Peer Review Report: Security-Programming

Reviewer: [Ching-Chun Wang / a1951826] **Review Date:** October 17, 2025
Repository: sahar-hub2/Security-Programming

Summary

This peer review evaluates a Secure Federated Chat Protocol (SOCP) implementation with end-to-end encryption using RSA cryptography and WebSockets. The review found that while the system implements core security features, it contains **intentional backdoors** for educational purposes and several **security vulnerabilities** that need attention.

1. Manual Testing

1.1 Functionality Testing

Core Features Implementation

Working Features:

- RSA key generation and management (4096-bit default)
- RSA-OAEP encryption (SHA-256) for end-to-end encryption
- RSA-PSS signatures (SHA-256) for message integrity
- Base64url encoding/decoding (no padding)
- WebSocket communication between client and server
- User registration and public key advertisement
- Direct encrypted messaging (`/tell` command)
- Broadcast messaging (`/all` command)
- User listing (`/list` command)
- File transfer capabilities (up to 50MB)
- SQLite persistence for user data
- Server-to-server federation support
- Introducer-based bootstrap mechanism

Protocol Compliance (SOCP v1.3):

- JSON envelope structure (type, from, to, ts, payload, sig)
- Message signing with RSA-PSS
- User UUID v4 identities (partially implemented)
- Server-to-server gossip protocol
- Some checklist items marked incomplete (see Checklist.md)

Missing/Incomplete Features:

- UUID v4 validation not consistently enforced
- Some federation features partially implemented

- Error handling could be more robust

Test Results Summary

Feature	Status	Notes
Key Generation	Pass	4096-bit RSA keys generated correctly
Encryption/Decryption	Pass	OAEP-SHA256 working as expected
Message Signing	Pass	PSS signatures verified correctly
Client-Server Comm	Pass	WebSocket connection stable
Direct Messages	Pass	E2EE working properly
Broadcast Messages	Pass	Signed but not encrypted (as designed)
File Transfer	Pass	Large file handling implemented
Persistence	Pass	SQLite database working

1.2 Ethical Backdoor Testing

BACKDOOR #1: Weak RSA Key Acceptance

Location: `keys.py` lines 79-86

Description:
When the environment variable `BACKDOOR_WEAK_KEYS=1` is set, the system generates **1024-bit RSA keys** instead of the secure 4096-bit keys, making them vulnerable to factorization attacks.

Code Evidence:

```
# BACKDOOR: downgrade key size when requested by environment variable
if os.getenv("BACKDOOR_WEAK_KEYS") == "1":
    # Intentionally weak 1024-bit RSA keys for demonstration/exercise
    priv_pem, pub_pem = _generate_rsa(1024)
    # Do not print the weakness by default — keep output minimal.
else:
    priv_pem, pub_pem = generate_rsa4096()
```

Exploitation Method:

1. Start server with: `BACKDOOR_WEAK_KEYS=1 python server.py --name serverA --port 8765`
2. Server generates weak 1024-bit keys
3. Attacker can factor the public key to derive private key
4. All encrypted messages become readable

PoC Verification:

- PoC script exists: `quarantine/poc_weak_key_register.py`
- Successfully registers users with 1024-bit keys
- Server accepts weak keys without warning (stealthy)

Impact: CRITICAL

- Complete compromise of encryption
- Attacker can decrypt all messages
- No visual indicator to users that weak keys are in use

BACKDOOR #2: Unsigned Gossip Acceptance

Location: `server.py` lines 952-969

Description:
When `BACKDOOR_TRUST_GOSSIP=1` is set, the server accepts `USER_ADVERTISE` messages **without signature verification**, allowing attackers to inject fake users into the network.

Code Evidence:

```
if not BACKDOOR_TRUST_GOSSIP:
    # Normal secure path: require sig and a known origin server pubkey
    if not sig_b64u or origin_sid not in server_addrs:
        return
    # ... signature verification ...
else:
    # BACKDOOR: when BACKDOOR_TRUST_GOSSIP=1, accept USER_ADVERTISE even
if unsigned
    # or signature doesn't match. For stealth, we only print a short
backdoor notice.
    if not sig_b64u:
        # unsigned advert — accepted under backdoor
        print(f"[BACKDOOR] accepting unsigned USER_ADVERTISE for {uid}
(origin={origin_sid})")
```

Exploitation Method:

1. Start vulnerable server: `BACKDOOR_TRUST_GOSSIP=1 python server.py --name serverA --port 8765`
2. Connect as malicious client
3. Send unsigned `USER_ADVERTISE` message with fake user details
4. Server accepts and broadcasts fake user to all clients

PoC Verification:

- PoC script exists: `quarantine/poc_inject_unsigned_advert.py`
- Successfully injects fake users without signatures
- Server logs show `[BACKDOOR]` acceptance message

Additional Security Concerns Found

1. Hardcoded Password (Medium Risk)

- **Location:** `server.py` line 402
- **Issue:** Default password `"default"` hardcoded
- **Impact:** Predictable credentials if used in production

2. SQL Injection Risk (Medium Risk)

- **Location:** `datavault.py` line 165
- **Issue:** String-based SQL query construction in debug function
- **Code:** `f"SELECT * FROM {table}"`
- **Impact:** Potential SQL injection in debug mode

3. Weak Password Hashing (Low-Medium Risk)

- **Location:** `datavault.py` lines 68-71
- **Issue:** Single SHA-256 hash with salt (no iteration count)
- **Recommendation:** Use PBKDF2, bcrypt, or Argon2

4. Excessive Exception Suppression (Low Risk)

- **Locations:** Multiple `except Exception: pass` blocks
- **Impact:** Silent failures make debugging difficult
- **Count:** 11 instances found by Bandit

---

2. Analysis with Tools

2.1 Code Quality Analysis (Pylint)

Server.py Results:

- **Module too long:** 1819 lines (limit: 1000)
- **Too many local variables:** Multiple functions with 15+ variables
- **Missing docstrings:** Many functions lack documentation
- **Style issues:** 50+ trailing whitespace, line length violations
- **Import organization:** Imports not at top of module
- **Broad exception catching:** Multiple `except Exception` blocks

Client.py Results:

- **Function redefinition:** `b64url_encode/decode` defined twice
- **Too many local variables:** `run_client` function with 34 variables
- **Too many branches:** 54 branches in message handler
- **Too many statements:** 334 statements in main function
- **Missing docstrings:** Core functions lack documentation
- **Unused imports:** `math` imported but not used

Keys.py Results:

- **Variable name shadowing:** `priv_pem`, `pub_pem` redefined in scopes
- **Missing docstrings:** Helper functions lack documentation
- **Import organization:** UUID import not at top
- **Broad exception catching:** Generic exception handling

Summary of Code Quality Issues:

Category	Count	Severity
Missing docstrings	15+	Medium
Trailing whitespace	50+	Low
Line too long	25+	Low
Broad exceptions	20+	Medium
Too many locals	5+	Low
Import issues	10+	Low

2.2 Security Analysis (Bandit)

Total Issues Found: 12

High Severity: 0

Medium Severity: 1

- **SQL Injection:** `datavault.py:165` - String-based query construction

Low Severity: 11

- **Try-Except-Pass:** 10 instances of silent exception suppression
- **Hardcoded Password:** 1 instance in server.py

Bandit Summary:

```
Total lines of code: 2352
Total issues (by severity):
    Undefined: 0
    Low: 11
    Medium: 1
    High: 0
```

Security Score: Medium Risk

Critical Findings:

1. No command injection vulnerabilities
2. No eval/exec usage

3. No pickle/marshal usage
4. SQL injection risk in debug code
5. Weak password storage mechanism
6. Silent exception handling could hide attacks

2.3 Backdoor Detection Results

Automated Detection Methods:

1. **Environment Variable Search:**

```
grep -r "BACKDOOR" *.py
```

   - Found: `BACKDOOR_TRUST_GOSSIP` in server.py (line 39)
   - Found: `BACKDOOR_WEAK_KEYS` in keys.py (line 80)

2. **Weak Cryptography Detection:**

```
grep -r "1024" *.py
```

   - Found: RSA 1024-bit key generation in keys.py (line 82)

3. **Signature Bypass Detection:**

   - Found: Conditional signature verification bypass (server.py:952-969)
   - Pattern: `if not BACKDOOR_TRUST_GOSSIP:` indicates conditional security

4. **Backdoor Documentation:**

   - `BACKDOOR_README.md` documents both backdoors explicitly
   - PoC scripts in `quarantine/` folder demonstrate exploits

Detection Effectiveness:

- **Manual Code Review:** 100% detection rate (both backdoors found)
- **Grep/Pattern Search:** 100% detection rate (environment vars are obvious)
- **Automated Tools (Bandit):** 0% detection rate (backdoors use legitimate crypto APIs)

Key Insight: The backdoors are **intentionally documented** for educational purposes, making them easy to find. In a real attack, such obvious markers would not exist.

---

3. Feedback and Recommendations

3.1 Positive Aspects

1. **Strong Cryptographic Foundation**

   - Correct implementation of RSA-4096, OAEP, and PSS

- Proper use of cryptography library APIs
- Base64url encoding follows spec correctly

2. **Good Architecture**
   - Clear separation of concerns (keys.py, server.py, client.py)
   - Modular design allows easy testing
   - Federation support shows advanced understanding
3. **Comprehensive Documentation**
   - README files provide clear setup instructions
   - Protocol documentation available
   - Backdoor documentation for educational review
4. **Educational Value**
   - Backdoors demonstrate real attack vectors
   - PoC scripts show exploitation methods
   - Good example of security vs. functionality trade-offs

3.2 Critical Issues

1. **BACKDOOR_WEAK_KEYS (CRITICAL)**
   - **Risk:** Complete encryption compromise
   - **Fix:** Remove environment variable check, enforce minimum 2048-bit keys
   - **Validation:** Add key size verification on import

```
def validate_key_size(pub_pem: bytes, min_bits: int = 2048) -> bool:
    pub = load_public_pem(pub_pem)
    if pub.key_size < min_bits:
        raise ValueError(f"Key size {pub.key_size} below minimum
{min_bits}")
    return True
```

2. **BACKDOOR_TRUST_GOSSIP (CRITICAL)**
   - **Risk:** Network impersonation and MITM attacks
   - **Fix:** Remove conditional bypass, always verify signatures
   - **Mitigation:** Implement strict signature verification at protocol level
3. **Hardcoded Credentials (HIGH)**
   - **Risk:** Predictable authentication
   - **Fix:** Remove default passwords, require user-provided credentials
   - **Best Practice:** Use environment variables or secure credential storage

3.3 Medium Priority Issues

1. **SQL Injection in Debug Code**
    - **Fix:** Use parameterized queries even in debug functions

```
cur = conn.execute("SELECT * FROM users WHERE table=?", (table,))
```

2. **Weak Password Hashing**
    - **Fix:** Implement PBKDF2 or Argon2

```
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2
# Use 100,000+ iterations
```

3. **Exception Handling**
    - **Fix:** Log exceptions instead of silently suppressing

```
except Exception as e:
    logging.error(f"Error: {e}")
```

3.4 Low Priority Issues

1. **Code Quality**
    - Break large functions into smaller units
    - Add comprehensive docstrings
    - Fix style violations (trailing whitespace, line length)
2. **Import Organization**
    - Move all imports to module top
    - Group standard, third-party, and local imports
3. **Type Hints**
    - Add type hints for better IDE support
    - Consider using mypy for type checking

---

4. Conclusion

This SOCP implementation demonstrates a **solid understanding of cryptographic protocols** and **secure messaging architecture**. The code quality is generally good, with functional encryption, signing, and federation capabilities.

However, the presence of **two critical backdoors** makes this version **completely unsuitable for production use**. These backdoors, while intentionally planted for educational purposes, demonstrate real-

world attack vectors:

1. **Weak key generation** enables decryption of all messages
2. **Unsigned message acceptance** allows network impersonation

For Educational Use: Excellent example of security vulnerabilities
For Production Use: Requires complete security hardening

Next Steps:

1. Remove all backdoor code immediately
2. Implement recommended security fixes
3. Conduct thorough security audit
4. Add comprehensive testing suite
5. Follow pre-deployment checklist before any production use
