STUDENT NAME: OKERE KINGSLEY

student id: 20050453



**KAYCEECHAT APP**

**Introduction**

In this project, I developed a secure chat app where users can send private messages that are encrypted and can only be read by the sender and the recipient. The main goal was to improve the security of the original chat app by adding strong encryption features. I started with an open-source chat app (<https://github.com/kerBiy/simple-chat>) and enhanced it with additional security measures, like message encryption and a more secure login system.

This report explains the improvements I made to the app, the tools I used, the challenges I faced, and ideas for further improvements.

**Technologies Used**

* Frontend: React with Vite, Socket.io client for real-time communication.
* Backend: Node.js, Express, Socket.io.
* Encryption: Node-forge, CryptoJS, Web Crypto API for end-to-end encryption.
* Database: MongoDB with Mongoose ORM for storing user and message data.
* Authentication: JWT (JSON Web Token) with RSA signatures.
* Deployment: Heroku for hosting the app.
* Real**-**timeCommunication: Socket.io for real-time messaging.

**Project Implementation**

1. **User Authentication and Registration**

To enhance the security of the original chat app, I improved the user authentication system to protect against unauthorized access and ensure safe user login. Below is an overview of the improvements made in the authentication process:

**User Registration Process**:

* The user begins by creating an account and providing their username, email, and password.
* To prevent password theft and brute-force attacks, I implemented password hashing using a secure hashing algorithm. This ensures that even if the database is compromised, the actual passwords remain unreadable.
* For enhanced security, I generated an RSA key pair for each user. The RSA key pair consists of a public key (used to encrypt data) and a private key (used to decrypt data). The RSA keys ensure that the user’s sensitive data is protected during communication.
* The user’s details, including the hashed password and RSA key pair, are then saved in a MongoDB database. This allows the system to securely store user credentials and keys for future use.

**User Login Process**:

* Logging in involves providing the user’s username and password.
* The frontend sends this information to the backend for verification. The server checks the password by comparing the provided password with the stored hashed password in the database.
* If the password is valid, the backend generates a JWT (JSON Web Token) for the user. This token contains essential user information (such as the user ID), and it is signed with the RSA private key. The signing ensures that the token is tamper-proof and can only be verified using the corresponding RSA public key.
* The generated token is returned to the frontend, where it is stored in the local storage and used in subsequent requests to authenticate the user.

**JWT Authentication:**

* JWT is used to manage secure user sessions. By signing the token with the user’s private RSA key, we ensure that the token cannot be tampered with, as only the server can generate valid tokens.
* This implementation helps protect against session hijacking and unauthorized access, ensuring that only authenticated users can access protected resources.

2. **End-to-End Encryption**

To further secure user communication, I implemented end-to-end encryption to ensure that messages are private and can only be read by the sender and the intended recipient. Here's an overview of how this encryption process works:

**Encryption Process**:

RSA Key Generation:

* Both the sender and receiver are assigned their own unique RSA public and private key pairs.
* These keys are used for encrypting and decrypting the symmetric AES encryption key.

AES Key Generation:

* When a sender wants to send a message, they generate a unique AES key. This key will be used to encrypt the actual message content.

Message Encryption:

* The sender encrypts the message using the generated AES key. This ensures that the message content is unreadable to anyone who intercepts it.

RSA Encryption of AES Key:

* The AES key itself is then encrypted using the receiver’s RSA public key. This step ensures that only the receiver, who has the corresponding RSA private key, can decrypt the AES key.

Sending Data:

* Both the encrypted message and the encrypted AES key are sent to the receiver.

**Decryption Process:**

RSA Decryption of AES Key:

* Upon receiving the encrypted message and AES key, the receiver uses their RSA private key to decrypt the AES key.

Message Decryption:

* Once the receiver has decrypted the AES key, they use it to decrypt the message content.

This hybrid encryption approach combining RSA and AES offers the best of both worlds:

* RSA provides strong security for key exchange, ensuring that only the recipient can decrypt the AES key.
* AES allows for fast and efficient encryption of the message content, making it suitable for real-time communication.

3. **JWT Authentication Implementation**

The implementation of JWT (JSON Web Token) is crucial to managing secure authentication sessions within the chat app. JWT ensures that users are authenticated each time they make requests to the server. Here’s a breakdown of how JWT was implemented:

**JWT Workflow**:

1. When a user logs in, the backend server generates a JWT token that includes user-specific information, such as the user ID and other relevant data.
2. The server signs the JWT token using the user’s RSA private key. Signing the token ensures that it is secure and tamper-proof. This is important because anyone can potentially create or modify tokens, but only the server can generate valid signed tokens.
3. The token is then sent back to the frontend and is typically stored in the local storage and is used in subsequent requests to authenticate the user.
4. On each subsequent request, the frontend includes the JWT token in the Authorization header of the HTTP request.
5. The backend verifies the JWT token by checking the signature against the RSA public key. If the token is valid and the user is authenticated, the server processes the request.

**4.Deployment on Render**

After completing all the demands of this project on the app, I deployed it on Render, a cloud platform that supports both frontend and backend hosting. I chose Render because it's easy to use, supports automatic deployments from GitHub, and handles environment variables securely. This made my app available online for real users.

**Deployment Steps:**

**1. Code Upload to GitHub**  
I uploaded the full project (frontend and backend) to a GitHub repository to keep track of changes and connect it to Render for deployment.

**2. Link GitHub to Render**  
I created a Render account and linked it to my GitHub. This allowed Render to automatically pull and deploy changes whenever I pushed updates to the repository.

**3. Add Environment Variables**  
To keep sensitive data safe, I added environment variables in Render instead of putting them in the code. These included:

JWT\_SECRET for token-based authentication.

MONGO\_URI for connecting to MongoDB Atlas.

**4. Deploy Backend (Node.js)**  
I set up a WebService on Render for the backend:

Defined the entry point which is index.js.

Made sure it used the dynamic port (process.env.PORT).

Added a start script in package.json.

Render automatically installed all needed packages.

**5. Deploy Frontend (React)**  
I built the frontend with yarn build or npm run build to create production files. Then:

Set up a StaticSite service on Render.

Used build/ as the publish folder.

Render hosted the static files automatically.

**6.Domain and SSL**  
Render gave my app a free subdomain https://kayceechat-frontend.onrender.com and enabled HTTPS by default for secure access.

**7. Testing**

I tested the live app to confirm:

API routes worked properly.

The frontend connected to the backend.

Real-time messaging using Socket.IO worked.

JWT authentication worked across sessions.

**Potential Vulnerabilities**

1. **Key Management Issues**:
   * Currently, RSA private keys are stored in the user's local storage, which poses a security risk. If an attacker gains access to the user's device, they could potentially steal these keys.
2. **Encryption Issues**:
   * I used two different libraries for encryption (Node-forge, CryptoJS), which can create inconsistencies or potential vulnerabilities.
3. **JWT Storage**:
   * JWT tokens are currently stored in localStorage, which makes them vulnerable to attacks.
4. **Database Security**:
   * Encrypted messages are stored in a way that makes it difficult to search or index data efficiently. This is both good and bad as it secured and encrypted but also makes it hard to search or query the database.

**Conclusion**

I successfully developed a secure chat app by enhancing an open-source project with stronger encryption and better login features. The app now provides end-to-end encryption for messages, making sure that only the sender and receiver can read the messages. I also integrated JWT for secure authentication, improving the security of user sessions.

While the app is secure, there are still areas to improve, such as better management of encryption keys, safer storage of JWT tokens, and optimizing the database for encrypted message storage.