CS 5732 Steganography

By: Miles Kuhn, David Helgeson, George Popov

Intro

This paper introduces a steganography tool designed to encode and decode messages within images. The tool utilizes basic encryption techniques to enhance message security. We discuss the background of steganography, provide an in-depth explanation of how the tool functions, explore potential detection methods for security engineers and offer instructions for running the tool.

Steganography

Steganography is the art of concealing information within other non-secret data to avoid detection. Unlike cryptography, which focuses on making the message unreadable, steganography aims to make the existence of the message inconspicuous. Covert channels, a subset of steganography, involve the communication of information in a way that violates security policies.

Description of our Steganography Tool

Our tool falls under the category of steganography, specifically hiding messages within image files using a combination of encryption and pixel manipulation. Images provide an excellent medium for steganography due to their complexity and capacity to hide information in plain sight. The tool comprises several functions, including loading an image, hashing the image, encrypting and decrypting messages, and encoding and decoding messages within the image pixels.

Here is a brief overview:

1. An image file, specified in the terminal arguments, is loaded into memory and converted into a format suitable for manipulation (a PNG file specifically as it is a lossless image file type and will preserve pixel values after compression).
2. The script then computes the SHA-256 hash of the image file, serving as the encryption key.
3. Next, the script uses the Advanced Encryption Standard (AES) in Electronic Codebook (ECB) mode for basic encryption and decryption of messages.
4. After encrypting the message using the hash of the original image, the size of the message is set and the script verifies if the image has enough capacity to encode the entire message.
5. Next, the message is encoded into the image. Four bits of information are coded into each pixel by encoding a bit into each of the RGBA values' least significant bit (using an XOR). The first 8 pixels (32 bits) encode the length of the encrypted message in bytes. The encrypted file is then read as a byte stream into the image encoded in the same way.

6. Decoding the image does the reverse of the processes above. First, the length of the message is extracted by examining the first 8 pixels. The pixels, up to the length (in terms of bytes so two pixels per byte) are read (XORed from the original image) and turned into a bit string which is turned into bytes and then decrypted using the hash of the original image. Finally, the bytes are written into the file format specified in the command line arguments.

Creating the steganography tool took some trial and error and reading lots of documentation of Python libraries and online forums (they answer questions faster sometimes). Overall, the tool works quite well and seems very robust with only the data capacity limitations in the image and the weak encryption method (used as a proof of concept).

Steganalysis

Steganalysis is the process of detecting the presence of hidden information within seemingly innocuous data, often employed in steganography. While our steganography tool aims to conceal messages within images, security engineers can employ various steganalysis techniques to identify the covert channel or detect an encoded message.

Security engineers may manually inspect the image for visible artifacts, changes in color patterns, or inconsistencies in pixel values. The images themselves may be given a visual inspection which involves closely examining the image for any anomalies or irregularities that might indicate the presence of hidden information.

For our Python script specifically, a security engineer might do a statistical analysis of the least significant bits of the image. Even though the message is encrypted (so it will look more randomly distributed throughout the image), pixels will have different values where they should have the same values (especially in areas of the image where the color should be the same). For example, the pixels of an image of a lake should be the same blue on a diagram but the encoded information would make them slightly different. This kind of statistical analysis has not been attempted. Still, it would be very easy to notice different color values (of a similarly colored part of the image) given an RGBA readout of the image. The encoded information is not visually perceptible and a computer would be required for analysis. Understanding the contents of the message relies on the hash of the original image. And if the original image is possessed it's easy to notice information encoded in the encoded image. The first 32 bits (first 8 pixels) store the length of the encrypted message in bytes. This length is unencrypted and always has 16 (because of the padding required for AES) as a factor which would be very suspicious when recognized across multiple images. Additionally, the file sizes of the encoded image are larger than the unencoded one since AES-encrypted messages are difficult to compress (compression is done automatically by the png file format).

<u>Instructions for running the Python Script</u>

- Ensure you have the required dependencies installed: PIL, pycryptodome, hashlib, and bitstring.
- Open a terminal or command prompt in the location of the Python script.
- To encode an image, execute the script using the command: python3 steganography.py -e [image_file] [message_file]. Please note that the image file is a PNG as the pixel values will stay the same when compressed (and the pixel values are how the data is encoded). The message file can be of any file type so long as its size can fit in the image (the script will not attempt to encode the image if there's too much information.
- To decode an image, execute the script using the command: python3 steganography.py -d [image_file] [encoded_image_file][output_file]. The output file should have the same extension as the file encoded into the image; it can be a txt tile in all cases if desired but then only the byte stream will be saved as text.