Locating the Trojan inside an infected COVID-19 contact tracing app

By @cryptax

Published: 2020-09-25 · Archived: 2026-04-06 01:14:39 UTC

5 min read

Sep 18, 2020

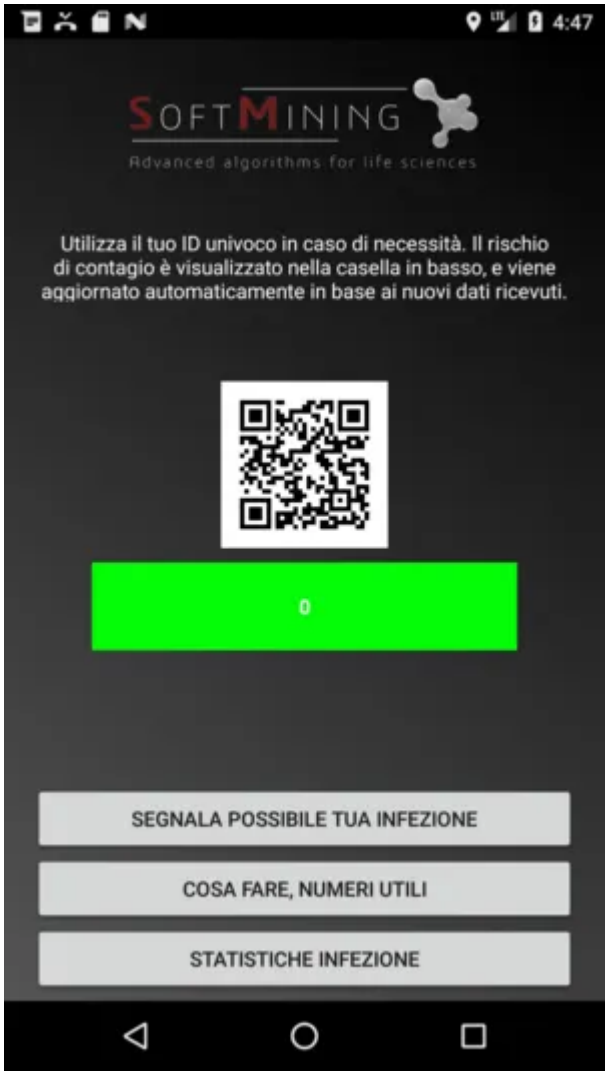
Update Sept 25, 2020: part 2 is available here.

An italian company, SoftMining, developed an Android COVID-19 contact tracing application "SM-COVID-19". Unfortunately, malware authors repackaged the application to include a Java-based **Meterpreter** backdoor from **Metasploit**.

The samples were discovered in March 2020, and you can find several blog posts on them (here, here). For instance, this one mentions the samples are "repackaged application injected with metasploit". Interesting! But **where is that "metasploit" in the samples? That's what we are going to discuss in this article.**

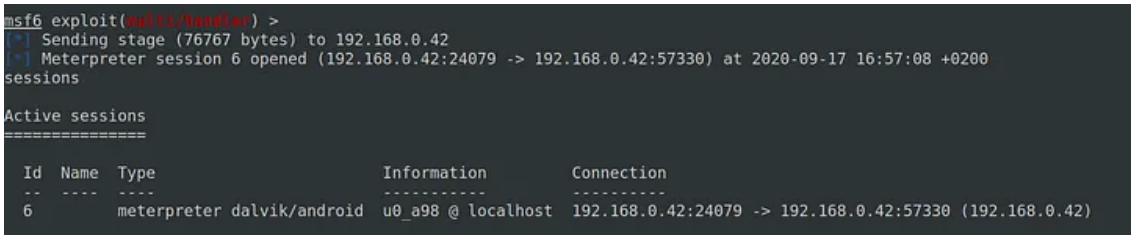
A remote shell for the attacker

When the victim launches the infected app on the smartphone, the legitimate COVID-19 application begins, but, additionally, in background, the malicious part connects to a remote server (samples were found connecting to IP addresses `87.19.73.8` and `95.239.79.156` — there may be others).



The malware includes the legitimate app. So, it is difficult for the victim to understand this is an infected version.

Press enter or click to view image in full size



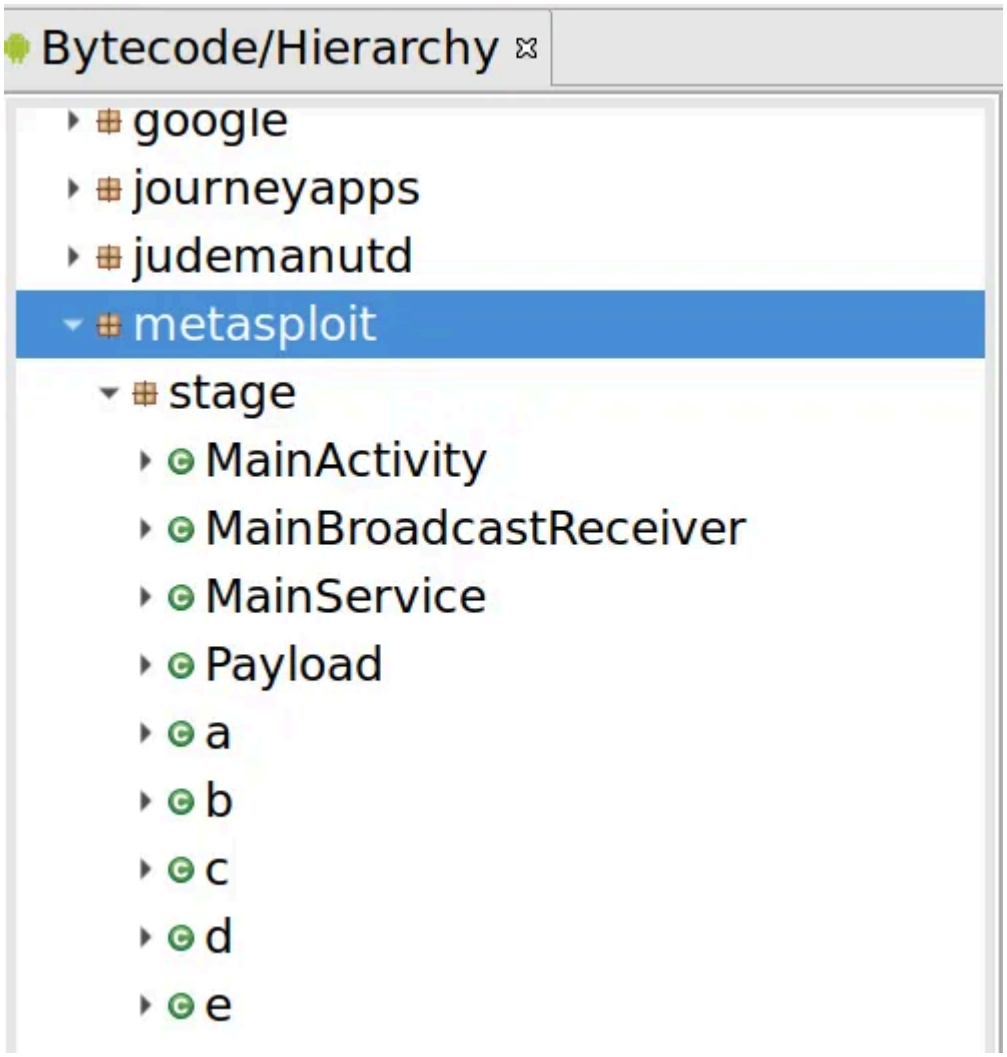
An infected version is connected to the attacker's server. For this study, we do not contact the real server (!) but our own local host, whose IP address is 192.168.0.42 (we'll explain how to do that later).

The attacker gets a shell with numerous commands: dump SMS, contacts, get a screenshot, get webcam view etc.

Seen from attacker's end. What s/he can have your smartphone do. Uses metasploit console (msfconsole)

Locating the trojan in the code (easy case)

Depending on the sample, the difficulty may vary. For example, in a non-obfuscated sample , the malicious part is obvious and located within the explicit hierarchy `com.metasploit.stage`.



In this sample (sha2`56: f3d452befb5e319251919f88a08669939233c9b9717fa168887bfebf43423aca)`, the injected meterpreter hasn't been obfuscated. Its code is located within com.metasploit.stage.

Locating the trojan in an obfuscated sample (intermediate)

But, of course, there are obfuscated samples 😉 where the name for the trojanized part isn't going to be so immediate. A close inspection of the Android manifest can help. We skip the permissions and jump to the "application" part.

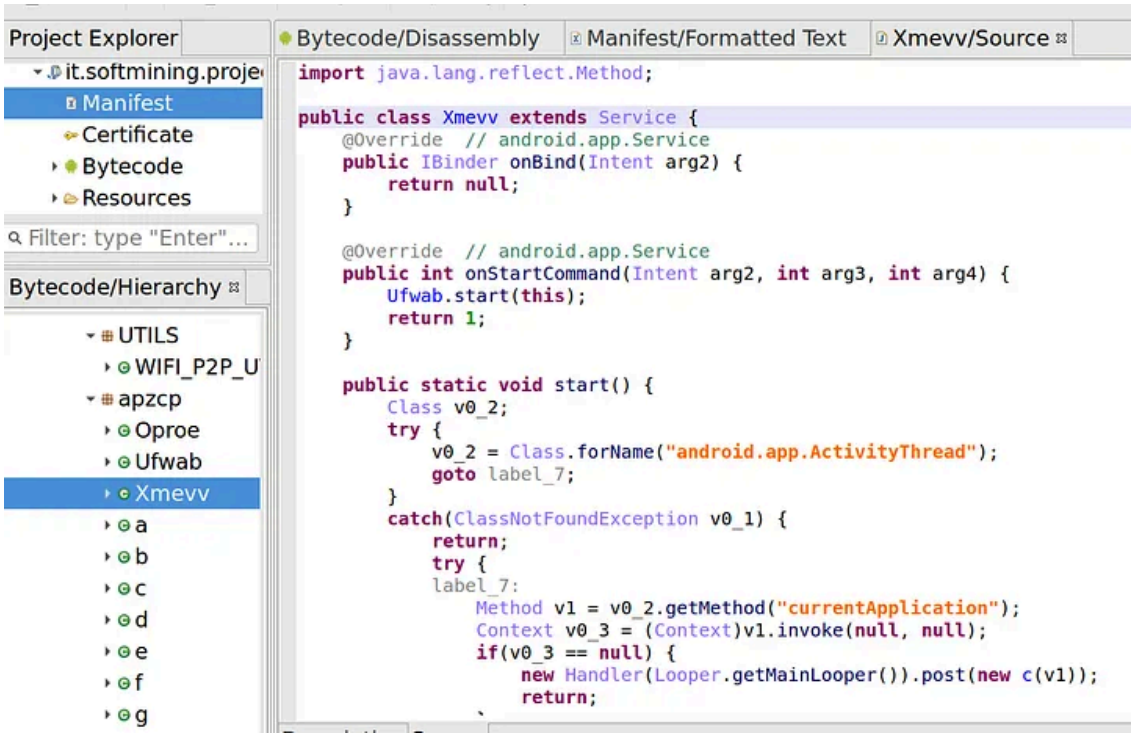
Press enter or click to view image in full size

Android manifest of sample 7b8794ce2ff64a669d84f6157109b92f5ad17ac47dd330132a8e5de54d5d1afc

First, notice the application uses multiple DEX (i.e the code is contained not only within `classes.dex`, but also `classes2.dex` and potentially more) with the label "androix.multidex.MultiDexApplication". We'll get back to that in a future post and it has some importance in that case.

Then, there is an activity named `it.softmining.projects.covid19.savelifestyle.SendRiskActivity`: this is an activity from the real application. Not suspicious. Later, the main activity `it.softmining.projects.covid19.savelifestyle.MainActivity`.Then, do you see it? There is a service whose name is `it.softmining.projects.covid19.savelifestyle.apzcp.Xmevv`. This is *not* part of the real application, and the fact the last part of the name `apzcp.Xmevv` is obfuscated when the beginning is not, *should immediately trigger an alarm* in your mind. We decompile the class and recognize the Meterpreter's MainService.

Press enter or click to view image in full size



Obfuscated name "Xmevv" for com.metasploit.stage.MainService. All other malicious classes are located in "apzcp" namespace.

Locating the trojan in difficult cases!

Now, what if the malicious activity or service names are not mentioned in the manifest (or you don't spot them)? Actually, I was unlucky, this is what happened with the *first* sample I examined 😓

Get @cryptax's stories in your inbox

sha256: `992f9eab66a2846d5c62f7b551e7888d03cea253fa72e3d0981d94f00d29f58a`

In this sample, the manifest only consisted of activities, services and receivers from the real app. So, I had trouble locating the malicious part.

In that case, I recommend *reading the code of the Meterpreter* and searching for similar parts in the infected application. There are 2 solutions to get the code of the Meterpreter:

1. Long (but educational): generate a Meterpreter APK and decompile it (with your favorite Android decompiler). This also has the advantage to show you exactly what to expect in the decompiled code.
2. Short: read the sources on GitHub (it's quick once you have the link, huh 😏)

There are several interesting parts to spot:

1. The payload uses a configuration byte array. The content in the sample will be different, but if you see a byte array in a class, give it a second look… especially that this byte string contains the IP address of the remote attacker's server !
   ```
   private static final byte [] configBytes = new byte[] { (byte) 0xde, (byte) 0xad, (byte) 0xba, (byte) 0xad, //placeholder /*8192 bytes */ 0, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, …
   };
   ```
2. Meterpreter connects to the remote server via a *Socket*. So, you can search in your Android app which part uses Sockets. There shouldn't be that many. You can typically use DroidLysis for that. Run the tool (`python3 droidlysis3.py --input thesample.apk --output dir`). Then, in the output directory, search for "Socket" in `autoanalysis.md` , and you get a list of all parts that call Socket methods (for the precise pattern that is matched, in your DroidLysis configuration, go and see `./conf/smali.conf` and search for a property named "socket": the pattern is listed just below).

   Press enter or click to view image in full size

```
106     private static void runStagefromTCP(String url) throws Exception {
107         Socket sock;
108         String[] parts = url.split(":");
109         int port = Integer.parseInt(parts[2]);
110         String host = parts[1].split("/")[2];
111         if (host.equals("")) {
112             ServerSocket server = new ServerSocket(port);
113             sock = server.accept();
114             server.close();
115         } else {
116             sock = new Socket(host, port);
117         }
```

Meterpreter source code creates a Socket server or a Socket client. Search for Sockets in your sample!

Press enter or click to view image in full size

DroidLysis shows the code instantiates a Socket and calls SocketServer.accept() and in a class named Snotq. This helps locate malicious code within the package.

3. Meterpreter reads an incoming payload Jar from the Socket with the remote server, and loads it using `DexClassLoader`. Same, we can use DroidLysis to spot which part of the code uses `DexClassLoader`. Once again, we are lucky, the only spots that use `DexClassLoader` are the malicious class `Snotq` and DroidLysis points it out. Yes, we are "lucky" because there could be non-malicious parts using `DexClassLoader`, but in reality, the use of `DexClassLoader` is often an excellent marker for suspicious activity. Even in the worse cases, you shouldn't have too many classes to inspect.

There are several other things you can notice in Meterpreter's code, but actually, those 3 are enough.

What if you don't know that Meterpreter is injected in your code? Well, in that case, you resort to the standard job of a malware analyst: decompile the sample, and analyze anything that looks out of place. I personally use DroidLysis to get hints at where to start searching, and as I said earlier, for instance, noticing use of `DexClassLoader`, or even Sockets, gives you nice hints to look at the corresponding classes.

Continued in part 2.

IOCs (March 2020):

- `992f9eab66a2846d5c62f7b551e7888d03cea253fa72e3d0981d94f00d29f58a`

- `f3d452befb5e319251919f88a08669939233c9b9717fa168887bfebf43423aca`
- `7b8794ce2ff64a669d84f6157109b92f5ad17ac47dd330132a8e5de54d5d1afc`

— cryptax
