Reverse Android malware like a Jedi Master

Axelle Apvrille, Fortinet (aapvrille@fortinet.com)

Abstract

In this paper, we use 4 new Android reverse engineering tools - Dexcalibur, House, MobSF and Quark - over malicious samples of 2020/2021. We explain how to use or customize the tools at best, and highlight their strengths and limitations.

Presenting tools

Every Jedi padawan has reversed Android malware using *Apktool, Baksmali* and a disassembler. Some of the more experienced have probably written automated plugins or scripts (for Radare, JEB) or implemented hooks using *Frida.* Those tools are excellent, useful to padawan and masters. But there are new tools: *Dexcalibur* (2019), *House* (2018), *Quark* (2019) and *MobSF* (2015). We focus on those 4 tools and their use to Android reverse engineering (RE).

- **Dexcalibur** [1] and **House** [2] can both be seen as web front-ends to Frida. They help set or customize Frida hooks on interesting functions. With Dexcalibur, Frida hooks can be enabled by simple mouse clicks. There is little need to know how Frida hooks are implemented, except when you need to customize them. With House, the approach remains close to the implementation: the web interface loads Frida templates. Those can be customized at will, and run. Only a few tasks (e.g class enumeration, HTTP access monitoring) are press-button style. In this paper, we use **Dexcalibur v0.7.9** and House cloned from its repository in March 2021.
- **MobSF**. It is an open source *"automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, malware analysis and security assessment framework capable of performing static and dynamic analysis"* [3]. It features both *static analysis*, helpful for overview of the sample and *dynamic analysis*, based on Frida hooks. In this paper, we use **MobSF 3.4.4 beta**.
- **Quark** is different, and only works for *static* analysis [4]. Quark's engine parses the sample's code to detect suspicious combinations of API calls and permissions. The combinations are described in *rules.* Over 150 rules

are shared in a GitHub repository, additional ones can be created easily. In this paper, we use **Quark version 21.5.1**.

Test samples

The Android malicious samples we refer to in this paper are listed in the table below. They have been selected because (1) they are recent and (2) exhibit particular features (packing, native library…).

Table 1: *List of test samples used in the paper*

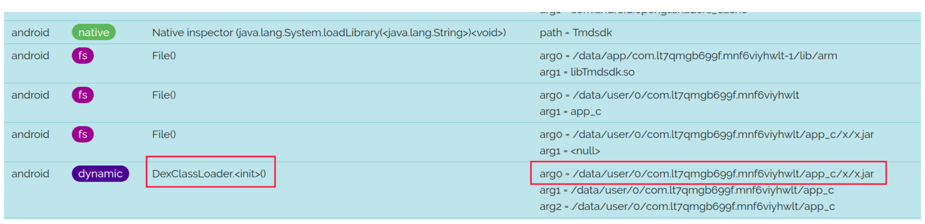
SHA256 Sum	Name	Date
1a8c17ad1a790554278b055bdb946d4597ba9af6b[illegible]	Android/EventBot [5]	April 2020
f82d6f24af2a4444c696c64060582d8aed6280da5[illegible]	Android/Sandr [6]	June 2020
f699f9e50e8401943321d757a9c1bab367473f10[illegible]	Riskware/Tenpack see here	February 2021
fd5f7648d03eec06c447c1c562486df10520b93ad[illegible] or 9379f91ddd9326bc1a8cf2fe4a22951d0859b2b7f88ffe68b023a97d59130810	Android/Flubot [7]	March 2021
a25363b68faa8188b99622d8909921a4026ea724[illegible]	Android/Oji details	May 2021
aad80d2ad20fe318f19b6197b76937bf7177dbb1[illegible]	Android/MoqHao details	May 2021
8810ca80d21173528be71109cd9e5a73afce98a08[illegible]	Android/Kfapp details	May 2021
dc215663af92d41f40f36088ec1b850b81092ea94[illegible], 0da75ac97f4ec8954a961c270bcbe75bd2671c65[illegible]	Android/Alien see this video	Sept 2020, May 2021

Using the 4 tools for common RE tasks

Unpacking malware

Malware analysts commonly encounter **packed Android malware** [8] (e.g ApkProtect, Bangcle…). The malicious payload is hidden in the APK, "unpacked" by a *packer* whose sole goal is to harden reverse engineering and conceal the payload. The most common implementation consists in using `DexClassLoader` to *dynamically load a hidden DEX file*. The 4 tools typically detect use of `DexClassLoader`. Quark detects it with a simple rule (example). The other 3 rely on Frida hooks.

In the Android API, the first argument of `DexClassLoader`'s constructor is the path of the DEX to load dynamically. Dexcalibur, House and MobSF show method arguments, hence they are able to display the path of the DEX. Then, the analyst just needs to retrieve the executable at this location, using `adb pull` and analyze it.



android	native	Native inspector (java.lang.System.loadLibrary(<java.lang.String>)<void>)	path = Tmdsdk

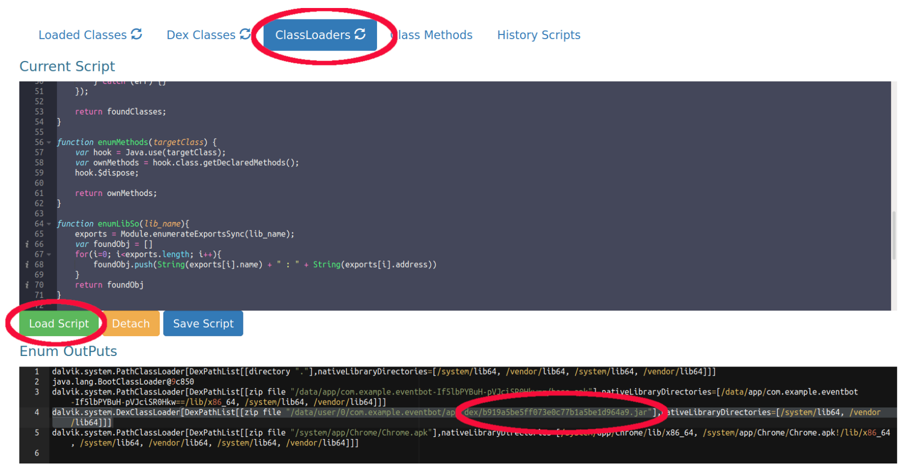
android	fs	File()	arg0 = /data/app/com.lt7qmgb699f.mnf6viyhwlt-1/lib/arm
			arg1 = libTmdsdk.so

android	fs	File()	arg0 = /data/user/0/com.lt7qmgb699f.mnf6viyhwlt
			arg1 = app_c

android	fs	File()	arg0 = /data/user/0/com.lt7qmgb699f.mnf6viyhwlt/app_c/x/x.jar
			arg1 = <null>

android	dynamic	DexClassLoader.<init>()	arg0 = /data/user/0/com.lt7qmgb699f.mnf6viyhwlt/app_c/x/x.jar
			arg1 = /data/user/0/com.lt7qmgb699f.mnf6viyhwlt/app_c
			arg2 = /data/user/0/com.lt7qmgb699f.mnf6viyhwlt/app_c

Figure 1: Dexcalibur detects live use of DexClassLoader in a sample of Riskware/Tenpack.



```
    });

    return foundClasses;
}

function enumMethods(targetClass) {
    var hook = Java.use(targetClass);
    var ownMethods = hook.class.getDeclaredMethods();
    hook.$dispose;

    return ownMethods;
}

function enumLibSo(lib_name){
    exports = Module.enumerateExportsSync(lib_name);
    var foundObj = []
    for(i=0; i<exports.length; i++){
        foundObj.push(String(exports[i].name) + " : " + String(exports[i].address))
    }
    return foundObj
}
```

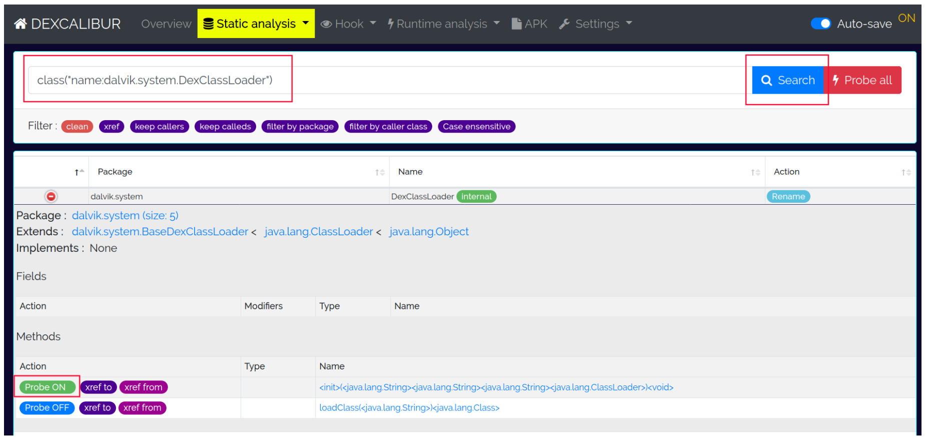
```
dalvik.system.PathClassLoader[DexPathList[[directory "."],nativeLibraryDirectories=[/system/lib64, /vendor/lib64, /system/lib64, /vendor/lib64]]]
java.lang.BootClassLoader@9c850
dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/com.example.eventbot-If5lbPYBuH-pVJciSR0Hkw==/base.apk"],nativeLibraryDirectories=[/data/app/com.example.eventbot
    -If5lbPYBuH-pVJciSR0Hkw==/lib/x86_64, /system/lib64, /vendor/lib64]]]
dalvik.system.DexClassLoader[DexPathList[[zip file "/data/user/0/com.example.eventbot/app_dex/b919a5be5ff073e0c77b1a5be1d964a9.jar"],nativeLibraryDirectories=[/system/lib64, /vendor
    /lib64]]]
dalvik.system.PathClassLoader[DexPathList[[zip file "/system/app/Chrome/Chrome.apk"],nativeLibraryDirectories=[/system/app/Chrome/lib/x86_64, /system/app/Chrome/Chrome.apk!/lib/x86_64
    , /system/lib64, /vendor/lib64, /system/lib64, /vendor/lib64]]]
```

Figure 2: With House, go to the Enumeration/ClassLoaders tab. This displays a customizable Frida script. Actually, it does not need any modification: press the "Load Script" button. The Enum Outputs show Android/EventBot loads dynamically a JAR.

With **Dexcalibur**, we have to make sure `DexClassLoader` is probed (by default, it is). If not, click on the "Probe ON" button. If for some reason the hook is not present at all, first search for it in the "Static Analysis" tab.

Beware not to hook only `DexClassLoader`: there are similar class loaders e.g `PathClassLoader` (replacing the deprecated `DexFile`) and `InMemoryDexClassLoader`. The latter, introduced in Android 8.0, does not load a payload DEX from a *file* but from *memory*. Consequently, there is no full path nor file to grab on the device. The solution in that case is to add a Frida hook that automatically dumps the payload byte array to a file.

Besides hooking class loaders, there are a few other strategies to unpack dynam-



DEXCALIBUR
Overview
Static analysis
Hook
Runtime analysis
APK
Settings
Auto-save ON
Search
Probe all
Filter : clean xref keep callers keep calleds filter by package filter by caller class Case ensensitive

```
class("name:dalvik.system.DexClassLoader")
```

	Package	Name	Action
	dalvik.system	DexClassLoader internal	Rename

Package : dalvik.system (size: 5)
Extends : dalvik.system.BaseDexClassLoader < java.lang.ClassLoader < java.lang.Object
Implements : None

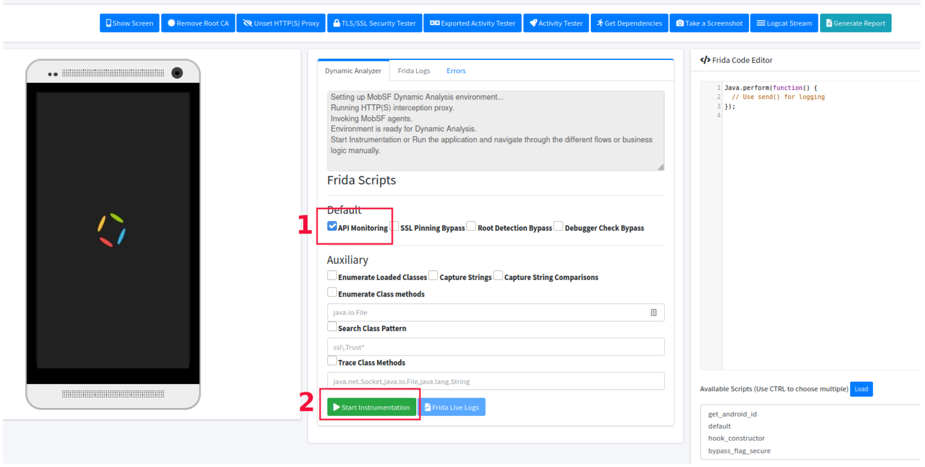
Fields

Action	Modifiers	Type	Name

Methods

Action	Type	Name
Probe ON xref to xref from		<init>(<java.lang.String><java.lang.String><java.lang.String><java.lang.ClassLoader>)<void>
Probe OFF xref to xref from		loadClass(<java.lang.String>)<java.lang.Class>

Figure 3: Search for DexClassLoader in the Static Analysis tab and select "Probe ON" to hook its constructor.



```
Setting up MobSF Dynamic Analysis environment...
Running HTTP(S) interception proxy.
Invoking MobSF agents.
Environment is ready for Dynamic Analysis.
Start Instrumentation or Run the application and navigate through the different flows or business
logic manually.
```

Frida Scripts

Default

[✓ API Monitoring] SSL Pinning Bypass Root Detection Bypass Debugger Check Bypass

Auxiliary

Enumerate Loaded Classes Capture Strings Capture String Comparisons

Enumerate Class methods: java.io.File

Search Class Pattern: ssl\.Trust*

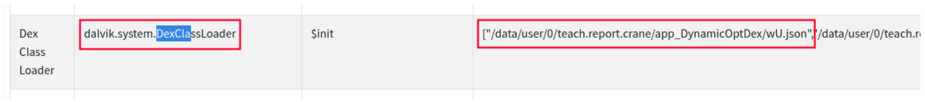
Trace Class Methods: java.net.Socket,java.io.File,java.lang.String

```
Java.perform(function() {
  // Use send() for logging
});
```

Available Scripts (Use CTRL to choose multiple)

get_android_id
default
hook_constructor
bypass_flag_secure

Figure 4: In MobSF, click API Monitoring, then Start Instrumentation. A new button "Live API Monitor" appears at the top, click on it, and see next image.



Dex Class Loader	dalvik.system.DexClassLoader	$init	["/data/user/0/teach.report.crane/app_DynamicOptDex/wU.json","/data/user/0/teach.r

Figure 5: Live API monitoring in MobSF detects Android/Alien loads a DEX.

ically [9]: hooking file creation or deletion (at Java level or system level) [10], dumping the memory (e.g. https://github.com/enovella/fridroid-unpacker).

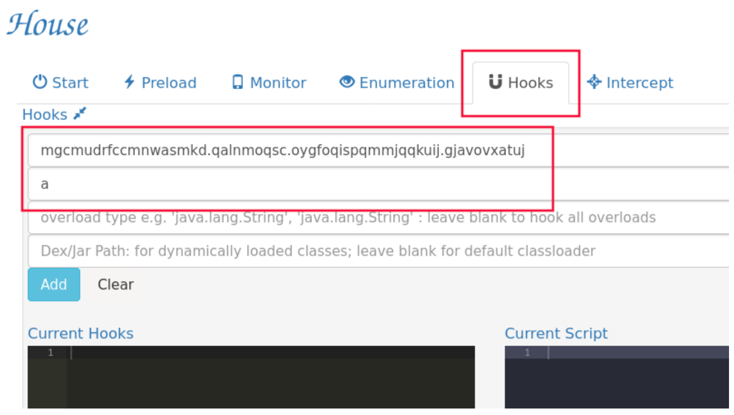
Some clever malware load DEX dynamically from *native libraries*. This is the case of *Android/MoqHao*. It is difficult to analyze with the 4 tools (see later). Finally, note VM-based packers and packers for native code are on their way [11], but haven't been used in malware yet.

Analyzing dynamically loaded DEX

All 4 tools have **difficulties to access/hook inside a dynamically loaded DEX**. This is an issue for RE because this dynamically DEX typically holds the interesting malicious features (payload), whereas the wrapping DEX (packer) is of no importance (apart hardening reverse).

There are two solutions:

1. **Manually**. Retrieve the payload DEX (see previous paragraph), and analyze it with a disassembler. Note only Quark is able to process the payload DEX, the other 3 tools do not support DEX. . .
2. **Automatically**. Write a Frida hook which hooks *inside* the dynamically loaded DEX. Writing such a hook requires experience (example). Then, load the hook in a dynamic analysis tool. In theory, it should work, except I have only managed to get it work with *House* (not Dexcalibur, nor MobSF), and even with House, the process is not reliable (sometimes it works, sometimes not!).



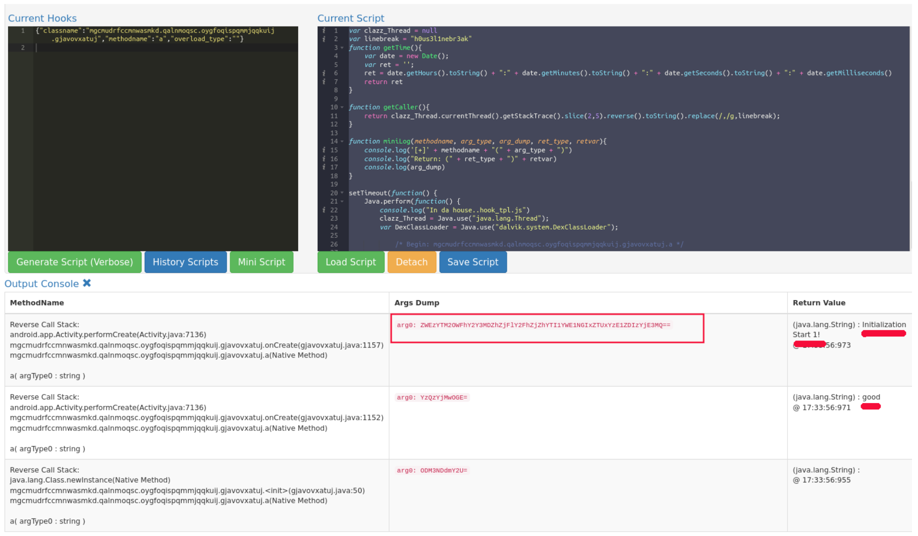
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj

a

overload type e.g. 'java.lang.String', 'java.lang.String' : leave blank to hook all overloads

Dex/Jar Path: for dynamically loaded classes; leave blank for default classloader

Figure 6: Configuring House to hook inside dynamic code. Fill class name and method. Despite the name do not fill the entry DEX/Jar path. . .



```
{"classname":"mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij
    .gjavovxatuj","methodname":"a","overload_type":""}
```

```
var clazz_Thread = null
var linebreak = "h0us3l1nebr3ak"
function getTime(){
    var date = new Date();
    var ret = '';
    ret = date.getHours().toString() + ":" + date.getMinutes().toString() + ":" + date.getSeconds().toString() + ":" + date.getMilliseconds()
    return ret
}

function getCaller(){
    return clazz_Thread.currentThread().getStackTrace().slice(2,5).reverse().toString().replace(/,/g,linebreak);
}

function miniLog(methodname, arg_type, arg_dump, ret_type, retvar){
    console.log('[+]' + methodname + "(" + arg_type + ")")
    console.log("Return: (" + ret_type + ")" + retvar)
    console.log(arg_dump)
}

setTimeout(function() {
    Java.perform(function() {
        console.log("In da house..hook_tpl.js")
        clazz_Thread = Java.use("java.lang.Thread");
        var DexClassLoader = Java.use("dalvik.system.DexClassLoader");

            /* Begin: mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.a */
```

MethodName	Args Dump	Return Value

Reverse Call Stack:
android.app.Activity.performCreate(Activity.java:7136)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.onCreate(gjavovxatuj.java:1157)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.a(Native Method)
 	
a(argType0 : string)
	arg0: ZWEzYTM2OWFhY2Y3MDZhZjFlY2FhZjZhYTI1YWE1NGIxZTUxYzE1ZDIzYjE3MQ==	(java.lang.String) : Initialization Start 1!
		[redacted]56:973

Reverse Call Stack:
android.app.Activity.performCreate(Activity.java:7136)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.onCreate(gjavovxatuj.java:1152)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.a(Native Method)
 	
a(argType0 : string)
	arg0: YzQzYjMwOGE=	(java.lang.String) : good
		@ 17:33:56:971

Reverse Call Stack:
java.lang.Class.newInstance(Native Method)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.<init>(gjavovxatuj.java:50)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.gjavovxatuj.a(Native Method)
 	
a(argType0 : string)
	arg0: ODM3NDdmY2U=	(java.lang.String) :
		@ 17:33:56:955

Figure 7: This screenshot of House shows live string de-obfuscation inside dynamically loaded DEX of Android/Alien.

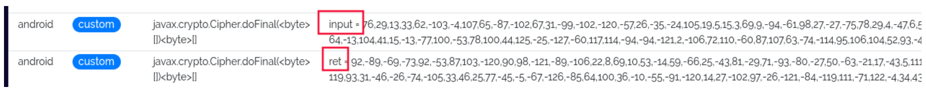
De-obfuscate strings

Another typical task for malware analysts is **string de-obfuscation**. It can be done *statically* with a stand-alone program, implemented after close reversing of the obfuscation code. Some advanced decompilers, such as JEB, automatically perform easy decryption/de-obfuscation, or allow the execution of custom scripts.

The other way to go is *dynamically*. This is much faster: nearly no RE or code to perform. The downside is, of course, that it will only de-obfuscate code it runs into. The strategy consists in hooking the de-obfuscation functions (to find via static reverse engineering) or, if standard encryption is used, hooking methods such as `Cipher.doFinal()` to get the plain text.

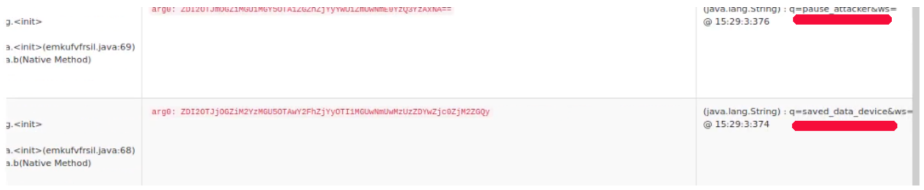
With Dexcalibur, search for the de-obfuscation method in the Static Analysis tab, and probe it. Then, slightly edit the hookClick on the "Probe OFF" button to turn it "ON." This adds the corresponding Frida hook. Then, in the Hook tab, select the hook and slightly edit its code to display the output (add the output variable to "data" JSON item).

```
var ret = meth_xxx.call(this , arg0);
/* In data, add "ret" to display the output */
send({ id:"yyyyyyy=",
           msg:"javax.crypto.Cipher.doFinal(<byte>[])<byte>[]",
       data:{ret}, action:"None before", after:true  });
```



```
android  custom  javax.crypto.Cipher.doFinal(<byte>  input = 76,29,13,33,62,-103,-4,107,65,-87,-102,67,31,-99,-102,-120,-57,26,-35,-24,105,19,5,15,3,69,9,-94,-61,98,27,-27,-75,78,29,4,-47,6,5
                 []<byte>[]                          64,-13,104,41,15,-13,-77,100,-53,78,100,44,125,-25,-127,-60,117,114,-94,-94,-121,2,-106,72,110,-60,87,107,63,-74,-114,95,106,104,52,93,-4
android  custom  javax.crypto.Cipher.doFinal(<byte>  ret = 92,-89,-69,-73,92,-53,87,103,-120,90,98,-121,-89,-106,22,8,69,10,53,-14,59,-66,25,-43,81,-29,71,-93,-80,-27,50,-63,-21,17,-43,5,111
                 []<byte>[]                          119,93,31,-46,-26,-74,-105,33,46,25,77,-45,-5,-67,-126,-85,64,100,36,-10,-55,-91,-120,14,27,-102,97,-26,-121,-84,-119,111,-71,122,-4,34,43
```

Figure 8: In Dexcalibur, the hooked encryption methods show the cipher text (input) and plain text (ret) for Android/Ksapp.



```
g.<init>                               arg0: ZDI2OTJmOGZiMGU1MGY5OTA1ZGZhZjYyYWU1ZmUwNmE0YZQ3YZAxNA==     (java.lang.String) : q=pause_attacker&ws=
                                                                                                         @ 15:29:3:376  [redacted]
a.<init>(emkufvfrsil.java:69)
a.b(Native Method)

g.<init>                               arg0: ZDI2OTJjOGZiM2YzMGU5OTAwY2FhZjYyOTI1MGUwNmUwMzUzZDYwZjc0ZjM2ZGQy     (java.lang.String) : q=saved_data_device&ws=
                                                                                                         @ 15:29:3:374  [redacted]
a.<init>(emkufvfrsil.java:68)
a.b(Native Method)
```

Figure 9: In this sample of Android/Alien, we configured House to hook the de-obfuscating function. Each time the function is called, House displays both the input (obfuscated) and the output (plain text).

Communication with the Command and Control (CnC) server

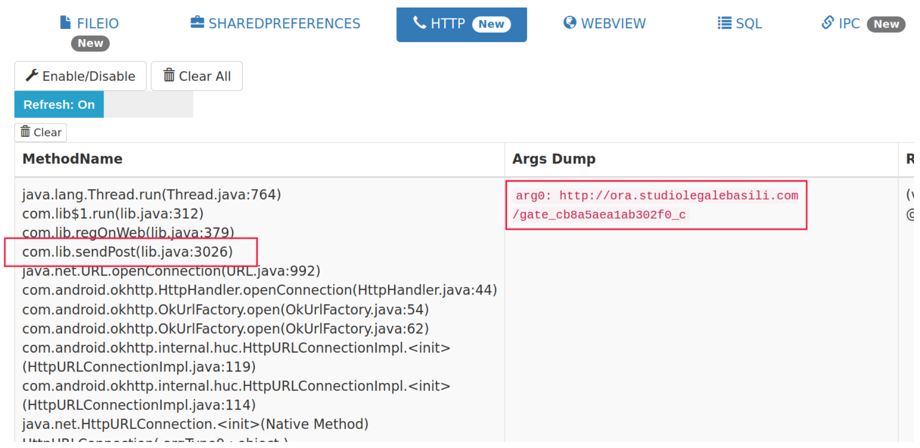
No tool is guaranteed to spot the IP address of the CnC. Nevertheless, they can help. For instance, MobSF's static analysis lists domains and IP addresses the malware uses. In practice, there are often False Positives (see this section), and sometimes, the tool completely misses the CnC altogether, especially when the malware is packed.

In some malware, the malicious code conceals the IP address of the CnC or any remote host it contacts. Dynamic analysis is helpful in such circumstances because we automatically get the resulting IP address / names. For example, with Dexcalibur, add hooks for the `URL` constructor and the `openConnection()` method.

Bypass Anti-Reverse tricks

Android malware sometimes attempt to detect emulators, debuggers, rooted environments, or even Frida hooks. While this has no effect on static analysis, it hardens dynamic analysis. Typically, protections are based on use of specific APIs (e.g. `isDebuggerConnected()`), presence of given files (e.g `su`), named pipes, processes, symbols or applications (e.g `com.noshufou.android.su`), debug / default values (e.g `15555215554` as phone number on emulators), stack trace or libc-level checks see Rootbeer, OWASP, strong-frida and [12].

For a malware analyst, the first step consists in detecting those protections. To do so, MobSF relies on APKiD, a tool which focuses on identifying packers, obfuscators, anti-vm and anti-debug. With Quark, we can typically implement a rule to detect use of `isDebuggerConnected`. Unfortunately, Quark will be

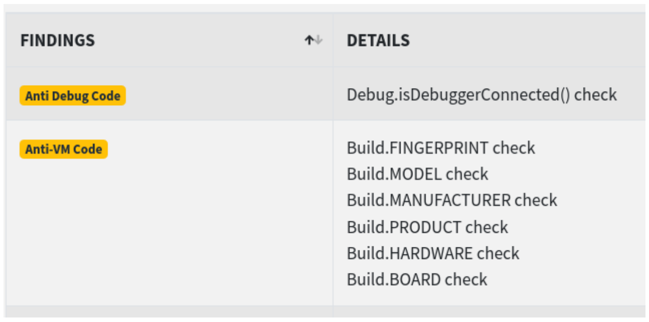


MethodName	Args Dump

java.lang.Thread.run(Thread.java:764)
com.lib$1.run(lib.java:312)
com.lib.regOnWeb(lib.java:379)
com.lib.sendPost(lib.java:3026)
java.net.URL.openConnection(URL.java:992)
com.android.okhttp.HttpHandler.openConnection(HttpHandler.java:44)
com.android.okhttp.OkUrlFactory.open(OkUrlFactory.java:54)
com.android.okhttp.OkUrlFactory.open(OkUrlFactory.java:62)
com.android.okhttp.internal.huc.HttpURLConnectionImpl.<init>
(HttpURLConnectionImpl.java:119)
com.android.okhttp.internal.huc.HttpURLConnectionImpl.<init>
(HttpURLConnectionImpl.java:114)
java.net.HttpURLConnection.<init>(Native Method)
	arg0: http://ora.studiolegalebasili.com
	/gate_cb8a5aea1ab302f0_c

Figure 10: House has been configured to enable HTTP monitoring (Monitor tab, then Enable/Disable button). Here it shows Android/EventBot contacts hxxp://ora.studiolegalebasili.com/. Better than Wireshark, House shows the call occurs from a method named com.lib.sendPost. We can hook that method to display every packet that gets sent.

inefficient by design on most other tricks (Quark cannot detect specific strings, files, pipes, processes. . .).

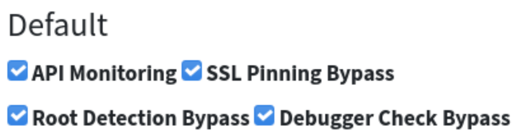


FINDINGS	DETAILS
Anti Debug Code	Debug.isDebuggerConnected() check
Anti-VM Code	Build.FINGERPRINT check Build.MODEL check Build.MANUFACTURER check Build.PRODUCT check Build.HARDWARE check Build.BOARD check

Figure 11: MobSF uses APKiD to detect anti-debug and anti-VM tricks, here in Android/Ghimob.

Once the protection is identified, we must try to *bypass* it. [13] hosts a collection of Frida bypasses. The hooks can be added/loaded to Dexcalibur, House or MobSF. MobSF even comes with built-in detection of root environment and debuggers.

Frida *native-level* hooks (e.g here) are not supported yet by Dexcalibur, and their status is uncertain for House and MobSF. If such as hook is required, we have to run our own Frida server and client. Also, some anti-Frida techniques, such



Default

✓ API Monitoring ✓ SSL Pinning Bypass

✓ Root Detection Bypass ✓ Debugger Check Bypass

Figure 12: Bypassing root/debugger detection is just a matter of clicking the right box in MobSF.

as `libc.so` tampering check [14], are way more difficult to bypass. Fortunately, we haven't ever seen such advanced techniques in malware.

Practical cases

Android/Oji.G!worm

The Android worm *Oji* has re-surfaced in May 2021, with a fake COVID-19 vaccine registration campaign. It propagates via SMS to victims located in India and using a specific operator. It also asks the victim to share the app on WhatsApp. Also, the sample I analyzed uses AES/CBC, but the code is wrong: the cipher text does not decrypt as it is not a multiple of 16.

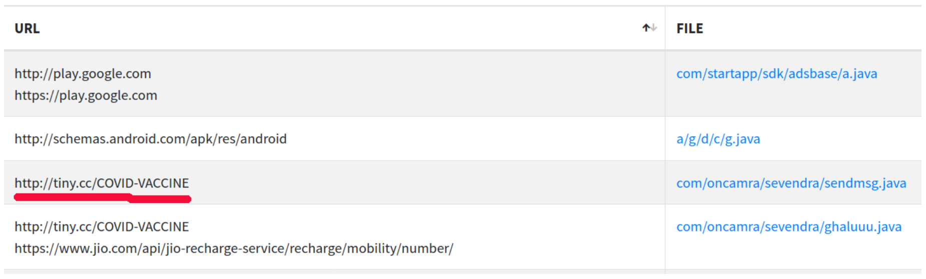
First, **all tools are heavily impacted by the fact they are unable to tell the difference between code from third party kits** (for example, this sample uses `com.startapp`, an in-app ads SDK) and the malicious part (`com.omcamra.sevendra`). As far as I know, the only tool which takes this into account is my own tool.

Quark manages to detect the malware reads contacts (under a slightly misleading crime name *"Read sensitive data(SMS, CALLLOG, etc)"*). The other results are not very relevant and polluted by alarms raised by third party kits. Also, the general threat level "Moderate Risk" and "total score 153" do not seem appropriate for malware analysis.

MobSF's static analysis is more helpful, but still polluted by references to third party kits. At least, the tables display the path of the code which performs the action, so it is easier to rule out third party code (e.g. StartApp). As for its dynamic analysis, it works well but it is unpleasant to use because of silly ergonomics issues (small columns, difficult to scroll, no search etc - probably this will be improved in future versions).

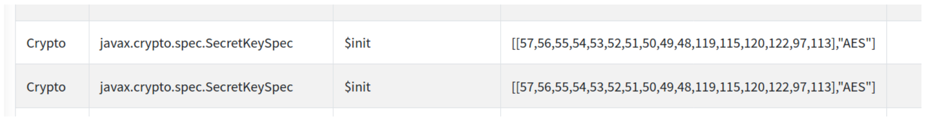
In this particular sample, the decryption fails (bug of the code?), but this is difficult to spot with MobSF. It does show in Android's *logcat* but there are many lines, and no particular highlight…

```
6-04 11:07:58.262 19083 19083 W System.err: java.lang.Exception: [decrypt]
  error:1e00006a:Cipher functions:OPENSSL_internal:DATA_NOT_MULTIPLE_OF_BLOCK_LENGTH
```



URL	FILE
http://play.google.com https://play.google.com	com/startapp/sdk/adsbase/a.java
http://schemas.android.com/apk/res/android	a/g/d/c/g.java
http://tiny.cc/COVID-VACCINE	com/oncamra/sevendra/sendmsg.java
http://tiny.cc/COVID-VACCINE https://www.jio.com/api/jio-recharge-service/recharge/mobility/number/	com/oncamra/sevendra/ghaluuu.java

Figure 13: MobSF detects the propagation URL hxxp://tiny.cc/COVID-VACCINE the sample uses.

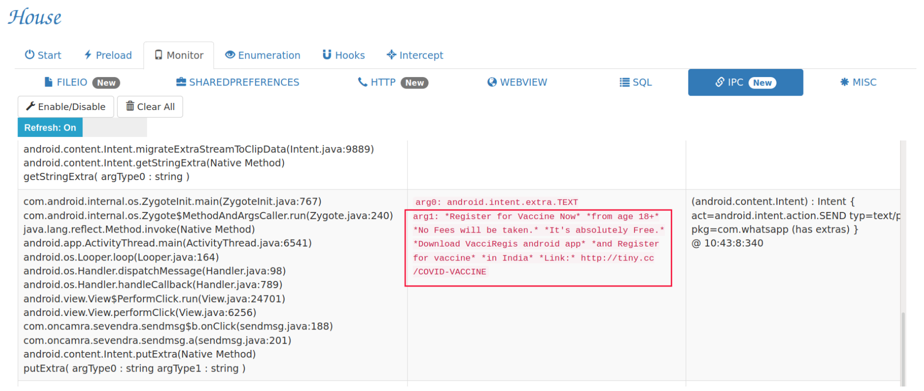


Crypto	javax.crypto.spec.SecretKeySpec	$init	[[57,56,55,54,53,52,51,50,49,48,119,115,120,122,97,113],"AES"]
Crypto	javax.crypto.spec.SecretKeySpec	$init	[[57,56,55,54,53,52,51,50,49,48,119,115,120,122,97,113],"AES"]

Figure 14: MobSF live API monitoring displays the AES key of Android/Oji. The same can be achieved with Dexcalibur and hooking SecretKeySpec.

```
06-04 11:07:58.262 19083 19083 W System.err:
  at com.oncamra.sevendra.ghaluuu.c(ghaluuu.java:240)
```

House is not the best choice for malware reconnaissance, but its monitoring section is helpful for the sample. For example, monitoring the *IPC section* clearly shows the message copied for WhatsApp, something the other tools won't show easily.



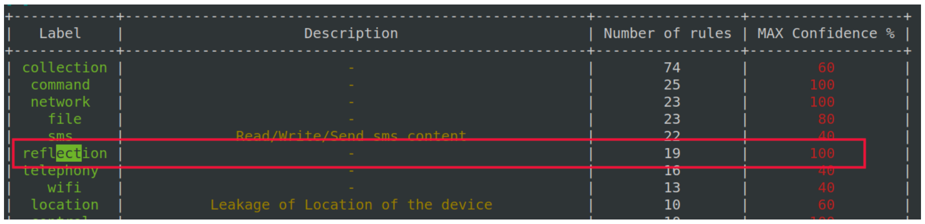
```
android.content.Intent.migrateExtraStreamToClipData(Intent.java:9889)
android.content.Intent.getStringExtra(Native Method)
getStringExtra( argType0 : string )

com.android.internal.os.ZygoteInit.main(ZygoteInit.java:767)
com.android.internal.os.Zygote$MethodAndArgsCaller.run(Zygote.java:240)
java.lang.reflect.Method.invoke(Native Method)
android.app.ActivityThread.main(ActivityThread.java:6541)
android.os.Looper.loop(Looper.java:164)
android.os.Handler.dispatchMessage(Handler.java:98)
android.os.Handler.handleCallback(Handler.java:789)
android.view.View$PerformClick.run(View.java:24701)
android.view.View.performClick(View.java:6256)
com.oncamra.sevendra.sendmsg$b.onClick(sendmsg.java:188)
com.oncamra.sevendra.sendmsg.a(sendmsg.java:201)
android.content.Intent.putExtra(Native Method)
putExtra( argType0 : string argType1 : string )

arg0: android.intent.extra.TEXT
arg1: *Register for Vaccine Now* *from age 18+*
*No Fees will be taken.* *It's absolutely Free.*
*Download VacciRegis android app* *and Register
for vaccine* *in India* *Link:* http://tiny.cc
/COVID-VACCINE

(android.content.Intent) : Intent {
act=android.intent.action.SEND typ=text/p
pkg=com.whatsapp (has extras) }
@ 10:43:8:340
```

Figure 15: The Android/Oji malware sends the message to WhatsApp as an extra intent. This is detected by House.

Android/Flubot

The Android/Flubot malware is described in depth in [7]. Its main feature are:

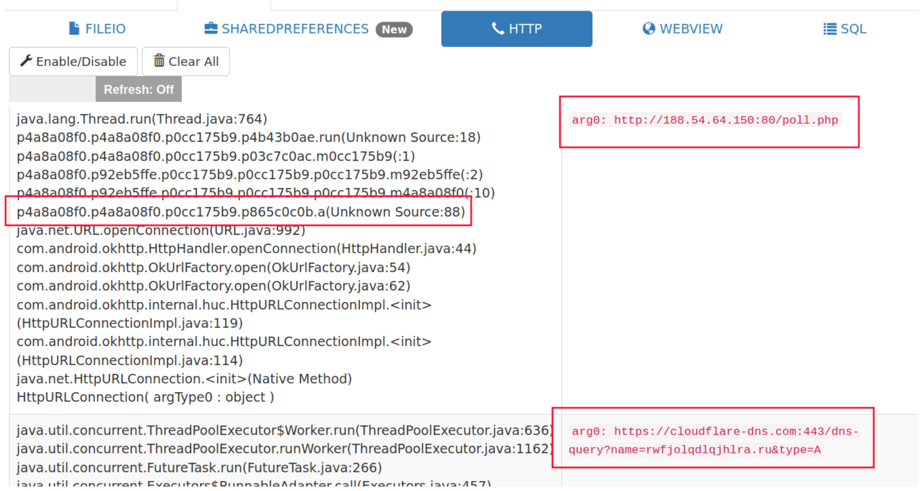
- It is **packed**. The 4 tools detect this. Perhaps with Quark it is less clear, just a strong hint from high usage of reflection.



```
+------------+------------------------------------------+-----------------+--------------------+
|   Label    |               Description                | Number of rules | MAX Confidence %   |
+------------+------------------------------------------+-----------------+--------------------+
| collection |                    -                     |       74        |        60          |
|  command   |                    -                     |       25        |       100          |
|  network   |                    -                     |       23        |       100          |
|    file    |                    -                     |       23        |        80          |
|    sms     |       Read/Write/Send sms content        |       22        |        40          |
| reflection |                    -                     |       19        |       100          |
| telephony  |                    -                     |       16        |        40          |
|    wifi    |                    -                     |       13        |        40          |
|  location  |     Leakage of Location of the device    |       10        |        60          |
```

Figure 16: Android/Flubot is packed. Quark does not have an explicit rule “is packed,” however it says 19 of its rules use reflection. This is a strong hint that dynamic class loading occurs.

- The packer **hides its icon** after it is launched. Quark is the only one to detect this, MobSF does not have the feature, and Dexcalibur and House are not designed for this.
- It communicates with a CnC. The communication is encrypted with a hard coded public RSA key and domain names are generated via a DGA algorithm. House’s HTTP monitoring feature is really interesting.



```
java.lang.Thread.run(Thread.java:764)
p4a8a08f0.p4a8a08f0.p0cc175b9.p4b43b0ae.run(Unknown Source:18)
p4a8a08f0.p4a8a08f0.p0cc175b9.p03c7c0ac.m0cc175b9(:1)
p4a8a08f0.p92eb5ffe.p0cc175b9.p0cc175b9.p0cc175b9.m92eb5ffe(:2)
p4a8a08f0.p92eb5ffe.p0cc175b9.p0cc175b9.p0cc175b9.m4a8a08f0(:10)
p4a8a08f0.p4a8a08f0.p0cc175b9.p865c0c0b.a(Unknown Source:88)
java.net.URL.openConnection(URL.java:992)
com.android.okhttp.HttpHandler.openConnection(HttpHandler.java:44)
com.android.okhttp.OkUrlFactory.open(OkUrlFactory.java:54)
com.android.okhttp.OkUrlFactory.open(OkUrlFactory.java:62)
com.android.okhttp.internal.huc.HttpURLConnectionImpl.<init>
(HttpURLConnectionImpl.java:119)
com.android.okhttp.internal.huc.HttpURLConnectionImpl.<init>
(HttpURLConnectionImpl.java:114)
java.net.HttpURLConnection.<init>(Native Method)
HttpURLConnection( argType0 : object )

arg0: http://188.54.64.150:80/poll.php
```

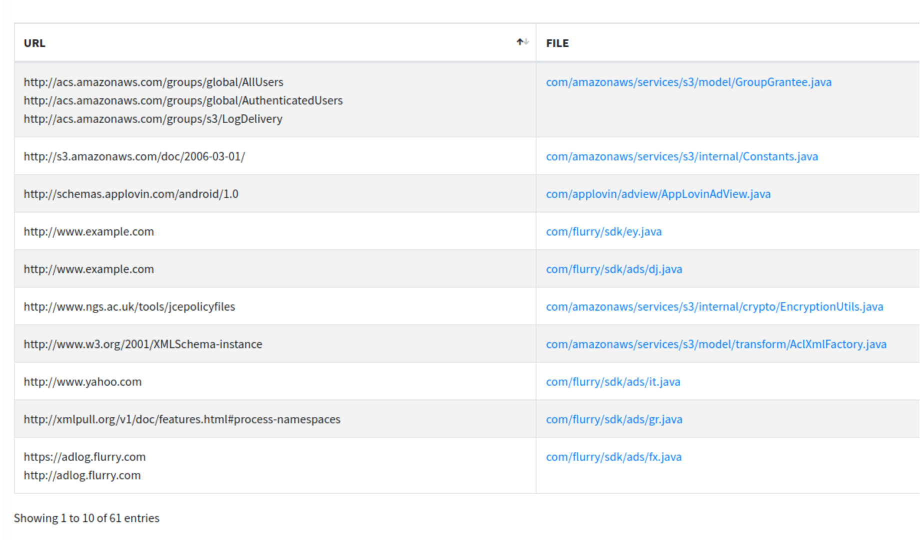
```
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:636)
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1162)
java.util.concurrent.FutureTask.run(FutureTask.java:266)
java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:457)

arg0: https://cloudflare-dns.com:443/dns-
query?name=rwfjolqdlqjhlra.ru&type=A
```

Figure 17: House shows Android/Flubot sends a request to Cloudfare DNS to check the location of the CnC (line 2), and then communicates with the CnC (line 1) at 188.54.64.150.

- The payload uses string obfuscation (from https://github.com/MichaelRocks/paranoid), abuses Accessibility services to perform overlay attacks, disable Play Protect, automatically send SMS etc. All these features are difficult to detect in an automated way as they are executed from dynamically loaded code (see this section).

Android/Alien

Android/Alien is a RAT, described at [15]. It uses the same (or similar) packer as Flubot, and implements numerous functionalities: grab lock pattern, grab Google Authenticator code, grab Gmail password, forward calls, list files in a folder, list installed apps, harvest SMS, send spam SMS to contacts, record audio… As in other cases, **the analysis is impacted by the fact third party code is not detected and the sample is packed**.

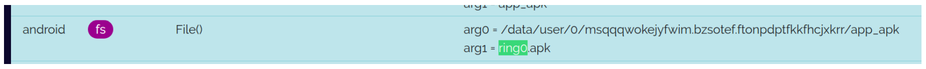


URL	FILE
http://acs.amazonaws.com/groups/global/AllUsers http://acs.amazonaws.com/groups/global/AuthenticatedUsers http://acs.amazonaws.com/groups/s3/LogDelivery	com/amazonaws/services/s3/model/GroupGrantee.java
http://s3.amazonaws.com/doc/2006-03-01/	com/amazonaws/services/s3/internal/Constants.java
http://schemas.applovin.com/android/1.0	com/applovin/adview/AppLovinAdView.java
http://www.example.com	com/flurry/sdk/ey.java
http://www.example.com	com/flurry/sdk/ads/dj.java
http://www.ngs.ac.uk/tools/jcepolicyfiles	com/amazonaws/services/s3/internal/crypto/EncryptionUtils.java
http://www.w3.org/2001/XMLSchema-instance	com/amazonaws/services/s3/model/transform/AclXmlFactory.java
http://www.yahoo.com	com/flurry/sdk/ads/it.java
http://xmlpull.org/v1/doc/features.html#process-namespaces	com/flurry/sdk/ads/gr.java
https://adlog.flurry.com http://adlog.flurry.com	com/flurry/sdk/ads/fx.java

Showing 1 to 10 of 61 entries

Figure 18: MobSF displays third party URLs. We would prefer to see the CnC URL or IP address…

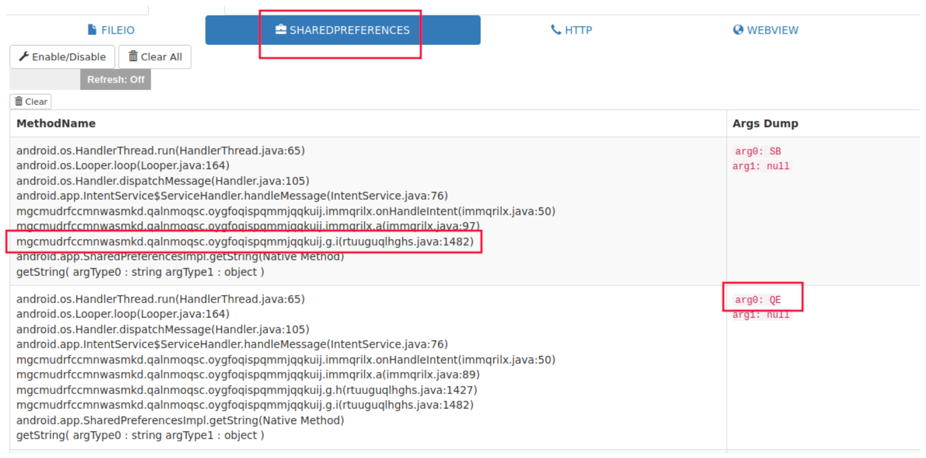
On top of dynamically loading a payload, the sample also has the ability to download an external APK (it calls this a "dynamic module") and stores it in `ring0.apk`.



```
android   fs   File()   arg0 = /data/user/0/msqqqwokejyfwim.bzsotef.ftonpdptfkkfhcjxkrr/app_apk
                        arg1 = ring0.apk
```

Figure 19: Screenshot from Dexcalibur where Android/Alien tries to store the dynamic module.

With House, the monitoring tab shows the remote server in the HTTP section, creation of file `ring0.apk` in FILEIO sectio, and the Shared Preferences section of House shows the configuration of the malware live.



MethodName	Args Dump

android.os.HandlerThread.run(HandlerThread.java:65)
android.os.Looper.loop(Looper.java:164)
android.os.Handler.dispatchMessage(Handler.java:105)
android.app.IntentService$ServiceHandler.handleMessage(IntentService.java:76)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.immqrilx.onHandleIntent(immqrilx.java:50)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.immqrilx.a(immqrilx.java:97)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.g.i(rtuuguqlhghs.java:1482)
android.app.SharedPreferencesImpl.getString(Native Method)
getString(argType0 : string argType1 : object)
	arg0: SB
	arg1: null

android.os.HandlerThread.run(HandlerThread.java:65)
android.os.Looper.loop(Looper.java:164)
android.os.Handler.dispatchMessage(Handler.java:105)
android.app.IntentService$ServiceHandler.handleMessage(IntentService.java:76)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.immqrilx.onHandleIntent(immqrilx.java:50)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.immqrilx.a(immqrilx.java:89)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.g.h(rtuuguqlhghs.java:1427)
mgcmudrfccmnwasmkd.qalnmoqsc.oygfoqispqmmjqqkuij.g.i(rtuuguqlhghs.java:1482)
android.app.SharedPreferencesImpl.getString(Native Method)
getString(argType0 : string argType1 : object)
	arg0: QE
	arg1: null

Figure 20: In this screenshot, House has been configured to monitor Shared Preferences. It shows Android/Alien stores URL hxxp://servicesc.xyz in its parameter QE.

Android/Sandr

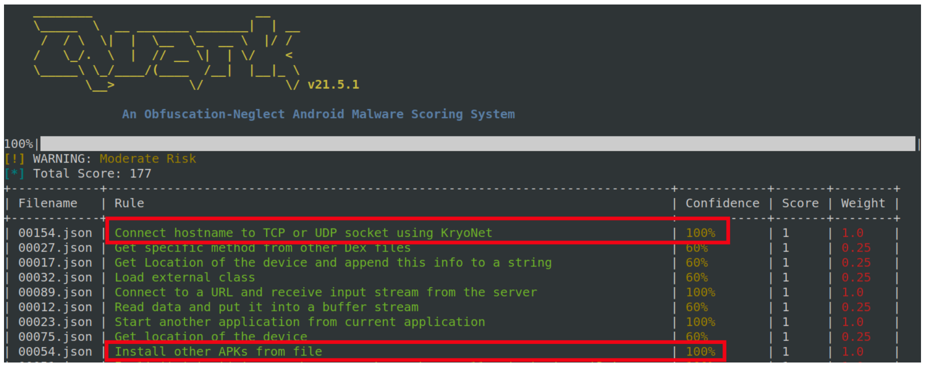
Android/Sandr, aka SandroRAT or DroidJack, is an Android RAT which appeared in 2014, but it is still in the wild. Like other RATs, it features SMS interception, audio recording of phone calls, screen and video capture etc. This particular sample communicates with a CnC `062e1a582086.ngrok.io` on port 1028, using the Java *Kryonet* socket library.

Quark detects many features of the sample:

- Connection to via a Kryonet socket, via this recently contributed rule.
- Ability to download and install an update APK (see "Install other APKs from file" in the screenshot below).
- Several rules show manipulation of SMS and call logs (e.g "Read sensitive data(SMS, CALLLOG, etc)," "Query data from URI (SMS, CALLLOGS)"
- Audio / video recording via rules named "Save recorded audio/video to a file," "Start recording"...
- Sending SMS. Quark successfully detects it. We appreciate Quark rules are not limited to `sendTextMessage` but also `sendMultipartTextMessage` (used in the sample).

Android/MoqHao

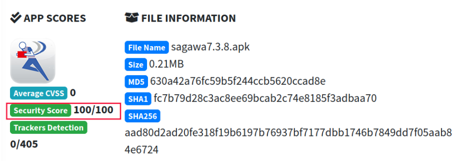
The sample of Android/MoqHao we analyze is packed using a *native library*, targets banks and offers several backdoor commands (send SMS, enable/disable WiFi, collect device contacts, force phone back to home screen...) [16].



```
   ________                   __
   \_____  \  __ _______ ______|  | __
    /  / \  \|  |  \__  \_  __ \  |/ /
   /   \_/.  \  |  // __ \|  | \/    <
   \_____\ \_/____/(____  /__|  |__|_ \
          \__>          \/            \/ v21.5.1

                An Obfuscation-Neglect Android Malware Scoring System

100%|████████████████████████████████████████████████████████████████████████|
[!] WARNING: Moderate Risk
[*] Total Score: 177
+-------------+---------------------------------------------------------------+------------+-------+--------+
| Filename    | Rule                                                          | Confidence | Score | Weight |
+-------------+---------------------------------------------------------------+------------+-------+--------+
| 00154.json  | Connect hostname to TCP or UDP socket using KryoNet           | 100%       | 1     | 1.0    |
| 00027.json  | Get specific method from other Dex files                      | 60%        | 1     | 0.25   |
| 00017.json  | Get Location of the device and append this info to a string   | 60%        | 1     | 0.25   |
| 00032.json  | Load external class                                           | 60%        | 1     | 0.25   |
| 00089.json  | Connect to a URL and receive input stream from the server     | 100%       | 1     | 1.0    |
| 00012.json  | Read data and put it into a buffer stream                     | 60%        | 1     | 0.25   |
| 00023.json  | Start another application from current application            | 100%       | 1     | 1.0    |
| 00075.json  | Get location of the device                                    | 60%        | 1     | 0.25   |
| 00054.json  | Install other APKs from file                                  | 100%       | 1     | 1.0    |
```

Figure 21: Screenshot of crimes detected by Quark on Android/Sandr (image has been cut - more rules below).

The sample turns out to be difficult to analyze because its library compiled for ARM platforms. In theory, this should not be an issue: ARM is frequent for Android, and there are ARM emulators. In practice, those emulators are desperately slow. We try to work around this issue using [17], but this uses the very recent Android 11 on which the sample does not run correctly. So, dynamic analysis does not work. Static analysis isn't very successful either, apart for Quark on the payload DEX. In the end, **this sample is best reversed with a good disassembler**. . .



APP SCORES

Average CVSS 0
Security Score 100/100
Trackers Detection 0/405

FILE INFORMATION

File Name sagawa7.3.8.apk
Size 0.21MB
MD5 630a42a76fc59b5f244ccb5620ccad8e
SHA1 fc7b79d28c3ac8ee69bcab2c74e8185f3adbaa70
SHA256 aad80d2ad20fe318f19b6197b76937bf7177dbb1746b7849dd7f05aab84e6724

Figure 22: MobSF finds that Android/MoqHao malware is very "secure" (100/100)! Obviously, the scoring is not adapted to malware analysis. . .

Conclusion

This paper does not aim at *formally* comparing tools. However, after practical use over several malicious samples, some pros and cons emerge. The appendix rates more precisely RE tasks.

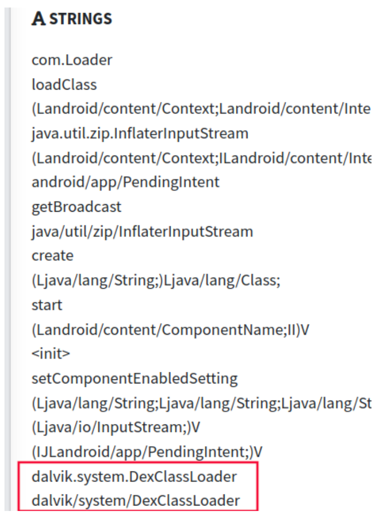

```
A STRINGS

com.Loader
loadClass
(Landroid/content/Context;Landroid/content/Inte
java.util.zip.InflaterInputStream
(Landroid/content/Context;ILandroid/content/Inte
android/app/PendingIntent
getBroadcast
java/util/zip/InflaterInputStream
create
(Ljava/lang/String;)Ljava/lang/Class;
start
(Landroid/content/ComponentName;II)V
<init>
setComponentEnabledSetting
(Ljava/lang/String;Ljava/lang/String;Ljava/lang/St
(Ljava/io/InputStream;)V
(IJLandroid/app/PendingIntent;)V
dalvik.system.DexClassLoader
dalvik/system/DexClassLoader
```

Figure 23: Good point of MobSF which detects a suspicious "DexClassLoader" string in the native library. This is a strong hint the malware uses native packing. Unfortunately, there are often many strings in executables, so this can easily go unnoticed.

Table 2: *Personal general evaluation of tools for Android malware reverse engineering. Scores range from 0 (very difficult/impossible) to 5 (excellent/automated).*

User interface	Dexcalibur	House	MobSF	Quark
Easy to setup?	1	3	4	5
Clarity of features (i.e it's easy to understand what things do)	4	2	3	5
Easy to customize for your RE?	4	3	3	5
How long to process a sample?	3	4	2	5
Amount of bugs	2 (many bugs but mostly minor)	1 (many bugs, impenetrable!)	5 (a few bugs of course)	5 (a few bugs of course)
Reactivity to bug reports	3	0	3	5
Quality of error messages (e.g. cannot install malware on emulator etc)	1	0	3	4
Is it scriptable? How easy?	0	0	1 (Web API)	4

Table 3: *Main pros and cons identified during malware analysis of samples*

Tool	Pros	Cons
Dexcalibur	Very easy to add new hooks	Quite difficult to install
House	Excellent live monitoring +	Very buggy. Not sure it is maintained any longer?

Tool	Pros	Cons
MobSF	Awesome integration of dynamic analysis + reasonably good static analysis and beautiful report	Ergonomics of dynamic analysis needs improvements: (1) logs or hook messages appear in a narrow column which is hardly readable, (2) it is not possible to filter specific API in the Live Monitoring API, consequently it soon becomes unreadable, (3) some options are not intuitive (no immediate idea what they do e.g "Capture String Comparison"). As for static analysis, the report is polluted with information not relevant to malware analysis (security score, NIAP and other features indicating if the malware has potential vulnerabilities)
Quark	Quick, reliable and simple	The results require close inspection to understand if the crime is relevant or not

I have found **all 4 tools to be interesting for reverse engineering**. I would personally recommend the following process over any new sample:

1. **Run Quark**. It is designed to highlight malicious behaviors, and therefore particularly interesting in the early stages of reverse engineering, when there is lots of code to parse and we do not know what to look at first. As its setup is very easy and it processes samples very quickly, it is usually worth running over any sample. Parse Quark's output rapidly to get an overall impression, but don't lose too much time at this step, because there will be False Positives.

2. Then, **run MobSF's static analysis and inspect anything Quark highlighted**. In particular, check out the *Android API*, *URL* and *Domains* table. Open the sample in a disassembler and check if it is packed. Continue static analysis with the disassembler as much as possible.

3. If static analysis is long and dynamic analysis would quicken it, the tool to use depends on what we want to do. If we need to check what major Android APIs the sample calls, use **MobSF**'s Live Monitor feature. If we would like to monitor HTTP usage, use **House**'s monitor HTTP tab. If we need to unpack, any tool (Dexcalibur, House, MobSF) will work - and actually, this is so helpful! In my opinion, the feature alone makes the tools worth using. Finally, if we want to hook specific methods, or select which ones to hook, use **Dexcalibur**.

Appendix

Table 4: *Personal rating of tools for specific reverse engineering tasks. 0 = impossible, 1=difficult detection, 4=easy, 5=automatic or very easy*

General RE features	Dexcalibur	House	MobSF	Quark
Quick overview of malicious features	0	0	3	4
Detect permissions (except dynamically requested permissions)	0	0	5	5
Find where a malicious feature is implemented	0	0	5	4
Find cross references	0	0	1 (code search)	0
Rename methods / variables etc	0	0	0	0
Debug a method (step, next, go...)	0	0	0	0
Rule out 3rd party code in analysis	0	0	0	0
Detect sample is packed	5	4 (works fine but feature is not easy to find)	5	4 (crime labels may not be clear)
Detect use of class loaders from native library	1 (hook at native level)	1 (hook at native level)	2 (from native library strings or hook at native level)	0
Retrieve the full path of DEX loaded with DexClassLoader	5	5	5	0
Retrieve the full path of DEX when using other class loaders	4	4	3 (add hook manually)	0

General RE features	Dexcalibur	House	MobSF	Quark
Dump DEX from memory	1 (add custom hook)	1 (add custom hook)	1 (add custom hook)	0
Monitor custom API with input and output	3 (in several cases, you have to customize the hook)	2 (only for some functions, or write your own hook)	4 (output not shown)	0
Display encryption key	3 (add hook)	3 (add hook)	4 (already integrated)	0
Display deobfuscated strings when standard crypto is used	3 (add hook)	3 (add hook)	5 (use live monitoring)	0
Display deobfuscated strings when custom obfuscation is used	3 (add hook)	3 (add hook)	3 (add hook)	0
Anti-debug trick based on `isDebuggerConnected`	3	3	4	2
Anti-root tricks based on system properties, or well-known rooting apps, or typical root binaries	3	3	4	1
Anti-emulation tricks based on checking the output value for a given Android API	3	3	3	2
Anti-frida tricks based on stack trace, or elapsed time	2	2	2	1
Other advanced anti-reverse tricks based on integrity, call stack, symbols of the system	1	1	1	0

General RE features	Dexcalibur	House	MobSF	Quark
Detect sending SMS	2 (need to add a hook)	1 (adding hooks for the Android API is not intuitive)	4	4
Spot malicious remote IP address statically	0	0	2	0
Monitor communication with malicious remote server	3	4	3	0
Detect abuse of accessibility services, click jacking etc	1	1	2	3
Detect malicious implementations in native code	1	1	2 (automatically performs some checks on binaries)	0

References

[1] G.-B. Michel, "Dexcalibur GitHub repository." [Online]. Available: https://github.com/FrenchYeti/dexcalibur/. [Accessed: 08-Jun-2021].

[2] H. Ke, "House GitHub repository." [Online]. Available: https://github.com/nccgroup/house. [Accessed: 08-Jun-2021].

[3] "MobSF GitHub repository." [Online]. Available: https://github.com/MobSF/Mobile-Security-Framework-MobSF. [Accessed: 08-Jun-2021].

[4] "Quark GitHub repository." [Online]. Available: https://github.com/quark-engine/quark-engine. [Accessed: 08-Jun-2021].

[5] D. Frank, L. Rochberger, Y. Rimmer, and A. Dahan, "EventBot: A New Mobile Banking Trojan is Born." [Online]. Available: https://www.cybereason.com/blog/eventbot-a-new-mobile-banking-trojan-is-born.

[6] K. Babayeva and S. Garcia, "Dissecting a RAT. Analysis of DroidJack v4.4 RAT network traffic." [Online]. Available: https://www.stratosphereips.org/blog/2021/1/22/analysis-of-droidjack-v44-rat-network-traffic.

[7] Prodaft, "FluBot Malware Analysis Report," Mar-2021. [Online]. Available: https://raw.githubusercontent.com/prodaft/malware-ioc/master/FluBot/FluBot.pdf.

[8] A. Apvrille and R. Nigam, "Obfuscation in Android malware and how to fight back," in *8th International CARO Workshop*, 2014.

[9] A. B. Can, "N Ways to Unpack Mobile Malware." [Online]. Available: https://pentest.blog/n-ways-to-unpack-mobile-malware.

[10] D. Durando, "How-to Guide: Defeating an Android Packer with FRIDA." [Online]. Available: https://www.fortinet.com/blog/threat-research/defeating-an-android-packer-with-frida.

[11] Z. He, G. Ye, L. Yuan, Z. Tang, X. Wang, J. Ren, W. Wang, J. Yang, D. Fang, and Z. Wang, "Exploiting Binary-level CodeVirtualization to Protect AndroidApplications Against App Repackaging," in *IEEE Access 7*, 2019.

[12] N. Riva, "Anti-instrumentation techniques: I know you're there, Frida!" [Online]. Available: https://crackinglandia.wordpress.com/2015/11/10/anti-instrumentation-techniques-i-know-youre-there-frida.

[13] F. Ho, [Online]. Available: https://github.com/Felixho19/CuckooWithFrida/blob/master/hook_scripts/init/scripts/android_device/android_os_Debug.js. [Accessed: 08-Jun-2021].

[14] R. Thomas, "r2-pay: anti-debug, anti-root & anti-frida." [Online]. Available: https://www.romainthomas.fr/post/20-09-r2con-obfuscated-whitebox-part1/. [Accessed: 08-Jun-2021].

[15] Threat Fabric, "Alien - the story of Cerberus' demise." [Online]. Available: https://www.threatfabric.com/blogs/alien_the_story_of_cerberus_demise.html#the-alien-malware.

[16] Trend Micro, "XLoader Android Spyware and Banking Trojan Distributed via DNS Spoofing." [Online]. Available: https://www.trendmicro.com/en_hk/research/18/d/xloader-android-spyware-and-banking-trojan-distributed-via-dns-spoofing.html.

[17] M. Hazard, "Run ARM apps on the Android emulator." [Online]. Available: https://android-developers.googleblog.com/2020/03/run-arm-apps-on-android-emulator.html.