Reflective Commentary - Secure Overlay Chat Protocol (SOCP)

Developed by Group 12 for the University of Adelaide Security Programming course, 2025

Introduction: Context and Purpose

This report reflects on Group 12's experience developing the Secure Overlay Chat Protocol (SOCP v1.3) for the Advanced Secure Programming assignment. It focuses on the design and implementation of a secure, standards-compliant version, alongside a deliberately backdoored version for controlled peer review. The commentary critically examines protocol decisions, cryptographic integrity, key management, testing, interoperability, and lessons learned from feedback, while addressing ethical considerations, assumptions, and challenges encountered during the creation and evaluation of vulnerabilities.

Reflection on the Standardised Protocol

The Secure Overlay Chat Protocol (SOCP) represents a robust, end-to-end encrypted communication framework designed for decentralized networks. Its n-to-n server mesh topology ensures that each server maintains knowledge of its local users while relying on a network-wide directory for routing messages securely between servers. Every payload is encrypted using RSA-4096 with OAEP for encryption and PSS for signatures, guaranteeing confidentiality and integrity without relying on symmetric key distribution. SOCP leverages WebSocket (RFC 6455) as the mandatory transport layer, providing real-time, bidirectional communication for both server-to-server and user-to-server messaging. Initially, our group considered TCP due to its straightforward, low-level connections and minimal overhead, reflected in early votes. However, during implementation we realized TCP alone would complicate browser compatibility, asynchronous messaging, and reliable propagation of public channel broadcasts. The voting ultimately favoured WebSocket, recognizing that standardizing on them simplifies client implementation, ensures interoperability, and better supports RSA-4096 encrypted payloads, content signatures, and fan-out of public channel messages. This highlights the tension between low-level control and practical deployment, showing how browser support, persistent connections, and secure broadcast channels decisively shape protocol adoption.

Implementing SOCP gives teams hands-on experience with distributed network coordination, end-to-end encryption, and secure message routing, highlighting the challenges of building a scalable, interoperable chat system. Its design enforces strong security RSA-4096 encryption, signed messages, and transport-level verification, while WebSocket enable persistent, browser-compatible bidirectional messaging, and gossip-based presence updates support fault-tolerant delivery. Trade-offs include higher computational overhead from RSA-only encryption, limited flexibility due to the mandatory public channel, and added complexity in managing per-user keys and broadcast fan-out. Multi-server routing and gossip updates may also introduce latency in larger networks. Overall, SOCP provides a rigorous, security-first framework that balances standardization, reliability, and real-world implementation challenges.

Backdoor Design and Exploitation

- **Weak-key acceptance** A conditional flag in the backdoored keys.py/key registration flow permits registration of RSA keys significantly weaker than the mandated RSA-4096. The backdoored PoC demonstrates registering a 1024-bit key and having the system accept it as if it were a 4096-bit key. Rationale: Purpose is to show how cryptographic strength is only as good as key management and verification. It is often assumed "if the algorithm is strong (RSA-OAEP/PSS) then the system is secure." Accepting weak keys demonstrates that logical or policy-level mistakes entirely undermine cryptography. How it’s subtle and plausible: The backdoor is subtle because all runtime messaging still uses RSA-OAEP and RSASSA-PSS APIs. Inspecting traffic looks encrypted, signed; only the key length verification step is lax. A reviewer who checks only "RSA is used" without verifying key sizes or certificate policies will miss the weakness. The system's outward behaviour remains functional; no immediate errors, which makes the flaw plausible in real-world deployments where legacy devices produce weak keys. Impact: Accepted weak keys allow offline cryptanalysis or key-forgery attacks, leading to impersonation, decryption, or signature forgery against participants that assume key strength. Evidence: poc_weak_key_register.py demonstrates registering a weak key and having the system accept it. Use this PoC to reproduce the behaviour in a controlled lab.

- **Unsigned gossip acceptance** the backdoored server.py contains a debug-mode conditional that, when enabled, allows acceptance and relay of USER_ADVERTISE and other introducer gossip frames without successful signature verification. This effectively disables the canonical signature check for federated advertisements and allows arbitrary adverts to be injected into the network. Rationale: Purpose is to demonstrate how a small logic change (bypassing one verification condition) can defeat an otherwise robust signature-based security model. The aim is to help reviewers see how trust assumptions in the gossip flow are fragile when signature verification is not enforced end-to-end. How it’s subtle and plausible: The gossip pipeline still uses the same JSON envelope format and signature fields; only the signature verification gate is bypassed under an internal flag. Reviewers may see normal envelope shapes and assume signature checks are present if they don't inspect the verification branch. Because messages continue to flow correctly, the issue is easily overlooked during functional testing. Impact: An injected unsigned advert can be propagated across servers, enabling attacker-controlled identities to be trusted and used for impersonation or message routing attacks. Evidence: poc_inject_unsigned_advert.py demonstrates how a crafted unsigned advert can be injected and propagated when the backdoor flag is active Detection and Mitigation • (Weak-key acceptance) Enforce cryptographic policy centrally (reject keys < RSA-3072 at import/generation; fail CI on violation). Mitigation can be done by removing runtime flags that short-circuit verification; make any debug bypass compile-time and blocked from production. • (Unsigned gossip acceptance) Centralize signature verification into an auditable function and add unit/property tests that assert every relayed advert passed verification. By adding CI regression tests (unsigned/invalid adverts, sub-policy keys) and log verification anomalies, this issue can be mitigated. Ethical considerations and safeguards applied the backdoors are intentionally confined to a clearly labelled backdoored_version/ and documented in BACKDOOR_README.md with explicit instructions to run only in isolated lab environments. Documented the PoCs and quarantine instructions to avoid accidental deployment. See *Appendix A — Backdoor Design and Exploitation (PoC)* for a lab-only reproduction checklist and PoC steps.

Evaluation of Received Feedback

All reviewing group successfully detected the two intentional backdoors - BACKDOOR_WEAK_KEYS (RSA-1024 key downgrade) and BACKDOOR_TRUST_GOSSIP (unsigned gossip acceptance). Reviewers demonstrated these through the included PoC scripts poc_weak_key_register.py and poc_inject_unsigned_advert.py, verifying that the backdoors allowed weak key generation and unsigned inter-server messages, leading to potential impersonation or data compromise. Several groups clearly described how environment variables gated these vulnerabilities and rated both as critical. They confirmed the educational intent was well-documented through BACKDOOR_README.md.

Reviewers also reported some additional weaknesses:

- SQL Injection risks through f-string–based queries in datavault.py.
- Weak password hashing (SHA-256 + salt only) and the presence of hardcoded default credentials. • Repository hygiene issues, such as committed .keys/ and data_vault.sqlite files, exposing sensitive data.
- Code-quality concerns, including missing docstrings, long functions, and import disorganization. Several reviewers went further and supplied fix examples, such as validate_key_size() checks, PBKDF2 password hashing, and structured exception logging.

The feedback was clear, specific, and actionable. All teams explained how to reproduce the vulnerabilities and offered direct remediation code. Their combined use of Bandit, Semgrep, and manual inspection revealed a valuable comparison: static tools failed to detect the intentional logic-level flaws, whereas manual reasoning achieved full success. The reviewers' suggestions directly influenced the secure rebuild: environment-variable gates were removed, RSA 4096-bit enforcement was added, PBKDF2 password hashing implemented, and .gitignore updated to exclude secret artifacts. Overall, the feedback process substantially improved both the robustness of our secure version and our understanding of how structured peer evaluation strengthens secure software development practices.

Feedback Provided to Other Groups

Sahar Hassan Alzahrani

The following feedback was provided by Sahar Hassan Alzahrani to groups 29, 100, and 42 on their secure chat system projects. Feedback for Group 29's chat system highlighted strengths in design and modern library usage, while identifying vulnerabilities such as hardcoded keys, insecure cookies, unsafe logging, and unencrypted WebSocket. A key challenge in this review was that I had not previously worked with some of these modern libraries, and the project included multiple components, including a frontend, which required additional effort to understand and correctly apply static and dynamic analysis tools. For Group 100's Secure Overlay Chat System, the review emphasized robust thread-safe design, structured message headers, and session-specific keys, yet critical issues were identified, including unauthenticated debug backdoors, weak MD5-derived IVs, binding to all network interfaces, broad exception handling, and input-validation deficiencies. A primary challenge in this case was interpreting complex runtime behaviours and reproducing crashes safely, which I addressed through focused fuzzing harnesses, manual code tracing, and embedding reproducible examples. Transitioning to Group 42, their Rust-based Secure Chat Protocol demonstrated strong type safety, structured identifier abstractions, and resilient error handling under fuzzed input. Static and dynamic analyses revealed timing side-channels in RSA signing, insecure WebSocket usage, unvalidated FromStr parsing, and weak key management, underscoring the importance of rigorous input validation, secure transport channels, and robust cryptographic hygiene. The challenge in reviewing this group was navigating Rust-specific idioms and cryptographic abstractions while maintaining clarity in recommendations.

Abidul Kabir

During the implementation of the secure chat system, the following were some of the feedback given to group 69, group 41 and group 25 by Abidul Kabir. Group 69 introduced a modular system that was well structured with working encryption and message handling. Nevertheless, the hardcoded or weak RSA parameters, the absence of TLS support, and the absence of authentication handshakes along with the risk of unsafe SQL query construction were found using both the static analysis (Bandit) and the manual one. The excessive exception silencing and absence of documentation was also identified by the code quality checks.

The Group 41 also offered a test mode, which posed severe threats of 1024-bit RSA keys, disabled replay protection and allowing identity registration. Although Bandit was able to detect only minor problems, these backdoors were identified during the manual review. The ability to distinguish between deliberate flaws and actual design weaknesses was another major problem, which was resolved by using ethical testing and config inspection.

Group 25 possessed a secure functional set up. Some of the key weaknesses were absence of TLS, poor key enforcement, and insecure validation with assert statements. The system failed to authenticate identities correctly and lacked important security functions such as message validation and formatted error processing. An inspection was performed both manually and using the static tools and showed several input validation and transport-layer vulnerabilities.
The main obstacle that was evident in all reviews was the correlation of the findings of tools with real-life attack situations. This was checked with combined Bandit, Pylint, manual tracing and runtime experiments.

Debasish Saha Pranta

Debasish Saha Pranta (a1963099) the following were feedback as given to Groups 38, 45, and 77 regarding their Secure Chat System projects. The Group 38 feedback on Secure Chat Architecture received positive mentions on the strong modular design and layered security concepts. The system combined an Introducer, peer servers and the clients who are in communication with each other via encrypted WebSocket messages. The strengths were good documentation, architectural separation that was professionally realized, and the use of RSA-based authentication. Nevertheless, construction problems like lack of some methods generated by Lombok did not allow full execution and dynamic testing. Weak input validation, absence of authentication on

WebSocket endpoints, disclosing keys and identifiers in logs, and inadequate error handling were some of the vulnerabilities that were detected during the process of performing static inspection. The key problem in this review was to diagnose errors in the compilation and test functionality without executing the complete system, which meant that it necessitated further manual examination and reasoning of the intended security flows.

In the case of Group 45, Secure Overlay Chat Prototype (SOCP), the review highlighted a well-written and readable Python-based implementation whose architecture is defined as a multi-server (master-local-client) architecture. The presence of good logging, modular design and understanding of the handling of public/ private keys were viewed as strengths. However, instability during the runtime, unfinished message processing and such essential weaknesses as the absence of validation of peer connections, plaintext-only message transfer, rate limiting and excessive logging of sensitive information have been observed. Dynamic testing revealed problems with the disconnection of clients and maintenance of user state. The key difficulty in this case was the consistency of reproducing these runtime crashes, long distance communication between servers, which was alleviated in the process of repetitive testing and debugging.

Lastly, the Secure Chat System by Group 77 showed good mastery of the cryptography and modular architecture. RSA-OAEP encryption and PSS signatures were well used in the project, which is an indication of good cryptographic hygiene. Such positive factors as replay protection, efficient file transfer using chunks, and structured logs that helped to analyze the logs were mentioned. Nonetheless, some weaknesses were identified such as unauthenticated introducer broadcasts, poor inter-server authentication, sensitive logging and absence of rate-limiting when it comes to file transfers. It was tested that there was a risk of denial-of-service and spoofing on federation channels. The main dilemma was to confirm these risks on federated components and how to comprehend a message propagation over a sequence of trust layers, which was tackled by a thorough static analysis and simulation of the runtime scenario.

In all reviews, Debasish showed that he used static inspection, limited runtime analysis, and manual reasoning to evaluate security, correctness and architectural soundness. The aggregate knowledge supported the paramount role of input validation, secure key management, authenticated communication, and resilience testing of constructing secure distributed chat systems.

Samin Yeasar Seaum

Samin Yeasar Seaum (a1976022) the following were feedback as given to Groups 43, 69, and 70 regarding their Secure Chat System projects.

Groups Reviewed: I offered formal peer reviews to Group 43, Group 69 and Group 70. The workflow of each review was identical: Pylint and Bandit were used to perform the static analysis, the codebase was examined manually, and the SOCP v1.3 compliance was checked as well as a list of recommendations issued by the professional feedback report was created.

Feedback Provided: In the case of Group 43, I concentrated on maintainability and compliance to the specification of SOCP. I reported two educational backdoors, two messages of the form: unsigned USER_ADVERTISE and weak key registration and suggested more rigorous checking signature validation, RSA-4096, and better exception handling.

In the case of Group 70, I have reviewed 4 backdoors: hard-coded secrets, placeholder signatures, weak default on RSA-keys, and binding to all interfaces. The feedback focused on the best practices in secret management, signing messages securely, exposure control in the network, and continuous-integration testing.

In the case of Group 69, I checked a more complicated node-based design and database layer. I found plaintext WebSocket use, weak key-acceptance, silent signature-verification failures and dynamic SQL risks. The recommendations I made were migration to wss:// transport, key-strength validation, fail-closed logic, and strict SQL parameterisation.

The common strengths found in all reviews included modular architectures, clear cryptographic knowledge, and coherent code style alongside incremental and viable security and maintainability enhancement.

Challenges Faced: The key issues were (1) the large, heterogeneous codebases that are based on different frameworks, (2) the separation of intentional educational vulnerabilities and real oversights, and (3) the interpretation of incomplete documentation of cryptographic flows.

How I Overcame Challenges: To ensure impartiality and consistency, I normalised the review process into repetitive phases, such as automated scanning, manual checks, compliance mapping, and report creation. I have also referred to the SOCP v1.3 brief and project rubric to establish which weaknesses were deliberately created. Questions to classmates on the discussion board helped in resolving vague decisions. Any analysis was conducted in closed system to prevent the risk of security.

Reflection: These reviews enhanced my practical skills in conducting secure-coding analysis, cryptographic enforcement, and positive feedback. It helped me to be more critical and supportive at the same time and to present technical findings in a clear and professional way, which is important in both collaboration and software development and in the actual cybersecurity evaluation.

Mahrin Alam Mahia
The following feedback was provided by Mahrin Alam Mahia (a1957342) to Groups 37, 101, and 97 on their Secure Overlay Chat System projects.
For Group 37, the review emphasized a well-functioning asynchronous chat overlay with correct RSA-OAEP and PSS usage, yet identified unsigned inter-server messages, auto-trust in Zeroconf discovery, and weak enforcement of key-import policies as major risks. Pylint (8.47/10) and Bandit scans confirmed overall good code quality after formatting, with earlier high counts traced to third-party dependencies.

For Group 101, the Python-based system correctly implemented end-to-end encryption and message signing, and the included proof-of-concept exploits successfully demonstrated intentional backdoors such as weak RSA key acceptance, routing-poison and replay vulnerabilities, and missing file-integrity checks. Pylint (7.94/10) and Bandit (low risk post-exclusion) corroborated minor operational weaknesses—broad exceptions, global state, and missing encodings—while a ConnectionClosedOK event highlighted runtime robustness limits rather than a security flaw.

For Group 97, the C++20 implementation demonstrated strong architectural design, correct RSA-4096 encryption, and well-managed dependencies, but lacked bootstrap signature validation, replay protection persistence, and TLS transport. cppcheck and clang-tidy flagged input-sanitization and logging deficiencies consistent with manual review. Overall, these reviews deepened understanding of cross-language security enforcement—from Python's runtime safety gaps to C++'s manual trust management—while reinforcing the need for validated introducer trust, replay defenses, and consistent cryptographic hygiene.

Engaging with diverse codebases across Python and C++ implementations highlighted how each language's security posture depends on disciplined design, testing, and documentation. This review process strengthened the understanding of secure protocol design, emphasizing that even well-structured systems can conceal subtle weaknesses in trust models, key management, and exception handling.

Maria Hasan Logno
Maria Hasan Logno (a1975478) has provided the feedback peer review to the following groups 26, 88 and 91.

Group 26 has developed a real-time chat system using WebSocket with claimed RSA/AES encryption. The system supports user authentication, private/group messaging, file transfer, and online user listing. However, the implementation exhibits significant security vulnerabilities and architectural weaknesses that fundamentally undermine its security posture.
Strengths:
- Clear documentation and setup instructions in README files
- Well-structured protocol definition in protocol.json
- Comprehensive feature set including file transfer and group messaging
- Proper use of asynchronous programming with asyncio and WebSocket
- Implementation of heartbeat mechanism for connection monitoring Critical Security

Weaknesses:
- Missing Encryption Implementation
- Despite claims of "RSA + AES encryption," the code contains no cryptographic implementation
- All messages are transmitted in plaintext over unencrypted WebSocket (ws://)
- No key generation, encryption, or decryption logic present in server code

Group 91 has developed a comprehensive WebSocket-based chat system with multiple security features and a well-structured project architecture. The implementation demonstrates good software engineering practices with proper documentation, dependency management, and database integration.
Strengths: Excellent documentation with clear setup instructions and troubleshooting guidance Proper use of requirements.txt for dependency management Comprehensive feature set including file transfer and multiple messaging types Database integration with SQLite and proper initialization scripts Support for multiple server instances with configuration files Implementation of user authentication with password requirements good project structure with separate server/client modules

Security Concerns Critical: README incorrectly states "Message encryption using SHA256" - SHA256 is a hashing algorithm, not encryption. This fundamental misunderstanding raises concerns about cryptographic implementation. Missing details on actual encryption methodology for message security Database initialization process could benefit from automated scripts rather than manual SQLite commands No mention of transport security (TLS/SSL) for WebSocket connections Areas for Improvement Clarify the actual encryption approach used for message security Consider implementing automated database setup scripts Add transport layer security for production deployment Include more details about the cryptographic architecture in documentation

Overview Group 88's EchoChat presents a sophisticated distributed chat system with a modern web frontend and robust backend architecture. The project demonstrates strong software engineering practices with comprehensive documentation and a well-structured technology stack. Strengths Excellent Architecture: Clean separation of concerns with dedicated servers for WebSocket communication, file handling, and authentication Modern Technology Stack: Use of Maven, Java 11, JWT authentication, and Vue.js frontend shows contemporary development practices Comprehensive Documentation: Clear quick-start guide with demo accounts and multiple operation modes (web and CLI) Security Features: JWT authentication, rate limiting, and SOCP protocol with encryption support Professional Configuration: Well-structured pom.xml with appropriate dependencies including Bouncy Castle for cryptography
Security Concerns Weak Demo Passwords: "demo123" passwords for all accounts represent a security risk and poor practice Debug Endpoint Exposure: /api/debug/users endpoint could leak sensitive user information in production Large File Uploads: 256MB maximum file size could enable denial-of-service attacks Missing Transport Security: No mention of TLS/SSL implementation for production deployment
Areas for Improvement Implement stronger password policies and remove hardcoded demo credentials Secure or remove debug endpoints in production environments Add file type validation and virus scanning for uploads Document encryption implementation details and key management Consider implementing end-to-end encryption for enhanced privacy

Full individual reports and evidence are attached in the *Appendix C*.

Reflection on AI Use

During this project, we used AI tools such as ChatGPT mainly to help with documentation, debugging, and clarifying concepts around encryption and network protocols. It was especially useful for explaining technical ideas in simpler terms, identifying coding errors, and suggesting ways to structure functions more clearly. Having quick, well-organised guidance made it easier to stay productive and maintain consistency across the project.

At the same time, we learned that AI has its own limits. It sometimes gave suggestions that didn’t fit the project’s context or referred to functions that didn’t exist. In areas like asynchronous communication and cryptography, we found that relying purely on AI outputs could be risky. Every recommendation needed careful testing and review to ensure it was correct and secure.

From an ethical perspective, we used AI responsibly, treating it as a learning and drafting aid, not as a replacement for our own work. It helped us learn faster, but human oversight remained essential. Overall, using AI felt like having a knowledgeable assistant, helpful for brainstorming and explanation, but never a substitute for real understanding. This experience reminded us that in Secure software development, critical thinking and verification are just as important as technical skill.

Reflection on testing

The security testing of secure_version was designed to comprehensively ensure both the correctness and resilience of the system. It began with static and dynamic analyses to detect potential vulnerabilities, including weak RSA keys, unsafe file handling, unsanitized inputs, missing signature verification, and high-severity issues flagged by Bandit. Building on this, integration tests validated end-to-end encrypted communication, message delivery, replay detection, and the persistence of cryptographic keys across sessions, providing a systems-level verification of protocol integrity. Complementary server and client module tests confirmed proper cryptographic signing, error handling, message routing, and protocol compliance under realistic scenarios, while cryptography utility tests ensured correct key generation, encryption/decryption, signature operations, and robust handling of malformed or invalid inputs. Security hardening checks reinforced this foundation by confirming the absence of backdoors, enforcing minimum key sizes, and verifying safe SQL parameterization and input validation. Additionally, vulnerabilities identified by Snyk, including path traversal and insecure dependency handling, were addressed to safeguard file operations and TLS authentication. Together, these carefully orchestrated testing activities provide a thorough and reliable evaluation of the system's security and functionality, with the full detailed test plan and results available in the appendix B for reference.

Reflection on testing interoperability

We experimented with interoperability of our Group 12 implementation of SOCP and Group 43 system with a shared introducer. Our introducer and server started correctly, but as the server of Group 43 tried to connect, it gave a reply of Unhandled message type from introducer: ERROR and this means that their introducer has limited capability that supports only one server. Their customer could also reach our server, but authentication failed as messages were not of the same format- our system has strict checks of timestamps and ID which their system lacked.

Such outcomes indicate that even though the two systems are based on the SOCP principles, the communication packet structure and the sequence of handshakes are slightly different. Our implementation can be federated to many servers, compared to the implementation in Group 43 which is a stand-alone network node. The testing, in general, has validated the resilience of our system and its compliance with standards, as well as given useful information about the differences in the protocols of independent implementations. Evidence is attached in the *Appendix C.*

Group Contributions

All members jointly contributed to the system's design, secure implementation, testing, and documentation.

- **Debasish Saha Pranta:** Led introducer and federation setup; co-developed and tested backdoor scenarios.

- **Samin Yeasar Seaum:** Focused on file transfer and cryptography; co-designed and validated backdoor logic.
- **Abidul Kabir:** Worked on networking and persistence; supported protocol integration and backdoor testing.
- **Sahar Alzahrani:** Developed the client framework, led testing and documentation; coordinated security validation.
- **Mahrin Mahia:** Drove backdoor PoC design and analysis; contributed mitigation and ethical safeguards.
- **Maria Hasan Logno:** Managed integration and interoperability testing; compiled testing evidence and appendices.

Conclusion

Working on the Secure Overlay Chat Protocol (SOCP) was a genuine learning curve that took us from simply building a working encrypted system to realising how easily human error can weaken even the strongest designs. By intentionally adding backdoors and later reviewing them through peer feedback, we saw how small lapses like a skipped validation, a misplaced check, or an unchecked configuration can quietly undermine the security we thought was airtight.

We learned the value of strict protocol discipline and the need to maintain clear trust boundaries between servers, clients, and introducers. It also reinforced the importance of writing code ethically; being transparent, documenting decisions, and isolating any vulnerabilities for safe testing. Building and analysing SOCP gave us hands-on experience with cryptographic integrity, where every signature and key exchange must be open to verification and accountability. Moreover, working on both secure and deliberately flawed versions of SOCP helped us develop that balanced perspective.

Group 12:

1. Debasish Saha Pranta (a1963099, debasishsaha.pranta@student.adelaide.edu.au)
2. Samin Yeasar Seaum (a1976022, saminyeasar.seaum@student.adelaide.edu.au)
3. Abidul Kabir (a1974976, abidul.kabir@student.adelaide.edu.au)
4. Sahar Alzahrani (a1938372, sahar.alzahrani@student.adelaide.edu.au)
5. Mahrin Mahia (a1957342, mahrin.mahia@student.adelaide.edu.au)
6. Maria Hasan Logno (a1975478, mariahasan.logno@student.adelaide.edu.au)

Appendix

Appendix A — Backdoor Design and Exploitation (PoC):

Conceptual exploit steps: This should be reproduced only in an isolated lab environment.

A. Weak Key Registration

1. Configure a test server instance using the backdoored_version/ server (enable backdoor or run the PoC according to the quarantine notes).
2. Use a client that generates a weak RSA key pair (1024-bit in the PoC) and attempt to register/advertise the user key to an introducer or server.
3. If accepted, the introducer or servers will include the weak-key advertisement into their registry.
4. Attacker extracts the weak public key and (offline) attempts cryptanalysis.
5. Observe that servers or clients accept messages signed or encrypted under that weak key as if it were strong.

B. Unsigned gossip acceptance

1. Configure the backdoored server with gossip verification bypass enabled.
2. Craft a USER_ADVERTISE JSON envelope with an arbitrary user_id and public key info, leaving signature fields empty or invalid.
3. Send the envelope to one server or the introducer via the gossip endpoint.
4. The compromised server accepts the advert and relays it to peers; other servers add the advertised user to their local registries.
5. The attacker can now send messages claiming to be the advertised user or observe how servers route traffic for that fake identity.

More explanations are provided in the BACKDOOR_README.md file in the Appendix/.

Appendix B — Testing Evidence:

Testing Evidence: See Testing Report

Appendix C — Peer Review:

Peer Review: Debasish Saha Pranta – Group 38 Review
Peer Review: Debasish Saha Pranta – Group 45 Review
Peer Review: Debasish Saha Pranta – Group 77 Review
Peer Review: Samin Yeasar Seaum – Group 43 Review
Peer Review: Samin Yeasar Seaum – Group 69 Review
Peer Review: Samin Yeasar Seaum – Group 70 Review
Peer Review: Abidul Kabir – Group 25 Review
Peer Review: Abidul Kabir – Group 41 Review
Peer Review: Abidul Kabir – Group 69 Review
Peer Review: Sahar Hassan Alzahrani – Group 100 Review
Peer Review: Sahar Hassan Alzahrani – Group 29 Review
Peer Review: Sahar Hassan Alzahrani – Group 42 Review
Peer Review: Mahrin Mahia – Group 101 Review
Peer Review: Mahrin Mahia – Group 37 Review
Peer Review: Mahrin Mahia – Group 97 Review
Peer Review: Maria Hasan Logno – Group 26 Review
Peer Review: Maria Hasan Logno – Group 88 Review
Peer Review: Maria Hasan Logno – Group 91 Review