How to analyze mobile malware: a Cabassous/FluBot Case study

By Jeroen Beckers

Published: 2021-04-19 · Archived: 2026-04-05 21:23:51 UTC

This blogpost explains all the steps I took while analyzing the Cabassous/FluBot malware. I wrote this while analyzing the sample and I've written down both successful and failed attempts at moving forward, as well as my thoughts/options along the way. As a result, this blogpost is **not** a writeup of the Cabassous/FluBot malware, but rather a step-by-step guide on how you can examine the malware yourself and what the thought process can be behind examining mobile malware. Finally, it's worth mentioning that all the tools used in this analysis are open-source / free.

If you want a straightforward writeup of the malware's capabilities, there's an excellent technical write up by ProDaft (pdf) and a writeup by Aleksejs Kuprins with more background information and further analysis. I knew these existed before writing this blogpost, but deliberately chose not to read them first as I wanted to tackle the sample 'blind'.

Our goal: Intercept communication between the malware sample and the C&C and figure out which applications are being attacked.

The sample

Cabassous/FluBot recently popped up in Europe where it is currently expanding quite rapidly. The sample I examined is attacking Spanish mobile banking applications, but German, Italian and Hungarian versions have been spotted recently as well.

In this post, we'll be taking a look at this sample (`acb38742fddfc3dcb511e5b0b2b2a2e4cef3d67cc6188b29aeb4475a717f5f95`). I've also uploaded this sample to the Malware Bazar website if you want to follow along.

This is live malware
Note that this is live malware and you should never install this on a device which contains sensitive information.

Starting with some static analysis

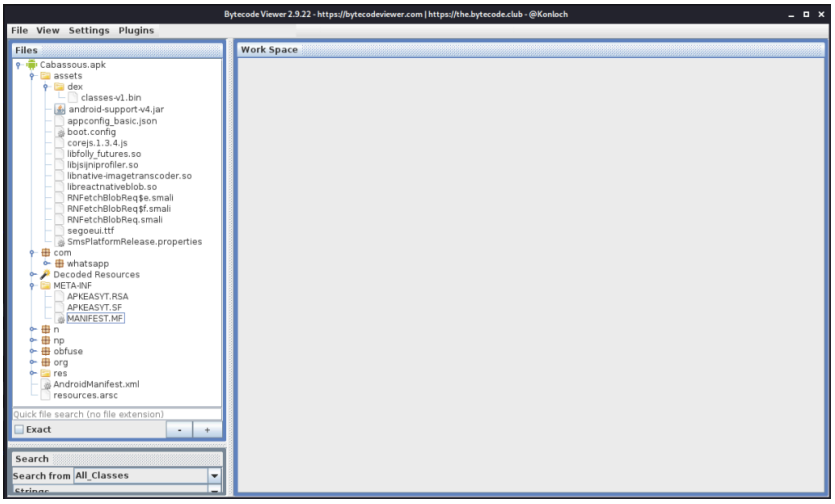
I usually make the mistake of directly going to dynamic analysis without some recon first, so this time I wanted to start things slow. It also takes some time to reset my phone after it has been infected, so I wanted to get the most out of my first install by placing Frida hooks where necessary.

First steps

The first thing to do is find the starting point of the application, which is listed in the AndroidManifest:

```
1  < activity android:name = "com.tencent.mobileqq.MainActivity" >
2      < intent-filter >
3          < action android:name = "android.intent.action.MAIN" />
4          < category android:name = "android.intent.category.LAUNCHER" />
5      </ intent-filter >
6  </ activity >
7  < activity android:name = "com.tencent.mobileqq.IntentStarter" >
8      < intent-filter >
9          < action android:name = "android.intent.action.MAIN" />
10     </ intent-filter >
11 </ activity >
```

So we need to find `com.tencent.mobileqq.MainActivity`. After opening the sample with Bytecode Viewer, there unfortunately isn't a com.tencent.mobileqq package. There are however a few other interesting things that Bytecode Viewer shows:



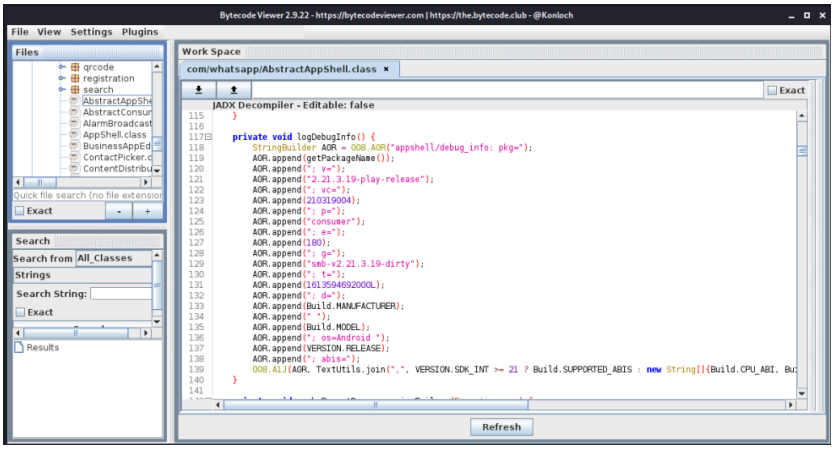
- There's a `classes-v1.bin` file in a folder called 'dex'. While this file probably contains dex bytecode, it currently isn't identified by the `file` utility and is probably encrypted.
- There is a `com.whatsapp` package with what appear to be legitimate WhatsApp classes
- There are three top-level packages that are suspicious: `n`, `np` and `obfuse`
- There's a `libreactnativeblob.so` which probably belongs to WhatsApp as well

Comparing the sample to WhatsApp

So it seems that the malware authors repackaged the official WhatsApp app and added their malicious functionality. Now that we know that, we can compare this sample to the official WhatsApp app and see if any functionality was added in the com.whatsapp folder. A good tool for comparing apks is apkdiff.

Which version to compare to?

I first downloaded the latest version of WhatsApp from the Google Play store, but there were way too many differences between that version and the sample. After digging around the com.whatsapp folder for a bit, I found the AbstractAppShell class which contains a version identifier: `2.21.3.19-play-release`. A quick google search leads us to apkmirror which has older versions for download.



So let's compare both versions using apkdiff:

```
1  python3 apkdiff.py .. /com .whatsapp_2.21.3.19-210319006_minAPI16\(x86\)\(nodpi\)_apkmirror.com.apk .. /Cabassous .apk
```

Because the malware stripped all the resource files from the original WhatsApp apk, apkdiff identifies 147 files that were modified. To reduce this output, I added 'xml' to the ignore list of apkdiff.py on line 14:

```
13  at = "at/"
14  ignore = ".*(align|apktool.yml|pak|MF|RSA|SF|bin|so|xml)"
15  count = 0
```

After running apkdiff again, the output is much shorter with only 4 files that are different. All of them differ in their labeling of try/catch statements and are thus not noteworthy.

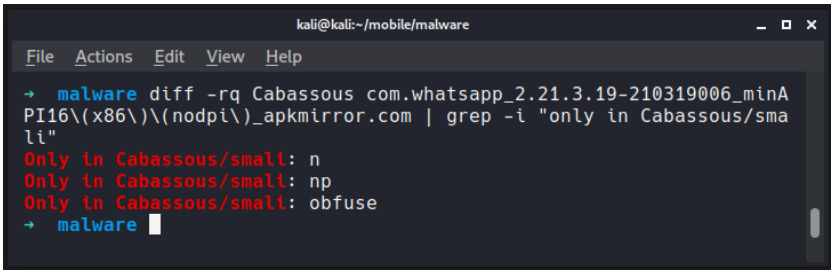
Something's missing…

It's pretty interesting to see that apkdiff doesn't identify the `n`, `np` and `obfuse` packages. I would have expected them to show up as being added in the malware sample, but apparently apkdiff only compares files that exist in both apks.

Additionally, apkdiff did not identify the encrypted dex file (`classes-v1.bin`). This is because, by default, apkdiff.py ignores files with the .bin extension.

So to make sure no other files were added, we can run a normal diff on the two smali folders after having used `apktool` to decompile them:

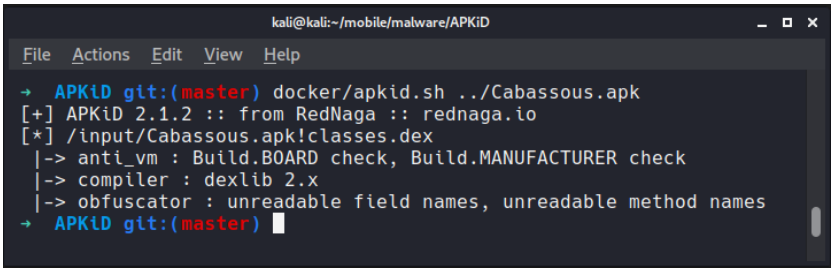
```
1  diff -rq Cabassous com.whatsapp_2.21.3.19-210319006_minAPI16\(x86\)\(nodpi\)_apkmirror.com | grep -i "only in Cabassous/smali"
```



It looks like no other classes/packages were added, so we can start focusing on the `n`, `np` and `obfuse` packages.

Examining the obfuscated classes

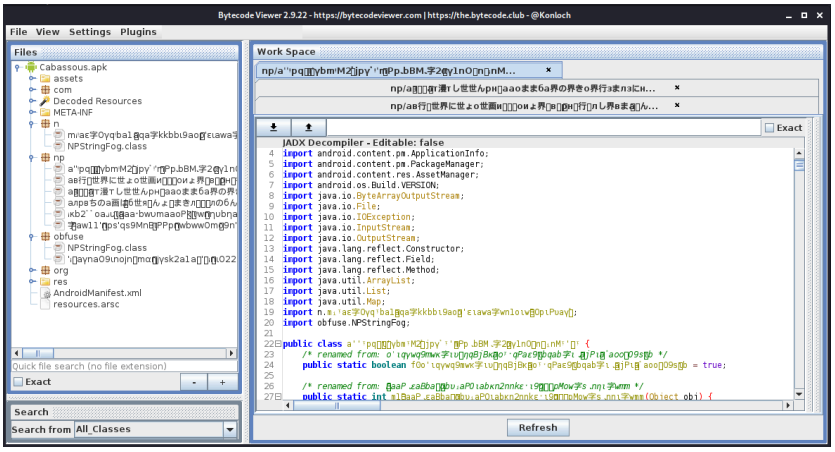
We still need to find the `com.tencent.mobileqq.MainActivity` class and it's probably inside the encrypted classes-v1.bin file. The com.tencent package name also tells us that the application has probably been packaged with the tencent packer. Let's use APKiD to see if it can detect the packer:



Not much help there; it only tells us that the sample has been obfuscated but it doesn't say with which packer. Most likely the tencent packer was indeed used, but it was then obfuscated with a tool unknown to APKiD.

So let's take a look at those three packages that were added ourselves. Our main goal is to find any references to `System.load` or `DexClassLoader`, but after scrolling through the files using different decompilers in Bytecode Viewer, I couldn't really find any. The classes use string obfuscation, control flow obfuscation and many of the decompilers are unable to decompile entire sections of the obfuscated classes.

There are however quite some imports for Java reflection classes, so the class and method names are probably constructed at runtime.

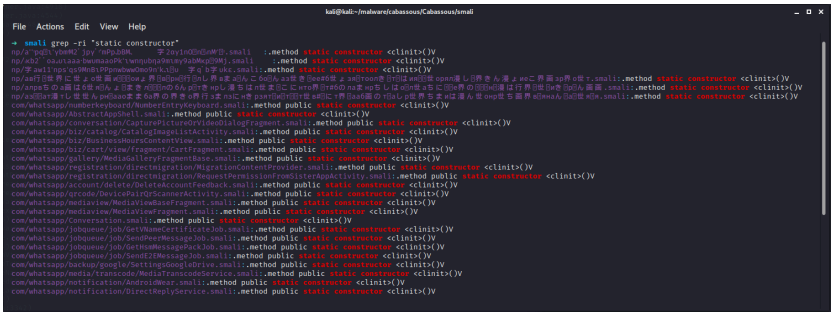


We could tackle this statically, but that's a lot of work. The unicode names are also pretty annoying, and I couldn't find a script that deobfuscates these, apart from the Pro version of the JEB decompiler. At this point, it would be better to move onto dynamic analysis and use some create Frida hooks to figure out what's happening. But there's one thing we need to solve first…

How is the malicious code triggered?

How does the application actually trigger the obfuscated functionality? It's not inside the MainActivity (which doesn't even exist yet), which is the first piece of code that will be executed when launching the app. Well, this is a trick that's often used by malware to hide functionality or to perform anti-debugging checks before the application actually starts. Before Android calls the MainActivity's `onCreate` method, all required classes are loaded into memory. After they are loaded in memory, all Static Initialization Blocks are executed. Any class can have one of these blocks, and they are all executed before the application actually starts.

The application contains many of these static initializers, both in the legitimate com.whatsapp classes and in the obfuscated classes:



Most likely, the classes-v1.bin file gets decrypted and loaded in one of the static initialization blocks, so that Android can then find the `com.tencent.mobileqq.MainActivity` and call its onCreate method.

On to Dynamic Analysis…

The classes-v1.bin file will need to be decrypted and then loaded. Since we are missing some classes, and since the file is inside a 'dex' folder, it's a pretty safe bet that it would decrypt to a dex file. That dex file then needs to be loaded using the DexClassLoader. A tool that's perfect for the job here is Dexcalibur by @FrenchYeti. Dexcalibur allows us to easily hook many interesting functions using Frida and is specifically aimed at apps that use reflection and dynamic loading of classes.

For my dynamic testing, I've installed LineageOS + TWRP on an old Nexus 5, I've installed Magisk, MagiskTrustUserCerts and Magisk Frida Server. I also installed ProxyDroid and configured it to connect to my Burp Proxy. Finally, I installed

Burp's certificate, made sure everything was working and then performed a backup using TWRP. This way, I can easily restore my device to a clean state and run the malware sample again and again for the first time. Since the malware doesn't affect the /system partition, I only need to restore the /data/ permission. You could use an emulator, but not all malware will have x86 binaries and, furthermore, emulators are easily detected. There are certainly drawbacks as well, such as the restore taking a few minutes, but it's currently fast enough for me to not be annoyed by it.

Resetting a device is easy with TWRP

Making and restoring backups is pretty straightforward in TWRP. You first boot into TWRP by executing '`adb reboot recovery`'. Each phone also has specific buttons you can press during boot, but using adb is much more nicer and consistent.

In order to create a backup, go to **Backup** and select the partitions you want to create a backup of. In this case, we should do `System`, `Data` and `Boot`. Slide the slider at the bottom to the right and wait for the backup to finish.

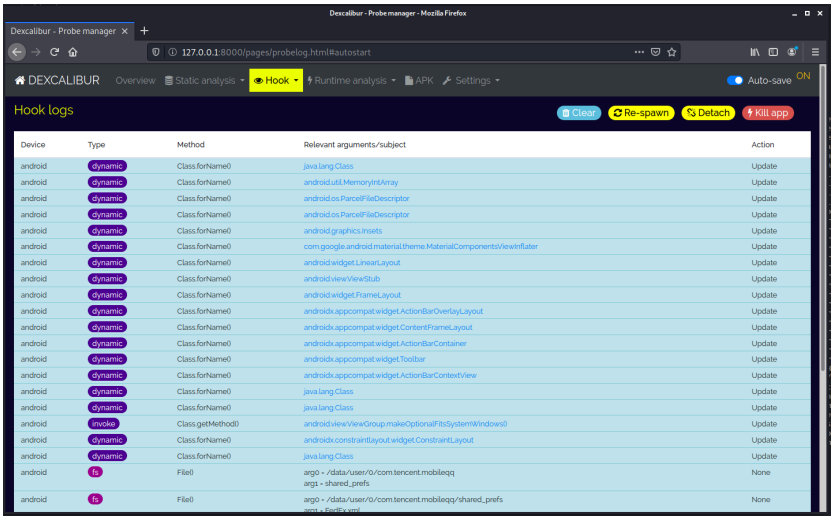
In order to restore a backup, go to **Restore** and select the backup you created earlier. You can choose which partitions you want to restore and then swipe the slider to the right again.

After setting up a device and creating a project, we can start analyzing. Unfortunately, the latest version of Dexcalibur wasn't too happy with the SMALI code inside the sample. Some lines have whitespace where it isn't supposed to be, and there are a few illegal constructions using array definitions and goto labels. Both of them were fixed within 24 hours of reporting which is very impressive!

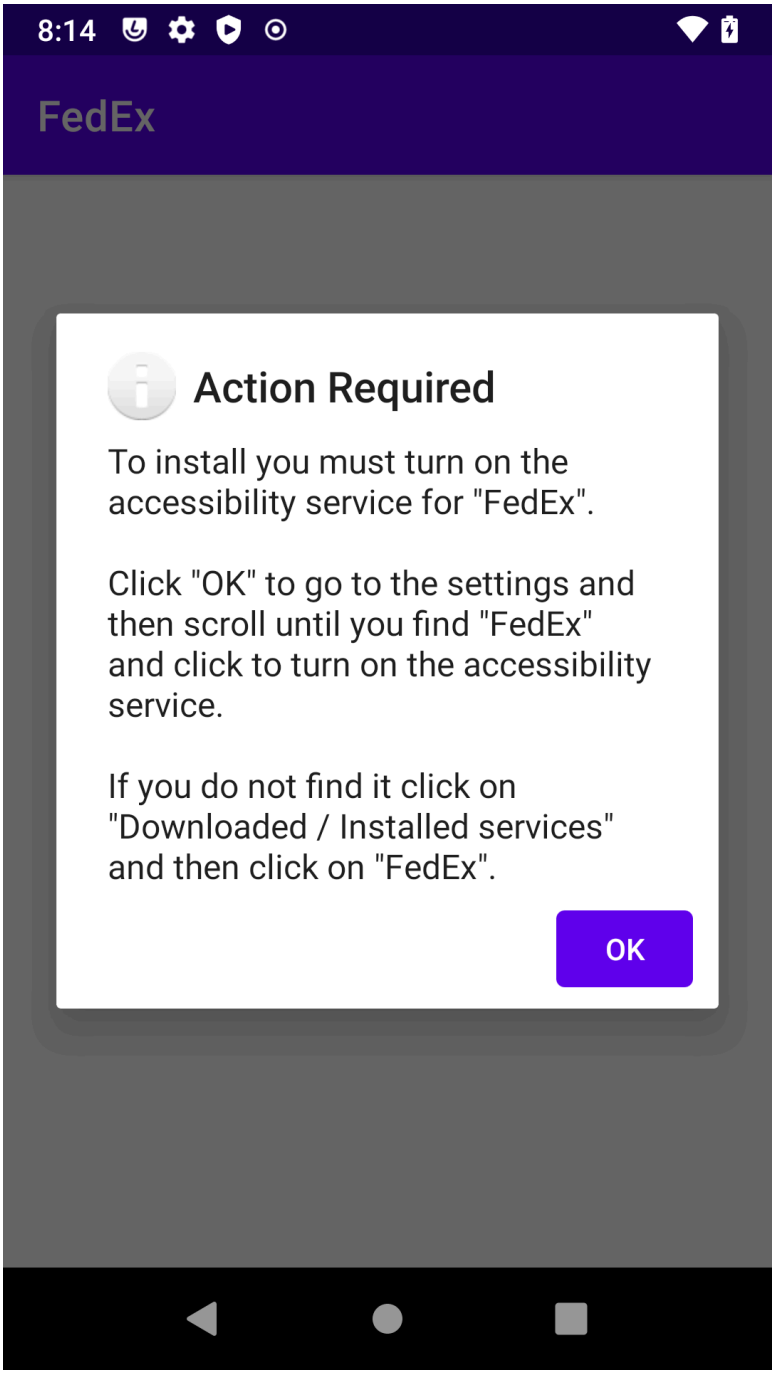
When something doesn't work…

Almost all the tools we use in mobile security are free and/or open source. When something doesn't work, you can either find another tool that does the job, or dig into the code and figure out exactly why it's not working. Even by just reporting an issue with enough information, you're contributing to the project and making the tools better for everyone in the future. So don't hesitate to do some debugging!

So after pulling the latest code (or making some quick hotpatches) we can run the sample using dexcalibur. All hooks will be enabled by default, and when running the malware Dexcalibur lists all of the reflection API calls that we saw earlier:



We can see that some visual components are created, which corresponds to what we see on the device, which is the malware asking for accessibility permissions.



At this point, one of the items in the hooks log should be the dynamic loading of the decrypted dex file. However, there's no such call and this actually had me puzzled for a little while. I thought maybe there was another bug in Dexcalibur, or maybe the sample was using a class or method not covered by Dexcalibur's default list of hooks, but none of this turns out to be the case.

Frida is too late 😕

Frida scripts only run when the runtime is ready to start executing. At that point, Android will have loaded all the necessary classes but hasn't started execution yet. However, static initializers are run during the initialization of the classes which is before Frida hooks into the Android Runtime. There's one issue reported about this on the Frida GitHub repository but it was closed without any remediation. There are a few ways forward now:

- We manually reverse engineer the obfuscated code to figure out when the dex file is loaded into memory. Usually, malware will remove the file from disk as soon as it is loaded in memory. We can then remove the function that removes the decrypted dex file and simply pull it from the device.
- We dive into the smali code and modify the static initializers to normal static functions and call all of them from the MainActivity.onCreate method. However, since the Activity defined in the manifest is inside the encrypted dex file,

we would have to update the manifest as well, otherwise Android would complain that it can't find the main activity as it hasn't been loaded yet. A real chicken/egg problem.

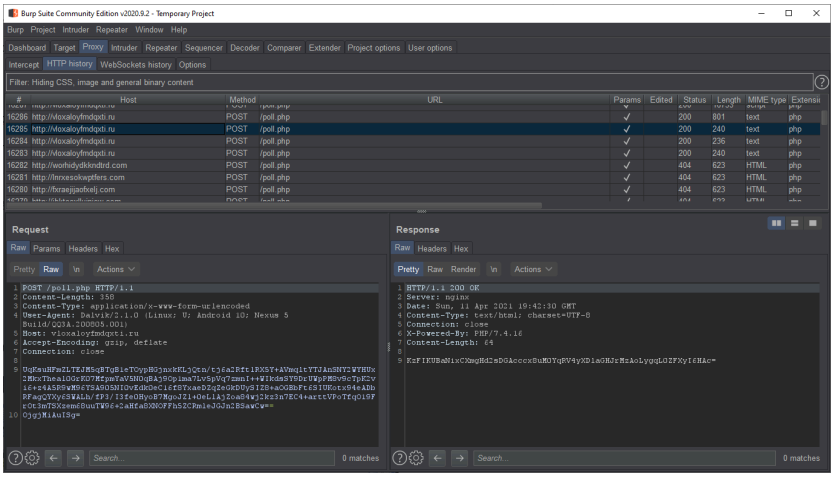
- Most (all?) methods can be decompiled by at least one of the decompilers in Bytecode Viewer, and there aren't too many methods, so we could copy everything over to a new Android project and simply debug the application to figure out what is happening. We could also trick the new application to decrypt the dex file for us.

But…. None of that is necessary. While figuring out why the hooks weren't called, I took a look at the application's storage and after the sample has been run once, it actually doesn't delete the decrypted dex file and simply keeps it in the app folder.

So we can copy it off the device by moving it to a world-readable location and making the file world-readable as well.

```
kali > adb shell
hammerhead:/ $ su
hammerhead:/ # cp /data/data/com.tencent.mobileqq/app_apkprotector_dex /data/local/tmp/classes-v1.bin
hammerhead:/ # chmod 666 /data/local/tmp/classes-v1.bin
hammerhead:/ # exit
hammerhead:/ $ exit
kali > adb pull /data/local/tmp/classes-v1.bin payload.dex
/data/local/tmp/classes-v1.bin: 1 file pulled. 18.0 MB/s (3229988 bytes in 0.171s)
```

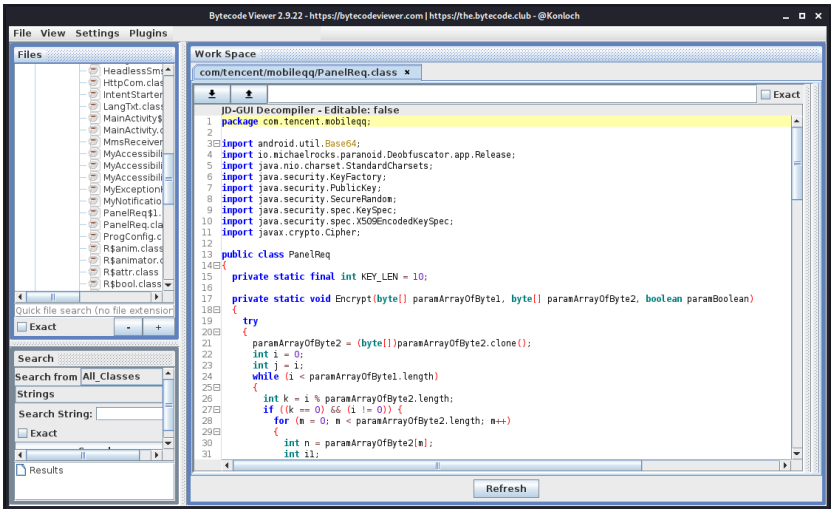
But now that we've got the malware running, let's take a quick look at Burp. Our goal is to intercept C&C traffic, so we might already be done!



While we are indeed intercepting C&C traffic, everything seems to be encrypted, so we're not done just yet.

… and back to static

Since we now have the decrypted dex file, let's open it up in Bytecode Viewer again:

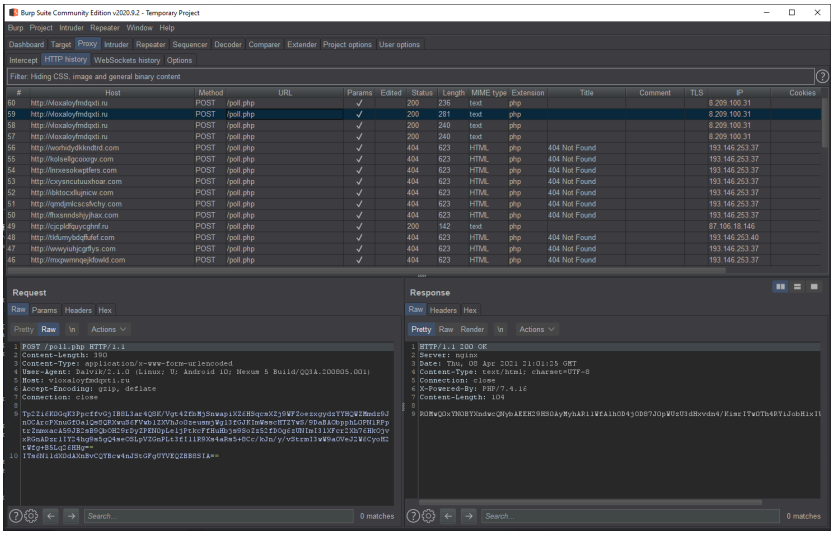


The payload doesn't have any real anti-reverse engineering stuff, apart from some string obfuscation. However, all the class and method names are still there and it's pretty easy to understand most functionality. Based on the class names inside the com.tencent.mobileqq package we can see that the sample can:

- Perform overlay attacks (`BrowserActivity.class`)
- Start different intens (`IntentStarter.class`)
- Launch an accessibility service (`MyAccessibilityService.class`)
- Compose SMS messages (`ComposeSMSActivity`)
- etc…

The string obfuscation is inside the `io.michaelrocks.paranoid` package (`Deobfuscator$app$Release.class`) and the source code is available online.

Another interesting class is `DGA.class` which is responsible for the Domain Generation Algorithm. By using a DGA, the sample cannot be taken down by sink-holing the C&C's domain. We could reverse engineer this algorithm, but that's not really necessary as the sample can just do it for us. At this point we also don't really care which domain it actually ends up connecting to. We can actually see the DGA in action in Burp: Before the sample is able to connect to a legitimate C&C it tries various different domain names (requests 46 – 56), after which it eventually finds a C&C that it likes (requests 57 – 60):



So the payloads are encrypted/obfuscated and we need to figure out how that's done. After browsing through the source a bit, we can see that the class that's responsible for actually communicating with the C&C is the `PanelReq` class. There are a few methods involving encryption and decryption, but there's also one method called 'Send' which takes two parameters and contains references to HTTP related classes:

```
1  public static String Send(String paramString1, String paramString2)
2  {
```

```
    try
    {
        HttpCom localHttpCom =  new com/tencent/mobileqq/HttpCom;
        localHttpCom.<init>();
        localHttpCom.SetPort( 80 );
        localHttpCom.SetHost(paramString1);
        localHttpCom.SetPath(Deobfuscator.app.Release.getString(-375422524606 44L));
        paramString1 = Deobfuscator.app.Release.getString(-37585202133604L);
```

We can be pretty sure that 'paramString1' is the hostname which is generated by the DGA. The second string is not immediately added to the HTTP request and various cryptographic functions are applied to it first. This is a strong indication that paramString2 will not be encrypted when it enters the Send method. Let's hook the Send method using Frida to see what it contains.

The following Frida script contains a hook for the `PanelReq.Send()` method:

```
Java.perform( function (){
    var PanelReqClass = Java.use( "com.tencent.mobileqq.PanelReq" );
    PanelReqClass.Send.overload( 'java.lang.String' , 'java.lang.String' ).implementation = function (hostname, payload){
        console.log( "hostname:" +hostname);
        console.log( "payload:" +payload);
        var retVal =  this .Send(hostname, payload);
        console.log( "Response:" + retVal)
        console.log( "------" );
        return retVal;
    }
});
```

Additionally, we can hook the Deobfuscator.app.Release.getString method to figure out which strings are returned after decrypting them, but in the end this wasn't really necessary:

```
var Release = Java.use("io.michaelrocks.paranoid.Deobfuscator$app$Release");
Release.getString.implementation = function (id){
    var retVal = this.getString(id);
    console.log(id + " > " + retVal);
    console.log("---")
    return retVal;
}
```

Monitoring C&C traffic

After performing a reset of the device and launching the sample with Frida and the overloaded Send method, we get the following output:

```
...
```

```
hostname:vtcslaabqljbnco[.]com
payload:PREPING,
Response:null
------
hostname:urqisbcliipfrac[.]com
payload:PREPING,
Response:null
------
hostname:vloxaloyfmdqxti[.]ru
payload:PREPING,
Response:OK
------
hostname:cjcpldfquycghnf[.]ru
payload:PREPING,
Response:null
------
Response:nullhostname:vloxaloyfmdqxti[.]ru
payload:PING,3.4,10,LGE,Nexus 5,en,127,
Response:
------
hostname:vloxaloyfmdqxti.ru
payload:SMS_RATE
Response: 10
------
hostname:vloxaloyfmdqxti[.]ru
payload:GET_INJECTS_LIST,com.google.android.carriersetup,org.lineageos.overlay.accent.black,com.android.cts.priv.ctsshim,org.line
Response:
------
hostname:vloxaloyfmdqxti[.]ru
payload:LOG,AMI_DEF_SMS_APP,1
Response:OK
------
hostname:vloxaloyfmdqxti[.]ru
payload:GET_SMS
Response:648516978,Capi: El envio se ha devuelto dos veces al centro mas cercano codigo: AMZIPH1156020
------
hostname:vloxaloyfmdqxti[.]ru
payload:GET_SMS
Response:634689547,No hemos dejado su envio 01101G573629 por estar ausente de su domicilio. Vea las opciones:
```

```
------
hostname:vloxaloyfmdqxti[.]ru
payload:GET_SMS
Response:699579720,Hola, no te hemos localizado en tu domicilio. Coordina la entrega de tu envio 279000650 aqui:
------
hostname:vloxaloyfmdqxti[.]ru
payload:LOG,AMI_DEF_SMS_APP,0
Response:OK
------
hostname:vloxaloyfmdqxti[.]ru
payload:PING,3.4,10,LGE,Nexus 5,en,197,
Response:
------
...
```

Some observations:

- The sample starts with querying different domains until it finds one that answers 'OK' (Line 14). This confirms with what we saw in Burp.
- It sends a list of all installed applications to see which applications to attack using an overlay (Line 27). Currently, no targeted applications are installed, as the response is empty
- Multiple premium text messages are received (Lines 36, 41, 46, …)

Package names of targeted applications are sometimes included in the apk, or a full list is returned from the C&C and compared locally. In this sample that's not the case and we actually have to start guessing. There doesn't appear to be a list of financial applications available online (or at least, I didn't find any) so I basically copied all the targeted applications from previous malware writeups and combined them into one long list. This does not guarantee that we will find all the targeted applications, but it should give us pretty good coverage.

In order to interact with the C&C, we can simply modify the Send hook to overwrite the payload. Since the sample is constantly polling the C&C, the method is called repeatedly and any modifications are quickly sent to the server:

```
Java.perform( function (){
    var PanelReqClass = Java.use( "com.tencent.mobileqq.PanelReq" );
    PanelReqClass.Send.overload( 'java.lang.String' , 'java.lang.String' ).implementation = function (hostname, payload){
        var injects= "GET_INJECTS_LIST,alior.banking[...]zebpay.Application,"
        if (payload.split( "," )[0] == "GET_INJECTS_LIST" ){
            payload=injects
        }
        console.log( "hostname:" +hostname);
        console.log( "payload:" +payload);
        var retVal = this .Send(hostname, payload);
        console.log( "Response:" + retVal)
```

```
        console.log( "------" );
        return retVal;
    }
});
```

Frida also automatically reloads scripts if it detects a change, so we can simply update the Send hook with new commands to try out and it will automatically be picked up.

Based on the very long list of package names I submitted, the following response was returned by the server to say which packages should be attacked:

```
-----
hostname:vloxaloyfmdqxti[.]ru
payload:GET_INJECTS_LIST,alior.banking[...]zebpay.Application
Response:com.bankinter.launcher,com.bbva.bbvacontigo,com.binance.dev,com.cajasur.android,com.coinbase.android
-----
```

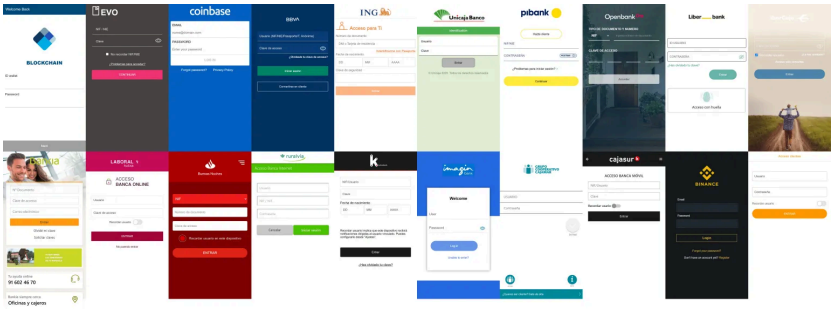
When the sample receives the list of applications to attack, it immediately begins sending the GET_INJECT command to get a HTML page for each targeted application:

```
---
hostname:vloxaloyfmdqxti[.]ru
payload:GET_INJECT,es.evobanco.bancamovil
Response:&lt;!DOCTYPE html>
&lt;html>
&lt;head>
    &lt;title>evo&lt;/title>
    &lt;link rel="shortcut icon" href="es.evobanco.bancamovil.png" type="image/png">
    &lt;meta charset="utf-8">
....
```

In order to view the different overlays, we can modify the Frida script to save the server's response to an HTML file:

```
if (payload.split( "," )[0] == "GET_INJECT" ){
        var file = new File( "/data/data/com.tencent.mobileqq/" +payload.split( "," )[1] + ".html" , "w" );
        file.write(retVal);
        file.close();
}
```

We can then extract them from the device, open them in Chrome, take some screenshots and end up with a nice collage:



Conclusion

The sample we examined in this post is pretty basic. The initial dropper made it a little bit difficult, but since the decrypted payload was never removed from the application folder, it was easy to extract and analyze. The actual payload uses a bit of string obfuscation but is very easy to understand.

The communication with the C&C is encrypted, and by hooking the correct method with Frida we don't even have to figure out how the encryption works. If you want to know how it works though, be sure to check out the technical writeups by ProDaft (pdf) and Aleksejs Kuprins.

Jeroen Beckers

Jeroen Beckers is a mobile security expert working in the NVISO Software Security Assessment team. He travels around the world teaching *SANS SEC575: iOS and Android Application Security Analysis and Penetration Testing* and is a co-author of OWASP Mobile Application Security (MAS) project, which includes:

- OWASP Mobile Application Security Testing Guide (MASTG)
- OWASP Mobile Application Security Verification Standard (MASVS)
- OWASP Mobile Application Security Weakness Enumeration (MASWE)
