Tampere University

COMP.SEC.300 Secure Programming

Spring 2025

Secure Programming Exercise Work

End-to-end encrypted real-time chat application

Repository: https://github.com/SalonenTeemu/e2ee-real-time-chat-app

Contents

1 General description 1
1.1 How to use the application 2
2 Use of AI 3
3 Used technologies 3
4 Structure of the program 4
5 Secure programming solutions 6
5.1 Authentication, key exchange, and end-to-end encryption flow 8
6 Security and unit testing 8
7 Security vulnerabilities 10
8 Suggestions for improvement 11

1 General description

This real-time chat application enables secure communication through end-to-end encrypted messaging. It features user registration, login, secure messaging, and role-based access control to ensure that only authorized users can access specific parts of the system, such as the chat page. To protect against DDoS and brute-force attacks, the system includes rate-limiting, and security logging tracks critical events for auditing purposes.

End-to-end encryption (E2EE) is implemented using cryptographic algorithms to ensure that all messages exchanged are private. Encryption occurs on the client side, meaning only the intended recipient can decrypt and read the messages. The user's private key is stored with encryption in the browser's indexed database and can only be decrypted with the user’s password. The key can be regenerated using the correct seed phrase (generated via BIP 39) and password, allowing for use across different environments.

The application can be run via Docker or locally, with the former requiring Docker engine and Docker compose installations and latter local PostgreSQL database installation. Basic unit tests for both frontend and backend ensure core functionality. Additionally, a Jenkins

security pipeline is included to generate security-related reports, which can be found in the repository under the ‘docs/security-reports’ folder.

1.1 How to use the application

The README.md file in the project root provides detailed instructions and requirements for installing and running the application. Once the application is running in the browser:

1. **User registration and login:**
   - The first step is to register a new user via the registration page.
   - After successful registration, proceed to log in through the login page.
2. **Seed phrase and private key:**
   - On the first login, the application generates a unique 24-word seed phrase, which is shown to the user.
   - This seed phrase should be stored securely by the user as it is critical for recovering the private key on other browsers or devices.
3. **Starting a chat:**
   - After saving their seed phrase, users are directed to the chat page, where they can initiate new chats.
   - On the left sidebar, users can search for other registered users and start a chat by selecting one.
   - To test the chat functionality, it's recommended to open a window in another browser (Chrome and Firefox for example), register a second user, and log in. Then, one user can search for and initiate a chat with the other.
4. **Sending and receiving messages:**
   - Messages can be securely exchanged between users.
   - If a user closes a chat or navigates away, incoming messages will trigger notifications to alert the user.
5. **Session timeout and reauthentication:**
   - For security, the decrypted private key is cleared from client memory after 5 minutes of inactivity or on page reload.
   - When this happens, the user will be prompted to re-enter their password to re-enable message encryption and decryption.
6. **Cross-environmental access:**
   - If a user logs in from a different environment (e.g., a new browser or device), they will be required to input both their password and seed phrase to regenerate

their private key, which is why it is essential that the user has securely saved their seed phrase during the initial setup.

2 Use of AI

Artificial intelligence tools such as ChatGPT and GitHub Copilot were used throughout the project to assist with coding and problem-solving. AI was particularly used in generating unit tests and resolving issues related to the security pipeline as it was difficult to find help with it from the internet. Additionally, AI was used to refine the wording of this report.

3 Used technologies

- **Deployment:** Locally or with Docker using Docker compose.
- **CI/CD security pipeline:** Jenkins.
- **Backend:** Node.js with Express.js. Jest and supertest for testing.
- **Frontend:** React with Tailwind CSS. Jest and @testing-library/react for testing.
- **Real-Time Communication:** Socket.io (backend) and Socket.io-client (frontend).
- **Database:** PostgreSQL.
- **Authentication:** JSON Web Tokens (JWT).
- **Password hashing:** Bcrypt library.
- **Encryption Algorithms (libraries used):**
  - XChaCha20-Poly1305 – message encryption (libsodium).
  - AES-256-GCM – additional database message encryption (Node.js crypto).
  - X25519 (Elliptic Curve Diffie-Hellman) – key exchange (libsodium).
  - XSalsa20-Poly1305 – private key encryption on client side (libsodium).
  - PBKDF2 + SHA-256 – password-based key derivation (Web Crypto API).
  - Ed25519 – intermediate key pair generation from seed (tweetnacl).
- **Seed Phrase Generation (library used):**
  - BIP39 – generates 24-word mnemonic seed phrases (bip39).
  - Uses tweetnacl for Ed25519 key pair generation from the seed, then converts to X25519 with libsodium.

4 Structure of the program

1. **Root directory:**
   - Jenkinsfile – Defines the CI/CD security testing pipeline.
   - docker-compose.yml – Docker configuration for deploying the app.
   - .env.example – Template for environment variables.
   - README.md – Setup instructions, usage, and project details.
2. **Frontend:**

   The frontend is structured into the following folders:

   - **Root** – Contains the Dockerfile, Jest configuration for testing, and Vite configuration for setting server details and custom headers.
   - **Components** – Includes the Home, Login/Register, Chat, and Restore pages, along with components like the Navbar, Password and Seed Phrase Modals.
   - **Context:**
     - AuthContext: Manages authentication and user state.
     - NotificationContext: Handles global notifications.
   - **Services:**
     - keys.ts and keyManager.ts: Manage key generation, storage, encryption, and memory clearing after inactivity.
     - socket.ts: Sets up the client-side Socket.io instance.
   - **Utils** – Helper functions for indexed database interaction, encryption, logging, input validation, seed phrase generation, API fetching, and message sanitization.
   - **Tests** – Unit tests for components, services, and utilities using Jest and React Testing Library.
3. **Backend:**

   The backend is structured into the following folders:

   - **Root** – Contains the Dockerfile and Jest configuration for testing.
   - **Controllers:**
     - authController.ts: Manages registration, login, logout, token handling, user profile retrieval and password verification.

    - chatController.ts, messageController.ts: Handle chat creation, retrieval, and message fetching.
    - keyController.ts: Handles saving and retrieving public keys.
    - userController.ts: Enables user search functionality.
- **DB:**
    - initDB.ts: Initializes the database schema and tables.
    - knex.ts: Sets up the Knex-based database connection.
    - queries/: Contains query files for chats, keys, messages, tokens, and users.
- **Middleware:**
    - cors.ts: Configures CORS settings.
    - rateLimiting.ts: Adds request rate limiting.
    - user.ts: Handles authentication and role-based access control.
- **Routes** – Defines Express routes for authentication, chat, key, message, and user-related actions.
- **Services:**
    - authService.ts: Manages access and refresh token creation, verification, and revocation.
    - socket.ts: Sets up and manages the backend Socket.io server, including middleware and rate limiting.
- **Utils:**
    - encryption.ts: Performs AES-256-GCM encryption and decryption for messages.
    - logger.ts: Logs key server events.
    - sanitize.ts, validate.ts: Cleans and validates input data.
    - tokenCleanupCron.ts: Daily cron job to remove expired/revoked refresh tokens from the database.
    - constants.ts, types.ts: Store reusable values and TypeScript types.
- **Tests** – Includes unit and integration tests for API endpoints, services, and utility functions using Jest and Supertest.

5 Secure programming solutions

Password strength and hashing:

- Passwords are required to be strong. They must be between 12 and 100 characters, containing lowercase, uppercase, numbers, and special characters. The check-password-strength NPM library is used to validate password strength on both the client and server sides. Passwords are hashed with the bcrypt NPM library before being stored in the database. During login, bcrypt is also used to compare the entered password with the stored hash to verify correctness.

Chat message sanitization:

- To prevent XSS attacks, chat messages are sanitized on both the client and server. The DOMPurify NPM library sanitizes messages before they are encrypted and sent to the server, as well as before being processed on the server side.

Session safety:

- User private keys and cached shared keys are stored in memory only during active sessions. After inactivity, keys are securely erased using sodium.memzero, preventing lingering sensitive data in memory.

Authentication:

- JWT Tokens: Users authenticate using JWT tokens. Upon login, tokens are generated and stored in secure, HttpOnly cookies (with the Secure flag enabled in production), ensuring better protection against session hijacking and XSS attacks. Tokens are refreshed using refresh tokens. Endpoints requiring authentication/authorization validate tokens and user roles. Expired or revoked refresh tokens are automatically cleaned from the database daily.

End-to-end encryption (E2EE):

- Messages are encrypted using XChaCha20-Poly1305 on the client side before being sent to the server, ensuring that only the intended recipient can read them.

Key exchange:

- The X25519 (Elliptic Curve Diffie-Hellman) algorithm is used for the key exchange between users. This ensures that the shared secret used to derive the session key is never exposed, making man-in-the-middle (MITM) attacks more difficult.

Private key generation and encryption:

- Private keys are derived from seed phrases generated using BIP39. These keys are encrypted with XSalsa20-Poly1305 before being stored in the browser's indexed database. The encryption key is derived from the user's password using PBKDF2.

Message storage encryption:

- Messages are additionally encrypted with AES-256-GCM before being stored in the database, providing extra security in case of database access breaches.

HTTP response headers:

- The server uses the helmet NPM library to automatically set default secure HTTP headers, mitigating common web vulnerabilities such as cross-site scripting (XSS). Default protections are applied, which include for example HTTPS enforcement, Content Security Policy (CSP), and X-Content-Type-Options.
- The client additionally configures strict security headers, including CSP, cross-origin resource policies, permissions policies, and caching directives, further reducing the risk of client-side attacks and data leakage.

Parameterized database queries:

- Parameterized queries are used for database interactions to prevent SQL injection attacks by ensuring user inputs are treated as data, not executable code.

Rate limiting:

- Server rate limiting is implemented using the rate-limiter-flexible NPM library to guard against brute force and DDoS attacks. Limits are applied to both REST API endpoints and WebSocket connections.

Security logging:

- Server-side events are logged with winston and winston-daily-rotate-file NPM libraries for audit and debugging purposes. The logs can be viewed from the

`backend/logs` folder. Client-side logging is limited to the development environment and disabled in production.

5.1 Authentication, key exchange, and end-to-end encryption flow

1. **User authentication:** Upon successful login, the server issues a JWT access token and refresh token, which are stored in cookies. These tokens are included with subsequent requests to authenticate the user.
2. **Private key generation:** On first login, a 24-word BIP39 seed phrase is generated. This seed is used to create the user's private key, which is then encrypted using XSalsa20-Poly1305 with a key derived from the user’s password (via PBKDF2) and stored in the browser’s indexed database. The seed phrase allows recovery of the private key on other environments (device and/or browser).
3. **Key exchange (ECDH):** When initiating a new chat, users perform a secure key exchange using X25519 (Elliptic Curve Diffie-Hellman). A shared secret is derived from each user's private key and the other party’s public key.
4. **Shared key usage:** The resulting shared key is used for encrypting and decrypting messages between the users.
5. **Client-side message encryption:** Messages are encrypted on the client using the shared key and the XChaCha20-Poly1305 algorithm before being sent to server, ensuring end-to-end confidentiality.
6. **Server-side storage encryption:** As an added layer of protection, the server encrypts incoming messages with AES-256-GCM before storing them in the database. The encryption key from the .env file is used. The encrypted message is then relayed to the intended recipient.
7. **Client-side message decryption:** The recipient decrypts messages on the client using the same shared key, ensuring only the intended users can read the content.

6 Security and unit testing

The application includes basic unit tests for both frontend and backend, located in their respective tests folders. While a few minor issues were identified and fixed during testing, no major problems were detected. However, this doesn’t guarantee the absence of issues.

Security testing was handled through a Jenkins CI/CD pipeline defined in the root Jenkinsfile, which documents each step of the process. The security reports generated by

the pipeline are stored in the `docs/security-reports` folder, including reports from both before and after fixes were applied.

Security tests in the pipeline:

- Static Application Security Testing (SAST) with Semgrep
- Software Composition Analysis (SCA) with OWASP Dependency-Check
- File System Scanning with Trivy
- Container Image Scanning with Trivy
- Dynamic Application Security Testing (DAST) with OWASP ZAP

Key findings and fixes:

SAST (Semgrep):

- Missing user in Dockerfiles: Resolved by creating non-root users in both frontend and backend Dockerfiles.
- Missing tag length in AES-GCM decryption: Fixed by specifying authTagLength: 16 in the crypto.createDecipheriv() options in the encryption.ts file in the backend.
- Pipeline environment-related low/medium risk alerts: Eliminated by excluding irrelevant semgrep environments from the report using --exclude semgrep-env/.

File System & Image Scanning (Trivy):

- File system scan: No vulnerabilities found.
- PostgreSQL image (postgres:17-alpine3.21): Contains known vulnerabilities in base libraries (e.g., libxml2, libxslt, xz-libs). These stem from Alpine Linux 3.21 and upstream dependencies. Not much can be done about these issues other than monitoring them and using the latest PostgreSQL image version available.
- Frontend & backend images: No vulnerabilities reported. While Alpine 3.21.3 has known CVEs, affected libraries are not included in the final builds.

DAST (OWASP ZAP):

- The OWASP ZAP scan reported: 0 high, 2 medium, 4 low and 4 informational vulnerabilities. All medium and low vulnerabilities were fixed except for one low-severity issue. Informational issues were acknowledged but not addressed due to their minimal impact.

Fixed vulnerabilities:

- Content Security Policy Header Not Set (Medium): Resolved by configuring CSP in the frontend Vite setup.
- Missing Anti-clickjacking Header (Medium): Fixed by adding Content-Security-Policy with frame-ancestors directive and setting X-Frame-Options header to DENY preventing clickjacking.
- Insufficient Site Isolation Against Spectre (Low): Resolved by setting Cross-Origin-Resource-Policy header to same-origin.
- Permissions Policy Header Not Set (Low): Addressed by adding strict feature restrictions: res.setHeader('Permissions-Policy', 'accelerometer=(), autoplay=(), camera=(), geolocation=(), gyroscope=(), magnetometer=(), microphone=(), payment=(), usb=()');
- X-Content-Type-Options Header Missing (Low): Resolved with setting X-Content-Type-Options header to nosniff to avoid MIME-sniffing on the response body.

Unresolved vulnerability:

- Timestamp Disclosure – UNIX (Low): Left unpatched, as it poses minimal risk unless combined with other serious vulnerabilities. This may stem from build metadata, library versions, or static timestamps in some dependencies.

7 Security vulnerabilities

Following the implemented security fixes, the reports highlight two issues mentioned in section 6:

- UNIX timestamp disclosure
  - Considered low-risk and exploitable only in combination with more severe issues. The timestamp may originate from build metadata or static content in dependencies.
- PostgreSQL base image vulnerabilities
  - The official postgres:17-alpine3.21 image contains known vulnerabilities in system libraries and Go’s standard library. These are inherited from the upstream Alpine Linux base and cannot be mitigated directly if the docker image is decided to be used as the database option.

In addition to the reports, currently the decrypted private keys are stored in client-side browser memory for up to 5 minutes of inactivity, after which they are cleared using sodium.memzero. While this balances usability and basic security, it is not ideal for protecting sensitive key material against advanced threats like memory scraping or browser exploits. A more secure and long-term solution for the application could involve offloading key management to a trusted device, such as:

- Native mobile app pairing, like WhatsApp Web does:
  - The mobile app would securely store the private key.
  - The browser client would forward encryption/decryption/signing requests via WebRTC or QR-code-based secure channels.
  - This separation would significantly reduce the exposure surface of private key material in the browser.

Also, to enable required cryptographic operations, the client-side Content Security Policy (CSP) includes the directive: script-src 'wasm-unsafe-eval'. This exception is necessary to support WebAssembly execution by the libsodium-wrappers library, which handles critical cryptographic functions such as key conversions and encryption. While this inclusion slightly relaxes the CSP, it is a targeted and justified exception but should be kept in mind.

8 Suggestions for improvement

To further enhance functionality, user experience, and security, the following features and improvements could be implemented:

Functional and user experience enhancements:

- User profile management: Enable users to update their profile information, such as display name or status.
- Profile avatar support: Add support for uploading and displaying profile pictures.
- Message status indicators: Display message delivery stages such as “Sent,” “Delivered,” and “Read” to improve communication feedback.
- Group chats: Implement support for multi-user conversations with group-level permissions and shared encryption keys.
- Message editing and deletion: Allow users to edit or delete messages after sending, with appropriate audit or visibility logic.

- Anonymous chat rooms: Support “throw-away” chat rooms that allow temporary E2EE communication without user registration or authentication.

Security improvements:

- Two-Factor Authentication (2FA): Integrate 2FA to strengthen account security, especially during login and sensitive actions.
- Forward secrecy: Implement ephemeral key exchanges (e.g., via a double ratchet algorithm) to ensure past messages remain secure even if long-term keys are compromised.
- Native mobile app integration: Improve private key security by pairing the browser-based app with a mobile app that holds the private key as mentioned in section 7.