Secure Chat — Project Report

Abstract

This report documents the design and implementation of Secure Chat, a compact browser-based one-to-one messaging prototype providing end-to-end encryption (E2EE). The system demonstrates a hybrid cryptographic approach: RSA-OAEP (asymmetric) for secure exchange of ephemeral session keys and AES-GCM (symmetric) for authenticated encryption of message payloads. The server functions strictly as a relay and public-key registry; all cryptographic operations and plaintext handling occur within clients' browsers. The objective is to provide an educational, runnable demo that illustrates core secure messaging concepts without external dependencies.

Introduction

Secure Chat was developed as a learning-focused implementation showing how modern secure messaging can be constructed from widely-understood primitives. The application leverages the Web Crypto API in the browser to generate and manage keys, ensuring private keys and plaintext remain on the client side. The backend is intentionally minimal—an Express + Socket.IO server that serves static assets, stores public keys in memory, and relays ciphertext. This separation reduces trust requirements on the server and allows students and researchers to experiment with E2EE flows.

Objectives

• Enable low-latency, real-time messaging using WebSockets (Socket.IO).
• Guarantee that plaintext messages are accessible only to sender and intended recipient (end-to-end confidentiality).
• Use RSA for secure exchange of ephemeral AES session keys and AES-GCM for message confidentiality and integrity.
• Ensure the server acts as a non-trusting relay that never handles private keys or plaintext.
• Support both local testing and remote testing via tunneling (for example, ngrok).

Tools Used

Implementation uses Node.js, Express, and Socket.IO for the backend. The frontend is plain HTML and JavaScript, relying on the Web Crypto API to perform RSA-OAEP key generation and AES-GCM encryption/decryption in-browser. Ngrok is recommended for exposing a local server during remote testing. Development assumed modern browsers (Chrome, Firefox) that support the SubtleCrypto interface.

Steps Involved in Building the Project

1. Project scaffolding: Initialize a Node.js project, install Express and Socket.IO, and create a static folder that will serve the frontend files (index.html and chat.js). Configure the server to serve static assets and accept socket connections. **2. Client-side crypto:** Implement RSA-OAEP key generation in the browser upon client connection. Export the public key in SPKI/base64 format and send it to the server as part of a `register` event. Keep the private key entirely in browser memory. **3. Key exchange protocol:** When a user initiates a chat, generate an ephemeral AES-GCM 256-bit session key, export the raw key bytes, encrypt those bytes with the recipient's RSA public key (RSA-OAEP), and send the resulting envelope to the recipient via the server. The recipient decrypts the envelope locally using their RSA private key and imports the AES key for subsequent message encryption/decryption. **4. Messaging:** For each chat message, the sender uses AES-GCM with a fresh 96-bit IV to encrypt the plaintext, which yields ciphertext and an authentication tag. The sender transmits the IV and ciphertext object; the server relays this opaque blob to the recipient, who decrypts and verifies authenticity client-side. **5. Testing and validation:** Validate the design by running two browser clients (local or via ngrok). Use developer tools to inspect network traffic and confirm that only ciphertext is visible on the wire. Verify console logs for key exchange events and successful decryption.

Security Analysis

The prototype defends against passive eavesdroppers by encrypting all message payloads with AES-GCM. AES-GCM also provides integrity and authenticity detection for ciphertext. Storing only public keys on the server reduces impact of server compromise: without clients' private keys, an attacker cannot decrypt captured ciphertext. However, several important security limitations remain: there is no authenticated identity binding (usernames can be impersonated), no forward secrecy is provided by the RSA-based exchange, and public-key distribution trusts the server (susceptible to key substitution). Additionally, the application does not provide offline message delivery or long-term key management; these are areas for future work.

Conclusion

Secure Chat is a concise educational prototype that successfully showcases client-side end-to-end encryption using a hybrid cryptographic approach. It demonstrates the practicality of keeping private keys and plaintext solely within client environments while delegating routing and public-key distribution to a simple backend. To transition to a production-grade system would require implementing authentication, key verification/fingerprinting, forward secrecy, offline delivery mechanisms, and robust server hardening. The current implementation provides a clear and extensible foundation for further study and iterative improvements.

Generated Secure Chat Project Report — 2 pages