Assignment 2: Secure Chat System Test Report

Course: CS-3002 Information Security

Semester: Fall 2025

1. Introduction

This document provides the methodology and evidence for a series of tests conducted on the Secure Chat System. The purpose of these tests is to empirically verify that the implemented protocol correctly achieves its stated security goals (CIANR) and is resilient against the threat model defined in the assignment.

The following tests were performed:

1. **Wireshark Capture (Confidentiality Test)**
2. **Invalid Certificate Test (Authenticity Test)**
3. **Message Tampering Test (Integrity Test)**
4. **Message Replay Test (Freshness Test)**
5. **Non-Repudiation Test (Offline Verification)**

2. Test 1: Wireshark Capture (Confidentiality)

- **Test Objective:** To verify that all sensitive communication (credentials and chat messages) is encrypted and unreadable to a passive eavesdropper on the network.
- **Methodology:**
    - Started Wireshark and began capturing packets on the loopback interface (`lo` or `Loopback Adapter`) with a display filter for the server's port (e.g., `tcp.port == 12345`).
    - Ran the server and client applications.
    - On the client, performed a **user registration** and then a **user login**.
    - Exchanged several chat messages between the client and server.
    - Stopped the Wireshark capture.
- **Expected Result:**
    - The initial `hello` and `server_hello` packets are in plaintext, and their JSON structure is readable. The public X.509 certificates are visible. This is expected.
    - All subsequent packets containing sensitive data—specifically the encrypted registration payload, the encrypted login payload, and all `MsgModel` chat packets—should appear as "Application Data" over TCP. The JSON payload itself should be unreadable ciphertext.
- Evidence:
  The screenshot below shows the Wireshark capture. The hello packet (No. 4) is visible, but the subsequent chat message (No. 12) shows a TCP segment where the data

payload is entirely encrypted ciphertext, confirming confidentiality.

3. Test 2: Invalid Certificate (Authenticity)

- **Test Objective:** To verify that the server will reject any client that presents a certificate *not* signed by the trusted Root CA.
- **Methodology:**
    1. Created a new, separate "Fake Root CA" by running:
       python scripts/gen_ca.py --name "Fake Attacker CA"
    2. Used this "Fake" CA to issue a new "fake" client certificate:
       python scripts/gen_cert.py --cn client.local --out certs/fake_client
    3. Temporarily modified app/client.py to load the fake identity:
       self.client_cert, self.client_key = pki.load_identity("certs/fake_client")
    4. Ran the server and then ran the modified client.
- **Expected Result:** The client attempts to connect. The server receives the `hello` message, attempts to verify the certificate, and fails because it was signed by "Fake Attacker CA," not the "FAST-NU Root CA" that the server trusts. The server console should print a `ValueError: BAD_CERT: Signature is invalid` and immediately close the connection. The client application should fail to connect.
- Evidence:
  The screenshot below shows the server console output upon the fake client's connection attempt.

4. Test 3: Message Tampering (Integrity)

- **Test Objective:** To verify that if an active attacker intercepts and modifies a message, the receiver will detect the tampering and reject the message.
- **Methodology:** This test requires modifying the code to simulate an attack, as a real-time MitM is complex.
    1. Modified the `app/client.py` file in the `run_chat_session` function.
    2. After the MsgModel was created and signed, but before it was sent, I manually tampered with the ciphertext:
       ct_b64 = utils.b64e(ct_bytes)
       hash_to_sign = ...
       sig_b64 = ...
       ct_b64 = ct_b64[1:] + "A" # Manually tamper with the ciphertext
       send_message(self.sock, protocol.MsgModel(..., ct=ct_b64, sig=sig_b64))
    3. Ran the server and client and attempted to send this tampered message.
- **Expected Result:** The server receives the `MsgModel`. It re-computes the hash `h = SHA256(seqno || ts || ct)` using the *tampered* ciphertext. This new hash `h'` will not match the original hash `h` that was signed. Therefore, the call to `sign.verify_signature` will return `False`, raising a `ValueError: SIGNATURE failure. Message tampered.` The server terminates the connection.

- Evidence:
  The screenshot below shows the server console output after receiving the tampered message from the client.

5. Test 4: Message Replay (Freshness)

- **Test Objective:** To verify that if an attacker captures and resends an old, valid message, the receiver will detect it as a replay and reject it.
- **Methodology:** This test was also simulated by modifying the client code.
  1. Modified `app/client.py`'s `run_chat_session` function.
  2. A valid message `msg_to_send` was created and sent normally.
  3. The exact same message was sent a second time, immediately afterward, without incrementing the client_seq number.
     send_message(self.sock, msg_to_send) # Send first time
     # ... server replies ...
     send_message(self.sock, msg_to_send) # Send exact same message again
- **Expected Result:** The server receives the first message, and its `client_seq` counter increments to 1. The server then receives the second message. It checks the message's `seqno` (which is still 1) against its internal counter (which is also 1). Since `1 <= 1` is true, the `if client_msg.seqno <= client_seq:` check passes, and the server raises a `ValueError: REPLAY attack detected.` The connection is terminated.
- Evidence:
  The screenshot below shows the server console output after receiving the replayed message.

6. Test 5: Non-Repudiation (Offline Verification)

- **Test Objective:** To verify that the generated transcript file and its final hash can be used as cryptographic proof of the conversation.
- **Methodology:**
  1. Ran a full chat session (register, login, send 3-4 messages) and shut it down cleanly.
  2. Located the generated transcript file in the `transcripts/` directory (e.g., `client_vs_server.local_...log`).
  3. Observed the final `TranscriptHash` printed in the client and server consoles (e.g., `[Transcript] Finalized. Hash: a1b2c3d4...`).
  4. Ran an independent SHA-256 hash check on the transcript file from a system shell.
- **Expected Result:** The hash computed by the external tool (e.g., `sha256sum` on Linux or `Get-FileHash` on PowerShell) must exactly match the `TranscriptHash` printed in the console by the applications.
- Evidence:
  The screenshot below shows two terminals.

1. The top terminal shows the client console at the end of a session, printing the final `TranscriptHash`.
2. The bottom terminal shows a `Get-FileHash` (or `sha256sum`) command being run on the corresponding `.log` file, and the output hash is identical.

7. Conclusion

All tests passed successfully, confirming that the implemented protocol and code correctly meet the CIANR security requirements. The system was proven to be secure against passive eavesdropping, active tampering, replay attacks, and identity spoofing (with an invalid certificate).