Multidex trick to unpack Android/BianLian

cryptax.medium.com/multidex-trick-to-unpack-android-bianlian-ed52eb791e56

@cryptax May 5, 2022

@cryptax

Jan 14

.

5 min read

This article explains how to unpack sample sha256 `5b9049c392eaf83b12b98419f14ece1b00042592b003a17e4e6f0fb466281368` which was served from http://videofullizlesite9356[.]site/ApiServices-Files92752/Down at the beginning of January 2022 (see this tweet). The malware poses as a Video player and appears to be a member of the *Bian Lian* malware family (see former analysis 1 and 2). Although the malware has similarities with BankBot, I believe this precise sample is *not a banker* malware (it does not attempt to steal your banking credentials or steal from your bank account) but the **Bian Lian bot** (bulk send of SMS, USSD calls etc).

Update Jan 17, 2022. Let's try and clarify. Bian Lian is a "generic" bot. It may be used to steal passwords of banking apps. Some current samples seem to be particularly interested in stealing credentials from turkish banks.

Kudos to @U039b with whom I began this reverse engineering, and @ReBensk, .

> **Summary (spoiler?) for those who don't want to read it all :)**
>
> The malware **does not** use `DexClassLoader` to unpack the payload DEX. Instead it loads the payload as a secondary DEX through multidex support. The packer re-implements **multidex** support and mainly changes names & adds asset decryption.
>
> You can use my Java program to decrypt the asset and access the payload.

What makes this sample difficult to unpack

The most common packing mechanism consists in loading a hidden DEX with `DexClassLoader`. The sample **does not use** `DexClassLoader` (nor `PathClassLoader` or `InMemoryDexClassLoader`). Therefore, we cannot unpack by creating a hook on `DexClassLoader` that dumps its first file argument.

Many **dynamic tools fail** (let me know if you find one that works): RMS times out while trying to launch the app, House fails, and Dexcalibur fails to load the malware's project due to a bug.

This malware uses another mechanism to side load its payload. **I'll discuss at the end of the article** (section **"***So, how is the DEX loaded if not with DexClassLoader?*"). But first, let's unpack, because actually if your goal is to analyze the payload, you don't really need to understand how it is loaded.

Detecting it is packed

The main activity is `com.pmmynubv.nommztx.MainActivity`, which is not present in the wrapping APK but in the contained payload.

```
uuid                : True (Creates a random identifier. Us
.)
vibrate             : True (Uses phone vibrations)
webview             : True (Displays a URL in the WebView.
ay custom pages with JavaScript, sometimes malicious...)
wifi                : True (Tests or scans for WiFi)
zip                 : True (Zips or unzips files)
packed              : True (None)
```

DroidLysis detects the sample is packed

Unpacking

Understanding the packing mechanism takes a little bit of time, because all strings are obfuscated (fortunately JEB decrypts nearly all of them) and `DexClassLoader` — typically to load a DEX — is not used.

The flow is the following. The manifest references `com.brazzers.naughty.g` as the application. Indeed, `g` extends `Application` and can be seen as the main entry point of the app. One of the first methods to be called is the protected method `attachBaseContext`.

```
@Override  // android.content.ContextWrapper
public void attachBaseContext(Context ctx) {
    super.attachBaseContext(ctx);
    try {
        a.install_multidex(this);
    }
    catch(Exception excp) {
    }
}
```

This is where the unpacking actually begins! For clarity, I renamed the method “install_multidex”. Its original name was of course obfuscated to “a”. It is a bit difficult to spot so much happens through this simple call…
From `attachBaseContext`, the malware calls a cascade of functions which (1) locate an asset named `G9ugwFtlG1.jwi`, (2) deflates it and (3) finally decrypts it using a home-made algorithm with hard coded key `GIUh9JHGUIGIUHGokfewrofij58YV6UhYUF7gjhgv`.

The strings `G9ugwFtlG1, .jwi, GIUh9JHGUIGIUHGokfewrofij58YV6UhYUF7gjhgv` are found in a malware’s configuration class com.brazzers.naughty.h, but note they are all obfuscated.

```
package com.brazzers.naughty;

public final class h {
    public static final String a = i.a("思恙悎恚");

    public static final String b = i.a("意怊恆忮悎恵恇悎意");

    public static final String c = i.a("怄意恨恟悎悎恇協");

    public static final String d = i.a("意悎恟恟怊恢惻意恟悎意悎恟惻意怵恟恕恩悎恁怵恕恚恙恰恨恪恥悵恟恟恪恟恵怄忮恙恟忮恺");

    public static final String e = i.a("$DATA");

    public static final String f = i.a("恟恕恨恟悵恟恟悎悎悎悎恚恚恵忮怵");

    public static final String g = i.a("恣怊恩恇悎恚恆悎思恆悎恨恇怄");

    public static final String h = i.a("思恟恵意忮恨恇恟");

    public static final String i = i.a("思悎恩忮");

    public static final String j = i.a("恇忮恟恆恟恙協忮怄忮");
}
```

Obfuscated bot configuration strings. De-obfuscation occurs through i.a()
The obfuscation is not complex: a XOR with a fixed character, nevertheless I was happy JEB did the work automatically for me!

```
public static {
    h.PAYLOAD_EXTENSION = ".jwi";
    h.PAYLOAD_FILENAME = "G9ugwFtIG";
    h.directory_name = "7G8Uwty";
    h.DECRYPTION_KEY = "GIUh9JHGUIGIUHGokfewrofij58YV6UhYUF7gjhgv";
    h.e = "";
    h.CODE_CACHE_SECONDARY_FOLDER_NAME = "hf8U6UUIwiqaGgo";
    h.LOCK_FILENAME = "T9etIiaI.uw87";
    h.dotHFGg8tq = ".hFGg8tq";
    h.EXTRACTED_SUFFIX = ".weg";
    h.CODE_CACHE_NAME = "tgqaqjyg7g";
}
```

Isn't that nice? JEB automatically decrypts strings when a simple algorithm is used. JEB doesn't do all the work though, you still have to reverse engineer to understand the meaning: I manually renamed obfuscated name h.a to h.PAYLOAD_EXTENSION etc.
If you want to follow the calls:

1. com.brazzers.naughty.g.attachBaseContext(Context)
2. com.brazzers.naughty.a.a(Context)
3. com.brazzers.naughty.a.a(Context, File, File…)
4. com.brazzers.naughty.b.a(Context, String…)
5. com.brazzers.naughty.b.c()
6. com.brazzers.naughty.b.a(ZipFile, ZipEntry…)
7. com.brazzers.naughty.k.a(String, InputStream, OutputStream)

We'll see later the malware is actually following the normal flow for loading secondary DEXes, except the names are obfuscated.

Automating the unpacking

I basically copied the malware's unpacking code in `com.brazzers.naughty.k.a`, did a little adaptation, and finally worked out a Java program that automatically unpacks the malware (get the code here). Run the program in the directory where `G9ugwFtIG1.jwi` is available (`./assets/7G8Uwty`), and it will unpack and create a `unpacked.zip` which contains the payload DEX.

```
decode(): processing 512 bytes...
decode(): processing 507 bytes...
decode(): processing 512 bytes...
decode(): processing 512 bytes...
decode(): processing 323 bytes...
decode(): finished reading
[+] Decrypted
----------------------------------------
```

Static program to unpack the asset — java UnpackJwi

```
$ unzip -l unpacked.zip Archive:  unpacked.zip  Length      Date    Time    Name-----
----  ---------- -----   ----   906456  2022-01-14 11:05   classes.dex---------
-------   906456                     1 file
```

So, how is the DEX loaded if not with DexClassLoader?

The malware **uses the multidex scheme** to load the payload as secondary DEX. This method has existed for a couple of years (e.g. Android/Rootnik using it in 2017), but I hadn't seen it for a while. In 2022, it seems we have a new packer in the wild which uses this technique as the same packer is used here in a sample of Flubot.

The technique consists in re-writing the way applications load multiple DEX. The Android code can be found here. The classes `MultiDex` and `MultiDexApplication` are the core of the functionality. They implement support of APKs with multiple DEX. Everything begins in indeed in `attachBaseContext` of `MultiDexApplication.` The malware re-implements `MultiDexApplication` , `MultiDex` and `MultiDexExtractor` with little changes part code re-organization and obfuscated names:

- `MultiDexApplication` is found in `com.brazzers.naughty.g`
- `MultiDex` is within in `com.brazzers.naughty.a`
- and `MultiDexExtractor` is `com.brazzers.naughty.b`

The two main functional changes are :

1. . The extracted DEX will be located in a directory named `hf8U6UUIwiqaGgo` instead of the standard `secondary-dexes` name, the extracted suffix will be `.weg` instead of `.zip` , and some other minor details like the lock file name is changed to `T9etIiaI.uw87` instead of `MultiDex.lock` . The goal is obviously to complicate reverse engineering, but also to make the files less noticeable should they be spotted during extraction on the device.
2. . Compare the original code with the malware's version below.

```
try {
    ZipEntry classesDex = new ZipEntry("classes.dex");
    // keep zip entry time since it is the criteria used by Dalvik
    classesDex.setTime(dexFile.getTime());
    out.putNextEntry(classesDex);

    byte[] buffer = new byte[BUFFER_SIZE];
    int length = in.read(buffer);
    while (length != -1) {
        out.write(buffer, 0, length);
        length = in.read(buffer);
    }
    out.closeEntry();
```

Original code from (not malicious!)

```
try {
    ZipEntry classesDex = new ZipEntry("classes.dex");
    classesDex.setTime(dexFile.getTime());
    out.putNextEntry(classesDex);
    InflaterInputStream iis = new InflaterInputStream(in);
    InflaterOutputStream ios = new InflaterOutputStream(out);
    k.decrypt(MultiDexExtractor.key, iis, ios);
    ios.close();
    iis.close();
    out.closeEntry();
}
```

Malware's version. The input is deflated and decrypted.
The next article will deal with the reverse engineering of the payload DEX.

More recent samples of January 14 (today)

Some newer samples of that malicious video application have been released today: see Twitter. I haven't looked in all samples yet, but at least the first one, 01658_Video_Oynatıcı.apk (sha256: `d105764cd5383acacd463517691a0a7578847a8174664fc2c1da5efd8a30719d`) does not use the same packer. It uses the common `DexClassLoader` method, actually the same packing mechanism as this sample. Watch for the unpacked payload in a file named `maXclr.json`.

```
generic_x86_64:/data/data/com.friend.bronze/app_DynamicOptDex # lsmaXclr.json
maXclr.json.prof
```

The payload DEX (`maXclr.json`) is a Bian Lian sample.

— the Crypto Girl