VB2021 localhost

7 - 8 October, 2021 / vblocalhost.com

REVERSE ANDROID MALWARE LIKE A JEDI MASTER

Axelle Apvrille

Fortinet, France

aapvrille@fortinet.com

ABSTRACT

In this paper, we use four new *Android* reverse engineering tools – *Dexcalibur*, *House*, *MobSF* and *Quark* – over malicious samples of 2020/2021. We explain how best to use or customize the tools, 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 plug-ins or scripts (for Radare, JEB) or implemented hooks using *Frida*. Those tools are excellent, useful to padawan and masters alike. But there are new tools: *Dexcalibur* (2019), *House* (2018), *Quark* (2019) and *MobSF* (2015). In this paper we focus on these four tools and their use in *Android* reverse engineering (RE).

- *Dexcalibur* [1] and *House* [2] can both be seen as web front-ends to *Frida* [3]. 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* 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'* [4]. It features both s*tatic 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 [5]. *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 rules 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 Table 1. They have been selected because (1) they are recent and (2) they exhibit particular features (packing, native library...).

SHA256 sum	Name	Date
`1a8c17ad1a790554278b055bdb946d4597ba9af6be3611ee6311b90c7f7848c5`	Android/EventBot [6]	April 2020
`f82d6f24af2a4444c696c64060582d8aed6280da578c4dea3bb71bd6a11ebcf8`	Android/Sandr [7]	June 2020
`f699f9e50e8401943321d757a9c1bab367473f102c0abfb57367e9252aae7fde`	Riskware/ Tenpack!Android	Feb 2021
`fd5f7648d03eec06c447c1c562486df10520b93ad7c9b82fb02bd24b6e1ec98a` or `9379f91ddd9326bc1a8cf2fe4a22951d0859b2b7f88ffe68b023a97d59130810`	Android/Flubot [8]	March 2021
`a25363b68faa8188b99622d8909921a4026ea7241df6377d0a6374d2b2b4e08c`	Android/Oji [9]	May 2021
`aad80d2ad20fe318f19b6197b76937bf7177dbb1746b7849dd7f05aab84e6724`	Android/MoqHao [10]	May 2021
`8810ca80d21173528be71109cd9e5a73afce98a080892643ffdcbe53ac9b6893`	Android/Ksapp [11]	May 2021
`dc215663af92d41f40f36088ec1b850b81092ea94a4a061a9ce88178daee965a`, `0da75ac97f4ec8954a961c270bcbe75bd2671c65cf25db45540b70f1ff403e31`	Android/Alien [12]	Sept 2020, May 2021

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

USING THE FOUR TOOLS FOR COMMON RE TASKS

Unpacking malware

Malware analysts commonly encounter packed *Android* malware [13] (e.g. ApkProtect, Bangcle, etc.). The malicious payload is hidden in the APK, 'unpacked' by a *packer* whose sole goal is to make reverse engineering more difficult and to conceal the payload. The most common implementation consists of using `DexClassLoader` to dynamically load a hidden DEX file. The four tools typically detect use of `DexClassLoader`. *Quark* detects it with a simple rule (example: [14]). The other three rely on *Frida* hooks.

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

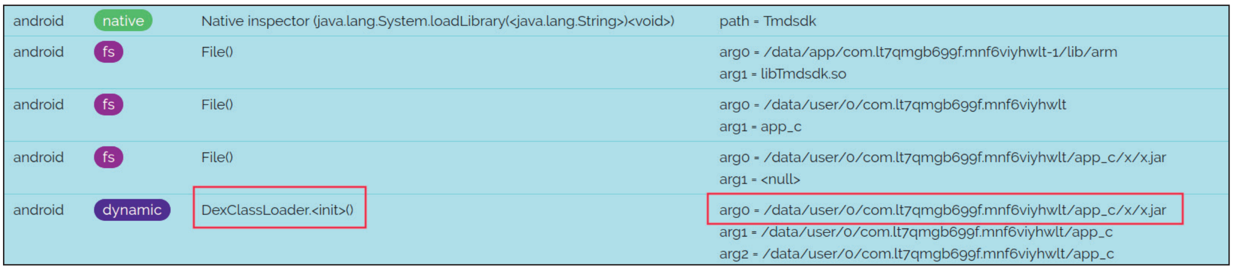


Figure 1: Dexcalibur detects live use of DexClassLoader in a sample of Riskware/Tenpack.

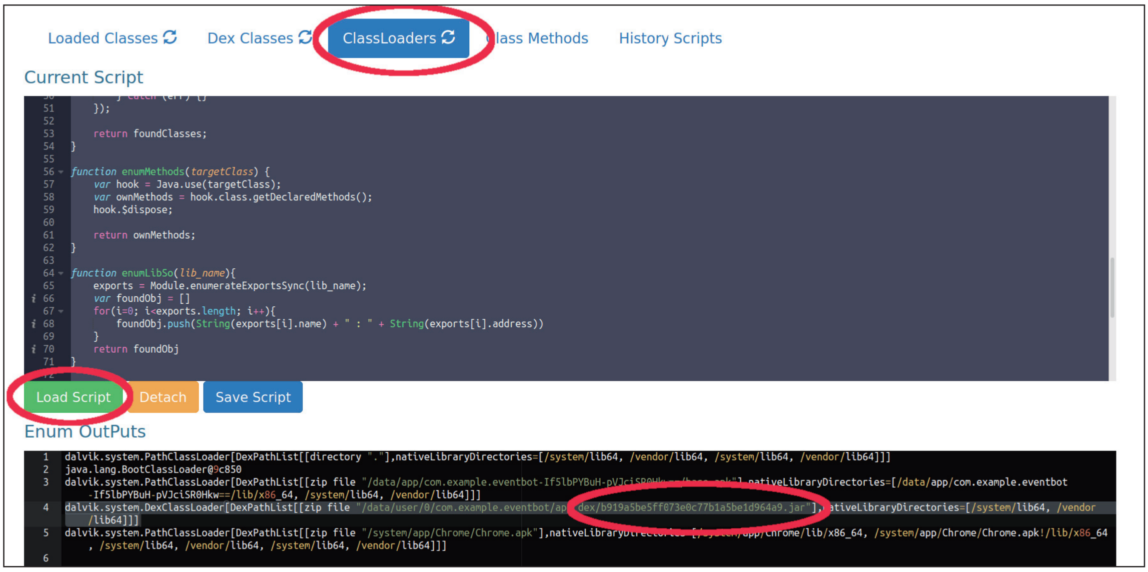


Figure 2: With House, go to the 'Enumeration'/'ClassLoaders' tab. This displays a customizable Frida script. Actually, it does not need any modification: press the 'Load Script' button. The 'Enum Outputs' show that Android/EventBot dynamically loads a JAR.

With *Dexcalibur*, we have to make sure `DexClassLoader` is probed (by default, it is). If it isn't, click on the 'Probe ON' button. If, for some reason, the hook is not present at all, you can 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 or 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 [15].

Besides hooking class loaders, there are a few other strategies for unpacking dynamically [16]: hooking file creation or deletion (at Java level or system level) [17], dumping the memory (e.g. [18]).

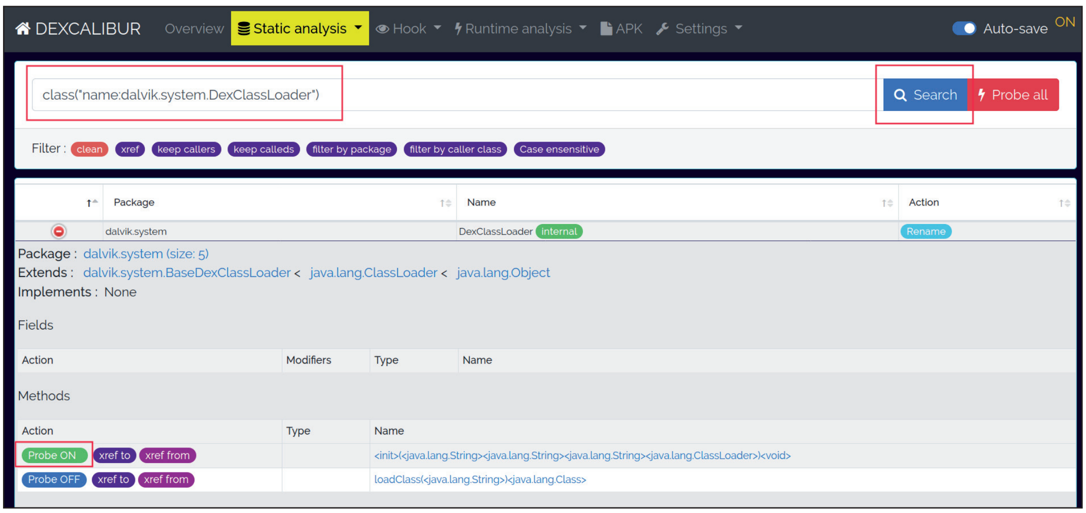


Figure 3: Search for DexClassLoader in the 'Static Analysis' tab and select 'Probe ON' to hook its constructor.

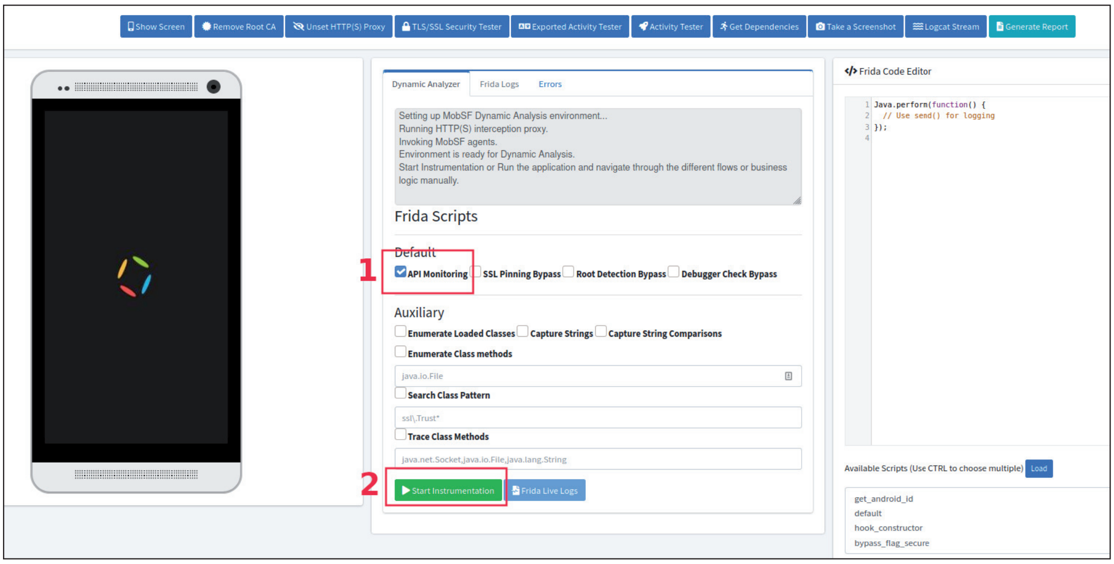


Figure 4: In MobSF, click 'API Monitoring', then 'Start Instrumentation'. A new button, 'Live API Monitor', appears at the top; click on it, and see next image.

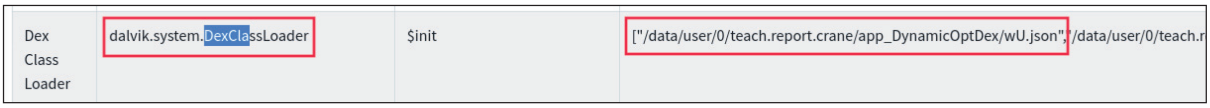


Figure 5: Live API monitoring in MobSF detects that Android/Alien loads a DEX.

Some clever pieces of malware load DEX dynamically from *native libraries*. This is the case of Android/MoqHao. It is difficult to analyse with the four tools (as we will see later). Finally, note that VM-based packers and packers for native code are on their way [19], but haven't been used in malware yet.

Analysing dynamically loaded DEX

All four tools have difficulties accessing/hooking inside a dynamically loaded DEX.

This is an issue for RE because it is this dynamically loaded DEX that typically holds the interesting malicious features (payload), whereas the wrapping DEX (packer) is of no importance (apart from making reversing harder).

There are two solutions:

1. **Manual solution**. Retrieve the payload DEX (see previous paragraph), and analyse it with a disassembler. Note that only *Quark* is able to process the payload DEX, the other three tools do not support DEX.
2. **Automatic solution**. Write a *Frida* hook that hooks *inside* the dynamically loaded DEX. Writing such a hook requires experience ([20]). Then, load the hook in a dynamic analysis tool. In theory, it should work, but I have only managed to get it to work with *House* (not *Dexcalibur* or *MobSF*), and even with *House* the process is not reliable (sometimes it works, sometimes it doesn't!).

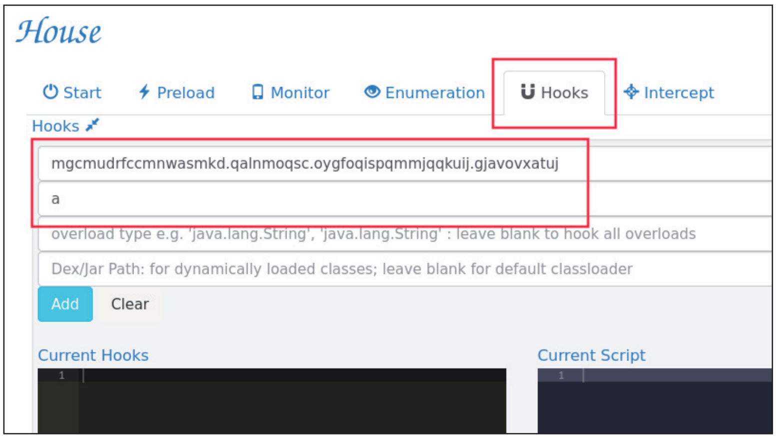


Figure 6: Configuring House to hook inside dynamic code. Fill class name and method. Despite the name do not fill the entry DEX/Jar path.

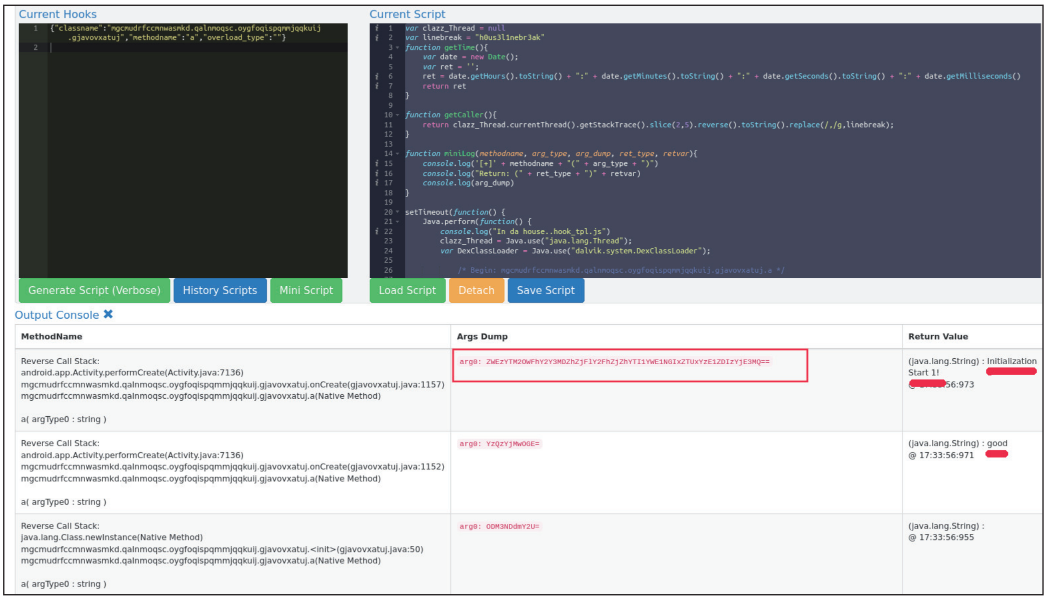


Figure 7: This screenshot of House shows live string de-obfuscation inside the dynamically loaded DEX of Android/Alien.

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 of 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 hook. Click 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:"Nonebefore", after:true
});
```

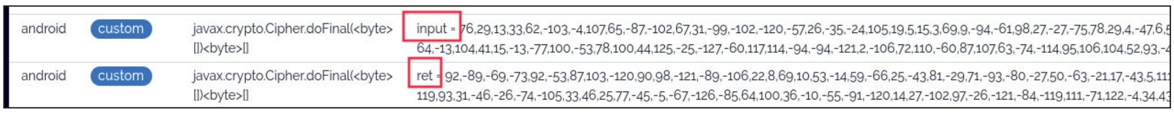


Figure 8: In Dexcalibur, the hooked encryption methods show the cipher text (input) and plain text (ret) for Android/Ksapp.

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 the domains and IP addresses used by the malware. In practice, there are often false positives, 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 attempts to detect emulators, debuggers, rooted environments, or even *Frida* hooks. While this has no effect on static analysis, it makes dynamic analysis harder. Typically, protections are based on the use of specific APIs (e.g. `isDebuggerConnected()`), the 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 [21], [22], [23] and [24].

For a malware analyst, the first step consists of detecting those protections. To do so, *MobSF* relies on APKiD [25], a tool that focuses on identifying packers, obfuscators, anti-VM and anti-debug tricks. With *Quark*, we can typically implement a rule to detect use of `isDebuggerConnected`. Unfortunately, *Quark* will be inefficient by design on most other tricks (*Quark* cannot detect specific strings, files, pipes or processes).

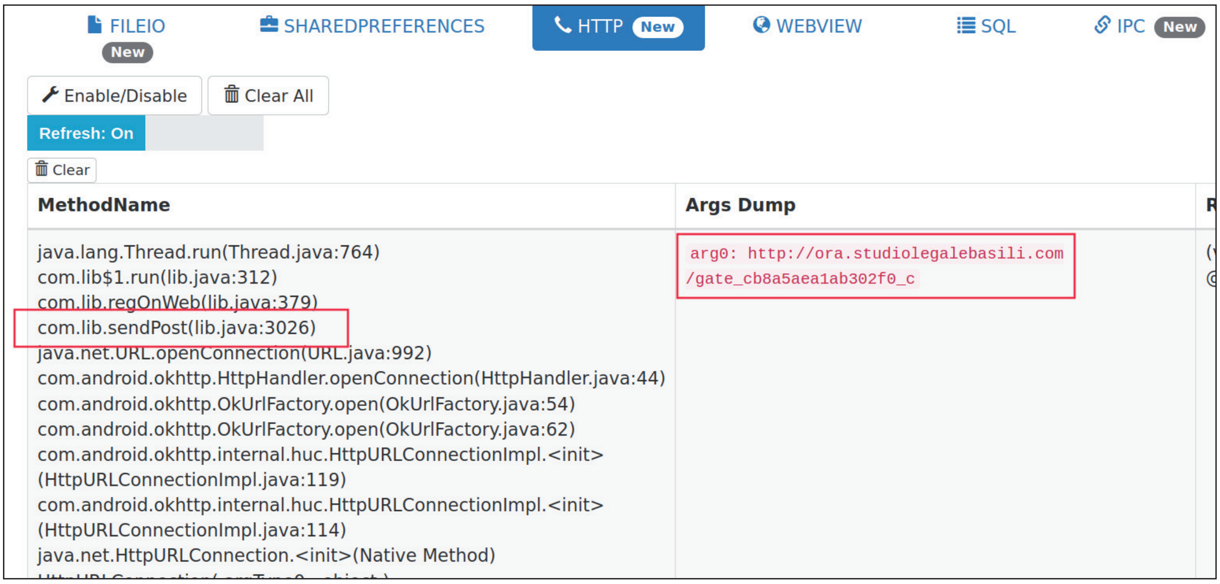


Figure 10: House has been configured to enable HTTP monitoring ('Monitor' tab, then 'Enable/Disable' button). Here it shows that Android/EventBot contacts hxxp://ora.studiolegalebasili.com/. Better than Wireshark, House shows that the call occurs from a method named com.lib.sendPost. We can hook that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

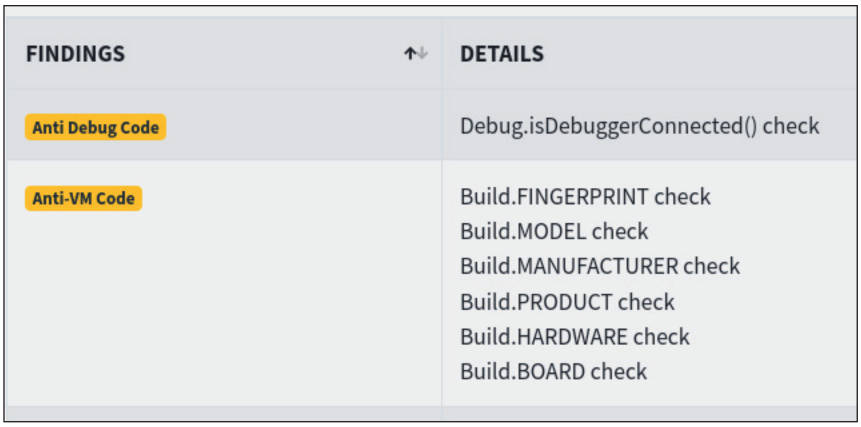


Figure11: MobSF uses APKiD to detect anti-debug and anti-VM tricks, here in Android/Ghimob.

Once the protection is identified, we must try to *bypass* it. [26] 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. [27]) are not supported yet by *Dexcalibur*, and their status is uncertain for *House* and *MobSF*. If such a hook is required, we have to run our own *Frida* server and client. Also, some anti-*Frida* techniques, such as the `libc.so` tampering check [28], are way more difficult to bypass. Fortunately, we haven't ever seen such advanced techniques in malware.

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

PRACTICAL CASES

Android/Oji.G!worm

The Android worm Oji 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*. The sample I analysed uses AES/CBC, but the code is wrong: the cipher text does not decrypt as it is not a multiple of 16.

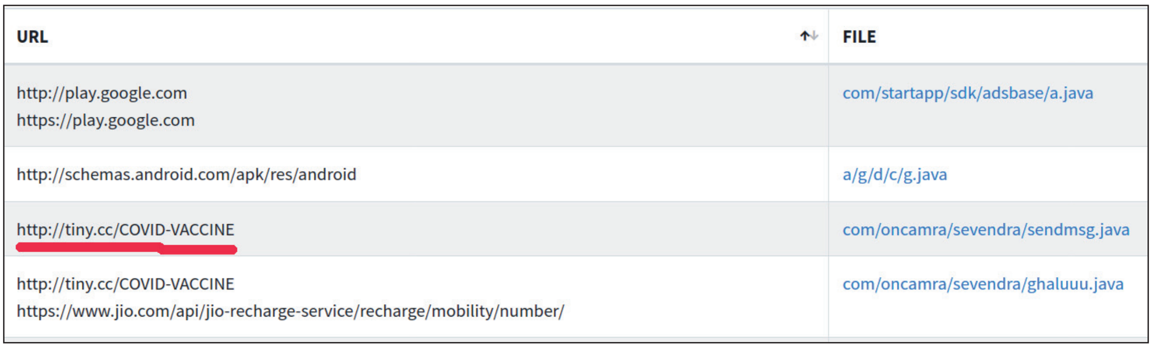
First, *all tools* are heavily impacted by the fact they are unable to tell the difference between code from third-party kits (for example, this sample uses `com.startapp`, an in-app ads SDK) and the malicious part (`com.omcamra.sevendra`). As far as I know, the only tool which takes this into account is my own tool [29].

Quark manages to detect that the malware reads contacts (under a slightly misleading crime name *'Read sensitive data(SMS, CALLLOG, etc.)* '). The other results are not very relevant and are 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).

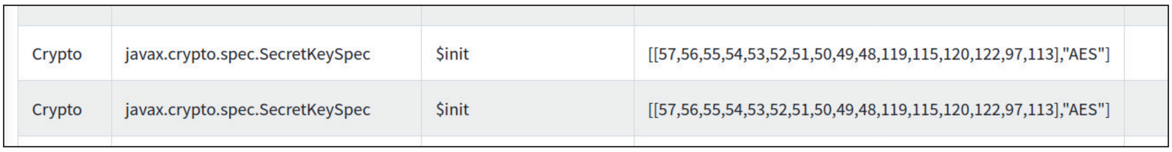
In this particular sample, the decryption fails (a bug of the code?), but this is difficult to spot with *MobSF*. It does show in *Android*'s *logcat* but there are many lines, and no particular highlight.

```
6-0411:07:58.2621908319083WSystem.err:    java.lang.Exception:   [decrypt]
error:1e00006a:Cipherfunctions: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 used by the sample: hxxp://tiny.cc/COVID-VACCINE.



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.

```
06-0411:07:58.262 19083 19083 WSystem.err:
atcom.oncamra.sevendra.ghaluuu.c(ghaluuu.java:240)
```

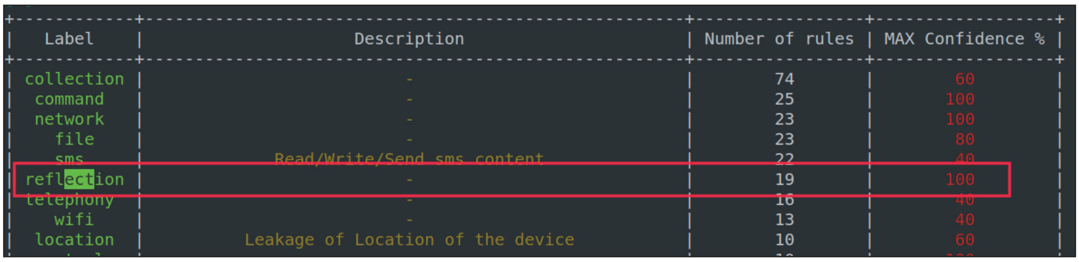
House is not the best choice for malware reconnaissance, but its monitoring section is helpful for the sample. For example, monitoring the *IPC section* clearly shows the message copied for *WhatsApp* – something the other tools won't show easily.

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

Android/Flubot

Android/Flubot is described in depth in [8]. Its main features are:

- It is packed. The four 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 tool 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.

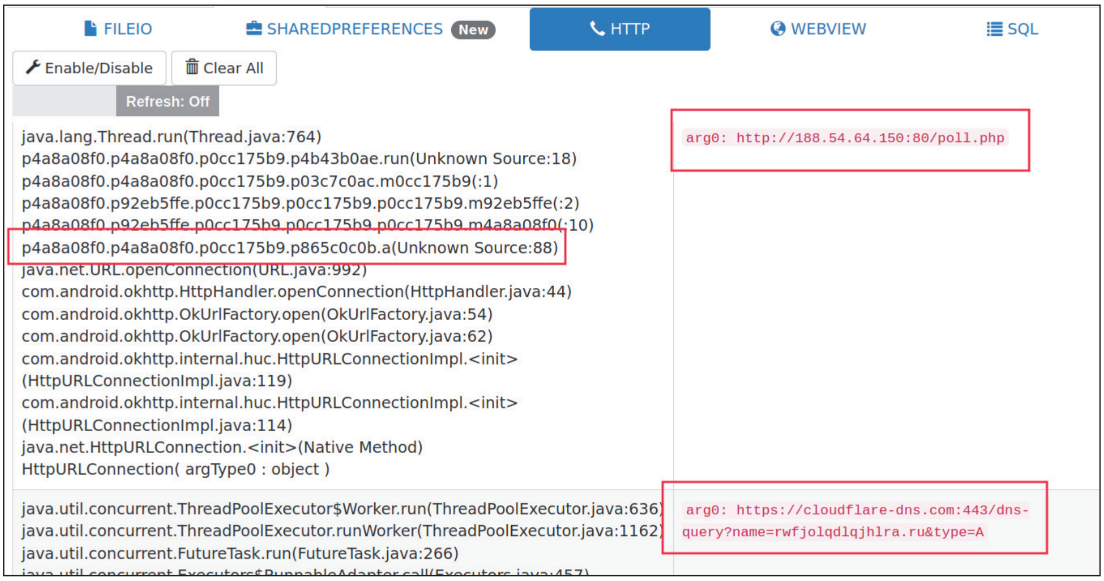


Figure 17: House shows that Android/Flubot sends a request to Cloudflare DNS to check the location of the CnC (line 2), and then communicates with the CnC (line1) at 188.54.64.150.

The payload uses string obfuscation (from [30]), and abuses Accessibility services to perform overlay attacks, disable *Play Protect*, automatically send SMSs, etc. All these features are difficult to detect in an automated way as they are executed from dynamically loaded code.

Android/Alien

Android/Alien is a RAT, described at [12]. 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 SMSs, send spam SMS to contacts, record audio, etc. 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`.

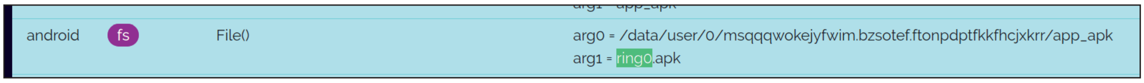


Figure 19: Screenshot from Dexcalibur where Android/Alien tries to store the dynamic module.

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

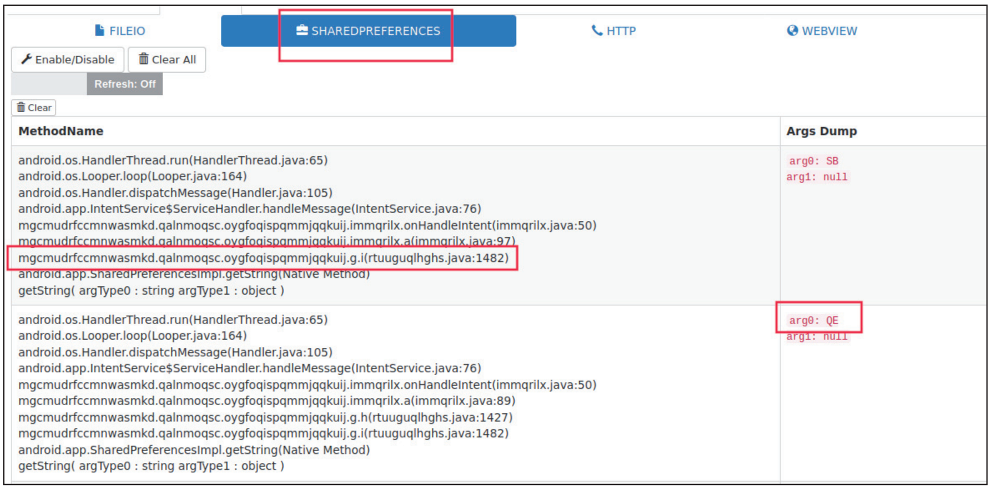


Figure 20: In this screenshot, House has been configured to monitor Shared Preferences. It shows that Android/Alien stores the URL hxxp://servicesc.xyz in its parameter QE.

Android/Sandr

Android/Sandr, a.k.a. 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 CnC via a Kryonet socket, via [31].
- 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 this. We appreciate that *Quark* rules are not limited to `sendTextMessage` but also `sendMultipartTextMessage` (used in the sample).

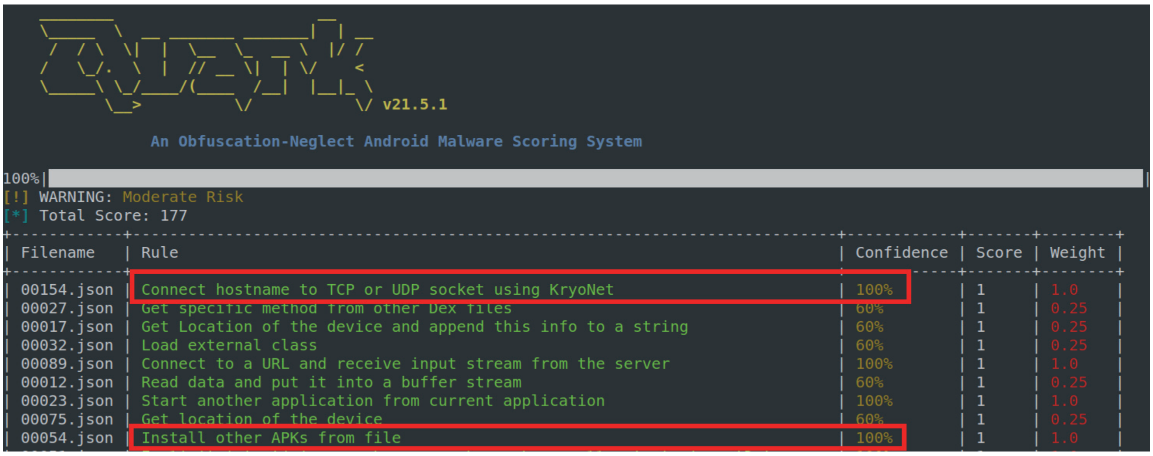


Figure 21: Screenshot of crimes detected by Quark on Android/Sandr (image has been cut – more rules below).

Android/MoqHao

The sample of Android/MoqHao we analyse is packed using a *native library*, targets banks and offers several backdoor commands (send SMS, enable/disable Wi-Fi, collect device contacts, force phone back to home screen, etc.) [32].

The sample turns out to be difficult to analyse because its library is compiled for ARM platforms. In theory, this should not be an issue: ARM is common for *Android*, and there are ARM emulators. In practice, those emulators are desperately slow. We try to work around this issue using [33], 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.

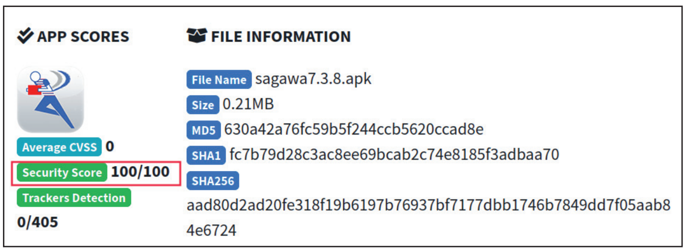


Figure 22: MobSF finds that Android/MoqHao malware is very 'secure' (100/100)! Obviously, the scoring is not adapted to malware analysis.

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.

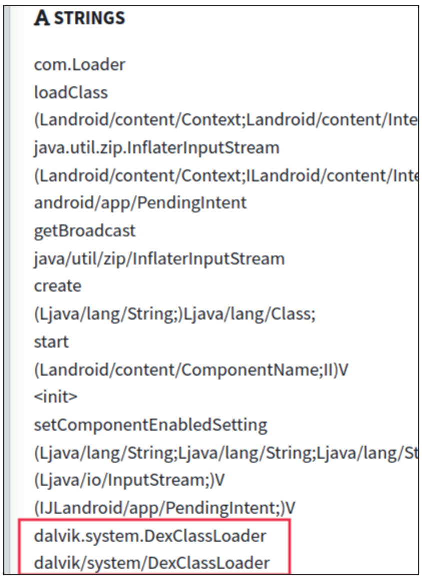


Figure 23: Good point of MobSF which detects a suspicious 'DexClassLoader' string in the native library. This is a strong hint that the malware uses native packing. Unfortunately, there are often many strings in executables, so this can easily go unnoticed.

User interface	**Dexcalibur**	**House**	**MobSF**	**Quark**
Easy to setup?	1	3	4	5
Clarity of features (i.e. ease of understanding what things do)	4	2	3	5
Easy to customize for your RE?	4	3	3	5
How long does it take to process a sample?	3	4	2	5
Number 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 easily?	0	0	1 (Web API)	4

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

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?
MobSF	Awesome integration of dynamic analysis + reasonably good static analysis and beautiful report	Ergonomics of dynamic analysis need improvements: (1) logs or hook messages appear in a narrow column which is hardly readable, (2) it is not possible to filter specific APIs 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

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

I have found *all four 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 behaviours, 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. 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, this feature alone makes the tools worth using. Finally, if we want to hook specific methods, or select which ones to hook, use *Dexcalibur*.

REFERENCES

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

[2] Ke, H. House GitHub repository. https://github.com/nccgroup/house. [Accessed 08 June 2021].

[3] Frida. https://frida.re/.

[4] MobSF GitHub repository. https://github.com/MobSF/Mobile-Security-Framework-MobSF. [Accessed 08 June 2021].

[5] Quark GitHub repository. https://github.com/quark-engine/quark-engine. [Accessed 08 June 2021].

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

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

[8] Prodaft. FluBot Malware Analysis Report, Mar-2021. https://raw.githubusercontent.com/prodaf t/malwareioc/master/FluBot/FluBot.pdf.

[9] Apvrille, A. Android/Oji worm fake COVID-19 vaccine registration campaign. https://cryptax.medium.com/android-oji-worm-fake-covid-19-vaccine-registration-campaign-80b6f9c79abe.

[10] Apvrille, A. A native packer for Android/MoqHao. https://cryptax.medium.com/a-native-packer-for-android-moqhao-6362a8412fe1.

[11] Apvrille, A. Blind try of MobSF over a suspicious Android sample. https://cryptax.medium.com/blind-try-of-mobsf-over-a-suspicious-android-sample-7fe7368a4804.

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

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

[14] quark-engine / quark-rules. https://github.com/quark-engine/quark-rules/blob/master/00149.json.

[15] Project: InMemoryDexClassLoader dump. https://codeshare.frida.re/@cryptax/inmemorydexclassloader-dump/.

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

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

[18] Novella, E. fridroid-unpacker. https://github.com/enovella/fridroid-unpacker. [Accessed 08 June 2021].

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

[20] Apvrille, A. Hooking methods inside dynamically loaded classes. https://github.com/cryptax/misc-code/blob/master/frida_hooks/dyndecrypt.tmpl.js.

[21] Alexander-Brown, S. Rootbeer GitHub repository. https://github.com/scottyab/rootbeer. [Accessed 08 June 2021].

[22] Android Anti-Reversing Defenses. https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering.

[23] Cong, F. Anti Frida tricks. https://github.com/feicong/strong-frida/blob/main/docs/README.md. [Accessed 08 June 2021].

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

[25] rednaga / APKiD. https://github.com/rednaga/APKiD.

[26] Ho, F. https://github.com/Felixho19/CuckooWithFrida/blob/master/hook_scripts/init/scripts/android_device/android_os_Debug.js. [Accessed 08 June 2021].

[27] Project: anti-frida-bypass. https://codeshare.frida.re/@enovella/anti-frida-bypass/.

[28] Thomas, R. r2-pay: anti-debug, anti-root & anti-frida. https://www.romainthomas.fr/post/20-09-r2con-obfuscatedwhitebox-part1/. [Accessed 08 June 2021].

[29] cryptax / droidlysis. https://github.com/cryptax/droidlysis.

[30] MichaelRocks / paranoid. https://github.com/MichaelRocks/paranoid.

[31] quark-engine / quark-rules. https://github.com/quark-engine/quark-rules/blob/master/00154.json.

[32] Trend Micro. XLoader Android Spyware and Banking Trojan Distributed via DNS Spoofing. https://www.trendmicro.com/en_hk/research/18/d/xloader-android-spyware-and-banking-trojandistributed-via-dns-spoofing.html.

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

APPENDIX

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 third-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
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

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
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
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

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