Android Reverse Engineering:

an introductory guide to malware analysis

The Android malware has followed an exponential growth rate in recent years, in parallel with the degree of penetration of this system in different markets. Currently, over 90% of the threats to mobile devices have Android as a main target. This scenario has led to the demand for professionals with a very specific knowledge on this platform.

The software reverse engineering, according to Chikofsky and Cross [1], refers to the process of analyzing a system to identify its components and their interrelationships, and create representations of the system in another form or a higher level of abstraction. Thus, the purpose of reverse engineering is not to make changes or to replicate the system under analysis, but to understand how it was built.

The best way to tackle a problem of reverse engineering is to consider how we would have built the system in question. Obviously, the success of the mission depends largely on the level of experience we have in building similar systems to the analyzed system. Moreover, knowledge of the right tools we will help in this process.

In this article we describe tools and techniques that will allow us, through a reverse engineering process, identify malware in Android applications.

To execute the process of reverse engineering over an application, we can use two types of techniques: static analysis and / or dynamic analysis. Both techniques are complementary, and the use of both provides a more complete and efficient vision on the application being discussed. In this article we focus only on static analysis phase, ie, we will focus on the analysis of the application by analyzing its source code, and without actually running the application.

Static analysis of Android application starts from the moment you have your APK file (Application PacKage). APK is the extension used to distribute and install applications for the Android platform. The APK format is similar to the JAR (Java ARchive) format and contains the packaged files required by the application.

If we unzip an APK file (for example, an APK corresponding to the application "Iron Man 3 Live Wallpaper" available at Play Store: *https://play.google.com/store/apps/details?id=cellfish.ironman3wp&hl=en*):

```
$ unzip cellfish.ironman3wp.apk
```

typically we will find the following resources: Figure 1.

An interesting resource is the "AndroidManifest.xml" file. In this XML file, all specifications of our application are declared, including Activities, Intents, Hardware, Services, Permissions required by the application [2], etc. Note that this is a binary XML

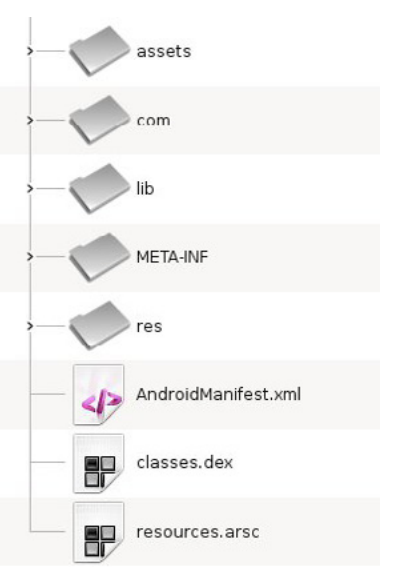

Figure 1. *Typical Structure of an APK File*

file, so if you want to read easily its contents you should convert it to a human-readable XML format.

The "AXMLPrinter2.jar" tool performs this task:

```
$ java -jar AXMLPrinter2.jar AndroidManifest.xml >
                AndroidManifest.xml.txt
$ less AndroidManifest.xml.txt
```

Another important resource that we find in any APK is the "classes.dex" file. This is a formatted DEX (Dalvik EXecutable) file containing the bytecodes that understands the DVM (Dalvik Virtual Machine). Dalvik is the virtual machine that runs applications and code written in Java, created specifically for the Android platform.

Since we want to analyze the source code of the application, we need to convert the DEX format to Java source code. To do this we will pass through an intermediate state. We will convert the DEX format to the compiled Java code (.class). Many tools exist for this purpose. One of the most used is "dex2jar".

This tool takes as input the APK file and generates a JAR file as output:

```
$ /vad/tools/dex2jar/d2j-dex2jar.sh cellfish.
                ironman3wp.apk
dex2jar cellfish.ironman3wp.apk -> cellfish.
                ironman3wp-dex2jar.jar
```

Now we only need to decompile the Java classes to get the source code. To do this, we can use the "JD-GUI" tool (Figure 3):

```
$ /vad/tools/jd-gui/jdgui cellfish.ironman3wp-
                dex2jar.jar
```

One of the first observations we draw from decompile the Java code in our example, is the fact that it has been used some code obfuscation tool that complicates the process of analyzing the application. The most common tools are "ProGuard" [3] and "DexGuard" [4].

Although these tools are commonly used to provide an additional layer of security and hinder the reverse engineering process, these applications can also be used in order to optimize the code and get a APK of a smaller size (eg, optimizing the bytecode eliminating unused instructions, renaming the class name, fields, and methods using short meaningless names, etc..).

In our example, we can deduce that the developers have used "ProGuard" (open source tool) because we can observe that some of the features offered by "DexGuard" are not been implemented in the analyzed code:

- The strings are not encrypted
- The code associated with logging functionality are not removed
- Does not exist encrypted files in the /assets resource
- There are no classes that have been entirety encrypted

Once we have access to source code, we can try to better understand how the application is built. "JD-GUI" allows us to save the entire application source code in a ZIP file, so you can perform new operations on this code using other tools. For example, to search for key terms on the entire code using the "grep" utility from the command line.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:versionCode="1"
        android:versionName="1.0"
        android:installLocation="0"
        package="cellfish.ironman3wp"
        >
        <uses-sdk
                android:minSdkVersion="9"
                android:targetSdkVersion="17"
                >
        </uses-sdk>
        <uses-feature
                android:name="android.software.live_wallpaper"
                >
        </uses-feature>
        <uses-feature
                android:name="android.hardware.touchscreen"
                android:required="false"
                >
        </uses-feature>
        <uses-feature
                android:glEsVersion="0x00020000"
                >
        </uses-feature>
        <uses-permission
                android:name="com.android.vending.BILLING"
                >
        </uses-permission>
        <uses-permission
                android:name="android.permission.INTERNET"
                >
        </uses-permission>
        <uses-permission
                android:name="android.permission.ACCESS_NETWORK_STATE"
                >
        </uses-permission>
        <uses-permission
                android:name="android.permission.VIBRATE"
                >
        </uses-permission>
AndroidManifest.xml.txt
```

Figure 2. *Contents of an AndroidManifest.xml File*

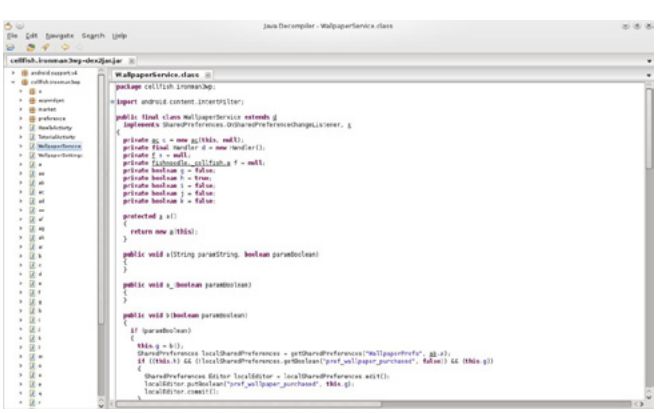

Figure 3. *Viewing the Source Code Decompiled with JD-GUI*

Although "JD-GUI" allows us to browse the entire hierarchy of objects in a comfortable manner, we generally find applications where there is a large number of Java classes to analyze, so we need to rely on other tools to facilitate the understanding of the code .

Following the aim that defined Chikofsky and Cross in reverse engineering, which is none other than that of understanding how the application is built, there is a tool that will help us greatly in this regard: "Understand".

According to the website itself, "Understand" is a static analysis tool for maintaining, measuring and analyzing critical or large code bases. Although is not purely a security tool (do not expect to use it as a vulnerability scanner), it helps us to understand the application code, which is our goal (Figure 4).

There are several online tools that have a similar purpose. For example, "Dexter" gives us detailed information about the application we want to analyze. As with any online service, our analysis is exposed to third party who can get to make use of our work, so we should always keep this in mind.

With the "Dexter" tool, is a simple as registering, create a project and upload the APK that we want to analyze. After the analysis, we can view information such as the following:

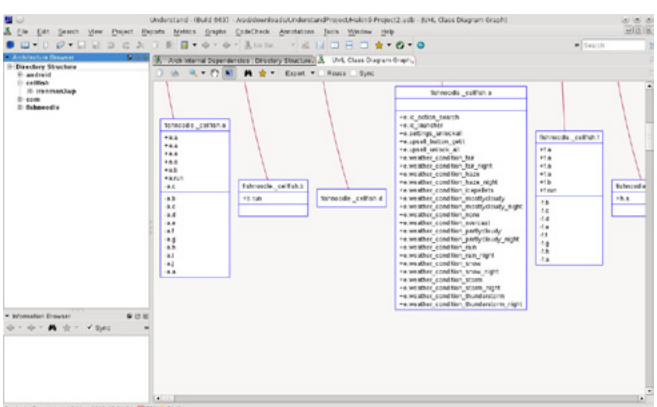

Figure 4. *Understand Showing the UML Class Diagram of the Application*

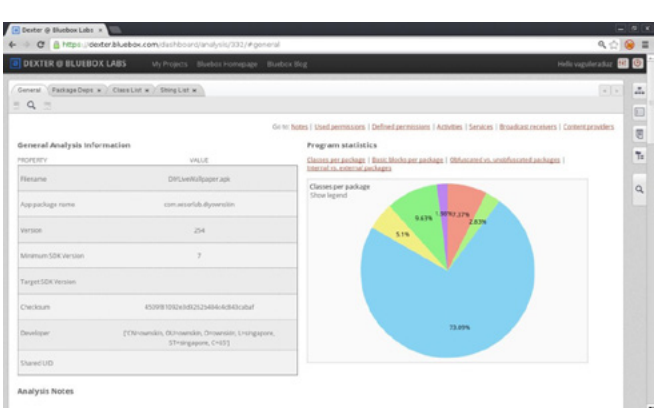

Figure 5. *Initial View of an Application Analysis with Dexter*

- Package dependency graph
- List of classes
- List of strings used by the application
- Defined permissions and used permissions
- Activities, Services, Broadcast Receivers, Content Providers
- Statistical data (percentage of obfuscated packages, use of internal versus external packages, classes per package, etc.).

Possibly, the power of this tool lies in its ease of use (all actions are performed through the browser) and navigating the class diagram and application objects (Figure 5).

Malware Identification in the Play Store

It's not a secret that Google's official store (the Play Store, which we have received an update in late April this year), hosts malware. Now, how do we identify those malicious applications? How do we know what they are really doing? Let us then how to answer these questions.

The techniques for introducing malware on a mobile application can be summarized in the following:

- Exploit any vulnerability in the web server hosting the official store (typically, for example, taking advantage of a XSS vulnerability)
- Enter malware in an application available at the official store (most users trust it and can be downloaded by a large number of potential users)
- Install not malicious applications that at some point installs malware (eg, include additional levels with malware into a widespread game)
- Use alternatives to official stores to post applications containing malware (usually, offering free applications that are not free in the official store)

When we talk about to introduce malware into an application, we can refer to two different scenarios:

- The published application contains code that exploits a vulnerability in the device, or
- The published application does not exploit any vulnerability, but contains code that can perform malicious actions and, therefore, the user is warned of the permissions required by the application as a step prior to installation.

In this article we focus on the second case: application with malicious code that exploits the user's trust.

How to Identify Malicious Applications on the Play Store?

A malicious application includes code that performs some action not expected by the user. For example, if a user downloads from the official store an application to change the wallpaper of his device, the user do not expect that this app can read his emails, can make phone calls or send SMS messages to premium accounts, for example.

A tool that allows us to quickly assess the existence of malicious code is "VirusTotal" [5]. For example, if we use the service offered by "VirusTotal" to analyze the APK of the "Wallpaper & Background Browser" application of the "Start-App" company, and available in the Play Store (*https://play.google.com/store/apps/details?id=com.startapp.wallpaper.browser*), we note that 12 of the 46 supported antivirus by this service, detect malicious code in the application. Exactly, the following:

- AhnLab-V3. Result: Android-PUP/Plankton
- AVG. Result: Android/Plankton
- Commtouch. Result: AndroidOS/Plankton.A.gen!Eldorado
- Comodo. Result: UnclassifiedMalware
- DrWeb. Result: Adware.Startapp.5.origin
- ESET-NOD32. Result: a variant of Android/Plankton.I
- F-Prot. Result: AndroidOS/Plankton.D
- F-Secure. Result: Application:Android/Counterclank
- Fortinet. Result: Android/Plankton.A!tr
- Sophos. Result: Andr/NewyearL-B
- TrendMicro-HouseCall. Result: TROJ_GEN.F47V0830
- VIPRE. Result: Trojan.AndroidOS.Generic.A (Figure 6)

Here's another example. If we search at the Play Store the "Cool Live Wallpaper" application (*https://play.google.com/store/apps/details?id=com.ownskin.diy_01zti0rso7rb*), developed by "Brankhox", we find the following information:

Package

```
com.ownskin.diy_01zti0rso7rb
```

Permissions

```
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.ACCESS_NETWORK_STATE
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.READ_SMS
android.permission.READ_CONTACTS
com.google.android.gm.permission.READ_GMAIL
android.permission.GET_ACCOUNTS
android.permission.ACCESS_WIFI_STATE
```

Potential malicious activities

- The application has the ability to read text messages (SMS or MMS)
- The application has the ability to read mail from Gmail
- The application has the ability to access user contacts

The questions we must ask is why and for what purpose the application need these permissions, like reading my email or access my contacts? It's really so intrusive as it sounds?

We will use some of the tools described above, to reverse engineer this application and see if it is using some of the more sensitive permissions that it requests.

Step 1: Get the APK file of the application

There are multiple ways to obtain an APK:

- Downloading an unofficial APK
  - Google: we can use the Google search engine to locate the APK.
  - Unofficial repositories: we can find the APK in several alternative markets [6] or other repositories like 4shared.com, apkboys.com, apkmania.co, aplicacionesapk.com, aptoide.com, flipkart.asia, etc.
- Downloading an official APK
  - Real APK Leecher [7]: This tool allows us to download the official APK for some applications.
  - SaveAPK [8]: This tool (required to have previously installed the „OI File Manager"

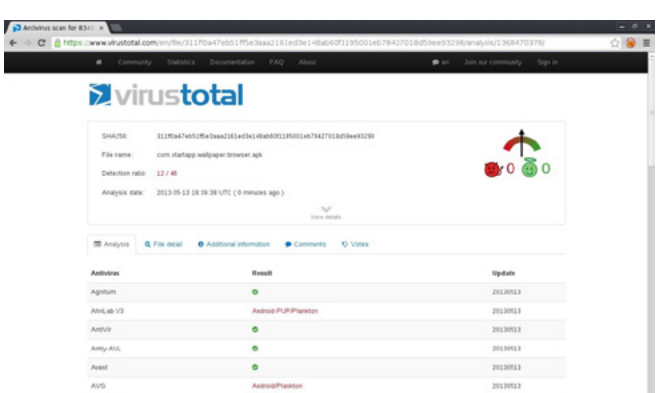

Figure 6. *Result of a VirusTotal Analysis on an APK*

application) available on the Play Store, lets us generate the APK if we have previously installed application on the device.

- Astro File Manager [9]: This tool is available in the Play Store, and we can get the APK if we have previously installed the application on the device. When performing a backup of the application, the APK is stored in the directory that is defined for backup.

Given the risk involved in dealing with malware, if we choose the option to download the APK existing in the Play Store from a previous installation of the application, we should use preferably an emulator [10] or a device of our test lab (Figure 7).

Figure 7. *Downloading an APK with APK Real Leecher*

Step 2: Convert the application from the Dalvik Executable format (.dex) to Java classes (.class)

The idea is to have the application code into a human-readable format. In this case, we use the "dex2jar" tool to convert the format Android to the Java format:

```
$ /vad/tools/d2j-dex2jar.sh com.ownskin.
                    diy_01zti0rso7rb.apk
dex2jar com.ownskin.diy_01zti0rso7rb.apk ->
   com.ownskin.diy_01zti0rso7rb-dex2jar.jar
```

Step 3: Decompile the Java code

Using a Java decompiler (like "JD-GUI"), we can obtain the Java source code from the .class files.

In our case, we will choose a fast track. "JD-GUI" allows us to save the entire application source code in a ZIP file. We'll keep this file as "com.ownskin.diy_01zti0rso7rb-dex2jar.src.zip", and unzip it to perform a manual scan.

We note that there are 353 Java source files:

```
$ find /vad/lab/Android/com.ownskin.diy_01zti0r
                    so7rb-dex2jar.src/ -type f | wc -l
353
```

Step 4: Find malicious code in the application

We can now search in any resource of the application to identify strings that may be susceptible of being used for malicious purposes. For example, we have previously identified that this application sought permission to read SMS messages.

Listing 1. *Finding Malicious Code in the Application*

```
$ cd /vad/lab/Android/com.ownskin.diy_01zti0rso7rb-dex2jar.src/
$ grep -i sms -r *
com/ownskin/diy_01zti0rso7rb/ht.java:import android.telephony.SmsMessage;
com/ownskin/diy_01zti0rso7rb/ht.java:    SmsMessage[] arrayOfSmsMessage = new
                SmsMessage[arrayOfObject.length];
com/ownskin/diy_01zti0rso7rb/ht.java:      arrayOfSmsMessage[0] = SmsMessage.createFromPdu((byte[])
                arrayOfObject[0]);
com/ownskin/diy_01zti0rso7rb/ht.java:      hs.a(this.a, arrayOfSmsMessage[0].getOriginatingAd-
                dress());
com/ownskin/diy_01zti0rso7rb/ht.java:      hs.c(this.a, arrayOfSmsMessage[0].getMessageBody());
com/ownskin/diy_01zti0rso7rb/hm.java:      if (!"SMS_MMS".equalsIgnoreCase(this.U))
com/ownskin/diy_01zti0rso7rb/hm.java:      a(Uri.parse("content://sms"));
com/ownskin/diy_01zti0rso7rb/hs.java:    Uri localUri = Uri.parse("content://sms");
com/ownskin/diy_01zti0rso7rb/hs.java:    this.P.l().registerReceiver(this.ac, new
                IntentFilter("android.provider.Telephony.SMS_RECEIVED"));
```

Let's see if the application actually use this permission (Listing 1).

Using the "grep" command, we identified that the following resources (Java classes) seem to contain some code that allows read access to the user's SMS:

- com/ownskin/diy_01zti0rso7rb/hm.java
- com/ownskin/diy_01zti0rso7rb/hs.java
- com/ownskin/diy_01zti0rso7rb/ht.java

Let's see the source code detail of these resources in JD-GUI:

- com/ownskin/diy_01zti0rso7rb/hm.java

```
…
if (!"SMS_MMS".equalsIgnoreCase(this.U))
        break label89;
       a(Uri.parse("content://sms"));
       a(Uri.parse("content://mms"));
…
```

- com/ownskin/diy_01zti0rso7rb/hs.java
  It creates a „localUri" object of the "Uri" class, calling the "parse" method to be used in the query to the Content Provider that allows to access to the SMS inbox:

```
…
public static final Uri a = localUri;
public static final Uri b = Uri.
                  withAppendedPath(localUri, "inbox");
…
```

```
static
  {
    Uri localUri = Uri.parse("content://sms");
  }
```

and registers a Receiver to be notified of the received SMS:

```
…this.P.l().registerReceiver(this.ac,new
                  IntentFilter("android.provider.
                  Telephony.SMS_RECEIVED"));
```

…• com/ownskin/diy_01zti0rso7rb/ht.java
This class implements a Broadcast Receiver. This is simply an Android component that allows the registered Receiver to be notified of events produced in the system or in the application itself.

In this case, the implemented Receiver is capable of receiving input SMS messages. And this notification occurs before that the internal SMS management application receive the SMS messages. This scenario is used by some malware, for example, to perform some action and then delete the received message before it is processed by the messaging application and be detected by the user.

In this example, when the user receives an SMS, the application identify its source and read the message, as shown in the following code: Listing 2.

As we can see (at this point, we can complete the process of analysis of the application by a dynamic analysis of it), in fact, the application accesses our SMS messages. However, it's im-

Listing 2. *When the User Receives an SMS, the Application Identify its Source and Read the Message*

```
…
public final void onReceive(Context paramContext, Intent paramIntent)
  {
    Object[] arrayOfObject = (Object[])paramIntent.getExtras().get("pdus");
    SmsMessage[] arrayOfSmsMessage = new SmsMessage[arrayOfObject.length];
    if (arrayOfObject.length > 0)
    {
      arrayOfSmsMessage[0] = SmsMessage.createFromPdu((byte[])arrayOfObject[0]);
      hs.a(this.a, arrayOfSmsMessage[0].getOriginatingAddress());
      hs.b(this.a, go.a(this.a.P.l(), hs.a(this.a)));
      if ((hs.b(this.a) == null) || (hs.b(this.a).length() == 0))
        hs.b(this.a, hs.a(this.a));
      hs.c(this.a, arrayOfSmsMessage[0].getMessageBody());
      hs.c(this.a);
    }
  }
…
```

Table 1. *Static Analysis Tools for Android Applications*

TOOL	DESCRIPTION	URL
Dexter	Static android application analysis tool	https://dexter.bluebox.com/
Androguard	Analysis tool (.dex, .apk, .xml, .arsc)	https://code.google.com/p/androguard/
smali/baksmali	Assembler/disassembler (dex format)	https://code.google.com/p/smali/
apktool	Decode/rebuild resources	https://code.google.com/p/android-apktool/
JD-GUI	Java decompiler	http://java.decompiler.free.fr/?q=jdgui
Dedexer	Disassembler tool for DEX files	http://dedexer.sourceforge.net/
AXMLPrinter2.jar	Prints XML document from binary XML	http://code.google.com/p/android4me/
dex2jar	Analysis tool (.dex and .class files)	https://code.google.com/p/dex2jar/
apkinspector	Analysis functions	https://code.google.com/p/apkinspector/
Understand	Source code analysis and metrics	http://www.scitools.com/
Agnitio	Security code review	http://sourceforge.net/projects/agnitiotool/

References

[1] "Reverse Engineering and Design Recovery: A Taxonomy". Elliot J. Chikofsky, James H. Cross. *http://win.ua.ac.be/~lore/Research/Chikofsky1990-Taxonomy.pdf*
[2] "Security features provided by Android" *http://developer.android.com/guide/topics/security/permissions.html*
[3] ProGuard Tool *http://developer.android.com/tools/help/proguard.html*
[4] DexGuard Tool *http://www.saikoa.com/dexguard*
[5] VirusTotal *http://www.virustotal.com*
[7] Alternative markets to the Play Store *http://alternativeto.net/software/android-market/*
[8] Real APK Leecher *https://code.google.com/p/real-apk-leecher/*
[9] SaveAPK *https://play.google.com/store/apps/details?id=org.mariotaku.saveapk&hl=en*
[10] Astro File Manager *https://play.google.com/store/apps/details?id=com.metago.astro&hl=en*
[11] "Using the Android Emulator" *http://developer.android.com/tools/devices/emulator.html*

portant to recall that we have accepted that the application can perform these actions, because we have accepted the permissions required and the application has informed to us of this situation prior to installation.

Similarly, we can verify as any application makes use of the various permits requested, with particular attention to those that may affect our privacy or which may result in a cost to us.

Some people sees no malware in this type of applications that take advantage of user trust, and has been the subject of controversy on more than one occasion. In any case, Google has decided to remove applications from the Play Store that can make an abuse of permits that these require to be confirmed by users who wish to use them. That does not mean, on the other hand, that there still exist such applications in Google's official store (Table 1).

VICENTE AGUILERA DIAZ

With over 10 years of professional experience in the security sector, Vicente Aguilera Diaz is co-founder of Internet Security Auditors (a Spanish firm specializing in security services), OWASP Spain Chapter Leader, member of the Technical Advisory Board of the RedSeguridad magazine, and member of the Jury of the IT Security Awards organized by the RedSeguridad magazine.

Vicente has collaborate in several open-source projects, is a regular speaker at industry conferences and has published several articles and vulnerabilities in specialized media. Vicente has the following certifications: CISA, CISSP, CSSLP, PCI ASV, ITIL Foundation, CEH|I, ECSP|I, OPSA and OPST.