# 



# Mobile Application Security Testing Guide

[date]

# Foreword

**Welcome to the work-in-progress version of the OWASP Mobile Security Testing Guide. Feel free to explore the exiting 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/>

*-- TODO [Write a clever and inspriring foreword that, while entertaining, also teaches the reader a valuable lesson. Will be added when the whole guide is finished ...] --*

**script kiddie**

*noun informal derogatory*

"a person who uses existing computer scripts or codes to hack into computers, lacking the expertise to write their own."

**The first rule of the OWASP Mobile Security Testing Guide is: Don't just follow the OWASP Mobile Security Testing Guide.**

# 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 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 Testing Guide, please join the discuss the MASVS or MSTG join 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 from based on the [contribution log](https://github.com/OWASP/owasp-mstg/graphs/contributors). For more details, see the [GitHub Repository README](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.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Project Leaders and Authors | Co-Authors | Top Contributors | Contributors | Reviewers |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bernhard Mueller, Sven Schleier | Romuald Szkudlarek, Francesco Stillavato, Pawel Rzepa, Abdessamad Temmar, Slawomir Kosowski | Jin Kung Ong, Alexander Antukh, Gerhard Wagner, Ryan Teoh, Daniel Ramirez Martin, Claudio André, Prathan Phongthiproek, Luander Ribeiro | Michael Helwig, Pishu Mahtani, Stefan Streichsbier, Ben Actis, Anatoly Rosencrantz, Ali Yazdani, Sebastian Banescu, Prabhant Singh, *Romantic668*, Stephen Corbiaux, *Demonbensa*, Jeroen Willemsen | Anant Shrivastava, Stephanie Vanroelen |

### 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 Mobile Security Testing Guide

Technological revolutions can happen quickly. Less than a decade ago, smartphones were clunky devices with little keyboards - expensive playthings for tech-savvy business users. Today, smartphones are an essential part of our lives. We've come to rely on them for information, navigation and communication, and they are ubiquitous both in business and in our social lives.

Apps running on those devices store our personal information, pictures, recordings, notes, account data, business information, location and much more. They act as clients that connect us to services we use on a daily basis, and as communications hubs that processes each and every message we exchange with others. Compromise a person's smartphone and you get unfiltered access to that person's life. When we consider that mobile devices are more readily lost or stolen and mobile malware is on the rise, the need for data protection becomes even more apparent.

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.

## Mobile Application Threats

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. Also leaving a device unattended, which allows adversaries temporary physical access (Evil-Maid attack) can already lead to full compromise of the device or steal data without the owner noticing it.

From the view of a mobile app, this means that 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. Here however we encounter another problem: Much depends on the device and operating system the app is running on, as well as its configuration. Is the keychain locked with a passcode? What if the device doesn't offer hardware-backed secure storage, as is the case with some Android devices? Can and should the app even verify this, or is it the responsibility of the user?

Another key difference to their more stationary cousins is that 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 - those baddies use whatever works).

## OWASP Mobile Top 10 2016

The OWASP Mobile Top 10 is the equivalent counterpart of the OWASP Top Ten Project, but is specifically designed to focus on the mobile application security. Most of the time, folks in the information security industry discuss about the "OWASP Top Ten" project but in fact, they are only referring to the web application security.

In this guide, we bring to your attention about its equivalent counterpart, the OWASP Mobile Top 10 2016, which is essentially an awareness document for mobile application security.

The OWASP Mobile Top 10 represents a broad consensus about what are the most critical mobile application security flaws identified in the actual mobile applications, derived as per the raw data obtained from various different vendors and consultants in the information security industry.

The following are the OWASP Mobile Top 10:

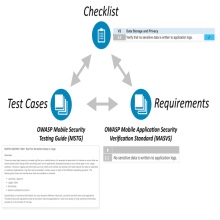
* M1 - Improper Platform Usage[1](https://github.com/pillfill/hiding-passwords-android/)
* Misuse of a mobile platform feature or failure to use platform security controls adequately
* Scope of coverage includes Android intents, platform permissions, misuse of TouchID, the Keychain, or some other security control that is part of the mobile operating system
* Some examples includes the violation of published guidelines, violation of convention or common practice, and any unintentional misuse
* M2 - Insecure Data Storage[2](https://developer.android.com/reference/java/security/KeyStore.html)
* Insufficient protection mechanisms towards user or app data stored locally in the mobile devices
* Scope of coverage includes an adversary that has attained a lost or stolen mobile device, malware or a repackaged app acting on the adversary's behalf that executes on the mobile device
* Data insecurely stored includes files such as SQLite databases, log files, XML files and cookies
* M3 - Insecure Communication[3]
* Insufficient protection mechanisms towards user or app data transmitted over the mobile device's carrier network or the internet
* Scope of coverage includes an adversary that shares the same Local Area Network (LAN), network devices or malware; and whether defensive mechanisms such as Certificate Pinning has been implemented in the mobile app
* M4 - Insecure Authentication[4]
* Lack of proper authentication methods and controls
* Scope of coverage includes the exploitation of authentication vulnerabilities like weak password policy
* M5 - Insufficient Cryptography[5]
* Usage of inadequately strong cryptographic standards, or poor crytography implementation and usages
* Scope of coverage includes the cracking of improperly encrypted data through physical access or mobile malware acting on an adversary's behalf
* M6 - Insecure Authorisation[6]
* Lack of proper roles and permissions validation and access rights controls
* Scope of coverage includes the exploitation of the authorization vulnerabilities like insecure direct object references
* M7 - Poor Code Quality[7]
* Insufficient consistency in coding patterns and lack of proper user data input validations and method calls
* Scope of coverage includes any plausible endpoints that can pass untrusted inputs to method calls made within the mobile app's code, resulting in potential exploitation via malware or phishing scams
* M8 - Code Tampering[8]
* Lack of runtime checks function that perform app code integrity checks
* Scope of coverage includes exploitation through code modification via malicious forms of the apps hosted in third-party app stores. Malicious attacker may also trick the user into installing the app via phishing attacks
* M9 - Reverse Engineering[9]
* Missing obfuscation methods
* Scope of coverage includes downloading the mobile app from an app store and analyze it within their own local environment using a suite of different tools to identify potential attack vectors
* M10 - Extraneous Functionality[10]
* Lack of logs and endpoints verification prior to publishing the production builds
* Scope of coverage includes the identification of hidden or extraneous functionality in the backend system or the mobile app itself, and then exploit it directly from their own systems without any involvement by end-users

To read more about the category of vulnerabilities and procedures to prevent them from compromising your mobile application, please refer to the OWASP Mobile Top 2016 Project Page11.

## 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 of 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 Testing Guide

-- TODO [Describe the organization of the current guide] --

## References

* [1](https://github.com/pillfill/hiding-passwords-android/) M1 - Improper Platform Usage - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) M2 - Insecure Data Storage - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage>
* [3] M3 - Insecure Communication - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication>
* [4] M4 - Insecure Authentication - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure_Authentication>
* [5] M5 - Insufficient Cryptography - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M5-Insufficient_Cryptography>
* [6] M6 - Insecure Authorisation - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M6-Insecure_Authorization>
* [7] M7 - Poor Code Quality - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality>
* [8] M8 - Code Tampering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M8-Code_Tampering>
* [9] M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>
* [10] M10 - Extraneous Functionality - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M10-Extraneous_Functionality>
* [11] OWASP Mobile Top 2016 Project Page - <https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10>

# Testing Processes and Techniques

## Mobile App Security Testing Methodology

-- TODO [Describe Mobile Security Testing methodology] --

### Testing Process

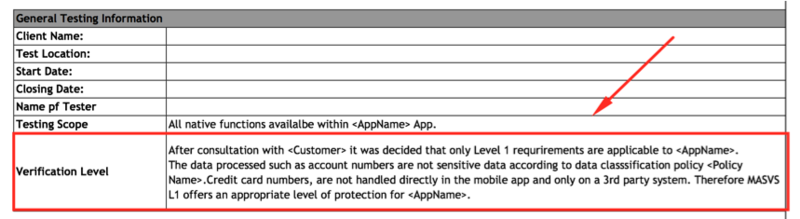
The following sections will show how to use the OWASP mobile application security checklist and testing guide during a security test.

#### Preparation - Defining The Baseline

First of all, you need to decide what security level of the MASVS to test against. The security requirements should ideally have been decided at the beginning of the SDLC - but unfortunately we are not living in an ideal world. At the very least, it is a good idea to walk through the checklist, ideally with an IT security representative of the enterprise, the app stakeholders of the project and make a reasonable selection of Level 2 (L2) controls to cover during the test.

The controls in MASVS Level 1 (L1) are appropriate for all mobile apps - the rest depends on the threat model and risk assessment for the particular app. Discuss with the app stakeholders to understand what are the requirements that are applicable and which are the ones that should be deemed out of scope for the scope of testing, perhaps due to business decisions or company policies. Also consider whether some L2 requirements may be needed due to industry regulations or local laws - for example, 2-factor-authentation (2FA) may be obligatory for a financial app, as enforced by the respective country's central bank and/or financial regulatory authority.

If security requirements were already defined during the SDLC, even better! Ask for this information and document it on the front page of the Excel sheet ("dashboard"). More guidance on the verification levels and guidance on the certification can be found in the [MASVS](https://github.com/OWASP/owasp-masvs).

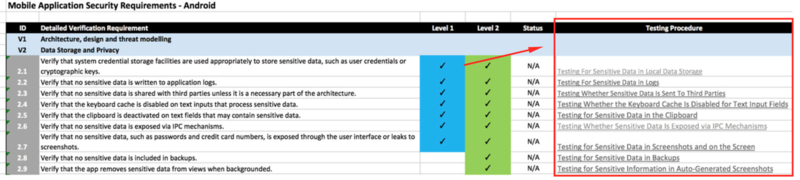


All involved parties need to agree on the decisions made and on the scope in the checklist, as this will present the baseline for all security testing, regardless if done manually or automatically.

#### Mobile App Security Testing

During a manual test, one can simply walk-through the applicable requirements down the checklist, one-by-one. For a detailed testing procedures, simply click on the link provided in the "Testing Procedure" column. These links will bring you to the respective chapter in the OWASP Mobile Security Testing Guide (MSTG), where detailed steps and examples are listed for reference and guidance purposes.

Also important is to note that the OWASP Mobile Security Testing Guide (MSTG) is still "Work In Progress" and being updated even as you are reading this paragraph, therefore, some test cases may not have been written yet or may be in a draft status. (Ideally, if you discover any missing content, you could contribute it yourself).



The status column can have one of the following three different values, that need to be filled out:

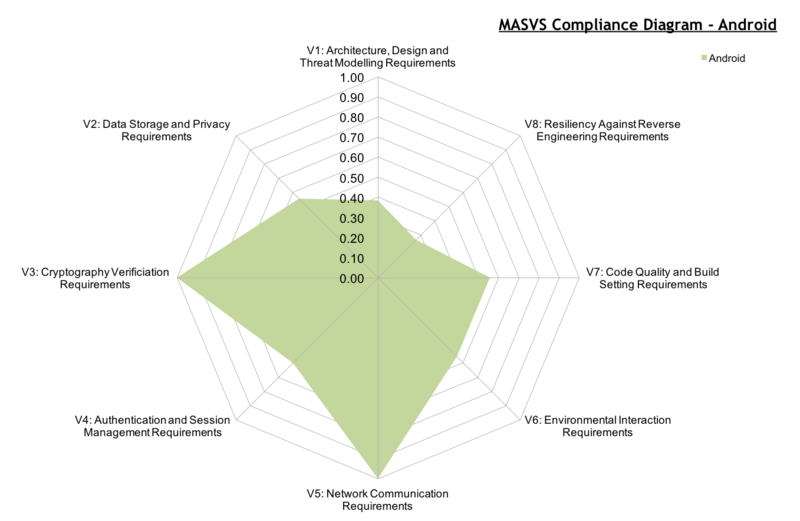
* **Pass:** Requirement is applicable to mobile app and implemented according to best practices.
* **Fail:** Requirement is applicable to mobile app but not fulfilled.
* **N/A:** Requirement is not applicable to mobile app.

#### Reverse Engineering Resiliency Testing

*Resiliency Testing* is a new concept introduced in the OWASP MSTG. This kind of testing is used if the app implements defenses against client-side threats, such as tampering and extracting sensitive information. As we know, such protection is never 100% effective. The goal in resiliency testing is to verify that no glaring weaknesses exist in the protection scheme, and that the expectations as to its effectiveness are met (e.g., a skilled reverse engineer should be forced to invest significant effort to do reach a particular goal).

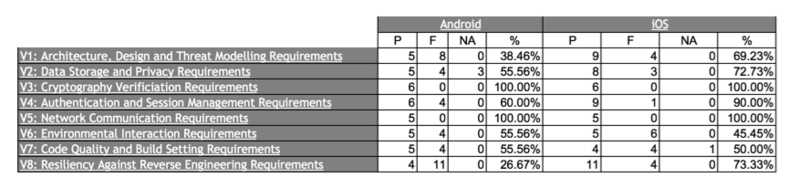
#### The Management Summary

A spider chart is generated on the fly according to the results of the requirements for both supported platforms (Android and iOS) in the "Management Summary" tab. You can use this in your report to point out areas that need improvement, and visualize progress over time.



The spider chart visualizes the ratio of passed and failed requirements in each domain. As can be seen above all requirements in "V3: Cryptography Verification Requirements" were set to "pass", resulting in a value of 1.00. Requirements that are set to N/A are not included in this chart.

A more detailed overview can also be found in the "Management Summary" tab. This table gives an overview according to the eight domains and breaks down the requirements according to it's status (Passed, Failed or N/A). The percentage column is the ratio from passed to failed requirements and is the input for the spider chart described above.



#### Risk Assessment

#### Reporting

## Vulnerability Analysis Techniques

### Static Analysis

In a Static Analysis approach, the developers must provide the source code or compiled IPA/APK files of the mobile application for programmatic analysis. The source code will be analyzed to ensure that there is sufficient and correct implementation of security controls, specifically on crucial components such as the authentication, authorization, session management and data storage mechanisms.

##### Pros of Static Analysis

* Great scalability, able to run on lots of mobile applications and can be easily repeated
* Great at identifying standard security vulnerabilities such as SQL injection flaws and etc.

##### Cons of Static Analysis

* May require access to the source code
* High number of false positives
* Unable to identify issues related to operational deployment environments

#### Automatic Code Analysis

In automatic code analysis, the tool will check the source code for compliance with a predefined set of rules or industry's best practices. It has been a standard development practice to use analytical methods to review and inspect the mobile application's source code to detect bugs and implementation errors.

The automatic code analysis tools will provide assistance with the manual code review and inspection process. The tool will typically display a list of warnings, identified through comparing the source code content with its own predefined set of rules or industry's best practices, and then flag all the instances which contains any forms of violations in terms of their programming standards. An automatic code analysis tool can also provide an automated or a programmer-assisted way to correct the issues found.

Some static code analysis tools encapsulate deep knowledge of the underlying rules and semantics required to perform the specific type of analysis, such that it does not require the code reviewer to have the same level of expertise as an expert. Many Integrated Development Environments (IDE) also provide basic automated code review functionality, to provide assistance in improving the security mechanisms implementation code in the mobile applications.

In the role of a penetration testing engagement, the use of automatic code analysis tools can be very handy as it could quickly and easily provide a first-level analysis of source code, to identify the low hanging fruits before diving deeper into the more complicated functions, where it is essential to thoroughly assess the method of implementation in varying contexts.

##### Open Source Static Analysis Tools (non-exhaustive list)

* [FindBugs](http://findbugs.sourceforge.net/) (Java)
* [PMD](https://pmd.github.io/) (Java)
* [VisualCodeGrepper](https://sourceforge.net/projects/visualcodegrepp/) (C/C++, C#, VB, PHP, Java and PL/SQL)
* [Agnitio](https://sourceforge.net/projects/agnitiotool/) (Objective-C, C#, Java & Android)

##### Commercial

* [Veracode](https://www.veracode.com/products/binary-static-analysis-sast) (Java, Objective-C, Swift, etc.)
* [CheckMarx](https://www.checkmarx.com/technology/static-code-analysis-sca/) (Java, Objective-C, Swift, etc.)

#### Manual Code Review

In manual code review, a human code reviewer will be looking through the source code of the mobile application, to identify security vulnerabilities through various techniques such as performing grep command on key words within the source code repository to identify usages of potentially vulnerable code, such as the likes of strcpy() function or equivalent vulnerable code.

The main difference between a manual code review and the use of any automatic code analysis tools is that in manual code review, it is better at identifying vulnerabilities in the business logic, standards violations and design flaws, especially in the situations where the code is technically secure but logically flawed. In such scenarios, the code snippet will not be detected by any automatic code analysis tool as an issue.

##### Crawling Code

Crawling code is the practice of scanning a code base of the review target in question. It is to look for key pointers wherein a possible security vulnerability might reside. In crawling code, it will look for certain APIs that are related to interfacing to the external functions which are key areas for an attacker to focus on.

For example, HTTP Request Strings like request.url or request.files, HTML Output like innerHtml or HtmlUtility, or even Database related codes like executeStatement or executeQuery are key indicators which are of interest in the process of crawling code.

##### Pros of Manual Code Review

* Proficient in deep diving into the various code paths to check for logical errors and flaws in the mobile application's design and architecture where automated analysis tools are not able to identify
* Great at detecting security issues like authorization, authentication and data validation as compared to automated code analysis tools

##### Cons of Manual Code Review

* Difficult for human code reviewers to identify subtle mistakes such as buffer overflows
* Requires expert human code reviewer who are proficient in both the language and the frameworks used in the mobile application, as it is essential to have a deep understanding of the security implmentation of the technologies used in the mobile application's source code
* Time consuming, slow and tedious; especially when mobile application source code nowadays usually has so many functionalities that it has a large number of lines of code (e.g. over 10K, or sometimes even over 100K)

### Dynamic Analysis

In a Dynamic Analysis approach, the focus is on the testing and evaluation of a program by executing data in a real-time manner, under different stimuli. The main objective of a dynamic analysis is to find the security vulnerabilities or weak spots in a program while it is running. Dynamic analysis is conducted against the backend services and APIs of mobile applications, where its request and response patterns would be analysed.

Usually, dynamic analysis is performed to check whether there are sufficient security mechanisms being put in place to prevent data validation attacks (e.g. cross-site scripting, SQL injection, etc.) and server configuration errors or version issues.

#### Pros of Dynamic Analysis

* Does not require to have access to the source code
* Does not need to understand how the mobile application is supposed to behave
* Able to identify infrastructure, configuration and patch issues that Static Analysis approach tools will miss

#### Cons of Dynamic Analysis

* Limited scope of coverage because the mobile application must be footprinted to identify the specific test area
* No access to the actual instructions being executed, as the tool is exercising the mobile application and conducting pattern matching on the requests and responses

#### Runtime Analysis

-- TODO [Describe Runtime Analysis : goal, how it works, kind of issues that can be found] --

#### 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](https://github.com/intrepidusgroup/mallory)
* [Wireshark](https://www.wireshark.org/)

#### Input Fuzzing

-- TODO [Write a better description, or remove this] --

Fuzz testing, is a method for testing software input validation by feeding it intentionally malformed input.  
Steps in fuzzing

* Identifying a target
* Generating malicious inputs
* Test case delivery
* Crash monitoring

[OWASP Fuzzing guide](https://www.owasp.org/index.php/Fuzzing)

Note: Fuzzing only detects software bugs. Classifying this issue as a security flaw requires further analysis by the researcher.

### Vulnerability Analysis Considerations

#### Eliminating False Positives

##### Cross-Site Scripting (XSS)

A typical reflected XSS attack is executed by sending a URL to the victim(s), which for example can contain a payload to connect to some exploitation framework like BeeF [2](https://developer.android.com/reference/java/security/KeyStore.html). When clicking on it a reverse tunnel is established with the Beef server in order to attack the victim(s). As a WebView is only a slim browser it is not possible for a user to insert a URL into a WebView of an App as no adress bar is available. Also clicking on a link will not open the URL in a WebView of an App, instead it will open directly within the browser of Android. Therefore a typical reflected Cross-Site Scripting attack that targets a WebView in an App is not applicable and will not work.

If an attacker finds a stored Cross-Site Scripting vulnerability in an endpoint, or manages to get a Man-in-the-middle (MITM) position and injects JavaScript into the response, then the exploit will be sent back within the response. The attack will then be executed directly within the WebView. This can become dangerous in case:

* JavaScript is not deactivated in the WebView (see OMTG-ENV-005)
* File access is not deactivated in the WebView (see OMTG-ENV-006)
* The function addJavascriptInterface() is used (see OMTG-ENV-008)

As a summary reflected XSS is no concern for a mobile App, but stored XSS or injected JavaScript through MITM can become a dangerous vulnerability if the WebView in use is configured insecurely.

##### Cross-Site Request Forgery (CSRF)

The same problem described with reflected XSS also applied to CSRF attacks. A typical CSRF attack is executed by sending a URL to the victim(s) that contains a state changing request like creation of a user account of triggering a financial transaction. As a WebView is only a slim browser it is not possible for a user to insert a URL into a WebView of an App and also clicking on a link will not open the URL in a WebView of an App. Instead it will open directly within the browser of Android. Therefore a typical CSRF attack that targets a WebView in an App is not applicable.

The basis for CSRF attacks, access to session cookies of all browser tabs and attaching them automatically if a request to a web page is executed is not applicable on mobile platforms. This is the default behaviour of full blown browsers. Every App has, due to the sandboxing mechanism, it's own web cache and stores it's own cookies, if WebViews are used. Therefore a CSRF attack against a mobile App is by design not possible as the session cookies are not shared with the Android browser.

Only if a user logs in by using the Android browser (instead of using the mobile App) a CSRF attack would be possible, as then the session cookies are accessible for the browser instance.

## 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 this 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 resiliency 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 resiliency assessment, the tester assumes the role of the reverse engineer and attempts to bypass the defenses.

In this guide, we'll cover basic tampering techniques such as patching and hooking, as well as common tools and processes for reverse engineering (and comprehending) mobile apps without access to the original source code. Reverse engineering is an immensely complex topic however - covering every possible aspect would easily fill several books. Links and pointers to useful resources are included in the "references" section at the end of each chapter.

### Before You Start

Before you 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 re-sign the app, or disable the default code signature verification facilities to run modified code.

#### Code Injection

Code injection is a very powerful technique that allows you to explore and modify processes 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 in 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"](https://github.com/nowsecure/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 [did not go that down that well with Saurik](https://www.reddit.com/r/ReverseEngineering/comments/50uweq/cycript_on_steroids_pumping_up_portability_and/).

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 Yampering" 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 lifecycle. 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 Yampering" 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 workflow 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](http://valgrind.org/docs/manual/dist.readme-android.html) 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 [4] and Angr [5].

##### 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 *conc*rete and symb*olic*), 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) [6].

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

#### Domain-Specific De-Obfuscation Attacks

-- TODO [Describe de-obfucscation of virtual machines and whiteboxes] --

### References

* [1](https://github.com/pillfill/hiding-passwords-android/) OWASP Mobile Application Security Verification Standard - <https://www.owasp.org/images/f/f2/OWASP_Mobile_AppSec_Verification_Standard_v0.9.2.pdf>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) The Importance of Manual Secure Code Review - <https://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/the-importance-of-manual-secure-code-review>
* [3] OWASP Code Review Introduction - <https://www.owasp.org/index.php/Code_Review_Introduction>
* [4] Radare2 - <https://github.com/radare/radare2>
* [5] Angr - <http://angr.io>
* [6] <https://triton.quarkslab.com/files/csaw2016-sos-rthomas-jsalwan.pdf>

# Testing Application Security on Android

## Android Platform Overview

This chapter is going to introduce Android on the architecture point of view and will provide the reader with detailed information on security mechanisms. Then, it will describe the structure of an Android application and will emphasize on the Inter Process Communication mechanism. Last, the way Android applications are published is explained to the reader.

### Android Architecture and Security Mechanisms

Android is an open source platform that can be found nowadays on many devices:

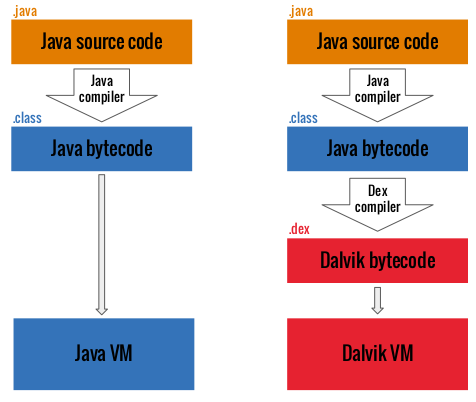
* Mobile Phones and Tablets
* Wearables
* "Smart" devices in general like TVs

It also offers an applicative environment that supports not only pre-installed apps on the device, but also 3rd party apps that can be downloaded from marketplaces like Google Play.

The software stack of Android comprises of different layers, where each layer is defining certain behavior and offering specific services to the layer above.



On the lowest level Android is using the Linux Kernel where the core operating system is built up on. The hardware abstraction layer defines a standard interface for hardware vendors. HAL 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 and then compiled again into the dex format. The dex files are then executed within the Dalvik VM.  
In the next image we 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 the successor of Dalvik VM was introduced, called Android Runtime (ART).

ART, a.k.a Android RunTime, was born with KitKat (Android 4.4). However, it has really been set for general use only in Lollipop (Android 5.0, API 21) in November 2014, where it replaced Dalvik : in KitKat, ART was only available in the 'Developer' menu to those who wanted to try it explicitly. When no user action was done to modify the normal behaviour of the mobile, Dalvik was used.

In Android, apps are executed into their own environnement in a Virtual Machine (VM), that was called Dalvik, located in the RunTime environnement. 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 environnements 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 environnement 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 hardwares 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 was 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 everytime 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 its battery for more time..

#### Android Users and Groups

Android is a system based on Linux, however it does not deal with users the same way Linux does. It does not have a */etc/password* file describing a list of Linux users in the system. Instead Android contains a fixed set of users and groups and they are used to isolate processes and grant permissions.  
File [system/core/include/private/android\_filesystem\_config.h](http://androidxref.com/7.1.1_r6/xref/system/core/include/private/android_filesystem_config.h) shows the complete list of the predefined users and groups mapped to numbers.  
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 \*/  
 ...

### Understanding Android Apps

#### Communication with the Operating System

In Android, apps are developed in Java, and the Operating System offers an API to interact with system resources: communication media (Wifi, Bluetooth, NFC, ...), files, cameras, geolocation (GPS), microphone, ... . System resources cannot be accessed directly, and APIs mediate the access for the user. At the time of writting this guide, the current version of Android API is 7.1.1 Nougat, API 25.

APIs have evolved a lot since Android creation (the first release happened in September 2008). Early versions are not supported anymore; however, Android is a living project and new features and bug fixes are periodically made.

Noteworthy recent 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) in August 2016 (new JIT compiler on ART)

After being developed, apps can be installed on mobiles from a variety of sources: locally through USB, from Google official store (Google Play) or from alternate 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, another way has to 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 phone, 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 (under developer settings) and has to be connected via USB.  
Once USB-Debugging is enabled, the connected devices can be viewed with the command

$ adb devices  
List of devices attached  
BAZ5ORFARKOZYDFA 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, meaning this is the app the developer uploaded to the market.  
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 used to cache app data on runtime including WebView caches.
* **code\_cache**: TBD
* **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 different files for the bytecode, assets, etc. When unzipped the following directory structure can 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 containting 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.

Since some resources inside the APK are compressed using non-standard algorithms (e.g. the AndroidManifest.xml), simply unzipping the file does not reveal all information. A better way is to use the tool apktool to unpack and uncompress the files. The following is a listing of the 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 openend 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 containting 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

When a new app gets installed on Android a new UID is assigned to it. Generally apps are assigned UIDs in the range of 10000 (*AID\_APP*) and 99999. Android apps also receive a user name based on its UID. As an 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, and it also 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](http://androidxref.com/7.1.1_r6/xref/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>

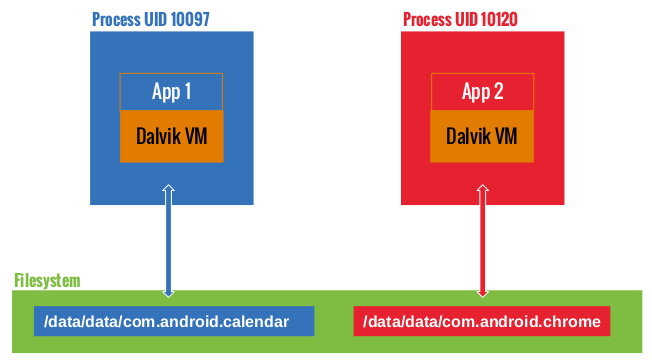
An important element to understand Android security is that all apps have the same level of privileges: both native and third-party apps are built on the same APIs and are run in similar environnements. Also, all apps are executed not as 'root', but with the user level of privileges. That means that, basically, apps cannot perform some actions or access 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 for Python, ...), mobiles need to be rooted.

#### The App Sandbox

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

When installing new apps (From Google Play or External Sources), a new folder is created in the filesystem in the path /data/data/<package name>. 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 to *read* and *write* only to the owner of the app.



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

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

drwx------ 4 u0\_a97 u0\_a97 4096 2017-01-18 14:27 com.android.calendar  
drwx------ 6 u0\_a120 u0\_a120 4096 2017-01-19 12:54 com.android.chrome

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 other data directory.  
An example how this is achieved in Nfc app:

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

The Android Framework is creating an abstraction layer for all the layers below, so developers can implement Android Apps and can utilize the capabilities of Android without deeper knowledge of the layers below. It also offers a robust implementation that offers common security functions like secure IPC or cryptography.

#### 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 Lifecycle

Android apps have their own lifecycles, that is under the control of the operating system. Therefore, apps need to listen to state changes and must be able to 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 not in use since a significant time).

Apps implement several event managers to handle events: for example, the onCreate handler implements what is 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 the 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 the rights required by the app (required permissions are listed, and filters can be implemented to refine the way the app will interact with the outside world). It also contains general metadata about the app, like its icon, its version number and the theme is uses for User Experience (UX). It may also list other information like the APIs it is compatible with (minimal, targeted and maximal SDK version) and the kind of memory is 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), the app version, relevant SDKs, required permissions, exposed content providers, used broadcast receivers with intent filters, and the description of the app itself with 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>

Manifests are text files and can be edited with any text editor. However, Android Studio, Google prefered IDE for Android development, embeds a graphical editor.

A lot more useful options can be added to manifest files. The reader is invited to refer to Android official documentation available at <https://developer.android.com/index.html> for more details: even if numerous examples are given in this section, in no way this guide should be considered as a reference on the topic.

##### Activities

Activities make up the visible part of any app. More specifically, one activity exists per screen (e.g. user interface) in an app: for instance, apps that have 3 different screens implement 3 different activities, where the user can 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 lifecycles and need to listen to system changes to be able to handle them accordingly. Activities can have the following states: active, paused, stopped and inactive. These states are managed by Android operating systems. 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 that situation, default actions are taken. However, usually at least the onCreate manager is overridden by app developers, as this is the place where most user interface components are declared and initialised. onDestroy may be overridden as well in case some resources need to be explicitly released (like network connections or connections to databases) or 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, ...); however, they must be integrated in activities to become useful: fragments cannot exist on their own. They have their own lifecycle, which is tied to the one of the activity that implements them.  
As they have their own lifecycle, 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 Android 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 fragmens.  
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.

##### Intents

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 kinds of intents : explicit and implicit.

* explicit intents exactly name the activity class to be used. 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"));

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 exists and is used to send and receive intents inside an app, without having them sent to the outside world (other apps or 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 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 permissoin 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 certificated 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 AndroidManifes.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 as 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 [9].

##### 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 treats 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* [9]

##### 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 on the security poin 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 you 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 refered 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 Environnements (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.

#### 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.

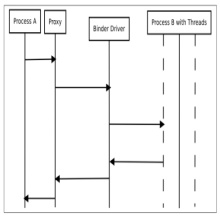
Whereas other vendors may review and approve apps before they are actually published, such things do not happen on Google Play; this way, a short release time can be expected between the moment when the developer starts the publishing process and the moment when the app is available to users.

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>.

### How Apps Communicate - Android IPC

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. A lot of Android system services, as well as all high-level IPC services, depend on Binder.

In the Binder framework, a client-server communication model is used. IPC clients communicate through a client-side proxy. This proxy connects 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. Developers write interfaces for remote services using the Android Interface Descriptor Language (AIDL).

  
*Binder Overview. Image source:* [*Android Binder by Thorsten Schreiber*](https://www.nds.rub.de/media/attachments/files/2011/10/main.pdf)

#### High-Level Abstractions

*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().

There are two types of Intents:

* Explicit intents specify the component to start by name (the fully-qualified class name).
* Implicit intents do not name a specific component, but instead declare a general action to perform, which allows a component from another app to handle it. When you create an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device.

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.

For activities and broadcast receivers, intents are the preferred mechanism for asynchronous IPC in Android. Depending on your app requirements, you might use sendBroadcast(), sendOrderedBroadcast(), or an explicit intent to a specific app component.

A BroadcastReceiver handles asynchronous requests initiated by an Intent.

Using Binder or Messenger is the preferred mechanism for RPC-style IPC in Android. They provide a well-defined interface that enables mutual authentication of the endpoints, if required.

-- TODO [Explain what vulnerabilities can be created while using IPC mechanisms. Give short examples in the form of code snippets] --

Android’s Messenger represents a reference to a Handler that can be sent to a remote process via an Intent

A reference to the Messenger can be sent via an Intent using the previously mentioned IPC mechanism

Messages sent by the remote process via the messenger are delivered to the local handler. Great for efficient call-backs from the service to the client

### References

-- TODO [Numbering and cleanup of references] -

* [Android Security](https://source.android.com/security/)
* [Android Developer: App Components](https://developer.android.com/guide/components/index.html)
* [HAL](https://source.android.com/devices/)
* "Android Security: Attacks and Defenses" By Anmol Misra, Abhishek Dubey
* [AProgrammer Blog](https://pierrchen.blogspot.com.br)
* [keesj Android internals](https://github.com/keesj/gomo)
* [Android Versions] (<https://en.wikipedia.org/wiki/Android_version_history>)
* "Professional Android 4 Application Development" by Reto MEIER
* [9] APK Signing - <https://source.android.com/security/apksigning/>

## Basic Security Testing on Android

### Testing Methods

#### Static Analysis

Static analysis is the act of looking into app components, source code and other resources without actually executing it. This test is focused on finding misconfigured or unprotected Android IPC components, as well as finding programming mistakes such as misuse of cryptography routines, find libraries with known vulnerabilities and even dynamic code loading routines.

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 analyser 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 analysers can otherwise be overwhelming and the effort can become counterproductive if an overly large report need to be manually investigated.

Static Analysis can be divided into two categories, **White box** and **Black box**. The first is when the source code is available and the other is when we only have the compiled application or library. We will now go into more details on each category.

##### With Source Code ("White-Box")

**White box testing** an app is the act of testing an app with the source code available. To accomplish the source code testing, you will want to have a setup similar to the developer. You will need a computer with the Android SDK and an IDE installed. It is also recommended to that you have access to either a physical device or an emulator, so you can debug the app.

Once you have the setup ready and the source code indexed by an IDE (Android Studio is recommended since it is the current IDE of choice by Google), you can start debugging and searching for interesting parts of code.  
Begin by testing each [Android Component](0x05a-Platform-Overview.md#app-components). Check whether they are exported and the enforcing permissions that are in place. Android Lint[15] can help in the identification of such problems.

Any Android component manipulating sensitive data (contacts, location, images, etc.) should be investigated carefully.

Proceed on to testing the libraries the application has embedded: some libraries contain known vulnerabilities and you should check for that. Some of the question you may want to answer are: what libraries are the app using? Which version of the libraries are being used? Do they have any known vulnerability?

Since you have the source code in hand, you can check for crypto mistakes in the implementation. Look for hard coded keys and implementation errors related to cryptography functions. Devknox[16] can help checking most common cryptographic mistakes since it is embedded to the IDE.

##### Without Source Code ("Black-Box")

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[17]), which can be installed on an Android device or reverse engineered with the goal to retrieve parts of the source code.

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. Applications that contain a native library can still be reverse engineered by require low level knowledge and the process is not automated.

More details and tools about the Android reverse engineering topic can be found at [Tampering and Reverse Engineering on Android](0x05b-Reverse-Engineering-and-Tampering.md) section.

Besides reverse engineering, there is a handful of automated tools that perform security analysis on the APK itself searching for vulnerabilities.  
Some of these tools are: QARK[18], Androbugs[19] and JAADAS[20].

#### 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 or 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, Burp Suite Professional or Fiddler 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 made from the endpoint.

#### Network Analysis

A proxy is the bread and butter to any form of security testing that involves a client-server application and that includes mobile application. It can be used to intercept network traffic between a mobile and a back-end server for testing Authorization, Session Management and so on. Some popular proxy tools for security testings are Burp Suite, OWASP ZAP and Charles.

#### Reverse Engineering

There are many reason to reverse engineer an application: to understand application security logic, to identify application secret and so on. More detail coverage on reverse engineering Android application are covered in "Tampering and Reverse Engineering on Android" **(-- TODO [add\_link] --)** page. Some popular tools that will be used are apktool, enjarify and more.

### Setting Up Your Testing Environment

When setting up the testing environment, this can become a challenging task. For example when testing on-site at client premises there might be restrictions when using an enterprise Access Point due to limitations in the connections that can be made (e.g. ports are blocked), making it more difficult to start a dynamic analysis of the app. Rooted phones might also not be allowed within the enterprise network due to companies policies. Also, root detection and other countermeasures implemented within an app can lead to significant extra work just to be able to finally test the app. Either way, the testing team responsible for the Android assessment need to work together with the app developer(s) and operation team in order to find a proper solution for a working testing environment.

This section will give an overview of different methods on how an Android app can be tested and will illustrate also its limitations. Due to the reasons stated above you should be aware of all possible testing methods to select the right one for your testing environment, but also to articulate restrictions so that everybody in the project is on the same page.

#### Preparation

The goal of a test is to verify if the app and the endpoint(s) it's communicating with, are implemented in a secure way. Several security controls like SSL Pinning or root detection might be implemented. These will slow down the testing dramatically and might already take days to bypass, depending on the implementation.

During the preparation phase it should be discussed with the company developing the mobile app, to provide two versions of the app. One app should be built as release to check if the implemented controls like SSL Pinning are working properly or can be easily bypassed. The same app should also be provided as debug build that deactivates certain security controls. Through this approach all scenarios and test cases can be tested in the most efficient way.

This approach needs of course to align with the scope of the engagement and if it's a black box or white box test (-- TODO [Link to section in MSTG describing Black and White Box] --). For a white box test, requesting for a production and debug build 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 it might be already the intention of the client to see what can be done in a certain amount of time with the production app and how effective the implemented security controls are.

Either way, the following items should be discussed with the company developing the mobile app and it should be decided if the implemented security controls can be adjusted to get the best out of the testing exercise.

##### 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 the Android OS installed on both the machine and the 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[3].

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 WiFi connection of the Android device need to be configured properly to point to the interception proxy in use[1](https://github.com/pillfill/hiding-passwords-android/).
* The CA certificate of the interception proxy need to be added to the trusted certificates in the certificate storage [2](https://developer.android.com/reference/java/security/KeyStore.html) 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 guarantee (always check the manufacturer policy before taking any action),
* may "brick" the device, e.g. render it unoperable 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 help 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 (which 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 xda forums[21].

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[4]. 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.

-- TODO [Develop section Using an Emulator : what can be done with it, difference with using a real Android mobile, availability, how to start and configure one] --

##### Setting Up a Web Proxy on Virtual Device

To set up a HTTP proxy on the emulator, follow the following procedure (this 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:

* Open the Settings Menu
* Tap on "Wireless & Networks" -> "Cellular Networks" or "Mobile Networks"
* Open "Access Point Names"
* Open the existing APN (e.g. "T-Mobile US")
* Enter "127.0.0.1" in the "Proxy" field and your proxy port in the "Port" field (e.g. "8080")
* 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 stora via Security Settings. For example, you can install the BURP (PortSwigger) CA certificate as follows.

1. Navigate to <http://burp/> using a web browser on the host, and download file 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[5]. 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, like GPS[6] or SMS[7] and many more.

#### Potential Obstacles

##### SSL Pinning

SSL Pinning is already a strong 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[11] or Android-SSL-TrustKiller[12]. 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[9] [13] or ADBI[10]
* Disassembling the APK, identify the SSL Pinning logic in smali code, patch it and reassemble the APK[7] [22]

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" **(-- TODO [add\_link] --)** for further details.

##### Root Detection

To implement root detection on Android, libraries like RootBeer[14] or custom checks are used to verify if the device is rooted or not. See also test case "Testing Root Detection" **(-- TODO [add\_link] --)** and "Testing Advanced Root Detection" for further details.

To be able to efficiently test during a white box test, a debug build with disabled root detection should be provided.

#### Software and Tools (non-comprehensive list)

The context of mobile security testing is a conjunction of multiple different tier of components: **application container**, **communications** and **back-end servers**. These three high-level attack surface will be the main attack surface for a mobile security testing.

* **Application container:** insecure data storage, poor resiliency against reverse engineering and etc.
* **Communication:** usage of insecure or unencrypted communication channel, missing certification pinning and etc.
* **Back-end Servers:** flawed authentication or security logic, vulnerable server side functions and etc.

There are various tool that can be leverage to conduct an effective mobile security testing and like any tool of choice it all depends on the matter of preference and budget. Some basic usage of tools will be covered in the following section and an extensive list of tools can be found in "Testing Tools" **(-- TODO [add\_link] --)** page.

#### Other Considerations

##### App Debug Build vs. Release Build

A debug build has several benefits, when provided during a (white box) test that allows a more comprehensive analysis:

* Debugger can be attached to the running App
* Debug log files of the App are available

-- TODO [Elaborate on Debug Build] --

See also test case "Testing If the app is Debuggable" **(-- TODO [add\_link] --)** for further details.

### References

-- TODO [Update References] --

* [1](https://github.com/pillfill/hiding-passwords-android/) Configuring an Android Device to Work With Burp - <https://support.portswigger.net/customer/portal/articles/1841101-Mobile%20Set-up_Android%20Device.html>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Installing Burp's CA Certificate in an Android Device - <https://support.portswigger.net/customer/portal/articles/1841102-installing-burp-s-ca-certificate-in-an-android-device>
* [3] Creating an Ad-hoc Wireless Network in OS X - <https://support.portswigger.net/customer/portal/articles/1841150-Mobile%20Set-up_Ad-hoc%20network_OSX.html>
* [4] Android Application Security Testing Guide: Part 2 - <http://resources.infosecinstitute.com/android-app-sec-test-guide-part-2/#gref>
* [5] Create and Manage Virtual Devices - <https://developer.android.com/studio/run/managing-avds.html>
* [6] GPS Emulation - <https://developer.android.com/studio/run/emulator-commandline.html#geo>
* [7] SMS Emulation - <https://developer.android.com/studio/run/emulator-commandline.html#sms>
* [8] Mobile Security Certificate Pinning - <http://blog.dewhurstsecurity.com/2015/11/10/mobile-security-certificate-pining.html>
* [9] Frida - <https://www.frida.re/docs/android/>
* [10] ADBI - <https://github.com/crmulliner/adbi>
* [11] SSLUnpinning - <https://github.com/ac-pm/SSLUnpinning_Xposed>
* [12] Android-SSL-TrustKiller - <https://github.com/iSECPartners/Android-SSL-TrustKiller>
* [13] Defeating SSL Pinning in Coin's Android Application - <http://rotlogix.com/2015/09/13/defeating-ssl-pinning-in-coin-for-android/>
* [14] RootBeet - <https://github.com/scottyab/rootbeer>
* [15] Android Lint - <https://sites.google.com/a/android.com/tools/tips/lint/>
* [16] devknox - <https://devknox.io/>
* [17] Android application package - <https://en.wikipedia.org/wiki/Android_application_package>
* [18] QARK - <https://github.com/linkedin/qark/>
* [19] Androbugs - <https://github.com/AndroBugs/AndroBugs_Framework>
* [20] JAADAS - <https://github.com/flankerhqd/JAADAS>
* [21] Guide to root mobile devices - <https://www.xda-developers.com/root/>
* [22] Bypassing SSL Pinning in Android Applications - <https://serializethoughts.com/2016/08/18/bypassing-ssl-pinning-in-android-applications/>

## 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 [1](https://github.com/pillfill/hiding-passwords-android/) 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 [2](https://developer.android.com/reference/java/security/KeyStore.html), 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. APKTool [3] 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.

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.

### Building a Reverse Engineering Environment For Free

With a little effort you can build a reasonable GUI-based reverse engineering environment for free. JD[4] is a free Java de-compiler that integrates with Eclipse[5] and IntelliJ IDEA [6]. Generally, we recommend using IntelliJ, as it is the more light-weight solution, works great for browsing the source code and also 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: Advice is based on the author's opinion and 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 [7]. 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.

More elaborate tasks such as program analysis and automated de-obfuscation can be achieved with open source reverse engineering frameworks such as Radare2 [8] and Angr [9]. You'll find usage examples for many of these free tools and frameworks throughout the guide.

#### Commercial Tools

##### JEB

JEB [10], a commercial decompiler, packs all the functionality needed for static and dynamic analysis of Android apps into a convenient 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 to a subscription model, so you'll need to pay a hefty monthly fee to use it.

##### IDA Pro

IDA Pro [11] understands ARM, MIPS and of course Intel ELF binaries, plus it can deal with Java bytecode. It also comes with remote 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

#### Statically Analyzing Java Code

Unless some mean anti-decompilation tricks have been applied, Java bytecode can be converted back into source code without issues using free tools. We'll be using UnCrackable 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/OMTG-Files/02\_Crackmes/01\_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. Once you have a JAR file, you can use any number of free decompilers to produce Java code - some popular decompilers are JD [4], Jad [10], Proycon [11] and CFR [12].

For this example, let's pick CFR as our decompiler of choice. CFR is under active development, and brand-new releases are made available regularly on the author's website [13]. 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.

For convenience, we have packaged the dex2jar and CFR libraries along with a Python script that can be downloaded from the OWASP MSTG GitHub repo [14]. Download apkx.py and apkx-libs.jar from the repository and you are ready to go. Run apkx.py to extract and decompile that Java classes from the APK:

$ wget https://raw.githubusercontent.com/OWASP/owasp-mstg/master/OMTG-Files/Download/apkx-0.9.tgz  
$ tar xzf apkx-0.9.tgz   
$ chmod +x apkx.py  
$ ./apkx.py UnCrackable-Level1.apk   
Extracting UnCrackable-Level1.apk to UnCrackable-Level1  
dex2jar UnCrackable-Level1/classes.dex -> UnCrackable-Level1/classes.jar  
Processing UnCrackable-Level1/classes.jar (use silent to silence)  
Processing sg.vantagepoint.a.a  
Processing sg.vantagepoint.a.b  
Processing sg.vantagepoint.a.c  
Processing sg.vantagepoint.uncrackable1.MainActivity  
Processing sg.vantagepoint.uncrackable1.a  
Processing sg.vantagepoint.uncrackable1.b  
Processing sg.vantagepoint.uncrackable1.c

You should now find the decompiled sources in the "Uncrackable-Level1/src" directory. 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.

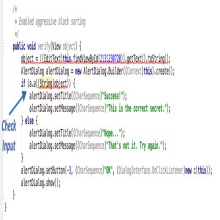
In the next dialog, pick any APK - we don't want to actually compile the project, so it really doesn't matter. Click "next" and choose "Add no Activity", then click "finish".

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

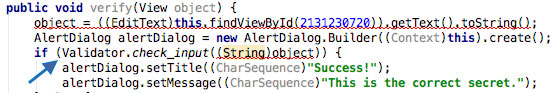
Now, open the "Uncrackable-Level1/src" directory in a file browser and drag the "sg" directory into the now empty "Java" folder in the IntelliJ project view (hold the "alt" key to copy the folder instead of moving it).

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 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.



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".



Congratulations - you just learned the fundamental process 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. The decompiled method should look as follows:

public static boolean check\_input(String string) {  
 byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);  
 byte[] arrby2 = new byte[]{};  
 try {  
 arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d806cf50473cc"), arrby);  
 arrby2 = arrby;  
 }  
 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". Damn ProGuard!), along with something that looks suspiciously like a hex-encoded encryption key (16 hex bytes = 128bit, a common key length). What exactly does this "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 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.

#### Statically Analyzing Native Code

Dalvik and ART both support the Java Native Interface (JNI), which defines defines a way for Java code to interact with native code written in C/C++. Just like on other Linux-based operating systes, 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 [x]. 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. The app is not excatly 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.py. This should extract the source into the HelloWorld/src directory.

$ wget https://raw.githubusercontent.com/OWASP/owasp-mstg/master/OMTG-Files/03\_Examples/01\_Android/01\_HelloWorld-JNI/HelloWorld-JNI.apk  
$ ./apkx.py HelloWorld-JNI.apk

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 architectur des. 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 an 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\_MainActivity\_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 multple 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 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 [x].

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.

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 initalized 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.

We've now covered the basics of static 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 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 behaviour. Next, we'll have a look at dynamic analysis methods that help you do just that.

#### Debugging and Tracing

Android apps support two different types of debugging: Java-runtime-level debugging using Java Debug Wire Protocol (JDWP) and Linux ptrace-style debugging on the native layer.

##### 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 7 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

-- TODO [Complete debugging howto - still some work to do] --

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 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).

Every debugger-enabled Java VM starts an extra JDWP thread for handling protocol packets from the debugger. If the system property ro.debuggable set to "1", this thread is started 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.

Using a JDWP debugger allows you to step through Java code, set breakpoints on Java methods, inspect instance variables of live objects, and many other useful things. You'll be using JDWP most of the time when debugging "normal" Android apps that don't do a lot of calls into native libraries.

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 however - after all, that's what *debug builds* are for. By default, Android disallows both JDWP and native debugging of release builds, and although this is easy to bypass, you'll still likely encounter some limitations and bugs, such as a lack of line breakpoints, method breakpoints being set at the wrong locations, and others. 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.

###### Repackaging

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 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 distibution 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 zipaling and apksigner tools are found in this directory.

UnCrackable App Level 1 is the perfect subject for practicing our new debugging powers, so let's start by repackaging UnCrackable-Level1.apk.

1. Use apktool to restore AndroidManifest.xml:

$ apktool d --no-src target\_app.apk

1. Add android:debuggable = “true” to the manifest:

<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.

$ apktool b  
$ zipalign -v 4 target\_app.recompiled.apk target\_app.recompiled.aligned.apk  
$ jarsigner -verbose -keystore ~/.android/debug.keystore target\_app.recompiled.aligned.apk signkey

1. Reinstall the app:

$ adb install target\_app.recompiled.aligned.apk

$ apktool d --no-src UnCrackable-Level1.apk

Set android:allowBackup="true" as described above.

$ 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  
$ adb install UnCrackable-Repackaged.apk

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.

The abd jdwp command lists

An important restriction is that line breakpoints usually won't work, as the release bytecode doesn't contain line information. Method breakpoints do work however.

$ adb shell ps | grep uncrackable  
u0\_a157 7328 201 1564936 50656 ffffffff 00000000 S sg.vantagepoint.uncrackable1  
$ adb forward tcp:7777 jdwp:7328  
$ jdb attach localhost:7777  
Initializing jdb ...  
>

> classes  
(...)  
sg.vantagepoint.a.a  
sg.vantagepoint.a.b  
sg.vantagepoint.a.c  
sg.vantagepoint.uncrackable1.MainActivity  
sg.vantagepoint.uncrackable1.a  
sg.vantagepoint.uncrackable1.b  
sg.vantagepoint.uncrackable1.c  
short[]  
short[][]  
sun.misc.Unsafe  
> methods  
sg.vantagepoint.uncrackable1.a a(java.lang.String)  
sg.vantagepoint.uncrackable1.a b(java.lang.String)  
(...)

> 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

##### The 'Wait For Debugger' Feature

The Developer options also contain the useful "Wait for Debugger" setting that allows you to suspend an app during startup. We'll revisit this option in a bit.

Note: Even with ro.debuggable set to 1 in default.prop, the app won't show up in the "debug app" list unless the android:debuggable flag is set to true in the Manifest.

###### 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 - its 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 was described in the NetSPI blog []

-- TODO [Debugging with IntelliJ] --

File -> New -> Project...

Choose "Android"

Name the project

Choose "Add no Activity"

##### 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, as well as sometimes Intel or MIPS).

To try it out, let's install HelloWorld-JNI.apk.

$ adb install HelloWorld-JNI.apk

If you followed the instructions at the start of this chapter, you should already have the Android NDK. The NDK ships with prebuilt versions of gdbserver for various architectures. Copy gdbserver to your device:

$ adb push $NDK/prebuilt/android-arm/gdbserver/gdbserver /data/local/tmp

$ 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 = 14342  
Listening on port 1234

$ adb forward tcp:1234 tcp:1234  
$ export TOOLCHAIN=[YOUR-NDK-PATH]/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86\_64/bin/  
$ $TOOLCHAIN/arm-linux-androideabi-gdb libnative-lib.so  
GNU gdb (GDB) 7.7  
(...)  
Reading symbols from libnative-lib.so...(no debugging symbols found)...done.  
(gdb) target remote :1234  
Remote debugging using :1234  
0xb6e0f124 in ?? ()

The problem: At this point it's already too late! The function has already run...

```bash  
$ adb shell  
android $ su  
android # /data/local/tmp/gdbserver --attach localhost:1234 14342

Go to "Developer Options" -> "Select debug app" and pick HelloWorldJNI. Activate the "Wait for debugger" switch.

Launch the app

$ adb jdwp  
14342  
$ adb forward tcp:7777 jdwp:14342  
$ { echo "suspend"; cat; } | jdb -attach localhost:7777  
> 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]

At this point, the library has been loaded.

$ adb shell  
android $ su  
android # /data/local/tmp/gdbserver --attach localhost:1234 14342

$ 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 ?? ()  
(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  
(...)

Set a breakpoint:

(gdb) b \*0xa3522e78  
Breakpoint 1 at 0xa3522e78  
(gdb) cont

In jdb:

main[1] resume  
All threads resumed.

In gdb:

Breakpoint 1, 0xa3522e78 in Java\_sg\_vantagepoint\_helloworldjni\_MainActivity\_stringFromJNI () from libnative-lib.so  
(gdb) disass $pc  
Dump of assembler code for function Java\_sg\_vantagepoint\_helloworldjni\_MainActivity\_stringFromJNI:  
=> 0xa3522e78 <+0>: ldr r2, [r0, #0]  
 0xa3522e7a <+2>: ldr r1, [pc, #8] ; (0xa3522e84 <Java\_sg\_vantagepoint\_helloworldjni\_MainActivity\_stringFromJNI+12>)  
 0xa3522e7c <+4>: ldr.w r2, [r2, #668] ; 0x29c  
 0xa3522e80 <+8>: add r1, pc  
 0xa3522e82 <+10>: bx r2  
 0xa3522e84 <+12>: lsrs r4, r7, #28  
 0xa3522e86 <+14>: movs r0, r0  
End of assembler dump.

##### 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 I didn’t know about for several weeks until I discovered it by accident) 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 the 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 Studo (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 ptrace() 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 (5) results in a Kernel panic on default Lolllipop 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 [11] - an extension to the DECAF dynamic analysis framework [15] - 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 [13] 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 [16]. 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 [17] 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 empoys SSL pinning.

In most cases, both issues can be fixed by making minor changes and re-packaging and re-signing the app (the exception are apps that run additional integrity checks beyond default Android code signing - in theses cases, you also have to patch out those additional checks as well).

##### Example 1: Repackaging an App for Debugging

In our first example, we'll modify the android:debuggable flag to enable debugging of a release app. You can reproduce this with any app downloaded from the Play Store.

##### Example 2: Disabling SSL Pinning

As seen in the previous Chapter, certificate pinning might hinder an analyst when analyzing the traffic. To help with this problem, the binary can be patched to allow other certificates. To demonstrate how Certificate Pinning can be bypassed, we will walk through the necessary steps to bypass Certificate Pinning implemented in an example application.  
Disassembling the APK using apktool

$ apktool d target\_apk.apk

Modify the Certificate Pinning logic:  
We need to locate where within the smali source code the certificate pinning checks are done. Searching the smali code for keywords such as “X509TrustManager” should point you in the right direction.  
In this case a search for “X509TrustManager” returned one class which implements an own Trustmanager. This file contains methods named “checkClientTrusted”, “checkServerTrusted” and “getAcceptedIssuers”.  
The “return-void” opcode was added to the first line of each of these methods. The “return-void” statement is a Dalvik opcode to return ‘void’ or null. For more Dalvik opcodes refer to <http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html>.  
In this context, return-void means that 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  
 iget-object v1, p0, Lasdf/t$a;->a:Ljava/util/ArrayList;  
  
 invoke-virtual {v1}, Ljava/util/ArrayList;->iterator()Ljava/util/Iterator;  
  
 move-result-object v1  
  
 :goto\_0  
 invoke-interface {v1}, Ljava/util/Iterator;->hasNext()Z

#### 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" [1](https://github.com/pillfill/hiding-passwords-android/). 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](https://docs.oracle.com/javase/tutorial/reflect/) 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

-- TODO [Detailed Xposed tutorial] --

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 we have to do is to replace the code with something that returns "false".

Using an Xposed module is one way to do this. Modules for Xposed are developed and deployed with Android Studio just like regular Android apps. The author, rovo89, provides a great [tutorial](https://github.com/rovo89/XposedBridge/wiki/Development-tutorial) showing how to write, compile and install a module.

Code:

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);  
 }  
  
 });  
 }  
}

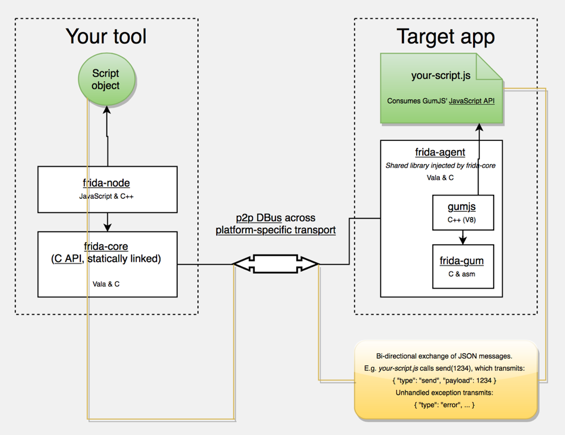
#### Dynamic Instrumentation with FRIDA

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 first 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/\*](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.

-- TODO [Detailed Frida tutorial] --

##### Installing Frida

To install Frida locally, simply use Pypi:

$ sudo pip install frida

Your Android device needs to be rooted to get Frida running. Download the frida-server binary from the [Frida releases page](https://github.com/frida/frida/releases). 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-android-arm.xz

Copy frida-server to the device and run it:

$ adb push frida-server /data/local/tmp/  
$ adb shell "chmod 755 /data/local/tmp/frida-server"  
$ adb shell "su -c /data/local/tmp/frida-server &"

With frida-server running, you should now be able to get a list of running processes with the following command:

$ frida-ps -U  
 PID Name  
----- --------------------------------------------------------------  
 276 adbd  
 956 android.process.media  
 198 bridgemgrd  
 1191 com.android.nfc  
 1236 com.android.phone  
 5353 com.android.settings  
 936 com.android.systemui  
(...)

The -U option lets Frida search for USB devices or emulators.

To trace specific (low level) library calls, you can use the frida-trace command line tool:

frida-trace -i "open" -U com.android.chrome

This generates a little javascript in \_\_handlers\_\_/libc.so/open.js 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 Fridas [Javascript API](https://www.frida.re/docs/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 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{7ccea78 G.ED..... ......ID 0,0-0,0 #7f0c01fc app:id/action\_bar\_black\_background}  
[\*] Instance found: android.view.View{2809551 V.ED..... ........ 0,1731-0,1731 #7f0c01ff app:id/menu\_anchor\_stub}  
[\*] Instance found: android.view.View{be471b6 G.ED..... ......I. 0,0-0,0 #7f0c01f5 app:id/location\_bar\_verbose\_status\_separator}  
[\*] Instance found: android.view.View{3ae0eb7 V.ED..... ........ 0,0-1080,63 #102002f android:id/statusBarBackground}  
[\*] Finished heap search

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:

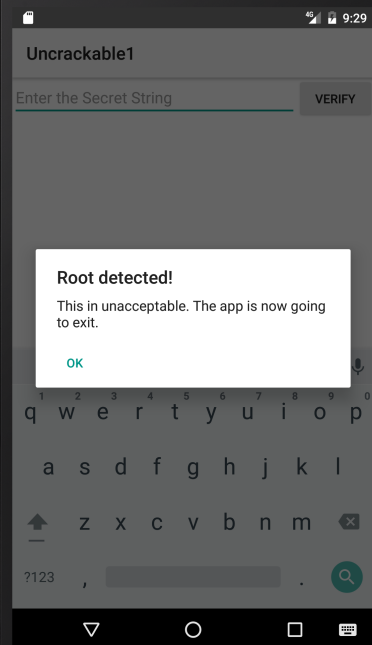
Java.perform(function () {  
 var view = Java.use("android.view.View");  
 var methods = view.class.getMethods();  
 for(var i = 0; i < methods.length; i++) {  
 console.log(methods[i].toString());  
 }  
});

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 in 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 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 to 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 retreive 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)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);  
 byte[] arrby2 = new byte[]{};  
 try {  
 arrby2 = arrby = sg.vantagepoint.a.a.a((byte[])a.b((String)"8d127684cbc37c17616d806cf50473cc"), (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.uncrackable1  
 \_\_\_\_  
 / \_ | Frida 9.1.16 - A world-class dynamic instrumentation framework  
 | (\_| |  
 > \_ | Commands:  
 /\_/ |\_| help -> Displays the help system  
 . . . . object? -> Display information about 'object'  
 . . . . exit/quit -> Exit  
 . . . .  
 . . . . More info at http://www.frida.re/docs/home/  
   
[\*] Starting script  
[USB::Android Emulator 5554::sg.vantagepoint.uncrackable1]-> [\*] onClick handler modified  
[\*] sg.vantagepoint.a.a.a modified  
[\*] onClick called.  
[\*] Decrypted: I want to believe

The hooked function outputted 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.

### 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.

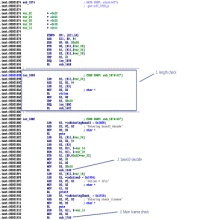
Amongst other things, this is useful in cases where we need to find the right inputs for reaching a certain block of code. In the following example, we'll use Angr to solve a simple Android crackme in an automated fashion. The crackme takes the form of a native ELF binary that can be downloaded here:

<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.



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, #0xFFFFFFF0  
.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. Congratulati"...  
.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 observer 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.  
JQAE6ACMABNAAIIA

### 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.

#### Preparing a Development Environment

-- TODO [Creating a Standalone Toolchain] --

For convenience, you can create a standalone toolchain create a standalone toolchain for Android Nougat (API 24):

$ $YOUR\_NDK\_PATH/build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-24 --install-dir=/tmp/my-android-toolchain

#### 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 > ../myinitd.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\_RWX 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 codenam” 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 5.1, 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-21 --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/mmcblk0p28

Then, dump the whole thing into a file:

$ adb shell "su -c dd if=/dev/block/mmcblk0p19 of=/data/local/tmp/boot.img"  
$ adb pull /data/local/tmp/boot.img

Next, extract the ramdisk 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](https://github.com/pillfill/hiding-passwords-android/).

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:

bash $ adb shell "su -c echo ABCD > /data/local/tmp/nowyouseeme" $ adb shell cat /data/local/tmp/nowyouseeme ABCDbash

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, facessat, 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:

#include <linux/kernel.h>  
#include <linux/module.h>  
#include <linux/moduleparam.h>  
#include <linux/unistd.h>  
#include <linux/slab.h>  
#include <asm/uaccess.h>  
  
asmlinkage int (\*real\_openat)(int, const char \_\_user\*, int);  
  
void \*\*sys\_call\_table;  
  
int new\_openat(int dirfd, const char \\_\_user\* pathname, int flags)  
{  
 char \*kbuf;  
 size\_t len;  
  
 kbuf=(char\*)kmalloc(256,GFP\_KERNEL);  
 len = strncpy\_from\_user(kbuf,pathname,255);  
  
 if (strcmp(kbuf, "/data/local/tmp/nowyouseeme") == 0) {  
 printk("Hiding file!\n");  
 return -ENOENT;  
 }  
  
 kfree(kbuf);  
  
 return real\_openat(dirfd, pathname, flags);  
}  
  
int init\_module() {  
  
 sys\_call\_table = (void\*)0xc000f984;  
 real\_openat = (void\*)(sys\_call\_table[\_\_NR\_openat]);  
  
return 0;  
  
}

To build the kernel module, you need the kernel sources and a working toolchain - since you 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 0xbf000000 (PO)

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 [1](https://github.com/pillfill/hiding-passwords-android/) 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 whole 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 [x]

### References

-- TODO [Sync with text] --

* [1](https://github.com/pillfill/hiding-passwords-android/) OWASP Mobile Crackmes - <https://github.com/OWASP/owasp-mstg/blob/master/OMTG-Files/02_Crackmes/List_of_Crackmes.md>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Android Studio - <https://developer.android.com/studio/index.html>
* [3] APKTool - <https://ibotpeaches.github.io/Apktool/>
* [4] JD - <http://jd.benow.ca/>
* [5] Eclipse - <https://eclipse.org/ide/>
* [6] IntelliJ IDEA - <https://www.jetbrains.com/idea/>
* [7] Smalidea - <https://github.com/JesusFreke/smali/wiki/smalidea>
* [8] Radare2 - <https://www.radare.org>
* [9] Angr - <http://angr.io/>
* [10] JEB -
* [11] IDA Pro - <https://www.hex-rays.com/products/ida/>
* [12] JAD - <http://www.javadecompilers.com/jad>
* [13] Proycon - <http://proycon.com/en/>
* [14] CFR - <http://www.benf.org/other/cfr/>
* [15] APKX - <https://github.com/OWASP/owasp-mstg/tree/master/OMTG-Files/01_Tools/01_Android/01_apkx>
* Frida - <https://www.frida.re>
* Bionic - <https://github.com/android/platform_bionic>
* DroidScope -
* DECAF - <https://github.com/sycurelab/DECAF>
* PANDA - <https://github.com/moyix/panda/blob/master/docs/>
* VxStripper -
* Dynamic Malware Recompliation - <http://ieeexplore.ieee.org/document/6759227/>
* UnCrackable Android App Level 1 - <https://github.com/OWASP/owasp-mstg/tree/master/OMTG-Files/02_Crackmes/01_Android/Level_01>
* NetSPI Blog - Attacking Android Applications with Debuggers - <https://blog.netspi.com/attacking-android-applications-with-debuggers/>
* <http://repo.xposed.info/module/de.robv.android.xposed.installer>
* <https://github.com/rovo89/XposedBridge/wiki/Development-tutorial>
* <https://github.com/JesusFreke/smali>
* <https://dl.packetstormsecurity.net/papers/general/HITB_Hacking_Soft_Tokens_v1.2.pdf>
* <https://en.wikipedia.org/wiki/Concolic_testing>

## Testing Data Storage

### Testing for Sensitive Data in Local Storage

#### Overview

-- TODO: What is sensitive information? Need to be described, ideally defined by the customer (data classification policy).

This 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 this kind 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 other attacks like social engineering (when PII is disclosed), session hijacking (if session information or a token is disclosed) or gather information from apps that have a payment option in order to attack and abuse it.

Storing data[1](https://github.com/pillfill/hiding-passwords-android/) is essential for many mobile applications, 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 available on the Android platform:

* Shared Preferences
* Internal Storage
* External Storage
* SQLite Databases

The following examples shows snippets of code to demonstrate bad practices that discloses sensitive information and also shows the different storage mechanisms in detail on Android.

##### Shared Preferences

SharedPreferences[2](https://developer.android.com/reference/java/security/KeyStore.html) is a common approach to store Key/Value pairs persistently in the filesystem by using a 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/<PackageName>/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 classes to operate 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/<PackageName>/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[3].
* 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[4].

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

##### SQLite Databases (Encrypted)

By using the library SQLCipher[5] 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)

##### Internal Storage

Files can be saved directly on the internal storage[6] of the device. By default, files saved to the internal storage are private to your application and other applications cannot access them. When the user uninstalls your application, 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 need to be checked, to make sure that only the app itself has access to the file 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 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[7] 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.getExternalFilesDir(), "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/<packagename>/) will not be deleted when the user uninstall the application.

#### 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 if sensitive data is processed insecurely.

* Check AndroidManifest.xml for permissions to read and write to external storage, like uses-permission android:name="android.permission.WRITE\_EXTERNAL\_STORAGE"
* Check the source code for functions and API calls that are used for storing data:
* Open the Java Files in an IDE or text editor of your choice or use grep on the command line to search for:
  + 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 with the source code:

* Check if simple bit operations are used, like XOR or Bit flipping to “encrypt” sensitive information like credentials or private keys that are 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 KeyStore[8].
* Check if keys are disclosed.

##### KeyChain and 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 should not be stored in clear text and 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[10]** and **Keystore[8]**. The following controls should therefore be used:

* Check if a key pair is created within the app by looking for the class KeyPairGenerator.
* Check that the application is using the 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 it’s usage.
* 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 pattern .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\_nam>/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, execute (rwx). Others should have no permissions to files, but may have the executable flag to directories.

#### 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 not stored in the first place on the device itself.

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 leave it unencrypted on the device.

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

* **Android KeyStore[8]**: The KeyStore provides a secure system level credential storage. It is important to note that the credentials are not actually stored within the KeyStore. 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 KeyStore is a secure container that makes it difficult for an attacker to retrieve the private key and guards the encrypted data. Nevertheless an attacker can access all keys on a rooted device in the folder /data/misc/keystore/. The KeyStore is encrypted using the user’s own lock screen pin/password, hence, when the device screen is locked the KeyStore is unavailable[9].
* **Android KeyChain[10]**: The KeyChain class is used to store and retrieve 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.
* Encryption or decryption functions that were self implemented need to be avoided. Instead use Android implementations such as Cipher[11], SecureRandom[12] and KeyGenerator[13].
* 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 (session token). If possible, use the AccountManager[14] 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[15] 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 inputting 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.
* As a security in depth measure code obfuscation should also be applied to the app, to make reverse engineering harder for attackers.

#### 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

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) Security Tips for Storing Data - <http://developer.android.com/training/articles/security-tips.html#StoringData>  
[2](https://developer.android.com/reference/java/security/KeyStore.html) SharedPreferences - <http://developer.android.com/reference/android/content/SharedPreferences.html>  
[3] SQLite Journal files - <https://www.sqlite.org/tempfiles.html>  
[4] SQLite Lock Files - <https://www.sqlite.org/lockingv3.html>  
[5] SQLCipher - <https://www.zetetic.net/sqlcipher/sqlcipher-for-android/>  
[6] Using Internal Storage - <http://developer.android.com/guide/topics/data/data-storage.html#filesInternal>  
[7] Using External Storage - <https://developer.android.com/guide/topics/data/data-storage.html#filesExternal>  
[8] Android KeyStore System - <http://developer.android.com/training/articles/keystore.html>  
[9] Use Android Keystore - <http://www.androidauthority.com/use-android-keystore-store-passwords-sensitive-information-623779/>  
[10] Android KeyChain - <http://developer.android.com/reference/android/security/KeyChain.html>  
[11] Cipher - <https://developer.android.com/reference/javax/crypto/Cipher.html>  
[12] SecureRandom - <https://developer.android.com/reference/java/security/SecureRandom.html>  
[13]KeyGenerator - <https://developer.android.com/reference/javax/crypto/KeyGenerator.html>  
[14] AccountManager - <https://developer.android.com/reference/android/accounts/AccountManager.html>  
[15] Secure Preferences - <https://github.com/scottyab/secure-preferences>

##### Tools

* Enjarify - <https://github.com/google/enjarify>
* JADX - <https://github.com/skylot/jadx>
* Dex2jar - <https://github.com/pxb1988/dex2jar>
* Lint - <http://developer.android.com/tools/help/lint.html>
* Sqlite3 - <http://www.sqlite.org/cli.html>

### 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 that are stored locally when being offline and being sent to the application developer once being online again or simply for usage statistics. However, logging sensitive data such as credit card number and 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[1](https://github.com/pillfill/hiding-passwords-android/), .log[a-Z]
* Logger Class
* StrictMode
* System.out/System.err.print

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

#### Static Analysis

Check the source code for usage of Logging functions, by searching for the following terms:

1. Functions and classes like:

* Log.d, Log.e, Log.i, Log.v, Log.w and Log.wtf
* Logger
* StrictMode

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

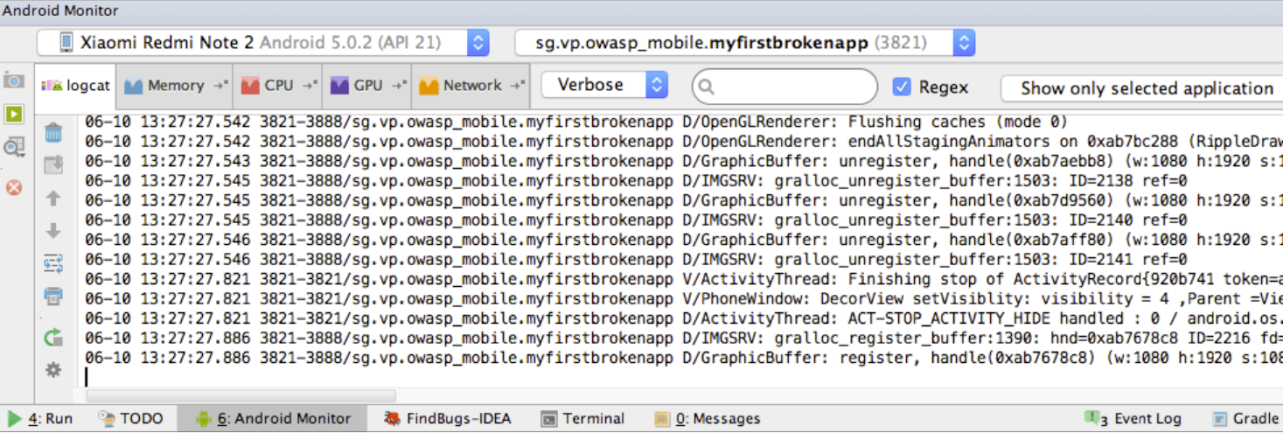
* logfile
* logging
* logs
* System.out.print | System.out.println

#### Dynamic Analysis

Use the mobile app extensively so that all functionality is at least triggered once.

1. 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.
2. Many application developers use still 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 and not only the output created by the log classes. In order to verify what data is written to logfiles and printed directly by using System.out.println() or printStackTrace() the code should be checked for these functions and the tool LogCat[2](https://developer.android.com/reference/java/security/KeyStore.html) can be used to check the 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 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 or DexGuard can be used to strip out logging portions in the code when preparing the production release. For example, to remove logging calls within an Android application, 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(...);  
}

#### 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

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) Overview of Class Log - <http://developer.android.com/reference/android/util/Log.html>  
[2](https://developer.android.com/reference/java/security/KeyStore.html) Debugging Logs with LogCat - <http://developer.android.com/tools/debugging/debugging-log.html>

##### Tools

* ProGuard - <http://proguard.sourceforge.net/>
* DexGuard - <https://www.guardsquare.com/dexguard>
* LogCat - <http://developer.android.com/tools/help/logcat.html>
* ClassyShark - <https://github.com/google/android-classyshark>

### 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 AnroidManifest.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 Proxy*[1](https://github.com/pillfill/hiding-passwords-android/) 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

[1](https://github.com/pillfill/hiding-passwords-android/) Configure Burp with Android - <https://support.portswigger.net/customer/portal/articles/1841101-configuring-an-android-device-to-work-with-burp>  
[2](https://developer.android.com/reference/java/security/KeyStore.html) 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 keying 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. This feature should therefore be disabled for input fields that are asking for sensitive information.

#### 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[1](https://github.com/pillfill/hiding-passwords-android/):

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

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) No suggestions for text - <https://developer.android.com/reference/android/text/InputType.html#TYPE_TEXT_FLAG_NO_SUGGESTIONS>

### Testing for Sensitive Data in the Clipboard

#### Overview

When keying in data into input fields, the clipboard[1](https://github.com/pillfill/hiding-passwords-android/) can be used to copy data in. The clipboard is accessible systemwide and therefore shared between the apps. This feature can be misused by malicious apps in order to get sensitive data.

#### 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.

#### 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;  
 }  
0  
 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

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) Copy and Paste in Android - <https://developer.android.com/guide/topics/text/copy-paste.html>

### Testing Whether Sensitive Data Is Exposed via IPC Mechanisms

#### 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[1](https://github.com/pillfill/hiding-passwords-android/)
* Services[2](https://developer.android.com/reference/java/security/KeyStore.html)
* Bound Services[9]
* AIDL[10]
* Intents[3]
* Content Providers[4]

#### Static Analysis

The first step is to look into the AndroidManifest.xml in order to detect and identify IPC mechanisms exposed by the app. You will want to identify elements such as:

* <intent-filter>[5]
* <service>[6]
* <provider>[7]
* <receiver>[8]

Except for the <intent-filter> element, check if the previous elements contain the following attributes:

* android:exported
* android:permission

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, *ContentProviders* can be used to access database information, while services can be probed to see if they return data. Also BroadcastReceiver and Broadcast intents can leak sensitive information if probed or sniffed.

**Vulnerable ContentProvider**

An example of a vulnerable *ContentProvider*:  
(and SQL injection **-- TODO [Refer to any input validation test in the project] --**

<provider android:name=".CredentialProvider"  
 android:authorities="com.owaspomtg.vulnapp.provider.CredentialProvider"  
 android:exported="true">  
</provider>

As can be seen in the AndroidManifest.xml above, the application exports the content provider. In the CredentialProvider.java file the query function need to be inspected to detect if any sensitive information is leaked:

public Cursor query(Uri uri, String[] projection, String selection,  
 String[] selectionArgs, String sortOrder) {  
 SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();  
 // the TABLE\_NAME to query on  
 queryBuilder.setTables(TABLE\_NAME);  
 switch (uriMatcher.match(uri)) {  
 // maps all database column names  
 case CREDENTIALS:  
 queryBuilder.setProjectionMap(CredMap);  
 break;  
 case CREDENTIALS\_ID:  
 queryBuilder.appendWhere( ID + "=" + uri.getLastPathSegment());  
 break;  
 default:  
 throw new IllegalArgumentException("Unknown URI " + uri);  
 }  
 if (sortOrder == null || sortOrder == ""){  
 sortOrder = USERNAME;  
 }  
 Cursor cursor = queryBuilder.query(database, projection, selection,  
 selectionArgs, null, null, sortOrder);  
 cursor.setNotificationUri(getContext().getContentResolver(), uri);  
 return cursor;  
 }

The query statement would return all credentials when accessing content://com.owaspomtg.vulnapp.provider.CredentialProvider/CREDENTIALS.

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

An example of a vulnerable broadcast is the following:

private void vulnerableBroadcastFunction() {  
 // ...  
 Intent VulnIntent = new Intent();  
 VulnIntent.setAction("com.owasp.omtg.receiveInfo");  
 VulnIntent.putExtra("ApplicationSession", "SESSIONID=A4EBFB8366004B3369044EE985617DF9");  
 VulnIntent.putExtra("Username", "litnsarf\_omtg");  
 VulnIntent.putExtra("Group", "admin");  
 }  
 this.sendBroadcast(VulnIntent);

#### Dynamic Analysis

Similar to White-box testing, you should decompile the application (if possible) and create a list of IPC mechanisms implemented by going through the AndroidManifest.xml file. Once you have the list, prove each IPC via ADB or custom applications to see if they leak any sensitive information.

* Vulnerable ContentProvider

In the case of the previous content provider, we can probe the content provider via ADB, but we need to know the correct URI. Once the APK has been decompiled, use the commands strings and grep to identify the correct URI to use:

$ strings classes.dex | grep "content://"  
com.owaspomtg.vulnapp.provider.CredentialProvider/credentials

Now you can probe the content provider via adb with the following command:

$ 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  
...

* Vulnerable Broadcast

To sniff intents install and run the application on a device (actual device or emulated device) and use tools like Drozer or Intent Sniffer to capture intents and broadcast messages.

#### Remediation

For an *activity*, *broadcast* and *service* the permission of the caller can be checked either by code or in the manifest.

If not strictly required, be sure that your IPC does not have the android:exported="true" value in the AndroidManifest.xml file, as otherwise this allows all other apps on Android to communicate and invoke it.

If the *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[11].  
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 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
* M2 - Insecure Data Storage

##### OWASP MASVS

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

##### CWE

* CWE-634 - Weaknesses that Affect System Processes

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) IPCBinder - <https://developer.android.com/reference/android/os/Binder.html>  
[2](https://developer.android.com/reference/java/security/KeyStore.html) IPCServices - <https://developer.android.com/guide/components/services.html>  
[3] IPCIntent - <https://developer.android.com/reference/android/content/Intent.html>  
[4] IPCContentProviders - <https://developer.android.com/reference/android/content/ContentProvider.html>  
[5] IntentFilterElement - <https://developer.android.com/guide/topics/manifest/intent-filter-element.html>  
[6] ServiceElement - <https://developer.android.com/guide/topics/manifest/service-element.html>  
[7] ProviderElement - <https://developer.android.com/guide/topics/manifest/provider-element.html>  
[8] ReceiverElement - <https://developer.android.com/guide/topics/manifest/receiver-element.html>  
[9] BoundServices - <https://developer.android.com/guide/components/bound-services.html>  
[10] AIDL - <https://developer.android.com/guide/components/aidl.html>  
[11] SendBroadcast - <https://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent)>

##### Tools

* Drozer - <https://labs.mwrinfosecurity.com/tools/drozer/>
* IntentSniffer - <https://www.nccgroup.trust/us/about-us/resources/intent-sniffer/>

### 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"

#### 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 information or asks for information to be keyed in.

If the information is masked, e.g. by replacing characters in the text field through asterisks 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

When backup options are available, it is important to consider that user data may be stored within the app data directory. The backup feature could potentially leak sensitive information such as session identifiers, usernames, email addresses, passwords, keys and much more. Consider to encrypt backup data and avoid to store any sensitive information that is not strictly required within the data directory of the app.

Besides a local backup, Android provides two ways for apps to backup their data to the cloud:

* Auto Backup for apps in Android 6.0 (available >= API level 23), which uploads the data to the user's Google Drive account.
* Key/Value Backup (Backup API or Android Backup Service), which uploads the data to the Android Backup Service.

#### Static Analysis

##### Local

In order to backup all your application data Android provides an attribute called allowBackup[1](https://github.com/pillfill/hiding-passwords-android/). 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) - $ 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,
* if sensitive information is protected through encryption before sending it to the cloud.

**Auto Backup**  
When setting the attribute android:allowBackup to true in the manifest file, auto backup is enabled. 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 packageNameOfTheDesiredAPK

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[6] or JRE8[7], 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 need to be explicitly set in order to deactivate it.

Sensitive information should not be sent in clear text to the cloud. It should either be:

* avoided to store 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[2](https://developer.android.com/reference/java/security/KeyStore.html), in case they should not be shared with 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

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) Documentation for the application tag - <https://developer.android.com/guide/topics/manifest/application-element.html#allowbackup>  
[2](https://developer.android.com/reference/java/security/KeyStore.html) IncludingFiles - <https://developer.android.com/guide/topics/data/autobackup.html#IncludingFiles>  
[3] Backing up App Data to the cloud - <https://developer.android.com/guide/topics/data/backup.html>  
[4] KeyValueBackup - <https://developer.android.com/guide/topics/data/keyvaluebackup.html>  
[5] BackupAgentHelper - <https://developer.android.com/reference/android/app/backup/BackupAgentHelper.html>  
[6] BackupAgent - <https://developer.android.com/reference/android/app/backup/BackupAgent.html>  
[7] Oracle JCE Unlimited Strength Jurisdiction Policy Files JRE7 - <http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html>  
[8] Oracle JCE Unlimited Strength Jurisdiction Policy Files JRE8 - <http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html>  
[9] AutoBackup - <https://developer.android.com/guide/topics/data/autobackup.html>

##### Tools

* Android Backup Extractor - <https://sourceforge.net/projects/adbextractor/>

### 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 next entered. However, this would leak sensitive information that is present within the app.

To verify if the application may expose sensitive information via task switcher, detect if the FLAG\_SECURE[1](https://github.com/pillfill/hiding-passwords-android/) 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 task-switcher button, to see the snapshot. As showed below, if FLAG\_SECURE is set (image on the left), the snapshot is entirely black, while if the FLAG\_SECURE is not set (image on the right), information within the activity are shown:

|  |  |
| --- | --- |
| FLAG\_SECURE not set | FLAG\_SECURE set |

|  |  |
| --- | --- |
| OMTG_DATAST_010_1_FLAG_SECURE | OMTG_DATAST_010_2_FLAG_SECURE |

#### 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);

Moreover, the following suggestions can also be implemented to enhance your application security posture:

* Quit the app entirely when backgrounded. This will destroy any retained GUI screens.
* Nullify the data on a GUI screen before leaving the screen or logging out.

#### 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

##### Info

[1](https://github.com/pillfill/hiding-passwords-android/) FLAG\_SECURE - <https://developer.android.com/reference/android/view/Display.html#FLAG_SECURE>

### 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 application 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.

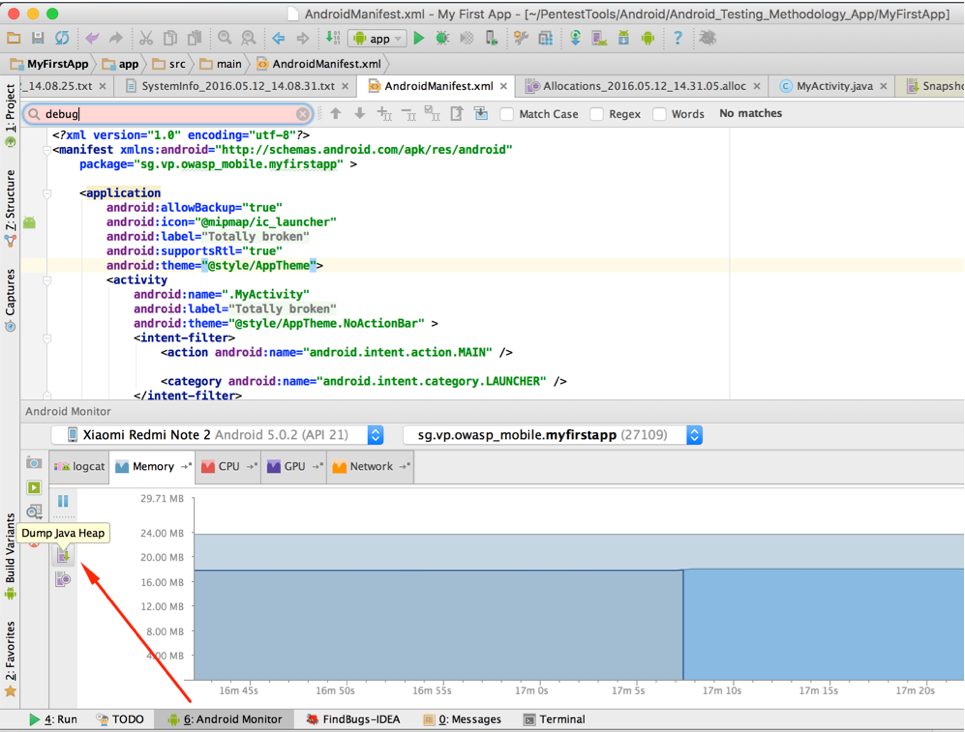
#### Static Analysis

It needs to be identified within the code when sensitive information is stored within a variable or processed and is therefore available within the memory. This information can then be used in dynamic testing when using the app.

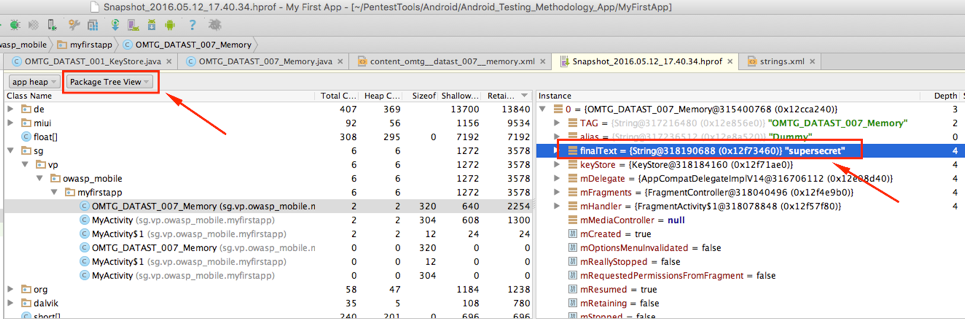
#### Dynamic Analysis

To analyse the memory of an app, the app must be **debuggable**.  
See the instructions in XXX (-- TODO [Link to repackage and sign] --) on how to repackage and sign an Android app to enable debugging for an app, if not already done. Also adb integration need to be activated in Android Studio in “*Tools/Android/Enable ADB Integration*” in order to take a memory dump.

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 analyse 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.



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

Before the *.hprof* file can be opened in MAT it needs to be converted. The tool *hprof-conf* 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 Username
* Decrypted information
* User or session related information
* Session ID
* Interaction with OS, e.g. reading file content

#### Remediation

In Java memory cannot be directly overwritten, instead the garbage collector will collect the object once no references are available anymore. To achieve this the object should be *nulled* immediately after usage to reduce the attack surface.

#### 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

##### Info

* Securely stores sensitive data in RAM - <https://www.nowsecure.com/resources/secure-mobile-development/coding-practices/securely-store-sensitive-data-in-ram/>

##### 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 secure 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 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 checked if they can be bypassed.

#### Remediation

Different checks on the Android device can be implemented by querying different system preferences from *Settings.Secure*[1](https://github.com/pillfill/hiding-passwords-android/). The *Device Administration API*[2](https://developer.android.com/reference/java/security/KeyStore.html) 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

* CWE -- TODO [Link to CWE issue] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Settings.Secure - <https://developer.android.com/reference/android/provider/Settings.Secure.html>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Device Administration API - <https://developer.android.com/guide/topics/admin/device-admin.html>

### 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 users.

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.  
-- TODO [Create content on Static Analysis for Verifying User Education Controls] --

#### 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.  
-- TODO [Develop content on Dynamic Analysis on Verifying User Education Controls] --

#### Remediation

Warnings should be implemented that address the key points listed in the overview section.  
-- TODO [Develop remediations on Verifying User Education Controls] --

#### 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

* CWE: -- TODO [Link to CWE issue] --

## Testing Cryptography

### Verifying Key Management

#### Overview

The use of a hard-coded or world-readable cryptographic key significantly increases the possibility that encrypted data may be recovered.

-- TODO [Develop overview on Verifying Key Management]

#### White-box Testing

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

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

Since the key is the same for all the users and it is trivial to obtain it, the advantages of having sensitive data encrypted are gone, and there is effectively no point in such encryption at all. 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[1](https://github.com/pillfill/hiding-passwords-android/):

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);  
}

#### Black-box Testing

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\_S3cr3t\_K3Y</string>  
 </resources>

* build configs, such as in local.properties or gradle.properties

Example:

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

* shared preferences, typically at /data/data/package\_name/shared\_prefs

#### Remediation

If you need to store a key for repeated use, use a mechanism, such as KeyStore[2](https://developer.android.com/reference/java/security/KeyStore.html), that provides a mechanism for long term storage and retrieval of cryptographic keys.

#### References

##### OWASP MASVS

* V3.1: "The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption"
* V3.5: "The app doesn't re-use the same cryptographic key for multiple purposes"
* V3.7: "All cryptographic keys are changeable, and are generated or replaced at installation time"

##### OWASP Mobile Top 10

* M6 - Broken Cryptography

##### CWE

* CWE-320: Key Management Errors
* CWE-321: Use of Hard-coded Cryptographic Key

##### Info

* <https://rammic.github.io/2015/07/28/hiding-secrets-in-android-apps/>
* <https://medium.com/@ericfu/securely-storing-secrets-in-an-android-application-501f030ae5a3#.7z5yruotu>

##### Tools

* [QARK](https://github.com/linkedin/qark)
* [Mobile Security Framework](https://github.com/ajinabraham/Mobile-Security-Framework-MobSF)

### Testing for Custom Implementations of Cryptography

#### Overview

The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Well-known techniques may exist to break the algorithm.

#### White-box Testing

Carefully inspect all the crypto methods, especially those which are directly applied to the sensitive data. Pay close attention to seemingly standard but modified algorithms. Remember that encoding is not encryption! Any appearance of direct XORing might be a good sign to start digging deeper.

#### Black-box Testing

Although fuzzing of the custom algorithm might work in case of very weak crypto, the recommended approach would be to decompile the APK and inspect the algorithm to see if custom encryption schemes is really the case (see "White-box Testing")

#### Remediation

When there is a need to store or transmit sensitive data, use strong, up-to-date cryptographic algorithms to encrypt that data. Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and use well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis.  
Do not develop custom or private cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If the algorithm can be compromised if attackers find out how it works, then it is especially weak.

##### OWASP MASVS

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

##### OWASP Mobile Top 10

* M6 - Broken Cryptography

##### CWE

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

### Verifying the Configuration of Cryptographic Standard Algorithms

#### Overview

-- TODO [Provide a general description of the issue "Verifying the Configuration of Cryptographic Standard Algorithms"] --

#### Static Analysis

-- TODO [Describe Static Analysis on Verifying the Configuration of Cryptographic Standard Algorithms : how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Clarify the purpose of "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Develop Static Analysis with source code of "Verifying the Configuration of Cryptographic Standard Algorithms"] --

##### Without Source Code

-- TODO [Develop Static Analysis without source code of "Verifying the Configuration of Cryptographic Standard Algorithms"] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Verifying the Configuration of Cryptographic Standard Algorithms" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying the Configuration of Cryptographic Standard Algorithms".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" to OWASP MASVS] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying the Configuration of Cryptographic Standard Algorithms"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Verifying the Configuration of Cryptographic Standard Algorithms"] --

* Enjarify - <https://github.com/google/enjarify>

### 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.

#### White-box Testing

Inspect the code to identify the instances of crypto algorithms throughout the application, and look for known weak ones, such as DES, RC2, CRC32, MD4, MD5, SHA1 and others. See "Remediation" section for a basic list of recommended algorithms.

Example of initialization of DES algorithm:

Cipher cipher = Cipher.getInstance("DES");

#### Black-box Testing

Decompile the APK and inspect the code to see if known weak crypto algorithms are in place (see "White-box Testing").

-- TODO [Give examples of black-box testing for "Testing for Insecure and/or Deprecated Cryptographic Algorithms"] --

#### Remediation

Periodically ensure that the cryptography has not become obsolete. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once regarded as strong. Examples of currently recommended algorithms[1](https://developer.android.com/reference/java/security/KeyStore.html):

* Confidentiality: AES-256
* Integrity: SHA-256, SHA-384, SHA-512
* 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

#### References

* [1](https://github.com/pillfill/hiding-passwords-android/): [Commercial National Security Algorithm Suite and Quantum Computing FAQ](https://cryptome.org/2016/01/CNSA-Suite-and-Quantum-Computing-FAQ.pdf)
* [2](https://developer.android.com/reference/java/security/KeyStore.html): [NIST Special Publication 800-57](http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf)

##### OWASP MASVS

* V3.3: "The app does not use cryptographic protocols or algorithms that are widely considered depreciated"
* V3.4: "Cryptographic modules use parameters that adhere to current industry best practices. This includes key length and modes of operation"

##### OWASP Mobile Top 10

* M6 - Broken Cryptography

##### CWE

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

##### Info

* <https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html>

##### Tools

* [QARK](https://github.com/linkedin/qark)
* [Mobile Security Framework](https://github.com/ajinabraham/Mobile-Security-Framework-MobSF)

### Testing Random Number Generation

#### Overview

When software generates predictable values in a context requiring unpredictability, it may be 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.

#### White-box Testing

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.  
Sample weak random generation code:

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);  
}

#### Black-box Testing

Knowing what type of weak PRNG is used, it can be trivial to write proof-of-concept to generate next random value based on previously observed ones, as it was done for Java Random[1](https://github.com/pillfill/hiding-passwords-android/). 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 "White-box Testing")

#### 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[2](https://developer.android.com/reference/java/security/KeyStore.html).  
In general, if a pseudo-random number generator 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[3]. A 128-bit seed is a good starting point for producing a "random enough" number.

Sample secure random generation:

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

* [1](https://github.com/pillfill/hiding-passwords-android/): [Predicting the next Math.random() in Java](http://franklinta.com/2014/08/31/predicting-the-next-math-random-in-java/)
* [2](https://developer.android.com/reference/java/security/KeyStore.html): [Generation of Strong Random Numbers](https://www.securecoding.cert.org/confluence/display/java/MSC02-J.+Generate+strong+random+numbers)
* [3]: [Proper seeding of SecureRandom](https://www.securecoding.cert.org/confluence/display/java/MSC63-J.+Ensure+that+SecureRandom+is+properly+seeded)

##### OWASP MASVS

* V3.6: "All random values are generated using a sufficiently secure random number generator"

##### OWASP Mobile Top 10

* M6 - Broken Cryptography

##### CWE

* CWE-330: Use of Insufficiently Random Values

##### Tools

* [QARK](https://github.com/linkedin/qark)

## Testing Authentication

### Verifying that Users Are Properly Authenticated

#### Overview

-- TODO [Provide a general description of the issue.] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Develop content on Verifying that Users Are Properly Authenticated with source code] --

##### Without Source Code

-- TODO [Develop content on Verifying that Users Are Properly Authenticated without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Verifying that Users Are Properly Authenticated" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue.] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update below reference "VX.Y" for "Verifying that Users Are Properly Authenticated"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying that Users Are Properly Authenticated"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Verifying that Users Are Properly Authenticated"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Session Management

#### Overview

-- TODO [Provide a general description of the issue "Testing Session Management".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Develop content on "Testing Session Management" with source code] --

##### Without Source Code

-- TODO [Develop content on "Testing Session Management" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Session Management" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Session Management".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Session Management"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Session Management"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Session Management"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing the Password Policy

#### Overview

-- TODO [Provide a general description of the issue "Testing the Password Policy".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Develop content on Testing the Password Policy with source code] --

##### Without Source Code

-- TODO [Develop content on Testing the Password Policy without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing the Password Policy" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing the Password Policy".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing the Password Policy"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing the Password Policy"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing the Password Policy"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing the Logout Functionality

#### Overview

Session termination is an important part of the session lifecycle. Reducing the lifetime of the session tokens to a minimum decreases the likelihood of a successful session hijacking attack.  
   
The scope for this test case is to validate that the application has a logout functionality and it effectively terminates the session on client and server side.

#### Testing

To verify the correct implementation of a logout functionality, dynamic analysis should be applied by using an interception proxy. This technique can be applied to both, Android and iOS platform.    
Static Analysis  
If server side code is available, it should be reviewed to validate that the session is being terminated as part of the logout functionality.  
The check needed here will be different depending on the technology used. Here are different examples on how a session can be terminated in order to implement a proper logout on server side:

* Spring (Java) - <http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/SecurityContextLogoutHandler.html>
* Ruby on Rails -  <http://guides.rubyonrails.org/security.html>
* PHP - <http://php.net/manual/en/function.session-destroy.php>
* JSF - <http://jsfcentral.com/listings/A20158?link>
* ASP.Net - <https://msdn.microsoft.com/en-us/library/ms524798(v=vs.90).aspx>
* Amazon AWS - <http://docs.aws.amazon.com/appstream/latest/developerguide/rest-api-session-terminate.html>

#### Dynamic Analysis

For a dynamic analysis of the application an interception proxy should be used. Please see section XXX on how to set it up.  
The following steps can be applied to check if the logout is implemented properly.

1. Log into the application.
2. Do a couple of operations that require authentication inside the application.
3. Perform a logout operation.
4. Resend one of the operations detailed in step 2 using an interception proxy. For example, with Burp Repeater. The purpose of this is to send to the server a request with the token that has been invalidated in step 3.  
      
   If the session is correctly terminated on the server side, either an error message or redirect to the login page will be sent back to the client. On the other hand, if you have the same response you had in step 2, then, this session is still valid and has not been correctly terminated on the server side.  
   A detailed explanation with more test cases, can also be found in the OWASP Web Testing Guide (OTG-SESS-006) [1](https://github.com/pillfill/hiding-passwords-android/).

#### Remediation

One of the most common errors done by developers to a logout functionality is simply not destroying the session object in the server side. This leads to a state where the session is still alive even though the user logs out of the application. The session remains alive, and if an attacker get’s in possession of a valid session he can still use it and a user cannot even protect himself by logging out or if there are no session timeout controls in place.  
   
To mitigate it, the logout function on the server side must invalidate this session identifier immediately after logging out to prevent it to be reused by an attacker that could have intercepted it.  
   
Related to this, it must be checked that after calling an operation with an expired token, the application does not generate another valid token. This could lead to another authentication bypass.  
   
Many Apps do not automatically logout a user, because of customer convenience. The user logs in once, afterwards a token is generated on server side and stored within the applications internal storage and used for authentication when the application starts instead of asking again for user credentials. There should still be a logout function available within the application and this should work according to best practices by also destroying the session on server side.

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing the Logout Functionality"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing the Logout Functionality"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.owasp.org/index.php/Testing_for_logout_functionality_(OTG-SESS-006)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) <https://www.owasp.org/index.php/Session_Management_Cheat_Sheet>

##### Tools

-- TODO [Add relevant tools for "Testing the Logout Functionality"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Excessive Login Attempts

#### Overview

-- TODO [Provide a general description of the issue "Testing Excessive Login Attempts".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark on "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Develop content on "Testing Excessive Login Attempts" with source code] --

##### Without Source Code

-- TODO [Develop content on "Testing Excessive Login Attempts" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Excessive Login Attempts" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Excessive Login Attempts".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Excessive Login Attempts"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Excessive Login Attempts"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Excessive Login Attempts"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Biometric Authentication

#### Overview

-- TODO [Provide a general description of the issue "Testing Biometric Authentication".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Develop content on "Testing Biometric Authentication" with source code] --

##### Without Source Code

-- TODO [Develop content on "Testing Biometric Authentication" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Biometric Authentication" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Biometric Authentication".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Biometric Authentication"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Biometric Authentication"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Biometric Authentication"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing the Session Timeout

#### Overview

Compared to web applications most mobile applications don’t have a session timeout mechanism that terminates the session after some period of inactivity and force the user to login again. For most mobile applications users need to enter the credentials once. After authenticating on server side an access token is stored on the device which is used to authenticate. If the token is about to expire the token will be renewed without entering the credentials again. Applications that handle sensitive data like patient data or critical functions like financial transactions should implement a session timeout as a security-in-depth measure that forces users to re-login after a defined period.  
   
We will explain here how to check that this control is implemented correctly, both in the client and server side.

#### Testing

To test this, dynamic analysis is an efficient option, as it is easy to validate if this feature is working or not at runtime using an interception proxy. This is similar to test case OMTG-AUTH-002 (Testing the Logout Functionality), but we need to leave the application in idle for the period of time required to trigger the timeout function. Once this condition has been launched, we need to validate that the session is effectively terminated on client and server side.  
This technique can be applied to both, Android and iOS platform.

#### Static Analysis

If server side code is available, it should be reviewed that the session timeout functionality is correctly configured and a timeout is triggered after a defined period of time.    
The check needed here will be different depending on the technology used. Here are different examples on how a session timeout can be configured:

* Spring (Java) - <http://docs.spring.io/spring-session/docs/current/reference/html5/>
* Ruby on Rails -  <https://github.com/rails/rails/blob/318a20c140de57a7d5f820753c82258a3696c465/railties/lib/rails/application/configuration.rb#L130>
* PHP - <http://php.net/manual/en/session.configuration.php#ini.session.gc-maxlifetime>
* ASP.Net - <https://msdn.microsoft.com/en-GB/library/system.web.sessionstate.httpsessionstate.timeout(v=vs.110).aspx>
* Amazon AWS - <http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/config-idle-timeout.html>  
     
  Some applications also have an autologoff functionality in the client side. This is not a mandatory feature, but helps to improve to enforce a session timeout.  To implement this, the client side needs to control the timestamp when the screen has been displayed, and check continuously if the time elapsed is lower than the defined timeout. Once that time matches or excesses the timeout, the logoff method will be invoked, sending a signal to the server side to terminate the session and redirecting the customer to an informative screen.  
  For Android the following code might be used to implement it [3]:

public class TestActivity extends TimeoutActivity {  
@Override protected void onTimeout() {  
// logout  
}  
@Override protected long getTimeoutInSeconds() {  
return 15 \* 60; // 15 minutes  
}

#### Dynamic Analysis

For a dynamic analysis of the application an interception proxy should be used. Please see section XXX on how to set it up.  
The following steps can be applied to check if the session timeout is implemented properly.

* Log into the application.
* Do a couple of operations that require authentication inside the application.
* Leave the application in idle until the session expires (for testing purposes, a reasonable timeout can be configured, and amended later in the final version)  
     
  Resend one of the operations executed in step 2 using an interception proxy. For example, with Burp Repeater. The purpose of this is to send to the server a request with the session ID that has been invalidated when the session has expired.  
  If session timeout has been correctly configured on the server side, either an error message or redirect to the login page will be sent back to the client. On the other hand, if you have the same response you had in step 2, then, this session is still valid, which means that the session timeout control is not configured correctly.  
  More information can also be found in the OWASP Web Testing Guide (OTG-SESS-007) [1](https://github.com/pillfill/hiding-passwords-android/).

#### Remediation

Most of the frameworks have a parameter to configure the session timeout. This parameter should be set accordingly to the best practices specified of the documentation of the framework. The best practice timeout setting may vary between 5 to 30 minutes, depending on the sensitivity of your application and the use case of it.  
Regarding autologoff, the pseudocode of the implementation should be as follow:

Function autologoff  
    Get timestamp\_start  
    While application\_is\_running  
        time=timestamp-timestamp\_start  
        If time=logoff\_condition  
            Call logoff  
        EndIf  
    EndWhile  
End

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing the Session Timeout"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing the Session Timeout"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) OWASP web application test guide <https://www.owasp.org/index.php/Test_Session_Timeout_(OTG-SESS-007)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) OWASP Session management cheatsheet <https://www.owasp.org/index.php/Session_Management_Cheat_Sheet>

##### Tools

-- TODO [Add relevant tools for "Testing the Session Timeout"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing 2-Factor Authentication

#### Overview

-- TODO [Provide a general description of the issue "Testing 2-Factor Authentication".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark on "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Develop content on Testing 2-Factor Authentication with source code] --

##### Without Source Code

-- TODO [Develop content on Testing 2-Factor Authentication without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing 2-Factor Authentication" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing 2-Factor Authentication".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing 2-Factor Authentication"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing 2-Factor Authentication"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing 2-Factor Authentication"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Step-up Authentication

#### Overview

-- TODO [Provide a general description of the issue "Testing Step-up Authentication".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark on "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Develop content on Testing Step-up Authentication with source code] --

##### Without Source Code

-- TODO [Develop content on Testing Step-up Authentication without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Step-up Authentication" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Step-up Authentication".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Step-up Authentication"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Step-up Authentication"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Step-up Authentication"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing User Device Management

#### Overview

-- TODO [Provide a general description of the issue "Testing User Device Management".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm remark on "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

--TODO [Develop content on Testing User Device Management with source code] --

##### Without Source Code

--TODO [Develop content on Testing User Device Management without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing User Device Management" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing User Device Management".] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing User Device Management"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing User Device Management"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing User Device Management"] --

* Enjarify - <https://github.com/google/enjarify>

## Testing Network Communication

### Testing for Unencrypted Sensitive Data on the Network

#### Overview

A functionality of most mobile applications requires sending or receiving information from services on the Internet. This reveals another surface of attacks aimed at data on the way. It's possible for an attacker to sniff or even modify (MiTM attacks) an unencrypted information if he controls any part of network infrastructure (e.g. an WiFi Access Point) [1](https://github.com/pillfill/hiding-passwords-android/). For this reason, developers should make a general rule, that any confidential data cannot be sent in a cleartext [2](https://developer.android.com/reference/java/security/KeyStore.html).

#### White-box Testing

Identify all external endpoints (backend APIs, third-party web services), which communicate with tested application and ensure that all those communication channels are encrypted.

#### Black-box Testing

The recommended approach is to intercept all network traffic coming to or from tested application and check if it is encrypted. A network traffic can be intercepted using one of the following approaches:

* Capture all network traffic, using Tcpdump. You can begin live capturing via command:
* adb shell "tcpdump -s 0 -w - | nc -l -p 1234"  
  adb forward tcp:1234 tcp:1234

Then you can display captured traffic in a human-readable way, using Wireshark

nc localhost 1234 | sudo wireshark -k -S -i –

* Capture all network traffic using intercept proxy, like OWASP ZAP [3] or Burp Suite [4] and observe whether all requests are using HTTPS instead of HTTP.

Please note, that some applications may not work with proxies like Burp or ZAP (because of customized HTTP/HTTPS implementation, or Cert Pinning). In such case you may use a VPN server to forward all traffic to your Burp/ZAP proxy. You can easily do this, using Vproxy.

It is important to capture all traffic (TCP and UDP), so you should run all possible functions of tested application after starting interception. This should include a process of patching application, because sending a patch to application via HTTP may allow an attacker to install any application on victim's device (MiTM attacks).

#### Remediation

Ensure that sensitive information is being sent via secure channels, using HTTPS [5], or SSLSocket [6] for socket-level communication using TLS.

Please be aware that SSLSocket **does not** verify hostname. The hostname verification should be done by using getDefaultHostnameVerifier() with expected hostname. Here [7] you can find an example of correct usage.

Some applications may use localhost address, or binding to INADDR\_ANY for handling sensitive IPC, what is bad from security perspective, as this interface is accessible for other applications installed on a device. For such purpose developers should consider using secure Android IPC mechanism [8].

#### OWASP MASVS

V5.1: "Sensitive data is encrypted on the network using TLS. The secure channel is used consistently throughout the app."

#### CWE

* CWE-319 - Cleartext Transmission of Sensitive Information - <https://cwe.mitre.org/data/definitions/319.html>

#### OWASP Mobile Top 10 2014

M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

#### References

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://cwe.mitre.org/data/definitions/319.html>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) <https://developer.android.com/training/articles/security-tips.html#Networking>
* [3] <https://security.secure.force.com/security/tools/webapp/zapandroidsetup>
* [4] <https://support.portswigger.net/customer/portal/articles/1841101-configuring-an-android-device-to-work-with-burp>
* [5] <https://developer.android.com/reference/javax/net/ssl/HttpsURLConnection.html>
* [6] <https://developer.android.com/reference/javax/net/ssl/SSLSocket.html>
* [7] <https://developer.android.com/training/articles/security-ssl.html#WarningsSslSocket>
* [8] <https://developer.android.com/reference/android/app/Service.html>

#### Tools

* Tcpdump - <http://www.androidtcpdump.com/>
* Wireshark - <https://www.wireshark.org/>
* OWASP ZAP - <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project>
* Burp Suite - <https://portswigger.net/burp/>
* Vproxy - <https://github.com/B4rD4k/Vproxy>

### Verifying the TLS Settings

#### Overview

Using encryption is essential when you are sending confidential data. However, encryption can defend your privacy, only if it uses enough strong cryptography. To reach this goal SSL-based services should not offer the possibility to choose weak cipher suite. A cipher suite is specified by an encryption protocol (e.g. DES, RC4, AES), the encryption key length (e.g. 40, 56, or 128 bits), and a hash algorithm (e.g. SHA, MD5) used for integrity checking. To ensure, that your encryption cannot be easily defeated, you should verify your TLS configuration that it does not use any weak cipher/protocol/key [1](https://github.com/pillfill/hiding-passwords-android/).

#### Static Analysis

Static analysis is not applicable for this point.

#### Dynamic Analysis

After identifying all servers communicating with your application (e.g. using Tcpdump, or Burp Suite) you should verify if a server/-s allow for using weak cipher/protocol/key. It can be done, using different tools:

* testssl.sh: via following command:

testssl.sh www.example.com:443

* sslyze: via following command:

sslyze --regular www.example.com:443

* O-Saft (OWASP SSL Advanced Forensic Tool): can be run in GUI mode via command:

o-saft.tcl

or via command. There are multiple options, which can be specified here [2](https://developer.android.com/reference/java/security/KeyStore.html), but the most general one, verifying certificate, ciphers and SSL connection is the following:

perl o-saft.pl +check www.example.com:443

#### Remediation

To properly configure transport layer protection for network communication, please follow the OWASP Transport Layer Protection cheat sheet [3].

#### References

##### OWASP Mobile Top 10 2014

M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

* V5.2: "The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards."

##### CWE

* CWE-327 - Use of a Broken or Risky Cryptographic Algorithm - <https://cwe.mitre.org/data/definitions/327.html>

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Testing for Weak SSL/TLS Ciphers - <https://www.owasp.org/index.php/Testing_for_Weak_SSL/TLS_Ciphers,_Insufficient_Transport_Layer_Protection_(OTG-CRYPST-001)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) O-Saft various tests - <https://www.owasp.org/index.php/O-Saft/Documentation#COMMANDS>
* [3] Transport Layer Protection Cheat Sheet - <https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet>

##### Tools

* testssl.sh- <https://testssl.sh>
* sslyze - <https://github.com/nabla-c0d3/sslyze>
* O-Saft - <https://www.owasp.org/index.php/O-Saft>

### Testing Endpoint Identify 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 decides for easier, but less secure (e.g. accepting any certificate) solutions to ease a development process what often is not fixed after going on production [1](https://github.com/pillfill/hiding-passwords-android/), exposing at the same time an application to man-in-the-middle attacks [2](https://developer.android.com/reference/java/security/KeyStore.html).

#### White-box Testing

There are 2 main issues related with validating TLS connection: the first one is verification if a certificate comes from trusted source and the second one is a check whether the endpoint server presents the right certificate [3].

##### Verifying server certificate

A mechanism responsible for verifying conditions to establish a trusted connection in Android is called TrustedManager. 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-sgined?

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

TrustManager[] trustAllCerts = new TrustManager[] {  
new X509TrustManager()  
{  
  
 public java.security.cert.X509Certificate[] getAcceptedIssuers()  
 {  
 return new java.security.cert.X509Certificate[] {};  
 }  
 public void checkClientTrusted(X509Certificate[] chain,  
 String authType) throws CertificateException  
 {  
  
 }  
 public void checkServerTrusted(X509Certificate[] chain,  
 String authType) throws CertificateException  
 {  
  
 }  
  
}};  
  
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, SSLSession session)  
 {  
 return true;  
 }  
};

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

HostnameVerifier NO\_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory  
 .ALLOW\_ALL\_HOSTNAME\_VERIFIER;

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

#### Black-box Testing

Improper certificate verification may be found using static or dynamic analysis.

* Static analysis approach is to decompile an application and simply look in a code for TrustManager and HostnameVerifier usage. You can find insecure usage examples in a "White-box Testing" section above. Such checks of improper certificate verification, may be done automatically, using a tool called MalloDroid [4]. It simply decompiles an application and warns you if it finds something suspicious. To run it, simply type this command:

./mallodroid.py -f ExampleApp.apk -d ./outputDir

Now, you should be warned if any suspicious code was found by MalloDroid and in ./outputDir you will find decompiled application for further manual analysis.

* Dynamic analysis approach will require usage of intercept proxy, e.g. Burp Suite. 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 you listener and click Edit button. 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 you listener and click Edit button. Then go to Certificate tab, check 'Generate a CA-signed certificate with a specific hostname' and type 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 you listener and click Edit button. Then go to Certificate tab, check 'Generate a CA-signed certificate with a specific hostname' and type 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.

**Note**, if you are interested in further MITM analysis or you face any problems with configuration of your intercept proxy, you may consider using Tapioca [6]. It's a CERT preconfigured VM appliance [7] for performing MITM analysis of software. All you have to do is deploy a tested application on emulator and start capturing traffic [8].

#### Remediation

Ensure, that the hostname and certificate is verified correctly. You can find a help how to overcome common TLS certificate issues here [2](https://developer.android.com/reference/java/security/KeyStore.html).

#### 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>

#### OWASP Mobile Top 10 2014

M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

#### References

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.owasp.org/images/7/77/Hunting_Down_Broken_SSL_in_Android_Apps_-_Sascha_Fahl%2BMarian_Harbach%2BMathew_Smith.pdf>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) <https://cwe.mitre.org/data/definitions/295.html>
* [3] <https://developer.android.com/training/articles/security-ssl.html>
* [4] <https://github.com/sfahl/mallodroid>
* [5] <https://support.portswigger.net/customer/portal/articles/1841101-configuring-an-android-device-to-work-with-burp>
* [6] <https://insights.sei.cmu.edu/cert/2014/08/-announcing-cert-tapioca-for-mitm-analysis.html>
* [7] <http://www.cert.org/download/mitm/CERT_Tapioca.ova>
* [8] <https://insights.sei.cmu.edu/cert/2014/09/-finding-android-ssl-vulnerabilities-with-cert-tapioca.html>

### Testing Custom Certificate Stores and SSL Pinning

#### Overview

Certificate pinning allows to hard-code in the client the certificate 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 take the CA out of games. Mobile applications that implements certificate pinning only have to connect to a limited numbers of server, so a small list of trusted CA can be 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 certificate is in .bks format
3. Pin the certificate to an instance of the default Apache Httpclient.

To analyze the correct implementations of the 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);

#### Dynamic Analysis

Black-box Testing can be performed by launching a MITM attack using your prefered Web Proxy to intercept [1](https://github.com/pillfill/hiding-passwords-android/) the traffic exchanged between client (mobile application) and the backend server. If the Proxy is unable to intercept the HTTP requests/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 [2](https://developer.android.com/reference/java/security/KeyStore.html).

#### References

##### OWASP Mobile Top 10 2014

M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### 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 - <https://cwe.mitre.org/data/definitions/295.html>

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) - Setting Burp Suite as a proxy for Android Devices: <https://support.portswigger.net/customer/portal/articles/1841101-configuring-an-android-device-to-work-with-burp>)
* [2](https://developer.android.com/reference/java/security/KeyStore.html) - OWASP Certificate Pinning for Android: <https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#Android>

### Verifying that Critical Operations Use Secure Communication Channels

#### Overview

For sensitive applications, like banking apps, OWASP MASVS introduces "Defense in Depth" verification level [1](https://github.com/pillfill/hiding-passwords-android/). Critical operations (e.g. user enrollment, or account recovery) of such sensitive applications are the most attractive targets from attacker's perspective. This creates a need of implementing advanced security controls for such operations, like adding additional channels (e.g. SMS and e-mail) to confirm user's action. Additional channels may reduce a risk of many attacking scenarios (mainly phishing), but only when they are out of any security faults.

#### Static Analysis

Review the code and identify those parts of a code which refers to critical operations. Verify if it uses additional channels to perform such operation. Examples of additional verification channels are following:

* token (e.g. RSA token, yubikey)
* push notification (e.g. Google Prompt)
* SMS
* email
* data from another website you had to visit/scan
* data from a physical letter or physical entry point (e.g.: data you receive only after signing a document at the office of a bank)

#### Dynamic Analysis

Identify all critical operations implemented in tested application (e.g. user enrollment, or account recovery, money transfer etc.). Ensure that each of critical operations, requires at least one additional channel (e.g. SMS, e-mail, token etc.). Verify if usage of such channel can be bypassed (e.g. turning off SMS confirmation without using any other channel).

#### Remediation

Ensure that critical operations require at least one additional channel to confirm user's action. Each channel must not be bypassed to execute a critical operation. If you are going to implement additional factor to verify user's identity, you may consider usage of Infobip 2FA library [2](https://developer.android.com/reference/java/security/KeyStore.html), one-time passcodes via Google Authenticator [3].

#### References

##### OWASP Mobile Top 10 2014

M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

* V5.5 "The app doesn't rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery."

##### CWE

* CWE-956 - Software Fault Patterns (SFPs) within the Channel Attack cluster - <https://cwe.mitre.org/data/definitions/956.html>

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) The Mobile Application Security Verification Standard - <https://github.com/OWASP/owasp-masvs/blob/master/Document/0x03-Using_the_MASVS.md>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Infobip 2FA library - <https://2-fa.github.io/libraries/android-library.html>
* [3] Google Authenticator for Android - <https://github.com/google/google-authenticator-android>

## Testing Platform Interaction

### 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 certificated as the application that declared the permission. Example: android.permission.ACCESS\_DOWNLOAD\_MANAGER

Full list of Android Permissions [here](https://developer.android.com/reference/android/Manifest.permission.html#ACCESS_LOCATION_EXTRA_COMMANDS).

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 permission can be easily defined in AndroidManifest.xml file, 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 define explicitly for its purpose with meaningful and accurate label and description.

Below is an example of a custom permission START\_MAIN\_ACTIVITY that 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 easily 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\_ACTIVTY is created, apps can request it using the uses-permision 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

##### With Source Code

###### Android Permissions

Permissions should be checked if they are really need 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 developer of the intention of every permission and removed those that are not needed.

###### Custom Permissions

Apart from enforcing custom permissions via application manifest file, it can also be enforce programmatically. This is not recommended as this can lead to permission leaking and perform an unauthorized operation. This can be verified by inspecting whether if all defined custom permission were enforce in android manifest file.

int canProcess = checkCallingOrSelfPermission(  
“com.example.perm.READ\_INCOMING\_MSG”);  
if (canProcess != PERMISSION\_GRANTED)  
throw new SecurityException();

##### Without Source Code

To review application permissions via Android Manifest file, the APK file will need to be unpacked with apktool. It will then generate a folder that contains the Android Manifest file.

$apktool d test.apk  
  
I: Using Apktool 2.2.1 on test.apk  
I: Loading resource table...  
I: Decoding AndroidManifest.xml with resources...  
I: Loading resource table from file: /Users/tnayr/Library/apktool/framework/1.apk  
I: Regular manifest package...  
I: Decoding file-resources...  
I: Decoding values \*/\* XMLs...  
I: Baksmaling classes.dex...  
I: Baksmaling classes2.dex...  
I: Copying assets and libs...  
I: Copying unknown files…  
I: Copying original files...

Within the manifest file, requested permissions will be declared as uses-permissions tag.

<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
 package="com.owasp.mstg.myapp" >  
 <uses-permission android:name="android.permission.RECEIVE\_SMS" />  
 ...  
</manifest>

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

#### Dynamic Analysis

Dynamic analysis is not applicable and a solid statement and result for this test case can only be done after reviewing the Android Manifest. See "Static Analysis" for details.

#### Remediation

Only permissions that are used within the app should be requested in the Android Manifest. All other permissions should be removed.

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 concerning this section] --

##### OWASP MASVS

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

##### CWE

-- TODO [Add reference to relevant CVE(s) : titles, links, ...] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Android Permissions - <https://developer.android.com/guide/topics/permissions/requesting.html>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Custom Permissions - <https://developer.android.com/guide/topics/permissions/defining.html>
* [3] 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

-- TODO [Add link to relevant tools] --

### Testing Input Validation and Sanitization

#### Overview

-- TODO [Provide a general description of the issue.] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Clarify the purpose of "[Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/).]" ] --

##### With Source Code

-- TODO [Develop content for "Testing Input Validation and Sanitization" with source code] --

##### Without Source Code

-- TODO [Develop content for "Testing Input Validation and Sanitization" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue.] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to MX OWASP Mobile Top 10 2014] --

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update below "X.Y" reference to MASVS] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add links and titles to relevant CWE] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add links to relevant tools] --

* Enjarify - <https://github.com/google/enjarify>

### 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&messsage=I%20QUIT!&sendImmediately=true. Using something like this embedded as a link on a web page, when clicked by a victim on their mobile device, calling the custom URI with maliciously crafted parameters might trigger an SMS to be sent by the vulnerable SMS application with attacker defined content.

For any application, each of these custom URL schemes needs to be enumerated, and the actions they perform need to be tested.

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Clarify the purpose of "[Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/).]" ] --

##### With Source Code

-- TODO [Develop content for "Testing Custom URL Schemes" with source code] --

##### Without Source Code

-- TODO [Develop content for "Testing Custom URL Schemes" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue.] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for the "Testing Custom URL Schemes" topic] --

##### OWASP MASVS

* V6.3: "The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected."

##### CWE

-- TODO [Add link to relevant CWE for "Testing Custom URL Schemes"]

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add link to tools for "Testing Custom URL Schemes"] --

### Testing For Sensitive Functionality Exposure Through IPC

#### Overview

-- TODO [Provide a general description of the issue.] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Clarify purpose of "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for "Testing For Sensitive Functionality Exposure Through IPC" with source code] --

##### Without Source Code

-- TODO [Add content for "Testing For Sensitive Functionality Exposure Through IPC" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue.] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for the "Testing For Sensitive Functionality Exposure Through IPC" topic] --

##### OWASP MASVS

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

##### CWE

-- TODO [Add links and titles for CWE related to the "Testing For Sensitive Functionality Exposure Through IPC" topic] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add links to relevant tools for the "Testing For Sensitive Functionality Exposure Through IPC" topic] --

### 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 it's 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 to attack the App and gain access to it's data.

#### Static Analysis

##### With Source Code

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("http://slashdot.org/");

Different settings can be applied to the WebView of which one is able 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 to check if JavaScript is activated.

webview.getSettings().setJavaScriptEnabled(true);

This allows the WebView to interpret JavaScript and execute it's command.

##### Without Source Code

-- TODO [Add content on "Testing JavaScript Execution in WebViews" without source code] --

#### 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) vulnerability 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[4].
* The HTTPS communication need to be implemented according to best practices to avoid MITM attacks. This means:
* whole communication is encrypted via TLS (see OMTG-NET-001),
* the certificate is checked properly (see OMTG-NET-002) and/or
* the certificate is even pinned (see OMTG-NET-004)
* Only files within the App data directory should be rendered in a WebView (see OMTG-ENV-007).

#### 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 (see also OMTG-NET-001) to protect the 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()[2](https://developer.android.com/reference/java/security/KeyStore.html) 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[3].

#### References

##### OWASP Mobile Top 10 2014

* M7 - Client Side Injection

##### 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>

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) setJavaScriptEnabled in WebViews - <https://developer.android.com/reference/android/webkit/WebSettings.html#setJavaScriptEnabled(boolean)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) clearCache() in WebViews - <https://developer.android.com/reference/android/webkit/WebView.html#clearCache(boolean)>
* [3] WebView Best Practices - <https://developer.android.com/training/articles/security-tips.html#WebView>
* [4] Stored Cross-Site Scripting - <https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_(OTG-INPVAL-002)>

##### Tools

-- TODO [Add link to tools for "Testing JavaScript Execution in WebViews"] --

### Testing WebView Protocol Handlers

#### Overview

Several schemas are available by default in an URI on Android and can be triggered within a WebView[3], e.g:

* http(s):
* file:
* tel:
* geo:

When using them in a link the App can be triggered for example to access a local file when using file:///storage/emulated/0/private.xml. This can be exploited by an attacker if he is able to inject JavaScript into the Webview to access local resources via the file schema.

-- TODO [Further develop content on "Testing WebView Protocol Handlers"] --

#### Static Analysis

##### With Source Code

The following methods are available for WebViews to control access to different resources[4]:

* 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 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 *ICE\_CREAM\_SANDWICH\_MR1* and below, and false for API level *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 ICE\_CREAM\_SANDWICH\_MR1 and below, and false for API level 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.

##### Without Source Code

-- TODO [Create content on "Testing WebView Protocol Handlers" with source code] --

#### 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.

-- TODO [Further develop content on dynamic analysis for "Testing WebView Protocol Handlers" ] --

#### Remediation

Set the following best practices in order to deactivate protocol handlers, if applicable[2](https://developer.android.com/reference/java/security/KeyStore.html):

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

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[1](https://github.com/pillfill/hiding-passwords-android/).

-- TODO [How to disable tel and geo schema?] --

#### References

##### OWASP Mobile Top 10 2014

* M7 - Client Side Injection

##### 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

-- TODO [Add links and titles to relevant CWE for "Testing WebView Protocol Handlers"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) File Access in WebView - <https://developer.android.com/reference/android/webkit/WebSettings.html#setAllowFileAccess%28boolean%29>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) WebView best practices - <https://github.com/nowsecure/secure-mobile-development/blob/master/en/android/webview-best-practices.md#remediation>
* [3] Intent List - <https://developer.android.com/guide/appendix/g-app-intents.html>
* [4] WebView Settings - <https://developer.android.com/reference/android/webkit/WebSettings.html>

##### Tools

-- TODO [Add links to relevant tools for "Testing WebView Protocol Handlers"] --

### Testing for Local File Inclusion in WebViews

#### Overview

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.

-- TODO [Further develop content on the overview for "Testing for Local File Inclusion in WebViews"] --

#### Static Analysis

##### With Source Code

Check the source code for the usage of WebViews. If a WebView instance can be identified check if local files are loaded through the method loadURL()[1](https://github.com/pillfill/hiding-passwords-android/).

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.

##### Without Source Code

-- TODO [Develop content for "Testing for Local File Inclusion in WebViews" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

Create a white-list that defines the web pages and it's protocols (HTTP or HTTPS) 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 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing for Local File Inclusion in WebViews"] --

##### OWASP MASVS

* V6.7: "The app does not load user-supplied local resources into WebViews."

##### CWE

-- TODO [Add reference to relevant CWE for "Testing for Local File Inclusion in WebViews"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) loadURL() in WebView - <https://developer.android.com/reference/android/webkit/WebView.html#loadUrl(java.lang.String)>

##### Tools

-- TODO [Add links to tools for "Testing for Local File Inclusion in WebViews"] --

### Testing Whether Java Objects Are Exposed Through WebViews

#### Overview

Android offers two different ways that enables JavaScript executed in a WebView to call and use native functions within an Android App:

* shouldOverrideUrlLoading()[4]
* addJavascriptInterface()[5]

**shouldOverrideUrlLoading**

This method gives the host application a chance to take over the control when a new URL is about to be loaded in the current WebView. The method shouldOverrideUrlLoading() is available with two different method signatures:

* boolean shouldOverrideUrlLoading (WebView view, String url)
* This method was deprecated in API level 24.
* boolean shouldOverrideUrlLoading (WebView view, WebResourceRequest request)
* This method was added in API level 24

**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[2](https://developer.android.com/reference/java/security/KeyStore.html).

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

##### With Source Code

**shouldOverrideUrlLoading**

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

The following example illustrates how the method can be used.

@Override  
public boolean shouldOverrideUrlLoading (WebView view, WebResourceRequest request) {  
 URL url = new URL(request.getUrl().toString());  
 // execute functions according to values in URL  
 }  
}

If an attacker has access to the JavaScript code, for example through stored XSS or MITM, he can directly trigger native functions if the exposed Java methods are implemented in an insecure way.

window.location = http://example.com/method?parameter=value

**addJavascriptInterface**

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.

##### Without Source Code

-- TODO [Add content on "Testing Whether Java Objects Are Exposed Through WebViews" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

If shouldOverrideUrlLoading() is needed, it should be verified how the input is processed and if it's possible to execute native functions through malicious JavaScript.

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

Another compliant solution is to define the API level to 17 (JELLY\_BEAN\_MR1) and above in the manifest file of the App. For these API levels, only public methods that are annotated with JavascriptInterface can be accessed from JavaScript[1](https://github.com/pillfill/hiding-passwords-android/).

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

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for the "Testing Whether Java Objects Are Exposed Through WebViews" issue] --

##### OWASP MASVS

* V6.8: "If Java objects are exposed in a WebView, verify that the WebView only renders JavaScript contained within the app package."

##### CWE

-- TODO [Add links and titles to relevant CWE for "Testing Whether Java Objects Are Exposed Through WebViews"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) DRD13 addJavascriptInterface() - <https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=129859614>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) WebView addJavascriptInterface Remote Code Execution - <https://labs.mwrinfosecurity.com/blog/webview-addjavascriptinterface-remote-code-execution/>
* [3] Method shouldOverrideUrlLoading() - <https://developer.android.com/reference/android/webkit/WebViewClient.html#shouldOverrideUrlLoading(android.webkit.WebView,%20java.lang.String)>
* [4] Method addJavascriptInterface() - <https://developer.android.com/reference/android/webkit/WebView.html#addJavascriptInterface(java.lang.Object>, java.lang.String)

##### Tools

-- TODO [Add links to tools for "Testing Whether Java Objects Are Exposed Through WebViews"] --

### Testing Object (De-)Serialization

#### Overview

An object and it's 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 sign and encrypt serialized data but, if the source code is available, this is always reversible.

#### Static Analysis

##### With Source Code

Search the source code for the following keywords:

* import java.io.Serializable
* implements Serializable

Check if serialized data is stored temporarily or permanently within the app's data directory or external storage and if it contains sensitive data.

\*\*[https://www.securecoding.cert.org/confluence/display/java/SER04-J.+Do+not+allow+serialization+and+deserialization+to+bypass+the+security+manager\*\*](https://www.securecoding.cert.org/confluence/display/java/SER04-J.+Do+not+allow+serialization+and+deserialization+to+bypass+the+security+manager**)

##### Without Source Code

-- TODO [Create content for "Testing Object (De-)Serialization" without source code] --

#### Dynamic Analysis

-- TODO [Create content for dynamic analysis of "Testing Object (De-)Serialization" ] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Object (De-)Serialization".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing Object (De-)Serialization"] --

##### OWASP MASVS

* V6.9: "Object serialization, if any, is implemented using safe serialization APIs."

##### CWE

-- TODO [Add link and title to CWE for "Testing Object (De-)Serialization"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Update Security Provider - <https://developer.android.com/training/articles/security-gms-provider.html>

##### Tools

-- TODO [Add link to relevant tools for "Testing Object (De-)Serialization"] --

### 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

##### With Source Code

Root detection can either be implemented by leveraging existing root detection libraries, such as Rootbeer[1](https://github.com/pillfill/hiding-passwords-android/), 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.

##### Without Source Code

-- TODO [Create content for "Testing Root Detection" without source code] --

#### 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[2](https://developer.android.com/reference/java/security/KeyStore.html) 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[1](https://github.com/pillfill/hiding-passwords-android/). 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 2014

-- TODO [Add link to OWASP Mobile Top 10 2414 for "Testing Root Detection"] --

##### OWASP MASVS

* V6.10: "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

-- TODO [Add link to relevant CWE for "Testing Root Detection"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) RootBeer - <https://github.com/scottyab/rootbeer>

##### Tools

* [2](https://developer.android.com/reference/java/security/KeyStore.html) RootCloak - <http://repo.xposed.info/module/com.devadvance.rootcloak2>

## Testing Code Quality and Build Settings

### Verifying That the App is Properly Signed

#### Overview

Android requires that all APKs be 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 prevents 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 be 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) 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 is belongs to the developer.

APK signatures can be verified using the apksigner tool.

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

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

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

$ jarsigner -verify -verbose -certs example.apk   
  
sm 11116 Fri Nov 11 12:07:48 ICT 2016 AndroidManifest.xml  
  
 X.509, CN=Android Debug, O=Android, C=US  
 [certificate is valid from 3/24/16 9:18 AM to 8/10/43 9:18 AM]  
 [CertPath not validated: Path does not chain with any of the trust anchors]  
(...)

The output for an APK signed with a Release 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=Awesome Corporation, OU=Awesome, O=Awesome Mobile, L=Palo Alto, ST=CA, C=US  
 [certificate is valid from 9/1/09 4:52 AM to 9/26/50 4:52 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. If you don't have the APK available locally, pull it from the device first:

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

#### Remediation

Developers need to make sure that release builds are signed with the appropriate certificate from the release keystore. In Android Studio, this can be done manually or by configuring creating a signing configuration and assigning it to the release build type [2](https://developer.android.com/reference/java/security/KeyStore.html).  
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

#### References

##### OWASP Mobile Top 10 2016

M7 - Client Code Quality

##### OWASP MASVS

* V7.1: "The app is signed and provisioned with valid certificate."

##### CWE

N/A

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Sign your App - <https://developer.android.com/studio/publish/app-signing.html>

##### Tools

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

### Testing If the App is Debuggable

#### Overview

The android:debuggable attiribute in the Application tag 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

Attempt to the running process with jdb. If debugging is disallowed, this should fail with the following error:

#### Remediation

Set the android:debuggable to false, or simply leave omit it from the Application tag.

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing If the App is Debuggable"] --

* VX.Y: ""

##### CWE

-- TODO [Add relevant CWE for "Testing If the App is Debuggable"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Application element - <https://developer.android.com/guide/topics/manifest/application-element.html>

### Testing for Debugging Symbols

#### Overview

-- TODO [Give an overview about the functionality and it's potential weaknesses] --

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

#### Static Analysis

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

To display debug symbols:

export $NM = $ANDROID\_NDK\_DIR/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86\_64/bin/arm-linux-androideabi-nm

$ $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

#### 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 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing for Debugging Symbols"] --

* VX.Y: ""

##### CWE

-- TODO [Add relevant CWE for "Testing for Debugging Symbols"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* Debugging with Android Studio - <http://developer.android.com/tools/debugging/debugging-studio.html>

##### Tools

-- TODO [Add relevant tools for "Testing for Debugging Symbols"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing for Debugging Code and Verbose Error Logging

#### Overview

-- TODO [Give an overview about the functionality and it's potential weaknesses] --

#### White-box Testing

-- TODO [Add content on white-box testing for "Testing for Debugging Code and Verbose Error Logging"] --

#### Black-box Testing

-- TODO [Add content on black-box testing for "Testing for Debugging Code and Verbose Error Logging"] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing for Debugging Code and Verbose Error Logging"] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing for Debugging Code and Verbose Error Logging"] --

* VX.Y: ""

##### CWE

-- TODO [Add relevant CWE for "Testing for Debugging Code and Verbose Error Logging"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* Debugging with Android Studio - <http://developer.android.com/tools/debugging/debugging-studio.html>

##### Tools

-- TODO [Add relevant tools for "Testing for Debugging Code and Verbose Error Logging"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Exception Handling

#### Overview

-- TODO [Give an overview about the functionality and it's potential weaknesses] --

#### White-box Testing

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

* Verify that the application use a [well-designed] (<https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=18581047>) (an unified) scheme to handle exceptions.
* Verify that the application doesn't expose sensitive information while handeling exceptions, but are still verbose enough to explain the issue to the user.
* C3

#### Black-box Testing

-- TODO [Describe how to test for this issue using static and dynamic analysis techniques. This can include everything from simply monitoring aspects of the app’s behavior to code injection, debugging, instrumentation, etc. ] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Exception Handling"] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Exception Handling"] --

* VX.Y: ""

##### CWE

-- TODO [Add relevant CWE for "Testing Exception Handling"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* Debugging with Android Studio - <http://developer.android.com/tools/debugging/debugging-studio.html>

##### Tools

-- TODO [Add relevant tools for "Testing Exception Handling"] --

* Enjarify - <https://github.com/google/enjarify>

### Verifying Compiler Settings

#### Overview

Since most Android applications are Java based, they are [immunue](https://www.owasp.org/index.php/Reviewing_Code_for_Buffer_Overruns_and_Overflows#.NET_.26_Java) to buffer overflow vulnerabilities.

#### White-box Testing

-- TODO [Describe how to assess this with access to the source code and build configuration] --

#### Black-box Testing

-- TODO [Describe how to test for this issue using static and dynamic analysis techniques. This can include everything from simply monitoring aspects of the app’s behavior to code injection, debugging, instrumentation, etc. ] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying Compiler Settings"] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Verifying Compiler Settings"] --

* VX.Y: ""

##### CWE

-- TODO [Add relevant CWE for "Verifying Compiler Settings"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* Debugging with Android Studio - <http://developer.android.com/tools/debugging/debugging-studio.html>

##### Tools

-- TODO [Add relevant tools for "Verifying Compiler Settings"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing for Memory Management Bugs

#### Overview

-- TODO [Give an overview about the functionality and it's potential weaknesses] --

#### White-box Testing

-- TODO [Add content for white-box testing "Testing for Memory Management Bugs"] --

#### Black-box Testing

-- TODO [Add content for black-box testing "Testing for Memory Management Bugs"] --

#### Remediation

-- TODO [Add remediations for "Testing for Memory Management Bugs"] --

#### References

##### OWASP Mobile Top 10 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

* V7.7: "In unmanaged code, memory is allocated, freed and used securely."

##### CWE

-- TODO [Add relevant CWE for "Testing for Memory Management Bugs"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* Debugging with Android Studio - <http://developer.android.com/tools/debugging/debugging-studio.html>

##### Tools

-- TODO [Add relevant tools for "Testing for Memory Management Bugs"] --

* Enjarify - <https://github.com/google/enjarify>

### Verifying that Java Bytecode Has Been Minified

#### Overview

Because 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.

#### White-box Testing

If source code is provided, build.gradle file can be check to see if obfuscation settings are set. From the example below, we can see that minifyEnabled and proguardFiles are set. It is common to see application exempts some class from obfuscation with "-keepclassmembers" and "-keep class", so it is important to audit proguard configuration file to see what class are exempted. The getDefaultProguardFile('proguard-android.txt') method gets the default ProGuard settings from the Android SDK tools/proguard/ folder and proguard-rules.pro is where you defined custom proguard rules. From our sample proguard-rules.pro file, we can see that many classes that extend common android classes are exempted, which should be done more granular on exempting specific classes or library.

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

#### Black-box Testing

If source code is not provided, apk can be decompile to verify if codebase have been obfuscated. dex2jar can be used to convert dex code to jar file. Tools like JD-GUI can be used to check if class, method and variable name is 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, obfuscator, and preverifier. 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 2014

* MX - Title - Link
* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Verifying that Java Bytecode Has Been Minified"] --

* VX.Y: ""

##### CWE

-- TODO [Add relevant CWE for Verifying that Java Bytecode Has Been Minified] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Configuring your application for release - <http://developer.android.com/tools/publishing/preparing.html#publishing-configure>
* Debugging with Android Studio - <http://developer.android.com/tools/debugging/debugging-studio.html>

##### Tools

-- TODO [Add relevant tools for Verifying that Java Bytecode Has Been Minified] --

* Enjarify - <https://github.com/google/enjarify>

## Testing Resiliency Against Reverse Engineering

### 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 of the 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 [1](https://github.com/pillfill/hiding-passwords-android/).

###### 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" [2](https://developer.android.com/reference/java/security/KeyStore.html).

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 vaidation 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 [3].

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 uses the stat system call to retrieve information about a file (example code adapted from rootinspector [9]), and returns 1 if the file exists.

jboolean Java\_com\_example\_statfile(JNIEnv \* env, jobject this, jstring filepath) {  
 jboolean fileExists = 0;  
 jboolean isCopy;  
 const char \* path = (\*env)->GetStringUTFChars(env, filepath, &isCopy);  
 struct stat fileattrib;  
 if (stat(path, &fileattrib) < 0) {  
 \_\_android\_log\_print(ANDROID\_LOG\_DEBUG, DEBUG\_TAG, "NATIVE: stat error: [%s]", strerror(errno));  
 } else  
 {  
 \_\_android\_log\_print(ANDROID\_LOG\_DEBUG, DEBUG\_TAG, "NATIVE: stat success, access perms: [%d]", fileattrib.st\_mode);  
 return 1;  
 }  
  
 return 0;  
}

**Executing su and other commands**

Another way of determining whether su exists is attempting to execute it through Runtime.getRuntime.exec(). This will throw an IOException if su is not in PATH. The same method can be used to check for other programs often found on rooted devices, such as busybox or the symbolic links that typically point to it.

**Checking running processes**

Supersu - by far the most popular rooting tool - runs an authentication daemon named daemonsu, so the presence of this process is another sign of a rooted device. Running processes can be enumerated through ActivityManager.getRunningAppProcesses() and manager.getRunningServices() APIs, the ps command, or walking through the /proc directory. As an example, this is implemented the following way in rootinspector [9]:

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 [5]. This can be checked as follows [6]:

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 [4].

##### 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 resiliency 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 enviroment 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?

Also note how well the root detection mechanisms are integrated within the overall protection scheme. For example, the detection functions should obfuscated and protected from tampering.

#### 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 mechansims, 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>

##### OWASP MASVS

* V8.3: "The app implements two or more functionally independent methods of root detection and responds to the presence of a rooted device either by alerting the user or terminating the app."

##### CWE

N/A

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) OWASP Mobile Crackmes - <https://github.com/OWASP/owasp-mstg/blob/master/OMTG-Files/02_Crackmes/List_of_Crackmes.md>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) SafetyNet Documentation - <https://developers.google.com/android/reference/com/google/android/gms/safetynet/SafetyNet>
* [3] SafetyNet: Google's tamper detection for Android - <https://koz.io/inside-safetynet/>
* [4] NetSPI Blog - Android Root Detection Techniques - <https://blog.netspi.com/android-root-detection-techniques/>
* [5] InfoSec Institute - <http://resources.infosecinstitute.com/android-hacking-security-part-8-root-detection-evasion/>
* [6] Android – Detect Root Access from inside an app - <https://www.joeyconway.com/blog/2014/03/29/android-detect-root-access-from-inside-an-app/>

##### Tools

* [7] rootbeer - <https://github.com/scottyab/rootbeer>
* [8] RootCloak - <http://repo.xposed.info/module/com.devadvance.rootcloak2>
* [9] rootinspector - <https://github.com/devadvance/rootinspector/>

### Testing Anti-Debugging

#### Overview

Debugging is a highly effective way of analyzing the runtime behaviour 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 behaviour). 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.

##### Sample Anti-JDWP-Debugging Methods

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 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 [2](https://developer.android.com/reference/java/security/KeyStore.html).

static boolean detect\_threadCpuTimeNanos(){   
 long start = Debug.threadCpuTimeNanos();  
  
 for(int i=0; i<1000000; ++i)   
 continue;  
  
 long stop = Debug.threadCpuTimeNanos();  
  
 if(stop - start < 10000000) {  
 return false;  
 }  
 else {  
 return true;  
 }

###### Messing With JDWP-related Data Structures

In Dalvik, the global virtual machine state is accessible through the DvmGlobals structure. The global variable gDvm holds a pointer to this structure. DvmGlobals contains various variables and pointers important for JDWP debugging that can be tampered with.

struct DvmGlobals {  
 /\*  
 \* Some options that could be worth tampering with :)  
 \*/  
  
 bool jdwpAllowed; // debugging allowed for this process?  
 bool jdwpConfigured; // has debugging info been provided?  
 JdwpTransportType jdwpTransport;  
 bool jdwpServer;  
 char\* jdwpHost;  
 int jdwpPort;  
 bool jdwpSuspend;  
   
 Thread\* threadList;  
   
 bool nativeDebuggerActive;  
 bool debuggerConnected; /\* debugger or DDMS is connected \*/  
 bool debuggerActive; /\* debugger is making requests \*/  
 JdwpState\* jdwpState;  
   
};

For example, setting the gDvm.methDalvikDdmcServer\_dispatch function pointer to NULL crashed the JDWP thread[2](https://developer.android.com/reference/java/security/KeyStore.html):

JNIEXPORT jboolean JNICALL Java\_poc\_c\_crashOnInit ( JNIEnv\* env , jobject ) {  
 gDvm.methDalvikDdmcServer\_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 behaviour of the debugging runtime can be manipulatedB ny 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 [3].

#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);  
  
 }  
 }  
}

##### Sample Anti-Native-Debugging Methods

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 [3].

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 [5]. 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 inticate 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 behaviour 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 enviroment. If nothing else works, you can modify the operating system to subvert the assumptions the developers made when designing the anti-debugging tricks.

###### Example: UnCrackable App for Android Level 2

-- TODO [Bypassing Debugger Detection - Solve UnCrackable 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'.

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");  
 }

-- TODO [Add a generic bypass script using Frida (?)] --

#v0.1  
   
import frida  
import sys  
   
session = frida.get\_remote\_device().attach("com.example.targetapp")  
   
script = session.create\_script("""  
   
var funcPtr = Module.findExportByName("libdvm.so", "\_Z25dvmDbgIsDebuggerConnectedv");  
Interceptor.replace(funcPtr, new NativeCallback(function (pathPtr, flags) {  
    return 0;  
}, 'int', []));  
""")   
  
def on\_message(message, data):  
    print(message)  
   
script.on('message', on\_message)  
script.load()  
sys.stdin.read()

#### 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?

Consider how the anti-debugging mechansims fit into the overall protection scheme. For example, anti-debugging defenses should obfuscated and protected from tampering.

Note that some anti-debugging implementations respond in a stealthy way so that changes in behaviour are not immediately apparent. For example, a soft token app might not visibly respond when a debugger is detected, but instead secretly alter the state of an internal variable so that an incorrect OTP is generated at a later point. Make sure to run through the complete workflow to determine if attaching the debugger causes a crash or malfunction.

#### 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 mechansims, or better integrating existing mechanisms with other defenses.

#### References

* [1](https://github.com/pillfill/hiding-passwords-android/) Matenaar et al. - Patent Application - MOBILE DEVICES WITH INHIBITED APPLICATION DEBUGGING AND METHODS OF OPERATION - <https://www.google.com/patents/US8925077>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Bluebox Security - Android Reverse Engineering & Defenses - <https://slides.night-labs.de/AndroidREnDefenses201305.pdf>
* [3] Tim Strazzere - Android Anti-Emulator - <https://github.com/strazzere/anti-emulator/>
* [4] Anti-Debugging Fun with Android ART - <https://www.vantagepoint.sg/blog/88-anti-debugging-fun-with-android-art>
* [5] ptrace man page - <http://man7.org/linux/man-pages/man2/ptrace.2.html>

### Testing File Integrity Checks

#### Overview

In the "Tampering and Reverse Engineering" chapter, we discussed Android's APK signature check and showed how to re-package and re-sign apps for reverse engineering purposes. Adding additional integrity checks to the app itself makes this process a bit 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.

##### Sample Implementation

Integrity checks usually calculate a checksum or hash over selected files. Files that are commonly protected include:

* AndroidManifest.xml
* classes.dex
* Native libraries (\*.so)

As well as any other files containing Java bytecode. The following sample implementation from the Android Cracking Blog [1](https://github.com/pillfill/hiding-passwords-android/) calculates a CRC over classes.dex and compares is 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;  
 }  
}

##### Bypassing File Integrity Checks

1. Patch out the anti-debugging functionality. Disable the unwanted behaviour 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.

#### Effectiveness Assessment

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 an cease to function.

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue.] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update "VX.Y" and description below] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

* N/A

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Android Cracking Blog - <http://androidcracking.blogspot.sg/2011/06/anti-tampering-with-crc-check.html>

##### Tools

-- TODO [Add link to relevant tool for "Testing File Integrity Checks"] --

* Enjarify - <https://github.com/google/enjarify>

### 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 as increased risk by installing such tools.

##### Detection Methods

-- TODO [Add list of tools and associated files, processes, libs, etc. etc. Cover the most important tools..."] --

* Substrate for Android
* Xposed
* Frida
* Introspy-Android
* Drozer
* RootCloak
* Android SSL Trust Killer

###### File Checks

###### Checking Running Processes

###### Checking Loaded Libraries

##### Bypassing Detection

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of sentence "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Detection of Reverse Engineering Tools".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" and description below] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Detection of Reverse Engineering Tools"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add link to tools for "Testing Detection of Reverse Engineering Tools"] --

* Enjarify - <https://github.com/google/enjarify>

### 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.

#### Detection Techniques

There are several static 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 indicaters stem from the build.prop file

API Method Value Meaning  
Build.ABI armeabi possibly emulator  
BUILD.ABI2 unknown possibly emulator  
Build.BOARD unknown emulator  
Build.Brand generic emulator  
Build.DEVICE generic emulator  
Build.FINGERPRINT generic emulator  
Build.Hardware goldfish emulator  
Build.Host android-test possibly emulator  
Build.ID FRF91 emulator  
Build.MANUFACTURER unknown emulator  
Build.MODEL sdk emulator  
Build.PRODUCT sdk emulator  
Build.RADIO unknown possibly emulator  
Build.SERIAL null emulator  
Build.TAGS test-keys emulator  
Build.USER android-build emulator

It should be noted that the build.prop file can be edited on a rooted android device, or modified when compiling AOSP from source. Either of these techniques would bypass the static string checks above.

The next set of static indicators utilize the Telephony manager. All android emulators have fixed values that this API can query.

API Value Meaning  
TelephonyManager.getDeviceId() 0's emulator  
TelephonyManager.getLine1 Number() 155552155 emulator  
TelephonyManager.getNetworkCountryIso() us possibly emulator  
TelephonyManager.getNetworkType() 3 possibly emulator  
TelephonyManager.getNetworkOperator().substring(0,3) 310 possibly emulator  
TelephonyManager.getNetworkOperator().substring(3) 260 possibly emulator  
TelephonyManager.getPhoneType() 1 possibly emulator  
TelephonyManager.getSimCountryIso() us possibly emulator   
TelephonyManager.getSimSerial Number() 89014103211118510720 emulator  
TelephonyManager.getSubscriberId() 310260000000000 emulator  
TelephonyManager.getVoiceMailNumber() 15552175049 emulator

Keep in mind that a hooking framework such as Xposed or Frida could hook this API to provide false data.

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of sentence "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

#### Bypassing Emulator Detection

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Emulator Detection".] --

#### References

* [1](https://github.com/pillfill/hiding-passwords-android/) Timothy Vidas & Nicolas Christin - Evading Android Runtime Analysis via Sandbox Detection - <https://users.ece.cmu.edu/~tvidas/papers/ASIACCS14.pdf>

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" and description] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Emulator Detection"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add links to tools for "Testing Emulator Detection"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Memory Integrity Checks

#### Overview

-- TODO [Provide a general description of the issue "Testing Memory Integrity Checks".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of sentence "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "-- TODO [Add content on "Testing Memory Integrity Checks" with source code] --".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below and description] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Memory Integrity Checks"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add link to relevant tools for "Testing Memory Integrity Checks"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Device Binding

#### Overview

-- TODO [Provide a general description of the issue "Testing Device Binding".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Device Binding".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below + description] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Device Binding"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add link to tools for "Testing Device Binding"] --

* Enjarify - <https://github.com/google/enjarify>

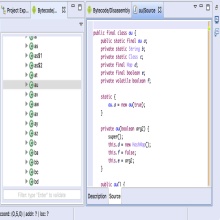
### Testing Obfuscation

#### Overview

-- TODO [Add content for overview on "Testing Obfuscation"] --

##### Simple Tricks

* Modifying the DEX file so static analysis tools can't load it;
* Using dynamic class loading and reflection to obfuscated the control flow;
* Pack or encrypt portions of the code and/or data;
* Frequently jumping between Java and native code.

  
*Identifier renaming with ProGuard.*

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of sentence "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

-- TODO [Add content on "Testing Obfuscation" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

-- TODO [Dumping process memory] --

#! /usr/bin/env python  
import re  
import sys  
  
pid = sys.argv[1]  
startaddr = sys.argv[2]  
endaddr = sys.argv[3]  
  
mem\_file = open("/proc/" + pid + "/mem", 'r', 0)  
out\_file = open("./" + pid + "\_" + startaddr + "-" + endaddr + ".dump", 'w')  
  
start = int(startaddr, 16)  
end = int(endaddr, 16)  
mem\_file.seek(start)  
chunk = mem\_file.read(end - start)  
  
out\_file.write(chunk)  
  
mem\_file.close()  
out\_file.close()

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Obfuscation".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below and description] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Obfuscation"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add links to relevant tools for "Testing Obfuscation"] --

* Enjarify - <https://github.com/google/enjarify>

# Testing Application Security on iOS

## iOS Platform Overview

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.

### The iOS Security Architecture

The core features of the iOS security architecture:

* Secure Boot
* Sandbox
* Code Signing
* Encryption and Data Protection
* General Exploit Mitigations

A very good and detailed analysis of iOS security architecture has been done by [Johnatan Levin in MacOS and iOS Internals Vol. 3](http://www.newosxbook.com/2ndUpdate.html) [4]

#### 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 devices 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. It's 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 possibilites. 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 else. 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.

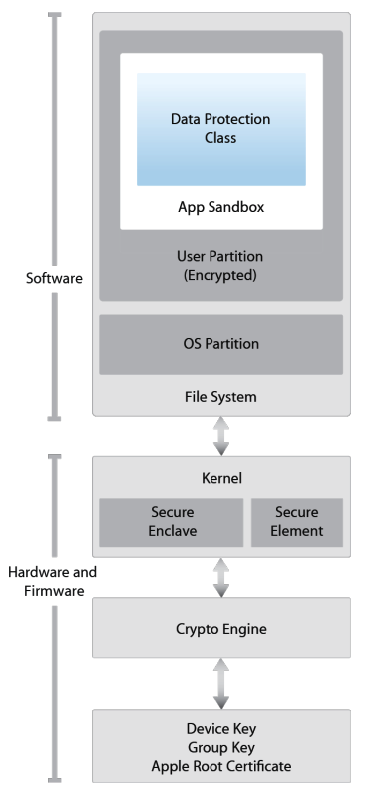
#### General Exploit Mitigations

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.

-- TODO [Further develop section on iOS General Exploit Mitigation] --

  
*iOS Security Architecture (iOS Security Guide)*

### 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.

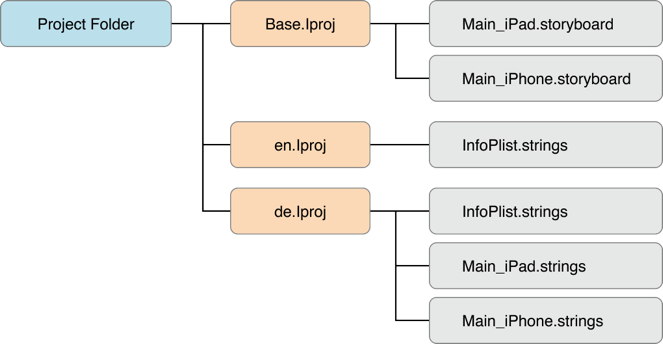
#### IPA Payloads - A Closer Look

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.



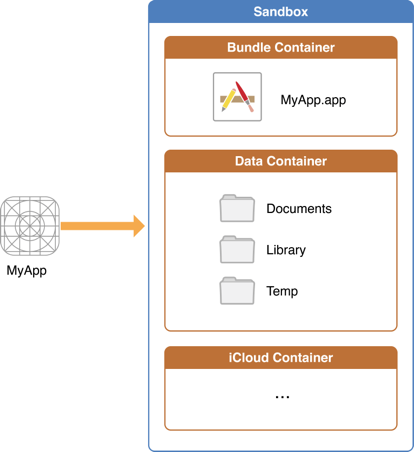
On a jailbroken device, you can recover the IPA for an installed iOS app using IPA Installer (see also [Testing Processes and Techniques](Document/0x05b-Testing-Process-and-Techniques-iOS.md)). Note that during mobile security assessments, developers will often provide you with the IPA directly. They 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 synchronise 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. [usbmux]

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 [1](https://github.com/pillfill/hiding-passwords-android/).

-- TODO [Develop section on iOS Code Signing and Encryption] --

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 releasees 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 [3]:

* The app process is restricted to it's 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.

### References

* [1](https://github.com/pillfill/hiding-passwords-android/) Apple's Crystal Prison and the Future of Open Platforms - <https://www.eff.org/deeplinks/2012/05/apples-crystal-prison-and-future-open-platforms>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Decrypting iOS binaries - <https://mandalorian.com/2013/05/03/decrypting-ios-binaries/>
* [3] Jonathan Levin, Mac OS X and iOS Internals, Wiley, 2013
* [4] Johnatan Levin, MacOS and iOS Internals, Volume III: Security & Insecurity
* [iOS Technology Overview](https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007898-CH1-SW1)
* [iOS Security Guide](https://www.apple.com/business/docs/iOS_Security_Guide.pdf)
* [How iOS Security Really Works](https://developer.apple.com/videos/play/wwdc2016/705/)
* [usbmuxd](http://www.libimobiledevice.org/)
* [usbmux](http://wikee.iphwn.org/usb:usbmux)
* [AppSync](https://cydia.angelxwind.net/?page/net.angelxwind.appsyncunified)
* [ipainstaller](https://github.com/autopear/ipainstaller)
* [Hockey Flight](https://hockeyapp.net/)
* [Testflight](https://developer.apple.com/testflight/)

## 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

**Requirements for iOS testing lab**

Bare minimum is:

* Laptop with admin rights, VirtualBox with Kali Linux
* WiFi network with client to client traffic permitted (multiplexing through USB is also possible)
* Hopper Disassembler
* At least one jailbroken iOS device (with desired iOS version)
* Burp Suite tool

Recommended is:

* Macbook with XCode and Developer's Profile
* WiFi network as previously
* Hopper Disassembler or IDA Pro with Hex Rays
* At least two iOS devices, one jailbroken, second non-jailbroken
* Burp Suite tool

### Jailbreaking iOS

In the iOS world, jailbreaking means among others disabling Apple's code signing mechanisms so that apps not signed by Apple can be run. 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.  
There's an important different between exploit chain and jailbreak. The former will disable iOS system protections like code signing or MAC, but will not install Cydia store for you. A jailbreak is a complete tool that will leverage exploit chain, disable system protections and install Cydia.

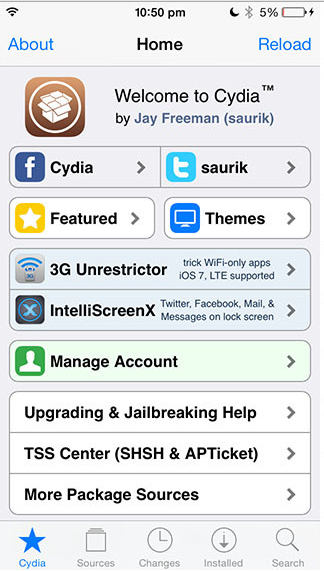
Developing a jailbreak for any given version of iOS is not an easy endeavor. As a security tester, you'll most likely want to use publicly available jailbreak tools. Even so, we recommend studying the techniques used to jailbreak various versions of iOS in the past - you'll encounter many highly interesting 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 vulnerabilty in the Photos app (CVE-2015-7037). A great book on iOS Security Internals has been written and published by Jonathan Levin. This won't be very useful for iOS application security testing, but will definitely help dive into the world of iOS exploitation and jailbreak analysis [11]

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.

Jailbreaking methods vary across iOS versions. Best choice is to check if a [public jailbreak is available for your iOS version](https://canijailbreak.com/). Beware of fake tools and spyware that is often distributed around the Internet, often hiding behind domain names similar to the jailbreaking group/author.

**Important** caveat regarding jailbreaking iOS: contrary to Android, you **can't** downgrade iOS version with one exception explained below. Naturally, this 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. One possible solution is to have at least two iOS devices: one that will be jailbroken and have all necessary tools for testing and second, 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 challenge-response process. The device will perform OS installation if and only if the response to 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 [this site](https://ipsw.me). More information on jailbreaking is available on [The iPhone Wiki](https://www.theiphonewiki.com/)

### Preparing your 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

1. Add the following repository to Cydia: https://build.frida.re
2. Install Frida from Cydia
3. Install following packages with aptitude

inetutils   
syslogd   
less   
com.autopear.installipa   
class-dump   
com.ericasadun.utilities   
odcctools  
cycript   
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

-- TODO [Add content on usbmuxd/tcprelay] --

$ ./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:

* Obtain IPA file
* Bypass jailbreak detection (if present)
* Bypass certificate pinning (if present)
* Inspect HTTP(S) traffic - usual web app test
* Abuse application logic by runtime manipulation
* Check for local data storage (caches, binary cookies, plists, databases)
* Check for client-specific bugs, e.g. SQLi, XSS
* Other checks like: logging to ASL with NSLog, application compile options, application screenshots, no app backgrounding

### Static Analysis

#### With Source Code

-- TODO [Add content on security Static Analysis of an iOS app with source code] --

#### Without Source Code

##### Folder structure

System applications can be found in /Applications  
For all the rest you can use installipa to navigate to appropriate folders [14]:

iOS8-jailbreak:~ root# installipa -l  
me.scan.qrcodereader  
iOS8-jailbreak:~ root# installipa -i me.scan.qrcodereader  
Bundle: /private/var/mobile/Containers/Bundle/Application/09D08A0A-0BC5-423C-8CC3-FF9499E0B19C  
Application: /private/var/mobile/Containers/Bundle/Application/09D08A0A-0BC5-423C-8CC3-FF9499E0B19C/QR Reader.app  
Data: /private/var/mobile/Containers/Data/Application/297EEF1B-9CC5-463C-97F7-FB062C864E56

As you can see, there are three main directories: Bundle, Application and Data. The Application directory is just a subdir 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.

##### Recovering an IPA File From an Installed App

###### From Jailbroken devices

You can use Saurik's IPA Installer to recover IPAs from apps installed on the device. To do this, install IPA installer console [1](https://github.com/pillfill/hiding-passwords-android/) via Cydia. Then, ssh into the device and look up the bundle id of the target app. For example:

iPhone:~ root# ipainstaller -l  
com.apple.Pages  
com.example.targetapp  
com.google.ios.youtube  
com.spotify.client

Generate the IPA file for using the following command:

iPhone:~ root# ipainstaller -b com.example.targetapp -o /tmp/example.ipa

###### From non-Jailbroken devices

If the app is available on itunes, you are able to recover the ipa on MacOS with the following simple steps:

* Download the app in itunes
* Go to your itunes Apps Library
* Right-click on the app and select show in finder

-- TODO [Further develop section on Static Analysis of an iOS app from non-jailbroken devices without source code] --

#### Dumping Decrypted Executables

On top of code signing, apps distributed via the app store are also protected using Apple's FairPlay DRM system. This system uses asymmetric cryptography to ensure that any app (including free apps) obtained from the app store only executes on the particular device it is approved to run on. The decryption key is unique to the device and burned into the processor. As of now, the only possible way to obtain the decrypted code from a FairPlay-decrypted app is dumping it from memory while the app is running. On a jailbroken device, this can be done with Clutch tool that is included in standard Cydia repositories [2](https://developer.android.com/reference/java/security/KeyStore.html). Use clutch in interactive mode to get a list of installed apps, decrypt them and pack to IPA file:

# Clutch -i

**NOTE:** Only applications distributed with AppStore are protected with FairPlay DRM. If you obtained your application compiled and exported directly from XCode, you don't need to decrypt it. The easiest way is to load the application into Hopper and check if it's being correctly disassembled. You can also check it with otool:

# otool -l yourbinary | grep -A 4 LC\_ENCRYPTION\_INFO

If the output contains cryptoff, cryptsize and cryptid fields, then the binary is encrypted. If the output of this comand is empty, it means that binary is not encrypted. **Remember** to use otool on binary, not on the IPA file.

#### Getting Basic Information with Class-dump and Hopper Disassembler

Class-dump tool can be used to get information about methods in the application. Example below uses Damn Vulnerable iOS Application [12]. As our binary is so-called fat binary, which means that it can be executed on 32 and 64 bit platforms:

$ unzip DamnVulnerableiOSApp.ipa  
  
$ cd Payload/DamnVulnerableIOSApp.app  
  
$ otool -hv DamnVulnerableIOSApp   
  
DamnVulnerableIOSApp (architecture armv7):  
Mach header  
 magic cputype cpusubtype caps filetype ncmds sizeofcmds flags  
 MH\_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK TWOLEVEL WEAK\_DEFINES BINDS\_TO\_WEAK PIE  
  
DamnVulnerableIOSApp (architecture arm64):  
Mach header  
 magic cputype cpusubtype caps filetype ncmds sizeofcmds flags  
MH\_MAGIC\_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK TWOLEVEL WEAK\_DEFINES BINDS\_TO\_WEAK PIE

Note architecture armv7 which is 32 bit and arm64. This design permits to deploy the same application on all devices.  
In order to analyze the application with class-dump we must create so-called thin binary, which contains only one architecture:

iOS8-jailbreak:~ root# lipo -thin armv7 DamnVulnerableIOSApp -output DVIA32

And then we can proceed to performing class-dump:

iOS8-jailbreak:~ root# class-dump DVIA32   
  
@interface FlurryUtil : ./DVIA/DVIA/DamnVulnerableIOSApp/DamnVulnerableIOSApp/YapDatabase/Extensions/Views/Internal/  
{  
}  
+ (BOOL)appIsCracked;  
+ (BOOL)deviceIsJailbroken;

Note the plus sign, which means that this is a class method returning BOOL type.  
A minus sign would mean that this is an instance method. Please refer to further sections to understand the practical difference between both.

Alternatively, you can easily decompile the application with Hopper Disassembler [13]. All these steps will be performed automatically and you will be able to see disassembled binary and class information.

Your main focus while performing static analysis would be:

* Identifying and undestanding functions responsible for jailbreak detection and certificate pinning
* For jailbreak detection, look for methods or classess containing words like jailbreak, jailbroken, cracked, etc. Please note that sometimes, the name of function performing jailbreak detection will be 'obfuscated' to slow down the analysis. Your best bet is to look for jailbreak detection mechanisms discussed in further section (cf. Dynamic Analysis - Jailbreak Detection)
* For certificate pinning, look for keywords like pinning, X509 or for native method calls like NSURLSession, CFStream, AFNetworking
* Understanding application logic and possible ways to bypass it
* Any hardcoded credentials, certificates
* Any methods that are used for obfuscation and in consequence may reveal sensitive information

#### 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-AF8CDCA8A2F1  
Application: /private/var/mobile/Containers/Bundle/Application/A8BD91A9-3C81-4674-A790-AF8CDCA8A2F1/UnCrackable Level 1.app  
Data: /private/var/mobile/Containers/Data/Application/A8AE15EE-DC8B-4F1C-91A5-1FED35258D87

You can now simply archive the data directory and pull it from the device 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 [23] 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: WOg1DfuH

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

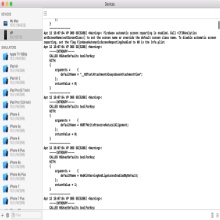
-- TODO [Dynamic analysis - copying data files, logs, from device, etc.] --

#### 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.



#### 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 singing. 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.

#### 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 [4]. An alternative way of repackaging using different tools was described by NCC group [5].

To reproduce the steps listed below, download "UnCrackable iOS App Level 1" from the OWASP Mobile Testing Guide repo [6]. 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 explains this process in great detail [5].

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 identitifier, 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 [3] 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 now, 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

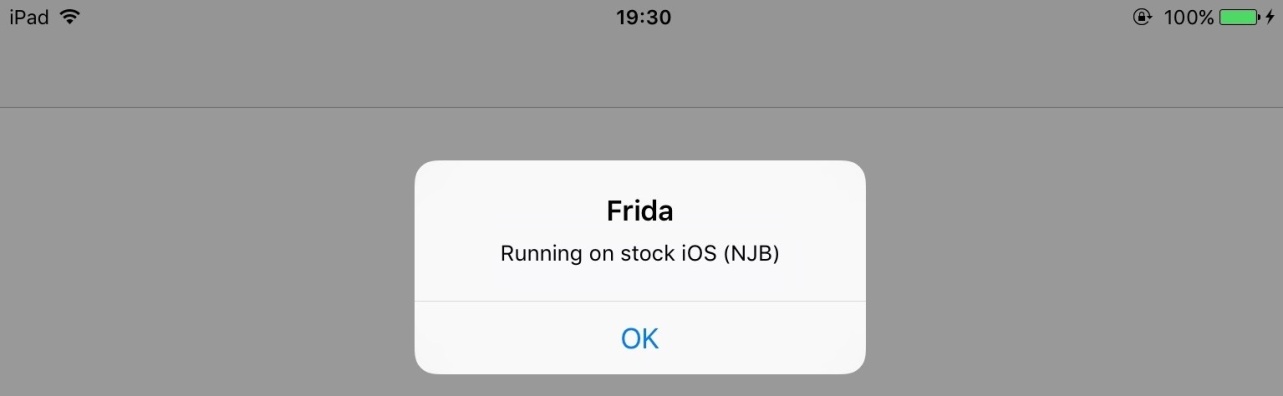
##### Installing and Running the App

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 an understanding of how the whole system works [7][8]. I also found Apple's entitlement troubleshooting page [9] to be a useful resource.

### Setting up Burp

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 should be possible to use usbmuxd [18] in order to connect to burp through USB.

The first step is to configure proxy of your burp to listen on all interfaces (alternatively only on the WiFi interface). Then we can configure our iDevice to use our proxy in advanced wifi settings. Portswigger provides good tutorial on setting an iOS Device and Burp [22].

### Bypassing Certificate Pinning

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 Cycript
* 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!

#### Recommendations

Certificate pinning is a good security practice and should be used for all applications handling sensitive information.  
[EFF's Observatory](https://www.eff.org/pl/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](https://www.eff.org/files/colour_map_of_CAs.pdf). 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 patters, refer to iOS Application Security by David Thiel [21]. 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'

### References

* [1](https://github.com/pillfill/hiding-passwords-android/) IPA Installer Console - <http://cydia.saurik.com/package/com.autopear.installipa>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Clutch - <https://github.com/KJCracks/Clutch>
* [3] Optool - <https://github.com/alexzielenski/optool>
* [4] Swizzler 2 - <https://github.com/vtky/Swizzler2/wiki>
* [5] iOS instrumentation without jailbreak - <https://www.nccgroup.trust/au/about-us/newsroom-and-events/blogs/2016/october/ios-instrumentation-without-jailbreak/>
* [6] Uncrackable Level 1 - <https://github.com/OWASP/owasp-mstg/tree/master/OMTG-Files/02_Crackmes/02_iOS/UnCrackable_Level1>
* [7] Maintaining Certificates - <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingCertificates/MaintainingCertificates.html>
* [8] Maintaining Provisioning Profiles - <https://developer.apple.com/library/content/documentation/IDEs/Conceptual/AppDistributionGuide/MaintainingProfiles/MaintainingProfiles.html>
* [9] Entitlements Troubleshooting - <https://developer.apple.com/library/content/technotes/tn2415/_index.html>
* [10] iOS-deploy - <https://github.com/phonegap/ios-deploy>
* [11] MacOS and iOS Internals, Volume III: Security & Insecurity - Johnathan Levin
* [12] Damn Vulnerable iOS Application - <http://damnvulnerableiosapp.com/>
* [13] Hopper Disassembler - <https://www.hopperapp.com/>
* [14] Introduction to iOS Application Security Testing - Slawomir Kosowski
* [15] The Mobile Application Hacker's Handbook - Dominic Chell, Tyrone Erasmus, Shaun Colley
* [16] Cydia Substrate - <http://www.cydiasubstrate.com>
* [17] Frida - <http://frida.re>
* [18] usbmuxd - <https://github.com/libimobiledevice/usbmuxd>
* [19] Jailbreak Detection Methods - <https://www.trustwave.com/Resources/SpiderLabs-Blog/Jailbreak-Detection-Methods/>
* [20] Bypassing OpenSSL Certificate Pinning -<https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2015/january/bypassing-openssl-certificate-pinning-in-ios-apps/>
* [21] iOS Application Security - David Thiel
* [22] Configuring an iOS Device to Work With Burp - <https://support.portswigger.net/customer/portal/articles/1841108-configuring-an-ios-device-to-work-with-burp>
* [23] KeyChain-Dumper - <https://github.com/ptoomey3/Keychain-Dumper/>

## Tampering and Reverse Engineering on iOS

### Environment and Toolset

-- TODO [Environment Overview] --

#### XCode and iOS SDK

-- TODO [Where to get XCode] --

#### Utilities

Class-dump by Steve Nygard [1](https://github.com/pillfill/hiding-passwords-android/) is a command-line utility for examining the Objective-C runtime information stored in Mach-O files. It generates declarations for the classes, categories and protocols.

Class-dump-dyld by Elias Limneos [2](https://developer.android.com/reference/java/security/KeyStore.html) allows dumping and retrieving symbols directly from the shared cache, eliminating the need to extract the files first. It can generate header files from app binaries, libraries, frameworks, bundles or the whole dyld\_shared\_cache. Is is also possible to Mass-dump the whole dyld\_shared\_cache or directories recursively.

MachoOView [3] is a useful visual Mach-O file browser that also allows for in-file editing of ARM binaries.

### Jailbreaking iOS

In the iOS world, jailbreaking means disabling Apple's code code signing mechanisms so that apps not signed by Apple can be run. 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) [3].

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.

-- TODO [Jailbreaking howto] --

#### 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. Also, MASVS L2 requires displaying a warning to the user, or terminate the app, when a jailbreak has been detected - the idea here is to inform users opting to jailbreak their device about the potential security implications (and not so much hindering determined reverse engineers).

We'll revisit this topic in the chapter "Testing Resiliency Against Reverse Engineering".

### Reverse Engineering iOS Apps

-- TODO [Overview] --

#### Static Analysis

-- TODO [Basic static analysis ] --

#### Debugging

iOS ships with a console app, debugserver, that allows for remote debugging using gdb or lldb. By default however, debugserver cannot be used to attach to arbitrary processes (it is usually only used for debugging self-developed apps deployed with XCode). To enable debugging of third-part apps, the task\_for\_pid entitlement must be added to the debugserver executable. An easy way to do this is adding the entitlement to the debugserver binary shipped with XCode [5].

To obtain the executable mount the following DMG image:

/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/ DeviceSupport/<target-iOS-version//DeveloperDiskImage.dmg

You’ll find the debugserver executable in the /usr/bin/ directory on the mounted volume - copy it to a temporary directory. Then, create a file called entitlements.plist with the following content:

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/ PropertyList-1.0.dtd">  
<plist version="1.0">  
<dict>  
 <key>com.apple.springboard.debugapplications</key>  
 <true/>  
 <key>run-unsigned-code</key>  
 <true/>  
 <key>get-task-allow</key>  
 <true/>  
 <key>task\_for\_pid-allow</key>  
 <true/>  
</dict>  
</plist>

And apply the entitlement with codesign:

codesign -s - --entitlements entitlements.plist -f debugserver

Copy the modified binary to any directory on the test device (note: The following examples use usbmuxd to forward a local port through USB).

$ ./tcprelay.py -t 22:2222  
$ scp -P2222 debugserver root@localhost:/tmp/

You can now attach debugserver to any process running on the device.

VP-iPhone-18:/tmp root# ./debugserver \*:1234 -a 2670  
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89  
 for armv7.  
Attaching to process 2670...

### Tampering and Instrumentation

#### Hooking with MobileSubstrate

##### Example: Deactivating Anti-Debugging

#import <substrate.h>  
  
#define PT\_DENY\_ATTACH 31  
  
static int (\*\_my\_ptrace)(int request, pid\_t pid, caddr\_t addr, int data);  
  
  
static int $\_my\_ptrace(int request, pid\_t pid, caddr\_t addr, int data) {  
 if (request == PT\_DENY\_ATTACH) {  
 request = -1;  
 }  
 return \_ptraceHook(request,pid,addr,data);  
}  
  
%ctor {  
 MSHookFunction((void \*)MSFindSymbol(NULL,"\_ptrace"), (void \*)$ptraceHook, (void \*\*)&\_ptraceHook);  
}

#### Cycript and Cynject

Cydia Substrate (formerly called MobileSubstrate) is the de-facto standard framework for developing run-time patches (“Cydia Substrate extensions”) on iOS. It comes with Cynject, a tool that provides code injection support for C. By injecting a JavaScriptCore VM into a running process on iOS, users can interface with C code, with support for primitive types, pointers, structs and C Strings, as well as Objective-C objects and data structures. It is also possible to access and instantiate Objective-C classes inside a running process. Some examples for the use of Cycript are listed in the iOS chapter.

Cycript injects a JavaScriptCore VM into the running process. Users can then manipulate the process using JavaScript with some syntax extensions through the Cycript Console.

-- TODO [Add use cases and example for Cycript] --

* Obtain references to existing objects
* Instantiate objects from classes
* Hooking native functions
* Hooking objective-C methods
* etc.\*  
  <http://www.cycript.org/manual/>

Cycript tricks:

<http://iphonedevwiki.net/index.php/Cycript_Tricks>

#### Frida

-- TODO [Develop section on Frida] --

### References

* [1](https://github.com/pillfill/hiding-passwords-android/) Class-dump - <http://stevenygard.com/projects/class-dump/>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Class-dump-dyld - <https://github.com/limneos/classdump-dyld/>
* [3] MachOView - <https://sourceforge.net/projects/machoview/>
* [3] Jailbreak Exploits on the iPhone Dev Wiki - <https://www.theiphonewiki.com/wiki/Jailbreak_Exploits#Pangu9_.289.0_.2F_9.0.1_.2F_9.0.2.29>)
* [4] Stack Overflow - <http://stackoverflow.com/questions/413242/how-do-i-detect-that-an-ios-app-is-running-on-a-jailbroken-phone>
* [5] Debug Server on the iPhone Dev Wiki - <http://iphonedevwiki.net/index.php/Debugserver>

## Testing Data Storage

### Testing Local Data Storage

#### Overview

Storing data is essential for many mobile applications, for example in order to keep track of user settings or data a user might has keyed in that needs to stored locally or offline. Data can be stored persistently by a mobile application in various ways. The following table shows mechanisms that are available on the iOS platform, that should usually be not considered to store sentive data.

* CoreData/SQLite Databases
* NSUserDefaults
* Property List (Plist) files
* Plain files

#### Dynamic Analysis

A way to identify if sensitive information like credentials and keys are stored insecurely and without leveraging the native functions from iOS is to analyse the app data directory. It is important to trigger as much app functionality as possbile before the data is analysed, as the app might only store system credentials as specific functionality is triggered by the user. A static analysis can then be performed for the data dump based on generic keywords and app specifc data. Identify how the application stores data locally on the iOS device.

Steps :

1. Proceed to trigger functionality that stores potential sensitive data.
2. Connect to the iOS device and browse to the following directory (this is applicable to iOS version 8.0 and higher): /var/mobile/Containers/Data/Application/$APP\_ID/
3. Perform a grep command of the data that you have stored, such as: grep -irn "USERID".
4. If the sensitive data is being stored in plaintext, it fails this test.

Manual dynamic analysis such as debugging can also be leveraged to verify how specific system credentials are stored and processed on the device. As this approach is more time consuming and is likely conducted manually, it might be only feasible for specific use cases.

-- TODO [Add content on Dynamic Testing of "Testing Local Data Storage "] --

#### Static Analysis

Ideally sensitive information should not be stored on the device at all. If there is a requirement to store sensitive information on the device itself, several functions/API calls are available to protect the data on IOS devices by using for example the Keychain.

During the static analysis it should be checked if sensitive data is stored permanently on the device. The following frameworks and functions should be checked when handling sensitive data.

##### CoreData/SQLite Databases

* Core Data is a framework that you use to manage the model layer objects in your application. It provides generalized and automated solutions to common tasks associated with object life cycle and object graph management, including persistence. Core Data operates on a sqlite database at lower level.
* sqlite3: The ‘libsqlite3.dylib’ library in framework section is required to be added in an application, which is a C++ wrapper that provides the API to the SQLite commands.

##### NSUserDefaults

The NSUserDefaults class provides a programmatic interface for interacting with the defaults system. The defaults system allows an application to customize its behavior to match a user’s preferences. Data saved by NSUserDefaults can be viewed from the application bundle. It also stores data in a plist file, but it's meant for smaller amounts of data.

##### Plain files / Plist files

* NSData: Creates static data objects, and NSMutableData creates dynamic data objects. NSData and NSMutableData are typically used for data storage and are also useful in Distributed Objects applications, where data contained in data objects can be copied or moved between applications.
* Options for methods used to write NSData objects: NSDataWritingWithoutOverwriting, NSDataWritingFileProtectionNone, NSDataWritingFileProtectionComplete, NSDataWritingFileProtectionCompleteUnlessOpen, NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication
* Store Data as part of the NSData class with: writeToFile
* Managing File Paths: NSSearchPathForDirectoriesInDomains, NSTemporaryDirectory
* The NSFileManager object lets you examine the contents of the file system and make changes to it. A way to create a file and write to it can be done through createFileAtPath.

#### Remediation

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

The following is a list of best practice used for secure storage of certificates and keys and sensitve data in general:

* For small amounts of sensitive data such as credentials or keys use the [Keychain Services](https://developer.apple.com/reference/security/1658642-keychain_services?language=objc) to securely store it locally on the device. Keychain data is protected using a class structure similar to the one used in file Data Protection. These classes have behaviors equivalent to file Data Protection classes, but use distinct keys and are part of APIs that are named differently. The the default behaviour is kSecAttrAccessibleWhenUnlocked. For more information have a look at the available modes [Keychain Item Accessibility](https://developer.apple.com/reference/security/1658642-keychain_services/1663541-keychain_item_accessibility_cons).
* Cryptographic functions that have been self implemented to encrypt or decrypt local files should be avoided.
* Avoid insecure storage functions for sensitive information such as credentials and keys as illustrated in chapter OMTG-DATAST-001-2.

#### References

* [Keychain Services Programming Guide](https://developer.apple.com/library/content/documentation/Security/Conceptual/keychainServConcepts/iPhoneTasks/iPhoneTasks.html)
* [IOS Security Guide](https://www.apple.com/business/docs/iOS_Security_Guide.pdf)
* [File System Basics](https://developer.apple.com/library/content/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/FileSystemOverview/FileSystemOverview.html)
* [Foundation Functions](https://developer.apple.com/reference/foundation/1613024-foundation_functions)
* [NSFileManager](https://developer.apple.com/reference/foundation/nsfilemanager)
* [NSUserDefaults](https://developer.apple.com/reference/foundation/userdefaults)

##### OWASP MASVS

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

##### OWASP Mobile Top 10

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

##### 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

### 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 that are stored locally when being offline and being sent to the application developer/company once online again or for usage statistics. However, logging sensitive data such as credit card number and session IDs might expose the data to attackers or malicious applications.  
Log files can be created in various ways on each of the different operating systems. The following list shows the mechanisms that are available on iOS:

* NSLog Method
* printf-like function
* NSAssert-like function
* Macro

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

#### Dynamic Analysis

Proceed to a page on the iOS application that contains input fields which prompt users for their sensitive information. Two different methods are applicable to check for sensitive data in log files:

* Connect to the iOS device and execute the following command:
* tail -f /var/log/syslog
* Connect your iOS device via USB and launch Xcode. Navigate to Windows > Devices, select your device and the respective application.

Proceed to complete the input fields prompt and if the sensitive data are displayed in the output of the above command, it fails this test.

#### Static Analysis

Check the source code for usage of predefined/custom Logging statements using the following keywords :

* For predefined and built-in functions:
* NSLog
* NSAssert
* NSCAssert
* fprintf
* For custom functions:
* Logging
* Logfile

#### Remediation

Use a define to enable NSLog statements for development and debugging, and disable these before shipping the software. This can be done by putting the following code into the appropriate PREFIX\_HEADER (\*.pch) file:

#ifdef DEBUG  
# define NSLog (...) NSLog(\_\_VA\_ARGS\_\_)  
#else  
# define NSLog (...)  
#endif

#### References

-- TODO [Add references for section "Testing for Sensitive Data in Logs"] --

##### OWASP MASVS

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

##### OWASP Mobile Top 10

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

##### CWE

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

### 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 behaviour within the App, selling banner advertisements or to create a better user experience. Interacting with these services abstracts the complexity and neediness to implement the functionality on its own and to reinvent the wheel.

The downside is that a developer doesn’t know in detail what code is executed via 3rd party libraries and therefore giving up visibility. Consequently it should be ensured that not more information as needed is sent to the service and that no sensitive information is disclosed.

3rd party services are mostly implemented in two ways:

* By using a standalone library.
* By using a full SDK.

#### Static Analysis

-- TODO [Add content on black-box testing of "Testing Whether Sensitive Data Is Sent to Third Parties"] --

#### 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 Proxy* or OWASP ZAP, to intercept the traffic exchanged between client and server. A complete guide can be found [here](http://FIXME). Once we are able to route the traffic to the interception proxy, we can try to sniff the traffic from the App. 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.
* When decompiling the App, API calls and/or functions provided through the 3rd party library should be reviewed on a source code level to identify if they are used accordingly to best practices.

#### 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.

#### References

-- TODO [Add content on References of "Testing Whether Sensitive Data Is Sent to Third Parties"] --

##### OWASP MASVS

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

##### OWASP Mobile Top 10

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

##### CWE

* CWE-359 "Exposure of Private Information ('Privacy Violation')": [Link to CWE issue]

### Testing for Sensitive Data in the Keyboard Cache

#### Overview

In order to simplify keyboard input by providing autocorrection, predicative input, spell checking, etc., most of keyboard input by default is cached in /private/var/mobile/Library/Keyboard/dynamic-text.dat

This behavior is achieved by means of UITextInputTraits protocol, which is adopted by UITextField, UITextView and UISearchBar. Keyboard caching is influenced by following properties:

* var autocorrectionType: UITextAutocorrectionType determines whether autocorrection is enabled or disabled during typing. With autocorrection enabled, the text object tracks unknown words and suggests a more suitable replacement candidate to the user, replacing the typed text automatically unless the user explicitly overrides the action. The default value for this property is UIText​Autocorrection​Type​Default, which for most input methods results in autocorrection being enabled.
* var secureTextEntry: BOOL identifies whether text copying and text caching should be disabled and in case of UITextField hides the text being entered. This property is set to NO by default.

#### Dynamic Analysis

1. Reset your iOS device keyboard cache by going through: Settings > General > Reset > Reset Keyboard Dictionary
2. Proceed to use the application's functionalities. Identify the functions which allow users to enter sensitive data.
3. Dump the keyboard cache file dynamic-text.dat at the following directory (Might be different in iOS below 8.0):  
   /private/var/mobile/Library/Keyboard/
4. Look for sensitive data such as username, passwords, email addresses, credit card numbers, etc. If the sensitive data can be obtained through the keyboard cache file, it fails this test.

#### Static Analysis

Check with the developers directly if there is any implementation to disable keyboard cache.

* Search through the source code provided to look the following similar implementations.

textObject.autocorrectionType = UITextAutocorrectionTypeNo; textObject.secureTextEntry = YES;

* Open xib and storyboard files in Interface Builder and verify states of Secure Text Entry and Correction in Attributes Inspector for appropriate objects.

#### Remediation

The application must ensure that data typed into text fields which contains sensitive information must not be cached. This can be achieved by disabling the feature programmatically by using the textObject.autocorrectionType = UITextAutocorrectionTypeNo directive in the desired UITextFields, UITextViews and UISearchBars. For data that should be masked such as PIN and passwords, set the textObject.secureTextEntry to YES.

UITextField \*textField = [ [ UITextField alloc ] initWithFrame: frame ];  
textField.autocorrectionType = UITextAutocorrectionTypeNo;

#### References

* [UIText​Input​Traits protocol](https://developer.apple.com/reference/uikit/uitextinputtraits)

##### OWASP MASVS

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

##### OWASP Mobile Top 10

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

##### CWE

* CWE-524: Information Exposure Through Caching

### Testing for Sensitive Data in the Clipboard

#### Overview

-- TODO [Add content on overview of "Testing for Sensitive Data in the Clipboard"] --

#### Black-box Testing

Proceeed to a view in the application that has input fields which prompt the user for sensitive information such as username, password, credit card number, etc.

Enter some values and double tap on the input field.

If the "Select", "Select All", and "Paste" option shows up, proceed to tap on the "Select", or "Select All" option, it should allow you to "Cut", "Copy", "Paste", or "Define".

The "Cut" and "Copy" option should be disabled for sensitive input fields, since it will be possible to retrieve the value by pasting it.

If the sensitive input fields allow you to "Cut" or "Copy" the values, it fails this test.

#### White-box Testing

Search through the source code provided to look for any implemented subclass of UITextField.

@interface name\_of\_sub\_class : UITextField  
action == @select(cut:)  
action == @select(copy:)

#### Remediation

Possible remediation method:

@interface NoSelectTextField : UITextField  
  
@end  
  
@implementation NoSelectTextField  
  
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender {  
 if (action == @selector(paste:) ||  
 action == @selector(cut:) ||  
 action == @selector(copy:) ||  
 action == @selector(select:) ||  
 action == @selector(selectAll:) ||  
 action == @selector(delete:) ||  
 action == @selector(makeTextWritingDirectionLeftToRight:) ||  
 action == @selector(makeTextWritingDirectionRightToLeft:) ||  
 action == @selector(toggleBoldface:) ||  
 action == @selector(toggleItalics:) ||  
 action == @selector(toggleUnderline:)  
 ) {  
 return NO;  
 }  
 return [super canPerformAction:action withSender:sender];  
}  
  
@end

<http://stackoverflow.com/questions/1426731/how-disable-copy-cut-select-select-all-in-uitextview>

#### References

-- TODO [Add references for "Testing for Sensitive Data in the Clipboard"] --

### Testing Whether Sensitive Data Is Exposed via IPC Mechanisms

#### Overview

-- TODO [Add content on overview of "Testing Whether Sensitive Data Is Exposed via IPC Mechanisms"] --

#### Black-box Testing

-- TODO [Add content on black-box testing of "Testing Whether Sensitive Data Is Exposed via IPC Mechanisms"] --

#### White-box Testing

-- TODO [Add content on white-box testing of "Testing Whether Sensitive Data Is Exposed via IPC Mechanisms"] --

#### Remediation

-- TODO [Add remediation on "Testing Whether Sensitive Data Is Exposed via IPC Mechanisms"] --

#### References

-- TODO [Add references for "Testing Whether Sensitive Data Is Exposed via IPC Mechanisms"] --

### Testing for Sensitive Data Disclosure Through the User Interface

##### Overview

-- TODO [Add content on overview for "Testing for Sensitive Data Disclosure Through the User Interface"] --

#### Black-box Testing

-- TODO [Add content on black-box testing of "Testing for Sensitive Data Disclosure Through the User Interface"] --

#### White-box Testing

-- TODO [Add content on white-box testing of "Testing for Sensitive Data Disclosure Through the User Interface"] --

#### Remediation

-- TODO [Add remediation of "Testing for Sensitive Data Disclosure Through the User Interface"] --

#### References

-- TODO [Add references for "Testing for Sensitive Data Disclosure Through the User Interface"] --

### Testing for Sensitive Data in Backups

#### Overview

-- TODO [Add content on overview of "Testing for Sensitive Data in Backups"] --

#### Black-box Testing

-- TODO [Add content on black-box testing of "Testing for Sensitive Data in Backups"] --

#### White-box Testing

-- TODO [Add content on white-box testing of "Testing for Sensitive Data in Backups"] --

#### Remediation

-- TODO [Add content on remediation of "Testing for Sensitive Data in Backups"] --

#### References

-- TODO [Add references for "Testing for Sensitive Data in Backups"] --

### Testing For Sensitive Information in Auto-Generated Screenshots

#### Overview

Manufacturers want to provide device users an aesthetically pleasing effect when an application is entered or exited, hence they introduced the concept of saving a screenshot when the application goes into the background. This feature could potentially pose a security risk for an application, as the screenshot containing sensitive information (e.g. a screenshot of an email or corporate documents) is written to local storage, where it can be recovered either by a rogue application on a jailbroken device, or by someone who steals the device.

#### Black-box Testing

Proceed to a page on the application which displays sensitive information such as username, email address, account details, etc. Background the application by hitting the Home button on your iOS device. Connect to the iOS device and proceed to the following directory (Might be different in iOS below 8.0):

/var/mobile/Containers/Data/Application/$APP\_ID/Library/Caches/Snapshots/

If the application caches the sensitive information page as a screenshot, it fails this test.

It is highly recommended to have a default screenshot that will be cached whenever the application enters background.

#### White-box Testing

While analyzing the source code, look for the fields or screens where sensitive data is involved. Identify if the application sanitize the screen before being backgrounded.

#### Remediation

Possible remediation method that will set a default screenshot:

@property (UIImageView \*)backgroundImage;  
   
- (void)applicationDidEnterBackground:(UIApplication \*)application {  
    UIImageView \*myBanner = [[UIImageView alloc] initWithImage:@"overlayImage.png"];  
    self.backgroundImage = myBanner;  
    [self.window addSubview:myBanner];  
}

This will cause the background image to be set to the "overlayImage.png" instead whenever the application is being backgrounded. It will prevent sensitive data leaks as the "overlayImage.png" will always override the current view.

#### References

-- TODO [Add references for "Testing For Sensitive Information in Auto-Generated Screenshots" ] --

### Testing the Device-Access-Security Policy

#### Overview

-- TODO [Add content for overview of "Testing the Device-Access-Security Policy"] --

#### Static Analysis

-- TODO [Add content for static analysis of "Testing the Device-Access-Security Policy"] --

#### Dynamic Analysis

-- TODO [Add content for dynamic analysis of "Testing the Device-Access-Security Policy"] --

#### Remediation

-- TODO [Add remediation of "Testing the Device-Access-Security Policy"] --

#### References

##### OWASP MASVS

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

##### OWASP Mobile Top 10

* M1 - Improper Platform Usage

##### CWE

* CWE: -- TODO [Add link to CWE issue] --

### 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 users.

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

* App shows after starting it the first time a list of data it is storing locally and remotely. This can also be a link to an external ressource 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 as secure and applies to best practice password policies.
* If the user is installing the App on a rooted device a warning should be shown that this is dangerous and deactivates security controls on OS level and is more likely to be prone to Malware. See also OMTG-DATAST-011 for more details.
* If a user installed the App on an outdated Android version a warning should be shown. See also OMTG-DATAST-010 for more details.

-- TODO [What else can be a warning on Android?] --

#### Static Analysis

-- TODO [Add content for static analysis of "Verifying User Education Controls"] --

#### 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 education purpose.

-- TODO [Further develop content of dynamic analysis of "Verifying User Education Controls"] --

#### Remediation

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

-- TODO [Further develop remediation of "Verifying User Education Controls"] --

#### References

-- TODO [Add references for "Verifying User Education Controls"] --

##### 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."

##### OWASP Mobile Top 10

* M1 - Improper Platform Usage

##### CWE

* CWE: -- TODO [Add link to CWE issue for "Verifying User Education Controls"] --

## Testing Cryptography

### Verifying Cryptographic Key Management

#### Overview

Proper cryptographic key management is often one of pitfalls of mobile applications. Although, platform provides standard system API like keychain, sometimes developers seem to either not use it at all, or use it improperly.

#### Static Analysis

During static analysis, the most important part is to understand how the application is using cryptographic algorithms. Let us divide applications into three main categories

1. An application is a pure online application, where authentication, authorization is done online with application server and no information is stored locally.
2. An application is mainly an offline application, where authentication and authorization is done purely locally. Application information is stored also locally.
3. An application is mix of first two, i.e. it supports both: online and offline authentication, some information may be stored locally and some or all actions that are performed online may be performed offline.

* A good example of such an app, may be point of sale, where seller may sell products. The app requires connection to the internet, so that it can communicate with backend and update information on products that were sold, cash amount, etc. However, there might be a business requirement that this app must also work in offline mode and would synchronize all information once it connects back to the internet. This will be a mixed app type, i.e. online and offline.

The following checks would be performed in both applications:

* Ensure that no keys/passwords are stored within the source code. Pay special attention to any 'administrative' or backdoor accounts enabled in the source code. Storing fixed salt within application 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 hardcoding keys.
* If the application 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. hardcoded in the app)

Proper way would be to generate client certificate upon user registration/first login and then store it in the keychain.

* Ensure that the keys/passwords/logins are not stored in application data. This can be included in the iTunes backup and increase attack surface. Keychain is the only appropriate place to store credentials of any type (password, certificate, etc.).
* Ensure that keychain entries have appropriate protection class. The most rigorous being kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly which translates to: entry unlocked only if passcode on the device is set and device is unlocked; the entry is not exportable in backups or by any other means.

The following checks would be performed in the offline application:

* 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.

##### With Source Code

-- TODO [Create content of ""Verifying Cryptographic Key Management" with source code] --

##### Without Source Code

-- TODO [Create content of "Verifying Cryptographic Key Management" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Verifying Cryptographic Key Management" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying Cryptographic Key Management".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update below reference "VX.Y" for "Verifying Cryptographic Key Management"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying Cryptographic Key Management"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add link to relevant tools for "Verifying Cryptographic Key Management"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing for Custom Implementations of Cryptography

#### Overview

The use of a non-standard algorithm is dangerous because a determined attacker may be able to break the algorithm and compromise whatever data has been protected. Well-known techniques may exist to break the algorithm.

#### White-box Testing

Carefully inspect all the crypto methods, especially those which are directly applied to the sensitive data. Pay close attention to seemingly standard but modified algorithms. Remember that encoding is not encryption! Any appearance of direct XORing might be a good sign to start digging deeper.

#### Black-box Testing

Although fuzzing of the custom algorithm might work in case of very weak crypto, the recommended approach would be to decompile the app and inspect the algorithm to see if custom encryption schemes is really the case (see "White-box Testing")

#### Remediation

When there is a need to store or transmit sensitive data, use strong, up-to-date cryptographic algorithms to encrypt that data. Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and use well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis.  
Do not develop custom or private cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If the algorithm can be compromised if attackers find out how it works, then it is especially weak.

##### OWASP MASVS

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

##### OWASP Mobile Top 10

* M6 - Broken Cryptography

##### CWE

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

### Verifying the Configuration of Cryptographic Standard Algorithms

#### Overview

Apple provides libraries with implementations of most commonly used cryptographic algorithms. A good point of reference is Apple's Cryptographic Services Guide [1](https://github.com/pillfill/hiding-passwords-android/). It contains broad documentation on how to use standard libraries to initialize and use cryptographic primitives, which is also useful when performing source code analysis.  
For black-box testing, more useful is native C API, for instance CommonCryptor, that is most frequently used when performing cryptographic operations. Source code is partially available at the Apple open source repository [2](https://developer.android.com/reference/java/security/KeyStore.html).

#### Static Analysis

The main goal of static analysis is to ensure the following:

* 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), crooked random number generators like Dual\_EC\_DRBG (even if they are NIST certified). All of these should be marked as insecure and removed from the application or server.
* key lengths are in-line with industry standards and provide protection for sufficient amount of time. An online comparison of different key lenghts and protection they provide taking into account Moore's law is available on the web [3].
* cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

##### With Source Code

-- TODO [Create content for "Verifying the Configuration of Cryptographic Standard Algorithms" with source code] --

##### Without Source Code

If the app is using standard cryptographic implementations provided by Apple, the easiest way is to decompile the application and check for calls to functions from CommonCryptor, such as CCCrypt, CCCryptorCreate, etc. The [source code](https://opensource.apple.com/source/CommonCrypto/CommonCrypto-36064/CommonCrypto/CommonCryptor.h) contains signatures of all functions.  
For instance, CCCryptorCreate has following signature:

CCCryptorStatus CCCryptorCreate(  
 CCOperation op, /\* kCCEncrypt, etc. \*/  
 CCAlgorithm alg, /\* kCCAlgorithmDES, etc. \*/  
 CCOptions options, /\* kCCOptionPKCS7Padding, etc. \*/  
 const void \*key, /\* raw key material \*/  
 size\_t keyLength,   
 const void \*iv, /\* optional initialization vector \*/  
 CCCryptorRef \*cryptorRef); /\* RETURNED \*/

You can then compare all the enum types to understand which algorithm, padding and key material is being used. Pay attention to the keying material, if it's coming directly from a password (which is bad), or if it's comming from Key Generation Function (e.g. PBKDF2).  
Obviously, there are other non-standard libraries that your application might be using (for instance openssl), so you should check for these too.

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Verifying the Configuration of Cryptographic Standard Algorithms" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying the Configuration of Cryptographic Standard Algorithms".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Verifying the Configuration of Cryptographic Standard Algorithms"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying the Configuration of Cryptographic Standard Algorithms"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Apple Cryptographic Services Guide - <https://developer.apple.com/library/content/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Apple Open Source - <https://opensource.apple.com>
* [3] Keylength comparison - <https://www.keylength.com/>

##### Tools

-- TODO [Add links to relevant tools for "Verifying the Configuration of Cryptographic Standard Algorithms"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Random Number Generation

#### Overview

-- TODO [Provide a general description of the issue "Testing Random Number Generation".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for "Testing Random Number Generation" with source code] --

##### Without Source Code

-- TODO [Add content for "Testing Random Number Generation" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Random Number Generation" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Random Number Generation".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Random Number Generation"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Random Number Generation"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add links to relavant tools for "Testing Random Number Generation"] --

* Enjarify - <https://github.com/google/enjarify>

## Testing Authentication

### Verifying that Users Are Authenticated Properly

#### Overview

-- TODO [Provide a general description of the issue "Verifying that Users Are Authenticated Properly".] --

-- TODO [One recommended best practice is that authentication must be enforced on the server. List other recommendations here.] --

Some applications are doing the authentication relying on the client side, that means that the developer creates some method that will check the username and password on the client side instead of sending the credentials into the backend API. So in that situation, it is possible with the help of some tools to bypass login forms and get access to the application.

#### Static Analysis

##### With Source Code

-- TODO [Add content on "Verifying that Users Are Authenticated Properly" with source code] --

##### Without Source Code

-- TODO [Add content on "Verifying that Users Are Authenticated Properly" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Verifying that Users Are Authenticated Properly" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying that Users Are Authenticated Properly".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Verifying that Users Are Authenticated Properly"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying that Users Are Authenticated Properly"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Verifying that Users Are Authenticated Properly"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Session Management

#### Overview

-- TODO [Provide a general description of the issue "Testing Session Management".] --

##### Recommended best Practices

-- TODO [Develop content for Recommended best Practices for "Testing Session Management".] --

* A "logout" function should exist that allows the user to terminate the session.

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Add content on "Testing Session Management" with source code] --

##### Without Source Code

-- TODO [Add content on "Testing Session Management" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Session Management" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Session Management".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Session Management"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Session Management"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Session Management"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing the Logout Functionality

#### Overview

Session termination is an important part of the session lifecycle. Reducing the lifetime of the session tokens to a minimum decreases the likelihood of a successful session hijacking attack.  
   
The scope for this test case is to validate that the application has a logout functionality and it effectively terminates the session on client and server side.

#### Testing

To verify the correct implementation of a logout functionality, dynamic analysis should be applied by using an interception proxy. This technique can be applied to both, Android and iOS platform.    
Static Analysis

If server side code is available, it should be reviewed to validate that the session is being terminated as part of the logout functionality.

The check needed here will be different depending on the technology used. Here are different examples on how a session can be terminated in order to implement a proper logout on server side:

* Spring (Java) - <http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/web/authentication/logout/SecurityContextLogoutHandler.html>
* Ruby on Rails -  <http://guides.rubyonrails.org/security.html>
* PHP - <http://php.net/manual/en/function.session-destroy.php>
* JSF - <http://jsfcentral.com/listings/A20158?link>
* ASP.Net - <https://msdn.microsoft.com/en-us/library/ms524798(v=vs.90).aspx>
* Amazon AWS - <http://docs.aws.amazon.com/appstream/latest/developerguide/rest-api-session-terminate.html>

#### Dynamic Analysis

For a dynamic analysis of the application an interception proxy should be used. Please see section XXX on how to set it up.  
The following steps can be applied to check if the logout is implemented properly.

1. Log into the application.
2. Do a couple of operations that require authentication inside the application.
3. Perform a logout operation.
4. Resend one of the operations detailed in step 2 using an interception proxy. For example, with Burp Repeater. The purpose of this is to send to the server a request with the token that has been invalidated in step 3.  
      
   If the session is correctly terminated on the server side, either an error message or redirect to the login page will be sent back to the client. On the other hand, if you have the same response you had in step 2, then, this session is still valid and has not been correctly terminated on the server side.

A detailed explanation with more test cases, can also be found in the OWASP Web Testing Guide (OTG-SESS-006) [1](https://github.com/pillfill/hiding-passwords-android/).

#### Remediation

One of the most common errors done by developers to a logout functionality is simply not destroying the session object in the server side. This leads to a state where the session is still alive even though the user logs out of the application. The session remains alive, and if an attacker get’s in possession of a valid session he can still use it and a user cannot even protect himself by logging out or if there are no session timeout controls in place.  
   
To mitigate it, the logout function on the server side must invalidate this session identifier immediately after logging out to prevent it to be reused by an attacker that could have intercepted it.  
   
Related to this, it must be checked that after calling an operation with an expired token, the application does not generate another valid token. This could lead to another authentication bypass.  
   
Many Apps do not automatically logout a user, because of customer convenience. The user logs in once, afterwards a token is generated on server side and stored within the applications internal storage and used for authentication when the application starts instead of asking again for user credentials. There should still be a logout function available within the application and this should work according to best practices by also destroying the session on server side.

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing the Logout Functionality"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing the Logout Functionality"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.owasp.org/index.php/Testing_for_logout_functionality_(OTG-SESS-006)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) <https://www.owasp.org/index.php/Session_Management_Cheat_Sheet>

##### Tools

-- TODO [Add relevant tools for "Testing the Logout Functionality"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing the Password Policy

#### Overview

-- TODO [Provide a general description of the issue "Testing the Password Policy".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on "Testing the Password Policy" with source code] --

##### Without Source Code

-- TODO [Add content on "Testing the Password Policy" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing the Password Policy" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing the Password Policy".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing the Password Policy"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing the Password Policy"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing the Password Policy"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Excessive Login Attempts

#### Overview

We all have heard about brute force attacks, right? That is one of the simplest attack types, as already many tools are available that work out of the box. It also doesn’t require a deep technical understanding of the target, as only a list of username and password combinations is sufficient to execute the attack. Once a valid combination of credentials is identified access to the application is possible and the account can be compromised.  
   
To be protected against these kind of attacks, applications need to implement a control to block the access after a defined number of incorrect login attempts.  
   
Depending on the application that you want to protect, the number of incorrect attempts allowed may vary. For example, in a banking application it should be around three to five attempts, but, in a public forum, it could be a higher number. Once this threshold is reached It also needs to be decided if the account gets locked permanently or temporarily. Locking the account temporarily is also called login throttling.

#### Testing

It is important to clarify that this control is at the server side, so the testing will be the same for iOS and Android applications.  
Moreover, the test consists by entering the password incorrectly for the defined number of attempts to trigger the account lockout. At that point, the anti-brute force control should be activated and your logon should be rejected when the correct credentials are entered.

#### Static Analysis

If server side code is available, it should be reviewed to validate that the session is being terminated as part of the lockout functionality.  
Here, we need to check that there is a validation in the logon method that checks if the number of attempts in a credential equals to the maximum number of attempts set. In that case, no logon should be granted  
It worths reviewing too that, after a correct attempt, there is a mechanism in place to set the error counter to zero.

#### Dynamic Analysis

For a dynamic analysis of the application an interception proxy should be used. Please see section XXX on how to set it up.  
The following steps can be applied to check if the lockout mechanism is implemented properly.

1. Log in incorrectly for a number of times to trigger the lockout control (generally 3 to 15 incorrect attempts)
2. Once you have locked out the account, enter the correct logon details to verify if login is not possible anymore.  
   If this is correctly implemented, when the right password is entered, logon should be denied, as the credential has already been blocked.

#### Remediation

Lockout controls have to be implemented to prevent brute force attacks. See [3] for further mitigation techniques.  
It is interesting to clarify that incorrect logon attempts should be cumulative and not linked to a session. If you implement a control to block the credential in your 3rd attempt in the same session, it can be easily bypassed by entering the details wrong two times and get a new session. This will then give another free attempts.

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Excessive Login Attempts"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Excessive Login Attempts"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.owasp.org/index.php/Testing_for_Weak_lock_out_mechanism_(OTG-AUTHN-003)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) <https://www.owasp.org/index.php/Brute_force_attack>
* [3] <https://www.owasp.org/index.php/Blocking_Brute_Force_Attacks>

##### Tools

-- TODO [Add relevant tools for "Testing Excessive Login Attempts"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Biometric Authentication

#### Overview

-- TODO [Provide a general description of the issue "Testing Biometric Authentication".]

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for "Testing Biometric Authentication" with source code] --

##### Without Source Code

-- TODO [Add content for "Testing Biometric Authentication" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Biometric Authentication" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Biometric Authentication".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Biometric Authentication"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Biometric Authentication"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Biometric Authentication"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing the Session Timeout

#### Overview

Compared to web applications most mobile applications don’t have a session timeout mechanism that terminates the session after some period of inactivity and force the user to login again. For most mobile applications users need to enter the credentials once. After authenticating on server side an access token is stored on the device which is used to authenticate. If the token is about to expire the token will be renewed without entering the credentials again. Applications that handle sensitive data like patient data or critical functions like financial transactions should implement a session timeout as a security-in-depth measure that forces users to re-login after a defined period.  
   
We will explain here how to check that this control is implemented correctly, both in the client and server side.

#### Testing

To test this, dynamic analysis is an efficient option, as it is easy to validate if this feature is working or not at runtime using an interception proxy. This is similar to test case OMTG-AUTH-002 (Testing the Logout Functionality), but we need to leave the application in idle for the period of time required to trigger the timeout function. Once this condition has been launched, we need to validate that the session is effectively terminated on client and server side.  
This technique can be applied to both, Android and iOS platform.

#### Static Analysis

If server side code is available, it should be reviewed that the session timeout functionality is correctly configured and a timeout is triggered after a defined period of time.    
The check needed here will be different depending on the technology used. Here are different examples on how a session timeout can be configured:

* Spring (Java) - <http://docs.spring.io/spring-session/docs/current/reference/html5/>
* Ruby on Rails -  <https://github.com/rails/rails/blob/318a20c140de57a7d5f820753c82258a3696c465/railties/lib/rails/application/configuration.rb#L130>
* PHP - <http://php.net/manual/en/session.configuration.php#ini.session.gc-maxlifetime>
* ASP.Net - <https://msdn.microsoft.com/en-GB/library/system.web.sessionstate.httpsessionstate.timeout(v=vs.110).aspx>
* Amazon AWS - <http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/config-idle-timeout.html>  
     
  Some applications also have an autologoff functionality in the client side. This is not a mandatory feature, but helps to improve to enforce a session timeout.  To implement this, the client side needs to control the timestamp when the screen has been displayed, and check continuously if the time elapsed is lower than the defined timeout. Once that time matches or excesses the timeout, the logoff method will be invoked, sending a signal to the server side to terminate the session and redirecting the customer to an informative screen.

-- TODO [Change code below from Android code to iOS code + format it as code "Testing the Session Timeout"] --  
For Android the following code might be used to implement it [3]:

public class TestActivity extends TimeoutActivity {  
@Override protected void onTimeout() {  
// logout  
}  
@Override protected long getTimeoutInSeconds() {  
return 15 \* 60; // 15 minutes  
}

#### Dynamic Analysis

For a dynamic analysis of the application an interception proxy should be used. Please see section XXX on how to set it up.  
The following steps can be applied to check if the session timeout is implemented properly.

* Log into the application.
* Do a couple of operations that require authentication inside the application.
* Leave the application in idle until the session expires (for testing purposes, a reasonable timeout can be configured, and amended later in the final version)  
     
  Resend one of the operations executed in step 2 using an interception proxy. For example, with Burp Repeater. The purpose of this is to send to the server a request with the session ID that has been invalidated when the session has expired.  
  If session timeout has been correctly configured on the server side, either an error message or redirect to the login page will be sent back to the client. On the other hand, if you have the same response you had in step 2, then, this session is still valid, which means that the session timeout control is not configured correctly.  
  More information can also be found in the OWASP Web Testing Guide (OTG-SESS-007) [1](https://github.com/pillfill/hiding-passwords-android/).

#### Remediation

Most of the frameworks have a parameter to configure the session timeout. This parameter should be set accordingly to the best practices specified of the documentation of the framework. The best practice timeout setting may vary between 5 to 30 minutes, depending on the sensitivity of your application and the use case of it.  
Regarding autologoff, the pseudocode of the implementation should be as follow:

Function autologoff  
    Get timestamp\_start  
    While application\_is\_running  
        time=timestamp-timestamp\_start  
        If time=logoff\_condition  
            Call logoff  
        EndIf  
    EndWhile  
End

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing the Session Timeout"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing the Session Timeout"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) OWASP web application test guide <https://www.owasp.org/index.php/Test_Session_Timeout_(OTG-SESS-007)>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) OWASP Session management cheatsheet <https://www.owasp.org/index.php/Session_Management_Cheat_Sheet>

##### Tools

-- TODO [Add relevant tools for "Testing the Session Timeout"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing 2-Factor Authentication

#### Overview

-- TODO [Provide a general description of the issue "Testing 2-Factor Authentication".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Add content on "Testing 2-Factor Authentication" with source code] --

##### Without Source Code

-- TODO [Add content on "Testing 2-Factor Authentication" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing 2-Factor Authentication" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing 2-Factor Authentication".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" in "Testing 2-Factor Authentication"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing 2-Factor Authentication"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing 2-Factor Authentication"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Step-up Authentication

#### Overview

-- TODO [Provide a general description of the issue "Testing Step-up Authentication".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on "Testing Step-up Authentication" with source code] --

##### Without Source Code

-- TODO [Add content on "Testing Step-up Authentication" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Step-up Authentication" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Step-up Authentication".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Step-up Authentication"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Step-up Authentication"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Step-up Authentication"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing User Device Management

#### Overview

-- TODO [Provide a general description of the issue "Testing User Device Management".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Add content for "Testing User Device Management" with source code] --

##### Without Source Code

-- TODO [Add content for "Testing User Device Management" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing User Device Management" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing User Device Management".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing User Device Management"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing User Device Management"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing User Device Management"] --

* Enjarify - <https://github.com/google/enjarify>

## Testing Network Communication

### General Overview

Starting from iOS 9 applications must use exclusively HTTPS and TLS 1.2 with Forward Secrecy enabled. Using HTTP requires a developer to define an exception in Info.plist file, which specifies the domains that will be using insecure communications.

### Testing for Unencrypted Sensitive Data on the Network

#### Overview

-- TODO [Add content on "Testing for Unencrypted Sensitive Data on the Network"] --

#### Static Analysis

-- TODO [Develop content on static analysis of "Testing for Unencrypted Sensitive Data on the Network"] --  
Check Info.plist file in the application bundle to check if there are any endpoints allowed to communicate over HTTP.

#### Dynamic Analysis

-- TODO [Add content on static analysis of "Testing for Unencrypted Sensitive Data on the Network"] --

#### Remediation

-- TODO [Add remediations for "Testing for Unencrypted Sensitive Data on the Network"] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference to "VX.Y" below for "Testing for Unencrypted Sensitive Data on the Network"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing for Unencrypted Sensitive Data on the Network"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add tools for "Testing for Unencrypted Sensitive Data on the Network"] --

* Enjarify - <https://github.com/google/enjarify>

### Verifying the TLS Settings

#### Overview

-- TODO [Provide a general description of the issue "Verifying the TLS Settings".]

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for "Verifying the TLS Settings" with source code] --

##### Without Source Code

-- TODO [Add content for "Verifying the TLS Settings" without source code] --

#### Dynamic Analysis

A good way of checking server-side security it to use a script like [testssl](https://testssl.sh/).  
You can also download a compiled openssl version that supports **all ciphersuites and protocols including SSLv2**.  
After downloading the script itself and proper openssl version you can use it in following way:

$ OPENSSL=./bin/openssl.Linux.x86\_64 bash ./testssl.sh yoursite.com

The tool will also help identifying potential misconfiguration or vulnerabilities by highlighting them in red.  
If you want to store the report preserving color and format use aha:

$ OPENSSL=./bin/openssl.Linux.x86\_64 bash ./testssl.sh yoursite.com | aha > output.html

This will give you a HTML document that will match CLI output.

#### Remediation

Any vulnerability or misconfiguration should be solved either by patching or reconfiguring the server. Although iOS userbase is coherent compared to Android, beware of supported SSL/TLS versions on older systems.

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Verifying the TLS Settings"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying the TLS Settings"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add tools on "Verifying the TLS Settings"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Endpoint Identity Verification

#### Overview

-- TODO [Provide a general description of the issue "Testing Endpoint Identity Verification".]

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on "Testing Endpoint Identity Verification" with source code] --

##### Without Source Code

-- TODO [Add content on "Testing Endpoint Identity Verification" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Endpoint Identity Verification" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Endpoint Identity Verification".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Endpoint Identity Verification"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Endpoint Identity Verification"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Endpoint Identity Verification"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Custom Certificate Stores and SSL Pinning

#### Overview

Certificate pinning allows to hard-code in the client the certificate 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 take the CA out of games. Mobile applications that implements certificate pinning only have to connect to a limited numbers of server, so a small list of trusted CA can be hard-coded in the application.

#### White-box Testing

The code presented below shows how it is possible to check if the certificate provided by the server reflects the certificate hard-coded in the application. The method below implements the connection authentication tells the delegate that the connection will send a request for an authentication challenge.

The delegate must implement connection:canAuthenticateAgainstProtectionSpace: and connection: forAuthenticationChallenge. Within connection: forAuthenticationChallenge, the delegate must call SecTrustEvaluate to perform customary X509 checks. Below a snippet who implements a check of the certificate.

(void)connection:(NSURLConnection \*)connection willSendRequestForAuthenticationChallenge:(NSURLAuthenticationChallenge \*)challenge  
{  
 SecTrustRef serverTrust = challenge.protectionSpace.serverTrust;  
 SecCertificateRef certificate = SecTrustGetCertificateAtIndex(serverTrust, 0);  
 NSData \*remoteCertificateData = CFBridgingRelease(SecCertificateCopyData(certificate));  
 NSString \*cerPath = [[NSBundle mainBundle] pathForResource:@"MyLocalCertificate" ofType:@"cer"];  
 NSData \*localCertData = [NSData dataWithContentsOfFile:cerPath];  
 The control below can verify if the certificate received by the server is matching the one pinned in the client.  
 if ([remoteCertificateData isEqualToData:localCertData]) {  
 NSURLCredential \*credential = [NSURLCredential credentialForTrust:serverTrust];  
 [[challenge sender] useCredential:credential forAuthenticationChallenge:challenge];  
}  
else {  
 [[challenge sender] cancelAuthenticationChallenge:challenge];  
}

#### Black-box Testing

##### Server certificate validation

We start our analysis by testing the application's behaviour while establishing secure connection.  
Our test approach is to gradually relax security of SSL handshake negotiation and check which security mechanisms are enabled.

1. Having burp set up as a proxy in wifi settings, make sure that there is no certificate added to trust store (Settings -> General -> Profiles) and that tools like SSL Kill Switch are deactivated. Launch your application and check if you can see the traffic in Burp. Any failures will be reported under 'Alerts' tabl. If you can see the traffic, it means that there is no certificate validation performed at all! This effectively means that an active attacker can silently do MiTM against your application. If however, you can't see any traffic and you have an information about SSL handshake failure, follow the next point.
2. Now, install Burp certificate, as explained in [Basic Security Testing section](./0x06b-Basic-Security-Testing.md). If the handshake is successful and you can see the traffic in Burp, it means that certificate is validated against device's trust store, but the pinning is not performed. The risk is less significant than in previous scenario, as two main attack scenarios at this point are misbehaving CAs and phishing attacks, as discussed in [Basic Security Testing section](./0x06b-Basic-Security-Testing.md).
3. If executing instructions from previous step doesn't lead to traffic being proxied through burp, it means that certificate is actually pinned and all security measures are in place. However, you still need to bypass the pinning in order to test the application. Please refer to [Basic Security Testing section](./0x06b-Basic-Security-Testing.md) for more information on this.

##### Client certificate validation

Some applications use two-way SSL handshake, meaning that application verifies server's certificate and server verifies client's certificate. You can notice this if there is an error in Burp 'Alerts' tab indicating that client failed to negotiate connection.

There is a couple of things worth noting:

1. client certificate contains private key that will be used in key exchange
2. usually certificate would also need a password to use (decrypt) it
3. certificate itself can be stored in the binary itself, data directory or the keychain

Most common and improper way of doing two-way handshake is to store client certificate within the application bundle and hardcode the password. This obviously does not bring much security, because all clients will share the same certificate.

Second way of storing the certificate (and possibly password) is to use the keychain. Upon first login, the application should download personal certificate and store it securely in the keychain.

Sometimes application have one certificate that is hardcoded and used for first login and then personal certificate is downloaded. In this case, check if it's possible to still use the 'generic' certificate to connect to the server.

Once you have extracted the certificate from the application (e.g. using Cycript or Frida), add it as client certificate in Burp, and you will be able to intercept the traffic.

#### Remediation

As a best practice, the certificate should be pinned. This can be done in several ways, where most common include:

1. Including server's certificate in the application bundle and performing verification on each connection. This requires an update mechanisms whenever the certificate on the server is updated
2. Limiting certificate issuer to e.g. one entity and bundling the root CA's public key into the application. In this way we limit the attack surface and have a valid certificate.
3. Owning and managing your own PKI. The application would contain the root CA's public key. This avoids updating the application every time you change the certificate on the server, due to e.g. expiration. Note that using your own CA would cause the certificate to be self-singed.

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for Testing Custom Certificate Stores and SSL Pinning] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for Testing Custom Certificate Stores and SSL Pinning] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* Setting Burp Suite as a proxy for iOS Devices : <https://support.portswigger.net/customer/portal/articles/1841108-configuring-an-ios-device-to-work-with-burp>  
  References
* OWASP - Certificate Pinning for iOS : <https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning#iOS>

##### Tools

-- TODO [Add relevant tools for Testing Custom Certificate Stores and SSL Pinning] --

* Enjarify - <https://github.com/google/enjarify>

### Verifying that Critical Operations Use Secure Communication Channels

#### Overview

-- TODO [Provide a general description of the issue "Verifying that Critical Operations Use Secure Communication Channels".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on "Verifying that Critical Operations Use Secure Communication Channels" with source code] --

##### Without Source Code

-- TODO [Add content on "Verifying that Critical Operations Use Secure Communication Channels" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Verifying that Critical Operations Use Secure Communication Channels" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying that Critical Operations Use Secure Communication Channels".] --

#### References

##### OWASP Mobile Top 10 2014

* M3 - Insufficient Transport Layer Protection - <https://www.owasp.org/index.php/Mobile_Top_10_2014-M3>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Verifying that Critical Operations Use Secure Communication Channels"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Verifying that Critical Operations Use Secure Communication Channels"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Verifying that Critical Operations Use Secure Communication Channels"] --

* Enjarify - <https://github.com/google/enjarify>

## Testing Platform Interaction

### Testing App permissions

#### Overview

-- TODO [Provide a general description of the issue "Testing App permissions".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on Static analysis of "Testing App permissions" with source code] --

##### Without Source Code

-- TODO [Add content on Static analysis of "Testing App permissions" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing App permissions" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing App permissions".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing App permissions"] --

##### OWASP MASVS

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

##### CWE

-- TODO [Add relevant CWE for "Testing App permissions"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add tools for "Testing App permissions"] --

### Testing Input Validation and Sanitization

#### Overview

-- TODO [Provide a general description of the issue "Testing Input Validation and Sanitization".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for static analysis of "Testing Input Validation and Sanitization" with source code] --

##### Without Source Code

-- TODO [Add content for static analysis of "Testing Input Validation and Sanitization" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Input Validation and Sanitization".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing Input Validation and Sanitization"] --

##### 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

-- TODO [Add relevant CWE for "Testing Input Validation and Sanitization"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing Input Validation and Sanitization"] --

### Testing Custom URL Schemes

#### Overview

-- TODO [Provide a general description of the issue "Testing Custom URL Schemes".]

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on static analysis for "Testing Custom URL Schemes" with source code] --

##### Without Source Code

-- TODO [Add content on static analysis for "Testing Custom URL Schemes" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Custom URL Schemes" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Custom URL Schemes".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing Custom URL Schemes"] --

##### OWASP MASVS

* V6.3: "The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected."

##### CWE

-- TODO [Add relevant CWE for "Testing Custom URL Schemes"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing Custom URL Schemes"] --

### Testing for Sensitive Functionality Exposed Through IPC

#### Overview

-- TODO [Provide a general description of the issue "Testing for Sensitive Functionality Exposed Through IPC".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on static analysis of "Testing for Sensitive Functionality Exposed Through IPC" with source code] --

##### Without Source Code

-- TODO [Add content on static analysis of "Testing for Sensitive Functionality Exposed Through IPC" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing for Sensitive Functionality Exposed Through IPC" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing for Sensitive Functionality Exposed Through IPC".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing for Sensitive Functionality Exposed Through IPC"] --

##### OWASP MASVS

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

##### CWE

-- TODO [Add relevant CWE for "Testing for Sensitive Functionality Exposed Through IPC"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing for Sensitive Functionality Exposed Through IPC"] --

### Testing JavaScript Execution in WebViews

#### Overview

-- TODO [Provide a general description of the issue "Testing JavaScript Execution in WebViews".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on static analysis of "Testing JavaScript Execution in WebViews" with source code] --

##### Without Source Code

-- TODO [Add content on static analysis of "Testing JavaScript Execution in WebViews" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing JavaScript Execution in WebViews" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing JavaScript Execution in WebViews".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing JavaScript Execution in WebViews"] --

##### OWASP MASVS

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

##### CWE

-- TODO [Add relevant CWE for "Testing JavaScript Execution in WebViews"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing JavaScript Execution in WebViews"] --

### Testing WebView Protocol Handlers

#### Overview

-- TODO [Provide a general description of the issue "Testing WebView Protocol Handlers".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on static analysis of "Testing WebView Protocol Handlers" with source code) --

##### Without Source Code

-- TODO [Add content on static analysis of "Testing WebView Protocol Handlers" without source code) --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing WebView Protocol Handlers" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing WebView Protocol Handlers".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing WebView Protocol Handlers"] --

##### 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

-- TODO [Add relevant CWE for "Testing WebView Protocol Handlers"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing WebView Protocol Handlers"] --

### Testing for Local File Inclusion in WebViews

#### Overview

-- TODO [Provide a general description of the issue "Testing for Local File Inclusion in WebViews".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on static analysis of "Testing for Local File Inclusion in WebViews" with source code] --

##### Without Source Code

-- TODO [Add content on static analysis of "Testing for Local File Inclusion in WebViews" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing for Local File Inclusion in WebViews" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing for Local File Inclusion in WebViews".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014] --

##### OWASP MASVS

* V6.7: "The app does not load user-supplied local resources into WebViews."

##### CWE

-- TODO [Add relevant CWE for "Testing for Local File Inclusion in WebViews"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing for Local File Inclusion in WebViews"] --

### Testing Whether Java Objects Are Exposed Through WebViews

#### Overview

-- TODO [Provide a general description of the issue "Testing Whether Java Objects Are Exposed Through WebViews".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for static analysis of "Testing Whether Java Objects Are Exposed Through WebViews" with source code] --

##### Without Source Code

-- TODO [Add content for static analysis of "Testing Whether Java Objects Are Exposed Through WebViews" with source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Whether Java Objects Are Exposed Through WebViews" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Whether Java Objects Are Exposed Through WebViews".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014 for "Testing Whether Java Objects Are Exposed Through WebViews"] --

##### OWASP MASVS

* V6.8: "If Java objects are exposed in a WebView, verify that the WebView only renders JavaScript contained within the app package."

##### CWE

-- TODO [Add relevant CWE for "Testing Whether Java Objects Are Exposed Through WebViews"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing Whether Java Objects Are Exposed Through WebViews"] --

### Testing Object Serialization

#### Overview

-- TODO [Add overview for "Testing Object Serialization"] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content on static analysis of "Testing Object Serialization" with source code] --

##### Without Source Code

-- TODO [Add content on static analysis of "Testing Object Serialization" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Object Serialization" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Object Serialization".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014] --

##### OWASP MASVS

* V6.9: "Object serialization, if any, is implemented using safe serialization APIs."

##### CWE

-- TODO [Add relevant CWE for "Testing Object Serialization"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Update Security Provider - <https://developer.android.com/training/articles/security-gms-provider.html>

##### Tools

-- TODO [Add relevant tools for "Testing Object Serialization"] --

### Testing Jailbreak Detection

#### Overview

iOS implements containerization so that each app is restricted to its own sandbox. A regular app cannot access files outside its dedicated data directories, and access to system APIs is restricted via app privileges. As a result, an app’s sensitive data as well as the integrity of the OS is guaranteed under normal conditions. However, when an adversary gains root access to the mobile operating system, the default protections can be bypassed completely.

The risk of malicious code running as root is higher on jailbroken devices, as many of the default integrity checks are disabled. Developers of apps that handle highly sensitive data should therefore consider implementing checks that either prevent the app from running under these conditions, or at least warn the user about the increased risks.

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)." ] --

##### With Source Code

-- TODO [Add content on static analysis of "Testing Jailbreak Detection" with source code] --

##### Without Source Code

-- TODO [Add content on static analysis of "Testing Jailbreak Detection" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Jailbreak Detection" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Jailbreak Detection".] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add reference to OWASP Mobile Top 10 2014] --

##### OWASP MASVS

* V6.10: "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

-- TODO [add relevant CWE for "Testing Jailbreak Detection"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add relevant tools for "Testing Jailbreak Detection"] --

## Testing Code Quality and Build Settings

### Verifying that the App is Properly Signed

#### Overview

-- TODO [Give an overview about the functionality and it's potential weaknesses] --

#### White-box Testing

-- TODO [Add content on white-box testing of "Verifying that the App is Properly Signed"] --

#### Black-box Testing

-- TODO [Add content on black-box testing of "Verifying that the App is Properly Signed"] --

#### Remediation

-- TODO [Add remediation for "Verifying that the App is Properly Signed"] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Verifying that the App is Properly Signed"] --

##### OWASP MASVS

* V7.1: "The app is signed and provisioned with valid certificate."

##### CWE

-- TODO [Add relevant CWE for "Verifying that the App is Properly Signed"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add tools for "Verifying that the App is Properly Signed"] --

### Testing Whether the App is Debuggable

#### Overview

-- TODO [Give an overview about the functionality "Testing Whether the App is Debuggable" and it's potential weaknesses] --

#### White-box Testing

1. Import the source code into the xCode Editor.
2. Check the project's build settings for 'DEBUG' parameter under "Apple LVM – Preprocessing" -> "Preprocessor Macros".
3. Check the source code for NSAsserts method and its companions.

#### Black-box Testing

This test case should be performed during White-box testing.

-- TODO [Develop content on black-box testing of "Testing Whether the App is Debuggable"] --

#### Remediation

Once you have deployed an iOS application, either through the App Store or as an Ad Hoc or Enterprise build, you won't be able to attach Xcode's debugger to it. To debug problems, you need to analyze Crash Logs and Console output from the device itself. Remove any NSLog calls to prevent debug leakage through the Console.

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing Whether the App is Debuggable"] --

##### OWASP MASVS

* V7.1: ""

##### CWE

-- TODO [Add relevant CWE for "Testing Whether the App is Debuggable"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add tools for "Testing Whether the App is Debuggable"] --

### Verifying that Debugging Symbols Have Been Removed

#### Overview

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

These symbols can be saved either in "Stabs" format or the DWARF format. When using the Stabs format, debugging symbols, like other symbols, are stored in the regular symbol table. With the DWARF format, debugging symbols are stored in a special "\_\_DWARF" segment within the the binary. DWARF debugging symbols can also be saved a separate debug-information file. In this test case, you verify that no debug symbols are contained in the release binary itself (either in the symbol table, ot the \_\_DWARF segment).

#### Static Analysis

Use gobjdump to inspect the main binary and any included dylibs for Stabs and DWARF symbols.

$ gobjdump --stabs --dwarf TargetApp  
In archive MyTargetApp:  
  
armv5te: file format mach-o-arm  
  
  
aarch64: file format mach-o-arm64

Gobjdump is part of binutils [1](https://github.com/pillfill/hiding-passwords-android/) and can be installed via Homebrew.

#### Dynamic Analysis

Not applicable.

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying that Debugging Symbols Have Been Removed"] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Verifying that Debugging Symbols Have Been Removed"] --

##### OWASP MASVS

-- TODO [Add reference to OWASP MASVS for "Verifying that Debugging Symbols Have Been Removed"] --

* V7.1: ""

##### CWE

-- TODO [Add relevant CWE for "Verifying that Debugging Symbols Have Been Removed"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.gnu.org/s/binutils/>

##### Tools

-- TODO [Add tools for "Verifying that Debugging Symbols Have Been Removed"] --

### Testing for Debugging Code and Verbose Error Logging

#### Overview

-- TODO [Give an overview about the functionality "Testing for Debugging Code and Verbose Error Logging" and it's potential weaknesses] --

#### White-box Testing

-- TODO [Add content for white-box testing of "Testing for Debugging Code and Verbose Error Logging"] --

#### Black-box Testing

-- TODO [Add content for black-box testing of "Testing for Debugging Code and Verbose Error Logging"] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing for Debugging Code and Verbose Error Logging"] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing for Debugging Code and Verbose Error Logging"] --

##### OWASP MASVS

-- TODO [Add reference to OWASP MASVS for "Testing for Debugging Code and Verbose Error Logging"] --

* V7.1: ""

##### CWE

-- TODO [Add relevant CWE for "Testing for Debugging Code and Verbose Error Logging"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.gnu.org/s/binutils/>

##### Tools

-- TODO [Add tools for "Testing for Debugging Code and Verbose Error Logging"] --

### Testing Exception Handling

#### Overview

-- TODO [Give an overview about the functionality "Testing Exception Handling" and it's potential weaknesses] --

#### White-box Testing

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

* Verify that the application use a [well-designed] (<https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=18581047>) (an unified) scheme to handle exceptions.
* Verify that the application doesn't expose sensitive information while handeling exceptions, but are still verbose enough to explain the issue to the user.
* C3

#### Black-box Testing

-- TODO [Describe how to test for this issue "Testing Exception Handling" using static and dynamic analysis techniques. This can include everything from simply monitoring aspects of the app’s behavior to code injection, debugging, instrumentation, etc. ] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Exception Handling"] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing Exception Handling"] --

##### OWASP MASVS

-- TODO [Add reference to OWASP MASVS for "Testing Exception Handling"] --

* V7.1: ""

##### CWE

-- TODO [Add relevant CWE for "Testing Exception Handling"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.gnu.org/s/binutils/>

##### Tools

-- TODO [Add tools for "Testing Exception Handling"] --

### Verifying that the App Fails Securely

#### Overview

-- TODO [Give an overview about the functionality and it's potential weaknesses] --

#### White-box Testing

-- TODO [Add content on white-box testing for "Verifying that the App Fails Securely"] --

#### Black-box Testing

-- TODO [Describe how to test for this issue "Verifying that the App Fails Securely" using static and dynamic analysis techniques. This can include everything from simply monitoring aspects of the app’s behavior to code injection, debugging, instrumentation, etc. ] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Verifying that the App Fails Securely"] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Verifying that the App Fails Securely"] --

##### OWASP MASVS

-- TODO [Add reference to OWASP MASVS for "Verifying that the App Fails Securely"] --

* V7.1: ""

##### CWE

-- TODO [Add relevant CWE for "Verifying that the App Fails Securely"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) <https://www.gnu.org/s/binutils/>

##### Tools

-- TODO [Add tools for "Verifying that the App Fails Securely"] --

### Testing Compiler Settings

Although XCode set all binary security features by default, it still might be relevant to some old application or to check compilation options misconfiguration. The following features are applicable:

* **ARC** - Automatic Reference Counting - memory management feature
* adds retain and release messages when required
* **Stack Canary** - helps preventing buffer overflow attacks
* **PIE** - Position Independent Executable - enables full ASLR for binary

#### Overview

-- TODO [Give an overview about the functionality "Testing Compiler Settings" and it's potential weaknesses] --

#### White-box Testing

-- TODO [Describe how to assess this with access to the source code and build configuration] --

#### Black-box Testing

-- TODO [Add content on black-box testing for "Testing Compiler Settings"] --

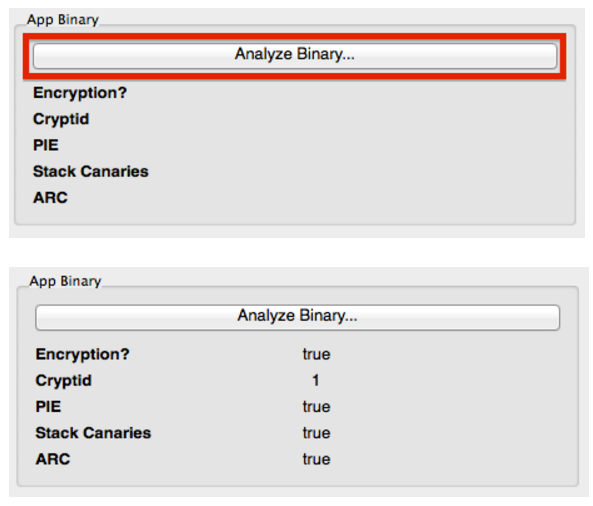
##### With otool :

Below are examples on how to check for these features. Please note that all of them are enabled in these examples:

* PIE:  
  ~~~  
  $ unzip DamnVulnerableiOSApp.ipa  
  $ cd Payload/DamnVulnerableIOSApp.app  
  $ otool -hv DamnVulnerableIOSApp  
  DamnVulnerableIOSApp (architecture armv7):  
  Mach header  
  magic cputype cpusubtype caps filetype ncmds sizeofcmds flags  
  MH\_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK TWOLEVEL  
  WEAK\_DEFINES BINDS\_TO\_WEAK PIE  
  DamnVulnerableIOSApp (architecture arm64):  
  Mach header  
  magic cputype cpusubtype caps filetype ncmds sizeofcmds flags  
  MH\_MAGIC\_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK TWOLEVEL  
  WEAK\_DEFINES BINDS\_TO\_WEAK PIE  
  ~~~
* Stack Canary:  
  ~~~  
  $ otool -Iv DamnVulnerableIOSApp | grep stack  
  0x0046040c 83177 \_\_\_stack\_chk\_fail  
  0x0046100c 83521 \_sigaltstack  
  0x004fc010 83178 \_\_\_stack\_chk\_guard  
  0x004fe5c8 83177 \_\_\_stack\_chk\_fail  
  0x004fe8c8 83521 \_sigaltstack  
  0x00000001004b3fd8 83077 \_\_\_stack\_chk\_fail  
  0x00000001004b4890 83414 \_sigaltstack  
  0x0000000100590cf0 83078 \_\_\_stack\_chk\_guard  
  0x00000001005937f8 83077 \_\_\_stack\_chk\_fail  
  0x0000000100593dc8 83414 \_sigaltstack  
  ~~~
* Automatic Reference Counting:  
  ~~~  
  $ otool -Iv DamnVulnerableIOSApp | grep release  
  0x0045b7dc 83156 \_\_\_cxa\_guard\_release  
  0x0045fd5c 83414 \_objc\_autorelease  
  0x0045fd6c 83415 \_objc\_autoreleasePoolPop  
  0x0045fd7c 83416 \_objc\_autoreleasePoolPush  
  0x0045fd8c 83417 \_objc\_autoreleaseReturnValue  
  0x0045ff0c 83441 \_objc\_release  
  [SNIP]  
  ~~~

##### With idb :

IDB automates the process of checking for both stack canary and PIE support. Select the target binary in the IDB gui and click the "Analyze Binary…" button.



#### Remediation

* Stack smashing protection

Steps for enabling Stack smashing protection within an iOS application:

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view its settings.
2. Verify that "–fstack-protector-all" option is selected under "Other C Flags" section.

* PIE support

Steps for building an iOS application as PIE :

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view its settings.
2. For iOS apps, set iOS Deployment Target to iOS 4.3 or later. For Mac apps, set OS X Deployment Target to OS X 10.7 or later.
3. Verify that "Generate Position-Dependent Code" is set at its default value of NO.
4. Verify that Don't "Create Position Independent Executables" is set at its default value of NO.

* ARC protection

Steps for enabling ACR protection within an iOS application :

1. In Xcode, select your target in the "Targets" section, then click the "Build Settings" tab to view its settings.
2. Verify that "Objective-C Automatic Reference Counting" is set at its default value of YES.

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing Compiler Settings"] --

##### OWASP MASVS

-- TODO [Add reference to OWASP MASVS for "Testing Compiler Settings"] --

* V7.1: ""

##### CWE

-- TODO [Add relevant CWE for "Testing Compiler Settings"] --

##### Info

* Technical Q&A QA1788 Building a Position Independent Executable : <https://developer.apple.com/library/mac/qa/qa1788/_index.html>
* idb : <https://github.com/dmayer/idb>

##### Tools

-- TODO [Add tools for "Testing Compiler Settings"] --

### Testing for Memory Management Bugs

#### Overview

-- TODO [Give an overview about the functionality "Testing for Memory Management Bugs" and it's potential weaknesses] --

#### White-box Testing

-- TODO [Add content for white-box testing of "Testing for Memory Management Bugs"] --

#### Black-box Testing

-- TODO [Add content for black-box testing of "Testing for Memory Management Bugs"] --

#### Remediation

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing for Memory Management Bugs"] --

##### OWASP MASVS

* V7.7: "In unmanaged code, memory is allocated, freed and used securely."

##### CWE

-- TODO [Add relevant CWE for "Testing for Memory Management Bugs"] --

##### Info

-- TODO [Add info sor "Testing for Memory Management Bugs"] --

##### Tools

-- TODO [Add tools for "Testing for Memory Management Bugs"] --

### Verifying that Java Bytecode Has Been Minifed

Not applicable on iOS.

## Testing Resiliency Against Reverse Engineering

### Testing Jailbreak Detection

#### Overview

In the context of reverse engineering defense, jailbreak detection mechansism are added to make it a bit more difficult to run the app on a jailbroken device, which in turn impedes some tools and techniques reverse engineers like to use. As is the case with most other defenses, jailbreak detection is not a very effective defense on its own, but having some checks sprinkled throughout the app can improve the effectiveness of the overall anti-tampering scheme. Typical jailbreak detection techniques on iOS include:

##### File-based Checks

Checking for the existence of files and directories typically associated with jailbreaks, such as:

/Applications/Cydia.app  
/Applications/FakeCarrier.app  
/Applications/Icy.app  
/Applications/IntelliScreen.app  
/Applications/MxTube.app  
/Applications/RockApp.app  
/Applications/SBSettings.app  
/Applications/WinterBoard.app  
/Applications/blackra1n.app  
/Library/MobileSubstrate/DynamicLibraries/LiveClock.plist  
/Library/MobileSubstrate/DynamicLibraries/Veency.plist  
/Library/MobileSubstrate/MobileSubstrate.dylib  
/System/Library/LaunchDaemons/com.ikey.bbot.plist  
/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist  
/bin/bash  
/bin/sh  
/etc/apt  
/etc/ssh/sshd\_config  
/private/var/lib/apt  
/private/var/lib/cydia  
/private/var/mobile/Library/SBSettings/Themes  
/private/var/stash  
/private/var/tmp/cydia.log  
/usr/bin/sshd  
/usr/libexec/sftp-server  
/usr/libexec/ssh-keysign  
/usr/sbin/sshd  
/var/cache/apt  
/var/lib/apt  
/var/lib/cydia  
/var/log/syslog  
/var/tmp/cydia.log

##### Checking File Permissions

Attempting to write a file to the /private/ directory. This should only be successful on jailbroken devices.

NSError \*error;  
NSString \*stringToBeWritten = @"This is a test.";  
[stringToBeWritten writeToFile:@"/private/jailbreak.txt" atomically:YES  
 encoding:NSUTF8StringEncoding error:&error];  
if(error==nil){  
 //Device is jailbroken  
 return YES;  
 } else {  
 //Device is not jailbroken  
 [[NSFileManager defaultManager] removeItemAtPath:@"/private/jailbreak.txt" error:nil];  
 }

##### Checking Protocol Handlers

Attempting to open a Cydia URL. The Cydia app store, which is installed by default by practically every jailbreaking tool, installs the cydia:// protocol handler.

if([[UIApplication sharedApplication] canOpenURL:[NSURL URLWithString:@"cydia://package/com.example.package"]]){

##### Calling System APIs

-- TODO [Fork-based check] --

Executing privileged actions. Calling the system() function with a NULL argument on a non jailbroken device will return ”0”; doing the same on a jailbroken device will return ”1”. This is since the function will check whether /bin/sh can be accessed, and this is only the case on jailbroken devices. Another possibility would be trying to write into a location outside the application’s sandbox. This can be done by having the application attempt to create a file in, for example, the /private directory. If the file is successfully created, it means the device is jailbroken.'

##### Using the Dynamic Loader

-- TODO [dyld-based check] --

##### SSH Loopback Connection

-- TODO [Connect to localhost:22] --

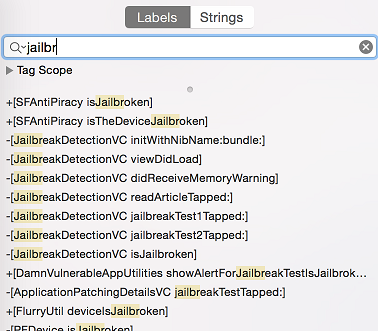
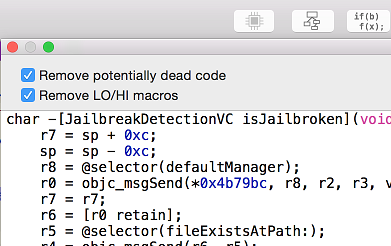
#### Bypassing Jailbreak Detection

Once you start the application, which has jailbreak detection enabled on a jailbroken device, you will notice one of the following:

1. The application closes immediately without any notification
2. There is a popup window indicating that the application won't run on a jailbroken device

In the first case, it's worth checking if the application is fully functional on non-jailbroken device. It might be that the application is in reality crashing or has a bug that causes exiting. This might happen when you're testing a preproduction version of the application.

Let's look on how to bypass jailbreak detection using once again Damn Vulnerable iOS application as an example.  
After loading the binary into Hopper, you need to wait until the application is fully disassembled (look at the top bar). Then we can look for 'jail' string in the search box. We see two different classes, which are SFAntiPiracy and JailbreakDetectionVC.  
You might also want to decompile the functions to see what they are doing and especially what do they return.

As you can see, there is a class method +[SFAntiPiracy isTheDeviceJailbroken] and instance method -[JailbreakDetectionVC isJailbroken]. The main difference for us is that we can inject cycript and call class method directly, whereas when it comes to instance method, we must first look for instances of target class. The function choose will look for the memory heap for known signature of a given class and return an array of instances that were found. It's important to put an application into a desired state, so that the class is indeed instantiated.

Let's inject cycript into our process (look for your PID with top):

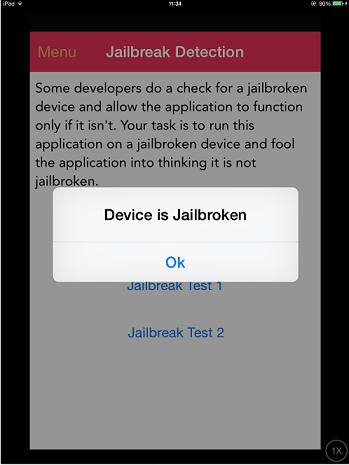
iOS8-jailbreak:~ root# cycript -p 12345  
cy# [SFAntiPiracy isTheDeviceJailbroken]  
true

As you can see our class method was called directly and returned true. Now, let's call -[JailbreakDetectionVC isJailbroken] instance method. First, we have to call choose function to look for instances of JailbreakDetectionVC class.

cy# a=choose(JailbreakDetectionVC)  
[]

Ooops! The returned array is empty. It means that there are no instances of this class registed within the runtime. In fact, we haven't clicked second 'Jailbreak Test' button, which indeed initializes this class:

cy# a=choose(JailbreakDetectionVC)  
[#"<JailbreakDetectionVC: 0x14ee15620>"]  
cy# [a[0] isJailbroken]  
True



Hence you now understand why it's important to have your application in a desired state.  
Now bypassing jailbreak detection in this case with cycript is trivial. We can see that the function returns Boolean and we just need to replace the return value. We can do it by replacing function implementation with cycript. Please note that this will actually replace function under given name, so beware of side effects in case if the function modifies anything in the application:

cy# JailbreakDetectionVC.prototype.isJailbroken=function(){return false}  
cy# [a[0] isJailbroken]  
false

  
In this case we have bypassed Jailbreak detection of the application!

Now, imagine that the application is closing immediately upon detecting that the device is jailbroken. In this case you have no chance (time) to launch cycript and replace function implementation. Instead, you would have to use CydiaSubstrate, use proper hooking function, like MSHookMessageEx and compile the tweak. There are good sources on how to perform this [15-16], however, we will provide possibly faster and more flexible approach.

**Frida** is a dynamic instrumentation framework, which allows you to use among other a JavaScript API to instrument the apps. One feature that we will use in bypassing jailbreak detection is to perform so-called early instrumentation, i.e. replace function implementation on startup.

1. First, ensure that frida-server is running on your iDevice
2. iDevice must be connected via USB cable
3. Use frida-trace on your workstation:

$ frida-trace -U -f /Applications/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp -m "-[JailbreakDetectionVC isJailbroken]"

This will actually start DamnVulnerableIOSApp, trace calls to -[JailbreakDetectionVC isJailbroken] and create JS hook with onEnter and onLeave callback functions. Now it's trivial to replace return value with value.replace() as shown in the example below:

onLeave: function (log, retval, state) {  
 console.log("Function [JailbreakDetectionVC isJailbroken] originally returned:"+ retval);  
 retval.replace(0);   
 console.log("Changing the return value to:"+retval);  
 }

Running this will have the following result:

```  
$ frida-trace -U -f /Applications/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp -m "-[JailbreakDetectionVC isJailbroken]:"

Instrumenting functions... `...  
-[JailbreakDetectionVC isJailbroken]: Loaded handler at "./**handlers**/\_\_JailbreakDetectionVC\_isJailbroken\_.js"  
Started tracing 1 function. Press Ctrl+C to stop.  
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1  
Changing the return value to:0x0  
/\* TID 0x303 \*/  
6890 ms -[JailbreakDetectionVC isJailbroken]  
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1  
Changing the return value to:0x0  
22475 ms -[JailbreakDetectionVC isJailbroken]  
```

Please note that there were two calls to -[JailbreakDetectionVC isJailbroken], which corresponds to two physical taps on the app GUI.

Frida is a very powerful and versatile tool. Refer to the documentation [3] to get more details.

-- TODO [a generic Frida script that catches many JB detection methods] --

Hooking Objective-C methods and native functions:

import frida  
import sys  
  
try:  
 session = frida.get\_usb\_device().attach("Target Process")  
except frida.ProcessNotFoundError:  
 print "Failed to attach to the target process. Did you launch the app?"  
 sys.exit(0);  
  
script = session.create\_script("""  
  
 // Handle fork() based check  
  
 var fork = Module.findExportByName("libsystem\_c.dylib", "fork");  
  
 Interceptor.replace(fork, new NativeCallback(function () {  
 send("Intercepted call to fork().");  
 return -1;  
 }, 'int', []));  
  
 var system = Module.findExportByName("libsystem\_c.dylib", "system");  
  
 Interceptor.replace(system, new NativeCallback(function () {  
 send("Intercepted call to system().");  
 return 0;  
 }, 'int', []));  
  
 // Intercept checks for Cydia URL handler  
  
 var canOpenURL = ObjC.classes.UIApplication["- canOpenURL:"];  
  
 Interceptor.attach(canOpenURL.implementation, {  
 onEnter: function(args) {  
 var url = ObjC.Object(args[2]);  
 send("[UIApplication canOpenURL:] " + path.toString());  
 },  
 onLeave: function(retval) {  
 send ("canOpenURL returned: " + retval);  
 }  
  
 });   
  
 // Intercept file existence checks via [NSFileManager fileExistsAtPath:]  
  
 var fileExistsAtPath = ObjC.classes.NSFileManager["- fileExistsAtPath:"];  
 var hideFile = 0;  
  
 Interceptor.attach(fileExistsAtPath.implementation, {  
 onEnter: function(args) {  
 var path = ObjC.Object(args[2]);  
 // send("[NSFileManager fileExistsAtPath:] " + path.toString());  
  
 if (path.toString() == "/Applications/Cydia.app" || path.toString() == "/bin/bash") {  
 hideFile = 1;  
 }  
 },  
 onLeave: function(retval) {  
 if (hideFile) {  
 send("Hiding jailbreak file...");MM  
 retval.replace(0);  
 hideFile = 0;  
 }  
  
 // send("fileExistsAtPath returned: " + retval);  
 }  
 });  
  
  
 /\* If the above doesn't work, you might want to hook low level file APIs as well  
  
 var openat = Module.findExportByName("libsystem\_c.dylib", "openat");  
 var stat = Module.findExportByName("libsystem\_c.dylib", "stat");  
 var fopen = Module.findExportByName("libsystem\_c.dylib", "fopen");  
 var open = Module.findExportByName("libsystem\_c.dylib", "open");  
 var faccesset = Module.findExportByName("libsystem\_kernel.dylib", "faccessat");  
  
 \*/  
  
""")  
  
def on\_message(message, data):  
 if 'payload' in message:  
 print(message['payload'])  
  
script.on('message', on\_message)  
script.load()  
sys.stdin.read()

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for static analysis of "Testing Jailbreak Detection" with source code] --

##### Without Source Code

-- TODO [Add content for static analysis of "Testing Jailbreak Detection" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Jailbreak Detection" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Jailbreak Detection".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference to "VX.Y" below for "Testing Jailbreak Detection"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Jailbreak Detection"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) - Jailbreak Detection Methods on the Trustware Spiderlabs Blog - <https://www.trustwave.com/Resources/SpiderLabs-Blog/Jailbreak-Detection-Methods/>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) - Dana Geist, Marat Nigmatullin: Jailbreak/Root Detection Evasion Study on iOS and Android - <http://delaat.net/rp/2015-2016/p51/report.pdf>
* [3] - <http://frida.re/>

##### Tools

-- TODO [Add relevant tools for "Testing Jailbreak Detection"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Anti-Debugging

#### Overview

Debugging is a highly effective way of analyzing the runtime behaviour of an app. It allows the reverse engineer to step through the code, stop execution of the app at arbitrary point, inspect and modify the state of variables, and a lot more.

-- TODO [Typical debugging defenses] --

typedef int (\*ptrace\_ptr\_t)(int \_request, pid\_t \_pid, caddr\_t \_addr, int \_data);  
  
#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"  
 );  
}

- (void)protectAgainstDebugger {  
 int junk;  
 int mib[4];  
 struct kinfo\_proc info;  
 size\_t size;  
   
 info.kp\_proc.p\_flag = 0;  
   
 // Initialize mib, which tells sysctl the info we want, in this case  
 // we're looking for information about a specific process ID.  
   
 mib[0] = CTL\_KERN;  
 mib[1] = KERN\_PROC;  
 mib[2] = KERN\_PROC\_PID;  
 mib[3] = getpid();  
   
  
 while(1) {  
   
 size = sizeof(info);  
 junk = sysctl(mib, sizeof(mib) / sizeof(\*mib), &info, &size, NULL, 0);  
 assert(junk == 0);  
  
 // We're being debugged if the P\_TRACED flag is set.  
   
 if ((info.kp\_proc.p\_flag & P\_TRACED) != 0) {  
 exit(0);  
 }  
   
 sleep(1);  
   
 }  
}

The app should either actively prevent debuggers from attaching, or terminate when a debugger is detected.

#### Bypassing Anti-Debugging Defenses

-- TODO [Bypass techniques] --

#import <substrate.h>  
  
#define PT\_DENY\_ATTACH 31  
  
static int (\*\_my\_ptrace)(int request, pid\_t pid, caddr\_t addr, int data);  
  
  
static int $\_my\_ptrace(int request, pid\_t pid, caddr\_t addr, int data) {  
 if (request == PT\_DENY\_ATTACH) {  
 request = -1;  
 }  
 return \_ptraceHook(request,pid,addr,data);  
}  
  
%ctor {  
 MSHookFunction((void \*)MSFindSymbol(NULL,"\_ptrace"), (void \*)$ptraceHook, (void \*\*)&\_ptraceHook);  
}

#### White-box Testing

-- TODO [Describe how to assess this with access to the source code and build configuration] --

#### Black-box Testing

-- TODO [Needs more detail] --

Attach a debugger to the running process. This should either fail, or the app should terminate or misbehave when the debugger has been detected. For example, if ptrace(PT\_DENY\_ATTACH) has been called, gdb will crash with a segmentation fault:

Note that some anti-debugging implementations respond in a stealthy way so that changes in behaviour are not immediately apparent. For example, a soft token app might not visibly respond when a debugger is detected, but instead secretly alter the state of an internal variable so that an incorrect OTP is generated at a later point. Make sure to run through the complete workflow to determine if attaching the debugger causes a crash or malfunction.

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Anti-Debugging"] --

#### References

##### OWASP Mobile Top 10 2014

-- TODO [Add link to OWASP Mobile Top 10 2014 for "Testing Anti-Debugging"] --

##### OWASP MASVS

* V...: ""

##### CWE

-- TODO [Add relevant CWE for "Testing Anti-Debugging"] --

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>

##### Tools

-- TODO [Add tools for "Testing Anti-Debugging"] --

### Testing File Integrity Checks

#### Overview

-- TODO [Implementation from UnCrackable2] --

int xyz(char \*dst) {  
 const struct mach\_header \* header;  
 Dl\_info dlinfo;  
  
 if (dladdr(xyz, &dlinfo) == 0 || dlinfo.dli\_fbase == NULL) {  
 NSLog(@" Error: Could not resolve symbol xyz");  
 [NSThread exit];  
 }  
  
 while(1) {  
   
 header = dlinfo.dli\_fbase; // Pointer on the Mach-O header  
 struct load\_command \* cmd = (struct load\_command \*)(header + 1); // First load command  
 // Now iterate through load command  
 //to find \_\_text section of \_\_TEXT segment  
 for (uint32\_t i = 0; cmd != NULL && i < header->ncmds; i++) {  
 if (cmd->cmd == LC\_SEGMENT) {  
 // \_\_TEXT load command is a LC\_SEGMENT load command  
 struct segment\_command \* segment = (struct segment\_command \*)cmd;  
 if (!strcmp(segment->segname, "\_\_TEXT")) {  
 // Stop on \_\_TEXT segment load command and go through sections  
 // to find \_\_text section  
 struct section \* section = (struct section \*)(segment + 1);  
 for (uint32\_t j = 0; section != NULL && j < segment->nsects; j++) {  
 if (!strcmp(section->sectname, "\_\_text"))  
 break; //Stop on \_\_text section load command  
 section = (struct section \*)(section + 1);  
 }  
 // Get here the \_\_text section address, the \_\_text section size  
 // and the virtual memory address so we can calculate  
 // a pointer on the \_\_text section  
 uint32\_t \* textSectionAddr = (uint32\_t \*)section->addr;  
 uint32\_t textSectionSize = section->size;  
 uint32\_t \* vmaddr = segment->vmaddr;  
 char \* textSectionPtr = (char \*)((int)header + (int)textSectionAddr - (int)vmaddr);  
 // Calculate the signature of the data,  
 // store the result in a string  
 // and compare to the original one  
 unsigned char digest[CC\_MD5\_DIGEST\_LENGTH];  
 CC\_MD5(textSectionPtr, textSectionSize, digest); // calculate the signature  
 for (int i = 0; i < sizeof(digest); i++) // fill signature  
 sprintf(dst + (2 \* i), "%02x", digest[i]);  
   
 // return strcmp(originalSignature, signature) == 0; // verify signatures match  
   
 return 0;  
 }  
 }  
 cmd = (struct load\_command \*)((uint8\_t \*)cmd + cmd->cmdsize);  
 }  
 }  
   
}

#### Bypassing File Integrity Checks

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for "Testing File Integrity Checks" with source-code] --

##### Without Source Code

-- TODO [Add content for "Testing File Integrity Checks" without source-code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing File Integrity Checks" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing File Integrity Checks".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing File Integrity Checks"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing File Integrity Checks"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Android Cracking Blog - <http://androidcracking.blogspot.com/2011/06/anti-tampering-with-crc-check.html>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing File Integrity Checks"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Detection of Reverse Engineering Tools

#### Overview

-- TODO [Provide a general description of the issue "Testing Detection of Reverse Engineering Tools".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content of static analysis of "Testing Detection of Reverse Engineering Tools" with source code] --

##### Without Source Code

-- TODO [Add content of static analysis of "Testing Detection of Reverse Engineering Tools" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Detection of Reverse Engineering Tools" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Detection of Reverse Engineering Tools".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Detection of Reverse Engineering Tools"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Detection of Reverse Engineering Tools"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Detection of Reverse Engineering Tools"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Memory Integrity Checks

#### Overview

-- TODO [Provide a general description of the issue "Testing Memory Integrity Checks".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for static analysis of "Testing Memory Integrity Checks" with source code] --

##### Without Source Code

-- TODO [Add content for static analysis of "Testing Memory Integrity Checks" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Memory Integrity Checks" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Memory Integrity Checks".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference below "VX.Y" for "Testing Memory Integrity Checks"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Memory Integrity Checks"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Memory Integrity Checks"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Device Binding

#### Overview

-- TODO [Provide a general description of the issue "Testing Device Binding".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for static analysis of "Testing Device Binding" with source code] --

##### Without Source Code

-- TODO [Add content for static analysis of "Testing Device Binding" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Device Binding" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Device Binding".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Device Binding"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Device Binding"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Device Binding"] --

* Enjarify - <https://github.com/google/enjarify>

### Testing Obfuscation

#### Overview

-- TODO [Provide a general description of the issue "Testing Obfuscation".] --

#### Static Analysis

-- TODO [Describe how to assess this given either the source code or installer package (APK/IPA/etc.), but without running the app. Tailor this to the general situation (e.g., in some situations, having the decompiled classes is just as good as having the original source, in others it might make a bigger difference). If required, include a subsection about how to test with or without the original sources.] --

-- TODO [Confirm purpose of remark "Use the <sup> tag to reference external sources, e.g. Meyer's recipe for tomato soup[1](https://github.com/pillfill/hiding-passwords-android/)."] --

##### With Source Code

-- TODO [Add content for static analysis of "Testing Obfuscation" with source code] --

##### Without Source Code

-- TODO [Add content for static analysis of "Testing Obfuscation" without source code] --

#### Dynamic Analysis

-- TODO [Describe how to test for this issue "Testing Obfuscation" by running and interacting with the app. This can include everything from simply monitoring network traffic or aspects of the app’s behavior to code injection, debugging, instrumentation, etc.] --

#### Remediation

-- TODO [Describe the best practices that developers should follow to prevent this issue "Testing Obfuscation".] --

#### References

##### OWASP Mobile Top 10 2016

* M9 - Reverse Engineering - <https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering>

##### OWASP MASVS

-- TODO [Update reference "VX.Y" below for "Testing Obfuscation"] --

* VX.Y: "Requirement text, e.g. 'the keyboard cache is disabled on text inputs that process sensitive data'."

##### CWE

-- TODO [Add relevant CWE for "Testing Obfuscation"] --

* CWE-312 - Cleartext Storage of Sensitive Information

##### Info

* [1](https://github.com/pillfill/hiding-passwords-android/) Meyer's Recipe for Tomato Soup - <http://www.finecooking.com/recipes/meyers-classic-tomato-soup.aspx>
* [2](https://developer.android.com/reference/java/security/KeyStore.html) Another Informational Article - <http://www.securityfans.com/informational_article.html>

##### Tools

-- TODO [Add relevant tools for "Testing Obfuscation"] --

* Enjarify - <https://github.com/google/enjarify>

# Appendix

## Security Testing in the Software Development life cycle

The history of software development is not that old after all, and it is easy to see that, rapidly, teams have stopped developing programs without any framework: we have all experienced the fact that, as the number of lines of code grows, a minimal set of rules are needed in order to keep work under control, meet deadlines, quality and budgets.

In the past, the most widely adopted methodologies were from the "Waterfall" family: development was done from a starting point to a final one, going through several steps, each of them happening one after the other in a predefined sequence. In case something was wrong during a given phase and something had to be changed in a former phase, it was possible to go only one step backward. This was a serious drawback of Waterfall methodologies. Even if they have strong positive points (bring structure, clarify where to put effort, clear and easy to understand, ...), they also have negative ones (creation of silos, slow, specialized teams, ...).

As time was passing and software development was maturing, also competition was getting stronger and stronger, and a need to react faster to market changes while creating software products with smaller budgets rose. The idea of having fewer structure with smaller teams collaborating together, breaking silos through the organization from marketing to production, became popular. The "Agile" concept was born (well known examples of Agile implementations are Scrum, XP and RAD), which was enabling more autonomous teams to work together in a faster manner.

Originally, security was not part of software development. It was seen as an afterthought, and was performed by Operation teams at the network level: those teams had to find ways to compensate for poor security in software programs! However, while this was possible when software programs were located inside a perimeter, the concept became obsolete as new ways to consume software emerged with Web and Mobile technologies. Nowadays, security has to be baked **inside** software as it is often very hard in this new paradigm to compensate for existing vulnerabilities.

The way to incorporate security during software development is to put in place a Secure SDLC (Software Development Life Cycle). A Secure SDLC does not depend on any methodology nor on any language, and it is possible to incorporate one in Waterfall or Agile: no excuse not to use one!  
This chapter will focus on Agile and Secure SDLC, in particular in the DevOps world. The reader will find below details on state-of-the-art ways to develop and deliver secure software in a fast-paced and collaborative manner that promotes autonomy and automation.

### Agile and DevOps

#### DevOps

DevOps refers to practices that focus on a close collaboration between all stakeholders involved in delivering software. DevOps is the logical evolution of Agile in that it enables software to be released to users as rapidly as possible. Besides the collaboration aspect, to a large extent, this is facilitated through heavy automation of the build, test and release process of software and infrastructure changes. This automation is embodied in the deployment pipeline.

##### -- Todo [Add deployment pipeline overview and description specific for mobile apps.] --

The term DevOps might be mistaken for only expressing collaboration between development and operations teams, however, as Gene Kim, a DevOps thought leader, puts it: “At first blush, it seems as though the problems are just between dev and ops," he says, "but test is in there, and you have information security objectives, and the need to protect systems and data. These are top-level concerns of management, and they have become part of the DevOps picture."

In other words, when you hear "DevOps" today, you should probably be thinking DevOpsQATestInfoSec.” (Source: <https://techbeacon.com/evolution-devops-new-thinking-gene-kim>)

Security is just as important for the business success as the overall quality, performance and usability of an application. As development cycles are shortened and deployment frequencies increased it is elementary to ensure that quality and security is built in from the very beginning.

From the human aspect, this is achieved by creating cross functional teams that work together on achieving business outcomes. This section is going to focus on the interaction with and integration of security into the development life cycle, from the inception of requirements, all the way until the value of the change is made available to users.

### General Considerations

* Release time for apple store
* Why are black listed, and how to avoid it.
* Common gotchas: Ensure that the app is always fully removed and re-installed. Otherwise there might be issues that are hard to re-produce.

### SDLC Overview

#### General description of SDLC

Whatever the development methodology that is being used, a SDLC always follows the same process:

* perform a risk assessment of the application and it's components to identify their risk profile. This risk profile typically depends on the risk appetite of the organization and the regulatory requirements for the application in scope. The risk assessment is additionally influenced by other factors such as whether the application is accessible from the internet, or what kind of data is processed and stored. A data classification policy determines which data is considered sensitive and prescribes how this data has to be secured.
* at the beginning of a project or a development cycle, at the same time when functional requirements are gathered, **Security Requirements** are listed and clarified. As use cases are built, **Abuse Cases** are added. Also, **Security Risks** are analysed, the same way other risks of the project (financial, marketing, industrial, ...) are handled. Teams (including development teams) may be trained on security if needed (Secure Coding, ...);
* for mobile applications the OWASP MASVS [todo: link to the other guide] can be leveraged to determine the security requirements based on the risk assessment that was conducted in this initial step. It is common, especially for agile projects, to iteratively review the set of requirements based on newly added features and new classes of data that is handled by the application.
* then, as architecture and design are ongoing, a foundational artefact must be performed: **Threat Modeling**. Based on the threat model, the **Security Architecture** is defined (both for software and hardware aspects). **Secure Coding rules** are established and the list of **Security tools** that will be used is created. Also, the strategy for **Security testing** is clarified;
* all security requirements and design considerations should be stored in the Application Life cycle Management System (ALM), which is typically known as issue tracker, that the development/ops team already uses to ensure that security requirements are tightly integrated into the development workflow. The security requirements should ideally also contain the relevant source code snippets for the used programming language, to ensure that developers can quickly reference them. Another strategy for secure coding guidelines is to create a dedicated repository under versioning control, that only contains these ode snippets, which has many benefits over the traditional apporach of storing these guidelines in word documents or PDFs.
* the next step is to develop software, including **Code Reviews** (usually with peers), **Static Analysis** with automated tools and **Unit Tests** dedicated to security;
* then comes the long-awaited moment to perform tests on the release candidate: **Penetration Testing** ("Pentests"), using both manual and automated techniques;
* and finally, after software has been **Accredited** by all stakeholders, it can be transitioned to Operation teams and safely put in Production.

The picture below shows all the phases with the different artefacts:  
-- TODO [Add a picture of a SDLC diagram that clarifies the description above] --

Based on the risks of the project, some artefacts may be simplified (or even skipped) while others may be added (formal intermediary approvals, documentation of certain points, ...). **Always keep in mind a SDLC is meant to bring risk reduction to software development and is a framework that helps put in place controls that will reduce those risks to an acceptable level.**  While this is a generic description of SDLC, always tailor this framework to the needs of your projects.

#### Diving into phases and artefacts

Now, let's have a closer look at the five phases listed above and let's clarify their main purposes, what is done while they take place and who performs them:

* **Initiation** phase: this is the first phase of a project, when requirements are gathered from the field and defined for the project. They should include both functional (e.g. what features will be created for the end user) and security (e.g. what security features will need to be implemented to allow end users to trust the software product) requirements. In this phase, all activities that need to happen before technical work starts and all others that can be anticipated will take place. This is also the moment when Proof of Concepts may be done and when the project viability is confirmed. Typically, teams close to business functions such as marketing (marketing people, or product owners, ...), management and finance are involved.
* **Architecture and Design** phase: after the project has been confirmed, the technical team will start working on early technical activities that will enable coding teams to be productive. In this matter, risks are analysed and relevant countermeasures identified and clarified. The architecture and coding approach as well as testing strategies and the appropriate tools are confirmed, and the different environnements (e.g. DEV, QA, SIT, UAT, PROD) are created and put in place. This phase is pivotal as its main goal is to go from a non-technical definition of needs to the point where technical teams are ready to give birth to code that will make up the software product. Typically, Architects, Designers, QA teams, Testers and AppSec experts are involved.
* **Coding** phase: this is the moment when code is produced and efforts become visible. This may be seen as the most important phase; however, one must keep in mind that all activities happening before and after the current phase are meant to support code creation and make sure it reaches proper standards for quality and security while meeting deadlines and budgets. In this phase, development teams work in the defined environnement to implement requirements following previously defined guidelines. The main people who are involved are developers.
* **Testing** phase: this is the phase when produced software is tested. As testing can take many forms (see detailed section on Security Testing in the SDLC below), testing activities may be performed during coding (the obvious goal being to discover issues as soon as possible). Depending on organizations, the project risk profile and techniques used, testing teams may be independent from coding teams. The main people involved during this phase are Testers. Test cases should exist that map tightly to the established security requirements and that are ideally presented in a way that allows codification and subsequently automated verification.
* **Release** phase: at this point of time, code has been created and tested. Its security level has been assessed; often, metrics are produced to support evidence that code meets the expected level of security. However, it now has to be transitioned to the Customer, e.g. it has to be accepted by stakeholders (Management, Marketing, ...) as able to create value on the market and be of economical interest to Customers; next to that, it will be made available to the market. It is not enough to produce secure software, but it now has to be safely transitioned to Production environnements, which in turn must be secured (both in the short term and in the long term); documentation for Operation teams may be created. In this phase, stakeholders (Management, Marketing, ...) are first involved, as well as technical teams (Testing, Operations, Quality, ...).

Even if the previous description seems to be "Waterfall-like", it also applies to Agile methodologies: the same logic is used, but in a more iterative manner. Some activities may be done only once (for instance project initiation), however smaller parts of similar activities will happen regularly all along the project (like bringing new requirements into light and clarifying them into user stories). In the same manner, testing will not happen only once at the end of a project, but, on each iteration, tests will focus on the amount of code that was produced in the iteration. This in-cycle testing is preferred over the out-of-cycle approach, because the longer it takes for developers to receive feedback, the more time it will take them to make the context switch.

### Security Testing in the SDLC

#### Overview

A well-known statement in software development (and many other fields as well) is that the sooner tests take place, the easier and more cost-effective it is to fix a defect. The same applies to defects related to cyber security: identifying (and fixing) vulnerabilities early in the development life cycle gives better results when it comes to produce secure software. In some ways, Quality Testing and Security Testing may share common aspects as both are meant to raise customer satisfaction.

Testing can be performed in many forms during the life cycle: using automated tools like Static Analysis, writing Unit Tests as code is being created, running Penetration Tests (either manually or with the help of scanning tools) after software has been developed. However, an emphasis should always be put on planning and preparing these efforts early in the Secure SDLC: a Test Plan should be initiated and developed at the beginning of the project, listing and clarifying the kind of tests that will be executed, their scope, how and when they will take place and with what budget. Also, abuse cases should be written early in the project (ideally at the same time when use cases are created) to provide guidance to test teams all along development. Finally, an artefact that should always be considered is Threat Modeling, which allow teams to focus on the right components in the architecture with the proper tests and proper coverage to ensure that the security controls have been implemented correctly.

The following diagram provides an overview of the way to perform test in the SDLC:

-- TODO [Add diagram to summarize the above paragraph and clarify the way test should be performed (planned, executed and reviewed)] --

#### Detailed Description

As stated before, several kinds of tests can be made along the SDLC. According to the risk profile of the targeted software, several kind of tests can be performed:

* **Static Analysis**: by nature, static analysis is about analysing source code without running it. The goal of this artefact is twofold: make sure the Secure Coding rules the team has agreed on are correctly implemented when writing code, and finding vulnerabilities. Often, specialized software tools are used to automate this task, as hundreds and thousands of lines of code may need to be analysed. However, the drawback is that tools can only find what they have been told to look for and, today, are not as successful as human beings. This is the reason why sometimes Static Analysis is performed by humans (in addition to tools or not): it may take more time for humans, but they have a more creative way to detect vulnerabilities. Examples of tools for Static Analysis are given in another section.
* **Unit Tests**: unit tests make up the family of tests that are the closest to source code (e.g. they are focused on a single unit) as they are performed along with code. According to the methodology in use, they can be created either before code is developed (known as Test Driven Development (TDD)) or right after. Whatever the case, the end goal is to verify that produced code is behaving as expected, but also that proper controls are put in place to prevent abuse cases (input filtering / validation, whitelisting, ...) and cannot be circumvented. Unit Tests are used to detect issues early in the development life cycle in order to fix them as quickly and effectively as possible. They are different from other forms of tests, like Integration / Verification / Validation tests, and may not be used to detect the same kind of issue. Often, Unit Tests are aided with tools; a few of them are listed in another section.
* **Penetration Testing**: this is the "king" of security tests, the one that is the most famous and often performed. However, one must keep in mind that they happen late in the development life cycle and that they cannot find every kind of flaw. They are too often constrained by available resources (time, money, expertise, ...), and as such should be complemented by other kind of tests. The current guide is about pentesting, and the reader will find a lot of useful information to conduct added-value tests and find even more vulnerabilities. Pentesting techniques include vulnerability scanning and fuzzing; however, Penetration Tests can be much more multi-facetted than these two examples. Useful tools are listed in another section.

A clear difference shall be made between quality testing and security testing: while quality testing is about making sure an explicitely planned feature has been implemented in the proper way, security testing is about making sure that:

* existing features cannot be used in a malicious way
* no new feature has unvoluntarily been introduced that could endanger the system or its users.

As a consequence, performing one type of tests is not enough to pretend having covered both types and that the produced software is both usable and secure. The same care should be given to both types of tests as they are of the same importance and that final users now put a strong emphasis both on quality (e.g. the fact features that are brought to them perform the way they expect them to) and security (e.g. that they can trust the software vendor that their money will not be stolen or their private life will remain private).

#### Defining a Test Strategy

The purpose of a test strategy is to define which tests will be performed all along the SDLC and how often. Its goal is twofold: make sure security objectives are met by the final software product, which are generally expressed by customers/legal/marketing/corporate teams, while being cost-effective. The test strategy is generally created at the beginning of a project, after risks have been clarified (Initiation phase) but before code production (Coding phase) starts. It generally takes place during the Architecture and Design phase. It takes inputs from activities such as Risk Management, Threat Modeling, Security Engineering, etc.

-- TODO [Add diagram (in the form of a workflow) showing inputs of a Test Strategy, and outputs (test cases, ...)] --

A Test Strategy does not always need to be formally written: it may be described through Stories (in Agile projects), quickly written in the form of checklists, or test cases could be written in a given tool; however, it definitely needs to be shared, as it may be defined by the Architecture team, but will have to be implemented by other teams such as Development, Testing, QA. Moreover, it needs to be agreed upon by all technical teams as it should not place unacceptable burdens on any of them.

Ideally, a Test Strategy addresses topics such as:

* objectives to be met and description of risks to be put under control.
* how these objectives will be met and risks reduced to an acceptable level: which tests will be mandatory, who will perform them, how, when, and at which frequency.
* acceptance criteria of the current project.

In order to follow its effectiveness and progress, metrics should be defined, updated all along the project and periodically communicated. An entire book could be written on the relevant metrics to choose; the best that can be said is that they depend on risk profiles, projects and organizations. However, some examples of metrics include:

* the number of stories related to security controls that are implemented,
* code coverage for unit tests on security controls and sensitive features,
* the number of security bugs found by static analysis tools upon each build,
* the trend of the backlog for security bugs (may be sorted by criticality).  
  These are only suggestions, and other metrics may be even more relevant in your case. Metrics are really powerful tools to get a project under control, provided they give a clear view and in a timely manner to project managers on what is happening and what needs to be improved to reach targets.

--

### Testing methods

#### White box

#### Grey box

#### Black box

### Team management

-- TODO [Develop content on Team Management in SDLC] --

* explain the importance of Separation of Duties (developers VS testers, ...)
* internal VS sub-contracted pentests

### Security Testing in the DevOps environments

#### Overview

As the frequency of deployments to production increases, and DevOps high-performers deploy to production many times a day, it is elementary to automated as many of the security verification tasks as possible. The best approach to facilitate that is by integrating security into the deployment pipeline. A deployment pipeline is a combination of continuous integration and continous delivery practices, which have been created to facilitate rapid development and receive almost instantaneous feedback upon every commit. More details on the deployment pipeline are provided in the section below.

#### The Deployment Pipeline

Depending on the maturity of the organisation, or development team, the deployment pipeline can be very sophisticated. In it's simplest form, the deployment pipeline consists of a commit phase. The commit phase commonly runs simple compiler checks, the unit tests suite, as well as creates a deployable artefact of the application which is called release candidate. A release candidate is the latest version of changes that has been checked into the trunk of the versioning control system and will be evaluated by the deployment pipeline to verify if it is in-line with the established standards to be potentially deployed to production.

The commit phase is designed to provide instant feedback to developers and as such is run on every commit to the trunk. Because of that, certain time constraints exists. Typically, the commit phase should run within 5 minutes, but in any case, shouldn't take longer than 10 minutes to complete. This time constraint is quite challenging in the security context, as many of the currently existing tools can't run in that short amount of time.

Todo: Automating security tools in Jenkins,...

### References

* Official (ISC)2 Guide to the CSSLP (ISC2 Press), Mano Paul - <https://www.amazon.com/Official-Guide-CSSLP-Second-Press/dp/1466571276/>
* Software Security: Building Security In (Addison-Wesley Professional), Gary McGraw - <https://www.amazon.com/Software-Security-Building-Gary-McGraw/dp/0321356705/>

## Testing Tools

To perform security testing different tools are available in order to be able to manipulate requests and responses, decompile Apps, investigate the behaviour of running Apps and other test cases and automate them.

* [Mobile Application Security Testing Distributions](#mobile-application-security-testing-distributions)
* [All-in-one Mobile Security Frameworks](#all-in-one-mobile-security-frameworks)
* [Tools for Android](#tools-for-android)
* [Reverse Engineering and Static Analysis](#reverse-engineering-and-static-analysis)
* [Dynamic and Runtime Analysis](#dynamic-and-runtime-analysis)
* [Bypassing Root Detection and SSL Pinning](#bypassing-root-detection-and-ssl-pinning)
* [Tools for iOS](#tools-for-ios)
* [Access Filesystem on iDevice](#access-filesystem-on-idevice)
* [Reverse Engineering and Static Analysis](#reverse-engineering-and-static-analysis)
* [Dynamic and Runtime Analysis](#dynamic-and-runtime-analysis)
* [Bypassing Root Detection and SSL Pinning](#bypassing-root-detection-and-ssl-pinning)
* [Tools for Network Interception](#tools-for-network-interception)
* [IDE](#ide)

### Mobile Application Security Testing Distributions

* [Appie](https://manifestsecurity.com/appie) - A portable software package for Android Pentesting and an awesome alternative to existing Virtual machines.
* [Android Tamer](https://androidtamer.com/) - Android Tamer is a Virtual / Live Platform for Android Security professionals.
* [AppUse](https://appsec-labs.com/AppUse/) - AppUse is a VM (Virtual Machine) developed by AppSec Labs.
* [Androl4b](https://github.com/sh4hin/Androl4b) - A Virtual Machine For Assessing Android applications, Reverse Engineering and Malware Analysis
* [Mobisec](http://sourceforge.net/projects/mobisec/) - Mobile security testing live environment.
* [Santoku](https://santoku-linux.com/) - Santoku is an OS and can be run outside a VM as a standalone operating system.
* [Vezir Project](https://github.com/oguzhantopgul/Vezir-Project) - Mobile Application Pentesting and Malware Analysis Environment.

### All-in-One Mobile Security Frameworks

* [Mobile Security Framework - MobSF](https://github.com/ajinabraham/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](https://github.com/mwrlabs/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.

### Tools for Android

#### Reverse Engineering and Static Analysis

* [Androguard](https://github.com/androguard/androguard) - Androguard is a python based tool, which can use to disassemble and decompile android apps.
* [Android Debug Bridge - adb](https://developer.android.com/studio/command-line/adb.html) - Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android device.
* [APKInspector](https://github.com/honeynet/apkinspector/) - APKinspector is a powerful GUI tool for analysts to analyze the Android applications.
* [APKTool](http://ibotpeaches.github.io/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.
* [Sign](https://github.com/appium/sign) - Sign.jar automatically signs an apk with the Android test certificate.
* [Jadx](https://github.com/skylot/jadx) - Dex to Java decompiler: Command line and GUI tools for produce Java source code from Android Dex and Apk files.
* [Oat2dex](https://github.com/testwhat/SmaliEx) - A tool for converting .oat file to .dex files.
* [FindBugs](http://findbugs.sourceforge.net/) + [FindSecurityBugs](http://h3xstream.github.io/find-sec-bugs/) - FindSecurityBugs is a extension for FindBugs which include security rules for Java applications.
* [Qark](https://github.com/linkedin/qark) - This tool is designed to look for several security related Android application vulnerabilities, either in source code or packaged APKs.
* [SUPER](https://github.com/SUPERAndroidAnalyzer/super) - SUPER is a command-line application that can be used in Windows, MacOS X and Linux, that analyzes .apk files in search for vulnerabilities. It does this by decompressing APKs and applying a series of rules to detect those vulnerabilities.
* [AndroBugs](https://github.com/AndroBugs/AndroBugs_Framework) - 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](https://github.com/HamiltonianCycle/ClassNameDeobfuscator) - Simple script to parse through the .smali files produced by apktool and extract the .source annotation lines.
* [Android backup extractor](https://github.com/nelenkov/android-backup-extractor) - Utility to extract and repack Android backups created with adb backup (ICS+). Largely based on BackupManagerService.java from AOSP.

#### Dynamic and Runtime Analysis

* [Cydia Substrate](http://www.cydiasubstrate.com/) - Cydia Substrate for Android enables developers to make changes to existing software with Substrate extensions that are injected in to the target process's memory.
* [Xposed Framework](http://forum.xda-developers.com/xposed/xposed-installer-versions-changelog-t2714053) - Xposed framework enables you to modify the system or application aspect and behaviour at runtime, without modifying any Android application package(APK) or re-flashing.
* [logcat-color](https://github.com/marshall/logcat-color) - A colorful and highly configurable alternative to the adb logcat command from the Android SDK.
* [Inspeckage](https://github.com/ac-pm/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](http://www.frida.re/) - The toolkit works using a client-server model and lets you inject in to running processes not just on Android, but also on iOS, Windows and Mac.
* [Diff-GUI](https://github.com/antojoseph/diff-gui) - A Web framework to start instrumenting with the avaliable modules, hooking on native, inject JavaScript using Frida.
* [AndBug](https://github.com/swdunlop/AndBug) - AndBug is a debugger targeting the Android platform's Dalvik virtual machine intended for reverse engineers and developers.
* [Cydia Substrate: Introspy-Android](https://github.com/iSECPartners/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](https://www.mwrinfosecurity.com/products/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.

#### Bypassing Root Detection and SSL Pinning

* [Xposed Module: Just Trust Me](https://github.com/Fuzion24/JustTrustMe) - Xposed Module to bypass SSL certificate pinning.
* [Xposed Module: SSLUnpinning](https://github.com/ac-pm/SSLUnpinning_Xposed) - Android Xposed Module to bypass SSL certificate validation (Certificate Pinning).
* [Cydia Substrate Module: Android SSL Trust Killer](https://github.com/iSECPartners/Android-SSL-TrustKiller) - Blackbox tool to bypass SSL certificate pinning for most applications running on a device.
* [Cydia Substrate Module: RootCoak Plus](https://github.com/devadvance/rootcloakplus) - Patch root checking for commonly known indications of root.
* [Android-ssl-bypass](https://github.com/iSECPartners/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](https://filezilla-project.org/download.php?show_all=1) - It supports FTP, SFTP, and FTPS (FTP over SSL/TLS).
* [Cyberduck](https://cyberduck.io) - Libre FTP, SFTP, WebDAV, S3, Azure & OpenStack Swift browser for Mac and Windows.
* [itunnel](https://code.google.com/p/iphonetunnel-usbmuxconnectbyport/downloads/list) - Use to forward SSH via USB.
* [iFunbox](http://www.i-funbox.com) - The File and App Management Tool for iPhone, iPad & iPod Touch.

#### Reverse Engineering and Static Analysis

* [otool](http://www.unix.com/man-page/osx/1/otool/) - The otool command displays specified parts of object files or libraries.
* [Clutch](http://cydia.radare.org/) - Decrypted the application and dump specified bundleID into binary or .ipa file.
* [Dumpdecrypted] (<https://github.com/stefanesser/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](http://stevenygard.com/projects/class-dump/) - A command-line utility for examining the Objective-C runtime information stored in Mach-O files.
* [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](https://www.hex-rays.com/products/ida/index.shtml) - 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](http://hopperapp.com/) - 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](http://www.radare.org/) - Radare2 is a unix-like reverse engineering framework and commandline tools.
* [iRET](https://www.veracode.com/iret-ios-reverse-engineering-toolkit) - The iOS Reverse Engineering Toolkit is a toolkit designed to automate many of the common tasks associated with iOS penetration testing.

#### Dynamic and Runtime Analysis

* [cycript](http://www.cycript.org) - 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](https://appsec-labs.com/cydia/) - AppSec Labs iNalyzer is a framework for manipulating iOS applications, tampering with parameters and method.
* [idb](https://github.com/dmayer/idb) - idb is a tool to simplify some common tasks for iOS pentesting and research.
* [snoop-it](http://cydia.radare.org/) - A tool to assist security assessments and dynamic analysis of iOS Apps.
* [Introspy-iOS](https://github.com/iSECPartners/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](http://cydia.radare.org/) - A tool to perform runtime analysis of IOS applications.
* [keychaindumper](http://cydia.radare.org/) - A tool to check which keychain items are available to an attacker once an iOS device has been jailbroken.
* [BinaryCookieReader](http://securitylearn.net/wp-content/uploads/tools/iOS/BinaryCookieReader.py) - A tool to dump all the cookies from the binary Cookies.binarycookies file.

#### Bypassing Root Detection and SSL Pinning

* [SSL Kill Switch 2](https://github.com/nabla-c0d3/ssl-kill-switch2) - Blackbox tool to disable SSL certificate validation - including certificate pinning - within iOS and OS X Apps.
* [iOS TrustMe](https://github.com/intrepidusgroup/trustme) - Disable certificate trust checks on iOS devices.
* [Xcon](http://apt.modmyi.com) - A tool for bypassing Jailbreak detection.
* [tsProtector] (<http://cydia.saurik.com/package/kr.typostudio.tsprotector8/>) - Another tool for bypassing Jailbreak detection.

### Tools for Network Interception

* [Tcpdump](http://www.androidtcpdump.com) - A command line packet capture utility.
* [Wireshark](https://www.wireshark.org/download.html) - An open-source packet analyzer.
* [Canape](http://www.contextis.com/services/research/canape/) - A network testing tool for arbitrary protocols.
* [Mallory](https://intrepidusgroup.com/insight/mallory/) - A Man in The Middle Tool (MiTM) that use to monitor and manipulate traffic on mobile devices and applications.
* [Burp Suite](https://portswigger.net/burp/download.html) - Burp Suite is an integrated platform for performing security testing of applications.
* [Charles Proxy](http://www.charlesproxy.com) - 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.
* [OWASP ZAP](https://github.com/zaproxy/zaproxy) - 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](http://www.telerik.com/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.

### IDEs

* [IntelliJ](https://www.jetbrains.com/idea/download/) - IntelliJ IDEA is a Java integrated development environment (IDE) for developing computer software.
* [Eclipse](https://eclipse.org/) - Eclipse is an integrated development environment (IDE) used in computer programming, and is the most widely used Java IDE.

## Suggested Reading

### Basic Knowledge

-- TODO [Add suggested reading on Basic Knowledge ] --

### 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>

#### 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 Backaalany (2014) *Practical Reverse Engineering*. Wiley. Available at: <http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118787315,subjectCd-CSJ0.html>
* Skakenunny, Hangcom *iOS App Reverse Engineering*. Online. Available at: <https://github.com/iosre/iOSAppReverseEngineering/>
* Bernhard Mueller (2016) *Hacking Soft Tokens - Advanced Reverse Engineering on Android*. HITB GSEC Singapore. Available at:
* Dennis Yurichev (2016) *Reverse Engineering for Beginners*. Online. Available at: <https://github.com/dennis714/RE-for-beginners>