SOFTENG 364 Assignment 2

Introduction

The chat application features real-time, one-on-one messaging between users over a network, built using a three-tier client-server architecture comprising the presentation, application logic, and data layers.

Communication occurs over a secure TCP connection using the TLS protocol, ensuring encrypted data transmission. User credentials are securely stored using BCrypt within a NoSQL mock database, UnQLite, which supports both in-memory and disk-based storage. Further details are provided in the System Architecture and Security Measures sections.

System Architecture

The chat application follows a three-tier client-server architecture, as illustrated in ***Figure 1***. This architecture is structured into three layers: the presentation layer, the application logic layer, and the data layer.

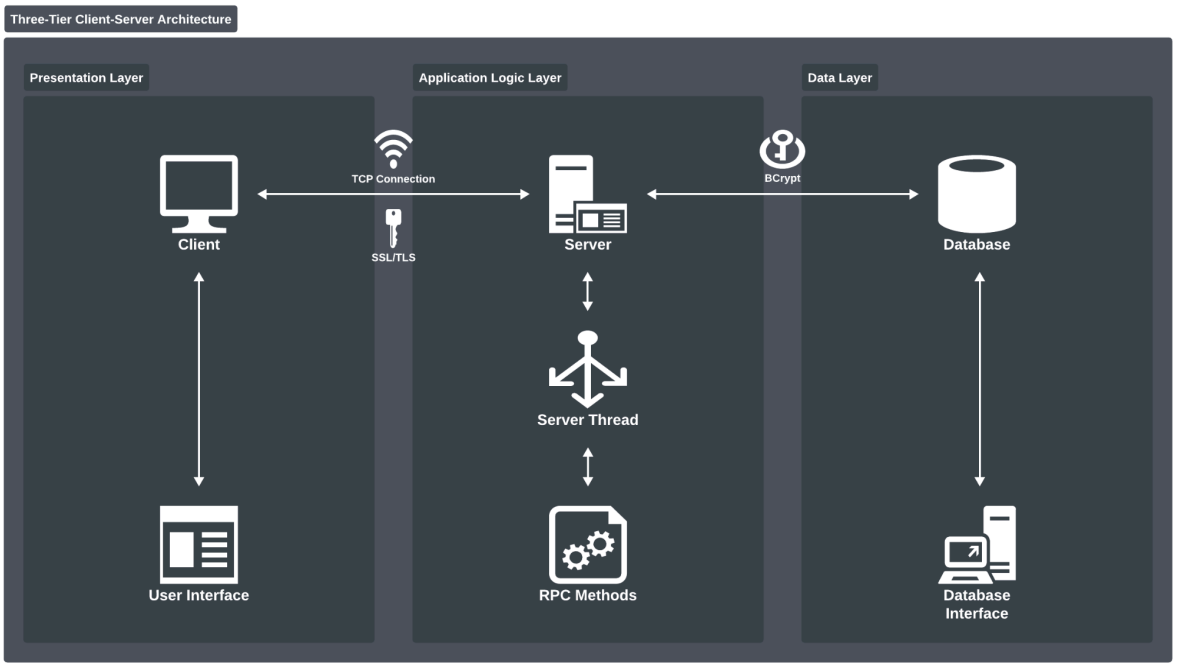

Figure 1 *- Three-Tier Client-Server Architecture*

Presentation Layer

The presentation layer includes the client and the user interface (UI), enabling the interaction between users and the server. The UI provides several key components: a login page for users with existing accounts to access the platform using their credentials, a sign-up page for new users to register, and a chat page where users can engage in a one-on-one conversation with another user.

Application Logic Layer

The application logic layer comprises the server, which acts as the intermediary between the client and the database. It handles all client requests through individual dedicated threads to avoid blocking input/output (I/O) operations. These requests invoke remote procedure call (RPC) methods on the server side, such as forwarding a client's message to other users.

Data Layer

The data layer includes the database, simulated by the NoSQL UnQLite mock database, which offers both in-memory and disk-based storage. A database interface manages operations between the server and the database. The database maintains two primary collections: a ***users*** collection, which stores user information such as usernames and passwords, and a ***messages*** collection, which records chat messages. Each message entry captures details like the content of the message, the associated user, and the message role (client or server), as both client messages and server notifications are stored.

Security Measures

To ensure robust application security, particularly in maintaining confidentiality, authentication, and message integrity, two essential files are generated: ***server.key*** and ***server.crt***. The ***server.key*** file contains the server's private key, and ***server.crt*** is the server's certificate, encompassing the server's public key and identity information. This certificate is self-signed, meaning it is signed using the server's own private key.

The generation of the server's public and private keys is performed using the OpenSSL RSA algorithm, which produces a 2048-bit key pair. This key pair ensures a high level of security through strong encryption. The code for generating the server keys and certificate is presented in ***Figure 2***.

```
#!/bin/bash

readonly KEY_PATH=.cache/keys
readonly CONFIG_PATH=.cache/configurations
readonly CERTIFICATE_PATH=.cache/certificates

mkdir -p ${KEY_PATH}
mkdir -p ${CERTIFICATE_PATH}

if [[ -f "${KEY_PATH}/server.key" && -f "${CERTIFICATE_PATH}/server.crt" ]]; then
    echo "Key and certificate already exist"
else
    echo "Generating key and certificate"
    openssl genrsa -out ${KEY_PATH}/server.key 2048
    openssl req -new -key ${KEY_PATH}/server.key -out ${CERTIFICATE_PATH}/server.csr -config ${CONFIG_PATH}/localhost.conf
    openssl x509 -req -days 365 -in ${CERTIFICATE_PATH}/server.csr -signkey ${KEY_PATH}/server.key -out ${CERTIFICATE_PATH}/server.crt -extensions v3_req -extfile ${CONFIG_PATH}/localhost.conf
    echo "Key and certificate generated"
fi
```

Figure 2 *- Code for Generating the Server's Key-Pair and Certificate*

To establish a secure TCP connection, both the client and server sockets are secured using SSL/TLS. The configurations for these secure sockets are provided in ***Figures 3*** and ***4***, respectively.

```
def get_secure_socket(self) -> SSLSocket:
    """Returns a secure socket wrapped with a TLS protection layer. It also
    allows for self-signed certificates to be verified.

    Returns: a secure socket wrapped with a protection layer; TLS
    """

    unsecure_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    context = SSLContext(PROTOCOL_TLSv1_2)
    context.load_verify_locations(self.__CERTIFICATE)
    context.set_ciphers(CIPHER)

    return context.wrap_socket(unsecure_socket, server_hostname=self.__HOST)
```

Figure 3 *- Client Socket Configuration*

```
def get_secure_socket(self) -> SSLSocket:
    """Returns a secure socket wrapped with a TLS protection layer.

    Returns: a secure socket wrapped with a protection layer; TLS
    """

    unsecure_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    context = SSLContext(PROTOCOL_TLSv1_2)
    context.load_cert_chain(certfile=self.__CERTIFICATE, keyfile=self.__KEY)
    context.load_verify_locations(self.__CERTIFICATE)
    context.set_ciphers(CIPHER)

    return context.wrap_socket(unsecure_socket, server_side=True)
```

Figure 4 *- Server Socket Configuration*

When a client initiates a connection to the server, a TLS handshake is triggered to establish a secure communication channel. During this handshake, the client and server exchange "hello" messages to negotiate the cipher suites to be used for the session. Once this negotiation is complete, the server sends its certificate (***server.crt***) to the client. This certificate contains the server's public key, which the client uses to verify the server's identity, ensuring that it is communicating with the legitimate server.

After successful verification of the server's certificate, the client generates a pre-master secret (a random byte sequence). This pre-master secret is encrypted with the server's public key and transmitted to the server. Both the client and server then use this pre-master secret, along with their respective nonces (random numbers generated

during the handshake), to independently compute a master secret. From this master secret, both parties derive a symmetric session key. This symmetric session key is utilised by the ***TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384*** [1] cipher to encrypt and decrypt all data transmitted during the session.

Additional Security Measures

To enhance the existing security measures, users are required to sign up or log in with valid credentials, including a username and password. User passwords are securely stored as cryptographic hashes in the database, utilising the BCrypt library to ensure protection against unauthorised access.

Wireshark

The Wireshark capture encapsulated in ***Figure 5***, shows the “hello” messages exchanged between the client and server during the TLS handshake. As described in previous sections, the server sends its certificate to the client, which includes the server’s public key.

Following the certificate exchange, the client initiates the client key exchange phase, where it generates the pre-master secret. This pre-master secret is encrypted with the server's public key and sent to the server to establish a secure connection. Additionally, the cipher specification is negotiated during this exchange, finalising the algorithms that will be used for encryption and integrity throughout the session.

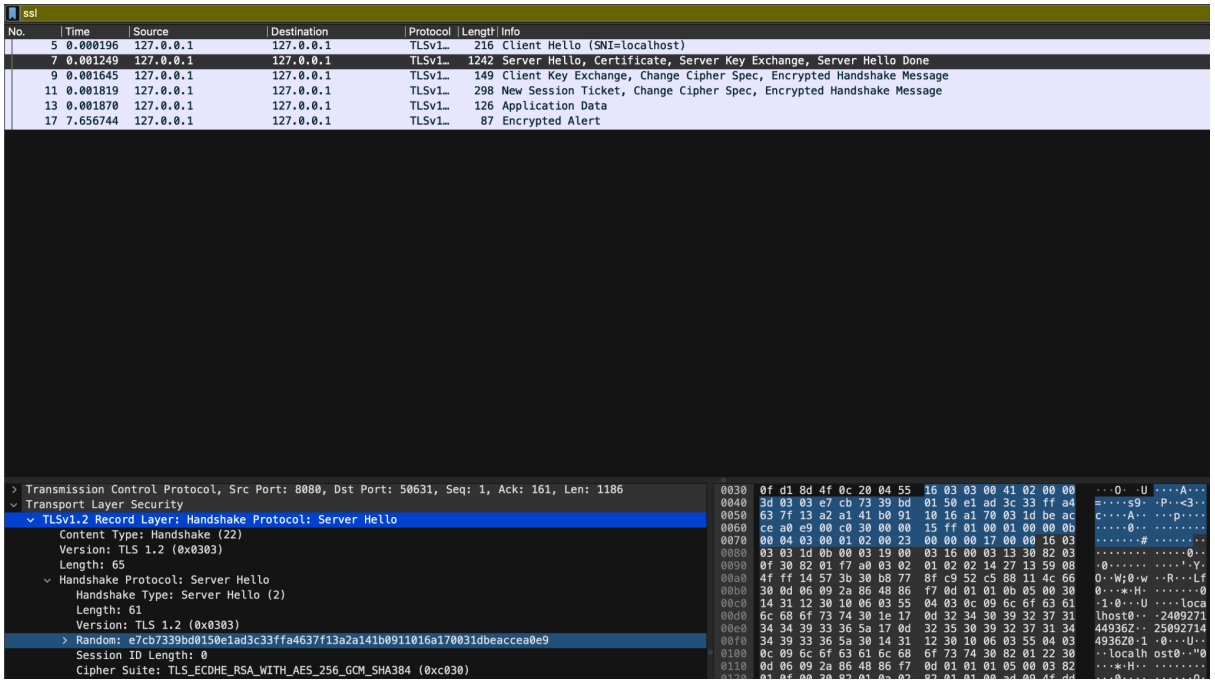

Figure 5 *- Wireshark Capture for TLS Handshake and Encryption*

Conclusion

The chat application employs a three-tier client-server architecture that ensures efficient communication and data management. Security is a priority, with the use of TLS to establish secure connections, encrypting data with symmetric key encryption after the handshake.

The Wireshark analysis confirms the secure exchange of keys and encryption protocols, ensuring confidentiality, integrity, and authentication. Additionally, user credentials are securely managed using BCrypt hashing to prevent unauthorised access. This combination of architectural design and security measures provides a reliable and secure environment for communication between users.

References

[1] Rudolph, H. C. (n.d.-b). *Ciphersuite info*. https://ciphersuite.info/cs/TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384/