Peer Review Report: Code Testing and Security Analysis (Group: 42) By: Sahar Hassan Alzahrani (a1938372)

Introudaction

This peer review report provides a detailed evaluation of Group 42's *Secure Chat Protocol* project, focusing on its code structure, testing methodology, and security posture. The review combines both static and dynamic analysis techniques to identify vulnerabilities in implementation, configuration, and runtime behavior. The assessment employs industry-standard tools such as cargo audit**,** cargo clippy**,** Semgrep**,** bun audit**,** and custom WebSocket fuzzers to examine the resilience of the system against known vulnerabilities, unsafe coding practices, insecure configurations, and input-handling flaws. The project demonstrates a strong grasp of secure software engineering principles and idiomatic Rust design. Key architectural strengths include a robust Identifier abstraction that improves maintainability and correctness through type-safe parsing and custom serialization logic.

Strengths in Implementation

The project demonstrates a robust and well-engineered identifier abstraction that enhances both safety and maintainability: the Identifier enum cleanly separates broadcast, UUID-based, and bootstrap address semantics, while a dedicated Id newtype encapsulates UUID handling and enforces correct parsing and display via FromStr, TryFrom, and Display implementations. Custom Serde Serialize/Deserialize logic preserves a compact, predictable wire format (mapping "*" and explicit bootstrap strings) and includes a backward-compatibility mapping, which helps maintain protocol stability across versions. The code leverages idiomatic Rust derives (Clone, Debug, Eq, Hash) and explicit defaults, and is accompanied by thorough unit tests that cover serialization, parsing, constructors, and formatting providing strong regression protection and making it easier to reason about input validation and attack surface during security reviews. Together, these design choices reduce ambiguity in message encoding, simplify static analysis, and provide a solid foundation for safe extension and runtime verification.

Testing Approaches and Tools Used

1. Static analysis:

To assess the security posture of Group 42's Secure Chat Protocol project, I employed a combination of static analysis and dependency vulnerability scanning**.** For the Rust server code, I used cargo audit to examine all crate dependencies for known vulnerabilities in the RustSec advisory database. This scan detected one medium-severity vulnerability in the rsa crate (v0.9.8), associated with a potential timing side-channel attack that could leak cryptographic key material. To complement this, we ran cargo clippy, a Rust linter that checks for unsafe patterns, logical

errors, and pedantic issues; no warnings or unsafe code were reported, indicating that the server code is well-structured and free from common Rust coding pitfalls. For the frontend, which uses React and Tauri, we employed Bun's dependency audit tool (bun audit) to analyze installed JavaScript packages. This audit revealed two low-severity advisories in the vite dev-server, related to potentially serving files incorrectly during development mode. These issues are unlikely to affect production builds but highlight the importance of keeping development dependencies up to date. Collectively, these static and dependency-based testing approaches provide an initial assessment of the system's attack surface, focusing on known vulnerabilities in third-party libraries, unsafe code patterns, and misconfigurations. Figure 1 illustrates the output from cargo audit, highlighting the medium-severity vulnerability detected in the Rust server's rsa crate**.** *Figure 2* shows the results of bun audit, which identified two low-severity vulnerabilities in the frontend vite package.

Using **Semgrep** to statically analyze the code, three main issues were identified: (1) the server.dockerfile does not specify a non-root USER, which could allow processes in the container to run as root, (2) the server's WebSocket connections use insecure ws:// URLs instead of secure wss://, and (3) another instance of insecure WebSocket usage was detected in the bootstrap process for server-server communication. These findings highlight areas where security best practices are not followed; however, the missing non-root USER in the Dockerfile is not critical**,** as the container is intentionally designed for a controlled, educational environment and does not affect the security of the chat protocol itself. While the Dockerfile and WebSocket findings address configuration-level and transport-layer concerns, a deeper examination of the Rust source code was performed using Semgrep to uncover potential vulnerabilities at the application logic and cryptographic level. The tool was configured with 5–11 Rust-specific rules targeting unsafe parsing, deserialization, and cryptographic key handling. Across multiple scans, Semgrep produced a total of four actionable findings. Notably, it flagged *secure_chat::id::from_str* as unvalidated: unknown strings were accepted as Bootstrap(String) without proper validation, which could allow malformed or hostile addresses. It also identified unsafe handling of private key files in *secure_chat::crypto.rs*, including direct reading of PEM files without additional security checks and use of NamedTempFile to store key material, which may leave sensitive data exposed on disk. Several rules failed to parse due to limitations in pattern matching, but the successfully parsed findings represent real code risks.

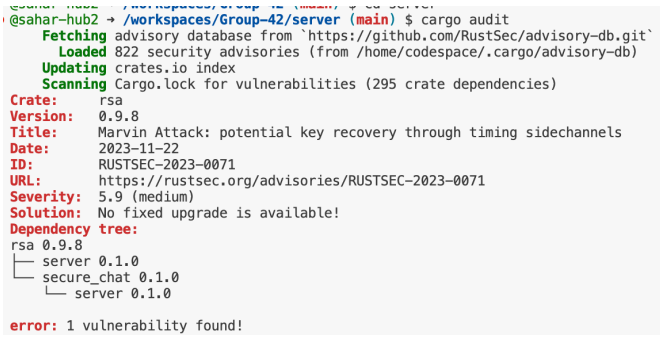

Figure 1: cargo audit output showing a medium-severity vulnerability in the Rust rsa crate.

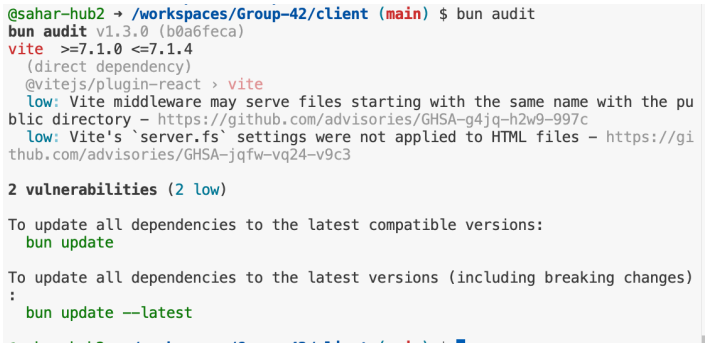

Figure 2: bun audit results revealing two low-severity vulnerabilities in the frontend vite package.

2. Dynamic fuzzing:

The dynamic testing workflow combined lightweight HTTP/WebSocket probes, an interactive WebSocket client, and a mutational fuzzer to evaluate both transport and parsing robustness. Using curl (raw HTTP/WebSocket handshake) and wscat/websocat (interactive WS client), I verified upgrade handling and origin enforcement. A crafted WebSocket upgrade request containing Origin: http://evil.example was accepted by the server, returning HTTP/1.1 101 Switching Protocols as shown in **Figure 3**, and establishing a live connection in wscat despite the forged origin, as illustrated in **Figure 4**. This confirms that the server permits WebSocket upgrades from arbitrary origins, exposing it to potential cross-origin attacks.

To further assess resilience, a lightweight web scanner (Nikto) was used to enumerate HTTP headers and revealed permissive or missing controls—specifically, the absence of X-Frame-Options and unenforced CORS policies within the WebSocket upgrade handler. Finally, a JSON mutational fuzzer (ws_fuzzer_safe.js) sent malformed, missing-field, and random-enum messages at high throughput. The server logs captured multiple parse errors such as invalid type: map, expected a string, unknown variant "RANDOM_123", and EOF while parsing an object, yet the application exhibited no panics or memory faults. Instead, it rejected invalid inputs gracefully and closed faulty connections cleanly. These results demonstrate strong error-handling resilience but highlight the need for stricter network-layer and input-validation controls.

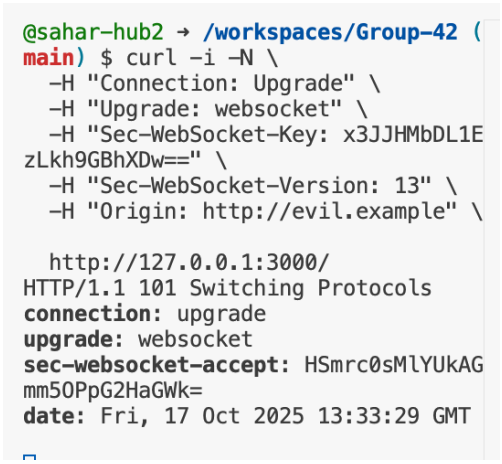

Figure 3: curl WebSocket upgrade handshake response including HTTP/1.1 101 Switching Protocols (server accepted the upgrade)

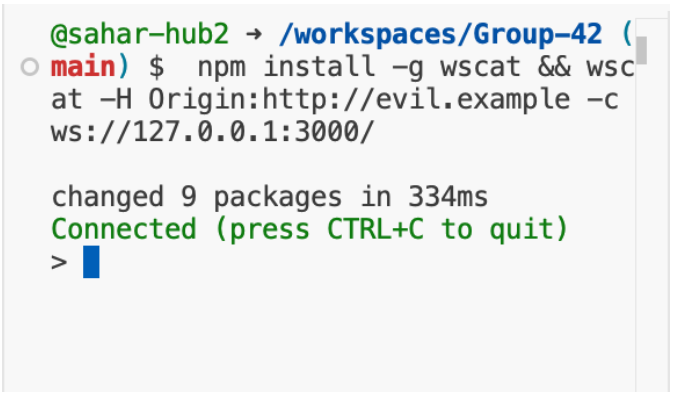

Figure 4: wscat client session showing Connected after supplying Origin: http://evil.example

Analysis of Test Results

Static dependency scanning identified one confirmed medium-severity vulnerability: the rsa crate v0.9.8, flagged by cargo audit for a timing side-channel (Marvin attack). In certain edge-case scenarios, this could leak information that weakens key secrecy. While this does not make the chat application trivially exploitable, an attacker capable of inducing carefully-timed signing operations and measuring timing externally could potentially extract sensitive key material. Mitigation options include upgrading to a patched rsa release when available, switching to a different actively-maintained RSA implementation, or adopting constant-time signing primitives—or moving to modern elliptic-curve algorithms with secure constant-time implementations.

Semgrep's static code analysis flagged configuration and code-level weaknesses. Insecure WebSocket usage (ws:// vs wss://) in bootstrap and announce code, and a Dockerfile run-as-root note, are operational risks rather than memory or authentication flaws. The insecure WebSocket URLs compromise network-layer confidentiality and integrity, allowing an active attacker to eavesdrop on or modify server-to-server messages. Recommendations include using wss:// (TLS) for inter-server connections and performing certificate or key pinning for bootstrap peers. Semgrep also highlighted real code vulnerabilities: the unvalidated FromStr fallback in secure_chat::id::from_str could allow untrusted input to be interpreted as a bootstrap identifier, potentially enabling internal network scans or address spoofing. Additionally, key-handling issues in secure_chat::crypto.rs—reading private keys directly from files without validation and storing temporary key files without restrictive permissions—expose the system to confidentiality risks if the filesystem is compromised. These should be remediated by enforcing strict parsing for identifiers (e.g., using SocketAddr), applying secure file permissions, avoiding persistent temp files, and improving secret management practices. Even though some Semgrep rules failed to parse, the successfully identified findings are genuine vulnerabilities.

Finally, the runtime findings show two distinct classes of issues. First, a confirmed operational vulnerability: the server accepts WebSocket upgrades from arbitrary origins (insufficient Origin/CORS validation) and lacks several common HTTP hardening headers. This allows cross-origin WebSocket connections — in practice an attacker-hosted webpage could open a WS to the server from a victim's browser (CSRF-style access) or probe protocol handlers remotely — and increases the risk surface if message-level authentication is incomplete. Second, the message-parsing code demonstrates robust defensive behavior: despite heavy fuzzing and many malformed messages, the server logged deserialization errors and closed connections without crashing or corrupting state, indicating strong use of Rust safety and good error handling. Recommended mitigations based on these dynamic results are: enforce a strict Origin allowlist (or require a secure token) at the WS upgrade handler and reject non-allowed origins with 403, switch inter-server transport to wss:// with peer verification/pinning, add standard HTTP security headers (e.g., X-Frame-Options, Referrer-Policy), and rate-limit or introduce progressive backoff on repeated parse failures to make fuzzing/probing more difficult..

Vulnerabilities

Vulnerability: Unvalidated FromStr Fallback in Identifier Parsing
Title: Accepting arbitrary strings as Bootstrap identifiers without validation
Severity: High (Protocol-level risk)
Location: `secure_chat/src/id.rs` → `from_str` function
Type: Input validation / Protocol flaw
CWE: CWE-20 (Improper Input Validation)
Summary: The `from_str` function treats unknown strings as `Bootstrap(String)` without proper validation, allowing untrusted input to be interpreted as bootstrap identifiers. This could enable address spoofing or internal network scans.
Evidence / How found: Detected via Semgrep rule `rust-unvalidated-fromstr-fallback`; review of the `from_str` code confirmed the fallback behavior.
Reproduction / Detection steps: Pass arbitrary string inputs to the server's identifier parsing function and observe that invalid or maliciously crafted inputs are accepted as `Bootstrap`.
Impact / Exploitation: An attacker could provide malicious identifiers, potentially probing or spoofing internal network nodes.
Mitigation: Enforce strict parsing, e.g., require valid `SocketAddr` for identifiers; reject malformed input rather than defaulting to bootstrap.
Confidence: High (confirmed in code).

Vulnerability: Unvalidated FromStr Fallback in Identifier Parsing
Title: Accepting arbitrary strings as Bootstrap identifiers without validation
Severity: High (Protocol-level risk)
Location: `secure_chat/src/id.rs` → `from_str` function
Type: Input validation / Protocol flaw
CWE: CWE-20 (Improper Input Validation)
Summary: The `from_str` function treats unknown strings as `Bootstrap(String)` without proper validation, allowing untrusted input to be interpreted as bootstrap identifiers. This could enable address spoofing or internal network scans.
Evidence / How found: Detected via Semgrep rule `rust-unvalidated-fromstr-fallback`; review of the `from_str` code confirmed the fallback behavior.
Reproduction / Detection steps: Pass arbitrary string inputs to the server's identifier parsing function and observe that invalid or maliciously crafted inputs are accepted as `Bootstrap`.
Impact / Exploitation: An attacker could provide malicious identifiers, potentially probing or spoofing internal network nodes.
Mitigation: Enforce strict parsing, e.g., require valid `SocketAddr` for identifiers; reject malformed input rather than defaulting to bootstrap.
Confidence: High (confirmed in code).

Vulnerability: Unsafe Private Key Handling
Title: Reading and storing private key material insecurely in temporary files

Severity: High (Confidentiality risk)
Location: `secure_chat/src/crypto.rs` → private key handling and `NamedTempFile` usage
Type: Cryptographic key management flaw
CWE: CWE-922 (Insecure Storage of Sensitive Information)
Summary: Private keys are read directly from PEM files without validation and stored in temporary files created without restrictive permissions. This could expose sensitive key material if the filesystem is compromised.
Evidence / How found: Detected via Semgrep rules `rust-read-private-key-file` and `rust-tempfile-key-usage`; manual inspection confirmed direct file reads and temporary file usage.
Reproduction / Detection steps: Examine the server's key-handling functions; attempt to access temporary files containing keys on the filesystem.
Impact / Exploitation: An attacker with access to the filesystem could recover private keys, compromising encrypted communications.
Mitigation: Apply secure file permissions, validate key files, avoid persistent temporary files, and adopt secure secret-handling practices.
Confidence: High (confirmed in code).

Vulnerability: Insecure Temporary File Creation in Tests
Title: NamedTempFile::new() creates temporary files with default, non-restrictive permissions
Severity: Medium (Information disclosure risk in testing or debug environments)
Location: secure_chat/src/crypto.rs — test module using NamedTempFile::new()
Type: Insecure temporary file handling
CWE: CWE-377 (Insecure Temporary File)
Summary: The use of NamedTempFile::new() in test code creates temporary files with default system permissions. On some systems, these files may be readable by other local users, potentially exposing cryptographic material or test data.
Evidence / How found: Detected by Semgrep rule rust-insecure-tempfile; confirmed by manual review of test code using NamedTempFile::new() without setting explicit file permissions.
Reproduction / Detection steps: Run tests and inspect /tmp or system temp directories; observe temporary key/test files with default permissions accessible by other users.
Impact / Exploitation: Local attackers could read residual temporary files containing sensitive or cryptographic material during or after test execution.
Mitigation: Use tempfile::Builder to set restrictive permissions (0o600) or explicitly delete temporary files after use. Avoid using unprotected temp files for key material, even in testing.
Confidence: Medium-High (confirmed in test code, limited to local exposure).

Vulnerability : Timing Side-Channel in RSA Signing
Title: Medium-severity timing leak in RSA crate v0.9.8
Severity: Medium (Potential cryptographic key leakage)
Location: Rust server → `rsa` crate v0.9.8 used in cryptographic signing operations
Type: Cryptographic timing side-channel
CWE: CWE-203 (Timing Attack)
Summary: The RSA signing implementation in the `rsa` crate may allow attackers to infer key material through timing measurements of signing operations.

Evidence / How found: Detected via `cargo audit` which flagged the known advisory in RustSec database. Manual code inspection confirmed use of vulnerable crate version in production cryptographic code.
Reproduction / Detection steps: Execute signing operations repeatedly and measure precise response timing; a remote attacker could analyze timing variations to extract partial key information.
Impact / Exploitation: An attacker capable of accurate timing measurements could gradually reconstruct secret keys, compromising encrypted communications or signatures.
Mitigation: Upgrade to a patched version of the `rsa` crate or replace with a constant-time signing implementation; consider migrating to modern elliptic-curve cryptography.
Confidence: High (confirmed in code and advisory database).

Vulnerability : Insecure WebSocket URLs
Title: Use of `ws://` instead of `wss://` for server connections
Severity: Medium (Potential network eavesdropping / message tampering)
Location: Rust server → bootstrap and inter-server communication code
Type: Insecure network transport
CWE: CWE-319 (Cleartext Transmission of Sensitive Information)
Summary: WebSocket connections are constructed using plain `ws://` URLs, which do not provide encryption or integrity protection.
Evidence / How found: Semgrep flagged usage of `ws://` URLs in bootstrap and server-server messaging routines. Manual inspection confirmed these are used for inter-server communication.
Reproduction / Detection steps: Intercept traffic between servers on the network; observe that messages are unencrypted and can be read or modified by a man-in-the-middle.
Impact / Exploitation: An active network attacker could eavesdrop on messages, modify contents, or inject spoofed messages, potentially compromising chat integrity or exposing sensitive session data.
Mitigation: Use `wss://` (TLS-secured WebSocket) with certificate validation for all network connections.
Confidence: High (confirmed insecure configuration and exploitable over network).

Short aggregated guidance :

- **Enforce strict input validation:**
  Update the from_str implementation in secure_chat::id.rs to reject arbitrary strings and validate inputs against explicit SocketAddr or UUID formats.
- **Harden key management practices:**
  Secure temporary file handling with restrictive permissions, avoid using NamedTempFile for sensitive material, and consider an in-memory or encrypted key store.
- **Secure transport channels:**
  Replace ws:// connections with wss:// (TLS) for all inter-server and bootstrap communications, and enable certificate or key pinning for mutual authentication.

- **Implement runtime security controls:**
  Enforce strict Origin/CORS validation in the WebSocket upgrade handler and add HTTP headers such as X-Frame-Options, Referrer-Policy, and Strict-Transport-Security.
- **Address dependency vulnerabilities:**
  Upgrade or replace the vulnerable *rsa* crate and adopt constant-time cryptographic primitives to prevent timing attacks.

Summary and Overall Feedback

Group 42's project exhibits a well-engineered and security-focused design, demonstrating clear modular organization, robust type safety, and effective use of Rust's ownership and trait systems to prevent common programming errors. The codebase reflects a strong understanding of secure software design principles, particularly through its structured identifier abstraction and use of custom serialization and parsing logic to maintain protocol consistency and prevent ambiguity in message handling. Static analysis identified several areas for improvement, including the use of an outdated and vulnerable RSA crate susceptible to timing side-channel attacks, unvalidated string parsing in the FromStr implementation that could permit spoofed identifiers, and insecure handling of private keys and temporary files. Additionally, insecure WebSocket connections (ws:// instead of wss://) were observed in bootstrap and inter-server communications, creating potential exposure to eavesdropping and tampering. Dynamic probing confirmed that the server remains stable and safe against malformed or adversarial input but uncovered a genuine configuration vulnerability: the WebSocket server accepts connections from arbitrary origins, leaving it open to cross-origin exploitation or CSRF-style attacks. Overall, the implementation is technically mature, leveraging Rust's strengths effectively and showing careful design in core logic and error handling. However, improving input validation**,** transport-layer security**,** and cryptographic key management will be crucial for hardening the system against practical attacks and ensuring protocol integrity.