SecureChat: A PKI-Enabled Encrypted Messaging System

CS-3002 Information Security – Assignment #2

Fall 2025

Name: Saeed Ud Din Ahmad
Roll Number: 22i-0938
Section: CS-H

---

1. Introduction

This project implements **SecureChat**, a console-based secure communication system demonstrating how classical cryptographic primitives combine to achieve:

- **Confidentiality**
- **Integrity**
- **Authenticity**
- **Non-Repudiation (CIANR)**

The system is built entirely at the **application layer**, without relying on TLS/SSL, ensuring students understand how cryptographic mechanisms work internally in secure protocols.

SecureChat uses:

- **AES-128 (ECB + PKCS#7 padding)** for message confidentiality
- **Diffie–Hellman (DH)** for shared secret derivation
- **RSA-2048 + SHA-256** signatures for authenticity & message integrity
- **X.509 PKI (self-built CA)** for identity verification
- **MySQL** for securely storing salted password hashes

- **Signed transcript receipts** for non-repudiation

This report documents the design decisions, implementation process, and security guarantees achieved by the system.

---

2. System Architecture

SecureChat is a client–server application built using plain TCP sockets (no TLS). All cryptographic protections are implemented manually.

System Components

Component	Function
`scripts/gen_ca.py`	Generates Root CA and self-signed certificate
`scripts/gen_cert.py`	Issues server/client certificates signed by CA
`app/crypto/aes.py`	AES-128 ECB encryption/decryption
`app/crypto/dh.py`	Classical DH key exchange + SHA-256 → AES key
`app/crypto/pki.py`	Certificate validation (CA signature, expiry, CN)
`app/crypto/sign.py`	RSA SHA-256 signing & verification
`app/storage/db.py`	MySQL storage with salted SHA-256 password hashing
`app/storage/transcript.py`	Session transcript + transcript hash
`app/server.py`	Server control plane, authentication, chat logic
`app/client.py`	Client control plane, authentication, chat logic

---

3. PKI Setup & Certificate Validation

A complete Public Key Infrastructure was implemented:

3.1 Root Certificate Authority (CA)

`gen_ca.py` generates:

- `root_ca.key` (private key)
- `root_ca.crt` (self-signed certificate)

The CA certificate is used to issue and validate all participants.

3.2 Server/Client Certificates

`gen_cert.py` produces:

- `server.key`, `server.crt`
- `client.key`, `client.crt`

Each certificate includes:

- CN (`server.local` / `client.local`)
- 1-year validity period
- RSA-2048 signing by Root CA

3.3 Certificate Validation Rules

During handshake:

- Server validates **client cert** CN = `client.local`
- Client validates **server cert** CN = `server.local`
- Expired or self-signed certs → `BAD_CERT`
- Signature must match CA public key
- Validity period must not be expired

This ensures **strong peer authentication**.

---

4. Registration & Login Security

User credentials are stored securely in MySQL.

4.1 Password Handling

Server does NOT store plaintext passwords.

For each user:

- A **16-byte random salt** is generated
- Password hash = `SHA256(salt || password)`
- Stored fields:

Field	Type
email	VARCHAR
username	VARCHAR UNIQUE
salt	VARBINARY(16)
pwd_hash	CHAR(64)

4.2 Credentials in Transit

Credentials are **never sent in plaintext**.

During registration/login:

1. A temporary DH key is derived (K_tmp)
2. Credentials are JSON-encoded
3. Encrypted with **AES-128(ECB)** using K_tmp

4. Base64 encoded & sent

This protects confidentiality before authentication is established.

---

5. Authenticated Key Exchange (Diffie–Hellman)

After authentication, a **fresh DH exchange** creates the **session key**:

- Client sends `(g, p, A = g^a mod p)`
- Server responds with `(B = g^b mod p)`
- Shared secret:

```
Ks = B^a mod p = A^b mod p
```

AES-128 Key Derivation

```
K = Trunc16( SHA256( big_endian(Ks) ) )
```

Security Properties

- **Perfect Forward Secrecy (PFS)** (new DH key per session)
- Shared key never sent over network
- Eavesdroppers cannot derive K

---

6. Encrypted Chat: Confidentiality, Integrity & Authenticity

Every chat message provides 4 protections:

Property	How Achieved
Confidentiality	AES-128(ECB) encryption
Integrity	SHA-256 digest over metadata + ciphertext
Authenticity	RSA-2048 signature of hash
Freshness	Sequence number + timestamps

6.1 Message Structure

```
{
  "type": "msg",
  "seqno": <int>,
  "ts": <unix ms>,
  "ct": <base64 ciphertext>,
  "sig": <base64 RSA signature>
}
```

6.2 Sender Side Workflow

1. `seqno` increases strictly
2. `ts` = now_ms()
3. Encrypt plaintext → AES-128 → ciphertext `ct`
4. Compute digest:

```
h = SHA256(seqno || ts || ct)
```

5. Sign digest:

```
sig = RSA_SIGN(h)
```

6.3 Receiver Side Workflow

- Reject message if `seqno <= last_seqno` → `REPLAY`
- Recompute digest & verify signature → else `SIG_FAIL`
- Decrypt ciphertext → plaintext

---

7. Non-Repudiation: Transcript & Session Receipt

SecureChat logs **every message** in an append-only transcript:

```
seqno | ts | ct_b64 | sig_b64 | peer_cert_fingerprint
```

After session termination:

7.1 Transcript Hash

```
TranscriptHash = SHA256(concatenate(all transcript lines))
```

7.2 Session Receipt (signed)

```
{
  "type": "receipt",
  "peer": "client" | "server",
  "first_seq": ...,
  "last_seq": ...,
  "transcript_sha256": "<hex>",
  "sig": "<base64 RSA signature>"
```

```
}
```

This provides **cryptographic proof** of communication.

Any modification to the transcript invalidates verification, achieving **non-repudiation**.

---

8. Testing & Security Validation

8.1 Wireshark Evidence

- All TCP payloads appear as unreadable ciphertext (AES encrypted)
- No plaintext credentials, messages, or salts visible

8.2 Invalid Certificate Test

- Self-signed client certificate → rejected (`BAD_CERT`)
- Wrong CN → rejected
- Expired certificate → rejected

8.3 Tampering Test

Manually flip a bit in `ct`:

→ Signature verification fails (`SIG_FAIL`)
→ Message rejected

8.4 Replay Test

Reinject previously captured message:

→ Rejected due to outdated `seqno` (`REPLAY`)

8.5 Non-Repudiation Test

- Recompute transcript hash offline
- Verify client/server signature using their certificates
- Modification of transcript breaks verification

---

9. Conclusion

SecureChat successfully demonstrates how classical cryptographic primitives combine to form a secure communication protocol providing **Confidentiality, Integrity, Authenticity, and Non-Repudiation**.

All requirements were implemented at the **application layer**, without relying on TLS.
The system is modular, transparent, and educational, providing full visibility into certificate handling, key exchange, authenticated encryption, and signature-based accountability.