Android Malware Analysis Lab

June 11, 2017

1 Background

1.1 Android Manifest File

[1] Every application must have an AndroidManifest.xml file in its root directory. The manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app's code. Among other things, the manifest file does the following

- It names the Java package for the application.
- It describes the components of the application, which include the activies, services, broadcast receivers, and content providers that compost that application. It also names the classes that implement each of the components and publishes their capabilities, such as the Intent messages that they can handle. These declarations inform the Android system of the components and the conditions in which they can be launched.
- It declares the permissions that the application must have in order to access protected parts of the API and interact with other applications. It also declares the permissions that others are required to have in order to interact with the app's components.

In Listing 1 an example of an app's manifest file is shown. From it, we can see that this app declares that it needs the INTERNET and RECEIVE_SMS permissions. Additionally, the app uses three components: ActivityOne, SmsReceiver, and myAppsService. ActivityOne is declared in lines 80-85. The intent-filter tag specifies the types of intents that an activity, service, or broadcast receive can respond to. An intent filter declares the capabilities of its parent component – what an activity or service can do and what types of broadcasts a receiver can handle. It opens the component to receiving the intents of the advertised type, while filtering out those that are not meaningful for the component. Lines 16-21 declare a broadcast receiver component named SmsReceiver.

[1] Portions of this section are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

From the intent filters, we see that the Android OS will notify SmsReceiver when the device receives a new text message. The final component this app uses is a service component named ServiceOfApp declared on lines 23-25.

The Android Manifest file provides a high-level abstraction of an app's behavior. When attempting to manually inspect the internal behaviors of an application statically, the manifest file is a good starting point. It provides key insights on the permissions an application is using, the components it is using, and how the application interacts with the Android OS and the outside world. Additional information about the contents and attributes of the manifest file can be found in the Android documentation [1].

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest ... package="com.myApplicationPackage">
3
4  <uses-permission android:name="android.permission.INTERNET"/>
5  <uses-permission android:name="android.permission.RECEIVE_SMS"/>
6
7  <application
8
9  <activity android:name=".ActivityOne">
10     <intent-filter>
11         <action android:name="android.intent.action.MAIN"/>
12         <category android:name="android.intent.category.LAUNCHER"/>
13     </intent-filter>
14 </activity>
15
16 <receiver android:name="SmsReceiver">
17     <intent-filter>
18         <action android:name=
19             "android.provider.Telephony.SMS_RECEIVED"/>
20     </intent-filter>
21 </receiver>
22
23 <service
24     android:name=".ServiceOfApp"
25 </service>
26
27 <provider android:authorities="de.ub0r.android.smsdroid" android:
       name=".MessageProvider"/>
28
29 /application>
30
31 </manifest>
```

Listing 1: An example of an app's Android Manifest File

1.2 Disassembling Android Apps

Android uses the Android application package (APK) format to distribute apps to Android devices. Apks are nothing more than a zip file containing resources and assembled Java code. However, if you were to simply unzip the apk you would only have two files: *classes.dex* and *resources.arsc*. Since viewing or editing compiled files is next to impossible, the apk file needs to be decoded or disassembled. If one wishes to analyze an app at the bytecode level, reverse engineering tools, such as Apktool [2] are available. Additionally, the app's Java source code can be partially reconstructed using JADX [3]. You will probably find both tools useful for completing this lab.

1.2.1 Disassembling Apps using Apktool

Apktool is a reverse engineering tool for Android apps. It can decode resources to nearly original form and rebuild them after making some modifications. It also makes working with an app easier because of the project like file structure and automation of some repetitive tasks like building apk, etc. [2]. The functionality of Apktool is well-documented and we will briefly describes how this tool can be used to decode and build apk files. More information about Apktool can be found in its documentation [2].

1.2.2 Decoding apps using Apktool

In this example, we will use Apktool to decompile a malicious apk that was found in the wild (a7f94d45c7e1de8033db7f064189f89e82ac12c1) [4]. The apk is a repackaged version of the CoinPirates game that includes a malicious payload.

Apktool provides a command line interface. Its most common use case is for decoding and disassembling apk files. If you need to decode an apk file, you use the d (decode) option and pass the apk file as an argument. An example is shown in Listing 2 on line 1.

```
1  $:apktool d a7f94d45c7e1de8033db7f064189f89e82ac12c1.apk
2  I: Using Apktool 2.2.1 on a7f94d45c7e1de8033db7f064189f89e82ac12c1.
     apk
3  I: Loading resource table...
4  I: Decoding AndroidManifest.xml with resources...
5  I: Loading resource table from file: /home/joey/.local/share/
     Apktool/framework/1.apk
6  I: Regular manifest package...
7  I: Decoding file-resources...
8  I: Decoding values */* XMLs...
9  I: Baksmaling classes.dex...
10 I: Copying assets and libs...
11 I: Copying unknown files...
12 I: Copying original files...
13 $:ls
14 a7f94d45c7e1de8033db7f064189f89e82ac12c1/
15 a7f94d45c7e1de8033db7f064189f89e82ac12c1.apk
```

Listing 2: Decoding an apk using Apktool.

If you look in the directory created you should see something similar to Listing 3. For this lab, we will focus mostly on the AndroidManifest.xml file, the res/ directory, and the smali/ directory. The app's resources, such as its images and layouts can be found in the res/ directory. In the smali/ directory, the original classes found in the classes.dex file can be found. Apktool converts the original classes.dex file into smali using baksmali[5], an assembler/disassembler for the dex format. We will discuss the contents of these files and smali syntax later on.

```
$:ls
AndroidManifest.xml  Apktool.yml  lib/  original/  res/  smali/
```

Listing 3: Contents of the directory created.

1.2.3 Building apk files using Apktool

Apktool also can rebuild an apk file from the decoded resources after making some modifications, such as modifying the smali code. To build an app you need to provide the b (build) parameter to Apktool and also provide the decoded directory as an argument like the example in Listing 4.

```
$Apktool b a7f94d45c7e1de8033db7f064189f89e82ac12c1/
I: Using Apktool 2.2.1
I: Checking whether sources has changed...
I: Checking whether resources has changed...
I: Building apk file...
I: Copying unknown files/dir...
```

Listing 4: Rebuilding an apk file using Apktool.

If you received no errors, the new apk should be found in the dist subdirectory of the directory provided as input. For example the apk created from running the command in Listing 4 is shown in Listing 5. In your working directory, you will still have a copy of the original apk file. It does not include any modifications you may have made.

```
$cd a7f94d45c7e1de8033db7f064189f89e82ac12c1/dist/
$ls
a7f94d45c7e1de8033db7f064189f89e82ac12c1.apk
```

Listing 5: **The location of the modified apk.**

The next step is to sign the apk you just created. **If the apk has not been signed it will fail to install on an emulator or real device.** The Android SDK provides a utility program called *apksigner* that is located in the Android/Sdk/build-tools/*SDK version*/ directory. We have provided this program on your VM (You can also use jarsigner if you prefer). For this lab, you should just sign the apk with the debug key, which is located in the debug.keystore file located in your $HOME/.android/ directory. An example of signing an apk is shown in Listing 6. You need to provide the location of the keystore after the –ks option and pass the apk file as an argument. You will be prompted for a password. The default password is android.

```
$apksigner sign --ks ~/.android/debug.keystore
    a7f94d45c7e1de8033db7f064189f89e82ac12c1.apk
Keystore password for signer #1:
$
```

Listing 6: **Signing your apk file (password is android).**

After you have signed your apk, install it onto the emulator to verify everything went correctly.

1.3 Making modification using Apktool

Apktool can also be useful for making small modifications to the underlying byte code. For example, let's assume a malicious app is using the anti-analysis check shown in Listing 7 to prevent the execution of any malicious behavior if the Build type is *eng*. Use apktool to disassemble this app, so that you can modify the code located in the smali directory. Use apktool to disassemble the app located in tutorialApps/emu-check.apk. After you have done so, open the file *emu-check/smali/com/myapplication/MainActivity.smali* in a text editor. You will see the code shown in Listing 8. The code shown is smali and is a representation of Dalvik bytecode. The Android Developer's website provides a page that discusses the types of instructions and arguments [6].

For the checkEnvironment method, the app is checking the model's build type to see if it is equal to the string "eng". In the bytecode, we see that the value of Build.TYPE is stored in register v0 on line 7. The string constant "eng" is stored in register v1 on line 9. The comparison of the strings is completed on line 11 and the result is stored in register v0. On line 13 we see that if the value stored in register v0 is equal to zero, then a jump to the cond_0 branch will occur. Therefore, if the Build.TYPE is not "eng" then a jump to cond_0 occurs and the malicious behavior will be triggered. Since we are on an emulator, our Build.TYPE will be "eng" and the jump will not occur. To force the control-flow to go to cond_0, change the statement on line 15 to "goto :cond_0". This will force the branch to occur every time the app runs. Build and sign the app. Install it onto the emulator (If you installed the previous version you will need to uninstall it first) and open the app. If you check logcat, you will see that the Build type is "eng". However, the app will now log the "do something malicious" instead.

```java
protected void checkEnvironment(){
    if ( Build.TYPE.equals("eng")) {
        Log.d(TAG, "checkEnvironment: do nothing");
        return;
    } else {
        Log.d(TAG, "checkEnvironment: do something malicious.");
    }
}
```

Listing 7: Prevents malicious behavior if the build type is eng.

```
# virtual methods
.method protected checkEnvironment()V
    .locals 2

    .prologue
    .line 21
    sget-object v0, Landroid/os/Build;->TYPE:Ljava/lang/String;

    const-string v1, "eng"

    invoke-virtual {v0, v1}, Ljava/lang/String;->equals(Ljava/lang/
    Object;)Z

    move-result v0

    if-eqz v0, :cond_0

    .line 22
    const-string v0, "MainActivity"

    const-string v1, "checkEnvironment: do nothing"

    invoke-static {v0, v1}, Landroid/util/Log;->d(Ljava/lang/String
    ;Ljava/lang/String;)I

    .line 27
    :goto_0
    return-void

    .line 25
    :cond_0
    const-string v0, "MainActivity"

    const-string v1, "checkEnvironment: do something malicious."

    invoke-static {v0, v1}, Landroid/util/Log;->d(Ljava/lang/String
    ;Ljava/lang/String;)I

    goto :goto_0
.end method
```

Listing 8: checkEnvironment in smali.

1.4 Using JADX to disassemble Dex to Java Source Code

JADX [3] is another tool that can be used to disassemble apk files. However, JADX disassembles the Dalvik byte code into JAVA source code. The translation is imperfect and will most likely be incomplete, but it is still useful for doing analysis. JADX provides two interfaces: a command line interface and a gui interface. For this lab, we will only discuss the gui interface. You can start the GUI interface of JADX by running jadx-gui from the command line. When the program first opens, it will ask the user to choose a file to disassemble. It supports apk, dex, jar, class, zip, and aar files. This discussion will only discuss using apk files. After you choose the apk file, JADX will begin disassembling the apk. When it's complete you should see the source code for each class in the Menu pane. If you review the source code, you can see it is not ideal, but it does provide insight into the app's behavior.

2 Using the disassembled Code To Search For Suspicious Behavior

Now that we have disassembled the apk file, we can begin analyzing the source code to identify suspicious behavior. Defining behavior within Android is challenging. Behavior that may be suspicious or malicious in one application may be expected behavior in another application. It is reasonable for a messaging app to access a user's contacts, but if a utility app, such as a flashlight app, accesses a user's contacts it should raise suspicion. Therefore, the behavior that makes an application potentially malicious is not a particular pattern, but the behavior in an application that is inconsistent with the end user's expectation. The easiest starting point for identifying any questionable behavior is by looking at the App's manifest file. The manifest file provides a high-level abstract of an app's behavior

2.1 Analyzing CoinPirates Manifest File

In JADX, the AndroidManifest.xml is located in the Resources/ directory. The highest level of security for Android is the permission system that protects the usage of sensitive behavior. The manifest file shows us that the CoinPirates app has access to 14 permissions. Malware often abuses the text messaging permissions to communicate with their C&C server and to try and send premium text messages without the user being aware.

```

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.
    ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.
    READ_PHONE_STATE" />
    <uses-permission android:name="android.permission.WRITE_SMS" />
    <uses-permission android:name="android.permission.RECEIVE_SMS"
    />
    <uses-permission android:name="android.permission.SEND_SMS" />
    <uses-permission android:name="android.permission.
    RECEIVE_BOOT_COMPLETED" />
    <uses-permission android:name="android.permission.
    CHANGE_NETWORK_STATE" />
    <uses-permission android:name="android.permission.
    WRITE_APN_SETTINGS" />
    <uses-permission android:name="android.permission.
    ACCESS_WIFI_STATE" />
    <uses-permission android:name="android.permission.
    CHANGE_WIFI_STATE" />
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <uses-permission android:name="com.android.browser.permission.
    READ_HISTORY_BOOKMARKS" />
    <uses-permission android:name="com.android.browser.permission.
    WRITE_HISTORY_BOOKMARKS" />
```

Listing 9: Permissions used by CoinPirates

After observing the permissions, the next goal is to vet the application by analyzing how the application uses the sensitive APIs that are protected by the suspicious permissions. Since malware writers often repackage their payload within real apps with 100's of classes, it would be too time-consuming to search through all the source code. Instead, we will focus on the entry points of the application.

3 Identifying Entry points into an Android applications

[2] Android applications are written using the Java programming language. Unlike conventional Java programs, Android applications do not have a *main()* function or a single entry point for execution. Instead, they are designed using components. App components make up the essential building blocks of an Android app. Each component is a different point through which the system can enter a developer's application. There are four different types of components: activities, services, content providers, and broadcast receivers. Each type of component serves a different role and the set of components used in an Android application define its overall behavior. The *activity* component creates user

[2]Portions of this section are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

interfaces. For example, a messaging application may have one activity that creates the user interface for allowing a user to input their message and another activity for allowing the user to view their contacts. The *service* component runs in the background to perform tasks. Unlike, activity components, service components do not have a user interface. For example, a service component can be used to play music in the background. The *content provider* component handles application data. Using content providers, an application can store data in files, SQLite databases, or other persistent storage locations an application can access. The *broadcast receiver* component responds to system-wide broadcast announcements. For example, the system may broadcast that a picture has been captured, and the broadcast receiver can alert the application of this action. In general, broadcast receivers do minimal work, but instead, alert other components that an event occurred.

Since the components are required to be declared in the manifest, this allows us to quickly identify any interesting entry points without having to search through the source code. To avoid detection, malware usually does not trigger until it receives commands from its C&C server. The two most common and efficient wants for this communication is through the network and sms. Since SMS can provide communication when the user does not have a wifi connection, it is usually preferred. Since this app has declared the RECEIVE_SMS permission, we know that it has the ability to receive broadcasts about arriving text messages through a broadcast receiver. If a broadcast receiver wants to receive a text message, it must specify that it can handle this action by adding the action to its intent filter inside the manifest file. The action required is shown in Listing 10.

```
<action android:name="android.provider.Telephony.SMS_RECEIVED" />
```

Listing 10: Action required to receive SMS broadcasts

In the CoinPirates manifest, we see that only one receiver has this ability, and the component's declaration provides us with enough information to identify the package and class name that declares the receiver. Additionally, the components declaration raises more suspicion. First, it is manipulating the naming convention and is located in the com.android package. Next, it has a priority of 10000. In Android, broadcasts can be ordered or sent to all apps at the same time. In general, applications with a higher priority will receive the broadcast first. Additionally, they have the choice of aborting the broadcast() or allowing it to be sent to the app with the next highest priority. Therefore, this behavior can be manipulated by malicious apps to hide the notification of received text messages[3]

[3] As of Android 4.4 this has been slightly adjusted. The default SMS app will always receive the broadcast first, regardless of priority.

```
    <receiver android:name="com.android.SMSReceiver">
        <intent-filter android:priority="10000">
            <action android:name="android.provider.Telephony.
SMS_RECEIVED" />
            <action android:name="android.provider.Telephony.
SMS_SENT" />
        </intent-filter>
    </receiver>
```

Listing 11: Action required to receive SMS broadcasts

If we use JADX to analyze the source code for the SMSReceiver class, we can identify any suspicious behavior that may occur when a text message is received. The Android OS notifies broadcast receivers by calling the receiver's onReceive method. Therefore, we should start our analysis from this point in the app. When looking over the source code of the onReceive method, we see that the method immediately queries a database called "mydb." The source code also shows us that the values received from the database are being compared to the sender's number and the contents of the sms body. Based on thee results of these comparisons, the app uses the needDel (delete text message) or needUpload variables to control the apps' control-flow.

Identifying suspicious entry point that are defined in the manifest file, allows us to quickly identify suspicious behavior. For example, After analyzing the SMSReceiver we see that it is being used by the C&C server to trigger malicious behavior. We also know that the app uses the "mydb" database to interpret the C&C servers commands. While the SMSReceiver app provides the most insight, the malicious app is also using two other receivers, AlarmReceiver and BootReceiver, to start the Monitor Service. We leaving analyzing the MonitorService component to the reader.

3.0.1 Triggering Malicious Behavior Dynamically

Using static analysis, we can identify the necessary events required to trigger malicious behavior in the app. Our next goal will be to leverage the details we extracted from the static analysis to dynamically generate the malicious behavior at run time.

In the case that the events necessary to trigger the malicious behavior is dependent on external sources, such as a text message being received, we will need to simulate these events. Android provides several tools for injecting events into the emulator, and you can read the full documentation on the Developer's Website [7]. One tool is the emulator console. Each running emulator instance provides a console that lets you query and control the emulated device environment. For example, you can use the console to manage port redirection, network characteristics, and telephony events while your application is running on the emulator. The console emulator will be useful for injecting events, such as text messages from a specific number or changing the location's device. The official documentation provides several examples.

3.1 Android Resources

A developer can provide an app with resources by placing it in a specific subdirectory of the res/ folder. Once you provide a resource in your application, you can use it by referencing its resource ID. Each resource is grouped into a "type" such as *string*, *layout*, or *drawable*.

When viewing an APK in JADX, you can find the resources an app uses in the Resources directory under the resources.arsc tab. After expanding the resources.arsc file, you can find many basic resources, such as hardcoded strings found in the values directory.

When JADX decompiles the APK back into source code, resources will be referenced by their ID in the R class, you can use this to create a mapping from the Resource ID to its original name in the res/resources.arsc/values subdirectory.

4 Lab Assignment

4.1 Scenario

- You have received a malware sample **sms.apk**. Your task is to discover what the malware does by analyzing it.
- You need to identify the components that are being used by the app to communicate with its C&C server.
- You need to identify any anti-analysis techniques being used by the app and remove them if necessary.
- You need to identify the commands that trigger the malicious behavior.

4.2 Grading

- Your answer must be in the correct format. If your answer does not follow the expected format, it will not be graded/regraded.

4.3 Project Structure

1. Tools
   (a) JADX
      i. Disassembles apk files into Java Source Code
   (b) Apktool
      i. Disassembles apk files into smali.
      ii. Rebuilds apk files.

4.4 Questionnaire

- To get your credit for the project, you have to answer the questionnaire in /android/report/assignment-questionnaire.txt
- For each stage, there are 4-5 questionnaire that inquires regarding the behavior of the malware.

4.5 Stage 1

4.5.1 Question 0: (5 points)

- Run the command **./start_server** and verify the server is active.
- Start the Android emulator.
- Use the **People** app that is preinstalled to add a contact to the device. The name of the contact should be your GT ID (e.g. JDoe2).
- Open the app that is named Messenger (not Messaging). This is the app installed from the sms.apk.
- The server will ask you if your GT ID is correct. If so, press 'y' and you will receive the first answer.
- Answer Format: You need to copy & paste the string between the answer tags. For example, if the output was < *answer* > 1234 < /*answer* >, your answer would be 1234. You should follow this answer format for all answers you receive from the server.
- After receiving your answer for Question 0, you can turn off the Android emulator and server. They will not be needed until later on. The remaining part of Stage 1 will be using JADX to analyze the sms.apk's source code.

4.5.2 Question 1: (10 points)

- **What is the name of the component that is used for communicating with the C&C server?**
- Related background sections: 1.1, 1.2, 3
- Answer Format: If the correct component was the receiver described below, the answer would be **com.android.AReceiver**.

```
        <receiver android:name="com.android.AReceiver">
            ...
        </receiver>
```

4.5.3 Question 2: (20 points)

Using SMS as a protocol for a C&C server is an important design decision that is different from traditional IP-based approaches known from infected PCs. The main advantages of an SMS-based approach instead of IP-based are the fact that it does not require steady connections, that SMS is ubiquitous, and that SMS can accommodate offline bots easily [8]. sms.apk is leveraging SMS to receive commands from its C&C server, you need to identify them.

- **When sms.apk receives a text message, it checks to see if the message matches a command. What are the commands?**
- Related background sections: 1.4, 2.1, 3, 3.1
- Answer Format: A list of commands (sms bodies) sms.apk receives from its C&C server. The list should be separated by end lines (one command per line).

At this point we should have enough information to trigger the malicious behavior. The C&C server can be started by running **./start_server** from the command line. Start the server and send the necessary text messages. Unfortunately, no malicious behavior will be exhibited. This is because the malicious app has placed anti-analysis techniques into the app to prevent analysis. Our next goal will be to find them and see if we can emulate these triggers or remove them.

4.5.4 Question 3: (20 points)

The Android/BadAccents malware, discussed in [8], contains two specific checks on the incoming SMS number. It checks for '84' and '82' numbers, which indicates that the malware expects SMS from a C&C SMS server either located in China or South Korea. It seems the app we are inspecting does something similar.

- **What country code does sms.apk require the incoming text message to have before the malicious behavior will be triggered?**
- Related background sections: 1.4, 3.1
- Answer Format: The country code required to trigger the commands (sms.apk only checks one country code).

4.6 Stage 2

From Stage 1, we know the required country code and the necessary commands to trigger the malicious behavior. However, even if we send the correct commands with the correct country code, sms.apk will still not exhibit any malicious behavior. In order to maximize the longevity of malware, malicious developers want to prevent analysis. Since the majority of dynamic analysis frameworks are based on emulation, malicious developers integrate anti-analysis techniques to change an app's behavior. If an app senses that the underlying environment is an emulator and not a real phone, it will change its behavior to not exhibit any suspicious behavior. In Stage 2 we will try to identify how sms.apk is checking if it is on an emulator. Then we will modify sms.apk to remove this check and trigger the malicious behavior.

4.6.1 Question 1: (15 points)

The most basic form of emulation detection is when a malicious app leverages a static heuristic. Static heuristics are pre-initialized values that provide information about the underlying environment [9]. Apps running on a system can check these static heuristics by calling Android APIs. For many of the values, the emulator will return values that are inconsistent with what would happen if the app was running on an real device. For example, if the *TelephonyManager.getDeviceId()* API returns all 0's, the device in question is an emulator. This is because this value cannot exist on a physical device.

A list of the possible static heuristics that can be found in sms.apk can be found in [10]. However, the one just mentioned would be a good starting point.

- **sms.apk is leveraging an Android API to identify if the underlying environment is emulated. The return value of the API provides sms.apk with a static heuristic about the emulated enviornment. sms.apk compares the returned value to a hard-coded string, what is the value of this string?**
- Related background sections: 1.4, 2.1, 3, 3.1
- Answer Format: The value of the string that the static heuristic is being compared to. For example, if emulation_check = "01234" your answer would be 01234.

4.6.2 Question 2: (30 points)

The final question is a two-step process. The first step will be to modify sms.apk and remove the environment check so that we can run sms.apk on an emulator. The second step will be sending the commands found in Stage 1 to the emulator and having it exhibit malicious behavior. Upon success, the C&C server will generate the final answers.

- **What are the strings the C&C server provides you with when you dynamically trigger the malicious behavior in sms.apk?**
- Answer format: For each command sent to sms.apk, the C&C server will print out a string. The answer to question 3 will be a list of strings, one for each command. In your report, place each string on a separate line.

4.6.3 Step 1:

- Prerequisites: Read Sections 1.2, it will discuss how to leverage apktool to disassemble an app into byte code. Additionally, you have been provided with a sample apk called *emu-check.apk*. Section 1.2 will walk you how to remove the emulation check in this basic app. If you have no previous experience modifying apks, it's recommended that you start off by removing the emulation check from *emu-check.apk* before working on sms.apk.

- To complete step 1, you will need to modify sms.apk, so that it will trigger its malicious behavior while running on an emulator.

- Note: The modification you are required to make is extremely small, if you find yourself modifying more than a few **characters** then you are going in the wrong direction.

4.6.4 Step 2:

- The final step of Question 3 should be straight forward if you have the correct answers for the previous steps. The first step should be starting up the server using the command **"./start_server"**. Once the server has started, you need to trigger the malicious behavior by sending it commands. If you are successful, the server will provide you with an answer for the respective command (the order does not matter). Copy and paste each answer to your report.

References

[1] Android. App manifest documentation. `https://developer.android.com/guide/topics/manifest/manifest-intro.html`.

[2] R Winsniewski. Apktool: a tool for reverse engineering android apk files. *URL: https://ibotpeaches.github.io/Apktool/(vi sited on 07/27/2016)*, 2012.

[3] skylot. Jadx, 2012. `https://github.com/skylot/jadx.git`.

[4] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[5] Jesus Freke. Smali/baksmali. *URL: http://code. google. com/p/smali.*

[6] Android. Dalvik bytecode. `https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`.

[7] Android. Controlling the emulator from the commandline. `https://developer.android.com/studio/run/emulator-commandline.html#events`.

[8] Siegfried Rasthofer, Irfan Asrar, Stephan Huber, and Eric Bodden. How current android malware seeks to evade automated code analysis. In *IFIP International Conference on Information Security Theory and Practice*, pages 187–202. Springer, 2015.

[9] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[10] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.