Detection of Encrypted Android Malware Using Dalvik Hooks

Ionuț-Mihăiță Pleșea, Silviu-Mihai Popescu

Computer Science and Engineering Department
University POLITEHNICA of Bucharest
Bucharest, Romania
{ionut.plesea, silviu.popescu}@cti.pub.ro

Abstract. There are several layers of analysis and isolation in the Android application environment, all develope with the explicit purpose of ensuring that the features of smartphones do not expose personal user information to malicious third parties. While the current set of barriers covers many attack vectors, the solutions are limited if the malware is delivered to the victim device as a seemingly inoffensive file only to be transformed using cryptographic primitives into a malicious application package. Since the transformation is done on the victim device, most safeguards are bypassed. We propose a system that detects tentative attacks which use the above mentioned approach by instrumenting the cryptographic primitives in order to detect if both input and output are valid files.

Keywords: Android, Dalvik, cryptography, malware, method hooking, AngeCryption, instrumentation

1 Introduction

Having been on the market for 6 years, the Android operating system and the environment around it have evolved continuously in reply to user demand. With this growth in popularity the need for providing adequate security for personal data has been increasingly pronounced.

While safety mechanisms have always been present in Android, it was only recently[3] that the Android Security Team highlighted the protection features available on more than half of the smartphones sold globally. A summary can be seen in Figure 1. With so many checks and analysis being done both on Google servers and on the device itself, successfully compromising a smartphone is increasingly difficult for an attacker and consequently innovative manners of bypassing these filters have been developed.

One approach employed by malicious agents is to use picture files bundled with the Android application. Normally pictures are bundled with applications for cosmetic purposes and therefore ignored and considered harmless. However, there is nothing preventing a determined attacker to use such a file as a wrapper

for hiding a harmful payload. Basically, the original application is a trojan, indistinguishable from an ordinary application.

The first incident of this type has been reported in September 2011[1]. An application called DroidCoupon pretended to offer various coupons in order to lure users to download it. However, it used an image file to hide a local privilege exploit. For all intents and purposes the special file behaved like an image file, but the DroidCoupon application, reading from a custom offset, extracted the local exploit and used it to gain root access on the smartphone. From that point, the attacker could violate security protections and leak private user data.

As time progressed, so has the complexity of the techniques used by attackers. In 2013, the Gamex malware also used an image file to hide the malicious payload[2]. However, its approach was a little more elegant. A file with the PNG extension was, in fact, a ZIP archive and by default would extract harmless files. Nevertheless, if the ZIP archive was XOR-ed with a certain key it produced another valid ZIP, which contained the malicious payload. This was a significand step up from previous incidents, as recognizing such behavior was not a trivial task.

Last but not least, in 2014, researchers went a step further[2]. Although the behavior of Gamex was atypical, the fact that it used XOR encryption meant that subsequent tentatives of the same manner could be thwarted. Therefore, they employed the cryptographic ciphers exposed by the Android framework in a rather unorthodox manner. In addition, they relied on the fact that parsers for specific file formats are error-prone. As a result, they managed to encrypt a PNG file using AES in CBC mode and obtain a valid APK file, an Android application that could be installed on the victim software. Using a standardized encryption algorithm to hide the malicious payload renders any analysis useless as it is computationally infeasible to crack AES.

Having outlined previous work with regards to the problem we attempt to solve, we will now take into account prior research that may aid in the development of an effective countermeasure. Since the current trend seems to be using more secure cryptographic ciphers, namely the ones provided by the Android framework itself, we consider instrumenting the framework itself a valid approach. More to the point, we wrap methods from Android's cryptographic

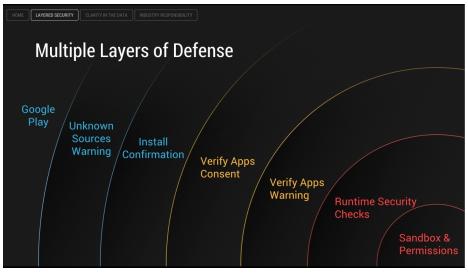

Fig. 1. Android Security Layers - Image courtesy of Andrian Ludwig

API and look for valid file headers in both the input and the output. Normally, depending whether we are encrypting of decrypting, either the input or the output should have a valid file signature, while the other should seem random. If both are not random, then there is a high probability that we are witnessing a tentative attack.

There are a few instrumentation frameworks available for the Android platform. Most are geared towards modifyig the user interface and adding features. They provide complex functionality. Therefore, they are not trivial to setup and, considering out target are cryptographic operations, may add non-negligible overhead.

Thankfully, security researchers have developed simplified alternatives. While lacking in features when compared with more mature instrumentation frameworks, DDI[4] is lightweight and will not induce a high computational overhead. It works by injecting a library in the Dalvik runtime that replaces the original entry points to the methods of interest. The Dalvik virtual machine is the part of the Android operating system that runs the individual applications. Therefore, a wrapper function here can be applied to the method of our choice in the context of the application we wish to monitor.

2 Instrumenting Cryptographic Primitives

The first step in detecting the above-mentioned attack vector would be to reliably intercept calls to methods pertaining to Android's cryptographic framework. Towards this outcome, we will require some of the functionality exposed by DDI, but with some modifications in order to facilitate preemptive detection and blocking of malicious applications. Figure 2 provides a high-level overview of our proposed solution. We will now outline each module, as well as their respective interactions.

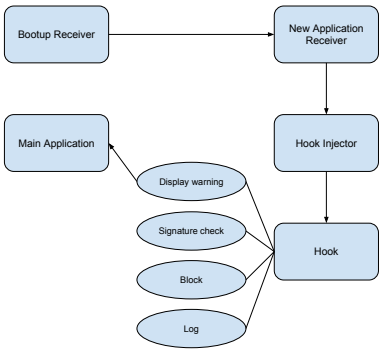

Fig. 2. Ciphermon Modules

2.1 Bootup Receiver

We have two prerequisites in order to properly detect malicious applications. The first is to have a stock Android system, in order to exclude the possibility that a malicious application is already installed. Such malware could analyze our solution during installation and take adequate defensive measures. The second requirement is that the detector start as soon as possible within the Android booting process. This is mandated since there is no telling when the user decides to install a new application and our solution must be already runnning and in active monitoring state.

To this end there is a need for a bootup receiver which waits for a notification from the Android system that booting has completed. When our application is notified that the device is ready for normal operation it enables the New Application Receiver.

2.2 New Application Receiver

Since our goal is to block malicious applications from running on the device, the best angle of approach towards solving this issue would be to detect when an application is installed. We can leverage the functionality exposed by the Android framework to detect when a new application is installed and notify the Hook Injector.

2.3 Hook Injector

Whenever a new application is installed, a new thread is spawned that injects the hook into the memory of Zygote, the Android daemon process that is responsible for launching new applications. The need for a dedicated thread is that, in order to inject the hook before the target application is launched, we must wait for Zygote to create a child process which will load the image of the target application.

This module is, in fact, an adaptation of the DDI framework and relies heavily on the work of Collin Mulliner. Since DDI was initially conceived for security research we had to retrofit it for active runtime detection.

2.4 Hook

The hook is a dynamic library that is loaded by the Hook Injector. This is done by changing the permissions of the code memory area and then writing ARM assembly that instruments the original function call with our custom wrapper. Withing this hook we let the original function run and then check the input and the output buffers using signatures. If both buffers have valid filetype signatures, we log the malicious attempt, block it from continuing and signal the Main Application.

2.5 Main Application

The Main Application module has a simple purpose, namely to issue a warning to the user. This notifies the user that there is a suspicious action being carried out by a certain application.

3 Ciphermon - Monitoring Cipher Operations

Since our research covers the Android operating systems, it is natural that the proposed solution comes as an Android application. To this end, we have made use of and repackaged Collin Mulliner's Dynamic Dalvik Instrumentation framework. Full credit is given for his excellent implementation of Dalvik hooking.

We have looked at the demonstrative application that researchers have previously[2] developed in order to highlight encrypted malware. While our approach is extensible, we have targeted this specific sample.

3.1 Tools and Constraints

The demonstrative application that highlights encrypted malware can be found on Github under the name *angecrypt*[1].

We wrapped our solution around it so when we designed our hook we specifically targeted a function called *doFinal()* [2] by taking its input and output and comparing them to signatures of known file formats (in this particular demonstrative application, the input was a PNG file and the output a valid APK file).

We used the Android SDK[3] to build the Bootup Receiver, the New Application Receiver and the Main Application modules, and relied on the Android NDK[4] when tackling the Hook Injector and the Hook modules.

One of the biggest constrain that we had was the fact that we used an emulator instead of a rooted Android phone due to financial limitations (warranty voiding due to rooting). Because of this we were unable to fully automate the Hook Injector and the hijacking of the above specified method had to be done manually.

4 Results

The main goal, i.e. detecting malicious action being carried out by a suspiciuos application (namely angecrypt), of our PoC (Proof-of-Concept) environment was achieved. We will further expand the process through which our result can be reproduced.

[1] `https://github.com/cryptax/angeapk`

[2] `http://developer.android.com/reference/javax/crypto/Cipher.html#doFinal()`

[3] `https://developer.android.com/sdk/index.html#Other`

[4] `https://developer.android.com/tools/sdk/ndk/index.html#Downloads`

4.1 Setup Tools and Projects

In order to reproduce our result, a rooted phone is needed or at least an emulator that has superuser access. In our experiments, we resorted to using an emulator with Android 4.4.2 and API level 19 that was emulating an ARMv7 architecture. It is considered implicit the availability of an Android SDK as well as an Android NDK.

One would also need the sources of *ADBI*[5] and *DDI*[6] projects developed by [4]. In order to use our hook instead of the examples provided in DDI, our project[7] sources have to be dumped in the ddi/examples folder.

Last but not least the suspicious application is required. That can be found on Github as well under the name emphangeapk [8]

4.2 PoC Reproduction

The suspiciuos APK can be found in the Github repository mentioned above. The *adb* tool can be used to install the app on the phone/emulator. Assuming your are in the angeapk folder:

```
adb install PocActivity-debug.apk
```

will install the APK.

We will now compile binaries that support the hooking platform. We need to be in the ADBI folder.

```
cd adbi/hijack/jni
ndk-build
adb push ../libs/armeabi/hijack /data/local/tmp/
cd ../..
cd instruments/base/jni
ndk-build
cd ../../../..
```

Next, we will compile the libraries needed for the dalvik machine. We need to be in the DDI folder.

```
cd ddi/dalvikhook/jni/libs
adb pull /system/lib/libdl.so
adb pull /system/lib/libdvm.so
cd ../
ndk-build
cd ../../..
```

[5] https://github.com/crmulliner/adbi
[6] https://github.com/crmulliner/ddi
[7] https://github.com/silviupopescu/ciphermon/blob/master/jni/mon.c
[8] https://github.com/cryptax/angeapk/tree/master/wrapping-apk/src/com/fortiguard/poc/angecrypt

Before compiling the libraries needed for the hook, make sure there is a folder called ciphermon in ddi/examples that contains the jni folder with our custom hook.

```
cd ddi/examples/ciphermon/jni
ndk-build
```

A library called *libciphermon.so* should be available now. We just push it on the phone

```
adb push ../libs/armeabi/libciphermon.so /data/local/tmp
```

and then run a shell within the phone/emulator and hook our library to the specified maliciuos application.

```
adb shell \# pentru conectat la dispozitiv Android
su
ps | grep ange \#to find the PID of said suspicious application (angecrypt)
cd /data/local/tmp
touch /data/local/tmp/ciphermon.log
chmod 777 /data/local/tmp/ciphermon.log
./hijack -d -p \$PID -l /data/local/tmp/libciphermon.so
cat /data/local/tmp/ciphermon/log
```

The output from the last command should contain information pertaining the use of the targeted suspicious application.

5 Conclusion and Further Work

Although not through a completely automated process, we managed to provide a Proof-of-Concept setting for succesfully detecting whether a suspicious application encodes/decodes malicious information to be used on our phone and stopping that application from executing its task.

Amongst our future work, an automation of the hooking process by using the Zygote, the Android daemon process that is responsible for launching new applications, to bypass manual hooking and the removal of the need for root access of the phone are two of the main focusing points of our development.

We also want to extend the current application to monitor not only one application but every application that is newly installed by the user as well as any combination of possibly encode/decode file formats to increase awareness regarding malicious aplications.

References

1. Droidcoupon. http://www.csc.ncsu.edu/faculty/jiang/DroidCoupon/.
2. A. Apvrille and A. Albertini. Hide Android Applications in Images. *Blackhat Europe 2014*, Amsterdam, Netherlands, October 14-17, 2014.

3. A. Ludwig. Android - Practical Security From the Ground Up. *VirusBulletin 2013*, Berlin, Germany, 2013.
4. C. Mulliner. Dynamic Dalvik Instrumentation Framework for Android. *30th Chaos Communication Congress*, Hamburg, Germany, December 27-30, 2013.