

<Development of a RASP for Android>

<Defending Android applications against dynamic analysis by expanding the ARMANDroid project>

Tanguy Snoeck



Research and Development project owner:
Tanguy Snoeck

Master thesis submitted under the supervision of
Wim Mees

Academic year
2020 – 2021

in order to be awarded the Degree of
Corporate Strategies

I hereby confirm that this thesis was written independently by myself without the use of any sources beyond those cited, and all passages and ideas taken from other sources are cited accordingly.

The author(s) gives (give) permission to make this master dissertation available for consultation and to copy parts of this master dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master dissertation.

The author(s) transfers (transfer) to the project owner(s) any and all rights to this master dissertation, code and all contribution to the project without any limitation in time nor space.

16/08/2021

Abstract

With an ever increasing number of mobile applications downloaded every day, Android security has been a hot topic of research for years. Because of its design, the Android operating system and more broadly, the Android environment is prone to a lot of varied threats. One of the most damaging threats for Android users is the development of malwares. Malware developers will often modify an existing application to redistribute it on non official platforms through a process called repackaging. To leverage such an attack, adversaries use a wide collection of techniques ranging from static analysis to code instrumentation. Countermeasures and defense mechanisms against this particular attack were proposed in the past, often with a focus on the prevention of application modification. As will be demonstrated in this thesis, focus must also be put on different attack vectors to effectively and durably protect Android applications.

Recently, products aiming at protecting applications from within appeared on the market with many companies offering Runtime Application Self Protection (RASP) solutions. Those products often claim more or less the same features: root detection, debugging detection and prevention, emulator detection, obfuscation and tampering prevention. Interestingly (most probably because of the commercial nature of these solutions), very few literature exist on the topic of RASP development.

This thesis aims at providing one of the first design and development document of a RASP solution for Android. The research presented in this paper heavily relies on the Anti-Repackaging through Multi-pattern Anti-tampering based on Native Detection (ARMAND) project, developed by the Italian researchers Alessio Merlo, Antonio Ruggia, Luigi Sciolla and Luca Verderame from the University of Genova.

My contribution on the topic will be the addition of effective protections against dynamic analysis to the ARMAND project which will help build a safer cyberspace for Android application and their users.

Keywords: RASP, Android, Security, Detection, Prevention

Acknowledgements

This project was easily one of the most ambitious I worked on to this day. It represents months of hard work and multiple experiments, and proof of concepts that worked but weren't viable. All those failed projects were far from useless though, as they helped me understand the problematic better and lead me towards the solution presented in this document. My interest for Android security dates back a long time and it was with great pleasure and excitement that I explored this topic during this year.

Such a project would not have been possible without some help though. I would like to thank first my thesis supervisor, M. Wim Mees, who helped me and supported my ideas from the beginning; he directly saw value in what I was trying to achieve and pointed my research in the right direction.

The work presented in this thesis relies heavily on the ARMAND project and without it, it would have been impossible to achieve what I did. I would therefore like to thank the team of researchers behind this project composed of Alessio Merlo, Antonio Ruggia, Luigi Sciolla and Luca Verderame who all agreed to collaborate with me.

Finally, I would like to thank M. Jeroen Bekers from the cybersecurity company Nviso who gave me additional advices and resources to help me during my research.

Table of Contents

Abstracts	I
Abstract	I
Table of Contents	VI
List of Figures	VI
List of Abbreviations	VII
1 Introduction	1
1.1 Motivations	1
1.1.1 Context	1
1.1.2 Problem statement	1
1.2 Project statement & contributions	2
1.2.1 Use cases	2
1.3 Organization of this document	3
2 Android system description	5
2.1 Introduction	5
2.2 Android platform	5
2.3 APK structure	6
2.4 Android build process	7
2.5 Android application sandbox	7
2.6 Android application components	8
2.6.1 Android Manifest	9
2.6.2 Intents	9
2.6.3 Activities	10
2.6.4 Services	10
2.6.5 Content providers	10
2.6.6 Broadcast receivers	10
2.7 Zygote	11
2.8 Summary	11
State of the Art	12
3 Threat model	13
3.1 Introduction	13
3.2 Device rooting	13
3.2.1 Magisk	13
3.3 Emulators	14
3.3.1 QEMU	14
3.3.2 Kernel-based virtual machine	15
3.4 Reverse engineering and static analysis techniques	15
3.4.1 Decompiling Java code	15
3.4.2 Disassembling native code	16
3.5 Dynamic analysis	16

3.5.1	Debugging	17
3.5.2	Tracing	18
3.5.3	Symbolic execution	19
3.6	Tampering and repackaging	20
3.6.1	Android manifest patching	20
3.6.2	Smali code patching	21
3.6.3	Java Bytecode manipulation	21
3.6.4	Java library injection	22
3.6.5	Native library injection	23
3.6.6	Repackaging	24
3.7	Dynamic instrumentation	25
3.7.1	Frida	25
3.7.2	Xposed	26
3.8	Summary	29
4	Application security	30
4.1	Introduction	30
4.2	Rooting detection	30
4.2.1	SafetyNet	30
4.2.2	Programmatic detection	31
4.3	Debugging detection	33
4.3.1	JDWP anti-debugging	33
4.3.2	Traditional anti-debugging	36
4.4	Reverse engineering and function hooking tools detection	38
4.4.1	Checking the application's signature	38
4.4.2	Checking the environment for related artifacts	38
4.4.3	Checking for open TCP ports	39
4.4.4	Checking for ports repoding to D-Bus authentication	39
4.4.5	Scanning process memory for known artifacts	39
4.5	Tampering detection	39
4.6	Emulator detection	40
4.6.1	Build.prop inspection	40
4.6.2	TelephonyManager	41
4.7	Obfuscation	41
4.7.1	Techniques taxonomy	42
4.7.2	Tools	43
4.8	Anti-repackaging	45
4.8.1	Common techniques	45
4.8.2	ARMAND	45
4.9	Summary	48
	Contribution	48
5	Implementation	49
5.1	Introduction	49
5.2	Current state of ARMANDroid	49
5.2.1	Overview	49
5.2.2	Java code injection	52

5.2.3	Native code injection	53
5.2.4	Java security checks	54
5.2.5	Native security checks	54
5.3	Contribution	54
5.3.1	Java protections	55
5.3.2	Native protections	58
5.4	Summary	59
6	Effectiveness Assessment	60
6.1	Introduction	60
6.2	Testing environment and methodology	60
6.3	Root detection	61
6.3.1	Checking for test key	61
6.3.2	Checking running processes	61
6.3.3	Checkig if su is on the PATH	62
6.3.4	Check files presence	62
6.4	Debug detection and prevention	62
6.4.1	Checking the debugger presence	62
6.4.2	Checking TracerPid	62
6.4.3	Debug prevention	63
6.5	Frida detection	63
6.5.1	Checking loaded libraries	63
6.5.2	Default port connection	64
6.6	Summary	65
7	Discussion	66
7.1	Introduction	66
7.2	Possible Improvements	66
7.2.1	More protections granularity	66
7.2.2	Notification to a remote server	66
7.2.3	Assets injection for better maintenance	66
7.2.4	Addition of modes	67
7.2.5	Protections improvements	67
7.2.6	Using a complementary behavior based approach	67
7.3	Conclusion	68
	Bibliography	70
	Appendices	71
A	Source code	71
A.1	Ptrace debugging prevention	71
A.2	Root detection - checking for suspicious files presence	72
A.3	Root detection - checking for suspicious package names	73
A.4	Root detection - checking if su is on PATH	74
A.5	Root detection - checking for test key	74
A.6	Emulator detection - checking suspicious package names	75
A.7	Emulator detection - checking for artifacts in build.prop	75
A.8	Emulator detection - checking the TelephonyManager API	76

A.9	Debug detection - checking if a JDWP debugger is attached	77
A.10	Debug detection - checking if a native debugger is attached	77
A.11	Debug prevention	79
A.11.1	Main method in DebugPrevention.cpp	79
A.11.2	Secondary method in utils.cpp	80
A.12	Frida detection - checking if default port is open	80
A.13	Frida detection - checking loaded libraries	80
A.14	Assets injection prototype	81

List of Figures

2.1	Platform Stack taken from Google's documentation	5
2.2	APK file structure	7
2.3	APK build process taken from Google's documentation	8
2.4	Application Sandbox	9
3.1	Frida's internal architecture taken from Frida's documentation	26
4.1	Obfuscation taxonomy by Hui X. et al	42
4.2	Java bomb implementation	46
4.3	Native-key bomb implementation	47
5.1	Logic bomb injection	51
5.2	Example of a logic bomb injected in a target application	52
6.1	Gdbserver fails to attach to the main thread of the application	63

List of Abbreviations

ARMAND	Anti-Repackaging through Multi-pattern Anti-tampering based on Native Detection
CRUD	Create Read Update Delete
QC	Qualified condition
RASP	Runtime Application Self-Protection
ROM	Read Only Memory
VM	Virtual Machine

Chapter 1

Introduction

1.1 Motivations

1.1.1 Context

For about 20 years now, smartphones have taken increasingly significant importance in our everyday life. Nowadays we use them for a lot of various tasks such as sending and receiving emails, consulting our favorite social media, transfer money through banking applications, etc ... They gained such importance that they've become an essential tool for a lot of people and that it would be almost impossible to live without one at this point.

At the beginning, smartphone operating systems were varied and depended on the device's manufacturer until 2007 and 2008 when two major companies released their own operating systems. In 2007 the first iPhone and the IOS operating systems were released by Apple and one year later, in 2008, the Android operating system was introduced by Google. Since then, the two operating systems have been fighting over market shares and in 2012, Android became the leading operating system worldwide with approximately 75 %market share according to the research firm IDC [28].

This much importance into people's everyday life attracted the attention of attackers and security of Android became an important concern. The Android platform owes its attractiveness to a couple of factors; its operating system is open source and there exists a variety of application stores to choose from, which don't enforce good enough security measures. On top of that, Android malware development was proven to be a lucrative practice as in 2014, a report showed that attackers can earn up to 12000 \$ through mobile malware related activities [37]. Another report also showed that in 2013, 98.05% of all mobile malwares detected targeted this platform confirming both the popularity of the operating system and the vulnerability of its design [20].

1.1.2 Problem statement

In this thesis I will focus on defending Android applications against dynamic analysis. To do that I will be explaining in details the design and implementation of a Runtime Application Self Protection (RASP) for Android.

In web application security, a Web Application Firewall (WAF) is a traditional solution used to increase security. Its purpose is to protect web applications and application programming interfaces (API) against attacks ranging from Denial of Service (DoS) to injection attacks and everything in between. They operate on the border of the application and thus sometimes lack the context needed to determine if a given input should be blocked or not. This results in a lack of accuracy which can be critical as it can, in some cases, not recognize an attack being made. RASPs products are the next logical step. According to Gartner, a RASP is "a security technology that is built or linked into an application or application runtime environment and is capable of controlling application execution and detecting and preventing real-time attacks." [12].

In the case of Android security, a RASP can be seen as an additional security layer in the application development that aims at controlling the application's behavior from

the inside to detect and mitigate threats at runtime. As we have seen in the introduction and will continue to demonstrate during this thesis, the Android environment is unsafe by nature and developers have always tried to increase their applications' security the best they could. Since the Android platform came out, multiple solutions and workarounds were found and used to protect Android applications as well as possible, but it usually was a cumbersome task for the developers.

To develop this product I will base my work on the ARMANDroid [38] project. ARMANDroid represents the state of the art in anti-repackaging solutions thanks to its advanced tampering protection system. ARMANDroid's aim is to protect applications against repackaging and contains therefore a great code injection mechanism which allows me to focus on the development of security checks and not on the code injection process. In this thesis I will showcase how I turned the ARMANDroid project to a full fledged RASP solution by expanding its set of features to include additional security protections against dynamic analysis.

Developing an SDK offering security controls and letting developers use it in their projects adds additional workload on the development team which then has to learn the documentation and use the solution the intended way. On top of letting the development's team deal with this security library, calls to the SDK's API can sometimes be bypassed by attackers by patching the source code. This is why an anti-tampering scheme such as ARMAND is useful in this situation. Thanks to this scheme, the added security checks cannot be removed by an attacker. ARMAND's implementation for Android, namely ARMANDroid, works on the packaged application in APK format by injecting the protections directly in the application's code. Since it requires no developer interaction, this solution could be used in an automated delivery process to add a security layer to applications before release.

1.2 Project statement & contributions

This project will offer multiple features all aiming at increasing the security of a packaged application with minimal developer interaction. The proposed solution in this thesis will offer the following set of protections:

- **Root & emulator detection** Preventing an application from running on a compromised device (i.e. a rooted physical device or an emulator) mitigates all kinds of attacks ranging from unauthorized data access to interception of communications, effectively slowing down attackers.
- **Debug detection & prevention** Preventing a production applications from being inspected with a debugger slows down the analysis of the app, which makes attacks harder to execute.
- **Dynamic analysis tools detection** Stopping the application from running when a hooking framework is present will mitigate attacks against it as those frameworks are used to analyze the app and bypass some of its security checks.

1.2.1 Use cases

The solution presented in this thesis has a lot of practical uses and goes beyond just a theoretical demonstration. In recent years, RASP products gained a lot of traction and

companies started developing their own version of the solution to then sell it commercially.

Mobile banking and Fintech

In Fintech (Financial technologies) applications, security is of prime importance because of the type of data processed. Moreover, the new European regulation PSD2 [25] in place since 2018 impose requirements that must be met by financial institutions when providing mobile applications to the public. Those requirements include:

- Implement monitoring mechanisms to take into account signs of malware infection.
- Ensure users have a secure environment to perform financial operations.
- Put security measures in place to enforce the user device's integrity.

The use of a RASP product can help companies become compliant with this regulation thanks to the security measures it adds to the companies' applications. A RASP can detect signs that a mobile application is installed on a compromised device or that the application itself has been modified and refuse to run the application until those risks have been remediated.

Mobile gaming

Patches or mod development in mobile gaming have been a known issue for years. Nowadays, patched versions of popular mobile games are being redistributed on non official app stores which makes them easily accessible to the public. Even though some of those mods don't have a high impact on the game and just give some additional capabilities to the gamer, others can have a much higher impact by, for example, unlocking paid content for free. Those types of mods are a real problem for game studios since in-app purchases are the only source of revenue for some free mobile games. Using a RASP product to protect a mobile game prevents or at least slows down the development of patches, ensuring that paid content won't be easily accessible to gamers.

Preventing malware development

As I will explain more in details later on, repackaging is the process of modifying an existing application to then distribute it on non official app stores. Through this process, attackers modify the behavior of the application to add malicious capabilities. Most of the time, those added capabilities have the only goal of providing a financial benefit to malware developers. Sometimes, such a modified application can have more pernicious intent and steal personal information from infected devices.

On top of the damage a malware infection does to the victim, the company owning the application suffer from loss of trust on the part of users. Such an event can sometimes make the news and hurt the company's reputation badly. For all these reasons, preventing repackaging should be a crucial goal for companies developing mobile applications.

1.3 Organization of this document

Chapter 2 will be dedicated to providing a thorough description of the relevant parts of the Android operating system and environment, limiting myself to what is needed to

understand the content of this thesis. In chapter 3, I will explain and describe the different tools and techniques used by adversaries to attack Android applications. Then in chapter 4, I will showcase which techniques exist that can prevent or at least mitigate such attacks in order to add a security layer to Android applications. In chapter 5 I will describe how I implemented and integrated the protection techniques discussed in chapter 3 with an existing project, ARMANDroid. In chapter 6, I will showcase the results of my work and experiments and how the implemented mechanisms effectively mitigate attacks. Finally in chapter 7 I will discuss what could be improved and the current limitations of my work.

Chapter 2

Android system description

2.1 Introduction

As any other operating system, Android is complex and offers a broad set of features for both users and developers. Therefore, this chapter will only explain the core concepts of the Android system needed to understand the rest of this thesis. Focus will be set on the composition and behavior of Android applications and a rigorous description of their technical details will be given. Understanding how applications are developed, packaged and installed on a device is of prime importance for the rest of the thesis as all those concepts were used to develop the work presented in this thesis. Each of those concepts gives to the security aware reader hints of what could be done by an adversary to attack such applications and consequently what could be done to protect them.

2.2 Android platform

The Android platform is composed of different layers with specific roles, each containing multiple components [24] as shown in figure 2.1.

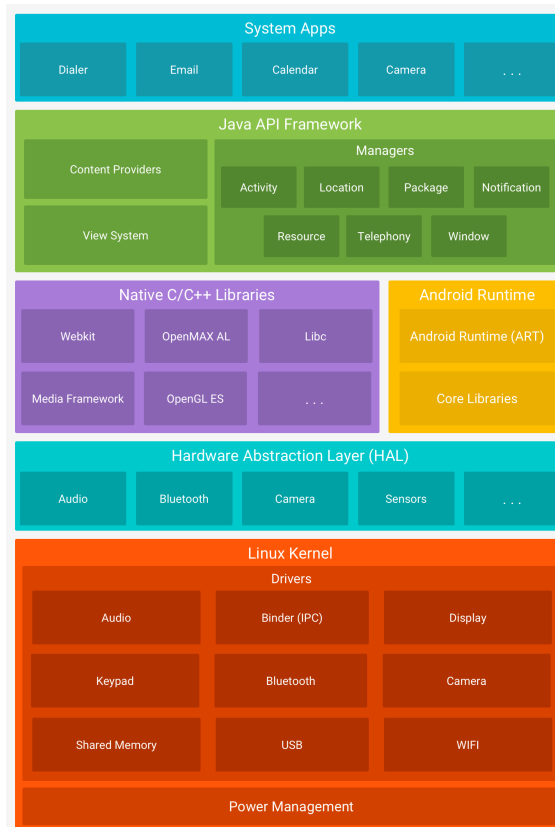


Figure 2.1 – Platform Stack taken from Google’s documentation

The Android Kernel Android is built off a Linux Kernel allowing for great modularity and portability. Linux' design makes it easy for manufacturers to develop hardware-specific features through the development of drivers.

Hardware Abstraction Level (HAL) This layer makes hardware capabilities available to the higher level Java API framework through interfaces. It consists in multiple library modules, each defining an interface to interact with a specific hardware component (e.g. camera or Bluetooth). Every time an API makes a call to access device hardware, the module of that component is loaded by the Android system.

Android Runtime (ART) ART is designed to run multiple virtual machines on low-memory devices by executing DEX files, a bytecode format replacing the Java bytecode contained in .class files in classic Java environments. It has been replacing the Dalvik virtual machine since Android Lollipop (version 5.0). In Android, each app runs in its own process and with its own instance of ART.

Native C/C++ libraries Many essential components and services of the Android system such as HAL or ART rely on libraries written in C and C++. Android provides Java framework APIs to enable apps to access some of those libraries. If an app requires C or C++ code, the developer can use the Android NDK to use those native libraries directly from native code.

Java API framework The developer interacts with the multiple services and components of the Android operating system through a variety of Java APIs.

System apps System apps are applications installed by default on the device that offer basic features you would expect for a smartphone such as SMS messaging or a calendar. These apps also offer their features to developers if they need it inside their own application.

2.3 APK structure

Android applications are packaged into an APK file which is in fact just a zip file that anyone can decompress to access the compiled sources. This file follows the structure described in figure 2.2

AndroidManifest.xml This file defines the activities used in the application along with permissions needed by the application.

classes.dex This file contains the bytecode of the application in dalvik bytecode.

resources.arsc This file contains all precompiled resources.

lib This directory contains the compiled code of native libraries that are platform dependent. It contains a sub-directory containing the library for each CPU architecture.

assets This directory contains all the visual and audio resources of the application.

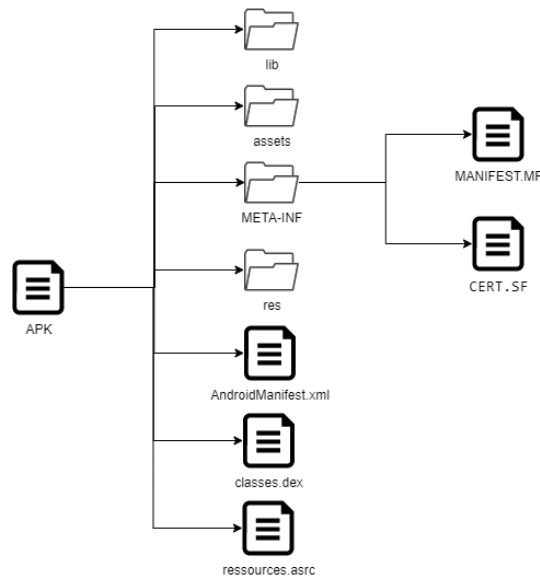


Figure 2.2 – APK file structure

META-INF This folder contains the certificate of the application along with the **MANIFEST.MF** file which contains metadata about the app and the **CERT.SF** file which contains the list of all files composing the APK and their corresponding SHA-1 digest.

res This folder contains the resources not compiled into **resources.arsc**.

2.4 Android build process

The process depicted in figure 2.3 starts by calling **aapt** (Android Asset Packaging Tool), which is a tool used to compile and package the app resources contained in the *res* folder. **R.java**, which contains the id of all available resources, and other auxiliary files are created during this step.

Next, the java compiler compiles all the source code and outputs their bytecode in **.class** files. The files can't be read directly by ART and need to be transformed first in **DEX** files. This is done by using the dex compiler or the more recent **d8** compiler and compiling those **.class** files.

Once the resources and the source code are compiled correctly, they are assembled into an **APK** file thanks to the **apkbuilder** utility. The application can't be uploaded yet on the playstore as it needs to be signed first. Multiple tools allow to sign an **APK** file, the recent **apksigner** offered with others Android cli programs.

The last step of this process is to call **zipalign** on the signed **APK** in order to reduce the RAM needed when running the application. **Zipalign** is an **APK** optimization tool that aligns on 4 bytes boundaries all uncompressed data such as images or raw files within the **APK**.

2.5 Android application sandbox

Android benefits from Linux-based protections to monitor and isolate applications from each other. Each application is assigned a unique user ID (UID) and runs in its own process.

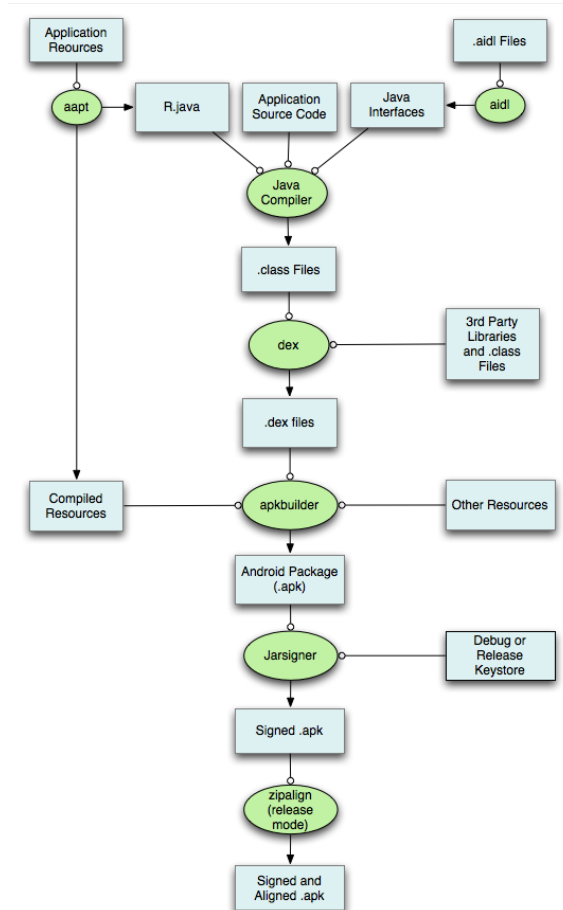


Figure 2.3 – APK build process taken from Google’s documentation

The UID mechanism is used to set up a kernel-level application sandbox. Security between applications and the underlying operating system is enforced at process level through standard Linux features such as user and group IDs that are assigned to apps. By default, the *least privilege* principle is applied to applications; they can’t interact with each other and have limited access to the operating system. For example, if a malicious app tries to read data from a legitimate application, it is blocked by the sandbox because it does not have the appropriate user privilege. Since the application sandbox lies at operating system level, the security model is not limited to Java code but also extends to both native code and OS-level applications.

Each new application is installed in a new directory named after the app’s package, which results in the following path: `/data/data/<package-name>`

If a developer develops two different apps but wants them to easily share data with each other, they can do that by signing the two applications with the same certificate and giving them the same UID by setting the `sharedUserId` entry in the apps’ manifests.

2.6 Android application components

The components described in this section are the main logical building blocks of Android applications and any developer has to know perfectly how to use them. In some cases, misuse of those components can lead to security risk and vulnerability, so understanding their behaviors and interactions is important.

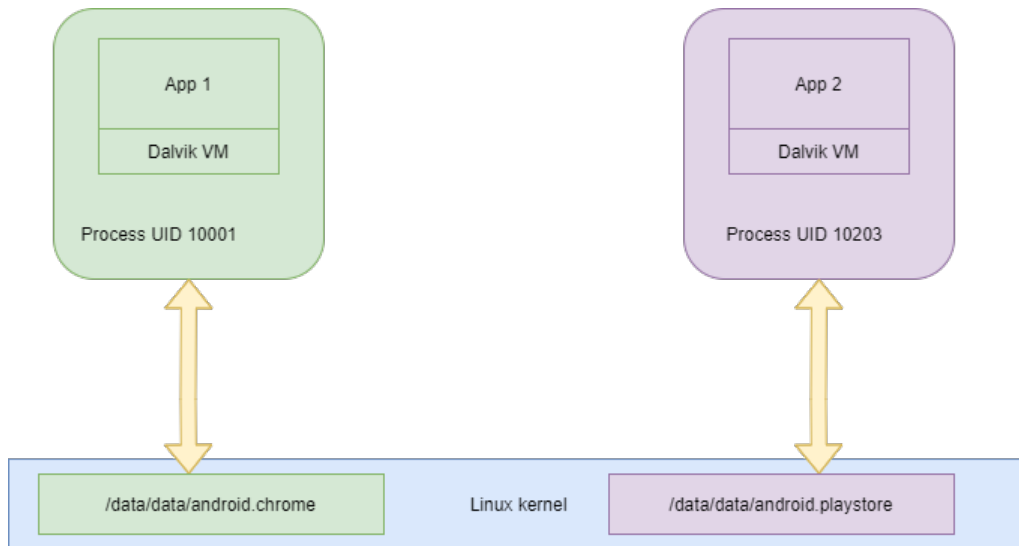


Figure 2.4 – Application Sandbox

2.6.1 Android Manifest

The `AndroidManifest.xml` file is a mandatory component of every Android application and is located at the root of the APK. The manifest file describes the application's structure, its components and the permissions the app requires. The file helps configuring the application and reading it helps understand what the application is doing. In this thesis I will often be referring to this file as simply the "manifest" file, which is not to be confused with the `MANIFEST.MF` file.

2.6.2 Intents

An intent is an asynchronous message to request an action from another component. Typically, an intent is used in three scenarios: starting a service, starting an activity and delivering a broadcast. There are two types of intents: implicit and explicit.

- **Explicit intents** specify the application which will satisfy the intent either by supplying the app's package name or a fully-qualified component class name. This kind of intent is usually used to start a component of the same app that sent the intent.
- **Implicit intents**, on the other hand, don't specify a particular component but declare a general action to perform, allowing a component from another app to handle it. For example, if an application wants to show a specific spot on a map, it can send an intent to request another app to do that.

Intent filters are expressions defined in the manifest file that specify the type of intents that the component would like to receive. When an application sends an implicit intent, Android finds the right component to start by checking intent filters of other applications. If there's a match with a component from another app, that component is started and passed the intent object as a parameter. This mechanism thus only works for implicit intent and if there is no declared intent filter for a component in the manifest, this component can only be started with an explicit intent.

2.6.3 Activities

An activity is basically one screen of the application and thus represents the main way of interacting with the user. They all work independently from each other but they interact to form a cohesive work flow inside the application. An application can start an activity of another application only if the other application allows it; for example, this mechanism allows the camera application to start an activity from an email application to easily share a picture.

From a technical point of view, all activities inherit from the `Activity` class. They contain all other user interface elements: fragments, views and layouts. Every activity must be declared in the Android Manifest as follows:

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

Activities have their own life cycle and switch from state to state in a deterministic manner; those state changes can be manipulated by the developer through the following methods: `onCreate`, `onSavedInstanceState`, `onStart`, `onResume`, `onPause`, `onRestart`, `onDestroy`.

If one of those methods is not implemented by the developer, default actions are taken by the operating system.

2.6.4 Services

A service is an Android component which runs in the background to perform tasks such as downloading a file over the network or playing music. Contrary to an activity, a service does not have a user interface. Services exist in two distinct flavours:

- **Started services** tell the system to keep them running until their job is done. The system will do its best to keep them alive and will kill them last because it would result in bad user experience (e.g. a music player shouldn't stop unless really necessary).
- **Bound services** run to serve another application. They run as a dependency to another process and the system knows that if process A needs process B, it should keep B and its service alive as long as A lives.

2.6.5 Content providers

A content provider is an Android component that abstracts a data source using a URI scheme. This data source can take multiple forms, be it a SQLite database or a file for example. This component is used by an application to perform usual CRUD operations on a specific data source. By default, a content provider created in an application is not accessible by other apps. To share it with another application, the content provider must be declared in the manifest file of the other application that wants to use it.

2.6.6 Broadcast receivers

Broadcast receivers allow applications to receive notifications from the system and from other apps. For example, an application might send a broadcast to other applications to tell them that data were downloaded and are now available for them to use or the system might tell the applications that there is no network connection. A broadcast receiver can, just like an activity, be declared in the Android Manifest:

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

As this example implies, broadcasts are delivered through intent objects and you use intent filters to specify which intents the broadcast receiver accepts.

Another way of creating a broadcast receiver is through code with the method `registerReceiver()` (the following example was taken from OWASP's MSTG [34]):

```
// Define a broadcast receiver
BroadcastReceiver myReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "Intent received by myReceiver");
    }
};

// Define an intent filter with actions that the broadcast receiver listens
// for
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction("com.ulb.myapplication.MY_ACTION");
// To register the broadcast receiver
registerReceiver(myReceiver, intentFilter);
// To un-register the broadcast receiver
unregisterReceiver(myReceiver);
```

To make sure that an application will not receive broadcasts from other applications, the developer can use a local broadcast manager. This ensures that all received broadcasts originated from the application itself and that broadcasts coming from other applications will be discarded. This increases security as no inter-process communication will take place.

2.7 Zygote

On Android devices, all processes are forked from the Zygote process. This process is started very early during the boot process and contains all the core libraries necessary for running applications. Once started, this process opens the socket `/dev/socket/zygote` and waits for incoming messages from local clients. Upon reception of such a message, it forks a new process which then loads and executes the application.

2.8 Summary

The concepts described in this chapter are of prime importance to understand the rest of this thesis. As stated before, the Android operating system is a complex machinery and its description focused on the development and deployment of applications to keep things relevant.

The reader should now understand the basics of the Android operating system, what the building blocks of an Android application are and how such an application is assem-

bled. The reader might also start to see what can be done to attack such a system and thus what could be done to protect it. Each of the components described in this chapter could potentially introduce vulnerabilities or be exploited in one way or another to perform malevolent actions on a device. The next chapter will explore this point of view by providing an overview of the different threats to the Android system.

Chapter 3

Threat model

3.1 Introduction

This chapter will provide a detailed description of the different static and dynamic analysis tools and techniques used by attackers to compromise an application. The given descriptions will sometimes be accompanied by examples of practical use. Most of those examples are inspired from the OWASP's Mobile Security Testing Guide (MSTG) [34]. It is important to keep in mind that the techniques described here represent only a fraction of the potential threats to Android systems and applications. This chapter should, however, give the reader a good overview of the possible attacks to Android applications, which is important to understand the mitigation techniques described in the next chapter.

3.2 Device rooting

Rooting (or jailbreaking) a device gives the user full control over its operating system. It allows the user to bypass restrictions such as the application sandbox. Those new privileges allow the user to use certain tools such as Frida to do code injection or function hooking more easily.

Rooting can be classified in two main categories as discussed in Long N. et al's paper [30] on Android rooting:

1. **Soft root** which is gaining privileged access on a booted system. This technique is entirely software-based and relies on the exploitation of vulnerabilities in the kernel or in a running process for example. This kind of rooting gave birth to "one-click rooting" applications. Rooting done in this fashion is usually temporary, mostly in devices which have a locked bootloader, as it checks the integrity of the system everytime it boots. To persist the root privilege, the bootloader must be unlocked and the recovery mode must undergo some manipulations.
2. **Hard root**, in contrary to soft root, requires physical interaction with the device. It works by booting the device in recovery mode and installing `su` or replacing the entire operating system with a new one that has `su` installed. On some devices the bootloader is locked, which makes the rooting process more difficult. The main advantage of this technique over soft rooting is that it is not temporary because it requires unlocking the bootloader which makes the rooting permanent.

A wide variety of tools exist in the open-source community and can take multiple forms: an APK, a binary to install on the system, etc ...

3.2.1 Magisk

Magisk [9] is probably the most popular rooting tools on the market at the moment. The reason for its popularity lies in the way modifications on the system are made. It offers a mode of operation wich is called *systemless* in which it does not make visible changes

to data on the system partition, but instead stores its modifications in the boot partition. This particularity offers the advantage of hiding all rooting related modifications from root-sensitive applications while allowing to use the official Android OTA (Over The Air) updates without unrooting the device first.

On top of its main rooting feature, Magisk also offers the application *Magisk hide* which allows more fine grained control over the rooting. With this application, the user can select which application installed on the device should not see Magisk and it will then hide its presence from the selected applications to evade the root detection controls inside those apps. To hide itself from a selected app, Magisk needs a way of detecting the app's start. Over the years, Magisk used different techniques to achieve this result:

Logcat monitoring. Logcat is a command line tool shipped with the Android SDK which dumps system and application information. Whenever an application starts, Logcat will log specific messages that are then used by Magisk hide to detect the app's process start. This method requires constant monitoring of system logs and thus introduces some overhead.

inotify. `inotify` is a Linux feature that allows to report changes in the file system to applications. Whenever an app starts, its APK file will be accessed from the file system. It is thus possible to use `inotify` to detect the `open()` system call to the target the APK to detect when the app's process starts.

ptrace. As explained in chapter 2, all applications' processes on Android are forked from the Zygote process. Therefore, it is possible to monitor all processes' start by attaching `ptrace` to Zygote and stepping through all fork events.

3.3 Emulators

An Android emulator imitates an Android device by reproducing virtually both its software and hardware architecture. Using an emulator offers advantages over a physical device, but also comes with downsides. The list of advantages include:

- Snapshot support
- Rooted by default
- Easy to restore if corrupted

The main downside of using an emulator is the performance overhead as emulators are most of the time way slower than physical devices. They also have limited hardware interactions, which might lead to errors when running certain native libraries because of the type of CPU the emulator uses.

3.3.1 QEMU

QEMU [18] based emulators offer complete virtualization of the device by taking into consideration RAM, CPU, battery performance and other hardware restrictions. The QEMU emulator can emulate different type of CPUs (x86, PowerPC, ARM and Sparc) which makes it a very versatile tool. The tool is useful for debugging because the state of the virtual machine can be easily stopped, saved or restored. QEMU is composed of several subsystems:

- CPU emulator (x86, PowerPC, ARM and Sparc).
- Emulated devices (e.g. VGA display, 16450 serial port, PS/2 mouse and keyboard, IDE hard disk).
- Generic devices used to allow communication between the emulated and the host device.
- Machine descriptions used to instantiate the emulated devices.
- Debugger.
- User interface.

This emulator or parts of it were used by other well known emulator projects such as VirtualBox [39] which integrates QEMU's recompiler or used QEMU's virtual hardware devices as a starting point for their own. It also powers the Android emulator project which is part of the Android SDK and is thus the by-default emulator for a big part of the Android developer community.

3.3.2 Kernel-based virtual machine

Kernel-based virtual machine (KVM for short) [7] is a Linux kernel module and a type 1 hypervisor allowing full virtualization. Unlike QEMU and type 2 hypervisors in general, the CPU is not emulated by the hypervisor which reduces the performance overhead. This module works by adding a character device (`/dev/kvm`) that exposes virtualization capabilities to userspace. With this driver, a process can run a virtual machine containing all the normal components of a virtualized system. Each virtual machine is therefore a process on the host and a virtual CPU is a thread in that process. Since QEMU's hardware virtualization is slow by nature, it uses KVM as an accelerator to increase performance.

3.4 Reverse engineering and static analysis techniques

Reverse engineering is the process of reconstructing a compiled piece of software which is not made to be human readable. Java or Dalvik bytecode can be easily decompiled back to a source code really close to the original Java code; if the application does not use additional native code, the application testing is then almost similar to white box testing. Native code reverse engineering, on the other hand, requires more skills because the disassembled code is far from the original source code.

Static analysis is the process of understanding the program's behavior without running it, as opposed to dynamic analysis. In a black box context, it is closely related to reverse engineering since in order to understand code from an APK an adversary must first reverse it.

In this section I will introduce the processes and tools used by attackers to reverse engineer and statically analyse Android applications.

3.4.1 Decompiling Java code

The most straightforward way of reversing Java code from an APK is to open it directly in Android Studio by clicking **Profile or debug APK** from the welcome screen. The

downside of this approach is that Smali code, which is harder to understand than Java code, is created from the bytecode.

Another tool that also produces Smali from Dalvik bytecode is `apktool` [2]. On top of that, it can reassemble the package of the application, which is useful for patching and applying changes to the Android Manifest for example.

Smali code is easier to understand than actual bytecode but it is still more difficult to understand than Java code. To produce Java from bytecode, a disassembler is needed and hopefully Java disassemblers usually work well with Android bytecode. Common free decompilers include:

- JD [21]
- jadx and jadx gui [36]
- CFR [19]

Once an APK is downloaded, an attacker can recompose the source code of an application to analyse it in order to find what defenses are in place and where they could potentially inject malicious code. `Dex2jar` [35] can convert the dex file of an APK to a jar which can then be decompiled and inspected.

3.4.2 Disassembling native code

Java code can communicate with native code written in C or C++ through the JNI which is supported by the Dalvik virtual machine and ART. To run on Android as on other Linux operating systems, native code is compiled into an ELF dynamic library with the extension *.so. The developer can then interact with this library by calling the `System.load()` function from Java code. Contrary to traditional Linux systems, Android does not rely on widely used C library such as glibc but instead uses a custom libc called Bionic. This library has been specifically designed by Google to run on devices with less memory and processor power than typical Linux systems.

When reversing an Android application, it is important to well understand a data structure: `JNIEnv`; this data structure is a pointer to pointers to function tables. It provides access to most common JNI functions which are accessible at a fixed offset through the `JNIEnv` pointer. This pointer is passed to every JNI function as the first parameter.

In order to reverse engineer native code, one needs any decompiler. There exists a wide variety of those tools but to give some examples, the most common ones are:

- IDA pro
- ghidra
- radare2

3.5 Dynamic analysis

As a defense against static analysis, developers will obfuscate the source code of their app before packaging it into an APK file; in this case, an attacker won't be able to perform static analysis and will have to run the application to study its behavior at runtime. It also allows the adversary to discover business logic flaws or certain vulnerabilities more easily than with static analysis.

Dynamic analysis can be done either on a rooted or non-rooted device. Device rooting is used a lot for dynamic analysis because it is required by a lot of tools and it offers complete control over the Android system.

3.5.1 Debugging

Debugging is probably the most well-known kind of dynamic analysis as it is common practice when developing software. Debugging tools are also useful to an attacker as they allow to explore memory at runtime and thus effectively bypass obfuscation. There are two different approaches when debugging Java applications on Android: the first one is to debug on the Java level with the Java Debug Wire Protocol (JDWP) and the second one is to debug on the native level through a ptrace-based debugger.

Java debugging

To debug Java code without having access to the source code of the app, one can use the jdb debugger and the Android utility adb. To do so, we must first open a listening socket on the host to forward the socket's incoming connections to the JDWP transport of a chosen process on the target device:

```
adb forward tcp:7777 jdwp:12345
```

Here 12345 is the process id that we are trying to debug. Next we want to attach the jdb debugger to the correct process:

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

The debugger is now attached to the correct process, all threads are suspended and we can start using it to explore memory at runtime. Jdb offers a lot of features, for example:

- list all loaded classes
- print details about a class and list its methods and fields
- print local variables an current stack frame
- print information about an object
- set a method a breakpoint
- remove a method breakpoint
- assign new value to field, variable or array element

Native code debugging

Since native code is compiled into regular ELF libraries, we can use the regular set of tools used to debug native libraries such as gdb for example. In order to debug native code, the Android NDK (Native Development Kit) must be installed on the host as it provides the `gdbserver` utility which is used to debug processes on the smartphone. Next we must copy the `gdbserver` binary to the testing device:

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

Then we attach gdbserver to the right process:

```
/data/local/tmp/gdbserver --attach localhost:1234 12690
```

Here 12690 is the id of the process we want to debug. Gdbserver is now correctly attached and is waiting for debugging clients on port 1234. We can now use adb to forward this port to a local port on the host:

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

The last step of this process is to use the prebuilt instance of gdb included in the NDK toolchain to perform the debugging:

```
$TOOLCHAIN/bin/gdb libnative-lib.so
```

We can now use all the features of gdb, including the important `target remote :1234`, indicating that we are debugging a remote process from port 1234.

3.5.2 Tracing

Function tracing or code instrumentation is a technique allowing to print the execution of a program by printing information such as the list of methods executed, time spent in those methods or even their parameters and return values.

In some cases (e.g. Frida or Xposed), tracing tools also allow to modify methods parameters and return values, which is a very powerful capability for an adversary. Such a technique allows to easily bypass some programmatic checks. For example we could imagine tampering with a specific function checking the validity of a X509 certificate and simply setting its return value to true.

Jdb execution tracing

Jdb, the java debugger discussed in section 3.5.1 offers the ability to dump all method entries and exits after a specific breakpoint with the command `trace go methods`.

Tracing system calls with strace

The previous methods worked on the application level but it is possible to also trace system calls to the Linux kernel with the strace utility. Intercepting such calls to the kernel gives a good overview of what a user process is doing and can also allow adversaries to deactivate low level tampering defenses.

Strace is a Linux utility not included with the Android SDK but can be built from source with the Android NDK. Since strace is internally using the `ptrace` system call to attach to the target process, it is vulnerable to anti-debugging measures which prevent it to work.

Tracing system calls with ftrace

Ftrace is another tracing utility directly included in the Linux kernel. This tool requires a rooted device but offers more transparency than strace because it does not use the `ptrace` system call.

This utility is installed by default on most Android versions and can be easily enabled:

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

Ftrace log files and directories are all located in the `/sys/kernel/debug/tracing` directory.

Method tracing with frida

Frida [4] (which will be described in section 3.7.1) is an open-source toolkit which provides dynamic code instrumentation by injecting the QuickJs [10] Javascript engine into the instrumented process. Its aim is to allow Javascript execution inside native apps on Android and IOS.

Using *frida-trace*, it is trivial to perform method tracing:

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

In the above command, the `-U` option is used to tell Frida to connect to the USB device, the `-i` option is used to specify the name of the function to trace and `com.android.chrome` is the fully qualified name of the application to monitor.

```
$ frida-trace -U -i "Java_*" com.android.chrome
```

Here, *frida-trace* will monitor all JNI methods as they all start with the *Java_* string by using a regular expression.

Oftentimes binaries will be stripped (i.e. their debugging information have been removed to shrink their size) and thus do not contain method names. In this case, the user can trace a method with its memory address as shown below:

```
$ frida-trace -U com.android.chrome -a "libjpeg.so!0x4793c"
```

3.5.3 Symbolic execution

According to Wikipedia [14], symbolic execution is *a means of analyzing a program to determine what inputs cause each part of a program to execute..* It translates the program's semantic into a logical formula in which some variables are represented by symbols with specific constraints. This formula is then used as an input by a constraint solver which finds the right values for those symbols to trigger each execution path.

In other words, symbolic execution uses mathematics to analyze a program without actually running it. Execution of the program is done by a *symbolic execution engine* which maintains, for each explored control flow path:

1. A first order Boolean formula that contains the conditions satisfied by the branches taken along that path.
2. A symbolic memory store that maps variables to symbolic expressions.

In the end of the process, a model checker based on a Satisfiability Modulo Theories (SMT) checker is used to verify if the final formula can be satisfied by some assignment of actual values to the program's symbolic arguments.

As powerful as this approach is, it still has downsides:

- Loops and recursions can cause the analysis to never return a value.
- A big code base with multiple conditional branches or nested conditions may lead to path explosion.
- In some cases, complex equations generated by the symbolic execution engine may not be solvable by the model checker because of their current limitations.
- The symbolic execution engine does not handle well system or library calls or network events.

Those downsides can be mitigated by combining symbolic execution with other techniques such as dynamic (aka concrete) execution. This specific combination of concrete and symbolic execution is referred to as concolic (**con**crete and **sym**bo**lic**) execution. It is also possible to combine reverse engineering or static analysis with symbolic execution to start the symbolic execution at a specific place in the program rather than starting at its entry point.

Although multiple symbolic execution tools exist, the most common is Angr [1]. According to its documentation page, Angr is *a python framework for analyzing binaries. It combines both static and dynamic symbolic ("concolic") analysis, making it applicable to a variety of tasks.* It is a very versatile tool which is not limited to symbolic execution; its features include also disassembly, program instrumentation, control-flow analysis, data-dependency analysis, decompilation thanks to a large set of plugins.

3.6 Tampering and repackaging

Tampering means modifying the source code of the application to change its behavior. It has a lot of different use cases which include bypassing security controls or adding new features to the application. Repackaging means making the modified application ready for deployment and use.

In this section we will have a look at the most common techniques to leverage tampering, i.e. Android manifest manipulation, Smali code patching, Java code injection and native code injection, as well as the repackaging process and tools.

3.6.1 Android manifest patching

Patching an application's manifest allows an adversary to add certain permissions to the application or to make it debuggable. Simply uncompressing the APK does not allow us to edit the file as it is encoded in Android's binary XML and we need `apktool` to decode it:

```
$ apktool d --no-src target.apk
```

Here the `d` option specifies that we want to decode `target.apk` and `--no-src` means that we don't want to disassemble the DEX file.

Now that the file is editable we can add `android:debuggable="true"` in the application tag:

```
<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">
```

3.6.2 Smali code patching

As discussed in section 3.4.1, `apktool` can be used to decompile Dalvik bytecode to Smali source code:

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

This resulting Smali code can then be modified with a simple text editor:

```
.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
```

This example was taken from OWASP’s MSTG [34] and describes the patching of the method `checkServerTrusted` which is responsible for verifying that the server to which the app connects to over SSL is trusted. The patching modifies the method’s return value to make it return immediately, effectively bypassing the check.

Once the Smali file has been modified, we can convert the application’s files back to Dalvik bytecode with the command:

```
$ apktool b target_apk
```

Where `target_apk` is the folder containing the modified Smali files.

3.6.3 Java Bytecode manipulation

In the previous section we discussed how to manipulate Smali code which is an intermediary representation of Java bytecode, but it is also possible to modify Java bytecode directly thanks to various libraries and frameworks. The most common are:

ASM From the official documentation [3], *ASM is an all purpose Java bytecode manipulation and analysis framework*. It is a mature library which offers tons of features and is well documented. It has been used in a variety of different projects such as the OpenJDK, the Groovy and Kotlin compiler or Gradle to name a few. It makes bytecode manipulation possible through an API which, although well documented, requires thorough knowledge of the Java class file format.

Soot According to its official github page [13], *Soot is a Java optimization framework*. It allows the developer to analyze and optimize java bytecode by offering four different

bytecode representations.

Javassist Javassist's goal is to make Java bytecode manipulation simple [6]. It has comprehensive and to-the-point tutorials allowing any new user to get started quickly. The strength of javassist resides in its two levels API: source and bytecode level. The source level API allows users to edit a class file without knowledge of the specifications of Java bytecode, and if the user wants to have more control on the operations performed, the bytecode level API allows to edit bytecode more directly.

Below you will find an example of bytecode manipulation of an Android application with Javassist:

```
pool = ClassPool.getDefault();
pool.insertClassPath("c:\\Users\\username\\AppData\\Local\\Android\\platforms\\
android-30\\android.jar");
pool.importPackage("com.test");

mainActivity = pool.get(mainActivity);
onCreate = mainActivity.getDeclaredMethod("onCreate");
```

A `ClassPool` is a container of `CtClass` object representing a class file. It reads a class file on demand for constructing a `CtClass` object and records the constructed object for responding later accesses. Then, the method `insertClassPath()` registers the class or jar path that was used for loading the class that is given in parameter. In this example, the provided path corresponds to the `android.jar` file as it is needed to use the Android API.

The method `pool.get(mainActivity)` will return a `CtClass` object representing the main activity that we can extract from the app's manifest. Inside this activity, the `onCreate()` method is the first method called when the activity starts and I can get a reference to that method by calling `getDeclaredMethod("onCreate")` on the `CtClass` object representing the main activity.

Now that we have a reference to the `onCreate` method, we can inject it with whatever code we want:

```
String injected_code_stmt = "{Log.d(\"ULB\", \"Injected code\")}";
onCreate.insertBefore(injected_code_stmt);
mainActivity.writeFile("output.test");
```

One of the downside of working with javassist is that bytecode manipulation is done through strings. When one wants to inject code in a class file, the code to inject must be in the form of a String type variable between curly braces. To actually inject the code to the class file, we use the `insertBefore` method with the statement to be injected passed as a parameter. This method will inject the provided statement at the beginning of the specified method, in our case `onCreate`. Finally, the `writeFile` method will write the modified class file of the object `mainActivity` to the path specified in parameter.

3.6.4 Java library injection

Thanks to `dex2jar` [35], it is possible to convert an APK to a jar file. A jar file being a compressed file format, it is possible to uncompress it to recover its content. A jar is composed of folders representing the different packages, each containing their corresponding class files. In order to inject any library, we need to first compile the wanted Java code

to bytecode with a Java compiler i.e. `javac` if we want to do it manually or an IDE e.g. IntelliJ. Then we need to create a folder with the right package name inside the uncompressed jar folder and finally we copy the class files we want to inject inside that folder.

The last step is to compress the entire jar again and convert it back to the `classes.dex` file with a dex compiler such as `dex` or `d8`.

```
$ dx --dex classes.dex modified_jar.jar
```

Here, `classes.dex` is the output file and `modified_jar.jar` is the input jar we want to convert to DEX bytecode. We can then put the resulting `classes.dex` file back in the uncompressed APK folder structure and repackage it to a fully functional APK as described in section 3.6.6.

This section and the previous one can both be automated and combined to inject any classes we want into an APK and then modify its bytecode to call methods from the injected classes. It is a common method for malware development but it can also be used to add new features to an application or to protect it.

3.6.5 Native library injection

Smali patching and bytecode manipulation

There are different techniques to inject a native library inside a target APK. We can use the Smali code patching technique described in section 3.6.2 for example:

```
const-string v0, "inject"  
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

This example shows the injection of a call to the method `System.loadLibrary()` which loads the library `libinject.so`. Keep in mind that the injected library must be placed in the right architecture folder (i.e. `armabi-v7a`, `arm64-v8a` or `x86`) of the `lib` folder in the APK for this technique to work.

We can achieve the same result with the bytecode manipulation technique described in section 3.6.3:

```
String injected_code_stmt = "{System.loadLibrary(inject)}";  
onCreate.insertBefore(injected_code_stmt);  
mainActivity.writeFile("output.test");
```

In this example, `onCreate` is a `CtClass` object representing a reference to the `onCreate` method of the `mainActivity`.

ELF format manipulation

Native code is compiled in a shared library (s.o. file extension) which is itself an ELF file. An ELF executable includes a list of other shared libraries that are linked to the executable in order for it to work. This list can be modified to include arbitrary dependencies which will then be injected in the process.

The ELF file format was not created to be human readable and its manual modification is thus not a trivial task. Fortunately, this can be done more easily using the LIEF (Library to Instrument Executable Formats) library in Python:

```
import lief

libnative = lief.parse("libnative.so")
libnative.add_library("libinject.so") # Injection!
libnative.write("libnative.so")
```

In this example, `libinject.so` is added as a dependency of the library `libnative.so`.

Preloading symbols

All the tampering techniques described until now used code modification to add a native library in an application. It is also possible to use the loader of the underlying operating system to achieve the same result. On Linux, it is possible to load an additional library with the `LD_PRELOAD` environment variable.

Symbols loaded by the library set by this environment variable have top priority, meaning that the loader will first look into this library when resolving symbols, overriding the original ones. This allows an adversary to wrap common libc functions such as `fopen`, `read`, `write` or `strcmp` in order to print their parameters for example. This is particularly useful if they are dealing with an obfuscated library where understanding low level functions can help better understand the library's behavior.

On the Android system, this mechanism varies slightly because of the Zygote process. Each application is forked from this process which is started early during the system boot up. Simply setting `LD_PRELOAD` on Android is thus not possible but there is a workaround. The `setprop` feature lets you do the same thing:

```
$ setprop wrap.com.foo.bar LD_PRELOAD=/data/local/tmp/libpreload.so
```

Where `wrap.com.foo.bar` is the package name of the application we want to load the library into.

3.6.6 Repackaging

Once we have made the changes we wanted to the application, we have to repackage it and sign it to make it ready for deployment. To repackage the app we can either compress its content folder or we can use `apktool`:

```
$ apktool b content_folder -o modified_target.apk
```

This command builds the content folder into the `modified-target.apk`. Next we must use the `zipalign` utility included in the Android SDK at this location: `[SDK-Path]/build-tools/[version]` to align its resources correctly (not doing this step can fail the installation of the application on a device).

```
$ zipalign -v 4 modified_target.apk aligned_target.apk
```

The last step of the repackaging process is to sign the APK with a valid certificate. The JDK provides a utility for managing keys and certificate called `keytool`. The following command creates a keystore and a key and stores it in the keystore:

```
$ keytool -genkey -v -keystore ulb.keystore -alias signkey -keyalg RSA
    -keysize 2048 -validity 20000
```

Now that the key is created, we can use it with the `apksigner` utility provided with the Android Sdk at the location: `[SDK-Path]/build-tools/[version]`:

```
$ apksigner sign --ks ulb.keystore --ks-key-alias signkey aligned_target.apk
```

The application is now repackaged, signed and ready for deployment; we can install it on a device with `adb`:

```
$ adb install aligned_target.apk
```

3.7 Dynamic instrumentation

3.7.1 Frida

We already discussed some of the capabilities offered by Frida tool in section 3.7.1 and I will now explain how it works internally.

Code injection is a vast topic and a lot of different techniques exist. For example, Xposed [15] permanently modifies the Android app loader, providing hooks for running your own code every time a new process is started. On the contrary, Frida injects code directly in the process memory. Once attached to a process:

- Frida uses `ptrace` internally to hook a running process. It is then used to allocate a chunk of memory to install a mini-bootstrapper.
- The bootstrapper then starts a new thread, connects to the Frida debugging server which was previously installed and started on the device and loads a shared library that contains the Frida agent (`frida-agent.so`).
- The agent creates a bi-directional communication channel back to the tool (e.g. your custom Python script)
- The hooked thread restores its original state and resumes to allow normal process execution.

Frida offers 3 different modes of operation:

Injected: This mode of operation is very flexible and offers a lot of capabilities. It lets the user either spawn a new program, attach to an already running program or hijack one as it is being spawned. Other features include listing all installed applications, running processes and connected devices. A connected device is typically an Android (or IOS) device where *frida-server* is running. *Frida-server* will expose *frida-core* over TCP listening on `localhost:27042` by default. To use this mode of operation the device needs to be rooted.

Embedded: This mode lets the user use Frida on jailed (i.e. non rooted) devices by injecting the shared library *frida-gadget* into the target apk. Simply loading the library lets the user interact with it remotely using tools from the Frida toolkit such as *frida-trace*. This mode also offers the ability to run scripts from the filesystem without any outside communication.

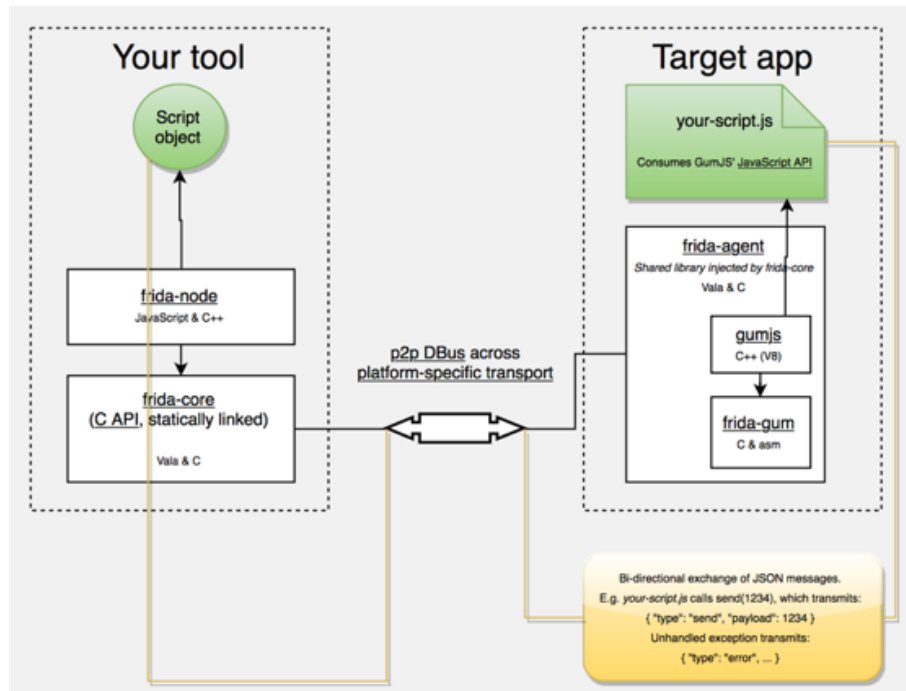


Figure 3.1 – Frida’s internal architecture taken from Frida’s documentation

Preloaded: This mode uses the same logic as `DYLD_INSERT_LIBRARIES` and `LD_PRELOAD`. Those two variables allow the user to specify the path to libraries which will be loaded before any other. If Frida is used in this mode of operation, frida-gadget will run autonomously and load a script from the filesystem.

To perform complex operations with Frida, it is possible to write custom scripts in Javascript.

3.7.2 Xposed

Xposed is a framework that allows the user to import modules to the ROM. Those modules allow to modify the behavior of the system or applications at runtime without modifying APKs or re-flashing a new ROM. From a technical standpoint, Xposed extends the Zygote process and exports APIs that allow Java code execution when a new process is started. This additional Java code is executed in the context of the instrumented process, which makes it possible to hook and override Java methods belonging to the started application. The changes made to applications are applied in memory and are therefore not persistent as they last only during the app’s execution.

This tool requires a rooted device to work and modules installation is done through the Xposed installer app, which offers a nice GUI for modules management. Just like Frida disposes of a wide variety of open-source scripts, Xposed benefits from a long list of modules created by the open-source community. Creating custom modules can be a powerful asset to have in the toolbox as they can be used to bypass security measures of Android applications.

Method overriding

Using Frida it is possible to override a method to arbitrarily modify its content. For example, an adversary could override a method performing a root check on a device:

```
setImmediate(function() { //prevents timeout
    console.log("[*] Starting script");

    Java.perform(function() {
        var mainActivity = Java.use("com.ulb.MainActivity");
        mainActivity.rootCheck.implementation = function(v) {
            console.log("[*] MainActivity.a called");
        };
        console.log("[*] MainActivity.a modified");

    });
});
```

In the above example, the function `setImmediate` calls the function passed in parameter as soon as possible on Frida's Javascript thread. The function `Java.perform()` is needed by Frida to interact with the JVM and Java methods. Then a wrapper for the `MainActivity` class is created which allows the adversary to retrieve and manipulate its methods. A reference to the methods performing the root check is retrieved with `mainActivity.rootCheck.implementation` and its body is then overridden by a function performing a simple `console.log()`, effectively bypassing the check implemented in the initial method.

Let us name this script `root_detection_bypass.js` and run it with Fria:

```
$ frida -U -f ulb.com.vulnerable_application -l root_detection_bypass.js
--no-pause
```

Return value interception

With Frida, an adversary can intercept return values of specific methods. In the following example we will consider an hypothetical class called `DecryptionHelper` that possesses a method called `decrypt(String encrypted_value, byte[] key)` that takes two parameters: the value to decrypt and the decryption key. The value of this method is never actually returned to the user but its return value is used to check if the provided password corresponds to the expected password. A straight forward approach to find the right password would be to brute-force it or reverse engineer the decryption routine but both approaches take time and effort. A better technique would be to intercept the return value of the `decrypt` method and return it:

```
setImmediate(function() { //prevent timeout
    console.log("[*] Starting script");

    Java.perform(function() {
        var decryptionClass = Java.use("com.ulb.DecryptionHelper");
        decryptionClass.decrypt.implementation = function(arg1, arg2) {
            var retval = this.decrypt(arg1, arg2);
            var password = '';
            for(var i = 0; i < retval.length; i++) {
                password += String.fromCharCode(retval[i]);
            }
        }
    });
});
```

```

        console.log("[*] Decrypted: " + password);
        return retval;
    };
});
});

```

In the above example, we first decrypt the value by calling the original `decrypt` method, then we convert it to a `String` object with the `for` loop and we finally return the decrypted value which we can then use as our password.

An other example would be bypassing a root check. Let's assume the target application has a root detection mechanism which crashes the application if the device is rooted. Let us consider an hypothetical application with package name `com.example.target` which possesses a class `Security` which itself implements a `rootcheck` method. This security mechanism can be bypassed with the following Xposed module.

```

package com.ulb.be;

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.target"))
            return;

        findAndHookMethod("com.example.target.security.rootcheck",
            lpparam.classLoader, "c", new XC_MethodHook() {
                @Override

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

            });
    }
}

```

An Xposed module can have several entry points. In this example, the entry point (or the moment this module is loaded if you will) is specified with the `IXposedHookLoadPackage` interface which indicates that this module will trigger whenever an application is loaded. The first line of the `handleLoadPackage` checks whether the package name of the loaded application corresponds to the package name of the target application. The function `findAndHookMethod` is responsible for hooking the right method in the target application. In our case, this method is `com.example.target.security.rootcheck` and is passed as the first parameter of `findAndHookMethod`. The last argument to this

method must be an instance of `XC_MethodHook` which, in this example, is instantiated as an anonymous class. Inside that class, the `beforeHookedMethod` is overridden so that it always returns false, bypassing the root check.

3.8 Summary

In this chapter different tools and techniques allowing to perform static and dynamic analysis were showcased. The reader should now have a clear understanding of the technical details of the attacks discussed in this chapter and the damage they can do to Android applications. Everything that was discussed in this chapter is used intensively by adversaries to bypass security measures implemented in applications or to develop malwares for example. New tools and techniques are developed regularly and even though the presented list is not exhaustive, it still represents a good overview of threats to Android applications.

Understanding this threat landscape helps us grasp what defense mechanisms should be put in place to protect applications against those attacks. A state of the art description of protections against those attacks will be discussed in the next chapter.

Chapter 4

Application security

4.1 Introduction

This chapter will present the different techniques used to protect against the threats discussed in previous chapter. For example, making sure that applications run in a safe environment (i.e. not on an emulator or rooted device) is of prime importance as unsafe environments give full control of the device to adversaries. Dynamic analysis through debuggers and instrumentation tools is another powerful tool in the attackers' toolkit. Fortunately dynamic analysis utilities can be detected and sometimes even prevented from working by a variety of different techniques that will be showcased. Even if static analysis is the most basic technique an attacker could use, it is still very useful to understand the behavior of an application. The different obfuscation techniques described in this chapter mitigate static analysis by modifying the application's code to make it less readable and understandable. Finally, the ARMAND anti-tampering scheme will be presented.

All these security mechanisms are important to understand to implement them in an application afterwards. An application extended with those security measures will be able to respond accordingly if it detects a threat, and it will make sure to make as difficult as possible any reverse engineering effort against it.

4.2 Rooting detection

4.2.1 SafetyNet

SafetyNet is a Google API that offers a set of different services as APIs. The SafetyNet attestation API checks the integrity of a device. The documentation [26] states that this API *"should be used as a part of your abuse detection system to help determine whether your servers are interacting with your genuine app running on a genuine Android device"*. It works by creating a profile of the device which correlates hardware and software information. The profile is then compared to a list of device profiles that have passed Android compatibility testing. A response is then sent back to the application that indicates if the device successfully passed the test. The following response example is taken directly from Google's documentation:

```
{
  "timestampMs": 9860437986543,
  "nonce": "R2Rra24fVm5xa2Mg",
  "apkPackageName": "com.package.name.of.requesting.app",
  "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
                                certificate used to sign requesting app"],
  "ctsProfileMatch": true,
  "basicIntegrity": true,
  "evaluationType": "BASIC",
}
```

The two most relevant fields in this response are `basic integrity` and `ctsProfileMatch`. `BasicIntegrity` is set to true if the general integrity of the device and its API passed the SafetyNet test. Many rooted devices fail `basic integrity` as most emulators or devices with signs of tampering are detected. `CtsProfileMatch`, on the other hand, will be set to false to indicate devices that were not certified by Google. For example, devices with a custom ROM, with an unlocked bootloader or with a system image that was built from Android source files. Those two fields are complementary but `ctsProfileMatch` offers a stricter indication on the device's integrity.

4.2.2 Programmatic detection

File existence check

One of the most used techniques to determine if a device is rooted or not is to check for common rooting scripts or apps presence on a file system.

For example, the Superuser APK is an Android application that roots the device when installed. As a developer, you can check if this application is installed on the device; if it is, the device is most probably rooted. Although this rooting detection technique is powerful and efficient, the file list needs to be updated regularly to include new suspicious files to look for.

```
/system/app/Superuser.apk
/system/etc/init.d/99SuperSUDaemon
/dev/com.koushikdutta.superuser.daemon/
/system/xbin/daemonsu
```

Once a device is rooted, those rooting utilities often import additional binaries which you can look for additional evidence of rooting. You should look for busybox and the su binary in multiple places as its installation path often changes.

```
/sbin/su
/system/bin/su
/system/bin/failsafe/su
/system/xbin/su
/system/xbin/busybox
/system/sd/xbin/su
/data/local/su
/data/local/xbin/su
/data/local/bin/su
```

Checking if su is on the PATH

Another way to look for su on a device is to programmatically check the PATH variable.

```

public static boolean checkRoot(){
    for(String pathDir : System.getenv("PATH").split(":")){
        if(new File(pathDir, "su").exists()) {
            return true;
        }
    }
    return false;
}

```

Executing su and other commands

If you have concerns a certain rooting utility such as su might be installed on the device, a straight forward way to verify that is to call the `Runtime.getRuntime().exec` method. If the binary is not on the PATH an `IOException` will be thrown.

Checking running processes

On top of installing files or binaries, rooting tools often run daemon processes which can be detected by inspecting what is running in memory. Running processes can be inspected with the `ps` command, with the `ActivityManager.getRunningProcesses` and `manager.getRunningServices` APIs or by browsing through the `/proc` directory. The following example is taken from the open source project `rootinspector` [11]:

```

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

Another technique is to use the Android package manager to list all the installed packages on the device. Following is a list of package names that belong to popular rooting tools:

```

com.thirdparty.superuser
eu.chainfire.supersu

```

```
eu.chainfire.supersu  
com.koushikdutta.superuser  
com.zachspng.temprootremovejb  
com.ramandroid.appquarantine  
com.topjohnwu.magisk
```

Checking for writable partitions and system directories

Normally, system and data directories are mounted read-only so if you happen to find them mounted read-write, that might be a sign the device is rooted. To test this, one can look for the file systems mounted with the "rw" flag or try to create a file in data directories.

Checking for custom Android builds

When the Android kernel is compiled, the `tags` entry in the file `/system/build.prop` are edited to indicate the type of compilation. If the kernel was compiled with an official developer key, the entry will indicate a release-key. On the other hand, if the kernel was compiled with a custom generated key, the entry will indicate a test-key. This entry can easily be checked in Java like so:

```
private boolean isTestKeyBuild() {  
    String str = Build.TAGS;  
    if ((str != null) && (str.contains("test-keys"))) {  
        return true;  
    }  
  
    return false;  
}
```

4.3 Debugging detection

As mentioned in section 3.5.1, there exist two debugging protocols on Android: JDWP debugging and native level debugging. Debugging being a powerful tool for an attacker, good debugging detection and prevention mechanisms can seriously impact the attacker's motivation to tamper with an application. In order to have strong debugging protection, multiple defenses on the two types of debugging should be applied to an application.

4.3.1 JDWP anti-debugging

JDWP is the protocol used for communication between the debugger and the Java Virtual Machine. Even if it is disabled by the application's developer, an attacker can easily enable it by patching its manifest file and changing the `ro.debuggable` system property which enables debugging for every application.

Checking the debuggable flag in Application info

In the manifest file, the `android.debuggable` attribute determines whether the JDWP thread should start for the application. It is possible to retrieve its value programmatically

by using the `ApplicationInfo` object. If the flag is set, the manifest has been tampered with and now permits debugging.

```
public static boolean isDebuggable(Context context){

    return ((context.getApplicationContext().getApplicationInfo().flags &
        ApplicationInfo.FLAG_DEBUGGABLE) != 0);

}
```

µ

Checking if a JDWP debugger is attached

The `android.os.Debug` class offers a method to determine if a debugger is connected.

```
public static boolean detectDebugger() {
    return Debug.isDebuggerConnected();
}
```

It is possible to perform the same check on the native level by accessing `DvmGlobals` global structure.

```
JNIEXPORT jboolean JNICALL
    Java_com_test_debugging_DebuggerConnectedJNI(JNIenv * env, jobject obj) {
    if (gDvm.debuggerConnected || gDvm.debuggerActive)
        return JNI_TRUE;
    return JNI_FALSE;
}
```

Timer checks

Since debugging slows down the application's execution, debugging can be detected by measuring the difference in execution time of the process. The `Debug.threadCpuTimeNanos()` method indicates the amount of time that the current thread has been executing code and can be used to perform such a check.

```
static boolean detect_threadCpuTimeNanos(){
    long start = Debug.threadCpuTimeNanos();

    for(int i=0; i<10000000; ++i)
        continue;

    long stop = Debug.threadCpuTimeNanos();

    if(stop - start < 10000000) {
        return false;
    }
    else {
        return true;
    }
}
```

Tampering with JDWP-related data structures

In the Dalvik environment, the global virtual machine state is accessible via the `DvmGlobals` structure. This structure is itself accessible via the `gDvm` variable which holds a pointer to this structure. `DvmGlobals` contains multiple important variables and pointers needed for JDWP debugging which 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, to crash the JDWP thread one can set the `gDvm.methdDalvikDdmcServer_dispatch` function pointer to NULL:

```

JNIEXPORT jboolean JNICALL Java_poc_c_crashOnInit ( JNIEnv* env , jobject ) {
    gDvm.methdDalvikDdmcServer_dispatch = NULL;
}

```

Since Android 5.0 (Lollipop), ART replaces the Dalvik virtual machines and the `gDvm` variable is not accessible anymore but it is still possible to disable debugging through similar techniques. ART exports some of the vtables of JDWP-related classes as global symbols (in C++, vtables are tables that hold pointers to class methods). Amongst those exported vtables, the `JdwpSocketState` and `JdwpAdbState` handle JDWP connections via network sockets and adb, respectively. Overwriting the method pointers in the associated vtables allows to manipulate the behavior of the debugging runtime.

For example, overwriting the address of the function `jdwpAdbState::ProcessIncoming` with the address of `JdwpAdbState::Shutdown` will cause the debugger to disconnect immediately.

```

extern "C"

JNIEXPORT void JNICALL Java_sg_vantagepoint_jdwpctest_MainActivity_JDWPfun(
    JNIEnv *env,
    jobject /* this */) {

```

```

void* lib = dlopen("libart.so", RTLD_NOW);

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);

}
}

```

4.3.2 Traditional anti-debugging

Traditional debugging differs from JDWP debugging by the fact that it is based on the Linux `ptrace` system call. This system call is used to monitor and control the execution of the process being debugged and to change that process' memory and registers. `ptrace` is the main way of implementing system call tracing and breakpoint debugging in native code. Limiting the debugging checks to JDWP debugging is thus not enough as they won't catch classical debuggers based on `ptrace`.

Checking TracerPid

When an application is being debugged and a breakpoint is set on native code from Android Studio, the needed files will be copied to the target device and *lldb-server*, which uses `ptrace` under the hood, will be started and attached to the target process. At that point if we inspect the status file of the debugged process, located at either `/proc/<pid>/status` or `/proc/self/status`, we can see that the `TracerPid` field has a value different from 0, indicating that the process is being debugged.

```

$ adb shell ps -A | grep com.example.hellojni
u0_a271 11657 573 4302108 50600 ptrace_stop 0 t com.example.hellojni
$ adb shell cat /proc/11657/status | grep -e "^TracerPid:" | sed
"s/^TracerPid:\t//"
TracerPid: 11839
$ adb shell ps -A | grep 11839
u0_a271 11839 11837 14024 4548 poll_schedule_timeout 0 S lldb-server

```

Note that this check only works if the native part of the target app is being debugged; if it does not contain any native code or if the native code is not being actively debugged, the `TracerPid` field will be 0 and the check will not work.

Using Fork and ptrace

To prevent debugging of a process we can fork a child process and attach it to the parent with `ptrace`. If someone tries to debug the parent process afterwards, they will be notified that it is not possible because it is already being monitored. Below is an implementation example:

```
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);
        }
    }
}
```

A simple bypass of this mechanism would be to kill the child and thus freeing the parent from being monitored. To mitigate this trivial bypass, more elaborate schemes exist:

- forking multiple processes that trace each other.
- keeping track of running processes to make sure the monitoring children stay alive.
- monitoring the `/proc` values, such as `TracerPID` in `/proc/pid/status`.

It is possible to improve the first example with those ideas. After the `fork`, we launch in the parent an other thread that checks the child's status. The child's behavior will change depending on the type of build of the application; i.e. debug or release mode which is indicated by the `android:debuggable` flag in the manifest:

Release mode: The call to `ptrace` fails and the child crashes with a segmentation fault (exit code 11).

Debug mode: The `ptrace` call works and the child should run for ever. As a result, a call to `waitpid(child_pid)` should never return. If it does return, this indicates something abnormal and action should be taken.

The code of this protection is shown in appendix A.1.

4.4 Reverse engineering and function hooking tools detection

Reverse engineering or function hooking tools always leave some kind of trace in the file system. Some of those tools rely on the rooting of the device or the fact that the app is debuggable. It is thus possible to detect them through various methods and take action depending on the type of tool detected. In this section I will be focusing on detection of the Frida tool. As discussed in section 3.7.1, in its *injected* mode, Frida needs the `frida-server` binary to run on the target device. When it is attached to a target application, Frida injects a *frida-agent* into the memory of the app. This is something that can be detected by looking at `/proc/<pid>/maps`:

```
vbox86p:/ # cat /proc/18370/maps | grep -i frida
71b6bd6000-71b7d62000 r-xp /data/local/tmp/re.frida.server/frida-agent-64.so
71b7d7f000-71b7e06000 r--p /data/local/tmp/re.frida.server/frida-agent-64.so
71b7e06000-71b7e28000 rw-p /data/local/tmp/re.frida.server/frida-agent-64.so
```

Another common way of using Frida is through the embedding of the *frida-gadget* shared library into the apk and making the application loading it as one of its native libraries. Inspecting the app's memory maps after starting the application shows the embedded *frida-gadget* as *libfrida-gadget.so*

```
vbox86p:/ # cat /proc/18370/maps | grep -i frida

71b865a000-71b97f1000 r-xp
    /data/app/sg.vp.owasp_mobile.omtg_android-.../lib/arm64/libfrida-gadget.so
71b9802000-71b988a000 r--p
    /data/app/sg.vp.owasp_mobile.omtg_android-.../lib/arm64/libfrida-gadget.so
71b988a000-71b98ac000 rw-p
    /data/app/sg.vp.owasp_mobile.omtg_android-.../lib/arm64/libfrida-gadget.so
```

Those two techniques are quite simple to implement and thus equally simple to bypass. Detecting Frida effectively can get more complicated.

4.4.1 Checking the application's signature

Description In its *embedded* mode of operation, Frida can be used by injecting the library *frida-gadget* inside the target application. If the actual signature of the application differs from the one pinned in the APK, this could indicate that the library was injected.

Bypass technique A trivial way to bypass this would be to patch the APK to replace the pinned signature by the signature of the APK once it has been injected with *frida-gadget*.

4.4.2 Checking the environment for related artifacts

Description An artifact is an indicator that a reverse engineer tool was used on a device; it can be a package file, a binary, a library, a process or any temporary file. With Frida, a good example would be the `frida-server` binary running in the target device. It can be detected by inspecting the running services (e.g. with the Java method `getRunningServices`) and processes (e.g. with `ps`) and looking for one whose name is `frida-server`. Another solution would be to walk through the list of loaded libraries and search for the ones with "frida" in their names.

Bypass technique Android 7.0 removed the ability for an app developer to access running services or processes which were not started by the app itself; it thus prevents us to find daemons like `frida-server`. Even if we were able to access those daemons, bypassing this technique would be as easy as renaming the Frida artifact (i.e. `frida-server`, `frida-gadget` or `frida-agent`).

4.4.3 Checking for open TCP ports

Description By default, the `frida-server` process binds to TCP port 27042. The fact that it is open on a device might indicate that Frida is being used.

Bypass technique 27042 is the default TCP port but any other port can be used instead by specifying it in a command line argument.

4.4.4 Checking for ports repending to D-Bus authentication

Description As `frida-server` uses the D-Bus protocol to communicate with other processes, you can expect it to respond to the D-Bus AUTH message. We can thus detect Frida by sending a D-Bus AUTH message to every open ports and wait for a reponse from `frida-server`.

Bypass technique This technique is quite robust but does not work with other modes of operation that do not use the `frida-server` daemon.

4.4.5 Scanning process memory for known artifacts

Description Some strings can be found in all Frida libraries and be a good indicator of its presence on a device. For example, the string "LIBFRIDA" is present in all versions of `frida-gadget` and `frida-agent`. We can look for this string by iterating through the memory mappings listed in `/proc/self/maps` or `/proc/<pid>/maps` depending on the Android version.

Bypass techniques Again, this technique is robust but can still be bypassed by patching the Frida libraries and removing the "LIBFRIDA" strings.

4.5 Tampering detection

On top of signature validation which is trivial to bypass, two other types of checks can be used to verify the integrity of the application's files at runtime:

1. *Code integrity checks.* A CRC or hash can be computed on different types of files included in the application: `classes.dex`, native libraries or resource files. Those checks can implemented in both Java and native code. It is worth noting that this specific type of check only moves the problem elsewhere. Indeed, in addition to computing and replacing the app's signature, the attacker now has to compute and replace the CRC's or hashes of all modified files. This is still doable for a motivated attacker but adds an additional layer of security.

2. *File storage integrity checks.* This kind of check protects the integrity of files stored on an SD card or public storage and the integrity of key-value pairs that are stored in `SharedPreferences` by the application.

```
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;
    }
}
```

This example calculates a CRC over the `classes.dex` file and compares it to the expected value which is stored in the class `R` which contains strings used in the application.

4.6 Emulator detection

Running an application on an emulator gives the adversary a lot of capabilities and advantages compared to running it on a physical device. Often emulators are rooted by default and it is easier to make changes to the Android operating system as it removes the risk of bricking a real and costly device. If an application can detect it is being run on an emulator, it can then act accordingly by simply exiting or sending an alert to a remote server for example.

4.6.1 Build.prop inspection

The file `build.prop` is a system file that contains build properties and settings. Some of its contents are specific to the device or the device's manufacturer while others vary depending on the Android's version.

Some strings might indicate the device is being emulated:

<u>API Method</u>	<u>Value</u>	<u>Meaning</u>
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.USER	android-build	emulator

Looking for those strings in this file is not a reliable detection technique as this file can be edited if the device is rooted or if the Android system was compiled from the source.

4.6.2 TelephonyManager

Another detection method is to use Android's `TelephonyManager` API. Differences can be found between values returned by some methods of the API when running on an emulator and when running on a physical device:

<u>API</u>	<u>Value</u>	<u>Meaning</u>
<code>getDeviceId()</code>	0's	emulator
<code>getLine1Number()</code>	155552155	emulator
<code>getNetworkCountryIso()</code>	us	possibly emulator
<code>getNetworkType()</code>	3	possibly emulator
<code>getNetworkOperator().substring(0,3)</code>	310	possibly emulator
<code>getNetworkOperator().substring(3)</code>	260	possibly emulator
<code>getPhoneType()</code>	1	possibly emulator
<code>getSimCountryIso()</code>	us	possibly emulator
<code>getSimSerialNumber()</code>	89014103211118510720	emulator
<code>getSubscriberId()</code>	3102600000000000	emulator
<code>getVoiceMailNumber()</code>	15552175049	emulator

It is important to note that those values can still be tampered with by hooking frameworks such as Frida or Xposed.

4.7 Obfuscation

Wikipedia defines obfuscation as [...] *the deliberate act of creating source or machine code that is difficult for humans to understand*. Obfuscation is a complex, ever-evolving topic that is still ongoing research. I will therefore give only an overview of the different software obfuscation techniques following the taxonomy proposed in [40] by Hui X. et al.

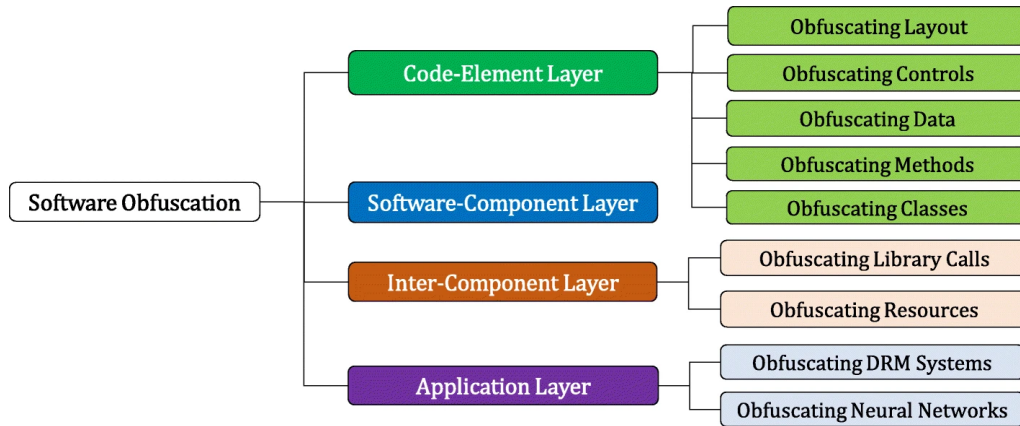


Figure 4.1 – Obfuscation taxonomy by Hui X. et al

4.7.1 Techniques taxonomy

Code element layer

This type of obfuscation modifies the instructions while keeping the original syntax intact. For example changing variable, classes and methods names for meaningless ones is an effective way to obfuscate source code. Another popular solution is the addition of junk code in the source. Junk code are instructions that are not functional and therefore have no real purpose in the source code to make adversarial program analysis harder.

Software component layer

As opposed to code element layer obfuscation, software element layer obfuscation does not focus on obfuscating code syntax or elements. For example *code virtualization* converts the original machine instructions into opcode for a specific virtual machine. To interpret this opcode at runtime, a lightweight VM is embedded in the program. DexGuard [27], a popular RASP solution by GuardSquare uses this technique to obfuscate Android applications.

Another example of this type of obfuscation is *decompilation prevention*. Reverse engineering non-scripting language such as C or C++ requires a disassembler. Disassembly of the binary can be prevented by introducing decompilation errors in the software.

Inter-component layer

As modern software engineering relies heavily on third party libraries and frameworks, released software often contains code that cannot be modified directly by developers. Such third party code can be reverse engineered by adversaries and leak important information about the piece of software being analyzed. Inter-component layer obfuscation is thus the process of obfuscating the released software as whole, including resources and libraries. For example, *resource encryption* encrypts every file that is not source code to hide its content to an attacker.

Application layer

Application layer obfuscation focuses on obfuscating specific types of software characterized by common features with ad-hoc mechanisms. For example, the structure of machine

learning models can leak information about them which is an issue for private ones. A solution is to train the target model all the way then reload the same well-trained model into a shallow network. This new network will be able to perform the same computations as the previous one but won't have the same learning capabilities. This defense mechanism is effective at preventing attackers from learning anything useful about the original network from the shallow one.

4.7.2 Tools

Proguard

Proguard [22] is an open-source project which shrinks, optimizes and obfuscates Android applications source code. It is shipped as part of the Android SDK and has its own Gradle module. Proguard operates at compile time on the Javabyte code level and outputs optimized Davlik bytecode. It offers three distinct type of features:

- It shrinks the code by detecting and removing unused classes, fields, methods and attributes.
- It optimizes the bytecode and removes unused instructions.
- It obfuscates the code by renaming the classes, fields and methods using short and meaningless names.

Configuration of the tool is done through `.proguard` files which contain Proguard rules. For example, a project might have some data classes that are serializable and renaming their fields might break the application at some point; to prevent Proguard from renaming that class, a developer can add the following rule to its proguard file:

```
-keep class com.ulb.ac.model.Example
```

To use it in an Android Studio project, the `build.gradle` file must be modified to include the following build instructions:

```
buildTypes {  
    release {  
        minifyEnabled true  
        proguardFiles getDefaultProguardFile('proguard-android.txt'),  
            'proguard-rules.pro'  
    }  
}
```

This tool works in a white-box fashion because a developer can't use it once the application is packaged and ready for deployment.

Obfuscapk

Obfuscapk [16] is an open-source CLI tool that works in a black box fashion on an APK file. Its focus is on obfuscation of the Davlik bytecode through a set of multiple different obfuscation techniques:

Debug removal. This technique removes debug information from the source code. Those debug information consist in the line numbers, types or method names for

example. Removing those meta-data effectively reduces the amount of useful information for the reverse engineering process.

Call indirection. This technique modifies the control flow graph of the application without modifying the code semantics. To achieve that, it wraps method calls in new methods; for example given method m_1 , it creates method m_2 which invokes m_1 and replaces call to m_1 by a call to m_2 .

Goto. Given a method, a `goto` instruction is inserted in the code which points to the end of the method and another `goto` pointing to the instruction is inserted right after the first `goto`.

Reorder. This technique changes the order of basic code blocks. When an if statement is found in the code, its condition is inverted e.g. `if x < 10` becomes `if x >= 10` and the code is reorganised to accommodate this change while keeping the initial logic.

Arithmetic branch. The goal of this technique is to insert useless code to make static analysis harder without introducing performance overhead. In practice, the inserted code is composed of arithmetic operations and of branch instructions which depend on the result of those operations but are crafted in such a way that those branches are never actually taken.

Nop. *No-operation* instructions are instructions that do nothing. Inserting such instructions makes static analysis harder.

Method overload. This technique takes advantage of the method overloading mechanism in Java. Given a method, it will create a new void method with the same name and arguments, but it will also add new arguments and fill the body of that method with random arithmetic computations.

Reflection. This technique complexifies the method call process by using the reflection feature of Java. It analyzes the code looking for candidate method calls (i.e. not a constructor, public visibility, ...) and replaces it with a call to a custom method that will invoke the original one through Java's reflection APIs. This technique avoids all calls to the Android framework.

Advanced reflection. This technique follows the same logic as the previous one but it specifically targets calls to dangerous Android APIs.

When Obfuscapk starts, it automatically generates a random secret key of 32 ASCII characters that is used to encrypt the following resources:

- Native libraries.
- Asset files (videos, photos, text files, ...).
- Strings contained in the `strings.xml` resource file.
- Constant strings in the source code.

Encrypting those resources can effectively complexify static analysis but has the disadvantage of adding significant performance issues because the encrypted resources must be decrypted to be used.

4.8 Anti-repackaging

4.8.1 Common techniques

The goal of an anti-repackaging protection is to prevent an application from being modified, repackaged and redistributed by protecting the application's code from being tampered with. The application is protected by a set of *detection nodes* injected in some particular locations in the code. Those detection nodes perform integrity checks (aka anti-tampering controls) when executed at runtime. More specifically, those checks compare the signature of a specific part of the APK (i.e. either its code base or its resources) with a value pre-computed during the building of the application. If the check fails, a repackaging is detected and the detection node usually exits the application, preventing it from being effectively run if modified.

The main task of an attacker when trying to repack a protected application is to identify and remove detection nodes. To protect them, most repackaging solutions hide them in *logic bombs*. A logic bomb protects a detection node by encrypting it using state-of-the-art encryption schemes. The key used to encrypt and decrypt those detection nodes is derived from a constant value used in the program. In order to get a fully working APK with no protections, the attacker must find and remove all logic bombs using a combination of static and dynamic analysis.

Logic bomb implementation

Logic bombs are placed in the code at specific locations called *qualified conditions*. Those conditions are if statements of the following form:

```
if (v == C) {  
  
}
```

where *v* is a variable and *C* is a constant. The original condition is modified to compare the pre-computed hash value of *C* with the result of the hash function applied to *v* plus some salt. The detection nodes are embedded in the body of that condition and encrypted using the original value of *C* as the encryption key. The fact that cryptographic hashing functions are one way by nature protects the logic bombs; in order to find the key needed to decrypt them, the attacker must first recover the original value of *C* which takes a long time when brute forcing. Inside those logic bombs different checks are implemented either in Java or native code to improve reliability.

It is important to note that following this scheme, an attacker could easily remove encrypted logic bombs by patching the corresponding smali file as it would not influence the normal behavior of the application; a solution to this issue will be discussed in the next subsection. Also, the decryption at runtime introduces a performance overhead to the protected application.

4.8.2 ARMAND

ARMAND, for Anti-Repackaging through Multi-pattern Anti-tampering based on Native Detection, is a novel anti-tampering approach based on the embedding of logic bombs and anti-tampering detection nodes directly in the APK file. Its aim is to provide solutions to current limitations in the state-of-the-art techniques by offering the following features:

1. **Use of multiple patterns** to place and hide logic bombs at multiple locations in the application. Current techniques use a single common pattern making detection and deactivation of logic bombs easier. To solve this issue, ARMAND uses multiple distinct patterns for logic bombs and anti-tampering checks to harden the detection and removal phase.
2. **Rely on native code** to hide anti-tampering protections. Native code being harder to reverse engineer than Java code, it makes the detection and removal of security controls difficult. It is thus a suitable location to include logic bombs and anti-tampering checks.
3. **Optimize the effectiveness of the detection nodes.** Current approaches struggle to find the right balance between inserting too many logic bombs and checks which will not be triggered at runtime and inserting too few of them to avoid performance decrease. ARMAND solves this issue by adopting an efficient deployment technique of logic bombs by making sure that they won't be included in unused code.
4. **Use honeypot bombs.** Honeypot bombs are fake logic bombs designed to increase the complexity of the detection and removal phases.
5. **Hide the anti-repackaging protection.** Stealth was often overlooked in the design of anti-repackaging mechanisms leading to obvious indicators that the end application was protected. Increased stealthiness of such mechanisms makes it hard for an attacker to understand that the target application is protected.

Multi-patterning logic bombs

As discussed in the *logic bombs implementation* subsection, the currently described logic bomb implementation pattern allows easy detection and removal by an attacker. To solve this issue, ARMAND adopts six different types of logic bombs:

1. **Java bomb.** In this type of logic bomb shown in figure 4.2, one or multiple anti-tampering checks are added in Java to the body of the if statement and encrypted using the value of C as the key.

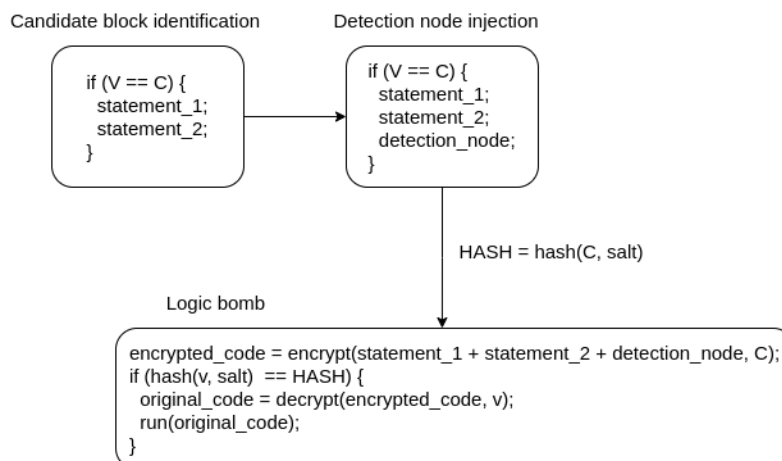


Figure 4.2 – Java bomb implementation

2. **Native key bomb.** The body of the if statement of the qualified condition is encrypted using the return value of an anti-tampering check implemented in native code as the key as shown in figure 4.3.

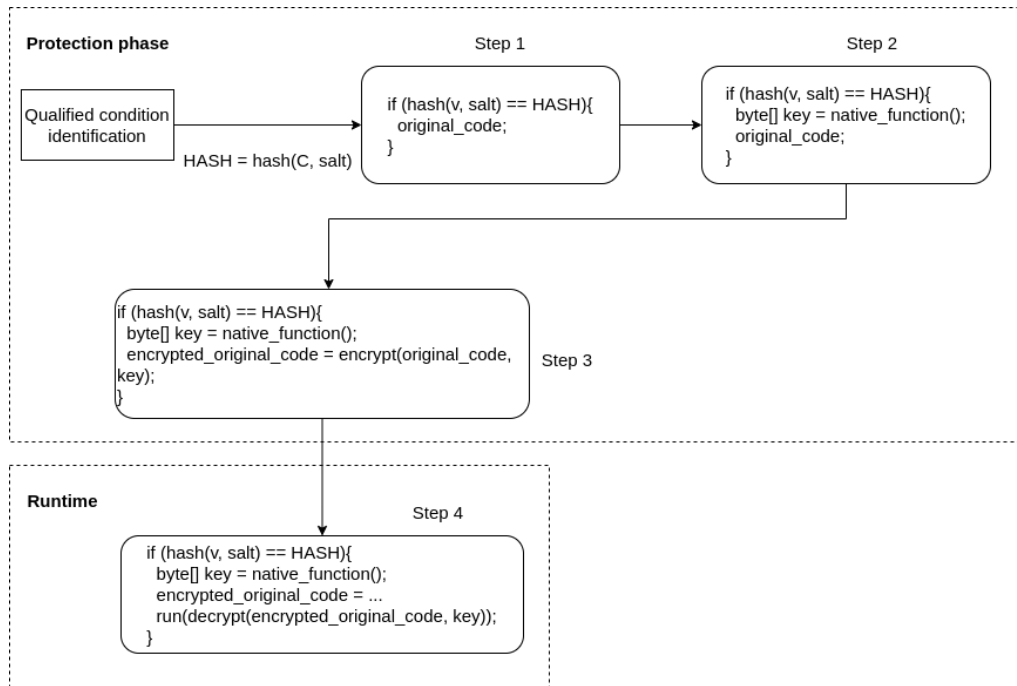


Figure 4.3 – Native-key bomb implementation

Once a qualified condition is identified, its if statement is transformed into a hash comparison of $\text{hash}(V, \text{salt})$ and $\text{hash}(C, \text{salt})$ as described in Step 1. In the body of the qualified condition, a call to a native function performing an anti-tampering check is made (Step 2), which returns the key to encrypt the original code (Step 3) if no tampering is detected. Step 4 shows the behavior of the bomb at runtime; if no tampering is detected, the encrypted original code is decrypted using the key returned by the native function call and the decrypted original code is then executed normally.

3. **Native anti-tampering bomb** A call to a native function performing anti-tampering checks is injected to the body of the qualified condition which is then encrypted. In contrary to the native key bomb, the original code is not encrypted with the return value of the native function but with the value of V instead.
4. **Java anti-tampering & native key bomb** This bomb uses the same scheme as the native key bomb but adds anti-tampering checks in Java to the body of the qualified condition. The body is then encrypted following the same mechanism as the native key bomb.
5. **Java & native anti-tampering bomb** This logic bomb is a combination of the Java anti-tampering and native anti-tampering logic bombs. The anti-tampering checks are placed both in Java in the body of the qualified condition and in the native function. The entire body is then encrypted with the value of V as the key.
6. **Honeypot bomb** There is no anti-tampering checks included in the encrypted original code; this bomb is only here to delay the attacker in their reverse engineering

efforts.

On top of all those logic bombs, some anti-tampering checks are distributed in random locations in the code of the application, meaning that they are not protected by a logic bomb. To detect and remove them, the attacker will have to manually inspect the entire code base.

Bypass mitigations and hiding strategies

In state-of-the-art anti-tampering schemes, all anti-tampering checks of the same nature share a single implementation written either in Java or in native code. This creates a single point of failure that an attacker can exploit by rewriting the check's implementation by leveraging static analysis and performing Smali code patching for example. By doing so, the attacker would successfully bypass all anti-repackaging checks using the rewritten implementation. ARMAND mitigates this issue by replicating the code of the anti-tampering check in each detection node to make sure that it can't be easily bypassed.

On top of that, ARMAND introduces the concept of nested logic bombs, i.e. one logic bomb can contain one or more logic bombs. This mechanism forces the attacker to decrypt each and every nested logic bomb in order to remove the included anti-tampering checks. ARMAND also randomizes what sort of logic bomb will be included in qualified condition and which sort of anti-tampering check will be included inside them to make their identification harder for the attacker.

4.9 Summary

The wide range of security protections described in this chapter represent the state-of-the-art in defenses against static and dynamic analysis and application tampering. The reader should now have a good understanding of those threats and what can be done to mitigate them. To increase the security of an Android application, a set of defenses should be chosen and implemented in the target application. To ensure maximal security, the *defense in depth* principle should be applied, each security measure acting as an additional security layer. Those defenses should therefore be as varied as possible and should be implemented in such a way that they all work together flawlessly.

In next chapter, I will showcase how I used some defense mechanisms described previously to increase Android applications security. The chapter will also describe the technical details of the ARMANDroid project, as doing so is important to understand how such a software can be augmented with additional protections and how their implementation is done in practice.

Chapter 5

Implementation

5.1 Introduction

This chapter will provide an overall description of the implementation of ARMANDroid, the implementation of the ARMAND scheme for Android. After that global overview, I will explain more in details the Java and native code injection process. Lastly, I will showcase the different security measures I implemented and how I integrated them with ARMANDroid so that they are effectively injected in the target application to protect it.

5.2 Current state of ARMANDroid

5.2.1 Overview

ARMANDroid is shipped in different forms: a docker image and a jar. The tool works directly on a packaged application, i.e. an application in APK format. It takes multiple inputs as arguments : P_{Java_AT} , P_{native_key} and P_{native_AT} and a package name (PN). The three first arguments specify the probability to include Java security checks and one of the six types of logic bombs as described in section 4.8.2. The last parameter PN specifies the package name in which to include those logic bombs / security checks. If PN is not specified, the tool defaults to the entire application.

The tool uses the soot [13] library extensively to make its changes to the target manipulation. Its powerful API is used throughout the project to perform the bytecode manipulation technique described in section 3.6.3 that allows to modify the target application's behavior as we want by modifying classes or methods bodies or inject new classes for example.

ARMANDroid is composed of the following packages:

bin Contains the `native-build.sh` script which compiles all the `c++` files containing native checks into a single shared library.

embedded Contains the different Java classes that will be injected as is in the target APK.

Mafefiles Contains the file `CMakeLists.txt` that is needed by the utility `cmake` to compile the `c++` files.

models Contains all the classes representing an object needed by the program (e.g. Method or StatisticContainer). This package also contains all the Java checks that will be included in logic bombs during the protection phase.

NativeFunctions Contains all the native checks separated in several `c++` files.

sootTransformer Contains all the classes that will transform the target application in one way or another.

util Contains miscellaneous classes needed for processing the target APK.

The protection process can be divided in three main steps:

1. Application pre-processing. During this phase, ARMANDroid unpacks the APK to extract its `classes.dex` file(s). Soot then converts it to Jimple for easier manipulation of the code. Finally, ARMANDroid scans the code looking for qualified conditions for later use and injects a set of Java anti-tampering checks.
2. Bomb injection. For each qualified condition found during the pre-processing phase, the tool includes one of the six types of logic bombs described in section 4.8.2.
3. Application packaging. During this last phase, the tool converts the Jimple code back to Dalvik bytecode, adds the native libraries, updates the `AndroidManifest.xml` file and outputs the protected APK file ready to be signed and deployed.

Application pre-processing

The role of this step is to :

1. Unpack the application and prepare it for processing.
2. Process the application's codebase to look for qualified conditions.
3. Inject Java security controls.

Application unpacking and processing preparation The tool unpacks the apk and extracts its `classes.dex` file. The bytecode of the app is then converted to Jimple with Soot.

Qualified conditions identification The Jimple code is scanned for instructions that would qualify to integrate logic bombs. The structure of such qualified conditions is described in section 4.8.1. Two different types of qualified conditions are currently searched for in the code: if and switch statements. In the case of a switch statement, each of its cases is considered as a separated qualified condition.

An AST (Abstract Syntax Tree), called *transformation tree* in the code, is built for each qualified condition discovered with the qualified condition at the top of the tree, used as the root node. If nested qualified conditions are discovered, those are added to the AST as leaves to indicate a hierarchy.

During this phase, the tool also stores references to the first and the last instruction of the block being encrypted and put inside a logic bomb.

Java security controls injection The security controls injected in Java during this phase are not included in a logic bomb but are injected at random places in the code. For each parsed instruction, the tool has a probability of P_{Java_AT} to include a Java security control before it. The resulting probability to add at least one Java security control in a set of n instructions is $P_{Java_AT_embedded} = 1 - (1 - P_{Java_AT})^n$

Bomb injection

Each transformation tree created during the *pre-processing* phase is iterated through to process all their qualified conditions. During this phase, every qualified condition is going to be transformed into a logic bomb as shown in figure 5.1.

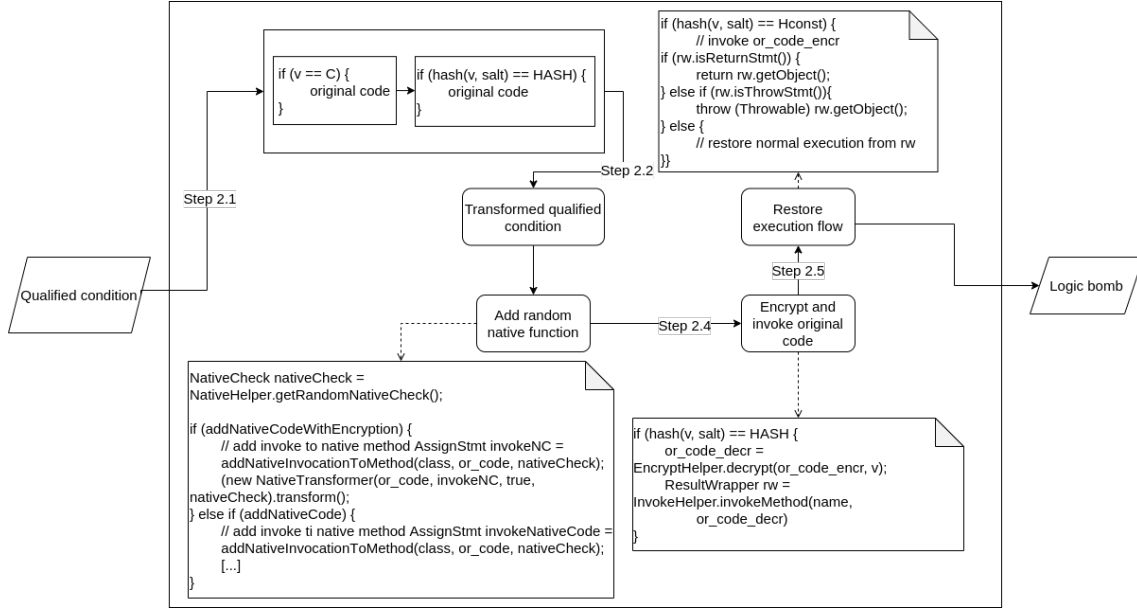


Figure 5.1 – Logic bomb injection

First (step 2.1), the condition is transformed into `hash(V, salt) == HASH` where `V` is the variable to be hashed, `HASH` is the result of the hashing algorithm applied on the constant and `salt` is a pseudo-random value. Then (step 2.2), the tool processes the original code contained within the qualified condition to make sure it will be correctly executed after the transformation. The important thing to understand is the execution flow once the original code has been executed. Once the original code has been executed, four distinct scenarios are possible:

1. The original code may jump to the next set of instructions
2. The original code may throw an exception
3. The original code may jump to a different block
4. The original code may terminate the execution by returning an object.

During step 2.3, the tool injects a native call implementing either a Native-key or a Native AT function, depending on the content of the original code. The type of logic bomb to be implemented depends on the content of the original code but also on the input parameters P_{native_key} and P_{native_AT} . To be more specific, if the original code contains at least one or more Java AT (with probability $P_{Java_AT_embedded}$), the logic bomb will be a Java AT & Native-key bomb with a probability of P_{native_key} , while the probability of having a Java & Native AT bomb is of $P_{native_AT|\neg native_key}$. In the case where no native functions are injected, the qualified condition will contain a Java bomb with a probability of $P_{Java_bomb}=1-P_{native}$, where P_{native} equals the join probabilities of having a logic bomb with a native or a native AT control: $P_{native} = P_{native_key} \cup P_{native_AT|\neg native_key}$.

In step 2.4 the modified body of the qualified condition is encrypted with AES-128 using the value of the constant to derive the key and compressed using base64 encoding. ARMANDroid then adds the instructions needed to decrypt and invoke the body of the condition. To do so, it embeds two Java classes in the target APK; `EncryptHelper` and `InvokeHelper`. `EncryptHelper` is used to decrypt the encrypted body of the condition

```

293     public boolean onOptionsItemSelected(MenuItem item) {
294         int $i0 = item.getItemId();
295         Integer $r4 = Integer.valueOf($i0);
296         if (HashHelper.match($r4, -9223370409524953198L, new byte[]{-4, -
297             ResultWrapper $r10 = InvokeHelper.invokeMethod("com.android.i
298             if ($r10.isReturnStmt()) {
299                 return ((Boolean) $r10.getReturnValue()).booleanValue();
300             }
301             if ($r10.isThrowStmt()) {
302                 throw ((Throwable) $r10.getReturnValue());
303             }
304             DoLogin doLogin = (DoLogin) $r10.getRestoredVariables()[0];
305             return true;
306         } else if ($i0 != R.id.action_exit) {
307             return super.onOptionsItemSelected(item);
308         } else {
309             Context $r3 = getBaseContext();
310             Log.d("RASP_DEBUG", "DETECT DEBUGGER");
311             if (Debug.isDebuggerConnected()) {
312                 Log.d("RASP_DEBUG", "Debugger is connected");
313             }
314             Intent r2 = new Intent($r3, LoginActivity.class);
315             r2.addFlags(67108864);
316             startActivity(r2);
317             return true;
318         }
319     }

```

Figure 5.2 – Example of a logic bomb injected in a target application

using the decryption key. Once the decryption is done, `InvokeHelper` loads the byte array resulting from the decryption and invokes the `execute` method. This class uses `InMemoryDexClassLoader` to load and execute the code in the form of a byte array, those bytes being bytes from the DEX file. Finally during step 2.5, ARMANDroid invokes the `ResultWrapper` object to recover the original execution flow.

Figure 5.2 shows a logic bomb injected in a method called `onOptionsItemSelected` at line 296. Notice how the comparison between the constant and the variable is done with the class `HashHelper`. It also shows a security check injected in plain Java that checks for the presence of a Debugger at line 310. This screenshot was taken from the JadxGui decompiler on a modified application.

5.2.2 Java code injection

The classes containing the security controls are placed in the package `javaChecks` which is a subpackage of the `model` package. Each class of `javaChecks` is loaded and converted to Jimple by the `initJavaAntiTamperingMap` method:

```

SootClass checkClass = new SootClass("models.javaChecks.JavaChecks");
File inputFile = new File("/tmp/" + UUID.randomUUID().toString());
InputStream inputStream =
    ApplicationHelper.class.getClassLoader().getResourceAsStream("models/
javaChecks/JavaChecks.class");
try {
    FileUtils.copyInputStreamToFile(inputStream, inputFile);
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
new CoffiClassSource("JavaChecks", new
    FoundFile(inputFile)).resolve(checkClass);

```

The method starts by loading the class `className` as a `SootClass`. Then

the content of the class file corresponding to `className` is dumped in a temporary file. Finally, the line: `new CoffiClassSource(className, new FoundFile(inputFile)).resolve(checkClass)` converts the class to Jimple to facilitate further manipulation and analysis.

Once the class containing methods to be injected is transformed to Jimple, it is possible to extract it with the wanted methods:

```
SootMethod originMethod = checkClass.getMethod("void
    checkAppSignature(android.content.Context,byte[])");
Body body = originMethod.getSource().getBody(originMethod, "jtp");
AT_METHODS.put("checkAppSignature", new
    JavaCheckWrapper(originMethod.getName(), body,
    ApplicationHelper.getKey(), true, true));
```

This block of code takes the signature of the method to be injected as a parameter and uses it to load it from the `checkClass` SootClass. Then the method's body is extracted and a `JavaCheckWrapper` object is added to the `AT_METHODS` HashMap containing all the methods to be injected in the target application. The `JavaCheckWrapper` object represents a Java security control and takes as parameters the name of the method, its body, the expected value to be returned if the security control is an anti-tampering check and two Booleans that specify if the method takes a `Context` object as a parameter and if it throws an exception. This block of code is repeated for each method to be injected.

Next, the body of the injected method is merged with the body of the original method which contains the qualified condition. To keep this explanation concise I won't be explaining the injection process further.

5.2.3 Native code injection

The connection between Java code and the JNI is done as follows (as explained in the documentation [33]):

1. **Load the library from Java.** Let's assume the native library we want to use in our project is called `native-lib`. Then we have to use the following line of code to load it: `System.loadLibrary("native-lib")`. Usually this is done in a `static` block to ensure it is loaded before doing anything else.
2. **Declare the native function in the calling class.** This is done by declaring a native method without body like so: `public native void nativeFunction()`.
3. **Declare the function in a native file properly.** The signature of the native function must follow a strict convention to be callable from Java: `JNIEXPORT void JNICALL Java_com_example_ulb_be_nativeFunction()`.
4. **Create a CMakeList.txt file for compilation using cmake.** I won't detail every compilation step here but the `CMakeList.txt` is responsible for the dynamic linking of the different libraries used.

In the project, each native file contains only one protection. If another method is needed by a protection, it has to be placed in the source file `utils.cpp` and declared in the header file `utils.h`. Since every method's signature must follow the convention described above, their names are dynamically generated at compile time. Every native protections

is then placed in the same file, `vendor.cpp` which is then compiled to a library that is injected and called from the injected logic bombs. As for Java code injection, the actual process of injective calls to the native library with Soot is quite lengthy, which is why I won't explain this process further.

5.2.4 Java security checks

In its current state, ARMANDroid already provides a set of Java anti-tampering checks and some detection methods against debuggers and emulators in an early attempt at mitigating such threats:

- `checkAppSignature(Context context, byte[] expectedValue)`. This method checks if the application's signature is equal to the `expectedValue` passed in parameter.
- `checkEmulator()`. This method uses the technique described in section 4.6.1 to try to detect if the applications is running on an emulated device.
- `checkDebuggable(Context context)`. This method uses two different techniques to check if the `debuggable` flag is set in the manifest.

5.2.5 Native security checks

The project also contains a set of native anti-tampering controls:

- `SignatureCheck.cpp`: This check computes the signature of the application and returns it.
- `PackageName.cpp`: This method will convert the package name of the application to bytes and returns it.
- `FileChecksum.cpp`: This method will compute a checksum over the files given in parameters, e.g. resource files and return it.

The return values from all those anti-tampering methods are used in the logic bomb mechanism as encryption / decryption keys.

5.3 Contribution

In this section I will describe each of the implemented security measures and explain how they work. Most of them were already covered in chapter 4, so in this section I will explain the differences with their original implementation if any. The security measures that were implemented were chosen because of their efficiency and their relative simplicity to integrate with the ARMANDroid project. The reader should keep in mind that because of the wide variety of topic addressed in this thesis and the time constraint associated with such a work, I couldn't dive deep into each of the subjects.

There are four distinct types of checks that were added to ARMANDroid:

1. Root detection: checking if the device rooted.
2. Debug detection & prevention: checking if the application is being debugged and preventing it from being debugged if possible.

3. Emulator detection: checking if the device is in fact an emulator.
4. Frida detection: checking for the presence of the Frida tool on the device.

Those security measures are distributed in both the Java layer and the native layer to make reverse engineering harder. To prevent all the following protections from being removed thanks to Smali code patching, they are all added to logic bombs in order to fully benefit of the tampering protection offered by the ARMAND scheme.

5.3.1 Java protections

To integrate Java protections with ARMANDroid, I created one Java class per type of check, namely: `RootDetection`, `EmulatorDetection`, `DebugDetection` and `REToolsDEtection`. Each of those files are placed in the package `models.javaChecks`. In those classes, each method represents a check. Each method must be declared as static to be recognized by soot.

The `initJavaAntiTamperingMap` in its current state did not allow to add new checks easily. Therefore I decided to refactor it to break it down in two smaller methods: `getCheckClass` and `addMethod`.

Shutting down the application

For each security measure implemented in Java, the application fails silently if a threat is detected. To do so, each protection method throws a `RuntimeException` when the check fails. The exception is then caught and the `ActivityManager` API is then used in the catch block to shut down all activities leading to shutting down the app entirely. In order to not leak any technical information to an attacker, no error messages are displayed in the logs, so that the application fails silently:

```
catch (RuntimeException e) {
    List<ActivityManager.AppTask> appTasks =
        ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE))
        .getAppTasks();

    for (ActivityManager.AppTask task : appTasks) {
        task.finishAndRemoveTask();
    }
}
```

Root detection - checking for suspicious files presence

This check iterates over an array containing every suspicious file name and checks if they exist on the file system. The complete code for this check is show in appendix A.2

Root detection - checking for suspicious package names

In a similar fashion to the precedent check, this one iterates over an array containing every suspicious package names and uses the `PackageManager` API to check if they are installed on the device. The complete code of this check is shown in appendix A.3.

Root detection - checking if su is on the PATH

To check if the su binary is on the PATH environment variable, the PATH is retrieved with the method `System.getenv("PATH")`. Then the directories on the PATH are iterated and checked for the presence of su.

```
for (String pathDir : System.getenv("PATH").split(":")){
    if (new File(pathDir, "su").exists()) {
        Log.d("ROOT_DETECTION", "su is on the PATH");
        throw new RuntimeException("ROOT_DETECTION: su is on the PATH");
    }
}
```

The complete code of this check is shown in appendix A.4.

Root detection - checking for test key

To check if the ROM was built by a certified Android developer or not, the TAGS entry is retrieved from the `build.prop` file and looks if it contains the string "test-keys":

```
String tags = Build.TAGS;

if (tags != null && tags.contains("test-keys")) {
    throw new RuntimeException("ROOT_DETECTION check test key failed");
}
```

The complete code of this check is shown in appendix A.5.

Emulator detection - checking suspicious package names

While working with the Genymotion emulators, I noticed some packages installed on the device indicating it is in fact a Genymotion device. Indeed, multiple packages contain the word "genymotion" or the shorter "geny" in their names; this indicator can then be used to detect a Genymotion emulator. This check uses the `PackageManager` API from Android to get the list of installed packages and check if any of them contains the string "geny" in their names.

```
PackageManager packageManager = context.getPackageManager();
List<ApplicationInfo> installedPackages =
    packageManager.getInstalledApplications(PackageManager.GET_META_DATA);

for (int i = 0; i < installedPackages.size(); i++) {
    ApplicationInfo packageInfo = installedPackages.get(i);
    if (packageInfo.packageName.contains("geny")) {
        throw new RuntimeException("EMULATOR_DETECTION: Package detected " +
            packageInfo.packageName);
    }
}
```

The complete code of this check is shown in appendix A.6.

Emulator detection - checking for artifacts in build.prop

The file `build.prop` was already checked for artifacts by the ARMANDroid team in a first emulator detection attempt but more indicators can be checked as described in section 4.6.1. I thus added the ones that weren't already used for a more complete and effective emulator detection. Additional properties used for this check include looking at the type of board used or the radio version among others. The check works by getting a reference to the Build API and looking for suspicious values:

```
boolean ABIS_artifacts = false;

for (String supported_abi : Build.SUPPORTED_ABIS) {
    if (supported_abi.contains("unknown") || supported_abi.contains("armeabi"))
        ABIS_artifacts = true;
}

boolean isEmulator = ABIS_artifacts
    || Build.BOARD.equals("unknown")
    || Build.HOST.equals("android-test")
    || Build.ID.equals("FRF91")
    || Build.MANUFACTURER.equals("unknown")
    || Build.MODEL.equals("sdk")
    || Build.getRadioVersion().equals("unknown")
    || Build.getSerial() == null
    || Build.USER.equals("android-build");
```

The complete code of this check is shown in appendix A.7.

Emulator detection - checking the TelephonyManager API

In order to use the TelephonyManager API, the `READ_PHONE_STATE` permission must be set in the manifest of the application like so:

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

If the permission is set in the manifest and the user of the application already accepted it, the check will work normally. On the contrary, if the user refused the app's request to access the phone state or if the permission is not set in the manifest, an error will be thrown then caught and the check won't work.

```
TelephonyManager telephonyManager = (TelephonyManager)
    context.getSystemService(Context.TELEPHONY_SERVICE);
@SuppressWarnings("MissingPermission") boolean isEmulator =
    telephonyManager.getDeviceId().equals("0's")
    || telephonyManager.getLine1Number().equals("155552155")
    || telephonyManager.getNetworkCountryIso().equals("us")
    || telephonyManager.getNetworkType() == 3
    || telephonyManager.getNetworkOperator().substring(0, 3).equals("310")
    || telephonyManager.getNetworkOperator().substring(3).equals("260")
    || telephonyManager.getPhoneType() == 1
    || telephonyManager.getSimCountryIso().equals("us")
    || telephonyManager.getSimSerialNumber().equals("8901410321118510720")
    || telephonyManager.getSubscriberId().equals("310260000000000")
```

```
|| telephonyManager.getVoiceMailNumber().equals("15552175049");
```

For this check I get a reference to a `TelephonyManager` object then check all suspicious values of the API.

The complete code of this check is shown in appendix A.8.

Debug detection - checking if a JDWP debugger is attached

It is possible to easily check if a JDWP debugger is connected to the application by calling the method `Debug.isDebugEnabledConnected()`.

The complete code for this check is shown in appendix A.9.

5.3.2 Native protections

In its current implementation, ARMANDroid only uses native checks that return an expected value that is used as an encryption and decryption key in the logic bomb process. To be able to add native security protections that do not return an expected value (i.e. debugging prevention), I needed to modify the `Native AT` logic bomb implementation. I applied the following modifications to the project to achieve this result:

1. **Added the `hasExpectedValue` flag to the `NativeWrapper` class.** This is used to make a distinction between native functions which return an expected value and the others.
2. **Created a new function `addNativeInvocationOnlyToMethod` to the `TransformHelper` class.** This new function is heavily inspired by the `addNativeInvocationToMethod` function of the same class. This function is responsible for adding a native invocation to a specific method and store its return value in a variable. Since the new native checks do not return anything, `addNativeInvocationOnlyToMethod` will only invoke the given native function without storing its return value.
3. **Modified the return value of the native functions declaration in the calling classes to `void`.** To be able to call a native function from Java, we need to declare it in its calling class like so: `private native void String native_function()`. Formerly, all native functions had the `byte[]` return type and it needed to change to `void` to match the return type of the new native checks.

Shutting down the application

For protections implemented on the native layer, a call to `_exit()` is made. A call to `_exit()` doesn't shut down the entire app but closes the current activity. The `ActivityManager` then restarts the last activity that was open before the one that closed. If the activity that closed was the first activity, then the application is turned off entirely.

Frida detection - checking if default port is open

The utility `frida-server` listens by default on port 27042. Even though this port can be changed easily, checking if it is open could indicate Frida's presence on a device. This check opens a connection to that port on the native layer thanks to a socket. If the connection succeeds, the port is open and Frida might be installed on the device.

The complete code of this check is shown in appendix A.12.

Frida detection - checking for loaded libraries

When Frida is used to trace an application, it injects it with libraries that can be detected. The file `/proc/self/maps` contains a list of all the libraries loaded in the application. By reading this file it is therefore possible to look for the presence of a library containing the keyword "frida" to check if the application has been injected with a Frida library. Berdhard's Mueller [31] implementation of this check was reused here.

The complete code of this check is shown in appendix A.13.

Debug detection - checking tracer Pid

For the implementation of this check I reused code of the gperftools project [5] which implements an `isDebuggerAttached` method. As explained in section 4.3.2, when `gdbserver` is attached to a process, the status file of the debugged process is updated to show a value different than 0 in the field `TracerPid`.

This checks opens up the status file of the application's process, then reads it. If the value of the `TracerPid` differs from 0, a log message is shown to indicate that the debugger was detected.

The complete code of this check is shown in appendix A.10.

Debug prevention

The initial code used to prevent debugging described in section 4.3.2 needed to be modified to be integrated with the project. This protection is divided in two methods: the main one, `void antiDebug()` and `void* monitorPid(void* arguments)`. The later is placed in `utils.cpp` while `antiDebug` is placed in its own file, `DebugPrevention.cpp`.

The complete code of this check is shown in appendix A.11.

5.4 Summary

This chapter showed how different types of defense mechanisms could be integrated with the ARMANDroid project to provide increased security to Android applications. The reader should now have a clear understanding of the implementation details of each of those checks and how they were added to ARMANDroid. An anti-tampering solution such as ARMAND was of prime importance to make it as hard as possible for an attacker to remove every security check. Without it, it would have been trivial or at least easier to disable or simply remove every injected security measure. The fact that the protection mechanisms are spread between the Java and native layer is another important aspect of this work. Indeed, it increases the effort needed by an adversary to effectively bypass or remove every type of logic bombs and defense mechanisms.

In the next chapter I will showcase the methodology I used to check that every single protection showcased in this chapter are fully functional as well as potential bypass techniques that attackers could use to attack the proposed scheme.

Chapter 6

Effectiveness Assessment

6.1 Introduction

This chapter will present the experimental results I got from using the different protections showcased in previous chapter. To get those results I followed a methodology that will also be discussed in this chapter. To verify that every implemented security protection works properly I needed to trigger them one by one and make sure that they had the expected effect on the application. This verification process allowed me to remove some security checks that didn't work or didn't have the expected behavior. For each tested defense mechanism I will also briefly introduce theoretical bypass techniques that could make those security checks ineffective. This chapter will serve as a proof that the protections added to an unprotected application can effectively detect and prevent the desired threats.

6.2 Testing environment and methodology

Since I have less control on security protections once injected in an application and the ARMANDroid's protection process takes a long time I tested the different security protections following a two step process:

1. Each protection is implemented directly in a test project using Android Studio. That way I was able to debug and test them more easily. If the implementation of a protection works fine during this step I move on to the step 2.
2. The protection is integrated within ARMANDroid and injected in the target application. I then use various techniques to trigger the protection and see if it still works as the injection process can sometimes have unpredicted side effects if not done properly.

For step 1 of my testing process, I used the insecure bank [23] and the damn insecure and vulnerable app [29] open-source projects as target applications, which are both Android applications riddled with common vulnerabilities developed for testing and education purposes. Since they are both open-source I can import them in Android Studio and modify them as I wish. To run the protected application, I used a Google Nexus 6 emulator running on a x86 architecture that I downloaded with Genymotion.

After using ARMANDroid to protect the target application, I used the Jadx-Gui de-compiler [36] to explore the modified APK and check if the different pieces of code and files to be injected were indeed present in the protected application.

Since I used the ARMANDroid tool in its jar form, I ran the following command to modify a target APK:

```
$ java -jar anti-repackaging-framework.jar -k  
    "PKCS12:rasp.keystore:<password>:rasp" -a ~ /Android/Sdk/platforms/ -i  
    InsecureBankv2.apk
```

I then signed it with apksigner as seen in section 3.6.6:

```
$ apksigner sign --ks ulb.keystore --ks-key-alias signkey aligned_target.apk
```

and installed it on the emulator with `adb install`.

Once the modified application is correctly installed on the emulator, I used the `logcat` utility provided with the Android SDK to check the logs of the application looking for the log traces I put in the different methods I implemented. Those logs are present in each method and indicate that a security measure has been effectively triggered by the application. `Logcat` is a powerful tool that allows us to specify the `pid` (process id) of the application we want to monitor to filter out unnecessary logs. To get that `pid`, I use `adb` to run the `ps` command inside a shell initiated inside my virtual device:

```
$ adb shell "ps -e | grep insecure*"

```

The output looks like this:

```
u0_a67 3905 655 1443212 136320 ep_poll f00ccbb9 S com.android.insecurebankv2
```

3905 being the `pid` of the application `com.android.insecurebankv2`. I can now attach `logcat` to that particular process like so:

```
$ adb logcat --pid=3905
```

6.3 Root detection

6.3.1 Checking for test key

The `TAGS` entry of the `build.prop` file of the emulator does contain the string "test-keys":

```
vbox86p:/ # find / -type f -name build.prop -exec cat {} \; 2>> /dev/null | grep
tags
ro.build.tags.geny-def=test-keys
```

Possible bypass The `build.prop` file can be edited and the value of the `TAGS` entry can thus be changed to any arbitrary value.

6.3.2 Checking running processes

Since Android API 26 (Android Oreo), applications don't have access to other application's status. This means that several methods became unusable with this version. For example, `getRunningServices` only returns the caller's own services.

As a bypass to this restriction I tried iterating over the files in `/proc/<pid>` and to look for their status file. The status file contains a lot of interesting information including the process' name which could have been used to check if `su` or `frida-server` were running. Unfortunately, the kernel authorized reading only the application process' file and other system files but forbid me to access other processes' files.

6.3.3 Checkig if su is on the PATH

Since emulators provided by Genymotion are all rooted by default, su is already installed on the device and can be found in the directory `/system/bin` that is on the `PATH` environment variable:

```
vbox86p:/ # echo $PATH
/sbin:/system/sbin:/system/bin:/system/sbin:/vendor/bin:/vendor/sbin
vbox86p:/ # ls /system/bin | grep -w "su"
su
```

Possible bypass This check could be bypassed by renaming the `su` utility.

6.3.4 Check files presence

Since emulators provided by Genymotion are all rooted by default, some files from the list depicted in section 4.2.2 are already installed on the device. To be more specific, the files `/system/bin/su`, `/system/sbin/su` and `/system/sbin/busybox` are present on the test emulator.

Possible bypass This check relies on an up-to-date list of suspicious file names. Renaming those files on the target device could effectively bypass this check.

6.4 Debug detection and prevention

6.4.1 Checking the debugger presence

This check works as expected in detecting if a JDWP debugger is attached to the application. To trigger this check I attached a JDWP debugger to the target app:

```
$adb forward tcp:7777 jdwp:1826
$jdb -attach localhost:7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
```

In this example 1826 is the pid of the target application.

Possible bypass This protection can be bypassed with a dynamic instrumentation tool such as Frida or Xposed. Hooking the `Debug` API and changing the return value of the `isDebuggerConnected` to false would bypass this check.

6.4.2 Checking TracerPid

To verify the efficiency of this check the target application had to be actually debugged. To do so I pushed the `gdbserver` utility shipped with the Android NDK on my test emulator:

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

Then I attached `gdbserver` to the application's process:

```
$ adb shell "/data/local/tmp/gdbserver --attach localhost:12345 17566"
```

Here 12345 is the port on which `gdbserver` will wait for incoming debugging traffic and 17566 is the `pid` of the process being debugged.

Now `gdb` can be used on my host machine to debug the attached process:

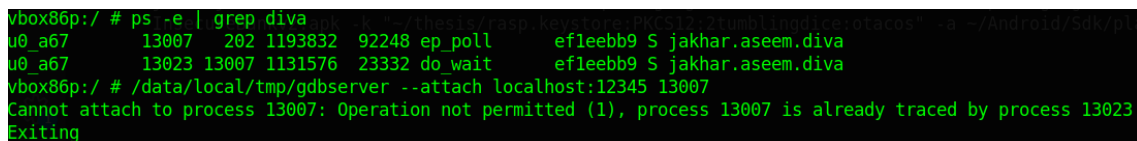
```
$ $TOOLCHAIN/bin/gdb
GNU gdb (GDB) 7.11
(...)
(gdb) target remote 192.168.57.1:12345
Remote debugging using :12345
(gdb) continue
Continuing.
```

From the moment `gdbserver` is attached, the application is paused and we cannot interact with it. In order to resume it and be able to navigate the app to trigger the right logic bomb, it is important to use the `continue` command in `gdb` to resume the app's execution.

Possible bypass This check relies on the `libc` library and its methods. Methods belonging to `libc` such as `read` or `open` could be hooked by Frida to modify their return value and effectively hide the fact that the application is being debugged.

6.4.3 Debug prevention

This check works as expected and prevents the application from being debugged as shown in figure 6.1. One downside to the implementation of this check with the ARMANDroid project is that to be the most effective, this protection should be triggered as soon as possible when the application is started. Unfortunately, the ARMAND scheme does not provide a lot of control over the placement of the logic bombs by design. That means that this check fails if an attacker attached a debugger before this protection was triggered, making it ineffective.



```
vbox86p:/ # ps -e | grep diva
u0_a67      13007    202 1193832  92248 ep_poll    efleebb9 S jakhar.aseem.diva
u0_a67      13023 13007 1131576  23332 do_wait    efleebb9 S jakhar.aseem.diva
vbox86p:/ # /data/local/tmp/gdbserver --attach localhost:12345 13007
Cannot attach to process 13007: Operation not permitted (1), process 13007 is already traced by process 13023
Exiting
```

Figure 6.1 – Gdbserver fails to attach to the main thread of the application

Possible bypass Besides the fact that this check should be triggered as soon as possible in the application lifecycle to be the most effective, this mechanism is still very powerful. An attacker wanting to bypass it would have to patch the call to `exit()` in the `libc` library to make it ineffective.

6.5 Frida detection

6.5.1 Checking loaded libraries

For this check, I used the latest version of `frida-server` (v15.0.8 at the time of writing) which I downloaded from its release repository. I also installed the utilities needed to

communicate with the instance of `frida-server` on the device, `frida-tools`:

```
$ pip install frida-tools
```

To install `frida-server` on the emulator, I used `adb push`:

```
$ adb push frida-server /data/local/tmp
```

Then I changed its rights to make it executable:

```
$ adb shell "chmod 755 /data/local/tmp/frida-server"
```

And finally I launched it in the background:

```
$ adb shell "/data/local/tmp/frida-server &"
```

In order for Frida to inject the `frida-agent` into the target application, I used the `frida-trace` command to perform method tracing as discussed in section 3.7.1. To do so I first need the device id of my emulator which I can get with the command `frida-ls-devices`:

```
$frida-ls-devices
Id Type Name
-----
local local Local System
192.168.57.104:5555 usb Google Nexus 6
socket remote Local Socket
```

Next I run `frida-trace` with either the package name or the `pid` of the application I want to instrument; in the following example I used its `pid`:

```
$ frida-trace - "read*" --attach-pid=9601 -D 192.168.57.104:5555
```

Here, `read*` means I am tracing all functions whose name starts with `read` and the option `-D` specifies the device id of my emulator.

Possible bypass Since this check looks for library with the string "frida" in their names, renaming the libraries would bypass this check.

6.5.2 Default port connection

The utility `frida-server` listens by default on port 27042. If this specific port is open, that could signify that `frida-server` is installed on the device. To trigger this check, the `frida-server` utility must be installed on the device and launched as shown in the previous check.

Possible bypass The Frida tool allows the user to specify an arbitrary port as an argument which would make this check ineffective.

6.6 Summary

My goal in this chapter was to show the effectiveness of the injected protections with log messages. As showcased in the previous chapter, a mechanism to shut down the application silently was also implemented and works effectively. Through the presented methodology, the reader should now have the ability to recreate my experiments and trigger the security checks the same way I did to verify that they are indeed working. The presented work is not perfect though, and I also introduced bypass mechanisms that work in theory but weren't tested. Indeed, the presented checks are not the most elaborate there are, but were the most accessible for a project of this size. Further work can definitely be done on all the subjects discussed to improve the security measures and implement more complex protections.

As always in information security, data protection is a cat and mouse game; for every new threat a protection mechanism is developed that will eventually be bypassed by some technique which in turn will be mitigated and so on. The next chapter will elaborate on this principle, showing the current limitations of my work as well as potential improvements.

Chapter 7

Discussion

7.1 Introduction

The final chapter of this thesis will explore the limitations of my work and the different aspects that could be improved. I focused my work on the implementation of a series of security measures aimed at protecting against certain types of threats, but others exist that could be mitigated using the mechanisms showcased in this thesis. Work can also be done to better protect against the threats tackled in this document. More elaborate defense schemes exist that can be used to protect further Android applications.

At the end of this chapter I will also provide closing words to this thesis and conclude my work.

7.2 Possible Improvements

7.2.1 More protections granularity

In its current state, this project adds logic bombs at specific places in the code which contain security measures. Those logic bombs are only injected in qualified conditions as discussed in section 4.8.1, which means that no one has real control over their location. A possible improvement scenario could be to add more control over the features which are blocked in the event of a threat detection. From a more technical point of view, we could create artificial qualified conditions at specific locations in the code that would then get triggered when using specific features. This could work side by side with the current implementation and improve it, not replace it.

For example, we could imagine that if the device is rooted some features might get blocked but not others. This would allow the management team of the company that owns the application to restrict the access to only some parts of their application to let users who have a rooted device still use non-critical features and put controls only where it is necessary.

7.2.2 Notification to a remote server

When a threat is detected, a notification could be sent to a monitoring server or SIEM (Security Information and Event Management) of the company, that could use these data to build statistics. This solution could then become part of a bigger infrastructure with possibly a web application to allow easier management and monitoring of the applications protected by this RASP.

7.2.3 Assets injection for better maintenance

Since the proposed scheme is signature-based, maintenance of the different signatures or artifacts is of prime importance. At the moment, all those signatures are hardcoded in the different checks for security reasons. Indeed, putting those signatures in a file, or, even better, multiple files, (we could imagine one file for each signature type e.g. one for

suspicious package names needed for root detection and one for the build.prop artifacts needed for emulator detection) and reading this file from a security check could offer easier maintenance but this would provide an attacker an easy bypass; all they would have to do would be to empty it.

A better solution would then be to encrypt the file to prevent its tampering. The key needed to decrypt and encrypt the file could be random and chosen at compile time and then placed within each logic bomb that needs it. That way the random key would then be encrypted with the rest of the code within that logic bomb and could be used to decrypt it at runtime when needed by the security checks of that logic bomb.

A prototype of this solution without the encryption logic is shown in appendix A.14

7.2.4 Addition of modes

The project presented in this thesis had two main objectives: detecting and preventing threats. Those two objectives could lead to an additional configuration option where the company using this product could choose between only detecting threats or also preventing them. This distinction is reminiscent of various security technologies such as IDSs (Intrusion Detection Systems) and IPSs (Intrusion Prevention Systems).

7.2.5 Protections improvements

As explained in the introduction of this chapter, my work in this thesis focused on mitigating dynamic analysis but there are other threats to Android applications. For example, when protecting against hooking frameworks, I focused my work on the Frida tool but Xposed is another powerful tool whose presence can be considered a threat. Work on detecting magisk could also be done as it is the most popular rooting tool at the moment. The debugging prevention feature I implemented in this project works only for native level debugging but JDWP debugging could also be mitigated by other mechanisms. Shutting down the application entirely from the native layer is also possible by recreating the same process as the one in the Java layer with JNI methods. This would ensure that a debugging effort with a native debugger would be fully mitigated.

At the moment and only in its public version, ARMANDroid uses the obfuscapk project to obfuscate the modified target application. This project only targets Java code and only encrypts resources such as native libraries. Obfuscation of native libraries can be improved further by using tools such as LLVM [8]. Using this tool, it is possible to modify bytecode of native libraries to obfuscate them as wanted.

7.2.6 Using a complementary behavior based approach

The approach used in this thesis to check for a safe environment relies entirely on artifacts to detect threats. Without an up-to-date list of known artifacts (or signatures), this approach loses in efficiency. Such a technique could be completed by an anomaly based approach where anomaly in the system are detected through machine learning algorithms which indicate an unsafe environment.

For example, many researches have been conducted on the topic of root detection using a behavioral approach through machine learning. Things like high number of installed applications on a device, or an increased network traffic can be indicators that a device is rooted [17]. Another example of such an approach is emulator detection; studies show that emulators tend to call more methods from the `libc` library than physical devices or that

real devices tend to make more system calls from other libraries than Bionic's libc [32]. Those indicators can be fed to a machine learning algorithm and help detecting the use of an emulator.

7.3 Conclusion

The motivation behind this work was the ever increasing number of Android applications and the growth of their attack surface. Even if Google is working to improve security of the platform with each release of their operating system, new attacks are emerging every day and applications will need more and more defense capabilities. What I wanted to demonstrate throughout this thesis is that even though Android's design is open by nature and thus prone to various attacks, defense mechanisms exist and can be combined to help protect applications. I think RASP products have a future in Android security as they can provide excellent security while reducing security stress in applications developers who can focus on the actual features and design of their apps.

In this thesis I analyzed the different threats to Android applications in the form of the tools and techniques used by adversaries to inspect the behavior of applications. This first analysis helped me identify the key points that needed protection to increase the security of applications. It then lead me to explore the numerous defense mechanisms against those attacks that have been developed along the years by the community of Android developers and security professionals. My contribution was then to select some of these defense mechanisms and integrate them in the ARMANDroid project. The chosen defense mechanisms include root and emulator detection as making sure that applications run in a safe environment can mitigate a lot of attacks. I also worked on debug detection and prevention as debuggers are powerful tools that can inspect and modify memory of applications at runtime. Finally I worked on detecting the Frida tool as it is widely used by attackers to perform dynamic analysis on applications. All those protections were implemented and tested in an effort to produce a fully functional solution.

The challenge of my work was to provide strong security through a single, cohesive and functional solution. The logic bomb mechanism and more broadly, the ARMAND anti-tampering scheme are the keystone of this project as they ensure that the added protections won't be easily removed or bypassed by adversaries. Since work can still be done on the subject of this thesis, I hope that this thesis will serve as a baseline to future students or researchers that want to help build a safer cyberspace for Android applications.

The different security features I implemented all work towards the goal of security protections that act as additional security layers in a bigger security scheme. Indeed, the solution presented in this thesis is no silver bullet and is not to be used as a single protection against attacks. It should be used as part of the defense strategy of a company developing Android applications and therefore work together with other traditional security products such as firewalls and so on.

It is important to keep in mind that security is a cycle and not a one shot effort. As new threats arise, defenders will find mitigation to those threats. In return adversaries will find counter measures and bypass techniques to those protections. Good security thus relies on a perpetual effort to find vulnerabilities and mitigate them in a never ending battle against adversaries.

Bibliography

- [1] Angr [Cited on page 20.]
- [2] Apktool [Cited on page 16.]
- [3] ASM java [Cited on page 21.]
- [4] Frida [Cited on page 19.]
- [5] Gperftools, <https://github.com/gperftools/gperftools> [Cited on page 59.]
- [6] Javassist [Cited on page 22.]
- [7] Kernel virtual machine, https://www.linux-kvm.org/page/Main_Page [Cited on page 15.]
- [8] Llvm, <https://llvm.org/> [Cited on page 67.]
- [9] Magisk [Cited on page 13.]
- [10] Quickjs [Cited on page 19.]
- [11] Rootinspector, <https://github.com/devadvance/rootinspector/blob/master/app/src/main/java/com/devadvance/rootinspector/Root.java> [Cited on page 32.]
- [12] Runtime application self-protection, <https://www.gartner.com/en/information-technology/glossary/runtime-application-self-protection-rasp> [Cited on page 1.]
- [13] Soot [Cited on pages 21 and 49.]
- [14] Symbolic execution, https://en.wikipedia.org/wiki/Symbolic_execution [Cited on page 19.]
- [15] Xposed [Cited on page 25.]
- [16] Aonzo, S., Georgiu, G.C., Verderame, L., Merlo, A.: Obfuscapk: An open-source black-box obfuscation tool for android apps. SoftwareX 11, 100403 (2020) [Cited on page 43.]
- [17] B., Y.S.N.E.A.: Insights into rooted and non-rooted android mobiledevices with behavior analytics (apr 2016) [Cited on page 67.]
- [18] Bellard, F.: QEMU, a fast and portable dynamic translator. In: 2005 USENIX Annual Technical Conference (USENIX ATC 05). USENIX Association, Anaheim, CA (Apr 2005) [Cited on page 14.]
- [19] Benfield, L.: Cfr, <https://github.com/leibnitz27/cfr> [Cited on page 16.]
- [20] Chebyshev, V., Unuchek, R.: Mobile malware evolution: 2013. Tech. rep., Kaspersky (2013) [Cited on page 1.]
- [21] decompiler, J.: Jd, <https://github.com/java-decompiler/jd-gui> [Cited on page 16.]

- [22] Dexguard: Proguard [Cited on page 43.]
- [23] dineshshetty: <https://github.com/dineshshetty/Android-InsecureBankv2> [Cited on page 60.]
- [24] documentation, A.: Platform Architecture [Cited on page 5.]
- [25] of European Union, C.: Commission delegated regulation (eu) 2018/389 of 27 november 2017 supplementing directive (eu) 2015/2366 of the european parliament and of the council with regard to regulatory technical standards for strong customer authentication and common and secure open standards of communication (2017), <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32018R0389> [Cited on page 3.]
- [26] Google: Safetynet, <https://developer.android.com/training/safetynet> [Cited on page 30.]
- [27] GuardSquare: Dexguard introduces code virtualization for android apps (jan 2019), <https://www.guardsquare.com/blog/dexguard-introduces-code-virtualization-android> [Cited on page 42.]
- [28] IDC: (nov 2012), <https://web.archive.org/web/20121103041944/http://www.idc.com/getdoc.jsp?containerId=prUS23771812> [Cited on page 1.]
- [29] LLP, P.S.L.: <https://github.com/payatu/diva-android> [Cited on page 60.]
- [30] Long N., N.T.C.S.K.S.J.: Android rooting: An arms race between evasion and detection. Hindawi 2017 (2017) [Cited on page 13.]
- [31] M., B.: The jiu-jitsu of detecting frida, <https://web.archive.org/web/20181227120751/http://www.vantagepoint.sg/blog/90-the-jiu-jitsu-of-detecting-frida> [Cited on page 59.]
- [32] N., A.G.H.B.S.: Differences in android behavior between realdevice and emulator: A malware detectionperspective (2019) [Cited on page 68.]
- [33] Oracle: Java native interface specification, <https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html> [Cited on page 53.]
- [34] OWASP: Mobile security testing guide. gitbook [Cited on pages 11, 13, and 21.]
- [35] pxb1988: dex2jar [Cited on pages 16 and 22.]
- [36] Skylot: Jadx, <https://github.com/skylot/jadx> [Cited on pages 16 and 60.]
- [37] Smith, R. Strazzere, T.: Dragon lady: An investigation into the industry behind the majority of russian-made malware. Tech. rep., Lookout (2013) [Cited on page 1.]
- [38] V., A.M.A.R.L.S.L.: Armand: Anti-repackaging through multi-pattern anti-tampering based on native detection [Cited on page 2.]
- [39] VirtualBox: https://www.virtualbox.org/wiki/Developer_FAQ [Cited on page 15.]
- [40] Xu, H., Zhou, Y., Ming, J., Lyu, M.: Layered obfuscation: a taxonomy of software obfuscation techniques for layered security. Cybersecurity 3(1), 9 (Apr 2020) [Cited on page 41.]

Appendix A

Source code

A.1 Ptrace debugging prevention

```
#include <jni.h>
#include <unistd.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <pthread.h>

static int child_pid;

void *monitor_pid() {

    int status;

    waitpid(child_pid, &status, 0);

    /* Child status should never change. */

    _exit(0); // Commit seppuku
}

void anti_debug() {

    child_pid = fork();

    if (child_pid == 0)
    {
        int ppid = getppid();
        int status;

        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
            waitpid(ppid, &status, 0);

            ptrace(PTRACE_CONT, ppid, NULL, NULL);

            while (waitpid(ppid, &status, 0)) {

                if (WIFSTOPPED(status)) {
                    ptrace(PTRACE_CONT, ppid, NULL, NULL);
                } else {
                    // Process has exited
                    _exit(0);
                }
            }
        }
    }
}
```

```
    }

    } else {
        pthread_t t;

        /* Start the monitoring thread */
        pthread_create(&t, NULL, monitor_pid, (void *)NULL);
    }
}

JNIEXPORT void JNICALL
Java_sg_vantagepoint_antidebug_MainActivity_antidebug(JNIEnv *env, jobject
instance) {

    anti_debug();
}
```

A.2 Root detection - checking for suspicious files presence

```
public static void checkFilePresence(Context context) {
    if (context == null)
        return;

    try {
        Log.d("ROOT_DETECTION", "CHECK_FILE_PRESENCE");

        String[] root_files = {
            "/system/app/Superuser.apk",
            "/system/etc/init.d/99SuperSUDaemon",
            "/dev/com.koushikdutta.superuser.daemon/",
            "/system/xbin/daemonsu",
            "/sbin/su",
            "/system/bin/su",
            "/system/bin/failsafe/su",
            "/system/xbin/su",
            "/system/xbin/busybox",
            "/system/sd/xbin/su",
            "/data/local/su",
            "/data/local/xbin/su",
            "/data/local/bin/su",
        };

        for (String file_name : root_files) {
            if (new File(file_name).exists()){
                Log.d("ROOT_DETECTION", "Detected file: " + file_name);
                throw new RuntimeException("ROOT_DETECTION: " + "Detected
                    file: " + file_name);
            }
        }
    } catch (RuntimeException e) {
        Log.d("ROOT_DETECTION", "Closing app");
    }
}
```

```

        List<ActivityManager.AppTask> appTasks =
            ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks()

        for (ActivityManager.AppTask task : appTasks) {
            task.finishAndRemoveTask();
        }
    }
}

```

A.3 Root detection - checking for suspicious package names

```

public static void checkInstalledPackages(Context context) {
    if (context == null)
        return;

    try {
        String[] package_names = {
            "com.thirdparty.superuser",
            "eu.chainfire.supersu",
            "com.noshufou.android.su",
            "com.koushikdutta.superuser",
            "com.zachspeng.temprootremovejb",
            "com.ramandroid.appquarantine",
            "com.topjohnwu.magisk",
        };

        Log.d("ROOT_DETECTION", "CHECK_INSTALLED_PACKAGES");

        PackageManager packageManager = context.getPackageManager();
        List<ApplicationInfo> installedPackages =
            packageManager.getInstalledApplications(PackageManager.GET_META_DATA);

        for (int i = 0; i < installedPackages.size(); i++) {
            ApplicationInfo packageInfo = installedPackages.get(i);

            for (String package_name : package_names) {
                if (packageInfo.packageName.equals(package_name)) {
                    //Log.d("ROOT_DETECTION", "detected package name " +
                        package_name);
                    throw new RuntimeException("ROOT_DETECTION: detected package
                        name " + package_name);
                }
            }
        }
    } catch (RuntimeException e) {
        //Log.d("ROOT_DETECTION", "Closing app");
        List<ActivityManager.AppTask> appTasks =
            ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks();

        for (ActivityManager.AppTask task : appTasks) {
            task.finishAndRemoveTask();
        }
    }
}

```

```

    }
}
}

```

A.4 Root detection - checking if su is on PATH

```

public static void checkSuOnPath(Context context) {
    if (context == null)
        return;

    try {
        Log.d("ROOT_DETECTION", "CHECK_SU_ON_PATH");

        for (String pathDir : System.getenv("PATH").split(":")){
            if (new File(pathDir, "su").exists()) {
                //Log.d("ROOT_DETECTION", "su is on the PATH");
                throw new RuntimeException("ROOT_DETECTION: su is on the PATH");
            }
        }
    } catch (RuntimeException e) {
        //Log.d("ROOT_DETECTION", "Closing app");
        List<ActivityManager.AppTask> appTasks =
            ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks();

        for (ActivityManager.AppTask task : appTasks) {
            task.finishAndRemoveTask();
        }
    }
}

```

A.5 Root detection - checking for test key

```

public static void isTestKeyBuild(Context context) {
    if (context == null)
        return;

    try {
        String tags = Build.TAGS;

        if (tags != null && tags.contains("test-keys")) {
            //Log.d("ROOT_DETECTION", "CHECK_TEST_KEY");
            throw new RuntimeException("ROOT_DETECTION check test key failed");
        }
    } catch (RuntimeException e) {
        //Log.d("ROOT_DETECTION", "Closing app test key");
        List<ActivityManager.AppTask> appTasks =
            ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks();

        for (ActivityManager.AppTask task : appTasks) {

```

```

        task.finishAndRemoveTask();
    }
}
}

```

A.6 Emulator detection - checking suspicious package names

```

public static void packageCheck(Context context) {
    if (context == null)
        return;

    try {
        PackageManager packageManager = context.getPackageManager();
        List<ApplicationInfo> installedPackages =
            packageManager.getInstalledApplications(PackageManager.GET_META_DATA);

        for (int i = 0; i < installedPackages.size(); i++) {
            ApplicationInfo packageInfo = installedPackages.get(i);
            if (packageInfo.packageName.contains("geny")) {
                //Log.d("EMULATOR DETECTION", "Package detected " +
                    packageInfo.packageName);
                throw new RuntimeException("EMULATOR_DETECTION: Package detected
                    " + packageInfo.packageName);
            }
        }
    } catch (RuntimeException e) {
        //Log.d("EMULATOR_DETECTION", "Closing app");
        List<ActivityManager.AppTask> appTasks = ((ActivityManager)
            context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks();

        for (ActivityManager.AppTask task : appTasks) {
            task.finishAndRemoveTask();
        }
    }
}

```

A.7 Emulator detection - checking for artifacts in build.prop

```

public static void buildCheck(Context context) {
    if (context == null)
        return;

    try {
        boolean ABIS_artifacts = false;

        for (String supported_abi : Build.SUPPORTED_ABIS) {
            if (supported_abi.contains("unknown") ||
                supported_abi.contains("armeabi")) ABIS_artifacts = true;
        }
    }
}

```

```
boolean isEmulator = ABIS_artifacts
    || Build.BOARD.equals("unknown")
    || Build.HOST.equals("android-test")
    || Build.ID.equals("FRF91")
    || Build.MANUFACTURER.equals("unknown")
    || Build.MODEL.equals("sdk")
    || Build.getRadioVersion().equals("unknown")
    || Build.getSerial() == null
    || Build.USER.equals("android-build");

if (isEmulator) {
    //Log.d("EMULATOR DETECTION", "Build check FAIL");
    throw new RuntimeException("EMULATOR_DETECTION: Build check
        FAIL");
} else {
    Log.d("EMULATOR DETECTION", "Build check PASS");
}
} catch (RuntimeException e) {
    //Log.d("EMULATOR_DETECTION", "Closing app");
    List<ActivityManager.AppTask> appTasks =
        ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks()

    for (ActivityManager.AppTask task : appTasks) {
        task.finishAndRemoveTask();
    }
}
}
```

A.8 Emulator detection - checking the TelephonyManager API

```
public static void telephonyCheck(Context context) {
    if (context == null)
        return;

    try {
        TelephonyManager telephonyManager = (TelephonyManager)
            context.getSystemService(Context.TELEPHONY_SERVICE);

        @SuppressWarnings("MissingPermission") boolean isEmulator =
            telephonyManager.getDeviceId().equals("0's")
                || telephonyManager.getLine1Number().equals("155552155")
                || telephonyManager.getNetworkCountryIso().equals("us")
                || telephonyManager.getNetworkType() == 3
                || telephonyManager.getNetworkOperator().substring(0,
                    3).equals("310")
                ||
                telephonyManager.getNetworkOperator().substring(3).equals("260")
                || telephonyManager.getPhoneType() == 1
                || telephonyManager.getSimCountryIso().equals("us")
    }
}
```

```

        ||
        telephonyManager.getSimSerialNumber().equals("89014103211118510720")
        || telephonyManager.getSubscriberId().equals("310260000000000")
        || telephonyManager.getVoiceMailNumber().equals("15552175049");

    if (isEmulator) {
        //Log.d("EMULATOR_DETECTION", "Telephony check FAIL");
        throw new RuntimeException("EMULATOR_DETECTION: Telephony check
            FAIL");
    }
} catch (SecurityException e) {
    return;
} catch (RuntimeException e) {
    //Log.d("EMULATOR_DETECTION", "Closing app");
    List<ActivityManager.AppTask> appTasks = ((ActivityManager)
        context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks();

    for (ActivityManager.AppTask task : appTasks) {
        task.finishAndRemoveTask();
    }
}
}
}

```

A.9 Debug detection - checking if a JDWP debugger is attached

```

public static void detectDebugger(Context context) {
    if (context == null)
        return;

    try {
        if (Debug.isDebuggerConnected()) {
            Log.d("DEBUG_DETECTION", "Debugger is connected");
        }
    } catch (RuntimeException e) {
        //Log.d("DEBUG_DETECTION", "Closing app");
        List<ActivityManager.AppTask> appTasks =
            ((ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE)).getAppTasks();

        for (ActivityManager.AppTask task : appTasks) {
            task.finishAndRemoveTask();
        }
    }
}
}

```

A.10 Debug detection - checking if a native debugger is attached

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include <dlfcn.h>
#include <bits/sysconf.h>
#include <asm/mman.h>
#include <sys/mman.h>
#include <unordered_map>
#include <link.h>
#include <elf.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include "libs/nativeutils/utils.h"
#define log(...) __android_log_print(ANDROID_LOG_DEBUG, "NATIVE", __VA_ARGS__);

%s
log("DEBUG DETECTION")
char buf[256]; // TracerPid comes relatively earlier in status output
int fd = open("/proc/self/status", O_RDONLY);

if (fd == -1) {
    log("Could not read file");
    return; // Can't tell for sure.
}

const int len = read(fd, buf, sizeof(buf));
bool rc = false;

if (len > 0) {
    const char *const kTracerPid = "TracerPid:\t";
    buf[len - 1] = '\0';
    const char *p = strstr(buf, kTracerPid);

    if (p != NULL) {
        rc = (strncmp(p + strlen(kTracerPid), "0\n", 2) != 0);
    }
}

close(fd);

if (rc) {
    log("Debugger is attached");
} else {
    log("Debugger not detected");
}
}
```

A.11 Debug prevention

A.11.1 Main method in DebugPrevention.cpp

```

#include <jni.h>
#include <string>
#include <android/log.h>
#include <dlfcn.h>
#include <bits/sysconf.h>
#include <asm/mman.h>
#include <sys/mman.h>
#include <unordered_map>
#include <link.h>
#include <elf.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include "libs/nativeutils/utils.h"
#define log(...) __android_log_print(ANDROID_LOG_DEBUG, "NATIVE", __VA_ARGS__);

%s

log("DEBUG DETECTION")
char buf[256]; // TracerPid comes relatively earlier in status output
int fd = open("/proc/self/status", O_RDONLY);

if (fd == -1) {
    log("Could not read file");
    return; // Can't tell for sure.
}

const int len = read(fd, buf, sizeof(buf));
bool rc = false;

if (len > 0) {
    const char *const kTracerPid = "TracerPid:\t";
    buf[len - 1] = '\0';
    const char *p = strstr(buf, kTracerPid);

    if (p != NULL) {
        rc = (strncmp(p + strlen(kTracerPid), "0\n", 2) != 0);
    }
}

close(fd);

if (rc) {
    log("Debugger is attached");
} else {
    log("Debugger not detected");
}

```

```
}
```

A.11.2 Secondary method in utils.cpp

```
void* monitor_pid(void* arguments) {
    int child_pid = *((int *) arguments);

    int status;

    waitpid(child_pid, &status, 0);

    kill(getpid(), SIGKILL); // Commit seppuku*/
}
```

A.12 Frida detection - checking if default port is open

```
void* check_default_port(void* ) {
    struct sockaddr_in sa;
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    inet_aton("127.0.0.1", &(sa.sin_addr));

    int i;
    int sock;
    int ret;
    char res[1024];
    char buff[5];

    if ((sock = socket(AF_INET , SOCK_STREAM , 0)) < 0) {
        log("unable to create socket");
        return NULL;
    }

    sa.sin_port = htons(27042);

    if (connect(sock , (struct sockaddr*)&sa , sizeof sa) != -1) {
        log("Default Frida port is open");
    }

    close(sock);
    return NULL;
}
```

A.13 Frida detection - checking loaded libraries

```
#include <jni.h>
#include <string>
#include <android/log.h>
```

```

#include "libs/nativeutils/utils.h"
#define log(...) __android_log_print(ANDROID_LOG_DEBUG, "NATIVE", __VA_ARGS__);

%s

char line[512];
FILE* fp;

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

if (fp) {
    while (fgets(line, 512, fp)) {
        if (strstr(line, "frida")) {
            log("frida library detected !");
            _exit(0);
        }
    }

    fclose(fp);
    log("frida library not detected");

} else {
    log("could not open /proc/self/maps");
}
}

```

A.14 Assets injection prototype

```

public static void injectAssets() {
    if (Options.v().process_dir().size() != 1) {
        throw new IllegalArgumentException("Not a single APK was provided");
    }

    try {
        Path sourceAPKPath = ManifestHelper.getAPKOutputPath(); // Path of the
        transformed apk before injecting assets
        String inputDir = "/tmp/test/assets/";
        JarFile jf = new
        JarFile(URLDecoder.decode(ApplicationHelper.class.getProtectionDomain().getCodeSource().get
        "UTF-8"));
        for (Enumeration<JarEntry> it = jf.entries(); it.hasMoreElements();) {
            // iterating through all the ARMANDroid jar's entries
            JarEntry entry = it.nextElement();
            if (entry.getName().startsWith("bin/assets/") &&
                !entry.isDirectory()) { // looking for the right files
                InputStream inputStream =
                NativeHelper.class.getClassLoader().getResourceAsStream(entry.getName());
                File f = new File(inputDir + entry.getName().split("/")[2]);
                FileUtils.copyInputStreamToFile(inputStream, f); // copying the files
                to the temp directory
            }
        }
    }
}

```

```
    }  
    // creating a temporary apk and adding the assets folder to it  
    File newTempApk = File.createTempFile("new-apk", ".apk");  
    FileOutputStream targetApkOutputStream = new  
    FileOutputStream(newTempApk.getAbsolutePath());  
    DexHelper.mergePathWithAPK(new ZipFile(sourceAPKPath.toString()),  
    Paths.get("/tmp/test"), new ZipOutputStream(targetApkOutputStream));  
    Files.delete(sourceAPKPath);  
    Files.move(newTempApk.toPath(), sourceAPKPath);  
    FileUtils.deleteDirectory(new File(inputDir));  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}
```
