Lab 9: Mobile Security

Jonathan Harijanto

March 16, 2017

CS 373: Defense Against the Dark Arts - Winter 2017

Abstract

The purpose of this blog is to describe my observation from the labs provided by Intel team. This blog will cover the testing methodology and the knowledge that was gain from completing the lab.

I. BLOG

A. What you looked at:

For this week, we were learning the topic of mobile security. There were two different labs where we had to analyze various infected apps (a.k.a malware) statically and dynamically. There were a lot of tools introduced for the analysis process such as JD-GUI – a Java decompiler, Apktool – a reverse engineering tool for Android apk files, Eclipse Juno – an IDE, dex2jar – a tool for reading Java .class files. Also, the samples that we analyzed were all Android apps because Android is the most-used mobile operating system in the world.

B. How you looked at it & What you found:

1) Lab 1::

There were two types of analysis process in this lab: static and dynamic. The static means I had to analyze the malware apps either by reading the results of a scanning service or reverse-engineering the code. The app that I had to analyze statically was called FakePlayer, the first Android malware (according to the instructor). First, I had to analyze that malware using a VirusTotal website – online malware scanner, by entering the hash number. I did this by doing `md5sum` command in the malware directory and inputting the hash on the online scanner site. I could see that this malware has something to do with sending a text message because the only permission needed by this app was `SEND_SMS`. Also, I noticed that the certificate listed is untrusted because it contains only few information.

The next step was to decompile the app with an Apktool so that I could see the de-codified version of AndroidManifest.xml. This file provides the app's essential information to the Android system such as the list of permission used and class being called in the app. The last step was to reverse engineered the code using dex2jar and JD-GUI. I had to open the correct directory and search for the `classes.dex` file. Then, I need to open the terminal to execute `sh ~/tools/dex2jar/d2j-dex2jar.sh classes.dex` command. This command created a jar file that contained all the java file of the apps. From there, I just need to analyze all these files using JD-GUI.

Based on my observation, I noticed that the malware (FakePlayer) sent **798657** text messages and the premium-rate numbers that the malware sending were **3353** and **3354**. I obtained this answer from a method call that looks like `sendTextMessage("3353", null, "798657", null, null)`. Also, the mechanism that are is automatically to execute the payload in the background was through `onCreate()` method.

The other part of the first lab was the dynamic analysis. Dynamic means that I analyzed the malware by executing it in real-time. The app that I had to analyze changed, this time I had to investigate a new one called `Walktxt.apk`. I started the analysis process by creating a new Android virtual device on Eclipse Juno. Next, I opened a terminal and set an environment for `adt-bundle-linux-x86_64-20140702`. In this bundle, there was an emulator that runs the emulator in Eclipse Juno. Essentially, I was trying to connect two emulators so they could communicate with each other as if I'm testing the app using two Android smartphones. Anyhow, to perform that, I went to the adt-bundle tools directory and entered the `./emulator @lollipop -tcpdump capture.pcap` command in the terminal. Then, I created a new phone number in this second emulator and installed the `Walktxt` app. Finally, I opened up my first emulator in Eclipse Juno and received a payload in the form of a text message from my second emulator.

2) Lab 2::

The first section of lab 2 was another static analysis. The malware that I had to analyze was called DroidDream, the first Android nightmare because its intention was to root a device using exploitation. Using the same procedure from the previous lab, I decompiled the APK using a much-shorter terminal command `java -jar /home/analysis/tools/AXMLPrinter2.jara AndroidManifest.xml > DecodedManifest.x` that will yield an XML de-codified file. From this manifest file, I could see that DroidDream required

lots of permissions such as INTERNET, READ_PHONE_STATE, CHANGE_WIFI_STATE, and ACCESS_WIFI_STATE. Furthermore, this app also used two different classes which are Activity and Service. Next, I converted the `classes.dex` file into a jar so I could take a look at the Java files. I was surprised to see this malware had many Java files just like a real app (approximately 20 different files). After analyzing these data for some time, I figured out that the entry points for the malware were from `AlarmsReceiver` and `Setting` class because both of them are of the class of `Service` – the component that starts and ends automatically in the background. Also, I noticed the execution of Exploid was at `prepareRawFile()` in `udevRoot` class and the drop was at `removeExploit()` in `udevRoot` class. On the other hand, the execution of RATC was at `RunExploid()` in `adbRoot` class and the drop was at `dopermroot()` in `AlarmReceiver` class. Last but not least, I managed to identify that **changing the Wi-Fi state** is the only action required to trigger the exploits.

Similar to the Lab 1, the other part of this lab was the dynamic analysis. Primarily, this lab wanted to show that one of the emulators that have been infected by a Trojan will pretend to generate a fake mobile transaction authentication number and later affect the victim phone through SMS message. In the end, the lab result would prove that a victim phone that's running Android Lollipop version didn't receive the malicious text. The reason is that there were some enhancements in Lollipop updates that could automatically filter several malicious actions. Unfortunately, I didn't finish the lab completely. My stopping point was at the stage where I must send SMS message with the command `/00005556`.

II. Conclusion

There was a lot of useful information about mobile security to take in. I was glad to know another new tool such as JD-GUI, Apktool, and dex2jar. Furthermore, I gained new concepts about mobile security issues, especially in Android platform. And last but not least, I learned how to perform basic static and dynamic analysis of Android applications.