Title: Penetration Testing for Android apps

Author: Amir Ghayour

Date: Summer 2020

Abstract:

With the fast growth of smartphones and their use in daily activities like banking, communication, and entertainment, securing mobile applications has become essential. This research looks at penetration testing methods for Android applications to find weaknesses that could put user data and application integrity at risk. The study examines several techniques, including reverse engineering with tools like apktool, dex2jar, and jd-gui to extract source codes. Dynamic code instrumentation using Frida to modify runtime processes and change application behavior without having to rebuild or resign the app. And man-in-the-middle (MITM) attacks are shown using Android Device Monitor and Burp Suite to intercept and analyze memory and network communications to extract sensitive information. Some of these techniques are demonstrated using Capture-the-Flag (CTF) applications. The findings highlight the need for strong security measures in Android applications and offer insights into tools and techniques that developers and security professionals can use to reduce risks.

1. Introduction:

Local Data Storage

The protection of sensitive data, such as user credentials and private information, is crucial to mobile security. If an app uses operating system APIs such as local storage or inter-process communication (IPC) improperly, the app might expose sensitive data to other apps running on the same device. It may also unintentionally leak data to cloud storage, backups, or the keyboard cache. Additionally, mobile devices can be lost or stolen more easily compared to other types of devices, so it's more likely an individual can gain physical access to the device, making it easier to retrieve the data.

Fragmentation is a problem we deal with especially on Android devices. Not every Android device offers hardware-backed secure storage, and many devices are running outdated versions of Android. For an app to be supported on these out-of-date devices, it would have to be created using an older version of Android's API which may lack important security features. For maximum security, the best choice is to create apps with the current API version even though that excludes some users.

Communication with Trusted Endpoints

Mobile devices regularly connect to a variety of networks, including public Wi-Fi networks shared with other (potentially malicious) clients. This creates opportunities for a wide variety of network-based attacks ranging from simple to complicated and old to new. It's crucial to maintain the confidentiality and integrity of information exchanged between the mobile app and remote service endpoints. As a basic requirement, mobile apps must set up a secure, encrypted channel for network communication using the TLS protocol with appropriate settings.

Authentication and Authorization

In most cases, sending users to log in to a remote service is an integral part of the overall mobile app architecture. Even though most of the authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. Unlike web apps, mobile apps often store long-time session tokens that are unlocked with user-to-device authentication features such as fingerprint scanning. While this allows for quicker login and better user experience (nobody likes to enter complex passwords), it also introduces additional complexity and room for error.

Mobile app architectures also increasingly incorporate authorization frameworks (such as OAuth2) that delegate authentication to a separate service or outsource the authentication process to an authentication provider. Using OAuth2 allows the client-side authentication logic to be outsourced to other apps on the same device (e.g. the system browser). Security testers must know the advantages and disadvantages of different possible authorization frameworks and architectures.

Interaction with the Mobile Platform

Mobile operating system architectures differ from classical desktop architectures in important ways. For example, all mobile operating systems implement app permission systems that regulate access to specific APIs. They also offer more (Android) or less rich (iOS) inter-process communication (IPC) facilities that enable apps to exchange signals and data. These platform-specific features come with their own set of pitfalls. For example, if IPC APIs are misused, sensitive data or functionality might be unintentionally exposed to other apps running on the device.

Code Quality and Exploit Mitigation

Traditional injection and memory management issues aren't often seen in mobile apps due to the smaller attack surface. Mobile apps mostly interact with the trusted backend service and the UI, so even if many buffer overflow vulnerabilities exist in the app, those vulnerabilities usually don't open up any useful attack vectors. The same applies to browser exploits such as cross-site scripting (XSS allows attackers to inject scripts into web pages) that are very prevalent in web apps. However, there are always exceptions. XSS is theoretically possible on mobile in some cases, but it's very rare to see XSS issues that an individual can exploit.

This protection from injection and memory management issues doesn't mean that app developers can get away with writing sloppy code. Following security best practices results in hardened (secure) release builds that are resilient against tampering. Free security features offered by compilers and mobile SDKs help increase security and mitigate attacks.

Anti-Tampering and Anti-Reversing

There are three things you should never bring up in polite conversations: religion, politics, and code obfuscation. Many security experts dismiss client-side protections outright. However, software protection controls are widely used in the mobile app world, so security testers need ways to deal with these protections. We believe there's a benefit to client-side protections if they are employed with a clear purpose and realistic expectations in mind and aren't used to replace security controls.

2. Basic Concepts:

.APK: Android Application Package, the mobile app installation file format.

.JAR: This format is a file package containing a large number of Java classes, metadata and resources. This is made based on zip format.

.class: a compiled java file that contains byte-code and is run by JVM.

Resources: These files are additional files and static contents that the application codes use.

Rooting: Rooting is the process by which users of Android devices can attain privileged control (known as root access) over various subsystems of the device, usually smartphones.

ADM: Android Device Monitor, explained in Tools.

adb: Android Debug Bridge (adb) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps. adb provides access to a Unix shell that you can use to run a variety of commands on a device.

EVABS: An open source CTF Android application that is intentionally vulnerable so as to act as a learning platform for Android application security beginners.

Manifest: Every app project must have an AndroidManifest.xml file (with precisely that name) at the root of the project source set. The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.

Activity: The Activity class is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle.

Intent: An Intent is a messaging object you can use to request an action from another app component. A link between two activities, transporting data between two activities, Service request for an activity, etc.

ContentProvider: Content providers are one of the primary building blocks of Android applications, providing content to applications. A content provider is only required if you need to share data between multiple applications.

Service: A Service is an application component that can perform long-running operations in the background. It does not provide a user interface. Once started, a service might continue running for some time, even after the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service can handle network transactions, play music, perform file I/O, or interact with a content provider, all from the background.

smail: smali/baksmali is an assembler/disassembler for the dex format used by dalvik, Android's Java VM implementation. The most common human readable format is known as *Smali* but A Dex file contains code which is ultimately executed by the Android Runtime.

Fridump: an open source memory dumping tool, primarily aimed to penetration testers and developers. Fridump is using the Frida framework to dump accessible memory addresses from any platform supported. It can be used from a Windows, Linux or Mac OS X system to dump the memory of an iOS, Android or Windows application.

3. Tools:

3.1 apktool:

For opening and decoding apk files.
“ A tool for reverse engineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications.

Features:

- Disassembling resources to nearly original form (including resources.arsc, classes.dex, 9.png. and XMLs)
- Rebuilding decoded resources back to binary APK/JAR
- Organizing and handling APKs that depend on framework resources
- Smali Debugging (Removed in 2.1.0 in favor of IdeaSmali)
- Helping with repetitive tasks “

Use below command for decompiling apk file:
apktool d appname.apk

You can prevent decoding sources and resources by -s and -r options.



Figure 3.1: decoded apk package by apktool

3.2 dex2jar:

Converting dex and apk to jar by using this command:
dex2jar appname.apk

3.3 jd-gui:

A graphical tool to open the source codes of .class files.
This tool is used to open the converted files by dex2jar.

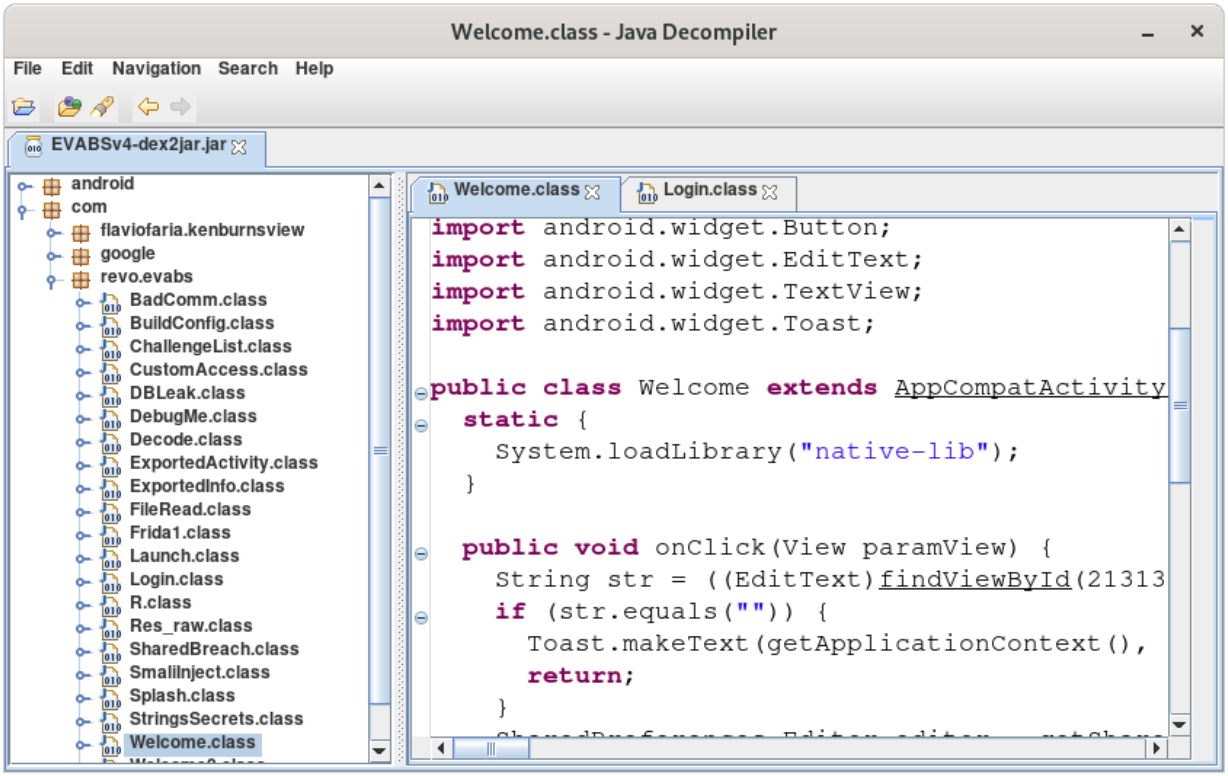


Figure 3.2: Jd_GUI environment

3.4 Burp Suite:

This tool is used for monitor, control and edit the data sent to and received from the internet.
“ Burp Suite is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application’s attack surface, through to finding and exploiting security vulnerabilities. “

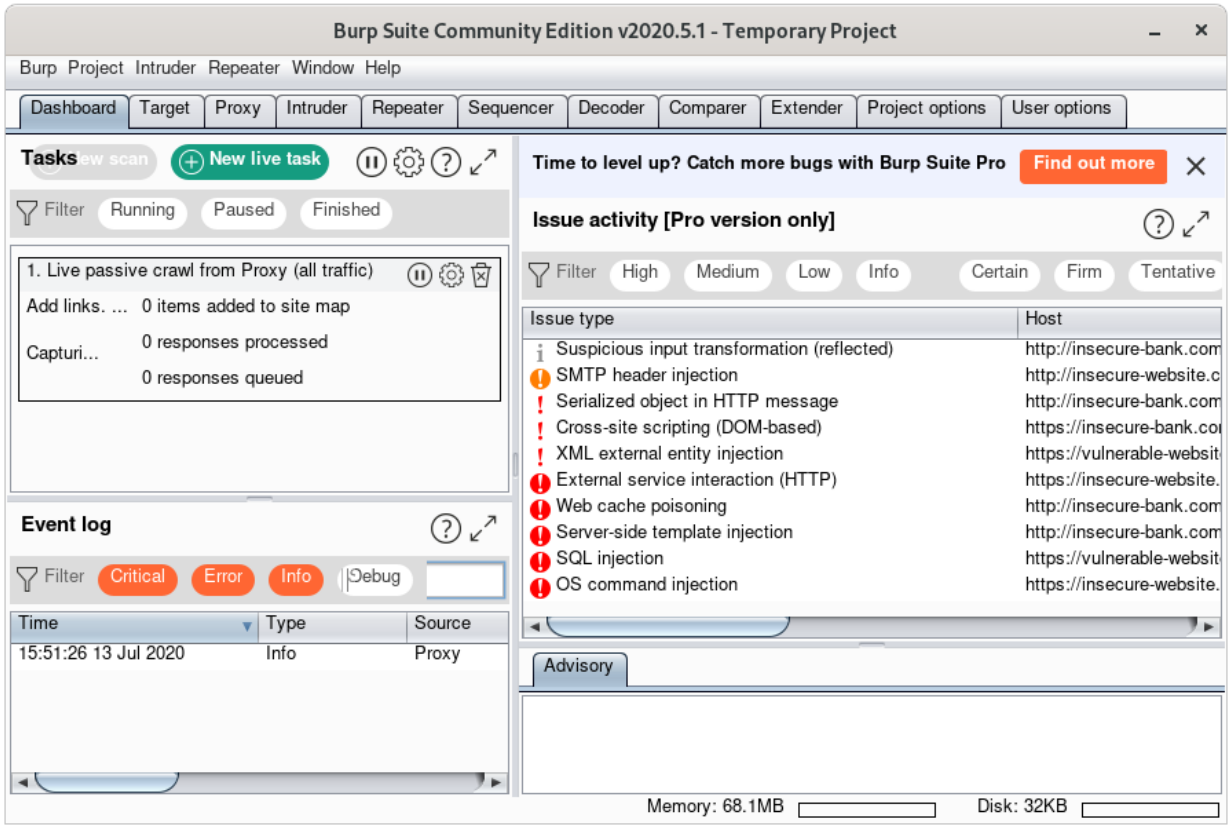


Figure 3.3: Burp Suite environment

For controlling the requests, go to the Proxy tab, click on Options. In the proxy listeners section, click on add and define an ip and a port. (127.0.0.1 for local(same) device). (enabling “all interface” to monitor android emulator). Then set the proxy in your program. And then go to the intercept tab and turn on the intercept if you want to edit the requests before sending them to the internet. Use the history tab for monitoring the sent and received data.
If you want to send the request again, you can send it to the repeater tab and after editing, send it to the internet again.
If you want to make it automatic, like a dictionary attack, you can use the Intruder tab.
Go to this address for more information: https://www.hacker101.com/playlists/burp_suite

3.5 A device with root access:

We need a phone or android emulator with root privilege to be able to do penetration testing more effectively and efficiently. The emulator installed along with Android Studio is suitable for this task. After opening Android Studio, create an android virtual machine from AVD Manager and run it.

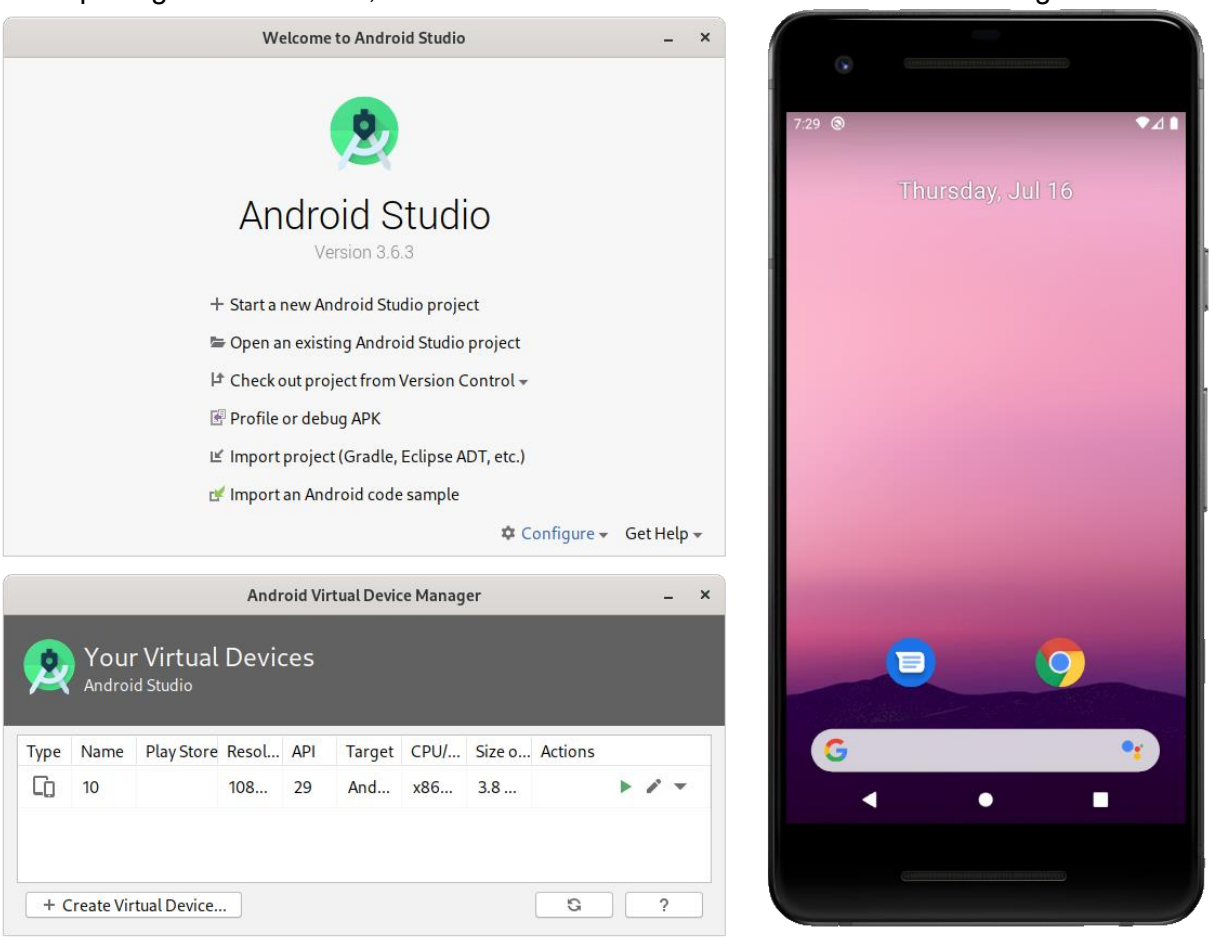


Figure 3.4: Android Studio on the top, AVD Manager at bottom, Android Emulator on the right

3.6 Android Device Monitor:

“ Android Device Monitor is a stand-alone tool that provides a graphical user interface for several Android application debugging and analysis tools. “

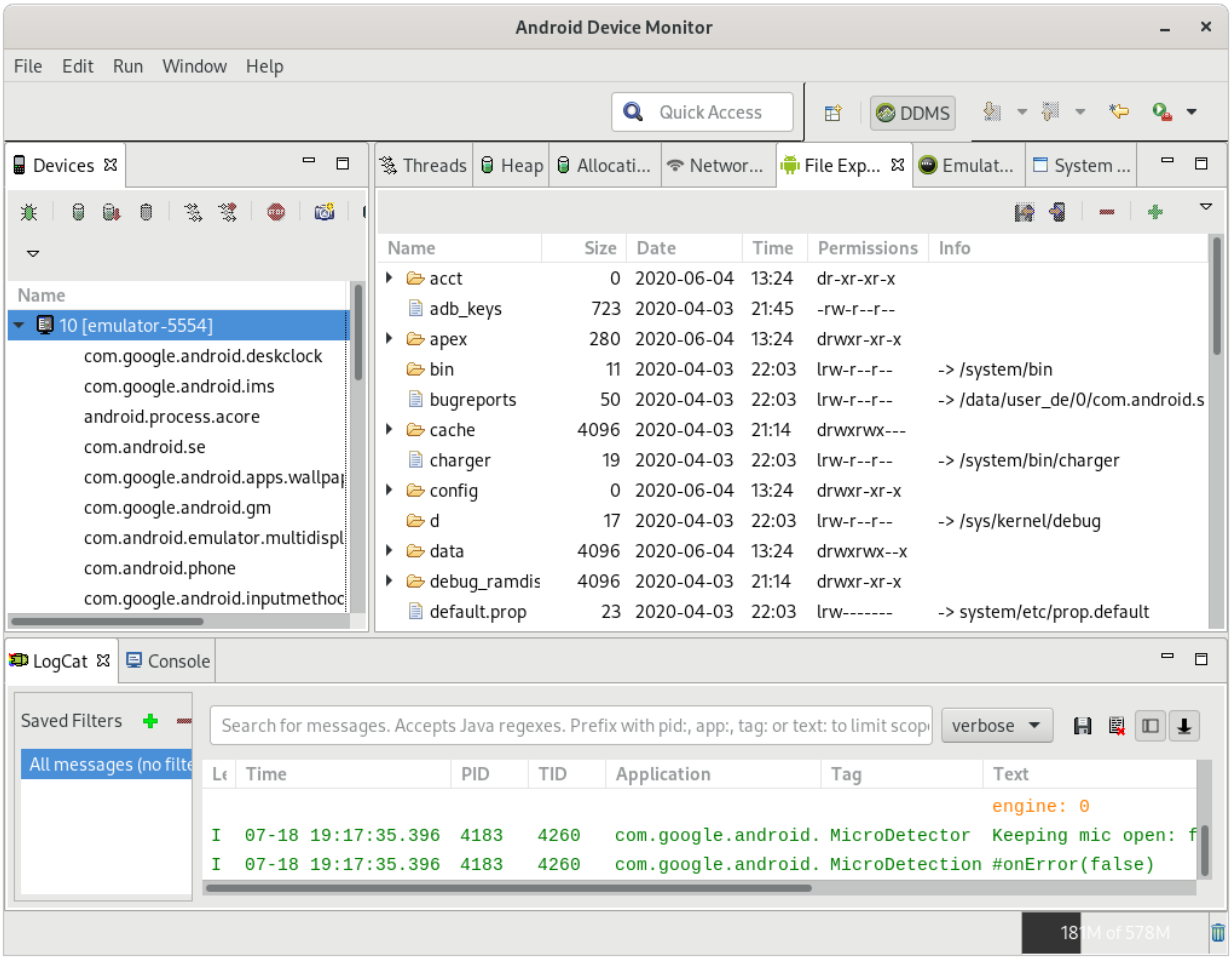


Figure 3.5: Android Device Monitor environment

Use *./monitor* command to run this application from Sdk/tools folder. After running an android emulator or connecting an android device, select the device name from Devices list.
Use below command to go to root mode:
./adb root

3.7 Frida:

“ It's a dynamic code instrumentation toolkit. It lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD, and QNX. Frida also provides you with some simple tools built on top of the Frida API. These can be used as-is, tweaked to your needs, or serve as examples of how to use the API. “
“ Frida can be used to hook into the running process of the application and modify the code on the fly without requiring any re-launching or re-packaging. “
We can use this tool to disable fingerprint verification, run applications that don’t allow you to run them on rooted devices or bypass a login page, etc.

“ Frida consists of two components, viz. a client and a server which communicate with each other over two ports(TCP) 27042 and 27043. “

We can use below command to install Frida client:

```
pip install frida-tools        # CLI tools
pip install frida              # Python bindings
npm install frida              # Node.js bindings
```

Use these links if you get into problems with installing Frida:
https://frida.re/docs/installation/
https://frida.re/docs/troubleshooting/

For installing the server part of Frida, “ you will need to root your device in case you haven’t done so already. It is technically also possible to use Frida without rooting your device, for example by repackaging the app to include frida-gadget, or using a debugger to accomplish the same. But, for this introduction we’re going to focus on the simplest case: a rooted device. “

You will also need the adb tool from the Android SDK that you can make sure adb can see your device:

```
adb devices -l
```

First off, download the latest frida-server for Android from https://github.com/frida/frida/releases and uncompress it.

```
unxz frida-server.xz
```

Switch to root by this command:

```
adb root
```

Move *frida-server* to an arbitrary location on your android device:

```
adb push frida-server /data/local/tmp/
```

After changing the permission:

```
adb shell "chmod 755 /data/local/tmp/frida-server"
```

we execute it (make sure to run with root privilege):

```
adb shell "/data/local/tmp/frida-server &"
```

Or we can execute it on the terminal inside the rooted device:

```
root@android:/ $ su
root@android:/ # /data/local/tmp/frida-server
```

“ Now, on your desktop it’s time to make sure the basics are working. Run: “

```
frida-ps -U
```

“ This should give you a process list along the lines of: “

```
 PID   NAME
1590   com.facebook.katana
13194  com.facebook.katana:providers
12326  com.facebook.orca
13282  com.twitter.android
…
```

Now we can start our penetration testing.

4. Penetration Testing:

In this section, we talk about some vulnerabilities and the way to find them and perform penetration testing.

4.1 Log sensitive information

Programmers must not log sensitive information. We can access this information by Android Device Monitor, LogCat section. For example on EVABS on level1, we get the below flag after clicking on “ log the key " in the LogCat section. (Figure 4.1)
EVABS{logging_info_never_safe}

4.2 Saving sensitive information in the source code and resources

Any data like the keys that is stored in source code and applications files, can be extracted. We can decode apk files with apktool and by analyzing AndroidManifest.xml, XMLs, smali files, etc, we can extract sensitive information.
Also with dex2jar and jd-gui, we can decompile and analyze the source code of apk files. (Figure 4.2)

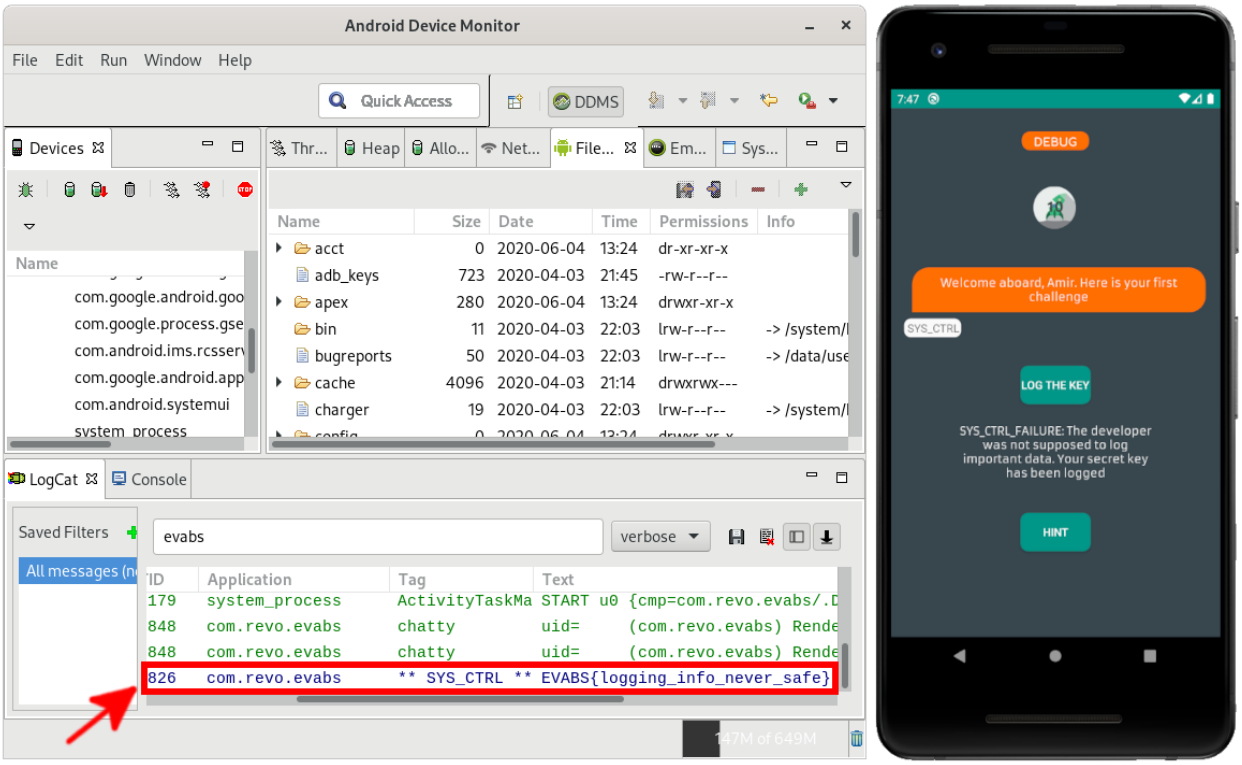


Figure 4.1: logged sensitive information in EVABS

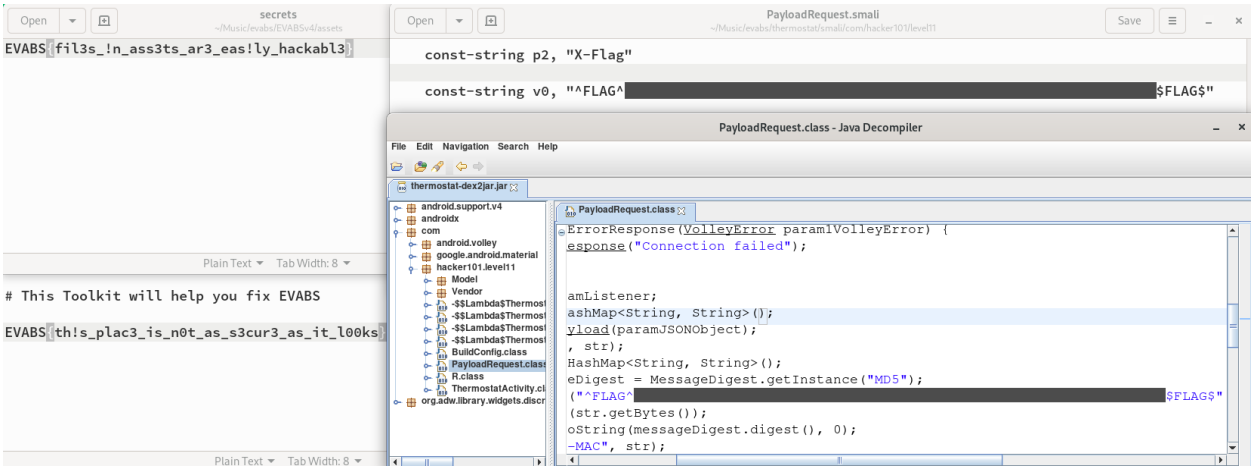


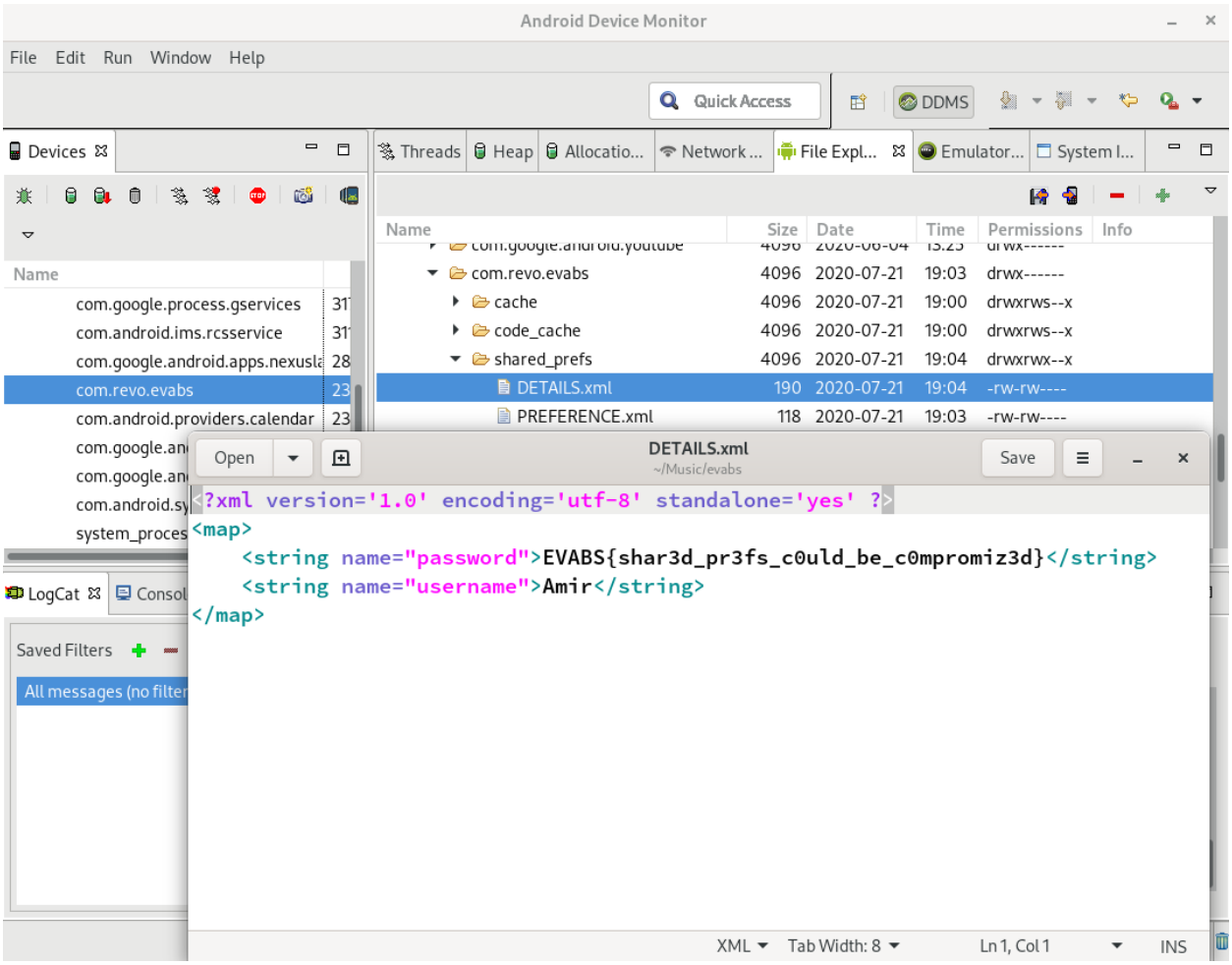
Figure 4.2: Stored sensitive information in EVABS and Thermostat applications

4.3 Smali Injection

When you depackage apk files with apktool, dex files get converted to smali files (readable format). We can now change smali codes, XMLs, etc, and pack it and sign it again by apktool. So we can crack and get extra access from applications this way.

4.4 /data/data/appname directory

We can access some data stored in this directory by having root access. (Figure 4.3)



Stored information in /data/data/com.revo.evabs directory

4.5 Local Databases

Generally, all the stored data in the device can be accessed by the owner of the device. So try to avoid storing sensitive information and if you need to, use encryption, root detection, etc, to make it difficult to access these data.
“ Protecting the app against the owner of the device is almost impossible without a server-side and is only a matter of time. “

4.6 Memory

Data stored in Memory is also accessible and extracted like any other storage on the device. For example, we can extract the memory used by a process by using Android Device Monitor or Fridump.

4.7 Connection Between the App and the Server

We can control the connection between the app and the server by Burp Suite and monitor and edit all the requests sent to and received from the server. So we can catch the static keys and non-encrypted data.

4.8 Exposed Components

“ In almost every Android application, developers expose activities without sufficient protections. Exposing activities can lead to various attacks. For example, an attacker or a malicious app installed on the same device, can call those exposed activities to invoke internal pages of the application. Calling internal pages puts the application at risk of phishing by manipulating users to enter details in the phishing app, as well as exposing a user to secret pages, such as admin panels or pages which should have been visible to paid/pro users only. “
All the activities must be declared in the Manifest. Most of the time, activities present a page of the program.
“ A user with root permissions (i.e. ADB) on the device can invoke any activity, simply by sending an intent to the desired activity. For example: “
adb shell am start -n <package_name>/<activity_name>

Assume when we run a program, we get a login page but we can just call the home page activity this way and bypass the login page. “ Of course, if a session/token is required in order to pull the data from the server, and every action is authorized against that session/token, then exposing internal pages is not a great risk. But as we know, not all apps have the server-side to protect them and some of them are standalone, client-side only, apps. In that case, there is little that can be done to protect them. But there are measures that can be taken to make them harder to attack. Encrypting data, obfuscating the code and implementing root-detection code in the app might make the time required to bypass it all not worthwhile for the potential attacker. But protecting the app against the owner of the device is almost impossible without a server-side and is only a matter of time. ”

“ The greater risk, however, is exposing the app to other applications/malware installed on the device. To allow apps to communicate with each other, the developers can explicitly expose the activity, by setting the android:exported attribute to true. This means that any app installed on the device can send an intent to launch the exported activity. “
The risk depends on how the app reacts to the received intent. A malicious app can send intent constantly to crash the application (market competition, phishing, etc) or even worse changing the behavior of the program. For example, the malicious app can send intent to a vulnerable banking app to make a transaction without even opening the application.
“ But most apps are exposed where least expected. Activities are not only exposed when explicitly setting the android:exported=”true” attribute. Although the default configuration is that the activity is not exported, when setting an to an activity, it automatically becomes exported. “

“ Intent filter specifies the types of intents that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of its parent component — what an activity or service can do and what types of broadcasts a receiver can handle. It opens the component to receiving intents of the advertised type, while filtering out those that are not meaningful for the component. “



Figure 4.4: configured intent-filter in manifest

“ As mentioned, the configured for the broadcast receiver in figure 4.4 sets the receiver as exported, which means that it can be called by any app installed on the device. “
“ There are different ways to exploit exposed components such as activities, content providers and broadcast receivers. As always, it’s in the hands of the developers to prevent such attacks.”

For protection, “ first, we configure the components in a secured manner. unless explicitly required – disable external exposure of any components by specifying android:exported=”false” in the application manifest. If a component has an intent-filter, by doing so, the component is no longer exposed to other apps.
Another way to protect components is to require permissions when calling them, by declaring the android:permission in the manifest, under the desired component. The name of a permission that the sending party must have in order to send an intent to the receiving app. This will be enforced by Android; if the attribute is not set, the receiver is not protected by a permission. (Figure 4.5) “

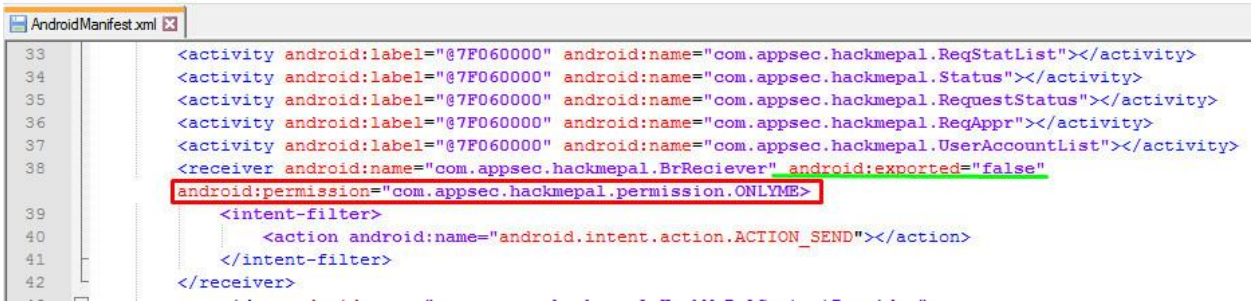


Figure 4.5: Preventing from attacks by declaring permissions and disabling exported

“ After we configured the manifest correctly, we still need to review our code. To avoid crashes, do not assume to have received the required arguments; verify before you assign them. “

4.9 dynamic code instrumentation

We do penetration testing on some apps by Frida and change the behavior of those apps.
1. Bypassing the login page for Sieve app. This is a password manager application.
As you see on figure 4.6, we can log in with the valid password and we have saved some passwords in the app.

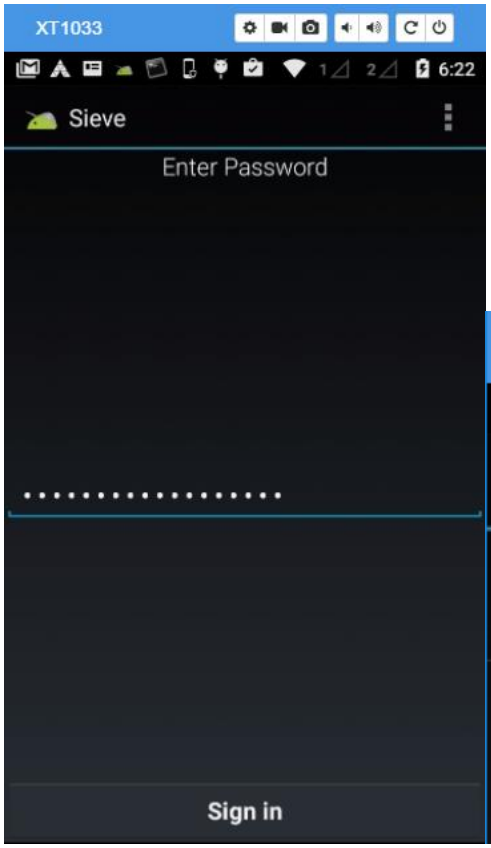


Figure 4.6: Sieve login page

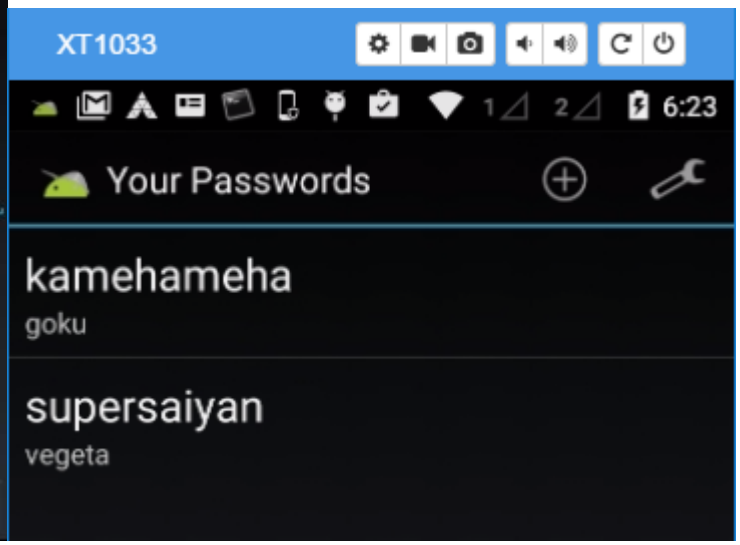


Figure 4.7: Sieve home page

Open your terminal and execute this command to get the name of the process for Sieve:
frida-ps -aU | grep -i "sieve"

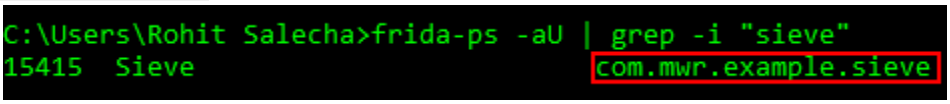


Figure 4.8: the name of the running process for Sieve

Before bypassing the login page, let's decompile the program and take a look at the source code. As mentioned, first we convert apk to jar by dex2jar then open it by jd-gui.
As you can see on figure 4.9, we found a method with the name, checkKeyResult which if we change the value of the boolean parameter to true, the application executes loginSuccessful method and if it is false, the app executes loginFailed method.

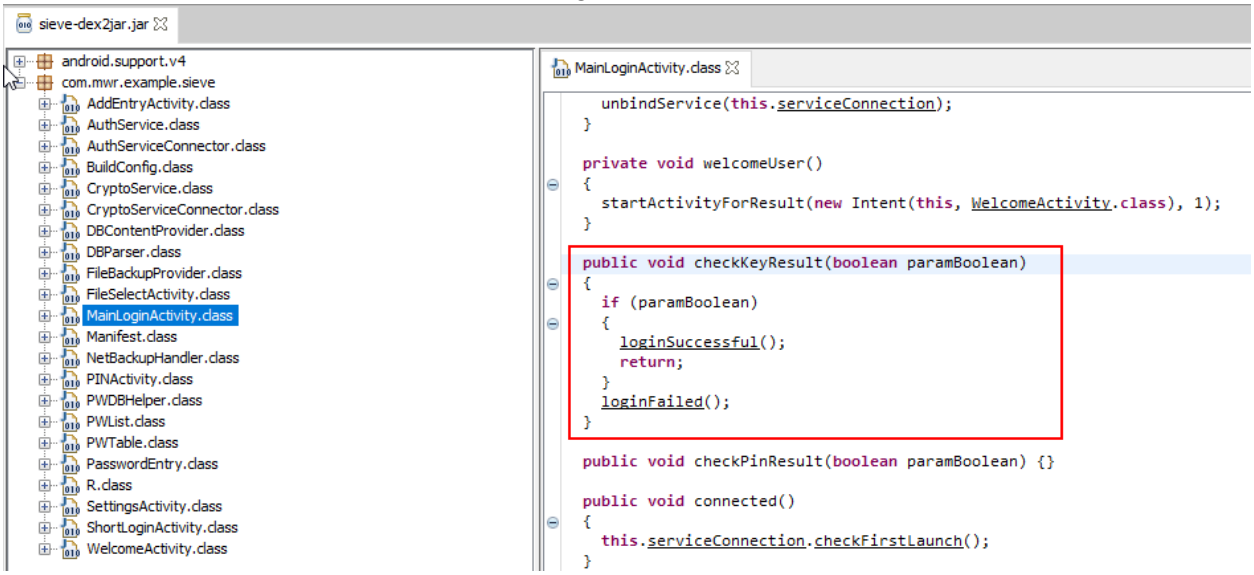


Figure 4.9: checkKeyResult method in Sieve app

Frida supports various languages like python, C, .Net and Swift. We will work with python:

```
import frida, sys

def on_message(message, data):
    if message['type'] == 'send':
        print("[*] {0}".format(message['payload']))
    else:
        print(message)

jscode = """
Java.perform(function () {
    //Obtain referrence of the Activity currently running
    var MainActivity = Java.use('com.mwr.example.sieve.MainLoginActivity');
            //Obtain reference of the function whcih needs to be called
    MainActivity.checkKeyResult.implementation = function (b) {
        send('checkKeyResult');
            //Calling the function and passing the boolean parameter as true
        this.checkKeyResult(true);

        console.log('Done:');
```

```
    };
});
"""

process = frida.get_usb_device().attach('com.mwr.example.sieve')
script = process.create_script(jscode)
script.on('message', on_message)
script.load()
sys.stdin.read()
```

We don’t change the method, only Frida controls the input and return of checkKeyResult method by changing the boolean parameter to true.
On figure 4.10, the lines 25 and 26 are used to attach the code to the process that we found. And the code that is going to be run in the process by Frida is written in JavaScript (lines 10 to 22).

Run the program and execute the python code. The python code will wait until checkKeyResult gets called. (Figure 4.11)
When you press Sign in, checkKeyResult gets called and Frida injects our javascript code and changes the boolean parameter to true. So we can bypass the login page. (Figure 4.12)

```
1  import frida, sys
2
3  def on_message(message, data):
4      if message['type'] == 'send':
5          print("[*] {0}".format(message['payload']))
6      else:
7          print(message)
8
9  jscode = """
10 Java.perform(function () {
11     //Obtain referrence of the Activity currently running
12     var MainActivity = Java.use('com.mwr.example.sieve.MainLoginActivity');
13     //Obtain reference of the function whcih needs to be called
14     MainActivity.checkKeyResult.implementation = function (b) {
15         send('checkKeyResult');
16     //Calling the function and passing the boolean parameter as true
17         this.checkKeyResult(true);
18
19         console.log('Done:');
20     };
21 });
22 """
23
24 process = frida.get_usb_device().attach('com.mwr.example.sieve')
25 script = process.create_script(jscode)
26 script.on('message', on_message)
27 script.load()
28 sys.stdin.read()
29
```

Figure 4.10: Python and JavaScript codes for bypassing Sieve login page

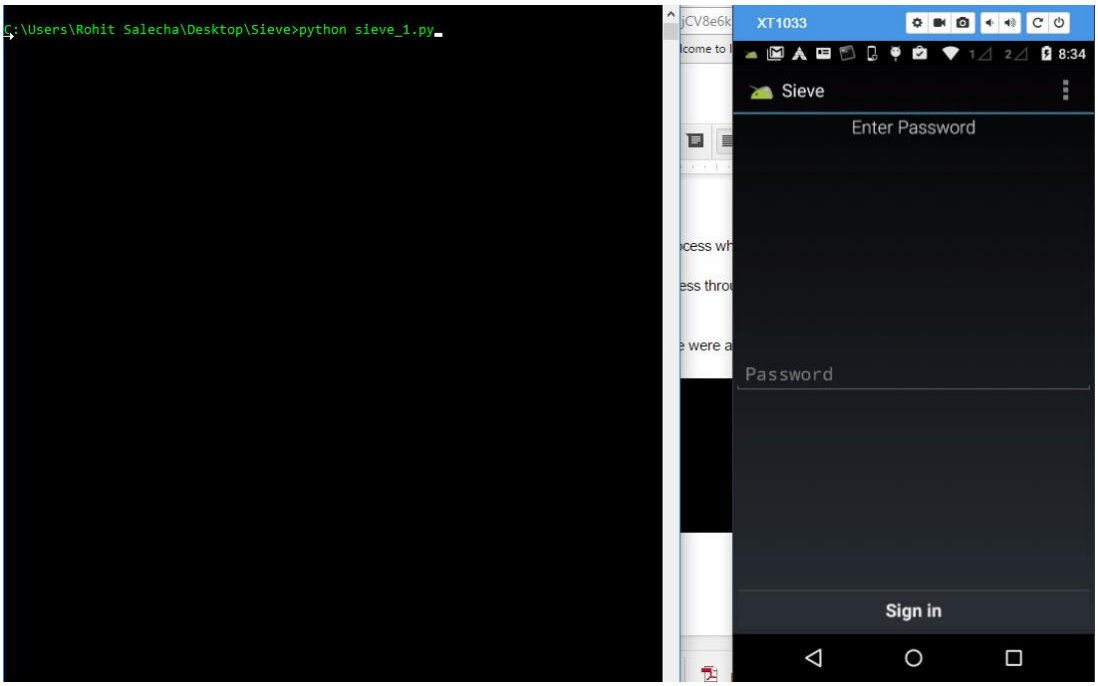


Figure 4.11: The python script is waiting on Sieve login page

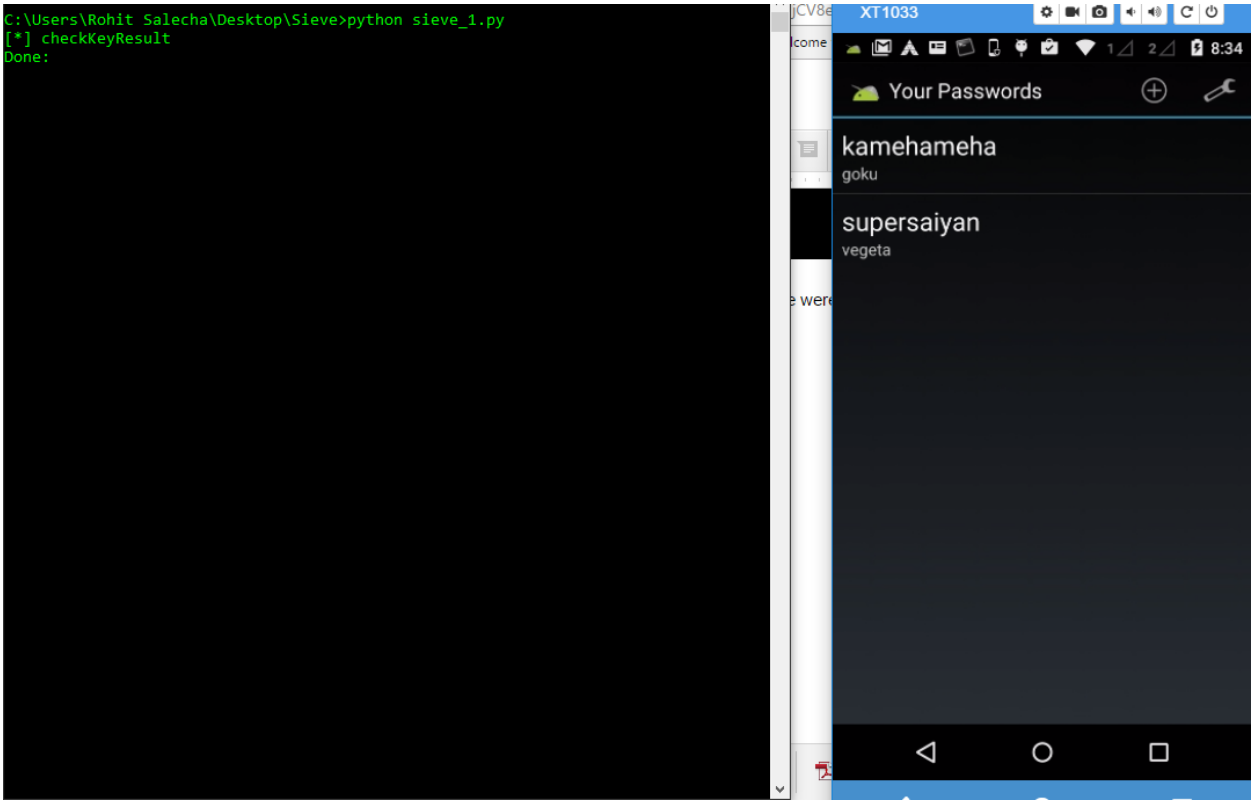


Figure 4.12: bypassed Sieve login page by Frida

```
import frida, sys

def on_message(message, data):
    if message['type'] == 'send':
        print("[*] {0}".format(message['payload']))
    else:
        print(message)

jscode = """
Java.perform(function () {
    var ShortLoginActivity = Java.use('com.mwr.example.sieve.ShortLoginActivity');
    ShortLoginActivity.submit.implementation = function(v){
        var service=this.serviceConnection.value
        for(var i=1225; i<1240; i++)
            {
                var result = service.checkPin(i+"");
                send(i + ": " + result);
            }
        console.log('Done:');
    };
});
"""

process = frida.get_usb_device().attach('com.mwr.example.sieve')
script = process.create_script(jscode)
script.on('message', on_message)
script.load()
sys.stdin.read()
```

Figure 4.13: Brute Forcing Pin in Sieve

2. Brute forcing PIN in Sieve

Sieve asks for login every time that we go back to the background and open the app again. So we can brute force it by Frida. (Figure 4.13)

3. Getting flag out of EVABSv4

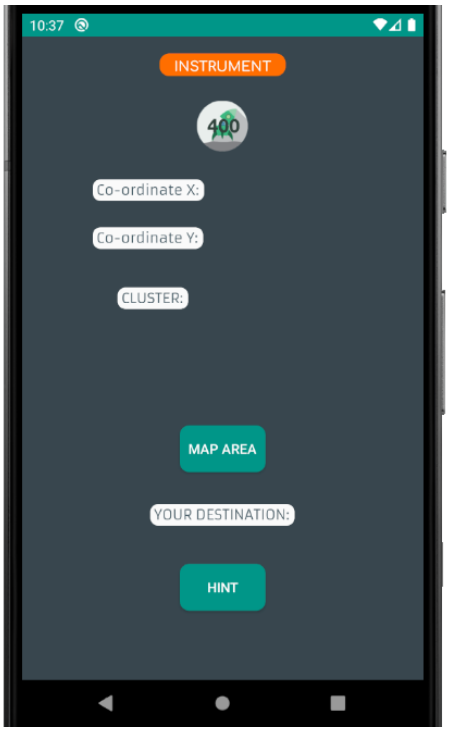


Figure 4.19: before clicking on MAP AREA in EVABS

As you see in Figure 4.20, by clicking on MAP AREA we get Co-ordinates Not Found and 25, 2, 50 on the top of the page. There is also no flag in the log of ADM.

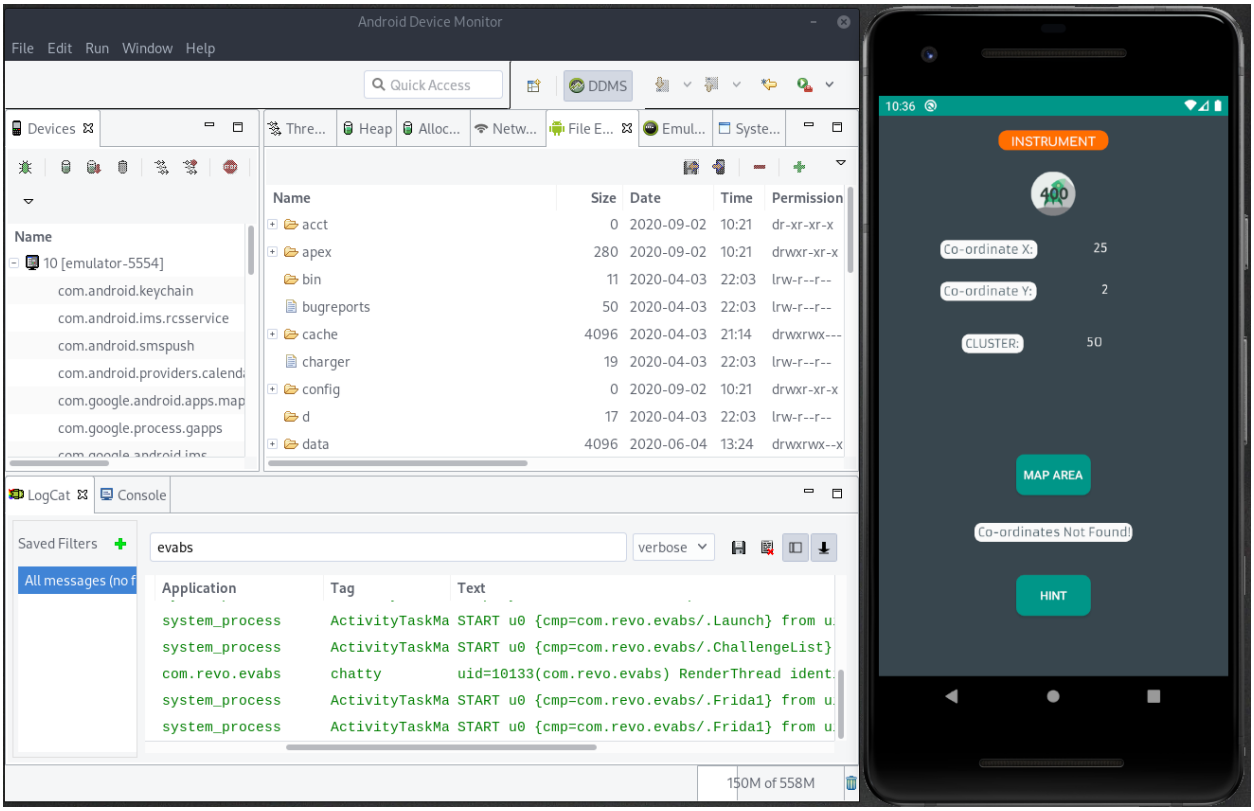


Figure 4.20: After clicking on MAP AREA in EVABS

Now let’s decompile the apk and take a look at the source codes.
As you see in the Figure 4.21, the source code for this section is in the Frida1 class. And there is a condition on the third box that if x > i + 150, flag is shown in the log of ADM otherwise show Co-ordinates Not Found message. We can see x = a*b in the second box and i is a random number between 0 to 69. By looking at the first box, we see that a is 25, b is 2. So we need to change one of these variables to make the multiply bigger that the if condition.

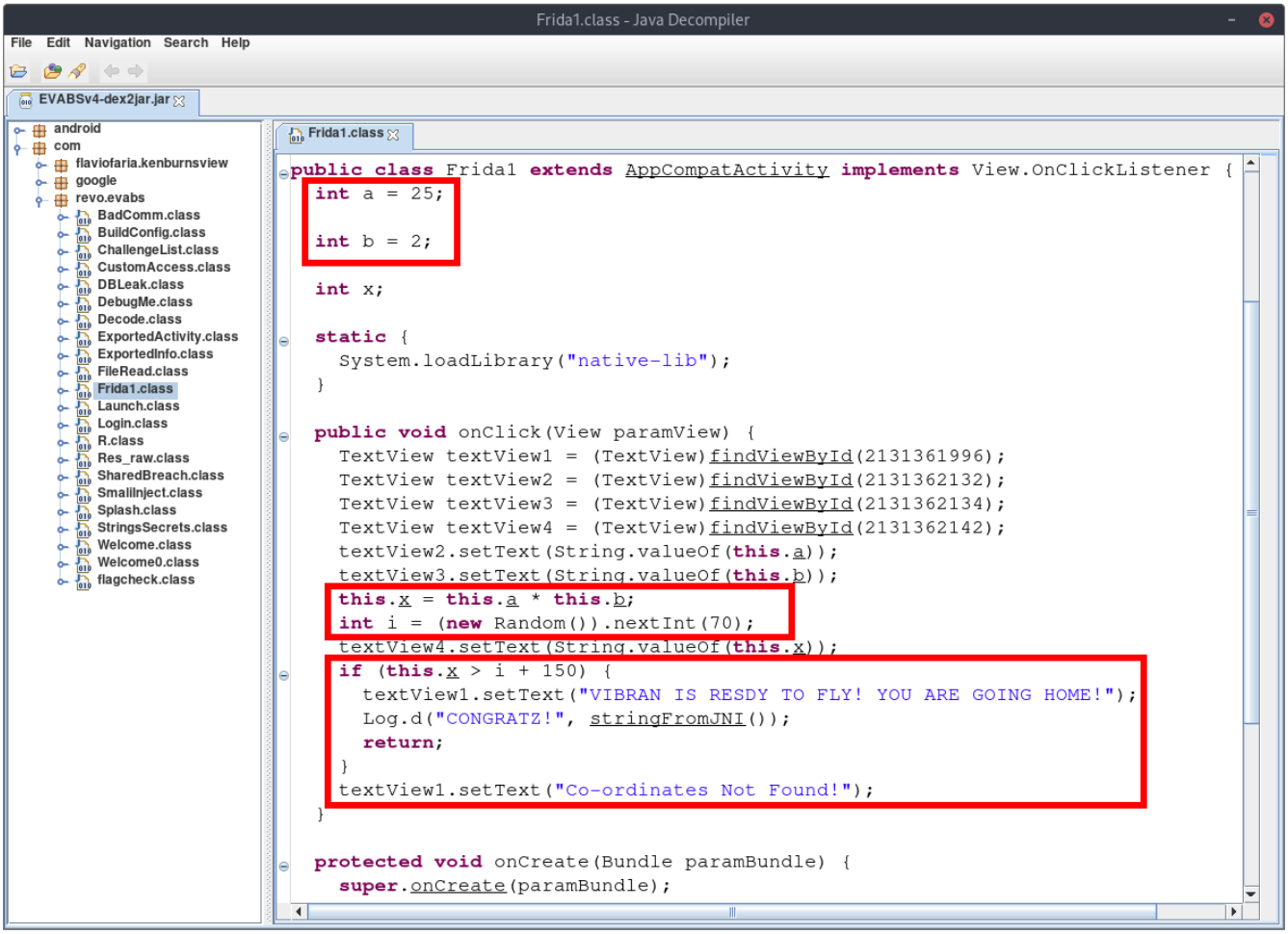


Figure 4.21: decompiled code in EVABS

Now we take a look at Python and JavaScript code.
The python code is like before, we only have changed the name of the process.
On the first line of JavaScript we define the Frida1 class and the next line we say if onClick called, it shows a message and changes to 200 and then calls the original onClick.
By running the python code and clicking on MAP AREA, as you see in figure 4.23, the value of a is 200 and the multiply of a and b is 400. So we get the flag in the log section in ADM.

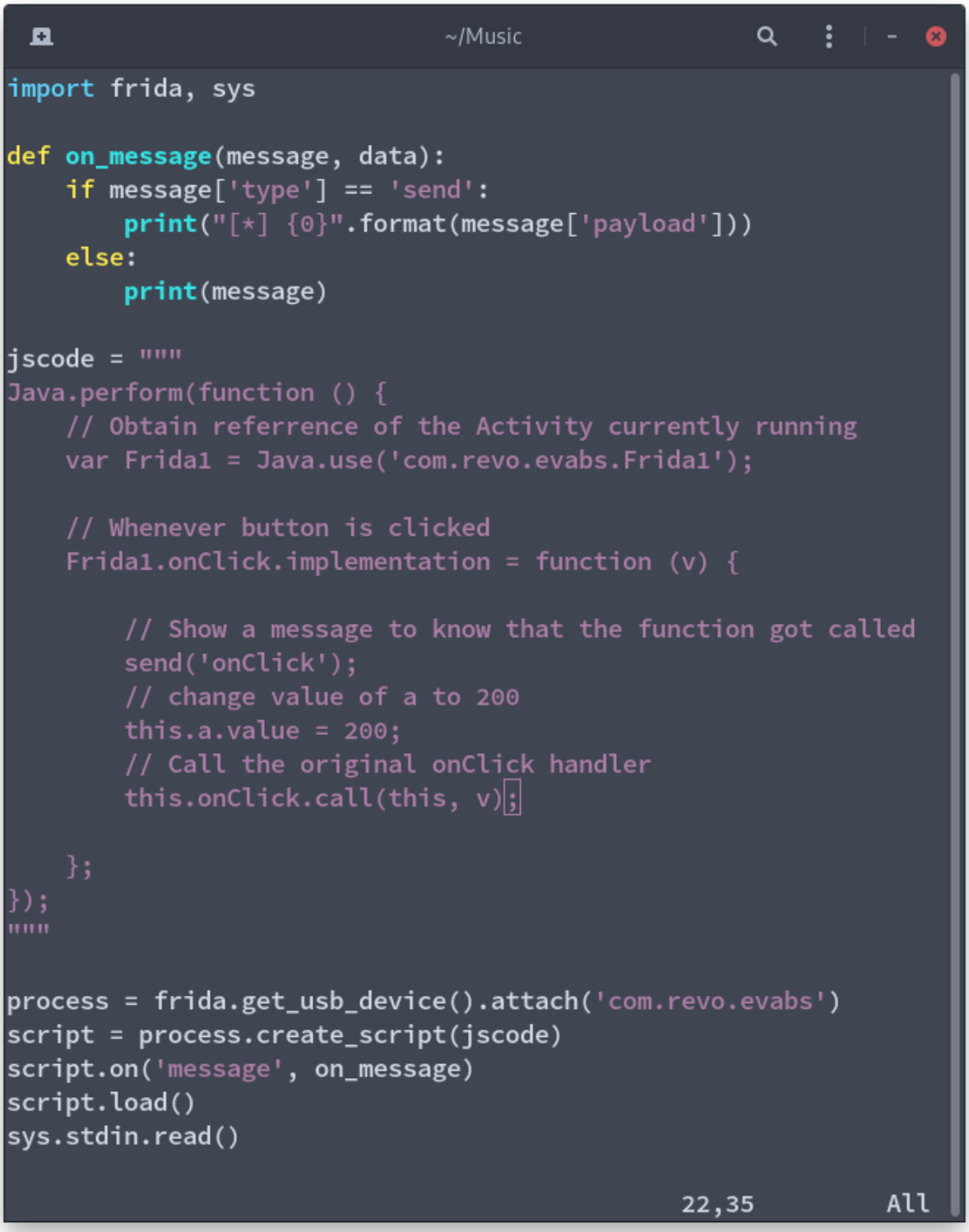


Figure 4.22: Python and JavaScript codes for EVABS app

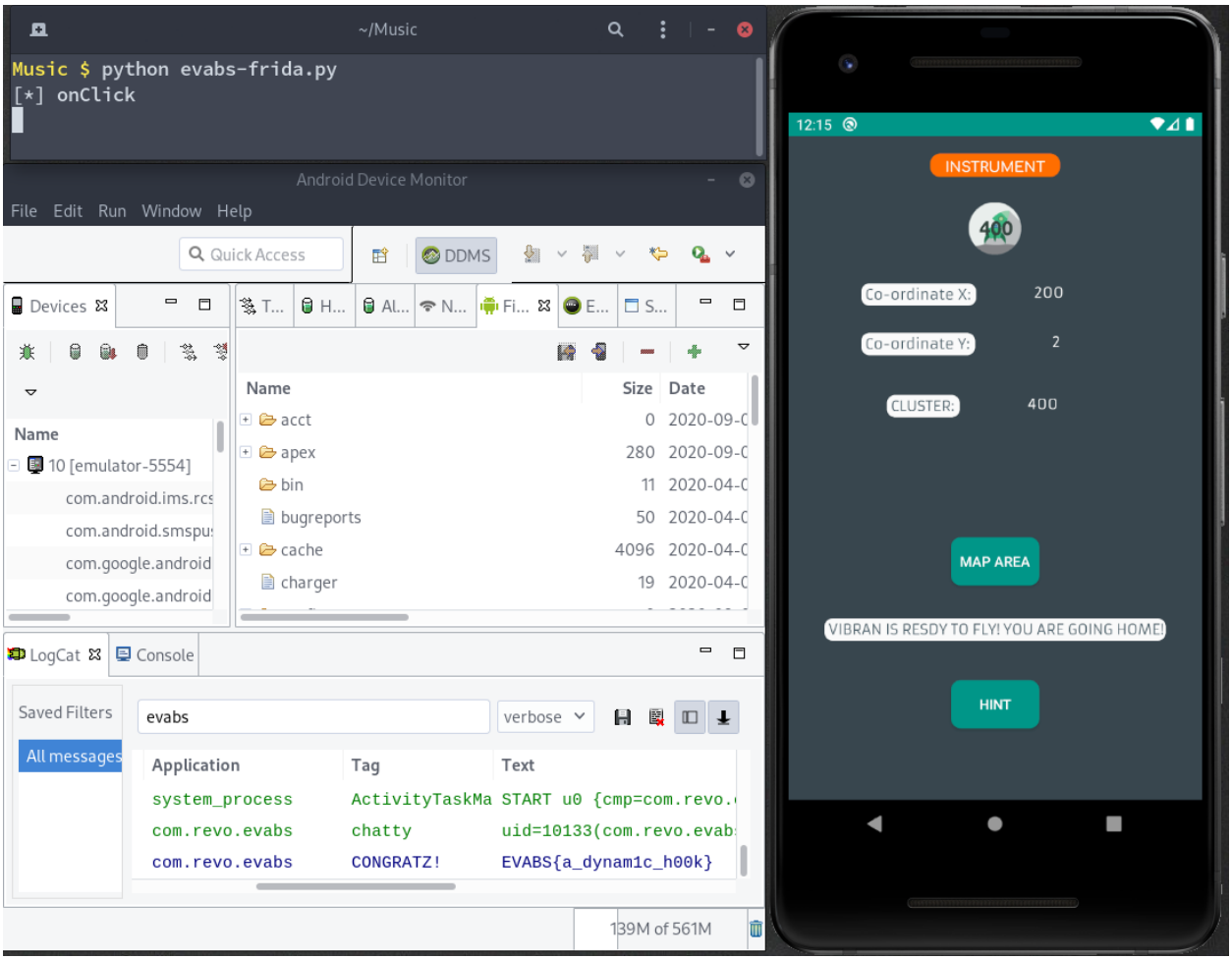


Figure 4.23: received flag in EVABS by Frida

References:

1. https://www.hacker101.com/
2. https://github.com/abhi-r3v0/EVABS
3. https://mobile-security.gitbook.io/mobile-security-testing-guide/overview/0x03-overview
4. https://www.hacker101.com/playlists/burp_suite
5. https://www.notsosecure.com/pentesting-android-apps-using-frida/
6. https://www.linkedin.com/pulse/hacking-android-apps-through-exposed-components-tal-melamed
7. https://frida.re/docs/home/
8. https://frida.re/docs/installation/
9. https://frida.re/docs/android/
10. https://github.com/frida/frida
11. https://developer.android.com/
12. https://developer.android.com/guide/topics/resources/providing-resources
13. https://developer.android.com/studio/command-line/adb
14. https://developer.android.com/guide/topics/manifest/manifest-intro
15. https://developer.android.com/guide/components/activities/intro-activities
16. https://developer.android.com/guide/components/services
17. https://en.wikipedia.org/wiki/JAR_(file_format)
18. https://en.wikipedia.org/wiki/Rooting_(Android)
19. https://stackoverflow.com/questions/8180180/what-is-an-activity-a-context-and-an-intent-in-android
20. https://stackoverflow.com/questions/30837450/what-is-smali-code-android
21. https://github.com/Nightbringer21/fridump
22. https://ibotpeaches.github.io/Apktool/
23. http://java-decompiler.github.io/
24. https://tools.kali.org/web-applications/burpsuite
25. https://youtu.be/LIVIzJEdE9w
26. https://stuff.mit.edu/afs/sipb/project/android/docs/tools/help/monitor.html