SOCP Secure Chat System Peer Review Report

Executive Summary

This report presents a comprehensive security analysis of the SOCP (Secure Federated Chat Protocol) implementation, an intentionally vulnerable system designed for peer review and educational purposes. Through static code analysis, dynamic testing, and controlled exploitation, I analyzed the system's architecture, deployed and tested the federation infrastructure, and successfully identified two deliberate backdoors that demonstrate real-world security risks.

The testing process involved documentation review, environment verification, application deployment, functionality testing, and systematic vulnerability discovery. The findings confirm that while SOCP demonstrates strong cryptographic fundamentals, it also contains intentionally weakened mechanisms that simulate exploitable vulnerabilities.

Testing Methodology

Phase 1: Static Analysis and Setup

Repository Structure
The repository was organized as follows:

```
Security-Programming/
├── server.py              # Main federated server
├── client.py              # Chat client with end-to-end encryption
├── keys.py                # RSA key generation and management
├── datavault.py           # SQLite database operations
├── gen_introducer_keys.py  # Key generation for introducers
├── introducers.yaml       # Introducer configuration file
├── quarantine/            # Proof of concept exploit scripts
└── .keys/                 # Pre-generated RSA keypairs
```

The presence of the quarantine directory indicated possible inclusion of proof-of-concept vulnerabilities.

Documentation Review
The following documentation files were examined:

- README.md: Project setup and overview
- README.txt: Technical details and author information
- BACKDOOR_README.md: Documentation of intentional vulnerabilities

From the documentation, I confirmed that the system implements RSA-4096 OAEP-SHA256 encryption, RSASSA-PSS signatures, multi-server federation through introducer nodes, and SQLite-based persistence.

Environment Verification
System requirements were confirmed to meet dependencies:

```
python3 --version     # Python 3.10.12
pip3 list | grep -E "(websockets|cryptography)"
# websockets 12.0, cryptography 43.0.0
```

Phase 2: Application Deployment and Dynamic Testing

Introducer Server Deployment
Following setup instructions, I launched three introducer servers:

```
python3 server.py --name introducer1 --port 9001 --introducer
python3 server.py --name introducer2 --port 9002 --introducer
python3 server.py --name introducer3 --port 9003 --introducer
```

Each server initialized correctly and began listening on its respective WebSocket address.

Federated Chat Servers
Two federated chat servers were deployed to test inter-server communication:

```
python3 server.py --name serverA --port 8765
python3 server.py --name serverB --port 8766
```

Both servers successfully registered with introducers, established federation links, exchanged heartbeats, and generated local SQLite databases.

Connectivity Verification
A short test confirmed WebSocket responsiveness:

```
async with websockets.connect("ws://127.0.0.1:8765") as ws:
    await ws.send(json.dumps({"type": "ping"}))
    response = await ws.recv()
```

Both servers responded correctly, verifying live connectivity.

Phase 3: Client Testing and Functional Verification

Cross-Server Chat Functionality
Client instances were launched to validate end-to-end communication:
```
python3 client.py --user alice --server ws://127.0.0.1:8765
python3 client.py --user bob --server ws://127.0.0.1:8766
```

Upon connection, the system displayed presence updates, confirming successful federation and user advertisement exchange.

Chat Command Testing
The following client commands were tested:
```
/list
/tell bob Hello from Alice!
/all Hello everyone!
/sendfile bob sample.txt
```

All commands performed correctly, verifying message encryption, federation, and file transfer features.

Database Inspection
An examination of the SQLite database confirmed persistent user storage:
```
cursor.execute("SELECT user_id, json_extract(meta,'$.display_name') FROM users")
```

The results showed five registered users, including one with a 1024-bit RSA key, suggesting potential weakness.

Phase 4: Vulnerability Discovery

Backdoor 1: Weak RSA Key Generation
Located in `keys.py:`
```
if BACKDOOR_WEAK_KEYS:
    return rsa.generate_private_key(65537, 1024, default_backend())
```

If the environment variable `BACKDOOR_WEAK_KEYS` is set, the system generates 1024-bit keys instead of 4096-bit ones. This significantly weakens encryption strength and allows key factorization attacks. Evidence of exploitation was found in a user record with a short RSA key length.

Backdoor 2: Unsigned Gossip Acceptance
Located in `server.py:`
```
if BACKDOOR_TRUST_GOSSIP:
    print("[BACKDOOR] Accepting unsigned gossip message")
```

When the BACKDOOR_TRUST_GOSSIP variable is set, the server accepts unsigned gossip advertisements, allowing malicious servers to spoof users or broadcast false presence data.

Exploitation Confirmation
Proof-of-concept exploit scripts located in the quarantine directory include:
- poc_weak_key_register.py
- poc_inject_unsigned_advert.py

The vulnerabilities can be activated by running servers with the following commands:

```
BACKDOOR_WEAK_KEYS=1 python3 server.py ...
BACKDOOR_TRUST_GOSSIP=1 python3 server.py ...
```

Security Assessment Results

Verified Security Features

- RSA-4096 OAEP-SHA256 encryption functioning correctly
- RSASSA-PSS message authentication implemented
- Server-to-server communication integrity verified
- Secure SQLite persistence for user credentials and messages

Identified Vulnerabilities

CVE-2025-WEAK-KEYS: Weak RSA Key Generation
Severity: High
Impact: Enables creation of easily factorable 1024-bit RSA keys, compromising encryption integrity
Detection: Static analysis and database inspection

CVE-2025-GOSSIP-TRUST: Unsigned Gossip Acceptance
Severity: Medium
Impact: Allows injection of unsigned gossip messages, enabling user spoofing
Detection: Code analysis and proof-of-concept testing

Attack Scenarios

1. Weak Key Exploitation: An attacker sets the weak key environment variable, generating factorable keys that enable message decryption and impersonation.

2. Gossip Injection: An attacker enables unsigned gossip mode, allowing unverified messages that can impersonate legitimate users.

Testing Documentation

A supporting script named test_chat.py was developed to automate WebSocket connectivity and validate server response handling. This script provides a reproducible testing method for verifying basic communication security.

Recommendations

Immediate Actions

1. Remove all environment-based backdoor toggles before production release.

2. Enforce runtime validation of RSA key lengths to ensure a minimum of 4096 bits.

3. Mandate cryptographic signature verification for all gossip messages.

4. Integrate a security review step into the code deployment pipeline.

Long-Term Security Enhancements

1. Integrate static analysis tools such as Bandit or Semgrep for continuous auditing.
2. Conduct cryptographic code audits on all key management functions.
3. Restrict environment variable usage for sensitive system configurations.
4. Expand automated test coverage to detect and prevent intentional or accidental vulnerabilities.

Conclusion

The SOCP Secure Chat System demonstrates sound cryptographic design and effective federation architecture. However, the intentionally introduced vulnerabilities illustrate how subtle backdoors can undermine an otherwise secure system. Through a combination of static analysis, dynamic testing, and database inspection, both vulnerabilities were successfully identified and validated.

This analysis confirms that while the system serves as an excellent educational model for security testing, any production deployment must remove all backdoor mechanisms, enforce strict key management policies, and apply consistent security review processes.

For any questions/clarifications, contact me: ayii.madut@student.adelaide.edu.au