Wack Chat

A project report submitted in fulfillment of Computer Security Course

Submitted by

Subhash Chandra, Teddy Diallo, Joshua Wiseman, Jamshed Karimnazarov

Under Supervision of

Prof. Dr Shanguing Zhao

School of Computer Science
University of Oklahoma, Norman, OK
May, 2024

Project Description

Wack Chat is a secure instant messaging platform designed to facilitate confidential point-to-point communication. The application features a dynamic front-end interface developed with React.js and integrated with a Firebase backend. The platform has the following key features:

- Account Management: Users can effortlessly register and log in to manage their profiles and maintain session integrity.
- Contact Search and Messaging: The system allows users to search for other registered individuals and engage in private messaging.
- Robust Security: Using protocols and algorithms discussed in our coursework, Wack Chat ensures that all communications are encrypted and secure.

This project serves as a practical application of our theoretical knowledge with the implementation of security measures, including encryption techniques and key management protocols.

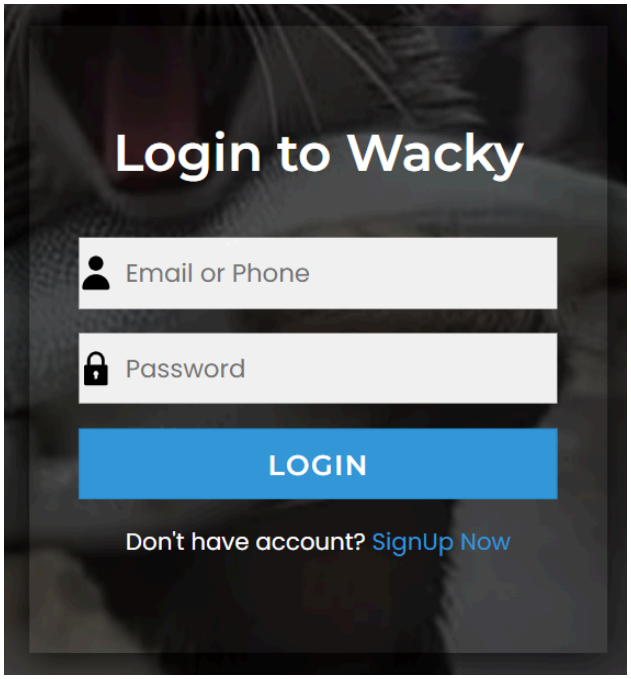
GitHub Repository: https://github.com/XixWISExiX/WackChat

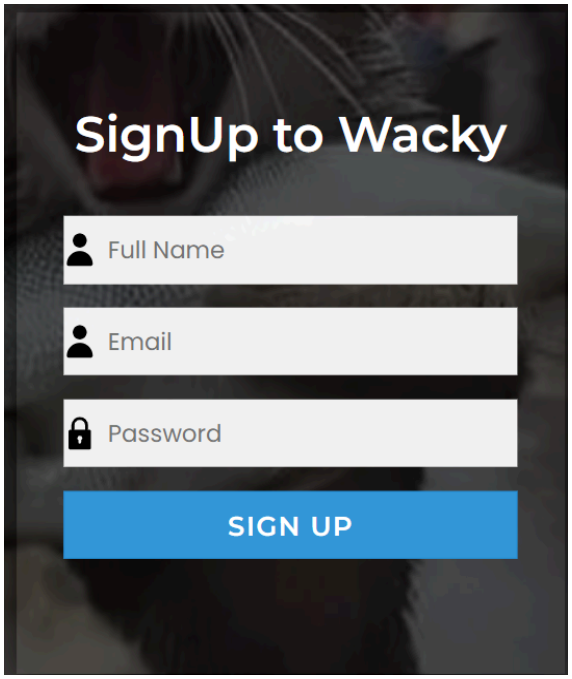
How to Access the Project

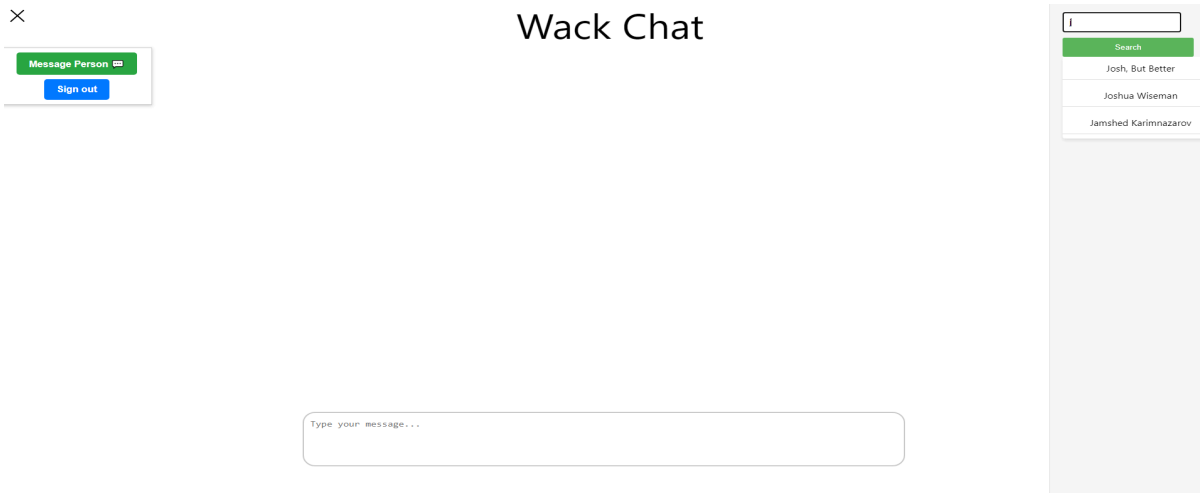
Wack Chat is hosted on GitHub, allowing users to easily download and set up the platform. To begin, clone the project repository from GitHub and execute “npm start” from the command line to launch the application (Note: you will not be able to access the application currently if you don’t have the .env.local file with the database information, this will be solved when application is deployed).

1. Initial Login Screen: Upon launching Wack Chat, you will be greeted by a login screen. Here, you can enter your email and password. The system checks these credentials against those stored in the database. If they match, you will gain access to the platform.
1. Account Creation: If you do not have an existing account, click the 'Sign Up' button to register. Follow the prompts to set up your new account, which will then be verified and stored securely in our database.
2. Messaging and Searching for Users: Once logged in, you can begin communicating by clicking the 'Message Person' button. This opens a search bar where you can look for other users. As you type, the system will suggest users from the database, allowing you to quickly find and select the person you wish to message.

The user interface is designed to be intuitive, facilitating smooth navigation and interaction within the application.





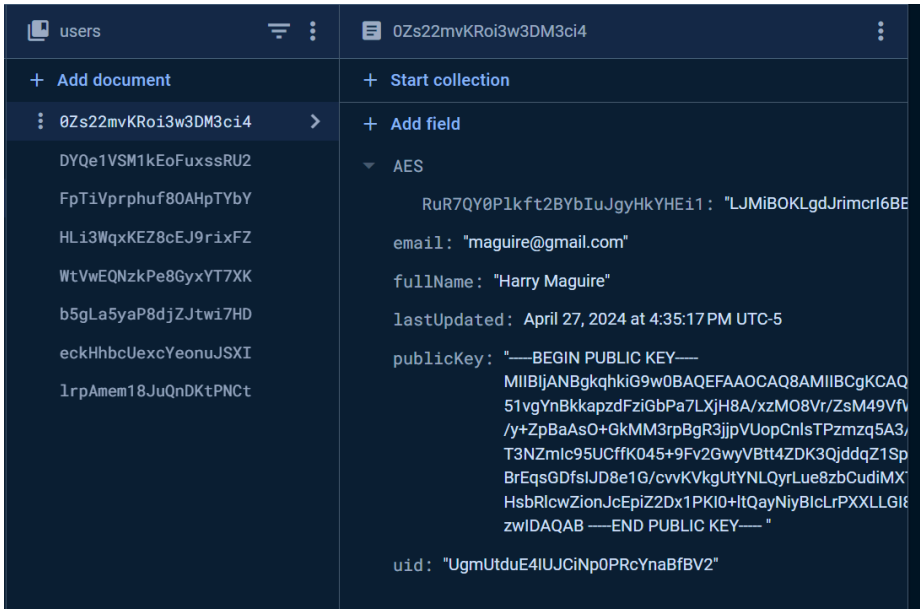


Key Management Protocols

Upon registration, each user is assigned a unique pair of cryptographic keys: a public key and a private key to communicate via RSA. These keys are essential for ensuring secure communication. The public key is available to other users to facilitate encrypted communications and is stored in the database, while the private key remains confidential and stored securely in local storage.

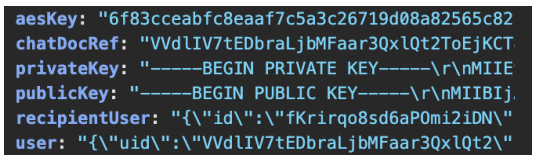
During a communication session, when a user searches for and selects another user to message, a session key is generated if one is not already generated.

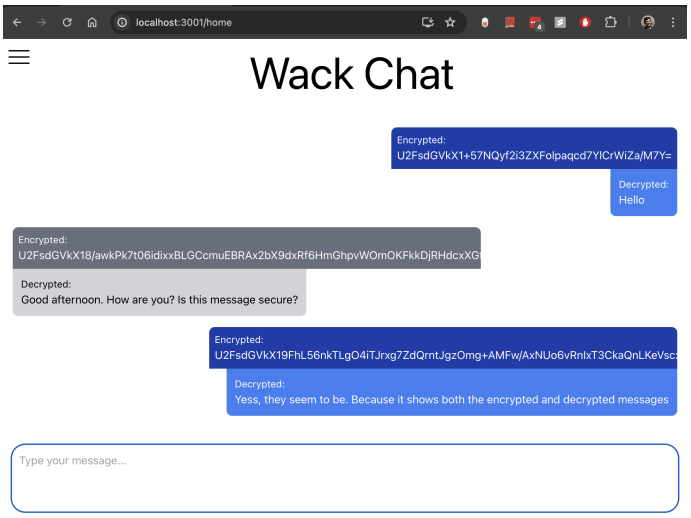
This session key plays a critical role in securing the communication session between the two parties. It is encrypted using the Advanced Encryption Standard (AES) algorithm, ensuring that only the intended recipients can decrypt it. The AES keys exchanged are encrypted with the user's public key and stored in their own user profile (stored in two locations in the database). Also, we are using AES (256 bit version) here because it is not possible to break in a reasonable amount of time in our day and age, hence using this over something like DES (56 bits). The **encrypted** session key is then securely stored in the database and is used for encrypting and decrypting messages during that session. The session key is specific to communication sessions between each two users.



Message Handling Protocols

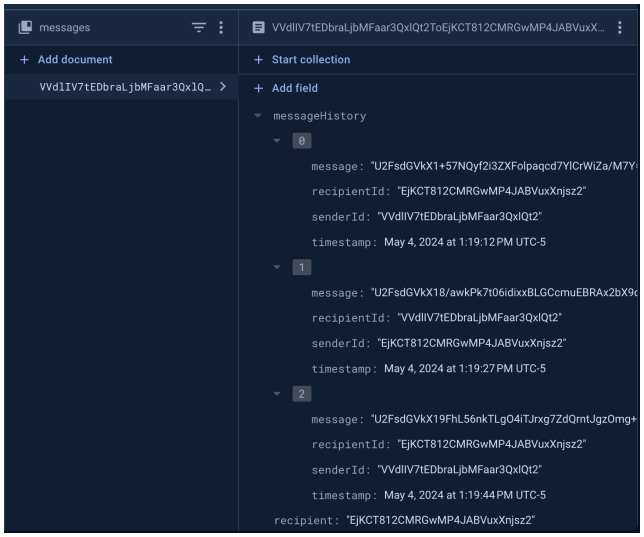
Once you send a request to the user, for both users the AES key gets decrypted and stored in local Storage (screenshot below) in the browser so you can use that for message encryption and decryption.





Users can then start interacting with each other as can be seen on the screenshot on the left. Each message gets encrypted with the AES key and is stored in the database. Once the message is added to the databases, the receiving user receives it and decrypts it with the AES key stored in the local storage. Both the encrypted and decrypted messages are shown on the user interface.

The image on the right shows how the messages are stored in the database. As you can see, only the encrypted messages are stored hence the system being end-to-end encrypted.



Encryption Technique (Extra Credit 2: "Pre shared passwords not needed")

How this is done on a technical level is that there is a function that checks whether a session key is generated. If there is a session key generated, it is located in the AES column of the database, and is represented in a hash map. The key is the receiver ID, and the value is the encrypted AES key (encrypted with the sender's public key). In the case where the key is not generated, another function is called to handle the session key process. The sender will go ahead and generate an AES key themselves, then encrypt it with the receiver public key and add the key:value pair as listed before to the database. Along with this, the connection must be established from the other end, meaning that now the receiver must have the key be the sender ID, and the value be the receiver's public key encrypted AES key. After this database update is done, the session key can be pulled from the database by both users. There is then a

call to another function to decrypt the AES session key with the user's private key to finally obtain the session key to use for messaging. Currently, this session key generation is done ONCE and never done again for ease of implementation.

The message encryption is built upon the session key generation and after the session key generation is complete, both users are now able to message each other. When a message is typed out and the user submits the message, it will call a function which takes the message and encrypts it with the AES session key. After which this encrypted message is sent to and stored in the database, under a chat table. When that chat table is updated, it pings the receiver of the message (therefore updating their chat logs with the new messages). After which, when the receiver next logs in, the application will grab the new database information and decrypt it with the AES session key, allowing for the receiver to look at the new messages sent.

Security Improvement Strategies (Extra Credit 1: "Improve the Encryption Algorithm")

- Generate a new session key that already exists between two users when both users log off the application. Constant regeneration of the session key will lead to potential resets in the attacker's brute force approach (testing all combinations) making it more computationally difficult to crack.
- The server itself could have a public and private key, acting like the CA. With this, you can send the newly made user your public key, have them verify you, and then have them generate a public and private key. Of which, the public key being sent to the database will be encrypted with the database's public key. Account generation doesn't happen as much as message generation, but could have still been prone to man-in-the-middle attacks, with this solution, that man-in-the-middle attack no longer exists.
- Having the website be hosted on HTTPS will also improve security, because it doesn't just rely on the custom encryption algorithms we have made. With our approach, we believe it to be secure, however, there could be bugs we don't see, and having something like HTTPS to help further encrypt messages will only make the application more secure than previously.

Conclusion

Q: With a key no less than 56 bits, what cipher you should use?

A: AES (128+ bits)

Q: DO NOT directly use the password as the key, how can you generate the same key between Alice and Bob to encrypt messages?

A: Digital Envelope using the previously generated public and private keys.

Q: What will be used for padding?

A: Random number generator which is like 16 bits long.

Q: A graphical user interface (GUI) is strongly preferred. When send a message, display the sent ciphertext. When receive a message, display the received ciphertext and decrypted plaintext.

A: Ok

Q: How should Alice and Bob set up an initial connection and also maintain the connection with each other on the Internet? (You may refer to socket/network programming in a particular computer language)

A: Connecting Firebase Values (Bob to Alice and Alice to Bob) along with update listeners to the messaging table, so GUI updates with recently sent messages.

Q: If Alice or Bob sends the same message multiple times (e.g., they may say “ok” many times), it is desirable to generate different ciphertext each time. How to implement this?

A: As previously mentioned, appending a random 16-bit number would help this, and you wouldn’t need to generate a random session key every time (generating a new session key every few hours or something like that would also work in addition to the previous).

Q: Design a key management mechanism to periodically update the key used between Alice and Bob. Justify why the design can enhance security.

A: Every hour, a new key will be generated. This would enhance security because the attacker can only crack an AES key in a certain amount of time, and the regeneration of the session key will reset that timer. Leading to an attacker's chance of cracking the session key to be extremely low.

Q: Have a good plan to show your design in the report (e.g., you may take the screenshot of your functions and the results, and then explain how your functions achieves the results.)

A: Answered in the previous parts.