Final Project Report

Secure Terminal Chat Room (Dockerized Multi Client Encrypted Chat)

Students: Adiyan Hossain and Andrew Tan
CECS 478 Computer Security
Date: December 8th, 2025

Github repo link : https://github.com/Adiyan542/CECS478-FinalProject

Problem Statement

Many simple and old chat programs send messages in plain text, which means anyone on the same network can capture and read them. This becomes a serious security risk because private conversations or sensitive information can leak without users knowing. Even simple terminal chat tools are unsafe if they do not use encryption. For this project, we will build a secure, multi-client terminal chat room running inside Docker containers. It will include one server and multiple clients that communicate in real time. All messages will be encrypted so that anyone trying to sniff the Docker network will only see unreadable data. We will use symmetric authenticated encryption to protect messages and prevent tampering. We will also focus on writing safe C code to avoid common bugs in network programs. Even though the chat room is simple, the main purpose is to learn how to combine networking, encryption, and Docker to create a secure and reliable communication system.

Threat Model

Asset

- The privacy of chat messages.
- The integrity of each message (nobody can modify it).
- The symmetric key used for encryption.
- The reliability of the server.

Potential Attackers

- Sniffing the Docker network and trying to read messages.
- Replaying old messages to confuse users.
- Tampering with messages to break the system.
- A malicious client sending bad data to crash the server.

Attack Surfaces

- The network path between clients and the server.
- The message parsing code in the server and clients.
- How the encryption key is stored in the container.
- The broadcast system that sends messages to all clients.

We assume all containers run in a private docker network, the key is shared ahead of time, the host machine is trusted, and attackers cannot break modern encryption. To defend the system, we will use authenticated encryption, nonces to stop replays, safe C coding practices, docker isolation, and strict message validation. These steps make sure that the chat system is resistant to basic eavesdropping and tampering.

Methods

To build the secure chat system, we combined Docker networking, socket programming in C, and modern cryptographic libraries. The system includes one server container and multiple client containers running inside a private Docker subnet. Each container is built using Ubuntu and compiled with the libsodium cryptography library for encryption and key exchange.

Implementation Steps

- **Docker Environment Setup** = We created a private Docker network (172.16.238.0/24) and defined a server and three clients in docker-compose.yml. Each container is isolated from the host and given a fixed IP address. This made testing easy and prevented outside devices from accessing the chat traffic.
- **Key Exchange Protocol (X25519)** = When a client connects, the server sends its public key and expects the client's public key in return. Both sides compute a shared secret using crypto_kx_* functions from libsodium. This produces a unique symmetric session key used only for that client’s messages.
- **Authenticated Encryption (ChaCha20 Poly1305)** = Every plaintext message is encrypted using crypto_aead_chacha20poly1305_ietf_encrypt() and decrypted using the matching function. We also added an 8 byte nonce to each message to prevent replay attacks.
- **Message Routing** = The server decrypts each incoming message, then re-encrypts it separately for every other connected client. Each client maintains its own send and receive nonce counters.
- **Safe C Coding Practices** = To avoid memory issues, we used bounds checking, flushed long inputs, ignored empty messages, and wiped secret keys from memory using sodium_memzero().
- **Experimentation and Testing** = They were run using both automatic demo scripts (make demo) and manual testing with multiple terminal windows. We also captured

packets using tcpdump to confirm encryption, and used logs to analyze system behavior.

Results

Encrypted Traffic Validation = We collected a packet capture using tcpdump to see what messages look like on the network. When we opened the file in Wireshark, everything appeared as random, unreadable data. This means the encryption is working.

Wireshark also showed that

- No plaintext messages ever appear on the network.
- Each packet includes a nonce and an authentication tag.
- Longer messages create larger encrypted packets, just like we expected.

System Behavior in Normal Use = When we tested the chat room manually, all clients connected successfully and exchanged keys with the server. Messages showed up right away for everyone, and the broadcast system worked without any problems. The server handled multiple clients at the same time with no errors.

Negative Testing

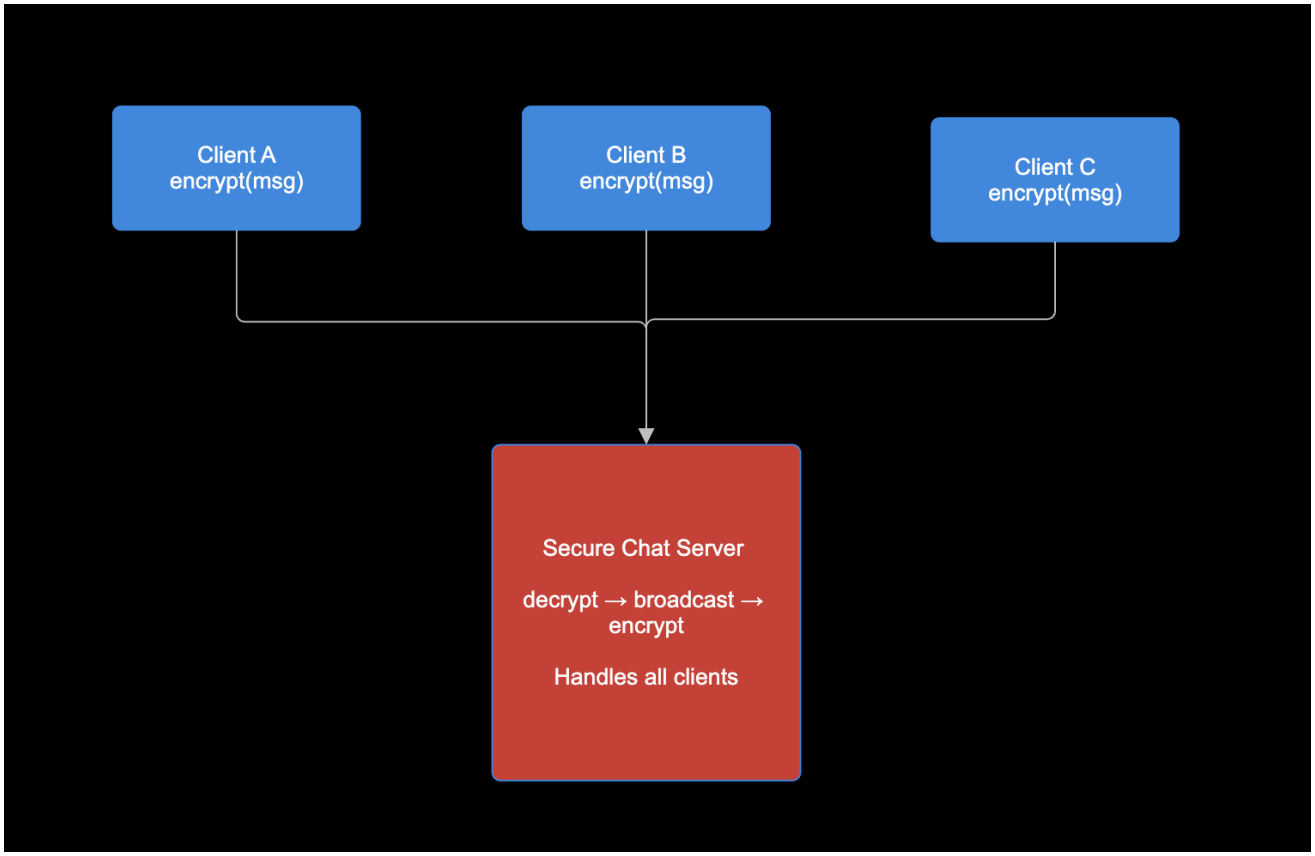
- Empty message: The client sent nothing and the system stayed stable and did not crash.
- Very long message (over 2000 characters): The message was still encrypted and broadcast correctly, and the system continued running normally.

Message Size Statistics = We made a CSV file that shows the size of different messages and how big they become after encryption. This helps us understand the overhead added by encryption and lets us compare plaintext and ciphertext sizes.

Logs and Server Output

- When clients join and leave
- When keys are exchanged
- When the server shuts down
- There were no decryption failures or crashes reported in any of the logs. This means the system behaved correctly throughout all experiments.

Architecture Diagram



Limitations and Future Work

Limitations

- The system has no message persistence, so chat history is lost when the server shuts down.
- It is not designed for real world deployment since it lacks TLS, certificate checks, user accounts, and other security protections used in production systems.
- The system is not very scalable since one server handles every message and key exchange, which may slow down performance with many clients.

Future Work

- Add user authentication (passwords or public keys) so unknown clients cannot join the chatroom.
- Support multiple chat rooms, private messages, or user groups to make the system more flexible.
- Measure performance metrics like message delay, CPU load, and encryption cost when more clients join.
- Improve logging and monitoring by exporting structured logs or adding dashboards.
- Build a web based or GUI client so users are not limited to terminal windows.

Contribution Log

Andrew Tan

- Lead designer and implemented the networking logic for client server communication
- Built the server's message routing and broadcast system
- Implemented socket handling, threading, and connection management
- Implemented the encryption layer using X25519 + ChaCha20 Poly1305 (libsodium)
- Wrote automated demo scripts
- Helped set up the docker environment and set up

Adiyan Hossain

- Helped test multi client manual chat sessions
- Assisted with negative test cases
- Contributed to debugging runtime issues and message flow
- Created and organized artifacts/release/ folder (pcap, logs, metrics)
- Wrote documentation (Architecture, Security, Runbook, Results, and more)
- Developed the presentation materials and testing scripts
- Led evaluation with tcpdump and Wireshark

Code Map

src/server/server.c = Andrew (Lead Designer and Implementation), Adiyan (Support)

- Connection handling & client threads
- Message broadcasting
- Join/leave events
- Integration with encryption and nonce checks

src/client/client.c = Andrew (Lead Designer and Implementation), Adiyan (Support)

- Encryption & decryption logic
- Key exchange (X25519)
- Nonce handling
- Message formatting & send/receive threads

Docker Setup (Both)

- Docker network architecture
- Container IP assignment
- Build environment (libsodium, gcc)

- Multi client orchestration

Demo Scripts = Andrew (Lead Designer and Implementation), Adiyan (Support)

- Automated chat simulation
- Negative testing scenarios

Documentation = Adiyan (Lead in terms of implementation), Andrew (Support)

- Architecture, Security, Runbook, Results, and project report writing. In addition to the presentation and demo videos.

Manual Testing and Debugging (Both)

- Multi client terminal testing
- Fixing runtime issues
- Verifying encryption and broadcast behavior