TAMPERE UNIVERSITY

COMP.SEC.300-Secure Programming

Report on: Secure Chat Forum

Presented by:

- Hussn Ul Maab, Roll: 153096592
- Sourav Podder, Roll: 152702560
- Zannatun Nayeem Chowdhury, Roll: 152847289

1. Introduction

Instant messaging has become a normal part of everyday life, but many popular apps still treat security as an after-thought. Data leaks, account take-overs, and privacy violations show what can happen when protection is weak. Our team decided to build a small yet complete chat forum that puts security first. The finished system works both in a web browser and in a mobile application, giving users familiar real-time messaging while adding seven carefully selected safeguards. Everything is open-source, and the full code can be viewed in the two GitHub repositories shared above.

2. Project Overview

The Secure Chat System was created by Hussn Ul Maab, Zannatun Nayeem Chowdhury, and Sourav Podder as part of the COMP.SEC.300 Secure Programming course. Our goal was clear: deliver a chat platform that remains usable for ordinary people yet bakes in defence layers normally found only in enterprise software. The solution had to be:

- **Cross-platform** — one code-base powering both a modern website and an Android/iOS app.
- **Real-time** — messages appear instantly without page reloads.
- **Security-centric** — every feature checked against common vulnerability classes before release.

3. Technology Stack and System Architecture

- **Backend** — *Node.js* with **Express** routes, **SQLite** for lightweight persistence, and **Socket.IO** for WebSocket messaging.
- **Web front-end** — vanilla HTML/CSS/JS enhanced with Socket.IO client and secure cookies for session storage.
- **Mobile front-end** — written in **Flutter**, using the same Socket.IO channel and device **SharedPreferences** to hold the JWT token safely.
- **Deployment** — both clients talk to a single Express server. All traffic moves through HTTPS, handled either by Nginx or by the built-in TLS support during development.

This full-stack design lets us re-use almost all business logic, share validation code, and push fixes quickly to every user.

4. Security-First Design Mind-Set

Before writing any controller or widget we drew a threat model. Standard attacker goals—steal credentials, read other people's messages, flood the service, abuse debug output—were mapped to countermeasures. Each countermeasure became one of the seven key security features described next. We deliberately chose defences that complement each other: if one ever fails there is a second layer to slow the attacker down or block them entirely.

5. The Seven Core Security Features

#	Feature	What It Does	How It Works in Code
1	**User Authentication**	Confirms that a person is really who they claim to be and stops password theft from revealing plain text secrets.	We hash every password with **bcrypt** (12 rounds) before storing it. After login the server issues a **JWT** that contains only a user ID and role. The token lives one hour, is signed with HS256, and is sent back as an **HTTP-only, Secure** cookie so client JavaScript cannot read it.
2	**Input Validation & Sanitisation**	Blocks code injection, cross-site scripting (XSS), and malicious SQL.	All incoming JSON is checked with **validator.js** for length, format, and character set. Text that might be rendered (usernames, chat body) is further cleaned by the **xss** library which escapes script tags and JavaScript event handlers. Query parameters are parameterised in SQLite to kill SQL injection attempts.
3	**Data Encryption**	Protects message content both on the wire and at rest.	During transit we require **TLS 1.3**. Inside the database each message row is encrypted with **AES-256-CBC**; the key is loaded from an environment variable so it never lives in the repository. Even if an attacker dumps the DB file, they still need the server secret to read any text.
4	**Role-Based Access Control (RBAC)**	Limits powerful actions to administrators.	A role claim lives inside the JWT. Middleware allows only admin users to hit routes such as /admin/clear-chat or /admin/users. In the UI, admins see an extra panel; normal users will not even get the button.
5	**Session Management**	Prevents hijacked or long-lived sessions from being abused.	JWT cookies use SameSite=Strict, expire in 60 minutes, and are stored securely. Token validation is enforced in all routes and socket connections. Expired or missing tokens result in immediate session rejection, reducing hijack risk.
6	**Rate Limiting & Account Lockout**	Thwarts brute-force attempts against the login endpoint.	With **express-rate-limit** each IP may send only five login requests per minute. After ten failed tries the account is locked for 15 minutes and an audit entry is stored.
7	**Secure Error Handling**	Stops information disclosure through stack traces.	A custom Express error handler intercepts every thrown error. In production it returns only a generic "Something went wrong" message and a unique correlation ID; full stack traces log internally for developers.

Together these controls address the OWASP Top 10 categories of A01 (Broken Access Control), A02 (Injection), A03 (Cryptographic Failures), A05 (Security Misconfiguration), and A07 (Identification & Authentication Failures), giving the platform defence-in-depth rather than a single point solution.

6. Implementation Highlights

Modular back end The server code is split into small files:

- auth.controller.js for login, signup, token refresh.
- chat.controller.js for message CRUD and Socket.IO events.
- security.middleware.js for validation, rate limiting, and JWT checks.

Shared validation The same schemas.js file is imported by both the Express routes and the Flutter client, ensuring that the web form, the mobile form, and the API all treat invalid data the same way.

Mobile adaptations On phones the Flutter *http* package rejects self-signed certificates, so we export a development CA bundle for easier local testing. Flutter's SharedPreferences stores only the JWT cookie header, never the user password.

Web client extras For desktop users we added copy-to-clipboard, emoji support, and a dark-mode toggle, yet the extra JavaScript never touches privileged routes, keeping the attack surface narrow.

7. Testing and Quality Assurance

Automated tests run in GitHub Actions on every push. We rely on:

- **Jest** — unit tests for utility functions, crypto helpers, and business rules.
- **Supertest** — spins up the Express app, fires HTTP calls, and checks for proper status codes, CORS headers, and cookie flags.
- **SQLite (in-memory)** — each test file starts with a blank database, so results never leak from one suite to another.
- **Socket.IO-client** — integration tests confirm that a message sent by user A actually appears in user B's listener array and that unauthenticated sockets are refused.

Manual quality checks were done with Burp Community Edition to scan for reflected XSS and with ADB logcat to make sure the mobile build never logs secrets.

8. User Experience and Interface

Users start by signing up with a username, email, and password. After logging in, they can instantly join the chat, send and receive messages in real time, and see other users' names and avatars. The web app offers emoji support and dark mode, while the mobile app provides a native look and stores session tokens securely using SharedPreferences. Admins have access to additional controls, including viewing active users and clearing chat history.

- **Live avatars and nicknames** make it easy to spot who is speaking.
- **Readable chat bubbles** and timestamps keep the conversation clean.
- **Toasts and spinners** show feedback when the network is slow.
- **Admin dashboard** lets privileged users view a list of active sessions and press a single "Clear History" button if something inappropriate is posted.

Performance stays smooth because Socket.IO streams only new messages, and Flutter's widget tree reuses existing components rather than repainting the whole screen.

Demo link of mobile app: SecureChatDemoVideoMobileApp.mp4

9. Limitations and Future Work

The current release already covers many major threats, yet we identified four areas that would raise the bar even more:

1. **End-to-End Encryption (E2EE)** — today the server can read plain text after decrypting the database field; switching to double-ratchet keys would lock even the administrators out of message content.
2. **Two-Factor Authentication (2FA)** — TOTP or WebAuth would reduce the impact of a leaked password hash.
3. **Scalable Data Store** — moving from SQLite to PostgreSQL or MongoDB would support large group chats and horizontal scaling.
4. **Push Notifications & Offline Caching** — FCM/APNs integration and encrypted local storage would improve usability without touching the threat model.

10. Conclusion

The Secure Chat Forum proves that strong protection does not have to sacrifice convenience. By layering seven complementary safeguards on top of a simple, reusable stack, we built a platform that resists common web attacks, prevents credential theft, and keeps private conversations private. The same code runs smoothly in the browser and on mobile devices, demonstrating that security-by-design can be both portable and approachable. Future enhancements such as E2EE and 2FA will push the project even further, but the current version is already suitable for classrooms, clubs, and small organisations that need a safe space to talk. Our hope is that other developers study the open repositories, reuse the modules, and continue to make everyday communication more secure for everyone.

USE OF AI IN REPORT

I hereby declare that the AI-based applications used in this work are as follows:

Application	Version
ChatGpt	4.0

Purpose of the use of AI: ChatGpt was utilized to identify grammatical and spelling mistakes and also helped me to structure the report properly.