

|  |  |
| --- | --- |
| Module Title: | Computer Systems Security 2025 |
| Module Code: | B9IS103 |
| Module Lecturer: | Paul Laird |
| Assignment Deliverable: | Secure Chat Application |
| Individual/Group: | Group of 3 |
| Student ID / Names: | **20053847, SRI SAI HARSHA NANDAN VEGIREDDI**  **20029994, VENKATA SAI THIRUVAKADU**  **20026584 , VENKATA KRISHNA SAI UDAY PALAVADI** |

Contents

[1. Overview 3](#_Toc193999783)

[2. Technology Stack 3](#_Toc193999789)

[3. Architectural Diagram 5](#_Toc193999791)

[4. Functional Requirements 6](#_Toc193999793)

[5. Non- Functional Requirements 7](#_Toc193999814)

[6. Security Requirements 7](#_Toc193999815)

[7. User- Flow 8](#_Toc193999816)

[8. Implementations Signup Page 9](#_Toc193999846)

[10. Limitations 12](#_Toc193999856)

[11. Conclusion 13](#_Toc193999857)

[12. Links 13](#_Toc193999863)

# Overview

# Secure Chat App is a fully encrypted real-time messaging application that is designed to provide secure chat sessions between authenticated users. With Flask (Python) used in the backend, React for responsiveness on the front end, Firebase Authentication to securely authenticate users, and either Web Socket or Firestore's real-time capability, the app provides a modern, secure chat experience.

# Essentially, the system employs end-to-end encryption that relies on a hybrid cryptography model—AES (Advanced Encryption Standard) to encrypt the actual text of the messages due to its speed and efficiency, and RSA encryption for sending the AES keys securely over the recipient's public key. This manner allows only the target recipient—who holds the corresponding private RSA key—access to decrypt and read the messages.

# Every user is assigned a special RSA key pair when they sign up. The public key is securely stored in the database but the private key is shown once and securely stored in the browser's Indexed DB after being password-encrypted at the user's option. Messages are never kept unencrypted and are encrypted on the device of the sender before leaving. The AES key and IV are decrypted by the receiver using their RSA private key to get the original message, all locally on the client.

# Implementation of JWT (JSON Web Tokens) ensures that the secured routes can be accessed and interacted with exclusively by authenticated users and server APIs. The application also includes a clean user interface with Google Sign-In functionality, dynamic user lists, and time-stamped chat windows to ensure a secure and faultless process of communication.

# This encryption-first and this architecture render the Secure Chat App extremely well-suited for environments where privacy, integrity, and identity authentication are of utmost importance.

# Technology Stack

# Frontend

* **React.js** – For building the responsive and interactive user interface.
* **React-Bootstrap** – For clean, modern UI components.
* **react-oauth/google** – For integrating Google Sign-In (OAuth 2.0).
* **Indexed DB** – To securely store the encrypted private RSA key on the user's browser.
* **JWT (JSON Web Token)** – For storing authentication tokens client-side (in local storage).

**Backend**

* **Flask (Python)** – Lightweight backend framework for handling APIs.
* **Flask-CORS** – To allow communication between frontend and backend (CORS headers).
* **Flask-Bcrypt** – For securely hashing passwords (if needed for fallback login).
* **JWT (PyJWT)** – For session-based token authentication.
* **Cryptography (Python)** – To generate RSA key pairs, perform RSA encryption/decryption, and AES encryption/decryption.

**Security**

* **RSA Encryption (2048-bit)** – For encrypting the AES key using the receiver’s public key.
* **AES Encryption (256-bit, CBC Mode)** – For encrypting chat messages.
* **JWT Tokens** – For stateless authentication of API access.

**Database**

**Firebase Firestore** – For storing:

* Encrypted messages
* AES key (encrypted with RSA)
* IVs (Initialization Vectors)
* User metadata (email, public key, etc.)

**Authentication**:

**Firebase Authentication with Google OAuth** – For validating user identity during signup/login.

**Deployment**

**Render** – For deploying frontend or backend (server).

# Architectural Diagram

# 

# Functional Requirements

# User Registration/Login using Google OAuth

# A user has to register or login using his/her Google account. It securely authenticates him/her and only allows genuine emails to be used.

# RSA Public/Private key pair generation on signup

# At user signup, an encryption key pair with public key and private key is generated using RSA encryption. It is the foundation for secure communication.

# Public key stored securely in database

# Public key is kept in Firestore and others use it to encrypt messages for the user. The key may be shared without compromising security.

# Private key shown only once for secure storage

# Private key is secret and is shown only to the user once (after signup) to securely store it within their browser (Indexed DB) for decryption in the future.

# JWT Token-based session authentication

# On a user's login, a JSON Web Token (JWT) is delivered. This token is stored in the browser and reused for all subsequent API requests to authenticate the user.

# AES-encrypted message sending using receiver's public key

# The messages are AES-encrypted (symmetric algorithm for high performance) and then the AES key is encrypted with RSA using the receiver's public RSA key.

# AES key and IV encrypted using RSA

# To permit the decryption by only the receiver, the AES key is encrypted with their public key and the IV to be used during decryption.

# Encrypted messages stored in Firestore

# The messages are stored in Firebase Firestore in an encrypted manner, thereby even when the database is compromised, the content remains indecipherable.

# The receiver decrypts the messages using their private key, AES key, and IV

# Upon receipt, the app decrypts the AES key with the user's private key and then decrypts the actual message using that AES key and IV.

# Chat page for sending and receiving encrypted messages

# Chat UI allows users to select a contact, view history of messages, and send new encrypted messages securely.

# Non- Functional Requirements

1. **Performance**

App should load within 2 seconds under normal conditions.Message encryption and decryption must happen in real-time.

1. **Scalability**

Flask backend is modular and scalable. Firebase Firestore can scale automatically with user/data growth.

1. **Usability**

Clean and simple UI using React-Bootstrap. Google Sign-In removes need for passwords. Private key is shown only once to reduce confusion.

1. **Maintainability**

Code is structured cleanly (frontend, backend, database separation).RESTful APIs make it easy to maintain and debug. Easily extendable to add new features in future.

1. **Reliability**

JWT-based session handling ensures secure and consistent access.Clear error messages shown for user actions (e.g., failed login, invalid token).

1. **Availability**

Firebase handles 24/7 availability of authentication and database.Flask backend is deployable on cloud platforms like Render.

# Security Requirements

1. **End-to-End Encryption**

Messages are encrypted on the sender’s device using AES.The AES key itself is encrypted using the receiver’s public RSA key.Only the receiver can decrypt it using their private RSA key.

1. **Google OAuth for Authentication**

Signup and login only allowed through Google accounts.Prevents fake or unauthorized users.Google ID token is verified before access is granted.

1. **JWT (JSON Web Token)**

JWT is issued after login and used for all protected API requests.Each JWT contains the user’s email and expiration timestamp.Expired or tampered tokens are rejected on the backend.

1. **Private Key Handling**

Private key is generated at signup and shown only once.Never stored on the backend or frontend.Must be securely stored by the user (e.g., password manager).

1. **Public Key Storage**

Stored in Firebase Firestore with the associated user email.Only public key is shared – safe to store and use.

1. **CORS Protection**

Flask backend is CORS-enabled only for requests from trusted frontend domain.

1. **No Hardcoded Keys**

No AES or RSA keys are stored or hardcoded in the code.All keys are generated per user/session.

1. **Secure Communication**

All API calls are made over HTTP (should be HTTPS in production).Sensitive data (like tokens or keys) is never exposed in logs or responses

# User- Flow

# User accesses the App

# Lands on Login Page.

# Google only sign in option.

# Google Signup / Sign In

# If first time for the user:

# Google email is verified.

# public/private key pair is established.

# Public key is stored in Firebase.

# Private key is shown once so the user can safely copy.

# If user already exists:

# JWT token is created to authenticate session.

# Redirected to Chat Page

# User is given the chat interface.

# Can search for other users by email to start a conversation.

# Sending a message

# User types in a message and selects a recipient.

# New AES key and IV are generated.

# Message is encrypted using AES.

# AES key is encrypted using receiver's public RSA key.

# Encrypted message, encrypted AES key, and IV are stored in Firestore.

# Receiving a message

# Receiver logs in and joins chat.

# They copy their private key securely in the UI.

# AES key decrypted using private RSA key.

# Message decrypted using AES key and IV.

# Decrypted messages displayed in UI.

# Logout or Token Expiry

# JWT token expires in ~1 hour.

# User needs to re-authenticate using Google.

# Implementations Signup Page

# 

# Login Page

# 

# Home Page

# 

# Profile Page

# 

# Challenges faced

# Major challenge faced while deployment has taken place as it sometimes gives connectivity issues with the database while making the deployment on Render. Also faced quite few challenges alongside this while performing the tests and these are the issues faced:

* **Private Key Management**
  + Private key shown only once for security.
  + User responsible for securely storing it.
* **Key Encryption Flow**
  + AES key encrypted with RSA and decrypted securely.
  + Matching encryption/decryption formats and padding (OAEP) was complex.
* **Firebase Integration**
  + Difficulties in securely structuring user, message, and metadata.
  + Firestore rules needed to restrict access and protect encrypted data.
* **JWT Authentication**
  + Ensured JWT validation before accessing sensitive endpoints.
  + Maintained user session securely across all routes.
* **WebSocket (Optional Attempted)**
  + Real-time chat attempted using WebSocket.
  + Eventually used Firebase polling for consistent updates.
* **UI Simplicity with Security**
  + Designed user-friendly interface with secure habits like private key copy prompts.
  + Balanced usability with encryption flow transparency.
* **CORS & OAuth Issues**
  + Faced CORS errors due to preflight failures and missing headers.
  + Google OAuth errors like invalid tool required accurate project and credential setup.

# Limitations

**Security Limitations**

* **Private Key Loss**: If the user loses the private key after signup, messages cannot be decrypted, and there's no way to recover them.
* **AES Key Handling**: AES key and IV are currently shared manually (through prompts), which is not user-friendly and could be insecure in real-world usage.
* **No Forward Secrecy**: Since RSA keys are static per user, compromise of one key exposes all messages for that user.
* **Token Expiry**: JWT tokens expire in 1 hour, but the system lacks proper refresh token or re-authentication mechanism.

**Feature Limitations**

* **No Group Chat Support**: Only one-to-one messaging is supported.
* **No Attachments**: The system only supports text messages, no files or media.
* **No Notification System**: Users don’t get notified when new messages arrive.
* **No Real-Time Updates**: Web Sockets are not fully integrated—message polling is manual.

# Conclusion

# This report presents a detailed analysis of the Secure Chat Application, describing all the key points from architecture and functionality to encryption flows and usability. It highlights the main features, including Google-based authentication, RSA and AES end-to-end encryption, JWT-based secure sessions, and Firebase for structured storage and scalability.

# The technology stack—built upon frameworks like Flask (Python) for backend, React.js for frontend, and Firebase Firestore for message storage—constitutes a lean yet secure foundation for a real-time messaging system. The use of public-private key cryptography (RSA) renders the messages readable by the intended recipient alone who is able to decrypt the messages, while AES encryption offers fast and secure message confidentiality.

# By walking through the user experience, from sign-up to decryption of messages, and addressing the challenges like secure key handling, CORS/OAuth setup, and Firebase integration, the report demonstrates technical sophistication along with thoughtful design decisions.

# Understanding how these elements work together allows developers, users, and reviewers to see not only how to use the app in a way that is secure, but where potential vulnerabilities might lie. Whether adding real-time updates, streamlining UI, or automating key exchange, this documentation offers a strong foundation for future development, improvement, and scaling.

# Concisely, this report not only documents the usage but enables additional comprehension, enabling developers and users to build more secure, more usable, and more scalable real-time communication applications.

# Links

|  |  |
| --- | --- |
| Github | <https://github.com/HarshaNandan1/SecureChatApp> |
| Deployment(Render) | <https://securechatapplication.onrender.com> |

# 