Abusing unlimited photo storage services — Getting unlimited whatever-kind-of-file storage.

Mario Bartolomé Manovel
mario.bartolome@gmail.com

June 26, 2018

Abstract

This paper, aims to show how easy is to abuse some of the major unlimited photo storage services, in order to get free storage for whatever-kind-of-file we may want to store.

Disclaimer: This is just for academic purposes. None of the contents of this article were, and should not be, used for profit.

1 Introduction

- What's an image?

Well, it kinda depends on the image we're seeing. If it's an JPEG/JPG image it will look really different from a PNG or RAW image, hexadecimally speaking. But basically, an image is just a codification of n bits/pixel. Where $n = 24$ on JPG, making 8bits per channel (RGB), and $n = [30, 48]$ on RAW making 10 to 16bits per channel.

- So, how does a computer knows that a file is an image?

Instead of opening the file, and reading everything it contains, and then trying to correspond the bytes to something familiar, OSs usually rely on the heading or *file signature*. The *file signature* are just a bunch of magic numbers, that an OS uses to find out what kind of file it is opening[1]. On images, these numbers are also called *SOI* or Start of Image. Of course, every opera needs a closure, so at the end of the images there's an *End of Image*, also called *EOI*.

- And, that's it?

Believe it or not, sometimes that's it.

[1] To offer you how to open it, and some other fancy stuff.

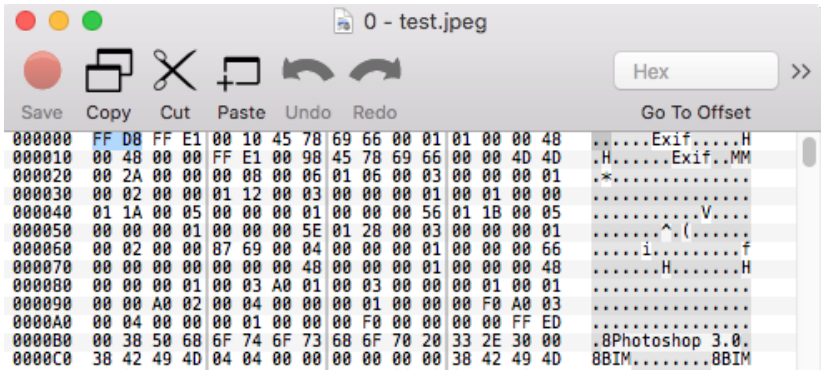

Figure 1: Hexadecimal SOI for a JPEG image.

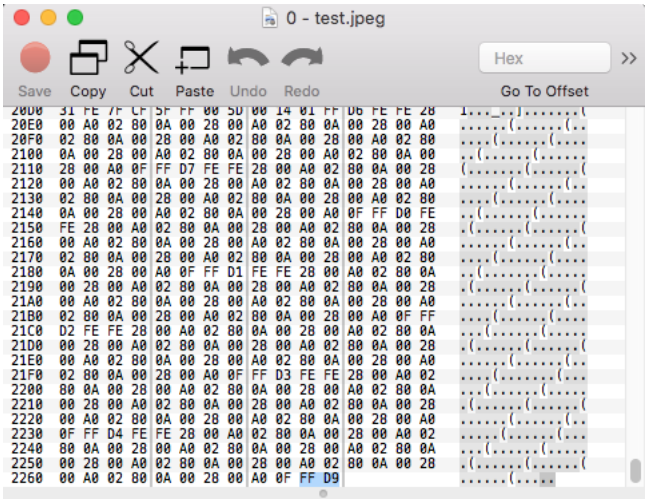

Figure 2: Hexadecimal EOI for a JPEG image.

2 Diving into the image

As you can see on Figure 1, the magic numbers, telling the OS this file is a JPEG image, *SOI*, correspond to `FF D8`. And the end bytes telling the OS the JPEG file is over are the ones highlighted on Figure 2, and corresponding to `FF D9`.

In fact, there's plenty of different bytes that, appended right after the *SOI*, help to identify a JPG/JPEG file. Some of the most common ones are listed below:

- `FF D8 FF E0 xx xx 4A 46 49 46 00`: JPEG/JFIF graphics file.
- `FF D8 FF E1 xx xx 45 78 69 66 00`: JPG file from a digital camera making use of EXIF[2].
- `FF D8 FF E8 xx xx 53 50 49 46 46 00`: JPG file on SPIFF[3].

EOI is always the same `FF D9`.

The funny part is, those `xx xx` bytes can be many different values, but for shake of simplicity we're going to use `00 10`, to identify our "image" as a JFIF-standard file. Also, the two bytes right after the *SOI*, can be used to identify the kind of camera that took the picture, for example: `FF D8 FF E2 ...` belongs to a Canon EOS/PowerShoot camera.

So what would happen if those bytes were written at the beginning of any kind of file? Will our OS *think* that the modified file is an image?

[2]Exchangeable Image File Format.
[3]Still Picture Interchange File Format.

3 POC ∨ GTFO

Well... let's give it a try!

I'm going to take just a simple file, let's say an mp3 file. And I'm going to append the *SOI* at the beginning of the file, and the *EOI* at the very end.

To make it even more clear, I will be appending a whole picture to the beginning of the mp3 file, and changing its extension. As you can see on Figure 3, there's an *EOI* but the file is not over (the next bunch of bytes, are the original mp3 file).

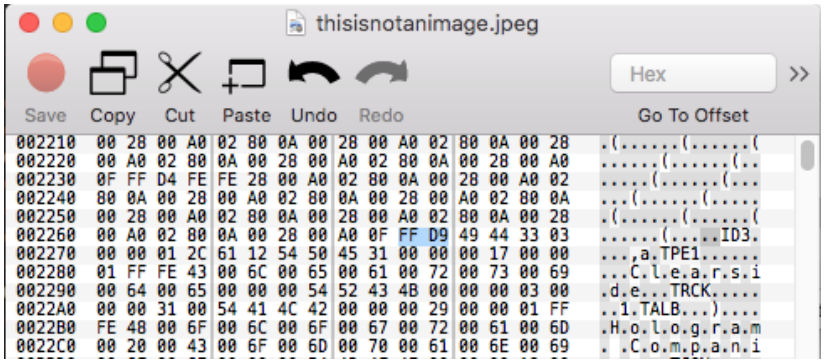

Figure 3: MP3 hex modified.

So what will the OS think about this file? Well... if you take a look at Figure 4, you will see how the OS seems to recognize the file as a JPEG file (well, it is, isn't it? but it's something else).

Figure 4: Not an img.

And... TADA! You're even able to display it, as you can see on Figure 5.

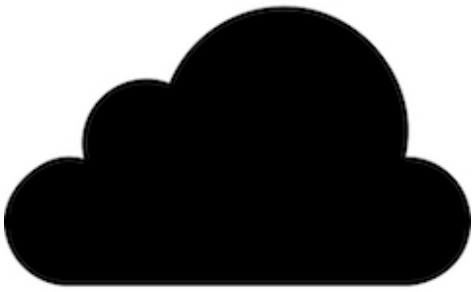

Figure 5: The file "thisisnotanimage.jpeg" fools even LaTeX!

4 Code

To simplify the job, even more, I wrote a simple snip of code that breaks any file into $2-3$MB pieces and creates a tiny database containing info about those fragments. It will append the image shown above (Figure 5), so it will be displayed on many services. The very same script is also able to reconstruct the original file from the pieces, making use of the database.

To recap:

- **Fragment to "images"**: pass the script the path of a big file (say 1.8GB) and run it. It will tear it down to pieces and generate a database (for that 1.8GB, the `.db` file weights around 34KB). Upload those pieces to your favorite free image storage service, and keep the `.db` file.
- **Reconstruction to original file**: download the pieces. Save them at the same folder as the `.db` file. Pass the script the path of the `.db` file and run it. It will reconstruct the original file.
- **Check original-reconstructed file integrity**: pass both files paths to the script. It will check they're equal.

The `code` is available at my repository on GitHub[4]. Fork it, like it and use it at your own discretion. Just remember, not a single free image storage service out there is going to like this. Not the affected ones at least...

[4] `https://github.com/MarioBartolome/AbuseImgStorage`