N Ways to Unpack Mobile Malware – Pentest Blog

Published: 2019-03-13 · Archived: 2026-04-05 18:29:32 UTC

This article will briefly explain methods behind the mobile malware unpacking. It will be focusing on Anubis since it is the latest trending malware for almost a year now. Actors use dropper applications as their primary method of distribution. Droppers find their ways to Google Play store under generic names thereby infecting devices with Anubis. An example of a such dropper may found in the references. There were at least forty cases in Google Play in the last fall targeting Turkish users. @LukasStefanko's twitter thread may be helpful to get an overview of such campaigns. Anubis malware already analysed by fellows from the industry in a detailed manner. Therefore readers should find it more valuable to have an article focusing on packer mechanisms of Anubis.

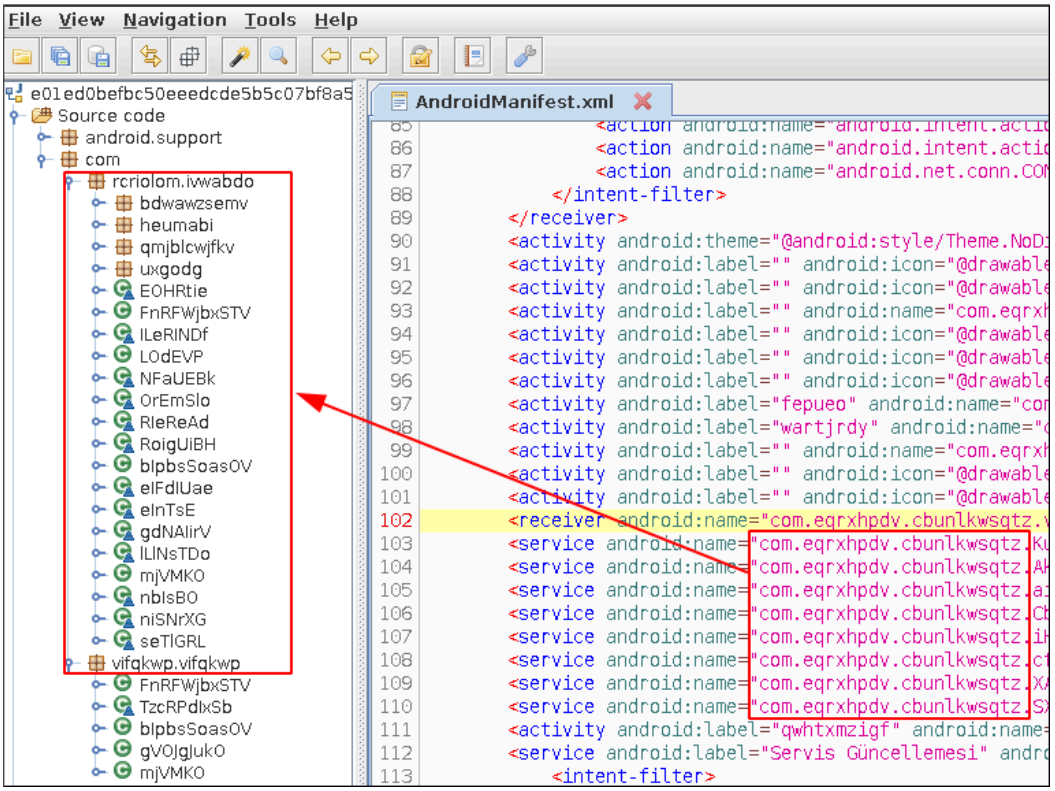
The sample used in this article is available at the references section. I strongly recommend downloading the sample and following through the article. I will be dividing this post into three sections.

Packers in Android Ecosystem

Mobile malwares also make use of packers to hide their malicious payloads from researchers and AV programs. This includes reflection, obfuscation, code-flow flattening and trash codes to make unpacking process stealthy. All mechanisms mentioned are used by the Anubis packer and therefore will be explored in this article.

Loading classes at runtime

Android applications must define their used services, receivers, activity classes in AndroidManifest file to use them. In Anubis samples, it is clear that there are many classes not defined in the Manifest file that are simply present in the source code.

- 

This means that a file with non-defined classes should be loaded into application at run-time. There are two main ways of run-time loading in Android:

From file:

- dalvik.system.DexFile.loadDex depreciated after API 26
- dalvik.system.DexClassLoader
- dalvik.system.PathClassLoader

From memory:

- dalvik.system.InMemoryDexClassLoader (not common in malwares)

Loading from the file requires a dex/jar file to be present in file system. Anubis unpacks the encrypted data file and then drops the decrypted version. Later on malware proceeds loading decrypted dex into the application. After loading with DexClassLoader, malware removes the decrypted dex file. Tracing the dexClassLoader should make the loading routine clear. Since dexClassLoader is a class of dalvik.system package “dalvik.system.dexClassLoader” should be in the code but it is nowhere to be found.

Reflection

Another useful method when dealing with malware is reflection. Reflection is an important concept in Java which lets you to call methods/classes without knowing about them in compile time. There are several classes/methods for reflection.

- java.lang.Class.forName
- java.lang.ClassLoader.loadClass
- java.lang.reflect.Method
- java.lang.Class.getMethods

Example usage of forName

cObj = Class.forName("dalvik.system.dexClassLoader");

cObj = Class.forName("dalvik.system.dexClassLoader");

```
cObj = Class.forName("dalvik.system.dexClassLoader");
```

cObj variable holds the class object of dexClassLoader. This enables program to call methods of any given class. The problem is to find where function calls are made to reflection methods.

Catching packers with Frida

frida is a dynamic instrumentation toolkit supported by nearly every operating system. Frida makes it possible to inject a piece of code to manipulate target program and also to trace program calls. In this case it will be used for tracing which reflection calls are made thereby analysing the threads. When previously mentioned function calls

are made, console.log will be called additionally. But before that, let’s take a quick recap on how to setup Frida on android emulator.

Download frida-server suitable with your emulator from:
(e.g Genymotion uses x86 architecture.)
https://github.com/frida/frida/releases.

adb push frida-server /data/local/tmp

adb push frida-server /data/local/tmp adb shell cd /data/local/tmp chmod +x frida-server ./frida-server &

```
adb push frida-server /data/local/tmp
adb shell
cd /data/local/tmp
chmod +x frida-server
./frida-server &
```

Frida tools should be installed in host machine by running
`pip install frida-tools`

After the setup, we can write a script to hook our target methods. We will start by defining variables for classes of our methods.

var classDef = Java.use('java.lang.Class');

var classLoaderDef = Java.use('java.lang.ClassLoader');

var loadClass = classLoaderDef.loadClass.overload('java.lang.String', 'boolean');

var forName = classDef.forName.overload('java.lang.String', 'boolean', 'java.lang.ClassLoader');

var reflect = Java.use('java.lang.reflect.Method')

var member = Java.use('java.lang.reflect.Member')

var dalvik = Java.use("dalvik.system.DexFile")

var dalvik2 = Java.use("dalvik.system.DexClassLoader")

var dalvik3 = Java.use("dalvik.system.PathClassLoader")

//var dalvik4 = Java.use("dalvik.system.InMemoryDexClassLoader")

var f = Java.use("java.io.File")

var url = Java.use("java.net.URL")

var obj = Java.use("java.lang.Object")

var fo = Java.use("java.io.FileOutputStream")

var classDef = Java.use('java.lang.Class'); var classLoaderDef = Java.use('java.lang.ClassLoader'); var loadClass = classLoaderDef.loadClass.overload('java.lang.String', 'boolean'); var forName = classDef.forName.overload('java.lang.String', 'boolean', 'java.lang.ClassLoader'); var reflect = Java.use('java.lang.reflect.Method') var member = Java.use('java.lang.reflect.Member') var dalvik = Java.use("dalvik.system.DexFile") var dalvik2 = Java.use("dalvik.system.DexClassLoader") var dalvik3 = Java.use("dalvik.system.PathClassLoader") //var dalvik4 = Java.use("dalvik.system.InMemoryDexClassLoader") var f = Java.use("java.io.File") var url = Java.use("java.net.URL") var obj = Java.use("java.lang.Object") var fo = Java.use("java.io.FileOutputStream")

```
var classDef = Java.use('java.lang.Class');
var classLoaderDef = Java.use('java.lang.ClassLoader');
var loadClass = classLoaderDef.loadClass.overload('java.lang.String', 'boolean');
var forName = classDef.forName.overload('java.lang.String', 'boolean', 'java.lang.ClassLoader');
var reflect = Java.use('java.lang.reflect.Method')
var member = Java.use('java.lang.reflect.Member')
var dalvik = Java.use("dalvik.system.DexFile")
var dalvik2 = Java.use("dalvik.system.DexClassLoader")
var dalvik3 = Java.use("dalvik.system.PathClassLoader")
//var dalvik4 = Java.use("dalvik.system.InMemoryDexClassLoader")
var f = Java.use("java.io.File")
var url = Java.use("java.net.URL")
var obj = Java.use("java.lang.Object")
var fo = Java.use("java.io.FileOutputStream")
```

We will be using this code snippet to change implementation of a method.

class.targetmethod.implementation = function(){

console.log("[+] targetmethod catched !")

return this.targetmethod()

class.targetmethod.implementation = function(){ console.log("[+] targetmethod catched !") stackTrace() return this.targetmethod() }

```
class.targetmethod.implementation = function(){
    console.log("[+] targetmethod catched !")
    stackTrace()
    return this.targetmethod()
}
```

`console.log("[+] {x} function catched !")` will enable us to see if the function is called. If function takes any parameters such as a string, logging those may become helpful during the analysis. Then we can get more

information about the thread we are in. Frida is able to call any android function including `getStackTrace()`. But that requires a reference to the current thread object. Let's start by getting instance of the thread class:

var ThreadDef = Java.use('java.lang.Thread');

var ThreadObj = ThreadDef.$new();

var ThreadDef = Java.use('java.lang.Thread'); var ThreadObj = ThreadDef.$new();

```
var ThreadDef = Java.use('java.lang.Thread');
var ThreadObj = ThreadDef.$new();
```

ThreadObj holds instance of the Thread class and `currentThread()` can be used to get thread according to https://developer.android.com/reference/java/lang/Thread.html.
We can now use `getStackTrace()` and also loop through stackElements to print the call stack.

console.log("------------START STACK----------------")

var stack = ThreadObj.currentThread().getStackTrace();

for (var i = 0; i < stack.length; i++) {

console.log(i + " => " + stack[i].toString());

console.log("------------END STACK----------------");

function stackTrace() { console.log("------------START STACK----------------") var stack = ThreadObj.currentThread().getStackTrace(); for (var i = 0; i < stack.length; i++) { console.log(i + " => " + stack[i].toString()); } console.log("------------END STACK----------------"); }

```
    function stackTrace() {
        console.log("------------START STACK----------------")
        var stack = ThreadObj.currentThread().getStackTrace();
        for (var i = 0; i < stack.length; i++) {
            console.log(i + " => " + stack[i].toString());
        }
        console.log("------------END STACK----------------");
    }
```

Printing call stack helps to identify call graph of reflections and unpacking mechanisms. For example dexClassLoader might have created with reflection. But when frida hooks into dexClassLoader and prints the call stack, we can see the functions before dexClassLoader is called. Unpacking routines are called at the very beginning of the application. Therefore frida should be attached as soon as possible to catch the unpacking process. Fortunately -f option in frida enables frida to spawn target app itself. frida accepts scripts with the -l parameter.
`frida -U -f appname -l dereflect.js`

Then frida waits input from the user to continue. `%resume` will resume the process. Full script is available at my github repository.
https://github.com/eybisi/nwaystounpackmobilemalware/blob/master/dereflect.js

```
[123vmxc 6.0::com.eqrxhpdv.cbunlkwsqtz]-> %resume
[123vmxc 6.0::com.eqrxhpdv.cbunlkwsqtz]-> Reflection => loadClass => com.rcriolom.ivwabdo.LOdEVP
Reflection => forName => java.io.FileOutputStream
[+] Output stream created with the file : /data/user/0/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.jar
[+] write catched
[!] Output stream closed
Reflection => forName => dalvik.system.DexClassLoader
Reflection => forName => java.lang.ClassLoader
[+] DexClassLoader Catched -> :/data/user/0/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.jar
[+] loadDex Catched -> /data/user/0/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.jar
```

Output without the stackTrace():

```
SpawnSpawned `com.eqrxhpdv.cbunlkwsqtz`. Use %resume to let the main thread start exe
[123vmxc 6.0::com.eqrxhpdv.cbunlkwsqtz]-> %resume
[123vmxc 6.0::com.eqrxhpdv.cbunlkwsqtz]-> Reflection => loadClass => com.rcriolom.ivw
Reflection => forName => java.io.FileOutputStream
[+] Output stream created with the file : /data/user/0/com.eqrxhpdv.cbunlkwsqtz/app_f
[+] write catched
--------------------------START STACK----------------------------------
message: {'type': 'send', 'payload': {'$handle': '0x10bec6', '': {}, '$weakRef': 1191
0 => dalvik.system.VMStack.getThreadStackTrace(Native Method)
1 => java.lang.Thread.getStackTrace(Thread.java:580)
2 => java.io.FileOutputStream.write(Native Method)
3 => java.io.OutputStream.write(OutputStream.java:82)
4 => java.lang.reflect.Method.invoke(Native Method)
5 => com.rcriolom.ivwabdo.mjVMKO.KBVcvShX(Unknown Source)
6 => com.rcriolom.ivwabdo.mjVMKO.KBVcvShX(Unknown Source)
7 => com.rcriolom.ivwabdo.bIpbsSoasOV.KBVcvShX(Unknown Source)
8 => com.rcriolom.ivwabdo.bIpbsSoasOV.KBVcvShX(Unknown Source)
9 => com.rcriolom.ivwabdo.bIpbsSoasOV.KBVcvShX(Unknown Source)
10 => com.rcriolom.ivwabdo.mjVMKO.KBVcvShX(Unknown Source)
11 => com.rcriolom.ivwabdo.LOdEVP.onCreate(Unknown Source)
12 => android.app.Instrumentation.callApplicationOnCreate(Instrumentation.java:1013)
13 => android.app.ActivityThread.handleBindApplication(ActivityThread.java:4707)
14 => android.app.ActivityThread.handleBindApplication(Native Method)
15 => de.robv.android.xposed.XposedBridge.invokeOriginalMethodNative(Native Method)
16 => de.robv.android.xposed.XposedBridge.handleHookedMethod(XposedBridge.java:360)
17 => android.app.ActivityThread.handleBindApplication(<Xposed>)
18 => android.app.ActivityThread.-wrap1(ActivityThread.java)
19 => android.app.ActivityThread$H.handleMessage(ActivityThread.java:1405)
20 => android.os.Handler.dispatchMessage(Handler.java:102)
21 => android.os.Looper.loop(Looper.java:148)
22 => android.app.ActivityThread.main(ActivityThread.java:5417)
```

With stackTrace()

Voila.
You can see the functions called before the write method. After tracing these interval functions, you can see `RNlkfTEUX` and `lqfRafMrGew` are called right before them. And turns out they are very important functions used in decryption of the encrypted file which we will come back later on.

How to Defeat Packers

We can divide unpacking methods into two sections. Both ways lead to the decrypted file.

Dynamically

- By hooking:

    - Intercept file.delete (Java level)
    - Intercept unlink syscall (system level)
- From memory:
    - Dump the memory with gameguardian
    - Dump the memory with custom tools

Statically:

- Hands on manual unpacking

Dynamically:
Intercepting methods is the easiest way.

By hooking : Java Level
When I first encountered Anubis and realized it was dropping a file, my first solution was hooking into file.delete function.

Java.perform(function() {

var f = Java.use("java.io.File")

f.delete.implementation = function(a){

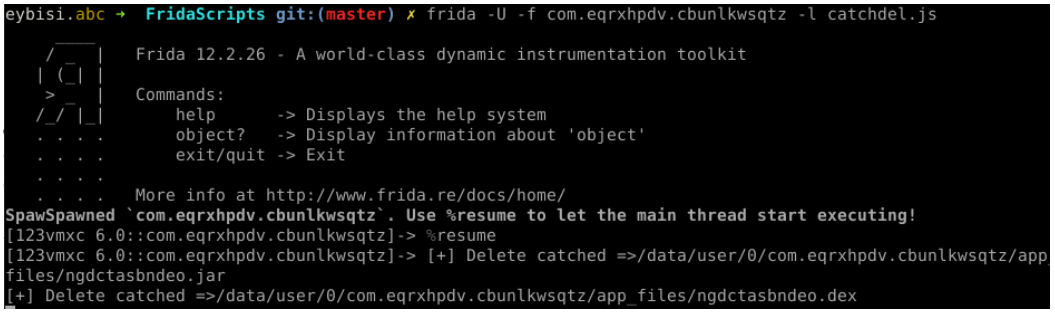
s = this.getAbsolutePath()

console.log("[+] Delete catched =>" +this.getAbsolutePath())

Java.perform(function() { var f = Java.use("java.io.File") f.delete.implementation = function(a){ s = this.getAbsolutePath() if(s.includes("jar")){ console.log("[+] Delete catched =>" +this.getAbsolutePath()) } return true } })

```
Java.perform(function() {
   var f = Java.use("java.io.File")
   f.delete.implementation = function(a){
       s = this.getAbsolutePath()
       if(s.includes("jar")){
          console.log("[+] Delete catched =>" +this.getAbsolutePath())
       }
       return true
   }
})
```

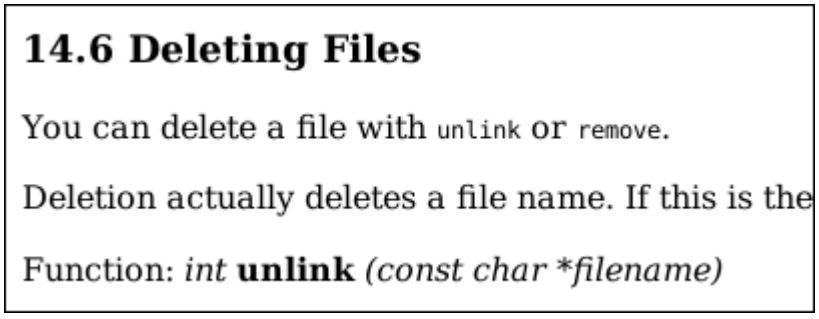
This piece of code always returns true to file.delete function. After intercepting we can pull the dropped jar file. ✔

- 

In addition to this we can automate our job with python bindings of frida and go through the folder that our target files are in. There are generally thousands of apks generated from those c&c servers. Since each of them can have different IP addresses embedded, an automated tool could make our life easier.

By Hooking : System Level

But what if malware uses native code to delete files? We can not always hook at Java level. We need to get deeper. What syscall is responsible for deleting file from file system and in libc?

- 

Unlink function takes one parameter, a pointer to filename. We can hook unlink with the help of findExportByName. Code is taken from https://www.fortinet.com/blog/threat-research/defeating-an-android-packer-with-frida.html but I tweaked little bit so deleted file will be printed.

var unlinkPtr = Module.findExportByName(null, 'unlink');

Interceptor.replace(unlinkPtr, new NativeCallback(function (a){

console.log("[+] Unlink : " + Memory.readUtf8String(ptr(a)))

var unlinkPtr = Module.findExportByName(null, 'unlink'); Interceptor.replace(unlinkPtr, new NativeCallback(function (a){ console.log("[+] Unlink : " + Memory.readUtf8String(ptr(a))) }, 'int', ['pointer']));

```
var unlinkPtr = Module.findExportByName(null, 'unlink');

Interceptor.replace(unlinkPtr, new NativeCallback( function (a){
    console.log("[+] Unlink : " + Memory.readUtf8String(ptr(a)))
    }, 'int', ['pointer']));
```

Let’s run the script.

```
eybisi.abc →  FridaScripts git:(master) ✗ frida -U -f com.eqrxhpdv.cbunlkwsqtz -l unlink.js
     ____
    / _  |   Frida 12.4.0 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/
SpaSpawned `com.eqrxhpdv.cbunlkwsqtz`. Use %resume to let the main thread start executing!
[123vmxc 6.0::com.eqrxhpdv.cbunlkwsqtz]-> %resume
[+] Unlink : /data/dalvik-cache/x86/data@app@com.eqrxhpdv.cbunlkwsqtz-1@base.apk@classes.dex
[123vmxc 6.0::com.eqrxhpdv.cbunlkwsqtz]-> [+] Unlink : /data/user/0/com.eqrxhpdv.cbunlkwsqtz/
[+] Unlink : /data/user/0/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.jar
[+] Unlink : /data/user/0/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.dex
```

We intercepted the unlink call, since our script just replaced code of original function with console.log() , file will not be deleted from the file system. ✔

From Memory:

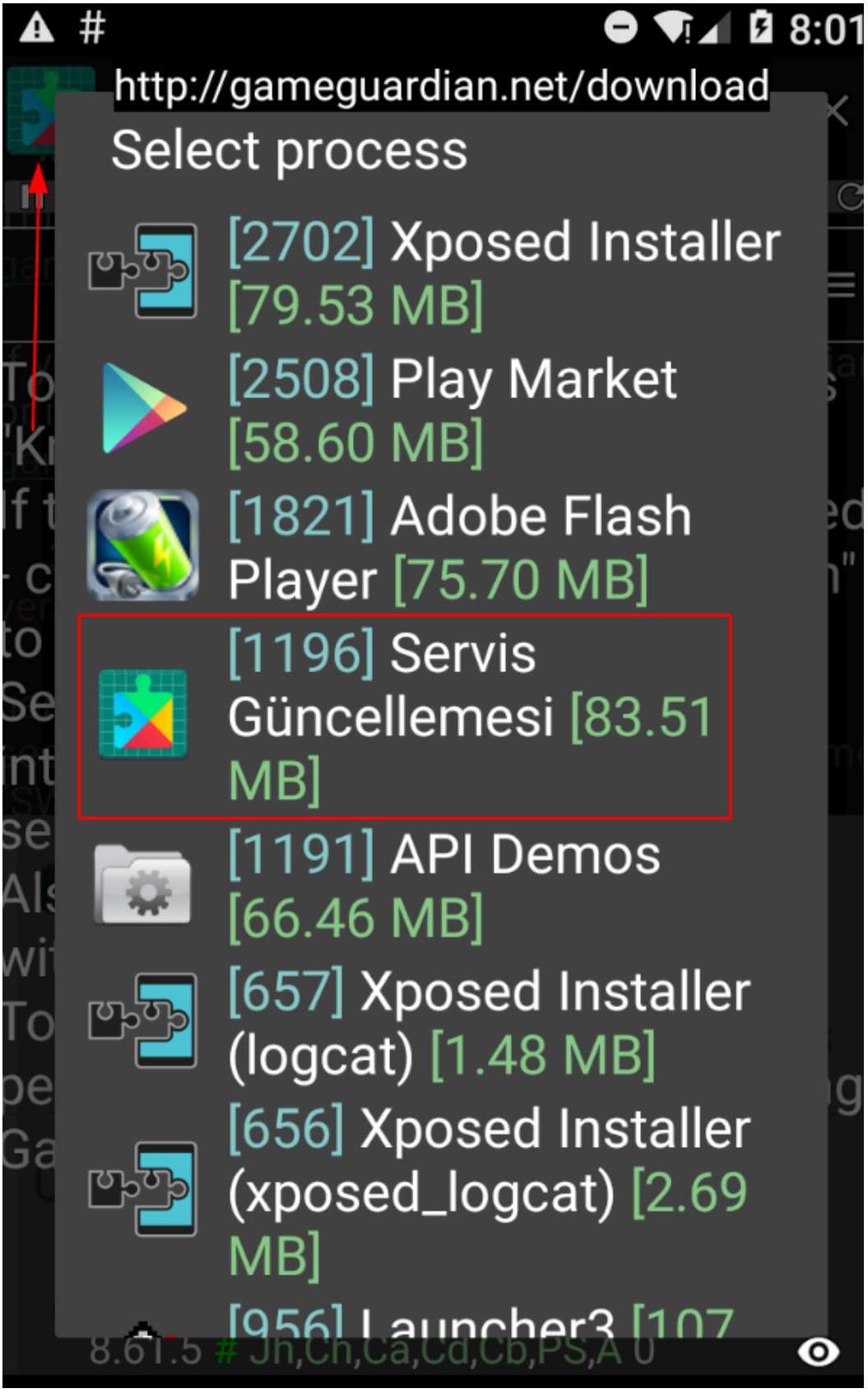
Even when file is deleted from file system because file was loaded into process, we can get trails of the deleted file from memory of that process. Since Android inherits from Linux, we can use /proc/pid folder to give us information about memory regions of a specified process. Let's look at our target with `cat /proc/pid/maps | grep dex` filtering the dex.

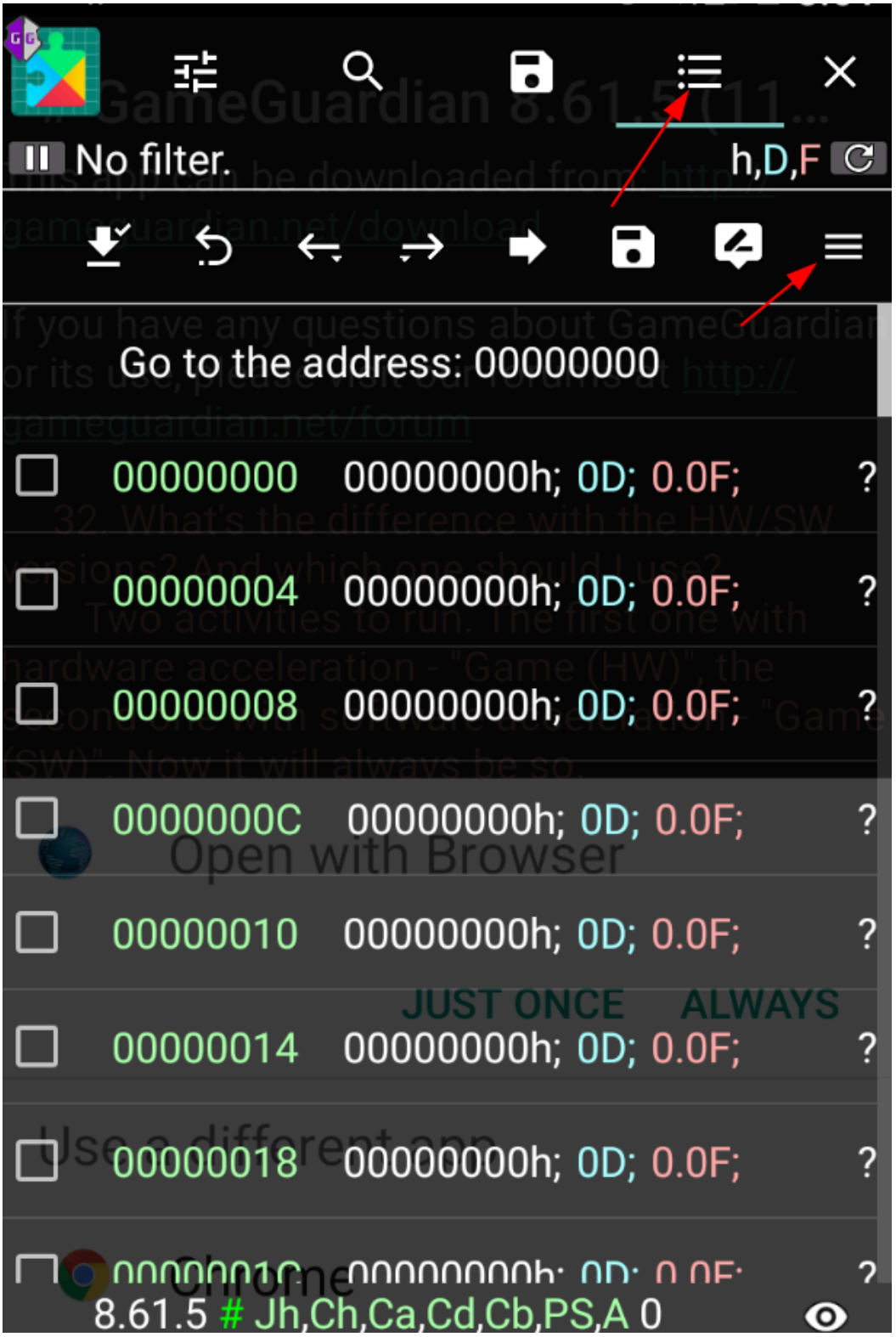
- ```
  root@vk1zcg915:/proc/1196 # cat maps | grep dex
  dfb5a000-dfbfd000 r--p 00000000 08:13 90432      /data/data/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.dex (deleted)
  dfbfd000-dfc7f000 r-xp 000a3000 08:13 90432      /data/data/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.dex (deleted)
  dfc7f000-dfc80000 rw-p 00125000 08:13 90432      /data/data/com.eqrxhpdv.cbunlkwsqtz/app_files/ngdctasbndeo.dex (deleted)
  e695d000-e69d4000 r--p 00000000 08:13 209        /data/app/com.eqrxhpdv.cbunlkwsqtz-1/oat/x86/base.odex
  e69d4000-e6a3f000 r-xp 00077000 08:13 209        /data/app/com.eqrxhpdv.cbunlkwsqtz-1/oat/x86/base.odex
  e6a3f000-e6a40000 rw-p 000e2000 08:13 209        /data/app/com.eqrxhpdv.cbunlkwsqtz-1/oat/x86/base.odex
  ```

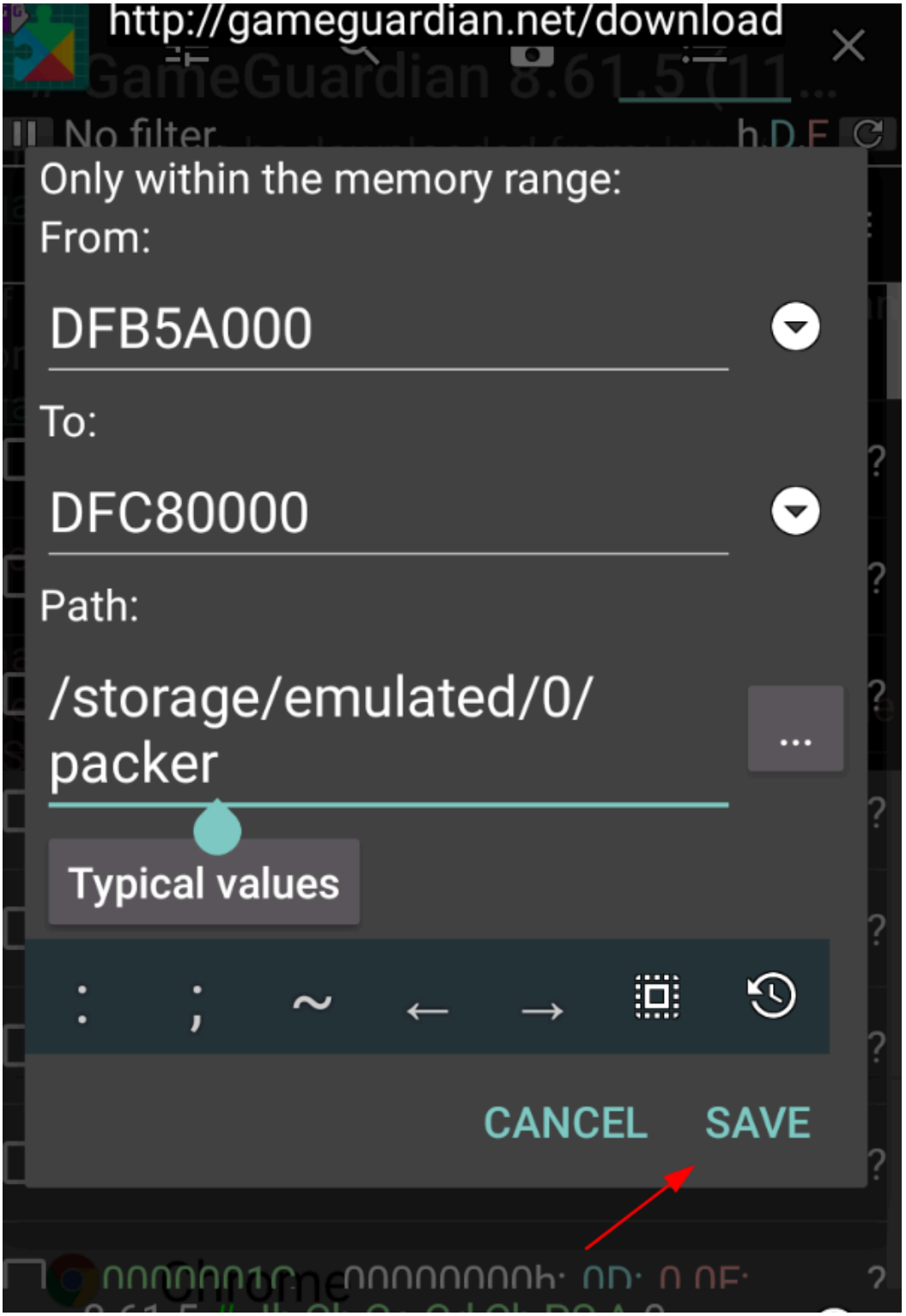
We have found the trails of dex files. Now we need to dump these sections.

Dump the Memory with Gameguardian:

First way is by "cheating" 🙂 There is a tool called GameGuardian which is used in game hacking. You can do many interesting things with GameGuardian but we will only use dump mechanism for now.
  ```

- 
8:01
http://gameguardian.net/download
Select process
[2702] Xposed Installer [79.53 MB]
[2508] Play Market [58.60 MB]
[1821] Adobe Flash Player [75.70 MB]
[1196] Servis Güncellemesi [83.51 MB]
[1191] API Demos [66.46 MB]
[657] Xposed Installer (logcat) [1.48 MB]
[656] Xposed Installer (xposed_logcat) [2.69 MB]

- 
No filter.
h,D,F
Go to the address: 00000000
00000000 00000000h; 0D; 0.0F; ?
00000004 00000000h; 0D; 0.0F; ?
00000008 00000000h; 0D; 0.0F; ?
0000000C 00000000h; 0D; 0.0F; ?
00000010 00000000h; 0D; 0.0F; ?
00000014 00000000h; 0D; 0.0F; ?
00000018 00000000h; 0D; 0.0F; ?
8.61.5 # Jh,Ch,Ca,Cd,Cb,PS,A 0

- 

Let’s start by installing and running the APK. Then launch GameGuardian and select the app name from left upmost button. Select right upmost button and the one underneath it. Now you can see dump memory option in menu. Put the hex codes of regions or select regions by clicking arrow buttons and press save. Yay!
We can pull dumped regions with :
`adb pull /storage/emulated/0/packer .` ✔
Then you will see 2 files in packer folder.
`com.eqrxhpdv.cbunlkwsqtz-dfb5a000-e0080000.bin com.eqrxhpdv.cbunlkwsqtz-maps.txt`

When examined with file command it detects our dex file as a data file.
We need to fix it by removing parts do not belong to our file.

Dump the Memory with Custom Tools:
Thanks to @theempire_h we can dump regions of memory of the target app with a C program.
https://github.com/CyberSaxosTiGER/androidDump
Here is how to dump a region with androidDump.

adb push androidDump /data/local/tmp

adb push androidDump /data/local/tmp adb shell cd /data/local/tmp chmod +x androidDump ./androidDump appname

```
adb push androidDump /data/local/tmp
adb shell
cd /data/local/tmp
chmod +x androidDump
./androidDump appname
```

It dumps 3 blobs of data. ✔

But after dumping it, `file` command still do not give us the correct type 😕 It turns out that we should modify the file a little bit. To find magic byte of dex I wrote this script.

with open(filename, 'rb') as f:

h = binascii.hexlify(content).split(b'6465780a')

h = b'6465780a' + b''.join(h)

dex = open(sys.argv[1][:-4]+".dex","wb")

dex.write(binascii.a2b_hex(h))

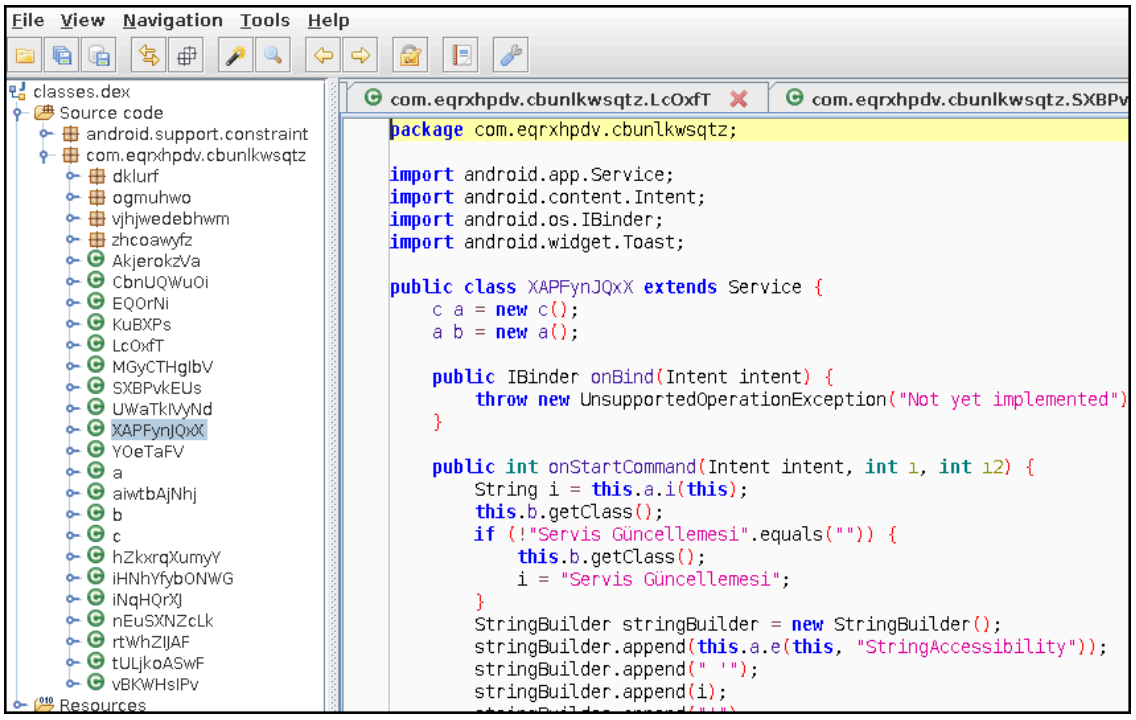
import binascii import sys filename = sys.argv[1] with open(filename, 'rb') as f: content = f.read() h = binascii.hexlify(content).split(b'6465780a') h.pop(0) h = b'6465780a' + b''.join(h) dex = open(sys.argv[1] [:-4]+".dex","wb") dex.write(binascii.a2b_hex(h)) dex.close()

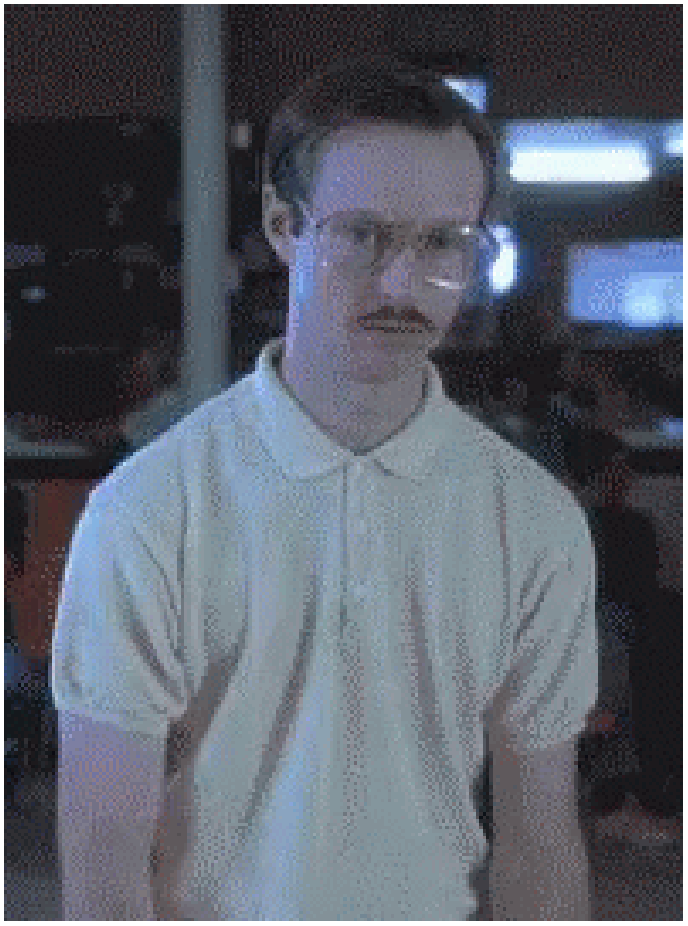
```
import binascii
import sys
filename = sys.argv[1]
with open(filename, 'rb') as f:
    content = f.read()
h = binascii.hexlify(content).split(b'6465780a')
h.pop(0)
h = b'6465780a' + b''.join(h)
dex = open(sys.argv[1][:-4]+".dex","wb")
```

```
dex.write(binascii.a2b_hex(h))
dex.close()
```

https://github.com/eybisi/nwaystounpackmobilemalware/blob/master/deDex.py

After running our script on the file, we open it.





We found our lost classes 🙂

Statically:

Here is a blog post explaining unpacking process from a different perspective.

https://sysopfb.github.io/malware,/reverse-engineering/2018/08/30/Unpacking-Anubis-APK.html

I found rc4 key with the help of stackTrace. But apparently searching for `^` value is a very efficient way to find RC4 routines for Anubis 🙂

To find rc4 key easily in JADX, here is quick tip:

- search “% length”
- right click to method you are in, press find Usage
- bArr2 will be used as rc4 key to decrypt.

Here is our sample’s decryption key as bArr2 . Does it look familiar?

```
private void RNlkfTEUX() {
    String str = "lrpilcgh spazaet ad";
    KBVcvShX();
    byte[] bArr = new byte[qTapfdmbPO.HQyWqNd];
    for (int i = 0; i < 22; i++) {
    }
    byte[] bArr2 = new byte[]{(byte) 97, (byte) -85, (byte) -52, (byte) -85, (byte) 108, (byte
    KBVcvShX(this.glNwLVgtvCo, 12636, str, this.WkzBTU);
    KBVcvShX(bArr2, bArr);
    KBVcvShX();
    str = "aineogsd itifac rgr";
    lqfRafMrGew(bArr2, bArr);
    String str2 = "nslipg srtptei jrtit";
}
```

```
eybisi.abc ➜  image anubis_manual.py "       byte[] bArr2 = new byte[]{(byte
) 97, (byte) -85, (byte) -52, (byte) -85, (byte) 108, (byte) -73, (byte) -13
, (byte) -31, (byte) 115, (byte) -32, (byte) -95, (byte) 10, (byte) 26, (byt
e) -46, (byte) -46, (byte) 64, (byte) 14, Byte.MAX_VALUE, (byte) -3, (byte)
18, (byte) 102, (byte) 27, (byte) 73, (byte) -46, (byte) 28, (byte) -115, (b
yte) -120, (byte) 22, (byte) 86, (byte) 116, (byte) -38, (byte) -123, (byte)
 121, (byte) -16, (byte) -121};
        " mediumcrop

Key = : b'a\xab\xcc\xabl\xb7\xf3\xe1s\xe0\xa1\n\x1a\xd2\xd2@\x0e\x7f\xfd\x12
f\x1bI\xd2\x1c\x8d\x88\x16Vt\xda\x85y\xf0\x87'
Zip header found : writing decrypted data to decryted.dex
eybisi.abc ➜  image file decrypted.dex
decrypted.dex: Zip archive data, at least v2.0 to extract
eybisi.abc ➜  image
```

After decrypting and unzipping, we get our dex.

```
public class a {
    public final String a = "http://usom.gov.tr/";
    public final String b = "";
    public final String c = "/olo/a1.php";
    public final String d = "https://t.me/japans587";
    public final String e = "/olo/a2.php";
    public final boolean f = true;
    public final boolean g = false;
    public final String h = "service";
    public final String i = "new";
    public final String j = "new";
    public final String k = "Servis Güncellemesi";
    public final int l = 10000;
    public final String m = "";
    public final int n = 12000;
    public boolean o = false;
    public String p = "<urlImage>";
    public boolean q = true;
    public int r = 300;
    public int s = 0;
    public int t = 1;
}
```

After extracting the config, there is one more step to get the address of c&c server. Malware gets page source of the telegram address and changes Chinese characters with ASCII letters. It then processes the base64 string. After

decoding base64, it uses `service` to decrypt data that encrypted with rc4 scheme. Here is a snippet for decrypting Chinese chars to c&c addresses.

https://github.com/eybisi/nwaystounpackmobilemalware/blob/master/solve_chinese.py

```
eybisi.abc ➜  Packer python solve_chinese.py "苏尔的开始死语并吸个
中是而比中阿斯并寞死符拉的在符比语并化拉屎拉吸并屎在寞比中意化拉
寞并妈个语努音拉中死符比亡阿号个要拼斯个语在斯个语需你比要拉脚死需
肏肏苏尔苏尔完" service
http://katkatkipoyran.site
```

I managed to decrypt the Anubis payload with Androguard without running the APK in an emulator! After dumping the dex file, my script will find the config class printing the c2 and the encryption key. Config class is in one of the `a,b or c` or in `ooooooooooooo{0,2}o` classes in newer versions.
By checking counts of “this” keywords in class source code I managed to decrypt all versions of anubis (lazy :P). Here is output of my script to get c2 and key from an Anubis sample.
https://github.com/eybisi/nwaystounpackmobilemalware/blob/master/getc2_imp.py

```
eybisi.abc ➜  Packer getc2_imp.py e01ed0befbc50eeedcde5b5c07bf8a51ab39c5b20ee6e1f5afe04e161d07
2f1d.apk
Key : [97, 171, 204, 171, 108, 183, 243, 225, 115, 224, 161, 10, 26, 210, 210, 64, 14, 127, 25
3, 18, 102, 27, 73, 210, 28, 141, 136, 22, 86, 116, 218, 133, 121, 240, 135]
[+] Filesize = 116515
[+] Zip header found. Decrypted payload : e01ed0befbc50eeedcde5b5c07bf8a51ab39c5b20ee6e1f5afe0
4e161d072f1d.apk.decrypted
[+] C&C and Key found !
C&C: https://t.me/japans587
Key: service
```

Conclusion

There are many ways to unpack mobile malware and trace packing mechanisms. We might see `dalvik.system.InMemoryDexClassLoader` used in the future. If this is used, delete hooks will not be able to catch dropped files because everything will be done in memory 🙂 But dumping memory will catch these methods. Knowing different ways always helps. If you have any question feel free to ask in comment section or through @0xabc0

Cheers.

Special thanks to @godelx0

Links & References

Dropper sample:
3c35f97b9000d55a2854c86eb201bd467702100a314486ff1dbee9774223bf0e
Anubis sample:
e01ed0befbc50eeedcde5b5c07bf8a51ab39c5b20ee6e1f5afe04e161d072f1d
https://codeshare.frida.re/@razaina/get-a-stack-trace-in-your-hook/
https://www.fortinet.com/blog/threat-research/defeating-an-android-packer-with-frida.html
https://medium.com/@fs0c131y/reverse-engineering-of-the-anubis-malware-part-1-741e12f5a6bd3
All materials:
https://github.com/eybisi/nwaystounpackmobilemalware
