Analyzing a variant of the GM Bot Android malware

January 6, 2017 By Pierluigi Paganini

http://securityaffairs.co/wordpress/55125/malware/gm-bot-android-malware.html

My Page

My friends at CyberBlog decided to analyze the GM Bot Android Malware as exercise aiming to receive feedback sand suggestions from the security community.

The sample explored is confirmed as a variant of the GM Bot Android malware – who's source was released publicly in early 2016. The code appears to have been forked by a second author and has additions that target the Danske Bank MobilePay application and the popular Danish Nem ID two factor authentication (2FA) system.

This article shows the process of walking through Static and Dynamic analysis to unlock the packed source code for the malware.

We see how even with basic static analysis a full picture of the intent of the malware can be readily assembled, and with a little debugging we can quickly get to readable source code.

Background

As part of my journey into Cyber Security I thought it would be interesting to see how modern mobile malware operates. I chose the following sample at random based on an article here.

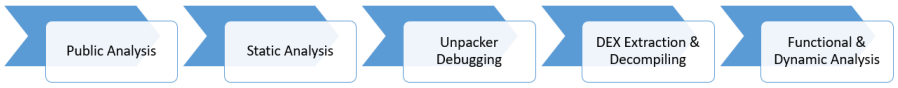
File Details

SHA256: 44ed4bbd5cdc13c28992c16e99a7dc58f5f95463e889dd494a433549754f7863
MD5: da88bdcb3d53d3ce7ab9f81d15be8497

A quick google search for these hashes will lead you to the file used if you would also like to explore this sample.

The article above demonstrates that the analyst has gone from sample to source code, but it is not clear how this is achieved. There are references to suggest that the code has been packed, but again no information on how it was unpacked for analysis.

This post will break down the process I used to analyse this sample, hopefully with enough detail to provide some tips and guidance for others wishing to attempt similar. The process I followed can be logically broken into the following stages:



Analysis Process

1. **Public Analysis** – What can we find out using existing public sources of information? What analysis has already been performed (automated or manual)?
2. **Static Analysis** – What can we determine from the sample without actually running it in an emulated environment?
3. **Packer Debugging** – Assuming the sample is packed (to frustrate analysis), how do we debug the unpacker to understand what is being loaded /run?
4. **DEX Extraction and De-compilation** – Once we have mapped out the function of the unpacker, how do we then recover the main code for the malware and reverse it?
5. **Functional & Dynamic Analysis** – once we have the extracted and reversed code, what do we see and how does this correlate with behavior in a safe emulated environment

Stage 1 – Public Analysis

First off let's see what we can find about this in the public domain. Searching for the file hashes on Virus Total, where we see approximately 50% of AV products have identified it as malicious:

VirusTotal Results

However, we also note that all classify it heuristically as a generic strain of malware – either a Trojan, Dropper, Fake Installer etc. Nothing to suggest it is in fact GM Bot Android, or any specific type of malware. Other than this we don't see much from google with either the SHA256, or MD5 hashes.

The original Security Intelligence article references IBM X-Force research, so this is the next stop – but again nothing immediately obvious with regards to this sample could be located.

A wider search of the internet reveals some history of GM bot, originally built and sold by Ganga Man on dark web forums. Following a dispute the source code for both client APK and C2 server were released publicly. A copy is hosted here on Github and will provide useful for cross referencing with this sample later in the analysis.

https://github.com/gbrindisi/malware/tree/master/android/gmbot

Stage 2 – Static Analysis

First up we are going to unpack the APK file using APK tool. This will unzip the contents, as well as providing a disassembly of the DEX code into Smali:

```
apktool d da88bdcb3d53d3ce7ab9f81d15be8497.apk
```

The results of this can be seen below and the tool has also provided a human readable version of the AndroidManifest.xml file.

```
root@kali:~/Desktop/GMBot/clean_analysis/da88bdcb3d53d3ce7ab9f81d15be8497# ls -lrt
total 68
-rw-r--r--  1 root root  5828 Dec  2 02:44 AndroidManifest.xml
drwxr-xr-x 22 root root  4096 Dec  2 02:44 res
drwxr-xr-x  3 root root  4096 Dec  2 02:44 smali
drwxr-xr-x  3 root root  4096 Dec  2 02:44 assets
drwxr-xr-x  3 root root  4096 Dec  2 02:44 unknown
drwxr-xr-x  3 root root  4096 Dec  2 02:44 original
-rw-r--r--  1 root root 37077 Dec  2 02:44 apktool.yml
root@kali:~/Desktop/GMBot/clean_analysis/da88bdcb3d53d3ce7ab9f81d15be8497#
```

Extracted APK files

First stop is to take a look at the Android Manifest file, that should provide an overview of the components of the application and permissions requested.

Manifest Analysis – AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="com.kzcaxog.mgmxluwswb"
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.WAKE_LOCK"/>
    <uses-permission android:name="android.permission.GET_TASKS"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW"/>
    <uses-permission android:name="android.permission.RECEIVE_SMS"/>
    <uses-permission android:name="android.permission.SEND_SMS"/>
    <uses-permission android:name="android.permission.READ_CONTACTS"/>
    <uses-permission android:name="android.permission.READ_SMS"/>
    <uses-permission android:name="android.permission.WRITE_SMS"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
    <application android:allowBackup="false" android:icon="@drawable/mms_menu_icon" android:label="@st
        <activity android:configChanges="keyboardHidden|orientation" android:excludeFromRecents="true"
android:name="com.kzcaxog.mgmxluwswb.njsbz" android:screenOrientation="portrait" android:theme="@andro
```

Android Manifest

Initial analysis shows a broad range of permissions that indicate malicious behavior including permissions to:

- control all SMS messages (send, receive, read, write, delete)
- list running applications
- read the phone's state, contacts, SD card data
- request to be a device administrator enabling remote wiping of the device with no warning to the user

A summarized view of referenced class files for the main application, activities (15) and services (2) can be seen below:

Type	Class	Intents
Application	com.igcfse.enscbo.wieroel	android.intent.action.MAIN android.intent.category.LAUNCHER
Activity	com.kzcaxog.mgmxluwswb.njsbz	
	com.kzcaxog.mgmxluwswb.qwualvowdz.vgfwd	
	com.kzcaxog.mgmxluwswb.qwualvowdz.vxfuy	
	com.kzcaxog.mgmxluwswb.qwualvowdz.hnyakaifct	
	com.kzcaxog.mgmxluwswb.qwualvowdz.hbefajvdl	
	com.kzcaxog.mgmxluwswb.qwualvowdz.ezrbsaxvq	
	com.kzcaxog.mgmxluwswb.wtzwsygveo	
	com.kzcaxog.mgmxluwswb.vgtlnziya	
	com.kzcaxog.mgmxluwswb.rteomzu	
	com.kzcaxog.mgmxluwswb.rijdmjq	
	com.kzcaxog.mgmxluwswb.yemgk	
	com.kzcaxog.mgmxluwswb.djpfilgsck	
	com.kzcaxog.mgmxluwswb.vquwa	
Service	com.kzcaxog.mgmxluwswb.nkyelz	
	com.kzcaxog.mgmxluwswb.clsmzvfmcx	

Classes Declared in Manifest -Application, Activities and Services

In addition, we see 4 further classes mapped as Broadcast Receivers which will process event messages (Android system Intents) as shown below:

Type	Class	Intents
Broadcast Receiver	com.kzcaxog.mgmxluwswb.eosyfvh	android.intent.action.BOOT_COMPLETED
		com.kzcaxog.mgmxluwswb.wakeup
		com.denmark.reportsent
		exts.whats.wakeup
	com.kzcaxog.mgmxluwswb.liayim	android.app.action.DEVICE_ADMIN_ENABLED
		android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
		android.app.action.ACTION_DEVICE_ADMIN_DISABLE_REQUESTED
	com.kzcaxog.mgmxluwswb.ztdmo	com.whats.process
	com.kzcaxog.mgmxluwswb.xlsensgp	android.provider.Telephony.SMS_RECEIVED

Broadcast Receiver Classes Declared in Manifest

From this we can see the application is capable of:

- Executing code when the phone is powered on (starting the application automatically)
- Receive notification when Device Admin is granted, requested or a request to disable admin is received (and hence interfere, or nag the user to enable it)
- Receive notification of a new inbound SMS – with high priority flag to ensure the code can intercept it first and potentially stop any further alerts (can be used to steal 2FA tokens)

Before proceeding with any reverse engineering of the code, the next step is to explore the other files in the APK for clues.

Files of interest

The following files were noted as of interest:

File: assets/fytluah.dat

A binary file with no immediately obvious format. Possible code to be unencrypted / unpacked at run time?

File: res/values/strings.xml

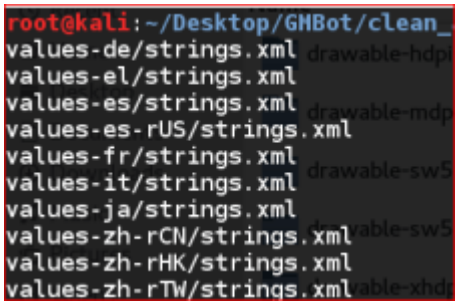
English language strings for the application, as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Adobe Flash</string>
    <string name="update">System update in progress. Please, Wait…</string>
    <string name="enter_phone">Enter phone number</string>
    <string name="hint_phone_number_not_denmark">Phone number</string>
    <string name="phone_country_code_with_plus">+45</string>
    <string name="phone_country_code">45</string>
    <string name="confirm_identity">Bekræft din identitet</string>
    <string name="hint_phone_number">mobilnummer</string>
    <string name="db_logo_description">Danske Bank</string>
    <string name="hint_cpr_number">CPR-nummer</string>
    <string name="hint_password">NemID Adgangskode</string>
    <string name="take_numid_picture">Tag et billede af din NemID nøglekort</string>
    <string name="retake_numid_picture">Tag et billede en gang til</string>
    <string name="send">SEND</string>
    <string name="card_number">Card number</string>
    <string name="install_id">1</string>
    <string name="hint_expiration_month">MM</string>
    <string name="hint_expiration_year">YY</string>
    <string name="expiration_date_separator">/</string>
    <string name="cvc_code">CVC</string>
    <string name="whatsapp">WhatsApp</string>
    <string name="google_play">Google Play</string>
    <string name="enter_credit_card">Enter details about the credit card for verifying payment details</string>
    <string name="cvc_code_popup_visa_description"><b>Discover, MasterCard, Visa, and Visa Electron</b>" The CVC is the last three
back of your card in the signature bar. "</string>
    <string name="cvc_code_popup_amex_description"><b>American Express</b>" The CVC is the four digits located on the front of the
    <string name="add_credit_card">Confirm credit card details.</string>
    <string name="add_instrument_continue">Continue</string>
    <string name="no_connection">No connection to server</string>
    <string name="phone_number">Phone number</string>
    <string name="first_name">First name</string>
    <string name="last_name">Last name</string>
    <string name="name_on_card">Cardholder name</string>
    <string name="change_password_restrict_label">Password</string>
</resources>
```

File: res/values/strings.xml (English Language Resource File)

The strings clearly indicate that this malware is targeting capturing victims credit card information. It is interesting to note that:

- The resource keys here are all in English, suggesting the original developer may be English speaking
- There are specific strings that are in Danish, despite this resource file being intended for English language
- In addition to English language strings we also see several other targeted countries:



Other Resource Files

File: res/values.xml

This file contains a list of country codes and specifically a group that are “non vbv”. This is understood to mean that they do not use the “Verified by Visa” process which is used to enforce additional verification checks during online purchases. It is likely that the attackers would seek to obtain additional VBV credentials via the malware in order to allow online purchases with the card details (or avoid these countries).

Verified By Visa Targeted Countries

Directory: res/drawable

Images and icons/logos including:

- Sample photo of Danish “Nem Id” – https://en.wikipedia.org/wiki/NemID
- Icon for Danske Bank mobile pay
- Mastercard secure code
- Icon for verified by visa
- Google play
- Flash icon (main application icon)
- Whatsapp

Additionally there are png images prefixed “overlay_”, indicating a possible use in fraudulent overlay activity.

Decompiling to Java source code

Next we attempt to reverse engineer the DEX file back to original Java source code. For this we use dex2jar as follows to translate the DEX file (in the APK) into a Java Class file archive:

```
Dex2jar da88bdcb3d53d3ce7ab9f81d15be8497.apk
```

The resulting jar file can then be disassembled using JD-GUI as follows:

```
java -jar ../../jd-gui-1.4.0.jar da88bdcb3d53d3ce7ab9f81d15be8497_dex2jar.jar
```

The resulting java classes that we see in JD-GUI show that there are only 4 java classes contained in the application. This is in direct contrast to the 16 different classes we saw declared in the application manifest. This confirms that there must be additional code that is loaded dynamically at run time – it is most likely that these four classes are in fact an unpacker.

Unpacker Classes

Examining the code we see that it is heavily obfuscated and has been crafted in a way to prevent clean decompiling of the code. This aside, we can start to get an understanding of the function of these four classes by examining the system classes that are imported (and therefore used) when the application is first executed.

After exporting the java source from JD-GUI and unzipping to a new folder, we can extract the imported classes from these files:

```
find . -type f -exec grep "^import" {} \; | sort -u
```

The classes we find are shown below :

Class	Imported Class
com.igcfse.enscbo.a	com.igcfse.enscbo.b
com.igcfse.enscbo.a	**java.io.RandomAccessFile**
com.igcfse.enscbo.a	**java.lang.reflect.Constructor**
com.igcfse.enscbo.b	android.app.Application
com.igcfse.enscbo.b	android.content.Context
com.igcfse.enscbo.b	com.igcfse.enscbo.a
com.igcfse.enscbo.b	**java.io.File**
com.igcfse.enscbo.b	**java.lang.reflect.Field**
com.igcfse.enscbo.b	**java.lang.reflect.Method**
com.igcfse.enscbo.c	android.content.Context
com.igcfse.enscbo.c	com.igcfse.enscbo.b
com.igcfse.enscbo.c	**java.io.FileDescriptor**

com.igcfse.enscbo.c	java.io.IOException
com.igcfse.enscbo.c	**java.lang.reflect.Constructor**
com.igcfse.enscbo.c	java.util.Random
com.igcfse.enscbo.wieroel	android.app.Application
com.igcfse.enscbo.wieroel	android.content.Context
com.igcfse.enscbo.wieroel	com.igcfse.enscbo.b

Essentially we have a very small set of libraries that are being imported and used. These consist of functionality for:

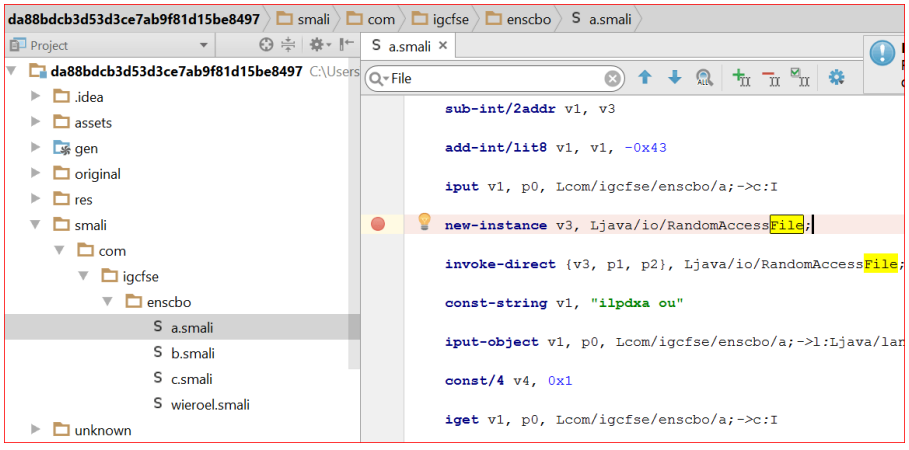
- General Android application and context classes (expected and needed for all android apps)
- File related classes (in **red**) – for access, reading and writing local files
- Java reflection classes (in **green**) – for creating new classes and instances and invoking methods dynamically

This confirms the hypothesis that we are most likely dealing with an unpacker that unpacks it's executable code from a local file resource (as opposed to pulling dynamically from network for example).

Stage 3 – Unpacker Debugging

As the Java code cannot be readily decompiled (due to protections injected by the malware author) we will instead debug the executable against the Smali assembly code. Smali is a disassembly of the DEX code used by the Dalvik Virtual Machine.

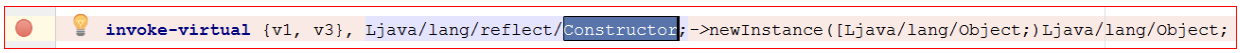
The Smali/Baksmali plugin for Android Studio is required, and then the output from Apktool is imported as a new project. We next set the breakpoints as required across the three classes that we are interested in (a,b,c):



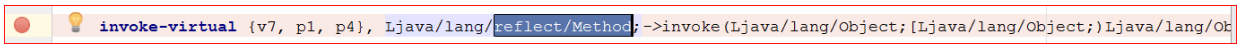
Setting Breakpoints in Android Studio

We will initially debug the calls to interesting reflection methods identified, which are as below:

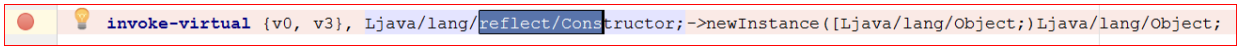
a.smali (a line that creates a new instance of a class based on a *java.lang.reflect.Constructor* instance)



b.smali (a line that invokes a method on an object via reflection)



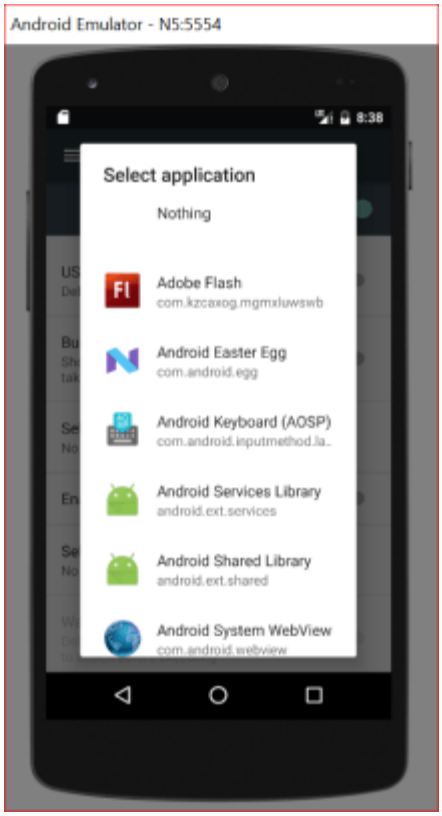
c.smali (a line similar to that described above for a.smali)



Now we install the application to the emulator (via ADB to ensure it doesn't start automatically as in some emulators).

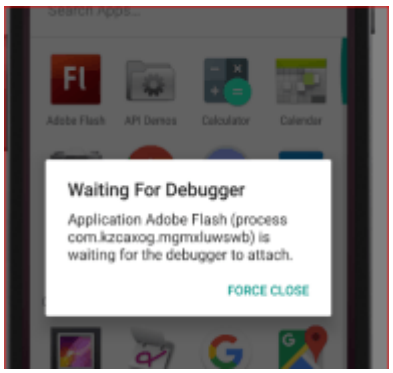
To enable the debugger to connect to the application, we perform the following prior to starting the application:

- Enable developer options by repeatedly clicking the build number in Settings > About device
- In developer options, choose "Select debug app" and choose the malicious application – "Adobe Flash"
- In developer options, enable the "wait for debugger"



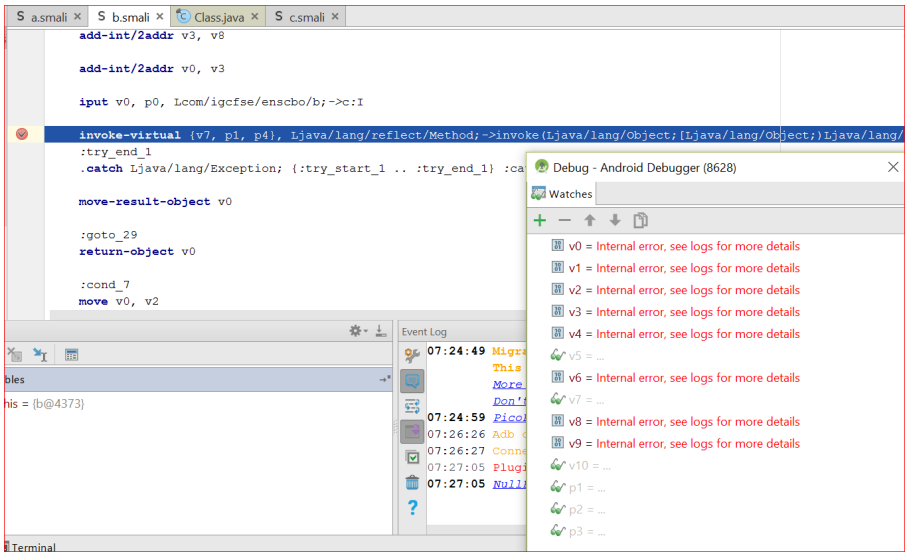
Selecting Debug Application

Now start the application from the launcher, you will be prompted to attach the debugger:



Attaching Debugger

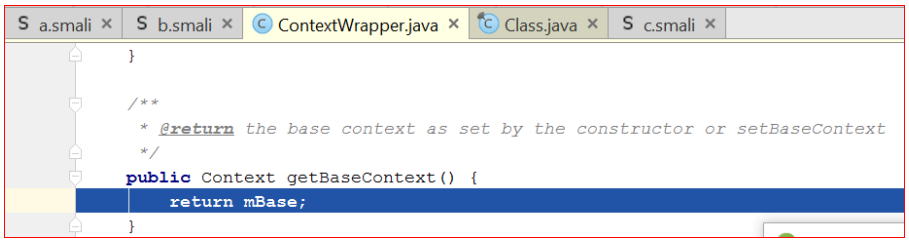
In Android Studio, attach the debugger using the icon. Choose the malicious application process. The debugger then stops at our first breakpoint as shown below:



First Breakpoint Reached

Note you should now set some variables to watch – as per above I have set v0 through v10 and p1 through p3. Our first breakpoint is hit and we see we are about to execute a method by reflection. Noting that we have not yet called newInstance() we can assume this is calling existing (loaded) classes – either one of the four loaded by the application, or some other Android framework classes.

Next we forces step into the method to see which method it is calling (the smali debugger seems a little buggy and we can't at this point see the parameters being passed).



Stepping into Reflected Method Call

An initial call to get the current context object -presumably to start retrieving local resources from the APK. We now allow the debugger to continue, and repeat this exercise several times to build up a flow of the reflected method calls:

```
Context android.context.ContextWrapper.getBaseContext()

//expected 2 arguments, got 1 – error in malware code, or to throw off debugging?
//Several more of these not shown
IllegalArgumentException java.lang.IllegalArgumentException(String s)

void Java.lang.reflect.setAccessible(boolean flag)

File android.app.getDataDir()
```

```
// returns /data/user/0/com.kzcaxog.mgmxluwswb/app_ydtjq
java.io.File.getAbsolutePath()

ContextImpl android.app.getImpl(Context context)

//filename is fytluah.dat
InputStream android.content.res.AssetManager.open(String fileName)
```

Pausing here, we can see the code is attempting to load the file that we had previously flagged as of interest in the static analysis section. Continuing we see the file is read, presumably decrypted and then written out again as a jar file:

```
int android.content.res.AssetManager.read(byte[] b)

//className = java.io.File
Class java.lang.Class.forName(String className)

//args = String "/data/user/0/com.kzcaxog.mgmxluwswb/app_ydtjq/gpyjzmose.jar"
T Java.lang.reflect.Constructor.newInstance(Object.. args)

void java.io.FileOutputStream.write(byte[] b) #25

void java.io.FileOutputStream.close()
```

Finally a *DexClassLoader* is invoked to load the additional code into the system:

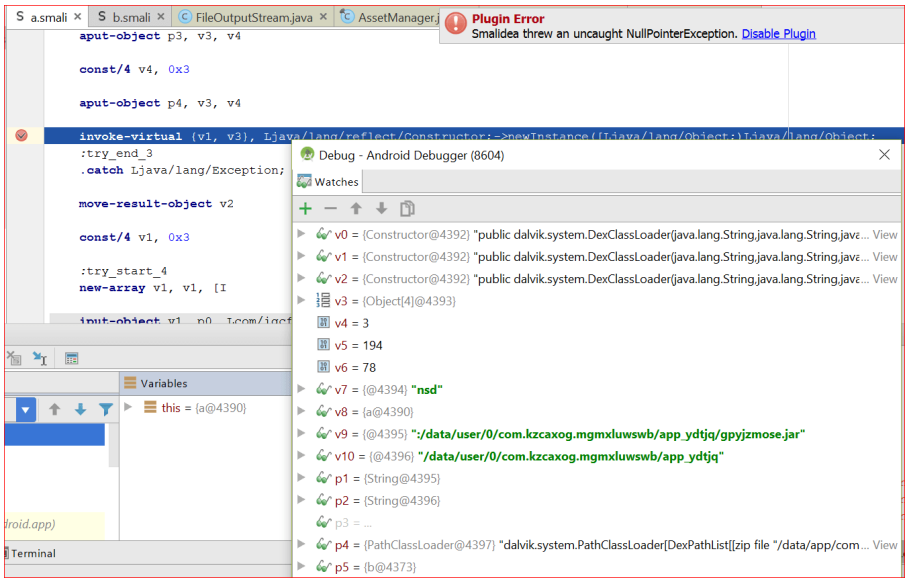
```
ClassLoader java.lang.Class.getClassLoader()

//className is dalvik.system.DexClassLoader
java.lang.Class.forName(String className)
```

Looking at the API for the DexClassLoader we can see that it takes two arguments – the location of the file to load, and a writeable area that it will use to re-write an optimised version of the code for the specific machine architecture – eg the Android Run Time (ART). Further information on this can be seen in the Android API documentation:

https://developer.android.com/reference/dalvik/system/DexClassLoader.html

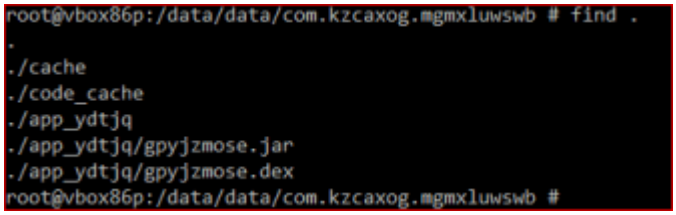
Stage 4 – DEX Extraction and Decompiling

We can see the exact location of the jar file in the debugger below, and the next step is to recover this file via ADB command line.



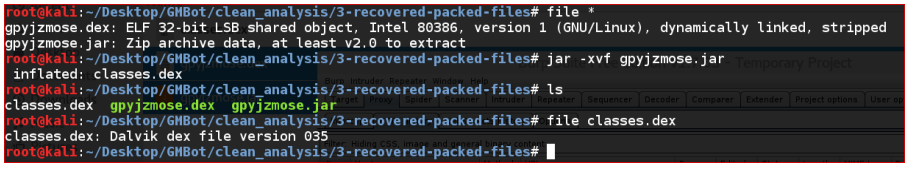
Debugging the Call to the DEXClassLoader

After execution of the classloader, connecting via ADB shell we see the two files, the original and the DEX optimised code:



Extracting the Payload Code

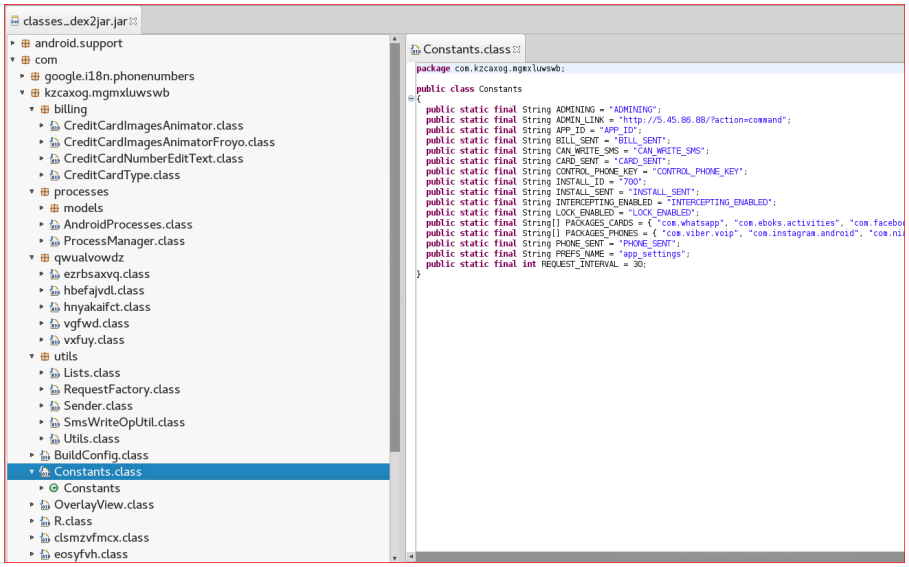
We copy these files to /sdcard/Download (+chmod) and then pull the .jar file to local machine for further analysis with adb pull.



Examining the files

Extracting the jar file we find the classes.dex file.

Repeating the steps to convert this to a jar file using dex2jar and decompiling with JD-GUI, we confirm we now have the full (un-obfuscated) source code for this malware sample.



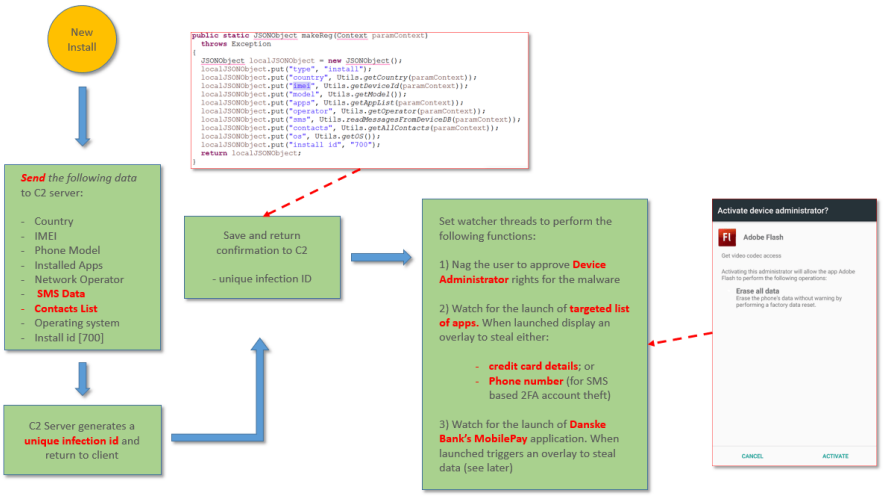
Decompiled Source Code

Stage 5 – Dynamic and Functional Analysis

First Installation

Upon initial analysis we can see the codebase bear remarkable similarities with the leaked source identified in the static analysis. However there are significant differences, and the code has been customised to specifically target the Danske Bank MobilePay application.

As the code is basically un-obfuscated, I'll now briefly walk through the key functionality of this malware, starting from first installation.



First Installation Process Overview

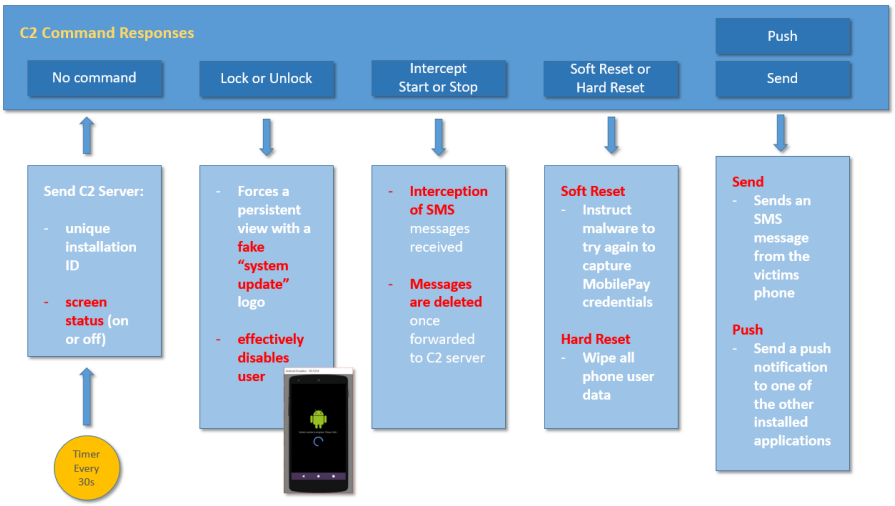
Upon first installation and execution the application will perform two primary functions. It will initially harvest a range of the users data, including phone contacts, all SMS messages and other key data and

send this to the C&C server. The C&C server then returns a unique installation identifier that is then used for all future communication to uniquely identify the compromised device.

Secondly the malware will then nag the user to accept the software as a device administrator. If the user declines the request is re-triggered, making it very difficult for most users to escape this screen without accepting. With this permission in place, the malware achieves two objectives:

1. The application cannot be un-installed by the user easily, without de-activating the device administrator. Attempting to do this will trigger the launching of overlays that prevent removing the device admin
2. At some point in the future, once further data has been stolen from the phone, the C2 server can issue a command to wipe the device, removing evidence of the infection and restoring the device to a factory state

Ongoing Operations – including after each reboot



Command and Control Process Flows

The malware maintains a regular heartbeat to the C2 server, which provides a mechanism for the attacker to issue specific commands to the device. Each hearbeat contains the installation ID and the current screen status. It is hypothesised that the attacker would ideally choose to execute malicious activities when the screen was off, and the user was not watching the phone.

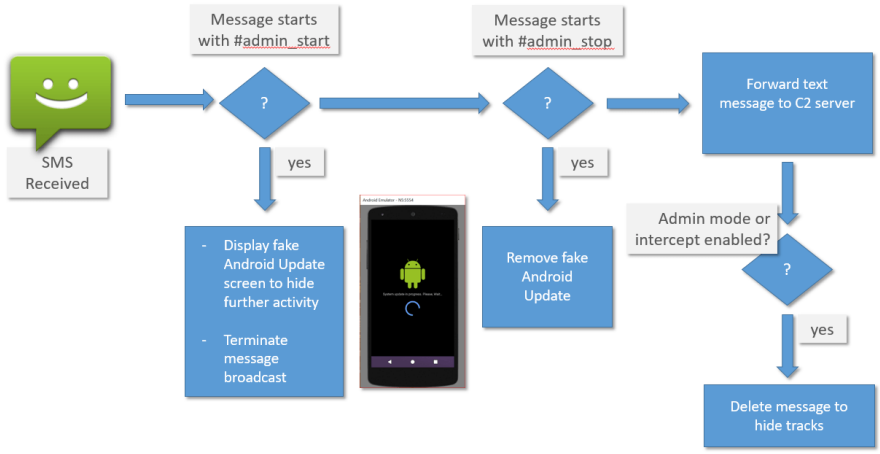
Firstly we see the ability to "lock" and "unlock" the phone. This simulates an Android software update screen, and effectively hides any other activity that is occurring behind the screen overlay (such as sending, receiving or deleting SMS messages). Additionally this could be used to disable the user, and prevent them from using the phone whilst their accounts or cards are being compromised in real time.

Next we see another function that is intended to intercept and forward SMS messages to the C2 server, and specifically trying to remove evidence that they ever existed by deleting them. This is used to steal 2FA credentials.

Next from a C2 server perspective we see two “reset” commands. The first, a “soft” reset, is used to reset the internal flag to re-attempt stealing Nem ID credentials. The second is the “hard” reset that performs and immediate wipe of the device data.

Finally, we see the ability to send an arbitrary SMS message to a mobile defined by the attacker and a function to launch a customised push notification to another application on the device. It was not clear what this could be used for.

SMS Remote Control



SMS Remote Control – “Admining Mode”

By listening for incoming SMS messages the malware could also trigger a fake Android update screen that would then harvest, forward and attempt to delete messages as they arrived on the phone. This mode could be enabled and disabled by customised SMS command messages delivered to the phone via SMS.

Automating Data Theft

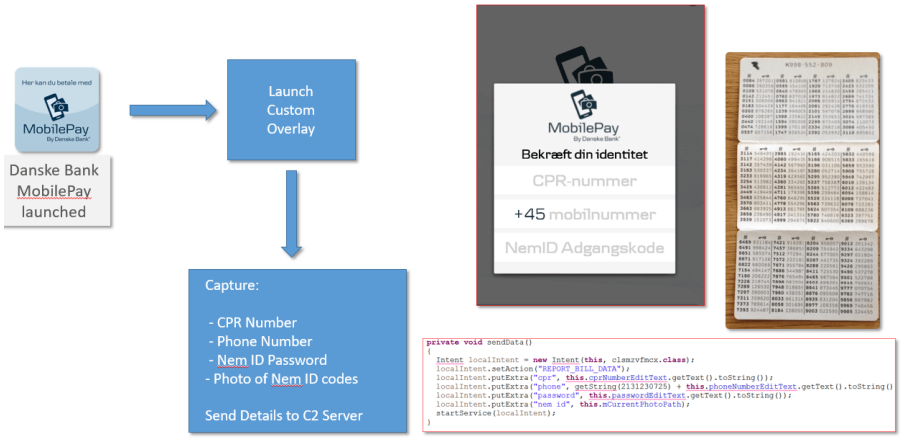
```
public class Constants
{
  public static final String ADMINING = "ADMINING";
  public static final String ADMIN_LINK = "http://5.45.86.88/?action=command";
  public static final String APP_ID = "APP_ID";
  public static final String BILL_SENT = "BILL_SENT";
  public static final String CAN_WRITE_SMS = "CAN_WRITE_SMS";
  public static final String CARD_SENT = "CARD_SENT";
  public static final String CONTROL_PHONE_KEY = "CONTROL_PHONE_KEY";
  public static final String INSTALL_ID = "700";
  public static final String INSTALL_SENT = "INSTALL_SENT";
  public static final String INTERCEPTING_ENABLED = "INTERCEPTING_ENABLED";
  public static final String LOCK_ENABLED = "LOCK_ENABLED";

  public static final String[] PACKAGES_CARDS = { "com.whatsapp", "com.eboks.activities", "com.facebook.orca",
                                                  "com.facebook.katana", "com.snapchat.android", "com.spotify.music",
                                                  "com.ubercab", "com.samsung.samsungupick", "com.netflix.mediaclient" };

  public static final String[] PACKAGES_PHONES = { "com.viber.voip", "com.instagram.android", "com.nianticlabs.pokemongo",
                                                   "com.viaplay.android", "com.hp.android.printservice", "dk.dr.webplayer",
                                                   "dk.guloggratis", "com.skype.raider" };
```

Decompiled Code Showing Targeted Applications

As per the original article and many of the indicators from the static analysis, the primary purpose of the application is to steal data by performing overlays on top of legitimate applications. The malware targets three specific classes of applications:

- Danske Bank's MobilePay application, with specific intent to steal Nem ID credentials
- Applications that trigger an attempt to steal credit card details via a custom overlay
- Applications that trigger an attempt to steal the users mobile phone number (possibly for triggering the "admining" mode described above)

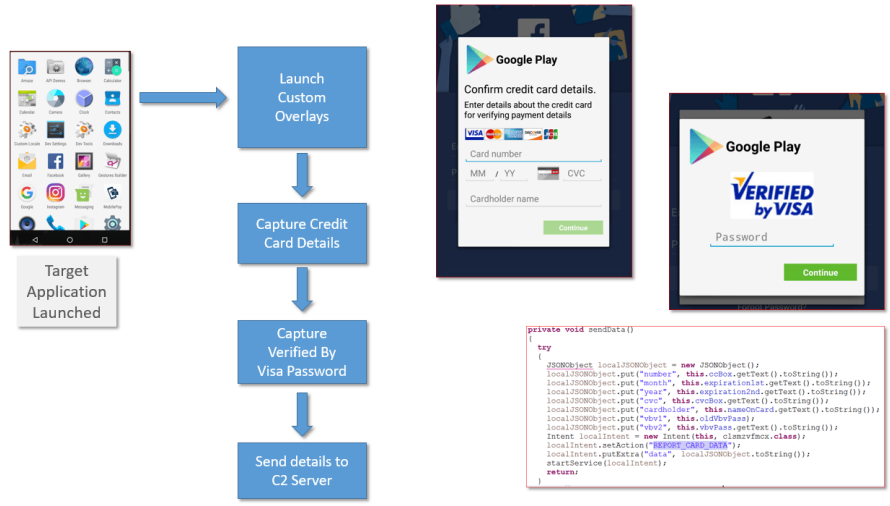
Danske Bank MobilePay



Danske Bank MobilePay Overlay Process

Upon launching the MobilePay application the overlay attempts to steal the users CPR number (unique social security type id), mobile number and Nem pass code. It then asks the user to take a photo of their Nem ID passbook, containing one time use codes which can be used by the attacker to then log into MobilePay (and other Danish systems) and issue payments.

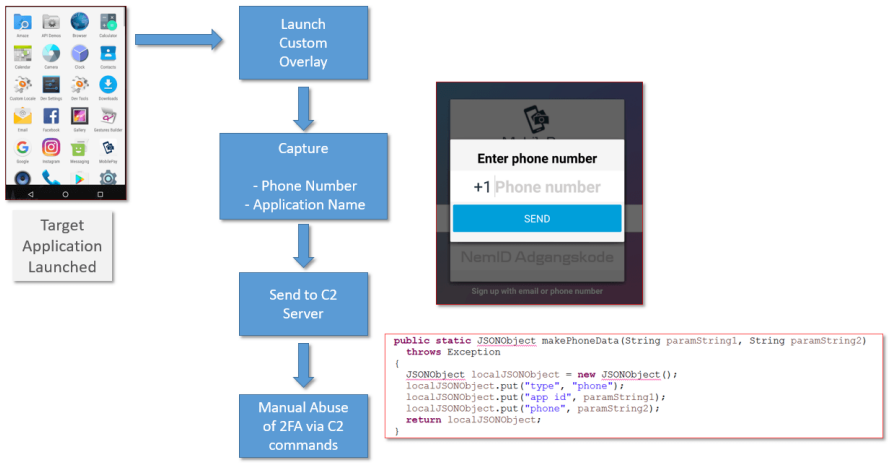
Stealing Credit Card Details



Credit Card Overlay Process

Upon launching one of the targeted applications, a credit card overlay is displayed with a configurable icon depending on the application launched. After basic card details are collected, the application then attempts to recover the Verified by Visa password for the user. These details are then forwarded to the C2 server.

Stealing Phone Numbers



Phone Number Overlay Process

Finally we see the functionality that is targeted to capture the user's phone number, presumably to enable further abuse of the victims account via abuse of text message 2FA.

Summary

The sample appears to be a specifically customised variant that is being used in a campaign to target the Danske Bank MobilePay application. We see evidence that it is probably not the original GM Bot

authors work – the coding style compared with the public source code is different, and the mix of languages in the resource files implies the sample has been adapted in a “quick and dirty” fashion to achieve the objectives.

This is a good example of how once released, complex code can be quickly and easily forked by less skilled authors and a pattern we also see today with the release of the Mirai botnet code. Quickly we see a spread of variants of the codebase that become harder to trace and detect and importantly attribute to any individual or group.

As ever, the best advice to prevent becoming a victim of such malware is to ensure that your phone is not configured to install 3rd party applications, and always review requests for permissions carefully – eg, are they aligned with the expected purpose of the application?

Open Questions

Due to time constraints there are a few further areas I would have liked to explore. I may pick these up in a subsequent post, but for the record they are:

1. The unpacked code contains included super user functionality from Chainfire’s SuperSu application. It’s not clear how or where this is used, no apparent attempt at rooting the device was seen in the unpacked code.
2. The debugger failed to return from the call to unpack the payload code. It is not clear if any further reflected actions were performed beyond this.
3. Given key indicators in the codebase, is it possible to search / locate other similar samples, or perhaps identify further C2 infrastructure

Any constructive feedback or comments most welcome.

About the author, the owner of the CyberBlog

I am an experienced IT consultant with a broad range of experience across different disciplines from development to large-scale Project Management. I have a passion for all things Cyber related but do not currently work in a Cyber related industry or role. I welcome and encourage all feedback!

Edited by Pierluigi Paganini

(Security Affairs – GM Bot Android, malware)

http://securityaffairs.co/wordpress/55125/malware/gm-bot-android-malware.html