Building a Secure Chat Application in Python with TLS Encryption

Learning objectives:

- Understanding how TLS ensures secure communication in client-server networks
- Set-up and manage SSL certificates in Python
- Implement and test a TLS-secured chat app using `socket` and `ssl`
- Analyze encrypted communication in Wireshark
- Identify improvements and limitations of base TLS implementations.

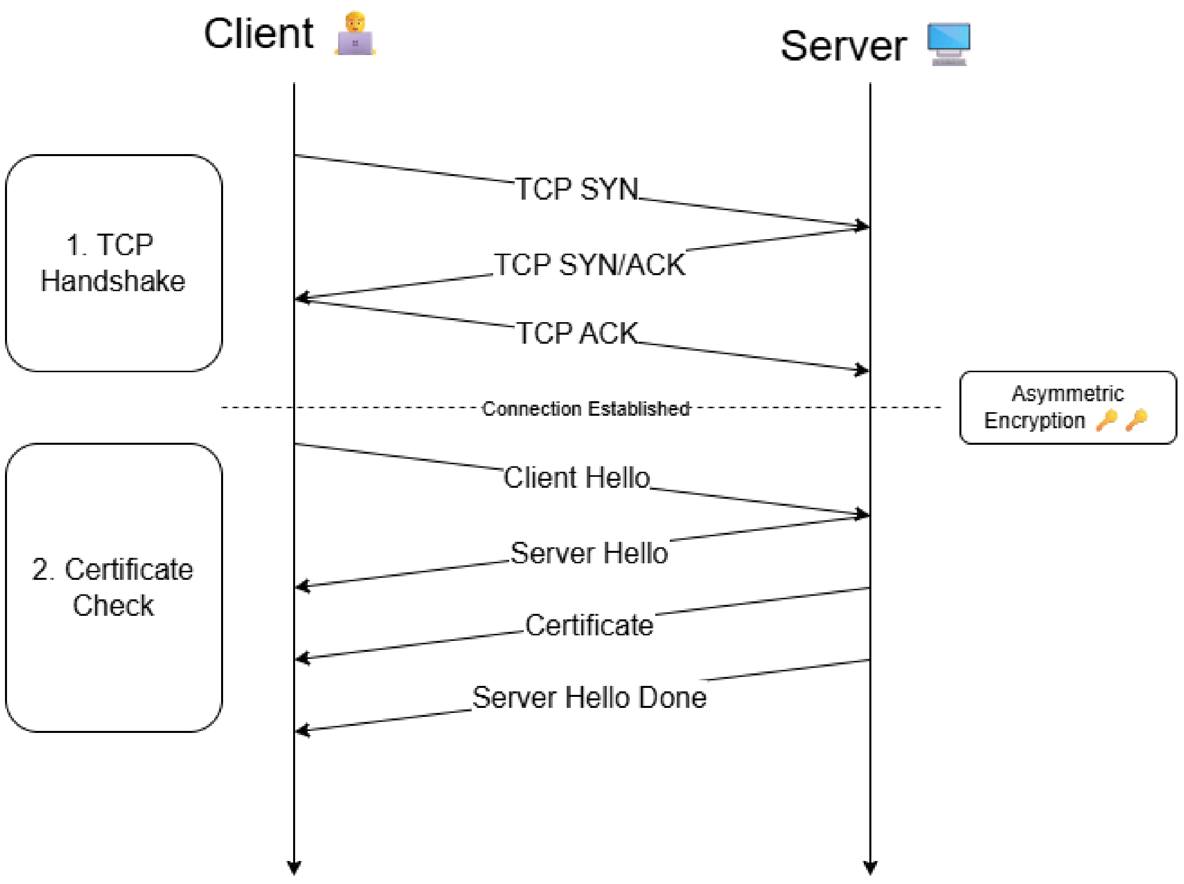
Introduction to Secure Communication

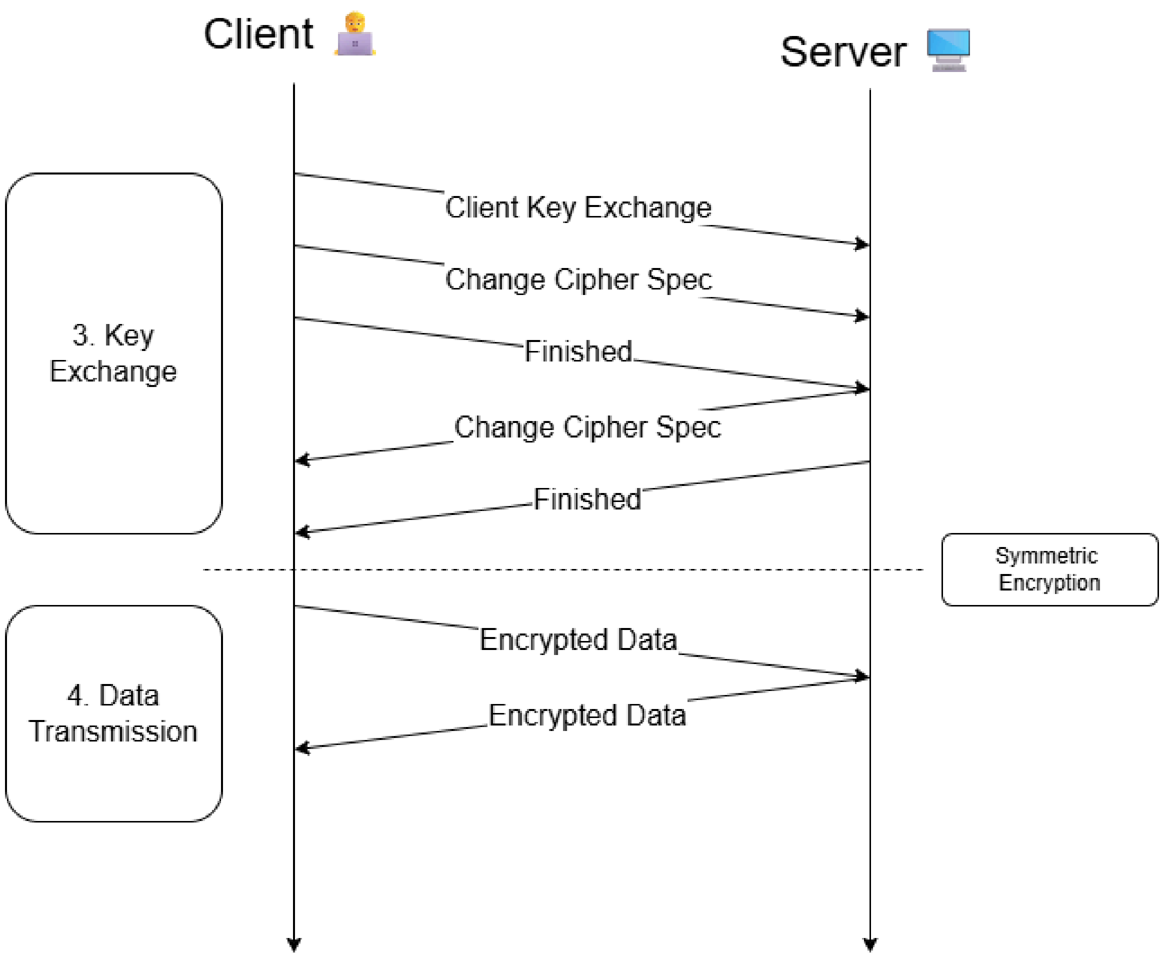
In today's digital world, network security is critical as sensitive data is constantly being exchanged over the internet. Without proper security, networks are vulnerable to malicious attacks. Protecting the network means safeguarding critical data, preventing costly data breaches and maintaining confidentiality, integrity, and availability. TLS is one of the most widely used protocols to achieve this. [1][2][3]

TLS (Transport Layer Protocol) is a cryptographic protocol which is designed to provide encryption over a network. It uses a combination of symmetric and asymmetric TLS handshake whereby the client and server exchange information and keys including the server's digital certificate for authentication, in order to establish a shared secret. This secret is used to encrypt all transmitted data and to protect sensitive information. [1][3]

Before TLS, a protocol called SSL (Secure Socket Layer) was used. However, SSL has too many vulnerabilities and is now deprecated and replaced by TLS with improved security and efficiency. [4]

TCP Handshake Introduction Tutorial: https://youtu.be/f5zFw34LwXE


Client
Server
1. TCP Handshake
TCP SYN
TCP SYN/ACK
TCP ACK
Connection Established
Asymmetric Encryption 🔑 🔑
2. Certificate Check
Client Hello
Server Hello
Certificate
Server Hello Done



Project Overview

This project focuses on implementing a secure chat app that enables secure communication between a client and a server over a network with its primary goal being to apply the core concepts of network security and communication protocols by creating a chat system that leverages TLS in order to protect data in transit.

This project is built using Python with the `socket` module to handle network communication, `threading` for multiple simultaneous clients, and `ssl` module to enable encryption (runs on TLS). Self-signed SSL certificates are also generated to authenticate the server and secure connection.

The architecture of the project is a client-server model with the server listening for incoming TLS connections, and each client connects to the server over some secure socket. Once connected, both parties exchange encrypted messages using shared symmetric key which was negotiated during the TLS handshake. This ensures confidentiality, integrity, and authentication.

Project Introduction Demo: https://youtu.be/cLIYvoGzI-0

Generating a Self-Signed Certificate

In a secure communication over TLS, certificates play a crucial role in establishing trust and encryption. This certificate contains a public key and information about the server, there is also a private key which isn't shared in this interaction. The certificate allows the client to verify the server's identity providing a sort of authentication. This prevents Man In The Middle (MITM) attacks. [5]

In production environments, certificates are typically signed by a trusted CA (Certificate Authority) which verifies the identity of the certificate's subject and then digitally signs the certificate. But for the development of this project in order to learn the internals of an application, we are going to use a self-signed certificate which is issued by the entity that the certificate is meant to verify rather than some third party CA. Although not trusted by default browsers or system, it enables us to use encrypted TLS sessions. [5][6]

In this project, a Python script called `generate_certs.py` is used to generate the necessary certificates and private key to produce 4 files: `server.crt`, `server.key`, `client.crt`, and `client.key`. These files represent the public and private key held by both the server and client used during verification. Here's a step by step explanation of the code.

For the creation of the program, we are going to import modules from Python's `cryptography` package to handle the key creation and certificate building.

```python
from datetime import datetime, timedelta, timezone
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa
```

After that we generate a private key using RSA to use with the self signed certificate.

```
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)
```

Then using the private key we will generate the public key by deriving it from the private key. This will go inside the certificate.

```
public_key = private_key.public_key()
```

After that we define the certificate's subject. For this we set the Common Name (CN) field to `localhost` or `client.localhost` depending upon its role.

```
subject = issuer = x509.Name([
    x509.NameAttribute(NameOID.COMMON_NAME, u"localhost"),
])
# For client:
# x509.NameAttribute(NameOID.COMMON_NAME, u"client.localhost")
```

After that we build the X.509 certificate using the `x509.CertificateBuilder()` to build a certificate that has the following properties:

- Subject and issuer
- Public key
- Random serial number
- Validity period of 1 year
- Subject Alternative Name (SAN) for hostname
- Sign it using SHA256 and the private key

```python
cert = x509.CertificateBuilder().subject_name(subject
).issuer_name(issuer
).public_key(public_key
).serial_number(x509.random_serial_number()
).not_valid_before(datetime.now(timezone.utc)
).not_valid_after(datetime.now(timezone.utc) + timedelta(days=365)
).add_extension(
    x509.SubjectAlternativeName([x509.DNSName(u"localhost")]),
    critical=False,
).sign(private_key, hashes.SHA256())
```

After that, save the private key and certificates in PEM format as that is how it is commonly used by TLS libraries.

```python
# Save private key
with open(f"{cert_name_prefix}.key", "wb") as f:
    f.write(private_key.private_bytes(...))

# Save certificate
with open(f"{cert_name_prefix}.crt", "wb") as f:
    f.write(cert.public_bytes(...))
```

After creating the script, run it for both the server and the client

```python
if __name__ == "__main__":
    generate_cert("server")
    generate_cert("client")
```

After running this code you should have the 4 files: `server.crt`, `server.key`, `client.crt`, and `client.key`. The final code for the implementation is at the end of this tutorial.

Generate Certificates Script Demo: https://youtu.be/pAb3NZET1VY

Implementing the Secure Server

The server for the secure chat app forms the hearts of the entire TLS-protected application. The server listens for client connections, upgrades them to secure TLS connection using SSL, and then handles messaging across multiple users through threading and message broadcasting.

For this, a script called `server.py` is being used. The following packages are being used for the server script:

```python
import socket
import ssl
import threading
import logging
from datetime import datetime
```

The process first begins by setting up a standard Python TCP socket bound to a certain IP address and port. And then to enable TLS, SSL context is being configured for which we load the self-signed certificate and private key.

```python
# Create base socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_socket.bind((host, port))
server_socket.listen(5)

# Create SSL context
ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_context.verify_mode = ssl.CERT_REQUIRED
ssl_context.load_cert_chain(certfile="server.crt", keyfile="server.key")
ssl_context.load_verify_locations("client.crt")
```

Once the server socket is ready, we then listen to any incoming connection requests. Each client connection is accepted and wrapped in a secure TLS socket to ensure that all communication is encrypted.

```
client_socket, address = server_socket.accept()
ssl_socket = self.ssl_context.wrap_socket(client_socket, server_side=True)
```

To all multiple client connections, the server handles each client in a separate thread which allows multiple clients to connect and chat concurrently.

```
client_thread = threading.Thread(target=handle_client, args=(client_socket, address))
client_thread.daemon = True
client_thread.start()
```

Inside the thread, the server continuously listens for messages from the client. When any message is received, the server broadcasts the message securely to all connected clients except the sender.

```
msg = ssl_socket.recv(1024).decode('utf-8')

for client in clients:
  if client != sender:
    client.send(msg.encode('utf-8'))
```

The server also continuously prints logs for new connections, disconnections, and messages, helping with debugging and monitoring.

```
logging.basicConfig(
  level=logging.INFO,
  format="%(asctime)s - %(levelname)s - %(message)s"
)
logger = logging.getLogger(__name__)

logger.info("Message log...")
logger.error("Error...")
```

This multi-threaded, TLS-secured architecture ensures a private and reliable chat experience for all users. The final code for this is at the end of this tutorial.

Secure Chat Server Overview: https://youtu.be/qWO7P5EdGC8

Implementing the Secure Client

In order to complete the secure chat system, we now need to implement the client that connects securely to the server using TLS encryption. The client's role is to initiate a secure connection, allow users to send messages, and then display those messages broadcasted from the server.

We begin first by creating a TCP socket and then wrapping it with SSL. Just for development purposes, we are going to disable certificate validation so that the client can connect without verifying the server certificate (but this creates insecure production as it leaves the client vulnerable to MITM attacks).

```python
# Create base socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Create SSL context
ssl_context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
ssl_context.load_cert_chain(certfile="client.crt", keyfile="client.key")
ssl_context.check_hostname = False # for development only
ssl_context.verify_mode = ssl.CERT_REQUIRED
ssl_context.load_verify_locations("server.crt")

# Wrap socket with SSL
ssl_socket = self.ssl_context.wrap_socket(client_socket, server_hostname=host)
```

After setting up the socket, we then connect to the server.

```python
ssl_socket.connect((host, port))
```

After connecting, we use two threads: one for sending user input, and another for receiving messages from the server. To send the message, we receive it as input, format it, and then send it to the server.

```python
message = input()
formatted_message = f"{username}: {message}"
ssl_socket.send(formatted_message.encode('utf-8'))
```

To receive messages, we start a separate thread that constantly listens for broadcasts from the server.

```
# receive messages
message = self.ssl_socket.recv(1024).decode('utf-8')

# Starting a receiving thread
receive_thread = threading.Thread(target=self.receive_messages)
receive_thread.daemon = True
receive_thread.start()
```

The final code for client can be found at the end of this tutorial.

Secure Chat Client Overview: https://youtu.be/xjBJmIwxrNE

Running the Chat App

To demonstrate the secure chat application, we first start by generating the certificates. We do so by first running the `generate_certs.py` script in the terminal:

```
$ python generate_certs.py
```

After this, we launch the server using the following command:

```
$ python server.py
```

Once the server is initialized with TLS and is ready to accept secure connections. In separate terminals, we then run two client instances:

```
$ python client.py
```

Once each client is connected, the user can then send messages that are securely broadcasted to all connected clients.

With this, both clients should now be able to exchange messages through the server. The server logs shows each message being received and relayed, along with the connection and disconnection events. Behind the scenes, all the communication is being encrypted using SSL/TLS. While the actual message content is visible in the terminal, it's not visible to anyone intercepting the traffic.

Final Demo With Wireshark: https://youtu.be/g0JeM9Pj2U8

Challenges, Limitations & Improvements

Now, while the secure chat application does successfully demonstrate TLS-encrypted communication and certificate-based authentication, there are still several limitations remaining that would need to be addressed for any real-world deployment.

One key issue is the improper certificate verification on the client side. Although TLS is used, the client explicitly disables hostname verification for ease during development. In production though, this is highly insecure, as it opens the door to MITM (Man In The

Middle) attacks. A trusted Certificate Authority (CA) should be used, and hostname verification must be enabled.

Secondly, the application lacks user-level authentication. Currently, any client with a valid certificate can connect. There is no password-based login or identity verification beyond just the certificate itself. This can be improved by using secure login interfaces.

Additionally, messages are not stored or encrypted at rest. While TLS secures messages in transit, they are not logged securely or stored for audit or history purposes. A secure backend with encrypted logging and access controls would be essential.

Limitation	Proposed Solution
Hostname verification disabled	Enable it and use a trusted CA
No user authentication	Implement a secure login
No message encryption at rest	Encrypt stored logs or use secure

Summary & Reflection

In this project, we developed a secure, certificate-based chat app using Python's `socket` and `ssl` libraries. The app features both server and multiple clients communication over TLS-encrypted channels, ensuring confidentiality and integrity of messages in transit.

Through this implementation, I gained hands-on experience with TLS handshake, digital certificates, and secure socket programming. I also learned the importance of proper certification, peer authentication, and how small misconfigurations (such as disabling hostname checks) can introduce serious vulnerabilities.

This project really helped deepen my understanding of SIT202 concepts, especially in areas such as secure communication protocols, client-server architecture, and real-time networking. It helped bridge the gap between theoretical knowledge and practical security implementation

References

[1] Wikipedia Contributors, “Transport Layer Security,” *Wikipedia*, Mar. 26, 2019. https://en.wikipedia.org/wiki/Transport_Layer_Security

[2]“Network Security Tutorial.” Available: https://training.apnic.net/wp-content/uploads/sites/2/2016/12/TSEC01.pdf

[3]CREDO23, “Understanding SSL/TLS: The Key to Secure Online Communication,” *DEV Community*, Jun. 06, 2023. https://dev.to/kadea-academy/understanding-ssltls-the-key-to-secure-online-communication-19a1

[4]“TLS vs SSL – SSL Versus TLS Protocols Explained,” *Ssltrust.com.au*, 2020. https://www.ssltrust.com.au/learning/ssl/tls-vs-ssl (accessed May 14, 2025).

[5]“How TLS/SSL Certificates Work | DigiCert,” *www.digicert.com*. https://www.digicert.com/how-tls-ssl-certificates-work

[6]“Self Signed Certificate vs CA Certificate — The Differences Explained,” *SectigoStore*, Apr. 23, 2020. https://sectigostore.com/page/self-signed-certificate-vs-ca/

Module Docs

“Welcome to pyca/cryptography — Cryptography 3.0.dev1 documentation,” *cryptography.io*. https://cryptography.io/en/latest/

“ssl — TLS/SSL wrapper for socket objects — Python 3.7.2 documentation,” *Python.org*, 2018. https://docs.python.org/3/library/ssl.html

Final Code

GitHub Link: https://github.com/Clupai8o0/sit202-3.2h.git

`generate_certs.py`

```
from datetime import datetime, timedelta, timezone

from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa

def generate_cert(cert_name_prefix="server"):
  # 1. Generate a private key
  private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
  )

  # 2. Generating public key from private key
  public_key = private_key.public_key()

  # 3. Generate certificate using X509
  subject = issuer = x509.Name([
    x509.NameAttribute(NameOID.COMMON_NAME, u"localhost")
```

```
   ])
   # Generating a different Certificate Name for client
   if cert_name_prefix == "client":
     subject = issuer = x509.Name([
       x509.NameAttribute(NameOID.COMMON_NAME, u"client.localhost"),
     ])

   # 4. Build the certificate
   cert = x509.CertificateBuilder().subject_name(subject
   ).issuer_name(issuer
   ).public_key(public_key
   ).serial_number(x509.random_serial_number()
   ).not_valid_before(datetime.now(timezone.utc)
   ).not_valid_after(datetime.now(timezone.utc) + timedelta(days=365)
   ).add_extension(
     x509.SubjectAlternativeName([x509.DNSName(u"localhost")]),
     critical=False,
   ).sign(private_key, hashes.SHA256())

   # 5. Save the certificate and private keys
   # Save the private key
   with open(f"{cert_name_prefix}.key", "wb") as f:
     f.write(private_key.private_bytes(
       encoding=serialization.Encoding.PEM,
       format=serialization.PrivateFormat.PKCS8,
       encryption_algorithm=serialization.NoEncryption()
     ))
   # Save the certificate
   with open(f"{cert_name_prefix}.crt", "wb") as f:
     f.write(cert.public_bytes(serialization.Encoding.PEM))

   # print success output
   print(f"Generate {cert_name_prefix}.key and {cert_name_prefix}.crt successfully")

 if __name__ == "__main__":
   generate_cert("server")
   generate_cert("client")
```

server.py

```
import socket
import ssl
import threading
import logging
from datetime import datetime

# Setting up logging
logging.basicConfig(
  level=logging.INFO,
  format="%(asctime)s - %(levelname)s - %(message)s"
)
logger = logging.getLogger(__name__)

class SecureChatServer:
  def __init__(self, host="localhost", port=8443):
    self.host = host
    self.port = port
    self.clients = []
    self.setup_server()

  def setup_server(self):
    """Set up the server socket with SSL/TLS"""
    try:
      # Create base socket
      self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # setup
TCP socket
      self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # al
low address reuse
      self.server_socket.bind((self.host, self.port))
      self.server_socket.listen(5)

      # Create SSL context
      self.ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
      self.ssl_context.verify_mode = ssl.CERT_REQUIRED # create SSL context requiring c
lient authentication
      self.ssl_context.load_cert_chain(certfile="server.crt", keyfile="server.key") # load th
e server's certificate and private key
      self.ssl_context.load_verify_locations("client.crt") # load client's certifiate to verify t
hem
```

```
    logger.info(f"Server initialized and listening on {self.host}:{self.port}")
  except Exception as e:
    logger.error(f"Error setting up server: {e}")
    raise

def handle_client(self, client_socket, address):
  """Handle individual client connections"""
  try:
    # Wrap socket with SSL
    ssl_socket = self.ssl_context.wrap_socket(client_socket, server_side=True)
    logger.info(f"Secure connection established with {address}")

    # Log client certificate
    cert = ssl_socket.getpeercert()
    logger.info(f"Client certificate for {address}: {cert}")

    # Add client to list
    self.clients.append(ssl_socket)

    while True:
      try:
        # Receive message
        message = ssl_socket.recv(1024).decode('utf-8')
        if not message:
          break

        # Log and broadcast message
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        formatted_message = f"[{timestamp}] {address}: {message}"
        logger.info(formatted_message)

        # Broadcast to all clients
        self.broadcast(formatted_message, ssl_socket)

      except ssl.SSLError as e:
        logger.error(f"SSL Error: {e}")
        break
      except Exception as e:
        logger.error(f"Error handling message: {e}")
```

```
          break

      except Exception as e:
        logger.error(f"Error handling client {address}: {e}")
      finally:
        # Clean up
        if ssl_socket in self.clients:
          self.clients.remove(ssl_socket)
        ssl_socket.close()
        logger.info(f"Connection closed with {address}")

  def broadcast(self, message, sender_socket=None):
    """Broadcast message to all connected clients except sender"""
    for client in self.clients:
      if client != sender_socket:
        try:
          client.send(message.encode('utf-8'))
        except Exception as e:
          logger.error(f"Error broadcasting to client: {e}")
          if client in self.clients:
            self.clients.remove(client)

  def start(self):
    """Start the server and accept any connections"""
    logger.info("Server started. Waiting for connections")
    try:
      while True:
        client_socket, address = self.server_socket.accept()
        logger.info(f"New connection from {address}")

        # Start a new thread for each client
        client_thread = threading.Thread(
          target=self.handle_client,
          args=(client_socket, address)
        )
        client_thread.daemon = True
        client_thread.start()
    except KeyboardInterrupt:
      logger.info("Server shutting down...")
    finally:
      self.server_socket.close()

if __name__ == "__main__":
  server = SecureChatServer()
  server.start()
```

client.py

```
import socket
import ssl
import threading
import logging
import sys

# Setup logging
logging.basicConfig(
  level=logging.INFO,
  format="%(asctime)s - %(levelname)s - %(message)s"
)
logger = logging.getLogger(__name__)

class SecureChatClient:
  def __init__(self, host="localhost", port=8443):
    self.host = host
    self.port = port
    self.username = None
    self.setup_client()

  def setup_client(self):
    """Set up the client with SSL/TLS"""
    try:
      # Create base socket
      self.client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

      # Create SSL context
      self.ssl_context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
      self.ssl_context.load_cert_chain(certfile="client.crt", keyfile="client.key")
      self.ssl_context.check_hostname = False # for development only
      self.ssl_context.verify_mode = ssl.CERT_REQUIRED
      self.ssl_context.load_verify_locations("server.crt")
```

```
            # Wrap socket with SSL
            self.ssl_socket = self.ssl_context.wrap_socket(self.client_socket, server_hostname=
self.host)

            logger.info("Client initialized")
        except Exception as e:
            logger.error(f"Error setting up client: {e}")
            raise

    def connect(self):
        """Connect to the server"""
        try:
            self.ssl_socket.connect((self.host, self.port))
            logger.info(f"Connected to server at {self.host}:{self.port}")
            return True
        except Exception as e:
            logger.error(f"Error connecting to server: {e}")
            return False

    def send_message(self, message):
        """Send a message to the server"""
        try:
            # Format message with username
            formatted_message = f"{self.username}: {message}"
            self.ssl_socket.send(formatted_message.encode('utf-8'))
        except Exception as e:
            logger.error(f"Error sending message: {e}")
            return False
        return True

    def print_message(self, message):
        """Print a message while preserving the input prompt"""
        # Clear the current line
        sys.stdout.write('\r\033[K')
        # Print the message
        print(message)
        # Reprint the prompt
        sys.stdout.write('> ')
        sys.stdout.flush()
```

```
def receive_messages(self):
  """Receive messages from the server"""
  try:
    while True:
      message = self.ssl_socket.recv(1024).decode('utf-8')
      if not message:
        break
      self.print_message(message)
  except Exception as e:
    logger.error(f"Error receiving message: {e}")
  finally:
    self.ssl_socket.close()
    logger.info("Disconnected from server")

def start(self):
  """Start the client and handle user input"""
  if not self.connect():
    return

  # Get username
  while not self.username:
    self.username = input("Enter your username: ").strip()
    if not self.username:
      print("Username cannot be empty")

  # Send join message
  self.send_message("has joined the chat")

  # Start a receive thread
  receive_thread = threading.Thread(target=self.receive_messages)
  receive_thread.daemon = True
  receive_thread.start()

  try:
    while True:
      sys.stdout.write('> ')
      sys.stdout.flush()
      message = input()
      if message.lower() == 'quit':
          self.send_message("has left the chat")
          break
        if not self.send_message(message):
          break
    except KeyboardInterrupt:
      logger.info("Client shutting down...")
      self.send_message("has left the chat")
    finally:
      self.ssl_socket.close()

if __name__ == "__main__":
  client = SecureChatClient()
  client.start()
```