EVOLUTION OF ANDROID EXPLOITS FROM A STATIC ANALYSIS TOOLS PERSPECTIVE

Anna Szalay & Jagadeesh Chandraiah
Sophos, UK

Email {anna.szalay, jagadeesh.chandraiah}@sophos.com

ABSTRACT

With *Android* being the fastest-growing mobile OS, and with a rapidly increasing number of *Android* malware samples, it is important to acknowledge the risk of exploitation of security vulnerabilities by malware.

According to Common Vulnerabilities and Exposures (CVE) data, over the past few years the total number of documented *Android* vulnerabilities has reached 36, with seven of them discovered in the last year. The most serious of the recent ones is the so-called 'Master Key' vulnerability (CVE-2013-4787), which is reported to have affected 99% of devices, compromising the APK signature validation process.

With the total number of *Android* samples in our database exceeding 900,000, and 2,000 new *Android* malware samples appearing every day, we estimate that approximately 10% of the samples exploit some vulnerability, of which one tenth will be a 'Master Key' exploit.

In this paper, we will investigate *Android* malware that has attempted to exploit vulnerabilities, and identify the most relevant threat families from the perspective of static analysis tools. The research will reveal the evolution of the threat families. Additionally, we will provide an evaluation of the various analysis tools that are currently available, exploring their successes and failures, and highlighting the differences between them. These results will be used to identify the best approach for future analysis, to ensure it keeps up with the rapid development of *Android* malware, and the increasing sophistication of device exploitation.

1. INTRODUCTION

According to *Gartner* [1], collectively, 2.5 billion devices are expected to be shipped in 2014, with *Android* expected to be loaded on more than one billion of them (see Figure 1).

The continuously growing popularity of *Android* devices, as well as the specifics of the *Android* environment, its open source nature, the non-complicated process of adding applications to *Google Play* and other *Android* application markets, forums and file-sharing sites, plus the possibility of redistribution in the form of 'cracked' and repackaged apps, all make it the number one target for mobile malware writers, with the number of malware samples in our database reaching 900,000 this year (see Figure 2).

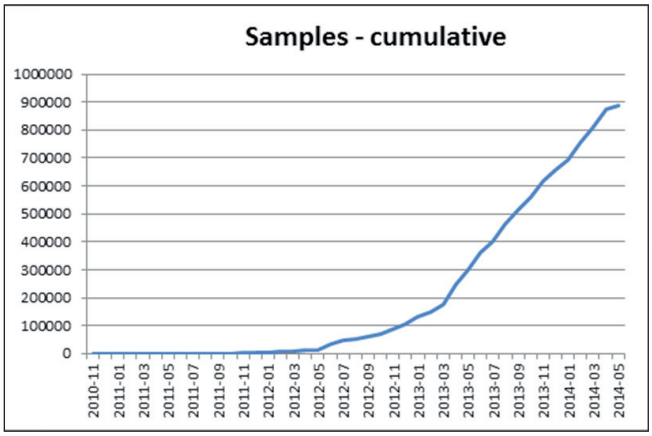

Figure 2: Android malware samples timeline.

It also means that exploitation of security vulnerabilities is inevitable, and it is important to acknowledge the risk of exploitation of vulnerabilities by malware.

The cumulative share of malware samples exploiting different vulnerabilities has reached 10% (Figure 3).

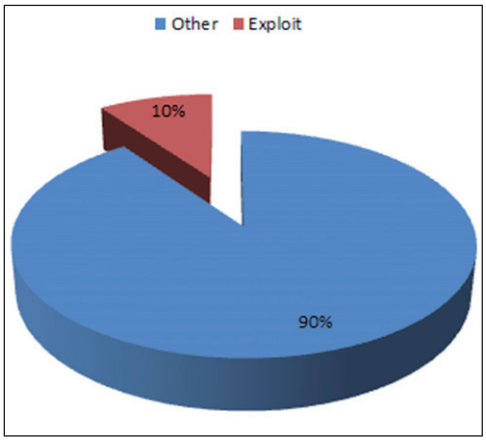

Figure 3: Share of malware exploit samples.

But when we consider the significance of the prevalent malware families that have surfaced over years, there are several

Operating System	2013	2014	2015
Android	879,821	1,170,952	1,358,265
Windows	325,127	339,068	379,299
iOS/Mac OS	241,416	286,436	324,470
Others	873,194	683,519	565,186
Total	**2,319,559**	**2,479,976**	**2,627,221**

Figure 1: Worldwide device shipment by operating system (thousands of units).

well-known families that have used an exploit(s). In this paper, we will aim to identify those families based on our collection of *Android* malware samples.

We will concentrate on *Android* exploits and their evolution by evaluating malware families using static analysis tools. We will look at the important families that have used exploits to obtain access to and gain control of infected devices, limiting our research to the families that could be considered as the most well known for exploiting one vulnerability or more.

We will evaluate popular analysis tools by running them against malware families containing the exploits and analysing the results. We will look at how successful they are, highlight their failures, and document our case studies in order to identify the best approach for similar cases in the future.

Our aim is to show how the development of *Android* malware and the increasing sophistication of *Android* exploits has made popular static analysis tools unreliable.

We will argue that the evolution of exploits needs to be matched by the development of analysis tools, and highlight the need to develop more robust static and dynamic tools in the future.

2. EXPLOITS OVERVIEW

Based on Common Vulnerabilities and Exposures (CVE) data [2], the total number of documented *Android* vulnerabilities has reached 36, with seven of them discovered in the last year, and six in the first quarter of 2014.

Since it began in 2010, the evolution of *Android* malware can be matched with the exploitation of *Android* vulnerabilities. Based on our samples, we will take a closer look at those that we think are the most significant milestones.

Android vulnerabilities exploitation timeline:

- **Q4 2010:** *Android WebKit* browser exploit
- **Q4 2010:** *Android* data-stealing vulnerability
- **Q1 2011:** *Android* local root exploit, a.k.a .'Rage against the cage' or 'Lotoor' exploit
- **Q2 2011:** *Android* ClientLogin protocol vulnerability
- **Q3 2011:** *Android* Gingerbreak root exploit, a.k.a. CVE-2011-1823
- **2012 …**
- **Q2 2013:** *Dex2jar* exploitation
- **Q2 2013:** 'Master Key' vulnerability
- **Q3 2013:** 'Extra Field' vulnerability

3. EXPLOITS EVOLUTION

3.1 Android WebKit browser exploit

The first serious proof-of-concept exploit for the *Android* platform was related to the *Android WebKit* browser and, while it was not related *only* to *Android* OS (and thus was not a platform vulnerability), it allowed remote attackers to execute arbitrary code or cause a denial of service (application crash) via a crafted HTML document, related to non-standard NaN representation. In other words, it was making *Android* devices vulnerable to drive-by exploits [3].

It has been reported that the targeted vulnerability was fixed by *Google* in the following *Android* release (*2.2 Froyo*). According to *Google*, *Froyo* was used by 36% of all *Android* devices at the time – which meant that the majority of devices could still successfully be attacked using the exploit [4].

3.2 Android data-stealing vulnerability

Next on our list is an *Android* data-stealing vulnerability [5], a general vulnerability in *Android* which allowed a malicious website to obtain the contents of any file stored on the device's SD card. It would also be possible to retrieve a limited range of other data and files stored on the phone using this vulnerability. This is a simple exploit involving JavaScript and redirects, meaning it should also work on multiple handsets and multiple *Android* versions without any effort. One limiting factor of this exploit is that you have to know the name and path of the file you want to steal. However, there are a number of applications that store data with consistent names on the SD card, and pictures taken on the camera are stored with a consistent naming convention too, so it is not hard to guess the correct names and paths. This is not a root exploit either, meaning it runs within the *Android* sandbox and cannot reach all files on the system, only those on the SD card and a limited number of others.

3.3 Android local root exploit, a.k.a. 'rage against the cage' or 'Lotoor' exploit

At the beginning of 2011, the *Android* root was attacked by exploiting privilege escalation. Both exploits for the *Linux* kernel udev vulnerability and an adb privilege escalation attack are relatively old, but they worked with the versions of *Android* used by the majority of *Android* phones.

Note that we are still seeing samples with different variations of this exploit. The cumulative share of malware samples that use variations of this exploit has reached 2% of the total number of samples, as shown in Figure 4.

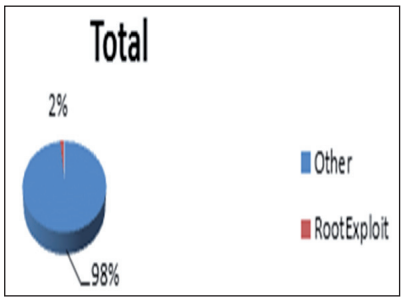

Figure 4: Malware root exploit samples share.

When looking at the different pieces of malware that have taken advantage of this vulnerability, up to 14% is made up of variants of the so-called 'classic' *Android* local root exploit samples based on using an ELF executable that comprises exploit code (see Figure 5).

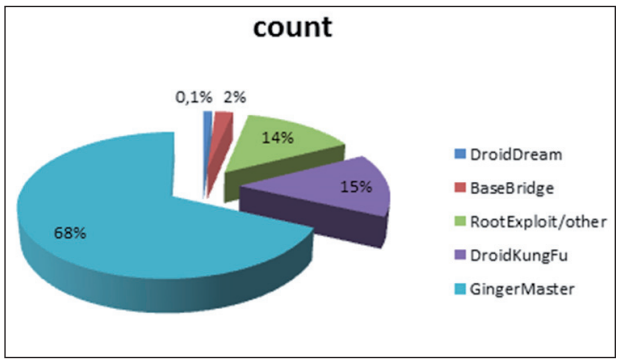

Figure 5: Share of root-exploit-based malware samples.

DroidKungFu is a well-known malware family that has taken root exploitation to a different level – it alone accounts for 15% of root-exploit-based malware samples.

DroidKungFu was included in repackaged apps that were made available through a number of alternative app markets and forums targeting Chinese-speaking users. This malware would add into the infected app a new service and a new receiver. The receiver would be notified when the system finished booting so that it could automatically launch the service without user interaction. DroidKungFu had encrypted udev and 'rage against the cage' exploits, and decrypted them upon running, executing and launching the attack.

Reports suggest that *Gingerbread* (*Android 2.3*) was the only *Android* version that was not vulnerable at the time, which would mean that 99% of *Android* phones were potentially affected.

Another 2% of the root exploits pie was occupied by the so-called BaseBridge family. This malware used a privilege escalation exploit to elevate its privileges and install additional malicious apps onto an *Android* device. It used HTTP to communicate with a central server and leaked potentially identifiable information. These malicious apps could send and read SMS messages, potentially costing the user money. In fact, it could even scan incoming SMS messages and automatically remove warnings that alert the user to the fact that they are being charged a fee for using premium rate services.

Also, insignificant in share, but a piece of malware that attracted public attention at the time, is DroidDream. It surfaced in spring-summer 2011 and represents the first *Android* botnet to take advantage of a root exploit. DroidDream became an '*Android Market* nightmare' when over 50 infected apps were identified and removed from the market [6].

3.4 Android ClientLogin protocol vulnerability

The *Android* ClientLogin protocol vulnerability has the highest score based on CVE details, and was discovered by German researchers in May 2011 [7]. It allowed remote attackers to gain privileges and access private data by interfering with the transmitting of an authentication token (authToken), meaning that it could potentially allow Wi-Fi traffic to be sniffed, and the authToken that had just been generated to be stolen. It has been reported that 99% of *Android* devices were at risk from this vulnerability at the time [8].

3.5 Android Gingerbreak root exploit

Another milestone in the evolution of exploiting vulnerabilities was claimed by GingerMaster, the first *Android* malware to use a root exploit on *Android 2.3* (*Gingerbread*), CVE-2011-1823 [7]. It takes up 68% of the root exploits pie!

The GingerMaster malware was repackaged into popular legitimate apps in order to attract user downloads and installation. Within the repackaged apps, the malware registered a receiver so that it would be notified when the system finished booting. Inside the receiver, it would silently launch a service in the background. Accordingly, the

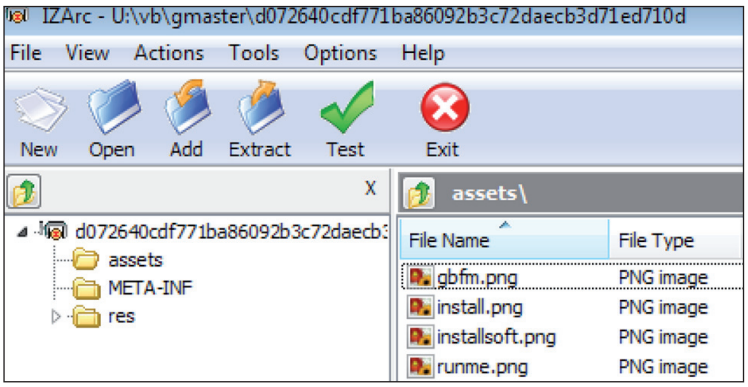

Figure 6: Inside GMaster APK, exploit code in a picture file.

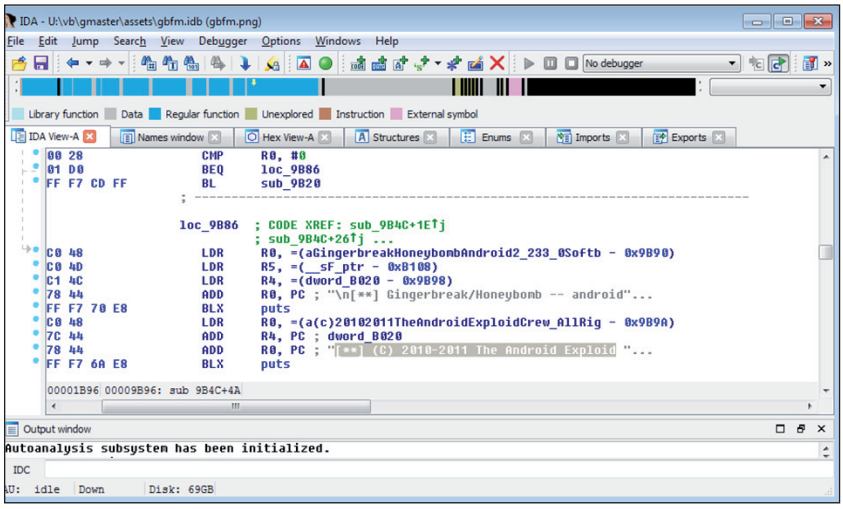

Figure 7: Inside GMaster APK, exploit code in a picture file.

background service would collect various pieces of information including the device ID, phone number and others (e.g. by reading /proc/cpuinfo) and then upload the information to a remote server. The actual exploit was packaged into the infected app in the form of a regular file named 'gbfm.png', which could be deciphered as 'Ginger Break for Me' (see Figures 6 and 7).

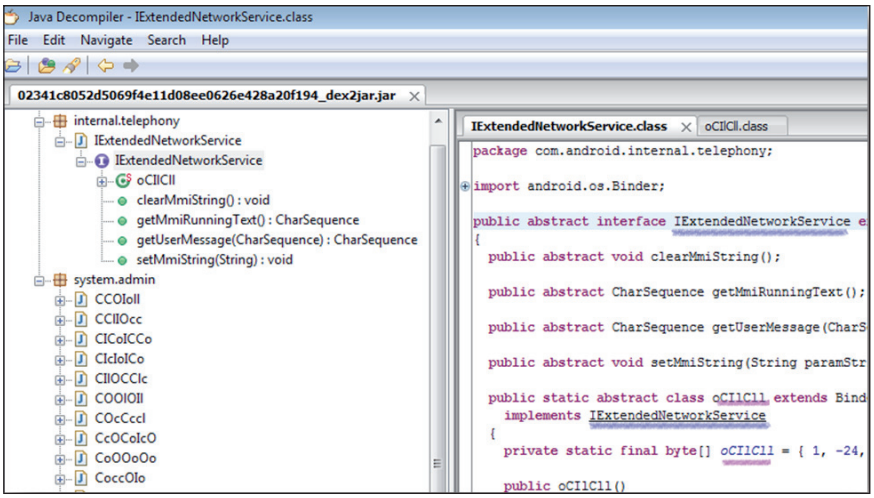

Figure 8: Obad obfuscation, example of decompiled by dex2jar dex code output.

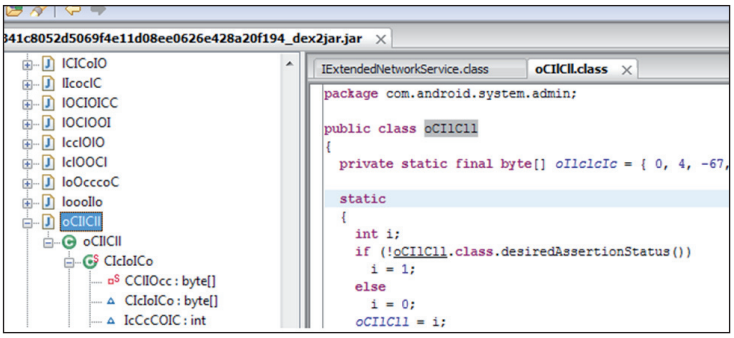

Figure 9: Obad obfuscation, example of decompiled by dex2jar dex code output.

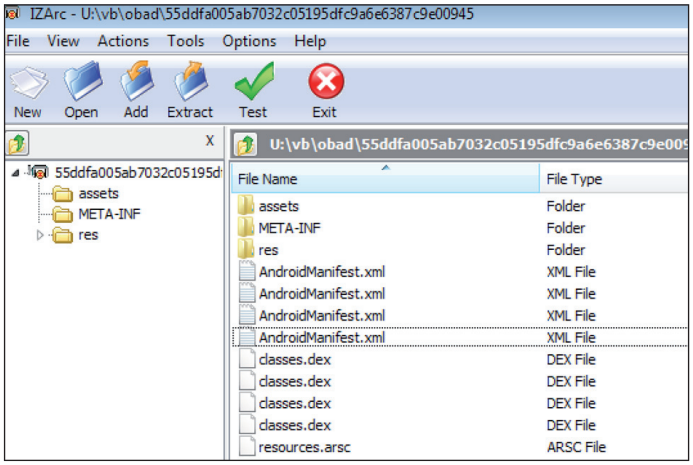

Figure 10: Inside 'Master Key' malware, example of multiple files.

3.6 Dex2jar exploitation: Obad

Labelled as one of the worst trojans ever, Obad hit the headlines in June 2013. It combined a few unknown exploits and made analysis incredibly difficult. Obad exploited the way in which the OS was processing the AndroidManifest.xml file (which is generated during the build process, and contains information about the application structure, including how different components are related and launched, as well as what permissions an application requests). Obad's authors also found a way to silently extend Device Administrator privileges so that the malware would not appear on the list of the applications running with these privileges. In addition, the malware authors introduced complex code encryption, with all external methods called via reflection, and all strings encrypted, including the names of classes and methods. Each class would have a local descriptor method which would obtain the string required for encryption from the locally updated byte array. See Figures 8 and 9.

From our point of view, however, Obad was most famous for finding an error in the *dex2jar* software. *Dex2jar* is one of the most popular and well-used static analysis tools. The disruption of the conversion of Davlik bytecode into Java bytecode by finding an error in the software was quite significant as it made static analysis extremely difficult.

3.7 'Master Key' vulnerability

In July 2013, it was reported that an *Android* APK signing had been compromised. The error was found in the way cryptographic signatures for applications were handled – this allowed attackers to execute arbitrary code via an application package file (APK) that is modified in a way that does not violate the cryptographic signature: CVE -2013-4787, a.k.a. *Android* security bug 8219321 and the 'Master Key' vulnerability [2]. In short, the application update validation process was compromised: it was found possible to repackage a legitimate application while inserting malicious code in the form of duplicates of the original AndroidManifest and classes.dex files, thus providing a way for a legitimate app to be updated with a malicious version.

An interesting addition to this vulnerability is that the simple unpacking of an APK using any Zip-based unpacker

will overwrite multiple files unless special settings are applied, thus complicating static analysis. See Figure 10.

3.8 'Extra Field' vulnerability

Following in the steps of 'Master Key', just a couple of weeks later a similar vulnerability that would allow the bypassing of code verification was found by Chinese researchers [9]. It was based on the way in which an APK file is verified as an archive and used an object extra field, hence the name 'Extra Field' vulnerability. It did not reach the same sample numbers as the 'Master Key' vulnerability, but was reported to have comparable possible implications due to the fact that it exploited fundamentals of the Android APK, which is a ZIP archive with some special object fields. The flaw was based on a signed-unsigned integer mismatch and, as some researchers have pointed out, relevant code testing could have prevented it [10]. Recognition of the exploited APK is based on identifying an object in a ZIP archive with the changed extra field length followed by the filename 'classes.dex', where the extra field length is FDFF, i.e. 65,533 (unsigned) or -3 (signed), as shown in Figure 11.

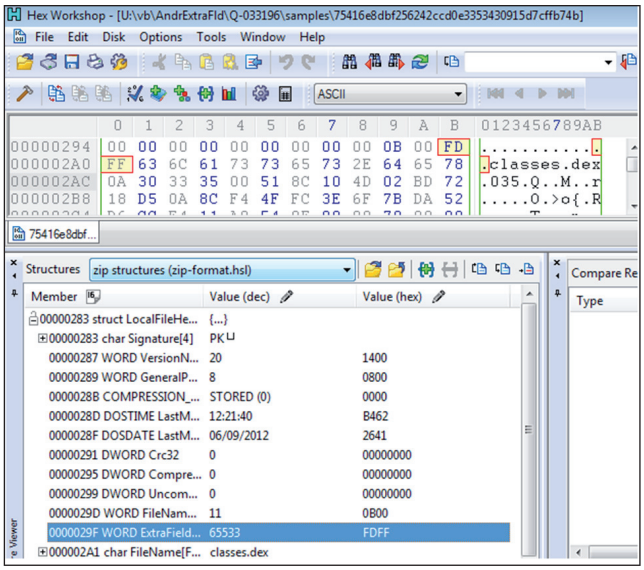

Figure 11: Example of malware exploiting the 'Extra Field' vulnerability, showing a changed field.

The fact that it was treated as a signed integer during the classes.dex checksum verification forced a verifier to step back exactly three bytes and read bytes starting from the 'dex' characters and following the contents of the extra field. However, on loading an APK, FDFF is treated as an unsigned integer, which causes the loader to go forward for the length of the extra field, i.e. 65,533 bytes. This could result in the loading of the malicious code that has been inserted into the hacked APK.

Google responded with a fix labelled 'Values in ZIP are unsigned' [11]. Both the 'Master Key' and 'Extra Field' vulnerabilities were based on compromising an *Android* app installer package (APK).

4. STATIC ANALYSIS TECHNIQUES AND TOOLS

In this section we will provide an overview of the static analysis techniques and tools used for analysing *Android* malware. There are different approaches for static analysis according to one's knowledge and tools arsenal, but here we will discuss the most widely used and straightforward technique. We can broadly classify the static analysis steps as follows:

- **Unarchiving** – the APK file is in ZIP format, to verify its contents we need to extract them using tools like *UnZip*.
- **Decoding** – AndroidManifest.xml and Resources.arsc are decoded using tools like *apktool* and *Androguard*.
- **Decompiling .dex** – *dex2jar* and *jd-gui* are used for converting .dex files to .jar format, and then decompiling to Java code respectively.
- **Disassembling .dex** – *smali/baksmali* and *IDA Pro* can be used to disassemble the .dex files.

5. EVALUATION OF STATIC ANALYSIS TOOLS AGAINST EXPLOIT SAMPLES

In this section, static analysis tools will be evaluated against popular exploit samples to highlight the challenges faced when analysing such samples. We won't explain the whole of each exploit, as we have already discussed the details earlier in the paper. Instead, we will discuss only the part that is relevant to the tools analysis. Although we have evaluated tools in the context of exploit samples, most of the evaluation will also be applicable to malware samples that don't use any exploits. The methodology used was to run the samples as an analyst would to verify for analysis, and all the samples that weren't successful in executing, or that posed a challenge for analysis, are discussed below.

The set of samples evaluated were:

Andr/MstrKey

Andr/DroidD (DroidDream)

Andr/DroidRt

Andr/Obad

Andr/Kongfu (DroidKungFu)

Andr/Gmaster (GinMaster)

5.1 Andr/MstrKey-A – challenge of multiple entries of the same file

Andr/MstrKey-A SHA1 – (APK)
78adcaa663d0f33ca014080870ff7a7e27461086

The Master Key vulnerability works by having multiple entries of the same filename [12]. As shown in Figure 12, there are multiple classes.dex, AndroidManifest.xml and icon.png files.

Because of the duplicate entries with same name, we should pay extra attention when using analysis tools to achieve what we want.

As the APK file is in ZIP format, the first thing we should do is to extract the APK to verify the classes.dex and

```
Archive:  78adcaa663d0f33ca014080870ff7a7e27461086
  Length      Date    Time    Name
---------  ---------- -----   ----
  2320255  21/07/2013 05:31   assets/fashion.jxp
      640  21/07/2013 05:31   res/layout/main.xml
     6780  21/07/2013 05:31   AndroidManifest.xml
     1404  21/07/2013 05:31   AndroidManifest.xml
     1476  21/07/2013 05:31   resources.arsc
     7542  21/07/2013 05:31   res/drawable-hdpi/icon.png
     2760  21/07/2013 05:31   res/drawable-ldpi/icon.png
     4278  21/07/2013 05:31   res/drawable-mdpi/icon.png
    90080  21/07/2013 05:31   classes.dex
     4908  21/07/2013 05:31   classes.dex
      635  21/07/2013 05:31   META-INF/MANIFEST.MF
      688  21/07/2013 05:31   META-INF/CERT.SF
     1007  21/07/2013 05:31   META-INF/CERT.RSA
---------                     -------
  2442453                     13 files
```

Figure 12: Andr/MstrKey-A APK file showing multiple files with the same name.

```
Archive:  78adcaa663d0f33ca014080870ff7a7e27461086
  inflating: assets/fashion.jxp
  inflating: res/layout/main.xml
  inflating: AndroidManifest.xml
replace AndroidManifest.xml? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

Figure 13: UnZip prompting to replace the existing file.

```
AndroidManifest.xml
apktool.yml

smali
└── com
    └── junction
        ├── fire
        │   └── AssetExtract.smali
        └── lexin
            └── exppack
                └── fashion
                    ├── R.smali
                    ├── R$string.smali
                    ├── GetPackage.smali
                    ├── R$layout.smali
                    ├── R$attr.smali
                    └── R$drawable.smali

assets
└── fashion.jxp

original
├── AndroidManifest.xml
└── META-INF
    ├── MANIFEST.MF
    ├── CERT.SF
    └── CERT.RSA

res
├── values
│   ├── public.xml
│   └── strings.xml
├── layout
│   └── main.xml
├── drawable-ldpi
│   └── icon.png
├── values-zh-rTW
│   └── strings.xml
├── drawable-mdpi
│   └── icon.png
├── values-zh-rCN
│   └── strings.xml
└── drawable-hdpi
    └── icon.png
```

Figure 14: Files after the 'apktool d' command.

AndroidManifest.xml files. When you try to extract the archive contents, if you are not aware of the exploit, it is possible to unintentionally overwrite the files with similar names, as shown in Figure 13.

The exploited APK file used for testing has two AndroidManifest files and two classes.dex files. *Apktool* is the tool most commonly used to decode AndroidManifest.xml files and generate smali code from classes.dex. In this case, *apktool* (v1. 5.2) decodes just one AndroidManifest file and one classes.dex file, but the dodgy permissions and code are in the other set of files. In order to decode these files, you have to fix the APK manually to decode the right files, or else decode them yourself manually. Even though this tool is designed to decode only one set of files, it would be useful for it to be able to decode more than one set in future.

5.2 Andr/DroidD – Dex header issue

SHA (dex) – b5b41c7c75182ced4121d01a6328f626aaf5a997

We came across an Andr/DroidD-Gen sample which *IDA Pro* showed as corrupt (Figure 15). When we investigated, we found that the classes.dex file in question had a dex036 header (Figure 16).

Figure 15: IDA 6.4adv corrupt error message.

```
00000000 6465 780A 3033 3600 1200 B06C 81A2 00EE dex.036....l....
00000010 2928 474B DB29 5212 E490 F298 3832 FEB0 )(GK.)R.....82..
00000020 1892 0000 7000 0000 7856 3412 0000 0000 ....p...xV4.....
00000030 0000 0000 4891 0000 6E02 0000 7000 0000 ....H...n...p...
00000040 5500 0000 280A 0000 5500 0000 7C0B 0000 U...(...U...|...
00000050 5E00 0000 780F 0000 ED00 0000 6812 0000 ^...x.......h...
00000060 1700 0000 D019 0000 6875 0000 B01C 0000 ........hu......
00000070 6270 0000 A77C 0000 7073 0000 2362 0000 bp...|..ps..#b..
00000080 DE7E 0000 1955 0000 F46F 0000 6066 0000 .~...U...o..`f..
00000090 CE60 0000 2E55 0000 AD64 0000 0C56 0000 .`...U...d...V..
000000A0 5455 0000 F77E 0000 E964 0000 2865 0000 TU...~...d..(e..
000000B0 9E67 0000 4265 0000 4A65 0000 5765 0000 .g..Be..Je..We..
000000C0 D780 0000 7481 0000 3681 0000 D77B 0000 ....t...6....{..
000000D0 7259 0000 0E62 0000 5B56 0000 E155 0000 rY...b..[V...U..
```

Figure 16: Dex file with dex036 header.

Offset	Size	Description
0x0	8	'Magic' value: "dex\n009\0"
0x8	4	Checksum
0xC	20	SHA-1 Signature
0x20	4	Length of file in bytes
0x24	4	Length of header in bytes (currently always 0x5C)
0x28	8	Padding (reserved for future use?)

Figure 17: Dex header format to offset 0x28.

According to the dex file format [13], the first eight bytes of the dex file are DEX_FILE_MAGIC (Figure 17), with

```
dex2jar classes.dex -> classes_dex2jar.jar
com.googlecode.dex2jar.DexException: while accept method:[Lcom/a;.a()Ljava/lang/String;]
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReader.java:694)
        at com.googlecode.dex2jar.reader.DexFileReader.acceptClass(DexFileReader.java:441)
        at com.googlecode.dex2jar.reader.DexFileReader.accept(DexFileReader.java:323)
        at com.googlecode.dex2jar.v3.Dex2jar.doTranslate(Dex2jar.java:85)
        at com.googlecode.dex2jar.v3.Dex2jar.to(Dex2jar.java:261)
        at com.googlecode.dex2jar.v3.Dex2jar.to(Dex2jar.java:252)
        at com.googlecode.dex2jar.tools.Dex2jarCmd.doCommandLine(Dex2jarCmd.java:110)
        at com.googlecode.dex2jar.tools.BaseCmd.doMain(BaseCmd.java:174)
        at com.googlecode.dex2jar.tools.Dex2jarCmd.main(Dex2jarCmd.java:34)
Caused by: com.googlecode.dex2jar.DexException: while accept code in method:[Lcom/a;.a()Ljava/lang/String;]
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReader.java:684)
        ... 8 more
Caused by: java.lang.RuntimeException: opcode format for 64 not found!
        at com.googlecode.dex2jar.reader.OpcodeFormat.get(OpcodeFormat.java:362)
        at com.googlecode.dex2jar.reader.DexCodeReader.findLabels(DexCodeReader.java:88)
        at com.googlecode.dex2jar.reader.DexCodeReader.accept(DexCodeReader.java:332)
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReader.java:682)
        ... 8 more
```

Figure 18: Error on dex2jar decompilation of file.

```
unknown opcode encountered - 40. Treating as nop.


UNEXPECTED TOP-LEVEL EXCEPTION:
org.jf.dexlib.Util.ExceptionWithContext: Index: 22789, Size: 269
        at org.jf.dexlib.Util.ExceptionWithContext.withContext(ExceptionWithContext.java:54)
        at org.jf.dexlib.IndexedSection.getItemByIndex(IndexedSection.java:77)
        at org.jf.dexlib.Code.InstructionWithReference.lookupReferencedItem(InstructionWithReference.java:79)
        at org.jf.dexlib.Code.InstructionWithReference.<init>(InstructionWithReference.java:57)
        at org.jf.dexlib.Code.Format.Instruction22c.<init>(Instruction22c.java:59)
        at org.jf.dexlib.Code.Format.Instruction22c.<init>(Instruction22c.java:40)
        at org.jf.dexlib.Code.Format.Instruction22c$Factory.makeInstruction(Instruction22c.java:103)
        at org.jf.dexlib.Code.InstructionIterator.IterateInstructions(InstructionIterator.java:82)
        at org.jf.dexlib.CodeItem.readItem(CodeItem.java:154)
        at org.jf.dexlib.Item.readFrom(Item.java:77)
        at org.jf.dexlib.OffsettedSection.readItems(OffsettedSection.java:48)
        at org.jf.dexlib.Section.readFrom(Section.java:143)
        at org.jf.dexlib.DexFile.<init>(DexFile.java:431)
        at org.jf.baksmali.main.main(main.java:280)
Caused by: java.lang.IndexOutOfBoundsException: Index: 22789, Size: 269
        at java.util.ArrayList.rangeCheck(ArrayList.java:635)
        at java.util.ArrayList.get(ArrayList.java:411)
        at org.jf.dexlib.IndexedSection.getItemByIndex(IndexedSection.java:75)
        ... 12 more
Error occured while retrieving the field_id_item item at index 22789
Error occured at code address 0
```

Figure 19: Error on baksmali 1.4.1, but works with baksmali 2.03.

Figure 20: Decompiled Obad file with error.

'dex\n<2 byte version number>\0'. In our database, we found that most of the malware had dex035 headers – which is an older, but still recognized version (API level 13 and earlier), and dex036 is used in the current version (4.x) of dex files [14]. This issue had already been noticed in *IDA*, and a fix has been made available [15], but at the time of writing this paper we still don't have a fix available for the default installation of *IDA* version 6.4adv.

5.3 Android/DroidRt, Andr/Obad-A – use of unfamiliar opcodes and decompilation

SHA (dex) – 90462f3ada7f4d551fc8f7d1e2672c4eea9e8cc8

In *Windows* PE files it is common to see use of obfuscation and anti-analysis codes in order to hinder sample analysis. We have seen previous work where illegal and improper use of opcodes has been suggested to break analysis tools [16]. Malware

authors and developers of commercial packers are aware of this, and use this technique extensively to make analysis of samples difficult.

We encountered an Andr/DroidRt sample which failed to decompile with *dex2jar* (Figure 18).

Dex2jar failed, saying that the format for opcode 64 could not be found. The opcode 64 documentation says 'Reads the byte static field identified by the field_id into vx' [16] – unfamiliar opcodes are usually used to delay reversing. Inserting junk bytes and bad opcodes is a known technique for thwarting reverse engineering, and in the past researchers have shown how insertion of these opcodes has broken analysis tools [16, 17] (these techniques are still used effectively). As an added bonus, this sample also failed to work with *baksmali 1.4.1* (Figure 19), but managed to work with the 2.03 version.

Obad has been called one of the most advanced pieces of *Android* malware. It has posed challenges in static analysis as it takes advantage of a vulnerability in the Davlik to Java conversion in *dex2jar* [18] (Figure 20).

Because of the error, the analyst would not be able to see a proper decompiled representation of the code and could come to the wrong conclusion if not analysed correctly. Obad is protected with the *dexguard* commercial packer [19] – this means that any application protected with this packer will produce the same issue with *dex2jar*.

6. BEST APPROACH

From our evaluation, we have found that several popular and commonly used tools are not suitable for effective static analysis of certain *Android* malware samples. While *Android* malware is growing both in complexity and volume at an exponential rate, the development of new tools and maintenance of existing ones are not matching that pace.

Even though most of the errors and tricks employed by malware authors to break analysis tools have been known to researchers for a while, the tools are not sufficiently well maintained to implement the necessary changes to overcome these issues. The *Android* SDK tools *baksmali*/*smali* are updated frequently, but they alone are not sufficient to provide full in-depth analysis of malware. In order to be able to understand how complex sophisticated malware works, analysts have to combine static analysis with dynamic tools. The consequence of this is that it buys more time for malware authors to continue their work while analysts spend more time analysing the samples. As a result, the security of *Android* applications and users is affected. To tackle this problem, *Google* should provide resources and funding for widely used tools and maintain them with regular updates to keep in line with malware advancements. Static analysis should be combined with the dynamic and sandbox environment.

7. CONCLUSION

Android malware is growing at a steady rate to match its counterparts in the PC world – we already have several malware families using *Android* exploits. From a web browser exploit to extensive root exploitation, cybercriminals have created complex pieces of malware that comprise multiple exploits and have a high level of obfuscation. Even an *Android* application installer, an APK file validation, has been compromised. By comparing the evolution of *Android* malware from an exploitation of vulnerabilities point of view with the evolution of tools used to analyse the exploit samples, we conclude that existing tools are not sufficiently well maintained to match advancements in malware.

We would like to add that, despite the fact that *Google* constantly attempts to improve its app verification processes, it is obvious that it is just playing catch-up. We keep seeing cases that involve compromised applications, whether it is down to the process of signature verification or compromised signatures. We anticipate that *Android* malware will grow and become more complex in the future. In order to combat it effectively, we need to find new ways of tackling it, keeping existing tools up to date and investing in new tools to make analysis easier.

REFERENCES

[1] Gartner (March 2014). http://techcrunch.com/2014/03/27/gartner-devices-forecast-2014/.

[2] http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html&sa=X&ei=rxZ6U4u2N6XN7AaNlYG4BA&ved=0CCgQ7xYoAA&biw=1920&bih=846.

[3] http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html.

[4] http://nakedsecurity.sophos.com/2010/11/08/pressure-to-improve-android-security-is-building-up/.

[5] http://thomascannon.net/blog/2010/11/android-data-stealing-vulnerability/.

[6] http://www.pcworld.com/article/221247/droiddream_becomes_android_market_nightmare.html.

[7] CVE-2011-2344. http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html.

[8] http://www.uni-ulm.de/en/in/mi/staff/koenings/catching-authtokens.html.

[9] http://www.h-online.com/open/news/item/Second-Android-signature-attack-disclosed-1918061.html.

[10] Ducklin, P. Anatomy of another Android hole – Chinese researchers claim new code verification bypass. http://nakedsecurity.sophos.com/2013/07/17/anatomy-of-another-android-hole-chinese-researchers-claim-new-code-verification-bypass/.

[11] https://android.googlesource.com/platform/libcore/+/9edf43dfcc35c761d97eb9156ac4254152ddbc55.

[12] Ducklin, P. Anatomy of a security hole – Google's 'Android Master Key' debacle explained. http://nakedsecurity.sophos.com/2013/07/10/anatomy-of-a-security-hole-googles-android-master-key-debacle-explained/.

[13] Dex File Format. http://source.android.com/devices/tech/dalvik/dex-format.html.

[14] Android 4.1.2 Dexfile.h source. http://osxr.org/android/source/dalvik/libdex/DexFile.h.

[15] http://www.strazzere.com/blog/2013/02/loose-documentation-leads-to-easy-disassembler-breakages/.

[16] http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.

[17] http://archive.hack.lu/2013/AbusingDalvikBeyondRecognition.pdf.

[18] Dex2Jar. http://code.google.com/p/dex2jar/.

[19] Dexguard. http://www.saikoa.com/dexguard.

[20] Unzip. http://linux.about.com/od/commands/l/blcmdl1_unzip.htm.

[21] Unuchek, R. The most sophisticated Android Trojan. https://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan.

[22] http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf.