Peer Review Report: Code Testing and Security Analysis (Group: 29)

Introudaction

This peer review evaluates the code implementation and security posture of the Group 29 chat system. The assessment focuses on both functional correctness and security resilience, employing a combination of manual code inspection, and static analysis (Snyk and Semgrep). The review identifies areas where the system demonstrates strong design practices, as well as potential vulnerabilities that could impact confidentiality, integrity, and operational robustness. The goal of this report is to provide actionable, detailed feedback to support learning and improvement, fostering a stronger understanding of secure web application development.

Strengths in Implementation

The system demonstrates several key strengths in its design and implementation. The code is well-structured, with a clear separation between backend and frontend modules, which enhances maintainability and readability. The adoption of modern libraries, including Fastify, Prisma, Next.js, and React, reflects current best practices in full-stack development and supports efficient, scalable application architecture. Client-side state management is handled thoughtfully, with UI components leveraging cookies and local storage to persist user preferences effectively. Additionally, the use of security-aware dependencies, such as established cryptography and input-handling libraries, minimizes the need for custom implementations and reduces the likelihood of introducing subtle security flaws. Collectively, these design choices demonstrate a careful and considered approach to building maintainable, functional, and secure web applications.

Testing Approaches and Tools Used

1. Manual Code Review

I systematically examined key frontend files, particularly sidebar.tsx and group-keys.ts, to locate areas where sensitive operations occur. This included checking for:

- Handling of cookies and client-side state.
- Cryptographic operations, including key generation and import.
- Patterns of hardcoded values, unsafe defaults, and unvalidated user inputs.

Manual review guided the focus of automated scanning and provided contextual understanding of potential weaknesses.

2. Static analysis (Snyk & Semgrep):

Snyk Code was applied to scan the entire codebase for known security issues. This tool performs **data flow analysis**, mapping sources of potentially sensitive information to sinks that could be exploited. The scan revealed two major findings:

- **Hardcoded cryptographic key** in *frontend/src/lib/group-keys.ts*, where a deterministic key is created and imported via crypto.subtle.importKey, exposing predictable key material to potential attackers.
- **Sensitive Cookie without Secure attribute** in *frontend/src/components/ui/sidebar.tsx*. Snyk traced the flow from the state variable storing sidebar visibility to the document.cookie sink, highlighting exposure of client-side state over insecure channels.

These findings are supported by stepwise traces in the Snyk Code output, showing the exact lines where sensitive sources are used and where the data flows to sinks. As illustrated in Figure 1 and 2, the screenshots from Snyk clearly demonstrate both the insecure cookie handling in sidebar.tsx and the hardcoded cryptographic key in group-keys.ts.

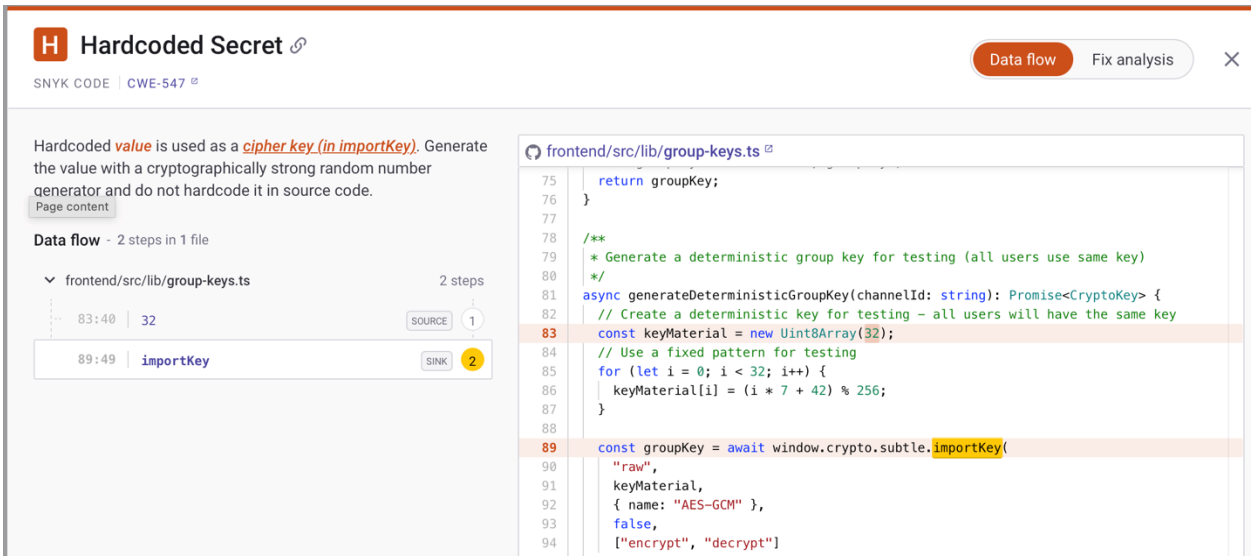

*Figure 1: Snyk Report (**Hardcoded cryptographic key**)*

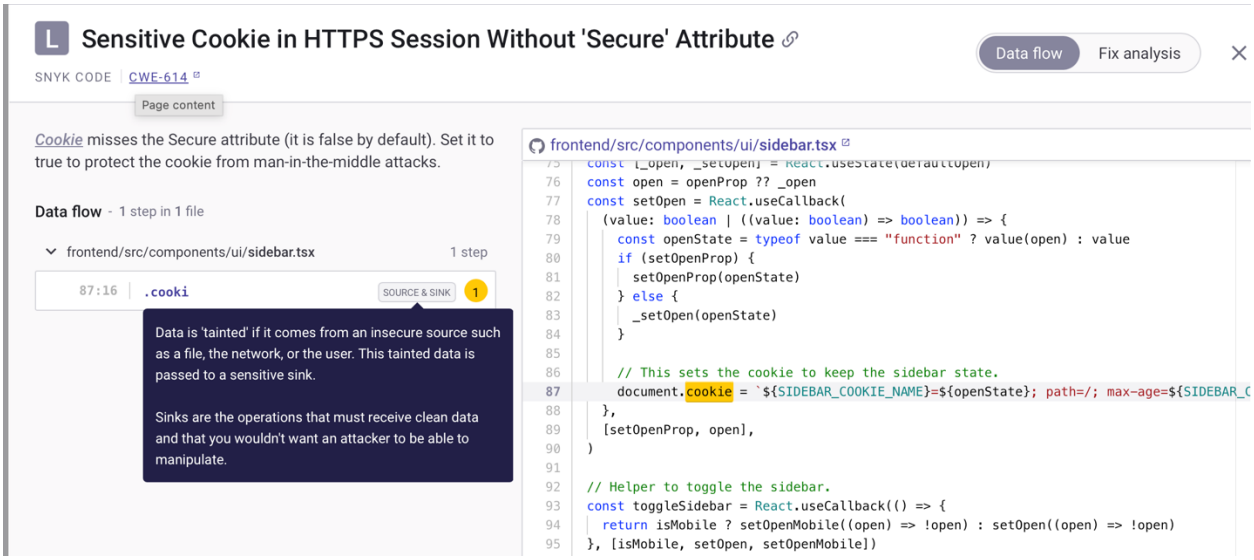

*Figure 2: Snyk Report (**Sensitive Cookie**)*

Semgrep was applied to scan the frontend and utility modules for security issues using its automatic rule configuration. This static analysis tool identifies common coding patterns that may lead to vulnerabilities. The scan revealed two notable findings:

- **Unsafe string formatting in frontend/src/contexts/ChatContext.tsx**, where dynamic variables are concatenated in a console.error statement (console.error(\Failed to decrypt ${messageType} message:`, error);), potentially allowing log message manipulation if untrusted input is included.
- **Insecure WebSocket connection in utilities/server-join.js**, where the WebSocket is established using ws://instead of the securewss://` protocol, exposing messages to potential interception or tampering.

These findings are illustrated in Figures 3, which show the Semgrep terminal output highlighting the unsafe string formatting in ChatContext.tsx and the insecure WebSocket usage in server-join.js

Figure 3: Semgrep Report

Analysis of Test Results

During analysis of the chat system, Snyk identified that the method generateDeterministicGroupKey in frontend/src/lib/group-keys.ts generates a hardcoded, deterministic cryptographic key. This key is imported using crypto.subtle.importKey and is then used in ChatProvider (frontend/src/contexts/ChatContext.tsx) when processing public channel messages (PUBLIC_CHANNEL_KEY_DELIVERY) if no existing group key is found. While log messages indicate that this key generation is intended for testing purposes, it is actively used during runtime to decrypt public messages as shown in Figure 4. This creates a potential security concern: the use of a predictable key could expose sensitive information if the system were deployed in a real environment. For testing and development, deterministic keys are useful for repeatability, but for production use, ephemeral or securely randomized keys would better preserve the confidentiality of messages and strengthen the overall security of the chat system. This observation

highlights an important distinction between development convenience and production-grade security within the implemented messaging workflow.

Figure 4: generateDeterministicGroupKey used in active run time.

Moreover, Semgrep flagged two issues in the codebase that could be improved for security and robustness. The first is an **unsafe format-string / logging pattern** in frontend/src/contexts/ChatContext.tsx (console.error(\Failed to decrypt ${messageType} message:`, error);). Since messageType comes from incoming messages, a malicious actor could potentially inject crafted content that corrupts log entries, which may confuse operators or obscure important audit trails. To improve this, it would be better to use constant format strings or explicitly sanitize dynamic values before logging. The second finding is an **insecure WebSocket connection** inutilities/server-join.js (ws:// used instead of wss://). This exposes all communication over plaintext, allowing attackers on the network to intercept, modify, or replay messages. Switching to wss:// with proper TLS validation would secure the channel and protect message confidentiality and integrity. Both issues are reachable during normal runtime, so addressing them would strengthen the system and improve resilience against potential attacks, while also maintaining clearer, safer operational logging.

Vulnerabilities

Vulnerability: Hardcoded Cryptographic Key (Predictable Key Material)
Title: Deterministic group key generation using hardcoded key material
Severity: High (Cryptographic weakness / confidentiality risk)
Location: frontend/src/lib/group-keys.ts → generateDeterministicGroupKey()
Type: Insecure cryptographic key management / predictable key usage
CWE: CWE-321 (Use of Hardcoded Cryptographic Key) / CWE-330 (Use of Insufficiently Random Values).
Summary: The function generateDeterministicGroupKey() generates a fixed cryptographic key using deterministic or hardcoded data before importing it via crypto.subtle.importKey. This deterministic behavior makes the derived key predictable and reusable across sessions, violating the confidentiality principle of cryptographic operations.
Evidence / How found: Snyk Code flagged the use of deterministic key generation in this file. Manual code review confirmed that the key material is not generated from a secure random source but rather derived from static input for convenience.
Reproduction / Detection steps: Locate the generateDeterministicGroupKey() function in frontend/src/lib/group-keys.ts and observe that it constructs the key deterministically before importing it using crypto.subtle.importKey.
Impact / Exploitation: An attacker who knows or can guess the key derivation method could

decrypt intercepted messages, impersonate users, or forge encrypted communications. This poses a severe confidentiality and integrity risk if the code is ever deployed in production.
Mitigation: Replace deterministic key generation with securely randomized key material (e.g., crypto.getRandomValues()), and ensure group keys are ephemeral and unique per session or group. For testing, use mock keys isolated from production builds.
Confidence: High (confirmed through static analysis and manual validation).

Vulnerability : Sensitive Cookie Without Secure Attribute
Title: Insecure cookie configuration for sidebar state persistence
Severity: Medium (Information disclosure / session risk)
Location: frontend/src/components/ui/sidebar.tsx → Cookie handling logic
Type: Insecure cookie attribute configuration
CWE: CWE-614 (Sensitive Cookie Without 'Secure' Attribute) / CWE-312 (Cleartext Storage of Sensitive Information)
Summary: The sidebar visibility state is stored in a client-side cookie without setting the Secure attribute. This allows the cookie to be transmitted over unencrypted HTTP channels, potentially exposing session-related information to attackers on the same network.
Evidence / How found: Snyk Code flagged this issue by tracing cookie storage operations that lack security attributes. Manual verification confirmed that the cookie was being created without Secure or SameSite attributes.
Reproduction / Detection steps: Search for cookie-setting logic in sidebar.tsx and confirm that the cookie is defined without Secure or SameSite properties.
Impact / Exploitation: Attackers on the same network could intercept the cookie if HTTP connections are used, potentially manipulating user interface states or leveraging the cookie for phishing-like UI manipulation.
Mitigation: Always set Secure, HttpOnly, and SameSite=Strict attributes when defining cookies; alternatively, store non-sensitive UI preferences in localStorage if appropriate.
Confidence: High (confirmed through Snyk trace and manual code inspection).

Vulnerability: Unsafe String Formatting in Logging
Title: Unvalidated dynamic data concatenated into log output
Severity: Low (Potential log injection / audit integrity risk)
Location: frontend/src/contexts/ChatContext.tsx → console.error(\Failed to decrypt ${messageType} message:`, error).
Type: Unsafe string interpolation in logging.
CWE: CWE-117 (Improper Output Neutralization for Logs).
Summary: Dynamic message types are directly interpolated into log statements without sanitization. If themessageTypevalue originates from untrusted network input, it could inject control characters or misleading text into logs.
Evidence / How found: Semgrep flagged unsafe dynamic formatting in logging statements. Manual inspection confirmedmessageTypecan originate from incoming messages.
Reproduction / Detection steps: Locate theconsole.errorcall and inspect howmessageType is constructed or passed. Injecting special characters or escape sequences could distort log entries.
Impact / Exploitation: Log manipulation could obscure real errors or mislead administrators

reviewing system behavior, reducing the reliability of operational monitoring and audit trails.
Mitigation: Use constant log format strings and sanitize or encode any dynamic input before inserting into logs.
Confidence: Medium (confirmed reachable during runtime).

Vulnerability : Insecure WebSocket Connection
Title: WebSocket connection established over unencrypted channel
Severity: High (Confidentiality and integrity risk / network eavesdropping)
Location: utilities/server-join.js → new WebSocket('ws://...')
Type: Unencrypted communication channel
CWE: CWE-319 (Cleartext Transmission of Sensitive Information)
Summary: The WebSocket connection to the server is initiated using the ws:// protocol, which transmits all traffic in plaintext. This exposes messages to interception or modification by attackers on the same network.
Evidence / How found: Semgrep identified the use of ws:// and flagged it as insecure. Manual validation confirmed that no conditional logic upgrades the connection to wss://.
Reproduction / Detection steps: Inspect utilities/server-join.js and observe the use of ws:// instead of wss:// when initializing the WebSocket connection.
Impact / Exploitation: Attackers can perform man-in-the-middle (MITM) attacks to read, alter, or replay messages exchanged between the client and server. This directly undermines confidentiality and message authenticity.
Mitigation: Use wss:// (WebSocket Secure) with TLS certificates properly configured. Ensure the backend WebSocket server supports encrypted connections.
Confidence: High (verified through static and manual review).

Short aggregated guidance :

- **Use Secure Cryptography:** Replace deterministic or hardcoded keys with securely generated, ephemeral keys using strong random sources (crypto.getRandomValues()).
- **Harden Cookies:** Always set Secure, HttpOnly, and SameSite=Strict attributes when storing sensitive information in cookies.
- **Sanitize Logging Inputs:** Avoid direct interpolation of untrusted data in logs; use constant format strings or explicitly sanitize dynamic content.
- **Encrypt Communications:** Upgrade all WebSocket connections to wss:// with proper TLS configuration to ensure confidentiality and integrity.
- **Separate Development and Production Practices:** Ensure that test conveniences, such as deterministic keys, do not propagate into production builds.

Summary and Overall Feedback

The chat system exhibits several notable strengths. Its structured implementation, with a clear separation between backend and frontend modules, enhances maintainability and readability, while the use of modern tooling, including Fastify, Prisma, Next.js, and React, demonstrates adherence to current full-stack development best practices. Client-side state management is thoughtfully handled via cookies and local storage, supporting a positive user experience, and the reliance on established cryptography and input-handling libraries, rather than custom implementations, reduces the likelihood of subtle security flaws. Despite these strengths, the review identifies areas where security practices can be improved. Deterministic, hardcoded keys used for group message decryption create predictable key material, which could compromise confidentiality in production environments. Certain client-side cookies lack Secure and SameSite attributes, leaving sensitive state information exposed over unencrypted channels. Dynamic values are directly interpolated into log outputs without sanitization, potentially enabling log manipulation or obscuring audit trails. Additionally, WebSocket connections established over ws:// transmit messages in plaintext, making communications vulnerable to interception or tampering. Overall, while the implementation demonstrates thoughtful design and solid functionality, addressing these security issues would further strengthen robustness and adherence to secure development principles.