Module: Computer Systems Security (B9IS103)

End-to-End Encrypted Chat and File Transfer

Student ID	**Name**
20041975	NusratUllah
20036759	Shivam Singh
20040108	Vikhyat Salian

Project Link: `https://chatappui-gs4s.onrender.com`

Contents

List of Code Snippets

1 Introduction

In this project, we have implemented a secure communication system consisting of two separate components, an end-to-end encrypted (E2EE) chat room and a peer-to-peer (P2P) file-sharing module.Chat room features a real-time message exchange with strong encryption, ensuring that conversations remain private and protected from unauthorized access. We have used SignalR, a real-time communication library in ASP.NET Core, which facilitates low-latency, bidirectional communication between clients and the server. In addition, the file sharing feature allows users to securely transfer files between each other directly on their devices, using WebRTC protocol. This component also uses SignalR, specifically as the signaling mechanism necessary to establish a WebRTC connection. We have used the Web Crypto API, cryptographic library present in modern browsers to handle encryption and decryption of messages and files. Users authenticate through GitHub OAuth, which not only verifies their identity but also allows the application to reference their public GitHub profile as a trusted source for their encryption public key. After logging in, users are prompted to add their private RSA key to the browser and provide a link to their public key(either as a file in a public GitHub repository or a GitHub Gist). This design enables secure, identity-verified communication without requiring users to pre-share keys through an external channel.
All together, these elements create a secure method for private messaging and exchanging files.
A high level user flow would be as follows:

1. Sign in with GitHub : The user logs in using GitHub OAuth to authenticate their identity.

2. Provide Public Key Location: The user shares a GitHub repository name and file path, or a Gist URL, where their RSA public key is hosted.

3. Upload Private Key: The user uploads their RSA private key to the browser for local decryption.

4. Key Pair Verification: The system verifies the uploaded private key matches the linked public key before allowing secure communication.

5. Choose an Action: The user can either, enter a chatroom or start a file transfer.

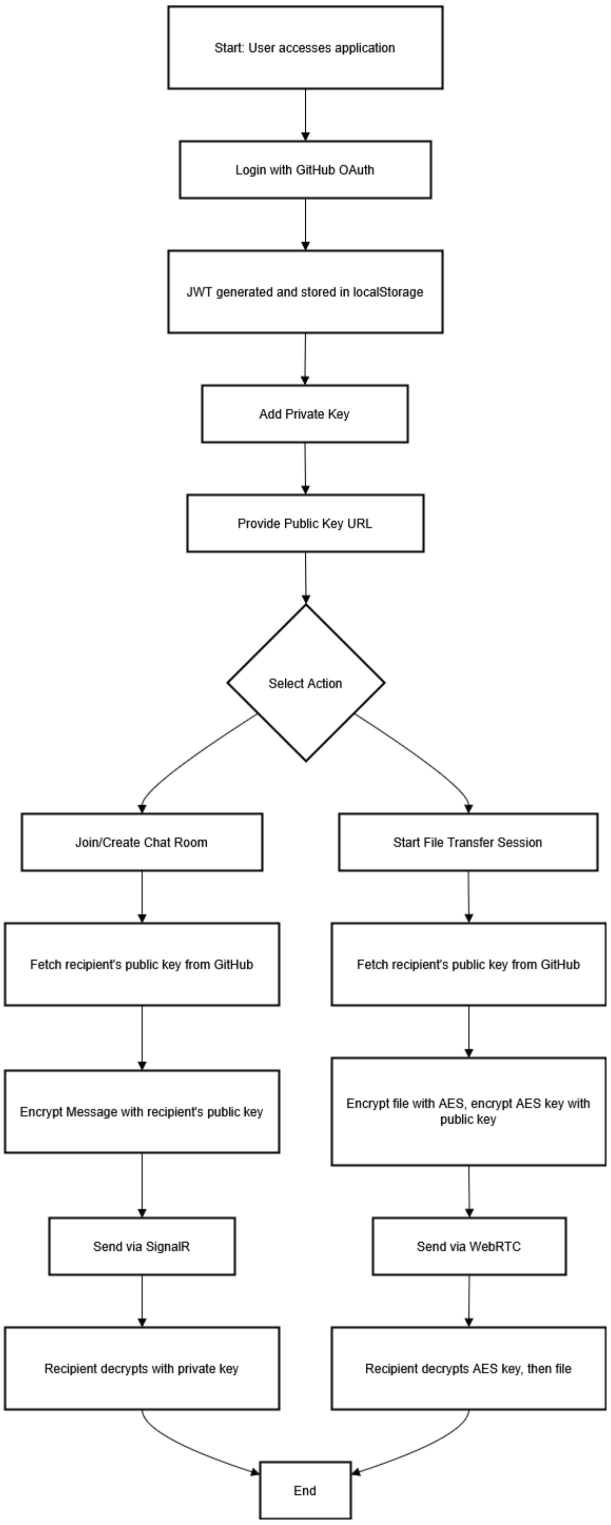
Start: User accesses application
Login with GitHub OAuth
JWT generated and stored in localStorage
Add Private Key
Provide Public Key URL
Select Action
Join/Create Chat Room
Start File Transfer Session
Fetch recipient's public key from GitHub
Fetch recipient's public key from GitHub
Encrypt Message with recipient's public key
Encrypt file with AES, encrypt AES key with public key
Send via SignalR
Send via WebRTC
Recipient decrypts with private key
Recipient decrypts AES key, then file
End

Figure 1.1: Overview of the system.

2 System Architecture

2.1 Frontend

The frontend is built on Angular and styling by Tailwind. It manages user interactions, authentication flow, message rendering, file selection and cryptographic operations.The initiation and setup of SignalR server and WebRTC connection is established through the Angular Code. The app uses the browser-native Web Crypto API to encrypt and decrypt messages and files client-side. After successful GitHub login, the JWT token issued by the backend is stored in localStorage to persist the authenticated session across browser reloads. Angular route guards are implemented to restrict access to protected routes('chat', 'file-share'), ensuring that only authenticated users and users who have added their keys can access these pages. For the public key, user can either share the repository with the public key file name or share the gist id. Private keys are loaded in browser memory and never sent to the server.

2.2 Backend

The backend is built using ASP.NET Core Web API and hosts the SignalR hubs used for both chat and file transfer signalling. It also defines endpoints for user log in and authentication. MongoDB is used to store user-related data such as GitHub userIds and email. After the user successfully authenticates through GitHub, the server generates and sends a JSON Web Token (JWT) to the same user to verify all future requests for authenticating the user to the SignalR hubs and API endpoints. Configured middleware checks the JWT over all real-time connections and API calls, restricting system access to valid authenticated users. Users authenticate through GitHub OAuth, it offers a simple, secure way of authenticating the identity.

2.2.1 Chat Messaging (SignalR)

The chat system leverages SignalR for real-time communication, ensuring low-latency delivery of messages between participants. Messages are encrypted on the client using the recipient's RSA public key before being transmitted. The encrypted payload is sent through the SignalR hub, and only the recipient can decrypt it using their private key, which is securely held in memory. This ensures that even the server facilitating the communication cannot access the message contents.

2.2.2 File Transfer (WebRTC and SignalR)

File sharing is implemented using the WebRTC protocol, which allows users to establish direct peer-to-peer connections. To initiate a file transfer session, the application first establishes a SignalR connection, which is used as a signaling channel. Once connected, a session is created and both users join it to begin the peer negotiation process. WebRTC peer connections are then instantiated on each client. Using SignalR, the application performs an offer/answer exchange to negotiate session parameters, followed by the exchange of ICE candidates required for NAT traversal. After a successful connection is established, a dedicated WebRTC data channel is created for transferring files. The file is split into smaller chunks, which are then sent sequentially over this encrypted data channel. Upon completion of the transfer, the peer connection is closed.

2.2.3 Deployment

The ASP.NET backend and Angular frontend have been containerised using Docker. We define Dockerfile in both the projects, which makes it easy to configure and simplifies dependency management. Containers are built and tested locally, and they are then pushed to the cloud for deployment.The app is deployed in Render a cloud platform that supports container-based deployement for hosting. Both services are hosted individually using the free tier of Render. The storage requirement for the system is minimal, since we're only storing the user data(email, github username), we've utilised the MongoDB Atlas free tier.

2.3 Encryption

The encryption logic for both chat messages and file transfers is implemented entirely on the client side using the Web Crypto API, a modern cryptographic library built into web browsers. This makes sure that all sensitive information is encrypted before being sent and decrypted only on the destination device, allowing for true end-to-end encryption (E2EE). To facilitate secure communication between users who have not previously exchanged keys, the application uses GitHub as both the authentication and identity. Users log in using GitHub OAuth, which provides their GitHub username and email for session identification. After logging in, each user is required to add their private RSA key to the browser and share the location of their public key. This public key can be hosted either as a file within a public GitHub repository or as a Gist. The implementation of importing the keys is detailed in *Appendix A, Listing 1 and Listing 2.* During encryption, peers retrieve the public key from the provided GitHub source to perform secure RSA-OAEP encryption, allowing secure message and file exchanges without manual key distribution. Through the RSA-OAEP algorithm, the chat messages are encrypted with the RSA public key of the recipient. Encrypted messages are sent through SignalR and decrypted by the recipient using their private RSA key stored in-memory. Storing the private key in memory, instead of local storage or session storage makes it immune to cross-site scripting attacks.This approach guarantees that only the intended recipient can access the message content.
While RSA-OAEP is sufficient for encrypting short messages directly, it is not suitable for large payloads such as files due to its size limitations. To address this, the system uses a hybrid encryption model for file transfers. AES-GCM is employed for encrypting the file content due to its efficiency and support for authenticated encryption. The symmetric AES key used for encryption is then encrypted using RSA-OAEP with the recipient's public key. This approach combines the scalability of symmetric encryption with the key security of asymmetric encryption.
To ensure the correctness of cryptographic operations and prevent key mismatch errors, the system includes a client-side key pair verification step. After a user adds their private key, the application verifies it against the associated public key (fetched from GitHub) by encrypting a short test string with the public key and attempting to decrypt it with the uploaded private key. This ensures the two keys form a valid pair before allowing any encrypted communication. This mechanism helps detect misconfigurations or tampered keys early in the process, reducing the risk of runtime encryption errors and improving overall trust in the system's key management. The implementation of this verification step is detailed in *Appendix B, Listing 3.*

3 Requirements

3.0.1 Functional Requirements

- The system shall require that users have a publicly visible email address associated with their GitHub account in order to successfully complete the authentication and identity verification process.
- The system shall allow users to authenticate using their GitHub accounts via OAuth.
- The system shall generate a JWT upon successful login and use it to authorize all API and SignalR requests.
- The system shall persist authenticated sessions using JWT stored in ‘localStorage‘.
- The system shall enable users to add their private key and public key.
- The system shall provide a real-time chat interface for users to exchange messages.
- The system shall allow peer-to-peer file sharing using WebRTC.
- The system shall retrieve the public key of the participating users in chat or file transfer.

3.0.2 Non-Functional Requirements

- The system shall maintain low latency for real-time chat and file transfer.
- The application shall be accessible through modern web browsers without additional plugins or extensions.
- The frontend and backend shall be containerized using Docker for consistent deployment.
- The system shall be hosted on a cloud platform (Render) and be accessible over HTTPS.
- The system shall provide a responsive UI using Angular and Tailwind CSS.

3.0.3 Security Requirements

- The system shall implement end-to-end encryption (E2EE) for all messages and file transfers.
- The public keys used for encryption shall be verifiable through the user’s GitHub profile to ensure authenticity.
- The system shall verify the integrity of uploaded private keys by checking that they correctly match the corresponding public key. This verification ensures that only valid key pairs are used for encryption and decryption operations.
- Messages shall be encrypted with the recipient’s public key using RSA-OAEP and decrypted using their private key.

- Files shall be encrypted with a unique AES-GCM key per file, with the key itself encrypted using RSA-OAEP.
- The system shall never transmit or store private keys on the server.
- All SignalR and API communications shall require a valid JWT for authorization.

4 Implementation

The encryption and decryption processes are implemented on the client side using the Web Crypto API, ensuring that all sensitive data is encrypted before transmission and decrypted only by the intended recipient. Two different approaches are used depending on the type of data being encrypted, RSA-OAEP for messages and a hybrid RSA-OAEP along with AES-GCM scheme for files.

4.1 Message Encryption with RSA-OAEP

For chat messages, the system uses RSA-OAEP to directly encrypt the message content. This method is well-suited for short text-based payloads and ensures that the message can only be decrypted by the recipient, who holds the corresponding private key. The public key is imported in spki format and used to encrypt the UTF-8 encoded message string. On the recipient's side, the encrypted message is decrypted using their private RSA key (imported in pkcs8 format).

```
async encryptMessage(publicKey: CryptoKey, message: string){
    const encryptedData = await crypto.subtle.encrypt(
      { name: 'RSA-OAEP' },
      publicKey,
      this.encoder.encode(message)
    );
    return this.arrayBufferToBase64(encryptedData);
  }
```

Code Snippet 1: Method to encrypt message.

```
async decryptMessageWithInMemoryPrimaryKey(
  encryptedMessage: string
){
  const binaryData = this.base64ToArrayBuffer(encryptedMessage);
  const decryptedData = await crypto.subtle.decrypt(
    { name: 'RSA-OAEP' },
    this.privateKey()!,
    binaryData
  );
  return this.decoder.decode(decryptedData);
}
```

Code Snippet 2: Method to decrypt message.

4.2 File Encryption Process

When encrypting a file, the client generates a random AES-256 key and a random 12-byte IV. The file is encrypted using AES-GCM with the generated key and IV. The AES key is then encrypted using the recipient's RSA public key. The final encrypted payload includes the encrypted file data, encrypted AES key, and the IV used for AES-GCM. The file is passed to the encrypt file function as an ArrayBuffer.

```
async encryptFile(
  publicKey: CryptoKey,
  fileData: ArrayBuffer
) {
// AES key generation
  const aesKey = await crypto.subtle.generateKey(
    { name: 'AES-GCM', length: 256 },
    true,
    ['encrypt', 'decrypt']
  );

  const iv = crypto.getRandomValues(new Uint8Array(12));

  // Encryptin the file:
  const encryptedFile = await crypto.subtle.encrypt(
    { name: 'AES-GCM', iv },
    aesKey,
    fileData
  );

  const exportedKey = await crypto.subtle.exportKey('raw', aesKey);

  // Encrypt the AES key with the RSA public key
  const encryptedKey = await crypto.subtle.encrypt(
    { name: 'RSA-OAEP' },
    publicKey,
    exportedKey
  );

  return {
    encryptedKey: this.arrayBufferToBase64(encryptedKey),
    encryptedFile,
    iv,
  };
}
```

Code Snippet 3: Method to encrypt file.

```
async decryptFile(
  privateKey: CryptoKey,
  encryptedKeyBase64: string,
  encryptedData: ArrayBuffer,
  ivArray: number[]
): Promise<ArrayBuffer> {
  try {
    const encryptedKey = this.base64ToArrayBuffer(encryptedKeyBase64);

    // Convert IV array back to Uint8Array
    const iv = new Uint8Array(ivArray);

    // Decrypt the AES key using private key
    const decryptedKeyBuffer = await crypto.subtle.decrypt(
      { name: 'RSA-OAEP' },
      privateKey,
      encryptedKey
    );

    const aesKey = await crypto.subtle.importKey(
      'raw',
      decryptedKeyBuffer,
      { name: 'AES-GCM', length: 256 },
      false,
      ['decrypt']
    );

    const decryptedData = await crypto.subtle.decrypt(
      { name: 'AES-GCM', iv },
      aesKey,
      encryptedData
    );

    return decryptedData;
  } catch (error) {
    console.error('Decryption error:', error);
    throw error;
  }
}
```

Code Snippet 4: Method to decrypt file.

Appendix

A Github Repositories

- Frontend : `https://github.com/shivam-ssingh/ChatAppUI`
- Backend : `https://github.com/shivam-ssingh/ChatAPI`

B Code Listing

Listing 1

```
async importPrivateKey(pem: string): Promise<CryptoKey> {
  const binaryDer = this.base64ToArrayBuffer(
    pem.replace(/-----.*?-----/g, '')
  );
  return await crypto.subtle.importKey(
    'pkcs8',
    binaryDer,
    { name: 'RSA-OAEP', hash: 'SHA-256' },
    true,
    ['decrypt']
  );
}
```

Code Snippet 5: Method to import Private key from added file.

Listing 2

```
async importPublicKey(pem: string): Promise<CryptoKey> {
  const binaryDer = this.base64ToArrayBuffer(
    pem.replace(/-----.*?-----/g, '')
  );
  return await crypto.subtle.importKey(
    'spki',
    binaryDer,
    { name: 'RSA-OAEP', hash: 'SHA-256' },
    true,
    ['encrypt']
  );
}
```

Code Snippet 6: Method to import Public key.

Listing 3

```
async verifyKeyPair() {
  try {
    const testString = 'Test message';
    const importedPublicKey = await this.importPublicKey(
      localStorage.getItem(StorageKeys.PUBLICKEY)!
    );
    const encryptedMessage = await this.encryptMessage(
      importedPublicKey,
      testString
    );
    const decryptedMessage = await
    this.decryptMessageWithInMemoryPrimaryKey(
      encryptedMessage
    );
    return testString === decryptedMessage;
  } catch (error) {
    console.log(error);
    return false;
  }
}
```

Code Snippet 7: Method to verify if the user added key pair is valid.