P2P E2EE Messaging Application

Introduction & usage

This peer-to-peer end-to-end encrypted messaging application demonstrates how modern web browsers can establish a secure, direct communication channel without routing message contents through a central server. Users begin by signing up for an account, which generates a unique elliptic-curve key pair in their browser. They then log in and receive a time-limited JWT (JSON Web Token) from the Authentication Server. Armed with this token, a user connects to the Signaling Server over WebSocket; the server's sole role is to relay WebRTC handshake messages (offer/answer and ICE candidates) to a chosen peer.

Once both browsers have exchanged signaling data, they open a native RTCDataChannel and perform an Elliptic-Curve Diffie-Hellman (ECDH) key derivation to agree on a shared AES-GCM key. From that moment on, all text messages are encrypted and decrypted locally by each client—no plaintext ever traverses any server. The Signaling and Authentication Servers remain idle during actual chat, ensuring true end-to-end confidentiality.

How to Use the Application

1. **Sign Up**
    - Open the browser at the application's URL and complete the **Sign Up** form by choosing a username and password.
    - The browser will generate your ECDH key pair, send your public key to the server, and securely store your private key in IndexedDB.
2. **Log In**
    - Enter the same credentials in the **Login** form to receive your JWT.
    - Successful login will reveal a **Connect** button.
3. **Connect to Signaling**
    - Click **Connect** to establish a WebSocket connection with the Signaling Server.
4. **Start a Call**
    - In the **Call** section, enter the username of the peer you wish to chat with and click **Start Call**.
    - Behind the scenes, your browser fetches your peer's public key, derives a shared AES-GCM key, creates the WebRTC PeerConnection, and sends an SDP offer.
5. **Encrypted Chat**
    - Once the DataChannel opens, the chat interface appears.

- Type your message and click **Send**; the message is encrypted client-side, transmitted directly to your peer, and decrypted automatically upon receipt.

Security choices.

Throughout this project, industry-standard cryptographic primitives and secure coding practices were deliberately selected to ensure robust end-to-end confidentiality, integrity, and authentication. First, each user is assigned a long-term public-key identity: upon signup, the browser generates an Elliptic-Curve Diffie-Hellman (ECDH) key pair on the P-256 curve, exports the public component to the server for storage, and retains the private component securely in IndexedDB. This approach not only anchors each user's identity in a scalable, cryptographically verifiable way, but also eliminates reliance on server-side session secrets when establishing peer-to-peer keys.

For password authentication, bcrypt was chosen with a work factor of 12—above the OWASP-recommended minimum of 10—to hash credentials before persisting them in SQLite. Bcrypt's built-in salting mechanism removes the need for external salt management and resists modern GPU-accelerated cracking attempts. All inter-service traffic between clients and the Authentication or Signaling servers runs over HTTPS/WSS; in development a self-signed certificate is used to simulate real TLS, but switching to a certificate from a recognized certificate authority (for example via Let's Encrypt) would require only minimal configuration changes.

Once peers have exchanged WebRTC signaling messages via the Signaling Server, they perform a one-time ECDH key agreement to derive a shared AES-GCM symmetric key. AES in Galois/Counter Mode (GCM) is widely regarded as the gold standard for authenticated encryption: it provides both confidentiality and integrity checks in one operation and is highly efficient for browser-based use. Each message uses a unique 96-bit initialization vector (IV) generated by the Web Crypto API's getRandomValues call, ensuring nonce uniqueness and safeguarding against replay or reuse attacks. All cryptographic operations—ECDH key generation, key import/export, symmetric encryption, and decryption—leverage the Web Crypto API directly, avoiding third-party JavaScript libraries and reducing supply-chain risks.

Finally, by deriving a new AES-GCM key at each WebRTC session—rather than reusing a static shared key—session-level forward secrecy is provided, so that compromise of one session's keys does not expose past or future sessions. Combined, these decisions create a tightly integrated security model in which users' private keys never leave their browsers, passwords are stored safely, transport layers are encrypted, and message contents remain opaque to any intermediary, including our own signaling infrastructure.

Testing

Functional testing was performed manually to verify that user signup, login, signaling, and message exchange all operated correctly and that no plaintext appeared in network traces. During these tests, browser developer tools were used to inspect WebSocket frames and confirm that all chat payloads were encrypted ciphertext. In parallel, the DevSecOps pipeline originally created in Exercise 6 was adapted to this project with minor adjustments—namely updating repository URLs, adding OWASP ZAP checks for WebSocket endpoints.

The static-analysis stage (Semgrep) flagged no critical or high-severity patterns. Dependency scanning via npm audit and Trivy (file-system) uncovered no vulnerabilities. Dynamic application security testing (DAST) with OWASP ZAP initially reported medium-severity alerts related to missing Content-Security-Policy headers on the JSON-only endpoints; these were determined not to pose a practical risk, since the servers do not serve browser-rendered HTML. A few informational alerts surfaced around overly verbose code comments, which were reviewed and deemed acceptable within this educational context. No critical or high-impact vulnerabilities were discovered, demonstrating that both the manual and automated testing processes affirmed the intended security posture.

Future improvements

Several enhancements are planned to extend the security and functionality of this P2P messaging application. First, message-level forward secrecy should be implemented by deriving a fresh ephemeral ECDH key pair for each message—ideally via a Double Ratchet algorithm—to ensure that even if a session key is compromised, past and future messages remain secure. Second, functional end-to-end tests should be integrated into the DevSecOps pipeline; for example, automated Puppeteer or Playwright scripts can verify that two headless browsers can complete a secure signaling handshake and exchange encrypted messages. Third, an offline message queuing mechanism would improve user experience by allowing messages to be encrypted and stored server-side (in a transient, encrypted form) when the recipient is offline and delivered automatically upon reconnection. Fourth, production-grade SSL/TLS certificates (e.g., from Let's Encrypt) should replace self-signed certificates to prevent browser warnings and support true trust in public deployments. Fifth, in anticipation of quantum-era threats, post-quantum key-exchange schemes approved by NIST (such as Kyber) could be evaluated and optionally incorporated into the key-agreement phase. Finally, introducing a TURN relay server (for example, Coturn) would ensure reliable connectivity in restrictive NAT and firewall environments, further broadening the application's real-world applicability.

AI acknowledgements:

Large language models (LLMs) were used in the following parts of this project:

- Debugging.
- Helping with the Jenkins pipeline setup.
- The bare bones html file was entirely created by Ai.
- Ai was used in this report for reformatting and improving readability.