Report for INFO2222 Project- Security

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

Our project used the barebones template of Winston (Link: [INFO2222 – Ed Discussion (edstem.org)](https://edstem.org/au/courses/10797/discussion/1292498)), 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.haspw() function before being stored in the database, and when a user tries to log in, the bcrypt.checkpw() function is used to check if the entered password matches the stored hash. 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 our computers; 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) Due to the certificate, we have created, our site operates on https, so 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. Our justification for SSL being sufficient is by this Ed Post:   
[INFO2222 – Ed Discussion (edstem.org)](https://edstem.org/au/courses/10797/discussion/1286708), where a staff has pointed out https is sufficient for transmitting password to and from the server securely.

4) To prevent offline pre-computation attacks, using the bcrypt library we added a randomly generated salt to the password before hashing and storing the password in the database, which can prove very effective against pre-computation attacks. To check if the inputted password matches the stored hash of the password, the inputted password is hashed and then compared with the stored hash using bcrypt to verify.

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.

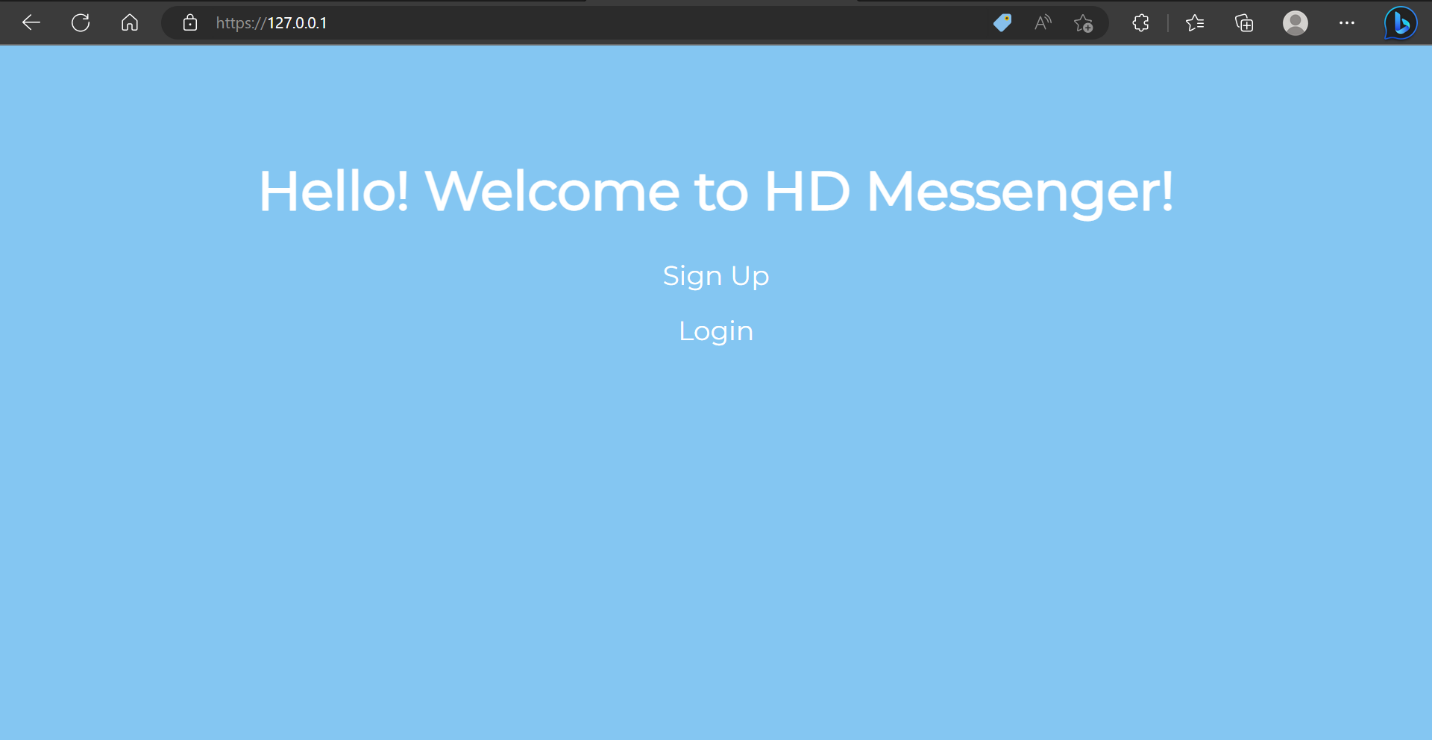
To provide evidence for these claims, we will provide pictures of code as well as our application in operation.

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



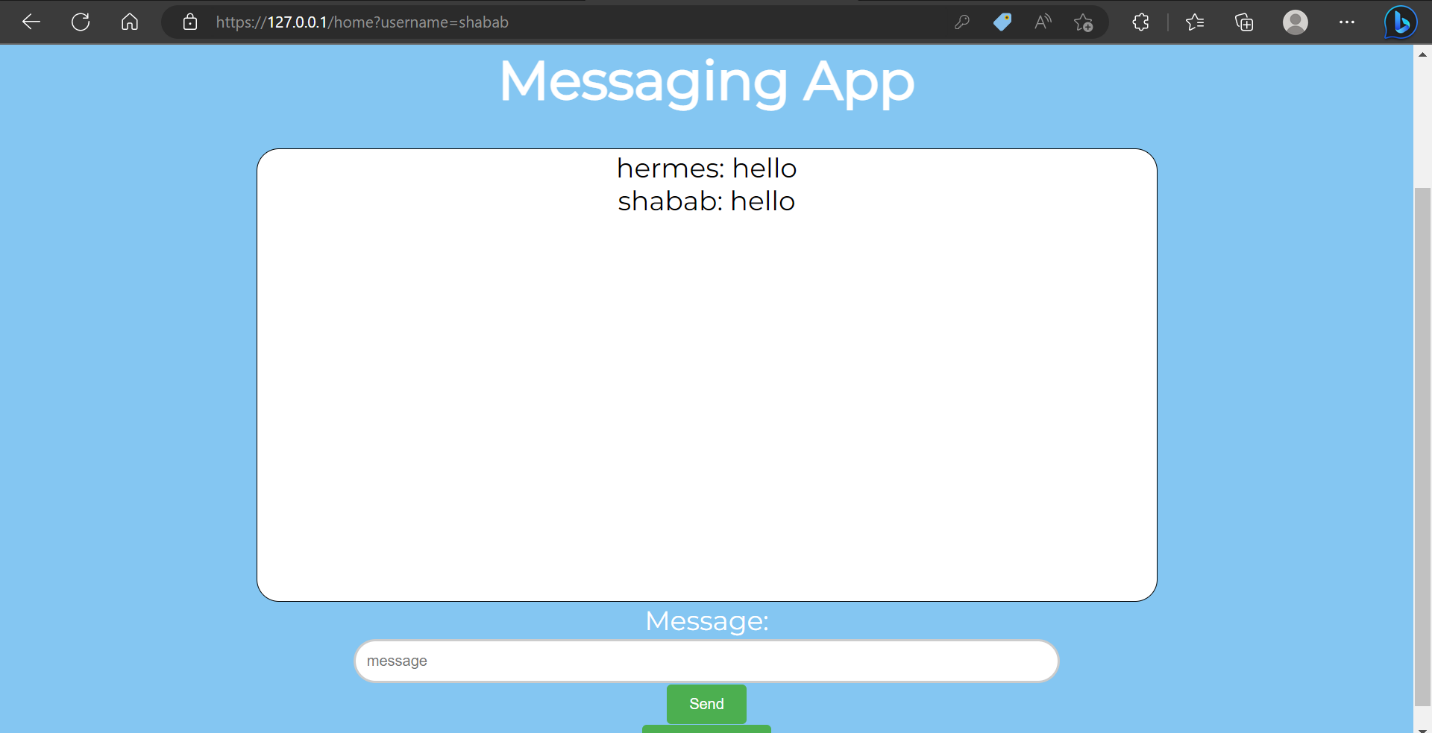
For point 5, we will show the end result, where two users can see their chats normally, **also**, to prove our messages are end-to-end encrypted, we will also demonstrate what happens if on the recipient side we don’t decrypt the message, that is, we will show you also what the server sees as the message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

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



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:
   1. In this case, both Hermes and Shabab say Hello



Shabab’s side

Graphical user interface, text

Description automatically generated

Hermes’s side

1. Now if we disable decrypting of the message on the **recipient** side, we will get messages in the form of what the server sees:
   1. In this case, both Hermes and Shabab have said hello:

Text

Description automatically generated

Shabab’s side

Graphical user interface, text

Description automatically generated

Hermes’s side

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

Now, our tasks were accomplished by the two of us equally: both contributed equally in all aspects, while Shabab implemented the end-to-end encryption and CSS, Terrence implemented the SSL certification for the https connection as well as salting and hashing the password, all of which is crucial for our security aspect of the program.