Steganography Chat Application Report

May 2, 2019

Prepared and submitted by:

Alaa Megahed	34-15309	T11
Hoda Hisham	34-03892	T15
Laila Elmalatawy	34-01810	T11
Mariem Kandil	34-11522	T11
Mona Hashem	34-00665	T11
Salma El-Ansary	34-02149	T11

1.0 Introduction

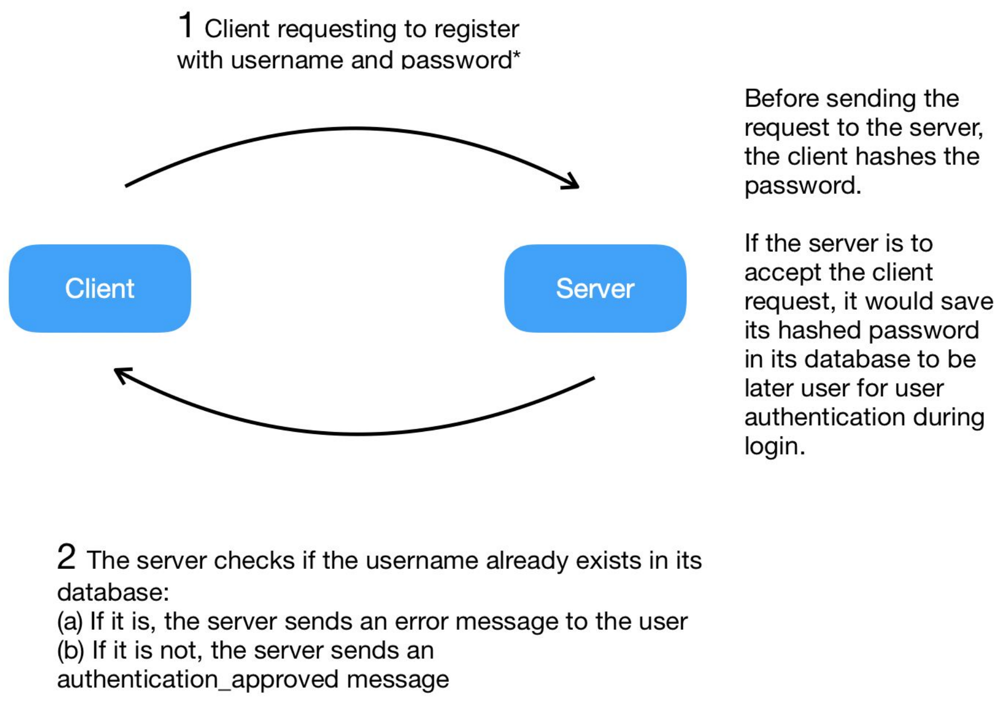
The chat application utilizes Steganography by hiding the content of messages in images. The application is composed of a lobby chat room where all users can send and receive broadcast messages. In addition, a user may choose any other online user and start a private chat with them. The application is secured using different security elements as Authentication and Hashing, Encryption (AES) and Key Exchange (Diffie–Hellman) and Access control through private chat channels.

2.0 Authentication

Passwords are used for user authentication. For a user to join the chat room, the user must first send a sign up request for the server with their chosen username and password (if they don't already have an account) or a login request (if they already have an account).
If the authentication request is approved, the user is then logged in and can start sending messages. The implemented approach of Authentication ensures Access Control by allowing only authenticated users to join the public chat or engage in any private ones.
Signup:

- For a user to sign up, they have to choose two things; a username and a password. There are some constraints, however, on their choice, for example, the username cannot include spaces, and the password has to be at least 8 characters long.

- The password is sent to the server hashed [7] with a randomly generated salt using SHA512 provided by python's hashlib library. The hashing adds an extra layer of security to block any password disclosure attempts made by unauthorized entities for example, man-in-the-middle, or unauthorized access to the database. The added salt ensures that if two users entered the same password, they would be hashed differently, and therefore preventing any kind of dictionary attack or popular password attack.
- The server checks if the username already exists in its database. If so, the server sends back an error message to the user prompting them to choose a different username. If not, the server inserts the user's credentials into the database, sends to the user an approval message, and adds the user to the list of current active users


1 Client requesting to register with username and password*
Before sending the request to the server, the client hashes the password.
If the server is to accept the client request, it would save its hashed password in its database to be later user for user authentication during login.
Client
Server
2 The server checks if the username already exists in its database:
(a) If it is, the server sends an error message to the user
(b) If it is not, the server sends an authentication_approved message

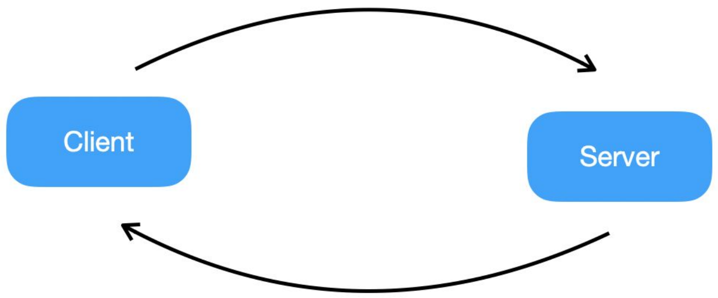
Sign up scenario

Login:

The users’ entered usernames and passwords are processed in 4 phases in order to grant them access to the chat room. First off, a message containing the username is sent to the server, the server then checks if the username already exists in the database. If so, it extracts the salt from the corresponding hashed password and sends it back to the client. The client uses the salt to hash the provided password and sends it back to the server, which by turn, compares the incoming hashed password with the stored one. In case of a match, the server sends an approval to the user and adds the user to the currently active users list. If an error is encountered at any of the phases, the server sends a descriptive error message back to the user.

1 Client requests to login using username

3 Client hashes the received salt and password and sends another request to the server



2 Server checks username:
(a) If found, the server extracts the salt from the hashed password and send it to client
(b) If not found, the server sends and error message

4 Server tries to verify the received hashed password with the stored hashed password for this username:
(a) If found, the server sends an authentication_approved message
(b) If not found, the server sends an error message

login scenario

3.0 Encryption:

Encryption is used to add an extra layer of security to block any attempts made by unauthorised entities to access the content of messages exchanged between the users and the server, for example man-in-the-middle attacks.
We use symmetric-key, block encryption algorithm, namely AES. We prefer it over public key encryption since it is simpler, faster, less expensive, yet it is efficient, secure and resistant against known attacks as long as the shared symmetric key is kept secret. As for exchanging the symmetric key, we use Diffie-Hellman algorithm, which is a practical method for public exchange of a secret key.

3.1 Key Exchange:

Each user has a shared symmetric key with the server, generated using Diffie-Hellman algorithm.
Given a shared prime number p and a shared base a, which is a primitive root modulo p, each entity generates a pair of public/private keys. The private key X_A is a 16-bit randomly generated number. The public key Y_A is calculated as follows: $Y_A = a^{XA} \bmod p$.
The public key is exchanged so that each entity can calculate the shared symmetric key K_{AB} given their secret key and the other entity's public key.
$K_{AB} = a^{XA*XB} \bmod p = Y_A^{XB} \bmod p = Y_b^{XA} \bmod p$.
The resulting shared key, however, is just a number and we need it to be a string of length 32 bytes, therefore we hash it using **SHA256** to produce a fixed-length string that can be used as a symmetric key for the AES.

3.2 AES:

We used AES algorithm imported from the Crypto library[6]. It has a fixed data block size of 16 bytes, so we add padding characters at the end of the message before encryption so that its total length would be a multiple of 16 bytes, then remove them after decryption to obtain the original message. As previously mentioned, we use a 32-byte long keys. The encryption is done in the default ECB mode.

4.0 Server-Client protocol:

Any data sent or received between the client and the server is folded into a JSON object that is made up of three fields; code, user and message.

Code carries the type of the message exchanged. We have 16 types of messages including “Request sign-up”, “start chat with a specific user”, and “send broadcast message to all users”.

The User field is either empty or contains a username depending on the type of message being sent. For example, when the client sends a private message to the server, the username is that of the recipient, however, when the server sends back the message to the recipient, the user field of this message carries the username of the sender.

Similarly, The message field also carries a value that is dependent on the message type. This value is encrypted using AES. It might carry the hashed password of the user, for instance, or the body of the message being sent.

Each of the three fields’ values is hidden in an image. The three images are then folded into the JSON object before it gets sent.

5.0 Steganography

Steganography has been used historically to hide information [4]. One of the remarkable examples of such technique in practice is writing the desired message using an invisible ink to be hidden within a fake visible message. Of the contemporary examples of this technique is to hide a text message within another text, image, audio or video file. Unless it is broken, steganography provides protection for both messages and communicating parties for the exchanged, even if encrypted, messages. However, once the system is discovered, it becomes entirely worthless. Moreover, steganography proves to cause an overhead for hiding information. In order to hide a relatively few bits of information, significantly larger data is constructed in order to conceal the message within [3, 4].

The app uses a combined technique merging encryption and steganography. Encryption provides another level of security if the steganography system is ever revealed. Additionally, steganography disguises the encrypted text to evade suspicion. Thus, this combination still has the trade-off of the overhead that steganography causes but secures the exchanged messages between the communicating users.

The library used [2] for steganography is ‘LSB-Steganography’. The file ‘LSBSteg.py’ contains the provided methods by the library. The idea depends on using the least significant bits of the pixel values of the image across the 3 RGB channels to hide the secret message. So first, the text size is saved on 2 bytes in the image, allowing up to 65,536 characters for the text message, to enable us to decode it properly. Afterwards, the text message characters, each encoded as 8 bits, are masked to the corresponding pixels one by one. If the least significant bits weren’t enough to contain the entire message, the second bits are then used, and so on. If the image can’t contain the message even after filling all the available bits, an error is generated. That brings us to the fact that enlarging the image relatively to the message size is preferable, as the larger the data, the more the image is altered. However in our implementation of the technique, we chose to generate random images ten times as big as the secret message and pass them to the library class.

The library provides several methods for encoding and decoding the messages in several formats, including: images, text messages, and binary files. In our code, we only used the ‘encode_text’ and ‘decode_text’ methods to hide the string messages entered by the user through the text fields in the UI.

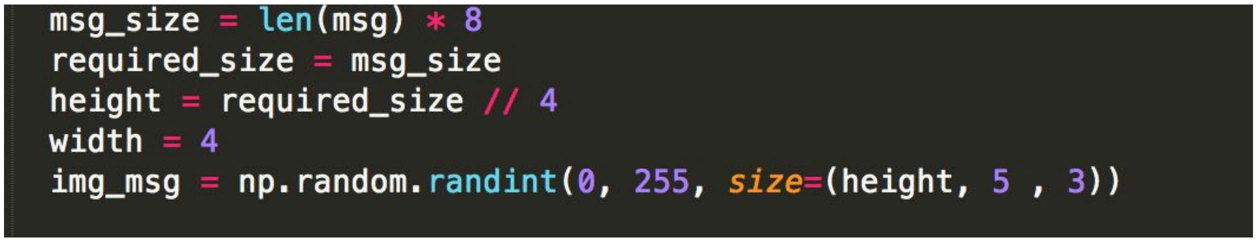
In the ‘common.py’ file, we implemented both encode and decode methods ‘encode_msg’ and ‘decode_msg’, in which we generate 3 different images using numpy random randint methods, to include the code, user and message information. We adjusted the sizes to completely include the text message hidden. Next, 3 instances of ‘LSBSteg’ class are created and the encode_text is called on each to compute the new image inscribing the secret message. Then, as previously mentioned, they are dumped as a single JSON object.

Capacity (ratio of secret message to hosting image/text):

We manually set the image dimensions to be 10 times as the text message size.

Generated image size = ((required_size // 4) * 5),

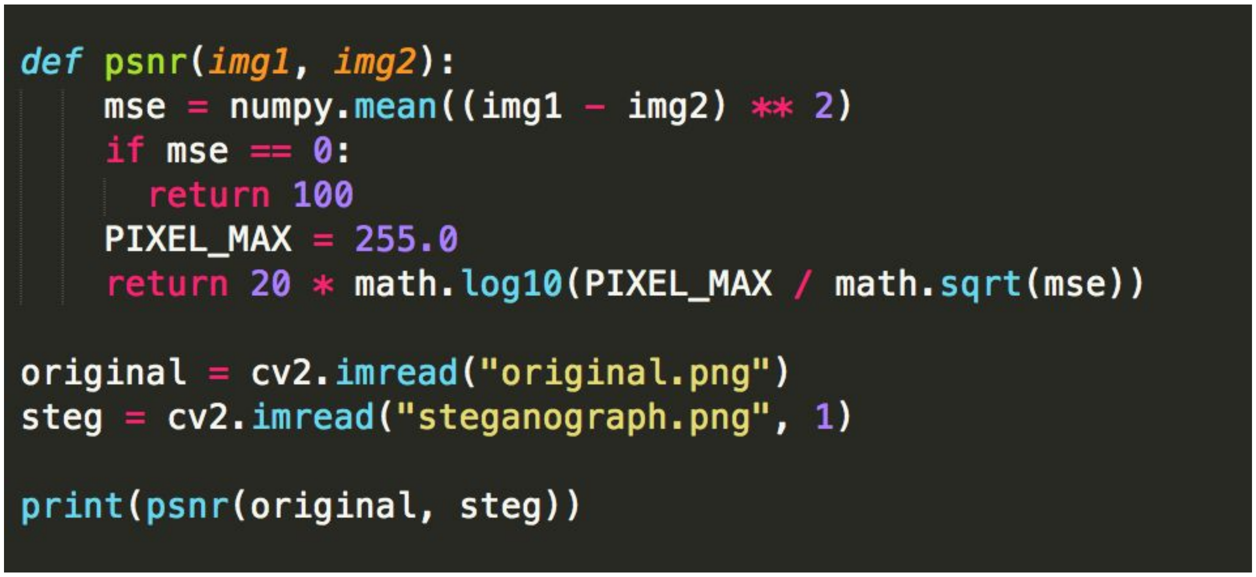
= (((len(msg) * 8) // 4) * 5) = len(msg) * 2 * 5 = len(msg) * 10 pixels



```
msg_size = len(msg) * 8
required_size = msg_size
height = required_size // 4
width = 4
img_msg = np.random.randint(0, 255, size=(height, 5 , 3))
```

So the ratio of the secret message to the hosting image size = **1/10**

Fidelity (distortion of image/text hiding the message, measured as Peak Signal to Noise Ratio):

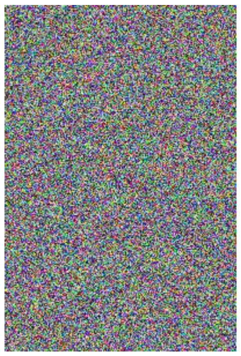


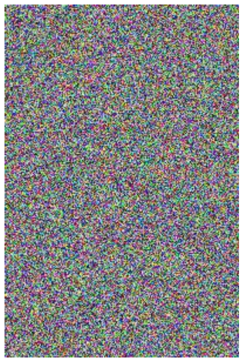
```
def psnr(img1, img2):
    mse = numpy.mean((img1 - img2) ** 2)
    if mse == 0:
      return 100
    PIXEL_MAX = 255.0
    return 20 * math.log10(PIXEL_MAX / math.sqrt(mse))

original = cv2.imread("original.png")
steg = cv2.imread("steganograph.png", 1)

print(psnr(original, steg))
```

We tested this formula on several images we send and we got an average score of 77.39973262532479 [5].

6.0 Man in the Middle Attack

Let’s suppose we have the instance where one user is sending a broadcasted message to the rest of the user saying “Hi again”. As previously mentioned this message was encoded in a randomly generated image. In this case the image to the right. Let’s suppose that somehow a man in the middle managed to intercept the packages being sent via the socket from the client to the server. If that person were to view the image of the message embedded in the json object, that person would see the following image to the left. Clearly it is very hard to tell the difference between those two images, let alone to tell that there is a message concealed within this message. However supposing that the MITM manages to somehow retrieve the concealed message. This is the message that he will see: '7vcqbgl/1Hbpv/Oh9o75jQ==', thanks to the encryption being applied to the message before the steganography concealing is applied to it.





References:

[1] Chat app in Python:
https://medium.com/swlh/lets-write-a-chat-app-in-python-f6783a9ac170

[2] Steganography library: https://github.com/RobinDavid/LSB-Steganography

[3] Steganography article: https://www.garykessler.net/library/steganography.html

[4] William Stallings. 2005. Cryptography and Network Security: Principles and Practice (4th ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[5] PSNR:
https://dsp.stackexchange.com/questions/38065/peak-signal-to-noise-ratio-psnr-in-python-for-an-image,
https://tutorials.techonical.com/how-to-calculate-psnr-value-of-two-images-using-python/?fbclid=IwAR0kgOPjM6dsc1R4Kg9cJiqyOfM0w38s4L4UpV96PvIcavR2tCo8E4VvRkU

[6] Crypto library: AES
https://www.dlitz.net/software/pycrypto/api/current/Crypto.Cipher.AES-module.html

[7] Hashing Passwords:
https://www.vitoshacademy.com/hashing-passwords-in-python/