Matryoshkryption - Write-up

Lena DAVID

March 4, 2018

1 Description

1.1 Challenge statement

1.1.1 First part

Wow! I heard about that conf from 31C3 on file formats tweaks and the result is pretty impressive.

Will you find what is hidden inside that matryioshka?

1.1.2 Second part

You need to solve the first part of the challenge first.

1.2 File provided to the challenger

`matryoshkryption.tgz`.

2 Write-up

This section describes an example of process that could lead to solving the challenge.

2.1 Starting up

The only provided file is a `.tgz` archive. Once decompressed, we get a single file, named `<file_shasum>.png`.

2.2 Handling the png file

An image viewer such as `feh` displays a photograph of a matryoshka.

Even though the picture itself does not seem particular in an way, some elements about the `png` file are untypical:

- running the `file` command on the `png` file should display informations relative to the picture (typically, something like: `<shasum>.png: PNG image data, 1400×1400, 8-bit/color RGB, non-interlaced`). This is definitely not the case here, since the output we get is: `<shasum>.png: data`.
- the file is 5.9MB large. This is huge, all the more so that the picture's dimensions are 1400×1400px, which is not particularly high. We could expect such a file to weigh a few hundreds of kylobytes, but not much more.

Ok, something is definitely hidden in that file (actually, we already knew that, otherwise the challenge would not have been categorized in steganography).

Opening the file with a text editor shows even more weirdness: the file starts as a standard png file (see below), but the first data chunk of the file is a "`aaaa`" chunk, which is strange for at least two reasons:

- the first data chunk of a png file should always be the `IHDR` chunk;
- `aaaa` is not an actual chunk type.

Some insights on the PNG file format
Understanding the PNG file format in detail is not necessary in order to solve this challenge, but some elements given below may help the reader get a better overview of how this step of the challenge has been built and on how to solve it.

- To be valid, a png file must start with `\x89\x50\x4e\x47\x0d\x0a\x1a\x0a`. These bytes are the "magic number" of the PNG format and makes files in that format identifiable as such.
- Data in png files are organized in "chunks", which are segments containing various data about the file. For instance, the `IHDR` chunk must be the first to appear in the file, and contains information such as the picture's width and height, as well as the bit depth, color type, compression method, filter method, and interlace method. This chunk is 13 bytes long. The `IDAT` chunk contains the actual data that makes the picture, and can be split into several chunks (all still called `IDAT`). The `IEND` chunk marks the end of the file.
- Manifold types of chunks exist, some of which are required in the file for it to be valid and some of which are optional. One can add custom chunks, which will most likely be ignored by the image viewer when the file is opened. Some chunks have a fixed size, some other a variable size with a fixed limit. The other are assumed to have a variable size and must weigh less than a limit which is hardcoded in the implementations of the standard (for example, `libpng` specifies that no chunk can weigh more than 8000000 bytes).

Without more information, it is a bit difficult to understand how to use what we have got so far. There must be something more somewhere in the file.
A common method to hide data into an image file - referred to as the *cover file* - consists in embedding the data to hide into the least significant bits of the picture - particularly if the format is lossless, which is the case for the `png` format. The basic idea is that for each of its pixels, for one or several color components (among red, green, blue), the last bit of the byte representing that component of that pixel is replaced with a bit of the data to conceal.
This method is called `LSB` (for **L**east **S**ignificant **B**it).
A variant consists in modifying the pixels of the picture so that the data is embedded visually: basically, in that context, the pixels of the cover file are modified so that isolating the plan made of the LSBs of one or several color components lets appear the hidden data. For matters of clarity, we will refer to this technique as *Visual LSB*.

Let's find out whether one method or the other has been used to hide any data into the image.
Even though we could write detection scripts to do so, a useful piece of software named `stegsolve` already exists to serve that purpose. Let's use it.

Trying to open the file with `stegsolve` directly, we encounter the following error message:

"`Failed to load file javax.imageio.IIOException: I/O error reading PNG header!`" That is probably due to the presence of the `aaaa` chunk in first position instead of the `IHDR` one. To solve that issue, a solution consists in opening the image in an image editor - such as GIMP - and to export it again to the png format.

Once that done, we can finally open the picture in `stegsolve`.

Let's start with visual LSB. Navigating between the differents plans of each component, we discover that the plan made of the LSBs of the green component holds the message "`key: <some_key>`" and that of the blue component the message "`iv: <some_iv>`". Both `<some_key>` and `<some_iv>` seem to be the hexadecimal representation of 16-byte long strings.

That will likely prove useful, although we do seemingly not have anything to encrypt or decrypt yet.

Let's have a look at the potential data in the "actual" LSBs of the picture (for instance using `Analyse > Data Extract` in `stegsolve`. Once again we find something interesting: more specifically, the plan made of the LSBs of the red component holds something that looks like a dictionary or a substitution alphabet.

In the end, LSB methods were indeed used to conceal data in that matryoshka picture. But nothing among what we have found so far provides an obvious hint on what to do next.

Even though we don't know what to do with what we have found, it may seem relevant to save the different elements - for example into files called `png_key`, `png_iv` and `alphabet`.

Since we have not got anything really more relevant to do, let's have a closer look at the challenge statement. It mentions a conference that was held at a venue called "31C3" and that addressed a topic related to file formats. Looking up "31C3 file formats" in a search engine, one of the first results links to the slides of Ange Albertini's conference "Funky File Formats". There are quite a lot of slides, but looking for the string "encryption" within them displays a single occurrence, related to a concept called "Angecryption". Looking for more information allows us to get a better understanding of it (see below).

Some insights on Angecryption

Angecryption was introduced by Ange Albertini in 2014. The core concept of it is to use the malleability one has got on the first plaintext block when using AES-128 CBC's decryption function by choosing the IV properly. More specifically, given a particular 16-bytes block of data considered as a ciphertext block, and given a chosen symmetric key, decrypting this ciphertext block with that symmetric key gives 16 bytes of data which is then xored with an IV to give the actual corresponding plaintext block. The IV can thus be chosen so that we get the value we want as plaintext block (IV =<output of AES_{DEC} > $\oplus$ <desired block>).

In particular, the IV can be chosen so that the first block of data looks like the beginning of a file of another type. Of course, this comes with some constraints as the specifications of the target file format must be respected for the resulting file to be valid. More details can be found at https://github.com/indrora/corkami

It thus appears very likely that what we have is some file that has been encrypted using angecryption with the key <some_key> and IV <some_iv> we found previously. Let's try to decrypt with the following script:

```python
#!/usr/bin/python

from Crypto.Cipher import AES
import argparse
import logging

# Logging
logger = logging.getLogger(__name__)
logging.basicConfig(level="DEBUG",
        format="%(asctime)s:%(name)s:%(lineno)s:%(levelname)s - %(message)s")


def solve(filename, outname, key, iv):
    f_in = open(filename, "rb")
    f_out = open(outname, "wb")
    key = open(key, "rb").read(16)
    iv = open(iv, "rb").read(16)

    logger.debug(key)
    logger.debug(iv)

    cipher = AES.new(key, AES.MODE_CBC, iv)

    indata = f_in.read()
    if len(indata) % 16 == 1:
        indata = indata[0:len(indata)-1]
    print(len(indata))
    out = cipher.decrypt(indata)
    f_out.write(out)


if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Decrypt angecrypted file\
            using provided key and iv")
    parser.add_argument("--input", "-i", required=True, help="File to de-angecrypt")
    parser.add_argument("--key", "-k", required=True, help="Key file")
    parser.add_argument("--iv", required=True, help="IV file")
    parser.add_argument("--output", "-o", required=True, help="Output file")

    args = parser.parse_args()
    solve(args.input, args.output, args.key, args.iv)
```

We run the following command: `python solve_angecryption.py --input <png_file> --key png_key --iv png_iv --output png_output`.

Running `file png_output`, we get the following output:
`png_output: PDF document, version .b`.

2.3 Handling the `pdf` file

Note: in that section and afterwards, the `pdf` file we just obtained will be refered to as `<pdf_file>`.

Once opened in a pdf viewer, the document displays a music score written in C clef. The musical rules seem to be followed; however, only quarter notes have been used, which is a bit unusual in the sense that the resulting song would probably be quite boring to listen to.

Anyways, listening to the resulting song is not really an option as we would have to generate the corresponding audio file, which would make it impossible for something particular to have been concealed in it.

Thus, what seems to be the thing to do is to decode the music score in terms of notes, which semmes even more likely that we have not used the alphabet we found earlier. This is quite a long task, which requires focus and patience, but is not too complex to achieve.

Once the music score decoded into a more textual form, we can put the alphabet in a way that makes it suitable for it to be handled by a script - for instance in the Python language. Afterwards, the only thing left to do is to use our substitution alphabet on our encoded message to get to something more understandable.

We thus obtain a message which gives us a new key/iv pair - which we can save in files named `pdf_key` and `pdf_iv`, as well as a flag, and a piece of advice stating that the pdf document might still be of use. **The flag we got validates the first part of the challenge**.

Having a closer look at the document, we notice that at the very end of it, some kind of copyright line is written. However, the second part of it is not intelligible, and its form suggests it is a base-64-encoded message. Once decoded - for example with `echo -en <b64_encoded_message> | base64 --decode`, we get a passphrase, that we can save as `passphrase`.

Since there is nothing obvious we can do with that passphrase, let's use the key/iv air we have found to get to the next step. The PDF document is likely another angecryption-encrypted file, so we use the same script as before with our new key/iv pair to try and find what is conceal in it:

```
python solve_angecryption.py --input pdf_file --key pdf_key --iv pdf_iv --output pdf_output.
```

Running `file pdf_output`, we get: `pdf_output: MPEG ADTS, layer III, v1, 128 kbps, 44.1 kHz, Monaural`.
The next file thus appears to be an audio file. After a quick search, we understand that its format is more or less equivalent to the `mp3` format.

2.4 Handling the mp3 file

Note: in that section and afterwards, the mp3 file we just obtained will be refered to as <mp3_file>.

Listening to the mp3 let us hear a morse-encoded message. At some point, we can also notice some additional noise.
Opening the file in Audacity makes it easier to decode the morse message since it provides us a way to visualize the file and thus to decode each character more conveniently. However, doing this requires the not-so-convenient step of understanding that our file is in the mp3 format (or something close) and renaming the file to something that holds the ".mp3" extension.
Decoding the morse message gives us a new key - we save it in a mp3_key file.

Having a look at the "noisy" part of the file does not directly reveal anything interesting. However, by switching to the Spectrogram view, we notice that something seems to be written there. After making the adequate adujstments to the view, we get a new IV, which we save to mp3_iv.

Once again, let's try to de-angecrypt our file with the new key/IV pair:

```
python solve_angecryption.py --input mp3_file --key mp3_key --iv mp3_iv --output mp3_output
```

.

Running `file mp3_output`, we get:

```
mp3_output: ISO Media, MP4 Base Media v1 [IS0 14496-12:2003]
```

.
The next file thus appears to be an mp4 file.

2.5 Handling the mp4 file

Note: in that section and afterwards, the mp4 file we just obtained will be refered to as <mp4_file>.

Watching the video informs us about something that occurred at the end of January, but does not provide us really useful information. Using tools such as binwalk does not help more, and looking at the bytes of the file by ourselves shows nothing useful either.

Since we can seemingly not use the file on its own, let's think for one second about elements that could help. We have used every auxiliary element we have found in the previous steps, except the string that looked like a potential passphrase that we got from the pdf document. That passphrase being the only unused element, it is very likely to be used in that step in some way. Nevertheless, we still do not know how to use it.

Let's have a closer look at it: we can note that there is something weird with the type case of it: it is mostly written in CamelCase but some characters are in uppercase even though they are not at the beginning of a word and some are in lowercase whereas they are at the beginning of a word. We can then notice that the part of the passphrase that has been altered seems to highlight the message "TrueCrypt". Searching for "TrueCrypt mp4" in a search engine displays that a TrueCrypt volume can easily be hidden in an mp4 file.

Trying to mount the mp4 file as a TrueCrypt volume proves vain, as the video is not recognized as a TrueCrypt volume. However, looking for more information about TrueCrypt makes us aware that TrueCrypt has been deprecated for a while, and that a common alternative to it is VeraCrypt.

Trying to mount the mp4 file as a VeraCrypt volume proves successful. Inside the volume, we find a single file named `flag`.
Its content validates the second part of the challenge.