. Confidentiality is achieved through use of encryption, with the aim of ensuring secrecy of the contents. Data integrity deals with maintaining and ensuring consistency of data and detection of tampering/modification. Authenticity ensures that the data comes from a trusted source. Since this is a testing guide and not a cryptography textbook, the following paragraphs provide only a very limited outline of relevant techniques and their usages in the context of mobile applications.

- Encryption ensures data confidentiality by using special algorithms to convert plaintext data into cipher text, which does not reveal any information about the original content. Plaintext data can be restored from the cipher text through decryption. Two main forms of encryption are symmetric (or secret key) and asymmetric (or public key). 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. It is fast and suitable for bulk data processing. Since everybody who has access to the key is able to decrypt the encrypted content, they require careful key management.
 - Public-key (or asymmetric) 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 should not 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 is typically only used to encrypt small amounts of data, such as symmetric keys for bulk encryption.
- Hash functions deterministically map arbitrary pieces of data into fixed-length values. It is typically easy to compute the hash, but difficult (or impossible) to determine the original input based on the hash. Cryptographic hash functions additionally guarantee that even small changes to the input data result in large changes to the resulting hash

values. Cryptographic hash functions are used for integrity verification, but do not provide authenticity guarantees.

- Message Authentication Codes, or 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 will be able to generate a valid MAC. The most commonly used type of MAC, called HMAC, relies on hashing as the underlying cryptographic primitive. As a rule, the full name of an HMAC algorithm also includes the name of the underlying hash, e.g. - HMAC-SHA256.
- Signatures combine asymmetric cryptography (i.e. - 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, or KDFs, are often confused with password hashing functions. KDFs do have many useful properties for password hashing, but were created with different purposes in mind. In context of mobile applications, it is the password hashing functions that are typically meant for protecting stored passwords.

Two uses of cryptography are covered in other chapters:

- Secure communications. TLS (Transport Layer Security) uses most of the primitives named above, as well a number of others. It is covered in the “Testing Network Communication” chapter.
- Secure storage. This chapter includes high-level considerations for using cryptography for secure data storage, and specific content for secure data storage capabilities will be found in OS-specific data storage chapters.

Testing for Custom Implementations of Cryptography

Overview

The use of non-standard or custom built cryptographic algorithms is dangerous because a determined attacker may be able to break the algorithm and compromise data that has been protected. Implementing cryptographic functions is time consuming, difficult and very likely to fail. Instead well-known algorithms that were already proven to be secure should be used. All mature frameworks and libraries offer cryptographic functions that should also be used when implementing mobile apps.

Static Analysis

Carefully inspect all the cryptographic methods used within the source code, especially those which are directly applied to sensitive data. All cryptographic operations (see the list in the introduction section) should come from the standard providers (for standard APIs for Android and iOS, see cryptography chapters for the respective platforms). Any cryptographic invocations which do not invoke standard routines from known providers should be candidates for closer inspection. Pay close attention to seemingly standard but modified algorithms. Remember that encoding is not encryption! Any appearance of bit manipulation operators like XOR (exclusive OR) might be a good sign to start digging deeper.

Remediation

Do not develop custom cryptographic algorithms, as it is likely they are prone to attacks that are already well-understood by cryptographers. Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and use well-tested implementations.

References

OWASP Mobile Top 10 2016

- M6 - Broken Cryptography

OWASP MASVS

- V3.2: "The app uses proven implementations of cryptographic primitives"

CWE

- CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Testing for Insecure and/or Deprecated Cryptographic Algorithms

Overview

Many cryptographic algorithms and protocols should not be used because they have been shown to have significant weaknesses or are otherwise insufficient for modern security requirements. Previously thought secure algorithms may become insecure over time. It is therefore important to periodically check current best practices and adjust configurations accordingly.

Static Analysis

The source code should be checked that cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG. Please note, that an algorithm that was certified, e.g., by NIST, can also become insecure over time. A certification does not replace periodic verification of an algorithm's soundness. All of these should be marked as insecure and should not be used and removed from the application code base.

Inspect the source code to identify the instances of cryptographic algorithms throughout the application, and look for known weak ones, such as:

- [DES](#), [3DES](#)
- RC2
- RC4
- [BLOWFISH](#)
- MD4
- MD5
- SHA1 and others.

On Android (via Java Cryptography APIs), selecting an algorithm is done by requesting an instance of the `Cipher` (or other primitive) by passing a string containing the algorithm name. For example, `Cipher cipher = Cipher.getInstance("DES");`. On iOS, algorithms are typically selected using predefined constants defined in `CommonCryptor.h`, e.g., `kCCAlgorithmDES`. Thus, searching the source code for the presence of these algorithm names would indicate that they are used. Note that since the constants on iOS are numeric, an additional check needs to be performed to check whether the algorithm values sent to CCCrypt function map to one of the deprecated/insecure algorithms.

Other uses of cryptography require careful adherence to best practices:

- For encryption, use a strong, modern cipher with the appropriate, secure mode and a strong key. Examples:
 - 256-bit key AES in GCM mode (provides both encryption and integrity verification.)
 - 4096-bit RSA with OAEP padding.
 - 224/256-bit elliptic curve cryptography.
- Do not use known weak algorithms. For example:
 - AES in ECB mode is not considered secure, because it leaks information about the structure of the original data.
 - Several other AES modes can be weak.
- RSA with 768-bit and weaker keys can be broken. Older PKCS#1 padding leaks information.
- Rely on secure hardware, if available, for storing encryption keys, performing cryptographic operations, etc.

Remediation

Periodically ensure that the cryptography has not become obsolete. Some older algorithms, once thought to require years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once considered as strong.

Examples of [currently recommended algorithms](#)

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

See also the following best practice documents for recommendations:

- ["Commercial National Security Algorithm Suite and Quantum Computing FAQ"](#)
- [NIST recommendations \(2016\)](#)
- [BSI recommendations \(2017\)](#)

References

OWASP Mobile Top 10

- M6 - Broken 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"

CWE

- CWE-326: Inadequate Encryption Strength
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Testing for Insecure Cryptographic Algorithm Configuration and Misuse

Overview

Choosing strong cryptographic algorithm alone is not enough. Often security of otherwise sound algorithms can be affected through their configuration. Most prominent for cryptographic algorithms is the selection of their used key length.

Static Analysis

Through source code analysis the following non-exhausting configuration options should be checked:

- Cryptographic salt, which should be at least the same length as hash function output
- Reasonable choice of iteration counts when using password derivation functions
- IVs being random and unique
- Fit-for-purpose block encryption modes
- Key management being done properly

Remediation

Periodically ensure that used key length fulfill [accepted industry standards](#). Also verify the used [security "Crypto" provider on the Android platform](#).

References

OWASP Mobile Top 10

- M6 - Broken 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"

CWE

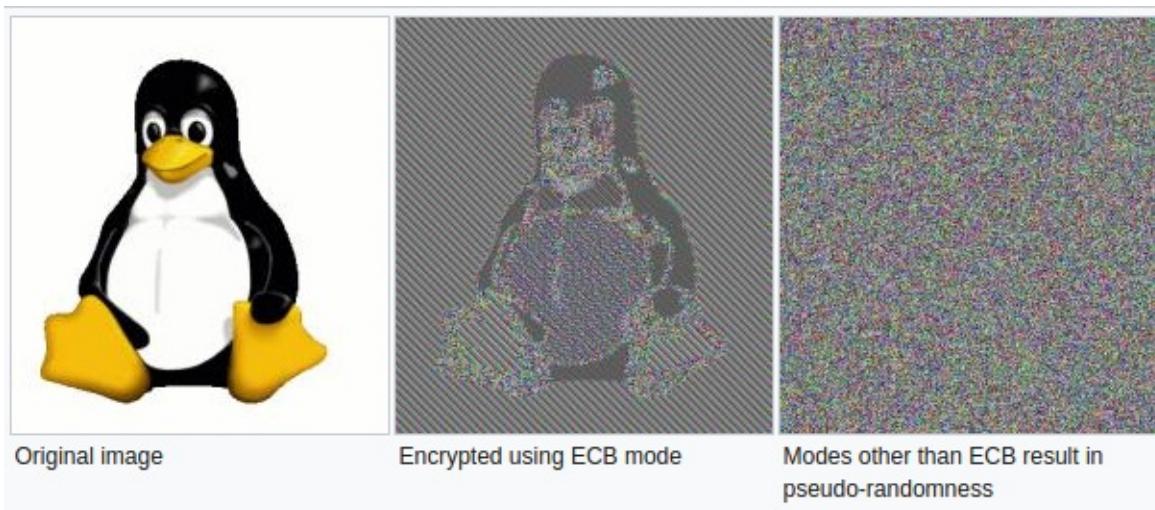
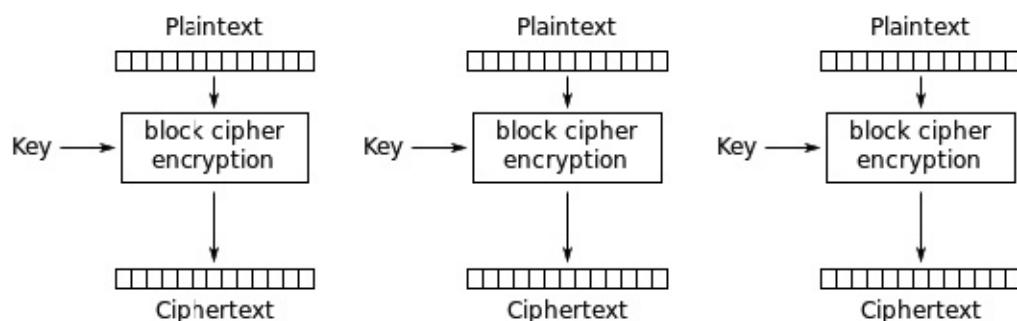
- CWE-326: Inadequate Encryption Strength
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Testing for Usage of ECB Mode

Overview

As the name implies, block-based encryption is performed upon discrete input blocks, e.g., 128 bit blocks when using AES. If the plain-text is larger than the block-size, it is internally split up into blocks of the given input size and encryption is performed upon each block. The so called block mode defines, if the result of one encrypted block has any impact upon subsequently encrypted blocks.

The [ECB \(Electronic Codebook\)](#) encryption mode should not be used, as it is basically divides the input into blocks of fixed size and each block is encrypted separately. For example, if an image is encrypted utilizing the ECB block mode, then the input image is split up into multiple smaller blocks. Each block might represent a small area of the original image. Each of which is encrypted using the same secret input key. If input blocks are similar, e.g., each input block is just a white background, the resulting encrypted output block will also be the same. While each block of the resulting encrypted image is encrypted, the overall structure of the image will still be recognizable within the resulting encrypted image.



Static Analysis

Use the source code to verify the used block mode. Especially check for ECB mode, e.g.:

```
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
```

Remediation

Use an established block mode that provides a feedback mechanism for subsequent blocks, e.g. Counter Mode (CTR). For storing encrypted data it is often advisable to use a block mode that additionally protects the integrity of the stored data, e.g. Galois/Counter Mode (GCM). The latter has the additional benefit that the algorithm is mandatory for each TLSv1.2 implementation -- thus being available on all modern platforms.

Also consult the [NIST guidelines on block mode selection](#).

References

OWASP Mobile Top 10

- M6 - Broken 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"

CWE

- CWE-326: Inadequate Encryption Strength
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Testing for Hardcoded Cryptographic Keys

Overview

The security of symmetric encryption and keyed hashes (MACs) is highly dependent upon the secrecy of the used secret key. If the secret key is disclosed, the security gained by encryption/MACing is rendered naught. This mandates, that the secret key is protected and should not be stored together with the encrypted data.

Static Analysis

The following checks should be performed against the source code:

- Ensure that no keys/passwords are hard coded and stored within the source code. Pay special attention to any 'administrative' or backdoor accounts enabled in the source code. Storing a fixed salt within the app or password hashes may cause problems too.
- Ensure that no obfuscated keys or passwords are in the source code. Obfuscation is

easily bypassed by dynamic instrumentation and in principle does not differ from hard coded keys.

- If the app is using two-way SSL (i.e. there is both server and client certificate validated) check if:
 - the password to the client certificate is not stored locally, it should be in the Keychain
 - the client certificate is not shared among all installations (e.g. hard coded in the app)
- If the app relies on an additional encrypted container stored in app data, ensure how the encryption key is used:
 - if key wrapping scheme is used, ensure that the master secret is initialized for each user, or container is re-encrypted with new key;
 - check how password change is handled and specifically, if you can use master secret or previous password to decrypt the container.

Mobile operating systems provide a specially protected storage area for secret keys, commonly named key stores or key chains. Those storage areas will not be part of normal backup routines and might even be protected by hardware means. The application should use this special storage locations/mechanisms for all secret keys.

Remediation

The secure and protected storage mechanisms provided by the OS should be used to store secret keys:

- [iOS: Managing Keys, Certificates, and Passwords](#)
- [Android: The Android Keystore System](#)
- [Android: Hardware-backed Keystore](#)

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."

CWE

- CWE-321 - Use of Hard-coded Cryptographic Key

Testing Key Generation Techniques

Overview

Cryptographic algorithms -- such as symmetric encryption or MACs -- expect a secret input of a given size, e.g. 128 or 256 bit. A native implementation might use the user-supplied password directly as an input key. There are a couple of problems with this approach:

- If the password is smaller than the key, then not the full key-space is used (the rest is padded, sometimes even with spaces)
- A user-supplied password will realistically consist mostly of displayable and pronounceable characters. So instead of the full entropy, i.e. 2^8 when using ASCII, only a small subset is used (approx. 2^6).
- If two users select the same password an attacker can match the encrypted files. This opens up the possibility of rainbow table attacks.

Static Analysis

Use the source code to verify that no password is directly passed into an encryption function.

Remediation

Pass the user-supplied password into a salted hash function or KDF; use its result as key for the cryptographic function.

References

OWASP Mobile Top 10

- M6 - Broken 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"

CWE

- CWE-330 - Use of Insufficiently Random Values

Testing for Stored Passwords

Overview

Normal hashes are optimized for speed, e.g., optimized to verify large media in short time. For password storage this property is not desirable as it implies that an attacker can crack retrieved password hashes (using rainbow tables or through brute-force attacks) in a short time. For example, when the insecure MD5 hash has been used, an attacker with access to eight high-level graphics cards [can test 200.3 Giga-Hashes per second](#), which will break the hash of weak passwords easily. A solution to this are Key-Derivation Functions (KDFs) that have a configurable calculation time. While this imposes a larger performance overhead this is negligible during normal operation but prevents brute-force attacks. Recently developed key derivation functions such as Argon2 or scrypt have been hardened against GPU-based password cracking.

Static Analysis

Use the source code to determine how the hash is calculated.

Remediation

Use an established key derivation function such as PBKDF2 ([RFC 2898](#)), [Argon2](#), [bcrypt](#) or scrypt ([RFC 7914](#)).

References

OWASP Mobile Top 10

- M6 - Broken 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"

CWE

- CWE-916 - Use of Password Hash With Insufficient Computational Effort

Android Platform Overview

This chapter introduces Android from an architecture point of view. It covers four key areas:

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 on the Android platform.

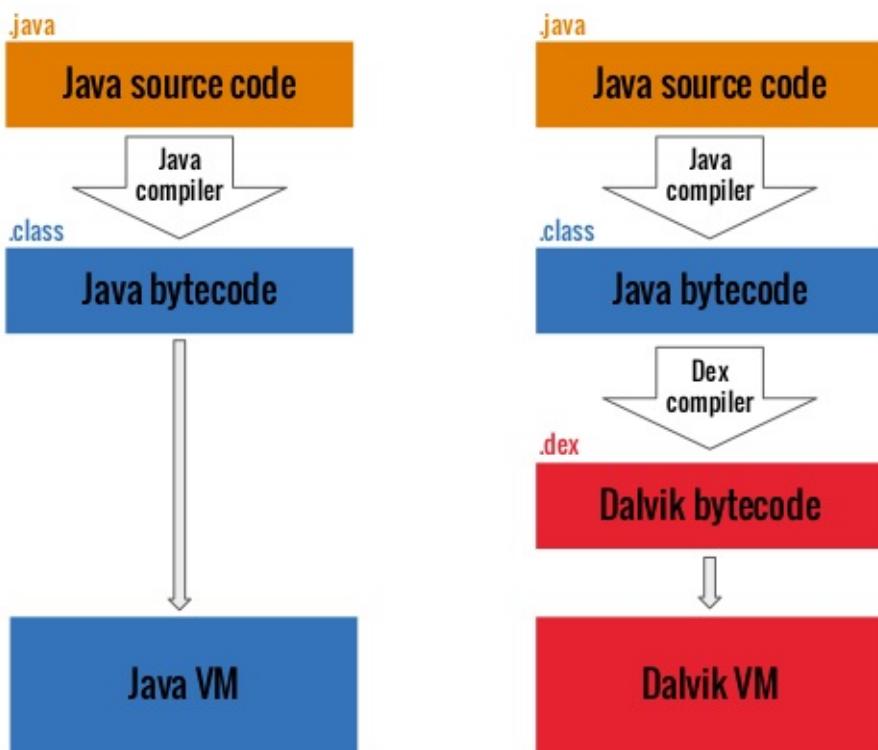
Android Security Architecture

Android is a Linux-based open source platform build by Google. Is is found on many commonly used devices including mobile phones and tablets, wearables, and "smart" devices like TVs. 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.

The software stack of Android comprises different layers. Each layer defines certain behavior and offers specific services to the layer above.



On the lowest level, Android uses the Linux Kernel upon which the core operating system is built. The hardware abstraction layer defines a standard interface for hardware vendors. HAL (Hardware Abstraction Layer) implementations are packaged into shared library modules (.so files). These modules will be loaded by the Android system at the appropriate time. The Android Runtime consists of the core libraries and the Dalvik VM (Virtual Machine). Apps are most often implemented in Java and compiled in Java class files. However since Android integrates a Dalvik VM, not JVM, the Java class files are then compiled again into the dex format. The dex files are packed into APK (a ZIP archive containing all resources, including the executable) and then unpacked and executed within the Dalvik VM. In the following image you can see the differences between the normal process of compiling and running a typical project in Java vs the process in Android using Dalvik VM.



With Android 4.4 (KitKat) the successor of Dalvik VM was introduced, called Android Runtime (ART). The advent of Android 5.0 (Lollipop) in November 2014 truly enabled general use. In KitKat, ART was only available in the 'Developer' menu to those who wanted to try it explicitly. When no user action was taken to modify the normal behavior of the mobile, Dalvik was used.

In Android, apps are executed into their own environment in a Virtual Machine (VM) called Dalvik and located in the runtime environment. Each VM emulates the whole mobile and gives access to relevant resources from the Linux kernel while controlling this access. Apps do not have direct access to hardware resources, and their execution environments are therefore separate from each other. This allows fine-grained control over resources and apps: for instance, when an app crashes, it does not prevent other apps from working and only their environment and the app itself have to be restarted. Also, the fact apps are not run directly on the mobile hardware allow the use of the same app (same bytecode) on different architectures (e.g. ARM, x86) as the VM emulates a common hardware for the app. At the same time, the VM controls the maximum amount of resources provided to an app, preventing one app from using all resources while leaving only few resources to others. In Android, apps are installed as bytecode (.dex files, see "Android Application Overview" section). In Dalvik, this bytecode is compiled at execution time into machine language suiting the current processor: such a mechanism is known as Just In Time (JIT). However, this means that such compiling is done every time an app is executed on a given mobile. As a consequence, Dalvik has been improved to compile an app only once, at installation time (the principle is called AOT, a.k.a. Ahead Of Time): ART was born, and compilation was

required only once, saving precious time at execution time (the execution time of an app may be divided by 2). Another benefit was that ART was consuming less power than Dalvik, allowing the user to use the mobile device and its battery longer.

Android Users and Groups

Even though the Android operating system is based on the Linux, it does utilize user accounts in the same way other Unix-like systems do. For instance, it does not have a `/etc/passwd` file containing a list of users in the system. Instead, Android utilizes the multi-user support of the Linux kernel to achieve application sandboxing, by running each application under a separate user (with some exceptions).

The file [system/core/include/private/android_filesystem_config.h](#) shows the complete list of the predefined users and groups used for system processes. UIDs (userIDs) for other applications are added as they are installed on the system. For more details you can check this [overview of Android application sandbox..](#)

File below depicts some of the users defined for Android Nougat:

```
#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

As already mentioned, Android apps are written in Java and compiled into a dex bytecode. System resources are not accessed directly. Instead, apps interact with system services using the Android Framework, an abstraction layer that offers high-level API easily usable from Java. For the most part, these services are used via normal Java method calls, and are translated to IPC calls to system services running in the background. Examples for system services include:

- * Connectivity (Wifi, Bluetooth, NFC, ...)
- * Files
- * Cameras
- * Geolocation (GPS)
- * Microphone

The framework also offers common security functions such as cryptography. At the time of writing this guide, the current version of Android is 7.1 (Nougat), API level 25.

APIs have evolved a lot since the first Android version (September 2008). Critical bug fixes and security patches are usually propagated several versions back. The oldest Android version supported at the time of writing this guide, is 4.4 (KitKat), API level 19.

Noteworthy API versions are:

Android 4.2 Jelly Bean (API 16) in November 2012 (introduction of SELinux)
Android 4.3 Jelly Bean (API 18) in July 2013 (SELinux becomes enabled by default)
Android 4.4 KitKat (API 19) in October 2013 (several new APIs and ART is introduced)
Android 5.0 Lollipop (API 21) in November 2014 (ART by default and many other new features)
Android 6.0 Marshmallow (API 23) in October 2015 (many new features and improvements, including granting fine-grained permissions at run time and not all or nothing at installation time)
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)

Apps can be installed on an Android device from a variety of sources: locally through USB, from Google's official store (Google Play Store) or from alternative stores.

App Folder Structure

Android apps installed (from Google Play Store or from external sources) are located at `/data/app/`. Since this folder cannot be listed without root. An alternative method must be used to get the exact name of the APK. To list all installed APKs, the Android Debug Bridge (adb) can be used. ADB allows a tester to directly interact with the real device. E.g. to gain access to a console on the device to issue further commands, list installed packages, start/stop processes, etc. To do so, the device has to have USB-Debugging enabled (can be found under developer settings) and has to be connected via USB. Alternatively, you can configure the device so that ADB can be [connected over TCP/IP](#). As in both cases ADB behaves the same, we further assume the default case where connection over USB is established.

Once USB-Debugging is enabled, the connected devices can be viewed with the following command:

```
$ adb devices
List of devices attached
BAZ50RFARKOZYDFA    device
```

Then the following command lists all installed apps and their locations:

```
$ adb shell pm list packages -f
package:/system/priv-app/MiuiGallery/MiuiGallery.apk=com.miui.gallery
package:/system/priv-app/Calendar/Calendar.apk=com.android.calendar
package:/system/priv-app/BackupRestoreConfirmation/BackupRestoreConfirmation.apk=com.android.backupconfirm
```

To pull one of those apps from the phone, the following command can be used:

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

This file only contains the “installer” of the app. This is the app the developer uploaded to the store. The local data of the app is stored 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 to cache app data on runtime including WebView caches.
- **code_cache**: The location of the application specific cache directory on the filesystem designed for storing cached code. On devices running Lollipop or later, the system will delete any files stored in this location both when your specific application is upgraded, and when the entire platform is upgraded. This location is optimal for storing compiled or optimized code generated by your application at runtime. Apps require no extra permissions to read or write to the returned path, since this path lives in their private storage.
- **databases**: This folder stores sqlite database files generated by the app at runtime, e.g. to store user data.
- **files**: This folder is used to store files that are created in the App when using the internal storage.
- **lib**: This folder used to store native libraries written in C/C++. These libraries can have file extension as .so, .dll (x86 support). The folder contains subfolders for the platforms the app has native libraries for:

- armeabi: Compiled code for all ARM based processors only
- armeabi-v7a: Compiled code for all ARMv7 and above based processors only
- arm64-v8a: Compiled code for all ARMv8 arm64 and above based processors only
- x86: Compiled code for x86 processors only
- x86_64: Compiled code for x86_64 processors only
- mips: Compiled code for MIPS processors only
- **shared_prefs:** This folder is used to store the preference file generated by an app at runtime to save current state of the app including data, configuration, session, etc. The file format is XML.

APK Structure

An app on Android is a file with the extension .apk. This file is a signed zip-file which contains all the application's resources, byte code, etc. When unzipped the following directory structure can usually be identified:

```
$ 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 app's package name, target and min API version, app configuration, components, user-granted permissions, etc.
- **META-INF:** This folder contains metadata of the app:
 - MANIFEST.MF: Stores hashes of app resources.
 - CERT.RSA: The certificate(s) of the app.
 - CERT.SF: The list of resources and SHA-1 digest of the corresponding lines in the MANIFEST.MF file.
- **assets:** A directory containing app assets (files used within the Android App like XML, Java Script or pictures) which can be retrieved by the AssetManager.
- **classes.dex:** The classes compiled in the DEX file format understandable by the Dalvik virtual machine/Android Runtime. DEX is Java Byte Code for Dalvik Virtual Machine. It is optimized for running on small devices.
- **lib:** A directory containing libraries that are part of the APK, for example 3rd party libraries that are not part of the Android SDK.
- **res:** A directory containing resources not compiled into resources.arsc.
- **resources.arsc:** A file containing precompiled resources, such as XML files for the

layout.

Some resources inside the APK are compressed using non-standard algorithms (e.g. the `AndroidManifest.xml`). This means that simply unzipping the file won't reveal all information. A better way is to use the tool 'apktool' to unpack and uncompress the files. The following is a list of the files contained in the apk:

```
$ 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`: This file is not compressed anymore and can be opened in a text editor.
- `apktool.yml` : This file contains information about the output of apktool.
- `assets`: A directory containing app assets (files used within the Android App like XML, Java Script or pictures) which can be retrieved by the AssetManager.
- `lib`: A directory containing libraries that are part of the APK, for example 3rd party libraries that are not part of the Android SDK.
- `original`: This folder contains the `MANIFEST.MF` file which stores meta data about the contents of the JAR and signature of the APK. The folder is also named as `META-INF`.
- `res`: A directory containing resources not compiled into `resources.arsc`.
- `smali`: A directory containing the disassembled Dalvik bytecode in Smali. Smali is a human readable representation of the Dalvik executable.

Linux UID/GID of Normal Applications

A newly installed app on Android is assigned a new UID. Generally apps are assigned UIDs in the range of 10000 (AID_APP) and 99999. Android apps receive a user name based on their UID. For example, apps with UID 10188 receive the user name u0_a188. If an app requested some permissions and they are granted, the corresponding group ID is added to the process of the app. For example, the user ID of the app below is 10188. It belongs to group ID 3003 (inet). That is the group related to android.permission.INTERNET permission. The result 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 are 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>
```

An important aspect of Android security is that all apps have the same level of privileges. Both native and third-party apps are built on the same APIs and run in similar environments. Apps are not executed at 'root' instead they hold user level privileges. This restricts the actions apps can perform as well as access to some parts of the file system. In order to be able to execute an app with 'root' privileges (inject packets in a network, run interpreters like Python etc.) mobiles need to be rooted.

Zygote

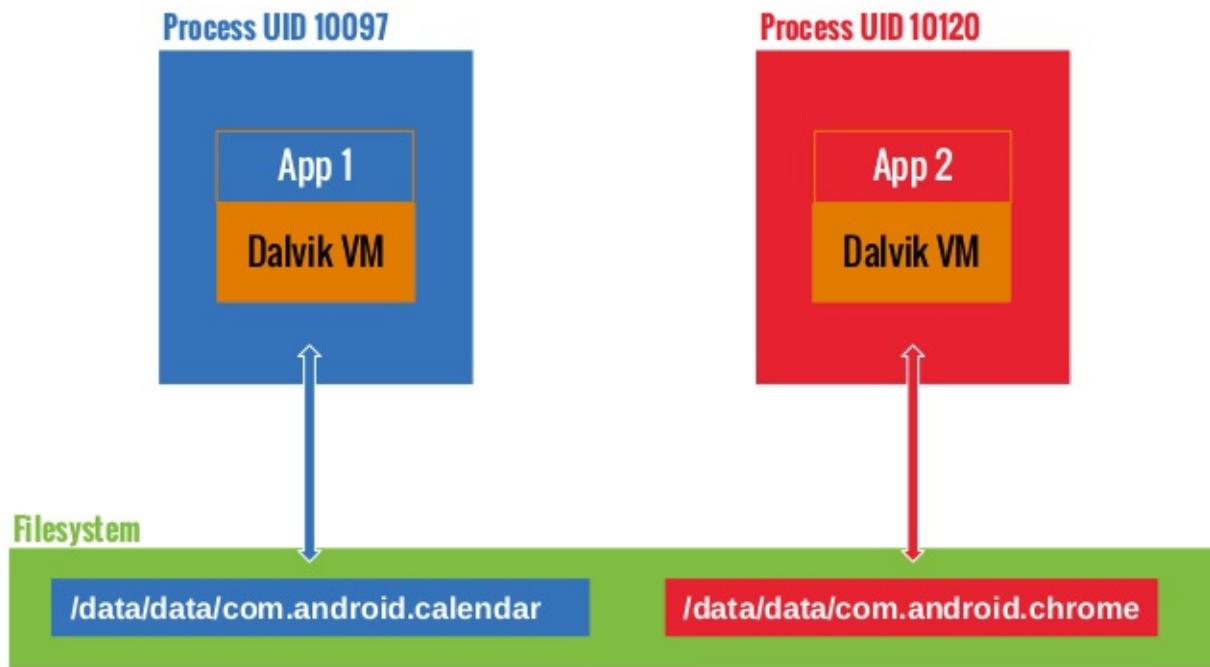
A process called `zygote` starts up during the [Android initialization process](#). Zygote is a system service used to launch apps. It opens up a socket in `/dev/socket/zygote` and listens on it for requests to start new applications. The Zygote process is a "base" process that contains all the core libraries that are needed by any app. When Zygote receives a request over its listening socket, it forks a new process which then loads and executes the app-specific code.

The App Sandbox

Apps are executed in the Android Application Sandbox thus enforcing isolation of app data and code execution from other apps on the device. This adds an additional layer of security.

When installing a new app (From Google Play Store or External Sources), a new folder is created in the file system in the path `/data/data/`. This folder is going to be the private data folder for that particular app.

Since every app has its own unique Id, Android separates app data folders configuring the mode *read* and *write* only to the owner of the app.



In this example, the Chrome and Calendar app are completely segmented with different UID and different folder permissions.

We can confirm this by looking at the filesystem permissions created for each folder:

<code>drwx----- 4 u0_a97</code>	<code>u0_a97</code>	<code>4096 2017-01-18 14:27 com.androi</code>
<code>d.calendar</code>		
<code>drwx----- 6 u0_a120</code>	<code>u0_a120</code>	<code>4096 2017-01-19 12:54 com.androi</code>
<code>d.chrome</code>		

However, if two apps are signed with the same certificate and explicitly share the same user ID (by including the `sharedUserId` in their `AndroidManifest.xml`) they can access each others data directory. See the following example on how this is achieved in the Nfc app:

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

App Components

Android apps are made of several high-level components that make up their architectures.

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.

Application Life Cycle

Android apps have their own lifecycles under the control of the operating system. Therefore, apps need to listen to state changes and react accordingly. For instance, when the system needs resources, apps may be killed. The system selects the ones that will be killed according to the app priority: active apps have the highest priority (actually the same as Broadcast Receivers), followed by visible ones, running services, background services, and last useless processes (for instance apps that are still open but have not been in use for a significant time).

Apps implement several event managers to handle events: for example, the `onCreate` handler implements what has to be done on app creation and will be called on that event. Other managers include `onLowMemory`, `onTrimMemory` and `onConfigurationChanged`.

Manifest

Every app must have a manifest file, which embeds content in XML format. The name of this file is standardized as `AndroidManifest.xml` and is the same for every app. It is located in the root tree of the `.apk` file in which the app is published.

A manifest file describes the app structure as well as its exposed components (activities, services, content providers and intent receivers) and requested permissions. Permission filters for IPC can be implemented to refine the way the app will interact with the outside world. The manifest file also contains general metadata about the app, like its icon, its

version number and the theme it uses for User Experience (UX). It may list other information like the APIs it is compatible with (minimal, targeted and maximal SDK version) and the [kind of storage it can be installed in \(external or internal\)](#)

Here is an example of a manifest file, including the package name (the convention is to use a url in reverse order, but any string can be used). It also lists the app version, relevant SDKs, required permissions, exposed content providers, used broadcast receivers with intent filters as well as 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>
```

A manifest is a text file and can be edited within Android Studio (the preferred IDE for Android development). A lot more useful options can be added to manifest files, which are listed in the official [Android Manifest file documentation](#).

Activities

Activities make up the visible part of any app. One activity exists per screen (e.g. user interface) so an app with three different screens is implementing three different activities. This allows the user to interact with the system (get and enter information). Activities are declared by extending the Activity class. They contain all user interface elements: fragments, views and layouts.

Activities implement manifest files. Each activity needs to be declared in the app manifest with the following syntax:

```
<activity android:name="ActivityName">  
</activity>
```

When activities are not declared in manifests, they cannot be displayed and would raise an exception.

In the same way as apps do, activities also have their own lifecycle and need to listen to system changes in order to handle them accordingly. Activities can have the following states: active, paused, stopped and inactive. These states are managed by 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 app developers. This is the place where most user interface components are declared and initialized. onDestroy may be overridden as well in case some resources need to be explicitly released (like network connections or connections to databases) or if specific actions need to take place at the end of the app.

Fragments

Basically, a fragment represents a behavior or a portion of user interface in an Activity. Fragments have been introduced in Android with version Honeycomb 3.0 (API level 11).

User interfaces are made of several elements: views, groups of views, fragments and activities. As for them, fragments are meant to encapsulate parts of the interface to make reusability easier and better adapt to different size of screens. Fragments are autonomous entities in that they embed all they need to work in themselves (they have their own layout, own buttons etc.). However, they must be integrated in activities to become useful: fragments cannot exist on their own. They have their own life cycle, which is tied to the one of the activity that implements them. As they have their own life cycle, the Fragment class contains event managers, that can be redefined or extended. Such event managers can be onAttach, onCreate, onStart, onDestroy and onDetach. Several others exist; the reader should refer to the [Android Fragment specification](#) for more details.

Fragments can be implemented easily by extending the Fragment class provided by Android:

```
public class myFragment extends Fragment {  
    ...  
}
```

Fragments don't need to be declared in manifest files as they depend on activities.

In order 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 simply with the following:

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

Fragments do not necessarily have a user interface: they can be a convenient and efficient way to manage background operations dealing with user interface in an app. For instance when a fragment is declared as persistent while its parent activity may be destroyed and created again.

Inter-Process Communication

As we know, every process on Android has its own sandboxed address space. Inter-process communication (IPC) facilities enable apps to exchange signals and data in a (hopefully) secure way. Instead of relying on the default Linux IPC facilities, IPC on Android is done through Binder, a custom implementation of OpenBinder. Most Android system services, as well as all high-level IPC services, depend on Binder.

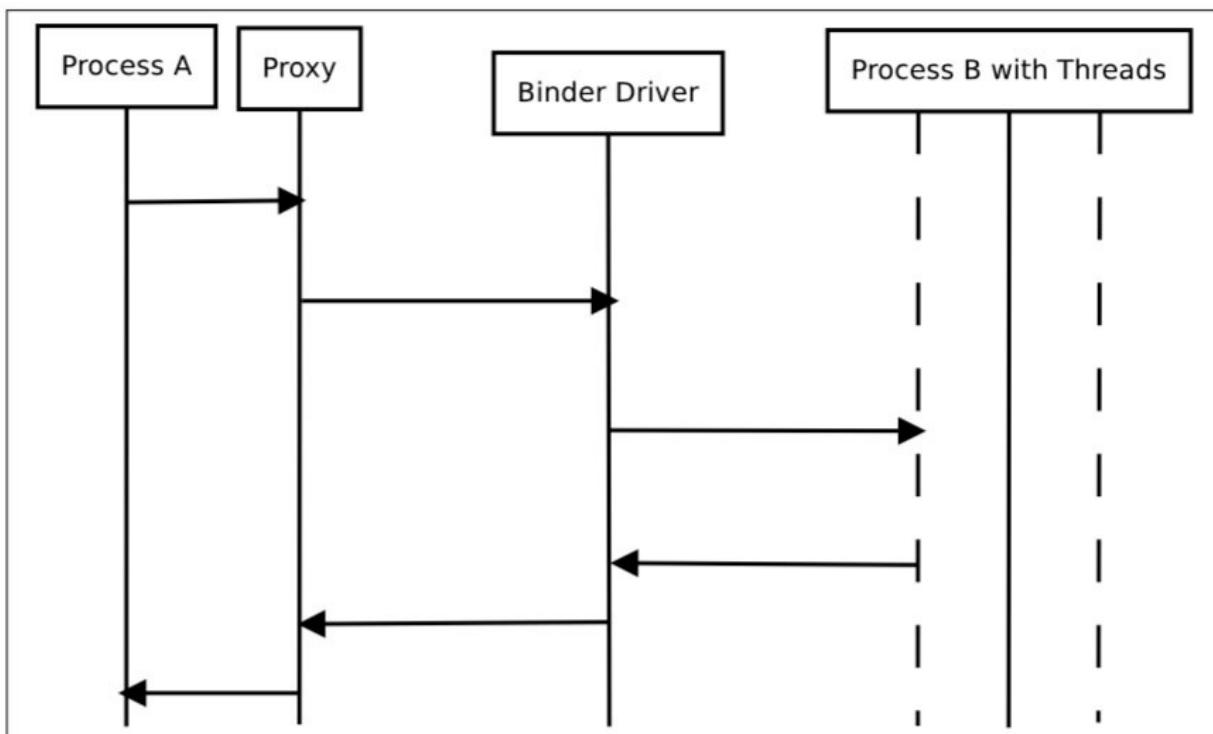
The term *Binder* stands for a lot of different things, including:

- Binder Driver - The kernel-level driver
- Binder Protocol - Low-level ioctl-based protocol used to communicate with the binder

driver

- IBinder Interface - well-defined behavior Binder objects implement
- Binder object - Generic implementation of the IBinder interface
- Binder service - Implementation of the Binder object. For example, location service, sensor service,...
- Binder client - An object using the binder service

In the Binder framework, a client-server communication model is used. To use IPC functionality, apps call IPC methods in proxy objects. The proxy object transparently marshalls the call parameters into a *parcel* and sends 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 is responsible for delivering messages to the destination object. From the view of the client app, all of this looks 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 for use by clients. Developers write interfaces for remote services using the Android Interface Descriptor Language (AIDL).

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 using the `addService` and retrieved by name using the `getService` static method in `android.os.ServiceManager`:

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

The list of system services can be queried using 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 a framework for asynchronous communication built on top of binder. This framework enables 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 communication between components 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 to start and carries any 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 various broadcasts for system events, such as when the system boots up or the device starts charging. You can deliver a broadcast to other apps by passing an Intent to `sendBroadcast()` or `sendOrderedBroadcast()`.

Intents are messaging components used between apps and components. They can be used by an app to send information to its own components (for instance, start inside the app a new activity) or to other apps, and may be received from other apps or from the operating system. Intents can be used to start activities or services, run an action on a given set of data, or broadcast a message to the whole system. They are a convenient way to decouple components.

There are two types of Intents. Explicit intents specify the component to start by name (the fully-qualified class name). For instance:

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

Implicit intents are sent to the system with a given action to perform 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 an app's manifest file that specifies the type of intents that 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, if you do not declare any intent filters for an activity, then it can be started only with an explicit intent.

Android uses intents to broadcast messages to apps, like an incoming call or SMS, important information on power supply (low battery for example) or network changes (loss of connection for instance). Extra data may be added to intents (through `putExtra` / `getExtras`).

Here is a short list of intents from the operating system. All constants are defined in the `Intent` class, and the whole list can be found in Android official documentation:

- ACTION_CAMERA_BUTTON
- ACTION_MEDIA_EJECT
- ACTION_NEW_OUTGOING_CALL
- ACTION_TIMEZONE_CHANGED

In order 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 to guarantee sensitive or private data do not leave the app perimeter (geolocation data for instance).

Broadcast Receivers

Broadcast Receivers are components that allow to receive notifications sent from other apps and from the system itself. This way, apps can react to events (either internal, from other apps or from the operating system). They are generally used to update a user interface, start services, update content or create user notifications.

Broadcast Receivers need to be declared in the Manifest file of the app. Any Broadcast Receiver must be associated to an intent filter in the manifest to specify which actions it is meant to listen with which kind of data. If they are not declared, the app will not listen to broadcasted messages. However, apps do not need to be started to receive intents: they are automatically started by the system when a relevant intent is raised.

An example of declaring a Broadcast Receiver with an Intent Filter in a manifest is:

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

When receiving an implicit intent, Android will list all apps that have registered a given action in their filters. If more than one app is matching, then Android will list all those apps and will require the user to make a selection.

An interesting feature concerning Broadcast Receivers is that they can be assigned a priority; this way, an intent will be delivered to all receivers authorized to get them according to their priority.

A Local Broadcast Manager can be used to make sure intents are received only from the internal app, and that any intent from any other app will be discarded. This is very useful to improve security.

Content Providers

Android is using SQLite to store data permanently: as it is in Linux, data is stored in files. SQLite is an open-source, light and efficient technology for relational data storage that does not require much processing power, making it ideal for use in the mobile world. An entire API is available to the developer with specific classes (Cursor, ContentValues, SQLiteOpenHelper, ContentProvider, ContentResolver, ...). SQLite is not run in a separate process from a given app, but it is part of it. By default, a database belonging to a given app is only accessible to this app. However, Content Providers offer a great mechanism to abstract data sources (including databases, but also flat files) for a more easy use in an app; they also provide a standard and efficient mechanism to share data between apps, including native ones. In order to be accessible to other apps, content providers need to be explicitly declared in the Manifest file of the app that will share it. As long as Content Providers are not declared, they are not 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. Whatever the nature of sources is (SQLite database, flat file, ...), the addressing scheme is always the same, abstracting what sources are and offering a unique scheme to the developer. Content providers offer all regular operations on databases: 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 components provided by Android operating system (in the form of the Service class) that will perform tasks in the background (data processing, start intents and notifications, ...), without presenting any kind of user interface. Services are meant to run processing on the long term. Their system priorities are lower than the ones active apps have, but are higher than inactive ones. As such, they are less likely to be killed when the system needs resources; they can also be configured to start again automatically when enough resources become available in case they get killed. Activities are executed in the main app thread. They are great candidates to run asynchronous tasks.

Permissions

Because Android apps are installed in a sandbox and initially it does not have access to neither user information nor access to system components (such as using the camera or the microphone), it provides a system based on permissions where the system has a predefined set of permissions for certain tasks that the app can request. As an example, if you want your app to use the camera on the phone you have to request the camera permission. On Android versions before Marshmallow (API 23) all permissions requested by an app were granted at installation time. From Android Marshmallow onwards the user have to approve some permissions during app execution.

Protection Levels

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

- *Normal*: Is the lower level of protection, it gives apps access to isolated application-level feature, 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 to the user 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 apps embedded in the system image or that were signed using the same certificate as the app that declared the permission.
Example: `android.permission.ACCESS_DOWNLOAD_MANAGER`

Requesting Permissions

Apps can request permissions of protection level Normal, Dangerous and Signature by inserting the XML tag `<uses-permission />` to its Android Manifest file. 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>
```

This will enable the app to read SMS messages at install time (before Android Marshmallow - 23) or will enable the app to ask the user to allow the permission at runtime (Android M onwards).

Declaring Permissions

Any app is able to expose its features or content to other apps installed on the system. It can expose the information openly or restrict it some apps by declaring a permission. The example below shows an app declaring a permission of protection level *signature*.

```
<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>
```

Only apps signed with the same developer certificate can use this permission.

Enforcing Permissions on Android Components

It is possible to protect Android components using permissions. Activities, Services, Content Providers and Broadcast Receivers all can use the permission mechanism to protect its interfaces. *Activities*, *Services* and *Broadcast Receivers* can enforce a permission by entering the attribute `android:permission` inside each 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 bit different. They allow separate permissions for read, write or access the Content Provider using a content URI.

- `android:writePermission` , `android:readPermission` : The developer can set separate permissions to read or write.
- `android:permission` : General permission that will control read and write to the Content Provider.
- `android:grantUriPermissions` : True if the Content Provider can be accessed using a content URI, temporarily overcoming the restriction of other permissions and False, if not.

Signing and Publishing Process

Once an app has been successfully developed, the next step is to publish it to share it with others. However, apps cannot simply be put on a store and shared: for several reasons, they need to be signed. This is a convenient way to ensure that apps are genuine and authenticate them to their authors: for instance, an upgrade to an app will only be possible if the update is signed with the same certificate as the original app. Also, this is a way to allow sharing between apps that are signed with the same certificate when signature-based permissions are used.

Signing Process

During development, apps are signed with an automatically generated certificate. This certificate is inherently insecure and is used for debugging only. Most stores do not accept this kind of certificates when trying to publish, therefore another certificate, with more secure features, has to be created and used.

When an application is installed onto an Android device, the Package Manager verifies that it has been signed with the certificate included in that APK. If the public key in the certificate matches the key used to sign any other APK on the device, the new APK has the option to share a UID with that APK. This facilitates interaction between multiple applications from the same vendor. Alternatively, it is also possible to specify security permissions the Signature protection level, restricting access to applications signed with the same key.

APK Signing Schemes

Android supports two application signing schemes: As of Android 7.0, APKs can be verified using the APK Signature Scheme v2 (v2 scheme) or JAR signing (v1 scheme). For backward compatibility, APKs signed with the v2 signature format can be installed on older

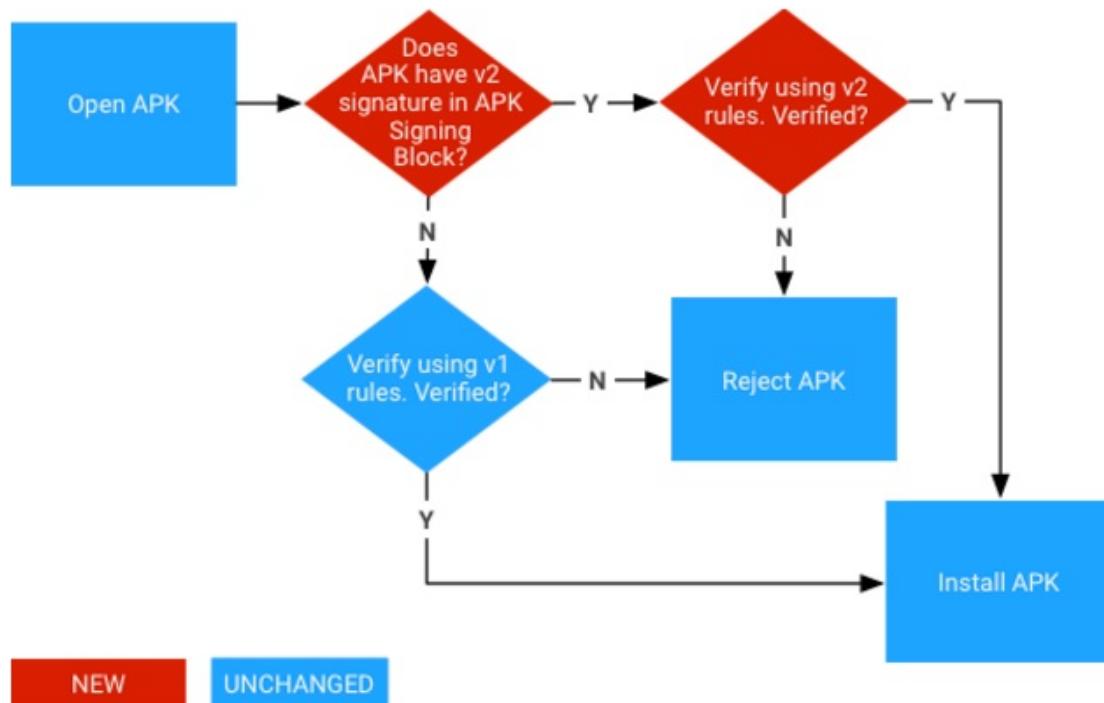
Android devices, as long as these APKs are also v1-signed. [Older platforms ignore v2 signatures and only verify v1 signatures.](#)

JAR Signing (v1 scheme):

In the original version of app signing, the signed APK is actually a standard signed JAR, which must contain exactly the entries listed in `META-INF/MANIFEST.MF`. All entries must be signed using the same certificate. This scheme does not protect some parts of the APK, such as ZIP metadata. The drawback with this scheme is that the APK verifier needs to process untrusted data structures before applying the signature, and discard data not covered by them. Also, the APK verifier must uncompress all compressed files, consuming considerable time and memory.

APK Signature Scheme (v2 scheme)

In 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, v2 scheme performs signature checking across the entire file. This form of APK verification is faster and offers more comprehensive protection against modification.



[APK signature verification process](#)

Creating Your Certificate

Android is using the public/private certificates technology to sign Android apps (.apk files): this permits to establish the authenticity of apps and make sure the originator is the owner of the private key. Such certificates can be self-generated and signed. Certificates are bundles

that contain different information, the most important from security point of view being keys: a public certificate will contain the public key of the user, and a private certificate will contain the private key of the user. Both the public and private certificates are linked together.

Certificates are unique and cannot be generated again: this means that, in case one or the two are lost, it is not possible to renew them with identical ones, therefore updating an app originally signed with a given certificate will become impossible.

The creator of an app can either reuse an existing private / public key pair that already exists and is stored in an available keystore, or generate a new pair.

Key pairs can be generated by the user with the keytool command (example for a key pair generated for my domain ("Distinguished Name"), using the RSA algorithm with a key length of 2048 bits, for 7300 days = 20 years, and that will be stored in the current directory in the secure file 'myKeyStore.jks'):

```
keytool -genkey -alias myDomain -keyalg RSA -keysize 2048 -validity 7300 -keystore myKeyStore.jks -storepass myStrongPassword
```

Safely storing a secret key and making sure it remains secret during its entire lifecycle is of paramount importance, as any other person who would get access to it would be able to publish updates to your apps with content that you would not control (therefore being able to create updates to your apps and add insecure features, access content that is shared using signature-based permissions, e.g. only with apps under your control originally). The trust a user places in an app and its developers is totally based on such certificates, hence its protection and secure management are vital for reputation and Customer retention. This is the reason why secret keys must never be shared with other individuals, and keys are stored in a binary file that can be protected using a password: such files are referred to as 'keystores'; passwords used to protect keystores should be strong and known only by the key creator (-storepass option in the command above, where a strong password shall be provided as an argument).

Android certificates must have a validity period longer than the one of the associated app (including its updates). For example, Google Play will require that the certificate remains valid till at least Oct 22nd, 2033.

Signing an Application

After the developer has generated its own private / public key pair, the signing process can take place. From a high-level point of view, this process is meant to associate the app file (.apk) with the public key of the developer (by encrypting the hash value of the app file with the private key, where only the associated public key can decrypt it to its actual value that anyone can calculate from the .apk file): this guarantees the authenticity of the app (e.g. that

the app really comes from the user who claims it) and enforces a mechanism where it will only be possible to upgrade the app with other versions signed with the same private key (e.g. from the same developer).

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; you should be aware of this and double-check this point in case others are able to access such files, and remove the information if needed.

Apps can be signed from the command line by using the 'apksigner' tool provided in Android SDK (API 24 and higher) or the 'jarsigner' tool from Java JDK in case of earlier Android versions. Details about the whole process can be found in Android official 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 ready for signing ('myUnsignedApp.apk') is going to be signed with a private key from the developer keystore 'myKeyStore.jks' located in the current directory and will become a signed app called 'mySignedApp.apk' ready for release on stores.

Zipalign

The `zipalign` tool should always be used to align an APK file before distribution. This tool aligns all uncompressed data within the APK, such as images or raw files, on 4-byte boundaries, which allows for improved memory management during app runtime. If using apksigner, zipalign must be performed before the APK file has been signed.

Publishing Process

The Android ecosystem is open, and, as such, it is possible to distribute apps from anywhere (your own site, any store, ...). However, Google Play is the more famous, trusted and popular store and is provided by Google itself.

Whereas other vendors may review and approve apps before they are actually published, such things do not happen on Google Play; this way, a short release time can be expected between the moment when the developer starts the publishing process and the moment when the app is available to users.

Publishing an app is quite straightforward, as the main operation is to make the signed .apk file itself downloadable. On Google Play, it starts with creating an account, and then delivering the app through a dedicated interface. Details are available on Android official documentation at <https://developer.android.com/distribute/googleplay/start.html>.

Basic Security Testing on Android

Setting Up Your Testing Environment

Setting up a testing environment can be a challenging task. When performing testing on-site at client premises, the restrictions on the enterprise wireless access points and networks may make dynamic analysis more difficult. Company policies may prohibit use of rooted phones or network testing tools (hardware and software) within the enterprise networks. Apps implementing root detection and other reverse engineering countermeasures may add a significant amount of extra work before further analysis can be performed.

To overcome these and other challenges, the testing team responsible for Android app assessment needs to work together with the app developers and the operation team in order to find a proper solution for an effective testing environment.

This section provides an overview of different methods of testing an Android app and illustrates their limitations. For the reasons stated above, not all testing methods documented here may be applicable for your testing environment. Being able to articulate the reasons for these restrictions will help all the project stakeholders to be on the same page.

Preparation

Security testing involves many invasive tasks such as monitoring and manipulating the network traffic between the mobile app and its remote endpoints, inspecting the app's data files, and instrumenting API calls. Security controls like SSL Pinning and root detection might impede these tasks and slow down testing dramatically.

To overcome these obstacles, it might make sense to request two build variants of the app from the development team. One variant should be provided as a release build to check if the implemented controls like SSL Pinning are working properly or can be easily bypassed. The second variant should also be provided as a debug build that deactivates certain security controls. This approach makes it possible to cover all scenarios and test cases in the most efficient way.

Of course, depending on the scope of the engagement, such approach may not be possible. For a white box test, requesting both production and debug builds will help to go through all test cases and give a clear statement of the security maturity of the app. For a black box test, the client might prefer the test to be focused on the production app, with the goal of evaluating the effectiveness of its security controls.

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

Software Needed on the Host PC or Mac

On the laptop or PC you are testing on, you should install at least the following.

1. JRE or JDK
2. Android SDK
3. Android device or emulator

OS Versions

Before starting to test any application, it is important to have all the required hardware and software. This does not only mean that you must have a configured machine ready to run auditing tools, but also that you have the correct version of Android OS installed on the testing device. Therefore, it is always recommended to ask if the application runs only on specific versions of Android OS.

Testing on a Real Device

Different preparation steps need to be applied before a dynamic analysis of a mobile app can be started. Ideally the device is rooted, as otherwise some test cases cannot be tested properly. See "Rooting your device" for more information.

The available setup options for the network need to be evaluated first. The mobile device used for testing and the machine running the interception proxy need to be placed within the same WiFi network. Either an (existing) access point is used or [an ad-hoc wireless network is created](#).

Once the network is configured and connectivity is established between the testing machine and the mobile device, several other steps need to be done.

- The proxy in the network settings of the Android device need to be [configured properly to point to the interception proxy in use](#).
- The CA certificate of the interception proxy need to be added to the trusted certificates in the certificate storage of the Android device. Due to different versions of Android and modifications of Android OEMs to the settings menu, the location of the menu to store a CA might differ.

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

Rooting Your Device

Risks of Rooting

As a security tester, you may want to root your mobile device: while some tests can be performed on a non-rooted mobile, some do require a rooted one. However, you need to be aware of the fact that rooting is not an easy process and requires advanced knowledge. Rooting is risky, and three main consequences need to be clarified before you may proceed: rooting

- Usually voids the device warranty (always check the manufacturer policy before taking any action),
- May "brick" the device, i.e., render it inoperable and unusable.
- Brings additional security risks as built-in exploit mitigations are often removed.

You need to understand that rooting your device is ultimately YOUR own decision and that OWASP shall in no way be held responsible for any damage. In case you feel unsure, always seek expert advice before starting the rooting process.

What Mobiles Can Be Rooted?

Virtually any Android mobile can be rooted. Commercial versions of Android OS, at the kernel level evolutions of Linux OS, are optimized for the mobile world. Here some features are removed or disabled, such as the possibility for a non-privileged user to become the 'root' user (who has elevated privileges). Rooting a phone means adding the feature to become the root user, e.g. technically speaking adding a standard Linux executable called `su` used for switching users.

The first step in rooting a mobile is to unlock its boot loader. The procedure depends on each manufacturer. However, for practical reasons, rooting some mobiles is more popular than rooting others, particularly when it comes to security testing: devices created by Google (and manufactured by other companies like Samsung, LG and Motorola) are among the most popular, particularly because they are widely used by developers. The device warranty is not nullified when the boot loader is unlocked and Google provides many tools to support the root itself to work with rooted devices. A curated list of guide on rooting devices from all major brands can be found in the [XDA forums](#).

See also "Android Platform Overview" for further details.

Restrictions When Using a Non-Rooted Device

For testing of an Android app a rooted device is the foundation for a tester to be able to execute all available test cases. In case a non-rooted device need to be used, it is still possible to execute several test cases to the app.

Nevertheless, this highly depends on the restrictions and settings made in the app. For example if backups are allowed, a backup of the data directory of the app can be extracted. This allows detailed analysis of leakage of sensitive data when using the app. Also if SSL Pinning is not used a dynamic analysis can also be executed on a non-rooted device.

Testing on the Emulator

All of the above steps to prepare a hardware testing device do also apply if an emulator is used. For dynamic testing several tools or VMs are available that can be used to test an app within an emulator environment:

- AppUse
- MobSF

It is also possible to simply create an AVD and use this for testing.

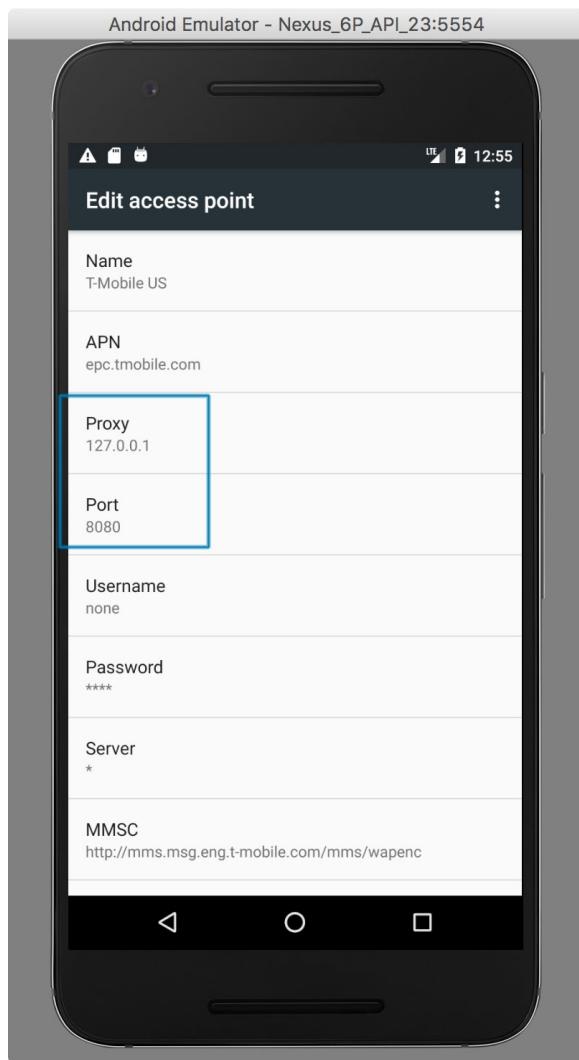
Setting Up a Web Proxy on Virtual Device

To set up a HTTP proxy on the emulator follow the following procedure, which works on the Android emulator shipping with Android Studio 2.x:

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 in the access point settings of the device:
2. Open the Settings Menu
3. Tap on "Wireless & Networks" -> "Cellular Networks" or "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 top-right menu and tap "save"



HTTP and HTTPS requests should now be routed over the proxy on the host machine. Try toggling airplane mode off and on if it doesn't work.

Installing a CA Certificate on the Virtual Device

An easy way to install a CA certificate is pushing the cert to the device and adding it to the certificate store via Security Settings. For example, you can install the PortSwigger (Burp) CA certificate as follows:

1. Start Burp and navigate to <http://burp/> using a web browser on the host, and 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 on `cacert.cer`

You should now 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

An Android Virtual Device (AVD) can be created by using the AVD manager, which is [available within Android Studio](#). The AVD manager can also be started separately from the command line by using the `android` command in the tools directory of the Android SDK:

```
$ ./android avd
```

Once the emulator is up and running a root connection can be established by using `adb`.

```
$ 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 of an emulator is therefore not needed as root access can be granted through `adb`.

Restrictions When Testing on an Emulator

There are several downsides when using an emulator. You might not be able to test an app properly in an emulator, if it's relying on the usage of a specific mobile network, or uses NFC or Bluetooth. Testing within an emulator is usually also slower in nature and might lead to issues on its own.

Nevertheless several hardware characteristics can be emulated, including [GPS](#), [SMS](#) and many more.

Testing Methods

Manual Static Analysis

In principle, we talk about white-box testing when the source code (or even better, the complete Android Studio project) is available, and black-box if only APK package is available. In Android app security testing however, the difference is not all that big. The majority of apps can be decompiled easily, and with some reverse engineering knowledge, having access to bytecode and binary code is almost as good as having the original code, except in cases where the release build is purposefully obfuscated.

To accomplish the source code testing, you will want to have a setup similar to the developer. You will need a testing environment on your machine with the Android SDK and an IDE installed. It is also recommended to have access either to a physical device or an emulator, so you can debug the app.

During **Black box testing** you will not have access to the source code in its original form. Usually, you will have the application package in hand (in [Android .apk format](#), which can be installed on an Android device or reverse engineered with the goal to retrieve parts of the source code).

An easy way on the CLI to retrieve the source code of an APK is through `apkx`, which also packages `dex2jar` and CFR and automates the extracting, 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 need to be tested:

```
$ 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 does not have any native library (code written in C/C++), the reverse engineering process is relatively easy and recovers almost the entire source code. Nevertheless, if the code is obfuscated, this process might become very time consuming and might not be productive. The same applies for applications that contain a native library. They can still be reverse engineered but require low level knowledge and the process is not automated.

More details and tools about the Android reverse engineering topic can be found in the "Tampering and Reverse Engineering on Android" section.

Automated Static Analysis

Static analysis should be supported through the usage of tools, to make the analysis efficient and to allow the tester to focus on the more complicated business logic. There are a plethora of static code analyzers that can be used, ranging from open source scanners to full blown enterprise ready scanners. The decision on which tool to use depends on the budget, requirements by the client and the preferences of the tester.

Some Static Analyzers rely on the availability of the source code while others take the compiled APK as input. It is important to keep in mind that while static analyzers can help us to focus attention on potential problems, they may not be able to find all the problems by itself. Go through each finding carefully and try to understand what the app is doing to improve your chances of finding vulnerabilities.

One important thing to note is to configure the static analyzer properly in order 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 the effort can become counterproductive if an overly large report need to be manually investigated.

Automated tools for performing security analysis on an APK are:

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

Dynamic Analysis

Compared to static analysis, dynamic analysis is applied while executing the mobile app. The test cases can range from investigating the file system and changes made to it on the mobile device to monitoring the communication with the endpoint while using the app.

When we talk about dynamic analysis of applications that rely on the HTTP(S) protocol, several tools can be used to support the dynamic analysis. The most important tools are so called interception proxies, like OWASP ZAP or Burp Suite Professional to name the most famous ones. An interception proxy allows the tester to have a Man-in-the-middle position in order to read and/or modify all requests made from the app and responses coming from the endpoint for testing Authorization, Session Management and so on.

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 begin using 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. Specifically, Mac OS versions from El Capitan don't have OpenSSL installed, so compiling pyOpenSSL doesn't work. You can resolve those issues by [installing OpenSSL manually]. To install openSSL, run:

```
$ brew install openssl
```

Drozer also depends on older versions of some libraries. In order not to mess up the system Python setup, it is better to install Python with homebrew and creating a dedicated environment with virtualenv (using a Python version management tool like [pyenv](#) is even better, but setting this up 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 that directory. Change 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. The pyOpenSSL version required by Drozer has a typo that prevents it from compiling successfully, so need to fix the source before compiling. Fortunately, ropnop has figured out 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:

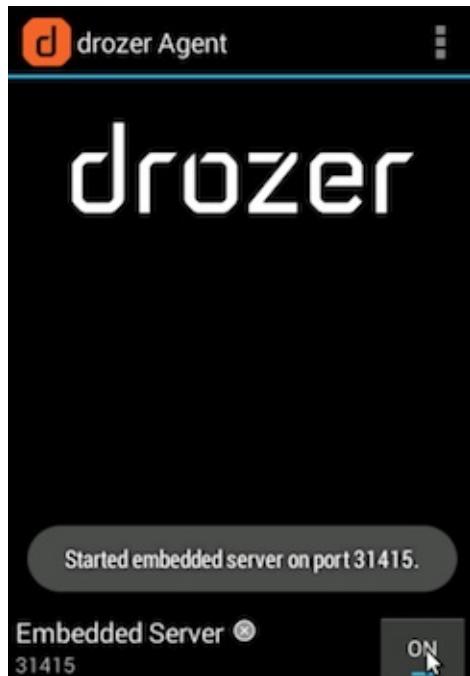
Drozer agent is the component running on the device itself. Download the latest Drozer Agent [here](#), and install it with adb.

```
$ 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 and you're ready to start exploring.

Open the Drozer application in running emulator and click the OFF button in the bottom of the app which will start a Embedded Server.



By default the server listens on port 31415. Forward this port to the localhost interface using adb, then run Drozer on the host to connect to the agent.

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

To show the list of all Drozer modules that can be executed in the current session use the "list" command.

Basic Drozer Commands:

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

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

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

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

- To see some basic information about the package, use

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

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

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

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

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

- To launch the activities exported, run the following command:

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

Using Modules:

Out of the box, Drozer provides modules to investigate 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 setup in your copy of Drozer. You can search for modules using 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 using 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 for immediate use.

Network Monitoring/Sniffing

Dynamic analysis by using an interception proxy can be straight forward if standard libraries in Android are used and all communication is done via HTTP. But what if XMPP or other protocols are used that are not recognized by your interception proxy? What if mobile application development platforms like [Xamarin](#) are used, where the produced apps do not use the local proxy settings of your Android phone? In this case we need to monitor and analyze the network traffic first in order to decide what to do next.

On Android it is possible to [remotely sniff all traffic in real-time by using tcpdump, netcat \(nc\) and Wireshark](#). First ensure 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
```

When executing `adb root` you might get an error saying `adbd cannot run as root in production builds`. If that's the case install `tcpdump` like this:

```
# 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: In order to use `tcpdump` you need root privileges on the phone!

`tcpdump` should now be working, so execute it once to see if it does. Once a few packets are coming in you can stop it 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
```

The first step in order to do remote sniffing of the network traffic on the Android phone by is executing `tcpdump` and pipe its output to netcat (nc):

```
$ tcpdump -i wlan0 -s0 -w - | nc -l -p 11111
```

Usually you want to monitor the wlan0 interface, in case you need another interface just list the available ones with `$ ip addr`.

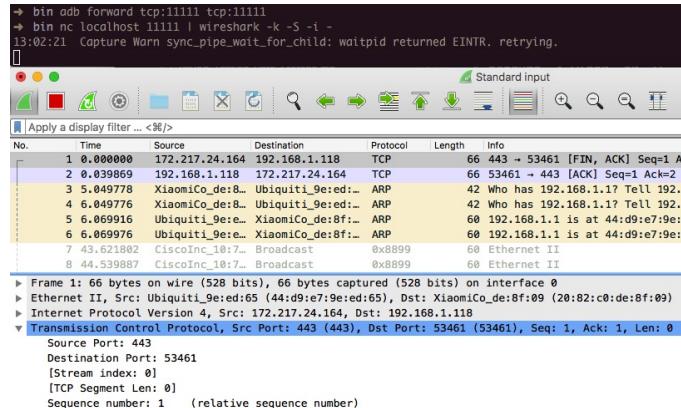
In order to access port 11111 opened by netcat, we need to forward the port via adb to your machine.

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

With the following command you are connecting to the forwarded port available on your local machine via netcat and piping it to Wireshark.

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

Wireshark should start and you should see the traffic from the wlan0 interface from the Android phone.



Xamarin

Xamarin is a mobile application development platform that is capable of producing [native Android 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 requests in your interception proxy, as the apps created by Xamarin do not use the local proxy settings of your Android 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:

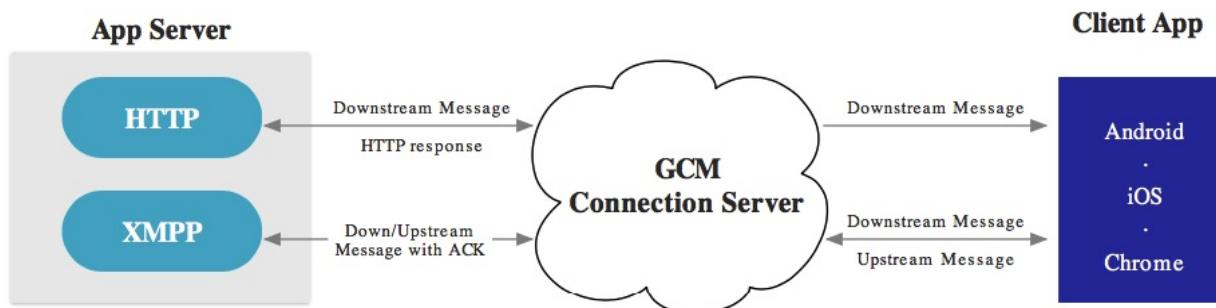
```
WebRequest.DefaultWebProxy = new WebProxy("192.168.11.1", 8080);
```

1. Use ettercap in order to get a man-in-the-middle position (MITM). When being MITM we only need to redirect port 443 to our interception proxy. See the next section about "Firebase/Google Cloud Messaging (FCM/GCM)" for the setup. The only difference is, that only port 443 need to be redirected to your interception proxy running on localhost.

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

Firebase/Google Cloud Messaging (FCM/GCM)

Firebase Cloud Messaging (FCM) is the successor of Google Cloud Messaging (GCM) and is a free service offered by Google and allows to send messages between an application server and client apps. The server and client app are communicating via the FCM/GCM connection server that is handling the downstream and upstream messages.



Downstream messages are sent from the application server to the client app (push notifications); upstream messages are sent from the client app to the server.

FCM is available for Android and also for iOS and Chrome. FCM provides two connection server protocols at the moment: HTTP and XMPP and there are several differences in the implementation, as described in the [official documentation](#). The following example demonstrates how to intercept both protocols.

Preparation

For a full dynamic analysis of an Android app FCM should be intercepted. To be able to intercept the messages several steps should be considered for preparation.

- [Install the CA certificate of your interception proxy into your Android phone.](#)
- A Man-in-the-middle attack should be executed so all traffic from the mobile device is

redirected to your testing machine. This can be done by using a tool like [ettercap](#). It can be installed by using brew on Mac OS X.

```
$ 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
```

FCM can use two different protocols to communicate with the Google backend, either XMPP or HTTP.

HTTP

The ports used by FCM for HTTP are 5228, 5229, and 5230. Typically only 5228 is used, but sometimes also 5229 or 5230 is used.

- Configure a local port forwarding on your machine for the ports used by FCM. The following example can be used on 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 need to listen to the port specified in the port forwarding rule above, which is 8080.

Xmpp

The [ports used by FCM over XMPP](#) are 5235 (Production) and 5236 (Testing).

- Configure a local port forwarding on your machine for the ports used by FCM. The following example can be used on 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 need to listen to the port specified in the port forwarding rule above, which is 8080.

Intercepting Messages

Your testing machine and the Android device need to be in the same wireless network. Start ettercap with the following command and replace the IP addresses with the one of the Android device and the network gateway in the wireless network.

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

Start using the app and trigger a function that uses FCM. You should see HTTP messages showing up 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=APA91bHWZNRCmf2ApntlGLEJO
0mEdYPOBIZ-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

When using ettercap you need to activate "Support invisible proxying" in Proxy Tab / Options / Edit Interface

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

As an alternative to a MITM attack executed on your machine, a Wifi Access Point (AP) or router can also be used instead. The setup would become a little bit more complicated, as port forwarding needs to be configured on the AP or router and need to point to your interception proxy that need to listen on the external interface of your machine. For this test setup tools like ettercap are not needed anymore.

Tools like Wireshark can be used to monitor and record the traffic for further investigation either locally on your machine or through a span port, if the router or Wifi AP offers this functionality.

Potential Obstacles

For the following security controls that might be implemented into the app you are about to test, it should be discussed with the project team if it is possible to provide a debug build. A debug build has several benefits when provided during a (white box) test, as it allows a more comprehensive analysis.

SSL Pinning

SSL Pinning is a mechanism to make dynamic analysis harder. Certificates provided by an interception proxy to enable a Man-in-the-middle position are declined and the app will not make any requests. To be able to efficiently test during a white box test, a debug build with deactivated SSL Pinning should be provided.

For a black box test, there are several ways to bypass SSL Pinning, for example [SSLUnpinning](#) or [Android-SSL-TrustKiller](#). Therefore bypassing can be done within seconds, but only if the app uses the API functions that are covered for these tools. If the app is using a different framework or library to implement SSL Pinning that is not implemented yet in those tools, the patching and deactivation of SSL Pinning needs to be done manually and can become time consuming.

To manually deactivate SSL Pinning there are two ways:

- Dynamical Patching while running the App, by using [Frida](#) or [ADB](#)
- [Identify the SSL Pinning logic in smali code, patch it and reassemble the APK](#)

Once successful, the prerequisites for a dynamic analysis are met and the apps communication can be investigated.

See also test case "Testing Custom Certificate Stores and SSL Pinning" for further details.

Root Detection

Root detection can be implemented using pre-made libraries like [RootBeer](#) or custom checks. An extensive list of root detection methods is presented in the "Testing Anti-Reversing Defenses on Android" chapter.

In 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, root detection can be disabled using a variety of methods which will be introduced later in this book.

Testing Data Storage on Android

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 Android offers for local data storage, as well as best practices for using them.

Note that "sensitive data" need to be identified in the context of each specific app. Data classification is described in detail in the chapter "Testing Processes and Techniques".

Testing for Sensitive Data in Local Storage

Overview

Conventional wisdom suggests saving as little sensitive data as possible on permanent local storage. However, in most practical scenarios, at least some types of user-related data need to be stored. For example, asking the user to enter a highly complex password every time the app is started isn't a great idea from a usability perspective. As a result, most apps must locally cache some kind of authentication token. Other types of sensitive data, such as personally identifiable information (PII), might also be saved if the particular scenario calls for it.

A vulnerability occurs when sensitive data is not properly protected by an app when persistently storing it. The app might be able to store it in different places, for example locally on the device or on an external SD card. When trying to exploit these kinds of issues, consider that there might be a lot of information processed and stored in different locations. It is important to identify at the beginning what kind of information is processed by the mobile application and keyed in by the user and what might be interesting and valuable for an attacker (e.g. passwords, credit card information, PII).

Consequences for disclosing sensitive information can be various, like disclosure of encryption keys that can be used by an attacker to decrypt information. More generally speaking an attacker might be able to identify this information to use it as a basis for further attacks like social engineering (when PII is disclosed), account hijacking (if session information or an authentication token is disclosed) or gather information from apps that have a payment option in order to attack and abuse it.

Storing data is essential for many mobile apps, for example in order to keep track of user settings or data a user has keyed in that needs to be stored locally or offline. Data can be stored persistently in various ways. The following list shows those mechanisms that are widely used on the Android platform:

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

The following snippets of code demonstrate bad practices that disclose sensitive information, but also show the different storage mechanisms on Android in detail.

Shared Preferences

[Shared Preferences](#) is a common approach to store Key/Value pairs persistently in the filesystem by using an XML structure. Within an activity the following code might be used to store sensitive information like a username and a password:

```
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 is called, the file key.xml is created with the provided data. This code is violating several best practices.

- The username and password is 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 content of `key.xml`

```
root@hermes:/data/data/sg.vp.owasp_mobile.myfirstapp/shared_prefs # ls -la
-rw-rw-r-- u0_a118 u0_a118    170 2016-04-23 16:51 key.xml
```

Please note that `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` were deprecated in API 17. Although this may not affect newer devices, applications compiled with `android:targetSdkVersion` set prior to 17 may still be affected, if they run on OS prior to Android 4.2 (`JELLY_BEAN_MR1`).

SQLite Database (Unencrypted)

SQLite is a SQL database that stores data to a `.db` file. The Android SDK comes with built in support for SQLite databases. The main package to manage the databases is `android.database.sqlite`. Within an activity the following code might be used to store sensitive information like a username and a password:

```
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 is called, the database file `privateNotSoSecure` is created with the provided data and is stored in clear text in `/data/data/<package-name>/databases/privateNotSoSecure`.

There might be several files available in the databases directory, besides the SQLite database.

- **Journal files**: These are temporary files used to implement atomic commit and rollback capabilities in SQLite.
- **Lock files**: The lock files are part of the locking and journaling mechanism designed to improve concurrency in SQLite and to reduce the writer starvation problem.

Unencrypted SQLite databases should not be used to store sensitive information.

SQLite Databases (Encrypted)

By using the library [SQLCipher](#) SQLite databases can be encrypted, by providing a password.

```
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, check if the password is hardcoded in the source, stored in shared preferences or hidden somewhere else in the code or file system. A secure approach to retrieve the key, instead of storing it locally could be to either:

- Ask the user every time for a PIN or password to decrypt the database, once the app is opened (weak password or PIN is prone to brute force attacks), or
- Store the key on the server and make it accessible via a web service (then the app can

only be used when the device is online).

Realm Databases

The [Realm Database for Java](#) is getting more and more popular amongst developers. The database and its content can be encrypted by providing a key in the configuration.

```
//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 encryption is not used, you should be able to obtain the data. If encryption is enabled, check if the key is hardcoded in the source or resources, whether it is stored unprotected in shared preferences or somewhere else.

Internal Storage

Files can be saved directly on the [internal storage](#) of the device. By default, files saved to the internal storage are private to your app and other apps cannot access them. When the user uninstalls your app, these files are removed. Within an activity the following code might be used to store sensitive information in the variable `test` persistently to the 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();  
}
```

The file mode needs to be checked to make sure that only the app itself has access to the file. This can be set by using `MODE_PRIVATE`. Other modes like `MODE_WORLD_READABLE` (deprecated) and `MODE_WORLD_WRITEABLE` (deprecated) are more lax and can pose a security risk.

It should also be checked what files are opened and read within the app by searching for the class `FileInputStream`. Part of the internal storage mechanisms is also the cache storage. To cache data temporarily, functions like `getCacheDir()` can be used.

External Storage

Every Android-compatible device supports a [shared external storage](#) that you can use to save files. This can be a removable storage media (such as an SD card) or an internal (non-removable) storage. Files saved to the external storage are world-readable and can be modified by the user when they enable USB mass storage to transfer files on a computer. Within an activity the following code might be used to store sensitive information in the file `password.txt` persistently to the external storage:

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

Once the activity is called, the file is created with the provided data and the data is stored in clear text in the external storage.

It's also worth to know that files stored outside the application folder (`data/data/<package-name>/`) will not be deleted when the user uninstall the application.

KeyChain

The [KeyChain class](#) is used to store and retrieve *system-wide* private keys and their corresponding certificate (chain). The user will be prompted to set a lock screen pin or password to protect the credential storage if it hasn't been set, if something gets imported into the KeyChain the first time. Please note that the KeyChain is system-wide: so every application can access the materials stored in the KeyChain.

KeyStore

The [Android KeyStore](#) provides a means of (more or less) secure credential storage. As of Android 4.3, it provides public APIs for storing and using app-private keys. An app can create a new private/public key pair to encrypt application secrets by using the public key and decrypt the same by using the private key.

The keys stored in the Android KeyStore can be protected such that the user needs to authenticate to access them. The user's lock screen credentials (pattern, PIN, password or fingerprint) are used for authentication.

Stored keys can be configured to operate in one of the two modes:

1. User authentication authorizes the use of keys for a duration of time. All keys in this mode are authorized for use as soon as the user unlocks the device. The duration for which the authorization remains valid can be customized for each key. This option can only be used if the secure lock screen is enabled. If the user disables the secure lock screen, any stored keys become permanently invalidated.
2. User authentication authorizes a specific cryptographic operation associated with one key. In this mode, each operation involving such a key must be individually authorized by the user. Currently, the only means of such authorization is fingerprint authentication.

The level of security afforded by the Android KeyStore depends on its implementation, which differs between devices. Most modern devices offer a hardware-backed KeyStore implementation. In that case, keys are generated and used in a Trusted Execution Environment or a Secure Element and are not directly accessible for the operating system. This means that the encryption keys themselves cannot be easily retrieved even from a rooted device. You can check whether the keys are inside the secure hardware, based on the return value of the `isInsideSecureHardware()` method which is part of the [class KeyInfo](#). Please note that private keys are often indeed stored correctly within the secure hardware, but secret keys, HMAC keys are, are not stored securely according to the KeyInfo on quite some devices.

In a software-only implementation, the keys are encrypted with a [per-user encryption master key](#). In that case, an attacker can access all keys on a rooted device in the folder `/data/misc/keystore/`. As the master key is generated using the user's own lock screen pin/password, the Android KeyStore is unavailable when the device is locked.

Older KeyStore Implementations

Older Android versions do not have a KeyStore, but do have the KeyStore interface from JCA (Java Cryptography Architecture). One can use various KeyStores that implement this interface and ensure secrecy and integrity to the keys stored in the KeyStore implementation. All implementations rely on the fact that a file is stored on the filesystem, which then protects its content by a password. For this, it is recommended to use the BouncyCastle KeyStore (BKS). You can create one by using the

```
KeyStore.getInstance("BKS", "BC");
```

, where "BKS" is the KeyStore name (BouncycastleKeyStore) and "BC" is the provider (BouncyCastle). Alternatively you can use SpongyCastle as a wrapper and initialize the KeyStore: `KeyStore.getInstance("BKS", "SC");`.

Please be aware that not all KeyStores offer proper protection to the keys stored in the KeyStore files.

Static Analysis

Local Storage

As already pointed out, there are several ways to store information within Android. Several checks should therefore be applied to the source code to identify the storage mechanisms used within the Android app and whether sensitive data is processed insecurely.

- Check `AndroidManifest.xml` for permissions to read from or write to external storage, like `uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"`
- Check the source code for keywords and API calls that are used for storing data:
 - File permissions like:
 - `MODE_WORLD_READABLE` OR `MODE_WORLD_WRITABLE`. IPC files should not be created with permissions of `MODE_WORLD_READABLE` or `MODE_WORLD_WRITABLE` unless it is required as any app would be able to read or write the file even though it may be stored in the app private data directory.
 - Classes and functions like:
 - `SharedPreferences` Class (Storage of key-value pairs)
 - `FileOutputStream` Class (Using Internal or External Storage)
 - `getExternal*` functions (Using External Storage)
 - `getWritableDatabase` function (return a `SQLiteDatabase` for writing)
 - `getReadableDatabase` function (return a `SQLiteDatabase` for reading)
 - `getCacheDir` and `getExternalCacheDirs` function (Using cached files)

Encryption operations should rely on solid and tested functions provided by the SDK. The following describes different “bad practices” that should be checked in the source code:

- Check if simple bit operations are used, like XOR or Bit flipping to “encrypt” sensitive information that is stored locally. This should be avoided as the data can easily be recovered.
- Check if keys are created or used without taking advantage of the Android onboard features like the Android KeyStore.
- Check if keys are disclosed.

Typical Misuse: Hardcoded Cryptographic Keys

The use of a hardcoded or world-readable cryptographic key significantly increases the possibility that encrypted data may be recovered. Once it is obtained by an attacker, the task to decrypt the sensitive data becomes trivial, and the initial idea to protect confidentiality fails.

When using symmetric cryptography, the key needs to be stored within the device and it is just a matter of time and effort from the attacker to identify it.

Consider the following scenario: An application is reading and writing to an encrypted database but the decryption is done based on a hardcoded key:

```
this.db = localUserSecretStore.getWritableDatabase("SuperPassword123");
```

Since the key is the same for all app installations it is trivial to obtain it. The advantages of having sensitive data encrypted are gone, and there is effectively no benefit in using encryption in this way. Similarly, look for hardcoded API keys/private keys and other valuable pieces. Encoded/encrypted keys is just another attempt to make it harder but not impossible to get the crown jewels.

Let's consider this piece of 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"
};
```

Algorithm to decode the original key in this case might look 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 places where secrets are usually hidden:

- 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>
```

- shared preferences (typically in the shared_prefs directory)
- build configs, such as in local.properties or gradle.properties

Example:

```
buildTypes {
    debug {
        minifyEnabled true
        buildConfigField "String", "hiddenPassword", "\"${hiddenPassword}\""
    }
}
```

KeyChain and Android KeyStore

When going through the source code it should be analyzed if native mechanisms that are offered by Android are applied to the identified sensitive information. Sensitive information must not be stored in clear text but should be encrypted. If sensitive information needs to be stored on the device itself, several API calls are available to protect the data on the Android device by using the **KeyChain** and **KeyStore**. The following should therefore be done:

- Check if a key pair is created within the app by looking for the class `KeyPairGenerator`.
- Check that the app is using the Android KeyStore and Cipher mechanisms to securely store encrypted information on the device. Look for the pattern `import java.security.KeyStore, import javax.crypto.Cipher, import java.security.SecureRandom` and corresponding usages.
- The `store(OutputStream stream, char[] password)` function can be used to store the KeyStore to disk with a specified password. Check that the password provided is not hardcoded and is defined by user input as this should only be known to the user. Look for the function `.store()`.

Dynamic Analysis

Install and use the app as it is intended and execute all functions at least once. Data can be generated when entered by the user, sent by the endpoint or it is already shipped within the app when installing it. Afterwards check the following items:

- Check the files that are shipped with the mobile application once installed in `/data/data/<package-name>/` in order to identify development, backup or simply old files that shouldn't be in a production release.
- Check if SQLite databases are available and if they contain sensitive information (usernames, passwords, keys etc.). SQLite databases are stored in `/data/data/<package-name>/databases`.
- Check Shared Preferences that are stored as XML files in the shared_prefs directory of

the app for sensitive information, which is in `/data/data/<package-name>/shared_prefs`.

- Check the file system permissions of the files in `/data/data/<package-name>`. Only the user and group created when installing the app (e.g. `u0_a82`) should have the user rights read, write and execute (`rwx`). Others should have no permissions to files, but may have the executable flag to directories.
- Check if there is a Realm database available in `/data/data/<package-name>/files/` and if it is unencrypted and contains sensitive information. The file extension is `realm` and the file name is `default` by default. Inspecting the Realm database is done with the [Realm Browser](#).

Remediation

The credo for saving data can be summarized quite easily: Public data should be available for everybody, but sensitive and private data needs to be protected or even better not get stored on the device in the first place.

If sensitive information (credentials, keys, PII, etc.) is needed locally on the device several best practices are offered by Android that should be used to store data securely instead of reinventing the wheel or leaving data unencrypted on the device.

The following is a list of best practices used for secure storage of certificates, keys and sensitive data in general:

- Encryption or decryption functions that were self implemented need to be avoided. Instead use Android implementations such as [Cipher](#), [SecureRandom](#) and [KeyGenerator](#).
- Username and password should not be stored on the device. Instead, perform initial authentication using the username and password supplied by the user, and then use a short-lived, service-specific authorization token or session identifier. If possible, use the [AccountManager](#) class to invoke a cloud-based service and do not store passwords on the device.
- Usage of `MODE_WORLD_WRITEABLE` or `MODE_WORLD_READABLE` should generally be avoided for files. If data needs to be shared with other applications, a content provider should be considered. A content provider offers read and write permissions to other apps and can make dynamic permission grants on a case-by-case basis.
- The usage of Shared Preferences or other mechanisms that are not able to protect data should be avoided to store sensitive information. SharedPreferences are insecure and not encrypted by default. [Secure-preferences](#) can be used to encrypt the values stored within Shared Preferences, but the Android KeyStore should be the first option to store data securely.
- Do not use the external storage for sensitive data. By default, files saved to the internal storage are private to your application and other applications cannot access them (nor

can the user). When the user uninstalls your application, these files are also removed.

- To provide additional protection for sensitive data, you might choose to encrypt local files using a key that is not directly accessible to the application. For example, a key can be placed in a KeyStore and protected with a user password that is not stored on the device. While this does not protect data from a root compromise that can monitor the user keying in the password, it can provide protection for a lost device without file system encryption.
- Set variables that use sensitive information to null once finished.
- Use immutable objects for sensitive data so it cannot be changed.

Please also check the [Security Tips data](#) in the Android developers guide.

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

OWASP MASVS

- V2.1: "System credential storage facilities are used appropriately to store sensitive data, such as user credentials or cryptographic keys."

CWE

- CWE-311 - Missing Encryption of Sensitive Data
- CWE-312 - Cleartext Storage of Sensitive Information
- CWE-522 - Insufficiently Protected Credentials
- 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>

Testing for Sensitive Data in Logs

Overview

There are many legit reasons to create log files on a mobile device, for example to keep track of crashes or errors or simply for usage statistics. Log files can be stored locally when being offline and being sent to the endpoint once being online again. However, logging sensitive data such as usernames or session IDs might expose the data to attackers or malicious applications and violates the confidentiality of the data. Log files can be created in various ways and the following list shows the mechanisms that are available on Android:

- [Log Class](#)
- [Logger Class](#)
- `System.out/System.err.print`

Static Analysis

The source code should be checked for logging mechanisms used within the Android App, by searching for the following keywords:

1. Functions and classes like:

- `android.util.Log`
- `Log.d | Log.e | Log.i | Log.v | Log.w | Log.wtf`
- `Logger`
- `System.out.print | System.err.print`

2. Keywords and system output to identify non-standard log mechanisms like:

- `logfile`
- `logging`
- `logs`

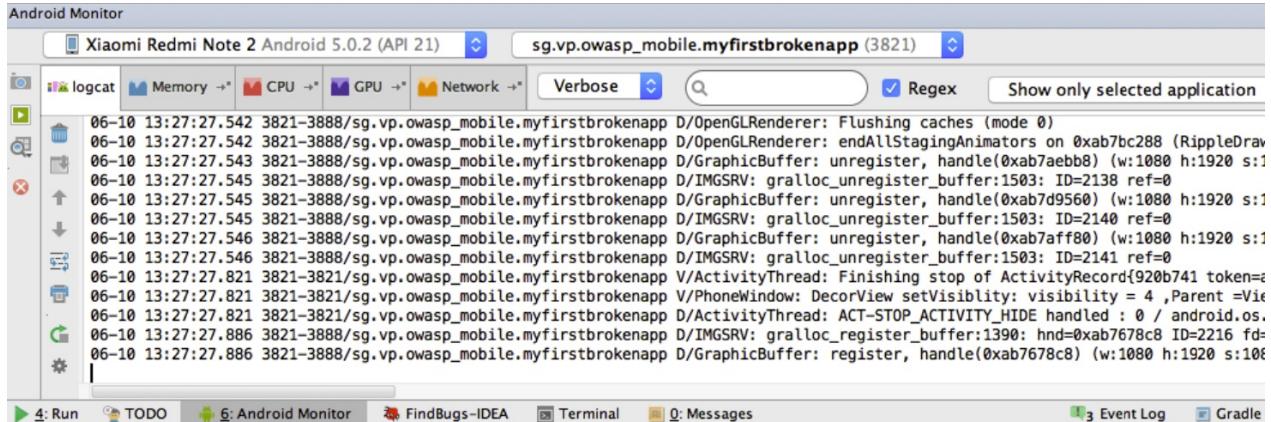
Dynamic Analysis

Use the mobile app extensively so that all functionality is at least triggered once. Afterwards identify the data directory of the application in order to look for log files (`/data/data/<package-name>`). Check if log data is generated by checking the application logs, as 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, the testing approach also needs to cover all output generated by the application during starting, running and closing of it. In order to verify what data is printed directly by using `System.out.println()` or `printStackTrace()` the tool [Logcat](#) can be used to check the app output. Two different approaches are available to execute Logcat.

- Logcat is already part of *Dalvik Debug Monitor Server (DDMS)* and is built into Android

Studio. If the app is in debug mode and running, the log output is shown in the Android Monitor in the Logcat tab. Patterns can be defined in Logcat to filter the log output of the app.



- Logcat can be executed by using adb in order to store the log output permanently.

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

Remediation

Ensure that a centralized logging class and mechanism is used and that logging statements are removed from the production release, as logs may be interrogated or readable by other applications. Tools like `ProGuard`, which is already included in Android Studio can be used to strip out logging portions in the code when preparing the production release. For example, to remove logging calls implemented with the class `android.util.Log`, simply add the following option in the `proguard-project.txt` configuration file of ProGuard:

```
-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(...);
}
```

Please note that the above example only ensures that calls to the methods offered by the `Log` class will be removed. However, if the string to be logged is dynamically constructed, the code for constructing the string might 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 has the string constructed explicitly:

```
Log.v(new StringBuilder("Private key [byte format]: ").append(key.toString()).toString());
```

What ProGuard guarantees is the 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 used](#).

This is potentially a security risk, as the (now unused) string leaks plain text data in memory which can be accessed over a debugger or by memory dumping.

Unfortunately, there is no silver bullet against this issue, but there are few options available:

- Implement a custom logging facility that takes simple arguments and does the construction of the log statements internally.

```
SecureLog.v("Private key [byte format]: ", key);
```

Then configure ProGuard to strip its calls.

- Remove logs on source level, instead of compiled bytecode level. Below is a simple Gradle task which comments out all log statements including any inline string builder.

```
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("(\\bLog\\b|ewidv|wtf)\\s*(\\[\\s*\\s*]*)>\\s*;", "/*$1*/")
                file.write(out);
            }
        }
    }
}
```

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

OWASP MASVS

- V2.2: "No sensitive data is written to application logs."

CWE

- CWE-117: Improper Output Neutralization for Logs
- CWE-532: Information Exposure Through Log Files
- CWE-534: Information Exposure Through Debug Log Files

Tools

- ProGuard - <http://proguard.sourceforge.net/>
- Logcat - <http://developer.android.com/tools/help/logcat.html>

Testing Whether Sensitive Data is Sent to Third Parties

Overview

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, like a Jar in an Android project that is getting included into the APK.
- By using a full SDK.

Static Analysis

Some 3rd party libraries can be automatically integrated into the app through a wizard within the IDE. The permissions set in the `AndroidManifest.xml` when installing a library through an IDE wizard should be reviewed. Especially permissions to access `SMS (READ_SMS)`, `contacts (READ_CONTACTS)` or the location (`ACCESS_FINE_LOCATION`) should be challenged if they are really needed to make the library work at a bare minimum, see also [Testing App Permissions](#). When talking to developers it should be shared to them that it's actually necessary to have a look at the differences on the project source code before and after the library was installed through the IDE and what changes have been made to the code base.

The same thing applies when adding a library or SDK manually. The source code should be checked for API calls or functions provided by the 3rd party library or SDK. The applied code changes should be reviewed and it should be checked if available security best practices of the library and SDK are applied and used.

The libraries loaded into the project should be reviewed in order to identify with the developers if they are needed and also if they are out of date and contain known vulnerabilities.

Dynamic Analysis

All requests made to external services should be analyzed if any sensitive information is embedded into them. Dynamic analysis can be performed by launching a Man-in-the-middle (MITM) attack using *Burp Suite Professional* or *OWASP ZAP*, to intercept the traffic exchanged between client and server. Once we are able to route the traffic to the interception proxy, we can try to sniff the traffic from the app to the server and vice versa. 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.

Remediation

All data that is sent to 3rd Party services should be anonymized, so no PII data is available. 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.

`AndroidManifest.xml` should only contain the permissions that are absolutely needed to work properly and as intended.

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

OWASP MASVS

- V2.3: "No sensitive data is shared with third parties unless it is a necessary part of the architecture."

CWE

- CWE-359 - Exposure of Private Information ('Privacy Violation')

Info

- [#nolan] Bulletproof Android, Godfrey Nolan - Chapter 7, Third-Party Library Integration

Tools

- Burp Suite Professional - <https://portswigger.net/burp/>
- OWASP ZAP - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

Testing Whether the Keyboard Cache Is Disabled for Text Input Fields

Overview

When typing in data into input fields, the software keyboard automatically suggests what data the user might want to key in. This feature can be very useful in messaging apps to write text messages more efficiently. For input fields that are asking for sensitive information like credit card data the keyboard cache might disclose sensitive information already when the input field is selected.

Static Analysis

In the layout definition of an activity, `TextViews` can be defined that have XML attributes. When the XML attribute `android:inputType` is set with the constant `textNoSuggestions` the keyboard cache is not shown if the input field is selected. Only the keyboard is shown and the user needs to type everything manually and nothing is suggested to him.

```
<EditText  
    android:id="@+id/KeyBoardCache"  
    android:inputType="textNoSuggestions"/>
```

Dynamic Analysis

Start the app and click into the input fields that ask for sensitive data. If strings are suggested the keyboard cache is not disabled for this input field.

Remediation

All input fields that ask for sensitive information, should implement the following XML attribute to [disable the keyboard suggestions](#):

```
android:inputType="textNoSuggestions"
```

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

OWASP MASVS

- V2.4: "The keyboard cache is disabled on text inputs that process sensitive data."

CWE

- CWE-524 - Information Exposure Through Caching

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 therefore be misused by malicious apps in order to get sensitive data stored in the clipboard.

Static Analysis

Input fields that are asking for sensitive information need to be identified and afterwards be investigated if any countermeasures are in place to mitigate the clipboard of showing up. See the remediation section for code snippets that could be applied.

Dynamic Analysis

Start the app and click into the input fields that ask for sensitive data. When it is possible to get the menu to copy/paste data the functionality is not disabled for this input field.

To extract the data stored in the clipboard, the Drozer module `post.capture.clipboard` can be used:

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

Remediation

A general best practice is overwriting different functions in the input field to disable the clipboard specifically for it.

```
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;
    }
});
```

Also `longclickable` should be deactivated for the input field.

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

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage
- M2 - Insecure Data Storage

OWASP MASVS

- V2.5: "The clipboard is deactivated on text fields that may contain sensitive data."

CWE

- CWE-200 - Information Exposure

Tools

- Drozer - <https://labs.mwrinfosecurity.com/tools/drozer/>

Testing Whether Stored Sensitive Data Is Exposed via IPC Mechanisms

Overview

As part of the IPC mechanisms included on Android, content providers allow an app's stored data to be accessed and modified by other apps. If not properly configured, they could lead to leakage of stored sensitive data.

Static Analysis

The first step is to look into the `AndroidManifest.xml` in order to detect content providers exposed by the app. Content providers can be identified through the `<provider>` element.

Check if the provider has the export tag set to "true" (`android:exported="true"`). Even if this is not the case, remember that if it has an `<intent-filter>` defined, the export tag will be automatically set to "true".

Finally, check if it is being protected by any permission tag (`android:permission`).

Permission tags allow to limit the exposure to other apps.

Inspect the source code to further 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(`

When exposing a content provider it should also be checked if parametrized [query methods](#) ("Query method in Content Provider Class") (`query()`, `update()`, and `delete()`) are being used to prevent SQL injection. If so, check if all inputs to them are properly sanitized.

As an example of a vulnerable content provider we will use the vulnerable password manager app [Sieve](#).

Inspect the AndroidManifest

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 can be seen in the `AndroidManifest.xml` above, the application exports two content providers. Note that one path ("/`Keys`") is being protected by read and write permissions.

Inspect the source code

In the `DBContentProvider.java` file the `query` function need to be inspected to detect if any sensitive information is 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`", being the latter not protected in the manifest and therefore vulnerable.

The query statement would return all passwords when accessing an URI including this path `Passwords/`. We will address this in the dynamic analysis below and find out the exact URI required.

Dynamic Analysis

Testing Content Providers

To begin dynamic analysis of an application's content providers, you should first enumerate the attack surface. This can be achieved using the Drozer module `app.provider.info` and providing the package name of the app:

```
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 the example, two content providers are exported, each not requiring any permission to interact with them, except for the `/Keys` path in the `DBContentProvider`. Using this information you can reconstruct part of the content URIs to access the `DBContentProvider`, because it is known that they must begin with `content://`. However, the full content provider URI is not currently known.

To identify content provider URIs within the application, Drozer's `scanner.provider.finduris` module should be used. This utilizes various techniques to guess paths and determine a list of accessible content URIs:

```
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/
```

Now that you have a list of accessible content providers, the next step is to attempt to extract data from each provider, which can be achieved using 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
```

In addition to querying data, Drozer can be used to update, insert 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 the use of SQLite databases for storing user data. Since these databases use SQL, they can be vulnerable to SQL injection. The Drozer module `app.provider.query` can be used 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 Pass
words

dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords
/ --selection ""
unrecognized token: '')" (code 1): , while compiling: SELECT * FROM Passwords WHERE ('
')
```

If vulnerable to SQL Injection, the application will return a verbose error message. SQL Injection in Android can be exploited 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 tables in the database:

```
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 can also be exploited to retrieve data from otherwise protected tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords
/ --projection "* FROM Key;--"
| Password | pin |
| thisismy password | 9876 |
```

These steps can be automated by using 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

A content provider can provide access to the underlying file system. This allows apps to share files, where the Android sandbox would otherwise prevent it. The Drozer modules `app.provider.read` and `app.provider.download` can be used to read or download files from exported file based content providers. These content providers can be susceptible to directory traversal vulnerabilities, making it possible to read otherwise protected files within the target application's sandbox.

```
dz> run app.provider.download content://com.vulnerable.app.FileProvider/../../../../...  
../../../../data/data/com.vulnerable.app/database.db /home/user/database.db  
Written 24488 bytes
```

To automate the process of finding content providers susceptible to directory traversal, the `scanner.provider.traversal` module should be used:

```
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 on a device:

```
$ 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  
...
```

Remediation

Set `android:exported` to "false" if the content is only meant to be accessed by the app itself. If not, set it to "true" and define proper read and write permissions.

Protect the content provider with the `android:protectionLevel` attribute set to `signature` protection, if it is only intended to be accessed by your own apps (signed with the same key). On the other hand, you may also want to offer access to other apps, for that you can apply a security policy by using the `<permission>` element and set a proper `android:protectionLevel`. When using `android:permission`, other applications will need to declare a corresponding `<uses-permission>` element in their own manifest to be able to interact with your content provider.

In order to avoid SQL injection attacks, use parameterized query methods such as `query()`, `update()`, and `delete()`. Be sure to properly sanitize all inputs to these methods because if, for instance, the `selection` argument is built out of user input concatenation, it could also lead to SQL injection.

You may also want to provide more granular access to other apps by using the `android:grantUriPermissions` attribute in the manifest and limit the scope with the `<grant-uri-permission>` element.

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage
- M2 - Insecure Data Storage

OWASP MASVS

- V2.6: "No sensitive data is exposed via IPC mechanisms."

CWE

- CWE-634 - Weaknesses that Affect System Processes

Tools

- Drozer - <https://labs.mwrinfosecurity.com/tools/drozer/>

Testing for Sensitive Data Disclosure Through the User Interface

Overview

In many apps users need to key in different kind of data to for example register an account or execute payment. Sensitive data could be exposed if the app is not masking it properly and showing data in clear text.

Masking of sensitive data within an activity of an app should be enforced to prevent disclosure and mitigate for example shoulder surfing.

Static Analysis

To verify if the application is masking sensitive information that is keyed in by the user, check for the following attribute in the definition of EditText:

```
    android:inputType="textPassword"
```

This will show dots in the text field instead of the keyed in characters.

Dynamic Analysis

To analyze if the application leaks any sensitive information to the user interface, run the application and identify parts of the app that either shows or asks for such information to be keyed in.

If the information is masked, e.g. by replacing characters in the text field through asterisks or dots the app is not leaking data to the user interface.

Remediation

In order to prevent leaking of passwords or pins, sensitive information should be masked in the user interface. The attribute `android:inputType="textPassword"` should therefore be used for EditText fields.

References

OWASP Mobile Top 10 2016

- M4 - Unintended Data Leakage

OWASP MASVS

- V2.7: "No sensitive data, such as passwords and pins, is exposed through the user interface."

CWE

- CWE-200 - Information Exposure

Testing for Sensitive Data in Backups

Overview

Like other modern mobile operating systems Android offers auto-backup features. The backups usually include copies of the data and settings of all apps installed on the device. An obvious concern is whether sensitive user data stored by the app might unintentionally leak to those data backups.

Given its diverse ecosystem, Android has a lot of backup options to account for.

- Stock Android has built-in USB backup facilities. A full data backup, or a backup of a particular app's data directory, can be obtained using the `adb backup` command when USB debugging is enabled.
- Google also 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 selected data to the Android Backup Service.
 - [Auto Backup for Apps](#): With Android 6.0 (\geq API level 23), Google added the "Auto Backup for Apps feature". This feature automatically syncs up to 25MB of app data to the user's Google Drive account.
- OEMs may add additional options. For example, HTC devices have a "HTC Backup" option that, when activated, performs daily backups to the cloud.

Static Analysis

Local

In order to backup all your application data Android provides an attribute called `allowBackup`. This attribute is set within the `AndroidManifest.xml` file. If the value of this attribute is set to **true**, then the device allows users to backup the application using Android Debug Bridge (ADB) via the command `$ adb backup`.

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 value is set to **true**, investigate whether the app saves any kind of sensitive data, check the test case "Testing for Sensitive Data in Local Storage".

Cloud

Regardless of using either key/value or auto backup, it needs to be identified:

- what files are sent to the cloud (e.g. SharedPreferences),
- if the files contain sensitive information and
- if sensitive information is protected through encryption before sending it to the cloud.

- **Auto Backup:** Auto Backup is configured through the boolean attribute

`android:allowBackup` within the application's manifest file. If not explicitly set, applications targeting Android 6.0 (API Level 23) or higher enable [Auto Backup](#) by default. The attribute `android:fullBackupOnly` can also be used to activate auto backup when implementing a backup agent, but this is only available from Android 6.0 onwards. Other Android versions will be using key/value backup instead.

```
android:fullBackupOnly
```

Auto backup includes almost all of the app files and stores them in the Google Drive account of the user, limited to 25MB per app. Only the most recent backup is stored, the previous backup is deleted.

- **Key/Value Backup:** To enable key/value backup the backup agent needs to be defined in the manifest file. Look in `AndroidManifest.xml` for the following attribute:

```
android:backupAgent
```

To implement the key/ value backup, either one of the following classes needs to be extended:

- [BackupAgent](#)
- [BackupAgentHelper](#)

Look for these classes within the source code to check for implementations of key/value backup.

Dynamic Analysis

After executing all available functions when using the app, attempt to make a backup using `adb`. If 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, you will have a `.ab` file in your current working directory. Run the following command to convert the `.ab` file into a `.tar` file.

```
$ dd if=mybackup.ab bs=24 skip=1|openssl zlib -d > mybackup.tar
```

Alternatively, use the [Android Backup Extractor](#) for this task. For the tool to work, you also 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 into your current working directory to perform your analysis for sensitive data.

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

Remediation

To prevent backing up the app data, set the `android:allowBackup` attribute to **false** in `AndroidManifest.xml`. If this attribute is not available the allowBackup setting is enabled by default. Therefore it needs to be explicitly set in order to deactivate it.

Sensitive information should not be sent in clear text to the cloud. Either,

- avoid storing the information in the first place, or
- encrypt the information at rest, before sending it to the cloud.

Files can also be excluded from [Auto Backup](#), in case they should not be shared with the Google Cloud.

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage
- M2 - Insecure Data Storage

OWASP MASVS

- V2.8: "No sensitive data is included in backups generated by the mobile operating system."

CWE

- CWE-530 - Exposure of Backup File to an Unauthorized Control Sphere

Tools

- Android Backup Extractor - <https://github.com/nelenkov/android-backup-extractor>

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. Sensitive data could be exposed if a user deliberately takes a screenshot of the application while sensitive data is displayed, or in the case of a malicious application running on the device, that is able to continuously capture the screen. This information is written to local storage, from which it may be recovered either by a rogue application on a rooted device, or by someone who steals the device.

For example, capturing a screenshot of a banking application running on the device may reveal information about the user account, his credit, transactions and so on.

Static Analysis

In Android, when the app goes into background a screenshot of the current activity is taken and is used to give a pleasing effect when the app is entered again. However, this would leak sensitive information that is present within the app.

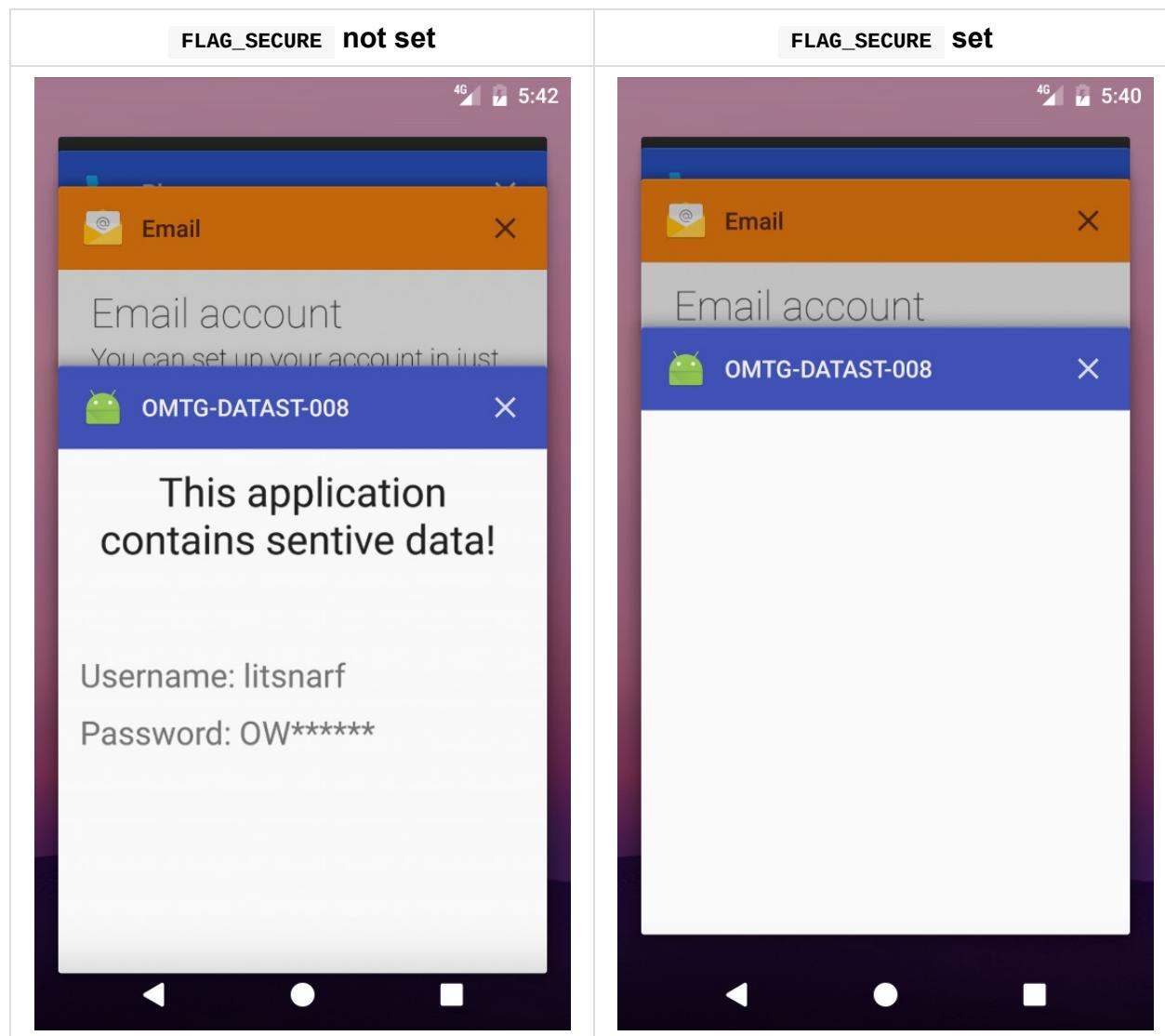
To verify if the application may expose sensitive information via app switcher, detect if the `FLAG_SECURE` option is set. You should be able to find something similar to the following code snippet.

```
LayoutParams.FLAG_SECURE
```

If not, the application is vulnerable to screen capturing.

Dynamic Analysis

During black-box testing, open any screen within the app that contains sensitive information and click on the home button so that the app goes into background. Now press the app-switcher button, to see the snapshot. As shown below, if `FLAG_SECURE` is set (image on the right), the snapshot is empty, while if the `FLAG_SECURE` is not set (image on the left), information within the activity is shown:



Remediation

To prevent users or malicious applications from accessing information from backgrounded applications use the `FLAG_SECURE` as shown below:

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,
    WindowManager.LayoutParams.FLAG_SECURE);

setContentView(R.layout.activity_main);
```

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage
- M2 - Insecure Data Storage

OWASP MASVS

- V2.9: "The app removes sensitive data from views when backgrounded."

CWE

- CWE-200 - Information Exposure

Testing for Sensitive Data in Memory

Overview

Analyzing the memory can help to identify the root cause of different problems, like for example why an application is crashing, but can also be used to identify sensitive data. This section describes how to check for sensitive data and disclosure of data in general within the process memory.

To be able to investigate the memory of an application a memory dump needs to be created first or the memory needs to be viewed with real-time updates. This is also already the problem, as the application only stores certain information in memory if certain functions are triggered within the application. Memory investigation can of course be executed randomly in every stage of the application, but it is much more beneficial to understand first what the mobile app is doing and what kind of functionalities it offers and also make a deep dive into the (decompiled) source code before making any memory analysis. Once sensitive functions are identified, like decryption of data, the investigation of a memory dump might be beneficial in order to identify sensitive data like a key or the decrypted information itself.

Static Analysis

First, you need to identify which sensitive information is stored in memory. Then there are a few checks that must be executed:

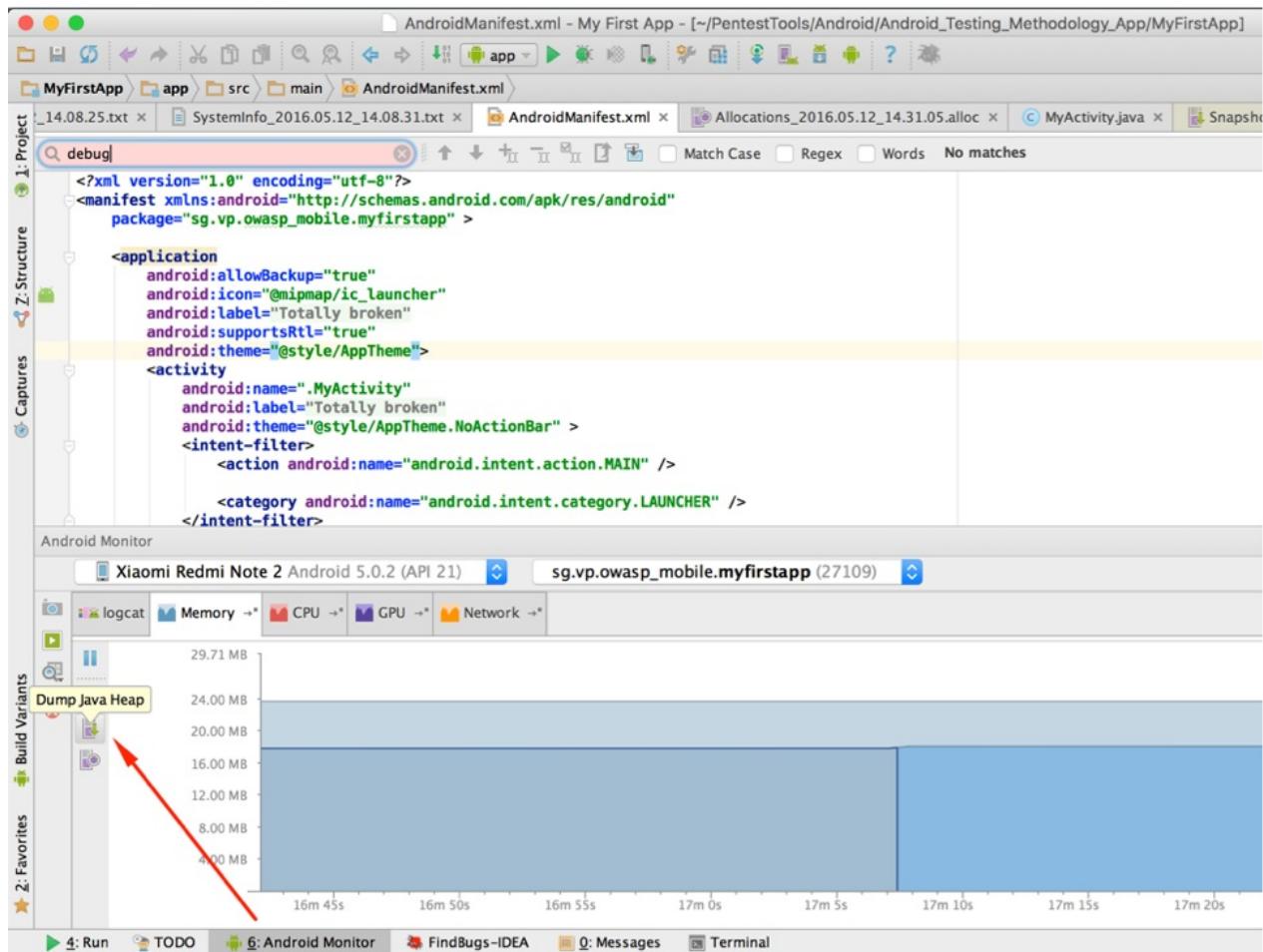
- Verify that no sensitive information is stored in an immutable structure. Immutable structures are not really overwritten in the heap, even after nullification or changing them. Instead, by changing the immutable structure, a copy is created on the heap.
`BigInteger` and `String` are two of the most used examples when storing secrets in memory.
- Verify that, when mutable structures are used, such as `byte[]` and `char[]` that all copies of the structure are cleared.

-NOTICE**: Destroying a key (e.g. `SecretKey secretKey = new SecretKeySpec("key".getBytes(), "AES"); secretKey.destroy();`) does not* work, nor nullifying the backing byte-array from `secretKey.getEncoded()` as the SecretKeySpec based key

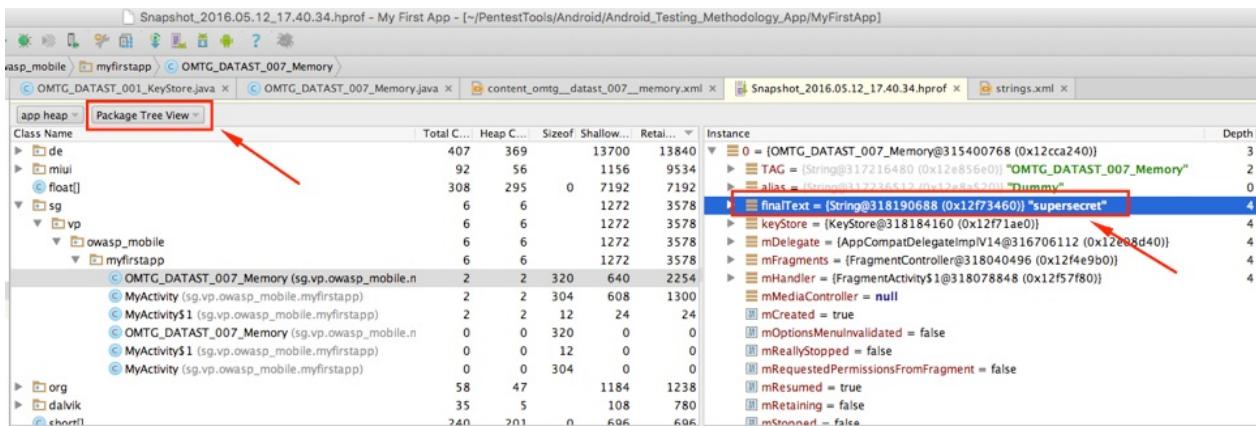
returns a copy of the backing byte-array. Therefore the developer should, in case of not using the `AndroidKeyStore` make sure that the key is wrapped and properly protected (see the remediation section for more details). Understand that an RSA key pair is based on `BigInteger` as well and therefore reside in memory after first use outside of the `AndroidKeyStore`. Lastly, 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.

Dynamic Analysis

For rudimentary analysis Android Studio built-in tools can be used. Android Studio includes tools in the *Android Monitor* tab to investigate the memory. Select the device and app you want to analyze in the *Android Monitor* tab and click on *Dump Java Heap* and a `.hprof` file will be created.



In the new tab that shows the `.hprof` file, the "Package Tree View" should be selected. Afterwards the package name of the app can be used to navigate to the instances of classes that were saved in the memory dump.



The `.hprof` file will be stored in the directory "captures", relative to the project path open within Android Studio. For deeper analysis of the memory dump the tool Eclipse Memory Analyzer (MAT) should be used.

Before the `.hprof` file can be opened in MAT it needs to be converted. The tool `hprof-conv` can be found in the Android SDK in the directory `platform-tools`.

```
./hprof-conv file.hprof file-converted.hprof
```

By using MAT, more functions are available, like usage of the Object Query Language (OQL). OQL is an SQL-like language that can be used to make queries in the memory dump. Analysis should be done on the dominator tree as only this contains the variables/memory of static classes.

To quickly discover potential sensitive data in the `.hprof` file, it is also useful to run the `string` command against it. When doing a memory analysis, check for sensitive information like:

- Password and/or usernames
- Decrypted information
- User or session related information
- Interaction with OS, e.g. reading file content

Remediation

In Java, no immutable structures should be used to carry secrets (e.g. `String`, `BigInteger`). Nullifying them will not be effective: the garbage collector might collect them, but they might remain in the JVM's heap for a longer period. Rather use byte-arrays (`byte[]`) or char-arrays (`char[]`) which are cleaned after the operations are done:

```

byte[] secret = null;
try{
    //get or generate the secret, do work with it, make sure you make no local copies
} finally {
    if (null != secret && secret.length > 0) {
        for (int i = 0; i < secret; i++) {
            array[i] = (byte) 0;
        }
    }
}

```

Also look into the best practices for [securely storing sensitive data in RAM](#)

Keys should be handled by the `AndroidKeyStore` or the `SecretKey` class needs to be adjusted. For a better implementation of the `SecretKey` one can use the `ErasableSecretKey` class below. This class consists of two parts:

- A wrapper class called `ErasableSecretKey` which takes care of building up the internal key, adding a clean method and a static convenience method. You can call the `getKey()` on a `ErasableSecretKey` to get the actual key.
- An internal `InternalKey` class which implements `javax.crypto.SecretKey, Destroyable`, so you can actually destroy it and it will behave as a `SecretKey` from JCE. The `destroyable` implementation first sets null bytes to the internal key and then it will put null as a reference to the `byte[]` representing the actual key. As you can see the `InternalKey` does not provide a copy of its internal `byte[]` representation, instead it gives the actual version. This will make sure that you will no longer have copies of the key in many parts of your application memory.

```

public class ErasableSecretKey implements Serializable {

    public static final int KEY_LENGTH = 256;

    private java.security.Key secKey;

    // Do not try to instantiate it: use the static methods.
    // The static construction methods only use mutable structures or create a new key
    directly.
    protected ErasableSecretKey(final java.security.Key key) {
        this.secKey = key;
    }

    //Create a new `ErasableSecretKey` from a byte-array.
    //Don't forget to clean the byte-array when you are done with the key.
    public static ErasableSecretKey fromByte(byte[] key) {
        return new ErasableSecretKey(new SecretKey.InternalKey(key, "AES"));
    }
}

```

```

//Create a new key. Do not forget to implement your own 'Helper.getRandomKeyBytes()
').
public static ErasableSecretKey newKey() {
    return fromByte(Helper.getRandomKeyBytes());
}

//clean the internal key, but only do so if it is not destroyed yet.
public void clean() {
    try {
        if (this.getKey() instanceof Destroyable) {
            ((Destroyable) this.getKey()).destroy();
        }
    } catch (DestroyFailedException e) {
        //choose what you want to do now: so you could not destroy it, would you run on? Or rather inform the caller of the clean method informing him of the failure?
    }
}

//convenience method that takes away the null-check so you can always just call ErasableSecretKey.clearKey(thekeytobecleared)
public static void clearKey(ErasableSecretKey key) {
    if (key != null) {
        key.clean();
    }
}

//internal key class which represents the actual key.
private static class InternalKey implements javax.crypto.SecretKey, Destroyable {
    private byte[] key;
    private final String algorithm;

    public InternalKey(final byte[] key, final String algorithm) {
        this.key = key;
        this.algorithm = algorithm;
    }

    public String getAlgorithm() {
        return this.algorithm;
    }

    public String getFormat() {
        return "RAW";
    }

    //Do not return a copy of the byte-array but the byte-array itself. Be careful
    : clearing this byte-array, will clear the key.
    public byte[] getEncoded() {
        if(null == this.key){
            throw new NullPointerException();
        }
        return this.key;
    }
}

```

```
//destroy the key.  
public void destroy() throws DestroyFailedException {  
    if (this.key != null) {  
        Arrays.fill(this.key, (byte) 0);  
    }  
  
    this.key = null;  
}  
  
public boolean isDestroyed() {  
    return this.key == null;  
}  
}  
  
public final java.security.Key getKey() {  
    return this.secKey;  
}  
}
```

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage
- M2 - Insecure Data Storage

OWASP MASVS

- V2.10: "The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use."

CWE

- CWE-316 - Cleartext Storage of Sensitive Information in Memory

Tools

- 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>

Testing the Device-Access-Security Policy

Overview

Apps that are processing or querying sensitive information should ensure that they are running in a trusted and secured environment. In order to be able to achieve this, the app can enforce the following local checks on the device:

- PIN or password set to unlock the device
- Usage of a minimum Android OS version
- Detection of activated USB Debugging
- Detection of encrypted device
- Detection of rooted device (see also "Testing Root Detection")

Static Analysis

In order to be able to test the device-access-security policy that is enforced by the app, a written copy of the policy needs to be provided. The policy should define what checks are available and how they are enforced. For example one check could require that the app only runs on Android Marshmallow (Android 6.0) or higher and the app is closing itself or showing a warning if the app is running on an Android version < 6.0.

The functions within the code that implement the policy need to be identified and checked if they can be bypassed.

Dynamic Analysis

The dynamic analysis depends on the checks that are enforced by app and their expected behavior and need to be validated if they can be bypassed.

Remediation

Different checks on the Android device can be implemented by querying different system preferences from [Settings.Secure](#). The [Device Administration API](#) offers different mechanisms to create security aware applications, that are able to enforce password policies or encryption of the device.

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage

OWASP MASVS

- V2.11: "The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode."

CWE

- N/A

Verifying User Education Controls

Overview

Educating users is a crucial part in the usage of mobile apps. Even though many security controls are already in place, they might be circumvented or misused through the user.

The following list shows potential warnings or advises for a user when opening the app the first time and using it:

- Showing a list of what kind of data is stored locally and remotely. This can also be a link to an external resource as the information might be quite extensive.
- If a new user account is created within the app it should show the user if the password provided is considered secure and applies to the password policy.
- If the user is installing the app on a rooted device a warning should be shown that this is dangerous and deactivates security controls at OS level and is more likely to be prone to malware. See also "Testing Root Detection" for more details.
- If a user installed the app on an outdated Android version a warning should be shown. See also "Testing the Device-Access-Security Policy" for more details.

Static Analysis

A list of implemented education controls should be provided. The controls should be verified in the code if they are implemented properly and according to best practices.

Dynamic Analysis

After installing the app and also while using it, it should be checked if any warnings are shown to the user, that have an educational purpose and are aligned with the defined education controls.

Remediation

Warnings should be implemented that address the key points listed in the overview section.

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage

OWASP MASVS

- V2.12: "The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app."

CWE

- N/A

Testing Cryptography in Android Apps

Verifying the Configuration of Cryptographic Standard Algorithms

Overview

A general rule in app development is that one should never attempt to invent their own cryptography. In mobile apps in particular, any form of crypto should be implemented using existing, robust implementations. In 99% of cases, this simply means using the data storage APIs and cryptographic libraries that come with the mobile OS.

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.

Remediation

Use cryptographic algorithm configurations that are currently considered strong, such those from [NIST](#) and [BSI](#). See also the "Remediation" section in the "Cryptography for Mobile Apps" chapter.

References

OWASP Mobile Top 10

- M6 - Broken 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"

CWE

- CWE-326: Inadequate Encryption Strength

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);  
}
```

Identify all instances of `SecureRandom` that are not created using the default constructor. Specifying the seed value may reduce randomness.

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).

Remediation

Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations with adequate length seeds. 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));
    }
}
```

References

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

Info

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

Testing Local Authentication in Android Apps

In local authentication, the app authenticates the user against credentials stored on the device itself. In other words, the user "unlocks" the app or some functionality within the app with a PIN, password or fingerprint that is verified only locally. This is sometimes done to allow users to more easily resume an existing session with the remote service, or as a means of step-up authentication to protect some critical functionality.

Testing Biometric Authentication

Overview

Android 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 (success, failure or error).

By using the fingerprint API in conjunction with the Android KeyGenerator class, apps can create a cryptographic key that must be "unlocked" with the user's fingerprint. This can be used to implement more convenient forms of user login. For example, to allow users access to a remote service, a symmetric key can be created and used to encrypt the user PIN or authentication token. By calling `setUserAuthenticationRequired(true)` when creating the key, it is ensured that the user must re-authenticate using their fingerprint to retrieve it. The encrypted authentication data itself can then be saved using regular storage (e.g. SharedPreferences).

Apart from this relatively reasonable method, fingerprint authentication can also be implemented in unsafe ways. For instance, developers might opt to assume successful authentication based solely on whether the `onAuthenticationSucceeded` callback is called. This event however isn't proof that the user has performed biometric authentication - such a check can be easily patched or bypassed using instrumentation. Leveraging the Keystore is the only way to be reasonably sure that the user has actually entered their fingerprint.

Static Analysis

Search for calls of `FingerprintManager.authenticate()`. The first parameter passed to this method should be a `cryptoobject` instance. [CryptoObject](#) is a wrapper class for the crypto objects supported by FingerprintManager. If this parameter is set to `null`, the fingerprint

auth is purely event-bound, which likely causes a security issue.

Trace back the creation of the key used to initialize the cipher wrapped in the CryptoObject.

Verify that the key was created using the `KeyGenerator` class, and that

`setUserAuthenticationRequired(true)` was called when creating the `KeyGenParameterSpec` object (see also the code samples below).

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, or some value derived from the secret.

Dynamic Analysis

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

Remediation

Fingerprint authentication should be implemented along the following lines:

Check whether fingerprint authentication is possible. The device must run Android 6.0 or higher (SDK 23+) and feature a fingerprint sensor. There are two pre-requisites that you need to check:

- The user must have protected their lockscreen

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

- Fingerprint hardware must be available:

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

- At least one finger should be registered:

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

- The application should have permission to ask for the users fingerprint:

```
context.checkSelfPermission(Manifest.permission.USE_FINGERPRINT) == PermissionResult.PERMISSION_GRANTED;
```

If any of those checks failed, the option for fingerprint authentication should not be offered.

When setting up fingerprint authentication, create a new AES key using the `KeyGenerator` class. Add `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();
```

Please note, that since Android 7 you can use the

`setInvalidatedByBiometricEnrollment(boolean value)` as a method of the builder. If you set this to true, then the fingerprint will not be invalidated when new fingerprints are enrolled. Even though this might provide user-convenience, it opens up a problem area when possible attackers are somehow able to social-engineer their fingerprint in.

To perform encryption or decryption, create a `Cipher` object and initialize it with the AES key.

```
SecretKey keyspec = (SecretKey)keyStore.getKey(KEY_ALIAS, null);

if (mode == Cipher.ENCRYPT_MODE) {
    cipher.init(mode, keyspec);
```

Note that the key cannot be used right away - it has to be authenticated through the `FingerprintManager` first. This involves wrapping `Cipher` into a `FingerprintManager.CryptoObject` which is passed to `FingerprintManager.authenticate()`.

```
cryptoObject = new FingerprintManager.CryptoObject(cipher);
fingerprintManager.authenticate(cryptoObject, new CancellationSignal(), 0, this, null);
```

If authentication succeeds, the callback method

`onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result)` is called, and the authenticated `CryptoObject` can be retrieved from the authentication result.

```
public void authenticationSucceeded(FingerprintManager.AuthenticationResult result) {  
    cipher = result.getCryptoObject().getCipher();  
  
    (... do something with the authenticated cipher object ...)  
}
```

Please bare in mind that the keys might not be always in secure hardware, for that you can do the following to validate the posture of the key:

```
SecretKeyFactory factory = SecretKeyFactory.getInstance(getEncryptionKey().getAlgorithm(), ANDROID_KEYSTORE);  
        KeyInfo secetkeyInfo = (KeyInfo) factory.getKeySpec(yourencryptionkeyhere, KeyInfo.class);  
secetkeyInfo.isInsideSecureHardware()
```

Please note that, on some systems, you can make sure that the biometric authentication policy itself is hardware enforced as well. This is checked by:

```
keyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware();
```

For a full example, see the [blog article by Deivi Taka](#).

References

OWASP Mobile Top 10 2016

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

OWASP MASVS

- 4.7 - "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

Testing Network Communication in Android Apps

Testing Endpoint Identity Verification

Overview

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](#).

Static Analysis

The static analysis approach is to decompile an application, if the source code was not provided. There are two main issues related with validating TLS connection that should be verified in the code:

- the first one is verification if a certificate comes from a trusted source and
- the second one is to check whether the endpoint server presents the right certificate

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());

```

Hostname Verification

Another security fault in TLS implementation is 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, SSLSocket 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](#).

Remediation

Ensure that the host name and certificate are verified correctly. Examples and common pitfalls can be found in the [official Android documentation](#).

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 valid 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 SSL Pinning

Overview

Certificate pinning allows to hard-code the certificate or parts of it into the app that is known to be used by the server. This technique is used to reduce the threat of a rogue CA and CA compromise. Pinning the server's certificate takes the CA out of the game. Mobile apps that implement certificate pinning only can connect to a limited numbers of servers, as a small list of trusted CAs or server certificates are hard-coded in the application.

Static Analysis

The process to implement the SSL 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 SSL 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/UwQAapahrjC0jYI3oLUX5AQxPBR02Jz6/E2pt0IeL
XA=")
        .build())
    .build();

```

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.

Remediation

The SSL pinning process should be implemented as described on the static analysis section. For further information please check the [OWASP certificate pinning guide](#).

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."

CWE

- CWE-295 - Improper Certificate Validation

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. (e.g. 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()`. 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. See the remediation section for possible examples.

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`.

Remediation

To make sure that the application is using a patched security provider, the application needs to use the `ProviderInstaller` class which comes with the Google Play services. The Google Play Services can be installed as a dependency in the build.gradle file by adding `compile 'com.google.android.gms:play-services-gcm:x.y.z'` (where x.y.z is a version number) in the dependencies block. Next, the `ProviderInstaller` needs to be called as early as possible by a component of the application. Here are two adjusted examples from Google on how this could work. 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. The first example shows how to do the installation synchronously, the second example shows how to do it asynchronously.

```

//this is a sync adapter that runs in the background, so you can run the synchronous p
atching.

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.
}

}

```

```

//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
        // onPostResume.
        mRetryProviderInstall = true;
    }
}
```

```
/**  
 * On resume, check to see if we flagged that we need to reinstall the  
 * provider.  
 */  
@Override  
protected void onPostResume() {  
    super.onPostResult();  
    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.).  
}  
}
```

The Android developer documentation also describes [how to update your security provider to protect against SSL exploits](#).

References

OWASP Mobile Top 10 2016

OWASP MASVS

- V5.6 "The app only depends on up-to-date connectivity and security libraries."

CWE

Testing Platform Interaction on Android

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 allow 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. 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.

Remediation

Only permissions that are needed within the app should be requested in the Android Manifest file and all other permissions should be removed.

Developers should take care to secure sensitive IPC components with the `signature` protection level, which will only allow applications signed with the same certificate to access the component.

References

[OWASP Mobile Top 10 2016](#)

- M1 - Improper Platform Usage -

https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage

OWASP MASVS

- V6.1: "The app only requires the minimum set of permissions necessary."

CWE

- CWE-250 - Execution with Unnecessary Privileges

Info

- [JeffSix] - Jeff Six, An In-Depth Introduction to the Android Permission Model -
https://www.owasp.org/images/c/ca/ASDC12-An_InDepth_Introduction_to_the_Android_Permissions_Modeland_How_to_Secure_MultiComponent_Applications.pdf

Tools

- AAPT - http://elinux.org/Android_aapt
- Drozer - <https://github.com/mwrlabs/drozer>

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.

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 category `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://0123
456789"
```

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.

Remediation

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".

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage -

https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage

OWASP MASVS

- V6.3: "The app does not export sensitive functionality via custom URL schemes, unless

these mechanisms are properly protected."

CWE

- CWE-939 - Improper Authorization in Handler for Custom URL Scheme

Tools

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

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".

Apart from that, remember that using the permission tag (`android:permission`) will also limit the exposure of a component to other applications.

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:la
aunchMode="singleTask" android:name=".MainLoginActivity" android:windowSoftInputMode="a
djustResize|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:ex
ported="true" android:finishOnTaskLaunch="true" android:label="@string/title_activity_
_file_select" android:name=".FileSelectActivity"/>
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true" android:ex
ported="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.MyBroadCast
Receiver">
    <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.

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);
        }
    }
}

```

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)
```

Remediation

If not strictly required, be sure that your 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.

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()`.

`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.

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.

References

OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage -
https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage

OWASP MASVS

- V6.4: "The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected."

CWE

- CWE-749 - Exposed Dangerous Method or Function

Tools

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

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 and execute its command.

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 Identify 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").

Remediation

JavaScript is disabled by default in a WebView and if not needed shouldn't be enabled. This reduces 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 your app is running on these devices, it must confirm that WebView objects [display only trusted content](#).

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

- V6.5: "JavaScript is disabled in WebViews unless explicitly required."

CWE

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

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.
- `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.

```
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.

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.

Remediation

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

```
//Should an attacker somehow find themselves in a position to inject script into a Web  
View, then they could exploit the opportunity to access local resources. This can be s  
omewhat 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 pub  
lic method setAllowFileAccess.  
webView.getSettings().setAllowFileAccess(false);  
  
webView.getSettings().setAllowFileAccessFromFileURLs(false);  
  
webView.getSettings().setAllowUniversalAccessFromFileURLs(false);  
  
webView.getSettings().setAllowContentAccess(false);
```

Access to files in the file system can be enabled and disabled for a WebView with `setAllowFileAccess()`. File access is enabled by default and should be deactivated if not needed. Note that this enables or disables [file system access](#) only. Assets and resources are still accessible using `file:///android_asset` and `file:///android_res`.

Create a white-list that defines the web pages and it's protocols that are allowed to be loaded locally and remotely. Loading web pages from the external storage should be avoided as they are read and writable for all users in Android. Instead they should be placed in the assets directory of the App.

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.

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

- V6.6: "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."

CWE

N/A

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()` "Method 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, a special annotation 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 MITM, he can directly call the exposed Java methods in order to exploit them.

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 using 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](#).

Remediation

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

Moreover pay attention if you imported libraries, e.g. for advertising, because they could use the methods mentioned before and bring the vulnerabilities in your app.

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>
```

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

- V6.7: "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-749 - Exposed Dangerous Method or Function

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](#), [Jackson](#) and others. 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](#).

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 reassess 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?).

Remediation

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 before you continue or the 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 you can actively use it (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.

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

- V6.8: "Object serialization, if any, is implemented using safe serialization APIs."

CWE

N/A

Testing Root Detection

Overview

Checking the integrity of the environment where the app is running is getting more and more common on the Android platform. Due to the usage of rooted devices several fundamental security mechanisms of Android are deactivated or can easily be bypassed by any app. Apps that process sensitive information or have built in largely intellectual property (IP), like gaming apps, might want to avoid to run on a rooted phone to protect data or their IP.

Keep in mind that root detection is not protecting an app from attackers, but can slow down an attacker dramatically and higher the bar for successful local attacks. Root detection should be considered as part of a broad security-in-depth strategy, to be more resilient against attackers and make analysis harder.

Static Analysis

Root detection can either be implemented by leveraging existing root detection libraries, such as [Rootbeer](#), or by implementing manually checks.

Check the source code for the string `rootbeer` and also the `gradle` file, if a dependency is defined for Rootbeer:

```
dependencies {  
    compile 'com.scottyab:rootbeer-lib:0.0.4'  
}
```

If this library is used, code like the following might be used for root detection.

```

RootBeer rootBeer = new RootBeer(context);
if(rootBeer.isRooted()){
    //we found indication of root
}else{
    //we didn't find indication of root
}

```

If the root detection is implemented from scratch, the following should be checked to identify functions that contain the root detection logic. The following checks are the most common ones for root detection:

- Checking for settings/files that are available on a rooted device, like verifying the `BUILD` properties for test-keys in the parameter `android.os.build.tags`.
- Checking permissions of certain directories that should be read-only on a non-rooted device, but are read/write on a rooted device.
- Checking for installed apps that allow or support rooting of a device, like verifying the presence of `Superuser.apk`.
- Checking available commands, like is it possible to execute `su` and being root afterwards.

Dynamic Analysis

A debug build with deactivated root detection should be provided in a white box test to be able to apply all test cases to the app.

In case of a black box test, an implemented root detection can be challenging if for example the app is immediately terminated because of a rooted phone. Ideally, a rooted phone is used for black box testing and might also be needed to disable SSL Pinning. To deactivate SSL Pinning and allow the usage of an interception proxy, the root detection needs to be defeated first in that case. Identifying the implemented root detection logic without source code in a dynamic scan can be fairly hard.

By using the Xposed module `RootCloak` it is possible to run apps that detect root without disabling root. Nevertheless, if a root detection mechanism is used within the app that is not covered in RootCloak, this mechanism needs to be identified and added to RootCloak in order to disable it.

Other options are dynamically patching the app with Friday or repackaging the app. This can be as easy as deleting the function in the smali code and repackage it, but can become difficult if several different checks are part of the root detection mechanism. Dynamically patching the app can also become difficult if countermeasures are implemented that prevent runtime manipulation/tampering.

Otherwise it should be switched to a non-rooted device in order to use the testing time wisely and to execute all other test cases that can be applied on a non-rooted setup. This is of course only possible if the SSL Pinning can be deactivated for example in smali and repackaging the app.

Remediation

To implement root detection within an Android app, libraries can be used like `RootBeer`. The root detection should either trigger a warning to the user after start, to remind him that the device is rooted and that the user can only proceed on his own risk. Alternatively, the app can terminate itself in case a rooted environment is detected. This decision is depending on the business requirements and the risk appetite of the stakeholders.

References

OWASP Mobile Top 10 2016

- M8 - Code Tampering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M8-Code_Tampering
- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

OWASP MASVS

- V6.9: "The app detects whether it is being executed on a rooted or jailbroken device. Depending on the business requirement, users are warned, or the app is terminated if the device is rooted or jailbroken."

CWE

N/A

Tools

- RootCloak - <http://repo.xposed.info/module/com.devadvance.rootcloak2>

Testing 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. 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.

If you don't have the APK available locally, pull it from the device first:

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

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.

Dynamic Analysis

Static analysis should be used to verify the APK signature.

Remediation

Developers need to make sure that [release builds](#) are signed with the appropriate certificate. In Android Studio, this can be done manually or by creating a signing configuration and assigning it to the release build type.

The signing configuration can be managed through the Android Studio GUI 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 your application for release](#) is also available in the official Android developer documentation.

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."

CWE

N/A

Tools

- jarsigner - <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html>

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

Static Analysis

Check in `AndroidManifest.xml` whether the `android:debuggable` attribute is set:

```

<?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>

```

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(ad  
b),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3003/inet),3006/net_  
bw_stats) 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.

Remediation

In the `AndroidManifest.xml` file, set the `android:debuggable` flag to false, as shown below:

```
<application android:debuggable="false">  
...  
</application>
```

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

- V7.2: "The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable)."

CWE

- CWE-215 - Information Exposure Through Debug Information

Tools

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

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 byte-code 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 Analysis

Static analysis should be used to verify for debugging symbols.

Remediation

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`.

Add the following to build.gradle:

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

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

- V7.3: "Debugging symbols have been removed from native binaries."

CWE

- CWE-215 - Information Exposure Through Debug Information

Tools

- GNU nm - https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_4.html

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();
}

```

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

```

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

Remediation

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.

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

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

- V7.4: "Debugging code has been removed, and the app does not log verbose errors or debugging messages."

CWE

- CWE-215 - Information Exposure Through Debug Information
- CWE-489 - Leftover Debug Code

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.

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.

Remediation

All functions in the app that process data that is coming from external and through the UI should be validated.

- 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.

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

Tools

- Drozer - <https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf>

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.
- 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.

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.

Remediation

There are a few best practices a developer should do:

- Ensure that the application use a well-designed and unified scheme to [handle exceptions](#).
- 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.
- When executing operations that involve high risk information, make sure you wipe the information in the finally block in java:

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

- Add a general exception-handler for uncaught exceptions 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 you need to call the initializer 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();
}

```

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

- V7.5: "The app catches and handles possible exceptions."
- V7.6: "Error handling logic in security controls denies access by default."

CWE

- CWE-388 - Error Handling

Tools

- Xposed - <http://repo.xposed.info/>

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. 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.

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;
    }
}

```

Remediation

ProGuard should be used to strip unneeded debugging information from the Java bytecode. 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'
        }
    }
    ...
}

```

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

- V7.8: "Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated."

CWE

- CWE-656 - Reliance on Security Through Obscurity

Tools

- ProGuard - <https://www.guardsquare.com/en/proguard>

Tampering and Reverse Engineering on Android

Its openness makes Android a favorable environment for reverse engineers. However, dealing with both Java and native code can make things more complicated at times. In the following chapter, we'll look at some peculiarities of Android reversing and OS-specific tools as processes.

In comparison to "the other" mobile OS, Android offers some big advantages to reverse engineers. Because Android is open source, you can study the source code of the Android Open Source Project (AOSP), modify the OS and its standard tools in any way you want. Even on standard retail devices, it is easily 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's 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 used on purpose to confuse reverse engineers. Developers sometimes use the native layer to "hide" data and functionality, or may structure their apps such that execution frequently jumps between the two layers. This can complicate things for reverse engineers (to be fair, there might also be legitimate reasons for using JNI, such as improving performance or supporting legacy code).

You'll need a working knowledge about both the Java-based Android environment and the Linux OS and Kernel that forms the basis of Android - or better yet, know all these components inside out. Plus, they need the right toolset to deal with both native code and bytecode running inside the Java virtual machine.

Note that in the following sections we'll use the [OWASP Mobile Testing Guide Crackmes](#) as examples for demonstrating various reverse engineering techniques, so expect partial and full spoilers. We encourage you to have a crack at the challenges yourself before reading on!

What You Need

At the very least, you'll need [Android Studio](#), which comes with the Android SDK, platform tools and emulator, as well as a manager app for managing the various SDK versions and framework components. With Android Studio, you also get an SDK Manager app that lets

you install the Android SDK tools and manage SDKs for various API levels, as well as the emulator and an AVD Manager application to create emulator images. 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 as well as other tools that interface with the Android platform. In general, these tools are backward-compatible, so you need only one version of those installed.
- The Android NDK. This is the Native Development Kit that contains prebuilt toolchains for cross-compiling native code for different architectures.

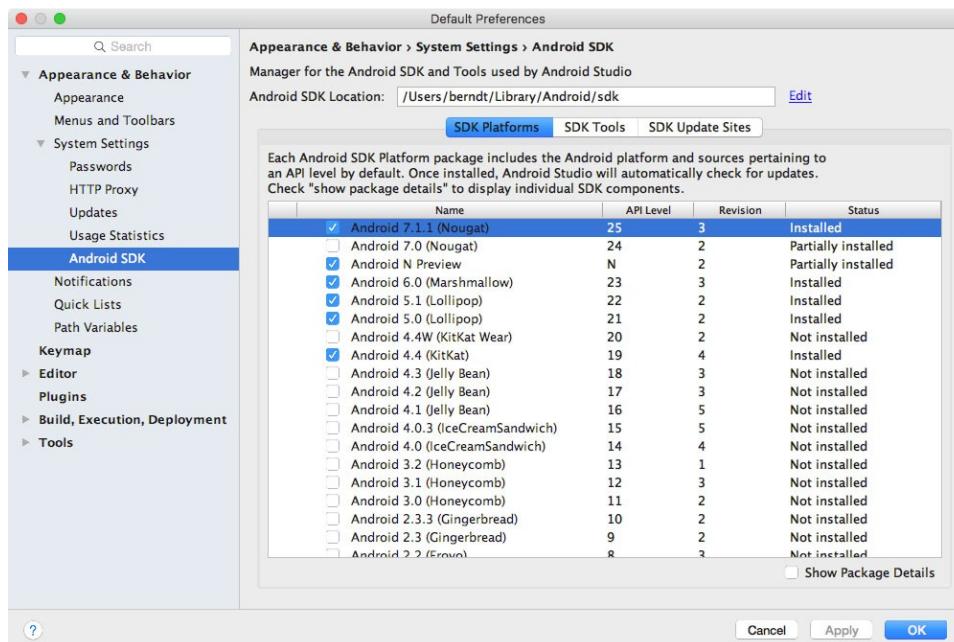
In addition to the SDK and NDK, you'll also something to make Java bytecode more human-friendly. Fortunately, Java decompilers generally deal well with Android bytecode. Popular free decompilers include [JD](#), [JAD](#), [Procyon](#) and [CFR](#). For convenience, we have packed some of these decompilers 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 examples below).

Other than that, it's 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 below.

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



Depending on your OS, installed SDKs are found at the following location:

Windows :

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

MacOS :

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

Note: On Linux, you'll need pick your own SDK location. Common locations are `/opt` , `/srv` , and `/usr/local` .

Setting up the Android NDK

The Android NDK contains prebuilt versions of the native compiler and toolchain. Traditionally, both the GCC and Clang compilers were supported, but active support for GCC ended with revision 14 of the NDK. What's the right version to use depends on both the device architecture and host OS. The prebuilt toolchains are located in the `toolchains` directory of the NDK, which contains one subdirectory per 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>

In addition to 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. Available native APIs vary by Android API level. Possible sysroots for respective Android API levels reside under `$NDK/platforms/`, each API-level directory contains subdirectories for the various CPUs and architectures.

One possibility to set 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 later. Run the following command, or add it to your `.bash_profile` or other startup script.

```
$ export TOOLCHAIN=/tmp/android-7-toolchain
```

Building a Reverse Engineering Environment For Free

With a little effort you can build a reasonable GUI-powered reverse engineering environment for free.

For navigating the decompiled sources we recommend using [IntelliJ](#), a relatively light-weight IDE that works great for browsing code and allows for 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 (note: This piece of advice is based on the author's personal bias).

If you don't mind looking at Smali instead of Java code, you can use the [smalidea plugin for IntelliJ](#) for debugging on the device Smalidea supports single-stepping through the bytecode, identifier renaming and 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.

More elaborate tasks such as program analysis and automated de-obfuscation can be achieved 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

Even though it is possible to work with a completely free setup, you might want to 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, this will save you a lot of time.

JEB

[JEB](#), a commercial decompiler, packs all the functionality needed for static and dynamic analysis of Android apps into an all-in-one package, is reasonably reliable and you get quick 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 when dealing with ProGuard-obfuscated bytecode. Of course convenience like this doesn't come cheap - and since version 2.0 JEB has changed from a traditional licensing model to a subscription-based one, so you'll need to pay a monthly fee to use it.

IDA Pro

[IDA Pro](#) understands ARM, MIPS and of course Intel ELF binaries, plus it can deal with Java bytecode. It also comes with debuggers for both Java applications and native processes. With its capable disassembler and powerful scripting 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 somewhat basic – you get the Smali disassembly but not

much more. There's no navigating the package and class structure, and some things (such as renaming classes) can't be done which can make working with more complex Java apps a bit tedious.

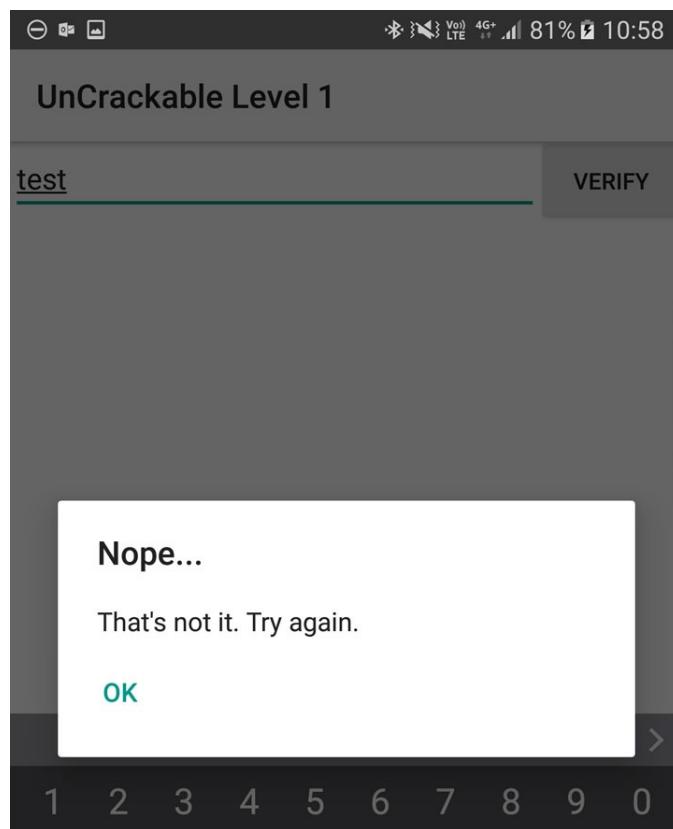
Reverse Engineering

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 runtime (dynamic analysis), or a combination of both.

Statically Analyzing Java Code

Unless some nasty, tool-breaking anti-decompilation tricks have been applied, Java bytecode can be converted back into source code without too many problems. 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!

Most likely, we're looking for a secret string stored somewhere inside the app, so the next logical step is to take a look inside. First, unzip the APK file and have a 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/res/mipmap-xxxhdpi-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 case, all the Java bytecode and data related to the app is contained in a file named `classes.dex` in the app root directory. This file adheres to the Dalvik Executable Format (DEX), an Android-specific way of packaging Java programs. Most Java decompilers expect plain class files or JARs as input, so you need to convert the `classes.dex` file into a JAR first. This can be done using `dex2jar` or `enjarify`.

Once you have a JAR file, you can use any number of free decompilers to produce Java code. In this example, we'll use CFR as our decompiler of choice. CFR is under active development, and brand-new releases are made available regularly on the author's website. Conveniently, CFR has been released under a MIT license, which means that it can be used freely for any purposes, even though its source code is not currently available.

The easiest way to run CFR is through `apkx`, which also packages `dex2jar` and automates the extracting, conversion and decompilation steps. Install it as follows:

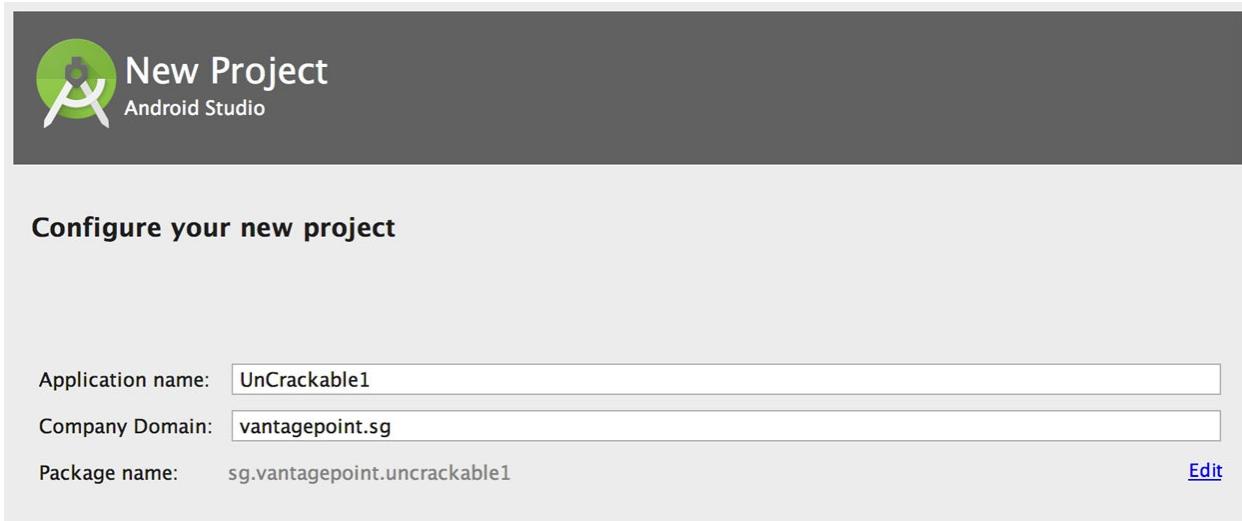
```
$ git clone https://github.com/b-mueller/apkx
$ cd apkx
$ sudo ./install.sh
```

This should copy `apkx` to `/usr/local/bin`. Run it on `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)
```

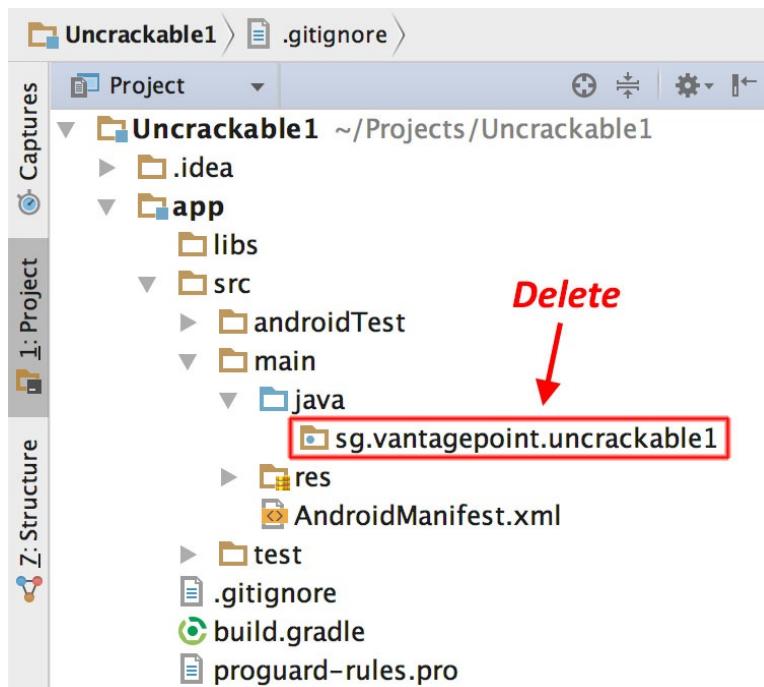
You should now find the decompiled sources in the directory `Uncrackable-Level1/src`. To view the sources, a simple text editor (preferably with syntax highlighting) is fine, but loading the code into a Java IDE makes navigation easier. Let's import the code into IntelliJ, which also gets us on-device debugging functionality as a bonus.

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, as IntelliJ uses the package name to identify the correct process.

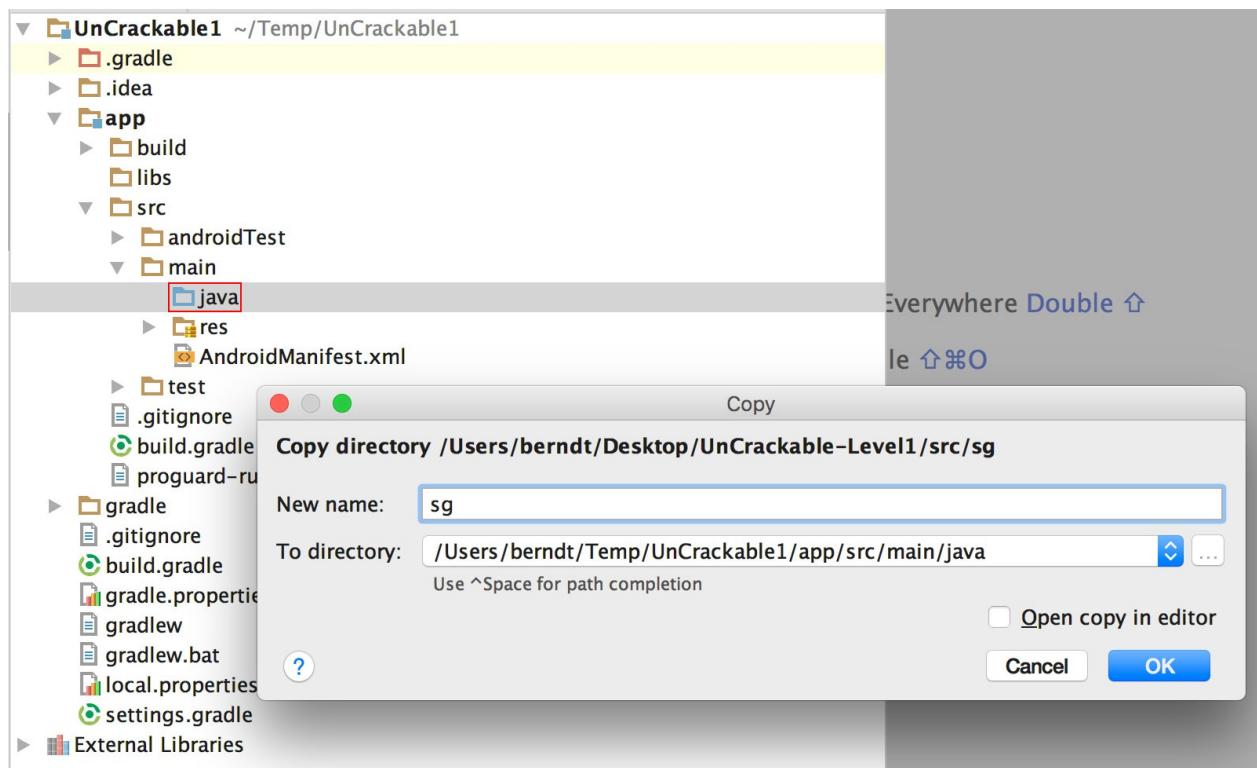


In the next dialog, pick any API number - we don't want to actually compile the project, so it really doesn't matter. Click "next" and choose "Add no Activity", then click "finish".

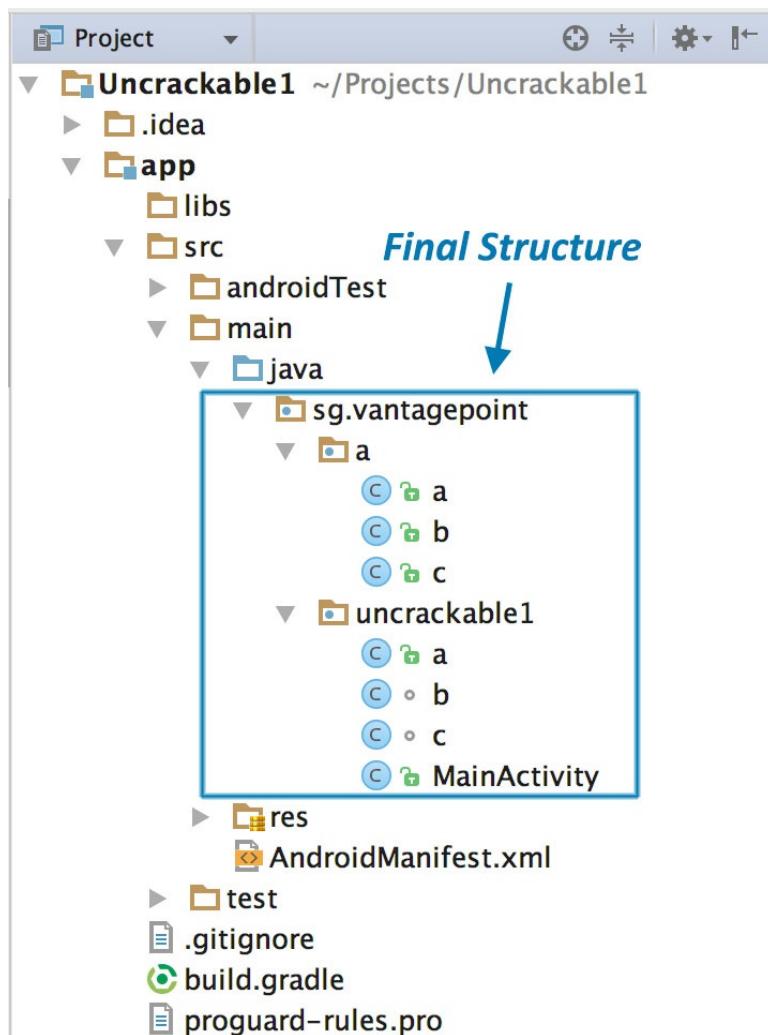
Once the project is created, expand the "1: Project" view on the left and navigate to the folder `app/src/main/java`. Right-click and delete the default package "sg.vantagepoint.uncrackable1" created by IntelliJ.



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 is done indexing the code, you can browse it just like any normal 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 bit more difficult to read, but with a fairly simple app like this one it won't cause you much of a headache - however, when analyzing a more complex app, it can get quite annoying.

A good practice to follow when analyzing obfuscated code is to annotate names of classes, methods and other identifiers as you go along. Open the `MainActivity` class in the package `sg.vantagepoint.a`. The method `verify` is what's called when you tap on the "verify" button. This method passes the user input to a static method called `a.a`, which returns a boolean value. It seems plausible that `a.a` is responsible for verifying whether the text entered by the user is valid or not, so we'll start refactoring 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.setPositiveButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}

```

Check Input

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 as you learn more about the class). `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 as follows:

```

public static boolean check_input(String string) {
    byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2
G0c=", (int)0);
    byte[] arrby2 = new byte[] {};
    try {
        arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d806cf50473c
c"), 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, we have a base64-encoded String that's passed to a function named `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 we are getting somewhere: It's simply standard AES-ECB. Looks like the base64 string stored in `arrby1` in `check_input` is a ciphertext, which is decrypted using 128bit AES, and then compared to the user input. As a bonus task, try to decrypt the extracted ciphertext and get the secret value!

An alternative (and faster) way of getting the decrypted string is by adding a bit of dynamic analysis into the mix - we'll revisit UnCrackable Level 1 later to show how to do this, so don't delete the project yet!

Statically Analyzing Native Code

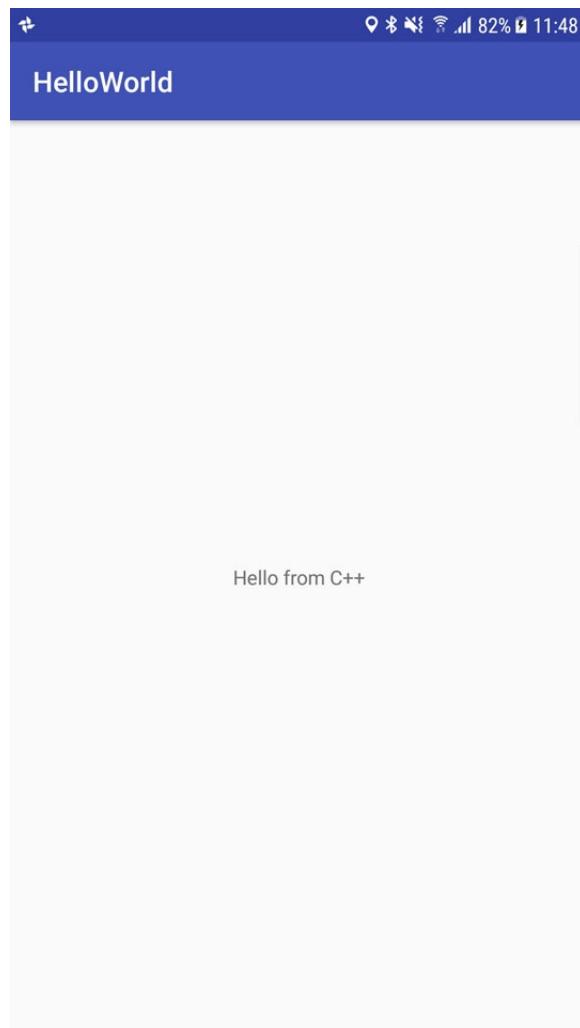
Dalvik and ART both support the Java Native Interface (JNI), which defines a way for Java code to interact with native code written in C/C++. Just like on other Linux-based operating systems, native code is packaged into ELF dynamic libraries ("*.so"), which are then loaded by the Android app during runtime using the `System.load` method.

Android JNI functions consist of native code compiled into Linux ELF libraries. It's pretty much 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 is not fully POSIX-compatible.

Download `HelloWorld-JNI.apk` from the OWASP MSTG repository and, optionally, install and run it on your emulator or Android device.

```
$ 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++". In fact, this is the default app Android generates when you create a new project with C/C++ support - enough however to show the basic principles of how JNI calls work.



Decompile the APK with `apkx`. This extract 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();
}

```

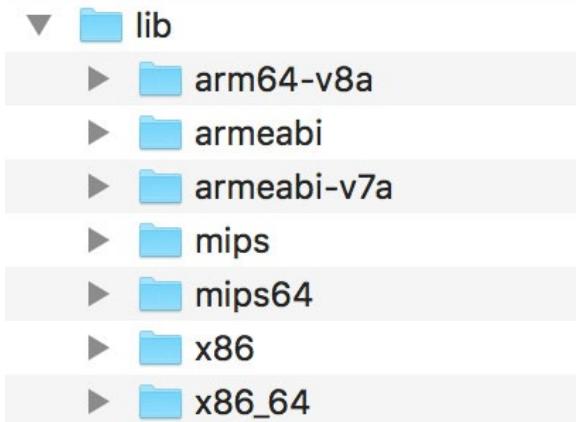
Note the declaration of `public native String stringFromJNI` at the bottom. The `native` keyword informs the Java compiler that the implementation for this method is provided in a native language. The corresponding function is resolved during runtime. Of course, this only works if a native library is loaded that exports a global symbol with the expected signature. This signature is composed of the package name, class name and method name. In our case for example, this means that the programmer must have implemented 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 a total of eight subdirectories named after different processor architectures. Each of this directories contains a version of the native library `libnative-lib.so`, compiled for the processor architecture in question. When `System.loadLibrary` is called, the loader selects the correct version based on what device the app is running on.



Following the naming convention mentioned above, we can expect the library to export a symbol named `Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI`. On Linux systems, you can retrieve the list of symbols using `readelf` (included in GNU binutils) or `nm`. On Mac OS, the same can be achieved 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_Mai
nActivity_stringFromJNI
```

This is the native function that gets eventually executed when the `stringFromJNI` native method is called.

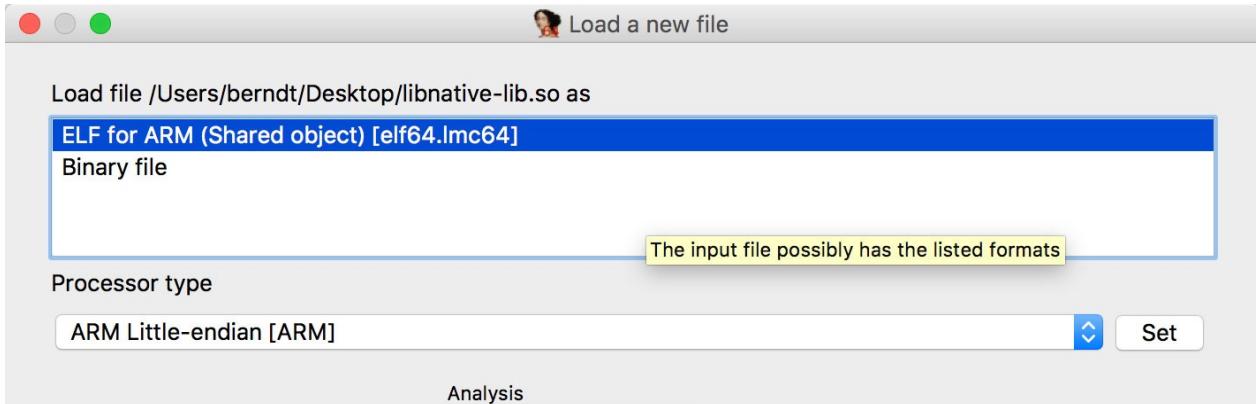
To disassemble the code, you can load `libnative-lib.so` into any disassembler that understands ELF binaries (i.e. every disassembler in existence). If the app ships with binaries for different architectures, you can theoretically pick the architecture you're most familiar with, as long as the disassembler knows how to deal with it. Each version is compiled from the same source and implements exactly the same functionality. However, if you're planning to debug the library on a live device later, it's usually wise to pick an ARM build.

To support both older and newer ARM processors, Android apps ship with multiple ARM builds compiled for different Application Binary Interface (ABI) versions. The ABI defines how the application's machine code is supposed to interact with the system at runtime. The following ABIs are supported:

- `armeabi`: ABI is for ARM-based CPUs that support at least the ARMv5TE instruction set.
- `armeabi-v7a`: This ABI extends `armeabi` to include several CPU instruction set extensions.
- `arm64-v8a`: ABI for ARMv8-based CPUs that support AArch64, the new 64-bit ARM architecture.

Most disassemblers will be able to deal with any of those architectures. Below, we'll be viewing the `armeabi-v7a` version in IDA Pro. It is located in `lib/armeabi-v7a/libnative-lib.so`. If you don't own an IDA Pro license, you can do the same thing with 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 it to jump to its address in the disassembly Window. "Ida View-A" should now show the disassembly of the function.

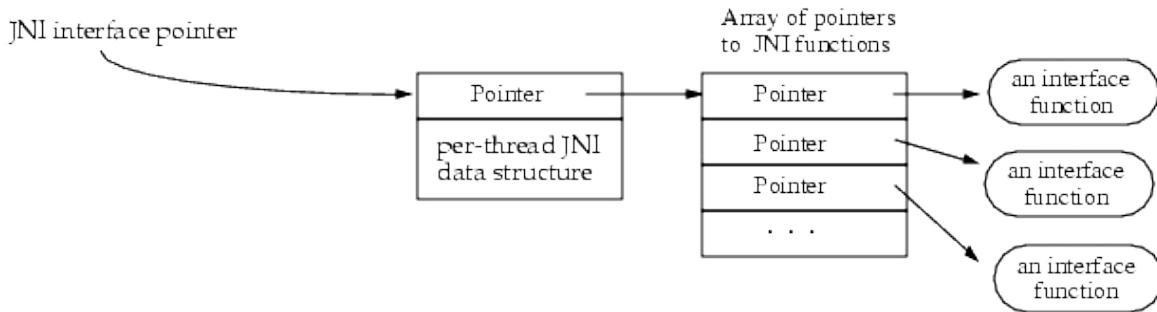
```

CODE16

EXPORT Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
LDR      R2, [R0]
LDR      R1, =(aHelloFromC - 0xE80)
LDR.W   R2, [R2,#0x29C]
ADD    R1, PC ; "Hello from C++"
BX     R2
; End of function Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI

```

Not a lot of code there, but let's analyze it. The first thing we 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 (located 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 the pc-relative offset of the string "Hello from C++" into R1. Note that this string is located directly after the end of the function block at offset 0xe84. The addressing relative to the program counter allows the code to run independent 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 into R2. This happens to be the `NewStringUTF` function. You can look the list of function pointers in `jni.h`, which is included in the Android NDK. The function prototype looks as follows:

```
jstring (*NewStringUTF)(JNIEnv*, const char*);
```

The function expects 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 ends.

Debugging and Tracing

So far, we've been using static analysis techniques without ever running our target apps. In the real world - especially when reversing more complex apps or malware - you'll find that pure static analysis is very difficult. Observing and manipulating an app during runtime makes it much, much easier to decipher its behavior. Next, we'll have a look at dynamic analysis methods that help you do just that.

Android apps support two different types of debugging: Java-runtime-level debugging using Java Debug Wire Protocol (JDWP) and Linux/Unix-style ptrace-based debugging on the native layer, both of which are valuable for reverse engineers.

Activating Developer Options

Since Android 4.2, the "Developer options" submenu is hidden by default in the Settings app. To activate it, you need to tap the "Build number" section of the "About phone" view seven times. Note that the location of the build number field can vary slightly on different devices - for example, on LG Phones, it is found under "About phone > Software information" instead. Once you have done this, "Developer options" will be shown at bottom of the Settings menu. Once developer options are activated, debugging can be enabled with the "USB debugging" switch.

Debugging Release Apps

Dalvik and ART support the Java Debug Wire Protocol (JDWP), a protocol used for communication between the debugger and the Java virtual machine (VM) which it debugs. JDWP is a standard debugging protocol that is 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).

Using 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 be using a JDWP debugger most of the time when debugging "normal" Android apps that don't do a lot of calls into native libraries.

In the following section, we'll show how to solve UnCrackable App for Android Level 1 using JDB only. Note that this is not an *efficient* way to solve this crackme - you can do it much faster using Frida and other methods, which we'll introduce later in the guide. It serves however well 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 typically the configuration on Android devices shipped to end users.

When reverse engineering apps, you'll often only have access to the release build of the target app. Release builds are not meant to be debugged - after all, that's what *debug builds* are for. If the system property `ro.debuggable` set to "0", Android disallows both JDWP and native debugging of release builds, and although this is easy to bypass, you'll still likely encounter some limitations, such as a lack of line breakpoints. Nevertheless, even an imperfect debugger is still an invaluable tool - being able to inspect the runtime state of a program makes it a *lot* easier to understand what's going on.

To "convert" a release build release 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 do this, 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 named "androiddebugkey".

The Java standard distribution includes `keytool` for managing keystores and certificates. You can create your own signing certificate and key and add it to the debug keystore as follows:

```
$ keytool -genkey -v -keystore ~/.android/debug.keystore -alias signkey -keyalg RSA -keysize 2048 -validity 20000
```

With a certificate available, you can now repackage the app using the following steps. Note that the Android Studio build tools directory must be in path for this to work - it is located at `[SDK-Path]/build-tools/[version]`. The `zipalign` and `apksigner` tools are found in this directory. Repackage UnCrackable-Level1.apk as follows:

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">
```

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. Note that in this case, `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
```

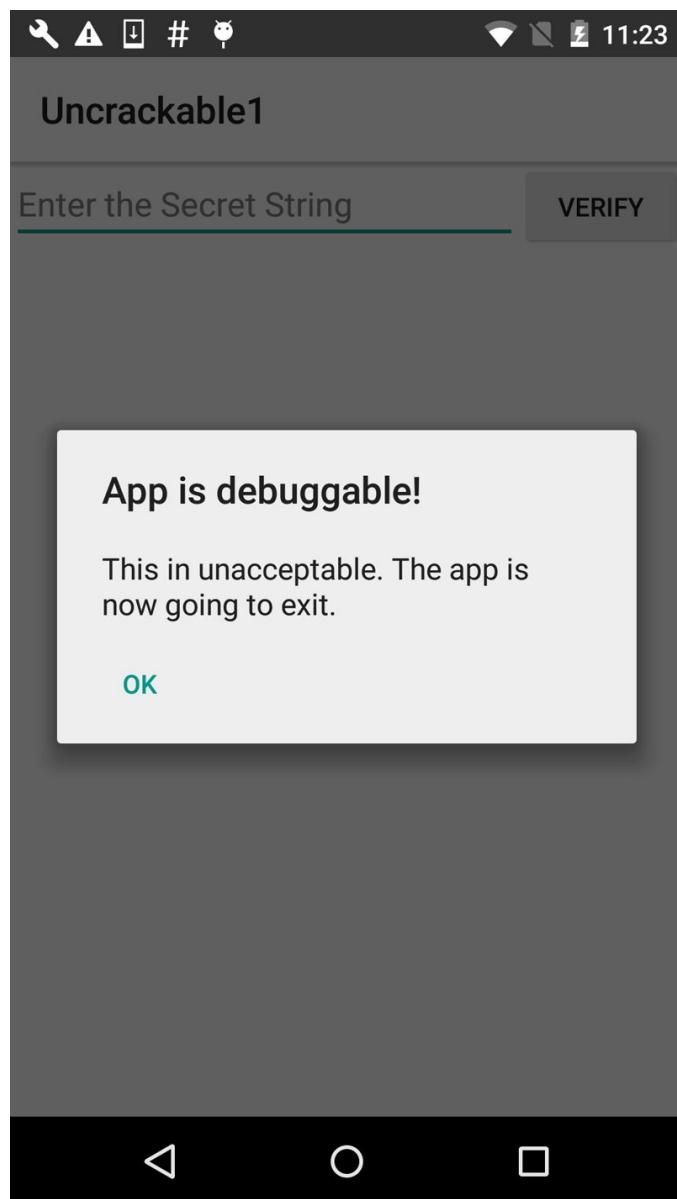
1. Reinstall the app:

```
$ adb install UnCrackable-Repackaged.apk
```

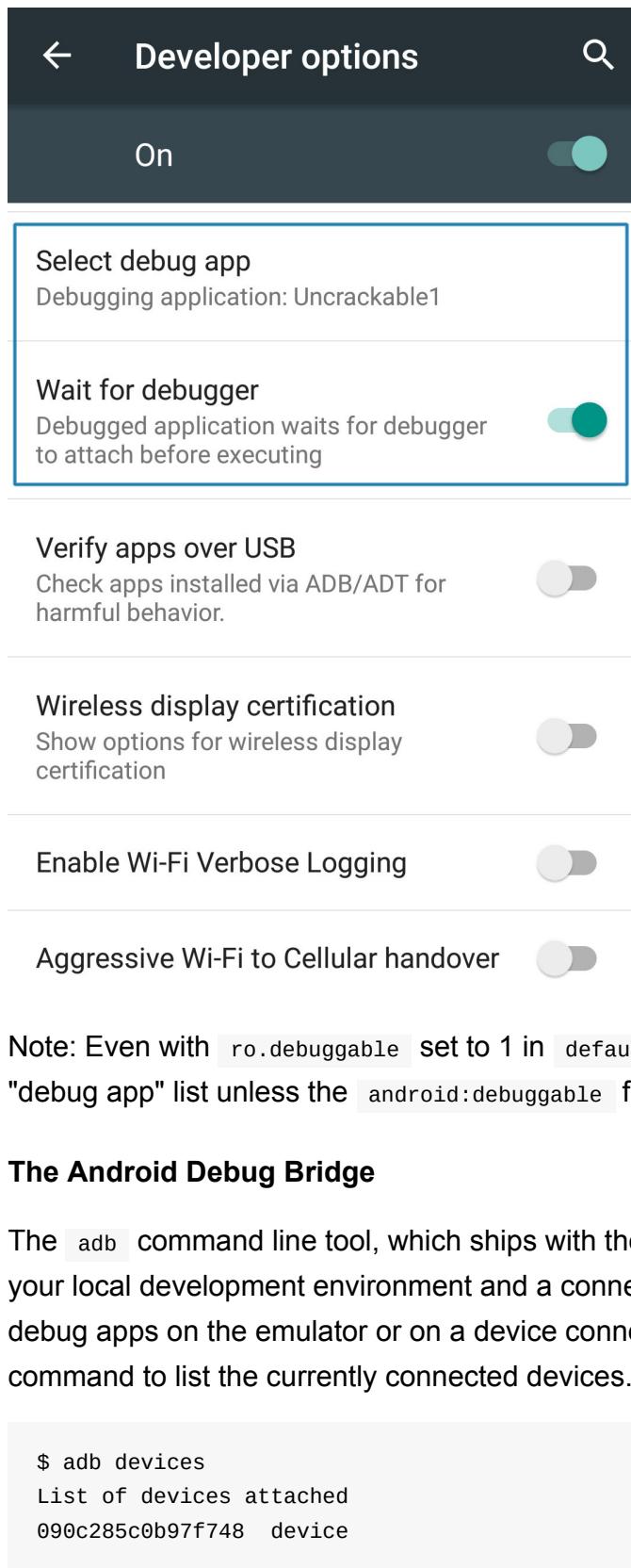
The 'Wait For Debugger' Feature

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 the OK button.

Fortunately, Android's Developer options contain the useful "Wait for Debugger" feature, which allows you to automatically suspend a selected app doing startup until a JDWP debugger connects. By using 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 Settings, 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. Commonly you'll debug apps on the emulator or on a device connected via USB. Use the `adb devices` command to list the currently 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 TCP connections to this socket to the JDWP transport of a chosen process.

```
$ adb jdwp
12167
$ adb forward tcp:7777 jdwp:12167
```

We're now ready to attach JDB. Attaching the debugger however causes the app to resume, which is something we don't want. Rather, we'd like to keep it suspended so we can do some exploration first. To prevent the process from resuming, we pipe the `suspend` command into jdb:

```
$ { echo "suspend"; cat; } | jdb -attach localhost:7777

Initializing jdb ...
> All threads suspended.
>
```

We are now attached to the suspended process and ready to go ahead with jdb commands. Entering `?` prints the complete list of. Unfortunately, the Android VM doesn't support all available JDWP features. For example, the `redefine` command, which would let us redefine the code for a class - a potentially very useful feature - 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 commands that work 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 of UnCrackable App Level 1 and think about possible solutions. A good approach would be to suspend the app at a state where the secret string is stored in a variable in plain text so we can retrieve it. Unfortunately, we won't get that far unless we deal with the root / tampering detection first.

By reviewing the code, we can gather that the method

`sg.vantagepoint.uncrackable1.MainActivity.a` is responsible for displaying the "This is unacceptable..." message box. This method hooks the "OK" button to a class that implements the `OnClickListener` interface. The `onClick` event handler on the "OK" button is what actually terminates the app. 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$Builder((Context)this).create();
    create.setTitle((CharSequence)title);
    create.setMessage((CharSequence)"This is unacceptable. The app is now going to
exit.");
    create.setButton(-3, (CharSequence)"OK", (DialogInterface$OnClickListener)new
b(this));
    create.setCancelable(false);
    create.show();
}

```

We can bypass this with a little runtime tampering. With the app still suspended, set a method breakpoint on `android.app.Dialog.setCancelable` and resume the app.

```

> stop in android.app.Dialog.setCancelable
Set breakpoint android.app.Dialog.setCancelable
> resume
All threads resumed.
>
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1]

```

The app is now suspended at the first instruction of the `setCancelable` method. You can print the arguments passed to `setCancelable` using the `locals` command (note that the arguments are incorrectly shown under "local variables").

```

main[1] locals
Method arguments:
Local variables:
flag = true

```

In this case, `setCancelable(true)` was called, so this can't be the call we're looking for. Resume the process using the `resume` command.

```

main[1] resume
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1] locals
flag = false

```

We've now hit 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 hit, until the alert box is finally displayed (the breakpoint will hit five or six times). The alert box should now be cancelable! Tap anywhere next to the box and it will close without terminating the app.

Now that the anti-tampering is out of the way we're ready to extract the secret string! In the "static analysis" section, we saw that the string is decrypted using AES, and then compared with the string entered into the messagebox. The method `equals` of the `java.lang.String` class is used to compare the input string with the secret. Set a method breakpoint on `java.lang.String.equals`, enter any text into the edit field, and tap the "verify" button. Once the breakpoint hits, you can read the method argument with the using 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 we are looking for!

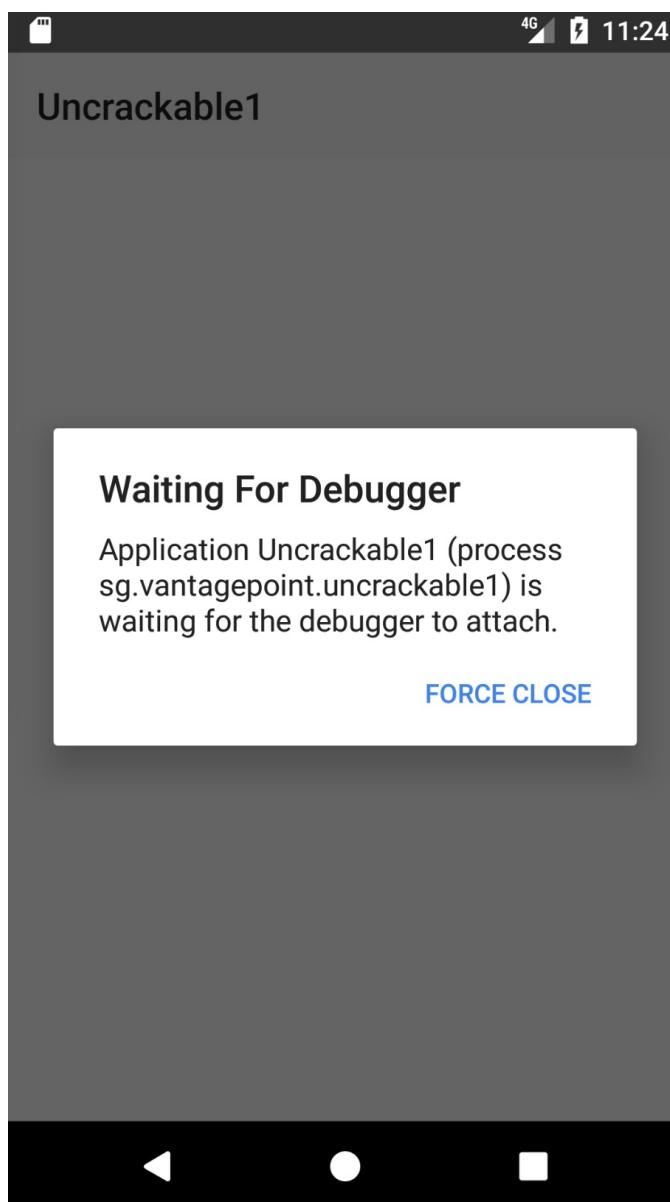
Debugging Using an IDE

A pretty neat trick is setting up a project in an IDE with the decompiled sources, which 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 through the GUI. The experience won't be perfect - it's not the original source code after all, so you can't set line breakpoints and sometimes things will simply not work correctly. Then again, reversing code

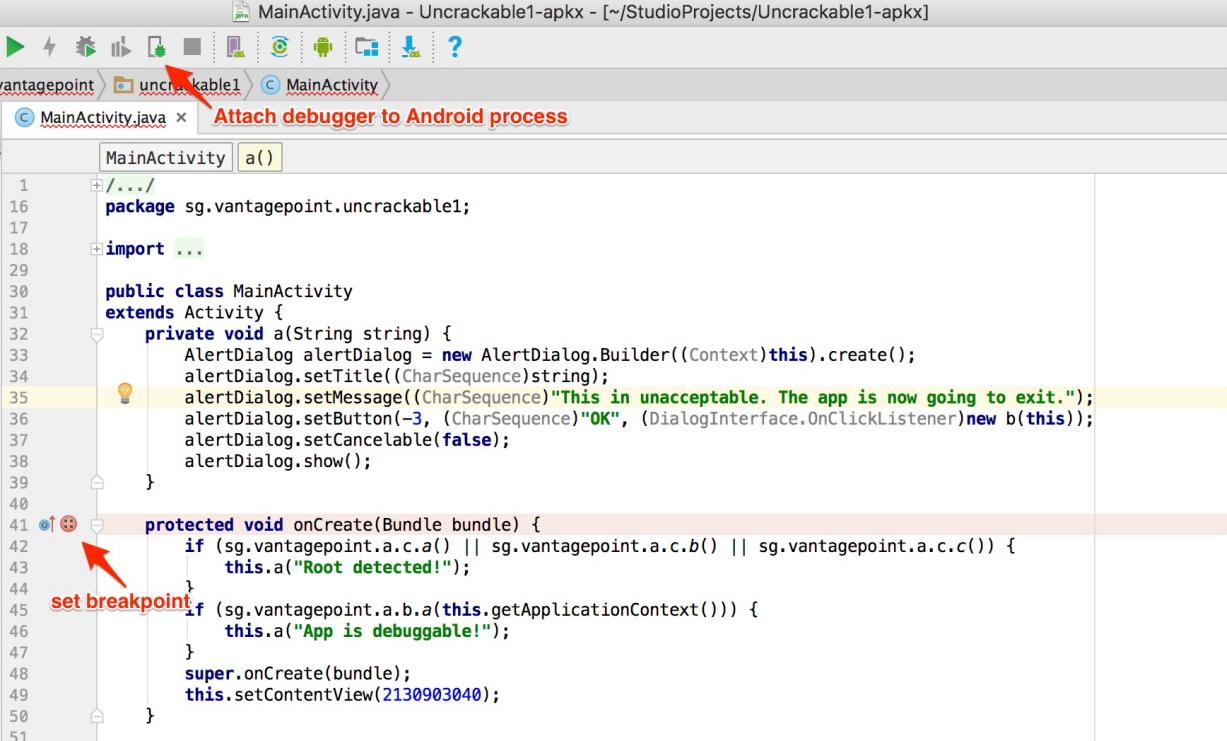
is never easy, and being able to efficiently navigate and debug plain old Java code is a pretty convenient way of doing it, so it's usually worth giving it a shot. A similar method has been described in the [NetSPI blog](#).

In order to debug an app from the decompiled source code, you should first create your Android project and copy the decompiled java sources into the source folder as described above at "Statically Analyzing Java Code" part. Set the debug app (in this tutorial it is Uncrackable1) and make sure you turned on "Wait For Debugger" switch from "Developer Options".

Once you tap the Uncrackable app icon from the launcher, it will get suspended in "wait for a debugger" mode.



Now you can set breakpoints and attach to the Uncrackable1 app process using the "Attach Debugger" button on the toolbar.



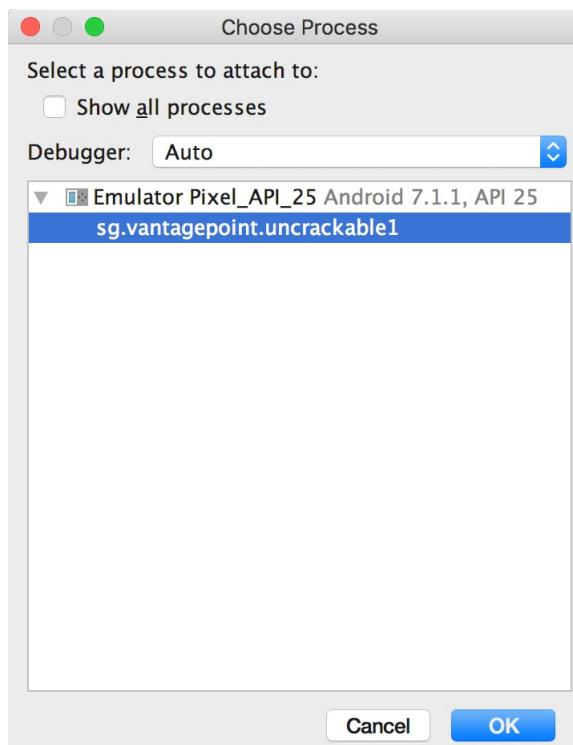
The screenshot shows the Android Studio interface with the code editor open to `MainActivity.java`. A red arrow points to the 'Breakpoint' icon in the toolbar at the top. Another red arrow points to the word 'set breakpoint' in the gutter of the `onCreate()` method. The code contains logic to detect root access and debuggability.

```

1  /...
16 package sg.vantagepoint.uncrackable1;
17
18 import ...
29
30 public class MainActivity
31 extends Activity {
32     private void a(String string) {
33         AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
34         alertDialog.setTitle((CharSequence)string);
35         alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
36         alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
37         alertDialog.setCancelable(false);
38         alertDialog.show();
39     }
40
41     protected void onCreate(Bundle bundle) {
42         if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
43             this.a("Root detected!");
44         }
45         if (sg.vantagepoint.a.b.a(this.getApplicationContext())) {
46             this.a("App is debuggable!");
47         }
48         super.onCreate(bundle);
49         this.setContentView(2130903040);
50
51

```

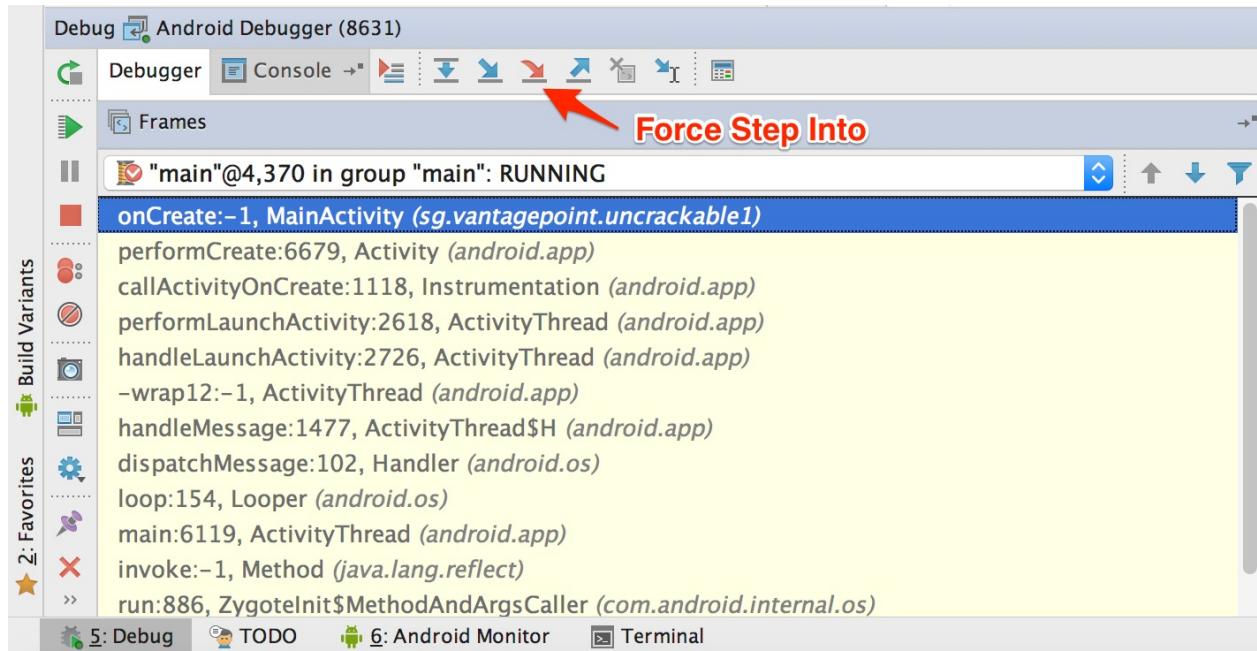
Note that only method breakpoints work when debugging an app from decompiled sources. Once a method breakpoint is hit, you will get the chance to single step throughout the method execution.



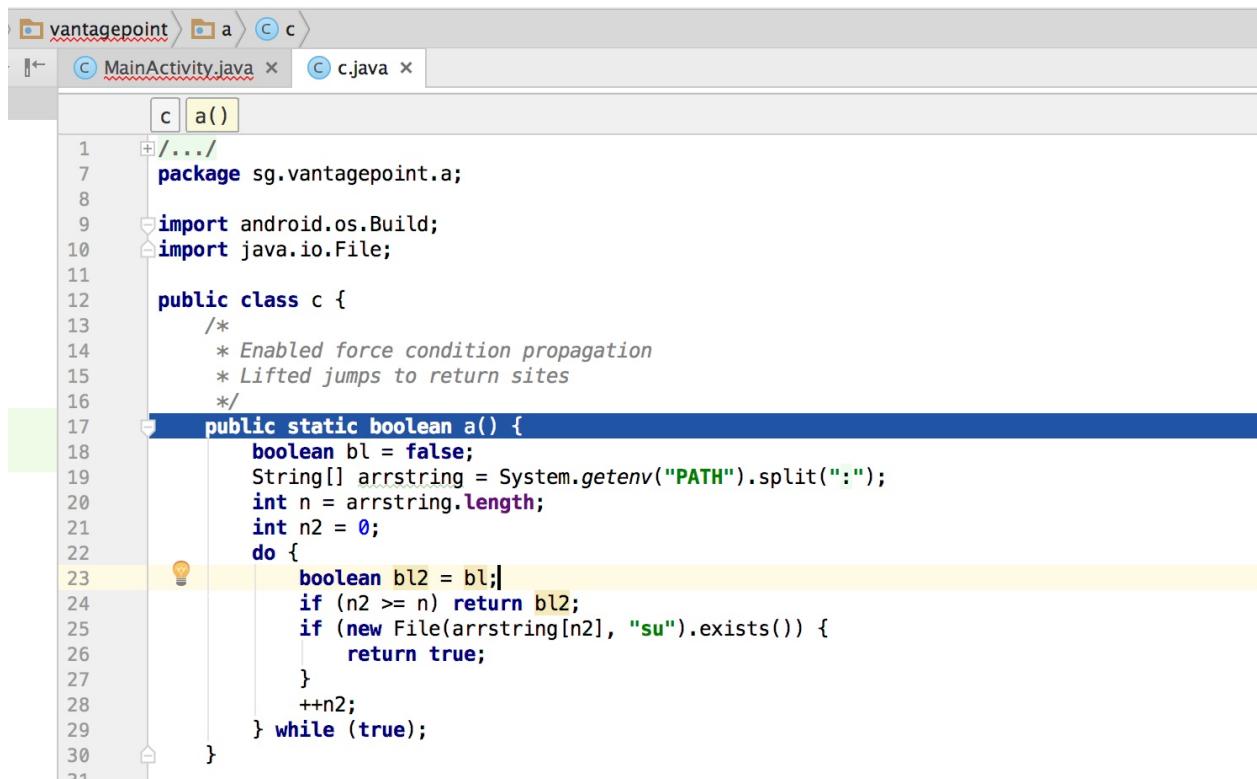
After you choose the Uncrackable1 application from the list, the debugger will attach to the app process and you will hit the breakpoint that was set on the `onCreate()` method. Uncrackable1 app triggers anti-debugging and anti-tampering controls within the

`onCreate()` method. That's why it is a good idea to set a breakpoint on the `onCreate()` method just before the anti-tampering and anti-debugging checks performed.

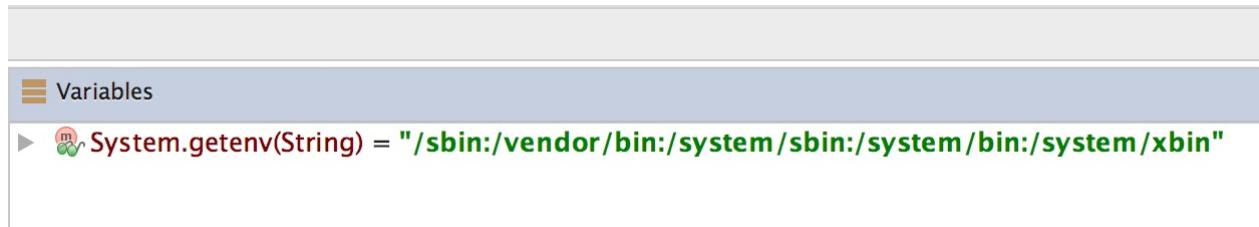
Next, we will single-step through the `onCreate()` method by clicking the "Force Step Into" button on the Debugger view. The "Force Step Into" option allows you to debug the Android framework functions and core Java classes that are normally ignored by debuggers.



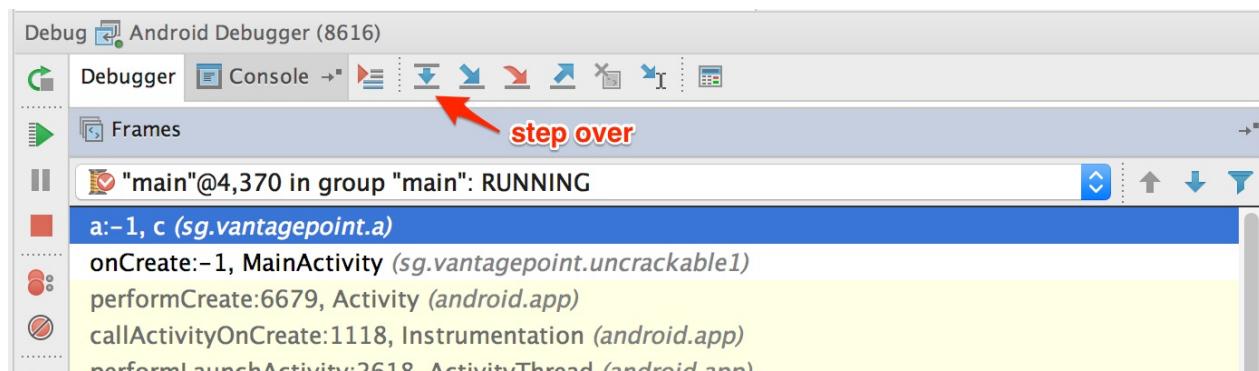
Once you "Force Step Into", the debugger will stop at the beginning of the next method which is the `a()` method of class `sg.vantagepoint.a.c`.



This method searches for "su" binary within well-known directories. Since we are running the app on a rooted device/emulator we need to defeat this check by manipulating variables and/or function return values.



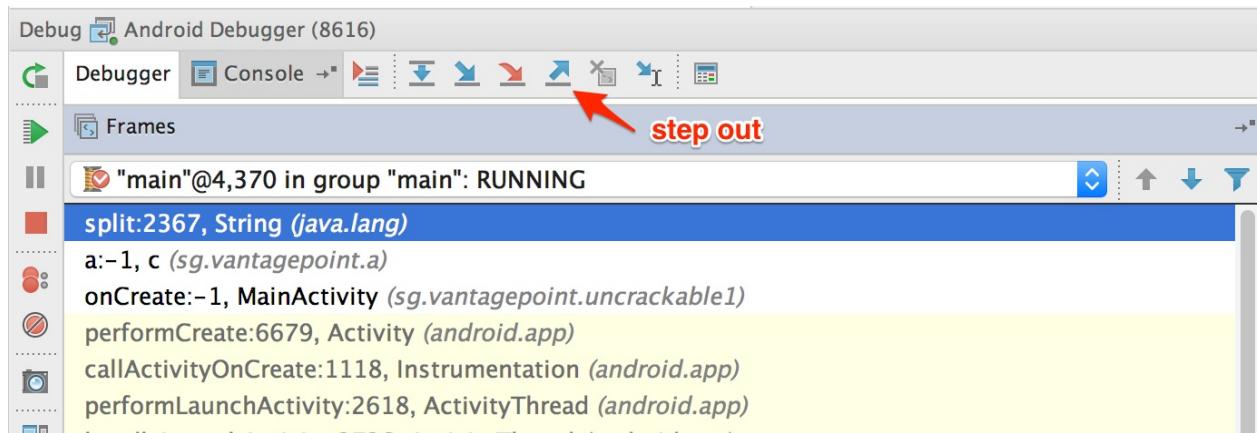
You can see the directory names inside the "Variables" window by stepping into the `a()` method and stepping through the method by clicking "Step Over" button in the Debugger view.



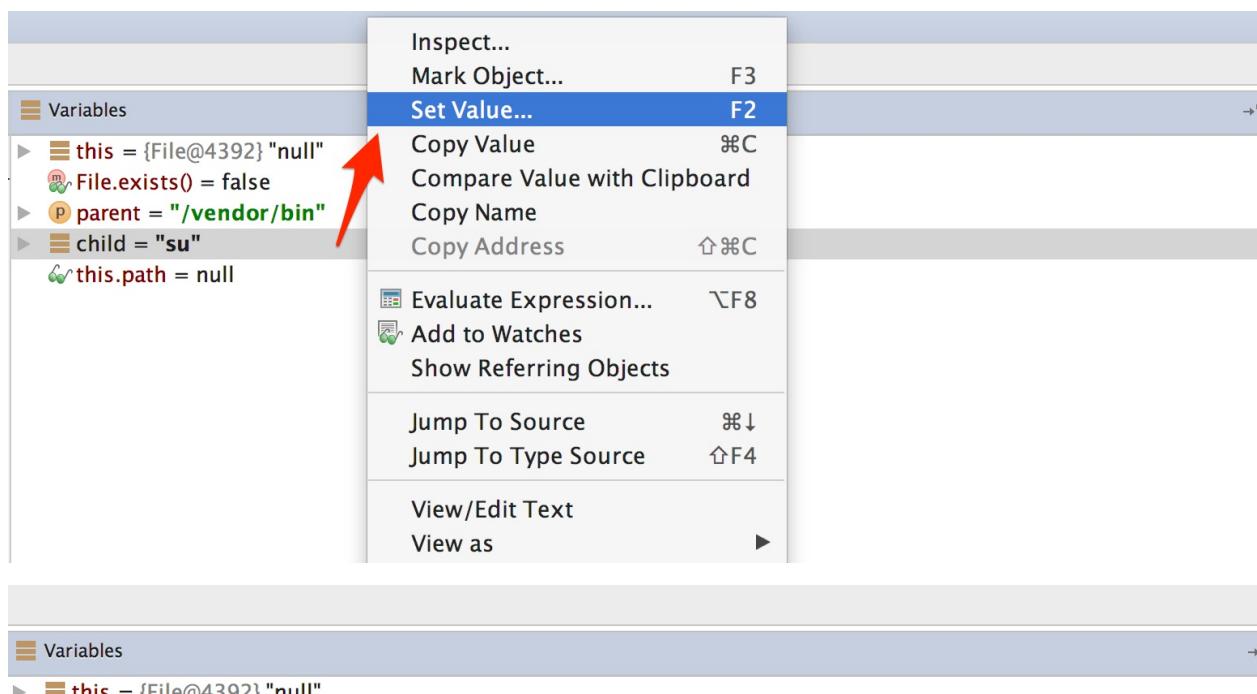
Step into the `System.getenv` method call y using the "Force Step Into" functionality.

After you get the colon separated directory names, the debugger cursor will return to the beginning of `a()` method; not to the next executable line. This is just because we are working on the decompiled code instead of the original source code. So it is crucial for the analyst to follow the code flow while debugging decompiled applications. Otherwise, it might get complicated to identify which line will be executed next.

If you don't want to debug core Java and Android classes, you can step out of the function by clicking "Step Out" button in the Debugger view. It might be a good approach to "Force Step Into" once you reach the decompiled sources and "Step Out" of the core Java and Android classes. This will help you to speed up your debugging while keeping eye on the return values of the core class functions.



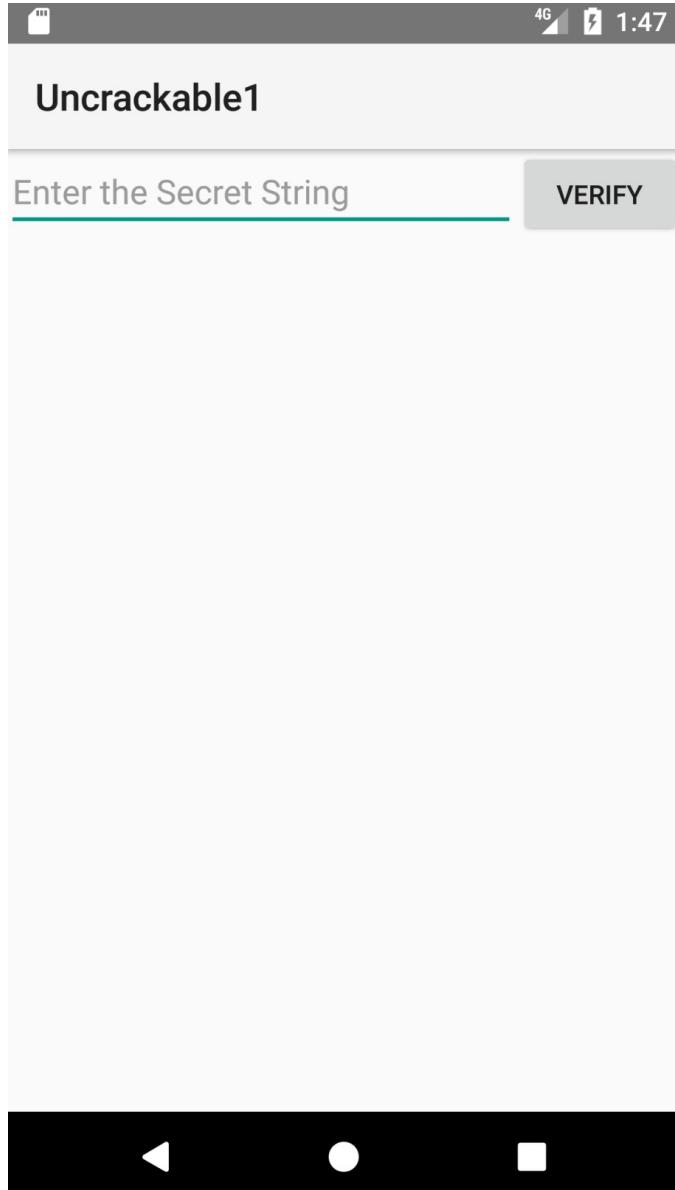
After it gets the directory names, `a()` method will search for the existence of the `</code>su</code>` binary within these directories. In order to defeat this control, you can modify the directory names (parent) or file name (child) at cycle which would otherwise detect the `su` binary on your device. You can modify the variable content by pressing F2 or Right-Click and "Set Value".



Once you modify the binary name or the directory name, `File.exists()` should return `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 to finally reach 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 hit the breakpoint. Then, single-step until you reach the call to `String.equals`. This is where user supplied input is compared with the secret string.

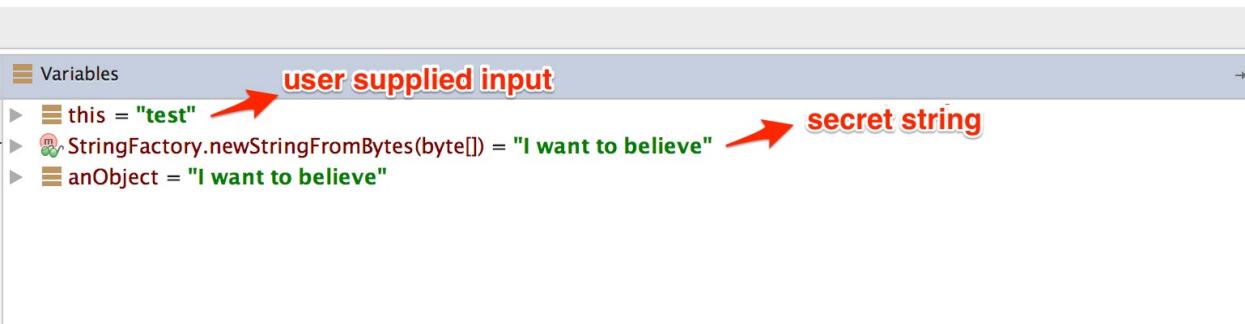


```

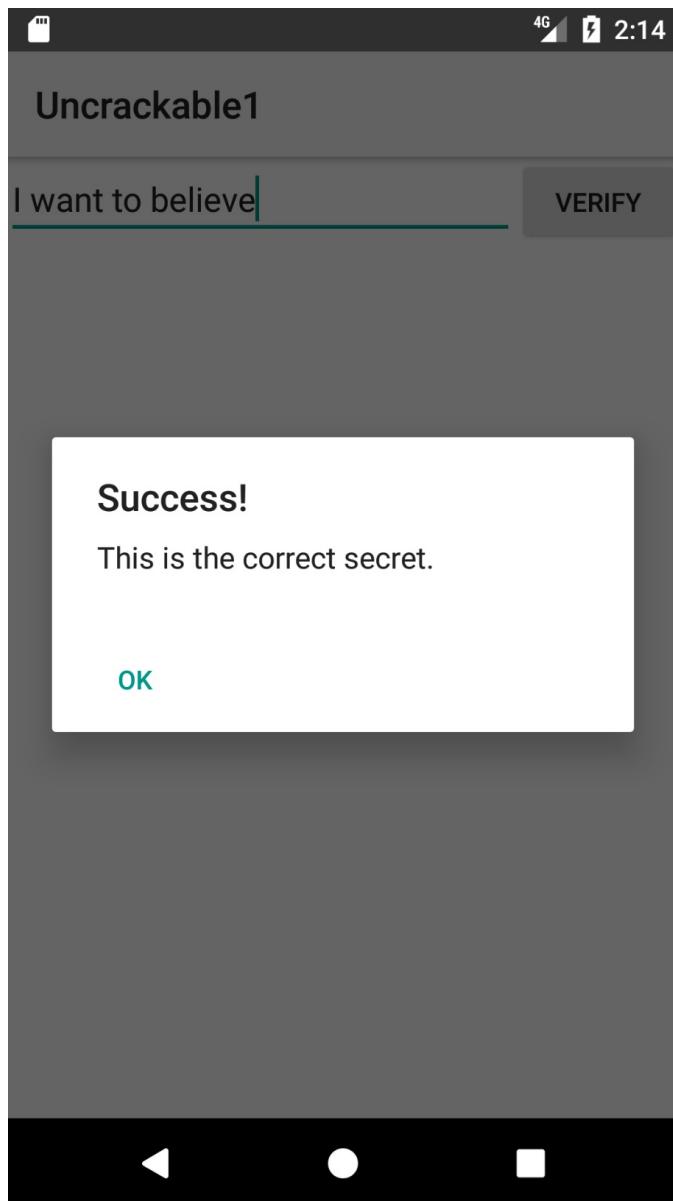
public static boolean a(String string) {
    byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkn08HT4Lv8dLat8FxR2G0c=", (int)0);
    byte[] arrby2 = new byte[]{};
    try {
        arrby = sg.vantagepoint.a.a.a(a.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        arrby2 = arrby;
    }
    catch (Exception exception) {
        Log.d((String)"CodeCheck", (String)(("AES error:" + exception.getMessage())));
    }
    if (string.equals(new String(arrby2))) {
        return true;
    }
    return false;
}

```

You can see the secret string in the "Variables" view at the time you reach the `String.equals` method call.



Variables
▶ <code>this = "test"</code>
▶ <code>mStringFactory.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 them using standard tools, including GDB and the built-in native debuggers of IDEs such as IDA Pro and JEB, as long as they support the processor architecture of the device (most devices are based on ARM chipsets, so this is usually not an issue).

We'll now set up our 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 start 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 /data/local/tmp
```

The `gdbserver --attach<comm> <pid>` command causes `gdbserver` to attach to the running process and bind to the IP address and port specified in `comm`, which in our 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` as follows.

```
$ 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` 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 using the `adb forward` command:

```
$ adb forward tcp:1234 tcp:1234
```

We'll now use the prebuilt version of `gdb` contained 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 ?? ()
```

We have successfully attached to the process! The only problem is that at this point, we're already too late to debug the JNI function `StringFromJNI()` as it only runs once at startup. We can again 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 start of the native function

`Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI()` before resuming the app.

Unfortunately, this isn't possible at this early point in execution because `libnative-lib.so` isn't yet mapped into process memory - it is loaded dynamically during runtime. To get this working, we'll first use JDB to gently control the process into the state we need.

First, we resume execution of the Java VM by attaching JDB. We don't want the process to resume immediately though, so we pipe the `suspend` command into JDB as follows:

```
$ adb jdwp
14342
$ adb forward tcp:7777 jdwp:14342
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
```

Next, we want to suspend the process at the point the Java runtime loads `libnative-lib.so`. In JDB, set a breakpoint on the `java.lang.System.loadLibrary()` method and resume the process. After the breakpoint has been hit, 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 have the effect that the app is "double-suspended" by both the Java VM and the Linux kernel.

```
$ 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 (we're done using JDB, so you can also detach it at this point). 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 hit when the first instruction of the JNI function is executed. You can now display a disassembly of the function using the `disassemble` command.

```
Breakpoint 1, 0xa3522e78 in Java_sg_vantagepoint_helloworldjni_MainActivity_stringFrom
JNI() from libnative-lib.so
(gdb) disass $pc
Dump of assembler code for function Java_sg_vantagepoint_helloworldjni_MainActivity_st
ringFromJNI:
=> 0xa3522e78 <+0>: ldr r2, [r0, #0]
  0xa3522e7a <+2>: ldr r1, [pc, #8] ; (0xa3522e84 <Java_sg_vantagepoint_helloworldjn
i_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 inner workings of 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 also offers basic execution tracing functionality. To trace an app right from the start we can pause the app using the Android "Wait for Debugger" feature or a `kill -STOP` command and attach JDB to set a deferred method breakpoint on an initialization method of our choice. Once the breakpoint hits, we activate method tracing with the `trace go methods` command and resume execution. JDB will dump all method entries and exits from that point on.

```
$ 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) a GUI tool included with Android Studio. At first glance it might not look like much, but make no mistake: Its Java method tracer is one of the most awesome tools you can have in your arsenal, and is indispensable for analyzing obfuscated bytecode.

Using DDMS is a bit confusing however: It can be launched in several ways, and different trace viewers will be launched depending on how the trace was obtained. There's a standalone tool called "Traceview" as well as a built-in viewer in Android Studio, both of which offer different ways of navigating the trace. You'll usually want to use the viewer built into Android studio which gives you a nice, zoom-able hierarchical timeline of all method calls. The standalone tool however is also useful, as it has a profile panel that shows the time spent in each method, as well as 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 are done, click the same button to stop the recording. The integrated trace view will open showing the recorded trace. You can scroll and zoom the timeline view using the mouse or trackpad.

Alternatively, execution traces can also be recorded in the standalone Android Device Monitor. The Device Monitor can be started from 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 the "Start Method Profiling" button. Click the stop button to stop recording, after which the Traceview tool will open showing the recorded trace. An interesting feature of the standalone

tool is the "profile" panel on the bottom, which shows an overview of the time spent in each method, as well as each method's parents and children. Clicking any of the methods in the profile panel highlights the selected method in the timeline panel.

As an aside, DDMS also offers convenient heap dump button that will dump the Java heap of a process to a `.hprof` file. More information on Traceview can be found in the Android Studio user guide.

Tracing System Calls

Moving down a level in the OS hierarchy, we 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 to get a rough idea of what a user process is doing, and is often the most efficient way to deactivate low-level tampering defenses.

Strace is a standard Linux utility that is used to monitor interaction between processes and the kernel. The utility is not included with Android by default, but can be easily built from source using the Android NDK. This gives us a very convenient way of monitoring system calls of a process. Strace however depends on the `ptrace()` system call to attach to the target process, so it only works up to the point that anti-debugging measures kick in.

As a side note, if the Android "stop application at startup" feature is unavailable we can use a shell script to make sure that strace attached immediately once the process is launched (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 be used to trace kernel system calls in a more transparent way than is possible with strace, which relies on the `ptrace` system call to attach to the target process.

Conveniently, ftrace functionality is found in the stock Android kernel on both Lollipop and Marshmallow. It 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 and 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 is used to set or display 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 us with an even more powerful way to instrument the kernel: It allows us to insert probes into (almost) arbitrary code addresses within kernel memory. Kprobes work by inserting a breakpoint instruction at the specified address. Once the breakpoint is hit, control passes to the Kprobes system, which then executes the handler function(s) defined by the user as well as the original instruction. Besides being great for function tracing, KProbes can be used to implement rootkit-like functionality such as file hiding.

Jprobes and Kretprobes are additional probe types based on Kprobes that allow hooking of function entries and exits.

Unfortunately, the stock Android kernel comes without loadable module support, which is a problem given that Kprobes are usually deployed as kernel modules. Another issue is that the Android kernel is compiled with strict memory protection which prevents patching some parts of Kernel memory. Using Elfmaster's system call hooking method results in a Kernel panic on default Lollipop and Marshmallow due to `sys_call_table` being non-writable. We can however use Kprobes on a sandbox by compiling our own, more lenient Kernel (more on this later).

Emulation-based Analysis

Even in its standard form that ships with the Android SDK, the Android emulator – a.k.a. “emulator” – is a somewhat capable reverse engineering tool. It 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 into a code cache, and executed. After execution of the basic block has completed, 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 the full QEMU feature set, including its monitoring, debugging and tracing facilities. QEMU-specific parameters can be passed to the emulator with the `-qemu` command line flag. We can use QEMU's built-in tracing facilities to log executed instructions and virtual register values. Simply 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. The `-d in_asm` option 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, it is not possible to generate a complete guest instruction trace with QEMU, 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 (save for hacking the source code). Even so, 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 to provide more complete tracing functionality. PANDA/PANDROID is your best if you're going for a CPU-trace based analysis, as 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 adds instrumentation on several levels, making it possible to fully reconstruct the semantics on the hardware, Linux and Java level.

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 or 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 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 upon certain QEMU events. The twist PANDA adds is its record/replay feature. This allows for an iterative workflow: The reverse engineer records an execution trace of some the target app (or some part of it) and then replays it over and over again, refining his analysis plugins with each iteration.

PANDA comes with some pre-made plugins, such as a stringsearch tool and a syscall tracer. Most importantly, it also supports Android guests and some of the DroidScope code has even been ported over. Building and running PANDA for Android (“PANDROID”) is relatively straightforward. To test it, clone Moiyx’s git repository and build PANDA as follows:

```
$ 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 specific Dalvik runtime of Android 2.3. Anyways, older versions of Android seem to run much faster in the emulator, so if you plan on using PANDA sticking with Gingerbread is probably best. 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 using 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 of modifying and instrumenting mobile apps. *Tampering* means making patches or runtime changes to the app to affect its behavior - usually in a way that’s to our advantage. For example, it could be desirable to deactivate SSL pinning or deactivate 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 is used rather loosely to refer to all kinds runtime 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 pop up 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 cannot intercept HTTPS traffic with a proxy because the app employs SSL pinning.

In most cases, both issues can be fixed by making minor changes and re-packaging and resigning the app (the exception are apps that run additional integrity checks beyond default Android code signing - in these cases, you also have to patch out those additional checks as well).

Example: Disabling SSL Pinning

Certificate pinning is an issue for security testers who want to intercept HTTPS communication for legitimate reasons. To help with this problem, the bytecode can be patched to deactivate SSL pinning. To demonstrate how Certificate Pinning can be bypassed, we will walk through the necessary steps to bypass Certificate Pinning implemented in an example application.

The first step is to disassemble the APK with `apktool` :

```
$ apktool d target_apk.apk
```

You then need to locate the certificate pinning checks in the Smali source code. Searching the smali 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 methods named `checkClientTrusted`, `checkServerTrusted` and `getAcceptedIssuers`.

Insert the `return-void` opcode was added to the first line of each of these methods to bypass execution. This causes each method to return immediately. With this modification, no certificate checks are performed, and the application will accept 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
    ige-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. By running Java code in the context of the newly instantiated app, it is 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 runtime of the process - no patches to the application files are made.

To use Xposed, you first need to install the Xposed framework on a rooted device. Modifications are then deployed in the form of separate apps ("modules") that 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 is 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 the `su` binary is found in any of them. Checks like this are easy to deactivate - all you have to do is to replace the code with something that returns "false". Method hooking using an Xposed module is one way to do this.

This method `XposedHelpers.findAndHookMethod` allows you to override existing class methods. From the decompiled code, we know that the method performing the check is called `c()` and located in the class `com.example.a.b`. An Xposed module that overrides the function to always return "false" looks as follows.

```
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);
            }
        });
    }
}
```

Modules for Xposed are developed and deployed with Android Studio just like regular Android apps. For more details on writing compiling and installing Xposed modules, refer to the tutorial provided by its author, [rovo89](#).

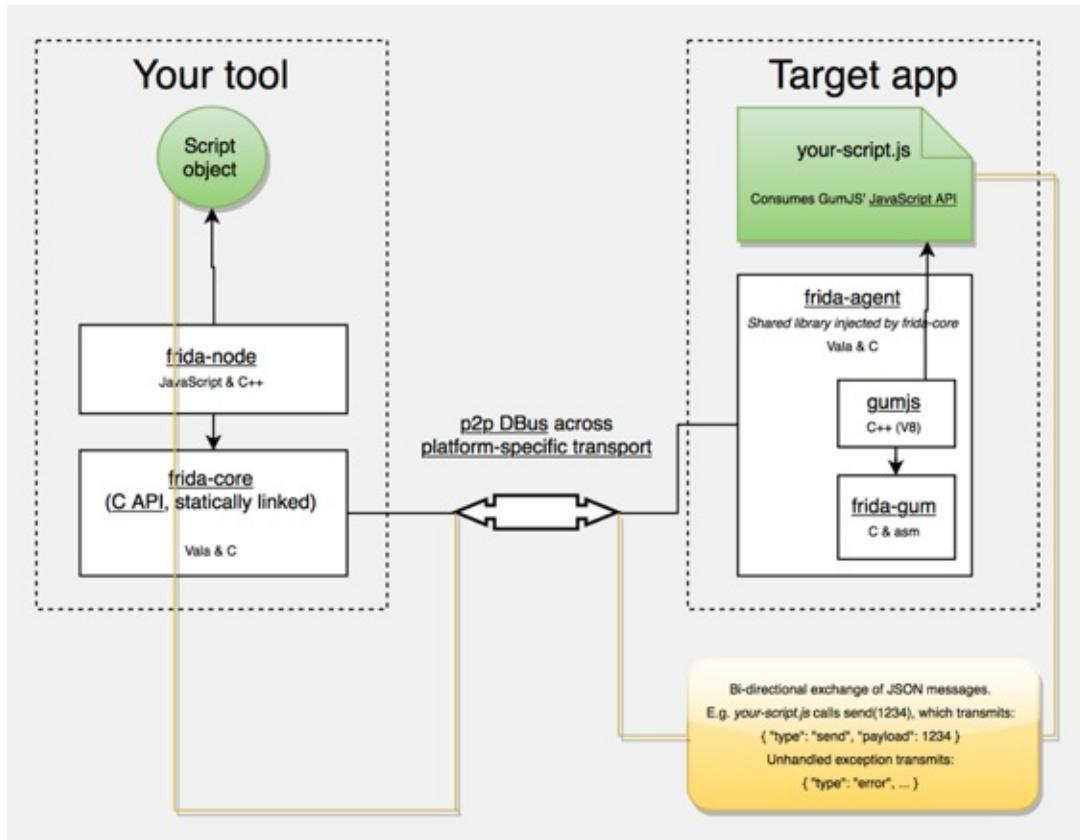
Dynamic Instrumentation with Frida

Frida "lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, Linux, iOS, Android, and QNX". While it was originally based on Google's V8 Javascript runtime, since version 9 Frida now uses Duktape internally.

Code injection can be achieved in different ways. For example, Xposed makes some permanent modifications to the Android app loader that provide hooks to run your own code every time a new process is started. In contrast, Frida achieves code injection by writing code directly into process memory. The process is outlined in a bit more detail below.

When you "attach" Frida to a running app, it uses ptrace to hijack a thread in 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 running on the device, and loads a dynamically generated library file containing the Frida agent and instrumentation code. The original, hijacked thread is restored to its original state and resumed, and execution of the process continues as usual.

Frida injects a complete JavaScript runtime into the process, along with a powerful API that provides a wealth of useful functionality, including calling and hooking of native functions and injecting structured data into memory. It also supports interaction with the Android Java runtime, such as interacting with objects inside the VM.



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.

Some features unfortunately don't work yet on current Android devices platforms. Most notably, the FRIDA Stalker - a code tracing engine based on dynamic recompilation - does not support ARM at the time of this writing (version 7.2.0). Also, support for ART has been

included only recently, so the Dalvik runtime is still better supported.

Installing Frida

To install Frida locally, simply use Pypi:

```
$ sudo pip install frida
```

Your Android device doesn't need to be rooted to get Frida running, but it's the easiest setup and we assume a rooted device here unless noted otherwise. Download the frida-server binary from the [Frida releases page](#). Make sure that the server version (at least the major version number) matches the version of your local Frida installation. Usually, Pypi will install the latest version of Frida, but if you are not sure, 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-and
roid-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` that Frida injects into the process and that traces all calls to the `open` function in `libc.so`. You can modify the generated script according to your needs, making use of Frida's [Javascript API](#).

To work with Frida interactively, you can use `frida CLI` which hooks into a process and gives you a command line interface to Frida's API.

```
frida -U com.android.chrome
```

You can also use frida CLI to load scripts via the `-l` option, 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. This 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 script above calls `Java.perform` to make sure that our 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 outputs 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 instantiated objects on the heap and work with them. 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.E..... ID 0,0-0,0 #7f0c01fc app
:id/action_bar_black_background}
[*] Instance found: android.view.View{2809551 V.E..... 0,1731-0,1731 #7f0c01
ff app:id/menu_anchor_stub}
[*] Instance found: android.view.View{be471b6 G.E..... I. 0,0-0,0 #7f0c01f5 app
:id/location_bar_verbose_status_separator}
[*] Instance found: android.view.View{3ae0eb7 V.E..... 0,0-1080,63 #102002f
android:id/statusBarBackground}
[*] Finished heap search

```

Notice that you can also make use of 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 its `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());
    }
});

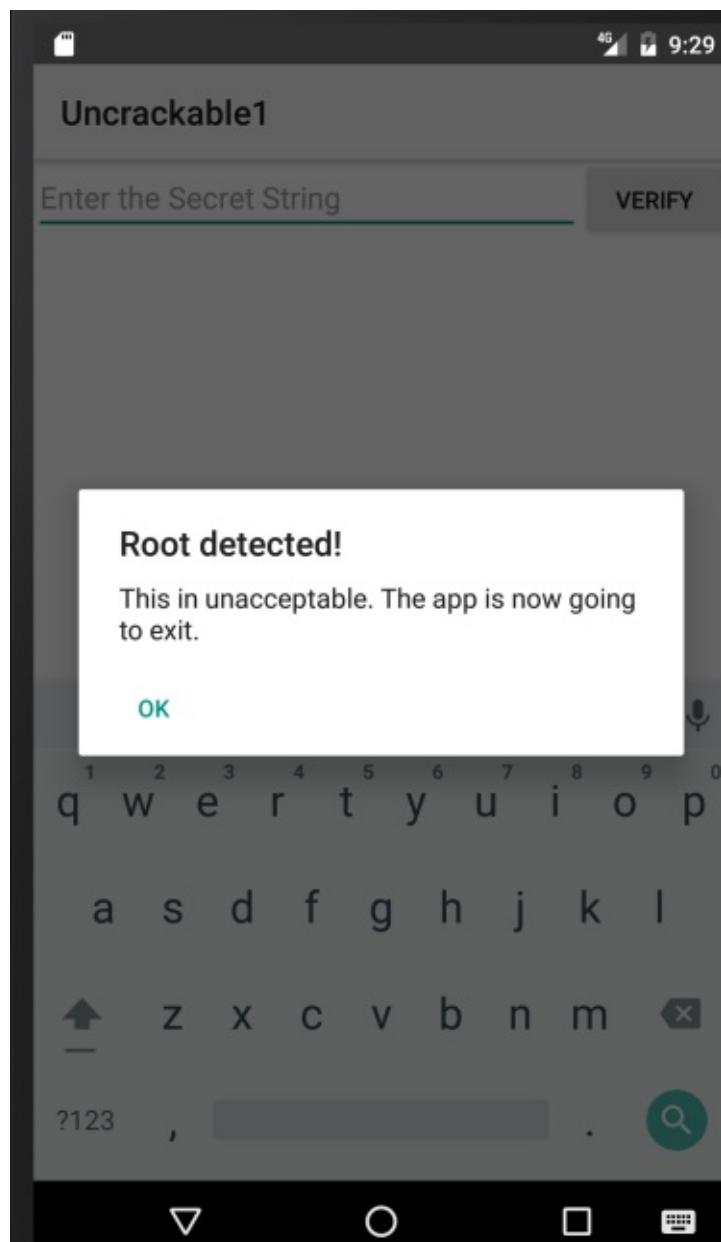
```

Besides loading scripts via `frida CLI`, Frida also provides Python, C, NodeJS, Swift and various other bindings.

Solving the OWASP Uncrackable Crackme Level1 with Frida

Frida gives you the possibility to solve the OWASP UnCrackable Crackme Level 1 easily. We have already seen that we can hook method calls with Frida above.

When you start the App on an emulator or a rooted device, you find that the app presents a dialog box and exits as soon as you press "Ok" because it detected root:



Let us see how we can prevent this. The decompiled main method (using CFR decompiler) 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 `if`-statement before 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 is where the dialog box gets displayed. There is a `AlertDialog.OnClickListener` callback set in the `setButton` method call which is responsible for closing the application via `System.exit(0)` after successful root detection. Using Frida, we 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 we intercept it using 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")

    })
})
```

We wrap our code in a `setImmediate` function to prevent timeouts (you may or may not need this), then call `Java.perform` to make use of Frida's methods for dealing with Java.

Afterwards we retrieve a wrapper for the class that implements the `OnClickListener` interface and overwrite its `onClick` method. Unlike the original, our new version of `onClick` just writes some console output and *does not exit the app*. If we inject our version of this method via Frida, the app should not exit anymore when we click the `ok` button of the dialog.

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 the OK button in the app. The app does not exit anymore.

We can now try to input a "secret string". But where do we get it?

Looking at the class `sg.vantagepoint.uncrackable1.a` you can see the encrypted string to which our 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)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2
G0c=", (int)0);
        byte[] arrby2 = new byte[] {};
        try {
            arrby2 = arrby = sg.vantagepoint.a.a.a((byte[])a.b((String)"8d127684cbc37c
17616d806cf50473cc"), (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`. The `string.equals` comparison compares our input to `arrby2`. So what we are after is the return value of `sg.vantagepoint.a.a.a`.

Instead of reversing the decryption routines to reconstruct the secret key, we 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 the 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 this:

```
michael@sixtyseven:~/Development/frida$ frida -U -l uncrackable1.js sg.vantagepoint.un
crackable1

    / _ |  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 modif
ied
[*] sg.vantagepoint.a.a.a modified
[*] onClick called.
[*] Decrypted: I want to believe
```

The hooked function outputted our decrypted string. Without having to dive too deep into the application code and its decryption routines, we were able to extract the secret string successfully.

We've now covered the basics of static/dynamic analysis on Android. Of course, the only way to *really* learn it is hands-on experience: Start by building your own projects in Android Studio and observing how your code gets translated to bytecode and native code, and have a shot at our cracking challenges.

In the remaining sections, we'll introduce a few advanced subjects including kernel modules and dynamic execution.

Binary Analysis Frameworks

Binary analysis frameworks provide you powerful ways of automating tasks that would be almost impossible to complete manually. In the section, we'll have a look at the Angr framework, a python framework for analyzing binaries that 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 a license key validator like this in the wild, but it should be useful enough to 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, precisely to make de-obfuscation more difficult).

Installing Angr

Angr is written in Python 2 and available from PyPI. It is easy to install on *nix operating systems and Mac OS using pip:

```
$ pip install angr
```

It is recommended to create a dedicated virtual environment with Virtualenv as some of its dependencies contain forked versions Z3 and PyVEX that overwrite the original versions (you may skip this step if you don't use these libraries for anything else - on the other hand, using Virtualenv is generally a good idea).

Quite comprehensive documentation for angr is available on Gitbooks, including an installation guide, tutorials and usage examples [5]. 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 does this by translating the program's semantics into a logical formula, whereby some variables are represented as symbols with specific constraints. By resolving the constraints, you can find out the conditions necessary so that some branch of the program gets executed.

Amongst other things, this is useful in cases where we need to find the right inputs for reaching a certain block of code. In the following example, we'll use Angr to solve a simple Android crackme in an automated fashion. The crackme takes the form of a native ELF binary that can be downloaded 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 we really know nothing about how a valid license key might look like. Where do we start? Let's fire up IDA Pro to get a first good look at what is happening.

```

.text:00001874 sub_1874          ; DATA XREF: start+4C↑o
.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:00001874     ADD     R11,SP,#4
.text:00001874     SUB    SP,SP,#0x28
.text:00001874     STRK   R0,[R11, #var_20]
.text:00001874     STRK   R1,[R11, #var_24]
.text:00001874     LDR    R3,[R11, #var_20]
.text:00001874     CMP    R3, #2
.text:00001874     BEQ    loc_1898
.text:00001874     BL     sub_16AB
.text:00001898
.text:00001898 loc_1898          ; CODE XREF: sub_1874+1C↑j
.text:00001898     LDR    R3,[R11, #var_24]
.text:00001898     ADD    R3,R3,#4
.text:00001898     LDR    R3,[R3]
.text:00001898     MOV    R0,R3      ; char *
.text:00001898     BL     strlen
.text:00001898     MOV    R3,R0
.text:00001898     CMP    R3, #0x10
.text:00001898     BEQ    loc_18BC
.text:00001898     BL     sub_16CC
.text:00001898
.text:00001898 loc_18BC          ; CODE XREF: sub_1874+40↑j
.text:00001898     LDR    R3, =(aEnteringBase32 - 0x18C8)
.text:00001898     ADD    R3,PC,R3      ; "Entering base32_decode"
.text:00001898     MOV    R0,R3      ; char *
.text:00001898     BL     puts
.text:00001898     LDR    R3,[R11, #var_24]
.text:00001898     ADD    R3,R3,#4
.text:00001898     LDR    R2,[R3]
.text:00001898     SUB    R2,R11, #-var_14
.text:00001898     SUB    R1,R11, #-var_18
.text:00001898     STR   R1,[SP, #0x2C+var_2C]
.text:00001898     MOV    R0,#0
.text:00001898     MOV    R1,R2
.text:00001898     MOV    R2, #0x10
.text:00001898     BL     sub_1340
.text:00001898     LDR    R3,[R11, #var_18]
.text:00001898     LDR    R2, =(aOutlenD - 0x1904)
.text:00001898     ADD    R2,PC,R2      ; "Outlen = %d\n"
.text:00001898     MOV    R0,R2      ; char *
.text:00001898     MOV    R1,R3
.text:00001898     BL     printf
.text:00001898     LDR    R3, =(aEnteringCheck - 0x1918)
.text:00001898     ADD    R3,PC,R3      ; "Entering check_license"
.text:00001898     MOV    R0,R3      ; char *
.text:00001898     BL     puts
.text:00001898     SUB    R3,R11, #-var_14
.text:00001898     MOV    R0,R3
.text:00001898     BL     sub_1760

```

The assembly code shows the main function starting at address 0x1874. It performs a length check at loc_1898 (length exactly 16), then decodes a 16-character base32 string into 10 bytes at loc_18BC using sub_1340. Finally, it calls sub_1760 to validate the license key. The code is annotated with arrows indicating the flow from the length check to the base32 decode, and from the base32 decode to the main license check.

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 luckily we can see some references to debugging strings: It appears that the input string is base32-decoded (call to sub_1340). At the beginning of main, there's also a length check at loc_1898 that verifies that the length of the input string is exactly 16. So we're looking for a 16 character base32-encoded string! The decoded input is then passed to the function sub_1760, which verifies the validity of the license key.

The 16-character base32 input string decodes to 10 bytes, so we know that the validation function expects a 10 byte binary string. Next, we have a 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. C
ongratulati"...
.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

```

We can see a loop with some XOR-magic happening at loc_1784, which supposedly decodes the input string. Starting from loc_17DC, we see a series of comparisons of the decoded values with values obtained from further sub-function calls. Even though this doesn't look like highly sophisticated stuff, we'd still need to do some more analysis to completely reverse this check and generate a license key that passes it. But now comes the twist: By using dynamic symbolic execution, we can construct a valid key automatically! The symbolic execution engine can map a path between the first instruction of the license check (0x1760) and the code printing the "Product activation passed" message (0x1840) and 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.

We need to provide several inputs to the symbolic execution engine:

- The address to start execution from. We initialize the state with the first instruction of the serial validation function. This makes the task significantly easier (and in this case, almost instant) to solve, as we avoid symbolically executing the Base32 implementation.
- The address of the code block we want execution to reach. In this case, we want to find a path to the code responsible for printing the "Product activation passed" message. This block starts at 0x1840.
- Addresses we don't want to reach. In this case, we're not interesting in any path that arrives at the block of code printing the "Incorrect serial" message, at 0x1854.

Note that Angr loader will load the PIE executable with a base address of 0x400000, so we have to add this to the addresses above. The solution looks as follows.

```

#!/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 obtained - it appears if we were simply reading the solution from memory. We are however reading from symbolic memory - neither the string nor the pointer to it actually exist! What's really happening is that the solver is computing possible concrete values that could be found at that program state, would we observe the actual program run 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.
JQAE6ACMABNAIIIA
```

Customizing Android for Reverse Engineering

Working on real device has advantages especially for interactive, debugger-supported static / dynamic analysis. For one, it is simply faster to work on a real device. Also, being run on a real device gives the target app less reason to be suspicious and misbehave. By instrumenting the live environment at strategic points, we can obtain useful tracing functionality and manipulate the environment to help us bypass any anti-tampering defenses the app might implement.

Customizing the RAMDisk

The initramfs is a small CPIO archive stored inside the boot image. It contains a few files that are required at boot time before the actual root file system is mounted. On Android, the initramfs stays mounted indefinitely, and it contains an important configuration file named `default.prop` that defines some basic system properties. By making some changes to this file, we can make the Android environment a bit more reverse-engineering-friendly. 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 causes all apps running on the system to be debuggable (i.e., the debugger thread runs in every process), independent of the `android:debuggable` attribute in the app's Manifest. Setting `ro.secure` to 0 causes `adbd` to be run as root. To modify `initrd` on any Android device, back up the original boot image using TWRP, or simply dump it with a command like:

```
$ adb shell cat /dev/mtd/mtd0 >/mnt/sdcard/boot.img
$ adb pull /mnt/sdcard/boot.img /tmp/boot.img
```

Use the `abootimg` tool as described in Krzysztof Adamski's how-to to extract the contents of the boot image:

```
$ mkdir boot
$ cd boot
$ ./abootimg -x /tmp/boot.img
$ mkdir initrd
$ cd initrd
$ cat ../initrd.img | gunzip | cpio -vid
```

Take note of the boot parameters written to `bootimg.cfg` – you will need to these parameters later 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 > ../myinitrd.img
```

Customizing the Android Kernel

The Android kernel is a powerful ally to the reverse engineer. While 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 a really unfair

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 of interacting with the OS environment. The standard way is through the APIs of the Android Application Framework. On the lowest level however, many important functions, such as allocating memory and accessing files, are translated into perfectly old-school Linux system calls. In 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 of function pointers (a.k.a. `sys_call_table` on Android).

The most straightforward way of intercepting system calls is injecting your own code into kernel memory, then overwriting the original function in the system call table to redirect execution. Unfortunately, current stock Android kernels enforce memory restrictions that prevent this from working. Specifically, stock Lollipop and Marshmallow kernel are built with the `CONFIG_STRICT_MEMORY_RXW` option enabled. This prevents writing to kernel memory regions marked as read-only, which means that any attempts to patch kernel code or the system call table result in a segmentation fault and reboot. A way to get around this is to build your own kernel: You can then deactivate this protection, and make many other useful customizations to make reverse engineering easier. If you're reversing Android apps on a regular basis, building your own reverse engineering sandbox is a no-brainer.

For hacking purposes, I recommend using an AOSP-supported device. Google's Nexus smartphones and tablets are the most logical candidates – kernels and system components built from the AOSP run on them without issues. Alternatively, Sony's Xperia series is also known for its openness. To build the AOSP kernel you need a toolchain (set of programs to cross-compile the sources) as well as 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 the "android-msm-hammerhead" branch (hammerhead is the codename of the Nexus 5, and yes, finding the right branch is a confusing process). Once the sources are downloaded, create the default kernel config with the command `make hammerhead_defconfig` (or `whatever_defconfig`, depending on 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 enable the most important tracing facilities, add loadable module support, and open up 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 are finished editing save the .config file and build the kernel.

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path_to_your_ndk/arm-eabi-4.8/bin/arm-eabi-
$ make
```

Once you are finished editing save the .config file. Optionally, you can now create a standalone toolchain for cross-compiling the kernel and later tasks. To create a toolchain for Android Nougat, run make-standalone-toolchain.sh from the Android NDK package as follows:

```
$ 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 the original boot image from your device. Look up the location of the boot partition as follows:

```
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/mmcblk0p
28
```

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 as well as some 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.

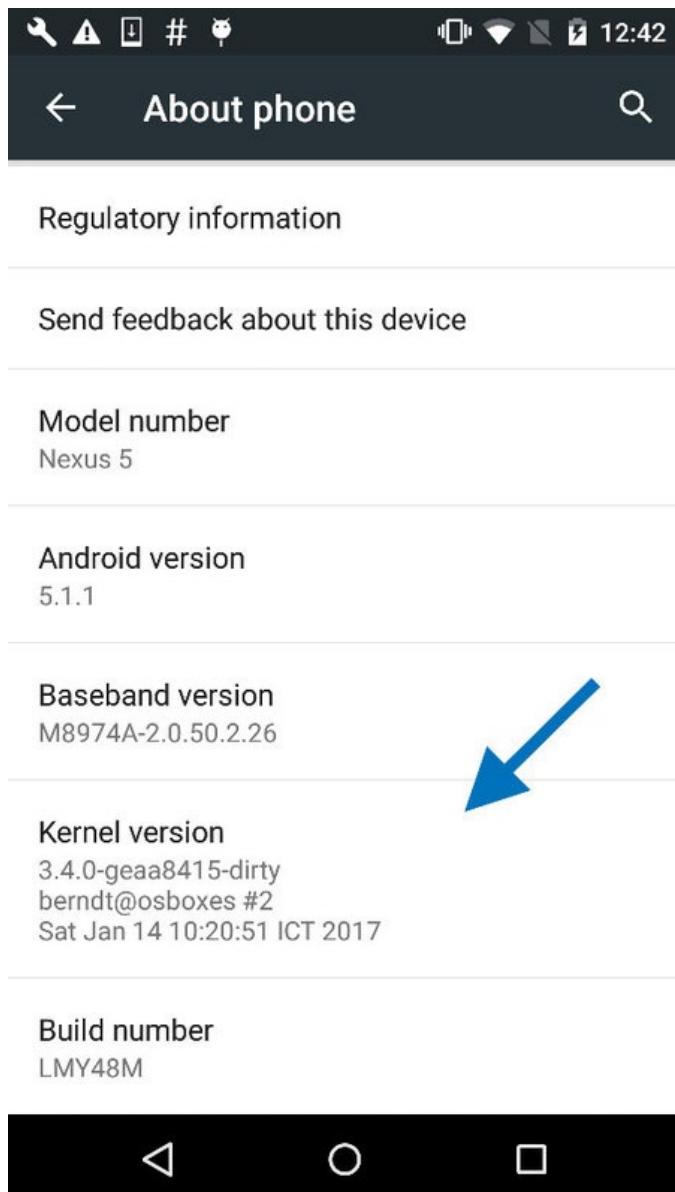
You can now use fastboot to test the new kernel. The "fastboot boot" command allows you to run the kernel without actually flashing it (once you're sure everything works, you can make the changes permanent with fastboot flash - but you don't have to). Restart the device in fastboot mode with the following command:

```
$ adb reboot bootloader
```

Then, use the "fastboot boot" command to boot Android with the new kernel. In addition to the newly built kernel and the original ramdisk, specify the kernel offset, ramdisk offset, tags offset and commandline (use the values listed in your previously extracted bootimg.cfg).

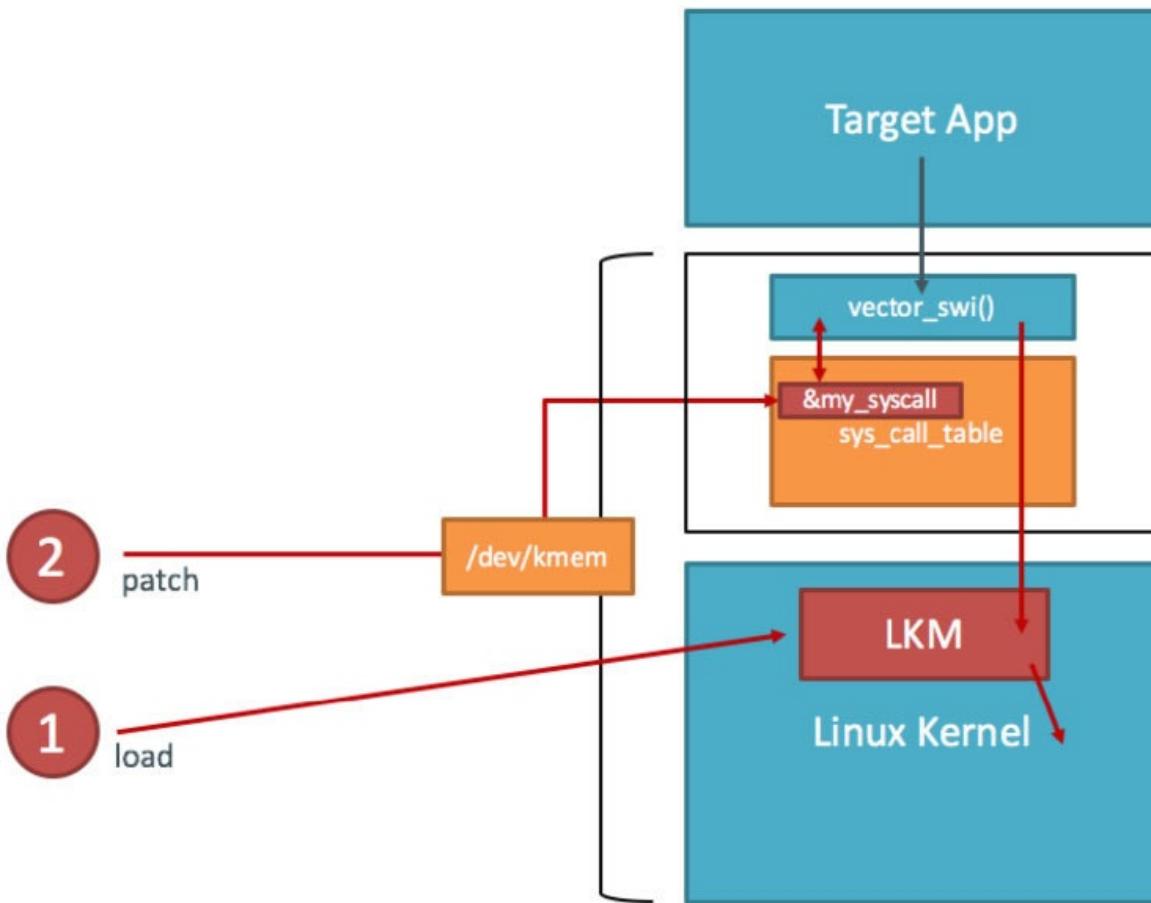
```
$ fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset 0x2900000 --tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

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 Using Kernel Modules

System call hooking allows us to attack any anti-reversing defenses that depend on functionality provided by the kernel. With our custom kernel in place, we can now use a LKM to load additional code into the kernel. We also have access to the /dev/kmem interface, which we can use to patch kernel memory on-the-fly. This is a classical Linux rootkit technique and has been described for Android by Dong-Hoon You [1].



The first piece of information we need is the address of `sys_call_table`. Fortunately, it is exported as a symbol in the Android kernel (iOS reversers are not so lucky). We 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 we need for writing our kernel module - everything else can be calculated using offsets taken from the Kernel headers (hopefully you didn't delete them yet?).

Example: File Hiding

In this howto, we're going to use a Kernel module to hide a file. Let's create a file on the device so we can hide it later:

```
$ adb shell "su -c echo ABCD > /data/local/tmp/nowyouseeme"
$ adb shell cat /data/local/tmp/nowyouseeme
ABCD
```bash
```

Finally it's time to write the kernel module. For file hiding purposes, we'll need to hook one of the system calls used to open (or check `for` the existence of) files. Actually, there many of those - `open`, `openat`, `access`, `accessat`, `faccessat`, `stat`, `fstat`, and more. For now, we'll only hook the `openat` system call - this is the syscall used by the `"/bin/cat"` program when accessing a file, so it should be servicable enough 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>

asm linkage 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 already built a complete kernel before, you are 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 the `kernel_hook.ko` file to the device and load it with the `insmod` command. Verify with the `lsmod` command 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 0xbff00000 (P0)
```

Now, we'll access `/dev/kmem` to overwrite the original function pointer in `sys_call_table` with the address of our newly injected function (this could have been done directly in the kernel module as well, but using `/dev/kmem` gives us an easy way to toggle our 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()`, as I found the latter to cause kernel panics for some reason. 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;
}

```

Build `kmem_util.c` using the prebuilt toolchain and copy it to the device. Note that from Android Lollipop, all executables must be compiled with PIE support:

```

$ /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 we start messing with kernel memory we still need to know the correct offset into the system call table. The openat system call is defined in unistd.h which is found 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 our replacement-openat. Again, we can get this address from /proc/kallsyms.

```
$ adb shell cat /proc/kallsyms | grep new_openat
bf000000 t new_openat [kernel_hook]
```

Now we have everything we 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 to point to our 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 should now be unable 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
```

Voilá! The file "nowyouseeme" is now somewhat hidden from the view of all usermode processes (note that there's a lot more you need to do to properly hide a file, including hooking stat(), access(), and other system calls, as well as hiding the file in directory listings).

File hiding is of course only the tip of the iceberg: You can accomplish a lot of things, including bypassing many root detection measures, integrity checks, and anti-debugging tricks. You can find some additional examples in the "case studies" section in [Bernhard Mueller's Hacking Soft Tokens Paper](#).

# Testing Anti-Reversing Defenses on Android

## 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 the `SafetyNetApi.attest()` method with returns a JWS message with the *Attestation Result*, and then check the following fields:

- `ctsProfileMatch`: Of "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 permissions: [%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.ramdroid.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?

For a more detailed assessment, apply the criteria listed under "Assessing Programmatic Defenses" in the "Assessing Software Protection Schemes" chapter.

## Remediation

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.

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - [https://www.owasp.org/index.php/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."

### CWE

N/A

## 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's 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<10000000; ++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 <dlfcn.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 '_ZTVN3art4JDWP12JdwpAdbStateE'.\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 <string>
#include <unistd.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

static int child_pid;

void *monitor_pid(void *) {
 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);
}
}

extern "C"

JNIEXPORT void JNICALL
Java_sg_vantagepoint_antidebug_MainActivity_antidebug(
 JNIEnv *env,
 jobject /* this */) {

 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?

For a more detailed assessment, apply the criteria listed under "Assessing Programmatic Defenses" in the "Assessing Software Protection Schemes" chapter.

## Remediation

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.

## 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.a
lgorithm);
 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, checksum
Length);
 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?

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - [https://www.owasp.org/index.php/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."

### CWE

- N/A

# 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 off. */
}
}

```

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.

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M9-Reverse\\_Engineering](https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering)

### OWASP MASVS

- V8.4: "The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device."

### CWE

N/A

### Tools

- frida - <https://www.frida.re/>

## 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           |
|------------------------------------------------------|----------------------|-------------------|
| g                                                    | 0's                  | emulator          |
| TelephonyManager.getDeviceId()                       | 155552155            | emulator          |
| or                                                   |                      |                   |
| TelephonyManager.getLine1Number()                    | us                   | possibly emulator |
| or                                                   |                      |                   |
| TelephonyManager.getNetworkCountryIso()              | 3                    | possibly emulator |
| or                                                   |                      |                   |
| TelephonyManager.getNetworkType()                    | 310                  | possibly emulator |
| or                                                   |                      |                   |
| TelephonyManager.getNetworkOperator().substring(0,3) | 260                  | possibly emulator |
| or                                                   |                      |                   |
| TelephonyManager.getPhoneType()                      | 1                    | possibly emulator |
| or                                                   |                      |                   |
| TelephonyManager.getSimCountryIso()                  | us                   | possibly emulator |
| or                                                   |                      |                   |
| TelephonyManager.getSimSerialNumber()                | 89014103211118510720 | emulator          |
| or                                                   |                      |                   |
| TelephonyManager.getSubscriberId()                   | 3102600000000000     | emulator          |
| or                                                   |                      |                   |
| TelephonyManager.getVoiceMailNumber()                | 15552175049          | emulator          |
| or                                                   |                      |                   |

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.

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M9-Reverse\\_Engineering](https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering)

### OWASP MASVS

- V8.5: "The app detects, and responds to, being run in an emulator."

### CWE

N/A

### Tools

N/A

## 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.

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-](https://www.owasp.org/index.php/Mobile_Top_10_2016-)

[M9-Reverse\\_Engineering](#)**OWASP MASVS**

- v8.6: "The app detects, and responds to, tampering the code and data in its own memory space."

**CWE**

N/A

## 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.

### 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"/>`.

## 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 MyInstanceIDServicr 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;
```

## 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.

## **Remediation**

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. Next, the Build.Serial was often used. Now, apps targeting Android O will get "UNKNOWN"

when they request the Build.Serial. 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 can 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.

The following three identifiers can be possibly used.

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M9-Reverse\\_Engineering](https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering)

### OWASP MASVS

- V8.11: "The app implements a 'device binding' functionality using a device fingerprint derived from multiple properties unique to the device."

### CWE

N/A

### Tools

- ADB & DDMS
- Android Emulator or two rooted devices.

## 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](https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering)

### OWASP MASVS

- 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."
- V8.13 - "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."

### CWE

---

- N/A

## Tools

- N/A

# iOS Platform Overview

iOS is the operating system that powers all of Apple's iDevices, including the iPhone, iPad, and iPod Touch. It is a derivative of Mac OS (formerly OS X), and as such runs a modified version of the XNU kernel. Compared to their Desktop relatives however, iOS apps run in a more restricted environment: They are isolated from each other on the file system level, and are significantly limited in terms of system API access.

iOS is more "closed" than Android: Apple also keeps tight control over which apps are allowed to run on iOS devices, and side-loading of apps is only possible with jailbreak or complicated workarounds.

Apps are sandboxed just like in Android, but in contrast to Android's Binder IPC, iOS offers very little IPC functionality. This means more limited options for developers, but also less potential attack surface.

The uniform hardware and tight integration between hardware and software creates another security advantage: For example, developers can rely on a hardware-backed keychain and file system encryption being available. Also, iOS updates are rolled out to a large percentage of users quickly, meaning less need to support older, less secure versions of iOS.

All of this doesn't mean however that iOS app developers need not worry about security. Topics like Data protection and Keychain, TouchID authentication, and network security still leave plenty of margin for errors. In the following chapters, we document the iOS security architecture, followed by security testing and reverse engineering howtos. We'll then map the seven categories of the MASVS to iOS and outline test cases for each requirement.

## The iOS Security Architecture

The iOS security architecture consists of five core features.

- Secure Boot
- Sandbox
- Code Signing
- Encryption and Data Protection
- General Exploit Mitigations

## Hardware Security

The iOS security architecture makes heavy use of hardware-based security features that enhance overall performance and security. Each device comes with two built-in AES 256-bit keys, UID and GID, fused/compiled into the application processor and Secure Enclave during manufacturing. There is no way to directly read these keys through software or debugging interfaces such as JTAG. Encryption and decryption operations are performed by hardware AES crypto-engines with exclusive access to the keys.

The GID is a common value shared between all processors in a class of devices and known to Apple, and is 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 used for device-level file system encryption. Because they are not recorded during manufacturing, not even Apple can restore the file encryption keys for a particular device.

To enable 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 the iOS device is powered on, it reads the initial instructions from the read-only Boot ROM, which bootstraps the system. This memory contains immutable code, together with Apple Root CA, which is etched in the silicon die during fabrication process, creating root of trust. In the next step, the Boot ROM code checks if signature of iBoot bootloader is correct. Once the signature is validated, the iBoot checks the signature of next boot stage, which is iOS kernel. If any of these step failed, the boot process is immediately terminated and the device enters recovery mode and displays "Connect to iTunes" screen. If, however, the Boot ROM fails to load, the device enters special low level recovery mode, which is called Device Firmware Upgrade (DFU). This is the last resort to recover the device to original state. There will be no sign of activity of the device, i.e. the screen will not display anything.

The entire process is called "Secure Boot Chain" and ensures that it is running only on Apple-manufactured devices. The Secure Boot chain consists of kernel, bootloaders, kernel extensions and baseband firmware. All new devices that have Secure Enclave coprocessor, i.e. starting from iPhone 5s also use secure boot process to ensure that the firmware within Secure Enclave is trusted.

## Sandbox

The sandbox is an access control technology that was provided for iOS and it is enforced at kernel level. Its purpose is to limit the impact and damage to the system and user data that may occur when an app is compromised.

The iOS Sandbox is derived from TrustedBSD MAC framework implemented as kernel extension 'Seatbelt'. iPhone Dev Wiki (<http://iphonedevwiki.net/index.php/Seatbelt>) provides some (a bit outdated) information about the sandbox.

As a principle, all user applications run under the same user `mobile`, with only a few system applications and services running as `root`. Access to all resources, like files, network sockets, IPCs, shared memory, etc. will be then controlled by the sandbox.

## Code Signing

Application code signing is different than in Android. In the latter you can sign with self-signed key and main purpose would be to establish root of trust for future application updates. In other words, to make sure that only the original developer of a given application would be able to update it. In Android, applications can be distributed freely as APK files or from Google Play. On the contrary, Apple allows app distribution only via App Store.

There exist at least two scenarios where you can install an application without App Store:

1. via Enterprise Mobile Device Management. This requires the company to have company-wise certificate signed by Apple
2. via sideloading - i.e. by signing the app with developer's certificate and installing it on one device. There is an upper limit of number of devices that can be used with the same certificate

Developer Profile and Apple-signed certificate is required in order to deploy and run an application. Developers need to register with Apple and join the Apple Developer Program and pay subscription fee (<https://developer.apple.com/support/compare-memberships/>) to get full range of development and deployment possibilities. Free account still allows you to compile and deploy an application via sideload.

## Encryption and Data Protection

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 level based crypto engine, based on 256-bit Advanced Encryption Standard (AES), that works in conjunction with a SHA-1 cryptographic hash function.

Besides that, there is unique identifier (UID) built into the device's hardware with an AES 256-bit key fused into the application processor. This UID is specific to the device and is not recorded elsewhere. As of writing, it is not possible for software or firmware to read it directly. As the key is burnt into the silicon chip, it cannot be tampered with or bypassed. It is only the crypto engine which can access it. It is through this that data is eventually cryptographically tied to a specific device and therefore cannot be related to any other identifier or device.

Building encryption into the physical architecture makes it easier to encrypt all data stored on an iOS device. This allows Apple to enable this level of encryption by default and disabling this is not permitted. The use of this encryption only functions as a way to only facilitate a fast, secure wipe of the system. This is an important feature, especially if a device is lost or stolen and remote wipe has been configured beforehand. Under such circumstances, a device's data can theoretically be erased before someone can hack or jailbreak it. But if a device can't be wiped quickly enough, a hacker can crack the security and get at sensitive data.

Data protection is implemented at the software level and works with the hardware and firmware encryption to provide a greater degree of security.

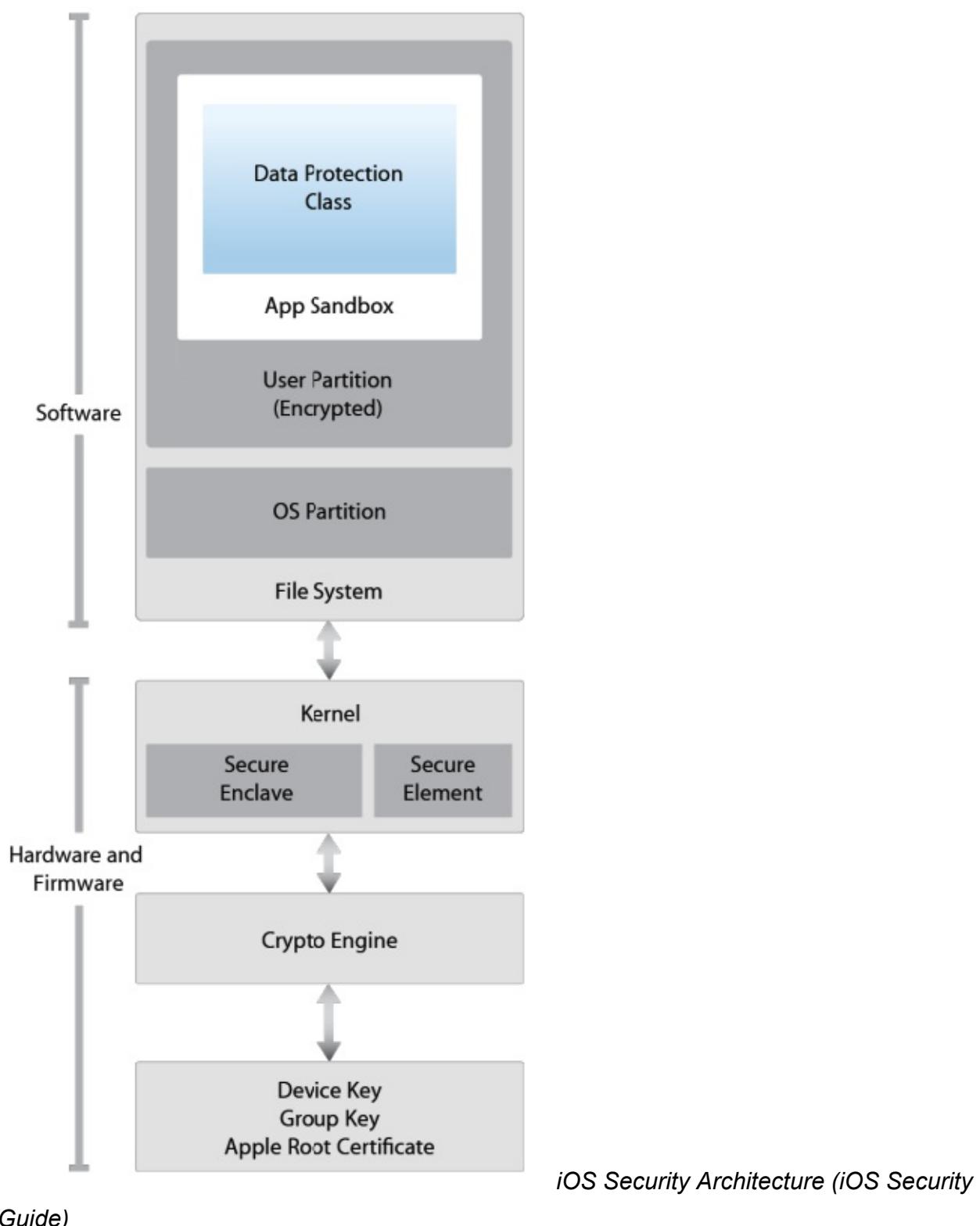
When data protection is enabled, each data file is associated with a specific class that supports a different level of accessibility and protects data based on when it needs to be accessed. The encryption and decryption operations associated with each class are based on multiple key mechanisms that utilizes the device's UID and passcode, plus a class key, file system key and per-file key. The per-file key is used to encrypt the file content. 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 and for a device to utilize data protection, a passcode must be used when accessing that device. The passcode not only unlocks the device, but also combined with the UID to create iOS encryption keys that are more resistant to hacking efforts and brute-force attacks. It is with this that users need to enable passcodes on their devices to enable data protection.

## General Exploit Mitigations

iOS currently implements two specific security mechanisms, namely address space layout randomization (ASLR) and eXecute Never (XN) bit, to prevent code execution attacks.

ASLR is a technique that does the job of randomizing the memory location of the program executable, data, heap and stack on every execution of the program. As the shared libraries need to be static in order to be shared by multiple processes, the addresses of shared libraries are randomized every time the OS boots instead of every time when the program is invoked.

Thus, this makes the specific memory addresses of functions and libraries hard to predict, thereby preventing attacks such as a return-to-libc attack, which relies upon knowing the memory addresses of basic libc functions.



## Further Reading

- [Apple iOS Security Guide](#)
- Jonathan Levin, Mac OS X and iOS Internals [#levin]

## Software Development on iOS

As with other platforms, Apple provides a Software Development Kit (SDK) for iOS that helps developers to develop, install, run and test native iOS Apps by offering different tools and interfaces. XCode Integrated Development Environment (IDE) is used for this purpose and iOS applications are implemented either by using Objective-C or Swift.

Objective-C is an object-oriented programming language that adds Smalltalk-style messaging to the C programming language and is used on macOS and iOS to develop desktop and mobile applications respectively. Both macOS and iOS are implemented by using Objective-C.

Swift is the successor of Objective-C and allows interoperability with the same and was introduced with Xcode 6 in 2014.

## Understanding iOS Apps

iOS applications are distributed in IPA (iOS App Store Package) archives. This IPA file contains all the necessary (for ARM compiled) application code and resources required to execute the application. The container is in fact a ZIP compressed file, which can be easily decompressed.

An IPA has a built-in structure for iTunes and App Store to recognize, The example below shows the high level structure of an IPA.

- /Payload/ folder contains all the application data. We will come back to the content 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 images used as the application's icon
- /iTunesMetadata.plist contains various bits of information, ranging from 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 for responding to user interactions on an Apple watch.

## IPA Payloads - A Closer Look

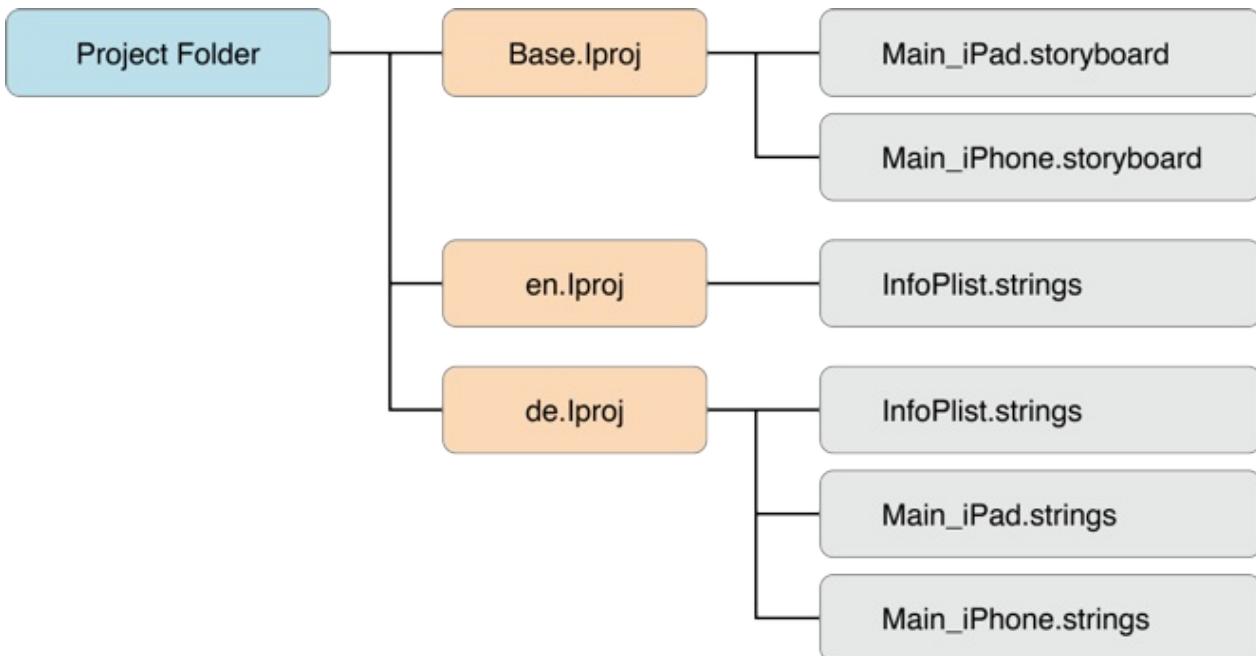
Let's take a closer look now at the different files that are to be found in the ZIP compressed IPA container. It is necessary to understand that this is the raw architecture of the bundle container and not the definitive form after installation on the device. It uses a relatively flat structure with few extraneous directories in an effort to save disk space and simplify access

to the files. The bundle contains the application executable and any resources used by the application (for instance, the application icon, other images, and localized content) in the top-level bundle directory.

- **MyApp:** The executable containing the application's code, which is compiled and not in a 'readable' format.
- **Application:** Icons used at specific times to represent the application.
- **Info.plist:** Containing configuration information, such as its bundle ID, version number, and display name.
- **Launch images:** Images showing the initial interface of the application 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:** Contains the default interface objects to load at application launch time. Other interface objects are then either loaded from additional nib files or created programmatically by the application.
- **Settings.bundle:** Contains any application-specific preferences using property lists and other resource files to configure and display them.
- **Custom resource files:** Non-localized resources are placed at the top level directory and localized resources are placed in language-specific subdirectories of the application bundle. Resources consist of nib files, images, sound files, configuration files, strings files, and any other custom data files you need for your application.

A language.lproj folder is defined for each language that the application supports. It contains the a storyboard and strings files.

- A storyboard is a visual representation of the user interface of an iOS application, showing screens of content and the connections between those screens.
- The strings file format consists of one or more key-value pairs along with optional comments.



On a jailbroken device, you can recover the IPA for an installed iOS app using IPA Installer (see also [Testing Processes and Techniques](#)). Note that during mobile security assessments, developers will often provide you with the IPA directly. They could send you the actual file, or provide access to the development specific distribution platform they use e.g. [HockeyApp](#) or [Testflight](#).

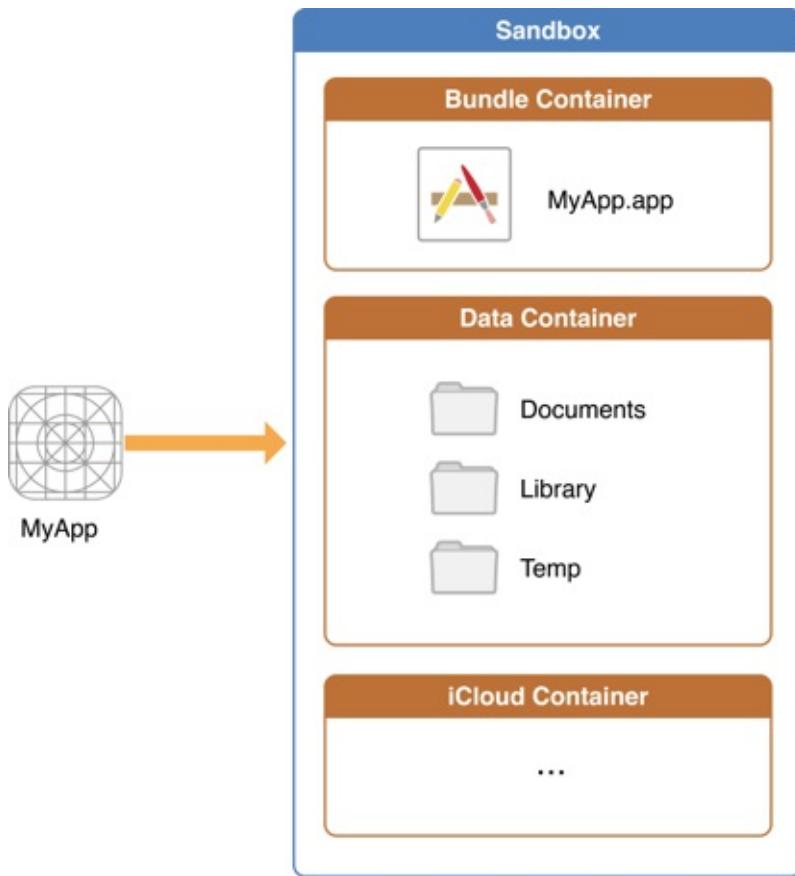
## App Structure on the iOS File System

Since iOS 8, changes were made to the way an application is stored on the device. On versions before iOS 8, applications would be unpacked to a folder in the /var/mobile/applications/ folder. The application would be identified by its UUID (Universal Unique Identifier), a 128-bit number. This would be the name of the folder in which we will find the application itself. Since iOS 8 this has changed however, so we will see that the static bundle and the application data folders are now stored in different locations on the filesystem. These folders contain information that we will need to closely examine during application security assessments.

- /var/mobile/Containers/Bundle/Application/[UUID]/Application.app contains the previously mentioned application.app data and stores the static content as well as the ARM compiled binary of the application. The content of this folder will be used to validate the code signature.
- /var/mobile/Containers/Data/Application/[UUID]/Documents contains all the data stored for the application itself. The creation of this data is initiated by the application's end user.
- /var/mobile/Containers/Data/Application/[UUID]/Library contains files necessary for the application e.g. caches, preferences, cookies, property list (plist) configuration files, etc.
- /var/mobile/Containers/Data/Application/[UUID]/Temp contains temporary files which do

not need persistence in between application launches.

The following figure represents the application's folder structure:



## The Installation Process

Different methods exist to install an IPA package on the device. The easiest solution is to use iTunes, which is the default media player from Apple. iTunes Packages exist for OS X as well as for Windows. iTunes allows you to download applications through the App Store, after which you can synchronize them to an iOS device. The App store is the official application distribution platform from Apple. You can also use iTunes to load an ipa to a device. This can be done by adding “dragging” it into the Apps section, after which we can then add it to a device.

On Linux we can make use of [libimobiledevice](#), a cross-platform software protocol library and set of tools to communicate with iOS devices natively. Through ideviceinstaller we can install packages over an USB connection. The connection is implemented using USB multiplexing daemon usbmuxd which provides a TCP tunnel over USB. During normal operations, iTunes communicates with the iPhone using this usbmux, multiplexing several “connections” over the one USB pipe. Processes on the host machine open up connections to specific, numbered ports on the mobile device.

On the iOS device, the actual installation process is then handled by installd daemon, which will unpack and install it. 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. This means that we can only install it after the code signature is valid. On a jailbroken phone this can however be circumvented using [AppSync](#), a package made available on the Cydia store. This is an alternate app store containing a lot of useful applications which leverage root privileges provided through the jailbreak in order to execute advanced functionalities. AppSync is a tweak that patches installd to allow for the installation of fake-signed IPA packages.

The IPA can also be installed directly from command line by using [ipainstaller](#). After copying the IPA onto the device, for example by using scp (secure copy), the ipainstaller can be executed with the filename of the IPA:

```
$ ipainstaller App_in_scope.ipa
```

## Code Signing and Encryption

Apple has implemented an intricate DRM system to make sure that only valid & approved code runs on Apple devices. In other words, on a non-jailbroken device, you won't be able to run any code unless Apple explicitly allows you to. You can't even opt to run code on your own device unless you enroll with the Apple developer program and obtain a provisioning profile and signing certificate. For this and other reasons, iOS has been [compared to a crystal prison](#)

In addition to code signing, *FairPlay Code Encryption* is applied to apps downloaded from the App Store. Originally, FairPlay was developed as a means of DRM for multimedia content purchased via iTunes. In that case, 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 is created and assigned to your account. The private key is stored securely on your device. This means that Fairplay-encrypted code can be decrypted only on devices associated with your account -- TODO [Be more specific] --. The usual way to obtain reverse FairPlay encryption is to run the app on the device and then dump the decrypted code from memory (see also "Basic Security Testing on iOS").

## The App Sandbox

In line with the "crystal prison" theme, sandboxing has been is a core security feature since the first releases of iOS. Regular apps on iOS are confined to a "container" that restrict access to the app's own files and a very limited amount of system APIs. Restrictions include

[#levin]:

- The app process is restricted to its own directory(below /var/mobile/Containers/Bundle/Application/) using a chroot-like mechanism.
- The mmap and mmprotect() system calls are modified to prevent apps from make writeable memory pages executable, preventing processes from executing dynamically generated code. In combination with code signing and FairPlay, this places strict limitations on what code can be run under specific circumstances (e.g., all code in apps distributed via the app store is approved by Apple).
- Isolation from other running processes, even if they are owned by the same UID;
- Hardware drivers cannot be accessed directly. Instead, any access goes through Apple's frameworks.

## App Permissions

In contrast to Android, iOS apps do not have preassigned permissions. Instead, the user is asked to grant permission during runtime when an app attempts to use a sensitive API for the first time. Once the app has asked for a permission, it is listed in the Settings > Privacy menu, allowing the user to modify the app-specific setting. Apple calls this permission concept [privacy controls](#).

Developers don't have the possibility to set the requested permissions directly - they are requesting them indirectly by using sensitive APIs. For example, when accessing the 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 permission from the user](#):

- Contacts
- Microphone
- Calendars
- Camera
- Reminders
- HomeKit
- Photos
- Health
- Motion activity and fitness
- Speech recognition
- Location Services
- Bluetooth sharing
- Media Library

- Social media accounts

## References

- [#levin] - Jonathan Levin, Mac OS X and iOS Internals, Wiley, 2013

# Basic Security Testing on iOS

## Foreword on Swift and Objective-C

Vast majority of this tutorial is relevant to applications written mainly in Objective-C or having bridged Swift types. Please note that these languages are fundamentally different. Features like method swizzling, which is heavily used by Cycript will not work with Swift methods. At the time of writing of this testing guide, Frida does not support instrumentation of Swift methods.

## Setting Up Your Testing Environment

In contrast to the Android emulator, which fully emulates the processor and hardware of an actual Android device, the simulator in the iOS SDK 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 an actual device don't run, making the simulator completely useless for black-box-analysis and reverse engineering.

Ideally you want to have a jailbroken iPhone or iPad available for running tests. That way, you get root access to the device and can install a variety of useful tools, making the security testing process easy. 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 less smooth experience.

For your mobile app testing setup you should have at least the following:

- Laptop with admin rights, VirtualBox with Kali Linux
- WiFi network with client to client traffic permitted (multiplexing through USB is also possible)
- At least one jailbroken iOS device (with desired iOS version)
- Burp Suite or other web proxy

If you want to get serious with iOS security testing, you need a Mac, for the simple reason that XCode and the iOS SDK are only available for Mac OS. Many tasks that you can do effortlessly on Mac are a chore, or even impossible, on Windows and Linux. The following setup is recommended:

- Macbook with XCode and Developer's Profile
- WiFi network as previously
- At least two iOS devices, one jailbroken, second non-jailbroken
- Burp Suite or other web proxy
- Hopper or IDA Pro

## Jailbreaking the iOS Device

iOS jailbreaking is often compared to Android rooting. Actually, we have three different things here and it is important to clearly distinguish them.

On the Android side we have:

- Rooting: this typically consists of installing the `su` binary within the existing system or replacing the whole system with an already rooted custom ROM. Normally, exploits are not required in order to obtain root access.
- Flashing custom ROMs (that might be already rooted): allows to completely replace the OS running on the device after unlocking the bootloader (which might require an exploit). There is no such thing on iOS as it is closed-source and *thanks* to the bootloader that only allows Apple-signed images to be booted and flashed (which is also the reason why downgrades/upgrades to not-signed-by-Apple iOS images are not possible).

On iOS side we have:

- Jailbreaking: colloquially, the word "jailbreak" is often used to refer to all-in-one tools that automate the complete jailbreaking process, from executing the exploit(s) to disable system protections (such as Apple's code signing mechanisms) and install the Cydia app store. If you're planning to do any form of dynamic security testing on an iOS device, you'll have a much easier time on a jailbroken device, as most useful testing tools are only available outside the App Store.

Developing a jailbreak for any given version of iOS is not an easy endeavor. As a security tester, you'll most likely want to use publicly available jailbreak tools (don't worry, we're all script kiddies in some areas). Even so, we recommend studying the techniques used to jailbreak various versions of iOS in the past - you'll encounter many highly interesting exploits and learn a lot about the internals of the OS. For example, Pangu9 for iOS 9.x [exploited at least five vulnerabilities](#), including a use-after-free bug in the kernel (CVE-2015-6794) and an arbitrary file system access vulnerability in the Photos app (CVE-2015-7037).

## Types of Jailbreaking Methods

In jailbreak lingo, we talk about tethered and untethered jailbreaking methods. In the "tethered" scenario, the jailbreak doesn't persist throughout reboots, so the device must be connected (tethered) to a computer during every reboot to re-apply it. "Untethered" jailbreaks need only be applied once, making them the most popular choice for end users.

## Benefits of Jailbreaking

A standard user will want to jailbreak in order to tweak the iOS system appearance, add new features or install *free* third party apps. However, for a security tester the benefits of jailbreaking an iOS Device go far beyond simply tweaking the system. They include but are not limited to the following:

- Removing the part of the security (and other) limitations on the OS imposed by Apple
- Providing root access to the operating system
- Allowing applications and tools not signed by Apple to be installed and run without any restrictions
- Debugging and performing dynamic analysis
- Providing access to the Objective-C Runtime

## Caveats and Considerations about Jailbreaking

Jailbreaking iOS devices is becoming more and more complicated as Apple keeps hardening the system and patching the corresponding vulnerabilities that jailbreaks are based on. Additionally, it has become a very time sensitive procedure as they stop signing these vulnerable versions within relative short time intervals (unless they are hardware-based vulnerabilities). This means that, contrary to Android, you can't downgrade iOS version with one exception explained below, which naturally creates a problem, when there is a major bump in iOS version (e.g. from 9 to 10) and there is no public jailbreak for the new OS.

A recommendation here is: if you have a jailbroken device that you use for security testing, keep it as is, unless you are 100% sure that you can perform a newer jailbreak to it. Additionally you can think of having a second one, which will be updated with every major iOS release and wait for public jailbreak to be released. Once a public jailbreak is released, Apple is quite fast in releasing a patch, hence you have only a couple of days to upgrade to the newest iOS version and jailbreak it (if upgrade is necessary).

The iOS upgrade process is performed online and is based on a challenge-response process. The device will perform the OS installation only if the response to the challenge is signed by Apple. This is what researchers call 'signing window' and explains the fact that you can't simply store the OTA firmware package downloaded via iTunes and load it to the device at any time. During minor iOS upgrades, it is possible that two versions are signed at the same time by Apple. This is the only case when you can possibly downgrade iOS version. You can check current signing window and download OTA Firmwares from the [IPSW Downloads website](#).

## How to Jailbreak iOS?

Jailbreaking methods vary across iOS versions. Best choice is to [check if a public jailbreak is available for your iOS version](#). Beware of fake tools and spyware that is often distributed around the Internet, often hiding behind domain names similar to the jailbreaking group/author.

Let's say you have a device running iOS 9.0, for this version you'll find a jailbreak (Pangu 1.3.0), at least for 64 bit devices. In the case that you have another version for which there's not a jailbreak available, you could still jailbreak it if you downgrade/upgrade to the target *jailbreakable* iOS version (via IPSW download and iTunes). However, this might not be possible if the required iOS version is not signed anymore by Apple.

The iOS jailbreak scene evolves so rapidly that it is difficult to provide up-to-date instructions. However, we can point you to some, at the time of this writing, reliable sources:

- [The iPhone Wiki](#)
- [Redmond Pie](#)
- [Reddit Jailbreak](#)

Note that obviously OWASP and the MSTG will not be responsible if you end up brickling your iOS device!

## Dealing with Jailbreak Detection

Some apps attempt to detect whether the iOS device they're installed on is jailbroken. The reason for this jailbreaking deactivates some of iOS' default security mechanisms, leading to a less trustable environment.

The core dilemma with this approach is that, by definition, jailbreaking causes the app's environment to be unreliable: The APIs used to test whether a device is jailbroken can be manipulated, and with code signing disabled, the jailbreak detection code can easily be patched out. It is therefore not a very effective way of impeding reverse engineers. Nevertheless, jailbreak detection can be useful in the context of a larger software protection scheme. We'll revisit this topic in the next chapter.

## Preparing the Test Environment



Once you have your iOS device jailbroken and Cydia is installed (as per screenshot), proceed as following:

1. From Cydia install aptitude and openssh
2. SSH to your iDevice
  - Two users are `root` and `mobile`
  - Default password is `alpine`
3. Add the following repository to Cydia: <https://build.frida.re>
4. Install Frida from Cydia
5. Install following packages with aptitude

```
inetutils
syslogd
less
com.autopear.installipa
class-dump
com.ericasadun.utilities
odcctools
cycrypt
sqlite3
adv-cmds
bigbosshackertools
```

Your workstation should have SSH client, Hopper Disassembler, Burp and Frida installed. You can install Frida with pip, for instance:

```
$ sudo pip install frida
```

## SSH Connection via USB

As per the normal behavior, iTunes communicates with the iPhone via the `usbmux`, which is a system for multiplexing several "connections" over one USB pipe. This system provides a TCP-like system where multiple processes on the host machine open up connections to specific, numbered ports on the mobile device.

`usbmuxd` is a socket daemon that watches for iPhone connections via USB. You can use it to map listening localhost sockets from the mobile device to TCP ports on your host machine. This conveniently allows you to SSH into your device independent of network settings. When it detects an iPhone running in normal mode, it will connect to it and then start relaying requests that it receives via `/var/run/usbmuxd`.

On MacOS:

```
$ brew install libimobiledevice
$ iproxy 2222 22
$ ssh -p 2222 root@localhost
iPhone:~ root#
```

Python client:

```
$./tcprelay.py -t 22:2222
$ ssh -p 2222 root@localhost
iPhone:~ root#
```

## Typical iOS Application Test Workflow

Typical workflow for iOS Application test is following:

1. Obtain IPA file
2. Bypass jailbreak detection (if present)
3. Bypass certificate pinning (if present)
4. Inspect HTTP(S) traffic - usual web app test
5. Abuse application logic by runtime manipulation
6. Check for local data storage (caches, binary cookies, plists, databases)
7. Check for client-specific bugs, e.g. SQLi, XSS
8. Other checks like: logging to ASL with NSLog, application compile options, application screenshots, no app backgrounding

## Static Analysis

### Without Source Code

#### Folder Structure

System applications can be found in "/Applications". For user-installed apps, you can use [IPA Installer Console](#) to navigate to the appropriate folders:

```
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-FF94
99E0B19C
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-FB062C86
4E56
```

As you can see, there are three main directories: `Bundle`, `Application` and `Data`. The `Application` directory is simply a subdirectory of `Bundle`. The static installer files are located in `Application`, whereas all user data resides in the `Data` directory.

The random string in the URI is application's GUID, which will be different from installation to installation.

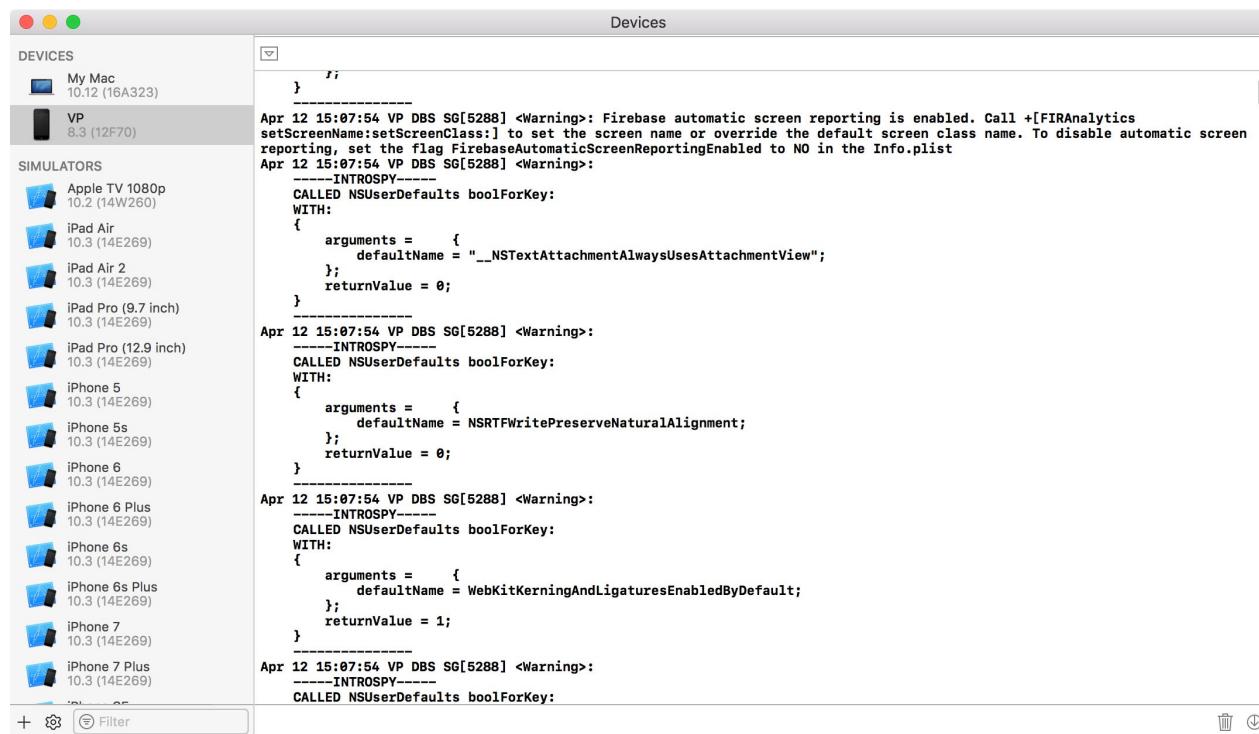
## Dynamic Analysis

### Monitoring Console Logs

Many apps log informative (and potentially sensitive) messages to the console log. Besides that, the log also contains crash reports and potentially 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 corner of the right section of the Devices window to expose the console log contents

To save the console output to a text file, click the circle with a downward-pointing arrow at the bottom right.



## Setting up a Web Proxy using BurpSuite

Burp Suite is an integrated platform for performing security testing of mobile and web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface, to finding and exploiting security vulnerabilities. It is a toolkit where Burp proxy operates as a web proxy server, and sits as a man-in-the-middle between the browser and web server(s). It allows the interception, inspection and modification of the raw HTTP traffic passing in both directions.

Setting up Burp to proxy your traffic through is pretty straightforward. It is assumed that you have both: iDevice and workstation connected to the same WiFi network where client to client traffic is permitted. If client-to-client traffic is not permitted, it is possible to use usbmuxd in order to connect to Burp through USB.

Portswigger also provides a good [tutorial on setting up an iOS Device to work with Burp](#).

### Configure the Burp Proxy Listener

- In Burp, go to the “Proxy” tab and then the “Options” tab.
- In the “Proxy Listeners” section, click the “Add” button.
- In the "Binding" tab, in the “Bind to port:” box, enter a port number that is not currently in use, e.g. “8082”.
- Then select the “All interfaces” option, and click "OK".
- The Proxy listener should now be configured and running.

### Configure Your Device to Use the Proxy

- In your iOS device, go to the “Settings” menu.
- Tap the “Wi-Fi” option from the "Settings" menu.
- Tap the “i” (information) option next to the name of your network.
- Under the "HTTP PROXY" title, tap the “Manual” tab.
- In the "Server" field, enter the IP address of the computer that is running Burp.
- In the “Port” field, enter the port number configured in the “Proxy Listeners” section earlier, in this example “8082”.

### Test the Burp configuration for HTTP Requests

- In Burp, go to the "Proxy Intercept" tab, and ensure that intercept is “on”.
- Open the browser on your iOS device and go to an HTTP web page.
- The request should be intercepted in Burp.

### Installing Burp's CA Certificate in an iOS Device:

- With Burp running, visit <http://burp> in your browser and click the “CA Certificate” link to download and install your Burp CA certificate.

### Test the Burp configuration for HTTPS Requests

- In Burp, go to the "Proxy Intercept" tab, and ensure that intercept is “on”.
- Open the browser on your iOS device and go to an HTTPS web page.
- The request should be intercepted in Burp.

## Setting up a Web Proxy using OWASP ZAP

-- TODO

## Bypassing Certificate Pinning

When you try to intercept the mobile app and server communication you might fail due to certificate pinning. The Certificate Pinning is a practice used to tighten security of TLS connection. When an application is connecting to the server using TLS, it checks if the server's certificate is signed with trusted CA's private key. The verification is based on checking the signature with public key that is within device's key store. This in turn contains public keys of all trusted root CAs.

Certificate pinning means that our application will have server's certificate or hash of the certificate hardcoded into the source code. This protects against two main attack scenarios:

- Compromised CA issuing certificate for our domain to a third-party
- Phishing attacks that would add a third-party root CA to device's trust store

The simplest method is to use `SSL Kill Switch` (can be installed via Cydia store), which will hook on all high-level API calls and bypass certificate pinning. There are some cases, though, where certificate pinning is more tricky to bypass. Things to look for when you try to bypass certificate pinning are:

- following API calls: `NSURLSession`, `CFStream`, `AFNetworking`
- during static analysis, try to look for methods/strings containing words like 'pinning', 'X509', 'Certificate', etc.
- sometimes, more low-level verification can be done using e.g. `openssl`. There are tutorials [20] on how to bypass this.
- some dual-stack applications written using Apache Cordova or Adobe Phonegap heavily use callbacks. You can look for the callback function called upon success and call it manually with `Crypt`
- sometimes the certificate resides as a file within application bundle. It might be sufficient to replace it with Burp's certificate, but beware of certificate's SHA sum that might be hardcoded in the binary. In that case you must replace it too!

Certificate pinning is a good security practice and should be used for all applications handling sensitive information. [EFF's Observatory](#) provides list of root and intermediate CAs that are by default trusted on major operating systems. Please also refer to a [map of the 650-odd organizations that function as Certificate Authorities trusted \(directly or indirectly\) by Mozilla or Microsoft](#). Use certificate pinning if you don't trust at least one of these CAs.

If you want to get more details on white-box testing and usual code patterns, refer to iOS Application Security by David Thiel. It contains description and code snippets of most-common techniques used to perform certificate pinning.

To get more information on testing transport security, please refer to section "Testing Network Communication".

## Dynamic Analysis On Jailbroken Devices

Life is easy with a jailbroken device: Not only do you gain easy access to the app's sandbox, you can also use more powerful dynamic analysis techniques due to the lack of code signing. On iOS, most dynamic analysis tools are built on top of Cydia Substrate, a framework for developing runtime patches that we will cover in more detail in the "Tampering and Reverse Engineering" chapter. For basic API monitoring purposes however, you can get away without knowing Substrate in detail - you can simply use existing tools built for this purpose.

### Copying App Data Files

Files belonging to an app are stored app's data directory. To identify the correct path, ssh into the device and retrieve the package information using IPA Installer Console:

```
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-AF8C
DCA8A2F1
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-91A5-1FED3525
8D87
```

You can now simply archive the data directory and pull it from the device using 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 the contents of the KeyChain on a jailbroken device. The easiest way of running 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 however that this binary is signed with a self-signed certificate with a "wildcard" entitlement, granting access to *all* items in the Keychain - if you are paranoid, or have highly sensitive private data on your test device, you might 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.

## 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 can do pretty much everything you need for a dynamical analysis (of course, you can't break out of the sandbox that way, but you usually don't need to). This technique however works only on 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 will refuse to run an app unless you get the provisioning profile and code signature header absolutely right. This requires you to learn

about a whole lot of concepts - different types of certificates, BundleIDs, application IDs, team identifiers, and how they are tied together using Apple's build tools. Suffice it to say, getting the OS to run a particular binary that hasn't been built using the default way (XCode) can be an daunting process.

The toolset we're going to use consists of optool, Apple's build tools and some shell commands. Our method is inspired by the resign script from [Vincent Tan's Swizzler project](#). An alternative way of repackaging using different tools was [described by NCC group](#).

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 using Frida.

## Getting a Developer Provisioning Profile and Certificate

The *provisioning profile* is a plist file signed by Apple that whitelists your code signing certificate on one or multiple devices. In other words, this is Apple explicitly allowing your app to run in certain contexts, 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 do the actual signing.

Depending on whether you're registered as an iOS developer, you can use one of the following two ways to obtain a certificate and provisioning profile.

### With an iOS developer account:

If you have developed and deployed apps iOS using Xcode before, you'll already have your own code signing certificate installed. Use the `security` tool to list your existing 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 [8]. The App ID can be anything - you can use the same App ID for re-signing multiple apps. Make sure you create a *development* profile and not a *distribution* profile, as you'll want to be able to debug the app.

In the examples below I'm using my own signing identity which is associated with my company's development team. I created the app-id "sg.vp.repackaged", as well as a provisioning profile aptly named "AwesomeRepackaging" for this purpose, and ended up

with the file AwesomeRepackaging.mobileprovision - exchange this with your own filename in the shell commands below.

### With a regular iTunes account:

Mercifully, Apple will issue a free development provisioning profile even if you're not a paying developer. You can obtain the profile with Xcode using your regular Apple account - simply build an empty iOS project and extract embedded.mobileprovision from the app container. The NCC blog post explains this process in great detail.

Once you have 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 later for code signing, so extract them to a separate plist file as shown below. It is also worth having a look at the contents of the file to check if everything looks 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 one app with this particular 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 Mach-O header of the main executable. [Optool](#) can be used to automate this process:

```
$ git clone https://github.com/alexzielenski/optool.git
$ cd optool/
$ git submodule update --init --recursive
```

We'll also use [ios-deploy](#) [10], a tools that enables deploying and debugging of iOS apps without using Xcode:

```
git clone https://github.com/alexzielenski/optool.git
cd optool/
git submodule update --init --recursive
```

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 OS X 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 use any zip tool to unpack the archive. Then, copy `FridaGadget.dylib` into the app directory, and add a load command to the "UnCrackable Level 1" binary using optool.

```
$ 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...
```

Such blatant tampering of course invalidates the code signature of the main executable, 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. The reason for this is that the "codesign" tool will read the Bundle ID from Info.plist during signing - a 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:

```
$ rm -rf Payload/F/_CodeSignature
$ /usr/bin/codesign --force --sign 8004380F331DCA22CC1B47FB1A805890AE41C938 Payload/UnCrackable\ Level\ 1.app/FridaGadget.dylib
Payload/UnCrackable Level 1.app/FridaGadget.dylib: replacing existing signature
$ /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
```

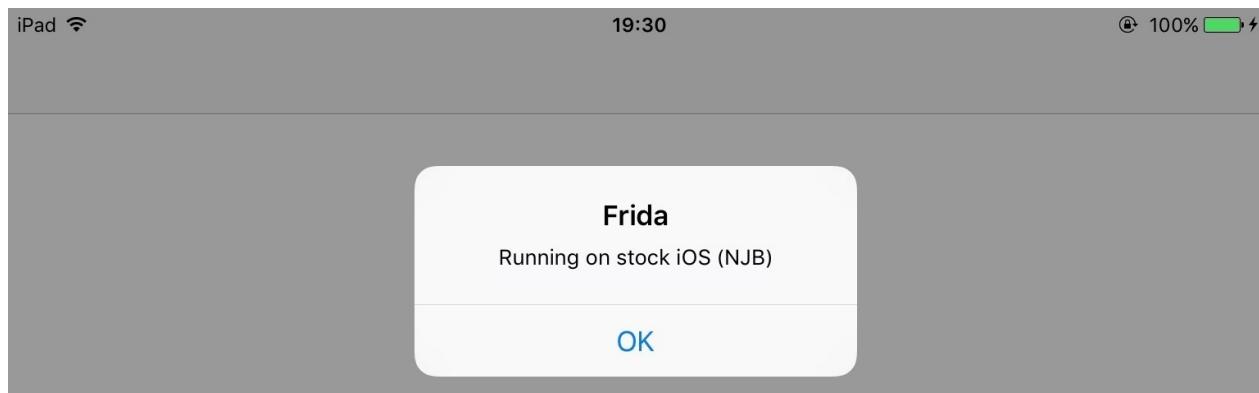
## Installing and Running the App

Now you should be all set for running 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 on the device 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.

If something goes wrong (which it usually does), mismatches between the provisioning profile and code signing header are the most likely suspect. In that case it is helpful to read the [official documentation](#) and gaining a deeper understanding of the code signing process. I also found Apple's [entitlement troubleshooting page](#) to be a useful resource.

## Network Monitoring/Sniffing

Dynamic analysis by using an interception proxy can be straight forward if standard libraries in iOS are used and all communication is done via HTTP. But what if XMPP or other protocols are used that are not recognized by your interception proxy? What if mobile application development platforms like Xamarin are used, where the produced apps do not use the local proxy settings of your iOS device? In this case we need to monitor and analyze the network traffic first in order to decide what to do next.

On iOS it is possible to remotely sniff all traffic in real-time by using Wireshark and [creating a Remote Virtual Interface](#) for your iOS device. First ensure you have Wireshark installed on your macOS workstation.

1. Connect your iOS device to your macOS workstation via a USB cable.
2. Ensure that both your iOS device and your macOS workstation are connected to the same network.
3. Open up "Terminal" on your 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 accordingly in Wireshark to display what you want to monitor, for example `ip.addr == 192.168.1.1 && http`.

# Testing Local Authentication in iOS Apps

In local authentication, the app authenticates the user against credentials stored on the device itself. In other words, the user "unlocks" the app or some functionality within the app with a PIN, password or fingerprint that is verified only locally. This is sometimes done to allow users to more easily resume an existing session with the remote service, or as a means of step-up authentication to protect some critical functionality.

## Testing Local Authentication

### Overview

On iOS, multiple possibilities exist for integrating local authentication into iOS apps. The [Local Authentication Framework](#) offers a set of APIs that display an authentication dialog to the user. In the context of connecting to a remote service, it is also possible (and recommended) to leverage the Keychain for implementing local authentication.

#### Local Authentication Framework

The Local Authentication Framework provides facilities for requesting a passphrase or TouchID authentication from users. With local authentication, an authentication prompt is displayed to the user programmatically using the function `evaluatePolicy` of the `LAContext` object.

Two policies are available that define the acceptable forms of authentication:

- `LAPolicyDeviceOwnerAuthentication`: The user is asked to perform TouchID authentication if available. If TouchID is not activated, they are asked to enter the device passcode. If the device passcode is not enabled, policy evaluation fails.
- `LAPolicyDeviceOwnerAuthenticationWithBiometrics`: The user is asked to perform TouchID authentication.

The `evaluatePolicy` function returns a boolean value indicating whether the user has authenticated successfully.

```

let myContext = LAContext()
let myLocalizedString = <#String explaining why app needs authentication#>

var authError: NSError? = nil
if #available(iOS 8.0, OSX 10.12, *) {
 if myContext.canEvaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, error: &authError) {
 myContext.evaluatePolicy(LAPolicy.DeviceOwnerAuthenticationWithBiometrics, localizedReason: myLocalizedString) { (success, evaluateError) in
 if (success) {
 // User authenticated successfully, take appropriate action
 } else {
 // User did not authenticate successfully, look at error and take appropriate action
 }
 }
 } else {
 // Could not evaluate policy; look at authError and present an appropriate message to user
 }
} else {
 // Fallback on earlier versions
}

```

*TouchID authentication using the Local Authentication Framework (official Apple code sample).*

## Using Keychain Services for Local Authentication

The iOS Keychain APIs can be used to implement local authentication. In that case, some secret authentication token (or other piece of secret data identifying the user) is stored in the Keychain. In order to authenticate to the remote service, the user then needs to unlock the Keychain using their passphrase or fingerprint and obtain the secret data (the Keychain mechanism is explained in more detail in the chapter "Testing Data Storage").

## Static Analysis

A common mistake when using local authentication is putting too much trust into the purely event-based (Local Authentication Framework) implementation. This type of authentication is effective on the user interface level, but can easily be bypassed through patching or instrumentation.

If the app you are testing implements local authentication, make sure that sensitive flows are protected using the Keychain services method. For example, in some apps an existing user session can be resumed using TouchID authentication. In that case, make sure that the

session credentials or tokens (e.g. refresh token) are stored securely in the Keychain as described above and "locked" with local authentication.

## Dynamic Analysis

If you identify a potential local authentication bypass issue, you can exploit by patching the app or using Cycript to instrument it. We'll explain how to do this in the "Reverse Engineering and Tampering" chapter.

## Remediation

The Local Authentication Framework makes it easy to add TouchID or similar authentication. For sensitive applications however, such as re-authenticating a user on a remote payment service, using the Local Authentication Framework is not recommended. Instead, the local authentication should be implemented by storing a user secret (e.g. refresh token) into the Keychain. This is done as follows:

- Use the `SecAccessControlCreateWithFlags()` API to obtain a security access control reference. Specify the `kSecAccessControlUserPresence` policy and `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` protection class.
- Inserting the data using the returned `SecAccessControlRef` in the attributes dictionary.

## References

### OWASP Mobile Top 10 2016

- M4 - Insecure Authentication - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M4-Insecure\\_Authentication](https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure_Authentication)

### OWASP MASVS

- 4.7 - "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

# Assessing Software Protection Schemes

Software protections are a controversial topic. Some security experts dismiss client-side protections outright. Security-by-obscURITY, they argue, is not *real* security, thus from a security standpoint no value is added. In the MASVS and MSTG we take a more pragmatic approach. Given that software protection controls are used fairly widely in the mobile world, we argue that there is *some* benefit to such controls, as long as they are employed with a clear purpose and realistic expectations in mind, and aren't used to *replace* solid security controls.

What's more, mobile app security testers encounter anti-reversing mechanisms in their daily work, and they not only need ways to "deal" with them to enable dynamic analysis, but also to assess whether these mechanisms are used appropriately and effectively. Giving clients advice like "you must use obfuscation" or "never obfuscate code because it's useless" doesn't cut it. However, most mobile app security testers have a background in network and web application security, and lack the reverse engineering and cracking skills required to form an opinion. On top of that, there is no methodology or even industry consensus on how anti-reversing schemes should be assessed.

The point of software-based reversing defenses is indeed to add obscurity - enough to deter some adversaries from achieving certain goals. There are several reason why developers choose to do this: For example, the intention could be to make it more difficult to steal the source code and IP, or to prevent malware running on the same device from tampering with the runtime behavior of the app.

Resilience testing is the process of evaluating the robustness of the a software protection scheme against particular threats. Typically, this kind of testing is performed using a black-box approach, with the objective of circumventing the software protection scheme and reaching a pre-defined goal, such as extracting sensitive assets. This process requires skills that are not typically associated with penetration testing: The tester must be able to handle advanced anti-reversing tricks and obfuscation techniques. Traditionally, this is the domain of malware analysts.

This form of testing can be performed in the context of a regular mobile app security test, or stand-alone to verify the effectiveness of a software protection scheme. The process consists of the following high-level steps:

1. Assess whether a suitable and reasonable threat model exists, and the anti-reversing controls fit the threat model;
2. Assess the effectiveness of the defenses in countering the identified threats using

hybrid static/dynamic analysis. In other words, play the role of the adversary, and crack the defenses!

3. In some scenarios, white-box testing can be added to assess specific features of the protection scheme in an isolated fashion (e.g., a particular obfuscation method).

Note that software protections must never be used as a replacement for security controls.

The controls listed in MASVR-R are intended to add threat-specific, additional protective controls to apps that also fulfill the MASVS security requirements.

The effectiveness of software protection schemes depends to some extent on originality and secrecy. Standardizing a particular scheme has the unfortunate side effect of making the scheme ineffective: Soon enough, a generic tool available for bypassing the scheme will be available. Instead of defining a standard way of implementing protection, we take the following approach:

1. List high-level requirements pertaining the reverse engineering processes against which should be defended against;
2. Highlight properties that determine the effectiveness of mechanisms and the overall scheme;
3. List robustness criteria for specific types of obfuscation and tampering;
4. Provide testers with knowledge, processes and tools for verifying effectiveness.

Item 1 and 2 are covered in the "Resilience Against Reverse Engineering" group of controls in the MASVS (MASVS-R), and further elaborated on in the Testing Guide. The MSTG also goes into great detail on item 3 and 4, and also documents a wide range of offensive and defensive techniques. However, it is impossible to provide a complete step-by-step guide for testing every possible protection scheme. To perform a meaningful assessment, the test must be performed by a skilled reverse engineer who is familiar with the state-of-the-art in mobile app reversing and anti-reversing.

## Assessing the Threat Model and Software Protection Architecture

Client-side protections are desirable in some cases, but unnecessary or even counter-productive in others. In the worst case, software protections cause a false sense of security and encourage bad programming practices. It is impossible to provide a generic set of resilience controls that "just works" in every possible case. For this reason, proper attack modeling is a necessary prerequisite before implementing any form of software protections. The threat model must clearly outline the client-side threats defended against. Note that the

threat model must be sensible. For example, hiding a cryptographic key in a white-box implementation is besides the point if the attacker can easily code-lift the white-box as a whole. Also, the expectations as to the effectiveness of scheme must be specified.

The [OWASP Reverse Engineering and Code Modification Prevention Project](#) lists the following technical threats associated with reverse engineering and tampering:

- Spoofing Identity - Attackers may attempt to modify the mobile application code on a victim's device to force the application to transmit a user's authentication credentials (username and password) to a third party malicious site. Hence, the attacker can masquerade as the user in future transactions;
- Tampering - Attackers may wish to alter higher-level business logic embedded within the application to gain some additional value for free. For instance, an attacker may alter digital rights management code embedded in a mobile application to attain digital assets like music for free;
- Repudiation - Attackers may disable logging or auditing controls embedded within the mobile application to prevent an organization from verifying that the user performed particular transactions;
- Information Disclosure - Attackers may modify a mobile application to disclose highly sensitive assets contained within the mobile application. Assets of interest include: digital keys, certificates, credentials, metadata, and proprietary algorithms;
- Denial of Service - Attackers may alter a mobile device application and force it to periodically crash or permanently disable itself to prevent the user from accessing online services through their device;
- Elevation of Privilege - Attackers may modify a mobile application and redistribute it in a repackaged form to perform actions that are outside of the scope of what the user should be able to do with the app.

## The Assessment Process

Software protection effectiveness can be assessed using the white-box or black-box approach. Just like in a "regular" security assessment, the tester performs static and dynamic analysis but with a different objective: Instead of identifying security flaws, the goal is to identify holes in the anti-reversing defenses, and the property assessed is *resilience* as opposed to *security*. Also, scope and depth of the assessment must be tailored to specific scenario(s), such as tampering with a particular function. Note that the resilience assessment can be performed as part of a regular security assessment.

## Design Review

Review and evaluate the design and implementation the software protection scheme and its individual components (anti-tampering, anti-debugging, device binding, obfuscating transformations, etc.).

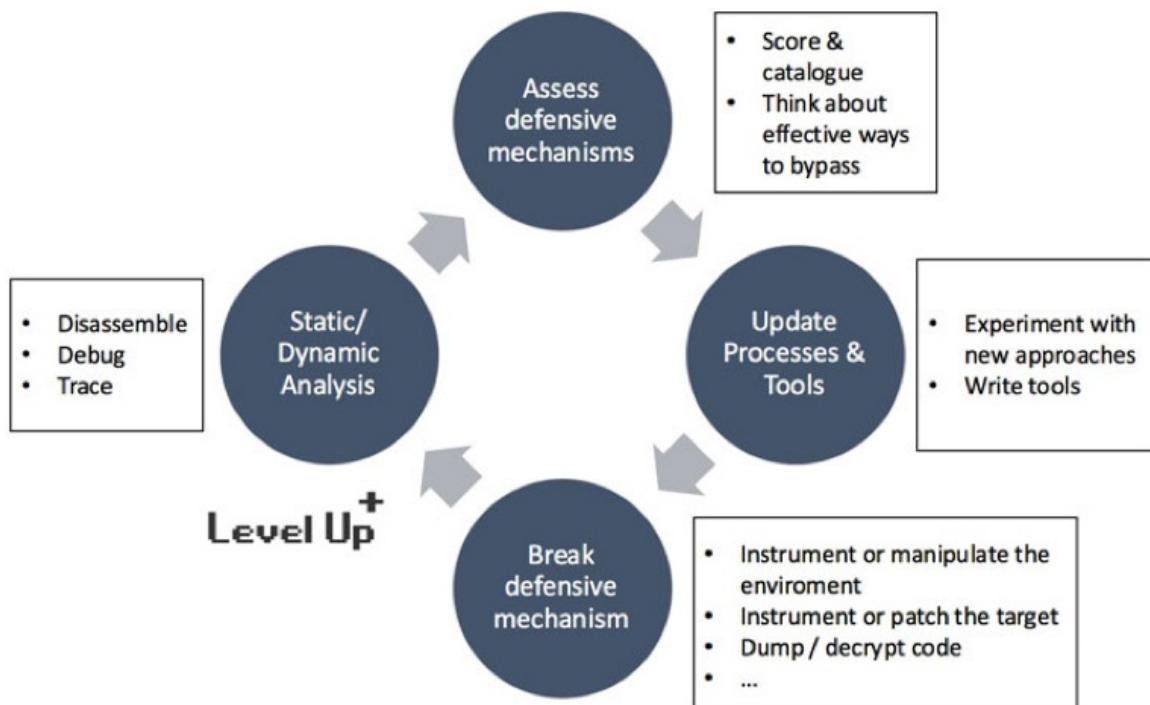
## Black-box Resilience Testing

Evaluate the robustness of their White-Box cryptographic solution against specific attacks. Without prior knowledge about the implementation, with the objective to break or circumvent the protections.

The advantage of the black-box approach is that it reflects the real-world effectiveness of the reverse engineering protections: The effort required by actual adversary with a comparable skill level and toolset would likely be close to the effort invested by the assessor.

--[ TODO ] --

Drawbacks: For one, the result is highly influenced by the skill level of the assessor. Also, the effort for fully reverse engineering a program with state-of-the-art protections is very high (which is exactly the point of having them), and some apps may occupy even experienced reverse engineers for weeks. Experienced reverse engineers aren't cheap either, and delaying the release of an app may not be feasible in an "agile" world.



## Obfuscation Effectiveness Assessment

Complex obfuscation schemes, such as custom implementations of white-box cryptography or virtual machines, are better assessed in an isolated fashion using the white-box approach. Such an assessment requires specialized expertise in cracking the particular type(s) of obfuscation. In this type of assessment, the goal is to determine resilience against current state-of-the-art de-obfuscation techniques, and providing an estimate of robustness against manual analysis.

## Key Questions

Any resilience test should answer the following questions:

**Does the protection scheme impede the threat(s) they are supposed to?**

It is worth re-iterating that there is no anti-reversing silver bullet.

**Does the protection scheme achieve the desired level of resilience?**

It is worth re-iterating that there is no anti-reversing silver bullet.

**Does the scheme defend comprehensively against processes and tools used by reverse engineers?**

--[ TODO ] --

**Are suitable types of obfuscation used in the appropriate places and with the right parameters?**

--[ TODO ] --

- Programmatic defense is a fancy word for "anti-reversing-trick". For a protection scheme to be considered effective, it must incorporate many of these defenses. "Programmatic" refers to the fact that these kinds of defenses *do* things - they are functions that prevent, or react to, actions of the reverse engineer. In this, they differ from obfuscation, which changes the way the program looks.
- Obfuscation is the process of transforming code and data in ways that make it more difficult to comprehend, while preserving its original meaning or function. Think about translating an English sentence into an French one that says the same thing (or pick a different language if you speak French - you get the point).

Note that these two categories sometimes overlap - for example, self-compiling or self-modifying code, while most would refer to as a means of obfuscation, could also be said to "do something". In general however it is a useful distinction.

Programmatic defenses can be further categorized into two modi operandi:

1. Preventive: Functions that aim to *prevent* anticipated actions of the reverse engineer. As an example, an app may use an operating system API to prevent debuggers from attaching.
2. Reactive: Features that aim to detect, and respond to, tools or actions of the reverse engineer. For example, an app could terminate when it suspects being run in an emulator, or change its behavior in some way if a debugger is detected.

You'll usually find a mix of all the above in a given software protection scheme.

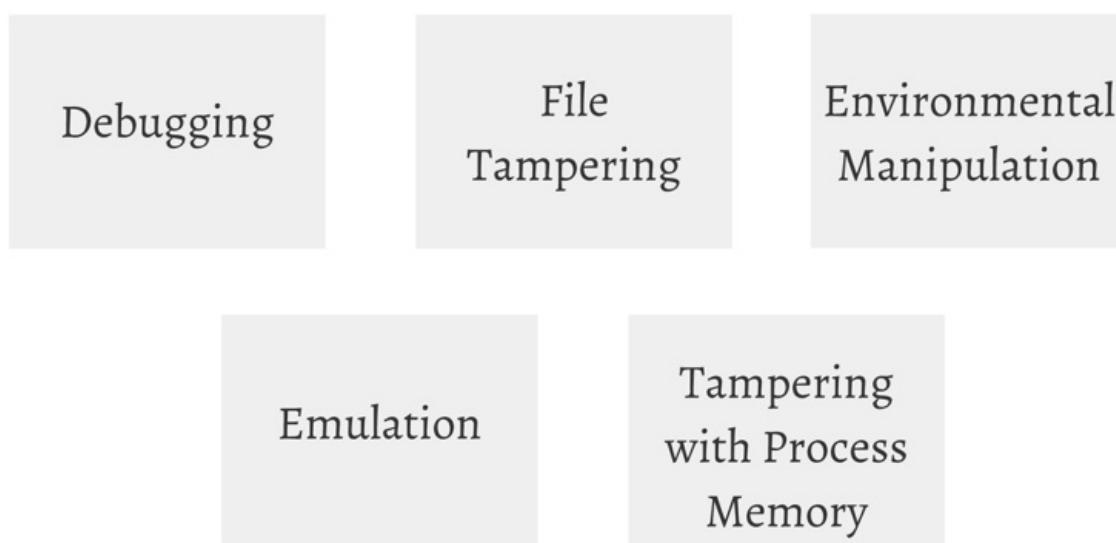
## Overall Effectiveness of Programmatic Defenses

The main motto in anti-reversing is **the sum is greater than its parts**. The defender wants to make it as difficult as possible to get a first foothold for an analysis. They want the adversary to throw the towel before they even get started! Because once the adversary does get started, it's usually only a matter of time before the house of card collapses.

To achieve this deterrent effect, one needs to combine a multitude of defenses, preferably including some original ones. The defenses need to be scattered throughout the app, but also work together in unison to create a greater whole. In the following sections, we'll describe the main criteria that contribute to the effectiveness of programmatic defenses.

### Coverage

--[ TODO ] --



8.1 The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.

8.2: The app implements prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered (Android: JDWP and ptrace, iOS: Mach IPC and ptrace).

8.3: The app detects, and responds to, tampering with executable files and critical data within its own container.

8.4: The app detects the presence of widely used reverse engineering tools and frameworks that support code injection, hooking, instrumentation and debugging.

8.5: The app detects, and responds to, being run in an emulator.

8.6: The app continually verifies the integrity of critical code and data structures within its own memory space.

## Amount and Diversity

--[ TODO ] --

As a general rule of thumb, at least two to three defensive controls should be implemented for each category. These controls should operate independently of each other, i.e. each control should be based on a different technique, operate on a different API Layer, and be located at a different location in the program (see also the criteria below). The adversary should not be given opportunities to kill multiple birds with the same stone - ideally, they should be forced to use multiple stones per bird.

8.7 The app implements multiple mechanisms to fulfill requirements 8.1 to 8.6. Note that resilience scales with the amount, diversity of the originality of the mechanisms used.

8.8 The detection mechanisms trigger different responses, including stealthy ones that don't simply terminate the app.

8.10: Obfuscating transformations and functional defenses are interdependent and well-integrated throughout the app.

## Originality

The effort required to reverse engineer an application highly depends on how much information is initially available to the adversary. This includes information about the functionality being reversed as well as knowledge about the obfuscation and anti-tampering techniques used by the target application. Therefore, the level of innovation that went into designing anti-reversing tricks is an important factor.

Adversaries are more likely to be familiar with ubiquitous techniques that are repeatedly documented in reverse engineering books, papers, presentations and tutorials. Such tricks can either be bypassed using generic tools or with little innovation. In contrast, a secret trick that hasn't been presented anywhere can only be bypassed by a reverser who truly understands the subject, and may force them to do additional research and/or scripting/coding.

Defenses can be roughly categorized into the following categories in terms of originality:

- Standard API: The feature relies on APIs that are specifically meant to prevent reverse engineering. It can be bypassed easily using generic tools.
- Widely known: A well-documented and commonly used technique is used. It can be bypassed using commonly available tools with a moderate amount of customization.
- Proprietary: The feature is not commonly found in reversing resources and research papers, or a known technique has been sufficiently extended / customized to cause significant effort for the reverse engineer.

## API Layer

Generally speaking, the less your mechanisms relies on operating system APIs to work, the more difficult it is to discover and bypass. Also, lower-level calls are more difficult to defeat than higher level calls. To illustrate this, let's have a look at a few examples.

As you have learned in the

```
#define PT_DENY_ATTACH 31

void disable_gdb() {
 void* handle = dlopen(0, RTLD_GLOBAL | RTLD_NOW);
 ptrace_ptr_t ptrace_ptr = dlsym(handle, "ptrace");
 ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
 dlclose(handle);
}
```

```

void disable_gdb() {

 asm(
 "mov r0, #31\n\t" // PT_DENY_ATTACH
 "mov r1, #0\n\t"
 "mov r2, #0\n\t"
 "mov ip, #26\n\t" // syscall no.
 "svc 0\n"
);
}

```

```

struct VT_JdwpAdbState *vtable = (struct VT_JdwpAdbState *)dlsym(lib, "_ZTVN3art4JDWP
12JdwpAdbStateE");

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);

```

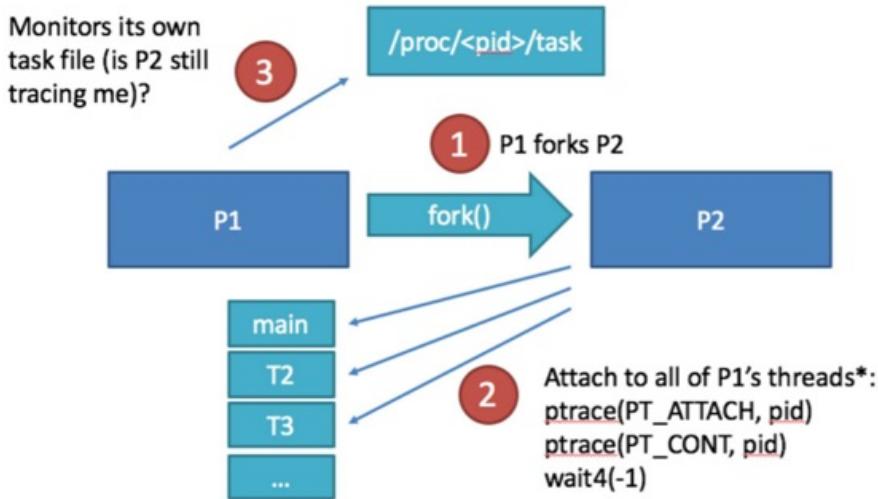
- System library: The feature relies on public library functions or methods.
- System call: The anti-reversing feature calls directly into the kernel.
- Self-contained: The feature does not require any library or system calls to work.

## Parallelism

Debugging and disabling a mechanism becomes more difficult when multiple threads or processes are involved.

- Single thread
- Multiple threads or processes

--[ TODO - description and examples ] --



## Response

Less is better in terms of information given to the adversary. This principle also applies to anti-tampering controls: A control that reacts to tampering immediately in a visible way is more easily discovered than a control that triggers some kind of hidden response with no apparent immediate consequences. For example, imagine a debugger detection mechanism that displays a message box saying "DEBUGGER DETECTED!" in big, red, all-caps letters. This gives away exactly what has happened, plus it gives the reverse engineer something to look for (the code displaying the messagebox). Now imagine a mechanism that quietly changes modifies function pointer when it detects a debugger, triggering a sequence of events that leads to a crash later on. This makes the reverse engineering process much more painful.

The most effective defensive features are designed to respond in stealth mode: The attacker is left completely unaware that a defensive mechanism has been triggered. For maximum effectiveness, we recommend mixing different types of responses including the following:

- Feedback: When the anti-tampering response is triggered, an error message is displayed to the user or written to a log file. The adversary can immediately discern the nature of the defensive feature as well as the time at which the mechanism was triggered.
- Indiscernible: The defense mechanism terminates the app without providing any error details and without logging the reason for the termination. The adversary does not learn information about the nature of the defensive feature, but can discern the approximate time at which the feature was triggered.
- Stealth: The anti-tampering feature either does not visibly respond at all to the detected tampering, or the response happens with a significant delay.

See also MASVS V8.8: "The app implements multiple different responses to tampering, debugging and emulation, including stealthy responses that don't simply terminate the app."

## Scattering

--[ TODO ] --

## Integration

--[ TODO ] --

# Assessing Obfuscation

The simplest way of making code less comprehensible is stripping information that is meaningful to humans, such as function and variable names. Many more intricate ways have been invented by software authors - especially those writing malware and DRM systems - over the past decades, from encrypting portions of code and data, to self-modifying and self-compiling code.

A standard implementation of a cryptographic primitive can be replaced by a network of key-dependent lookup tables so the original cryptographic key is not exposed in memory ("white-box cryptography"). Code can be into a secret byte-code language that is then run on an interpreter ("virtualization"). There are infinite ways of encoding and transforming code and data!

Things become complicated when it comes to pinpointing an exact academical definition. In [an often cited paper](#), Barak et. al describe the black-box model of obfuscation. The black-box model considers a program  $P'$  obfuscated if any property that can be learned from  $P'$  can also be obtained by a simulator with only oracle access to  $P$ . In other words,  $P'$  does not reveal anything except its input-output behavior. The authors also show that obfuscation is impossible given their own definition by constructing an un-obfuscable family of programs.

Does this mean that obfuscation is impossible? Well, it depends on what you obfuscate and how you define obfuscation. Barack's result only shows that *some* programs cannot be obfuscated - but only if we use a very strong definition of obfuscation. Intuitively, most of us know from experience that code can have differing amounts of intelligibility and that understanding the code becomes harder as code complexity increases. Often enough, this happens unintentionally, but we can also observe that implementations of obfuscators exist and are [more or less successfully used in practice](#) %20QoP%2039%20.pdf "Towards Experimental Evaluation of Code Obfuscation Techniques").

Unfortunately, researchers don't agree on whether obfuscation effectiveness can ever be proven or quantified, and there are no widely accepted methods of doing it. In the following sections, we provide a taxonomy of commonly used types of obfuscation. We then outline the requirements for achieving what we would consider *robust* obfuscation, given the deobfuscation tools and research available at the time of writing. Note however that the field is rapidly involving, so in practice, the most recent developments must always be taken into account.

## Obfuscation Controls in the MASVS

The [MASVS](#) lists only two requirements that deal with obfuscation. The first requirement is V8.12:

8.12 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.

This requirement simply says that the code should be made to look fairly incomprehensible to someone inspecting it in a common disassembler or decompiler. This can be achieved by doing a combination of the following.

### Stripping information

The first simple and highly effective step involves stripping any explanatory information that is meaningful to humans, but isn't actually needed for the program to run. Debugging symbols that map machine code or byte code to line numbers, function names and variable names are obvious examples.

For instance, class files generated with the standard Java compiler include the names of classes, methods and fields, making it trivial to reconstruct the source code. ELF and Mach-O binaries have a symbol table that contains debugging information, including the names of functions, global variables and types used in the executable.

Stripping this information makes a compiled program less intelligible while fully preserving its functionality. Possible methods include removing tables with debugging symbols, or renaming functions and variables to random character combinations instead of meaningful names. This process sometimes reduces the size of the compiled program and doesn't affect its runtime behavior.

### Packing, encryption, and other tricks

In addition to stripping information, there's many ways of making apps difficult and annoying to analyze, such as:

- Splitting up code and data between Java bytecode and native code;

- Encrypting strings;
- Encrypting parts of the code and data within the program;
- Encrypting whole binary files and class files.

This kind of transformations are "cheap" in the sense that they don't add significant runtime overhead. They form a part of every effective software protection scheme, no matter the particular threat model. The goal is simply to make it hard to understand what is going on, adding to the overall effectiveness of the protections. Seen in isolation, these techniques are not highly resilient against manual or automated de-obfuscation.

The second requirement, V8.13, deals with cases where obfuscation is meant to perform a specific function, such as hiding a cryptographic key, or concealing some portion of code that is considered sensitive.

8.13: 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 should be preferred over obfuscation whenever possible.

This is where more "advanced" (and often controversial) forms of obfuscation, such as white-box cryptography, come into play. This kind of obfuscation is meant to be truly robust against both human and automated analysis, and usually increases the size and complexity of the program. The methods aim to hide the semantics of a computation by computing the same function in a more complicated way, or encoding code and data in ways that are not easily comprehensible.

A simple example for this kind of obfuscations are opaque predicates. Opaque predicates are redundant code branches added to the program that always execute the same way, which is known a priori to the programmer but not to the analyzer. For example, a statement such as if  $(1 + 1) = 1$  always evaluates to false, and thus always result in a jump to the same location. Opaque predicates can be constructed in ways that make them difficult to identify and remove in static analysis.

Other obfuscation methods that fall into this category are:

- Pattern-based obfuscation, when instructions are replaced with more complicated instruction sequences
- Control flow obfuscation
- Control flow flattening
- Function Inlining
- Data encoding and reordering
- Variable splitting

- Virtualization
- White-box cryptography

## Obfuscation Effectiveness

To determine whether a particular obfuscation scheme is effective depends on the exact definition of "effective". If the purpose of the scheme is to deter casual reverse engineers, a mixture of cost-efficient tricks is sufficient. If the purpose is to achieve a level of resilience against advanced analysis performed by skilled reverse engineers, the scheme must achieve the following:

1. Potency: Program complexity must be increased by a sufficient amount to significantly impede human/manual analysis. Note that there is always a trade off between complexity and size and/or performance.
2. Resilience against automated program analysis. For example, if the type of obfuscation is known to be "vulnerable" to concolic analysis, the scheme must include transformations that cause problems for this type of analysis.

## General Criteria

--[ TODO - describe effectiveness criteria ] --

### Increase in Overall Program Complexity

--[ TODO ] --

### Difficulty of CFG Recovery

--[ TODO ] --

### Resilience against Automated Program Analysis

--[ TODO ] --

## The Use of Complexity Metrics

--[ TODO - what metrics to use and how to apply them] --

## Common Transformations

--[ TODO - describe commonly used schemes, and criteria associated with each scheme. e.g., white-box must incorporate X to be resilient against DFA, etc.] --

### Control-flow Obfuscation

--[ TODO ] --

### **Polymorphic Code**

--[ TODO ] --

### **Virtualization**

--[ TODO ] --

### **White-box Cryptography**

--[ TODO ] --

## **Background and Caveats**

--[ TODO ] --

## **Academic Research on Obfuscation Metrics**

[Colberg et al.](#) introduce potency as an estimate of the degree of reverse engineering difficulty. A potent obfuscating transformation is any transformation that increases program complexity. Additionally, they propose the concept of resilience which measures how well a transformation holds up under attack from an automatic de-obfuscator. The same paper also contains a useful taxonomy of obfuscating transformations.

Potency can be estimated using a number of methods. [Anaekart et al.](#) apply traditional software complexity metrics to a control flow graphs generated from executed code. The metrics applied are instruction count, cyclomatic number (i.e. number of decision points in the graph) and knot count (number of crossing in a function's control flow graph). Simply put, the more instructions there are, and the more alternate paths and less expected structure the code has, the more complex it is.

[Jacubowsky et al.](#) use the same method and add further metrics, such as number of variables per instruction, variable indirection, operational indirection, code homogeneity and dataflow complexity. Other complexity metrics such as [N-Scope](#), which is determined by the nesting levels of all branches in a program, can be used for the same purpose.

All these methods are more or less useful for approximating the complexity of a program, but they don't always accurately reflect the robustness of the obfuscating transformations. [Tsai et al.](#) attempt to remediate this by adding a distance metric that reflects the degree of difference between the original program and the obfuscated program. Essentially, this metric

captures how the obfuscated call graph differs from the original one. Taken together, a large distance and potency is thought to be correlated to better robustness against reverse engineering.

In the same paper, the authors also make the important observation is that measures of obfuscation express the relationship between the original and the transformed program, but are unable to quantify the amount of effort required for reverse engineering. They recognize that these measure can merely serve as heuristic, general indicators of security.

Taking a human-centered approach, [Tamada et al.](#) describe a mental simulation model to evaluate obfuscation. In this model, the short-term memory of the human adversary is simulated as a FIFO queue of limited size. The authors then compute six metrics that are supposed to reflect the difficulty encountered by the adversary in reverse engineering the program. Nakamura et. al. propose similar metrics reflecting the cost of mental simulation.

More recently, [Rabih Mosen and Alexandre Miranda Pinto proposed](#) the use of a normalized version of Kolmogorov complexity as a metric for obfuscation effectiveness. The intuition behind their approach is based on the following argument: if an adversary fails to capture some patterns (regularities) in an obfuscated code, then the adversary will have difficulty comprehending that code: it cannot provide a valid and brief, i.e., simple description. On the other hand, if these regularities are simple to explain, then describing them becomes easier, and consequently the code will not be difficult to understand. The authors also provide empirical results showing that common obfuscation techniques managed to produce a substantial increase in the proposed metric. They found that the metric was more sensitive than Cyclomatic measure at detecting any increase in complexity comparing to original unobfuscated code.

This makes intuitive sense and even though it doesn't always hold true, the Kolmogorov complexity metric appears to be useful to quantify the impact of control flow and data obfuscation schemes that add random noise to a program.

## Experimental Data

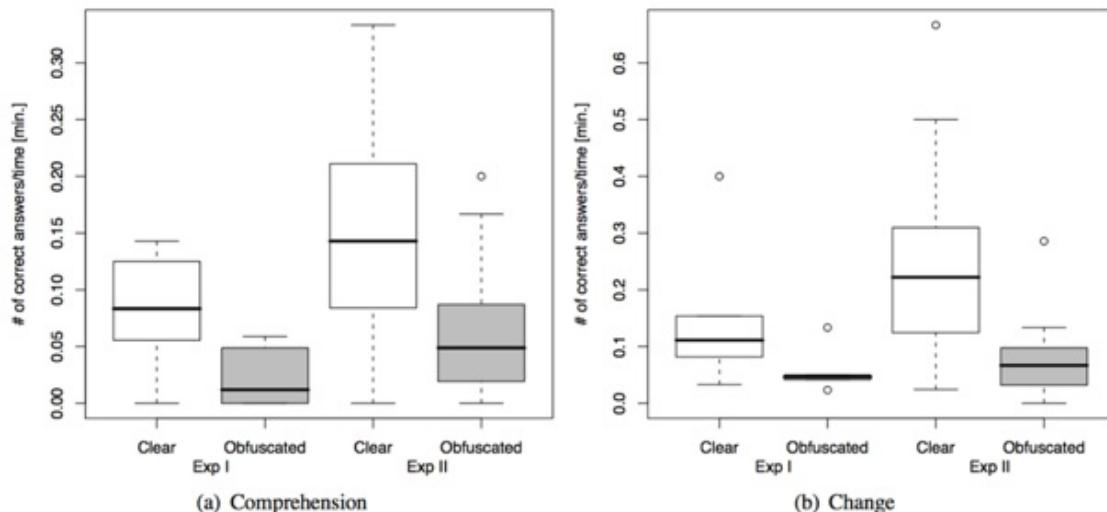
With the limitations of existing complexity measures in mind we can see that more human studies on the subject would be helpful. Unfortunately, the body of experimental research is relatively small - in fact, [the lack of empirical studies](#) is one of the main issues researchers face. There are however some interesting papers linking some types of obfuscation to higher reverse engineering difficulty.

[Nakamura et al.](#) performed an empirical study to investigate the impact of several novel cost metrics proposed in the same paper. In the experiment, twelve subjects were asked to mentally execute two different versions (with varying complexity) of three Java programs. At specific times during the experiment, the subjects were required to describe the program

state (i.e., values of all variables in the program). The accuracy and speed of the participants in performing the experiment was then used to assess the validity of the proposed cost metrics. The results demonstrated that the proposed complexity metrics (some more than others) were correlated with the time needed by the subjects to solve the tasks.

[Sutherland et al.](#) examine a framework for collecting reverse engineering measurement and the execution of reverse engineering experiments. The researchers asked a group of ten students to perform static analysis and dynamic analysis on several binary programs and found a significant correlation between the skill level of the students and the level of success in the tasks (no big surprise there, but let's count it as preliminary evidence that luck alone won't get you far in reverse engineering).

In a series of [controlled%20QoP%2039%20.pdf](#) "Towards Experimental Evaluation of Code Obfuscation Techniques") [experiments](#), M. Ceccato et al. tested the impact of identifier renaming and opaque predicates to increase the effort needed for attacks. In these studies, Master and PhD students with a good knowledge of Java programming were asked to perform understanding tasks or change tasks on the decompiled (either obfuscated or clear) client code of client-server Java applications. The experiments showed that obfuscation reduced the capability of subjects to understand and modify the source code. Interestingly, the results also showed that the presence of obfuscation reduced the gap between highly skilled attackers and low skilled ones: The highly skilled attackers were significantly faster in analyzing the clear source code, but the difference was smaller when analyzing the obfuscated version. Among other results, identifier renaming [was shown](#) to at least double the time needed to complete a successful attack.



*Boxplot of attack efficiency from the Ceccato et. al. experiment to measure the impact of identifier renaming on program comprehension. Subjects analyzing the obfuscated code gave less correct answers per minute.*

## The Device Binding Problem

In many cases it can be argued that obfuscating some secret functionality misses the point, as for all practical purposes, the adversary does not need to know all the details about the obfuscated functionality. Say, the function of an obfuscated program is to take an input value and use it to compute an output value in an indiscernible way (for example, through a cryptographic operation with a hidden key). In most scenarios, the adversary's goal would be to replicate the functionality of the program – i.e. computing the same output values on a system owned by the adversary. Why not simply copy and re-use whole implementation instead of painstakingly reverse engineering the code? Is there any reason why the adversary needs to look inside the black-box?

This kind of attack is known as code lifting and is commonly used for breaking DRM and [white-box cryptographic implementations](#). For example, an adversary aiming to bypass digital media usage could simply extract the encryption routine from a player and include it in a counterfeit player, which decrypts the digital media without enforcing the contained usage policies [TODO](#). Designers of white-box implementations have to deal with [another issue](#): one can convert an encryption routine into a decryption routine without actually extracting the key.

Protected applications must include measures against code lifting to be useful. In practice, this means binding the obfuscated functionality to the specific environment (hardware, device or client/server infrastructure) in which the binary is executed. Preferably, the protected functionality should execute correctly only in the specific, legitimate computing environment. For example, an obfuscated encryption algorithm could generate its key (or part of the key) using [data collected from the environment](#). Techniques that tie the functionality of an app to specific hardware are known as device binding.

Even so, it is relatively easy (as opposed to fully reverse engineering the black-box) to monitor the interactions of an app with its environment. In practice, simple hardware properties such as the IMEI and MAC address of a device are often used to achieve device binding. The effort needed to spoof these environmental properties is certainly lower than the effort required for understanding the obfuscated functionality.

What all this means is that, for most practical purposes, the security of an obfuscated application is only as good as the device binding it implements. For device binding to be effective, specific characteristics of the system or device must be deeply intertwined with the various obfuscation layers, and these characteristics must be determined in stealthy ways (ideally, by reading content directly from memory). Advanced device binding methods are often deployed in DRM and malware and [some research](#) has been published in this area.



# 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.

## 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.

## 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.

## 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, Mac OS 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](<https://github.com/CalebFenton/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

## Dynamic and Runtime Analysis

- [Cydia Substrate](#) - Cydia Substrate for Android enables developers to make changes to existing software with Substrate extensions that are injected into 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 SSL Pinning

- [Xposed Module: Just Trust Me](#) - Xposed Module to bypass SSL certificate pinning.
- [Xposed Module: SSLUnpinning](#).
- [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] ([https://github.com/limneos/weak\\_classdump](https://github.com/limneos/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

- [cscript](#) - Cycript 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.
- [keychaindumper](#) - 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](#)) 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 tools 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 captures 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>

### 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>
- Skakununny, 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>