Report for INFO2222 Project- Security

Members: Shabab (unikey- ssal6113) and Terry (unikey- chyu9221)

Our project used the barebones template of Winston (Link: INFO2222 – Ed Discussion (edstem.org)), while implementing our own security measures on it as well as our own personal CSS style.

1) In the app.py file, the *bcrypt* module is used to hash the user's password before storing it in the database. The password is **salted** and **hashed** using the brcypt.hashpw() function before being stored in the database – the salt is included in the output string of the function. This is a secure way to store passwords as even if an attacker gains access to the database, they will not be able to easily retrieve the plain-text passwords.

2) Using OpenSSL, we have created our own certificate that we signed as a self-certified root authority and in order to trust our certificate we included the certificate inside the trusted root authority folder of the machines used for local development; however, the certificate is solely for the localhost, and may not be utilized for official deployment, in which case a service like Let's Encrypt may be used.

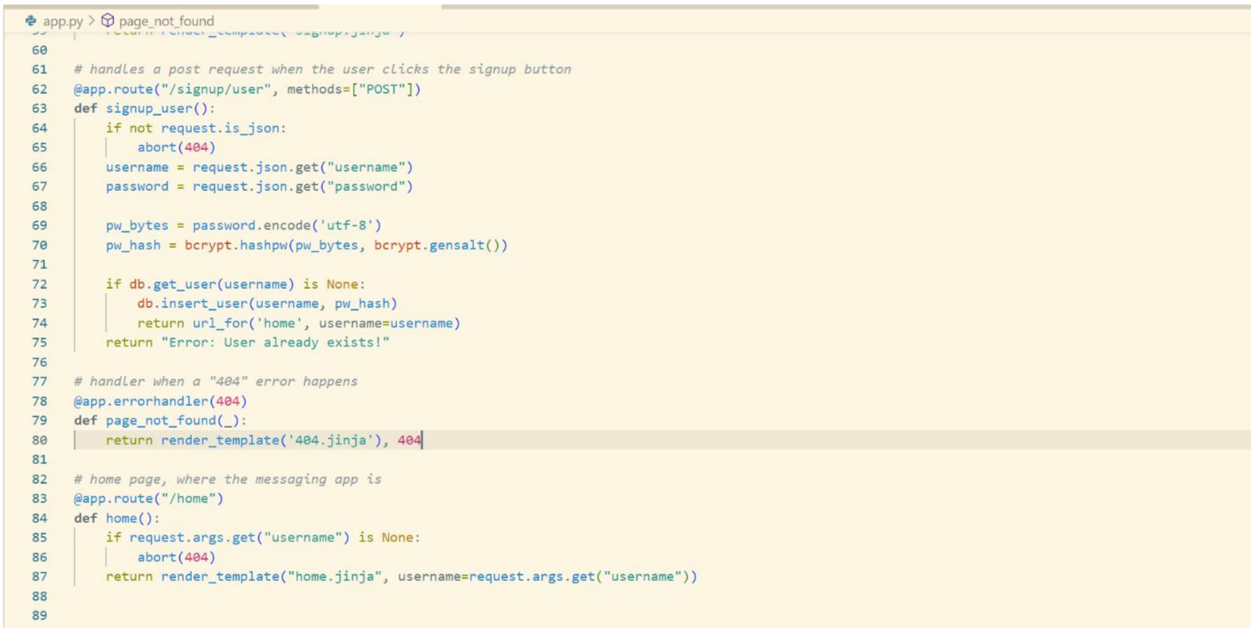
3) With the site operating on https with SSL/TLS, when a password is transmitted to the server for logging in, signing up etc., only the user and the server can see the password, as everyone in between will only see ciphertext. The SSL/TLS employs a Diffie-Hellman exchange mechanism where initially a public key and private key are used to create a session key which only the server and the client know of. As they are derived from the individual private keys, a man-in-the-middle will be unable to decipher the cipher text.

4) The database includes unique hashes – being generated with a salted password – which prevents offline pre-computation attacks. This is because salted passwords make it incredibly difficult to use a hash table. When a user tries to log in, the bcrypt.checkpw() function extracts the salt and re-computes the hash to check if the entered password + salt matches the original input which generated the stored hash. The checkpw() function is also intentionally computationally intensive, so as to reduce the efficiency of a brute force attack.

5) The most important part of our project. Here we used a symmetric encryption, where the secret master key is available to all clients but inaccessible to the server, and server code resides in the socket_routes.py. The server side can only forward messages to the client side. In the JavaScript part of our socket io module, where the client-side code is handled, before sending a message to a server, we encrypted it with the secret master key before sending. After the server broadcasts the encrypted message to all the recipients in the room, the recipient's side decrypts the password using the same key to obtain the message. Hence, the server can only see the ciphertext, and we will provide screenshots of the ciphertext below.

To provide evidence for these claims, we will provide pictures of code as well as our application in operation.

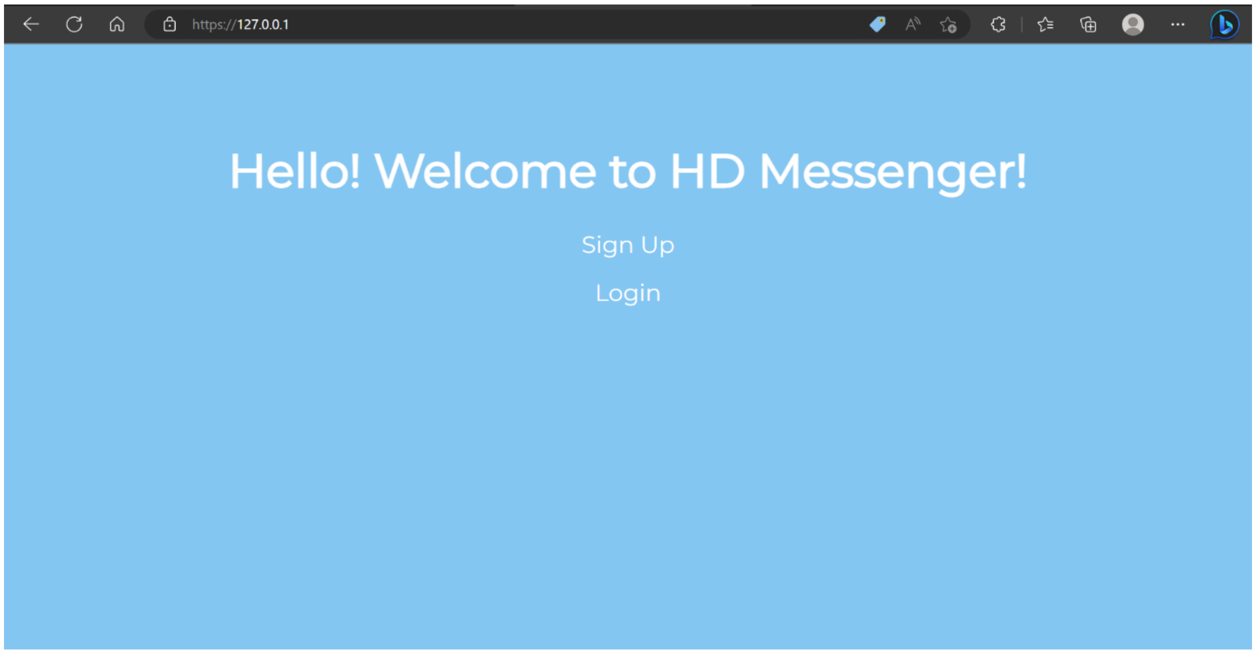
For points 1 and 4, the code for hashing and salting before storing passwords should suffice:


app.py > page_not_found

```python

# handles a post request when the user clicks the signup button
@app.route("/signup/user", methods=["POST"])
def signup_user():
    if not request.is_json:
        abort(404)
    username = request.json.get("username")
    password = request.json.get("password")

    pw_bytes = password.encode('utf-8')
    pw_hash = bcrypt.hashpw(pw_bytes, bcrypt.gensalt())

    if db.get_user(username) is None:
        db.insert_user(username, pw_hash)
        return url_for('home', username=username)
    return "Error: User already exists!"

# handler when a "404" error happens
@app.errorhandler(404)
def page_not_found(_):
    return render_template('404.jinja'), 404

# home page, where the messaging app is
@app.route("/home")
def home():
    if request.args.get("username") is None:
        abort(404)
    return render_template("home.jinja", username=request.args.get("username"))

```

Figure 1: src: password salting and hashing

For point 3 of our project spec, consider the lock on our webpage, certifying a trusted SSL certificate authorization:



Hello! Welcome to HD Messenger!

Sign Up

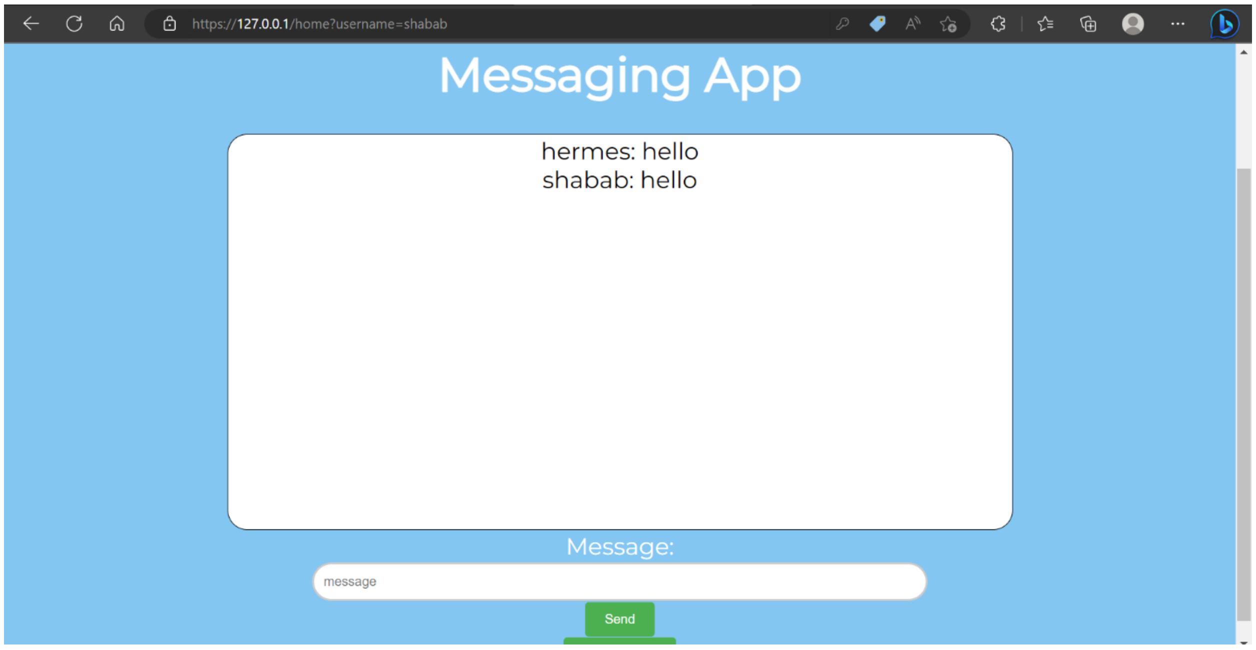
Login

Figure 2: Home page

For point 5, we will show the end result, where two users can see their chats normally, **also**, to prove our messages are end-to-end encrypted, we will also demonstrate what happens if on the recipient side we don't decrypt the message, that is, we will show you also what the server sees as the message is being transmitted to a recipient.

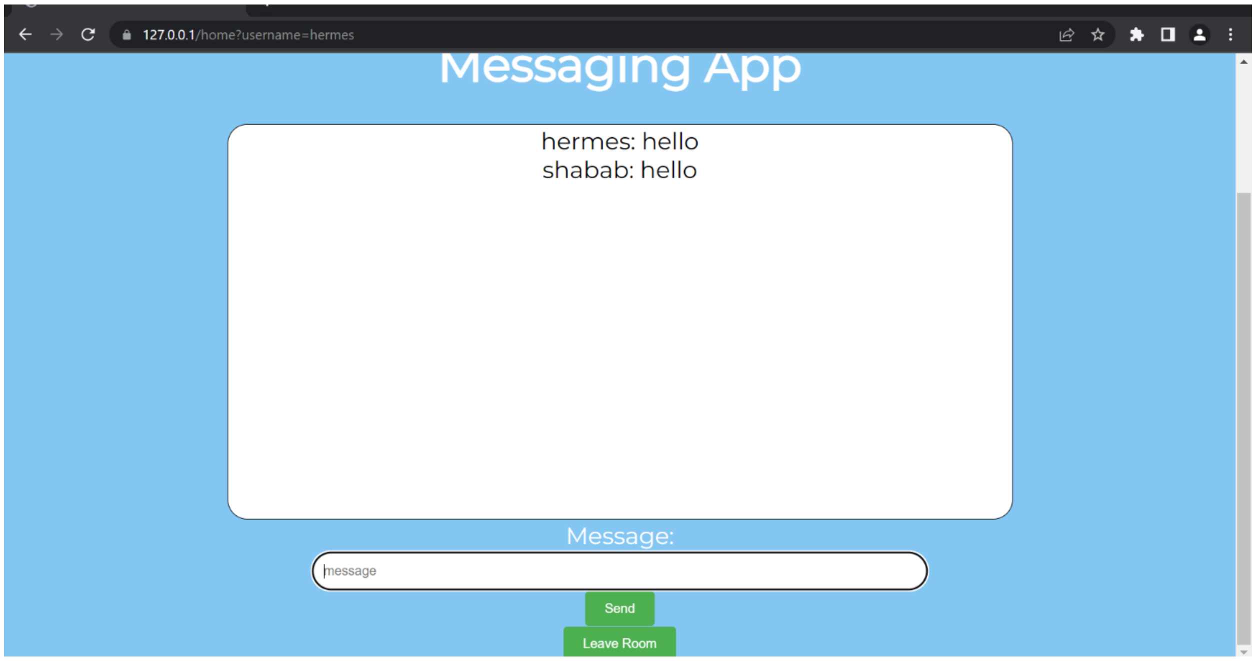
This means that anything transmitted from client to server, whether it be passwords or username, are fully encrypted, and anyone eavesdropping in the middle will simply receive ciphertext. So, it qualifies our point 3 of the project spec.

1) In the case where the end-to-end encrypted conversation is successfully encrypted and decrypted on the sender and recipient side respectively with server just receiving/sending ciphertext:
    a. In this case, both Hermes and Shabab say Hello



hermes: hello
shabab: hello

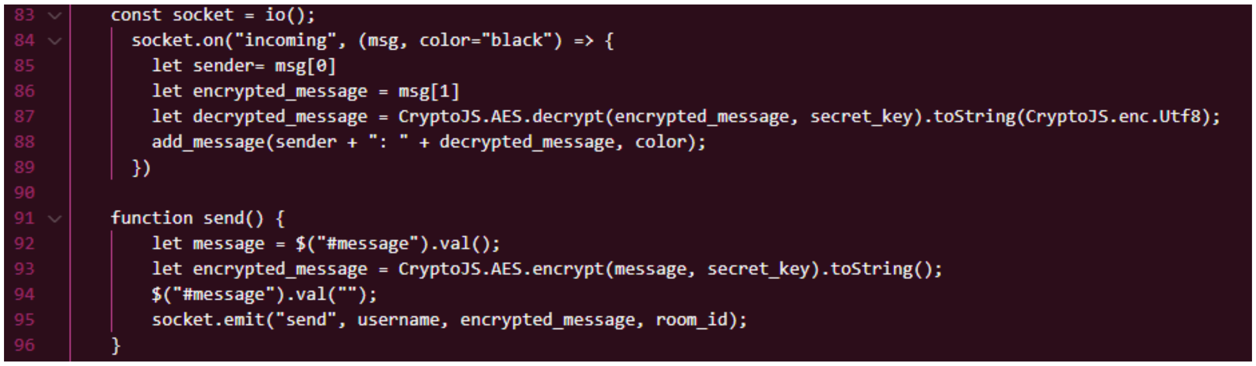
Figure 3: Shabab's client on Chrome


127.0.0.1/home?username=hermes

Messaging App

hermes: hello
shabab: hello

Figure 4: Hermes's client on Firefox

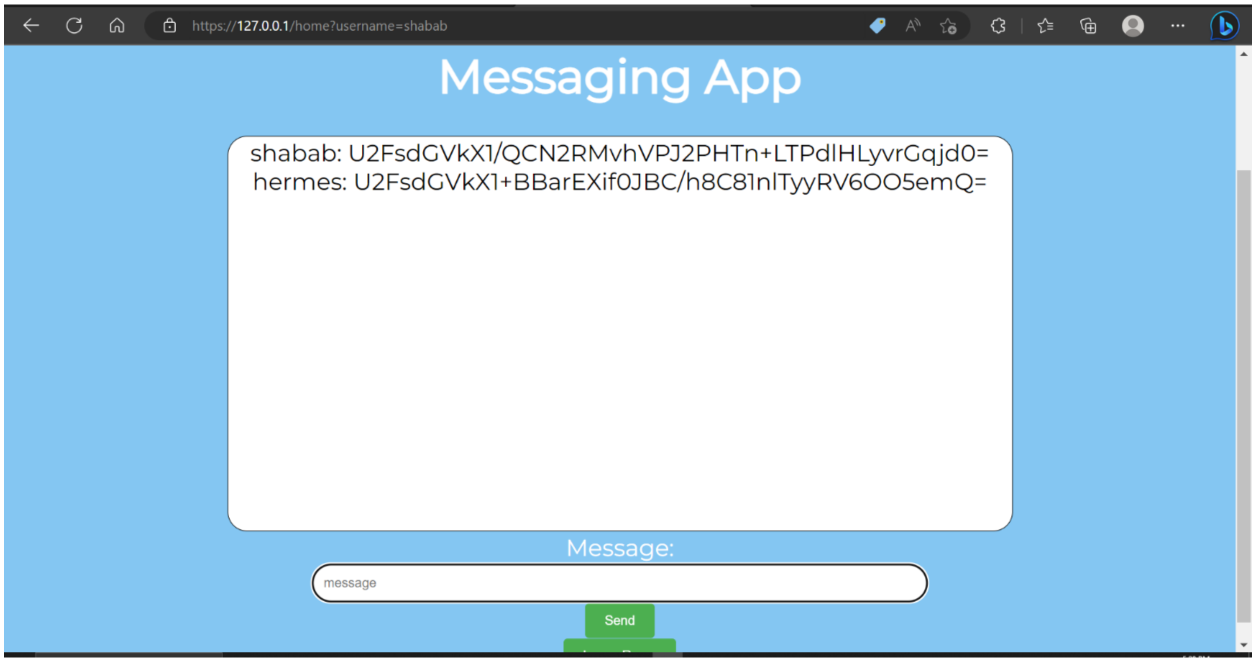


```
  const socket = io();
    socket.on("incoming", (msg, color="black") => {
      let sender= msg[0]
      let encrypted_message = msg[1]
      let decrypted_message = CryptoJS.AES.decrypt(encrypted_message, secret_key).toString(CryptoJS.enc.Utf8);
      add_message(sender + ": " + decrypted_message, color);
    })

  function send() {
      let message = $("#message").val();
      let encrypted_message = CryptoJS.AES.encrypt(message, secret_key).toString();
      $("#message").val("");
      socket.emit("send", username, encrypted_message, room_id);
  }
```

Figure 5: src: encryption and decryption

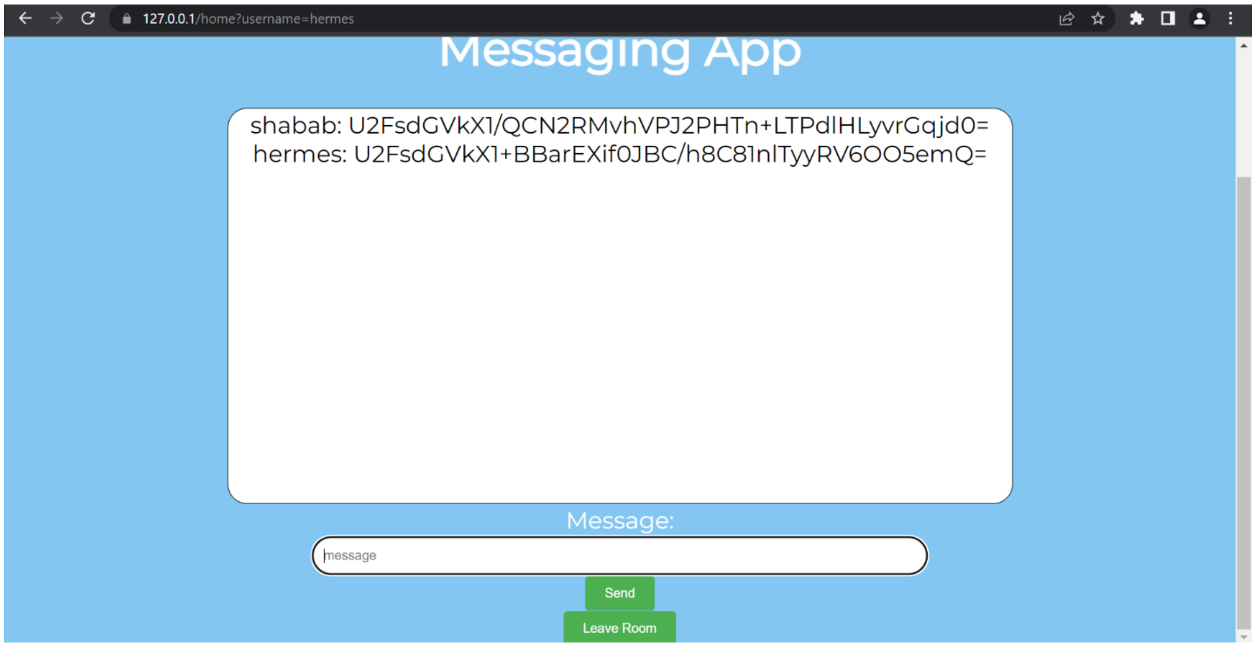
Now if we disable decrypting the message on the recipient side, we will get messages in the form of what the server sees (this can be shown in the server console but we display it on the client side to demonstrate the continuity):

b. In this case, both Hermes and Shabab have said hello:



shabab: U2FsdGVkX1/QCN2RMvhVPJ2PHTn+LTPdlHLyvrGqjd0=
hermes: U2FsdGVkX1+BBarEXif0JBC/h8C81nlTyyRV6OO5emQ=

Shabab’s side



shabab: U2FsdGVkX1/QCN2RMvhVPJ2PHTn+LTPdlHLyvrGqjd0=
hermes: U2FsdGVkX1+BBarEXif0JBC/h8C81nlTyyRV6OO5emQ=

Hermes’s side

Notice how both of their messages are encrypted and unable to be read by the server.

The division of our tasks was determined with the weighed points in mind. Shabab implemented the end-to-end encryption and CSS. Terry implemented the SSL certification for the https connection and the handling of the passwords. For every step of the project, research was shared for every feature developed.