Telecommunications Assignment 2

Name: Claire McNamara, Student Number: 17332568

SecureWebApp

A LOTR style groupchat where the admin has full control. Only users added by the admin can decrypt the posts/post messages. It implements a Hybrid Crypto System using RSA and AES in mode CBC – RSA. Each user has a public and private key, and the group itself has a symmetric key with which all posts are encrypted in. Two libraries are used to do the above - Cryptography for AES and Pycryptodome for RSA. The web framework used is Django and the front end was enhanced using bootstrap.

As it is a Django web app here is a link to the repository that I created for the project - https://github.com/mcnamacl/SecureWebApp

As it has also been deployed to Heroku (a web application hosting service), here is a link to the live version:

https://lotrsecuregroupchat.herokuapp.com/signup_show

In the readme of the Git Repository I have included two GIFs of both the application working locally and it working on Heroku. The GIF of it working locally is more detailed and so I would recommend looking at that to get a better insight into what it looks like/how it works.

The Different Branches

There are two separate branches. One named master and one named localsecurewebapp. The reason for this is that Heroku (the master branch) uses a different database system than the one I created and so the encoding/decoding of data was different.

localsecurewebapp

The Safe Version:

Running the local version means that the when the public and private keys are generated only the public key is stored in the database and the private key is written to a file on the local computer.

The main files in this branch:

views.py – this is the python code that runs the entire application. It handles all the different requests from the front end, adds new users to the database and verifies log ins, encrypts and decrypts the messages and generally handles everything. It can be found in the folder named groupchat.

models.py – this is where the information for the database is defined. There are three different models. One for defining the information that is stored about messages, one for defining information

about groups and one for defining what extra information is stored about a user. Also found in the groupchat folder.

Structure of the database:

Message

- Message content.
- Sender.

Group

- Group name.
- A relationship to messages.
- The current symmetric key for the group.

ExtraUserInfo

- The username for who the extra information relates to.
- Whether they are the Admin or not.
- The relationship to the group they are currently a part of.
- The symmetric key encrypted with their public key. (If they have added to The Fellowship)
- Their public key.

All frontend parts:

- HTML: groupchat/templates/groupchat
- CSS: groupchat/static/css

master

Heroku Version:

As I had the local version working safe and I needed to work on other projects I left a TODO in that to figure out a way using Django to download a file that I would create on the app with their private key.

Step-By-Step – Local Version

(Remote version is extremely similar except for the differences outlined above.)

Sign up for the admin is only ran once and this is for the Admin to create their password etc. They have to use the username Admin.

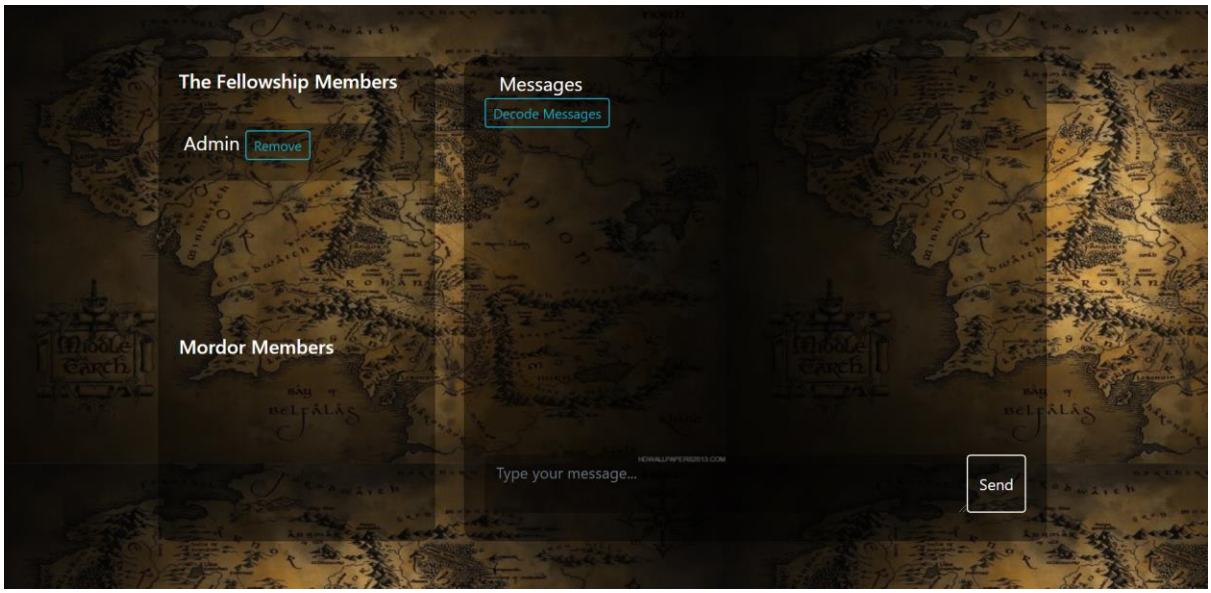
A user is created for the Admin that stores their name and password securely using the Django User Model.

An ExtraUserInfo entry is also added. This contains their status (admin or not), the group that they are in, their public key. Their private key is stored as a PEM file on the local computer.

During the admin's sign up process, in the background, two different groups are created – The Fellowship (the secure group) and Mordor (where new users end up when they first sign up and where you stay until approved to join by the Admin). This involves creating a symmetric key for The Fellowship and registering both groups in the database. This is the only not secure part of the whole process. The symmetric key for The Fellowship can be accessed by accessing the group information in the database however I have left that as a TODO as the rest of the code is secure.

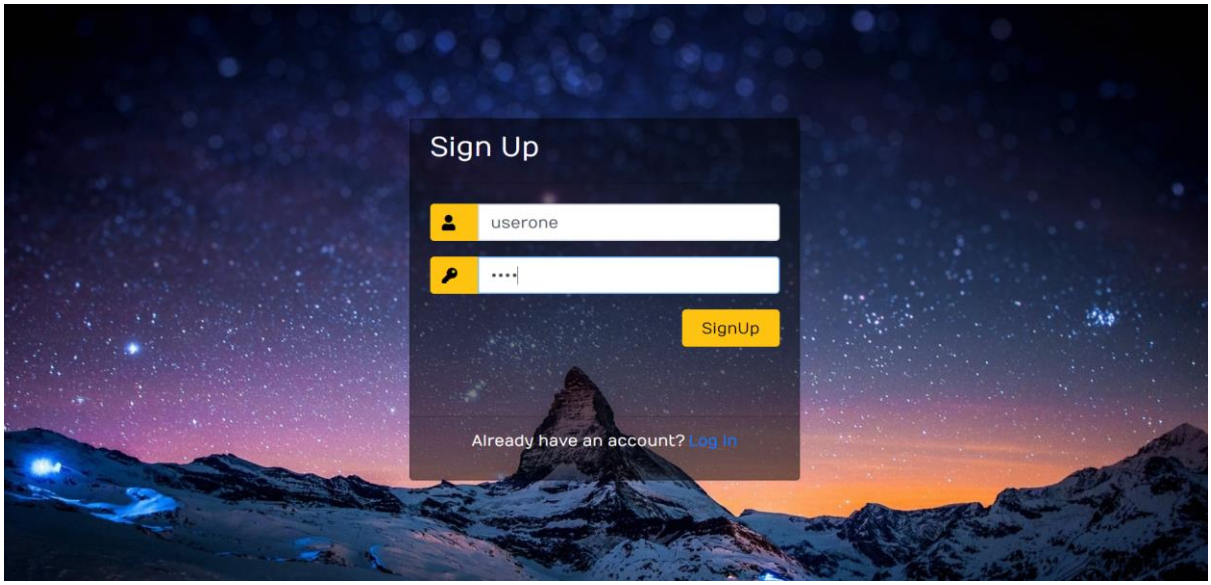
Once a symmetric key has been generated for The Fellowship, the Admin's public key is used to encrypt that symmetric key and it to the information stored in their ExtraUserInfo entry.

The admin is then brought to the admin page:

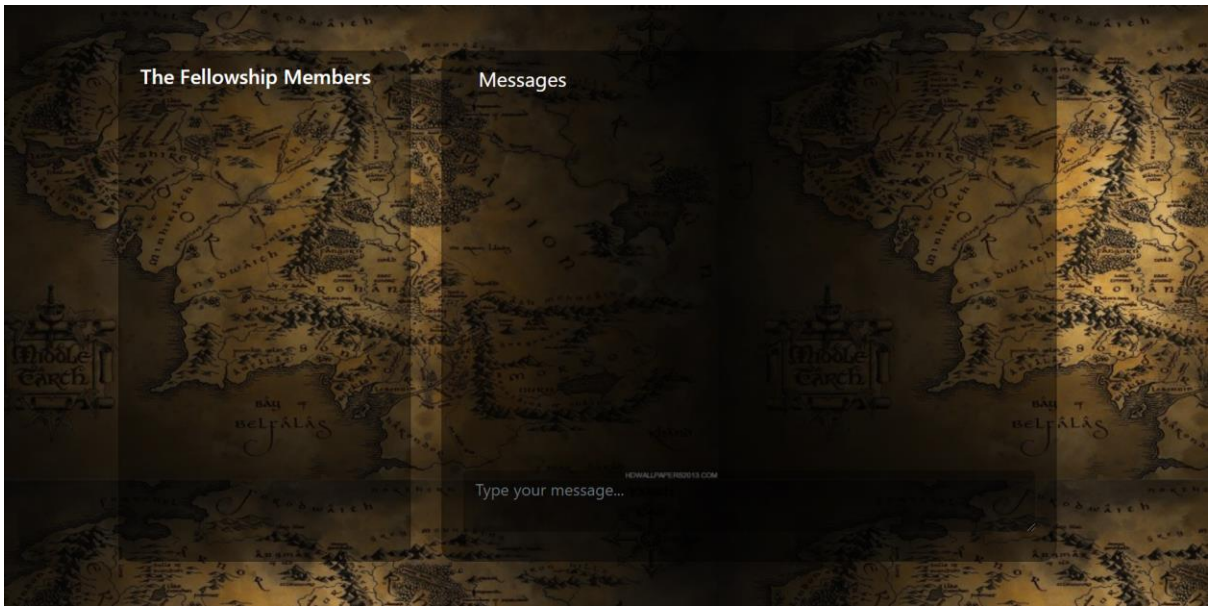


A new user signing up:

When a new user signs up a similar process happens to the above with a few notable exceptions.



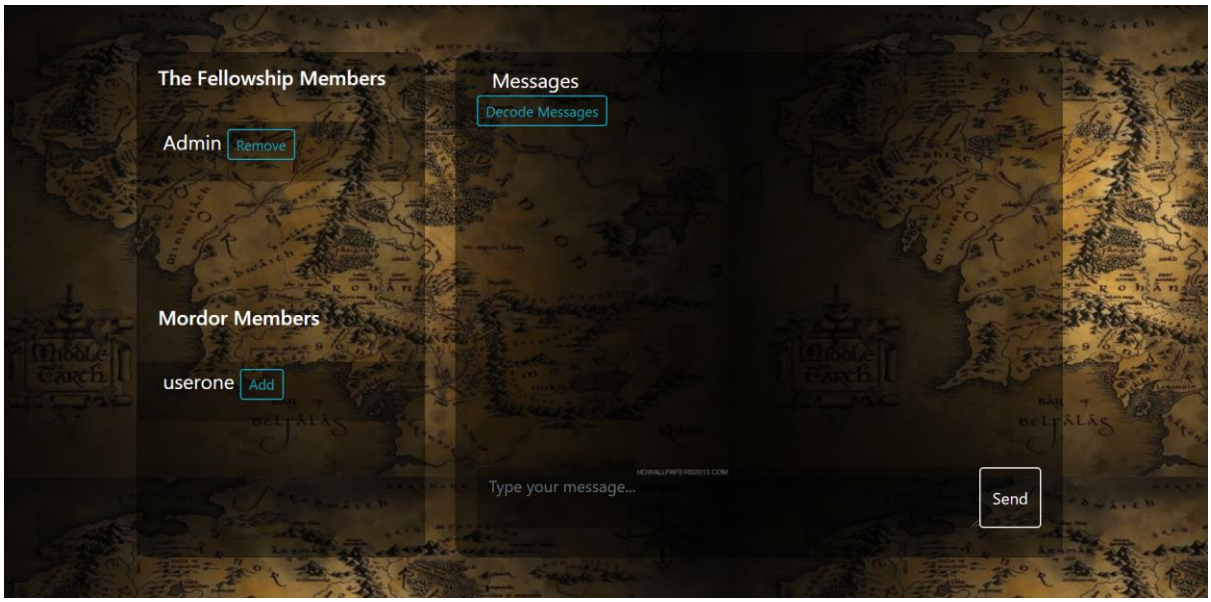
A new user is created with their username and password again, using the Django User Model and a new entry is added with the extra information. Instead of being added to The Fellowship they are immediately added to Mordor. A public and private key again are generated with the private key being stored locally and the public key being stored in the database. They are not given admin status and when the first sign up the page the are brought too looks like this:



As you can see, they cannot send messages or see who the other members of the group are.

Adding a new user:

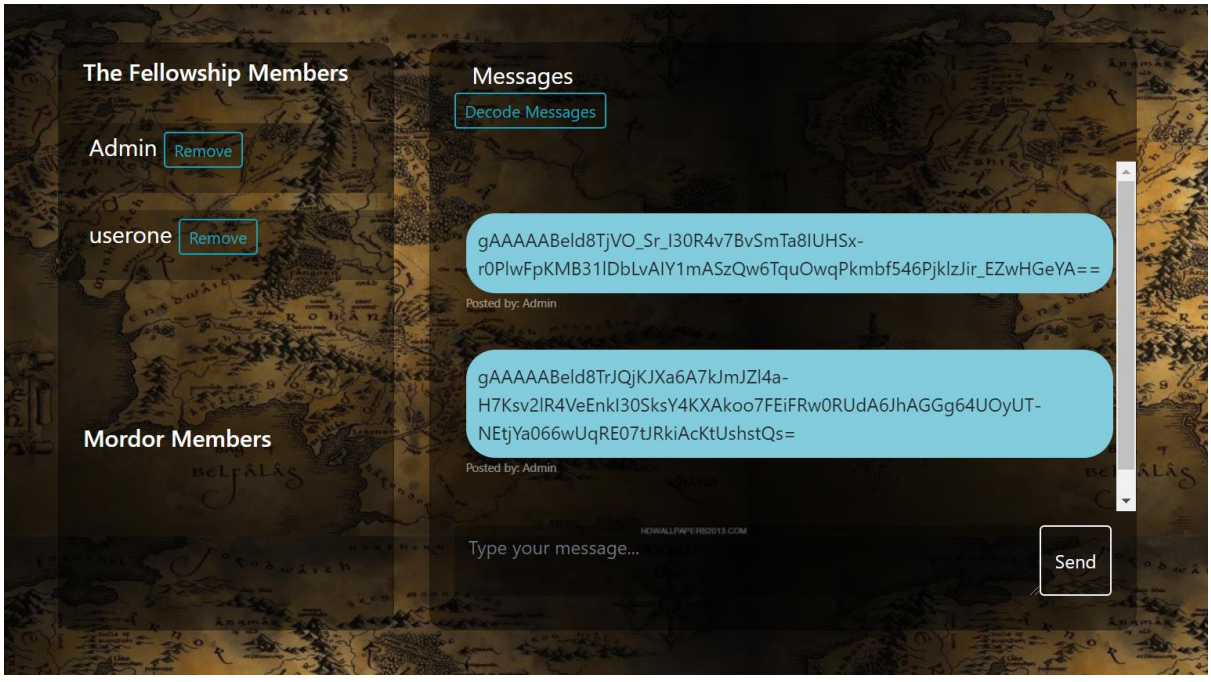
When a new user signs up they appear in Mordor on the admin page:



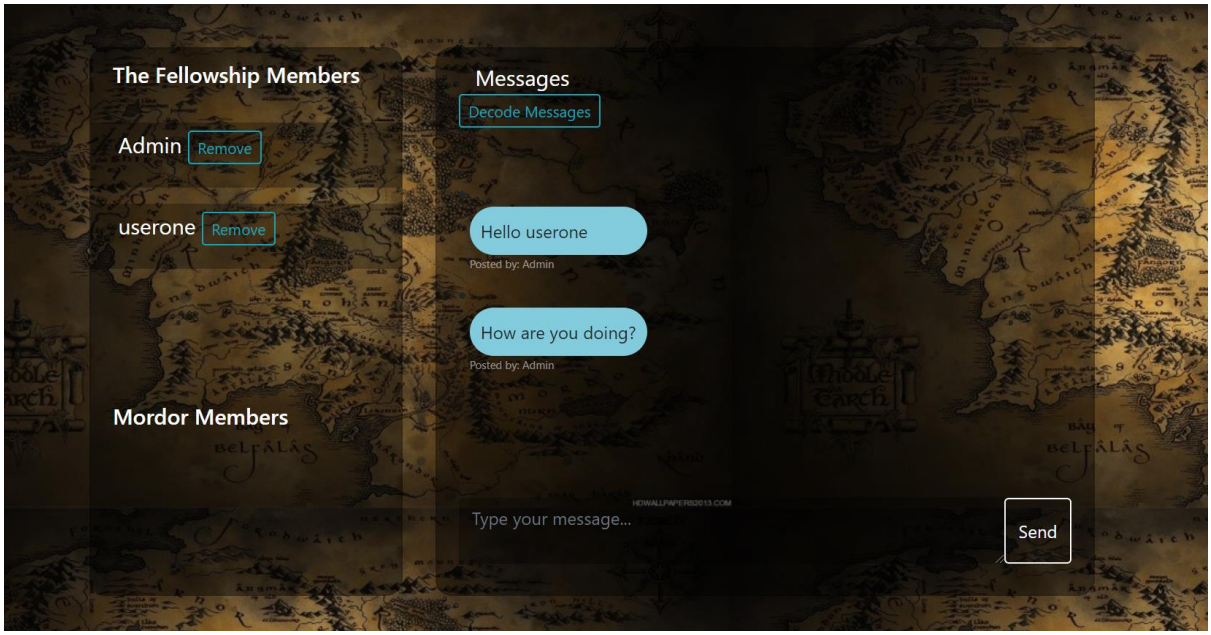
If the Admin chooses to add them, the current symmetric key is gotten from the Fellowship group. It is then encrypted using their public key and stored in the SQLITE database with their other information. There group info is updated also so it now reflects that they are part of The Fellowship.

Sending a message:

When a user sends a message it first looks like so:



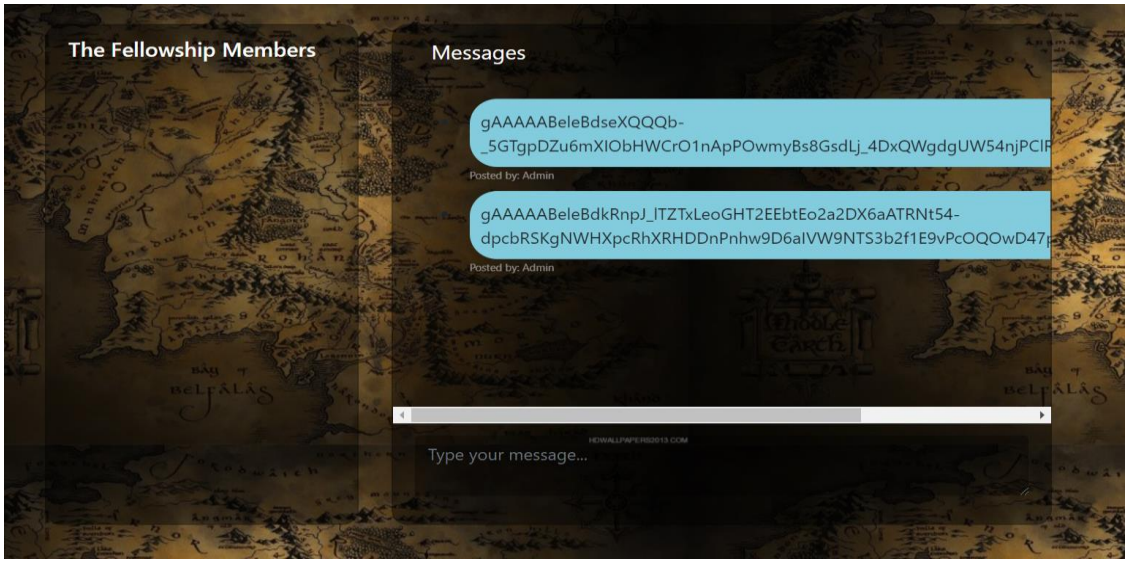
However once the decode button is pressed the messages are decrypted and displayed:



In the background, what is happening is that for every individual user who is part of the group, their encrypted symmetric key is being fetched from the database and their private key is being read from the PEM file stored locally. Using the private key the program decrypts the symmetric key and message by message decrypts them using the symmetric key which are then showed to the front end as decoded messages.

Removing a user:

When a user is removed, the symmetric key has to be updated. This means a new one is generated and all the messages are decrypted using the old key and re-encrypted using the new key. All users have their version of the symmetric key encoded with their public key and stored back in the database. As they have been removed from the private group, while they still can view the encrypted messages, they no longer can decrypt them and even if they try, the messages have been re-encrypted using a new symmetric key.



Resources Used:

https://cryptography.io/en/latest/fernet/

https://bootsnipp.com/snippets/vl4R7

https://pycryptodome.readthedocs.io/en/latest/src/introduction.html

https://cryptobook.nakov.com/symmetric-key-ciphers/aes-cipher-concepts

https://en.wikipedia.org/wiki/Hybrid_cryptosystem

https://scribles.net/deploying-existing-django-app-to-heroku/