Assignment 2: Secure Chat System Report

Course: CS-3002 Information Security

Semester: Fall 2025

1. Introduction

This report details the design and implementation of a secure client-server chat system. The primary objective of this project was to build a practical, application-layer cryptographic protocol from the ground up, without relying on existing standards like TLS/SSL.

The system was designed to meet four core security goals, collectively known as CIANR:

- **Confidentiality:** Ensuring that messages cannot be read by eavesdroppers.
- **Integrity:** Ensuring that messages cannot be tampered with or altered in transit.
- **Authenticity:** Ensuring that both the client and server can verify each other's identity.
- **Non-Repudiation:** Ensuring that neither party can deny having sent or received a message.

This was achieved by correctly integrating several cryptographic primitives: AES-128, RSA, X.509 Certificates, Diffie-Hellman (DH) key exchange, and SHA-256 hashing.

2. System Architecture

The application follows a standard client-server model, communicating over a plain TCP socket. The security is built entirely at the application layer. The system's architecture is composed of three main parts:

1. **Public Key Infrastructure (PKI):** A custom Root Certificate Authority (CA) was created using Python's `cryptography` library. This Root CA is the "root of trust" for the entire system and is used to issue and sign X.509 certificates for both the server (`server.local`) and the client (`client.local`).
2. **Server:** A multi-threaded Python server that listens for connections, manages the protocol flow, handles user authentication against a MySQL database, and facilitates the secure chat session.
3. **Client:** A console-based Python application that initiates the connection, handles user registration and login, and securely exchanges messages with the server.

3. Protocol Design and CIANR Implementation

The secure protocol is divided into distinct phases, each designed to systematically build a secure channel and achieve the CIANR goals.

3.1. Authenticity

Goal: To prove the client and server are who they claim to be.

Implementation: This is the first step in the protocol (PKI_CONNECT and CERT_VERIFY) and is achieved using our custom PKI.

1. The client connects and sends a hello message containing its X.509 certificate.
2. The server receives the client's certificate and verifies it using the app.crypto.pki.verify_certificate function. This function checks three things:
    - **Signature:** Is the certificate signed by the trusted Root CA (ca_cert.pem)?
    - **Validity:** Is the certificate's date range valid?
    - **Common Name (CN):** Does the certificate's CN match the expected value (client.local)?
3. If the client is verified, the server responds with a server_hello message containing its own certificate.
4. The client performs the exact same verification on the server's certificate, checking it against the Root CA and the expected CN (server.local).

Only after this mutual authentication is successful does the protocol proceed. This step ensures that an attacker cannot impersonate the server (MitM) or a valid client.

3.2. Confidentiality

Goal: To protect the content of all communications from being read by an attacker.

Implementation: This is achieved by encrypting all sensitive data using AES-128. Two separate AES keys are established at different points in the protocol using Diffie-Hellman key exchange.

1. **Authentication Key (k_auth):** After the PKI handshake, a *temporary* DH exchange is performed. Both parties compute a shared secret, which is then hashed to create k_auth. This key is used *only* to encrypt the subsequent registration or login message. This ensures that user credentials (email, password) are never sent in plaintext.
2. **Session Key (k_chat):** After a user is successfully authenticated, a *second* DH exchange is performed. This generates a new, separate shared secret that is hashed to create k_chat. This key is the main session key and is used to encrypt and decrypt all chat messages (MsgModel) for the duration of the session.

Because the Diffie-Hellman exchange only communicates public values, an eavesdropper cannot derive either k_auth or k_chat, guaranteeing confidentiality.

3.3. Integrity

Goal: To ensure that messages are not altered in transit.

Implementation: This is achieved by using a digital signature on a hash of the message content.

1. When a user (client or server) sends a message, it constructs a message payload containing a sequence number (seqno), a timestamp (ts), and the AES-encrypted ciphertext (ct).
2. It then computes a hash of this metadata: h = SHA256(seqno || ts || ct).

3. This hash h is then signed using the sender's private RSA key (e.g., client_private_key.pem).
4. The receiver gets the message and performs the reverse operation:
    - It re-computes the hash h using the seqno, ts, and ct from the message it received.
    - It verifies the received signature sig against the re-computed hash h, using the sender's public key (which it has from their certificate).

If an attacker modifies *any* part of the message (e.g., changes the ciphertext or the sequence number), the hash will not match, and the signature verification will fail. The application detects this as a SIGNATURE failure and terminates the connection.

3.4. Non-Repudiation

Goal: To create cryptographic proof that a specific user sent a specific message.

Implementation: This goal is achieved by combining the integrity mechanism with a persistent, signed transcript.

1. **Per-Message Proof:** Since every message is digitally signed (as described in 3.3), the signature sig on each message acts as a non-repudiable proof for that single message.
2. **Session Proof:** To prove the entire conversation, the app.storage.transcript.Transcript class is used.
    - Both client and server log every single message (with its seqno, ts, ct, and sig) to a local file.
    - This class also maintains a running SHA-256 hash of the entire transcript.
    - At the end of the conversation, the client and server exchange a final ReceiptModel message. This receipt contains the final transcript_sha256 hash, which is itself signed by the sender.

This signed receipt acts as the final, verifiable proof of the entire communication. An auditor can use this receipt, the transcript log, and the participants' public certificates to verify the entire conversation.

4. Additional Security Features

Beyond the CIANR goals, the protocol implements other essential security controls:

- **Replay Protection:** The seqno in each MsgModel is strictly increasing. The client and server both track the last-seen sequence number from the other party. If a message arrives with a seqno that is not greater than the last, it is discarded as a replay attack, and the connection is dropped.
- **Secure Credential Storage:** User passwords are not stored in the database. When a user registers, the client generates a 16-byte salt, computes pwd_hash = SHA256(salt + password), and sends the salt and pwd_hash to the server (this communication is

protected by k_auth). The server stores these values. For login, the server uses the stored salt and the provided password to re-compute the hash and check for a match.

5. Conclusion

This project successfully demonstrates how fundamental cryptographic primitives can be composed to build a secure, end-to-end communication protocol. By layering PKI for **Authenticity**, DH-derived AES keys for **Confidentiality**, and RSA-signed hashes for **Integrity** and **Non-Repudiation**, the system achieves a robust security posture against the defined threat model of passive and active attackers.