Web-based steganography tool

Deadline:

- ***04.06.2023***: HARD!

Changelog:

- *24.05.2023:* `/image/decode` & `/image/last/*` endpoint clarifications + final deadline!

Context

Steganography is the practice of hiding information within another medium (which is, usually, communicated in plain sight). If in encryption the difficulty lies in finding the secret (i.e.: the key) used to obfuscate the data, here the problem consists of detecting whether the data exists at all (the *plausible deniability* concept).

One steganog technique that is easy to understand consists of encoding messages into pixel data. As we all know, images are made out of pixels, and pixels are made out of three color channels: Red, Green, Blue. Usually, each channel is represented via an 8-bit value (0 - 255, or 0xFF). The higher the value, the more intense the color. Thus, it follows that altering the more significant bits of any channel will produce visible alterations: in the image below, we masked (i.e.: set to 0) the most significant bit of every channel, of every pixel.

But what happens if we play around with some of the less significant bits? Answer: no human will be able to tell the difference! As a result, one way to exfiltrate data is by splitting the message into bits and encoding them into the least significant bits of the image. If only the least significant bit is used, you will need 3px in order to encode 1 byte of data (with 1 bit to spare, or it could also be considered the next byte of our secret). For example:

```
# Note: using PIL representation, where each pi
pixels = [(0xff, 0x00, 0x04), (0xff, 0x19, 0x10
bin_message = [0, 1, 0, 0, 1, 0, 0, 1] # AS
# after the (color & 0xFE | msg_bit) bitwise op
enc_pixels = [(0xfe, 0x01, 0x04), (0xfe, 0x19,
```

The goal of this assignment is to write a simple Flask-based web application that helps you encode / decode a secret message into an existing image (uploaded by the user) using the steganography method described above. Furthermore, we also want the server containerized using Docker (together with its dependencies) such that, regardless of the machine, it can be easily started with minimal effort (especially for evaluation purposes!).

> This technique only works on lossless compression formats (`bmp`, `png`)!
>
> In contrast, lossy image formats (e.g., `jpeg`) may randomly alter the color data of the pixels, so the information concealed there will get corrupted. We will only consider the former case (no lossess)!

Specification

You must implement a Flask web server serving a basic User Interface with several (*when**, two) HTML forms for uploading images, plus specific backend routes for receiving the uploaded files, doing the actual steganography encoding / decoding processing and giving back the results.

In the following subsections, we define some minimal (required) aspects to be followed (especially to make the grading process easy to automate) + recommandations of the best approaches to consider (as notes / hints).

> We recommend to start this assignment by first writing small Python functions / modules and/or CLI script for running the steganography encoding / decoding algorithms.
>
> This will decouple the tasks (encoding / decoding vs web frontend), allowing you to partly validate the solution before continuing. OFC, bonus for using unit testing ;), although this is out of scope.

REST-ful Web API

A Web-based API (Application Programming Interface) is a contract between the provider of a service and the user wishing to make use of it. This usually consists of the format of the different URLs, specific parameters, HTTP methods they may be called with and request / response body formats. REpresentational State Transfer (REST) is a set of common principles / rules which makes such APIs consitent and easy to use.

Your Flask application must, too, adhere to such an API:

- `/`: serves the front HTML page;
- `/image/encode`: receives a "multipart/form-data" with the following fields (i.e., form names): `file` (the uploaded file, binary data) and `message` (the string message to embed into the image using steganography); responds with the generated image (as binary downloaded data); the encoded image should also be saved onto the server's disk for later retrieval requests;
- `/image/last/encoded`: retrieves (i.e., downloads / displays) the binary contents of the last encoded image (no arguments required);
- `/image/decode`: takes an encoded image and outputs the original steganography message (using the least significant bits technique); request should have a `multipart/form-data` content type and receive a `file` field with the stegano-encoded image, and output the decoded message (either as HTML page or as a simple `plain/text` output); it should also store the decoded text / binary data (if you wish to support it);
- `/image/last/decoded`: retrieves the last decoded plain text (or binary data, if you support this); do not output HTML here, just the raw data (used for test automation purposes).

Image formats: use any web-compatible lossless compression format (e.g., `png` is a safe choice; for advanced users: `webp` ;)).

As stated before, this is important for automating the grading process, so please respect it!

You may add any additional routes as required by your HTML+CSS-based UI (described below).

You may also add additional parameters (but they must be optional!) to the encode / decode endpoints if you wish to make the steganography technique customizable (e.g., use more significant bits or encode some redundancy, for bonuses ;)).

User Interface

The web frontend should present a (somewhat) friendly user interface with, at minimum, a simple front page (with a basic description) and the two upload form pages for steganography encoding / decoding.

All pages must have a common menu bar (hint: use a Jinja2 template!) directly linking to the three pages (index / encode / decode). We recommend the use of a CSS framework (e.g., Bootstrap) for easily adding vertical / horizontal menus to a HTML page.

The image upload forms should contain at least one file input, a textbox for the message to encode (for the encoding page) or a box to display the decoded image (for the decoding page). You should also show the last image uploaded to the server side-by-side with the form (e.g., as floating image; use the `/image/last/*` REST endpoints for this).

The design (aspect) of the web pages does not matter as long as it meets the requirements above and one is able to determine which link to press for accessing the required steganography encode / decode functions.

Containerization

In order for your web application to be easily deployable / shared (with us :P), you must add a Dockerfile installing all of its dependencies (use PIP requirements, ofc!). You may start from any base image, although we recommend `slim` due to its low disk footprint.

Thus, a containerized solution must work using the following steps:

- the `docker build -t iap-tema2 .` command should run successfully;
- `docker run -p 8080:80 -it iap-tema2` should start the Flask server and make it accessible on `http://localhost:8080`.

> Please follow the archiving conventions and have everything (especially the `Dockerfile`) inside its root directory!

Grading

The base assignment constitutes **4p** out of your final grade (10p homework + **4p** final grade). The 10p are split between the following tasks:

- **[40p] Stegano encode / decode script:** either working in console (via CLI scripts) or web-based (using Flask + HTML), as long as it works as intended!
- **[40p] Web UI (HTML Forms + Flask):** web-based frontend for uploading images and encoding / decoding secret messages using the described technique (Note: it must respect the given specification!);
- **[20p] Docker container:** write a (working) Dockerfile for easily building & running the server;
- **[up to 10p] Bonus ideas:**
  - A nice UX ;)
  - Implement both a CLI (using `argparse`) + Web frontend using a modular approach (code sharing!);
  - Extra steganography-related functionality (e.g., by adding additional form fields); e.g., add parameters for visualizing the data of specific color channels of an image using binary masking, use specific color channels / multiple bits for encoding the data etc.

Write a README (`.txt` / `.md`) containing a description of your implementation, design choices, any third party libraries used (e.g., PIL), challenges you encountered, etc. Feel free to add your feedback here as well.

The project's source code (i.e., no binary / generated files need to be included) must be archived (`.zip` please) and make sure the scripts (incl. Dockerfile) are placed directly in the root folder (i.e. depth 0) of the archive! Otherwise, the grading process will be slower ⇒ lower score :(

> **NOTE:** Assistants are free do deduct points for bad / illegible code!
>
> Also, please double-check if you followed all naming conventions!

Resources

- pillow is a image processing module that has support for many image formats and grants the developer access to the pixel data. You can install it using `pip3` (see our IDST labs for info regarding `pip` and virtual environments).
- argparse is a command line argument parser (useful if you want nice CLI scripts configurable with options).
- Flask web framework for Python.
- Jinja2 template engine (integrated with Flask).
- Docker container engine (Getting started tutorial).

FAQ

Q: Can I write the tool in something other than Python?
A: No. You have the Chip8 Bonus Assignment in C, if you want to be closer to the metal ;)

Q: What platform will this assignment be tested on?
A: Linux (though, you don't need to use any platform-specific APIs).

> **TODO:** Collect questions from Teams / lab and add them here.