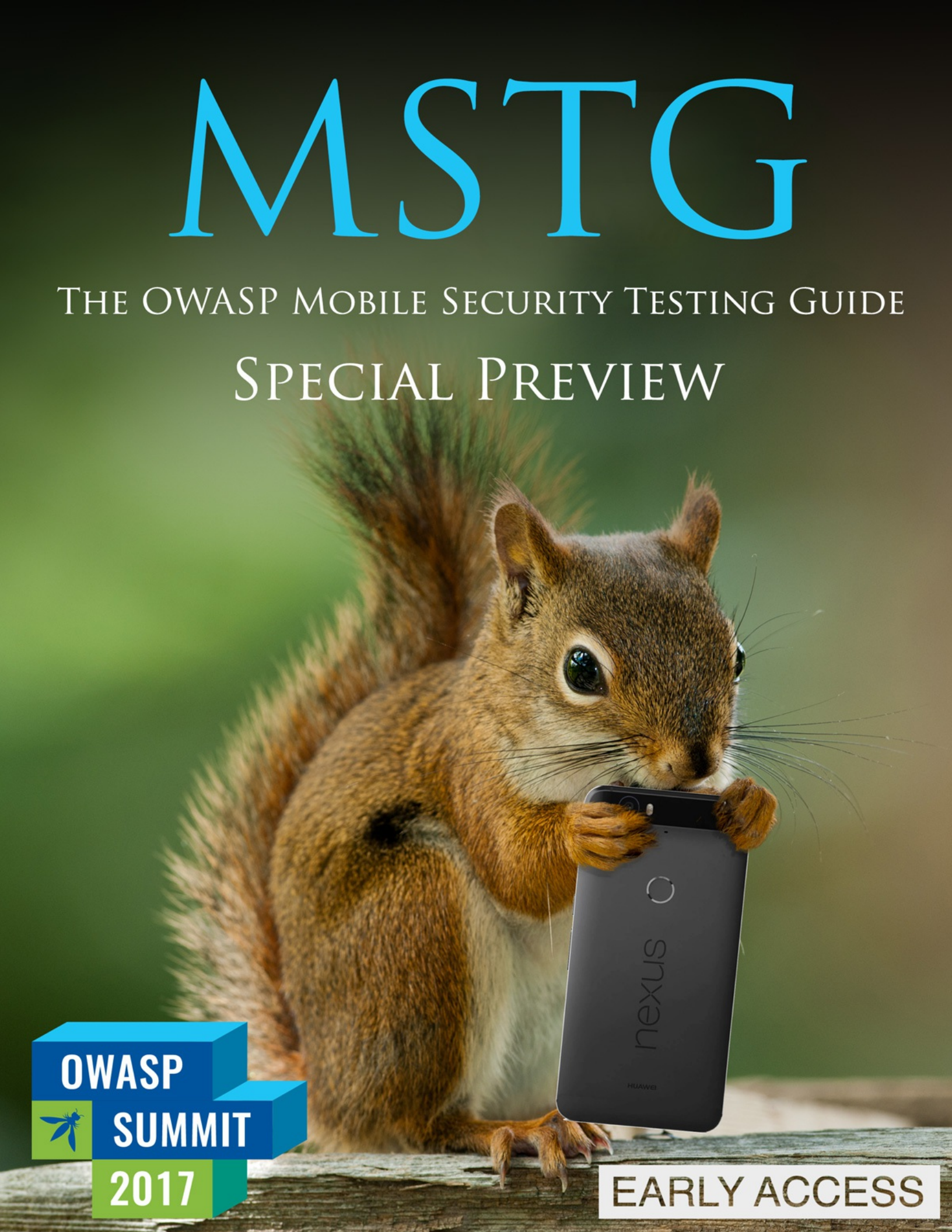
# MSTG

## The OWASP Mobile Security Testing Guide

## Special Preview




OWASP SUMMIT 2017

EARLY ACCESS

# Table of Contents

# Overview

# Android Security Testing - Sample Content

# Android Reverse Engineering - Sample Content

# Foreword

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

---

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

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

With the MSTG, we aim to create best practices for mobile security, along with a comprehensive set of security test cases to verify them. The best practices and test cases are packaged into beginner friendly, complete and practical guide to mobile app security testing and reverse engineering.

This is an early preview edition of the MSTG that contains sample chapters made out of the Android content in our GitHub repository. We made it so our OWASP Summit working would have a tangible result (and because OWASP asked us to). The final version of the guide, which will cover a wide range of Android, iOS, and OS-independent topics, is scheduled for release in the first quarter of 2018.

Our wholehearted thanks go to everyone who contributed to this project. We'd also like to thank the OWASP Foundation for bringing all of us together, and organizing and sponsoring this fantastic event.


OWASP
SUMMIT
2017
FURLAN!

# Frontispiece

# About the MSTG Summit Edition

The OWASP Mobile Security Testing Guide (MSTG) is a comprehensive manual for testing the security of mobile apps. It describes technical processes for verifying the controls listed in the OWASP Mobile Application Security Verification Standard (MASVS). The MSTG is meant to provide a baseline set of test cases for static and dynamic security tests, and to help ensure completeness and consistency of the tests.

The MSTG Summit Edition is an experimental proof-of-concept book created on the OWASP Summit 2017 in London. The goal was to improve the agile authoring process book deployment pipeline, as well as to demonstrate the viability of the project. Note that the content is not final and will likely change significantly in subsequent releases.

OWASP thanks the many authors, reviewers, and editors for their hard work in developing this guide. If you have any comments or suggestions on the Mobile Security Testing Guide, please join the discussion around MASVS and MSTG in the OWASP Mobile Security Project Slack Channel https://owasp.slack.com/messages/project-mobile_omtg/details/. You can sign up here:

http://owasp.herokuapp.com/

# Copyright and License

# Acknowledgments

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

## Authors

## Bernhard Mueller

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

## Sven Schleier

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

## Co-Authors

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

## Romuald Szularek

Romuald is a passionate cyber security & privacy professional with a unique blend of experiences over 15+ years in the Web, Mobile, IoT and Cloud domains. During his career, he has been dedicating spare time a variery of projects with the goal of advancing the sectors of software and security. He is also a teacher at the Nice Technological Institute (France).

## Jeroen Willemsen

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

# Top Contributors

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

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

# Contributors

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

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

## Reviewers

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

- Anant Shrivastava
- Sjoerd Langkemper

## Others

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

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

## Older Versions

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

**OWASP MSTG "Beta 2" (Google Doc)**

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

**OWASP MSTG "Beta 1" (Google Doc)**

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

# Introduction to the OWASP Mobile Security Testing Guide

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

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

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

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

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

# Key Areas in Mobile AppSec

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

## Local Data Storage

The protection of sensitive data, such as user credentials and private information, is a key focus in mobile security. Firstly, sensitive data can be unintentionally exposed to other apps running on the same device if operating system mechanisms like IPC are used improperly. Data may also unintentionally leak to cloud storage, backups, or the keyboard cache. Additionally, mobile devices can be lost or stolen more easily compared to other types of devices, so an adversary gaining physical access is a more likely scenario.

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

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

## Communication with Trusted Endpoints

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

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

## Authentication and Authorization

In most cases, user login to a remote service is an integral part of the overall mobile app architecture. Even though most of the the authentication and authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. In contrast to web apps, mobile apps often store long-time session tokens that are are then unlocked via user-to-device authentication features such as fingerprint scan. While this allows for a better user experience (nobody likes to enter a complex password every time they start an app), it also introduces additional complexity and the concrete implementation has a lot of room for errors.

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

## Interaction with the Mobile Platform

Android offers rich inter-process communication (IPC) facilities that 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.

Broadcast receivers are components that allow to receive notifications sent from other apps and from the system itself. 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.

From the perspective of mobile apps, it is important to make sure that IPC facilities don't expose sensitive data or functionality.

## Code Quality and Exploit Mitigation

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

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

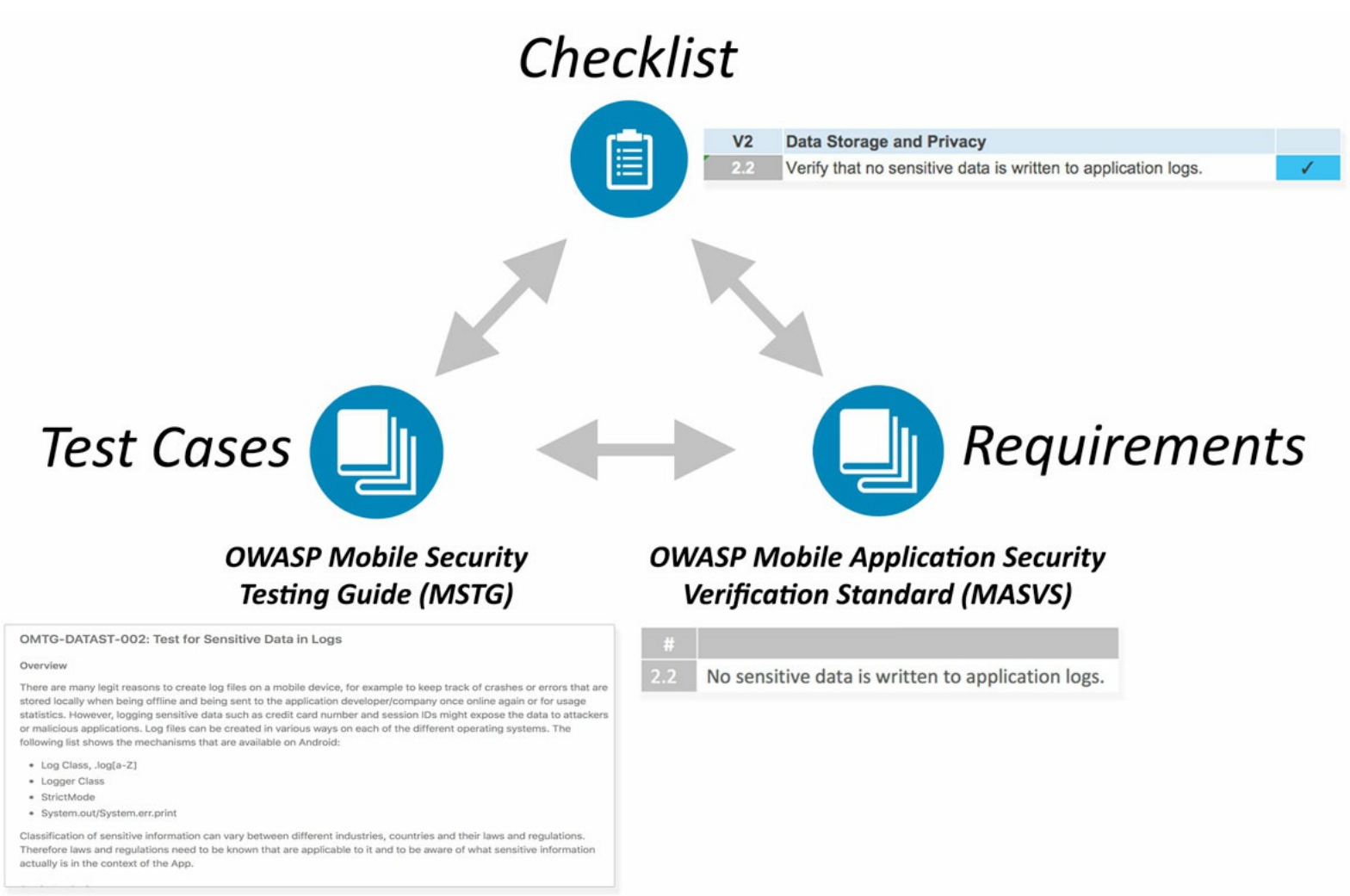
## Anti-Tampering and Anti-Reversing

There is three things you should never bring up in date conversations: Religion, politics and code obfuscation. Many security experts dismiss client-side protections outright. However, the fast is that software protection controls are widely used in the mobile app world, so security testers need ways to deal with them. We also think that there is *some* benefit to be had, as long as the protections are employed with a clear purpose and realistic expectations in mind, and aren't used to *replace* security controls.

# The OWASP Mobile AppSec Verification Standard, Checklist and Testing Guide

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

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




For example, the MASVS requirements can 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 this Book

This "Sample Bites" book contains sample chapters from the Android security testing and reverse engineering sections of the guide.

The book is organized as follows:

- The first three chapters, "Testing Local Data Authentication in Android apps", "Testing Data Storage on Android" and "Testing Cryptography in Android apps", contain sample test cases for important areas of Android appsec. This content
- The "Android Reverse Engineering Guide" chapter contains a complete guide to reverse engineering Android apps.
- The "Anti-Reversing Defenses on Android" chapter gives an overview of common software protection mechanisms used on Android and describes bypass techniques.

# Testing Local Authentication in Android Apps

Even though most of the the authentication and authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. The mobile security testing team is working on best practices and test cases for both remote and local authentication, and various possible combinations. This chapter contains a sample test case that deals with fingerprint authentication on Android.

## Testing Biometric Authentication

## Overview

Android 6.0 introduced public APIs for authenticating users via fingerprint. Access to the fingerprint hardware is provided through the `FingerprintManager` class [1]. An app can request fingerprint authentication by instantiating a `FingerprintManager` object and calling its `authenticate()` method. The caller registers callback methods to handle possible outcomes of the authentication process (success, failure or error).

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

Apart from this relatively reasonable method, fingerprint authentication can also be implemented in unsafe ways. For instance, developers might opt to assume successful authentication based solely on whether the `onAuthenticationSucceeded` callback [3] is called or when the Samsung Pass SDK is used for instance. This event however isn't proof that the user has performed biometric authentication - such a check can be easily patched or bypassed using instrumentation. Leveraging the Keystore is the only way to be reasonably sure that the user has actually entered their fingerprint.

Unless, of course, the Keystore is compromised. Which has been the case as reported in [5] and mostly explained in [6]. There are a few known CVEs registered for instance: CVE-2016-2431, CVE-2016-2432, CVE-2015-6639, CVE-2015-6647. Therefore one should always check the security patch-level:

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd",     Locale.getDefault());
sdf.parse(Build.VERSION.SECURITY_PATCH).after(sdf.parse("2016-05-01"));
```

## Static Analysis

First make sure that the actual Android SDK is used for fingerprint evaluation and not any vendor specific SDKs, such as Samsung Pass as it is inherently flawed.

Search for calls of `FingerprintManager.authenticate()` . The first parameter passed to this method should be a `CryptoObject` instance. `CryptoObject` is a wrapper class for the crypto objects supported by FingerprintManager [2]. If this parameter is set to `null` , the fingerprint auth is purely event-bound, which likely causes a security issue.

Trace back the creation of the key used to initialize the cipher wrapped in the CryptoObject. Verify that the key was created using the `KeyGenerator` class, and that `setUserAuthenticationRequired(true)` was called when creating the `KeyGenParameterSpec` object (see also the code samples below).

Verify the authentication logic. For the authentication to be successful, the remote endpoint **must** require the client to present the secret retrieved from the Keystore, or some value derived from the secret.

## Dynamic Analysis

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

## Remediation

Fingerprint authentication should be implemented along the following lines:

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

- The user must have protected their lockscreen

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

- Fingerprinthardware must be available:

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

- At least one finger should be registered:

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

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

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

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

When setting up fingerprint authentication, create a new AES key using the `KeyGenerator` class. Add `setUserAuthenticationRequired(true)` in `KeyGenParameterSpec.Builder` .

```
generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTORE);

generator.init(new KeyGenParameterSpec.Builder (KEY_ALIAS,
      KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
      .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
      .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
      .setUserAuthenticationRequired(true)
      .build()
);

generator.generateKey();
```

Please note, that since Android 7 you can use the `setInvalidatedByBiometricEnrollment(boolean value)` as a method of the builder. If you set this to true, then the fingerprint will not be invalidated when new fingerprints are enrolled. Even though this might provide user-convenience, it opens op a problem area when possible attackers are somehow able to social-engineer their fingerprint in.

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

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

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

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

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

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

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

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

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

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

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

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

For a full example, see the blog article by Deivi Taka [4].

# References

### OWASP Mobile Top 10 2016

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

### OWASP MASVS

- 4.6: "Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns "true" or "false"). Instead, it is based on unlocking the keychain/keystore."

### CWE

- CWE-287 - Improper Authentication
- CWE-604 - Use of Client-Side Authentication

**Info**

- [1] FingerprintManager - https://developer.android.com/reference/android/hardware/fingerprint/FingerprintManager.html
- [2] FingerprintManager.CryptoObject - https://developer.android.com/reference/android/hardware/fingerprint/FingerprintManager.CryptoObject.html
- [3] https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.Builder.html#setUserAuthenticationRequired(boolean)
- [4] Securing Your Android Appps with the Fingerprint API - https://www.sitepoint.com/securing-your-android-apps-with-the-fingerprint-api/#savingcredentials
- [5] Android Security Bulletins - https://source.android.com/security/bulletin/
- [6] Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption - http://bits-please.blogspot.co.uk/2016/06/extracting-qualcomms-keymaster-keys.html

# Testing Cryptography in Android Apps

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

- Encryption ensures data confidentiality by using special algorithms to convert the plaintext data into cipher text, which does not reveal any information about the original contents. The plaintext data can be restored from the cipher text through decryption. Two main forms of encryption are symmetric (or secret key) and asymmetric (or public key). In general, encryption operations do not protect integrity, but some symmetric encryption modes also feature that protection (see "Testing Sensitive Data Protection" section).
    - Symmetric-key encryption algorithms use the same key for both encryption and decryption. It is fast and suitable for bulk data processing. Since everybody who has access to the key is able to decrypt the encrypted content, they require careful key management.
    - Public-key (or asymmetric) encryption algorithms operate with two separate keys: the public key and the private key. The public key can be distributed freely, while the private key should not be shared with anyone. A message encrypted with the public key can only be decrypted with the private key. Since asymmetric encryption is several times slower than symmetric operations, it is typically only used to encrypt small amounts of data, such as symmetric keys for bulk encryption.
- Hash functions deterministically map arbitrary pieces of data into fixed-length values. It is typically easy to compute the hash, but difficult (or impossible) to determine the original input based on the hash. Cryptographic hash functions additionally guarantee that even small changes to the input data result in large changes to the resulting hash values. Cryptographic hash functions are used for integrity verification, but do not provide authenticity guarantees.
- Message Authentication Codes, or MACs, combine other cryptographic mechanism, such as symmetric encryption or hashes, with secret keys to provide both integrity and authenticity protection. However, in order to verify a MAC, multiple entities have to share the same secret

key, and any of those entities will be able to generate a valid MAC. The most commonly used type of MAC, called HMAC, relies on hash as the underlying cryptographic primitive. As a rule, full name of an HMAC algorithm also includes the name of the underlying hash, e.g. - HMAC-SHA256.
- Signatures combine asymmetric cryptography (i.e. - using a public/private keypair) with hashing to provide integrity and authenticity by encrypting hash of the message with the private key. However, unlike MACs, signatures also provide non-repudiation property, as the private key should remain unique to the data signer.
- Key Derivation Functions, or KDFs, are often confused with password hashing functions. KDFs do have many useful properties for password hashing, but were created with different purposes in mind. In context of mobile applications, it is the password hashing functions that are typically meant for protecting stored passwords.

Two uses of cryptography are covered in other chapters:

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

## References


- [1] Password Hashing Competition - https://password-hashing.net/


# Testing for Custom Implementations of Cryptography

## Overview

The use of non-standard or custom built cryptographic algorithms is dangerous because a determined attacker may be able to break the algorithm and compromise data that has been protected. Implementing cryptographic functions is time consuming, difficult and very likely to fail. Instead

well-known algorithms that were already proven to be secure should be used. All mature frameworks and libraries offer cryptographic functions that should also be used when implementing mobile apps.

## Static Analysis

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

## Remediation

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

## References

#### OWASP Mobile Top 10 2016

- M6 - Broken Cryptography

#### OWASP MASVS

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

#### CWE

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

#### Info

- [1] Supported Ciphers in KeyStore - https://developer.android.com/training/articles/keystore.html#SupportedCiphers

# Testing for Insecure and/or Deprecated Cryptographic Algorithms

## Overview

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

## Static Analysis

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

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

- DES, 3DES[6]
- RC2
- RC4
- BLOWFISH[6]
- MD4
- MD5
- SHA1 and others.

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

Other uses of cryptography require careful adherence to best practices:

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

# Remediation

Periodically ensure that the cryptography has not become obsolete. Some older algorithms, once thought to require years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once considered as strong. Examples of currently recommended algorithms[1], [2]:

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

384

# References

### OWASP Mobile Top 10

- M6 - Broken Cryptography

### OWASP MASVS

- V3.3: "The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices"
- V3.4: "The app does not use cryptographic protocols or algorithms that are widely considered depreciated for security purposes"

### CWE

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

### Info

- [1] Commercial National Security Algorithm Suite and Quantum Computing FAQ - https://cryptome.org/2016/01/CNSA-Suite-and-Quantum-Computing-FAQ.pdf
- [2] NIST Special Publication 800-57 - http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf
- [4] NIST recommendations (2016) - https://www.keylength.com/en/4/
- [5] BSI recommendations (2017) - https://www.keylength.com/en/8/
- [6] Sweet32 attack -- https://sweet32.info/

### Tools

- QARK - https://github.com/linkedin/qark
- Mobile Security Framework - https://github.com/ajinabraham/Mobile-Security-Framework-MobSF

# Verifying the Configuration of Cryptographic Standard Algorithms

## Overview

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

Android cryptography APIs are based on the Java Cryptography Architecture (JCA). JCA separates the interfaces and implementation, making it possible to include several security providers [8] that can implement sets of cryptographic algorithms. Most of the JCA interfaces and classes are defined in the `java.security.*` and `javax.crypto.*` packages. In addition, there are Android specific packages `android.security.*` and `android.security.keystore.*` .

The list of providers included in Android varies between versions of Android and the OEM-specific builds. Some provider implementations in older versions are now known to be less secure or vulnerable. Thus, Android applications should not only choose the correct algorithms and provide good configuration, in some cases they should also pay attention to the strength of the implementations in the legacy providers. You can list the set of existing providers as follows:

```
StringBuilder builder = new StringBuilder();
for (Provider provider : Security.getProviders()) {
    builder.append("provider: ")
              .append(provider.getName())
            .append(" ")
            .append(provider.getVersion())
            .append("(")
            .append(provider.getInfo())
            .append(")\n");
}
String providers = builder.toString();
//now display the string on the screen or in the logs for debugging.
```

Below you can find the output on the Emulator running Android 4.4 with Google Play APIs after the security provider has been patched:

```
provider: GmsCore_OpenSSL1.0 (Android's OpenSSL-backed security provider)
provider: AndroidOpenSSL1.0 (Android's OpenSSL-backed security provider)
provider: DRLCertFactory1.0 (ASN.1, DER, PkiPath, PKCS7)
provider: BC1.49 (BouncyCastle Security Provider v1.49)
provider: Crypto1.0 (HARMONY (SHA1 digest; SecureRandom; SHA1withDSA signature))
provider: HarmonyJSSE1.0 (Harmony JSSE Provider)
provider: AndroidKeyStore1.0 (Android KeyStore security provider)
```

For some applications that support older versions of Android, bundling an up-to-date library may be the only option. SpongyCastle (a repackaged version of BouncyCastle) is a common choice in these situations. Repackaging is necessary because BouncyCastle is included in the Android SDK. The latest version of SpongyCastle [6] likely fixes issues encountered in the earlier versions of BouncyCastle [7] that were included in Android. Note that the BouncyCastle libraries packed with Android are often not as complete as their counterparts from the Legion of the BounceyCastle. Lastly: bear in mind that packing large libraries such as SpongyCastle will often lead to a multidexed Android application.

Android SDK provides mechanisms for specifying secure key generation and use. Android 6.0 (Marshmallow, API 23) introduced the `KeyGenParameterSpec` class that can be used to ensure the correct key usage in the application.

Here's an example of using AES/CBC/PKCS7Padding on API 23+:

```
String keyAlias = "MySecretKey";

KeyGenParameterSpec keyGenParameterSpec = new KeyGenParameterSpec.Builder(keyAlias,
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
        .setRandomizedEncryptionRequired(true)
        .build();

KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
        "AndroidKeyStore");
keyGenerator.init(keyGenParameterSpec);

SecretKey secretKey = keyGenerator.generateKey();
```

The `KeyGenParameterSpec` indicates that the key can be used for encryption and decryption, but not for other purposes, such as signing or verifying. It further specifies the block mode (CBC), padding (PKCS7), and explicitly specifies that randomized encryption is required (this is the default.) `"AndroidKeyStore"` is the name of the cryptographic service provider used in this example.

GCM is another AES block mode that provides additional security benefits over other, older modes. In addition to being cryptographically more secure, it also provides authentication. When using CBC (and other modes), authentication would need to be performed separately, using HMACs (see the Reverse Engineering chapter). Note that GCM is the only mode of AES that does not support paddings.[3], [5]

Attempting to use the generated key in violation of the above spec would result in a security exception.

Here's an example of using that key to decrypt:

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

// byte[] input
Key key = keyStore.getKey(keyAlias, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
cipher.init(Cipher.ENCRYPT_MODE, key);

byte[] encryptedBytes = cipher.doFinal(input);
byte[] iv = cipher.getIV();
// save both the iv and the encryptedBytes
```

Both the IV and the encrypted bytes need to be stored; otherwise decryption is not possible.

Here's how that cipher text would be decrypted. The `input` is the encrypted byte array and `iv` is the initialization vector from the encryption step:

```
// byte[] input
// byte[] iv
Key key = keyStore.getKey(AES_KEY_ALIAS, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
IvParameterSpec params = new IvParameterSpec(iv);
cipher.init(Cipher.DECRYPT_MODE, key, params);

byte[] result = cipher.doFinal(input);
```

Since the IV (initialization vector) is randomly generated each time, it should be saved along with the cipher text ( `encryptedBytes` ) in order to decrypt it later.

Prior to Android 6.0, AES key generation was not supported. As a result, many implementations chose to use RSA and generated public-private key pair for asymmetric encryption using `KeyPairGeneratorSpec` or used `SecureRandom` to generate AES keys.

Here's an example of `KeyPairGenerator` and `KeyPairGeneratorSpec` used to create the RSA key pair:

```
Date startDate = Calendar.getInstance().getTime();
Calendar endCalendar = Calendar.getInstance();
endCalendar.add(Calendar.YEAR, 1);
Date endDate = endCalendar.getTime();
KeyPairGeneratorSpec keyPairGeneratorSpec = new KeyPairGeneratorSpec.Builder(context)
        .setAlias(RSA_KEY_ALIAS)
        .setKeySize(4096)
        .setSubject(new X500Principal("CN=" + RSA_KEY_ALIAS))
        .setSerialNumber(BigInteger.ONE)
        .setStartDate(startDate)
        .setEndDate(endDate)
        .build();

KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA",
        "AndroidKeyStore");
keyPairGenerator.initialize(keyPairGeneratorSpec);

KeyPair keyPair = keyPairGenerator.generateKeyPair();
```

This sample creates the RSA key pair with the 4096-bit key (i.e., modulus size).

# Static Analysis

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

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

Ensure that the best practices outlined in the Cryptography for Mobile Apps chapter are followed.

# Remediation

Use cryptographic algorithm configurations that are currently considered strong, such those from NIST[1] and BSI[2] recommendations.

## References

### OWASP Mobile Top 10

- M6 - Broken Cryptography

### OWASP MASVS

- V3.3: "The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices"

### CWE

- CWE-326: Inadequate Encryption Strength

### Info

- [1] NIST recommendations (2016) - https://www.keylength.com/en/4/
- [2] BSI recommendations (2017) - https://www.keylength.com/en/8/
- [3] Supported Ciphers in KeyStore - https://developer.android.com/training/articles/keystore.html#SupportedCiphers
- [4] Credential storage enhancements in Android 4.3 (August 21, 2013) - https://nelenkov.blogspot.co.uk/2013/08/credential-storage-enhancements-android-43.html
- [5] Cipher documentation - https://developer.android.com/reference/javax/crypto/Cipher.html
- [6] Spongy Castle - https://rtyley.github.io/spongycastle/
- [7] CVE Details Bouncey Castle - https://www.cvedetails.com/vulnerability-list/vendor_id-7637/Bouncycastle.html
- [8] Provider - https://developer.android.com/reference/java/security/Provider.html

# Testing for Hardcoded Cryptographic Keys

## Overview

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

## Static Analysis

The following checks would be performed against the used source code:

- Ensure that no keys/passwords are hard coded and stored within the source code. Pay special attention to any 'administrative' or backdoor accounts enabled in the source code. Storing 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 hard coded 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. hard coded in the app)
- if the app relies on an additional encrypted container stored in app data, ensure how the encryption key is used;
    - if key wrapping scheme is used, ensure that the master secret is initialized for each user, or container is re-encrypted with new key;
    - check how password change is handled and specifically, if you can use master secret or previous password to decrypt the container.

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

## Remediation

Using symmetric encryption with a hardcoded key is never a good idea. Instead, leverage the secure data storage and cryptography facilities offered by Android as described in the "Data Storage" chapter.

## References

### OWASP Mobile Top 10

- M6 - Broken Cryptography

### OWASP MASVS

- V3.1: "The app does not rely on symmetric cryptography with hardcoded keys as a sole method of encryption."

### CWE

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

### Info

- [1] iOS: Managing Keys, Certificates, and Passwords - https://developer.apple.com/library/content/documentation/Security/Conceptual/cryptoservices/KeyManagementAPIs/KeyManagementAPIs.html
- [2] Android: The Android Keystore System - https://developer.android.com/training/articles/keystore.html
- [3] Android: Hardware-backed Keystore - https://source.android.com/security/keystore/

# Testing Random Number Generation

## Overview

Cryptography requires secure pseudo random number generation (PRNG). Standard Java classes do not provide sufficient randomness and in fact may make it possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive

information.

In general, `SecureRandom` should be used. However, if the Android versions below KitKat are supported, additional care needs to be taken in order to work around the bug in Jelly Bean (Android 4.1-4.3) versions that failed to properly initialize the PRNG[4].

Most developers should instantiate `SecureRandom` via the default constructor without any arguments. Other constructors are for more advanced uses and, if used incorrectly, can lead to decreased randomness and security. The PRNG provider backing `SecureRandom` uses the `/dev/urandom` device file as the source of randomness by default.[5]

## Static Analysis

Identify all the instances of random number generators and look for either custom or known insecure `java.util.Random` class. This class produces an identical sequence of numbers for each given seed value; consequently, the sequence of numbers is predictable. The following sample source code shows weak random number generation:

```
import java.util.Random;
// ...

Random number = new Random(123L);
//...
for (int i = 0; i < 20; i++) {
  // Generate another random integer in the range [0, 20]
  int n = number.nextInt(21);
  System.out.println(n);
}
```

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

## Dynamic Analysis

Once an attacker is knowing what type of weak pseudo-random number generator (PRNG) is used, it can be trivial to write proof-of-concept to generate the next random value based on previously observed ones, as it was done for Java Random[1]. In case of very weak custom random generators it may be possible to observe the pattern statistically. Although the recommended approach would anyway be to decompile the APK and inspect the algorithm (see Static Analysis).

# Remediation

Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations with adequate length seeds. Prefer the no-argument constructor of `SecureRandom` that uses the system-specified seed value to generate a 128-byte-long random number[2]. In general, if a PRNG is not advertised as being cryptographically secure (e.g. `java.util.Random`), then it is probably a statistical PRNG and should not be used in security-sensitive contexts. Pseudo-random number generators can produce predictable numbers if the generator is known and the seed can be guessed[3]. A 128-bit seed is a good starting point for producing a "random enough" number.

The following sample source code shows the generation of a secure random number:

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
// ...

public static void main (String args[]) {
  SecureRandom number = new SecureRandom();
  // Generate 20 integers 0..20
  for (int i = 0; i < 20; i++) {
    System.out.println(number.nextInt(21));
  }
}
```

# References

### OWASP MASVS

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

**OWASP Mobile Top 10 2016**

- M6 - Broken Cryptography

**CWE**

- CWE-330: Use of Insufficiently Random Values

**Info**

- [1] Predicting the next Math.random() in Java - http://franklinta.com/2014/08/31/predicting-the-next-math-random-in-java/
- [2] 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
- [4] Some SecureRandom Thoughts - https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html
- [5] N. Elenkov, Android Security Internals, No Starch Press, 2014, Chapter 5.

# Testing Data Storage on Android

The protection of sensitive data, such as user credentials and private information, is a key focus in mobile security. In this chapter, you will learn about the APIs Android offers for local data storage, as well as best practices for using those APIs.

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

## Testing for Sensitive Data in Local Storage

## Overview

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

A vulnerability occurs when sensitive data is not properly protected by an app when persistently storing it. The app might be able to store it in different places, for example locally on the device or on an external SD card. When trying to exploit these 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] 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 widely used on the Android platform:

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

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

**Shared Preferences**

SharedPreferences[2] is a common approach to store Key/ Value pairs persistently in the filesystem by using an XML structure. Within an Activity the following code might be used to store sensitive information like a username and a password:

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

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

- The username and password is stored in clear text in `/data/data/<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 support for SQLite databases. The main package to manage the databases is `android.database.sqlite` . Within an Activity the following code might be used to store sensitive information like a username and a password:

```
SQLiteDatabase notSoSecure = openOrCreateDatabase("privateNotSoSecure",MODE_PRIVATE,null);
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Password VARCHAR);");
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');");
notSoSecure.close();
```

Once the activity is called, the database file `privateNotSoSecure` is created with the provided data and is stored in clear text in `/data/data/<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)

### Realm Databases

The Realm Database for Java [17] is getting more and more popular amongst developers. The database its contents can be encrypted by providing the configuration a key.

```
//the getKey() method either gets the keuy from the server or from a Keychain, or is deferred
from a password.
RealmConfiguration config = new RealmConfiguration.Builder()
  .encryptionKey(getKey())
  .build();

Realm realm = Realm.getInstance(config);
```

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

### 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 needs 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.

**KeyChain**

The KeyChain class [10] is used to store and retrieve *system-wide* private keys and their corresponding certificates (chain). The user will be prompted to set a lock screen pin or password to protect the credential storage if it hasn't been set, if something gets imported into the KeyChain the first time. Please note that the keychain is system-wide: so every application can access the materials stored in the KeyChain.

**KeyStore (AndroidKeyStore)**

The Android KeyStore [8] provides a means of (more or less) secure credential storage. As of Android 4.3, it provides public APIs for storing and using app-private keys. An app can create a new private/ public key pair to encrypt application secrets by using the public key and decrypt the same by using the private key.

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

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

1. User authentication authorizes the use of keys for a duration of time. All keys in this mode are authorized for use as soon as the user unlocks the device. The duration for which the authorization remains valid can be customized for each key. This option can only be used if the secure lock screen is enabled. If the user disables the secure lock screen, any stored keys become permanently invalidated.

2. User authentication authorizes a specific cryptographic operation associated with one key. In this mode, each operation involving such a key must be individually authorized by the user. Currently, the only means of such authorization is fingerprint authentication.

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

In a software-only implementation, the keys are encrypted with a per-user encryption master key [16]. In that case, an attacker can access all keys on a rooted device in the folder `/data/misc/keystore/` . As the master key is generated using the user's own lock screen pin/ password, the Android KeyStore is unavailable when the device is locked [9].

**Older Java-KeyStore**

Older Android versions do not have a KeyStore, but do have the KeyStore interface from JCA (Java Cryptography Architecture). One can use various KeyStores that implement this interface and provide secrecy and integrity protection to the keys stored in the keystore implementation. The impelemntations all rely on the fact that a file is stored on the filesystem, which then protects its contents by a password. For this, we recommend to use the BounceyCastle KeyStore (BKS). You can create one by using the `KeyStore.getInstance("BKS", "BC");` , where "BKS" is the keystore name (BounceycastleKeyStore) and "BC" is the provider (BounceyCastle). Alternatively you can use SpongeyCastle as a wrapper and initialize the keystore: `KeyStore.getInstance("BKS", "SC");` .

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

# Static Analysis

### Local Storage

As already pointed out, there are several ways to store information within Android. Several checks should therefore be applied to the source code to identify the storage mechanisms used within the Android app and whether or not sensitive data is processed insecurely.

- Check `AndroidManifest.xml` for permissions to read from or write to external storage, like `uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"`
- Check the source code for 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 Android KeyStore[8].
- Check if keys are disclosed.

### Typical Misuse: Hardcoded Cryptographic Keys

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

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

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

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

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

Let's consider this piece of code:

```
//A more complicated effort to store the XOR'ed halves of a key (instead of the key itself)
private static final String[] myCompositeKey = new String[]{
  "oNQavjbaNNSgEqoCkT9Em4imeQQ=","3o8eFOX4ri/F8fgHgiy/BS47"
};
```

Algorithm to decode the original key in this case might look like this[1]:

```
public void useXorStringHiding(String myHiddenMessage) {
  byte[] xorParts0 = Base64.decode(myCompositeKey[0],0);
  byte[] xorParts1 = Base64.decode(myCompositeKey[1],0);

  byte[] xorKey = new byte[xorParts0.length];
  for(int i = 0; i < xorParts1.length; i++){
    xorKey[i] = (byte) (xorParts0[i] ^ xorParts1[i]);
  }
  HidingUtil.doHiding(myHiddenMessage.getBytes(), xorKey, false);
}
```

Verify common places where secrets are usually hidden:

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

Example:

```
<resources>
    <string name="app_name">SuperApp</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
    <string name="secret_key">My_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

**KeyChain and Android KeyStore**

When going through the source code it should be analyzed if native mechanisms that are offered by Android are applied to the identified sensitive information. Sensitive information must not be stored in clear text but should be encrypted. If sensitive information needs to be stored on the device itself, several API calls are available to protect the data on the Android device by using the **KeyChain**[10] and **Android 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 Android KeyStore and Cipher mechanisms to securely store encrypted information on the device. Look for the pattern `import java.security.KeyStore`, `import javax.crypto.Cipher`, `import java.security.SecureRandom` and corresponding usages.
- The `store(OutputStream stream, char[] password)` function can be used to store the KeyStore to disk with a specified password. Check that the password provided is not hardcoded and is defined by user input as this should only be known to the user. Look for the 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.
- Check if there is a Realm database available and if it is unencrypted & contains sensitive information, which is in `/data/data<package_name>/files/<dbfilename>.realm`, where dbfilename

is `default` by default. Inspecting the Realm database is done with the RealmBrowser.

## Remediation

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

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

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

- Encryption or decryption functions that were self implemented need to be avoided. Instead use Android implementations such as Cipher[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] Security Tips for Storing Data - http://developer.android.com/training/articles/security-tips.html#StoringData [2] 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 [16] Nikolay Elenvok - Credential storage enhancements in Android 4.3 - https://nelenkov.blogspot.sg/2013/08/credential-storage-enhancements-android-43.html [17] Realm Database - https://realm.io/docs/java/latest/

**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 or simply for usage statistics. Log files can be stored locally when being offline and being sent to the endpoint once being online again. However, logging sensitive data such as

usernames or session IDs might expose the data to attackers or malicious applications and violates the confidentiality of the data. Log files can be created in various ways and the following list shows the mechanisms that are available on Android:

- Log Class[1]
- Logger Class
- System.out/System.err.print

## Static Analysis

Several checks should be applied to the source code to identify the logging mechanisms used within the Android app. This allows to identify if sensitive data is processed insecurely. The source code should be checked for logging mechanisms used within the Android App, by searching for the following:

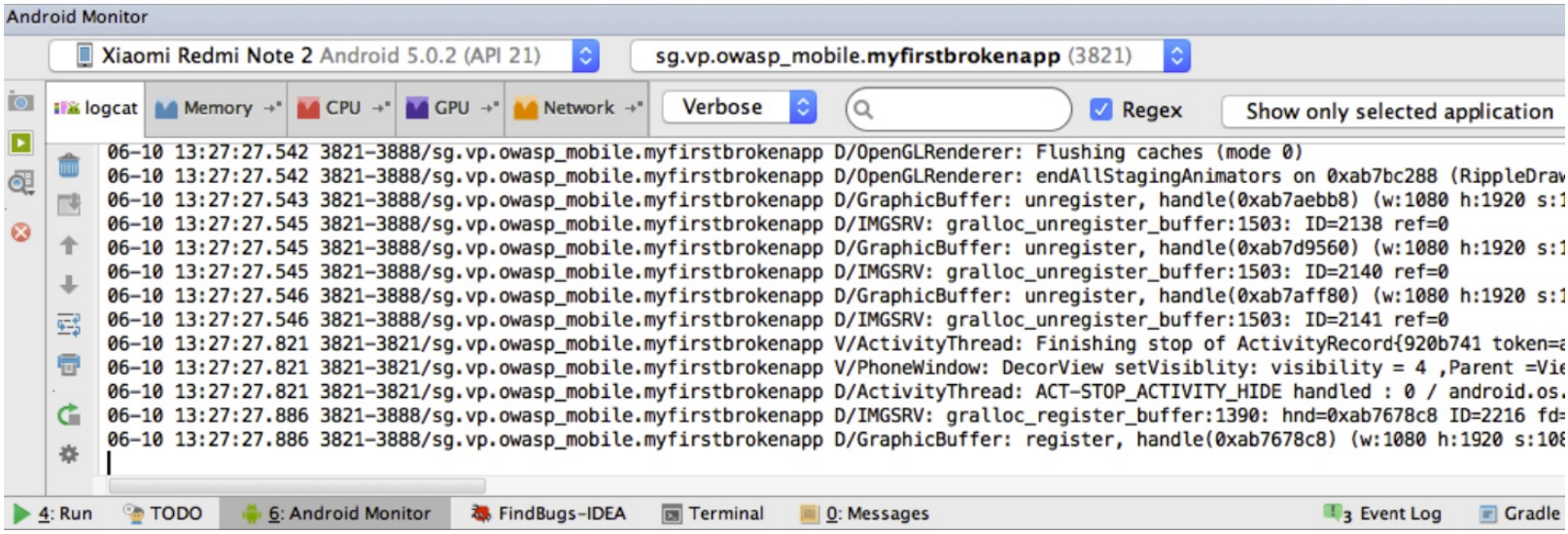
1. Functions and classes like:
    - `android.util.Log`
    - `Log.d` | `Log.e` | `Log.i` | `Log.v` | `Log.w` | `Log.wtf`
    - `Logger`
    - `System.out.print` | `System.err.print`
2. Keywords and system output to identify non-standard log mechanisms like:
    - logfile
    - logging
    - logs

## Dynamic Analysis

Use the mobile app extensively so that all functionality is at least triggered once. Afterwards identify the data directory of the application in order to look for log files ( `/data/data/package_name` ). Check if log data is generated by checking the application logs, as some mobile applications create and store their own logs in the data directory.

Many application developers still use `System.out.println()` or `printStackTrace()` instead of a proper logging class. Therefore the testing approach also needs to cover all output generated by the application during starting, running and closing of it. In order to verify what data is printed directly by using `System.out.println()` or `printStackTrace()` the tool `LogCat` [2] can be used to check the app output. Two different approaches are available to execute LogCat.

- LogCat is already part of *Dalvik Debug Monitor Server* (DDMS) and is built into Android Studio. If the app is in debug mode and running, the log output is shown in the Android Monitor in the LogCat tab. Patterns can be defined in LogCat to filter the log output of the app.




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

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

# Remediation

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

```
-assumenosideeffects class android.util.Log
{
public static boolean isLoggable(java.lang.String, int);
public static int v(...);
public static int i(...);
public static int w(...);
public static int d(...);
public static int e(...);
public static int wtf(...);
}
```

# 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] Overview of Class Log - http://developer.android.com/reference/android/util/Log.html
- [2] Debugging Logs with LogCat - http://developer.android.com/tools/debugging/debugging-log.html

### Tools

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

# Testing Whether Sensitive Data is Sent to Third Parties

## Overview

Different 3rd party services are available that can be embedded into the app to implement different features. These features can vary from tracker services to monitor the user behavior within the app, selling banner advertisements or to create a better user experience. Interacting with these services abstracts the complexity and neediness to implement the functionality on its own and to reinvent the wheel.

The downside is that a developer doesn't know in detail what code is executed via 3rd party libraries and therefore giving up visibility. Consequently it should be ensured that not more information as needed is sent to the service and that no sensitive information is disclosed.

3rd party services are mostly implemented in two ways:

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

## Static Analysis

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

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

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

## Dynamic Analysis

All requests made to external services should be analyzed if any sensitive information is embedded into them. Dynamic analysis can be performed by launching a Man-in-the-middle (MITM) attack using *Burp Proxy*[1] 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] Configure Burp with Android - https://support.portswigger.net/customer/portal/articles/1841101-configuring-an-android-device-to-work-with-burp [2] 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]:

```
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] 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] 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.

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

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

# Remediation

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

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

            public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
                return false;
            }
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] Copy and Paste in Android - https://developer.android.com/guide/topics/text/copy-paste.html

**Tools**

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

# 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]
- Services[2]
  - 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*:

```
<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

## Testing Content Providers

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

```
dz> run app.provider.info -a com.mwr.example.sieve
  Package: com.mwr.example.sieve
  Authority: com.mwr.example.sieve.DBContentProvider
  Read Permission: null
  Write Permission: null
  Content Provider: com.mwr.example.sieve.DBContentProvider
  Multiprocess Allowed: True
  Grant Uri Permissions: False
  Path Permissions:
  Path: /Keys
  Type: PATTERN_LITERAL
  Read Permission: com.mwr.example.sieve.READ_KEYS
  Write Permission: com.mwr.example.sieve.WRITE_KEYS
  Authority: com.mwr.example.sieve.FileBackupProvider
  Read Permission: null
  Write Permission: null
  Content Provider: com.mwr.example.sieve.FileBackupProvider
  Multiprocess Allowed: True
  Grant Uri Permissions: False
```

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

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

```
dz> run scanner.provider.finduris -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Unable to Query content://com.mwr.
example.sieve.DBContentProvider/
...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/Keys
Accessible content URIs:
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

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

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --
vertical
_id: 1
service: Email
username: incognitoguy50
password: PSFjqXIMVa5NJFudgDuuLVgJYFD+8w== (Base64
-
encoded)
email: incognitoguy50@gmail.com
```

In addition to querying data, Drozer can be used to update, insert and delete records from a vulnerable content provider:

- Insert record

```
dz> run app.provider.insert content://com.vulnerable.im/messages
                --string date 1331763850325
                --string type 0
                --integer _id 7
```

- Update record

```
dz> run app.provider.update content://settings/secure
                --selection "name=?"
                --selection-args assisted_gps_enabled
                --integer value 0
```

- Delete record

```
dz> run app.provider.delete content://settings/secure
                --selection "name=?"
                --selection-args my_setting
```

### SQL Injection in Content Providers

The Android platform promotes the use of SQLite databases for storing user data. Since these databases use SQL, they can be vulnerable to SQL injection. The Drozer module `app.provider.query` can be used to test for SQL injection by manipulating the projection and selection fields that are passed to the content provider:

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

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

If vulnerable to SQL Injection, the application will return a verbose error message. SQL Injection in Android can be exploited to modify or query data from the vulnerable content provider. In the following example, the Drozer module `app.provider.query` is used to list all tables in the database:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --
projection "*
FROM SQLITE_MASTER WHERE type='table';--"
| type  | name             | tbl_name         | rootpage | sql              |
| table | android_metadata | android_metadata | 3        | CREATE TABLE ... |
| table | Passwords        | Passwords        | 4        | CREATE TABLE ... |
| table | Key              | Key              | 5        | CREATE TABLE ... |
```

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

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

These steps can be automated by using the `scanner.provider.injection` module, which automatically finds vulnerable content providers within an app:

```
dz> run scanner.provider.injection -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Injection in Projection:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
Injection in Selection:
  content://com.mwr.example.sieve.DBContentProvider/Keys/
  content://com.mwr.example.sieve.DBContentProvider/Passwords
  content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

**File System Based Content Providers**

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

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

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

```
dz> run scanner.provider.traversal -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Vulnerable Providers:
  content://com.mwr.example.sieve.FileBackupProvider/
  content://com.mwr.example.sieve.FileBackupProvider
```

Note that `adb` can also be used to query content providers on a device:

```
$ adb shell content query --uri
content://com.owaspomtg.vulnapp.provider.CredentialProvider/credentials
Row: 0 id=1, username=admin, password=StrongPwd
Row: 1 id=2, username=test, password=test
...
```

### Vulnerable Broadcasts

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] IPCBinder - https://developer.android.com/reference/android/os/Binder.html [2] 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

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

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

- Stock Android has built-in USB backup facilities. A full data backup, or a backup of a particular app's data directory, can be obtained using the `abd backup` command when USB debugging is enabled.
- Google also provides a "Back Up My Data" feature that backs up all app data to Google's servers.
- Multiple Backup APIs are available to app developers:
    - Key/ Value Backup (Backup API or Android Backup Service) uploads selected data to the Android Backup Service.
    - Auto Backup for Apps: With Android 6.0 (>= API level 23), Google added the "Auto Backup for Apps feature". This feature automatically syncs up to 25MB of app data to the user's Google Drive account.
- OEMs may add additional options. For example, HTC devices have a "HTC Backup" option that, when activated, performs daily backups to the cloud.

# Static Analysis

### Local

In order to backup all your application data Android provides an attribute called `allowBackup` [1]. 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** Auto Backup is configured through the boolean attribute `android:allowBackup` within the application's manifest file. If not explicitly set, applications targeting Android 6.0 (API Level 23) or higher enable Auto Backup by default[10]. 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], 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] Documentation for the application tag - https://developer.android.com/guide/topics/manifest/application-element.html#allowbackup [2] 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 [10] Enabling AutoBackup - https://developer.android.com/guide/topics/data/autobackup.html#EnablingAutoBackup

**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.

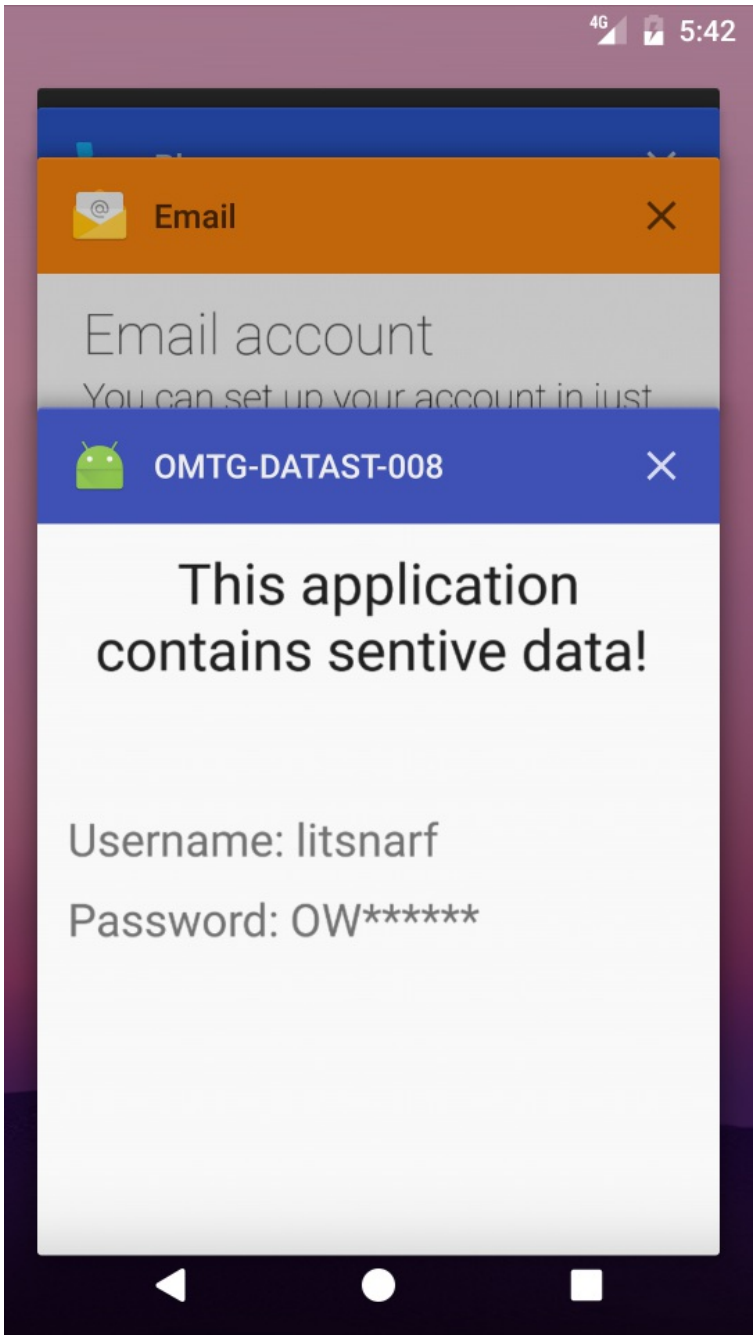
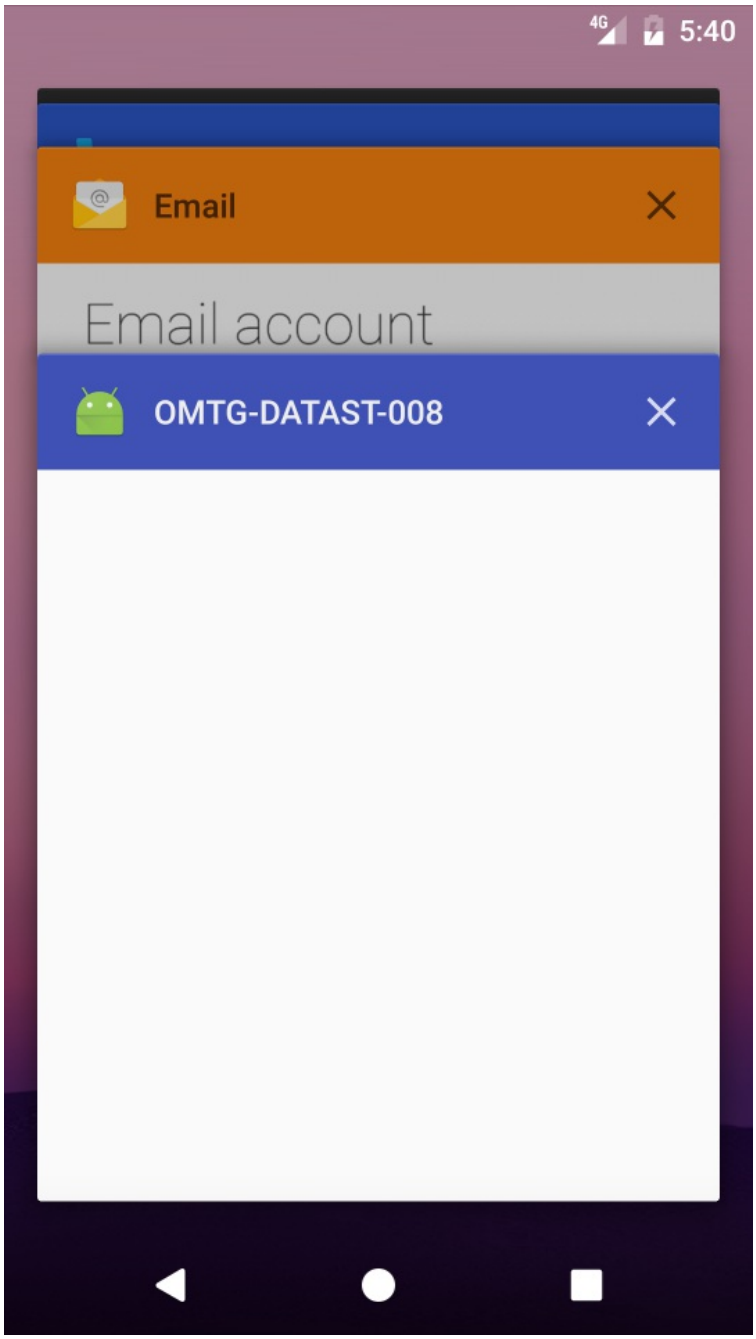
To verify if the application may expose sensitive information via task switcher, detect if the `FLAG_SECURE` [1] option is set. You should be able to find something similar to the following code snippet.

```
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 shown below, if `FLAG_SECURE` is set (image on the right), the snapshot is empty, while if the `FLAG_SECURE` is not set (image on the left), information within the activity are shown:

| `FLAG_SECURE` not set | `FLAG_SECURE` set |
| --- | --- |
|  |  |





# 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] 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

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

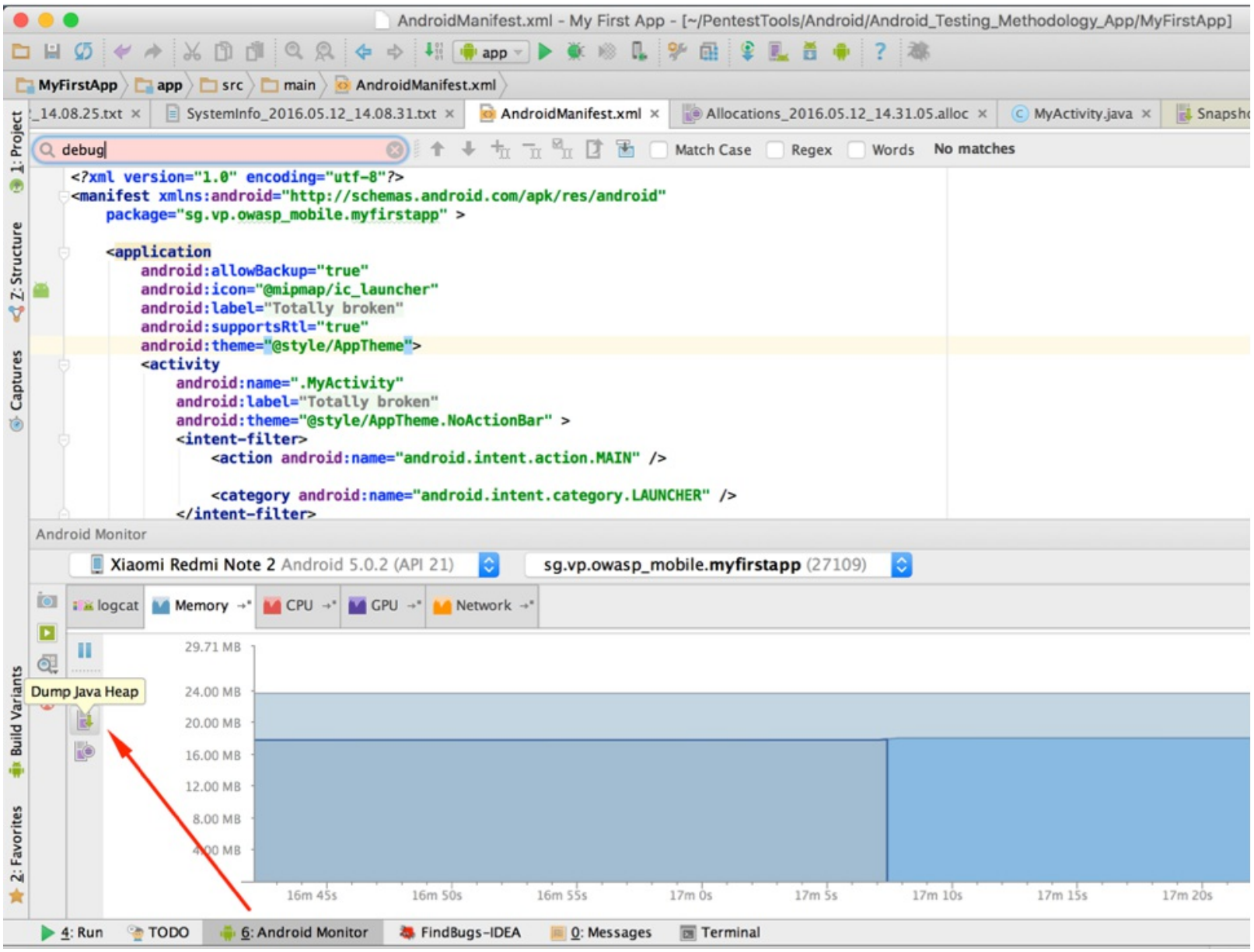
- Verify that no sensitive information is stored in an immutable structure. Immutable structures are not really overwritten in the heap, even after nullification or changing them. Instead, by changing the immutable structure, a copy is created on the heap. `BigInteger` and `String` are two of the most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

**NOTICE**: Destroying a key (e.g. `SecretKey secretKey = new SecretKeySpec("key".getBytes(), "AES"); secret.destroy();` ) does *not* work, nor nullifying the backing byte-array from `secretKey.getEncoded()` as the SecretKeySpec based key returns a copy of the backing byte-array. Therefore the developer should, in case of not using the `AndroidKeyStore` make sure that the key is wrapped and properly protected (see remediation for more details). Understand that an RSA keypair is based on `BigInteger` as well and therefore reside in memory after first use outside of the `AndroidKeyStore` . Lastly, some of the ciphers do not properly clean up their byte-arrays, for instance: the AES `Cipher` in `BounceyCastle` does not always clean up its latest working key.

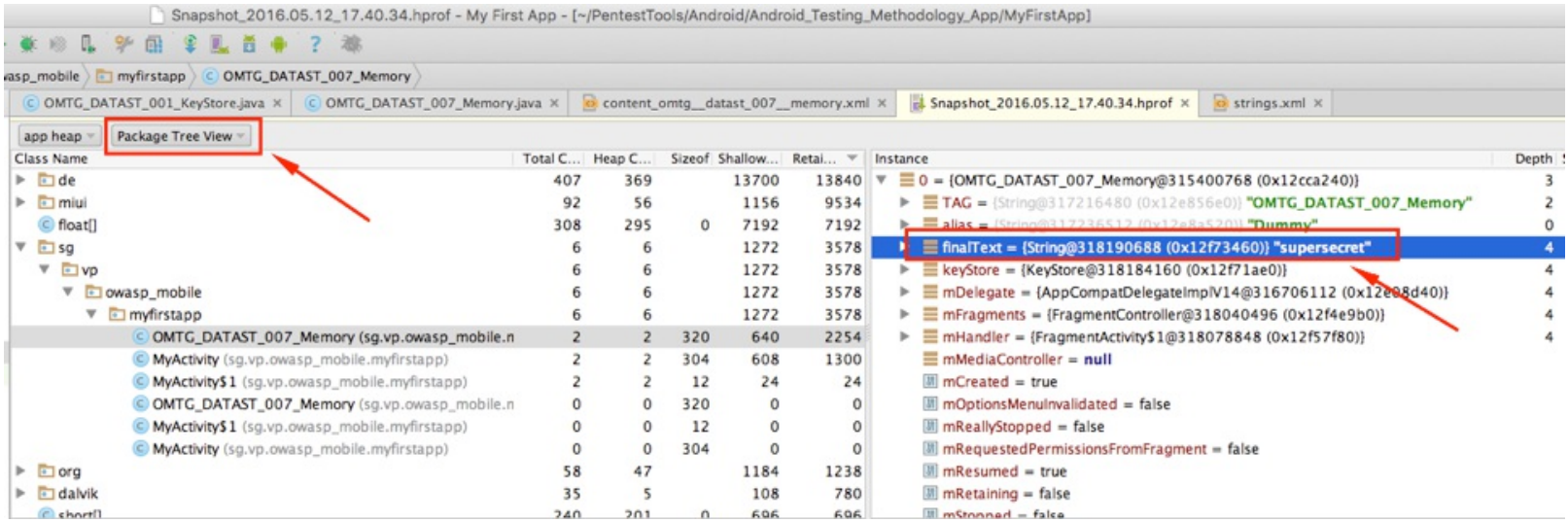
## Dynamic Analysis

To analyse the memory of an app in Android Studio, the app must be **debuggable**. See the instructions in "Reverse and Tampering" 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, no immutable structures should be used to carry secrets (E.g. `String` , `BigInteger` ). Nullifying them will not be effective: the Garbage collector might collect them, but they might remain in the JVMs heap for a longer period of time. Rather use byte-arrays ( `byte[]` ) or char-arrays ( `char[]` ) which are cleaned after the operations are done:

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

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

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

```
public class ErasableSecretKey implements Serializable {

    public static final int KEY_LENGTH = 256;

    private java.security.Key secKey;
```

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

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

  //clean the internal key, but only do so if it is not destroyed yet.
    public void clean() {
        try {
            if (this.getKey() instanceof Destroyable) {
                ((Destroyable) this.getKey()).destroy();
            }

        } catch (DestroyFailedException e) {
            //choose what you want to do now: so you could not destroy it, would you run on? Or
rather inform the caller of the clean method informing him of the failure?
        }
    }
  //convinience method that takes away the null-check so you can always just call
ErasableSecretKey.clearKey(thekeytobecleared)
    public static void clearKey(ErasableSecretKey key) {
        if (key != null) {
            key.clean();
        }
    }

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

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

```
            this.algorithm = algorithm;
        }

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

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

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

    //destroy the key.
        public void destroy() throws DestroyFailedException {
            if (this.key != null) {
                Arrays.fill(this.key, (byte) 0);
            }

            this.key = null;
        }

        public boolean isDestroyed() {
            return this.key == null;
        }
    }


    public final java.security.Key getKey() {
        return this.secKey;
    }

}
```

# References

**OWASP Mobile Top 10 2016**

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

**OWASP MASVS**

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

**CWE**

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

**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]. The *Device Administration API*[2] 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."

**Info**

- [1] Settings.Secure - https://developer.android.com/reference/android/provider/Settings.Secure.html
- [2] 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.

## Dynamic Analysis

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

## Remediation

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

## References

### OWASP Mobile Top 10 2016

- M1 - Improper Platform Usage

### OWASP MASVS

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

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

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

**Ultimately, the reverse engineer always wins.**

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

The bad news is that dealing with multi-threaded anti-debugging controls, cryptographic white-boxes, stealthy anti-tampering features and highly complex control flow transformations is not for the faint-hearted. The most effective software protection schemes are highly proprietary and won't be beaten

using standard tweaks and tricks. Defeating them requires tedious manual analysis, coding, frustration, and - depending on your personality - sleepless nights and strained relationships.

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

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

# Basic Tampering Techniques

## Binary Patching

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

One thing to keep in mind is that modern mobile OSes strictly enforce code signing, so running modified apps is not as straightforward as it used to be in traditional Desktop environments. Yep, security experts had a much easier life in the 90s! Fortunately, this is not all that difficult to do if you work on your own device - it simply means that you need to 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" that replaces Cycript's runtime with a Frida-based runtime called Mjølner[1]. 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[2].

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

## Static and Dynamic Binary Analysis

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

## Using Disassemblers and Decompilers

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

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

## Debugging and Tracing

In the traditional sense, debugging is the process of identifying and isolating problems in a program as part of the software development life cycle. The very same tools used for debugging are of great value to reverse engineers even when identifying bugs is not the primary goal. Debuggers enable suspending a program at any point during runtime, inspect the internal state of the process, and even modify the content of registers and memory. These abilities make it *much* easier to figure out what a program is actually doing.

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

## Advanced Techniques

For more complicated tasks, such as de-obfuscating heavily obfuscated binaries, you won't get far without automating certain parts of the analysis. For example, understanding and simplifying a complex control flow graph manually in the disassembler would take you years (and most likely drive you mad, way before you're done). Instead, you can augment your work flow with custom made

scripts or tools. Fortunately, modern disassemblers come with scripting and extension APIs, and many useful extensions are available for popular ones. Additionally, open-source disassembling engines and binary analysis frameworks exist to make your life easier.

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

## Dynamic Binary Instrumentation

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

The Valgrind README contains specific compilation instructions for Android - http://valgrind.org/docs/manual/dist.readme-android.html

## 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[3] and Angr [4].

**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) [5].

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

# References


- [1] Cycript fork powered by Frida - https://github.com/nowsecure/frida-cycript
- [2] Cycript on steroids: Pumping up portability and performance with Frida -

- [3] Radare2 - https://github.com/radare/radare2
- [4] Angr - http://angr.io
- [5] https://triton.quarkslab.com/files/csaw2016-sos-rthomas-jsalwan.pdf

# Android Reverse Engineering Guide

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] 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], which comes with the Android SDK, platform tools and emulator, as well as a manager app for managing the various SDK versions and framework components. With Android Studio, you also get an SDK Manager app that lets you install the Android

SDK tools and manage SDKs for various API levels, as well as the emulator and an AVD Manager application to create emulator images. Make sure that the following is installed on your system:

- The newest SDK Tools and SDK Platform-Tools packages. These packages include the Android Debugging Bridge (ADB) client as well as other tools that interface with the Android platform. In general, these tools are backward-compatible, so you need only one version of those installed.
- The Android NDK. This is the Native Development Kit that contains prebuilt toolchains for cross-compiling native code for different architectures.

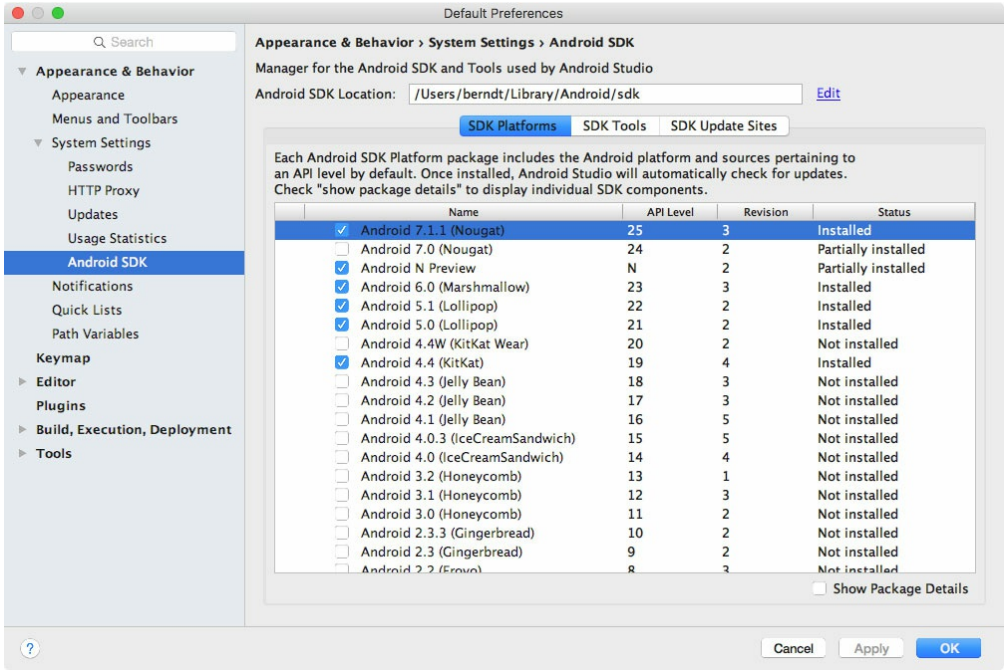
In addition to the SDK and NDK, you'll also something to make Java bytecode more human-friendly. Fortunately, Java decompilers generally deal well with Android bytecode. Popular free decompilers include JD [3], Jad [4], Procyon [5] and CFR [6]. For convenience, we have packed some of these decompilers into our `apkx` wrapper script [7]. This script completely automates the process of extracting Java code from release APK files and makes it easy to experiment with different backends (we'll also use it in some of the examples below).

Other than that, it's really a matter of preference and budget. A ton of free and commercial disassemblers, decompilers, and frameworks with different strengths and weaknesses exist - we'll cover some of them below.

## Setting up the Android SDK

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

- API 21: Android 5.0
- API 22: Android 5.1
- API 23: Android 6.0
- API 24: Android 7.0
- API 25: Android 7.1
- API 26: Android O Developer Preview




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

```
Windows:

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

MacOS:

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

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

# Setting up the Android NDK

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

| Architecture | Toolchain name |
|---|---|
| ARM-based | arm-linux-androideabi-<gcc-version> |
| x86-based | x86-<gcc-version> |
| MIPS-based | mipsel-linux-android-<gcc-version> |
| ARM64-based | aarch64-linux-android-<gcc-version> |
| X86-64-based | x86_64-<gcc-version> |
| MIPS64-based | mips64el-linux-android-<gcc-version> |

In addition to the picking the right architecture, you need to specify the correct sysroot for the native API level you want to target. The sysroot is a directory that contains the system headers and libraries for your target. Available native APIs vary by Android API level. Possible sysroots for respective Android API levels reside under `$NDK/platforms/`, each API-level directory contains subdirectories for the various CPUs and architectures.

One possibility to set up the build system is exporting the compiler path and necessary flags as environment variables. To make things easier however, the NDK allows you to create a so-called standalone toolchain - a "temporary" toolchain that incorporates the required settings.

To set up a standalone toolchain, download the latest stable version of the NDK [8]. Extract the ZIP file, change into the NDK root directory and run the following command:

```
$ ./build/tools/make_standalone_toolchain.py --arch arm --api 24 --install-dir /tmp/android-7-toolchain
```

This creates a standalone toolchain for Android 7.0 in the directory `/tmp/android-7-toolchain`. For convenience, you can export an environment variable that points to your toolchain directory - we'll be using this in the examples later. Run the following command, or add it to your `.bash_profile` or other startup script.

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

# Building a Reverse Engineering Environment For Free

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

For navigating the decompiled sources we recommend using IntelliJ [9], a relatively light-weight IDE that works great for browsing code and allows for basic on-device debugging of the decompiled apps. However, if you prefer something that's clunky, slow and complicated to use, Eclipse [10] is the right IDE for you (note: This piece of advice is based on the author's personal bias).

If you don't mind looking at Smali instead of Java code, you can use the smalidea plugin for IntelliJ for debugging on the device [11]. Smalidea supports single-stepping through the bytecode, identifier renaming and watches for non-named registers, which makes it much more powerful than a JD + IntelliJ setup.

APKTool [12] is a popular free tool that can extract and disassemble resources directly from the APK archive and disassemble Java bytecode to Smali format (Smali/Baksmali is an assembler/disassembler for the Dex format. It's also icelandic for "Assembler/Disassembler"). APKTool allows you to reassemble the package, which is useful for patching and applying changes to the Manifest.

More elaborate tasks such as program analysis and automated de-obfuscation can be achieved with open source reverse engineering frameworks such as Radare2 [13] and Angr [14]. You'll find usage examples for many of these free tools and frameworks throughout the guide.

## Commercial Tools

Even though it is possible to work with a completely free setup, you might want to consider investing in commercial tools. The main advantage of these tools is convenience: They come with a nice GUI, lots of automation, and end user support. If you earn your daily bread as a reverse engineer, this will save you a lot of time.

#### JEB

JEB [15], a commercial decompiler, packs all the functionality needed for static and dynamic analysis of Android apps into an all-in-one package, is reasonably reliable and you get quick support. It has a built-in debugger, which allows for an efficient workflow – setting breakpoints directly in the decompiled (and annotated sources) is invaluable, especially when dealing with ProGuard-

obfuscated bytecode. Of course convenience like this doesn't come cheap - and since version 2.0 JEB has changed from a traditional licensing model to a subscription-based one, so you'll need to pay a monthly fee to use it.

#### IDA Pro

IDA Pro [16] understands ARM, MIPS and of course Intel ELF binaries, plus it can deal with Java bytecode. It also comes with debuggers for both Java applications and native processes. With its capable disassembler and powerful scripting and extension capabilities, IDA Pro works great for static analysis of native programs and libraries. However, the static analysis facilities it offers for Java code are somewhat basic – you get the Smali disassembly but not much more. There's no navigating the package and class structure, and some things (such as renaming classes) can't be done which can make working with more complex Java apps a bit tedious.

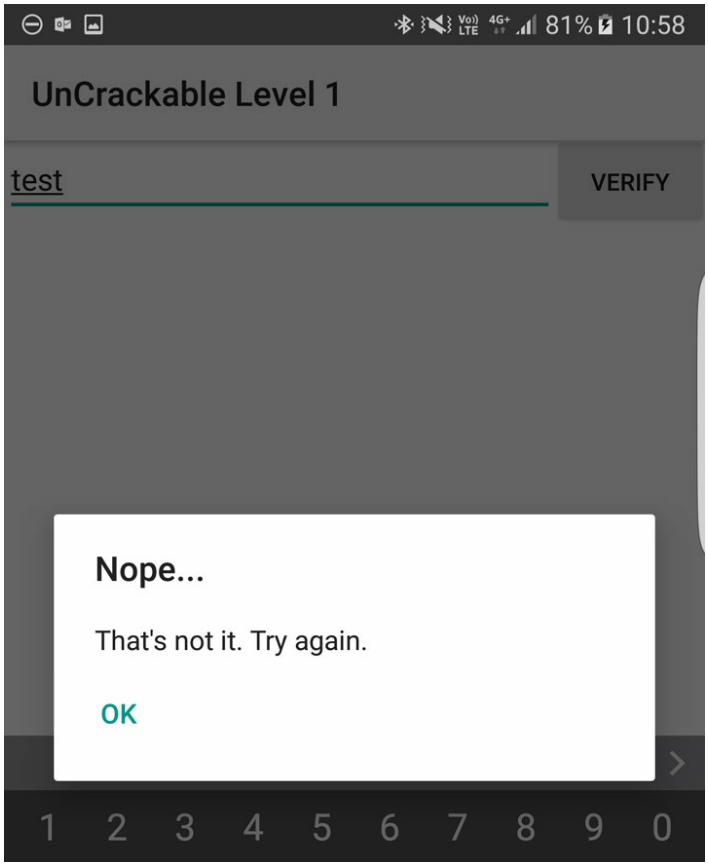
## Reverse Engineering

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

## Statically Analyzing Java Code

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

```
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Crackmes/Android/Level_01/UnCrackable-
Level1.apk
$ adb install UnCrackable-Level1.apk
```




Seems like we're expected to find some kind of secret code!

Most likely, we're looking for a secret string stored somewhere inside the app, so the next logical step is to take a look inside. First, unzip the APK file and have a look at the content.

```
$ unzip UnCrackable-Level1.apk -d UnCrackable-Level1
Archive:  UnCrackable-Level1.apk
  inflating: UnCrackable-Level1/AndroidManifest.xml
  inflating: UnCrackable-Level1/res/layout/activity_main.xml
  inflating: UnCrackable-Level1/res/menu/menu_main.xml
 extracting: UnCrackable-Level1/res/mipmap-hdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-mdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-xhdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-xxhdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-xxxhdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/resources.arsc
  inflating: UnCrackable-Level1/classes.dex
  inflating: UnCrackable-Level1/META-INF/MANIFEST.MF
  inflating: UnCrackable-Level1/META-INF/CERT.SF
  inflating: UnCrackable-Level1/META-INF/CERT.RSA
```

In the standard case, all the Java bytecode and data related to the app is contained in a file named `classes.dex` in the app root directory. This file adheres to the Dalvik Executable Format (DEX), an Android-specific way of packaging Java programs. Most Java decompilers expect plain class files or JARs as input, so you need to convert the classes.dex file into a JAR first. This can be done using `dex2jar` or `enjarify`.

Once you have a JAR file, you can use any number of free decompilers to produce Java code. In this example, we'll use CFR as our decompiler of choice. CFR is under active development, and brand-new releases are made available regularly on the author's website [6]. Conveniently, CFR has been released under a MIT license, which means that it can be used freely for any purposes, even though its source code is not currently available.

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

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

This should copy `apkx` to `/usr/local/bin`. Run it on `UnCrackable-Level1.apk`:

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

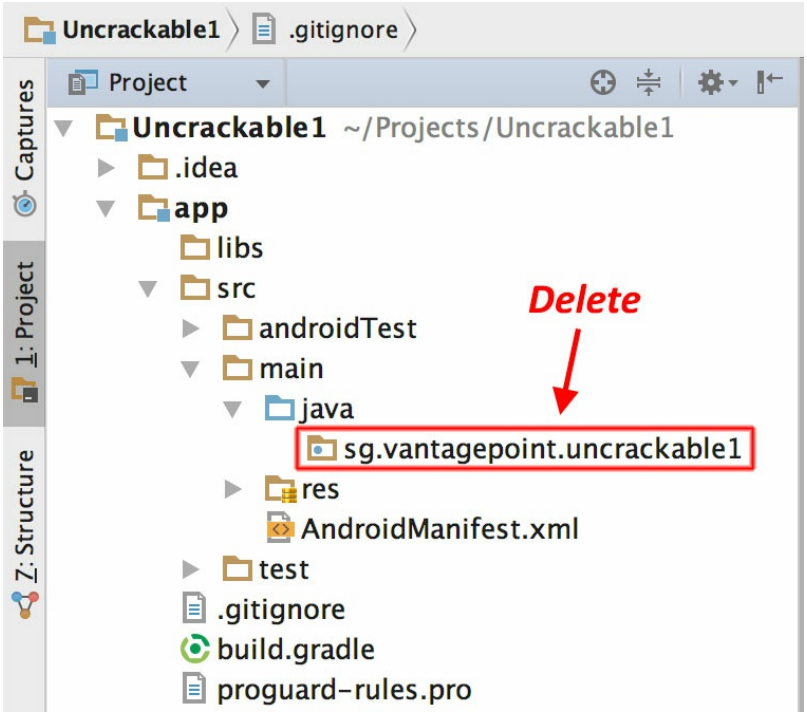
You should now find the decompiled sources in the directory `Uncrackable-Level1/src`. To view the sources, a simple text editor (preferably with syntax highlighting) is fine, but loading the code into a Java IDE makes navigation easier. Let's import the code into IntelliJ, which also gets us on-device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

Open IntelliJ and select "Android" as the project type in the left tab of the "New Project" dialog. Enter "Uncrackable1" as the application name and "vantagepoint.sg" as the company name. This results in the package name "sg.vantagepoint.uncrackable1", which matches the original package

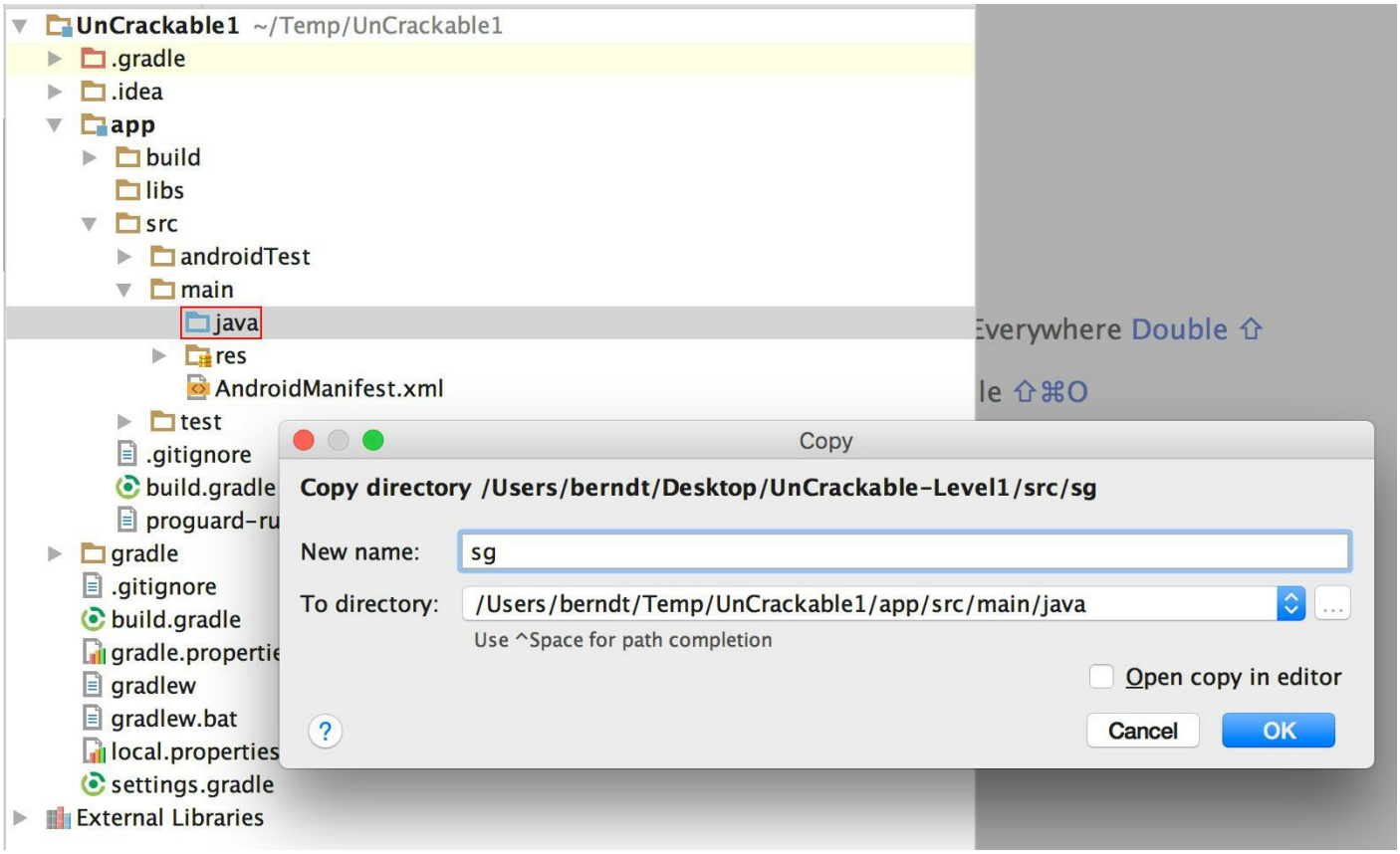
name. Using a matching package name is important if you want to attach the debugger to the running app later on, as Intellij uses the package name to identify the correct process.


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

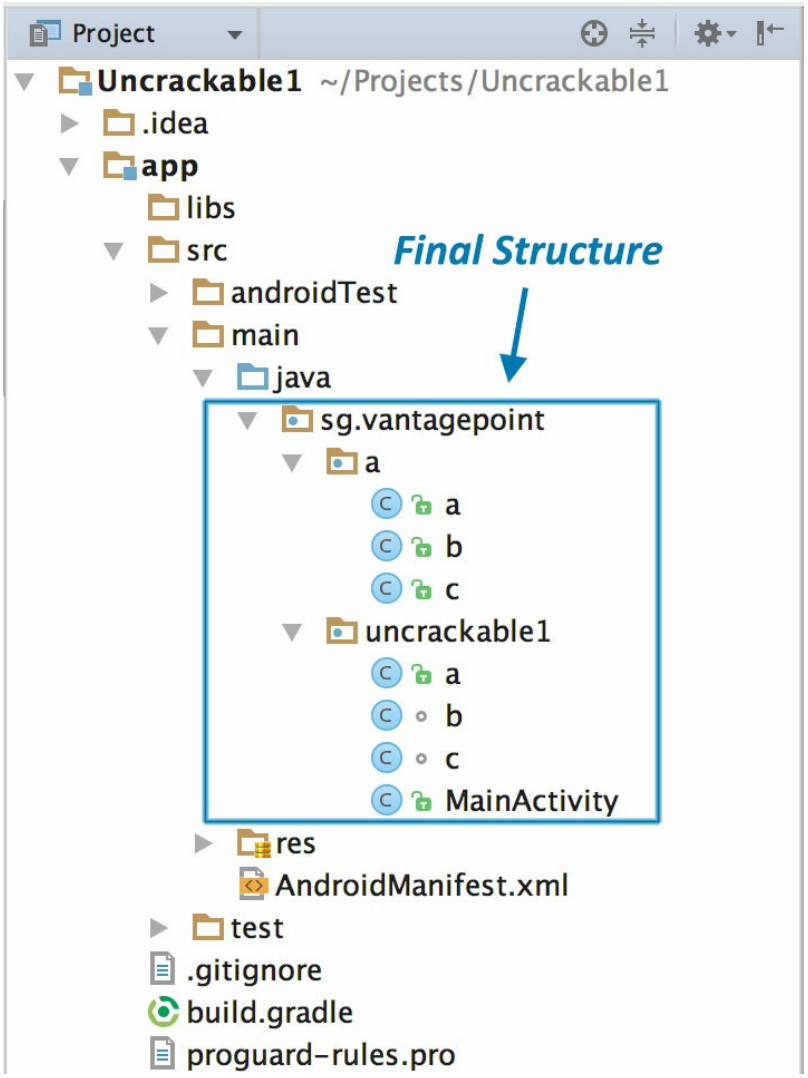
Once the project is created, expand the "1: Project" view on the left and navigate to the folder `app/src/main/java`. Right-click and delete the default package "sg.vantagepoint.uncrackable1" created by IntelliJ.




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

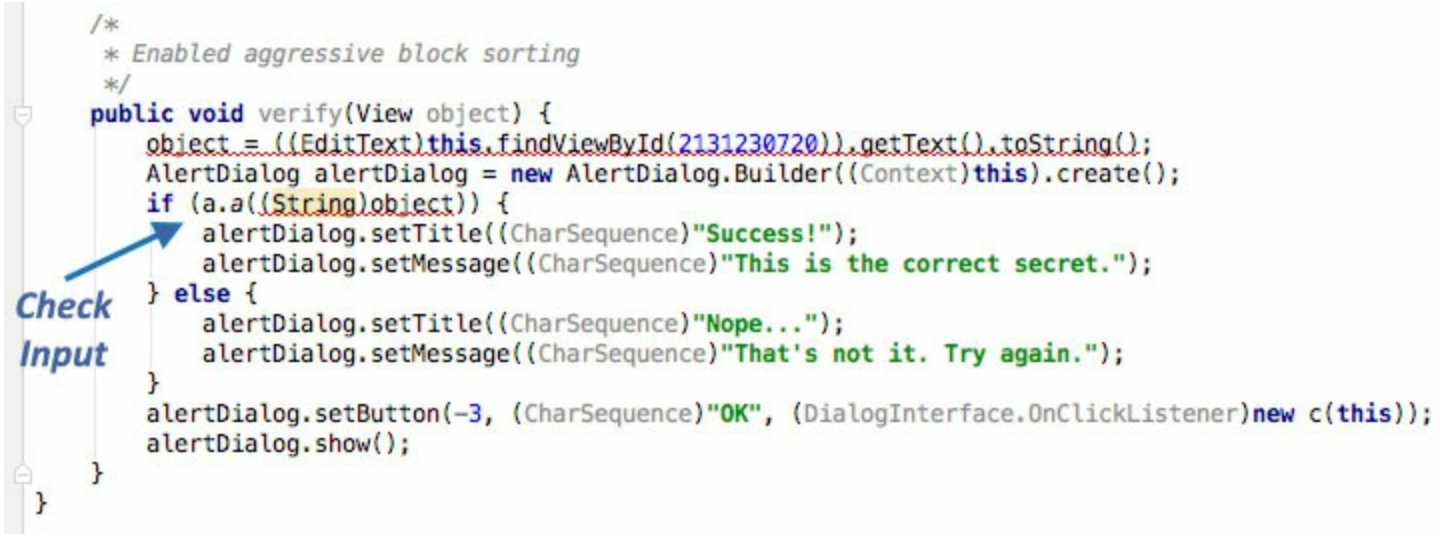



You'll end up with a structure that resembles the original Android Studio project from which the app was built.



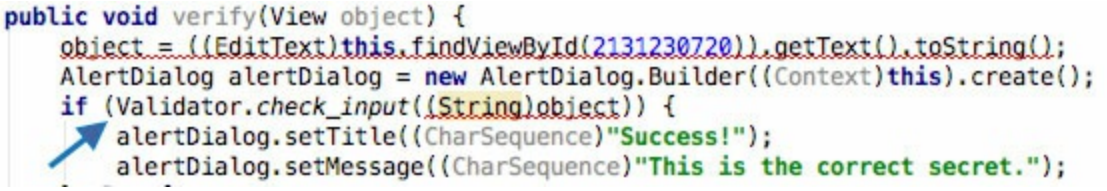

As soon as IntelliJ is done indexing the code, you can browse it just like any normal Java project. Note that many of the decompiled packages, classes and methods have weird one-letter names... this is because the bytecode has been "minified" with ProGuard at build time. This is a a basic type of obfuscation that makes the bytecode a bit more difficult to read, but with a fairly simple app like this one it won't cause you much of a headache - however, when analyzing a more complex app, it can get quite annoying.

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




```
/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (a.a((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
    } else {
        alertDialog.setTitle((CharSequence)"Nope...");
        alertDialog.setMessage((CharSequence)"That's not it. Try again.");
    }
    alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}
}
```

Right-click the class name - the first `a` in `a.a` - and select Refactor->Rename from the drop-down menu (or press Shift-F6). Change the class name to something that makes more sense given what you know about the class so far. For example, you could call it "Validator" (you can always revise the name later as you learn more about the class). `a.a` now becomes `Validator.a` . Follow the same procedure to rename the static method `a` to `check_input` .



```
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (Validator.check_input((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
```

Congratulations - you just learned the fundamentals of static analysis! It is all about theorizing, annotating, and gradually revising theories about the analyzed program, until you understand it completely - or at least, well enough for whatever you want to achieve.

Next, ctrl+click (or command+click on Mac) on the `check_input` method. This takes you to the method definition. The decompiled method looks as follows:

```
    public static boolean check_input(String string) {
        byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=",
(int)0);
        byte[] arrby2 = new byte[]{};
        try {
            arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d806cf50473cc"),
arrby);
            arrby2 = arrby;
        }sa
        catch (Exception exception) {
            Log.d((String)"CodeCheck", (String)("AES error:" + exception.getMessage()));
        }
        if (string.equals(new String(arrby2))) {
            return true;
        }
        return false;
    }
```

So, we have a base64-encoded String that's passed to a function named `a` in the package `sg.vantagepoint.a.a` (again everything is called `a`) along with something that looks suspiciously like a hex-encoded encryption key (16 hex bytes = 128bit, a common key length). What exactly does this particular `a` do? Ctrl-click it to find out.

```
public class a {
    public static byte[] a(byte[] object, byte[] arrby) {
        object = new SecretKeySpec((byte[])object, "AES/ECB/PKCS7Padding");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, (Key)object);
        return cipher.doFinal(arrby);
    }
}
```

Now we are getting somewhere: It's simply standard AES-ECB. Looks like the base64 string stored in `arrby1` in `check_input` is a ciphertext, which is decrypted using 128bit AES, and then compared to the user input. As a bonus task, try to decrypt the extracted ciphertext and get the secret value!

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

## Statically Analyzing Native Code

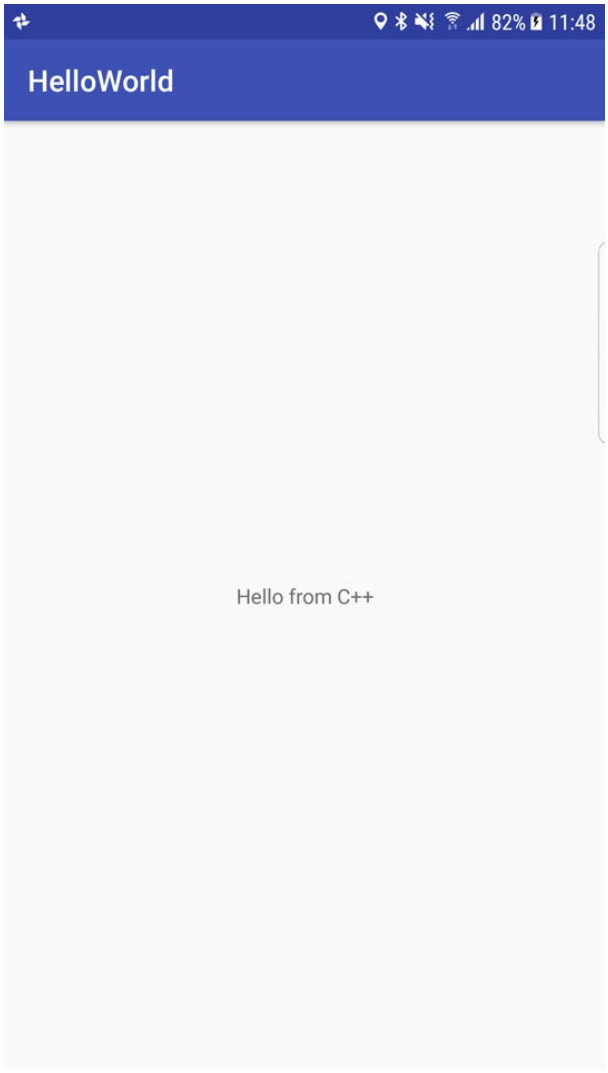
Dalvik and ART both support the Java Native Interface (JNI), which defines defines a way for Java code to interact with native code written in C/C++. Just like on other Linux-based operating systes, native code is packaged into ELF dynamic libraries ("*.so"), which are then loaded by the Android app 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 [17]. Bionic adds support for important Android-specific services such as system properties and logging, and is not fully POSIX-compatible.

Download HelloWorld-JNI.apk from the OWASP MSTG repository and, optionally, install and run it on your emulator or Android device.

```
$ wget HelloWord-JNI.apk
$ adb install HelloWord-JNI.apk
```

This app is not exactly spectacular: All it does is show a label with the text "Hello from C++". In fact, this is the default app Android generates when you create a new project with C/C++ support - enough however to show the basic principles of how JNI calls work.




Decompile the APK with `apkx` . This extract the source code into the `HelloWorld/src` directory.

```
$ wget https://github.com/OWASP/owasp-mstg/blob/master/OMTG-
Files/03_Examples/01_Android/01_HelloWorld-JNI/HelloWord-JNI.apk
$ apkx HelloWord-JNI.apk
Extracting HelloWord-JNI.apk to HelloWord-JNI
Converting: classes.dex -> classes.jar (dex2jar)
dex2jar HelloWord-JNI/classes.dex -> HelloWord-JNI/classes.jar
```

The MainActivity is found in the file `MainActivity.java` . The "Hello World" text view is populated in the `onCreate()` method:

```
public class MainActivity
extends AppCompatActivity {
    static {
        System.loadLibrary("native-lib");
    }

    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        this.setContentView(2130968603);
        ((TextView)this.findViewById(2131427422)).setText((CharSequence)this.stringFromJNI());
    }

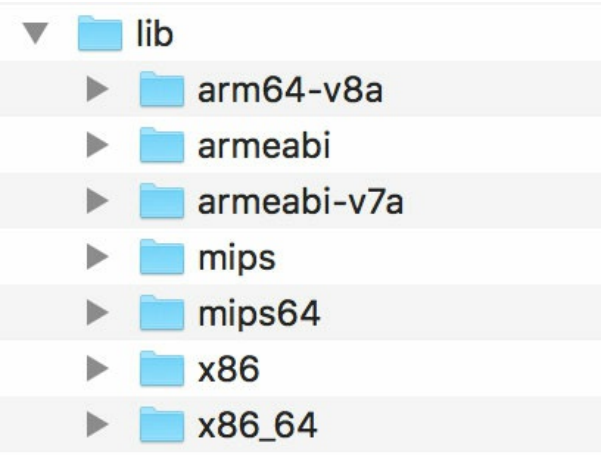
    public native String stringFromJNI();
}

}
```

Note the declaration of `public native String stringFromJNI` at the bottom. The `native` keyword informs the Java compiler that the implementation for this method is provided in a native language. The corresponding function is resolved during runtime. Of course, this only works if a native library is loaded that exports a global symbol with the expected signature. This signature is composed of the package name, class name and method name. In our case for example, this means that the programmer must have implemented the following C or C++ function:

```
JNIEXPORT jstring JNICALL Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI(JNIEnv
*env, jobject)
```

So where is the native implementation of this function? If you look into the `lib` directory of the APK archive, you'll see a total of eight subdirectories named after different processor architectures. Each of this directories contains a version of the native library `libnative-lib.so` , compiled for the processor architecture in question. When `System.loadLibrary` is called, the loader selects the correct version based on what device the app is running on.




Following the naming convention mentioned above, we can expect the library to export a symbol named `Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI`. On Linux systems, you can retrieve the list of symbols using `readelf` (included in GNU binutils) or `nm`. On Mac OS, the same can be achieved with the `greadelf` tool, which you can install via Macports or Homebrew. The following example uses `greadelf`:

```
$ greadelf -W -s libnative-lib.so | grep Java
     3: 00004e49   112 FUNC    GLOBAL DEFAULT   11
Java_sg_vantagepoint_helloworld_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.

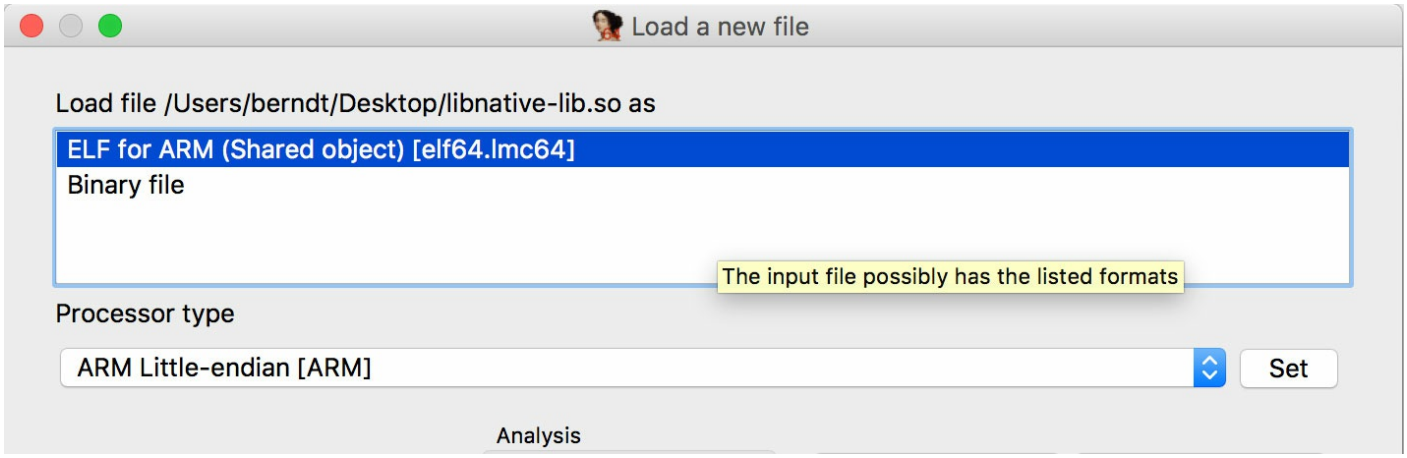
To support both older and newer ARM processors, Android apps ship with multple ARM builds compiled for different Application Binary Interface (ABI) versions. The ABI defines how the application's machine code is supposed to interact with the system at runtime. The following ABIs are supported:

- 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.

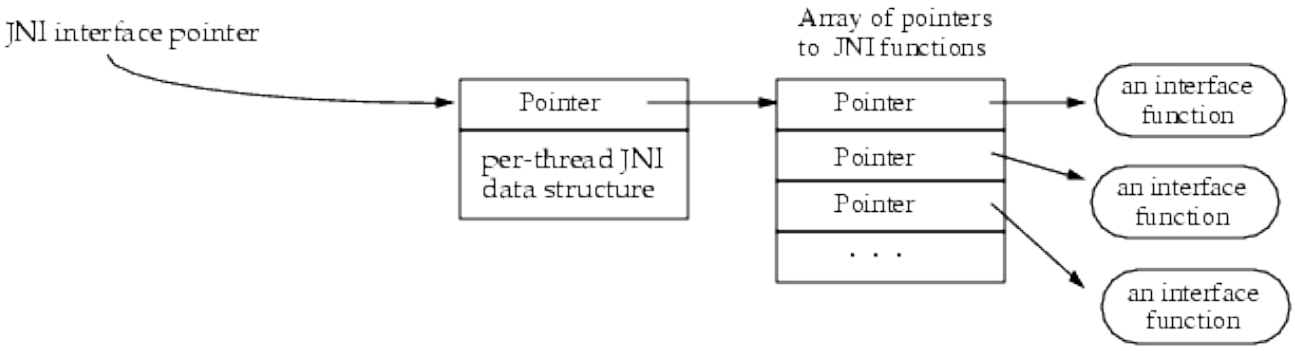
Most disassemblers will be able to deal with any of those architectures. Below, we'll be viewing the `armeabi-v7a` version in IDA Pro. It is located in `lib/armeabi-v7a/libnative-lib.so`. If you don't own an IDA Pro license, you can do the same thing with demo or evaluation version available on the Hex-Rays website [13].

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.

# Debugging and Tracing

So far, we've been using static analysis techniques without ever running our target apps. In the real world - especially when reversing more complex apps or malware - you'll find that pure static analysis is very difficult. Observing and manipulating an app during runtime makes it much, much easier to decipher its behavior. Next, we'll have a look at dynamic analysis methods that help you do just that.

Android apps support two different types of debugging: Java-runtime-level debugging using Java Debug Wire Protocol (JDWP) and Linux/Unix-style ptrace-based debugging on the native layer, both of which are valuable for reverse engineers.

### Activating Developer Options

Since Android 4.2, the "Developer options" submenu is hidden by default in the Settings app. To activate it, you need to tap the "Build number" section of the "About phone" view 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**

Dalvik and ART support the Java Debug Wire Protocol (JDWP), a protocol used for communication between the debugger and the Java virtual machine (VM) which it debugs. JDWP is a standard debugging protocol that is supported by all command line tools and Java IDEs, including JDB, JEB, IntelliJ and Eclipse. Android's implementation of JDWP also includes hooks for supporting extra features implemented by the Dalvik Debug Monitor Server (DDMS).

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

In the following section, we'll show how to solve UnCrackable App for Android Level 1 using JDB only. Note that this is not an *efficient* way to solve this crackme - you can do it much faster using Frida and other methods, which we'll introduce later in the guide. It serves however well an an introduction to the capabilities of the Java debugger.

### Repackaging

Every debugger-enabled process runs an extra thread for handling JDWP protocol packets. this thread is started only for apps that have the `android:debuggable="true"` tag set in their Manifest file's `<application>` element. This is typically the configuration on Android devices shipped to end users.

When reverse engineering apps, you'll often only have access to the release build of the target app. Release builds are not meant to be debugged - after all, that's what *debug builds* are for. If the system property `ro.debuggable` set to "0", Android disallows both JDWP and native debugging of release builds, and although this is easy to bypass, you'll still likely encounter some limitations, such as a lack of line breakpoints. Nevertheless, even an imperfect debugger is still an invaluable tool - being able to inspect the runtime state of a program makes it *a lot* easier to understand what's going on.

To "convert" a release build release into a debuggable build, you need to modify a flag in the app's Manifest file. This modification breaks the code signature, so you'll also have to re-sign the 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 `zipalign` and `apksigner` tools are found in this directory. Repackage UnCrackable-Level1.apk as follows:

1. Use apktool to unpack the app and decode AndroidManifest.xml:

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

1. Add android:debuggable = “true” to the manifest using a text editor:

```
<application android:allowBackup="true" android:debuggable="true"
android:icon="@drawable/ic_launcher" android:label="@string/app_name"
android:name="com.xxx.xxx.xxx" android:theme="@style/AppTheme">
```

1. Repackage and sign the APK.

```
$ cd UnCrackable-Level1
$ apktool b
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
$ cd ..
$ apksigner sign --ks  ~/.android/debug.keystore --ks-key-alias signkey UnCrackable-
Repackaged.apk
```

Note: If you experience JRE compatibility issues with `apksigner`, you can use `jarsigner` instead. Note that in this case, `zipalign` is called *after* signing.

```
$ jarsigner -verbose -keystore ~/.android/debug.keystore UnCrackable-Repackaged.apk signkey
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
```

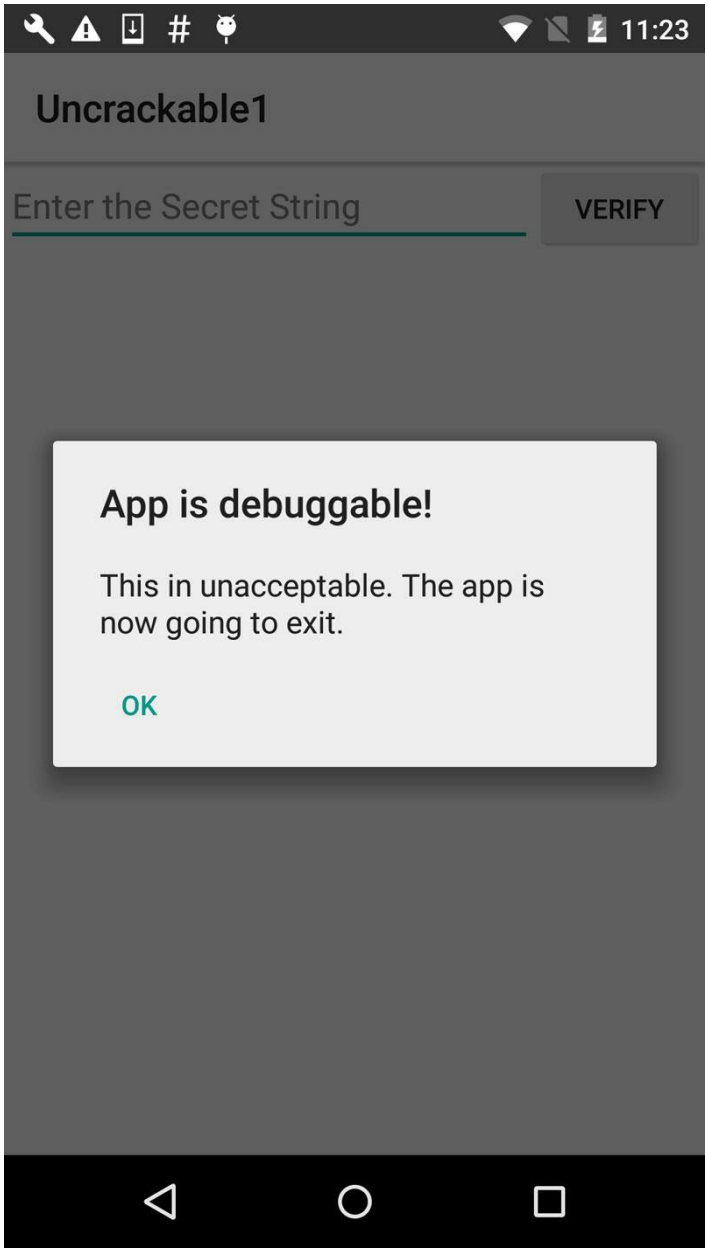
1. Reinstall the app:

```
$ adb install UnCrackable-Repackaged.apk
```

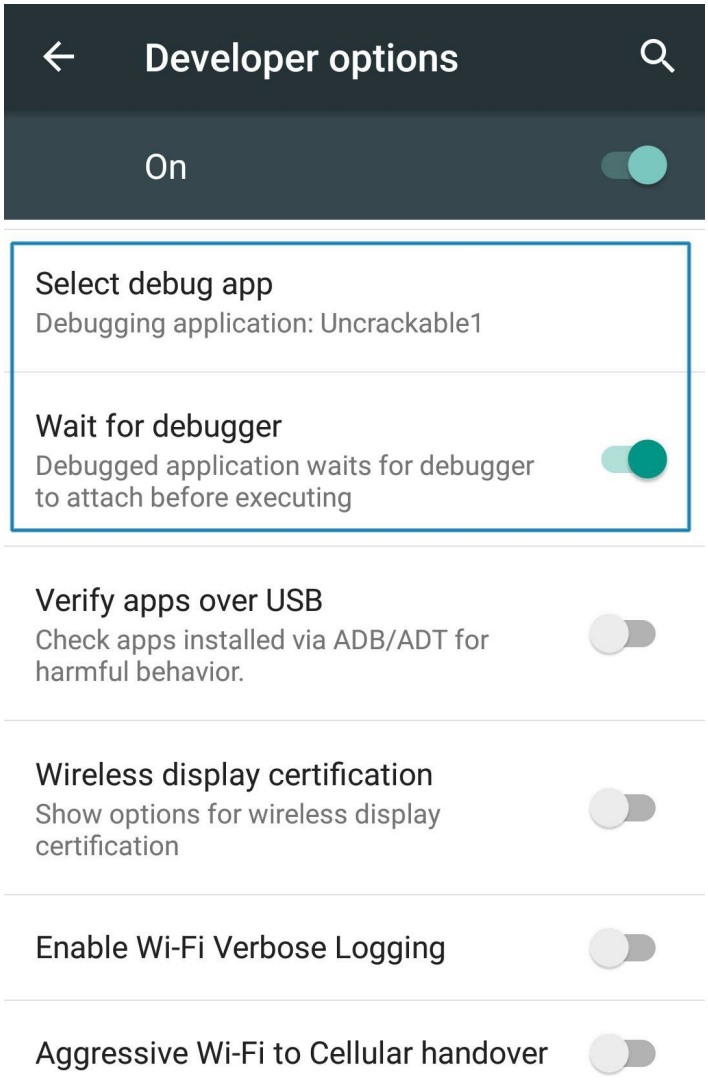
**The 'Wait For Debugger' Feature**

UnCrackable App is not stupid: It notices that it has been run in debuggable mode and reacts by shutting down. A modal dialog is shown immediately and the crackme terminates once you tap the OK button.

Fortunately, Android's Developer options contain the useful "Wait for Debugger" feature, which allows you to automatically suspend a selected app doing startup until a JDWP debugger connects. By using this feature, you can connect the debugger before the detection mechanism runs, and trace, debug and deactivate that mechanism. It's really an unfair advantage, but on the other hand, reverse engineers never play fair!




In the Developer Settings, pick `Uncrackable1` as the debugging application and activate the "Wait for Debugger" switch.




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

### The Android Debug Bridge

The `adb` command line tool, which ships with the Android SDK, bridges the gap between your local development environment and a connected Android device. Commonly you'll debug apps on the emulator or on a device connected via USB. Use the `adb devices` command to list the currently connected devices.

```
$ adb devices
List of devices attached
090c285c0b97f748  device
```

The `adb jdwp` command lists the process ids of all debuggable processes running on the connected device (i.e., processes hosting a JDWP transport). With the `adb forward` command, you can open a listening socket on your host machine and forward TCP connections to this socket to the JDWP transport of a chosen process.

```
$ adb jdwp
12167
$ adb forward tcp:7777 jdwp:12167
```

We're now ready to attach JDB. Attaching the debugger however causes the app to resume, which is something we don't want. Rather, we'd like to keep it suspended so we can do some exploration first. To prevent the process from resuming, we pipe the `suspend` command into jdb:

```
$ { echo "suspend"; cat; } | jdb -attach localhost:7777

Initializing jdb ...
> All threads suspended.
>
```

We are now attached to the suspended process and ready to go ahead with jdb commands. Entering `?` prints the complete list of. Unfortunately, the Android VM doesn't support all available JDWP features. For example, the `redefine` command, which would let us redefine the code for a class - a potentially very useful feature - is not supported. Another important restriction is that line breakpoints won't work, because the release bytecode doesn't contain line information. Method breakpoints do work however. Useful commands that work include:

- classes: List all loaded classes
- class / method / fields : Print details about a class and list its method and fields
- locals: print local variables in current stack frame
- print / dump : print information about an object
- stop in : set a method breakpoint
- clear : remove a method breakpoint
- set = : assign new value to field/variable/array element

Let's revisit the decompiled code of UnCrackable App Level 1 and think about possible solutions. A

good approach would be to suspend the app at a state where the secret string is stored in a variable in plain text so we can retrieve it. Unfortunately, we won't get that far unless we deal with the root / tampering detection first.

By reviewing the code, we can gather that the method `sg.vantagepoint.uncrackable1.MainActivity.a` is responsible for displaying the "This in unacceptable..." message box. This method hooks the "OK" button to a class that implements the `OnClickListener` interface. The `onClick` event handler on the "OK" button is what actually terminates the app. To prevent the user from simply cancelling the dialog, the `setCancelable` method is called.

```
private void a(final String title) {
      final AlertDialog create = new AlertDialog$Builder((Context)this).create();
      create.setTitle((CharSequence)title);
      create.setMessage((CharSequence)"This in unacceptable. The app is now going to exit.");
      create.setButton(-3, (CharSequence)"OK", (DialogInterface$OnClickListener)new b(this));
      create.setCancelable(false);
      create.show();
  }
```

We can bypass this with a little runtime tampering. With the app still suspended, set a method breakpoint on `android.app.Dialog.setCancelable` and resume the app.

```
> stop in android.app.Dialog.setCancelable
Set breakpoint android.app.Dialog.setCancelable
> resume
All threads resumed.
>
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1]
```

The app is now suspended at the first instruction of the `setCancelable` method. You can print the arguments passed to `setCancelable` using the `locals` command (note that the arguments are incorrectly shown under "local variables").

```
main[1] locals
Method arguments:
Local variables:
flag = true
```

In this case, `setCancelable(true)` was called, so this can't be the call we're looking for. Resume the process using the `resume` command.

```
main[1] resume
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1] locals
flag = false
```

We've now hit a call to `setCancelable` with the argument `false` . Set the variable to `true` with the `set` command and resume.

```
main[1] set flag = true
 flag = true = true
main[1] resume
```

Repeat this process, setting `flag` to `true` each time the breakpoint is hit, until the alert box is finally displayed (the breakpoint will hit 5 or 6 times). The alert box should now be cancelable! Tap anywhere next to the box and it will close without terminating the app.

Now that the anti-tampering is out of the way we're ready to extract the secret string! In the "static analysis" section, we saw that the string is decrypted using AES, and then compared with the string entered into the messagebox. The method `equals` of the `java.lang.String` class is used to compare the input string with the secret. Set a method breakpoint on `java.lang.String.equals` , enter any text into the edit field, and tap the "verify" button. Once the breakpoint hits, you can read the method argument with the using the `locals` command.

```
> stop in java.lang.String.equals
Set breakpoint java.lang.String.equals
>
Breakpoint hit: "thread=main", java.lang.String.equals(), line=639 bci=2

main[1] locals
Method arguments:
Local variables:
other = "radiusGravity"
main[1] cont

Breakpoint hit: "thread=main", java.lang.String.equals(), line=639 bci=2

main[1] locals
Method arguments:
Local variables:
other = "I want to believe"
main[1] cont
```

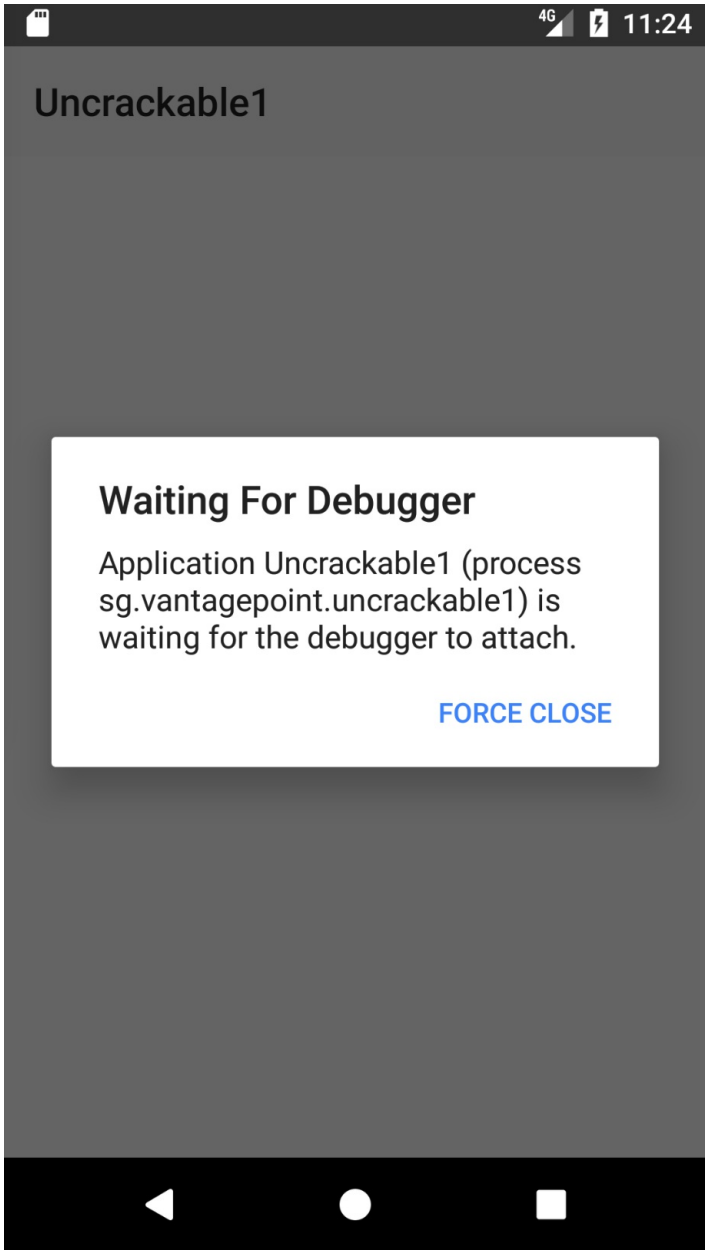
This is the plaintext string we are looking for!

### Debugging Using an IDE

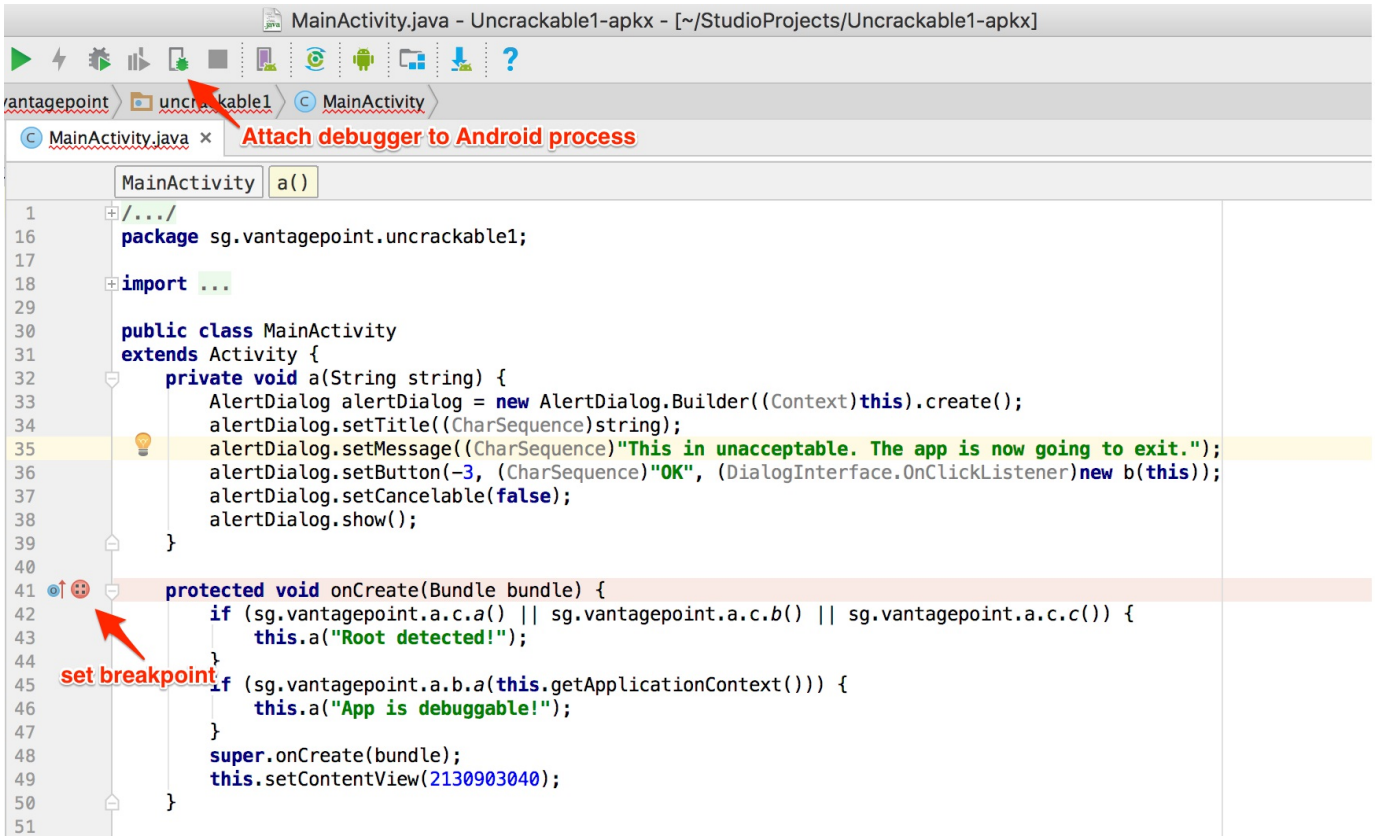
A pretty neat trick is setting up a project in an IDE with the decompiled sources, which allows you to set method breakpoints directly in the source code. In most cases, you should be able single-step through the app, and inspect the state of variables through the GUI. The experience won't be perfect - it's not the original source code after all, so you can't set line breakpoints and sometimes things will simply not work correctly. Then again, reversing code is never easy, and being able to efficiently navigate and debug plain old Java code is a pretty convenient way of doing it, so it's usually worth giving it a shot. A similar method has been described in the NetSPI blog [18].

In order to debug an app from the decompiled source code, you should first create your Android project and copy the decompiled java sources into the source folder as described above at "Statically Analyzing Java Code" part. Set the debug app (in this tutorial it is Uncrackable1) and make sure you turned on "Wait For Deugger" switch from "Developer Options".

Once you tap the Uncrackable app icon from the launcher, it will get suspended in "wait for a debugger" mode.




Now you can set breakpoints and attach to the Uncrackable1 app process using the "Attach Debugger" button on the toolbar.



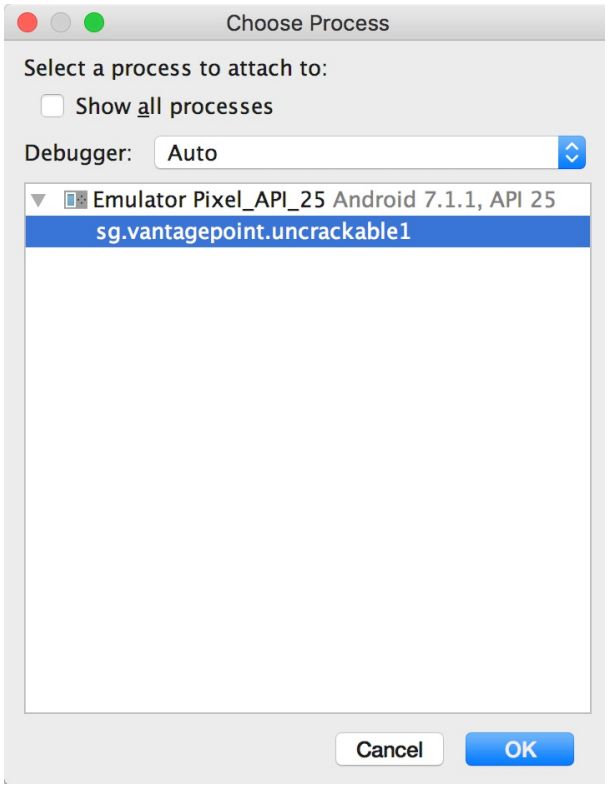

```
/.../
package sg.vantagepoint.uncrackable1;

import ...

public class MainActivity
extends Activity {
    private void a(String string) {
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
        alertDialog.setTitle((CharSequence)string);
        alertDialog.setMessage((CharSequence)"This in unacceptable. The app is now going to exit.");
        alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
        alertDialog.setCancelable(false);
        alertDialog.show();
    }

    protected void onCreate(Bundle bundle) {
        if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
            this.a("Root detected!");
        }
        if (sg.vantagepoint.a.b.a(this.getApplicationContext())) {
            this.a("App is debuggable!");
        }
        super.onCreate(bundle);
        this.setContentView(2130903040);
    }
```

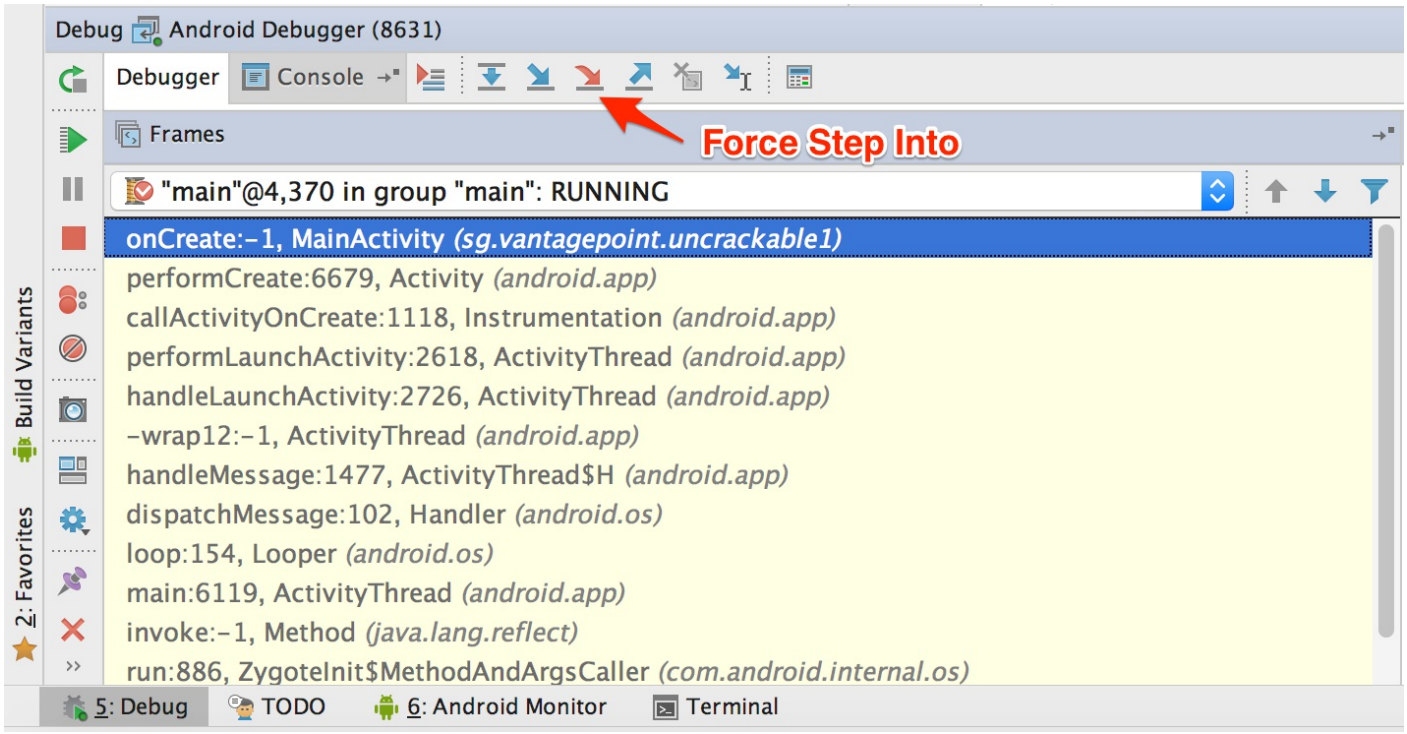
Note that only method breakpoints work when debugging an app from decompiled sources. Once a method breakpoint is hit, you will get the chance to single step throughout the method execution.




After you choose the Uncrackable1 application from the list, the debugger will attach to the app process and you will hit the breakpoint that was set on the `onCreate()` method. Uncrackable1 app triggers anti-debugging and anti-tampering controls within the `onCreate()` method. That's why it is a

good idea to set a breakpoint on the `onCreate()` method just before the anti-tampering and anti-debugging checks performed.

Next, we will single-step through the `onCreate()` method by clicking the "Force Step Into" button on the Debugger view. The "Force Step Into" option allows you to debug the Android framework functions and core Java classes that are normally ignored by debuggers.




Once you "Force Step Into", the debugger will stop at the beginning of the next method which is the `a()` method of class `sg.vantagepoint.a.c` .

```
vantagepoint > a > c
MainActivity.java × | c.java ×
c | a()
```

```
1   /.../
7   package sg.vantagepoint.a;
8
9   import android.os.Build;
10  import java.io.File;
11
12  public class c {
13      /*
14       * Enabled force condition propagation
15       * Lifted jumps to return sites
16       */
17      public static boolean a() {
18          boolean bl = false;
19          String[] arrstring = System.getenv("PATH").split(":");
20          int n = arrstring.length;
21          int n2 = 0;
22          do {
23              boolean bl2 = bl;
24              if (n2 >= n) return bl2;
25              if (new File(arrstring[n2], "su").exists()) {
26                  return true;
27              }
28              ++n2;
29          } while (true);
30      }
31
```

This method searches for "su" binary within well known directories. Since we are running the app on a rooted device/emulator we need to defeat this check by manipulating variables and/or function return values.

```
Variables
▶ System.getenv(String) = "/sbin:/vendor/bin:/system/sbin:/system/bin:/system/xbin"
```

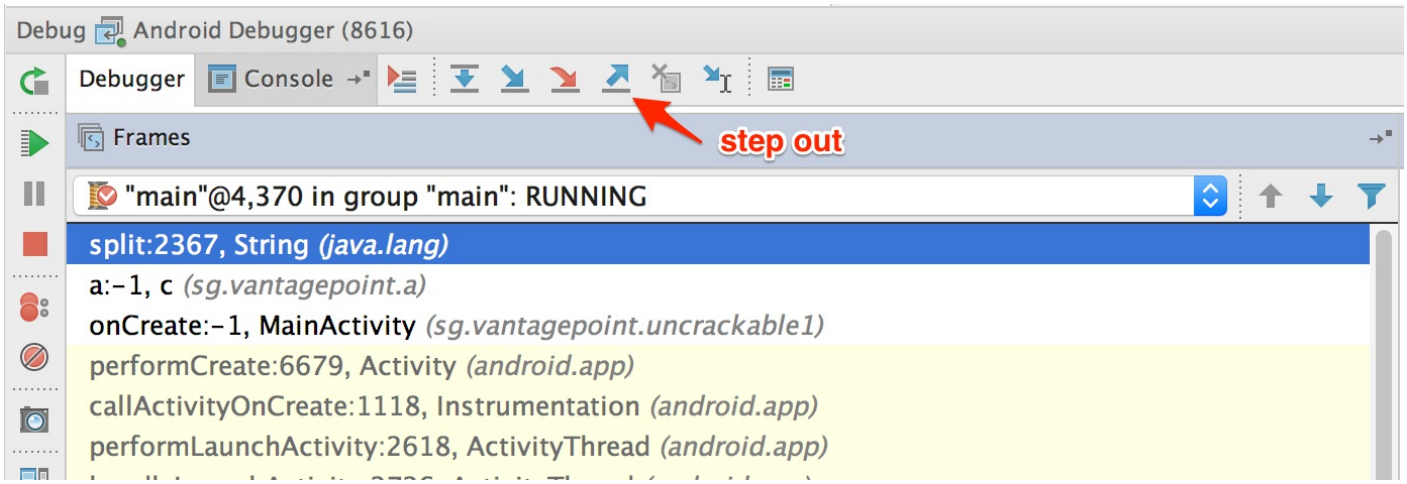
You can see the directory names inside the "Variables" window by stepping into the `a()` method and stepping through the method by clicking "Step Over" button in the Debugger view.

```
Debug  Android Debugger (8616)
Debugger | Console
Frames                                   step over
"main"@4,370 in group "main": RUNNING
a:-1, c (sg.vantagepoint.a)
onCreate:-1, MainActivity (sg.vantagepoint.uncrackable1)
performCreate:6679, Activity (android.app)
callActivityOnCreate:1118, Instrumentation (android.app)
```

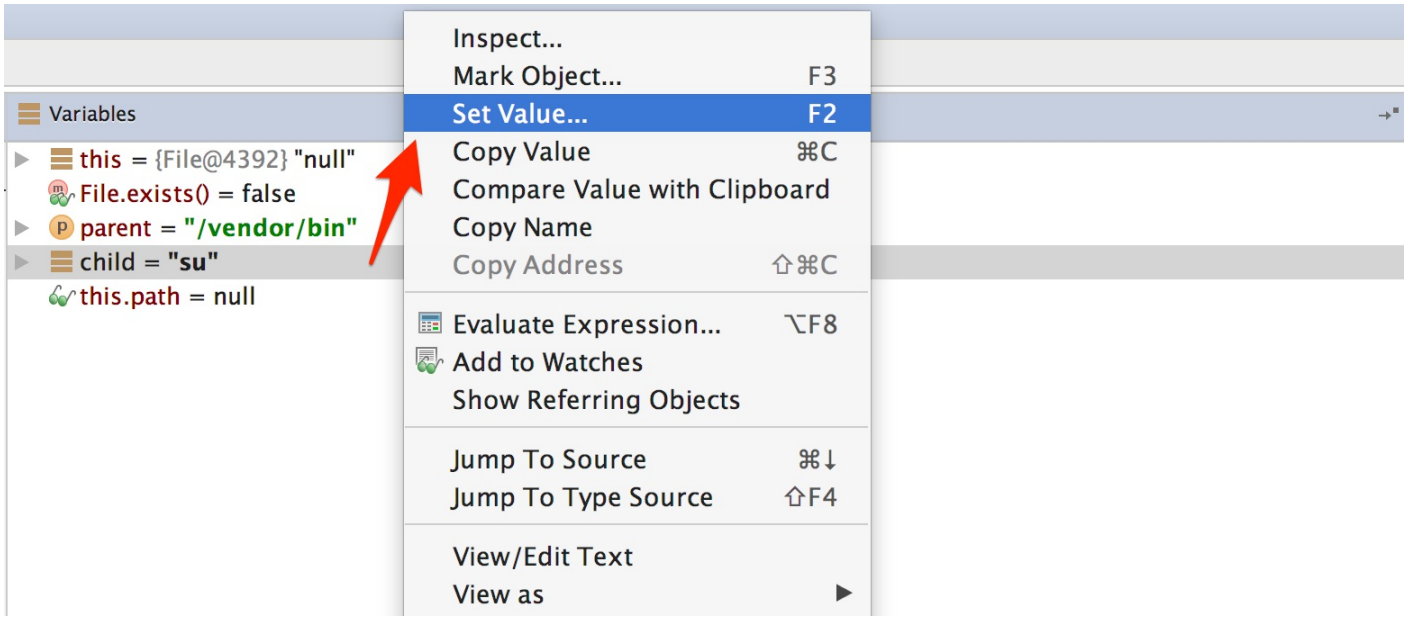
Step into the `System.getenv` method call y using the "Force Step Into" functionality.

After you get the colon seperated directory names, the debugger cursor will return back to the beginning of `a()` method; not to the next executable line. This is just because we are working on the decompiled code insted of the original source code. So it is crucial for the analyst to follow the code flow while debugging decompiled applications. Otherwise, it might get complicated to identify which line will be executed next.

If you don't want to debug core Java and Android classes, you can step out of the function by clicking "Step Out" button in the Debugger view. It might be a good approach to "Force Step Into" once you reach the decompiled sources and "Step Out" of the core Java and Android classes. This will help you to speed up your debugging while keeping eye on the return values of the core class functions.

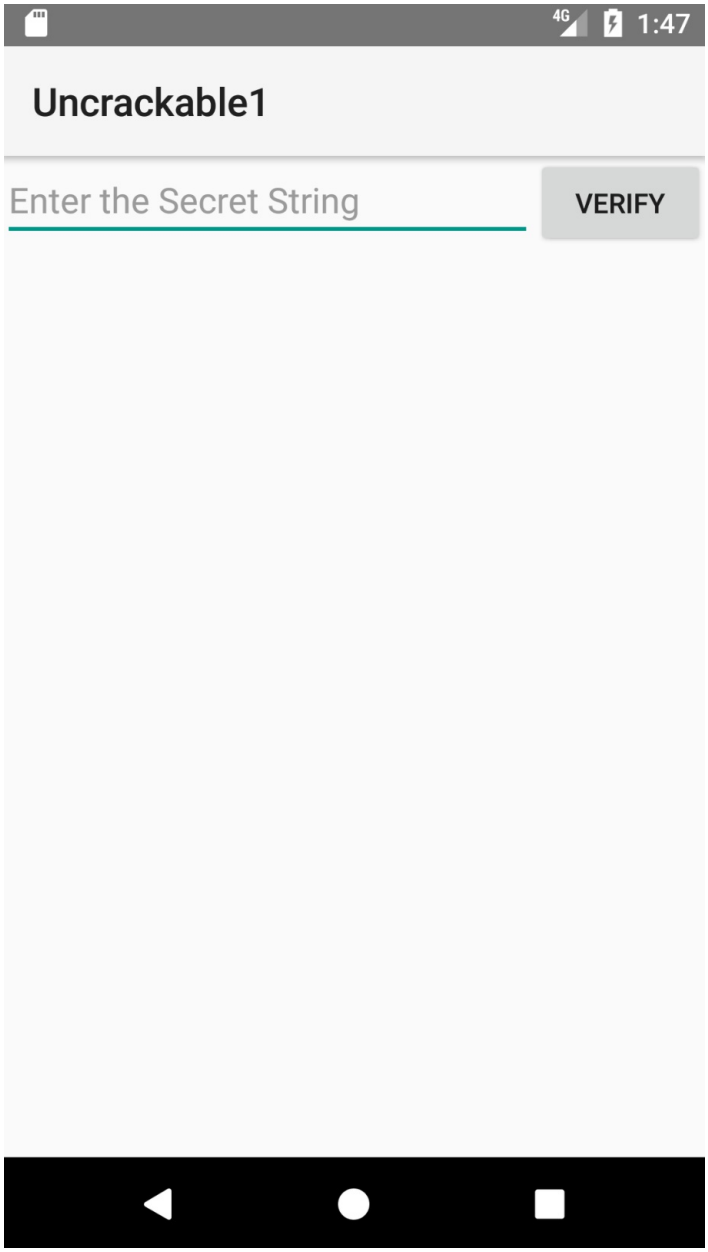



After it gets the directory names, `a()` method will search for the existence of the </code>su</code> binary within these directories. In order to defeat this control, you can modify the directory names (parent) or file name (child) at cycle which would otherwise detect the `su` binary on your device. You can modify the variable content by pressing F2 or Right-Click and "Set Value".




Once you modify the binary name or the directory name, `File.exists` should return `false` .


This defeats the first root detection control of Uncrackable App Level 1. The remaining anti-tampering and anti-debugging controls can be defeated in similar ways to finally reach secret string verification functionality.



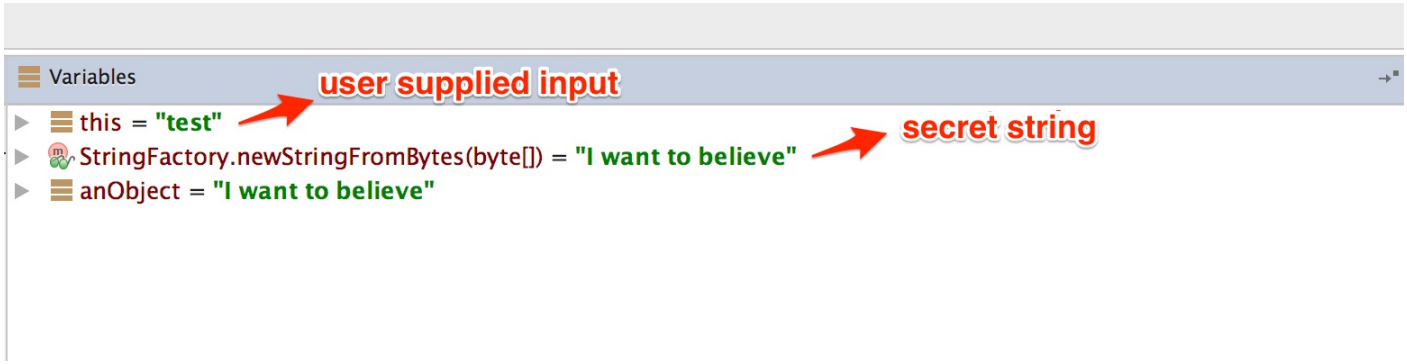

```
/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (a.a((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
    } else {
        alertDialog.setTitle((CharSequence)"Nope...");
        alertDialog.setMessage((CharSequence)"That's not it. Try again.");
    }
    alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}
```

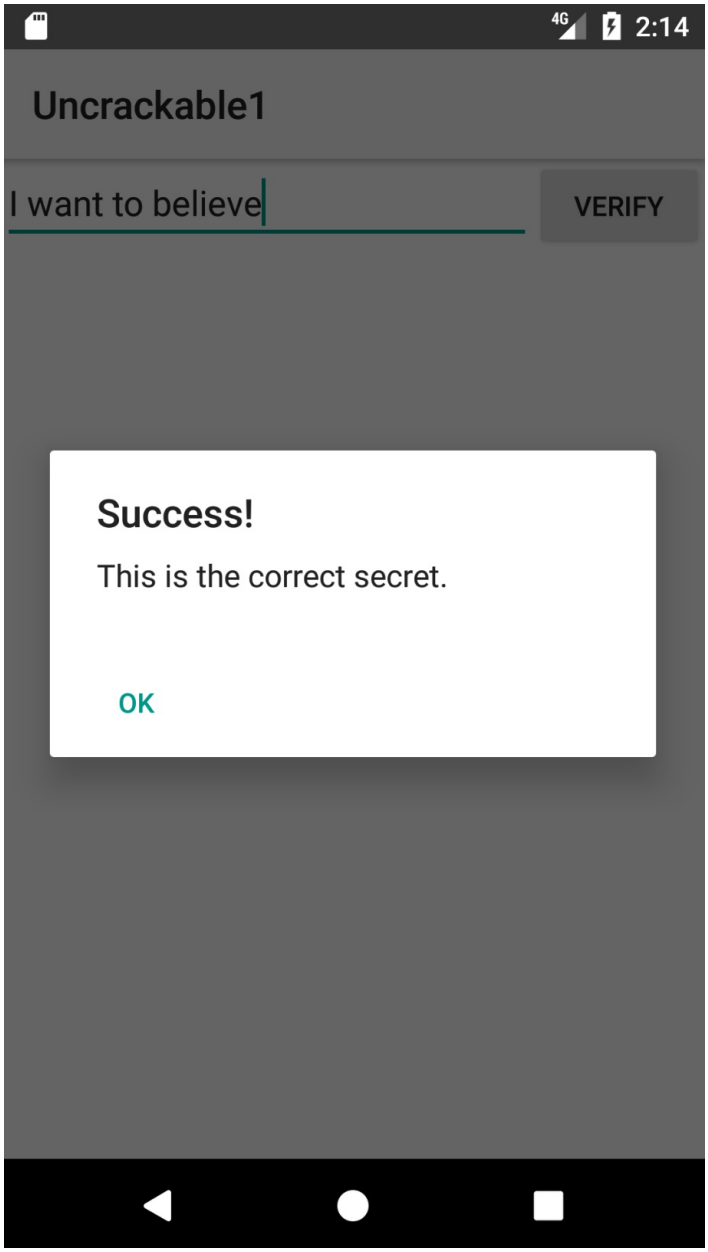
The secret code is verified by the method `a()` of class `sg.vantagepoint.uncrackable1.a` . Set a breakpoint on method `a()` and "Force Step Into" when you hit the breakpoint. Then, single-step until you reach the call to `String.equals` . This is where user supplied input is compared with the secret string.

```
public static boolean a(String string) {
    byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);
    byte[] arrby2 = new byte[]{};
    try {
        arrby = sg.vantagepoint.a.a.a(a.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        arrby2 = arrby;
    }
    catch (Exception exception) {
        Log.d((String)"CodeCheck", (String)("AES error:" + exception.getMessage()));
    }
    if (string.equals(new String(arrby2))) {
        return true;
    }
    return false;
}
```

set breakpoint

You can see the secret string in the "Variables" view at the time you reach the `String.equals` method call.






### Debugging Native Code

Native code on Android is packed into ELF shared libraries and runs just like any other native Linux program. Consequently, you can debug them using standard tools, including GDB and the built-in native debuggers of IDEs such as IDA Pro and JEB, as long as they support the processor architecture of the device (most devices are based on ARM chipsets, so this is usually not an issue).

We'll now set up our JNI demo app, HelloWorld-JNI.apk, for debugging. It's the same APK you downloaded in "Statically Analyzing Native Code". Use `adb install` to install it on your device or on an emulator.

```
$ adb install HelloWorld-JNI.apk
```

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

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

The `gdbserver --attach<comm> <pid>` command causes gdbserver to attach to the running process and bind to the IP address and port specified in `comm` , which in our case is a `HOST:PORT` descriptor. Start HelloWorld-JNI on the device, then connect to the device and determine the PID of the HelloWorld process. Then, switch to the root user and attach `gdbserver` as follows.

```
$ adb shell
$ ps | grep helloworld
u0_a164   12690 201   1533400 51692 ffffffff 00000000 S sg.vantagepoint.helloworldjni
$ su
# /data/local/tmp/gdbserver --attach localhost:1234 12690
Attached; pid = 12690
Listening on port 1234
```

The process is now suspended, and `gdbserver` listening for debugging clients on port `1234` . With the device connected via USB, you can forward this port to a local port on the host using the `abd forward` command:

```
$ adb forward tcp:1234 tcp:1234
```

We'll now use the prebuilt version of `gdb` contained in the NDK toolchain (if you haven't already, follow the instructions above to install it).

```
$ $TOOLCHAIN/bin/gdb libnative-lib.so
GNU gdb (GDB) 7.11
(...)
Reading symbols from libnative-lib.so...(no debugging symbols found)...done.
(gdb) target remote :1234
Remote debugging using :1234
0xb6e0f124 in ?? ()
```

We have successfully attached to the process! The only problem is that at this point, we're already too late to debug the JNI function `StringFromJNI()` as it only runs once at startup. We can again solve this problem by activating the "Wait for Debugger" option. Go to "Developer Options" -> "Select debug app" and pick HelloWorldJNI, then activate the "Wait for debugger" switch. Then, terminate and re-launch the app. It should be suspended automatically.

Our objective is to set a breakpoint at the start of the native function `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI()` before resuming the app. Unfortunately, this isn't possible at this early point in execution because `libnative-lib.so` isn't yet mapped into process memory - it is loaded dynamically during runtime. To get this working, we'll first use JDB to gently control the process into the state we need.

First, we resume execution of the Java VM by attaching JDB. We don't want the process to resume immediately though, so we pipe the `suspend` command into JDB as follows:

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

Next, we want to suspend the process at the point the Java runtime loads `libnative-lib.so` . In JDB, set a breakpoint on the `java.lang.System.loadLibrary()` method and resume the process. After the breakpoint has been hit, execute the `step up` command, which will resume the process until `loadLibrary()` returns. At this point, `libnative-lib.so` has been loaded.

> stop in java.lang.System.loadLibrary resume All threads resumed. Breakpoint hit: "thread=main", java.lang.System.loadLibrary(), line=988 bci=0 step up main[1] step up
>
> Step completed: "thread=main", sg.vantagepoint.helloworldjni.MainActivity.(), line=12 bci=5

main[1]

```
Execute <code>gdbserver</code> to attach to the suspended app. This will have the effect that
the app is "double-suspended" by both the Java VM and the Linux kernel.


```bash
$ adb forward tcp:1234 tcp:1234
$ $TOOLCHAIN/arm-linux-androideabi-gdb libnative-lib.so
GNU gdb (GDB) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
(...)
(gdb) target remote :1234
Remote debugging using :1234
0xb6de83b8 in ?? ()
```

Execute the `resume` command in JDB to resume execution of the Java runtime (we're done using JDB, so you can also detach it at this point). You can start exploring the process with GDB. The `info sharedlibrary` command displays the loaded libraries, which should include `libnative-lib.so` . The `info functions` command retrieves a list of all known functions. The JNI function `java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI()` should be listed as a non-debugging symbol. Set a breakpoint at the address of that function and resume the process.

```
(gdb) info sharedlibrary
(...)
0xa3522e3c  0xa3523c90  Yes (*)     libnative-lib.so
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x00000e78  Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
(...)
0xa3522e78  Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
(...)
(gdb) b *0xa3522e78
Breakpoint 1 at 0xa3522e78
(gdb) cont
```
```

Your breakpoint should be hit when the first instruction of the JNI function is executed. You can now display a disassembly of the function using the `disassemble` command.

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

From here on, you can single-step through the program, print the contents of registers and memory, or tamper with them, to explore the inner workings of the JNI function (which, in this case, simply returns a string). Use the `help` command to get more information on debugging, running and examining data.

**Execution Tracing**

Besides being useful for debugging, the JDB command line tool also offers basic execution tracing functionality. To trace an app right from the start we can pause the app using the Android "Wait for Debugger" feature or a `kill -STOP` command and attach JDB to set a deferred method breakpoint on an initialization method of our choice. Once the breakpoint hits, we activate method tracing with the `trace go methods` command and resume execution. JDB will dump all method entries and exits from that point on.

```
$ adb forward tcp:7777 jdwp:7288
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> All threads suspended.
> stop in com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()
Deferring breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>().
It will be set after the class is loaded.
> resume
All threads resumed.M
Set deferred breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()

Breakpoint hit: "thread=main", com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>
(), line=44 bci=0
main[1] trace go methods
main[1] resume
Method entered: All threads resumed.
```

The Dalvik Debug Monitor Server (DDMS) a GUI tool included with Android Studio. At first glance it might not look like much, but make no mistake: Its Java method tracer is one of the most awesome tools you can have in your arsenal, and is indispensable for analyzing obfuscated bytecode.

Using DDMS is a bit confusing however: It can be launched in several ways, and different trace viewers will be launched depending on how the trace was obtained. There's a standalone tool called "Traceview" as well as a built-in viewer in Android Studio, both of which offer different ways of navigating the trace. You'll usually want to use the viewer built into Android studio which gives you a nice, zoom-able hierarchical timeline of all method calls. The standalone tool however is also useful, as it has a profile panel that shows the time spent in each method, as well as the parents and children of each method.

To record an execution trace in Android studio, open the "Android" tab at the bottom of the GUI. Select the target process in the list and 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 the `ptrace()` system call to attach to the target process, so it only works up to the point that anti-debugging measures kick in.

As a side note, if the Android "stop application at startup: feature is unavailable we can use a shell script to make sure that strace attached immediately once the process is launched (not an elegant solution but it works):

```
$ while true; do pid=$(pgrep 'target_process' | head -1); if [[ -n "$pid" ]]; then strace -s
2000 - e “!read” -ff -p "$pid"; break; fi; done
```

### Ftrace

Ftrace is a tracing utility built directly into the Linux kernel. On a rooted device, ftrace can be used to trace kernel system calls in a more transparent way than is possible with strace, which relies on the ptrace system call to attach to the target process.

Conveniently, ftrace functionality is found in the stock Android kernel on both Lollipop and Marshmallow. It can be enabled with the following command:

```
$ echo 1 > /proc/sys/kernel/ftrace_enabled
```

The `/sys/kernel/debug/tracing` directory holds all control and output files and related to ftrace. The following files are found in this directory:

- available_tracers: This file lists the available tracers compiled into the kernel.
- current_tracer: This file is used to set or display the current tracer.
- tracing_on: Echo 1 into this file to allow/start update of the ring buffer. Echoing 0 will prevent further writes into the ring buffer.

### KProbes

The KProbes interface provides us with an even more powerful way to instrument the kernel: It allows us to insert probes into (almost) arbitrary code addresses within kernel memory. Kprobes work by inserting a breakpoint instruction at the specified address. Once the breakpoint is hit, control passes to the Kprobes system, which then executes the handler function(s) defined by the user as well as the original instruction. Besides being great for function tracing, KProbes can be used to implement rootkit-like functionality such as file hiding.

Jprobes and Kretprobes are additional probe types based on Kprobes that allow hooking of function entries and exits.

Unfortunately, the stock Android kernel comes without loadable module support, which is a problem given that Kprobes are usually deployed as kernel modules. Another issue is that the Android kernel is compiled with strict memory protection which prevents patching some parts of Kernel memory.

[16]</code> results in a Kernel panic on default

Using Elfmaster's system call hooking method [16]</code> 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 - an extension to the DECAF dynamic analysis framework [20] - 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 [21] 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 [22]. 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 [23] 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: Disabling SSL Pinning**

Certificate pinning is an issue for security testers who want to intercepts HTTPS communiction for legitimate reasons. To help with this problem, the bytecode can be patched to deactivate SSL pinning. To demonstrate how Certificate Pinning can be bypassed, we will walk through the necessary steps to bypass Certificate Pinning implemented in an example application.

The first step is to disassemble the APK with `apktool` :

```
$ apktool d target_apk.apk
```

You then need to locate the certificate pinning checks in the Smali source code. Searching the smali code for keywords such as “X509TrustManager” should point you in the right direction.

In our example, a search for "X509TrustManager" returns one class that implements a custom Trustmanager. The derived class implements methods named `checkClientTrusted` , `checkServerTrusted` and `getAcceptedIssuers` .

Insert the `return-void` opcode was added to the first line of each of these methods to bypass execution. This causes each method to return immediately. return value. With this modification, no certificate checks are performed, and the application will accept all certificates.

```
.method public checkServerTrusted([LJava/security/cert/X509Certificate;Ljava/lang/String;)V
  .locals 3
  .param p1, "chain"  # [Ljava/security/cert/X509Certificate;
  .param p2, "authType"   # Ljava/lang/String;

  .prologue
  return-void      # <-- OUR INSERTED OPCODE!
  .line 102
  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" [24]. Technically, it is an extended version of Zygote that exports APIs for running Java code when a new process is started. By running Java code in the context of the newly instantiated app, it is possible to resolve, hook and override Java methods belonging to the app. Xposed uses reflection to examine and modify the running app. Changes are applied in memory and persist only during the runtime of the process - no patches to the application files are made.

To use Xposed, you first need to install the Xposed framework on a rooted device. Modifications are then deployed in the form of separate apps ("modules") that can be toggled on and off in the Xposed GUI.

### Example: Bypassing Root Detection with XPosed

Let's assume you're testing an app that is stubbornly quitting on your rooted device. You decompile the app and find the following highly suspect method:

```
package com.example.a.b

public static boolean c() {
  int v3 = 0;
  boolean v0 = false;

  String[] v1 = new String[]{"/sbin/", "/system/bin/", "/system/xbin/", "/data/local/xbin/",
    "/data/local/bin/", "/system/sd/xbin/", "/system/bin/failsafe/", "/data/local/"};

    int v2 = v1.length;

    for(int v3 = 0; v3 < v2; v3++) {
      if(new File(String.valueOf(v1[v3]) + "su").exists()) {
         v0 = true;
         return v0;
      }
    }

    return v0;
}
```

This method iterates through a list of directories, and returns "true" (device rooted) if the `su` binary is found in any of them. Checks like this are easy to deactivate - all you have to do is to replace the code with something that returns "false". Methok hooking using an Xposed module is one way to do this.

This method `XposedHelpers.findAndHookMethodfindAndHookMethod` allows you to override existing class methods. From the decompiled code, we know that the method performing the check is called `c()` and located in the class `com.example.a.b` . An Xposed module that overrides the function to always return "false" looks as follows.

```
package com.awesome.pentestcompany;

import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
import de.robv.android.xposed.IXposedHookLoadPackage;
import de.robv.android.xposed.XposedBridge;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class DisableRootCheck implements IXposedHookLoadPackage {

    public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
        if (!lpparam.packageName.equals("com.example.targetapp"))
            return;

        findAndHookMethod("com.example.a.b", lpparam.classLoader, "c", new XC_MethodHook() {
            @Override

            protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
                XposedBridge.log("Caught root check!");
                param.setResult(false);
            }

        });
    }
}
```

Modules for Xposed are developed and deployed with Android Studio just like regular Android apps. For more details on writing compiling and installing Xposed modules, refer to the tutorial provided by its author, rovo89 [24].

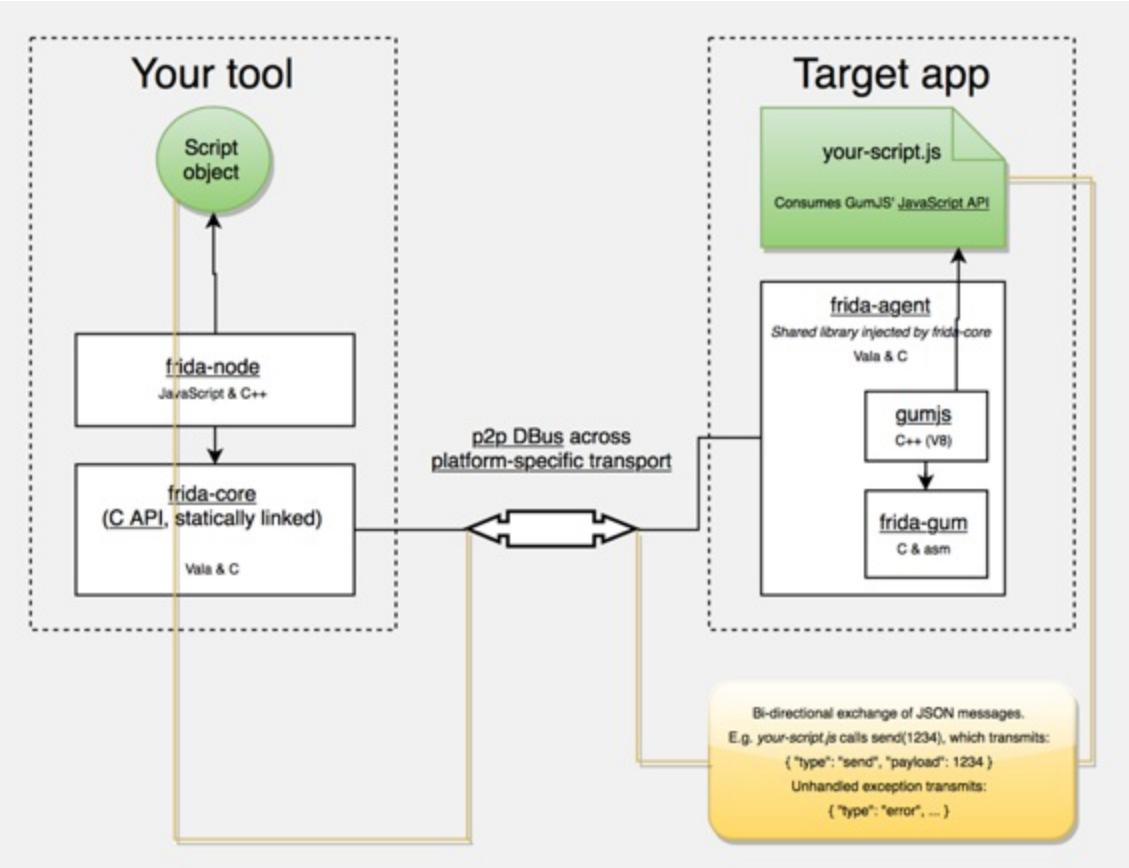
# Dynamic Instrumentation with FRIDA

Frida "lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, Linux, iOS, Android, and QNX" [26]. While it was originally based on Google's V8 Javascript runtime, since version 9 Frida now uses Duktape internally.

Code injection can be achieved in different ways. For example, Xposed makes some permanent modifications to the Android app loader that provide hooks to run your own code every time a new process is started. In contrast, Frida achieves code injection by writing code directly into process memory. The process is outlined in a bit more detail below.

When you "attach" Frida to a running app, it uses ptrace to hijack a thread in a running process. This thread is used to allocate a chunk of memory and populate it with a mini-bootstrapper. The bootstrapper starts a fresh thread, connects to the Frida debugging server running on the device, and loads a dynamically generated library file containing the Frida agent and instrumentation code. The original, hijacked thread is restored to its original state and resumed, and execution of the process continues as usual.

Frida injects a complete JavaScript runtime into the process, along with a powerful API that provides a wealth of useful functionality, including calling and hooking of native functions and injecting structured data into memory. It also supports interaction with the Android Java runtime, such as interacting with objects inside the VM.




*FRIDA Architecture, source: http://www.frida.re/docs/hacking/*

Here are some more APIs FRIDA offers on Android:

- Instantiate Java objects and call static and non-static class methods;
- Replace Java method implementations;
- Enumerate live instances of specific classes by scanning the Java heap (Dalvik only);
- Scan process memory for occurrences of a string;
- Intercept native function calls to run your own code at function entry and exit.

Some features unfortunately don't work yet on current Android devices platforms. Most notably, the FRIDA Stalker - a code tracing engine based on dynamic recompilation - does not support ARM at the time of this writing (version 7.2.0). Also, support for ART has been included only recently, so the Dalvik runtime is still better supported.

**Installing Frida**

To install Frida locally, simply use Pypi:

```
$ sudo pip install frida
```

Your Android device doesn't need to be rooted to get Frida running, but it's the easiest setup and we assume a rooted device here unless noted otherwise. Download the frida-server binary from the Frida releases page. Make sure that the server version (at least the major version number) matches the version of your local Frida installation. Usually, Pypi will install the latest version of Frida, but if you are not sure, you can check with the Frida command line tool:

```
$ frida --version
9.1.10
$ wget https://github.com/frida/frida/releases/download/9.1.10/frida-server-9.1.10-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.

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:

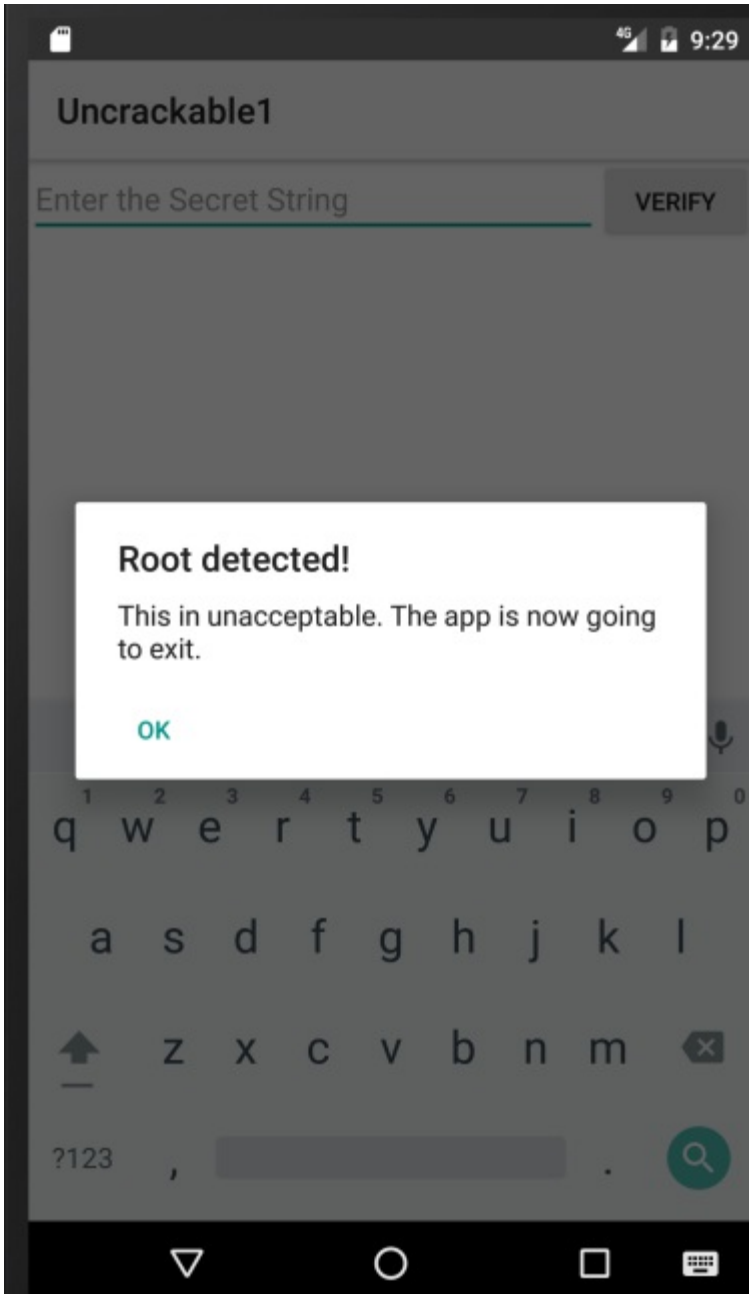
```
Java.perform(function () {
    var view = Java.use("android.view.View");
    var methods = view.class.getMethods();
    for(var i = 0; i < methods.length; i++) {
        console.log(methods[i].toString());
    }
});
```

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.

We've now covered the basics of static/dynamic analysis on Android. Of course, the only way to *really* learn it is hands-on experience: Start by building your own projects in Android Studio and observing how your code gets translated to bytecode and native code, and have a shot at our cracking challenges.

In the remaining sections, we'll introduce a few advanced subjects including kernel modules and dynamic execution.

# Binary Analysis Frameworks

Binary analysis frameworks provide you powerful ways of automating tasks that would be almost impossible to complete manually. In the section, we'll have a look at the Angr framework, a python framework for analyzing binaries that is useful for both static and dynamic symbolic ("concolic") analysis. Angr operates on the VEX intermediate language, and comes with a loader for ELF/ARM binaries, so it is perfect for dealing with native Android binaries.

Our target program is a simple license key validation program. Granted, you won't usually find a license key validator like this in the wild, but it should be useful enough to demonstrate the basics of static/symbolic analysis of native code. You can use the same techniques on Android apps that ship with obfuscated native libraries (in fact, obfuscated code is often put into native libraries, precisely to make de-obfuscation more difficult).

# Installing Angr

Angr is written in Python 2 and available from PyPI. It is easy to install on *nix operating systems and Mac OS using pip:

```
$ pip install angr
```

It is recommended to create a dedicated virtual environment with Virtualenv as some of its dependencies contain forked versions Z3 and PyVEX that overwrite the original versions (you may skip this step if you don't use these libraries for anything else - on the other hand, using Virtualenv is generally a good idea).

Quite comprehensive documentation for angr is available on Gitbooks, including an installation guide, tutorials and usage examples [5]. A complete API reference is also available [6].

# Using the Disassembler Backends

# Symbolic Execution

Symbolic execution allows you to determine the conditions necessary to reach a specific target. It does this by translating the program’s semantics into a logical formula, whereby some variables are represented as symbols with specific constraints. By resolving the constraints, you can find out the conditions necessary so that some branch of the program gets executed.

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

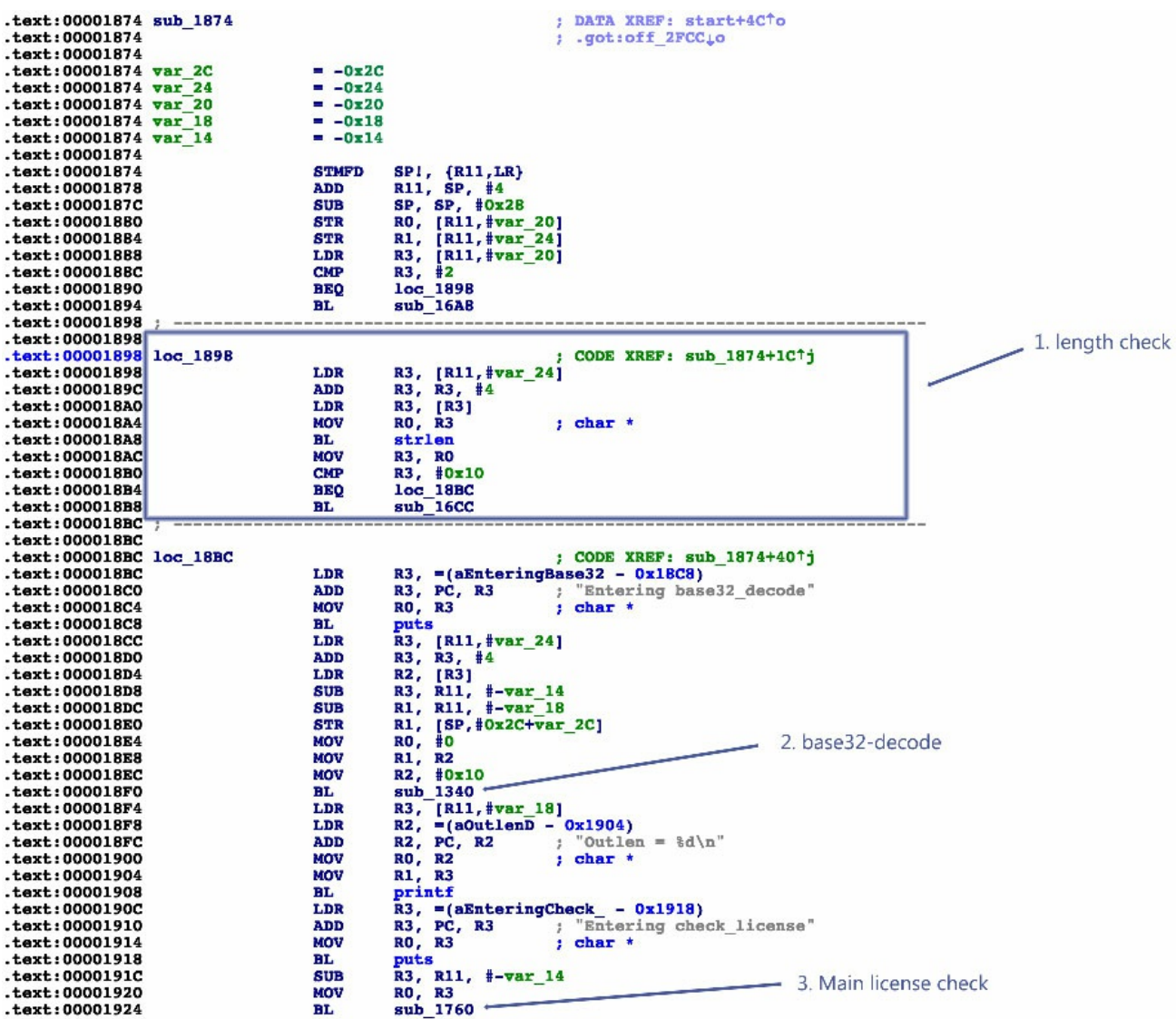
here:

https://github.com/angr/angr-doc/tree/master/examples/android_arm_license_validation

Running the executable on any Android device should give you the following output.

```
$ adb push validate /data/local/tmp
[100%] /data/local/tmp/validate
$ adb shell chmod 755 /data/local/tmp/validate
$ adb shell /data/local/tmp/validate
Usage: ./validate <serial>
$ adb shell /data/local/tmp/validate 12345
Incorrect serial (wrong format).
```

So far, so good, but we really know nothing about how a valid license key might look like. Where do we start? Let's fire up IDA Pro to get a first good look at what is happening.



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.

# 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 Nougat, run make-standalone-toolchain.sh from the Android NDK package as follows:

```
$ cd android-ndk-rXXX
$ build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-24 --install-
dir=/tmp/my-android-toolchain
```

Set the CROSS_COMPILE environment variable to point to your NDK directory and run "make" to build the kernel.

```
$ export CROSS_COMPILE=/tmp/my-android-toolchain/bin/arm-eabi-
$ make
```

# Booting the Custom Environment

Before booting into the new Kernel, make a copy of the original boot image from your device. Look up the location of the boot partition as follows:

```
root@hammerhead:/dev # ls -al /dev/block/platform/msm_sdcc.1/by-name/
lrwxrwxrwx root     root              1970-08-30 22:31 DDR -> /dev/block/mmcblk0p24
lrwxrwxrwx root     root              1970-08-30 22:31 aboot -> /dev/block/mmcblk0p6
lrwxrwxrwx root     root              1970-08-30 22:31 abootb -> /dev/block/mmcblk0p11
lrwxrwxrwx root     root              1970-08-30 22:31 boot -> /dev/block/mmcblk0p19
(...)
lrwxrwxrwx root     root              1970-08-30 22:31 userdata -> /dev/block/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.

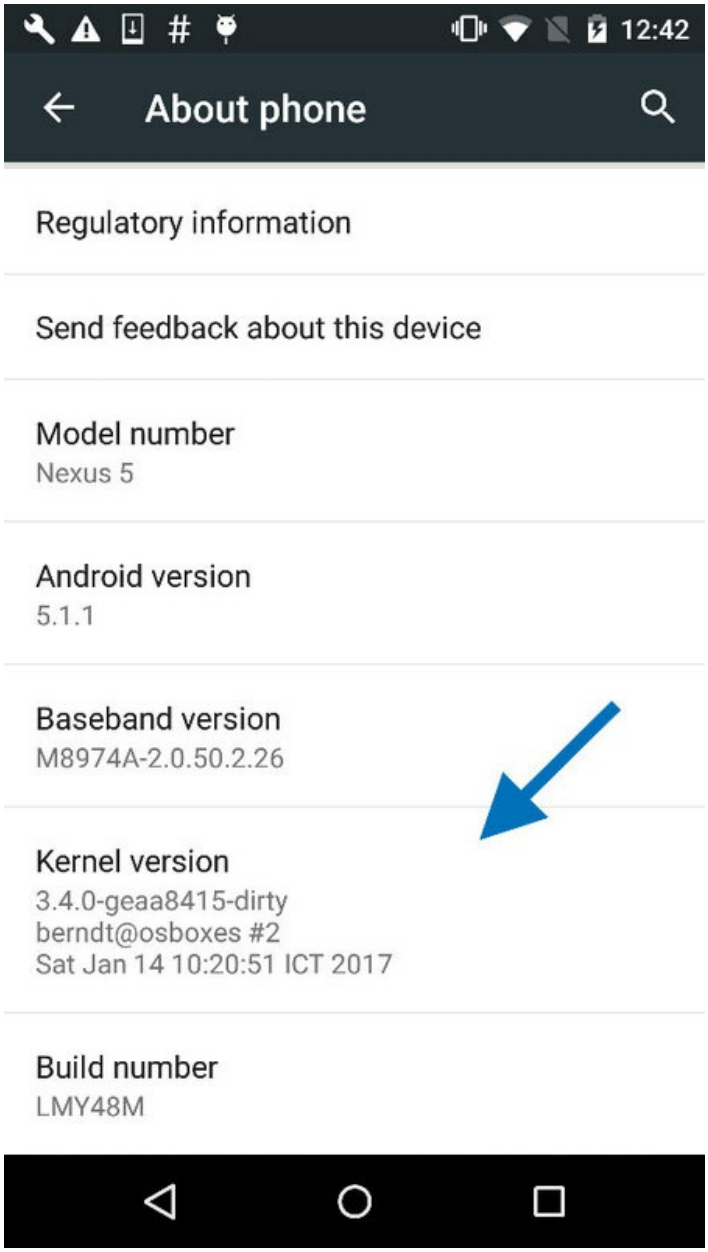
You can now use fastboot to test the new kernel. The "fastboot boot" command allows you to run the kernel without actually flashing it (once you're sure everything works, you can make the changes permanent with fastboot flash - but you don't have to). Restart the device in fastboot mode with the following 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).

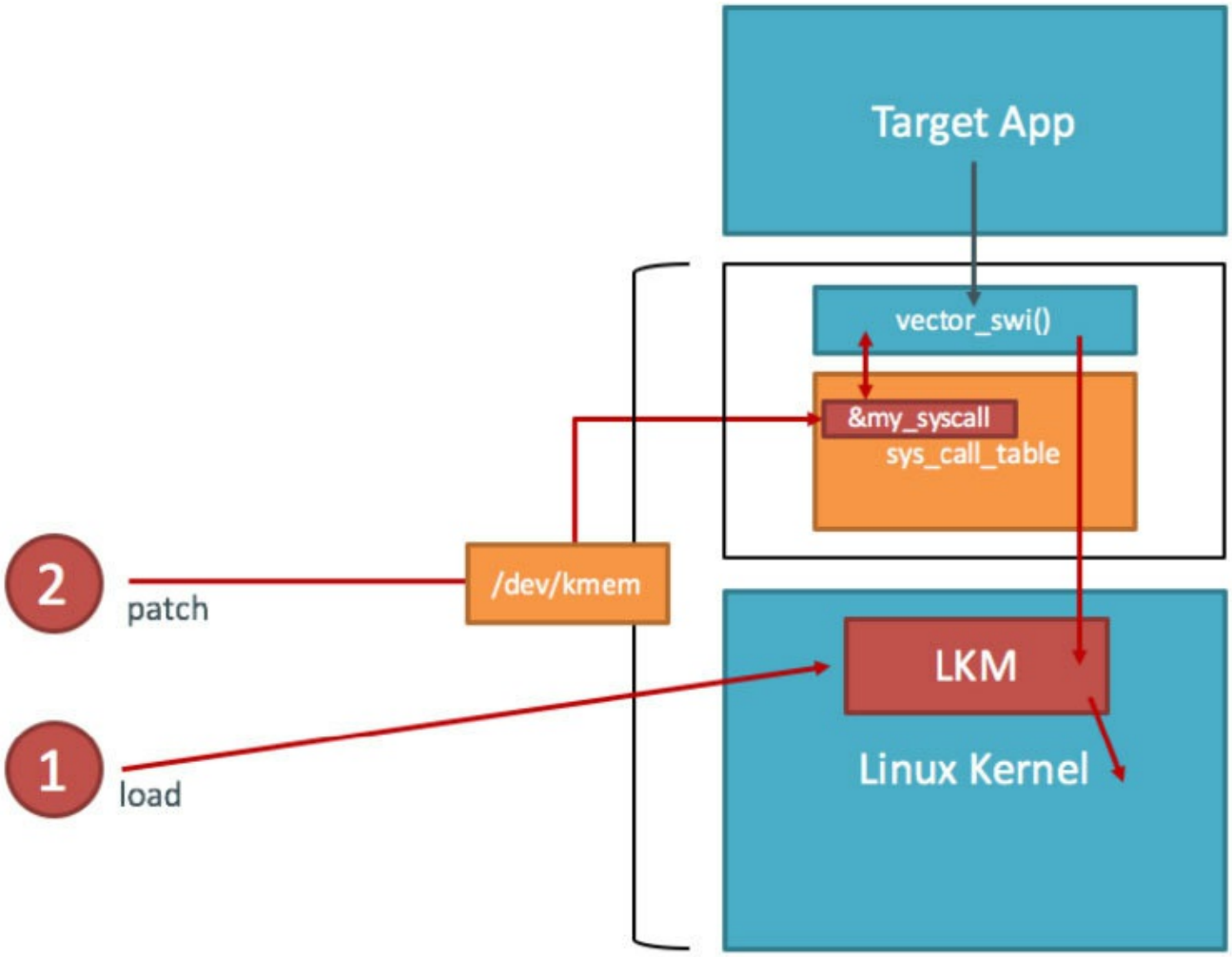
```
$ fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset
0x2900000 --tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead
user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

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].




The first piece of information we need is the address of sys_call_table. Fortunately, it is exported as a symbol in the Android kernel (iOS reversers are not so lucky). We can look up the address in the /proc/kallsyms file:

```
$ adb shell "su -c echo 0 > /proc/sys/kernel/kptr_restrict"
$ adb shell cat /proc/kallsyms | grep sys_call_table
c000f984 T sys_call_table
```

This is the only memory address we need for writing our kernel module - everything else can be calculated using offsets taken from the Kernel headers (hopefully you didn't delete them yet?).

**Example: File Hiding**

In this howto, we're going to use a Kernel module to hide a file. Let's create a file on the device so we can hide it later:

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

Finally it's time to write the kernel module. For file hiding purposes, we'll need to hook one of the system calls used to open (or check for the existence of) files. Actually, there many of those - open, openat, access, accessat, 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:

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/unistd.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

asmlinkage int (*real_openat)(int, const char __user*, int);

void **sys_call_table;

int new_openat(int dirfd, const char \__user* pathname, int flags)
{
  char *kbuf;
  size_t len;

  kbuf=(char*)kmalloc(256,GFP_KERNEL);
  len = strncpy_from_user(kbuf,pathname,255);

  if (strcmp(kbuf, "/data/local/tmp/nowyouseeme") == 0) {
    printk("Hiding file!\n");
    return -ENOENT;
  }

  kfree(kbuf);

  return real_openat(dirfd, pathname, flags);
```

```
}

int init_module() {

  sys_call_table = (void*)0xc000f984;
  real_openat = (void*)(sys_call_table[__NR_openat]);

return 0;

}
```

To build the kernel module, you need the kernel sources and a working toolchain - since you 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 [19] 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 the Bernhard Mueller's Hacking Soft Tokens Paper [27].

# References


- [1] UnCrackable Mobile Apps - https://github.com/OWASP/owasp-mstg/blob/master/Crackmes/
- [2] Android Studio - https://developer.android.com/studio/index.html
- [3] JD - http://jd.benow.ca/
- [4] JAD - http://www.javadecompilers.com/jad

- [5] Proycon - http://proycon.com/en/
- [6] CFR - http://www.benf.org/other/cfr/
- [7] apkx - APK Decompilation for the Lazy - https://github.com/b-mueller/apkx
- [8] NDK Downloads - https://developer.android.com/ndk/downloads/index.html#stable-downloads
- [9] IntelliJ IDEA - https://www.jetbrains.com/idea/
- [10] Eclipse - https://eclipse.org/ide/
- [11] Smalidea - https://github.com/JesusFreke/smali/wiki/smalidea
- [12] APKTool - https://ibotpeaches.github.io/Apktool/
- [13] Radare2 - https://www.radare.org
- [14] Angr - http://angr.io/
- [15] JEB Decompiler - https://www.pnfsoftware.com
- [16] IDA Pro - https://www.hex-rays.com/products/ida/
- [17] Bionic libc - https://github.com/android/platform_bionic
- [18] NetSPI Blog - Attacking Android Applications with Debuggers - https://blog.netspi.com/attacking-android-applications-with-debuggers/
- [19] Phrack Magazine - Android Platform based Linux kernel rootkit
- [20] DECAF - https://github.com/sycurelab/DECAF
- [21] PANDA - https://github.com/moyix/panda/blob/master/docs/
- [22] VxStripper - http://vxstripper.pagesperso-orange.fr
- [23] Dynamic Malware Recompliation - http://ieeexplore.ieee.org/document/6759227/
- [24] Xposed - http://repo.xposed.info/module/de.robv.android.xposed.installer
- [25] Xposed Development Tutorial - https://github.com/rovo89/XposedBridge/wiki/Development-tutorial
- [26] Frida - https://www.frida.re
- [27] Hacking Soft Tokens on - https://packetstormsecurity.com/files/138504/HITB_Hacking_Soft_Tokens_v1.2.pdf

# Anti-Reversing Defenses on Android

This chapter contains sample content about anti-reversing defenses on Android. It describes commonly used defenses, as well as bypass techniques for each type of defense.

## 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].

##### 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].

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 devic`e 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 resilience assessment, disabling the root detection mechanisms is your first step.

You can use a number of techniques to bypass these checks, most of which were introduced in the "Reverse Engineering and Tampering" chapter:

1. Renaming binaries. For example, in some cases simply renaming the "su" binary to something else is enough to defeat root detection (try not to break your 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.

# 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] OWASP Mobile Crackmes - https://github.com/OWASP/owasp-mstg/blob/master/OMTG-Files/02_Crackmes/List_of_Crackmes.md
- [2] 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/

# 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.

**Anti-JDWP-Debugging Examples**

In the chapter "Reverse Engineering and Tampering", we talked about JDWP, the protocol used for communication between the debugger and the Java virtual machine. We also showed that it 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].

```
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]:

```
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);

        }
    }
}
```

### Anti-Native-Debugging Examples

Most Anti-JDWP tricks (safe for maybe timer-based checks) won't catch classical, ptrace-based debuggers, so separate defenses are needed to defend against this type of debugging. Many "traditional" Linux anti-debugging tricks are employed here.

#### Checking TracerPid

When the `ptrace` system call is used to attach to a process, the "TracerPid" field in the status file of the debugged process shows the PID of the attaching process. The default value of "TracerPid" is "0" (no other process attached). Consequently, finding anything else than "0" in that field is a sign of debugging or other ptrace-shenanigans.

The following implementation is taken from Tim Strazzere's Anti-Emulator project [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.

# References


- [1] Matenaar et al. - Patent Application - MOBILE DEVICES WITH INHIBITED APPLICATION DEBUGGING AND METHODS OF OPERATION - https://www.google.com/patents/US8925077
- [2] 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


# File Integrity Checks

# Overview

There are two file-integrity related topics:

1. *The application-source related integrity checks:* In the "Tampering and Reverse Engineering" chapter, we discussed Android's APK code signature check. We also saw that determined reverse engineers can easily bypass this check by re-packaging and re-signing an app. To make this process more involved, a protection scheme can be augmented with CRC checks on the app bytecode and native libraries as well as important data files. These checks can be implemented both on the Java and native layer. The idea is to have additional controls in place so that the only runs correctly in its unmodified state, even if the code signature is valid.
2. *The file storage related integrity checks:* When files are stored by the application using the SD-card or public storage, or when key-value pairs are stored in the `SharedPreferences`, then their integrity should be protected.

#### Sample Implementation - application-source

Integrity checks often calculate a checksum or hash over selected files. Files that are commonly protected include:

- AndroidManifest.xml
- Class files *.dex
- Native libraries (*.so)

The following sample implementation from the Android Cracking Blog [1] 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;
 }
}
```

**Sample Implementation - Storage**

When providing integrity on the storage itself. You can either create an HMAC over a given key-value pair as for the Android `SharedPreferences` or you can create an HMAC over a complete file provided by the filesystem. When using an HMAC, you can either use a bouncy castle implementation to HMAC the given content or the AndroidKeyStore and then verify the HMAC later on: There are a few steps to take care of. In case of the need for encryption. Please make sure that you encrypt and then HMAC as described in [2].

When generating an HMAC with BouncyCastle:

1. Make sure BounceyCastle or SpongeyCastle are registered as a security provider.

2. Initialize the HMAC with a key, which can be stored in a keystore.
3. Get the bytearray of the content that needs an HMAC.
4. Call `doFinal` on the HMAC with the bytecode.
5. Append the HMAC to the bytearray of step 3.
6. Store the result of step 5.

When verifying the HMAC with BouncyCastle:

1. Make sure BounceyCastle or SpongeyCastle are registered as a security provider.
2. Extract the message and the hmacbytes as separate arrays.
3. Repeat step 1-4 of generating an hmac on the data.
4. Now compare the extracted hamcbytes to the result of step 3.

When generating the HMAC based on the Android keystore, then it is best to only do this for Android 6 and higher. In that case you generate the key for hmacking as described in [3]. A convinient HMAC implementation without the `AndroidKeyStore` can be found below:

```
public enum HMACWrapper {
    HMAC_512("HMac-SHA512"), //please note that this is the spec for the BC provider
    HMAC_256("HMac-SHA256");

    private final String algorithm;

    private HMACWrapper(final String algorithm) {
        this.algorithm = algorithm;
    }

    public Mac createHMAC(final SecretKey key) {
        try {
            Mac e = Mac.getInstance(this.algorithm, "BC");
            SecretKeySpec secret = new SecretKeySpec(key.getKey().getEncoded(),
this.algorithm);
            e.init(secret);
            return e;
        } catch (NoSuchProviderException | InvalidKeyException | NoSuchAlgorithmException e) {
            //handle them
        }
    }
```

```
public byte[] hmac(byte[] message, SecretKey key) {
    Mac mac = this.createHMAC(key);
    return mac.doFinal(message);
}

public boolean verify(byte[] messageWithHMAC, SecretKey key) {
    Mac mac = this.createHMAC(key);
    byte[] checksum = extractChecksum(messageWithHMAC, mac.getMacLength());
    byte[] message = extractMessage(messageWithHMAC, mac.getMacLength());
    byte[] calculatedChecksum = this.hmac(message, key);
    int diff = checksum.length ^ calculatedChecksum.length;

    for (int i = 0; i < checksum.length && i < calculatedChecksum.length; ++i) {
        diff |= checksum[i] ^ calculatedChecksum[i];
    }

    return diff == 0;
}

public byte[] extractMessage(byte[] messageWithHMAC) {
    Mac hmac = this.createHMAC(SecretKey.newKey());
    return extractMessage(messageWithHMAC, hmac.getMacLength());
}

private static byte[] extractMessage(byte[] body, int checksumLength) {
    if (body.length >= checksumLength) {
        byte[] message = new byte[body.length - checksumLength];
        System.arraycopy(body, 0, message, 0, message.length);
        return message;
    } else {
        return new byte[0];
    }
}

private static byte[] extractChecksum(byte[] body, int checksumLength) {
    if (body.length >= checksumLength) {
        byte[] checksum = new byte[checksumLength];
        System.arraycopy(body, body.length - checksumLength, checksum, 0, checksumLength);
        return checksum;
    } else {
        return new byte[0];
    }
}
```

```
    static {
        Security.addProvider(new BouncyCastleProvider());
    }
}
```

Another way of providing integrity, is by signing the obtained byte-array. Please check [3] on how to generate a signature. Do not forget to add the signature to the original byte-array.

#### Bypassing File Integrity Checks

*When trying to bypass the application-source integrity checks*

1. Patch out the anti-debugging functionality. Disable the unwanted 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.

*When trying to bypass the storage integrity checks*

1. Retrieve the data from the device, as described at the secion for device binding.
2. Alter the data retrieved and then put it back in the storage

## References

#### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

#### OWASP MASVS

-- V8.3: "The app detects, and responds to, tampering with executable files and critical data".

#### CWE

- N/A

#### Info

- [1] Android Cracking Blog - http://androidcracking.blogspot.sg/2011/06/anti-tampering-with-crc-check.html
- [2] Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm - http://cseweb.ucsd.edu/~mihir/papers/oem.html
- [3] Android Keystore System - https://developer.android.com/training/articles/keystore.html

# 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

Popular reverse engineering tools, if installed in an unmodified form, can be detected by looking for associated application packages, files, processes, or other tool-specific modifications and artefacts. In the following examples, we'll show how different ways of detecting the frida instrumentation framework which is used extensively in this guide. Other tools, such as Substrate and Xposed, can be detected using similar means. Note that DBI/injection/hooking tools can often also be detected implicitly through runtime integrity checks, which are discussed separately below.

##### Example: Ways of Detecting Frida

An obvious method for detecting frida and similar frameworks is to check the environment for related artefacts, such as package files, binaries, libraries, processes, temporary files, and others. As an example, I'll home in on fridaserver, the daemon responsible for exposing frida over TCP. One could use a Java method that iterates through the list of running processes to check whether fridaserver is running:

```
public boolean checkRunningProcesses() {

  boolean returnValue = false;

  // Get currently running application processes
  List<RunningServiceInfo> list = manager.getRunningServices(300);

  if(list != null){
    String tempName;
    for(int i=0;i<list.size();++i){
      tempName = list.get(i).process;

      if(tempName.contains("fridaserver")) {
        returnValue = true;
      }
    }
  }
  return returnValue;
}
```

This works if frida is run in its default configuration. Perhaps it's also enough to stump some script kiddies doing their first little baby steps in reverse engineering. It can however be easily bypassed by renaming the fridaserver binary to "lol" or other names, so we should maybe find a better method.

By default, fridaserver binds to TCP port 27047, so checking whether this port is open is another idea. In native code, this could look as follows:

```
boolean is_frida_server_listening() {
    struct sockaddr_in sa;

    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(27047);
    inet_aton("127.0.0.1", &(sa.sin_addr));

    int sock = socket(AF_INET , SOCK_STREAM , 0);

    if (connect(sock , (struct sockaddr*)&sa , sizeof sa) != -1) {
      /* Frida server detected. Do something… */
    }

}
```

Again, this detects fridaserver in its default mode, but the listening port can be changed easily via command line argument, so bypassing this is a little bit too trivial. The situation can be improved by pulling an nmap -sV. fridaserver uses the D-Bus protocol to communicate, so we send a D-Bus AUTH message to every open port and check for an answer, hoping for fridaserver to reveal itself.

```
/*
 * Mini-portscan to detect frida-server on any local port.
 */

for(i = 0 ; i <= 65535 ; i++) {

    sock = socket(AF_INET , SOCK_STREAM , 0);
    sa.sin_port = htons(i);

    if (connect(sock , (struct sockaddr*)&sa , sizeof sa) != -1) {

        __android_log_print(ANDROID_LOG_VERBOSE, APPNAME,  "FRIDA DETECTION [1]: Open Port:
%d", i);

        memset(res, 0 , 7);

        // send a D-Bus AUTH message. Expected answer is “REJECT"

        send(sock, "\x00", 1, NULL);
        send(sock, "AUTH\r\n", 6, NULL);

        usleep(100);

        if (ret = recv(sock, res, 6, MSG_DONTWAIT) != -1) {

            if (strcmp(res, "REJECT") == 0) {
               /* Frida server detected. Do something… */
            }
        }
    }
    close(sock);
}
```

We now have a pretty robust method of detecting fridaserver, but there's still some glaring issues. Most importantly, frida offers alternative modes of operations that don't require fridaserver! How do we detect those?

The common theme in all of frida's modes is code injection, so we can expect to have frida-related libraries mapped into memory whenever frida is used. The straightforward way to detect those is walking through the list of loaded libraries and checking for suspicious ones:

```
char line[512];
FILE* fp;

fp = fopen("/proc/self/maps", "r");

if (fp) {
    while (fgets(line, 512, fp)) {
        if (strstr(line, "frida")) {
            /* Evil library is loaded. Do something… */
        }
    }

    fclose(fp);

    } else {
       /* Error opening /proc/self/maps. If this happens, something is of. */
    }
}
```

This detects any libraries containing "frida" in the name. On its surface this works, but there's some major issues:

- Remember how it wasn't a good idea to rely on fridaserver being called fridaserver? The same applies here - with some small modifications to frida, the frida agent libraries could simply be renamed.
- Detection relies on standard library calls such as fopen() and strstr(). Essentially, we're attempting to detect frida using functions that can be easily hooked with - you guessed it - frida. Obviously this isn't a very solid strategy.

Issue number one can be addressed by implementing a classic-virus-scanner-like strategy, scanning memory for the presence of "gadgets" found in frida's libraries. I chose the string "LIBFRIDA" which appears to be present in all versions of frida-gadget and frida-agent. Using the following code, we iterate through the memory mappings listed in /proc/self/maps, and search for the string in every executable section. Note that I ommitted the more boring functions for the sake of brevity, but you can find them on GitHub.

```
static char keyword[] = "LIBFRIDA";
num_found = 0;

int scan_executable_segments(char * map) {
    char buf[512];
    unsigned long start, end;

    sscanf(map, "%lx-%lx %s", &start, &end, buf);

    if (buf[2] == 'x') {
        return (find_mem_string(start, end, (char*)keyword, 8) == 1);
    } else {
        return 0;
    }
}

void scan() {

    if ((fd = my_openat(AT_FDCWD, "/proc/self/maps", O_RDONLY, 0)) >= 0) {

    while ((read_one_line(fd, map, MAX_LINE)) > 0) {
        if (scan_executable_segments(map) == 1) {
            num_found++;
        }
    }

    if (num_found > 1) {

        /* Frida Detected */
    }

}
```

Note the use of my_openat() etc. instead of the normal libc library functions. These are custom implementations that do the same as their Bionic libc counterparts: They set up the arguments for the respective system call and execute the swi instruction (see below). Doing this removes the reliance on public APIs, thus making it less susceptible to the typical libc hooks. The complete implementation is found in syscall.S. The following is an assembler implementation of my_openat().

```
#include "bionic_asm.h"

.text
    .globl my_openat
    .type my_openat,function
my_openat:
    .cfi_startproc
    mov ip, r7
    .cfi_register r7, ip
    ldr r7, =__NR_openat
    swi #0
    mov r7, ip
    .cfi_restore r7
    cmn r0, #(4095 + 1)
    bxls lr
    neg r0, r0
    b __set_errno_internal
    .cfi_endproc

    .size my_openat, .-my_openat;
```

This is a bit more effective as overall, and is difficult to bypass with frida only, especially with some obuscation added. Even so, there are of course many ways of bypassing this as well. Patching and system call hooking come to mind. Remember, the reverse engineer always wins!

To experiment with the detection methods above, you can download and build the Android Studio Project. The app should generate entries like the following when frida is injected.

**Bypassing Detection of Reverse Engineering Tools**

1. Patch out the anti-debugging functionality. Disable the unwanted behaviour by simply overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook file system APIs on the Java and native layers. Return a handle to the original file instead of the modified file.
3. Use Kernel module to intercept file-related system calls. When the process attempts to open the modified file, return a file descriptor for the unmodified version of the file instead.

Refer to the "Tampering and Reverse Engineering section" for examples of patching, code injection and kernel modules.

## References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

### OWASP MASVS

- V8.4: "The app detects the presence of widely used reverse engineering tools, such as code injection tools, hooking frameworks and debugging servers."

### CWE

N/A

### Info

- [1] Netitude Blog - Who owns your runtime? - https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/

### Tools

- frida - https://www.frida.re/

# Emulator Detection

## Overview

In the context of anti-reversing, the goal of emulator detection is to make it a bit more difficult to run the app on a emulated device, which in turn impedes some tools and techniques reverse engineers like to use. This forces the reverse engineer to defeat the emulator checks or utilize the physical device. This provides a barrier to entry for large scale device analysis.

# Emulator Detection Examples

There are several indicators that indicate the device in question is being emulated. While all of these API calls could be hooked, this provides a modest first line of defense.

The first set of 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.

# Bypassing Emulator Detection

1. Patch out the emulator detection functionality. Disable the unwanted behaviour by simply overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook file system APIs on the Java and native layers. Return innocent looking values (preferably taken from a real device) instead of the tell-tale emulator values. For example, you can override the `TelephonyManager.getDeviceID()` method to return an IMEI value.

Refer to the "Tampering and Reverse Engineering section" for examples of patching, code injection and kernel modules.

# References

### OWASP Mobile Top 10 2016

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

**OWASP MASVS**

- V8.5: "The app detects, and response to, being run in an emulator using any method."

**CWE**

N/A

**Info**

- [1] Timothy Vidas & Nicolas Christin - Evading Android Runtime Analysis via Sandbox Detection - https://users.ece.cmu.edu/~tvidas/papers/ASIACCS14.pdf

**Tools**

N/A

# Runtime Integrity Checks

# Overview

Controls in this category verify the integrity of the app's own memory space, with the goal of protecting against memory patches applied during runtime. This includes unwanted changes to binary code or bytecode, functions pointer tables, and important data structures, as well as rogue code loaded into process memory. Intergrity can be verified either by:

1. Comparing the contents of memory, or a checksum over the contents, with known good values;
2. Searching memory for signatures of unwanted modifications.

There is some overlap with the category "detecting reverse engineering tools and frameworks", and in fact we already demonstrated the signature-based approach in that chapter, when we showed how to search for frida-related strings in process memory. Below are a few more examples for different kinds of integrity monitoring.

### Runtime Integrity Check Examples

### Detecting tampering with the Java Runtime

Detection code from the dead && end blog [3].

```
try {
  throw new Exception();
}
catch(Exception e) {
  int zygoteInitCallCount = 0;
  for(StackTraceElement stackTraceElement : e.getStackTrace()) {
    if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit")) {
      zygoteInitCallCount++;
      if(zygoteInitCallCount == 2) {
        Log.wtf("HookDetection", "Substrate is active on the device.");
      }
    }
    if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
        stackTraceElement.getMethodName().equals("invoked")) {
      Log.wtf("HookDetection", "A method on the stack trace has been hooked using Substrate.");
    }
    if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
        stackTraceElement.getMethodName().equals("main")) {
      Log.wtf("HookDetection", "Xposed is active on the device.");
    }
    if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
        stackTraceElement.getMethodName().equals("handleHookedMethod")) {
      Log.wtf("HookDetection", "A method on the stack trace has been hooked using Xposed.");
    }

  }
}
```

### Detecting Native Hooks

With ELF binaries, native function hooks can be installed by either overwriting function pointers in memory (e.g. GOT or PLT hooking), or patching parts of the function code itself (inline hooking). Checking the integrity of the respective memory regions is one technique to detect this kind of hooks.

The Global Offset Table (GOT) is used to resolve library functions. During runtime, the dynamic linker patches this table with the absolute addresses of global symbols. *GOT hooks* overwrite the stored function addresses and redirect legitimate function calls to adversary-controlled code. This type of hook can be detected by enumerating the process memory map and verifying that each GOT entry points into a legitimately loaded library.

In contrast to GNU `ld`, which resolves symbol addresses only once they are needed for the first time (lazy binding), the Android linker resolves all external function and writes the respective GOT entries immediately when a library is loaded (immediate binding). One can therefore expect all GOT entries to point valid memory locations within the code sections of their respective libraries during runtime. GOT hook detection methods usually walk the GOT and verify that this is indeed the case.

*Inline hooks* work by overwriting a few instructions at the beginning or end of the function code. During runtime, this so-called trampoline redirects execution to the injected code. Inline hooks can be detected by inspecting the prologues and epilogues of library functions for suspect instructions, such as far jumps to locations outside the library.

## Bypassing Runtime Integrity Checks

Make sure that all file-based detection of reverse engineering tools is disabled. Then, inject code using Xposed, Frida and Substrate, and attempt to install native hooks and Java method hooks. The app should detect the "hostile" code in its memory and respond accordingly. For a more detailed assessment, identify and bypass the detection mechanisms employed and use the criteria listed under "Assessing Programmatic Defenses" in the "Assessing Software Protection Schemes" chapter.

Work on bypassing the checks using the following techniques:

1. Patch out the integrity checks. Disable the unwanted behaviour by overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook APIs to hook the APIs used for detection and return fake values.

Refer to the "Tampering and Reverse Engineering section" for examples of patching, code injection and kernel modules.

## References

**OWASP Mobile Top 10 2016**

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

**OWASP MASVS**

- V8.6: "The app detects, and responds to, tampering the code and data in its own memory space."

**CWE**

N/A

**Info**

- [1] Michael Hale Ligh, Andrew Case, Jamie Levy, Aaron Walters (2014) *The Art of Memory Forensics*. Wiley. "Detecting GOT Overwrites", p. 743.
- [2] Netitude Blog - "Who owns your runtime?" - https://labs.nettitude.com/blog/ios-and-android-runtime-and-anti-debugging-protections/
- [3] dead && end blog - Android Anti-Hooking Techniques in Java - http://d3adend.org/blog/?p=589

# Device Binding

## Overview

The goal of device binding is to impede an attacker when he tries to copy an app and its state from device A to device B and continue the execution of the app on device B. When device A has been deemed trusted, it might have more privileges than device B, which should not change when an app is copied from device A to device B.

In the past, Android developers often relied on the Secure ANDROID_ID (SSAID) and MAC addresses. However, the behavior of the SSAID has changed since Android O and the behavior of MAC addresses have changed in Android N [1]. Google has set a new set of recommendations in their SDK documentation regarding identifiers as well [2].

When the source-code is available, then there are a few codes you can look for, such as:

- The presence of unique identifiers that no longer work in the future
  - `Build.SERIAL` without the presence of `Build.getSerial()`
  - `htc.camera.sensor.front_SN` for HTC devices
  - `persist.service.bdroid.bdadd`
  - `Settings.Secure.bluetooth_address` , unless the system permission LOCAL_MAC_ADDRESS is enabled in the manifest.
- The presence of using the ANDROID_ID only as an identifier. This will influence the possible binding quality over time given older devices.
- The absence of both InstanceID, the `Build.SERIAL` and the IMEI.

```
TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

Furthermore, to reassure that the identifiers can be used, the AndroidManifest.xml needs to be checked in case of using the IMEI and the Build.Serial. It should contain the following permission: `<uses-permission android:name="android.permission.READ_PHONE_STATE"/>` .

There are 3 methods which allow for device binding:

- augment the credentials used for authentication with device identifiers. This can only make sense if the application needs to re-authenticate itself and/or the user frequently.
- obfuscate the data stored on the device using device-identifiers as keys for encryption methods. This can help in binding to a device when a lot of offline work is done by the app or when access to APIs depends on access-tokens stored by the application.
- Use a token based device authentication (InstanceID) to reassure that the same instance of the app is used.

The following 4 identifiers can be possibly used:

- Google InstanceID
- IMEI
- Serial

- SSAID

**Google InstanceID**

Google InstanceID [5] uses tokens to authenticate the application instance running on the device. The moment the application has been reset, uninstalled, etc., the instanceID is reset, meaning that you have a new "instance" of the app. You need to take the following steps into account for instanceID:

1. Configure your instanceID at your Google Developer Console for the given application. This includes managing the PROJECT_ID.
2. Setup Google play services. In your build.gradle, add:

```
apply plugin: 'com.android.application'
 ...

 dependencies {
     compile 'com.google.android.gms:play-services-gcm:10.2.4'
 }
```

3. Get an instanceID

```
String iid = InstanceID.getInstance(context).getId();
//now submit this iid to your server.
```

4. Generate a token

```
String authorizedEntity = PROJECT_ID; // Project id from Google Developer Console
String scope = "GCM"; // e.g. communicating using GCM, but you can use any
                      // URL-safe characters up to a maximum of 1000, or
                      // you can also leave it blank.
String token = InstanceID.getInstance(context).getToken(authorizedEntity,scope);
//now submit this token to the server.
```

5. Make sure that you can handle callbacks from instanceID in case of invalid device information, security issues, etc. For this you have to extend the `InstanceIDListenerService` and handle the

callbacks there:

```
public class MyInstanceIDService extends InstanceIDListenerService {
  public void onTokenRefresh() {
    refreshAllTokens();
  }

  private void refreshAllTokens() {
    // assuming you have defined TokenList as
    // some generalized store for your tokens for the different scopes.
    // Please note that for application validation having just one token with one scopes can be
enough.
    ArrayList<TokenList> tokenList = TokensList.get();
    InstanceID iid = InstanceID.getInstance(this);
    for(tokenItem : tokenList) {
      tokenItem.token =
        iid.getToken(tokenItem.authorizedEntity,tokenItem.scope,tokenItem.options);
      // send this tokenItem.token to your server
    }
  }
};
```

Lastly register the service in your AndroidManifest:

```
<service android:name=".MyInstanceIDService" android:exported="false">
  <intent-filter>
        <action android:name="com.google.android.gms.iid.InstanceID"/>
  </intent-filter>
</service>
```

When you submit the iid and the tokens to your server as well, you can use that server together with the Instance ID Cloud Service to validate the tokens and the iid. When the iid or token seems invalid, then you can trigger a safeguard procedure (e.g. inform server on possible copying, possible security issues, etc. or removing the data from the app and ask for a re-registration).

Please note that Firebase has support for InstanceID as well [4].

**IMEI & Serial**

Please note that Google recommends against using these identifiers unless there is a high risk involved with the application in general.

For pre-Android O devices, you can request the serial as follows:

```
String serial = android.os.Build.SERIAL;
```

From Android O onwards, you can request the device its serial as follows:

1. Set the permission in your Android Manifest:

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

2. Request the permission at runtime to the user: See https://developer.android.com/training/permissions/requesting.html for more details.
3. Get the serial:

```
String serial = android.os.Build.getSerial();
```

Retrieving the IMEI in Android works as follows:

1. Set the required permission in your Android Manifest:

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

2. If on Android M or higher: request the permission at runtime to the user: See https://developer.android.com/training/permissions/requesting.html for more details.

3. Get the IMEI:

```
TelephonyManager tm = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

**SSAID**

Please note that Google recommends against using these identifiers unless there is a high risk involved with the application in general. you can retrieve the SSAID as follows:

```
String SSAID = Settings.Secure.ANDROID_ID;
```

The behavior of the SSAID has changed since Android O and the behavior of MAC addresses have changed in Android N `[1]` . Google has set a new set of recommendations in their SDK documentation regarding identifiers as well `[2]` . Because of this new behavior, we recommend developers to no relie on the SSAID alone, as the identifier has become less stable. For instance: The SSAID might change upon a factory reset or when the app is reinstalled after the upgrade to Android O. Please note that there are amounts of devices which have the same ANDROID_ID and/or have an ANDROID_ID that can be overriden. Next, the Build.Serial was often used. Now, apps targetting Android O will get "UNKNOWN" when they request the Build.Serial.

# References

**OWASP Mobile Top 10 2016**

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

**OWASP MASVS**

- V8.10: "The app implements a 'device binding' functionality using a device fingerprint derived from multiple properties unique to the device."

**CWE**

N/A

**Info**

- [1] Changes in the Android device identifiers - https://android-

developers.googleblog.com/2017/04/changes-to-device-identifiers-in.html

- [2] Developer Android documentation - https://developer.android.com/training/articles/user-data-ids.html
- [3] Documentation on requesting runtime permissions - https://developer.android.com/training/permissions/requesting.html
- [4] Firebase InstanceID documentation - https://firebase.google.com/docs/reference/android/com/google/firebase/iid/FirebaseInstanceId
- [5] Google InstanceID documentation - https://developers.google.com/instance-id/