

ASSIGNMENT REPORT

CS-3002 Information Security

Sawab Akbar 22i-1158
Section F



FAST NUCES, Islamabad
Department of Computer Science

1. Introduction

This report details the design and implementation of SecureChat, an application-layer secure messaging system. The primary goal of this project was to build a chat application from the ground up, implementing cryptographic primitives explicitly to achieve **Confidentiality, Integrity, Authenticity, and Non-Repudiation (CIANR)**, without relying on standard transport-layer security like TLS/SSL.

The system is built in Python and uses a custom-defined protocol to handle certificate exchange, key agreement, user authentication, and encrypted, signed messaging.

2. Implementation Details

The project was implemented in distinct phases, each building a new layer of security.

Phase 1: Public Key Infrastructure (PKI)

The foundation of the system's trust model is a custom PKI.

- `scripts/gen_ca.py`: This script generates the root "SecureChat Root CA," creating `ca.key` and `ca.crt`.
- `scripts/gen_cert.py`: This script uses the CA's key to issue signed certificates for the server (`server.crt`, CN=localhost) and the client (`client.crt`, CN=client).
- `app/crypto/pki.py`: This module implements the critical `validate_certificate` function, which is called during the initial handshake. This function performs three essential checks as per **Req 2.1.v**:
 1. **Signature Validity**: Checks that the peer's certificate was signed by our known `ca.crt`.
 2. **Expiry**: Checks that the current time is within the certificate's `not_valid_before` and `not_valid_after` timestamps.
 3. **Common Name (CN)**: Checks that the certificate's CN matches the expected identity (e.g., the client *must* see "localhost" from the server).

Phase 2: Control Plane (Authentication)

This phase establishes a mutually authenticated channel and handles secure user login.

1. **Mutual Authentication (Req 2.1)**: The client and server first connect and perform a certificate exchange. The server sends `server.crt`, and the client sends `client.crt`. Both parties use `pki.validate_certificate` to verify the other. If verification fails, the connection is immediately dropped.
2. **Temporary Key Exchange (Req 2.2)**: After mutual auth, the client and server perform a temporary Diffie-Hellman (DH) exchange (`app/crypto/dh.py`) to establish a temporary AES-128 key. This key is derived using the `K = Trunc16(SHA256(Ks))` function.

3. **Secure Credentials (Req 2.2):** The temporary AES key is used to encrypt the user's registration or login details. The client sends a `SecureRegister` or `SecureLogin` JSON message, which is encrypted using AES-128 (ECB mode, as specified) via `app/crypto/aes.py`.
4. **Database Storage (Req 2.2.5):** The server's `app/storage/db.py` module handles user registration. It generates a 16-byte random `salt` and stores the user with a `pwd_hash = SHA256(salt || password)`. When a user logs in, `db.py` fetches the salt, re-computes the hash, and uses `hmac.compare_digest` to prevent timing attacks.

Phase 3: Session Key Establishment

After a user is successfully logged in, the temporary AES key is discarded.

- **Final Key Exchange (Req 2.3):** The client and server immediately perform a **second, separate Diffie-Hellman exchange**.
- **Session Key:** The shared secret from this exchange is used to derive the *final* 16-byte AES session key. This key is used for the remainder of the session to encrypt all chat messages.

Phase 4: Data Plane (Encrypted Chat)

This phase implements the live, secure chat.

- **Threading (app/chat.py):** The `ChatSession` class uses two threads: one for listening to network messages (`_network_listen_thread`) and one for listening to user keyboard input (`_user_input_thread`).
- **Message Format (Req 2.4.i):** Each message is a JSON object with four fields: `seqno`, `ts` (timestamp), `ct` (ciphertext), and `sig` (signature).
- **Confidentiality (Req 2.4.i):** The plaintext message is encrypted using the *final session key* with AES-128 (`aes.encrypt`). The resulting ciphertext is Base64-encoded and sent as the `ct` field.
- **Authenticity & Integrity (Req 2.4.ii):** Before sending, a `data_hash = SHA256(seqno || ts || ct)` is computed. This hash is then signed using the sender's private key (`client.key` or `server.key`) via `app/crypto/sign.py`. This RSA-SHA256 signature is sent as the `sig` field.

Phase 5: Non-Repudiation

This phase ensures a verifiable log of the conversation.

- **Transcript Logging (Req 2.5.i):** The `app/storage/transcript.py` module writes every single sent and received message to an append-only log file (e.g., `logs/transcript_client_i...log`). Each line is formatted as `seqno | ts | ct | sig | peer-cert-fp`.
- **Session Receipt (Req 2.5.ii):** When a user types `quit`, the `ChatSession` computes a `TranscriptHash = SHA256(all_log_lines)`. This hash is then signed by the user's private key to create a `SessionReceipt`. This receipt is sent to the peer, and the connection is closed.

3. Testing & Evidence (Requirement 3)

The following tests were conducted to verify the security of the application.

1. Wireshark Test (Req 3.i - Confidentiality)

Wireshark was used to capture traffic on the loopback interface (`lo0`) on port 12345. After the initial certificate exchange, all subsequent data packets (DH exchanges, login, and chat) were unreadable. The packet data pane clearly shows the encrypted JSON payloads, proving confidentiality.

[--- INSERT SCREENSHOT HERE ---] (Your screenshot should show Wireshark capturing on `lo0`, filtered by `tcp.port == 12345`, with a packet selected, and the "Data" pane showing unreadable text.)

2. Invalid Certificate Test (Req 3.ii - Authenticity)

If the server presents a certificate not signed by our `ca.crt`, the client's `pki.validate_certificate` function fails. This throws an exception, and the client terminates the connection, preventing a Man-in-the-Middle (MitM) attack.

3. Tamper Test (Req 3.iii - Integrity)

Message integrity is guaranteed by the RSA signature. In `app/chat.py`, the `_verify_message` function re-computes the message hash and calls `sign.verify()`. If an attacker tampered with the ciphertext (`ct`) in transit, the hash would not match, `sign.verify()` would return `False`, and the message would be rejected as invalid.

4. Replay Test (Req 3.iv - Integrity)

Replay attacks are prevented by the `recv_seqno` state variable in the `ChatSession` class. The `_verify_message` function first checks if `msg.seqno <= self.recv_seqno`. If an attacker re-sends an old message, its sequence number will be lower than or equal to the current expected number, and the message will be rejected.

5. Offline Verification (Req 2.5.ii - Non-Repudiation)

The `verify_transcript.py` script provides offline verifiability. This script takes a log file, the peer's certificate, and the peer's copy-pasted `SessionReceipt`. It re-computes the `TranscriptHash` from the log file and verifies the peer's signature against it.

- **Test 1: Valid Transcript:** Running the verifier on an unmodified log file and the correct receipt shows a "SUCCESS" message.
- **Test 2: Tampered Transcript:** If even one character in the log file is altered, the `TranscriptHash` changes, and the verification **fails**.

This proves that a user cannot deny having sent the messages recorded in a valid transcript, achieving non-repudiation.

[--- INSERT SCREENSHOT HERE ---] (Your screenshot should show the output of `verify_transcript.py` running and showing "--- SUCCESS ---".)

4. Conclusion

This project successfully implements a secure chat application from basic cryptographic primitives. By layering mutual PKI authentication, Diffie-Hellman key exchanges, and AES-encrypted, RSA-signed messages, the application achieves all core CIANR security goals. The final implementation of non-repudiation through signed transcripts provides a verifiable and robust system.