

Project Report: Stego - A Web-Based Steganography Tool

Submitted by: MT2024120 Priyansh Agrahari
Project: Stego (Web Application)
Topic: Text Steganography (#7)
Date: 24 October 2025
Link: <https://github.com/r3dacted42/stego> (Repository)
<https://r3dacted42.github.io/stego/> (Deployment)

Contents

Development Process.....	3
Image Steganography (LSB).....	3
Handling Payloads.....	3
Text Steganography (Zero-Width Unicode).....	3
UI/UX Refinements.....	4
Technical Decisions.....	5
Key Learnings & Challenges.....	6
The Alpha Channel Pitfall.....	6
Bytes vs. Characters.....	6
Demonstration.....	7
Image Steganography.....	7
Encoding.....	7
Decoding.....	8
Text Steganography.....	10
Encoding.....	10
Decoding.....	10
Conclusion.....	12

Development Process

The development of the Stego application progressed iteratively, focusing initially on core steganography logic and later refining the user interface and handling edge cases. The goal was to create a tool demonstrating image and text steganography, performing processing client-side for privacy and speed. Vue 3 with TypeScript and Vite was chosen for the frontend framework. Web Workers were identified early on as crucial for handling potentially heavy image processing without blocking the main UI thread.

Image Steganography (LSB)

Development started with the image encoding/decoding logic using the Least Significant Bit (LSB) method. The initial implementation used a 4-bit-per-pixel approach (RGBA). Web Workers (`imgEncProc.worker.ts`, `imgDecProc.worker.ts`) were developed using `OffscreenCanvas` for efficient pixel data manipulation, and a message size calculation worker (`imgEncMsgSize.worker.ts`) was subsequently integrated.

Significant data corruption issues were encountered during decoding, which manifested as failing `fetch()` calls. This was traced back to the browser's premultiplied alpha handling when saving PNGs, which altered RGB values if the Alpha channel's LSB was modified. The LSB strategy was revised to use only 3 bits per pixel (RGB), completely skipping the Alpha channel. This resolved the corruption.

Handling Payloads

Functionality was added to allow both plain text and arbitrary files as payloads. Files were read using `FileReader` and converted to Base64 data URLs. To preserve the original filename for decoding, a non-standard `name=` parameter was injected into the Base64 string (e.g., `data:mime/type;name=filename.ext;base64,...`).

Helper functions, `getNamedB64` and `getUrlAndNameFromB64`, were developed to manage this enhanced Base64 format during both encoding and decoding processes.

Text Steganography (Zero-Width Unicode)

Various text steganography methods were explored (visual vs. linguistic), and the Zero-Width Character (ZW) method was selected as the most practical and web-friendly. An efficient quaternary encoding approach was implemented:

ZW Character	Encoding
Zero-width space (U+200B)	00
Zero-width joiner (U+200D)	01
Zero-width non-joiner (U+200C)	10
Zero-width no-break space (U+FEFF)	11

To ensure reliable decoding, a 4-byte length header, encoded as 16 Zero-Width (ZW) characters, was adapted from the image implementation. An interleaving strategy was implemented to distribute ZW characters throughout the carrier text for better stealth, rather than appending them all at the end. Encoding (`txtEncProc.worker.ts`) and decoding (`txtDecProc.worker.ts`) workers were created, with careful attention paid to matching the bit-order (LSB-first in this case) during byte reconstruction in the decoder.

UI/UX Refinements

Encoding and decoding interfaces (e.g., `ImageEncode.vue`, `ImageDecode.vue`) were developed, alongside an `ImageDropZone.vue` component for streamlined image file input. Key functionalities implemented include real-time UTF-8 based byte counting for textareas (not using `maxlength` on `<textarea>`), preview and download options for encoded files, and error handling via `<dialog>` elements for worker errors. Additionally, a comprehensive `README.md` and introductory text for the `Home.vue` component were created to elaborate on the methods and their limitations.

Technical Decisions

Stego leverages client-side processing for all steganographic operations, ensuring user privacy and eliminating server costs. To maintain a responsive UI, Web Workers are crucial for offloading CPU-intensive tasks like pixel manipulation and ZW character processing from the main thread, with OffscreenCanvas employed within image workers for efficient pixel access.

TypeScript enhances type safety throughout the project, particularly in defining message interfaces between the main thread and workers. For image steganography, LSB (Least Significant Bit) is used, specifically avoiding the Alpha channel. For text, ZW characters were chosen due to their invisibility and resilience to copy-pasting, with quaternary encoding and interleaving implemented for efficiency and stealth. Both methods utilize 4-byte length headers to ensure the decoder accurately reads the message.

File handling involves Base64 encoding to represent files as text strings, compatible with both steganography methods, and filenames are injected into the data URL for a better decoding experience. The UI is built with Vue 3 (Composition API) for a reactive and component-based structure, while holiday.css provides consistent semantic styling.

Key Learnings & Challenges

The Alpha Channel Pitfall

The most significant technical challenge encountered was understanding and resolving image data corruption, specifically caused by premultiplied alpha when modifying the Alpha channel's LSB, which underscored the importance of comprehending underlying browser and file format behaviors. A key learning from this experience is to avoid modifying the Alpha channel LSB for reliable PNG steganography.

Bytes vs. Characters

A crucial realization during development was the discrepancy between the textarea's character count, determined by `maxlength`, and the actual payload limit, which is measured in bytes due to the variable-length nature of UTF-8 encoding. This understanding was vital for ensuring the accuracy of the user interface. A key learning from this experience was the importance of employing `TextEncoder` for real-time byte calculation, which provides accurate feedback to the user regarding the actual data capacity.

Code robustness was improved by implementing typed communication between the main thread and web workers using TypeScript interfaces for `postMessage` and `onmessage`. A crucial aspect was the ensuring of the bit/byte reconstruction order in the decoder, which precisely matched the encoder's logic, specifically using LSB-first for text encoding. The practical limitations of these steganography methods, such as their sensitivity to image compression and ZW character stripping, were highlighted by the project, emphasizing their primary suitability for educational demonstration rather than truly secure communication.

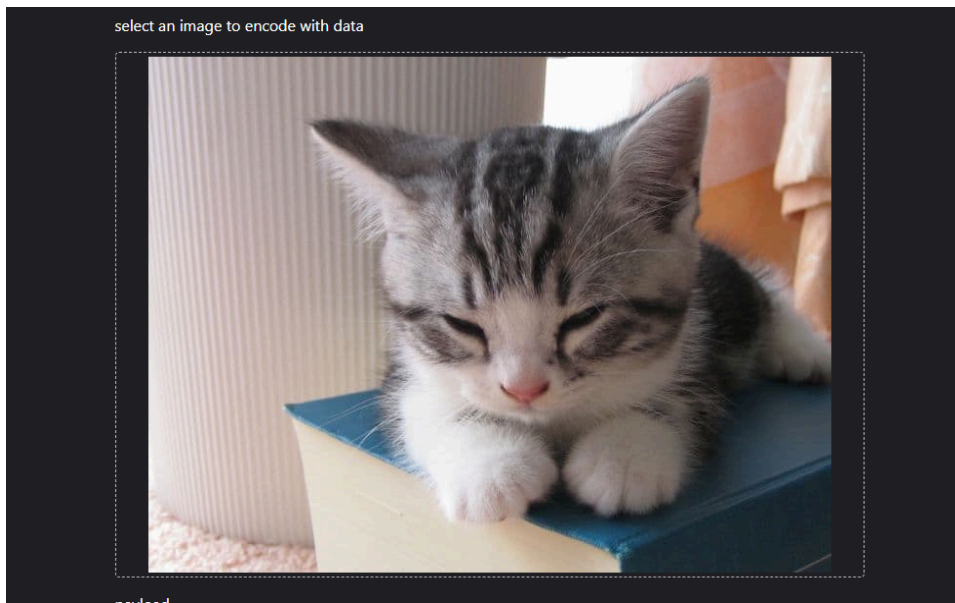
Demonstration

Image Steganography

Encoding

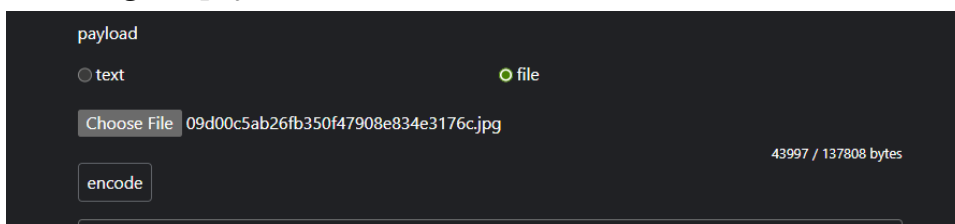
Available at <https://r3dacted42.github.io/stego/#/text/encode>.

1. Providing the carrier image:



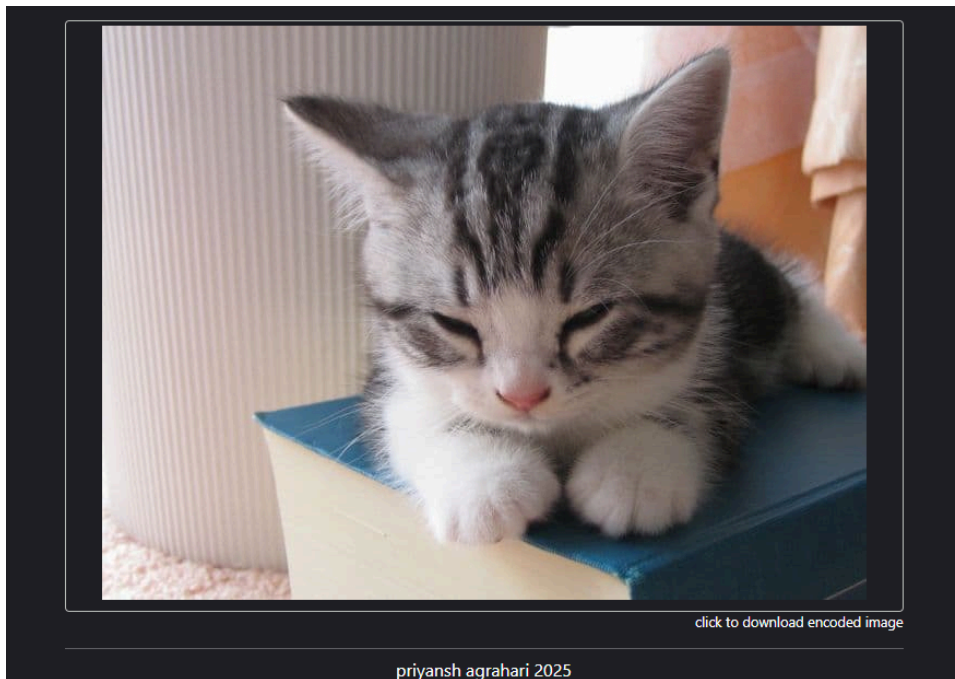
The image can be dropped onto the upload area, or can be selected using a modal by clicking on it.

2. Providing the payload data:



The payload can be a simple text message as well, but an image is used for demonstration.

3. Encoding the payload:

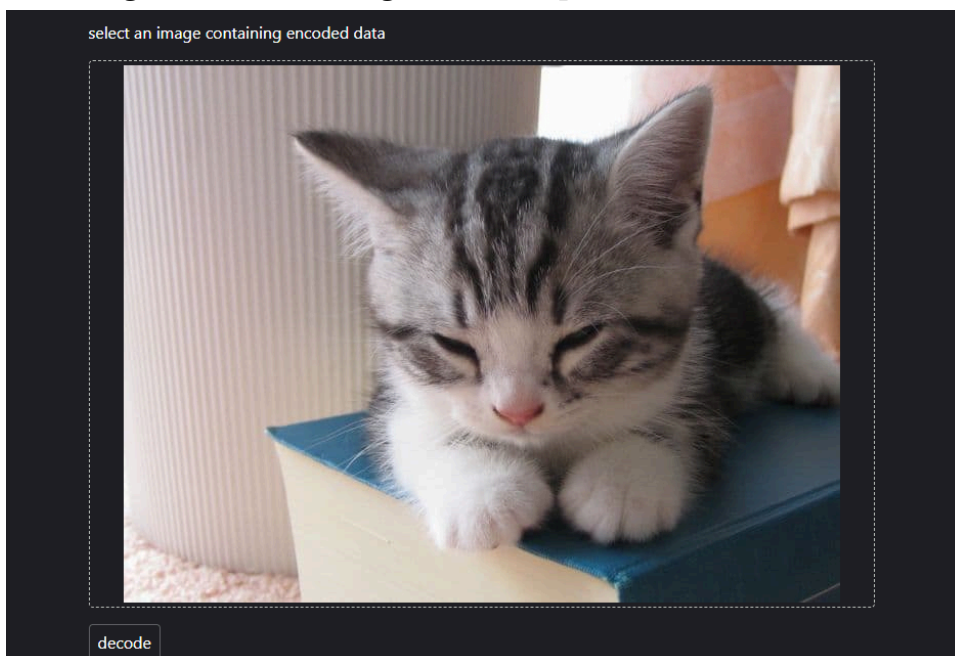


The image can now be saved to the disk by clicking on it.

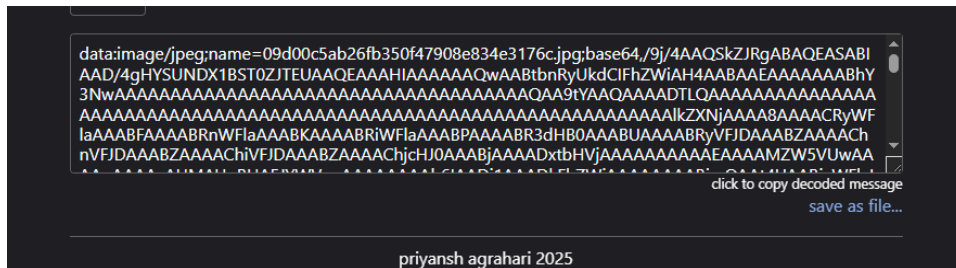
Decoding

Available at <https://r3dacted42.github.io/stego/#/text/decode>.

1. Providing the encoded image (from the previous section):



2. Decoding the message:



Clicking on “save as file...” downloads the JPEG file. It looks like this:

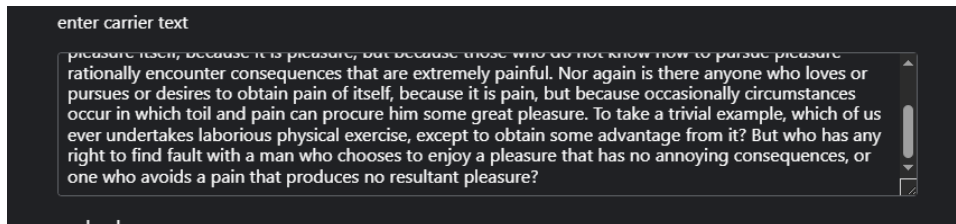


Text Steganography

Encoding

Available at <https://r3dacted42.github.io/stego/#/text/encode>.

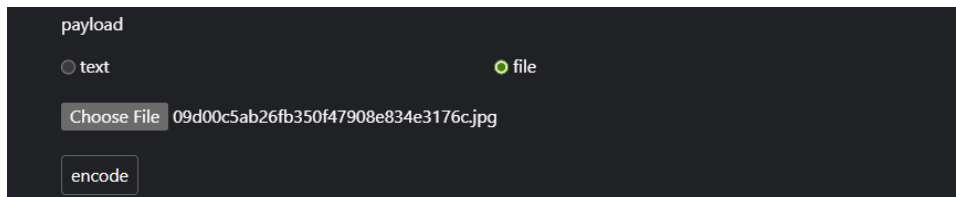
1. Providing the carrier text:



enter carrier text

pleasure itself, because it is pleasure, but because those who do not know how to pursue pleasure rationally encounter consequences that are extremely painful. Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure. To take a trivial example, which of us ever undertakes laborious physical exercise, except to obtain some advantage from it? But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure?

2. Providing the payload data:



payload

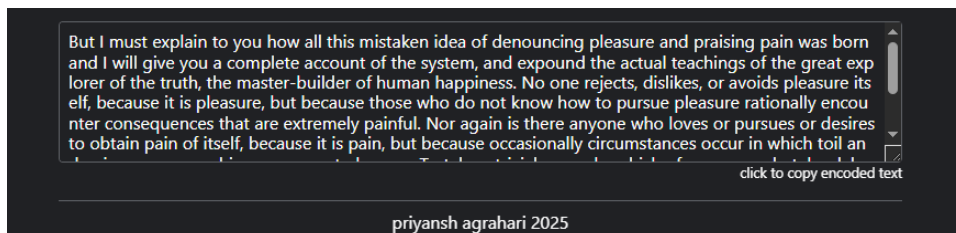
☐ text ☒ file

Choose File 09d00c5ab26fb350f47908e834e3176c.jpg

encode

The payload can be a simple text message as well, but an image is used for demonstration.

3. Encoding the payload:



But I must explain to you how all this mistaken idea of denouncing pleasure and praising pain was born and I will give you a complete account of the system, and expound the actual teachings of the great explorer of the truth, the master-builder of human happiness. No one rejects, dislikes, or avoids pleasure itself, because it is pleasure, but because those who do not know how to pursue pleasure rationally encounter consequences that are extremely painful. Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure. To take a trivial example, which of us ever undertakes laborious physical exercise, except to obtain some advantage from it? But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure?

click to copy encoded text

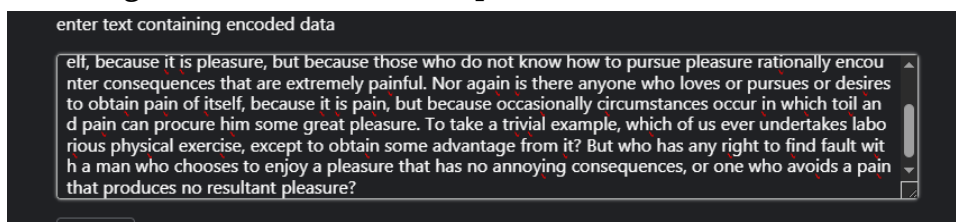
priyansh agrahari 2025

The encoded text can be copied to the clipboard by clicking on it.

Decoding

Available at <https://r3dacted42.github.io/stego/#/text/decode>.

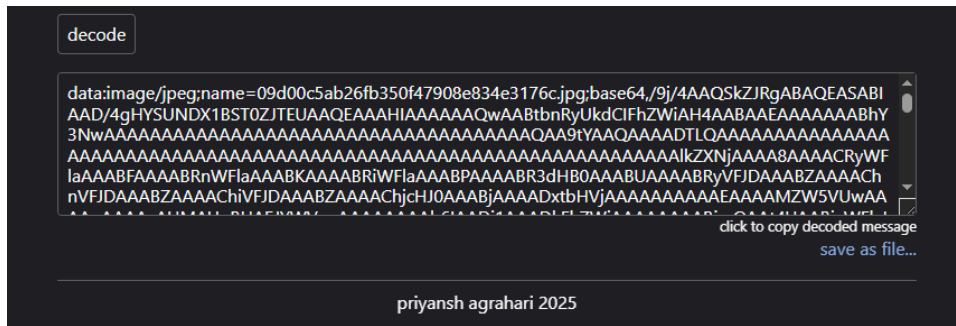
1. Providing the encoded text (from previous section):



enter text containing encoded data

elf, because it is pleasure, but because those who do not know how to pursue pleasure rationally encounter consequences that are extremely painful. Nor again is there anyone who loves or pursues or desires to obtain pain of itself, because it is pain, but because occasionally circumstances occur in which toil and pain can procure him some great pleasure. To take a trivial example, which of us ever undertakes laborious physical exercise, except to obtain some advantage from it? But who has any right to find fault with a man who chooses to enjoy a pleasure that has no annoying consequences, or one who avoids a pain that produces no resultant pleasure?

2. Decoding the message:



Clicking on “save as file...” downloads the JPEG file. It looks like this:



Conclusion

The final Stego application successfully implements both image and text steganography.

For image encoding, it accepts a carrier image (PNG recommended) and either a text message or a file, calculates the maximum capacity (3 bits/pixel minus header), encodes the payload (including filename) into the image's RGB LSBs, and provides a preview and download link for the resulting PNG. Image decoding then extracts LSB data, displaying the result as text or a downloadable file with its original name.

Text encoding uses interleaved Zero-Width characters (quaternary encoding) to embed a payload (text or file) into a carrier text, producing a visually identical output. Text decoding filters and decodes the ZW sequence to extract the message, presenting it as text or a downloadable file. The user experience is enhanced with clear interfaces, capacity indicators, file drop zones, previews, download links, and error messages.

Overall, the Stego project successfully met its objective, creating a functional web application that demonstrates LSB image and Zero-Width character text steganography, overcoming key technical challenges and providing valuable insights into their practical implementation and limitations, with Web Workers effectively maintaining UI responsiveness.

Thank you