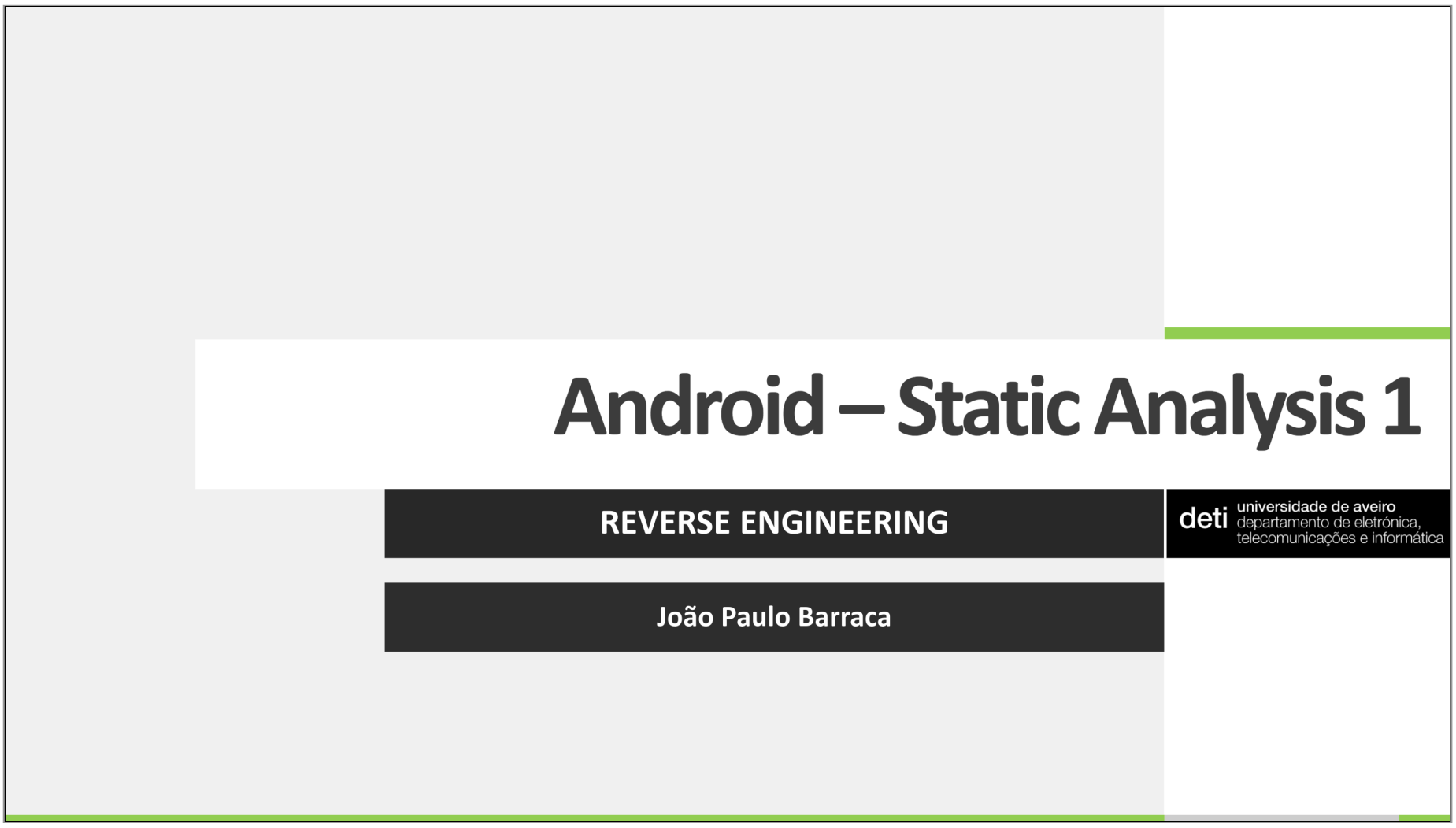
Android Static Analysis

Lecture Notes

Analyzing Android applications through static analysis methods as a first approach towards reversing.

Previous Next



Download here

Practical Tasks

Setup

Several tools should be installed along the execution of the guide. Most will be available on a `kali` system through `apt`. You do not need to install everything before the guide. You can install applications as you go.

The complete setup would be:

```
$ sudo apt install unzip apktool jadx androguard binutils quark-engine google-android-cmdline-tools-12.0-installer google-android-platform-tools-installer go
```

The command line tools can be setup as:

```
$ sudo sdkmanager "platforms;android-29"
$ sudo sdkmanager "system-images;android-27;default;arm64-v8a"
$ avdmanager create avd -n Android8 -k "system-images;android-27;default;arm64-v8a" -c 1000M
```

If you have Virtualization Acceleration (Direct or nested), and a Intel/AMD CPU, you can use `x86_64` instead of `arm64-v8a`, as it will be much faster.

After this step an emulator should be available. Check it with:

```
$ avdmanager list
Available Android Virtual Devices:
    Name: Android8
    Path: /home/user/.android/avd/Android10.avd
  Target: Default Android System Image
          Based on: Android 8.0 ("Oreo") Tag/ABI: default/arm64-v8a
  Sdcard: 512 MB
```

An emulator can be started with:

```
$ emulator @Android8 -qemu -machine virt
```

Exercise 1

Compile the project Hello.zip using Android Studio. You may run it in an emulated environment or on your own cell phone (it is safe, trust me :D) to see what it does (not much, trust me again :D). After you compile it, in the project folder there will be a sub folder named `build` and inside it an `APK`. This is the application package for us to use.

You may unzip the `APK` to access the contents directly:

```
$ unzip app.apk
```

In order to recover the Java code from the `classes.dex` file, the jadx tool will help. Because the `APK` is not obfuscated, the capability to recover the source code will be very good. Compare the source code you have in Android Studio with the decompiled code from jadx.

Q: How good it matches the original? Can you build a new project and recompile it?

If you examine the files, you will notice that some are not readable. One example is the `AndroidManifest.xml`. This file is actually compressed and encoded to a binary format. You may use apktool in alternative to `unzip`, which will further process the `APK` and decode the `AndroidManifest.xml` and many other files.

It will also produce a reasonable number of `smali` files (more on this later).

```
$ apktool d app-debug.apk
```

After the application is decompiled, and you have access to all resources, you may recompile it back, and repack it. If everything works as expected, you will end up with a new application, which is a clone of the first one. There are multiple approaches for this, depending on the tools you use and if there are changes. The simplest approach is to use the Android Studio or the Command line version installed in the setup section, creating a new project and compiling it back to an `APK`.

Q: Can you do it?

The second approach may be required if the Java code produced is not valid. Because Android Studio cannot compile the code, it will not produce the final `APK`. Fortunately, other tools allow repacking and resigning the application. This process may be useful if the changes do not include the code, or if the code is modified in a different way.

To repack you may execute:

```
$ apktool b extracted_apk -o app-release-mod.apk --use-aapt2
```

To sign it you may execute:

```
$ java -jar uber-apk-signer.jar --apks app-name.apk
```

And then the new Android application is ready to be installed directly into the phone or virtual environment.

```
$ adb install app.apk
```

The NahamCon 2021 Andra challenge

Using this knowledge, I'm sure you can solve the NahamCon2021 Andra challenge. Probably jadx and Androguard are enough. There is no need to install the application, as static analysis is all that is required.

Get the challenge file here, analyze it, and get the `flag{***}`.

Exercise 2

In the previous exercise you compiled and installed an application into the Android system. As we discussed, the application will be optimized on installation (`ART`) or on the first run (`DALVIK`), and will produce an optimized object. In both cases, the application will be stored at `/data/app/*`

You can navigate the Android system using `adb` from the Android Platform Tools. This will work better on a emulated environment or on a rooted phone.

```
$ adb shell
```

You can get the installation path of the application by listing all applications, and then querying the path for the matching class.

```
$ pm list packages | grep hello
```

Should return `pt.ua.deti.hello`.

```
$ pm path pt.ua.deti.hello
```

Check how the application is installed and how files are created. This object is in the `/data/app` folder. Look for the `odex` files. Objdump is a multipurpose tool which can show some information about `odex` files:

```
$ objdump -T base.odex

base.odex:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
00001000 g    DO .rodata        0000e000 oatdata
0000effc g    DO .rodata        00000004 oatlastword
0000f000 g    DO .dex   00000000 oatdex
003c340d g    DO .dex   00000004 oatdexlastword
```

You can also decompile the `DEX` code inside the `OAT`, but the process is lossy and you cannot go back to Java The first thing to do is to obtain the `boot.oat` file, which contains the symbols required for `base.odex`.

```
$ adb pull /system/framework/arm/boot.oat boot.oat
```

Then the file can be decompiled:

```
$ java -jar baksmali.jar -x -c boot.oat -d framework base.odex -o out
```

Interestingly if you use `readelf`, you may notice that, while having different extensions, both `boot.oat` and `base.odex` are of the same format.

Q: What is this format?

Exercise 3

So… our Hello application is leaking a password to the log. How could that happen? Check the log and get the password:

```
$ adb logcat
```

You will need to restart the application.

This presents a security risk and should be fixed. You can change it like in the previous situation, however the obfuscation may introduce weird behaviors in the decompiler.

The actual fix would be to change the source code, but as we are Reversing Android applications, let's do it *the wrong way* by modifying the `smali` code!!. For this, follow these instructions:

Unpack the `APK` and convert the Java bytecode to `smali`

```
$ apktool d app-release.apk  --use-aapt2
```

This will create a folder named `app-release`. Inside this folder you can find the `smali`representation of the application bytecode.

Remember the class where the leaking instruction was? `smali` structure follows the rules set by Java, with the name of the file matching the name of the public class.

Find the class, find de instruction, and just replace it with a `nop`.

Then you can repack the application using:

```
$ apktool b app-release --use-aapt2
```

And then sign it:

```
$ java -jar uber-apk-signer-1.2.1.jar --apks app-release/dist/app-release.apk
```

Install it to the phone and test it again:

```
$ adb uninstall pt.ua.deti.hello
```

```
$ adb install app-release/dist/app-release-aligned-debugSigned.apk
```

Q: Did it worked? Is the application still leaking the password?

This explores how easy it is to change an application and create clones. Changes to resources (images, themes) is trivial. Adding new code may not be that difficult, and then we end up with lots of clones and fake applications in the App store.

Exercise 4

DO NOT INSTAL THIS APPLICATION ON A REAL PHONE

There is the suspicion that an application (`ThaiCamera.apk`) is sending Premium SMS without user consent, or without notifying the user of what it is doing. Please check it and write your findings. You can find the application here

Describe the entry points to the application, if there is any suspicious activity, and how it may take place. It is recommended that you actually do a small writeup of your findings as it helps to structure your knowledge. For this task, you need to analyse the behavior of the code. It's not a simple grep/find. You need to look for suspicious behavior.

We can discuss your findings during the next class.

DO NOT INSTAL THIS APPLICATION ON A REAL PHONE

TIP: look which Android methods are used to send SMS.

Exercise 5

DO NOT INSTAL THIS APPLICATION ON A REAL PHONE

The goal of this exercise is to apply our `DEX` reverse engineering skills to find a vulnerability in an Android application (`FotaProvider.apk`). This example is a little more complex and will introduce us to reversing across different components of the application.

Assume that you are auditing a set of phones for security issues prior to allowing them onto your enterprise network. You are going through the applications that come pre-installed. For this pre-installed application, you are concerned that there may be a vulnerability that allows it to run arbitrary commands.

A command can be executed with `Runtime.exec()`, `Processbuilder()` and `system()` (in native code)

To start this exercise, follow these steps:

- Download the application:FotaProvider.apk
- Start jadx and open `FotaProvider.apk`
- Use the application manifest to identify interesting components and the bytecode to analyze the code, intent filters, exported services, or services with intent filters, and userids.
- Find a code path that allows other applications or code on the phone to run arbitrary commands as a privileged user (through an intent maybe?)
- Search for `getRuntime()`, `ProcessBuilder()` or `system()`.

Suggested questions for you to answer:

- How do we execute arbitrary commands?
- How could the app receive remote commands?
- What would be a starting point for the classes?
- Put it all together and how does it work?

The solution is here Android App Reverse Engineering - Exercise #3 Solution, but only break the glass if you are really stuck.

Further exploration

Check this repository as it contains an interesting list of Android Malware Samples. You may be able to get some information from the easy ones.

WARNING: Never install these samples into a real phone!

Tools

- apktool: Performs static analysis of the apk, allowing extensive inspection
- Uber `APK` signer: Signs apk back and helps with several other related processes
- smali/backsmali: converts to and from smali
- dex2jar: converts the dex file into a JAR. Then any Java decompiler can open it
- Java Decompiler (JD): Decompiles Java applications
- lief: Python library to interact with many types of binaries, including dex and oat
- jadx: integrated decompiler from DEX to Java code (whenever possible)
- androguard: swiss knife to manipulate android files
- Android Studio: official development environment for Android
- objdump: Dumps binary objects, including their sections, metadata and code
- readelf: Displays information about `ELF` objects
- Quark: Taint Analysis and Flow Analysis of Android applications

Credits

Some examples are modeled after the Android Workshop from Amanda Rousseau (malwareunicorn) and Maddie Stone (maddiestone). Follow them on Twitter as they post great content.

PREVIOUS
File Types

Last updated on 23 Feb 2024