Hide Android Applications in Images

Axelle Apvrille, Fortinet, FortiGuard Labs
120, rue Albert Caquot 06410 Biot, France
aapvrille@fortinet.com

Ange Albertini, Corkami
ange@corkami.com

September 18, 2014

Abstract

With AngeCryption, [Alb14b] has demonstrated it is possible to encrypt any input into a chosen JPG or PNG image. For a mobile malware author, this is particularly interesting when applied to Android packages (APK). Indeed, in that case, an attacker can craft a seemingly genuine wrapping APK which contains a valid image (e.g a logo) as resource or asset. However, the code is able to transform this unsuspicious image into another APK, carrying the malicious payload. The attacker installs that APK, and performs his/her nefarious deeds.

Such an attack is highly likely to go unnoticed, because the wrapping APK hardly has anything suspicious about it, and nothing about the payload APK leaks as it is encrypted. Additionally, the attack works with any payload and currently on any version of Android.

In short, what you see is on the left (an image). What there really is on the right (an Android application).

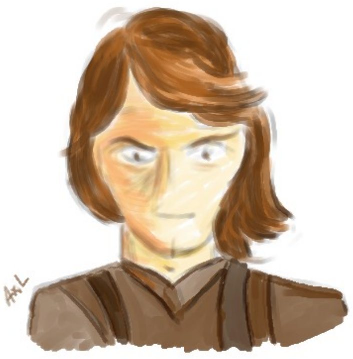


Figure 1: A PNG of Anakin Skywalker :)

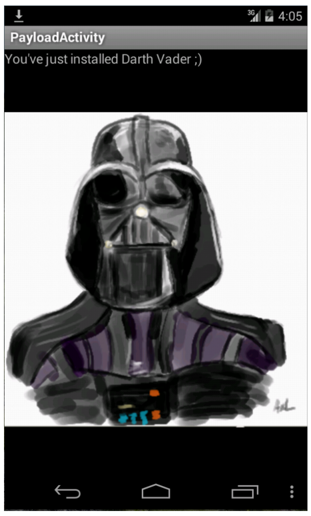


Figure 2: Hidden payload Android application

In this talk, we show the Proof of Concept application we have built. Naturally, the payload APK is harmless (it displays an image of Darth Vader). It is hidden in a genuine image of Anakin Skywalker. We show Android disassemblers are unable to detect anything of the real payload. We explain our PoC's implementation and how this attack works. Finally, we discuss potential solutions to mitigate such an attack.

The PoC has been sent to the Android Security Team on May 27, 2014, accepted and will be fixed in future releases. We are happy to be in the Android Security hall of fame [HOF14].

1 Origins

Android malware authors have always liked to hide malicious parts in applications they create or trojan. For example, Android/DroidCoupon.A!tr hides a rooting exploit in a PNG image inside the sample's raw resource directory (see Table 1). Images are usually found in all genuine applications, they are consequently a good choice for malware authors to hide malicious code and hope it won't be detected.

Android malware name and sha256 hash	**Year of discovery**	**Obfuscation**
Android/Gamex.A!tr ae7a20692250f85d7a2ed205994f2d26f2d695aef15a9356938454bccbbbd069	2013	Assets contains a file named logos.png. This is not a PNG, but a ZIP, and it de-zips to different valid output whether XORed with key (18) or not.
Android/SmsZombie.A!tr 45099416acd51a4517bd8f6fb994ee0bb9408bdd80dd906183a3cdb4b39c4791	2012	Hides malicious package in a33.jpg
Android/DroidCoupon.A!tr 94112b350d0feceff0a788fb042706cb623a55b559ab4697cb10ca6200ea7714	2011	The Rage Against the Cage exploit is hidden in a PNG file in raw resources

Table 1: Examples of samples hiding malicious packages in resource files

In some cases, they even try more tricky solutions like in Android/Gamex.A!tr where the asset, misleadingly named logos.png, unzips differently (and correctly) whether XORed with a key or not. This is close to what researchers commonly refer to as *polyglots* [Alb14a]: a file which is valid for different formats. So, how about creating a PNG/APK polyglot? An APK is nothing more than a ZIP, and creating a polyglot file which is valid for both formats (PNG and ZIP) is actually quite easy. PNG files must start with a PNG file header at offset 0. ZIP tools however are less restrictive and allow the ZIP file header to be located somewhere else later in the file. So, a PNG/APK polyglot can be simply creating by prefixing a PNG to an APK:

```
$ cat myphoto.png app.apk > polyglot.png
$ file polyglot.png
polyglot.png: PNG image data, 382 x 385, 8-bit/color ...
$ unzip -l polyglot.png
Archive:  polyglot.png
warning [polyglot.png]:  134834 extra bytes at beginning...
  Length      Date    Time    Name
---------  ---------- -----   ----
      636  2014-04-04 14:03   res/layout/main.xml
     1248  2014-04-04 14:03   AndroidManifest.xml
      892  2014-04-04 14:03   resources.arsc
...
```

Fortunately (or unfortunately depending from which angle you look at it), Android is more restrictive than common zip tools, and it refuses to load such APKs:

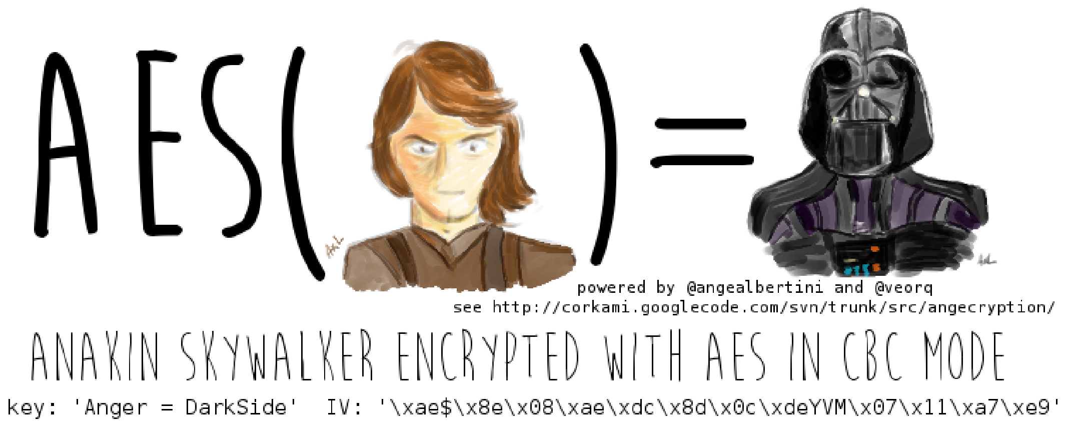

AES( ) =
powered by @angealbertini and @veorq
see http://corkami.googlecode.com/svn/trunk/src/angecryption/
ANAKIN SKYWALKER ENCRYPTED WITH AES IN CBC MODE
key: 'Anger = DarkSide' IV: '\xae$\x8e\x08\xae\xdc\x8d\x0c\xdeYVM\x07\x11\xa7\xe9'

Figure 3: When encrypted with AES-CBC and the given key and IV, the image of Anakin Skywalker encrypts to the image of Darth Vader. This proves it is possible to manipulate the output of encryption bytes.

```
$ adb install polyglot.png
Whoops: didn't find expected signature
read_central_directory_entry failed
file 'polyglot.png' is not a valid zip file
```

So, creating a PNG/APK polyglot actually fails. We then contemplate a tricker solution: AngeCryption. A valid PNG image will turn into a valid APK after decryption.

2 AngeCryption

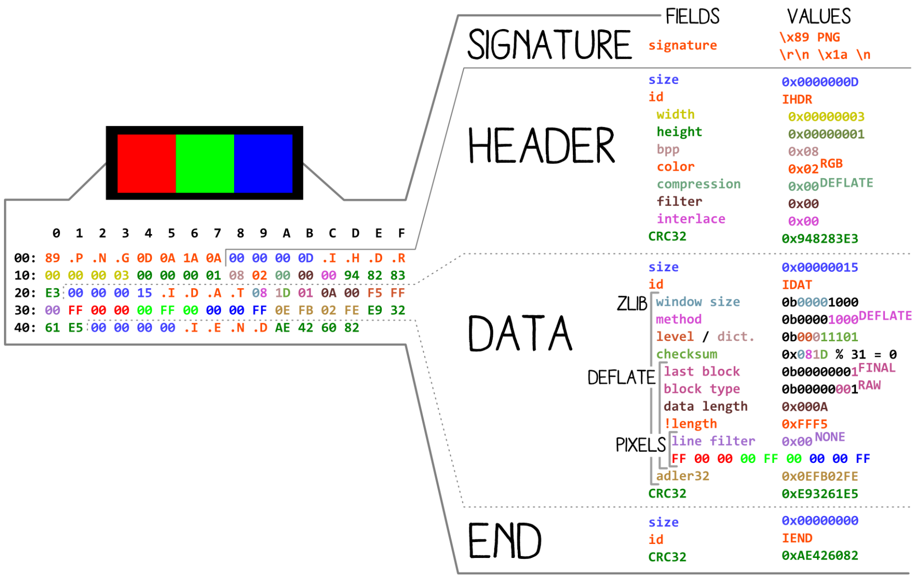
2.1 Introduction

AngeCryption [Alb14b] has demonstrated it is possible to encrypt any input file into a JPG or PNG image. For example, in [Apv14] we have shown we are able to encrypt Anakin Skywalker into Darth Vader, and reciprocally decrypt Darth Vader into Anakin Skywalker. See picture below.

Encrypting with AES-CBC the image of Anakin with the specified key and Initialisation Vector results in the image of Darth Vader - and reciprocally.

2.2 Short crypto reminder

A short reminder on cryptography is required to explain AngeCryption. However, we insist that AngeCryption is understandable without knowing much more about cryptography. AES (like DES, Blowfish etc) is block cipher, i.e it processes input data of a given size only. In the case of AES, the block size is of 16 bytes. This means that, essentially, AES is only able to process 16 bytes. To chain bigger documents, people use chaining methods which explain how to process each blocks. CBC, Cipher Block Chaining, is a common chaining method [NIS01].



```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

00: 89 .P .N .G 0D 0A 1A 0A 00 00 00 0D .I .H .D .R
10: 00 00 00 03 00 00 00 01 08 02 00 00 00 94 82 83
20: E3 00 00 00 15 .I .D .A .T 08 1D 01 0A 00 F5 FF
30: 00 FF 00 00 00 FF 00 00 00 FF 0E FB 02 FE E9 32
40: 61 E5 00 00 00 00 .I .E .N .D AE 42 60 82
```

Figure 4: PNG file format illustration by Ange Albertini

For its operation, it requires an additional parameter which is called the Initialisation Vector (IV). Exactly, if we call P_i the 16-byte chunks of plaintext (unencrypted data), K the encryption key, IV the 16-byte initialization vector, then C_i the corresponding ciphertext blocks (encrypted data) are computed like this: $C_0 = AES_K(P_0 \oplus IV)$ and $C_i = AES_K(P_i \oplus C_{i-1})$.

2.3 How AngeCryption works

A given input file cannot encrypt into a given output image as such, without a little trick. We do not *exactly* encrypt the input file but something that *looks identical*, and it does not *exactly* output the expected image, but something that *looks identical*.

Precisely, the format of PNG files is illustrated at Figure 4

We output a PNG with the following layout.

- File Header. PNG start with a fixed 8-byte 'signature' which is `0x89 PNG 0x0d 0x0a 0x1a 0xa`. It needs to be correct for the file to be recognized as a valid PNG.

- Garbage chunk. We insert here data which will be ignored by the tool that reads the file as image, e.g comment chunks, end of object. For PNGs, a chunk is made of:

  - Chunk length (4 bytes)
  - Chunk id (4 bytes). We use a dummy chunk type like 'aaaa', so that tools ignore this chunk.
  - Chunk data.
  - CRC32 of chunk data and id.

- Header chunk (IHDR). In theory, PNG specifications [RPB99] mandate that PNGs begin with a header chunk (IHDR). In practice, this is seldom enforced by tools as we will demonstrate.

- Data chunk (IDAT). Those are the chunks that contains the correct image data blocks
- End chunk (IEND), which terminates the PNG.

To get such an output, we:

- Select an appropriate IV. We want our first cipher block C_1 to be equal to the PNG file header (8 bytes) + chunk length (4 bytes) + chunk id (4 bytes). By chance, this fits in an AES block (they are 16 bytes long). We also know what P_0 looks like: the first block of the input file. Finally, we know K. So, we select IV such that: $IV = AES_k^{-1}(C_i) \oplus P_i$. In other words, we select the IV we need to get the first cipher block we want. Note that in real encryption cases, IV is selected randomly.
- Generate a modified input file. This modified input data has appended data at the end. This appended data is the *decrypted* CRC32 checksum + target image file blocks + end chunk. Why? Because when we encrypt decrypted data, we get the original data. So, if we encrypt those decrypted image blocks, we get the image blocks: $AES_k(AES_k^{-1}(P_i)) = P_i$.

Note that AngeCryption is independant of AES, CBC or PNG. It just requires that:

- First cipherblock can be controlled (e.g via IV selection)
- Source format tolerate appended data
- Header + chunk declaration data fits in block size.

2.4 Tool

An AngeCryption tool is released on `http://corkami.googlecode.com/svn/trunk/src/angecryption/angecrypt.py`.

For the end-user, the goal is to be able to encrypt an input file into a target image. To do so, AngeCryption must manipulate the input file into a modified file, without altering its content. The command line is:

```
$ python angecrypt.py inputfile targetimage modifiedinput key algorithm
```

where:

- inputfile. The initial file the end-user wants to encrypt into a target image.
- targetimage. That's what we want the encrypted file to look like. Current supported formats for output are PDF, PNG, JPG, FLV.
- modifiedinput. That's the input file modified by angecrypt.py so that encryption will indeed encrypt to targetimage. Manipulation of the input file is required: encrypting the input file directly cannot produce the targetimage. However, note that modifiedfile remains *content-identical* to inputfile, i.e if inputfile is an APK, then modifiedfile.apk is also a valid APK which installs the same application.
- key. The encryption key to use. Provide a string of valid length.

- algorithm. Encryption algorithm to use. Currently, this is either AES128-CBC or 3DES-EDE2-CBC, with no padding.

So, as output, the tool produces:

1. A modified input file.
2. A generated Python script to test and encrypt the input file. In this script, the appropriate IV to use is generated.

3 PoC application

3.1 Goal

Essentially, what we wish to prove is that **it is possible to embed undetectable, valid and runnable bytecode** in a wrapping APK. By *undetectable*, we mean that static analysis, such as disassembly, of the wrapping APK does not reveal anything particular about that bytecode (apart if we undo the encryption packing).

Important notes:

- Our PoC embeds an Android application, but we could basically have it embed whatever we want. For example, just a Dalvik Executable.
- We have not paid attention to obfuscating the wrapping APK. We believe this is outside the scope of the paper, as there is no particular problem in using off-the-shelf obfuscators such as ProGuard, DexGuard etc.
- We have not tried either to hide the installation of the payload APK. There are several ways to do this, for instance using the DexClassLoader class [Str12]. As this is a known technique, we believe this is also outside the scope of this paper.

3.2 Demo

We have created a PoC Android application containing a valid PNG image of Anakin Skywalker, which installs a payload APK that displays an image of Darth Vader.

We disassemble the PoC application and check that the string is nowhere to be seen in the wrapping APK. For example, with baksmali:

```
$ java -jar baksmali.jar -o ./smali PocActivity-debug.apk
$ grep -r "Darth Vader" ./smali
$
```

The assets only contain one valid (viewable) PNG file (see Figure 1):

```
$ file anakin.png
anakin.png: PNG image data, 3684634137 x 1133979790, 45-bit
```

Now, let's show it works:

1. Install the application:

```
adb install PocActivity-debug.apk
```

2. Launch the application (Figure 5) and press the button.
3. This triggers the hidden payload APK (Figure 6)! This demonstration works with any payload application. **Note trickier implementations can conceal the installation of the payload APK**.

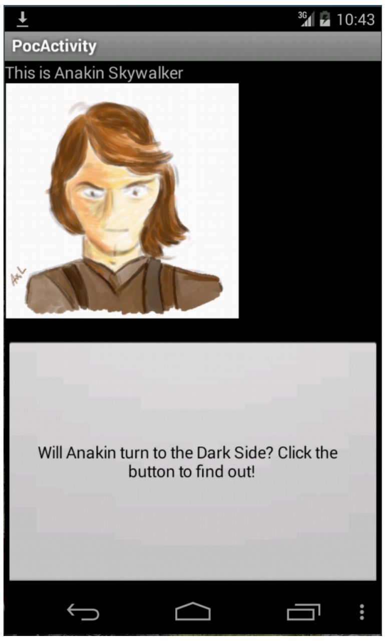

10:43
PocActivity
This is Anakin Skywalker
Will Anakin turn to the Dark Side? Click the button to find out!

Figure 5: Wrapping APK screenshot. Press the button to trigger the install of the hidden payload APK

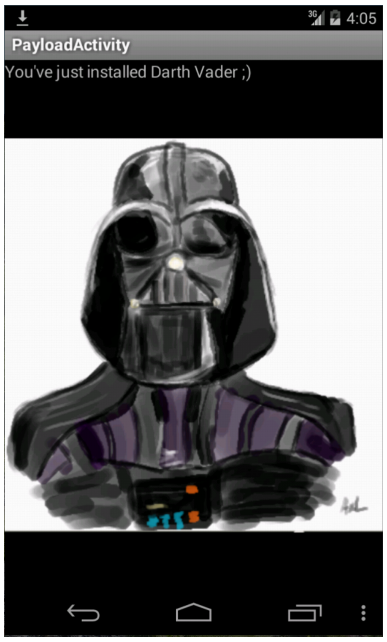


Figure 6: Hidden payload APK

3.3 Implementation

3.3.1 Creating the encrypted payload APK

We have implemented a dummy payload APK. In our case, this APK simply displays an image, but anything is possible. We are going to encrypt this APK into a PNG of Anakin Skywalker.

```
$ python angecrypt.py payload.apk anakin-original.png payload-similar.apk
     'Anakin= DarkSide' aes
```

For example, this outputs `payload-similar.apk` and generates a Python script to encrypt it:

```
algo = AES.new('Anakin= DarkSide', AES.MODE_CBC,
   '\xd3\x9e\x0c\xef#p*\xa3\xe9\x8a\xcc:+\xf0\x1a\xec')
with open('payload-similar.apk', "rb") as f:
        d = f.read()
d = algo.encrypt(d)
with open("dec-" + 'anakin-original.png', "wb") as f:
        f.write(d)
```

As such, the demo cannot immediately work, because payload-similar.apk does not unzip correctly[1]:

```
$ unzip -l payload-similar.apk
Archive:  payload-similar.apk
  End-of-central-directory signature not found.  Either this file is not
  a zipfile, or it constitutes one disk of a multi-part archive.  In the
  latter case the central directory and zipfile comment will be found on
  the last disk(s) of this archive.
unzip:  cannot find zipfile directory in one of payload-similar.apk or
        payload-similar.apk.zip, and cannot find payload-similar.apk.ZIP, period.
```

Unzipping fails because some tools like unzip do not accept too much appended data at the end, the end being marked by a record called **End-of-central-directory** (EOCD). To work around, we add another EOCD at the end of our APK: there are thus 2 EOCD in the APK.

```
$ dupe_eocd.py payload-similar.apk
```

Precisely, the generated APK has the following format:

- Payload APK - unmodified. That's the application that displays Darth Vader in our PoC. Note there will be an EOCD at the end of the application.
- $AES_{-1}(CRC32 + IHDR + IDAT + IEND)$ where
  - CRC32 is the CRC32 checksum of the garbage chunk we inserted in the image.
  - IHDR corresponds to the PNG header chunk
  - IDAT corresponds to the PNG data chunks. That's where the pixels of Anakin are.
  - IEND corresponds to a terminating PNG chunk
- Padding bytes. Those bytes ensure that the APK size is a multiple of 16, so that it can be processed by AES as such.
- EOCD

The unzip tool is happy with this APK.
Finally, we encrypt this APK (using the script which was generated earlier[2]:

```
$ python encrypt-apk.py
```

The APK does encrypt to a valid PNG of Anakin Skywalker (see Figure 7) because:

- The IV has been selected so that AES(beginning of APK)= Header + garbage chunk
- Appended data corresponds to decrypted data. Encrypting decrypted data is like doing nothing: $AES(AES_{-1}(CRC32 + Anakin + IEND)) = CRC32 + Anakin + IEND$. We get the garbage chunk CRC32 checksum, the chunk containing the image of Anakin and the end chunk.
- In crypto, CBC chaining also affects blocks after itself. So, adding padding bytes and EOCD to the file only affects cipherblocks after the end chunk. AES(Padding bytes+EOCD) produces garbage bytes at the end of the image. Those bytes are ignored because they appear after the end chunk.

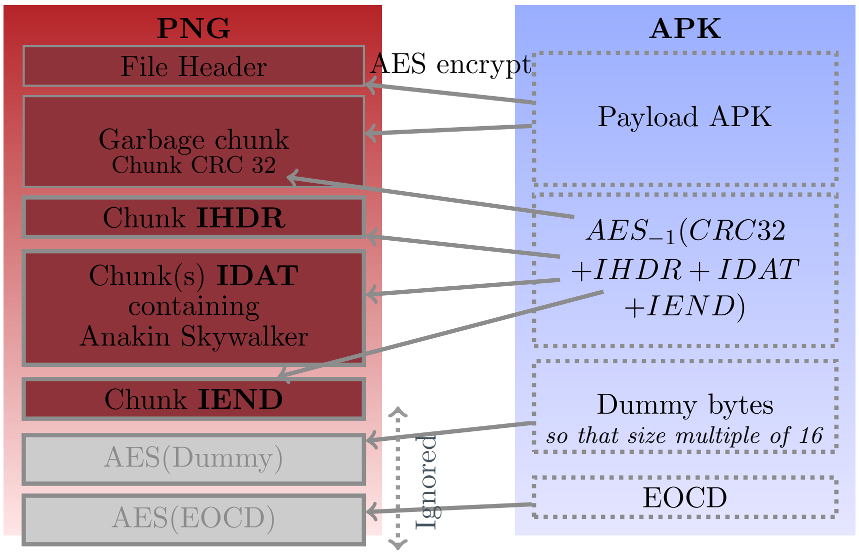


Figure 7: Real layout of PNG and APK in the demo

We check that the encrypted APK looks like a real PNG of Anakin Skywalker. It does :) This is the PNG we will insert in the assets of a wrapping APK. Just to be sure it will work, we check that the decrypted PNG installs on Android:

```
$ python decrypt-png.py
$ adb install decrypted.apk
2893 KB/s (402896 bytes in 0.135s)
        pkg: /data/local/tmp/decrypted.apk
Success
```

3.3.2 Wrapping APK

There are several ways to implement such an APK. We chose a simple and straight forward method (but it can be enhanced):

- Read the asset using `AssetManager.getAssets()` and open the PNG. The payload APK could also have been hidden in raw resources.
- Decrypt the asset using cipher AES/CBC/NoPadding.
- Write the decrypted APK to the SD card. Another writeable location would do fine too. Alternatively, there are ways to load DEX files as byte arrays (see `openDexFile` in dalvik.system.DexFile).
- Install the APK using the following piece of code:

  ```
  Intent intent =new Intent(Intent.ACTION_VIEW);
  intent.setDataAndType(Uri.fromFile(new File(filename)),
  ```

[1]Android packages (.apk) actually are Zip files (.zip)
[2]Do not forget to customize the filenames to encrypt the *modified duplicate EOCD* `payload-similar.apk` file.

```
    "application/vnd.android.package-archive");
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);
```

There are other ways to do this (calling `setClassName()` for example). This implementation has the advantage of working on Android 4.4, but it triggers an install screen that an attacker might want to hide. Concealing the install screen is feasible (e.g using `DexClassLoader`), but is out of the scope of this paper.

4 Status

This PoC has been tested over Android 4.4.2 (latest version at the time of writing the paper).

The PoC has been sent to the Android Security Team on May 27, 2014. On June 6, 2014, the team implemented a fix prohibiting any appended data after EOCD. We are however uncertain the fix makes sure to look after the *first* EOCD, and have consequently filed an issue on June 19,2014. The Android Security Team answered they'd be looking into it.

5 Conclusion

It works :) We are able to hide a payload APK inside an apparently genuine APK. This genuine wrapping APK does not reveal anything of the payload APK. This can be used maliciously by an attacker to hide his real intents, or by packers.

How can we detect this?
Currently, there is no real way to detect what the payload APK does, apart from actually decrypting the PNG. Security engineers might consider the following:

- Keep an eye on applications which are decrypting resources or assets. Our wrapping code for instance is easy to understand. It could be obfuscated though.
- Run the application in a sandbox and check for unexpected behaviour. Indeed, this technique hides the payload application/bytecode, not the behaviour. When the wrapping APK is launched, the payload PAK will get executed.
- Add stronger constraints to APKs so that an image can no longer decrypt to a valid APK. For instance, forbidding appended data in the ZIP format after the first End Of Central Directory will have AngeCryption fail. This is the option the Android Security Team chose and have started fixing on June 6th 2014. `https://android-review.googlesource.com/#/c/96603/`

References

[Alb14a] Ange Albertini. This PDF is a JPEG; or This Proof of Concept is a Picture of Cats. *Journal of PoC —— GTFO*, 3, 2014.

[Alb14b] Ange Albertini. When AES(*)=*, April 2014. `https://corkami.googlecode.com/svn/trunk/src/angecryption/slides/AngeCryption.pdf`.

[Apv14] Axelle Apvrille. AngeCryption at Insomni'Hack, March 2014. http://blog.fortinet.com/AngeCryption-at-Insomni-Hack/.

[HOF14] Android Security Acknowledgements, 2014. https://source.android.com/devices/tech/security/acknowledgements.html.

[NIS01] NIST. Recommendation for Block Cipher Modes of Operation - Modes and Techniques, 2001. Special Publication 800-38A, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[RPB99] Glenn Randers-Pehrson and Thomas Boutell. PNG (Portable Network Graphics) Specification. Technical report, Massachusetts Institute of Technology (MIT), 1999. Version 1.2 http://www.libpng.org/pub/png/spec/1.2/PNG-Contents.html.

[Str12] Tim Strazzere. Dex Education: Practicing Safe Dex. In *BlackHat USA*, July 2012. http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf.