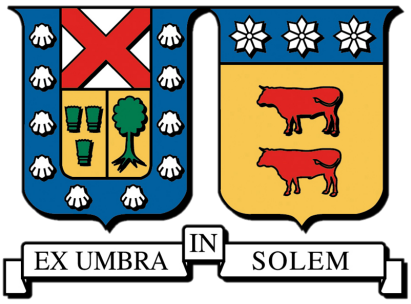
UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

DEPARTAMENTO DE INFORMATICA

CIUDAD - CHILE



INGENIERÍA INVERSA DE APLICACIONES MÓVILES PARA DESCUBRIMIENTO DE VULNERABILIDADES: AMENAZAS Y DEFENSAS

DANIEL FRANCISCO TAPIA RYBERTT

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN INFORMATICA.

PPROFESOR GUÍA : RAUL MONGE

ABRIL 2017

TABLE OF CONTENTS

List of Tables

Table **Page**

List of Figures

CHAPTER 1

CONCEPTUAL FRAMEWORK

Mobile applications have changed the way we interact with the world around us, now we don't rely on a desktop and constant refreshing of web applications to get the data we need to start our day, nor do we need to stay put in a place with wired connections since we can get our data on the go. Mobile apps help get your data customized by the vendor without having to worry about the device or hardware underneath it, this abstraction helps developers code applications that will work in a wide variety of mobile phones.

As it was in the 1990's with the rise of desktop computers into mainstream use, the market was bombarded with multiple viruses, worms and trojans aimed at users in order to steal, forge, or otherwise break the system it infected. This trend has now caught up to mobile phones, the new frontier.

As of now there are really only two key players in the mobile world, Google and Apple, and both have similar ecosystems with gaping difference in how they approach security. While Apple's approach is to have an App Store where all apps that are published can be downloaded with a press of a button. But before being allowed inside the App Store, an aspiring app must have passed a rigorous testing and verification by Apple, in which they must abide by their terms and license agreement.

Google's approach varies significantly, any developer who wishes to publish into the play store, can do so without a mainstreamed verification process. Google checks apps for security before they enter the Android Market, but there isn't an Android security policy to which all those apps adhere. Android uses sandboxing, which limits how an app can interact with other apps and the OS, thus limiting the effects of any malware as well. But some Android phone malware is designed to trick users into lifting these limits, by presenting seemingly authentic requests for permission to access other apps and OS components. A virus can then use these permissions to

attack the full mobile platform. For the most part, Android app security is good, but the chance for malicious apps to make their way into the Android Market is still present [**?**].

1.1 Android OS

Android is an open source, Linux-based software stack created for a wide array of devices and form factors. The following diagram shows the major components of the Android platform.

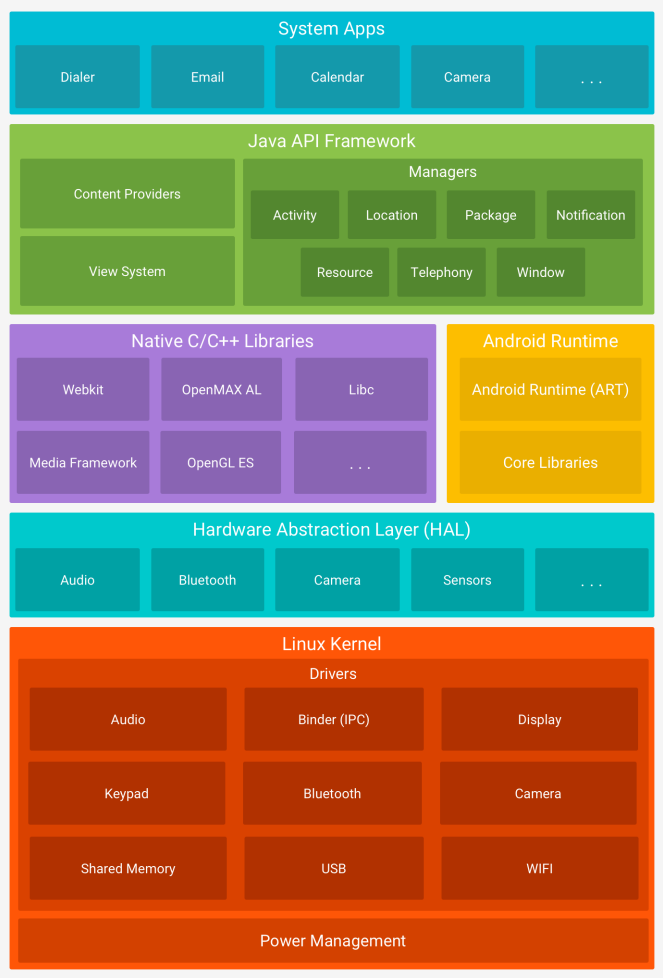


Figure 1.1: Android Stack

1.1.1 Linux Kernel

The foundation of the Android platform is the Linux kernel. For example, the Android Runtime (ART) relies on the Linux kernel for underlying functionalities such as threading and low-level memory management.

Using a Linux kernel allows Android to take advantage of key security features and allows device manufacturers to develop hardware drivers for a well-known kernel.

1.1.2 Hardware Abstraction Layer (HAL)

The hardware abstraction layer (HAL) provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. The HAL consists of multiple library modules, each of which implements an interface for a specific type of hardware component, such as the camera or bluetooth module. When a framework API makes a call to access device hardware, the Android system loads the library module for that hardware component.

1.1.3 Android Runtime

For devices running Android version 5.0 (API level 21) or higher, each app runs in its own process and with its own instance of the Android Runtime (ART). ART is written to run multiple virtual machines on low-memory devices by executing DEX files, a bytecode format designed specially for Android that's optimized for minimal memory footprint. Build tool chains, such as Jack, compile Java sources into DEX bytecode, which can run on the Android platform.

Prior to Android version 5.0 (API level 21), Dalvik was the Android runtime. If your app runs well on ART, then it should work on Dalvik as well, but the reverse may not be true.

1.1.4 Native C/C++ Libraries

Many core Android system components and services, such as ART and HAL, are built from native code that require native libraries written in C and C++. The Android platform provides Java framework APIs to expose the functionality of some of these native libraries to apps. For example, you can access OpenGL ES through the Android framework's Java OpenGL API to add support for drawing and manipulating 2D and 3D graphics.

1.1.5 Java API Framework

The entire feature-set of the Android OS is available through APIs written in the Java language. These APIs form the building blocks needed to create Android apps by simplifying the reuse of core, modular system components and services.

1.2 Mobile Applications

The Android platform suffers the issue of fragmentation – there are multiple versions of Android in the market, even on current devices. Manufacturers often make their own changes to Android, so they could be behind Google's current reference release. In addition, carriers and manufacturers

may not update their devices' Android version when Google does, or they take months or even years to do so.

As a result, many people within the same organization might be using outdated versions that could be riddled with security vulnerabilities. People focus on malware risks of Android, but arguably the greater risk is that fragmentation creates different user experiences, this variety of user experiences makes it hard to educate employees about how to take security measures, because the experience on each device is different. As a result many multiple users end up with different versions of Android which in turn means different versions of any one application.

This difference between versions means that users could be vendor locked into a version of android, for example Samsung has statistically released the next flavor of Android about 6-10 months after Google has released it, leaving users open to attack during this wide window of not being able to get the latest updates and patches.

Any developer suffers the consequences of this delay, from a security standpoint , they must now version an application to be able to support multiple versions of the Android OS, keeping new features off a specific version , etc.

1.3 Threats and Risks in the Android Eco System

Google has placed numerous measures in order to keep apps as secure as possible, one the main security pillars is the Android Security Sandbox (AAD) , which isolates app data and code execution from other apps, it is an application framework with robust implementations of common security functionality such as cryptography, permissions, and secure Inter Process Communication (IPC).

Even though Android makes it easy to publish apps to the Play Store, it makes it way to easy to publish malicious applications, in order to attempt to prevent this a security measure from 2010 that is still in effect today is that every application has to be signed before being installed on a device, but in order to circumvent this, a hacker may use a self signed certificate before publishing to the Play Store, even more appalling is the fact that users may also download apps from third party stores such as slideme.org and androidlib.com.

Apps can't just be installed and access every component of the phone, mainly because of the sandboxing mentioned earlier. In order to achieve full compromise, the user must allow and application permissions, these permissions are defined in advanced by the developer (in a file called AndroidManifest.xml) and the user is prompted with the permission requirements of the app before installation.

1.4 How to prevent mobile threats

There is no silver bullet that can prevent security breaches in the software world, and Android is no different. Preventing reverse engineering on an application is not 100% possible but it be

slowed down quite a bit, techniques such as obfuscation make understanding and modifying the underlying code much harder (but not impossible). In the end, one has to understand that an application can't be protected from modifications and any protections in place can be disabled or removed.

That being said, there are numerous methods for preventing mobile threats such as

1. Usage of tools like ProGuard. These will obfuscate the code, and make it harder to read when decompiled, if not impossible

2. Move the most critical parts of the service out of the app, and into a webservice, hidden behind a server side language like PHP or use the NDK to write them natively into .so files, which are much less likely to be decompiled than apks. Decompiler for .so files don't even exists as of now (and even if they did, it wouldn't be as good as the Java decompilers). Finally the usage of SSL/TLS when interacting between the server and device is mandatory for securing communication.

3. When storing values on the device, don't store them in a raw format, but instead use an algorithm to calculate the real value. Although this approach does in a way protect against tampering to obtain actual values (an attacker can still modify the entry value and get random results) it is not a security measure.

4. In cases of payment processing apps, sending of raw data is not always the most secure way of processing the data, one approach to prevent interception or modification of values being sent to the server is to have base values that add up to the final value, for example, instead of sending the value 1000 to the server, the value is decomposed into smaller numbers that are assigned to strings, so 1000 could be decomposed as 10*10+100*4+500*1 and the numbers 10,100, and 500 are not sent as number but as names of people such as 10 = Daniel , 100 = Max, 500 = Bob and so on and the backend knows the conversion from string to number. This prevents a hacker from sending values to the endpoint and makes it harder to understand what is happening

5. Insertion of random bits of code that has no meaning in an attempt to confuse whomever is trying to reverse engineer the application. Common practice is to insert classes that will catch the attention of anyone who is looking at the code, for example , compiling classes that generate a Fibonacci sequence called CreditCard.java is guaranteed to make the attacker waste valuable time.

1.4.1 Requirements

The application requires only Java 1.7 (JRE 1.7) and some basic knowledge of Android SDK, AAPT and the Smali language alongside some in depth understanding of computer architecture, assembler and hexadecimal numbers.

1.4.2 Installation

The first library to install is the ia32-libs if the system is running on top a x64 architecture and install the apktool script and the apktool.jar and move both of them to a directory that is in the $PATH variable, for example the /usr/lib/bin directory.

CHAPTER 2

STATE OF THE ART

Reverse Engineering is the process of unearthing information or architecture knowledge from any object, software or process and attempting to modify it and reconstruct it with or without new functionality using the information gained from tearing down the object granularly and the understanding of the process while analyzing its components and inner workings in detail.

2.1 State of the art

2.1.1 As of now

To Reverse engineering an android application, a vast amount of knowledge and understanding of all the technology involved in the development of an application is required. Not only is the Android development know-how a necessity but also the same levels of mastery in regards to the external components involved in the buildings phase.

When hearing the term reverse engineering in the software development world, one most likely associates it to hacking or attacking an application, and while it is true that reverse engineering techniques are used by malicious hackers to find vulnerabilities or exploits to produce viruses, worms, or just malware in general.

Reverse engineering allows the attacker to gain familiarity with the application and its inner workings to locate entry vectors or globally known vulnerabilities or bad practices that couldn't possible be known or found out while externally analyzing the application through testing.

There are many ways to preform reverse engineering , but the standard procedure always consists of obtaining an executable, disassemble it through some tool, scrutinize the output of the disassemble, preform memory mappings and other analysis techniques to view variables and

attempt to obtain the original source code if successful. Even though the retrieval of the source code is not always necessary or even possible, tampering the variables and checking validations is still the best way to go in order to get an idea of what is hiding underneath all that binary code.

In an Android/Java environment the process of compiling a source code and getting an executable consists of the following.

```
.java files -> .class files -> one classes.dex file
```

Preventing this attack has to be the main priority of any enterprise that wishes to avoid code modifications by third parties. Perhaps the most well known method for accomplishing this objective is a technique know as code obfuscation.

2.1.2 What is Obfuscation

In order to make potential attacker's life harder and hide code, what is needed is the ability to perform a process called obfuscation. Obfuscation is a deliberate act of creating code that is difficult for humans to understand. It can be done in various ways: renaming all variables and class names so that they are a gibberish, flattening the directory structure, moving methods between files, adding garbage code, changing strings to int/hex array equivalents etc. It is used in various languages. In Android Java the code is compiled, but in language like Javascript it's sent with plain text - obfuscation might be a logical step there, too. [**?**]

Next up, a comparison between normal Java code and it's obfuscated counterpart.

```
1  /*Example Java function*/
2
3  function NewObject(prefix)
4  {
5      var count=0;
6      this.SayHello=function(msg)
7      {
8            count++;
9            alert(prefix+msg);
10     }
11     this.GetCount=function()
12     {
13           return count;
14     }
15 }
```

```
var obj=new NewObject("Message : ");
obj.SayHello("You are welcome.");
```

```
/*Obfuscated Java code*/

var _0x888f=["\x53\x61\x79\x48\x65\x6C\x6C\x6F", "\x47\x65\x74\x43\x6F\x75\x6E\
x74","\x4D\x65\x73\x73\x61\x67\x65\x20\x3A\x20","\x59\x6F\x75\x20\x61\x72\x65\x
20\x77\x65\x6C\x63\x6F\x6D\x65\x2E"] ;function NewObject(_0xd6eex2){var _0xd6ee
x3=0;this[_0x888f[0]]=function(_0xd6eex4){_0xd6eex3++;alert(_0xd6eex2+_0xd6eex4
);};this[_0x888f[1]]=function(){return _0xd6eex3};}var obj= new NewObject(_0x8
88f[2]);obj.SayHello(_0x888f[3]);
```

In Android development there is a tool called ProGuard, which is responsible for obfuscation and optimization of the code. It has a wide variety of options, but we have to be careful which ones are chosen. ProGuard can sometimes break the code, if used improperly, it can cause errors in code that uses reflection or in libraries that do.

2.1.3 ApkTool

In order to start the reverse engeering process of an apk, the first step is to decompile the application, one way to accomplish this is through ApkTool. This CLI program carries within it the following features.

1. Disassembling resources to nearly original form (including resources.arsc, classes.dex, 9.png. and XMLs)

2. Rebuilding decoded resources back to binary APK/JAR

3. Organizing and handling APKs that depend on framework resources

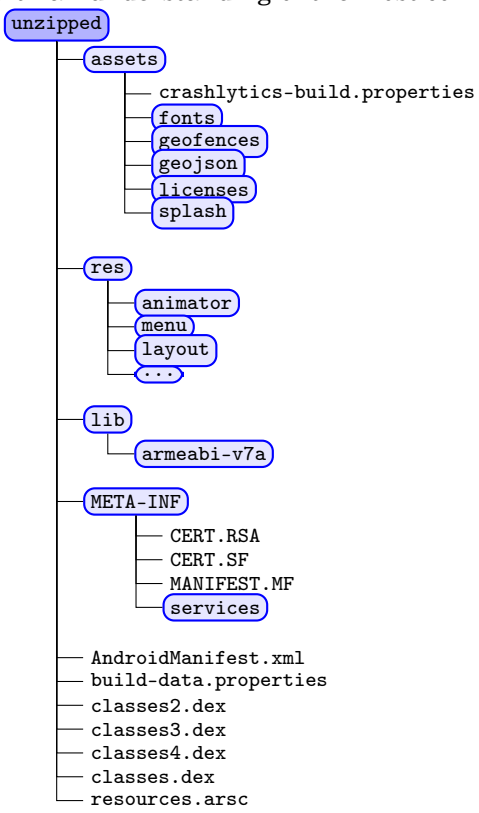
4. Smali Debugging (Removed in 2.1.0 in favor of IdeaSmali)

5. Helping with repetitive tasks

ApkTool is the most mainstream decompiler in android reverse engineering world, it is free , easy to use and well documented.

2.1.4 Analysis of Decompiled APKs

Once an APK has been decompiled, a thorough analysis of the files generated must be made and that begins with an understanding of the most common files and directories found within a decompiled APK.



As it can be seen the .apk format is really just a zipped archive with the binaries compiled, a breakdown of these directories is as follows.

1. classes.dex

   Contains compiled application code, transformed into Dex bytecode. You might see more than one DEX file in your APK if you are using multidex to overcome the 65536 method limit. Beginning with Android 5.0 which introduced the ART runtime, these are compiled into OAT files by the ahead-of-time compiler at install time and put on the device's data partition.

2. res/

   This folder contains most XML resources (e.g. layouts) and drawables (e.g. PNG, JPEG) in folders with various qualifiers, like -mdpi and -hdpi for densities, -sw600dp or -large for screen sizes and -en, -de, -pl for languages. Please note that any XML files in res/ have been transformed into a more compact, binary representation at compile time, so you won't be able to open them with a text editor from inside the APK.

3. resources.arsc

Some resources and identifiers are compiled and flattened into this file. It's normally stored in the APK without compression for faster access during runtime. Compressing this file manually might seem like an easy win, but is actually not a good idea for at least two reasons. One, Play Store compresses any data for transfer anyway and two, having the file compressed inside the APK would waste system resources (RAM) and performance (especially app startup time).

4. AndroidManifest.xml

   Similar to other XML resources, your application Manifest is transformed during compilation into a binary format. Play Store uses certain information contained in the AndroidManifest to decide if an APK can be installed on a device, checking against allowed densities or screen sizes and available hardware and features (such as a touchscreen). If you want to inspect those Manifest entries after compilation, you can use the aapt tool from the Android SDK:

   ```
   $ apt dump badging your_app.apk
   ```

5. libs/ This folder contains shared objects files (.so) which are mostly just C/C++ binaries that android accesses through JNI and must be included in the APK in order to take advantage of methods and calls not available in Java. This folder is sometimes not included in some decompiled apks, depends on the application.

6. assets/

   This folder is used for any file assets that will not be used as Android-type resources. Most commonly this will be font files or game data, like levels and textures, as well as any other application data that you want to open directly as a file stream.

7. META-INF/

   This folder is present in signed APKs and contains a list of all files in the APK with their signatures. The way signing in Android works currently is that it verifies the signatures against uncompressed file contents from the archive, one by one. This has some interesting consequences. Because every entry in a ZIP file is stored separately, this means that you can change individual files' compression level without re-signing. The signature verification will fail however if you remove any file from the archive after it is signed. One more thing to note about how a signed APK is created is that the zipalign tool is used as the last stage of the build. If you change the contents of the archive by hand, normally you will have to re-sign, then zipalign before uploading the APK to the Play Store.

2.2 System Specifications

All programs and builds will be done under my own personal computer which is an ASUS ZenBook UX305, Processor Intel®Core ™5Y10/5Y71 Processor, LPDDR3 1600 MHz SDRAM, 8 GB of RAM, 256GB SSD hard drive, Integrated Intel®HD Graphics 5300 on top of a Deepin OS 15.3 operating system.

CHAPTER 3

ANATOMY OF AN ANDROID APP

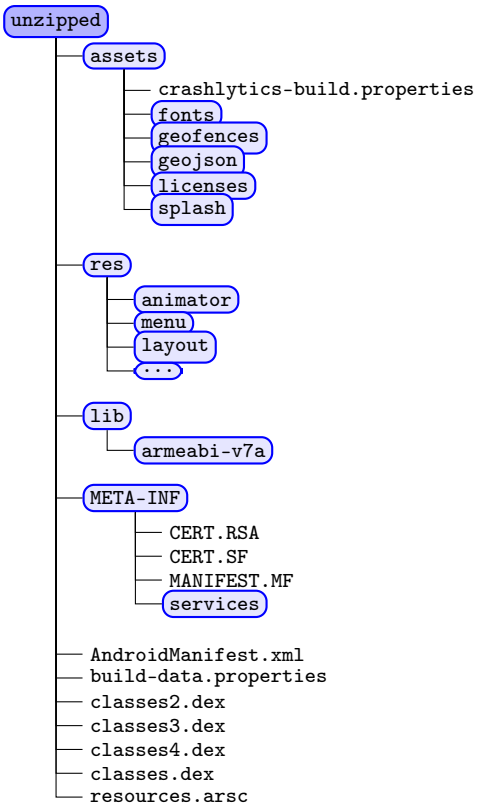
3.0.1 Application

Next an app has to be chosen and it has to be an app that has the .apk extension, The app chosen is the Uber app version 3.126.0, which when downloaded, we obtain a com.ubercab-3.126.0-www.APK4Fun.com.apk file. The procedure that follows is to unzip de .apk and examine the contents, executing unzip command yields the following.

```
danielftapiar $> unzip com.ubercab-3.126.0-www.APK4Fun.com.apk
danielftapiar $> inflating: AndroidManifest.xml
danielftapiar $>    creating: assets/
danielftapiar $>   inflating: assets/crashlytics-build.properties
danielftapiar $>    creating: assets/fonts/
danielftapiar $>   inflating: assets/fonts/ClanPro-Book.otf
danielftapiar $>   inflating: assets/fonts/ClanPro-Medium.otf
danielftapiar $>   inflating: assets/fonts/ClanPro-NarrBook.otf
danielftapiar $>   inflating: assets/fonts/ClanPro-NarrMedium.otf
danielftapiar $>   inflating: assets/fonts/ClanPro-NarrNews.otf
danielftapiar $>   inflating: assets/fonts/ClanPro-News.otf
danielftapiar $> ...
```

```
danielftapiar $> apktool d com.ubercab-3.126.0-www.APK4Fun.com.apk
danielftapiar $> Picked up _JAVA_OPTIONS:  -Dawt.useSystemAAFontSettings
danielftapiar $> I: Using Apktool 2.2.1 on com.ubercab-3.126.0-www.APK4Fu
danielftapiar $> I: Loading resource table...
danielftapiar $> I: Decoding AndroidManifest.xml with resources...
danielftapiar $> I: Loading resource table from file: /home/danielftapiar
danielftapiar $> I: Regular manifest package...
danielftapiar $> I: Decoding file-resources...
danielftapiar $> I: Decoding values */* XMLs...
danielftapiar $> I: Baksmaling classes.dex...
danielftapiar $> I: Baksmaling classes2.dex...
danielftapiar $> I: Baksmaling classes3.dex...
danielftapiar $> I: Baksmaling classes4.dex...
danielftapiar $> I: Copying assets and libs...
danielftapiar $> I: Copying unknown files...
danielftapiar $> I: Copying original files...
```

3.0.2 Directory Listing



As it can be seen the .apk format is really just a zipped archive with the binaries compiled, a breakdown of these directories is as follows.

3.0.3 classes.dex

Contains compiled application code, transformed into Dex bytecode. You might see more than one DEX file in your APK if you are using multidex to overcome the 65536 method limit. Beginning with Android 5.0 which introduced the ART runtime, these are compiled into OAT files by the ahead-of-time compiler at install time and put on the device's data partition.

3.0.4 res/

This folder contains most XML resources (e.g. layouts) and drawables (e.g. PNG, JPEG) in folders with various qualifiers, like -mdpi and -hdpi for densities, -sw600dp or -large for screen sizes and -en, -de, -pl for languages. Please note that any XML files in res/ have been transformed into a more compact, binary representation at compile time, so you won't be able to open them with a text editor from inside the APK.

3.0.5 resources.arsc

Some resources and identifiers are compiled and flattened into this file. It's normally stored in the APK without compression for faster access during runtime. Compressing this file manually might seem like an easy win, but is actually not a good idea for at least two reasons. One, Play Store compresses any data for transfer anyway and two, having the file compressed inside the APK would waste system resources (RAM) and performance (especially app startup time).

3.0.6 AndroidManifest.xml

Similar to other XML resources, your application Manifest is transformed during compilation into a binary format. Play Store uses certain information contained in the AndroidManifest to decide if an APK can be installed on a device, checking against allowed densities or screen sizes and available hardware and features (such as a touchscreen). If you want to inspect those Manifest entries after compilation, you can use the aapt tool from the Android SDK:

```
$ apt dump badging your_app.apk
```