Introduction to Mobile Security Testing

Approaches and Examples using OWASP MSTG

OWASP German Day 20.11.2018

Carlos Holguera
Carlos Holguera [olˈɣera]

- Security Engineer working at ESCRYPTE GmbH since 2012

- Area of expertise:
  - Mobile & Automotive Security Testing
  - Security Testing Automation
Index

1. Why?
2. From the Standard to the Guide
3. Vulnerability Analysis
4. Information Gathering
5. Penetration Testing
6. Final Demos
1 Why?
Why?

- Trustworthy sources?
- Right Methodology?
- Latest Techniques?

☑ MASVS is the WHAT
☑ MSTG is the HOW

Online videos, articles, trainings ??
2 From the Standard to the Guide
From the Standard to the Guide

ONE DOES NOT SIMPLY
SECURITY TEST
From the Standard to the Guide
OWASP Mobile Application Security Verification Standard

This is the official Github Repository of the OWASP Mobile Application Security Verification Standard (MASVS). The MASVS establishes baseline security requirements for mobile apps that are useful in many scenarios, including:

- In the SDLC - to establish security requirements to be followed by solution architects and developers;
- In mobile app penetration tests - to ensure completeness and consistency in mobile app penetration tests;
- In procurement - as a measuring stick for mobile app security, e.g. in form of questionnaire for vendors;
- Et cetera.

The MASVS is a sister project of the OWASP Mobile Security Testing Guide.

Getting the MASVS

PDF downloads are available on the Releases page. The current release is MASVS version 1.1. The MASVS is also available in different languages:

- Spanish
- Russian

Open on GitHub  Read it on GitBook
# Security Verification Requirements

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>L1</th>
<th>L2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5.2</td>
<td>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.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5.3</td>
<td>The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5.4</td>
<td>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.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>The app doesn’t rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery.</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>The app only depends on up-to-date connectivity and security libraries.</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
## From the Standard to the Guide

**OWASP Mobile Application Security Verification Standard**

<table>
<thead>
<tr>
<th>Mobile Application Security Requirements - Android</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ID</strong></td>
</tr>
<tr>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>V1.1</td>
</tr>
<tr>
<td>V1.2</td>
</tr>
<tr>
<td>V1.3</td>
</tr>
<tr>
<td>V1.4</td>
</tr>
<tr>
<td>V1.5</td>
</tr>
<tr>
<td>V1.6</td>
</tr>
<tr>
<td>V1.7</td>
</tr>
<tr>
<td>V1.8</td>
</tr>
<tr>
<td>V1.9</td>
</tr>
<tr>
<td>V1.10</td>
</tr>
</tbody>
</table>

### V2 Data Storage and Privacy

<table>
<thead>
<tr>
<th>ID</th>
<th>Detailed Verification Requirement</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Status</th>
<th>Testing Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Verify that system credential storage facilities are used appropriately to store sensitive data, such as user credentials or cryptographic keys.</td>
<td>✓</td>
<td>✓</td>
<td>Pass</td>
<td>Testing For Sensitive Data in Local Data Storage</td>
</tr>
<tr>
<td>2.2</td>
<td>Verify that no sensitive data is written to application logs.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing For Sensitive Data in Logs</td>
</tr>
<tr>
<td>2.3</td>
<td>Verify that no sensitive data is shared with third parties unless it is a necessary part of the architecture.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing Whether Sensitive Data Is Sent To Third Parties</td>
</tr>
<tr>
<td>2.4</td>
<td>Verify that the keyboard cache is disabled on text inputs that process sensitive data.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing Whether the Keyboard Cache Is Disabled for Sensitive Data</td>
</tr>
<tr>
<td>2.5</td>
<td>Verify that the clipboard is deactivated on text fields that may contain sensitive data.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing for Sensitive Data in the Clipboard</td>
</tr>
<tr>
<td>2.6</td>
<td>Verify that no sensitive data is exposed via IPC mechanisms.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing Whether Sensitive Data Is Exposed via IPC Mechanisms</td>
</tr>
<tr>
<td>2.7</td>
<td>Verify that no sensitive data, such as passwords or pins, is exposed through the user interface.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing for Sensitive Data Disclosure Through the User Interface</td>
</tr>
<tr>
<td>2.8</td>
<td>Verify that no sensitive data is included in backups generated by the mobile operating system.</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>Testing Sensitive Data in Backups</td>
</tr>
<tr>
<td>2.9</td>
<td>Verify that the app removes sensitive data from views when backgrounded.</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>Testing Sensitive Information in Auto-Generated Views</td>
</tr>
<tr>
<td>2.10</td>
<td>Verify that the app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>Testing Sensitive Data in Memory</td>
</tr>
<tr>
<td>2.11</td>
<td>Verify that the app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.</td>
<td>✓</td>
<td>✓</td>
<td>N/A</td>
<td>Testing the Device-Access-Security Policy</td>
</tr>
</tbody>
</table>

---

Get from GitHub

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

Or, as they say: "Do a pull request!"
From the Standard to the Guide
OWASP Mobile Security Testing Guide

GitHub Search or clone & grep

MASVS Refs. on each chapter

Android Network APIs

References

OWASP Mobile Top 10 2016

OWASP MASVS
- V5.3: "The app verifies the X.509 certificate before connection is established. Only certificates signed by a trusted CA are accepted."
- V5.4: "The app either uses its own certificate or requests a certificate from a certificate server. The certificate server subsequently does not establish connectivity if the certificate is not signed by a trusted CA."  
- V5.6: "The app only depends on up-to-date connectivity and security libraries."

$ scp -P 2222 root@localhost:/tmp/data.tgz
...
### Dumping KeyChain Data
[Keychain-dumper](https://github.com/ptoomey3/Keychain-Dumper/) lets you dump a jailbroken device's KeyChain contents. The easiest way to get the tool is to download the binary from its GitHub repo:

Next, for asymmetric operations, Apple provides [SecKey](https://opensource.apple.com/source/Sec57746.51c3/SecKey.h.auto.html "SecKey"). Apple provides a nice guide in its [Developer Doc](https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/using_keychain_keys_for_encryption) on how to use this.

Source: https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-ios, tested in the "Arcane" library

When you need to store the key, it is recommended to use the Keychain as long as the protection class: KSecAttrAccessibleAlways. Storing keys in any other location, such as the NSUserDefaults, is prone...
3 Vulnerability Analysis
Vulnerability Analysis

**Static Analysis (SAST)**

**Manual Code Review**
- `grep` & line-by-line examination
- **expert** code reviewer proficient in both language and frameworks

**Automatic Code Analysis**
- Speed up the review
- Predefined set of rules or industry best practices
- False positives! A **security professional** must always review the results.
- False negatives! Even worse ...

**Dynamic Analysis (DAST)**

**Testing and evaluation of apps**
- Real-time execution
- Manual
- Automatic

**Examples of checks**
- disclosure of data in transit
- authentication and authorization issues
- server configuration errors.

**Recommendation:** SAST + DAST + security professional
Vulnerability Analysis

Static Analysis

Check the app's source code for logging mechanisms by searching for the following keywords:

- Functions and classes, such as:
  - `android.util.Log`
  - `Log.d` | `Log.e` | `Log.i` | `Log.v` | `Log.w` | `Log` | `Logger`

- Keywords and system output:
  - `System.out.print` | `System.err.print`
  - `logfile` | `logging` | `logs`

While preparing the production release, you can use tools delete logging-related code. To determine whether all the have been removed, check the ProGuard configuration file:

```
-prepareinsideffects class android.util.Log
```

Dynamic Analysis

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

Many application developers still use `System.out.println` or `printStackTrace` instead of a proper logging class. Therefore, your testing strategy must include all output generated while the application is starting, running and closing. To determine what data is directly printed by `System.out.println` or `printStackTrace`, you can use `Logcat`. There are two ways to execute Logcat:

- `Logcat` is part of `Dalvik Debug Monitor Server (DDMS)` and Android Studio. If the app is running in debug mode, the log output will be shown in the Android Monitor on the Logcat tab. You can filter the app's log output by defining patterns in Logcat.

* OWASP, Mobile Security Testing Guide, 2018 (0x05d-Testing-Data-Storage.html)
Vulnerability Analysis
Demo App

The MSTG Hacking Playground App

Open on GitHub
Vulnerability Analysis

Manual Code Review

Example: Android original source code

```java
public void decryptString() {
    // BTW: Really bad idea, as this is the raw private key. Should be stored in the keystore
    String rawKeys = "4zInk+d4j1Q3m1B1ELctXg==:4aZtzwbniebvM7yC4/GIa2ZmJpSzqrAFtVkJ1Rm+Q4=";
    AesCbcWithIntegrity.SecretKeys privateKey = null;
    try {
        privateKey = AesCbcWithIntegrity.keys(rawKeys);
    }
    catch (InvalidKeyException e) {
        e.printStackTrace();
    }

    String cipherTextString = "6WpfZkgKMJsPhHNhWoSpVg==:6/TgUCXrAyAa21UMPWhx8hHOWwjWEHFp3VIsz3Ws37ZU==:C0mWyNQjcf6n7eBSFZzC";
    AesCbcWithIntegrity.CipherTextIvMac cipherTextIvMac = new AesCbcWithIntegrity.CipherTextIvMac(cipherTextString);
    try {
        plainText = AesCbcWithIntegrity.decryptString(cipherTextIvMac, privateKey);
    }
    catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
```
Vulnerability Analysis
Manual Code Review

Example: Android *decompiled* source code

```java
/*
 * public void decryptString()
 * @param string2
 * @param aesCbcWithIntegrity$SecretKeys
 * @param this.plainText
 * @return
 */
public void decryptString() {
    String string2 = "4zInk+d4jlQ3m1B1ELctxg=:4aZtzwpbniebvM7yC4/GIa2ZmJpSzqrAftVkJ91Rm+Q4=";
    AesCbcWithIntegrity$SecretKeys aesCbcWithIntegrity$SecretKeys = null;
    try {
        aesCbcWithIntegrity$SecretKeys = AesCbcWithIntegrity$SecretKeys.keys(string2);
    } catch (InvalidKeyException invalidKeyException) {
        invalidKeyException.printStackTrace();
    }
    String string3 = "6WpfZkgKMJsPhHNhWoSpVg=:6/TgUCXrAa2lUMPWhx8hHOWjWEHFp3VIsz3Ws37ZU=:C0mWyNQjcfv1MpmLxJXhXgNvAODjE=";
    try {
        AesCbcWithIntegrity$CipherTextIvMac aesCbcWithIntegrity$CipherTextIvMac = new AesCbcWithIntegrity$CipherTextIvMac();
        String string4;
        this.plainText = string4 = AesCbcWithIntegrity$CipherTextIvMac.decryptString(aesCbcWithIntegrity$CipherTextIvMac);
        return;
    }
}
```

If the goal of obfuscation is to protect sensitive computations, an obfuscation scheme is used that is both appropriate for the particular task and robust against manual and automated de-obfuscation methods, considering currently published research. The effectiveness of the obfuscation scheme must be verified through manual testing. Note that hardware-based isolation features are preferred over obfuscation whenever possible.
Example: iOS original source code

```
- (void)storeCredentialsInKeychain {
    NSMutableDictionary *storeCredentials = [NSMutableDictionary dictionary];

    // Prepare keychain dict for storing credentials.
    [storeCredentials setObject:(id)CFBridgingRelease(kSecClassGenericPassword) forKey:(id)CFBridgingRelease(kSecClass)];

    // Store password encoded.
    [storeCredentials setObject:[self.password.text dataUsingEncoding:NSUTF8StringEncoding] forKey:(id)CFBridgingRelease(kSecValueData)];
    [storeCredentials setObject:self.username.text forKey:(id)CFBridgingRelease(kSecAttrAccount)];

    // Access keychain data for this app, only when unlocked. Imp to have this while
    // adding as well as updating keychain item. This is the default, but best practice
    // to specify if apple changes its API.
    [storeCredentials setObject:(id)CFBridgingRelease(kSecAttrAccessibleWhenUnlocked) forKey:(id)CFBridgingRelease(kSecAttrAccessible)];

    // Query Keychain to see if credentials exist.
    OSStatus results = SecItemCopyMatching((CFDictionaryRef) CFBridgingRetain(storeCredentials), nil);

    // If username exists in keychain...
    if (results == errSecSuccess) {
        // NSDictionary *dataFromKeyChain = NULL;
        CFDataRef dataFromKeyChain;

        // There will always be one matching entry, thus limit resultSet size to 1.
        [storeCredentials setObject:(id)CFBridgingRelease(kSecValueData) forKey:(id)CFBridgingRelease(kSecAttrAccount)];
    }
```

*OWASP iGoat A Learning Tool for iOS App Pentesting and Security, 2018 (iGoat)*
Vulnerability Analysis

Manual Code Review

Example: iOS disassembled “source code”
Vulnerability Analysis
Automatic Code Analysis

Example: Static Analyzer

+ SSL Connection Checking
URLs that are NOT under SSL (Total: 1):
http://xmlpull.org/v1/doc/features.html#process-namespaces
=> Lcom/mwr/example/sieve/DBParser;->getPIN(Ljava/io/InputStream;)Ljava/lang/String;
=> Lcom/mwr/example/sieve/DBParser;->getKey(Ljava/io/InputStream;)Ljava/lang/String;
=> Lcom/mwr/example/sieve/DBParser;->readfile(Ljava/io/InputStream;)Ljava/util/List;

+ SSL Certificate Verification Checking
This app DOES NOT check the validation of SSL Certificate. It allows self-signed, expired or mismatch CN certificates for SSL connection.
This is a critical vulnerability and allows attackers to do MITM attacks without your knowledge.
If you are transmitting users’ username or password, these sensitive information may be leaking.
Reference:
(1) OWASP Mobile Top 10 doc: https://www.owasp.org/index.php/Mobile_Top_10_2014-M3
(3) https://www.securecoding.cert.org/confluence/pages/viewPage.action?pageId=134807561
This vulnerability is much more severe than Apple’s “goto fail” vulnerability: http://goo.gl/eFloW
Please do not try to create a "X509Certificate" and override "checkClientTrusted", "checkServerTrusted", and "getAcceptedIssuers"
functions with blank implementation.
We strongly suggest you use the existing API instead of creating your own X509Certificate class.
Please modify or remove these vulnerable code:
[Confirm Vulnerable]

must be always evaluated by a professional
4 Information Gathering
Information Gathering

Information Gathering

Identifies

- General Information
- Sensitive Information

... on the target that is publically available. E.g. about the OS and its APIs

Evaluates the risk by understanding

- Existing Vulnerabilities
- Existing Exploits

... especially from third party software.
Information Gathering

Android Platform Overview

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

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

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

Android Security Architecture

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

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

Example: Open `OMTG_DATAST_011_Memory.java` and observe the `decryptString` implementation.

```java
public void decryptString() {
    // BTW: Really bad idea, as this is the raw private key. Should be stored in the keystore
    String rawKeys = "4zInk+d4jQ3m18Elctxg==:4aZtwpbniebvM7yC4/GIa2ZmJpSzqrAFtVvK91Rm+Q4=";
    AesCbcWithIntegrity.SecretKeys privateKey = null;
    try {
        privateKey = AesCbcWithIntegrity.keys(rawKeys);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    }

    String cipherTextString = "6WpfZkgKM3sPhHnWrSpVg==:6/TgUCXrAuAA21UMPWhx8hHOwjlWEFP3ViSz3Ws37ZU=:C0mbwYHqycf6n7eB5ZmkXqxdu55CjU01c5qfw02
    AesCbcWithIntegrity.CipherTextIvMac cipherTextIvMac = new AesCbcWithIntegrity.CipherTextIvMac(cipherTextString);
    try {
        plainText = AesCbcWithIntegrity.decryptString(cipherTextIvMac, privateKey);
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
```
Information Gathering

Let me google that for you...

How to include in project?

Copy and paste

It's a single very simple Java class, AesCbcWithIntegrity.java that works across most or all versions of Android. The class should be easy to paste into an existing codebase.
Information Gathering

Got all original crypto code inclusive crypto params.
5 Penetration Testing
Penetration Testing

**Preparation**

Coordination with the client

- Define **scope** / focus
- Request source code
- Release and debug apps
- Understand **customer worries**

Identifying Sensitive Data

- at rest: **file**
- in use: **address space**
- in transit: **tx to endpoint, IPC**

**Intelligence Gathering**

Environmental info

- Goals and **intended use** (e.g. Flashlight)
- What if compromised?

Architectural Info

- Runtime protections (jailbreak, emulator..?)
- Which OS (old versions?)
- Network Security
- Secure Storage (what, why, how?)
Penetration Testing

Mapping

Based on all previous information

- UNDERSTAND the target
- LIST potential vulnerabilities
- DRAW sensitive data flow
- DESIGN a test plan, use MASVS

Complement with automated scanning and manually exploring the app

Exploitation

- Exploit the vulnerabilities identified during the previous phase
- Use the MSTG
- Find the true positives

Reporting

- Essential to the client
- Not so fun?
- It makes you the bad guy
- Security not integrated early enough in the SDLC?
Penetration Testing (a.k.a. Pentesting)

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

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

- **Intelligence Gathering** - analyzing the environmental and architectural context of the app to gain a general contextual understanding.

- **Mapping the Application** - based on information from the previous phases; may be complemented by automated scanning and manually exploring the app. Mapping provides a thorough understanding of the app, its entry points, the data it holds, and the main potential vulnerabilities. These vulnerabilities can then be ranked according to the damage their exploitation would cause so that the security tester can prioritize them. This phase includes the creation of test cases that may be used during test execution.

- **Exploitation** - in this phase, the security tester tries to penetrate the app by exploiting the vulnerabilities identified during the previous phase. This phase is necessary for determining whether vulnerabilities are real (i.e., true positives).

- **Reporting** - in this phase, which is essential to the client, the security tester reports the vulnerabilities he or she has been able to exploit and documents the kind of compromise he or she has been able to perform, including the compromise's scope (for example, the data he or she has been able to access illegitimately).

Penetration Testing

Penetration Testing is conducted in four phases*

Penetration Testing

However

- Multiple attack vectors
- Multiple steps
- Different combinations give different full attack vectors

So penetration testing usually looks more like this ...
Penetration Testing

Demo Spoiler

Download the app
unpack it
It’s android, be happy!
Dex to jar
decompile
Inspect the code
What do you want?

The plain text?

Patch smali
get smali
Make the app debuggable
Find stuff: keys, cipherText, classes

Replicate crypto operations in java
Re-package
Re-sign
Re-install
debug
Re-install
logcat
Read the logs
javac
run

The plain text
Penetration Testing

Techniques

decomposition
fuzzing
traffic interception
tampering
disassembly
hooking

dynamic binary instrumentation

traffic dump
root detection
man-in-the-middle

debugging
binary patching
Penetration Testing

ANDROID TESTING GUIDE
- Platform Overview
- Setting up a Testing Environment for Android Apps
- Testing Data Storage on Android
- Android Cryptographic APIs
- Local Authentication on Android
- Android Network APIs
- Android Platform APIs
- Code Quality and Build Settings for Android Apps
- Tampering and Reverse Engineering on Android
- Android Anti-Reversing Defenses

IOS TESTING GUIDE
- Platform Overview
- Setting up a Testing Environment for iOS Apps
- Data Storage on iOS
- iOS Cryptographic APIs
- Local Authentication on iOS
- iOS Network APIs
- iOS Platform APIs
- Code Quality and Build Settings for iOS Apps
- Tampering and Reverse Engineering on iOS
- iOS Anti-Reversing Defenses

All techniques and testing methods categorized

One for Android, one for iOS. All happy 😊
Penetration Testing

Reverse Engineering

Reverse engineering is the process of taking an app apart to find out how it works. You can do this by examining the compiled app (static analysis), observing the app during run time (dynamic analysis), or a combination of both.

Statically Analyzing Java Code

Java bytecode can be converted back into source code without many problems unless some nasty, tool-breaking anti-decompilation tricks have been applied. We'll be using UnCrackable App for Android Level 1 in the following examples, so download it if you haven't already. First, let's install the app on a device or emulator and run it to see what the cracker is about.

```bash
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Crackmes/Android/Level_01/UnCrackable-Level11.apk
$ adb install UnCrackable-Level11.apk
```

Penetration Testing

Tampering and Runtime Instrumentation

First, we'll look at some simple ways to modify and instrument mobile apps. Tampering means making patches or run-time changes to the app to affect its behavior. For example, you may want to deactivate SSL pinning or binary protections that hinder the testing process. Runtime Instrumentation encompasses adding hooks and runtime patches to observe the app's behavior. In mobile app-sec however, the term loosely refers to all kinds of run-time manipulation, including overriding methods to change behavior.

Patching and Re-Packaging

Making small changes to the app Manifest or bytecode is often the quickest way to fix small annoyances that prevent you from testing or reverse engineering an app. On Android, two issues in particular happen regularly:

1. You can't attach a debugger to the app because the android:debuggable flag is not set to true in the Manifest.
2. You can't intercept HTTPS traffic with a proxy because the app employs SSL pinning.

In most cases, both issues can be fixed by making minor changes to the app and then re-signing and re-packaging it. Apps that run additional integrity checks beyond default Android code-signing are an exception—in these cases, you have to patch the additional checks as well.

Example: Disabling Certificate Pinning

Certificate pinning is an issue for security testers who want to intercept HTTPS communication for

Penetration Testing
Example Scenario Automotive-Mobile Testing

Frida 12.2.25 - A world-class dynamic instrumentation toolkit
Commands:
  help   -> Displays the help system
  object? -> Display information about 'object'
  exit/quit -> Exit

More info at http://frida.re/docs/home/

CAN
Bluetooth
Mobile Apps
6 Demo 1 Mobile Penetration Testing

Let’s decrypt that encrypted string!
Demo 1

App: MSTG-Hacking-Playground (011_MEMORY)

```java
public void decryptString() {
    // BTW: Really bad idea, as this is the raw private key. Should be stored in the keystore
    String rawKeys = "4zInk+d4j1Q3m181Elctxg==:4aza7twbpniiebvM7yC4/GIaZmJpSzqrAFTvK91rm+Q4=";
    AesCbcWithIntegrity.SecretKeys privateKey = null;
    try {
       privateKey = AesCbcWithIntegrity.keys(rawKeys);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
    }
}

String cipherTextString = "6wpfZkgKMJoPhNwSpVg==6/TgUCXrAuAa21UMPhx8HjHOyjWEHFP3Vlsz3W3s37ZU:=-C0mbywQyjcf6n7eB5FzmkXqxdu55CjUO1c5qFW02"

AesCbcWithIntegrity.CipherTextIvMac cipherTextIvMac = new AesCbcWithIntegrity.CipherTextIvMac(cipherTextString);
try {
    plainText = AesCbcWithIntegrity.decryptString(cipherTextIvMac, privateKey);
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}
```

We have the keys and the ciphertext.

But the plaintext remains inside this variable. And now?
What do you want?

- Download the app
- Dex to jar
- decompile
- Inspect the code

It’s android, be happy!

Patch smali
- get smali
- Make the app debuggable
- Find stuff: keys, cipherText, classes

Replicate crypto operations in java
- Repackage
- Re-sign
- Re-install
- debug
- Re-package
- Re-sign
- Re-install
- run
- javac
- logcat
- Read the logs

The plain text?
- hooking

The plain text ✓
- The plain text?
What do you want?

Download the app
unpack it
It’s android, be happy!
Dex to jar
decompile
Inspect the code
Find stuff: keys, cipherText, classes
google

The plain text?

hooking
The plain text ✓
```javascript
Java.perform(function(){
    console.log("\n[*] script loaded. Open OMTG_DATAST_011_MEMORY\n\n");
    var clazz = Java.use("com.tozny.crypto.android.AesCbcWithIntegrity");

    clazz.decryptString.overload('com.tozny.crypto.android.AesCbcWithIntegrity$CipherTextIVMac', 'com.tozny.crypto.android.AesCbcWithIntegrity$CipherTextIVMac').implementation = function (cipherText, privateKey) {
        console.log("\n[*] decryptString called");
        console.log("\n[*] cipherText: " + cipherText);
        console.log("\n[*] privateKey: " + privateKey);

        var ret = this.decryptString.overload('com.tozny.crypto.android.AesCbcWithIntegrity$CipherTextIVMac', 'com.tozny.crypto.android.AesCbcWithIntegrity$CipherTextIVMac').call(this, cipherText, privateKey);
        console.log("\n[*] plainText: ' + ret);
        return ret;
    }
});
```
Demo 1

Frida 12.2.25 - A world-class dynamic instrumentation toolkit

Commands:
- help: Displays the help system
- object: Displays information about 'object'
- exit/quit: Exit

More info at http://www.frida.re/docs/home/

Attaching...

[*] script loaded. Open OMTG_DATAST_011_MEMORY

[Android Emulator 5554::sg.vp.owasp_mobile.owt.android]->

[*] decryptString called

[*] cipherText: 6fmp2zkgrW13sPhN/nhioSpVg==
[*] privateKey: 4zInk+4j3Q3d81ELctsG==
[*] plainText: You got the decrypted message. Well done.

[Android Emulator 5554::sg.vp.owasp_mobile.owt.android]>

Thank you for using Frida!

~/owasp $
6 Demo 2 Mobile Penetration Testing

Let’s get the crypto keys!
Demo 2

App: MSTG-Hacking-Playground (001_KEYSTORE)

Extraction prevention

Key material of Android Keystore keys is protected from extraction using two security measures:

- **Key material never enters the application process.** When an application performs cryptographic operations using an Android Keystore key, behind the scenes plaintext, ciphertext, and messages to be signed or verified are fed to a system process which carries out the cryptographic operations. If the app’s process is compromised, the attacker may be able to use the app’s keys but will not be able to extract their key material (for example, to be used outside of the Android device).

- **Key material may be bound to the secure hardware (e.g., Trusted Execution Environment (TEE), Secure Element (SE)) of the Android device.** When this feature is enabled for a key, its key material is never exposed outside of secure hardware. If the Android OS is compromised or an attacker can read the device’s internal storage, the attacker may be able to use any app’s Android Keystore keys on the Android device, but not extract them from the device. This feature is enabled only if the device’s secure hardware supports the particular combination of key algorithm, block modes, padding schemes, and digests with which the key is authorized to be used. To check whether the feature is enabled for a key, obtain a `KeyInfo` for the key and inspect the return value of `KeyInfo.isInsideSecurityHardware()`.
Demo 2

Options:
- Download the app
- Dex to jar
- decompile
- Inspect the code
- What do you want?

Steps:
- unpack it
- get smali
- Patch smali
- Re-package
- Re-sign
- Re-install
- debug
- hooking
- The crypto keys

Additional:
- It's android, be happy!
- Make the app debuggable
- Find stuff: keys, classes
- google
Demo 2

- Download the app
- unpack it
- It’s android, be happy!
- Dex to jar
- decompile
- Inspect the code
- Find stuff: keys, classes
- What do you want?

The crypto keys

hooking

The crypto keys ✓
```javascript
console.log("\n[*] script loaded. Open OMTG_DATAST_001_KEYSTORE\n\n");
var clazz = Java.use("sg.vp.owasp_mobile.OMTG_Android.OMTG_DATAST_001_KeyStore");

clazz.decryptString.overload("java.lang.String").implementation = function (alias) {
    console.log("\n[*] decryptString called");
    console.log("\n[*] alias: " + alias);

    this.decryptString.overload("java.lang.String").call(this, alias);
};

var RSAPublicKey = Java.use("java.security.interfaces.RSAPublicKey");
var RSAKey = Java.use("java.security.interfaces.RSAKey");
var RSAPrivateKey = Java.use("java.security.interfaces.RSAPrivateKey");

var OpenSSLRSAPrivateKey = Java.use("com.android.org.conscrypt.OpenSSLRSAKey");
var OpenSSLKey = Java.use("com.android.org.conscrypt.OpenSSLKey");

OpenSSLKey.isEngineBased.overload().implementation = function(){
    console.log("\n[*] OpenSSLKey.isEngineBased called");
    return false;
}

var NativeCrypto = Java.use("com.android.org.conscrypt.NativeCrypto");

var Cipher = Java.use("javax.crypto.Cipher");
Cipher.init.overload('int', 'java.security.Key').implementation = function(opmode, key){
    console.log("\n[*] Cipher.init called");
    console.log("\n[*] mode: " + opmode);

    if (opmode == 2)[
        console.log("\n\n[*] decryption with private key!");
    }
};
```

Demo 2

```
[*] script loaded. Open OMTG_DATAST_001_KEYSTORE

[Android Emulator 5554::sg.vp.ovasp_mobile.omtg.android] =>
[*] Cipher.init called
[*] mode: 1
[*] encryption with public key!
[*] key: OpenSSLRSAPublicKey(modulus=a8c24af6e16ed7a7d8386152c46cf793eebcb8ad73d2cf4e817b8aee0df5c5b364af7e6848c172ce2f7106c8351
d579efedc8135f39b6b0b0723cf6f8f7f7e2ee87f0f1a8721abc3f5e52f8e17f30f02b3571f177325f0c9f00f06dc23ab7e88541d413d907b2833791
d10000062c02b56c4b0f8b6388d385e3a12eedced3e6a81781e7863ad6f52b6df43a2b695d65a3e89d29c310335125c55ec0ce9a6596d1e24f
d8ea24d6b95a4c9a7a1ab72b92842a187e6f1f468c7534ba0c4ca6f6b84dbd1a578d738c77c101cb4add41e0e0a803555f0b8bd1d0c4a1938b33dcf142c6120f723, publicExponent: 65537
[*] key PublicExponent: 65537
[*] key modulus: 28238441178569597924049031695659430280030061533595319448200565515561676277023204878153438882218886953658807578372109824136319602512069286943634620432160575970786193800924224451163139013980715077117530844874823931
[4] 25759607898249048131987058178057711754710138023022333390148732051502473582757065475721304672363212306745935911038200322333901487320515024735827570654757213046723632123067459359110382
[*] decryptString called
[*] alias: Dummy
[*] OpenSSLKey.isEngineBased called
[*] Cipher.init called
[*] mode: 2
[*] decryption with private key!
[*] OpenSSLKey.isEngineBased called
[*] Private Key encoded: 30 82 01 02 01 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00 01 02 03 00 00 00 00
[*] OpenSSLKey.isEngineBased called
[*] OpenSSLKey.isEngineBased called
[*] Exception in priv_key_getPrivateExponent(): java.lang.NullPointerException: privateExponent == null
```

```
OMTG_DATAST_001_KeyStore

nkylI1qVauD6vojvcbo2BSRMsfAYSYV22Abiude/op2zk59ngq+Efo319aoag/7qfsXkcmNug/nbfFlM4eQ40Uigk0HznXPryZBNWQpNdcO14xkMPmPfHMrnaTRxbgrArHiPVsQDVwVvQ7b9AGTTYc3KhLHu8PbmboT07Qytphdu0ily4T8vYgWYRgx1XRsQ==

hola

```

```
ENCRIPT

q w e r t y u i o p
a s d f g h j k l
```

```
DECRIPT

[hold]
[...]
[hole]

```

```
?123 , .
```

```
```
**ASN.1 JavaScript decoder**

```plaintext
SEQUENCE (3 elem)
  INTEGER 0
  SEQUENCE (2 elem)
    OBJECT IDENTIFIER 1.2.840.113549.1.1.1 rsaEncryption (PKCS #1)
    NULL
  OCTET STRING (1 elem)
    SEQUENCE (3 elem)
      INTEGER 0
      INTEGER (2048 bit) 28228411879695972905460913761596582048200381
      INTEGER 65537
```

**7.2 Private-key syntax**

An RSA private key shall have ASN.1 type RSAPrivateKey:

```
RSAPrivateKey ::= SEQUENCE {
  version Version,
  modulus INTEGER, -- n
  publicExponent INTEGER, -- e
  privateExponent INTEGER, -- d
  prime1 INTEGER, -- p
  prime2 INTEGER, -- q
  exponent1 INTEGER, -- d mod (p-1)
  exponent2 INTEGER, -- d mod (q-1)
  coefficient INTEGER -- (inverse of q) mod p
}
```
Takeaways

- Read the MSTG
- Use the MASVS
- Play with Crackmes
- `grep` harder
- Learn FRIDA
- Learn 🤖🤖🤖
- Contribute!
- Have fun :)
References

RTFM

STG
References

- OWASP Mobile Security Testing Guide
  https://mobile-security.gitbook.io/mobile-security-testing-guide
  https://github.com/OWASP/owasp-mstg

- OWASP Mobile Application Security Verification Standard
  https://mobile-security.gitbook.io/masvs/
  https://github.com/OWASP/owasp-masvs

- OWASP iGoat - A Learning Tool for iOS App Pentesting and Security
  https://github.com/OWASP/igoat

- OWASP MSTG-Hacking-Playground Android App
  https://github.com/OWASP/MSTG-Hacking-Playground

- OWASP MSTG Crackmes
  https://github.com/OWASP/owasp-mstg/tree/master/Crackmes
Thank you, any questions?