PNG LSB Steganography

50 Shades of RSA
Cameron Moberg, Brandon Ruoff, Devin Hudson

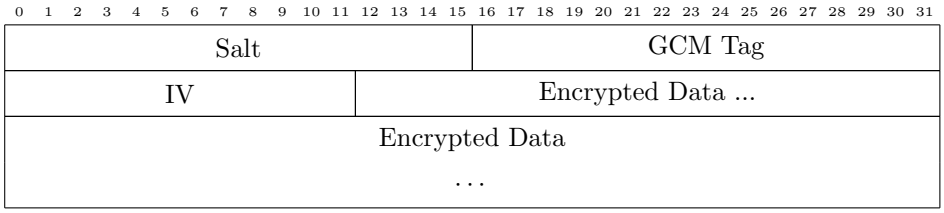
December 1, 2018

Encryption

The ability to encrypt and decrypt a desired file is not required per-se, but it is very helpful to have. Our specific file encryption algorithm is AES256-GCM. We chose Galois/Counter Mode (GCM) because of its performance, efficiency, and data integrity ability. GCM encryption can take advantage of parallel processing, which is where it gets the performance gain.

To encrypt, The user must pass in the file name to encrypt and the encryption key. Then we create salt from python's 'os.urandom' library, and derive a 32 byte key using the key, salt, and the PBKDF2 algorithm. It then writes to the file, in the following order 16 bytes of salt[1], 16 byte tag, 12 byte IV, and then the encrypted data.

Below is a visualization, in bytes, of the now-encrypted file.



The decryption is the reverse of this – the user passes in the encrypted file, and the encryption key. The file then reads the salt, the tag, the iv, and then the encrypted data. The key is then derived from the salt and provided key, and then attempts to decrypt the data and write to the output file.

Steganography

Steganography, from the Greek *steganos*, or "covered writing", is the hiding of secret data within seemingly ordinary other data, that will get extracted

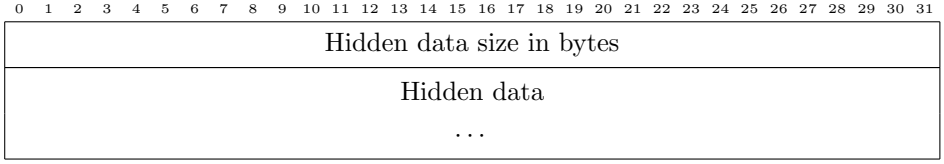
[1]Salt usually is not stored directly with the encrypted data, but in this case there was no other option

by someone who knows how it was concealed. This differs from encryption, since steganography alone shouldn't bring any attention to itself, and should go undetected.

So we're faced with a theoretical problem: we have to get secret data out of Germany in order to inform the allies of the Nazi's plans, however, the Nazi's are aware of any attempt to smuggle data and will not let any seemingly encrypted data out of the country. So we come up with a solution – to hide the data in "plain sight" in a `.png` file. Our specific implementation is a variation of Least Significant Bit (LSB) steganography where we hide the data in the least significant bit of each pixels red value. Unfortunately, just because a png file is lossless, doesn't mean it isn't compressed. In order to achieve this task we had to use the `libpng` C library to extract, edit, and write-back the pixels.

The specific encoding implementation is as follows: the user supplies the cover `png` file, a file with the data to be hidden, and the desired output `png` file name. The program will parse the `png` file, "hide" the length of the text in the first 32 pixels, and then hide the data bit by bit into the following pixels.

Below is a visualization, in bits, of the steganography file. This does not include the image bits that have no been modified, so in-between every bit is 7 bits of cover file data.



The decoding implementation is similar – the user passes the file with the concealed data and the desired output cipher file. The program then finds the length of the hidden data, extracts the data, and writes it byte by byte to the desired output file.

Issues

Since this implementation depends heavily on the size of the cover image size, we must consider that how large the hidden data can be. Since each pixel holds 1 bit of hidden data, and we have to use 32 pixels to store the size, the maximum size can be determined from this formula:

$$\frac{height * width + 32}{8}$$

Since we only use LSB in the red value, the image will at most have the red values ± 1 of the original. The following table shows the maximum error for an increasing number of bits used to hide data.

We could also store data in the Green, Blue, and Alpha values rather than just the red values, however, Alpha could be more easily detected – imagine if

Number bits used	Maximum error
1	±1
2	±3
3	±7
4	±15
5	±31
6	±63
7	±127
8	±255

we go from full opacity to just 99% opacity. This can also be said of the other bits, but it is much more common for color variations to be the case rather than the alpha not being all 100%. So we'll limit it to just RGB values. This opens up a lot more data storing capabilities, but the more we edit the file, the more steganalysis techniques will notice that something is different about this file and the higher chance we are found out.

Using the imagemagick's `compare` program, we call the command shown below.

```
compare <ORIGINAL.PNG> <STEG.PNG> -compose src <OUTPUT.PNG>
```

This allows us to see if there are different pixels between the original image and the hidden image. For example, below is what happens when we use RGB to hide our data instead of just red.

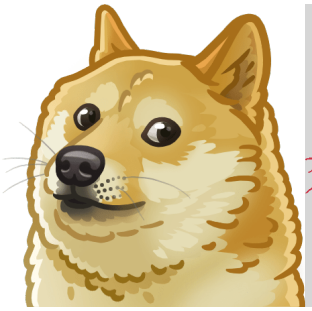


Figure 1: Original Image

Figure 2: RGB Diff File

Figure 3: Red Only Diff

Granted, it is still fairly easy to tell there is a difference, but not so obvious as to develop a pattern to discover it.

Now, if we take a look at using more bits to store our data we will see this:

Figure 4: Original Image

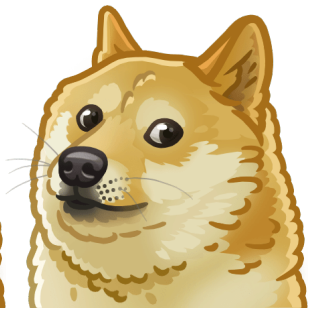


Figure 5: Least 1 bit(s)

Figure 6: Least 2 bit(s)

Figure 7: Least 3 bit(s)

Figure 8: Least 4 bit(s)

Figure 9: Least 5 bit(s)

Figure 10: Least 6 bit(s)

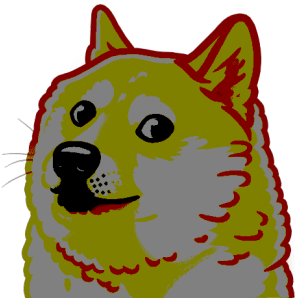


Figure 11: Least 7 bit(s)

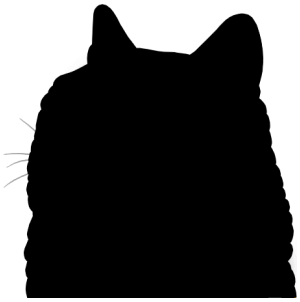


Figure 12: Least 8 bit(s)

Some very slight artifacts can be seen by using the least 3 bits, however, using the least 4, all the way up to all 8 bits show fairly large degradation of image quality. So in a way, although not perceptible to the human eye, image quality loss occurs after using more than just the bottom two bits.

Test Cases

Plaintext too large

```
$> identify doge1.png
doge1.png PNG 453x452 453x452+0+0 8-bit sRGB 131858B 0.010u 0:00.000
$> ls -l f.txt
-rw-r--r-- 1 bold bold 25592 Dec 1 16:16 f.txt
$> ./lsb_steg.out -e doge1.png f.txt output.png
Plaintext too large for given PNG.
```

So we see, given a file that is 453×452, and plaintext that contains $25,592$ bytes we should be able to store

$$\frac{453 \times 452 - 32}{8} = 25,590.5 \text{ bytes}$$

Since it is 2 bytes over our storage limit, we exit with an error condition.

Plaintext just right

So with a plaintext file that fits into the cover image, we get this output.

```
$> identify doge1.png
doge1.png PNG 453x452 453x452+0+0 8-bit sRGB 131858B 0.010u 0:00.000
$> ls -l f.txt
-rw-r--r-- 1 bold bold 25590 Dec 1 16:16 f.txt
$> ./lsb_steg.out -e doge1.png f.txt output.png
Successfully hid f.txt contents to output.png.
$> ./lsb_steg.out -d output.png output.txt
Successfully extracted hidden output.png contents to output.txt.
$> diff f.txt output.txt
$> echo $?
0
$>
```

No plaintext

Given no plaintext to hide, it still executes properly.

```
$> identify doge1.png
doge1.png PNG 453x452 453x452+0+0 8-bit sRGB 131858B 0.010u 0:00.000
$> ls -l f.txt
-rw-r--r-- 1 bold bold 0 Dec 1 16:16 f.txt
$> ./lsb_steg.out -e doge1.png f.txt output.png
Successfully hid f.txt contents to output.png.
$> ./lsb_steg.out -d output.png output.txt
Successfully extracted hidden output.png contents to output.txt.
$> diff f.txt output.txt
```

```
$> echo $?
0
$>
```

Responsibilities

- Devin Hudson
    1. File Transfer Code
    2. Transfer Testing
    3. Steg Testing
- Cameron Moberg
    1. Write-up
    2. Steganography Code
    3. PNG research
    4. Steg Testing
- Brandon Ruoff
    1. Encryption Code
    2. Encryption research
    3. Steg Testing

```
import argparse
import os
import sys

/*
* Python File Encryption and Decryption
* Cameron Moberg, Brandon Ruoff, Devin Hudson
*/
from cryptography.exceptions import InvalidTag
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

SALT_LEN = 16
TAG_LEN = 16
IV_LEN = 12


def aes_decrypt_file(f_name, key, mode=modes.GCM):
    """
    Decrypts a given file's contents from the given key.

    It reads the salt, IV, and GCM tag from the file, and then attempts to decipher the
    cipher text

    :param f_name: File who's contents will be decrypted
    :param key: Key to decrypt the contents with
    :param mode: AES mode, defaults to GCM
    :return: The original plain text if decryption is correct.
    """

    with open(f_name, 'rb') as f:
        text = f.read()

        salt = text[0:SALT_LEN]
        tag = text[SALT_LEN:SALT_LEN + TAG_LEN]
        iv = text[SALT_LEN + TAG_LEN:SALT_LEN + IV_LEN + TAG_LEN]
        cipher_text = text[SALT_LEN + IV_LEN + TAG_LEN:]

        key = derive_key(key.encode('utf-8'), salt)
        cipher = Cipher(algorithm=algorithms.AES(key),
                        mode=mode(iv),
                        backend=default_backend())
        decryptor = cipher.decryptor()

        try:
```

```
            return decryptor.update(cipher_text) + decryptor.finalize_with_tag(tag)
        except InvalidTag:
            raise Exception("Unable to decrypt text.")


def aes_encrypt_file(f_name, key, iv=os.urandom(IV_LEN), mode=modes.GCM):
    """
    Encrypts the contents of a file, and saves it to "f_name.aes"

    The file is laid out with the first 16 bytes as salt, next 12 as IV, and next 16 as GCM tag,
    with the remaining bytes the cipher text to decrypt.

    :param f_name: File to read and encrypt
    :param key: The key to encrypt the file with
    :param iv: The initialization vector, if not supplied, is a 12 byte random number,
        12 bytes has been shown to be the best if its random, since it doesn't require
        additional computations to encrypt it, but is still computationally secure.
    :param mode: The encryption mode, defaults to GCM, the method only uses AES to create
        cipher text
    """
    salt = os.urandom(SALT_LEN)
    key = derive_key(key.encode('utf-8'), salt)

    encryptor = Cipher(algorithm=algorithms.AES(key),
                       mode=mode(iv),
                       backend=default_backend()).encryptor()

    with open(f_name, 'rb') as f:
        f_text = f.read()
        cipher_text = encryptor.update(f_text) + encryptor.finalize()
        with open(f_name + '.aes', 'wb') as o:
            # salt 16 bytes
            # tag 16 bytes
            # iv 12 bytes
            o.write(salt)
            o.write(encryptor.tag)
            o.write(iv)
            o.write(cipher_text)


def derive_key(key, salt):
    """
    Given a key and a salt, derives a cryptographically secure key to be used in
    following computations. This is to allow any size key as input to the program, as
    we can extend it to the required multiple of 16,24,32 that AES requires

    :return:  Derived key from python's cryptography library
    """
```

```
    backend = default_backend()
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=2 ** 20,
        backend=backend
    )
    return kdf.derive(key)


def main():
    args = parse_args(sys.argv[1:])

    if args.encrypt:
        aes_encrypt_file(f_name=args.input,
                         key=args.key)
    else:
        with open("decrypted", "wb") as f:
            f.write(aes_decrypt_file(f_name=args.input,
                                     key=args.key))


def parse_args(args):
  parser = argparse.ArgumentParser(description='Encrypt a file with AES encryption.')
  group = parser.add_mutually_exclusive_group(required=True)
  group.add_argument('-e', '--encrypt', help='Flag that we encrypt the file.', action='store_true')
  group.add_argument('-d', '--decrypt', help='Flag decrypt the file.', action='store_true')
  parser.add_argument('-k', '--key', help='The key to encrypt the file with', required=True)
  parser.add_argument('-i', '--input', help='The data file you want hidden', required=True)
  return parser.parse_args(args)


if __name__ == '__main__':
    main()
```

```
/*
* Python File Transfer Utility
* Cameron Moberg, Brandon Ruoff, Devin Hudson
*/
import argparse
import socket
import sys
import threading

CHUNK_SIZE = 1024


class Client(object):
    def __init__(self, port, host):
        self.port = port
        self.host = host

    def send_file(self, f_name):
        sock = socket.socket()
        sock.connect((self.host, self.port))
        with open(f_name, 'rb') as f:
            while True:
                data = f.read(CHUNK_SIZE)
                if not data:
                    break
                sock.send(data)


class Server(object):
    def __init__(self, port=0, host='0.0.0.0'):
        self.host = host
        self.port = port

    def handle_conn(self, conn, addr, f_name):
        print("Connection from: {}".format(addr))
        with open(f_name, 'wb') as f:
            while True:
                data = conn.recv(CHUNK_SIZE)
                if not data:
                    break
                f.write(data)
        print("File written: {}".format(f_name))
        conn.close()

    def listen(self, f_name):
        s = socket.socket()
        s.bind((self.host, self.port))
```

```
48
49          print("Server started on port: {}".format(s.getsockname()[1]))
50
51          s.listen()
52          while True:
53              conn, addr = s.accept()
54              listener = threading.Thread(target=self.handle_conn, args=(conn, addr, f_name))
55              listener.start()
56
57
58  def main():
59      args = parse_args(sys.argv[1:])
60
61      if args.server:
62          Server(args.port, args.host).listen(args.file)
63      else:
64          Client(args.port, args.host).send_file(args.file)
65
66
67  def parse_args(args):
68    parser = argparse.ArgumentParser(description='Encrypt a file with AES encryption.')
69    group = parser.add_mutually_exclusive_group(required=True)
70    group.add_argument('-s', '--server', help='Flag that we encrypt the file.', action='store_true')
71    group.add_argument('-c', '--client', help='Flag to send file.', action='store_true')
72    parser.add_argument('-ho', '--host', help='IP of listener', default='0.0.0.0')
73    parser.add_argument('-p', '--port', help='The port that the server is listening', type=int, default=0)
74    parser.add_argument('-f', '--file',
75                        help='If in client mode, sends the specified file. In server mode saves contents'
76                             'to that file', required=True)
77    return parser.parse_args(args)
78
79
80  if __name__ == '__main__':
81      main()
```

```c
/*
 * Cameron Moberg, Devin Hudson, Brandon Ruoff
 * read_png_file and write_png_file logic from GitHub user @niw
 */
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <png.h>

#define SIZE 32
#define SBYTE 8

int width, height;
png_byte color_type, bit_depth;
png_bytep * row_pointers;

long file_size_bytes(FILE * fp) {
    /*
     * Function: file_size_bytes
     * --------------------
     * Determines the number of bytes that a file contains.
     *
     *  fp: file to determine size of
     *
     *  returns: File size in bytes
     */
    fseek(fp, 0L, SEEK_END);
    long sz = ftell(fp);
    rewind(fp);
    return sz;
}

long bstr_to_dec(const char * str, int len) {
    /*
     * Function: bstr_to_dec
     * --------------------
     * Converts a char array of 1s and 0s (binary) to decimal
     *
     *  str: pointer to string to convert
     *  len: length of the string to convert
     *
     *  returns: long of decimal representation of binary str
     */
    long val = 0;

    for (int i = 0; i < len; i++)
```

```
        if (str[i] == 1)
            val = val + pow(2, len - 1 - i);

    return val;
}

void write_size_to_px(long size, png_bytep * row_pointers) {
    /*
     * Function: write_size_to_px
     * --------------------
     *  Given the row pointers, will write a long, 4 bytes, to pixels 1 bit per pixel.
     */
    unsigned char parsed_size[SIZE];
    for (int i = 0; i < SIZE; i++) /* Iterate 32 times for the 4 bytes per long */ {
        png_bytep px = & (row_pointers[i / width][i * 4]);
        px[0] = (px[0] & ~1) | size >> 31 - i;
    }
}
long extract_size_of_cipher() {
    /*
     * Function: extract_size_of_cipher
     * --------------------
     *  After parsing a png file, reads the first 32 pixels and
     *  parses it into a long, so 0 0 0 0 0 1 1 would be returned as a long 3.
     *
     *  returns: Cipher size as a long
     */
    unsigned char parsed_size[SIZE];
    for (int i = 0; i < SIZE; i++) { /* Iterate 32 times for the 4 bytes per long */
        png_bytep px = & (row_pointers[i / width][i * 4]);
        parsed_size[i] = (char)(px[0] & 1);
    }
    return bstr_to_dec(parsed_size, SIZE);
}

void read_png_file(char * fn) {
    /*
     * Function: read_png_file
     * --------------------
     *  Given a png's file name, opens that png and converts it, if need be,
     *  and then parses it into the global variables above.
     *
     *  returns: Nothing, but updates global variables.
     */
    FILE * fp = fopen(fn, "rb");

    png_structp png = png_create_read_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);
    if (!png || !fp)
```

```
96          exit(1);
97
98      png_infop info = png_create_info_struct(png);
99      if (!info)
100         exit(1);
101
102     if (setjmp(png_jmpbuf(png))) abort();
103
104     png_init_io(png, fp);
105     png_read_info(png, info);
106
107     width = png_get_image_width(png, info);
108     height = png_get_image_height(png, info);
109     color_type = png_get_color_type(png, info);
110     bit_depth = png_get_bit_depth(png, info);
111
112     // Read any color_type into 8bit depth, RGBA format.
113     // See http://www.libpng.org/pub/png/libpng-manual.txt
114
115     if (bit_depth == 16)
116         png_set_strip_16(png);
117
118     if (color_type == PNG_COLOR_TYPE_PALETTE)
119         png_set_palette_to_rgb(png);
120
121     // PNG_COLOR_TYPE_GRAY_ALPHA is always 8 or 16bit depth.
122     if (color_type == PNG_COLOR_TYPE_GRAY && bit_depth < 8)
123         png_set_expand_gray_1_2_4_to_8(png);
124
125     if (png_get_valid(png, info, PNG_INFO_tRNS))
126         png_set_tRNS_to_alpha(png);
127
128     // The included types are RGB, Grayscale or Palette
129     if (color_type < PNG_COLOR_TYPE_GRAY_ALPHA)
130         png_set_filler(png, 255, PNG_FILLER_AFTER);
131
132     if (color_type == PNG_COLOR_TYPE_GRAY || color_type == PNG_COLOR_TYPE_GRAY_ALPHA)
133         png_set_gray_to_rgb(png);
134
135     png_read_update_info(png, info);
136
137     row_pointers = (png_bytep * ) malloc(sizeof(png_bytep) * height);
138     for (int y = 0; y < height; y++) {
139         row_pointers[y] = (png_byte * ) malloc(png_get_rowbytes(png, info));
140     }
141
142     png_read_image(png, row_pointers);
143
```

```
    fclose(fp);
}

void write_png_file(char * filename) {
    /*
     * Function: write_png_file
     * --------------------
     *  Given a png's file name, opens that png and parses and writes the global
     * variables into a valid png file form and writes it.
     *
     *  returns: Nothing, but updates global vars to file.
     */
    FILE * fp = fopen(filename, "wb");
    png_structp png = png_create_write_struct(PNG_LIBPNG_VER_STRING, NULL, NULL, NULL);

    if (!fp || !png)
        exit(1);

    png_infop info = png_create_info_struct(png);

    if (!info || setjmp(png_jmpbuf(png)))
        exit(1);

    png_init_io(png, fp);

    // Output is 8bit depth, RGBA format.
    png_set_IHDR(
        png,
        info,
        width, height,
        8,
        PNG_COLOR_TYPE_RGBA,
        PNG_INTERLACE_NONE,
        PNG_COMPRESSION_TYPE_DEFAULT,
        PNG_FILTER_TYPE_DEFAULT
    );

    png_write_info(png, info);
    png_write_image(png, row_pointers);
    png_write_end(png, NULL);

    for (int y = 0; y < height; y++)
        free(row_pointers[y]);

    free(row_pointers);

    fclose(fp);
}
```

```c


void add_steg_png(char * plaintext_fn) {
    /* Hides the given plaintext file into the global variables, which will then be
     * written to file later.
     */
    FILE * fp = fopen(plaintext_fn, "rb");
    int byte_r = 0;

    unsigned char buffer[256];

    if (file_size_bytes(fp) > (height * width - 32) / SBYTE)
    {
        printf("Plaintext too large for given PNG.\n");
        exit(1);
    }
    write_size_to_px(file_size_bytes(fp), row_pointers);

    // Set init_i to SIZE since we just wrote the file size.
    int offset = SIZE;
    long png_idx = 32;

    while (byte_r = (fread(buffer, sizeof(unsigned char), 256, fp))) {
        int byte_counter, byte_index = 0;
        for (int i = offset; i < offset + (byte_r * SBYTE); i++) {
            png_bytep row = row_pointers[png_idx / width];
            png_bytep px = & (row[(png_idx % width) * 4]);
            unsigned char c = buffer[byte_index];
            // Replace last bit of px[0] with current bit of character
            px[0] = (px[0] & ~1) | (c >> SBYTE - 1 - (i % 8));

            // One character finished
            if (byte_counter++ % SBYTE == SBYTE - 1) {
                byte_index++;
            }
            png_idx++;
        }
        offset = 0;
    }
}

void extract_steg_from_png(char * plaintext_fn) {
  /* Extracts the hidden steganography from parsed file
   * then writes it to the provided file.
   */
  FILE * fp = fopen(plaintext_fn, "wb");

  long cipher_size = extract_size_of_cipher();
```

```
  unsigned char parsed_byte[SBYTE];
  for (int i = SIZE; i < SIZE + (cipher_size * SBYTE); i++) {
      png_bytep row = row_pointers[i / width];
      png_bytep px = & (row[(i % width) * 4]);
      parsed_byte[i % SBYTE] = px[0] & 1;

      if (i % SBYTE == SBYTE - 1) {
          char conv_byte = (char) bstr_to_dec(parsed_byte, SBYTE);
          fwrite( & conv_byte, sizeof(unsigned char), 1, fp);
      }
  }
  fclose(fp);
}

int main(int argc, char * argv[]) {
  // Encoding is executed like:
  // ./a.out -e <COVER.PNG> <CIPHER.TXT> <OUTPUT.PNG>
  if (strcmp(argv[1], "-e") == 0) {
      // Cover png
      read_png_file(argv[2]);
      // Cipher text
      add_steg_png(argv[3]);
      // Steganography PNG
      write_png_file(argv[4]);
  }
  // Decoding is executed like:
  // ./a.out -d <STEG.PNG> <OUTPUT.TXT>
  else if (strcmp(argv[1], "-d") == 0) {
      // Steganography PNG
      read_png_file(argv[2]);
      // Cipher text output file
      extract_steg_from_png(argv[3]);
  } else
      printf("Arguments not understood.");

  return 0;

```