Inside KangaPack: the Kangaroo packer with native decryption

cryptax.medium.com/inside-kangapack-the-kangaroo-packer-with-native-decryption-3e7e054679c4

@cryptax June 23, 2023

@cryptax

--

In this blog post, we unpack a malicious sample sha256: `2c05efa757744cb01346fe6b39e9ef8ea2582d27481a441eb885c5c4dcd2b65b` . The core decryption of the payload is implemented at native level. I named the packer **KangaPack** (you'll understand why when reading this article)**,** it also goes under the name *Packed.57103*, I am unaware of any other name.

Teaser: from decompiled code, we'll see exactly how the packer decrypts the payload, we'll use JEB decompiler to decompile an ARM library, we'll use ImHex with a DEX pattern to understand where the payload is hidden.

Where to begin

The sample is packed: its main (`com.dsfdgfd.sdfsdf.MainActivity`) and receiver (`com.shounakmulay.telephony.sms.IncomingSmsReceiver`) are not available from the wrapping APK:

```
  <application android:appComponentFactory="androidx.core.app.CoreComponentFactory"
android:extractNativeLibs="true" android:icon="@mipmap/ic_launcher" android:label="遠
通電收ETC" android:name="com.hwgapkspv.gouhwkh.BQddpmHvTsWgSIexmtrw"
android:networkSecurityConfig="@xml/network_security_config"
android:usesCleartextTraffic="true">    <receiver android:exported="true"
android:initOrder="1048"
android:name="com.shounakmulay.telephony.sms.IncomingSmsReceiver"
android:permission="android.permission.BROADCAST_SMS">      <intent-filter>
<action android:initOrder="1048"
android:name="android.provider.Telephony.SMS_RECEIVED"/>      </intent-filter>
</receiver>    <activity
android:configChanges="density|fontScale|keyboard|keyboardHidden|layoutDirection|local
 android:exported="true" android:hardwareAccelerated="true" android:initOrder="1048"
android:launchMode="singleTop" android:name="com.dsfdgfd.sdfsdf.MainActivity"
android:theme="@style/LaunchTheme" android:windowSoftInputMode="adjustResize">
<meta-data android:initOrder="1096"
android:name="io.flutter.embedding.android.NormalTheme"
android:resource="@style/NormalTheme"/>
```

The application’s entry point is `com.hwgapkspv.gouhwkh.BQddpmHvTsWgSIexmtrw`. We decompile its `attachBaseContext()`. Let’s look into it step by step.

The malware unzips its own APK in a working directory `/data/data/<PACKAGE_NAME>/app_JKnBkiqZbmLnxDv/HIFlFQ`.

```
protected void attachBaseContext(Context base) {        int i;
super.attachBaseContext(base);        // get application source dir        File
app_srcdir = new File(this.getApplicationInfo().sourceDir);        File dir =
this.getDir("JKnBkiqZbmLnxDv", 0);  // app_JKnBkiqZbmLnxDv        // get sub
directory HIFlFQ        File HIF_dir = new File(dir, "HIFlFQ");        ArrayList list
= new ArrayList();        // if ./app_JKnBkiqZbmLnxDv/HIFlFQ create it an unzip the
application        if(!HIF_dir.exists()) {
Ysaplgfew9laf2lsdsa.unzip_metainf(app_srcdir, HIF_dir, true);        }...
```

Then, it parses each unzipped file:

```
this.dex_extension = ".dex";this.classes_dex_file = "classes.dex";// get all files of
the unzipped dirFile[] arr_file = new File(dir, "HIFlFQ").listFiles();  int v =
0;while(v < arr_file.length) {   ...
```

Whenever it encounters a DEX file, it reads it, decrypts, parses the resulting ZIP and adds to a list each file of that ZIP. By default, the code is obfuscated, I have edited the names to ease its understanding. We’ll look into `writeDex` (real name: `odgnstswehaxibqwemcbvdand`) `do_decrypt_file` (real name: `bhwi8sma09d23ssva`) afterwards.

In our case, there is a single DEX file: `classes.dex` .

```
File file = arr_file[v];
String f = file.getName();
if(f.equals(this.classes_dex_file)) {
    // adds ".dex" extension to filename
    // if filename is "classes.dex" -> "classes.dex.dex"
    String the_dex_dex = file.getPath() + this.dex_extension;



    // reads the input file as a byte array and writes that to the dex dex file
    this.writeDex(BQddpmHvTsWgSIexmtrw.file2bytes(file), the_dex_dex);



    // decrypts the file (f) and writes the result to the_dex_dex
    // the decrypted file is a ZIP: lists files inside it
    File[] arr_file1 = this.do_decrypt_file(new File(the_dex_dex), f).listFiles();

    // this loop is basically for(i=0; i<arr_file1.length; i++)     int max =
arr_file1.length;    i = 0;    while(true) {        label_20:        if(i >= max) {
goto loop_finished;        }        // add each file to a list
list.add(arr_file1[i]);    break;
```

Once this is done, the malware installs the listed payload files on the smartphone, thus making their code available to the malware. We’ll see that as well later.

Finding the payload

We dig into `writeDex` (real name: `odgnstswehaxibqwemcbvdand`). We notice the method

1. Retrieves an integer from the last 4 bytes of the file
2. Allocates a buffer whose size is represented by that integer
3. Reads the required length before that integer and stores it in a resulting file.

```
private void writeDex(byte[] bytearray, String filename) throws IOException {
byte[] dexlen = new byte[4];        // we copy the last 4 bytes of the file in
dexlen. This is an int.        System.arraycopy(bytearray, bytearray.length - 4,
dexlen, 0, 4);          // read the bytes of dexlen and convert to int.        int
dex_length = new DataInputStream(new ByteArrayInputStream(dexlen)).readInt();
System.out.println(Integer.toHexString(dex_length));        byte[] newdex = new
byte[dex_length];        // we copy the last DEX_length bytes (-4) to newdex array
System.arraycopy(bytearray, bytearray.length - 4 - dex_length, newdex, 0,
dex_length);          File file = new File(filename);        try {
FileOutputStream localFileOutputStream = new FileOutputStream(file);
localFileOutputStream.write(newdex);            localFileOutputStream.close();  // we
write that to filename            return;        }        catch(IOException
localIOException) {            throw new RuntimeException(localIOException);        }
}
```

This means that the payload is actually embedded inside the wrapping `classes.dex` itself!

To start unpacking manually, we read the last bytes of `classes.dex`: 00 04 A5 D0. This corresponds to a length of 304592 bytes.

```
$ hexdump -C classes.dex | tail -n 30005eae0  3e 15 97 d7 d5 f8 74 16  5f ed b5 8c 6c
aa d0 af  |>.....t._...l...|0005eaf0  00 04 a5 d0
|....|0005eaf4
```

The original `classes.dex` file has a length of 387828 bytes. So, we need to copy bytes starting at 387828–304592–4=83232. Not surprisingly, the extract part has no known format: it is encrypted.

```
$ dd if=classes.dex of=todecrypt skip=83232 count=304592 bs=1304592+0 records
in304592+0 records out304592 bytes (305 kB, 297 KiB) copied, 0,610131 s, 499 kB/s$
file todecrypt todecrypt: data
```

Let’s come back to where the encrypted payload is. It is — except for the last 4 bytes — at the end of the original (packing)`classes.dex`. How is that possible? We load the DEX in ImHex and apply the DEX pattern.

File_size, inside the DEX header, is 387828.

The DEX header has the correct length of 387828. This means we do not have “a DEX, and then an encrypted payload”, but that the encrypted payload is *included* in the DEX format itself.

DEX data ends at 0x1451F included.
A DEX normally ends after its link_data, which is just after its data section. Here, we have no link_data, so the DEX ends at the end the data section, i.e at 0x1451F. Notice that 0x14520 = 83232 which is exactly the offset for the encrypted payload.

NB. For correct pattern detection of DEX files in ImHex, I added the following to struct Dex in `dex.hexpat` . I will push the update soon, but meanwhile, you can do it for yourselves:

```
    u8 data[header.data_size] @header.data_off;    map_list map_list @
header.map_off;    u8 link_data[header.link_size] @ header.link_off;
```

So, the packer cleverly embeds the encrypted payload inside the DEX file format, fixes the size, signature and checksum.

Layout of packer’s classes.dex

Understanding decryption

Now let’s go back to `do_decrypt_file` (real name: `bhwi8sma09d23ssva`). The function calls `abddesCrypt`.

```
 byte[] arr_b = Molfwernpozxswfsg.abcdesCrypt(Fsolwtym0asmsaw.File2byte(file));
FileOutputStream decrypted_file = new FileOutputStream(new File(file.getPath()));
decrypted_file.write(arr_b); decrypted_file.flush(); decrypted_file.close();
```

And as I said in the introduction, this calls native code:

```
import android.content.Context;



public class Molfwernpozxswfsg {
    static {
        System.loadLibrary("tahagmaxss");
        System.loadLibrary("apksadfsalkwes");
    }

    public static native byte[] abcdesCrypt(byte[] arg0) {    }
```

The function is implemented in `libapksadfsalkwes.so`. We decompile it. The code is straight forward. The encrypted payload is retrieved from v5 and stored in ciphertext_ptr (the names have been edited for clarity). Then, the code allocates buffers and initializes

OpenSSL's EVP API. AES decryption is initialized for CBC mode using a symmetric key and an IV. The routine asks to decrypt the ciphertext (`EVP_DecryptUpdate`) and retrieves the result (`EVP_DecryptFinal_ex`)

adcdesCrypt function decompiled by JEB
To decrypt, we need to find the key and IV. See they point to the same address!

```
LOAD.data:0002E000 AES_SECRET_KEY   dd 27C7Ch                                  ; xref:
Java_com_hwgapkspv_gouhwkh_Molfwernpozxswfsg_abcdesCrypt+6Ch (data-dyn-adv) /
Java_com_hwgapkspv_gouhwkh_Molfwernpozxswfsg_abcensCryption+7Ch (data-dyn-adv) /
Java_com_hwgapkspv_gouhwkh_Molfwernpozxswfsg_abcdesCrypt+64h (data-dyn-adv) /
Java_com_hwgapkspv_gouhwkh_Molfwernpozxswfsg_abcensCryption+74h (data-dyn-adv) /
2DD00h (ptr)LOAD.data:0002E004 AES_IV          dd 27C7Ch   ....rodata:00027C7C
aJ2K10uXshMh9UGP db "j2K10uXshMh9UGPS",0
```

We have all we need to decrypt the payload.

```
from Crypto.Cipher import AESciphertext =
open('encrypted_payload','rb').read()key=b'j2K10uXshMh9UGPS'iv=keycipher=AES.new(key,
AES.MODE_CBC, iv)plaintext=cipher.decrypt(ciphertext)f =
open('decrypted.zip','wb')f.write(plaintext)f.close()
```

The resulting file is a ZIP. It contains the payload DEX.

Loading the payload

Once decrypted, the malware calls an `install` function (real name: `ir68n8s9tdadlbjh`). The code is difficult to understand, but fortunately, we don't have to: I have already seen this code, it is part of Android itself. Compare it with this source code.

```
private static void install(ClassLoader loader, List additionalClassPathEntries, File
optimizedDirectory) throws IllegalArgumentException, IllegalAccessException,
NoSuchFieldException, InvocationTargetException, NoSuchMethodException {
IOException[] dexElementsSuppressedExceptions1;            Object pathListField =
BQddpmHvTsWgSIexmtrw.findField(loader, "pathList").get(loader);            ArrayList
suppressedExceptions = new ArrayList();            Log.d("BQddpmHvTsWgSIexmtrw",
"Build.VERSION.SDK_INT " + Build.VERSION.SDK_INT);
if(Build.VERSION.SDK_INT >= 23) {
BQddpmHvTsWgSIexmtrw.expandFieldArray(pathListField, "dexElements",
BQddpmHvTsWgSIexmtrw.makePathElements(pathListField, new
ArrayList(additionalClassPathEntries), optimizedDirectory, suppressedExceptions));
} — Cryptax
```

It implements installation of all files listed in `additionalClassPathEntries` argument. In that list, we'll have the `classes_ogr.dex` file. Thus, the file will be loaded and everything inside (`com.dsfdgfd.sdfsdf.MainActivity` for instance) becomes available.

Packer naming

This packer isn’t recognized by APKiD (yet) and I don’t know where it is from. If you have any clue, please contact me Twitter or Mastodon.social (handle: @cryptax). Meanwhile, I am going to name it. As it contains the payload inside the DEX file inside, I’ll call it **KangaPack** (kudos to Kangaroos).

— Cryptax