Chair of IT Security
Brandenburg University of Technology

Asya Mitseva, M.Sc.
Fabian Mildner, B.Sc.
Prof. Dr.-Ing. Andriy Panchenko

Summer Term 2025

Part V: Mobile Security

Deadline: 2nd July 2025

Cyber Security Lab (Ethical Hacking)
Android Repackaging and Device Rooting

Introduction

Nowadays, mobile devices are an essential part of our daily life. We use them for communication with other people, web browsing and navigation. However, the introduction of smarter features for mobile devices has also opened up more potential for security issues. One of the major threats rises due to the extendability of the core system by apps. Although the apps obtained from the app store *should* be trustworthy thanks to certain review processes, it can still happen that a malicious application makes it into circulation. This risk is even higher when a user decides to install applications from third parties or directly from the Internet.

The *repackaging attack* is a common type of attack on Android devices. Attackers modify a popular app downloaded from the app store, reverse engineer it, add some malicious functionality, and then upload the modified app to the app store or third party stores. Users can easily be fooled to install the modified app, because it is hard to differentiate between the modified app and the original one. Once the modified app is installed, the integrated malicious code can launch attacks, usually in the background. For example, in March 2011, it was found that *DroidDream Trojan* had been embedded into more than 50 apps on the official Android app store and had infected many users by further abusing vulnerabilities in the Android operating system.

Mounting a repackaging attack is easy, as the binary code of Android apps can be easily reverse engineered due to a lack of protection on plain binaries. In this laboratory exercise, we will get familiar with the reverse engineering of Android applications in order to first steal secret information from the application and then inject malicious functionality.

Disclaimer: The mobile applications provided and/or developed during this laboratory exercise are not intended to be installed on physical devices. To avoid possible damage, please use an

emulated Android device only for this laboratory exercise!

Preparation of the Experimental Setup

Installing Necessary Software

For the purposes of this laboratory exercise, you will need to develop for and emulate a virtual Android device. The most convenient way to achieve this is the usage of *Android Studio* [1]. Android Studio is a fully featured IDE for Android applications, including features such as syntax highlighting, refactoring tools, and integrated Android emulators. To install it on Ubuntu, you can issue the following commands:

```
$ sudo apt-get install libcanberra-gtk-module android-sdk
$ sudo apt-get install android-studio
```

In case you do not find `android-studio` in your Linux distribution's package repository, or are not running on Linux at all, you may also gain the Installer for Android Studio from the official Website: https://developer.android.com/studio.

Upon launching Android Studio, you are able to create a project – we suggest a blank project for now. Once you are within the project, navigate to the Device Manager – you should be able to note that an Android Virtual Device (AVD) has already been prepared for you. Otherwise, you can create one yourself. You are able to launch your emulator from the device manager, and play around with the virtual device. It is also possible to launch the emulator from the command-line, as it is installed under `/Android/Sdk/emulator/emulator`. For example, you can run `./emulator -list-avds` to get a list of virtual devices, and `./emulator -avd <devices name>` to run one of them.

When the virtual device is running, you are also able to interact with it through the *Android Debug Bridge (adb)*. The command-line utility `adb` may already be available to you at this point (if not, it may be found in a Linux package called `android-tools` or similar, depending on your distribution). Furthermore, there are other useful tools that will be of help in solving the subsequent tasks. This includes the reverse-engineering tools *apktool* [3] and *JADX* [4], as well as the static code analyser *Androwarn* [2]. You are also free to do additional research for other tools that may help you solve the tasks of this task sheet.

Getting Familiar with the APK Format

Android applications are delivered in APK format. This format is a zip container for the source files and all other resources needed to run the application. If you unzip an application, you can observe the following structure:

The included XML file contains the configuration of the application, e.g., package name, target SDK version, granted permissions, and other related parameters. The `classes.dex` file contains the actual source code of the application. However, it is compiled in a special form of Java byte code optimized for the Dalvik virtual machine / Android runtime environment. To convert the code into a human friendly format, you need to decompile the application instead of just unzipping it. To do this, you can use `apktool` [3] with the decompilation option `d`:

```
$ apktool d application.apk -o apktool_out
```

The resources will be extracted to `apktool_out` in a human readable form. The decoded source code is in a smali format, which is equivalent to the disassembly of the machine byte code. Although the smali code is easier to read, it still makes the manual reverse engineering more difficult. Therefore, one needs an automated analyzer.

Androwarn [2] is a static code analyzer written in Python. It is primarily used to check existing applications for malicious behavior. However, due to its user friendly usage, it is also a good choice for creating a high level summary of the application, e.g., imported libraries, existing classes. Androwarn can be installed, e.g., using pip [5]:

```
$ pip install androwarn
```

Then, to generate an HTML-based report providing a high level summary of the application, you can use the following command:

```
$ androwarn -i Application.apk -v 3 -r html
```

This kind of report might be helpful to get more familiar with the application.

Beside the decompilation of the application by using `apktool`, you can also decompile the application to the original Java code. To this end, you can use the tool `Jadx` [4]. Beside a basic command line interface, it offers a graphical user interface as well. However, you could also use the code generated by `Jadx` only and import the resulting reversed code into a new Android Studio project. To generate the underlying source code from the application, you can use the following command:

```
$ jadx -d Application Application.apk
```

Please note that the usage of `apktool` is recommended when it comes to changing the reversed code. Recompiling the decompiled and possibly modified source code obtained by `Jadx` may result in additional trouble.

Tasks

Task 1: Installing the Application

Before we start hacking, it is good to get familiar with the provided app and our toolset, first. The goal of this task is to install the application on an Android Virtual Device - to this end, you will have to use the Android Debug Bridge (ADB). Launch the app and see what it is about. We will try to crack it in the next task.

Task 2: Finding out the Secret String

As you surely noticed, the application provided with this laboratory exercise is protected by a simple pass-sphere protection. The objective of this task is to obtain the secret key necessary to run the application (without brute-forcing it).

For the sake of simplicity, the functionality of the application is very limited. It is going to provide a simple output of the text message: “This is the correct secret”, in case the input secret is correct. Thus, if you have obtained the correct key, you will get this message after entering the key. This simple functionality also makes it easier to analyze the application binary with the tools you have available.

Hint: The secret can be obtained in different ways; however, static analysis is the most common approach. Some measures have been taken to protect the key, however - during your analysis, you may want to pay attention to keywords related to cryptography.

Task 3: Removing the Root Protection

In this task, we will consider what happens when running the application in root mode, i.e. in a virtual device which is rooted. First of all, you shall obtain a rooted Android Virtual Device, and test the application on it. What different behavior are you observing when the app runs on a rooted device? There are multiple ways to obtain a rooted Android virtual device. The simplest way might be to add a new device with API Version 24, which is rooted by default.

Next, you should work to remove the rooted device protection mechanism that is part of the application. To this end, you need to go through the following steps, which illustrate the Android *repackaging attack*:

1. Decompress and decompile the application, e.g., by using `apktool`.
2. Identify the sections responsible for the altered program behavior on rooted devices.
3. Edit the resulting code in order to accomplish the task requirements. Note that, while the code generated by `Jadx` can be helpful in analyzing the program functionality, it is safer to modify and use the smali-code generated by `apktool` for the repackaging.
4. Repackage the application and re-install it on your emulated device to test the modified behavior.

As part of this task, you will also need to inform yourself about the repackaging attack and the security mechanisms implemented to avoid it. In particular, think about the use of digital signatures and developer keys, and how you can make the device install your app despite these facts.

Hint: There are multiple ways to obtain a rooted Android Virtual Device. Unfortunately, you may note that the default device created by Android Studio can *not* be rooted without the usage of special scripts made for this purpose - this is due to certain security precautions relating to the Services of the Google Play Store. You can evade this issue by creating a new virtual device and making sure that, under “Services”, you select “Google APIs” instead of “Google Play Store”. It can be noted that devices with this setting and with API Version 24 already come rooted by default!

Task 4: Adding Malicious Functionality to the Application

After we have gotten familiar with the repackaging attack in the previous task, we now want to extend the application with some actual malicious functionality that can be a real threat to a user installing it. To achieve this, we will inject malicious code into the smali code of the application before repackaging it. One way to do this would be to modify an existing smali file by adding some malicious logic to it. However, another, often more convenient approach is to add a completely new component to the existing application. This new component will be independent from the existing application and, thus, will not affect the existing behavior of the program. Since each component can be placed in a separate smali file, this approach allows us to add additional, malicious functionality without having to modify the existing smali files.

There are four types of components in Android applications: *activity*, *service*, *broadcast receiver*, and *contentprovider*. The first two components are the most commonly used by applications, while the second two are rather uncommon. You can add any of these components in the provided application, but the more important task for an attacker is to find a way to trigger the malicious code without being identified by the users. Although there are possible solutions for all components, the easiest one is the broadcast receiver, which can be triggered by certain signals (broadcasts) sent by the system. For example, just after the device has finished booting, it sends a `BOOTCOMPLETED` signal to all apps that care about it. You can write a broadcast receiver that listens to this broadcast, allowing the malicious code to be triggered automatically each time the device reboots.

The goal of this task is to extend the application in such a way that it deletes all contacts in the address book of the user upon booting. To achieve this, you are tasked to add a broadcast receiver which listens to the `BOOTCOMPLETED` signal. When receiving the message, the application should iterate over the address book and delete each contact one-by-one. Please note that such functionality may require the application to ask for additional user permissions. Research how to inject and register the new functionality into the existing application and grant the extended privileges to the application. Afterwards, test the modified/extended application by adding dummy contacts to your address book and starting the app. Note that a restart of the Android Virtual Device will be required to notice results, since the malicious functionality proposed here is triggered by the `BOOTCOMPLETED` signal.

Hints:

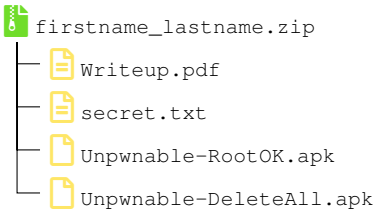
1. Since you do not need to edit existing smali files, it is possible to develop the broadcast receiver in a high-level language using Android Studio. Then, the obtained binary file can be reversed to a smali file, which is easy to inject.
2. Do not forget to register your broadcast receiver to your modified application by editing the file `Manifest.xml`.
3. Keep in mind that a change of the source code may also require a change of permission settings.

Preparation of Your Submission

Your submission for this lab should contain the final versions of any and all source code you have written and files you have generated, along with a detailed writeup about the tasks you solved. Your writeup should be given in an appropriate format, containing detailed descriptions of how you solved each task, what theoretical insights you gained, what observations you have made as well as images / screenshots / listings to aid the understanding of your approach. Your writeup should be submitted in PDF format.

As always, detailed descriptions as to how you solved the tasks are important and need to be submitted in your `Writeup.pdf`. For this lab, you should also submit a file containing the secret from the first task, along with the modified applications you created for the third task (`Unpwnable-RootOK.apk`) and the fourth task (`Unpwnable-DeleteAll.apk`).

Please submit your solutions in a packed zip file named after your full name, with the following structure and minimum required contents:



Preparation for the Q&A Session

Prepare yourself for a Q&A session of up to 40 minutes. During the Q&A session, you should be prepared to *explain and demonstrate* how you solved the tasks. Take special care to explain how your solution works and why you took the approach you took. Note that simply presenting a piece of code without any explanation will not yield any points! Last but not least, you should be ready to answer spontaneous (theoretical) questions asked by the reviewer in relation to the topics of this lab.

References

[1] *Android Studio*. URL: https://developer.android.com/studio (cit. on p. 3).

[2] *Androwarn - Yet another static code analyzer for malicious Android applications*. URL: https://github.com/maaaaz/androwarn (cit. on pp. 3, 4).

[3] *apktool - A tool for reverse engineering Android apk files*. URL: https://ibotpeaches.github.io/Apktool/ (cit. on pp. 3, 4).

[4] *JADX - Dex to Java decompiler*. URL: https://github.com/skylot/jadx (cit. on pp. 3, 5).

[5] *pip - package installer for python*. URL: https://pypi.org/project/pip/ (cit. on p. 4).