Practical Cryptography

Coursework

Elia Mamdouh Samy

TKH ID:- 202100513

Source code:-

https://github.com/eliamamdouh/CW_Cryptography_Elia_Mamdouh.git

Table of Contents

Introduction

The idea of my application was to build a secure client-server chatroom application that allows multiple clients to connect to the same server. The application uses three different cryptographic encryption mechanisms to ensure that the application maintains confidentiality, integrity, and secure authentication for the client connecting. The application uses RSA to ensure that the messages that are being sent between the client and the server are encrypted on both sides to ensure that only the intended recipient can read the message. Our second encryption mechanism is AES, which encrypts the conversation after it is saved to ensure that no one can read it unless they have the encryption key. The last encryption mechanism we have is hashing the passwords of the users in a separate file to ensure that no one can read them.

The objective of the application is to provide a secure user experience when chatting with other clients on the server, from having secure user authentication to having secure communication via encryption, protecting user credentials, and mitigating common security attacks. In the next part of the report, I will dive deeper into the encryption techniques we used in this application and explain how we implemented and used them.

The Cryptographic Encryption Techniques Used

In this section, I will dive deeper into the 3 encryption mechanisms we used in this application and explain in detail how they work and how they are implemented in my application as well as the attacks they can protect the application from and how they do that.

The 3 encryption techniques:-

1- Hashing algorithm for password storing
2- RSA encryption for message transfer
3- AES for storing the conversation encrypted

The Hashing Algorithm

The application utilizes the SHA-256 hashing algorithm, a secure method that generates a 256-bit (32-byte) hash value from any input data. This ensures data integrity and security, as

hashing is a one-way function; thus, the original password cannot be easily retrieved from the hash output.

In the application, the SHA-256 hashing algorithm is used to hash user passwords before storing them in the user_credentials text file. This file is responsible for holding all user credentials that are referenced during user authentication. By hashing the passwords, we prevent them from being stored in plaintext. Consequently, if the credentials file were to be compromised, attackers would only access the hashed versions of the passwords, rather than the original values.

During the login process, when a user enters their password, it is hashed and then compared to the stored hash for authentication. The application operates by reading the hash value, not the original password.

Attacks that can happen to my hashing algorithm and how SHA-256 and the security mechanisms I implemented help in mitigating these attacks:-

1- The first attack that can happen is the brute-force attack, where the attacker tries all possible combinations of characters to find a match for the hash. Our hashing algorithm and security mechanisms help mitigate this attack by requiring our users to have a minimum of 8 characters, which must include numbers, letters, and special characters, significantly increasing the total number of possible combinations and making brute force computationally expensive. The SHA-256 itself has a high computational cost, which slows down each brute-force attempt compared to other hashing algorithms.

2- The second attack is the dictionary attack, in which attackers use a list of precomputed hash values for common passwords, use another list for common usernames, and try to match any one of them to get access to any valid credentials. Our application helps mitigate this type of attack by ensuring password strength requirements: Your system makes it less probable that users use poor, common password choices by enforcing mixed-case letter combinations, numbers, and special characters. Lack of Predictability: The more complex the password, the more likely users are to avoid predictability; thus, reducing the rate of a successful attack.

3- The third attack we are going to talk about is the rainbow table attack, in which attackers use a precomputed table of hashes for every possible password to reverse the hash and

retrieve the original plaintext password. But because our system used a complex password, so it Is less likely to appear in precomputed rainbow tables, making the attack less effective.

Recommendations in the future for better hashing security mechanisms:-

1) Salting Implementation: One unique salt per user could give extra security against rainbow table and dictionary attacks.
2) Regular Password Audits: Changes in passwords by users should be encouraged periodically to reduce the risk of compromised password reuse.

The RSA Encryption

The application uses RSA encryption, which is a widely known asymmetric algorithm, to ensure secure communication between the client and the server and the same thing in the other way, which means that the RSA ensures that the messages that have been sent from the client to the server and from the server to the client are both securely encrypted using RSA keys, that only allow the intended recipient to be able to read it. The RSA relies on a pair of keys, as we said: a public key and a private key. The public key is used to be sent to the intended recipient for the encrypting of the data, and then when the data is encrypted, it gets sent again to be decrypted by the private key; this ensures that even if an attacker intercepts the encrypted message, they cannot decrypt it without access to the private key.

In our application, RSA encryption is used for encrypting and securely transmitting sensitive data between the messages that are being sent across the server and the client, as well as secure transmitting between the signup and login processes which we will explain in detail later in the report. This ensures if an attacker intercepts the data, they would not be able to decrypt it without access to the corresponding private key.

The application works by first having the server generate a public-private key pair using the RSA algorithm. Then, the public key is shared with the clients, while the private key sits with the server for future decryption processes. So, when a client sends sensitive information such as login credentials, it gets encrypted by the server's public key before sending. This means that whether it is a login or signup process, before sending the credentials to the server, it gets encrypted by the server's public key and then sent to be decrypted in the server with the server's

private key and then gets compared with the hash to allow the login. Regarding the messages that are being sent across the server and client, when the client sends a message, it first gets encrypted using the server's public key and then gets decrypted at the server, and the same thing happens on the server side; any message is first encrypted using the client's public key and then gets sent to be decrypted with the client's private key.

Attacks that can happen to the RSA encryption and how RSA and the security mechanisms I implemented help in mitigating these attacks:-

1- Man-in-the-Middle (MitM) Attack:-

In a typical MitM attack, communications between the client and server are intercepted by the attacker with the intention of modifying the data.

RSA reduces this possibility since even after the interceptor has the encrypted data, he doesn't have the server's private key to decrypt or modify the message; it is possible only at the server.

2- Eavesdropping:-

Since this communication is done in plain text from client to server, an attacker could sniff it easily and get the credentials.

RSA encryption ensures that even if the data gets intercepted, it is unreadable without the private key, hence safeguarding users' sensitive data.

3- Replay Attacks:-

A replay attack is when an attacker replays a certain packet that was responsible for successful log-in for example, so RSA prevents the playback of any of the intercepted data. If each message is differently encrypted with another server's public key, an attacker cannot replay an old message to impersonate the user. Another security step we have is that each server has a session ID, which will ensure that the attacker cannot replay an old message from a different server.

Recommendations in the future for better RSA security mechanisms:-

- Key Rotation**:** Regularly changing RSA keys helps limit the risk of long-term exposure if a key is compromised.

- Stronger Key Generation**:** Using larger RSA keys (like 4096 bits) makes encryption stronger but can slow down performance.
- SSL/TLS Encryption**:** Adding SSL/TLS encryption ensures the entire communication is secure, protecting both data and the connection from eavesdropping.

The AES Encryption

AES encryption is a widely known symmetric encryption algorithm, that ensures the confidentiality of the data by encrypting it with a single key, this is why it is a symmetric encryption algorithm unlike RSA, which is asymmetric and requires both a public and a private key, AES uses the same key for both encryption and decryption, making it efficient for large amounts of data. In our application, AES encryption is used to securely encrypt and decrypt the conversation data that gets saved when the server closes, which holds the conversation between the clients. This ensures that even if an attacker intercepts the encrypted data that gets stored and saved in a file, they cannot decrypt it without access to the correct encryption key.

The workflow of the AES encryption in our application goes as follows:

3) It first starts with the key derivation, in our case, we are using the Password-based key derivation function (PBKDF32), which derives the key for the AES encryption using a password and a salt that we configure In the code to be used to derive the key. The password is set to “PrivateConvo123”, which Is a shared secret between the server and the clients. Then we have the unique salt “UniqueSaltHere”, that ensures that the key derivation is unique for each instance, even if the same password is used. The two inputs to the PBKDF2 derive a 256-bit key for AES. It follows then that even if the password is poor, it has a strong key derivation because of the amount of computational effort needed to derive the key.

4) Then we have the padding because AES operates on fixed block sizes of 128 bits, so if the data that is being encrypted is not a multiple of the block size, it must be padded to ensure that the data fits into the AES blocks. The padding function(PKCS7) used in the application adds extra bytes to the data until its length is a multiple of the block size, ensuring proper encryption.

5) Then we have the encryption and decryption process itself, which works by using the AES in CBC (Cipher Block Chaining) mode, which combines each plaintext block with the previous cipher text before encryption. This works as follows: the first cipher text is padded first to fit the 16byte fixed block for the AES and then it gets combined with the IV (initialized vector), which is the first 16byte of the AES key, which is 256 bits (32 bytes). They are combined using the XOR operation and then produce the result, then we encrypt this result using the key, so we combine the plaintext with the Iv and then XOR them together to encrypt the result of that. The next plaintext block will then be the same but for the IV of the second block, it will be the cipher text we got from the first block after encrypting it, then they got XORed and then encrypted to produce the second block, and so on. This ensures that identical blocks of plaintext will produce different ciphertexts, improving security. After encrypting at the end, it gets encoded by base64 for transmission. Upon receiving the encrypted conversation, the data is first decoded from Base64. The ciphertext is then decrypted using the same AES key and the derived IV, returning the original plaintext.
6) Messages of conversations are encrypted with AES, ensuring confidentiality and integrity from client to server. Messages stored in a file do so in encrypted format; this file is decryptable by only the server or clients who also possess the correct password and salt to generate the AES key.

Attacks that can happen to the AES encryption and how AES and the security mechanisms I implemented help in mitigating these attacks:-

- Brute Force Attack: AES-256 will resist brute-force attacks since its key space is comparably very huge, as much as 2^256 possible keys, which will make it hard for an attacker to decrypt data without a valid key.
- Chosen Plaintext or Ciphertext Attack: Also, to make this type of attack ineffective, AES in CBC mode XORs each plaintext block with the previously sent ciphertext block, making it computationally infeasible for an attacker to extract information from the ciphertext.
- Replay Attacks: AES can be combined with either session IDs, timestamps, or nonces, enabling the generation of unique messages that cannot be used in replay attacks.

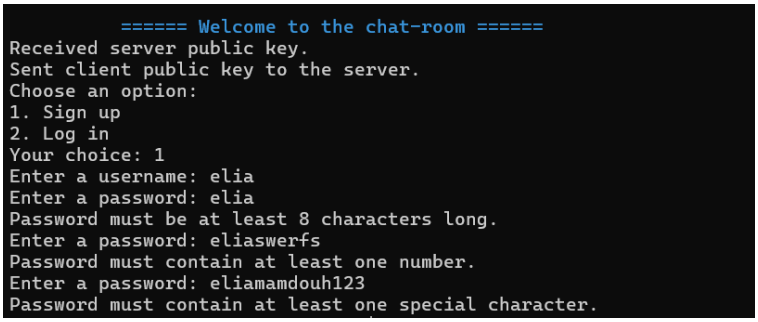
Recommendations in the future for better RSA security mechanisms:-

- Key Rotation: Routine rotation of AES keys reduces exposure time and minimizes risk in case one is compromised and thus keeps encrypted data continually secure.
- Stronger Salt and Key Derivation: modern key derivation functions like Argon2 provide better resistance against brute-force and dictionary attacks because of their strong salts, which increase key security.
- SSL/TLS Encryption: It provides full-channel encryption with SSL/TLS along with AES; therefore, any man-in-the-middle attack is prohibited, and integrity and authenticity are guaranteed.

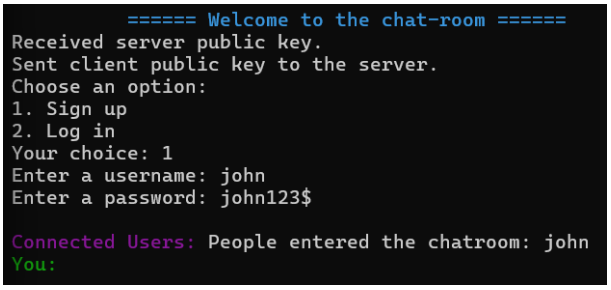
Test Cases for everything in the application

1) showing failed and successful sign-up attempts

Failed:-



Successful:-



2) showing failed and successful login-up attempts

Failed:

```
          ====== Welcome to the chat-room ======
Received server public key.
Sent client public key to the server.
Choose an option:
1. Sign up
2. Log in
Your choice: 2
Enter your username: elia
Enter your password: elia
Operation failed. Please check your credentials or try again.
```

Successful:

```
          ====== Welcome to the chat-room ======
Received server public key.
Sent client public key to the server.
Choose an option:
1. Sign up
2. Log in
Your choice: 2
Enter your username: elia
Enter your password: eliamam123$

Connected Users: People entered the chatroom: elia
You:
```

3) the server accepts clients and shows the session ID that gets created each time

```
          ====== Welcome to the chat-room ======
Server is running. Session ID: 38d8909d-eb77-4ac7-9a24-76457dcad2dc
```

```
Connection accepted from ('127.0.0.1', 33875).
Received name: elia
elia has joined
```

4) the server and client interface when several clients join in

The server-side chatroom:

```
Connection accepted from ('127.0.0.1', 33875)
Received name: elia
elia has joined
elia: hello
elia: welceme
elia: to
elia: this chatroom
Connection accepted from ('127.0.0.1', 33888)
Received name: samy
samy has joined
samy: hello
samy: welceom
samy: to
samy: my
samy: chatrrom
```

The client-side chat room:

```
      ====== Welcome to the chat-room ======
Received server public key.
Sent client public key to the server.
Choose an option:
1. Sign up
2. Log in
Your choice: 2
Enter your username: samy
Enter your password: samy123

Connected Users: People entered the chatroom: elia, samy
You: hello
You: welceom
You: to
You: my
You: chatrrom
[2024-12-03 22:20:09] elia: welcoe
[2024-12-03 22:20:11] elia: okay
You:
```

5) showing that the RSA encryption is working on the client and server side

The client-side:

```
      ====== Welcome to the chat-room ======
Received server public key.
Sent client public key to the server.
```

The server side:

```
john has left
Connection closed for john.
This conversation with john was successfully encrypted using RSA Encryption.
```

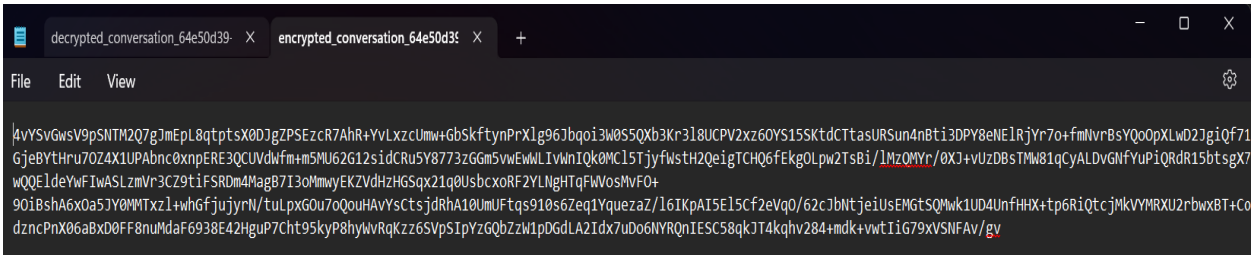
6) showing that the SHA-256 encryption works successfully

```
elia 14864b8099208730833cbb6da2fecc43943a711a9218a497d1d59b4d13a037b8
mamdouh fe32ab34604b641b7be90421889b06068dcf1e926f2347b9b9085cd9445e775e
ramez 271fea9e82ade7fb704fde86cc53549f8496ae1269168feffa1b869e343a9471
john d70097baed3078a124763b25ca25daa8780a42dd62278942212b8192799e6f9d
```

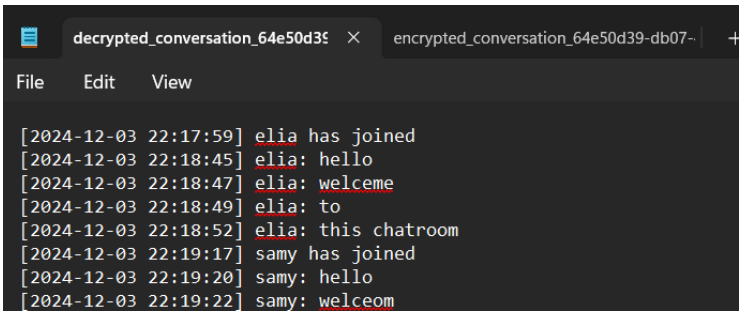
7) showing that the AES encrypts and decrypts the conversation and saves it with the session ID

```
Shutting down the server...
Encrypted conversation saved as: encrypted_conversation_64e50d39-db07-4bf0-8
d4c-9e657a77d665.txt
Decrypted conversation saved as: decrypted_conversation_64e50d39-db07-4bf0-8
d4c-9e657a77d665.txt
Server shut down successfully.
```

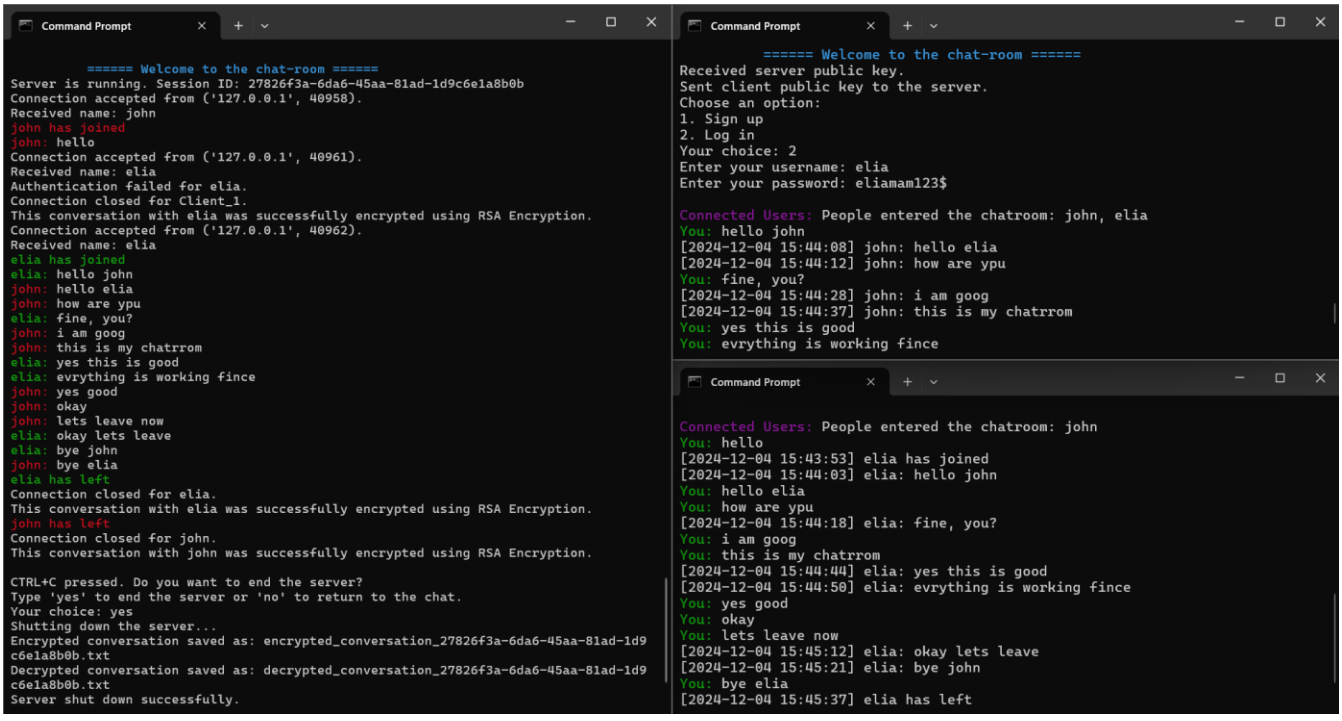
8) The encrypted version of the conversation using AES:



The decrypted version of the conversation using AES:



9) A full working chatroom with multiple clients that shows everything in the application:



Explanation of both the Server and Client code

The client code:-

```python
import socket
import threading
import signal
import sys
import re
import os
import hashlib
import base64
import struct
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import padding, rsa

MAX_LEN = 200
colors = ["\033[31m", "\033[32m", "\033[33m", "\033[34m", "\033[35m", "\033[36m"]
def_col = "\033[0m"

client_socket = None
exit_flag = False
server_public_key = None  # Store the server's public key
client_private_key = None  # Store the client's private key
```

I will begin by explaining the client code and how our application performs all the encryption techniques and comes. The first thing is to import the necessary libraries that we are going to use their functions to perform all the functionalities of the application. To enable safe network connectivity and data handling, the code imports a socket library. It makes use of threading to allow for simultaneous message sending and receiving and sockets for network interactions. While re and os provide input validation and file system interfaces, signal and sys handle system-level functions such as signal handling and system controls. Base64 and hashlib are used for safe encoding and data integrity, while struct helps with binary data management. The cryptography.hazmat.primitives module powers cryptographic operations, providing RSA encryption/decryption, hashing, padding, and key serialization to guarantee communication secrecy and authenticity.

Next, we defined some variables such as a placeholder for the client's socket, a flag to indicate when the client should terminate the connection, and 2 variables to store the server's public key and the client's private key to be used later in the RSA encryption.

```python
def catch_ctrl_c(_signal, _frame):
    global exit_flag
    exit_flag = True
    str_msg = "#exit"
    try:
        encrypted_msg = encrypt_message_to_server(str_msg)
        send_encrypted_message(encrypted_msg)
    except Exception as e:
        print(f"Error while sending exit message: {e}")
    client_socket.close()
    os._exit(0)

def encrypt_message_to_server(message):
    global server_public_key
    if not server_public_key:
        print("Server public key not available. Cannot encrypt message.")
        return b''

    encrypted = server_public_key.encrypt(
        message.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    # Encode the encrypted message in Base64
    encrypted_base64 = base64.b64encode(encrypted)
    return encrypted_base64
```

The image above contains 2 functions, that we are going to explain:

7) The catch_ctrl_c function, which handles the CTRL+C signal, happens when the user presses the signal, it encrypts an exit message, sends it to the server, and sets the flag to true to ensure that the client wants to close the connection, and at then it closes the client socket and exits the program
8) The encrypt_message_to_server function encrypts the message using the server's public key with the RSA encryption method. The encryption uses the OAEP (Optimal Asymmetric Encryption Padding) scheme, which applies a random seed (called a *masking function*) to the message before combining it with the message. This process is followed by hashing the combined result using the SHA-256 hashing algorithm. The resulting data is then encrypted using the server's public key. Finally, the encrypted message is encoded in Base64 to safely transmit it over a text-based protocol.

```python
def send_encrypted_message(encrypted_msg):
    # Send the length of the message first (4 bytes, big-endian)
    msg_length = struct.pack('>I', len(encrypted_msg))
    client_socket.sendall(msg_length + encrypted_msg)

def send_message(client_socket):
    global exit_flag
    while not exit_flag:
        try:
            msg = input(colors[1] + "You: " + def_col)
            if msg:  # Only send if the message is not empty
                encrypted_msg = encrypt_message_to_server(msg)
                send_encrypted_message(encrypted_msg)
            if msg == "#exit":
                exit_flag = True
                return
        except Exception as e:
            print(f"Error occurred while sending message: {e}")
            break
```

The image above contains 2 functions, that we are going to explain:

9) The send_encrypted message is responsible for sending the encrypted message to the server. The first thing for the server to correctly interpret the size of the message and send it, it calculate the length of the encrypted message and pack it into 4 bytes. Then the packed length and the encrypted message are sent to the server using the "client_socket.sendall()". Sending the length first allows the server to know how much data it is going to expect when receiving the message.

10) Then we have the send_message function, which handles the process of reading the messages from the user and sending them to the server. To constantly ask the user for input, it runs a loop to keep the user always ready to send a message. If the user sends a message, it gets encrypted using the encrypt_message_to_server function, and then the message is sent to the server using the function we defined earlier. The loop breaks if only the #exit flag is set to true, which terminates the session and stops the user from sending any more messages.

```python
def recv_message(client_socket):
    global exit_flag
    while True:
        if exit_flag:
            return
        try:
            # Receive the length of the message first
            raw_msglen = recvall(client_socket, 4)
            if not raw_msglen:
                break
            msglen = struct.unpack('>I', raw_msglen)[0]
            # Receive the encrypted message data
```

```
            encrypted_message_base64 = recvall(client_socket, msglen)
            if not encrypted_message_base64:
                continue
            # Decrypt the message using client's private key
            msg = decrypt_message_from_server(encrypted_message_base64)
            if not msg:
                continue
            print("\033[2K\r" + msg)
            print(colors[1] + "You: " + def_col, end='', flush=True)
        except Exception as e:
            print(f"Error receiving message: {e}")
            break

def recvall(sock, n):
    # Helper function to receive n bytes or return None if EOF is hit
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None
        data += packet
    return data
```

The image above contains 2 functions, that we are going to explain:

11) The recv_message function is responsible for handling the received message from the server, which continuously listens for messages from the server until the flag is set and exits the loop. At first, it receives the length of the incoming message using the recvall function (this function is to ensure that the full 4-byte length is received from the socket), then unpacks the length using “struct.unpack” to tell us the actual length of the message in integers. After that, the server sends the actual encrypted message to the client, because the server sends the message in two parts: one time for the length to tell the client how many bytes to expect for the encrypted message, and the second part is the client reads exactly the length to receive the actual encrypted message. The “recvall” function ensures that the client gets the full length at first.

```python
def signup_user(client_socket):
    username = input("Enter a username: ")

    # Password validation loop
    while True:
        password = input("Enter a password: ")
        if len(password) < 8:
            print("Password must be at least 8 characters long.")
        elif not re.search(r"[A-Za-z]", password):
            print("Password must contain at least one letter.")
        elif not re.search(r"\d", password):
            print("Password must contain at least one number.")
        elif not re.search(r"[!@#$%^&*(),.?\":{}|<>]", password):
            print("Password must contain at least one special character.")
        else:
            break  # Exit loop if password meets all criteria

    # Hash the password before sending
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    credentials = f"{username} {hashed_password}"

    try:
        # Encrypt the action indicator and credentials
        action_encrypted = encrypt_message_to_server('SIGNUP')
        credentials_encrypted = encrypt_message_to_server(credentials)

        # Send the encrypted action indicator and credentials
        send_encrypted_message(action_encrypted)
        send_encrypted_message(credentials_encrypted)

        # Receive and decrypt the response
        response = receive_encrypted_response()
        return response == '1'
    except Exception as e:
        print(f"An error occurred during signup: {e}")
        return False
```

The signup function in the image above is responsible for prompting the user with his credentials including a username and password, and then validating the password to meet several criteria, including the length, contain numbers, letters, and special characters. Then the password is hashed using the SHA-256 hashing algorithm before being sent with the username to the server. Then both the action “SIGNUP” and the credentials are encrypted using the “encrypt_message_to_server” function and then transmitted to the server. The credentials in the signup undergo 2 encryptions, first the hashing then it takes this hash and encrypt it using RSA and then gets transmitted to the server. Then the server’s response to indicate whether it was a successful signup process or not gets received, decrypted, and checked to determine if the signup is successful. Any error that is caught in the process is being handled to be displayed.

```python
def login_user(client_socket):
    username = input("Enter your username: ")
    password = input("Enter your password: ")
    # Hash the password before sending
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    credentials = f"{username} {hashed_password}"

    try:
        # Encrypt the action indicator and credentials
        action_encrypted = encrypt_message_to_server('LOGIN')
        credentials_encrypted = encrypt_message_to_server(credentials)
        # Send the encrypted action indicator and credentials
        send_encrypted_message(action_encrypted)
        send_encrypted_message(credentials_encrypted)
        # Receive and decrypt the response
        response = receive_encrypted_response()
        return response == '1'
    except Exception as e:
        print(f"An error occurred during login: {e}")
        return False

def receive_encrypted_response():
    try:
        # Receive the length of the message first
        raw_msglen = recvall(client_socket, 4)
        if not raw_msglen:
            return None
        msglen = struct.unpack('>I', raw_msglen)[0]
        # Receive the encrypted response
        encrypted_response = recvall(client_socket, msglen)
        if not encrypted_response:
            return None
        # Decrypt the response using client's private key
        response = decrypt_message_from_server(encrypted_response)
        return response
    except Exception as e:
        print(f"Error receiving encrypted response: {e}")
        return None
```

The 2 functions that will be explained now are the login_user and the receive_encrypted_response functions. The login_user function first takes the username and password that is entered, then it hashes the password because the way hashes work is that they are not authenticated by decrypting the hash, no, but by hashing the password given and comparing it to see if the hash is there or not. Then the function encrypts the login credentials sends them to the server and waits for the server's response to check whether the login was successful or not. The second function is responsible for receiving the encrypted server response and then decrypting it using the client's private key to uncover the real message using the decrypt_message_from_server function, and then lastly returning the decrypted response.

```python
def display_users(client_socket):
    # Encrypt and send the request to get the user list
    encrypted_msg = encrypt_message_to_server('GET_USERS')
    send_encrypted_message(encrypted_msg)
    # Receive the length of the message first
    raw_msglen = recvall(client_socket, 4)
    if not raw_msglen:
        print("Failed to receive user list.")
        return
    msglen = struct.unpack('>I', raw_msglen)[0]
    # Receive the encrypted message data
    encrypted_user_list = recvall(client_socket, msglen)
    if not encrypted_user_list:
        print("Failed to receive user list.")
        return
    # Decrypt the user list
    user_list = decrypt_message_from_server(encrypted_user_list)
    if user_list:
        print(colors[4] + "\nConnected Users: " + def_col + user_list)
    else:
        print("Failed to decrypt user list.")

def receive_server_public_key():
    global server_public_key
    try:
        pem_public_key = client_socket.recv(2048)
        server_public_key = serialization.load_pem_public_key(pem_public_key)
        print("Received server public key.")
    except Exception as e:
        print(f"Failed to receive server public key: {e}")
        sys.exit()
```

The first function is responsible for getting the user list by name to display it on the client side, this is done by first encrypting a message “GET_USERS”, this message is intended to request the user list from the server after the message gets encrypted, encoded, and sent to the server. Then it receives the length of the encrypted message, then the encrypted user list, and finally decrypts it to display the list of connected users. If any step fails, an error message is printed.

The second function is simple, all it does is receive the server’s public key that will used for encrypting anything we want to send to the server, it will receive the key in PEM format from the server, load it using the cryptography library to make it usable for encryption, and stores it in a global variable server_public_key.

```python
def send_client_public_key(client_public_key):
    try:
        pem_public_key = client_public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        client_socket.sendall(pem_public_key)
        print("Sent client public key to the server.")
    except Exception as e:
        print(f"Failed to send client public key: {e}")
        sys.exit()
```

```python
def decrypt_message_from_server(encrypted_message_base64):
    global client_private_key
    try:
        encrypted_message = base64.b64decode(encrypted_message_base64)
        decrypted = client_private_key.decrypt(
            encrypted_message,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return decrypted.decode()
    except Exception as e:
        print(f"Error decrypting message from server: {e}")
        return None
```

The first function is responsible for sending the client's public key to the server for the server to encrypt everything with its, in order when it gets to the client side it can get decrypted using the client's private key, the same as we did when we declared a function to receive the server's public key. This works by getting the client's public key and converting it to PEM format using public_bytes(), which serializes it into a byte string to be sent to the server. In the end, if successful, a confirmation message is printed; otherwise, it prints the error and exits the program.

The second function is responsible for decrypting messages
that the server has encrypted with the client's private key. The function works as follows:

- encrypted_message_base64 is first decoded into its original encrypted byte form in Base64.
- The encrypted message is then decrypted using the client's private key via the OAEP padding scheme, using SHA-256 for both the hashing algorithm and the mask generation function.
- If decryption is successful, it decodes the decrypted message bytes to string and returns. In case of an error, it prints an error message and returns None

It reverses the process of encrypting a message.

Then we have the last function, the main function of the file, which tells which function is going to execute and in which order, it is the building block of the code. It shows the flow of the application and shows how the application becomes together as a whole.

The main function:

1- Socket Creation: A socket is created, using client_socket, used for establishing a TCP connection to the server at 127.0.0.1 on port 10000.
2- Connection Handling: This program attempts to connect to a server, with an error in case of invalid connections. If it can't, the program exists.
3- Signal Handling: In fact, it sets the signal handler for SIGINT (example CTRL+C) to handle that with the catch_ctrl_c function to disconnect cleanly.
4- Server Public Key: The method obtains the server's public key by calling receive_server_public_key.
5- RSA Key Pair: The method rsa.generate_private_key will generate a random RSA key pair in a private key for the client.
6- Public Key Send: This method, send_client_public_key will send the client's public key to the server.
7- It asks the user if he wants to sign up or log in. According to the input, it calls a function signup_user or login_user.
8- Login Check: The program exists in case of failure of login due to disconnection.
9- Display Users: It calls the display_users function passing an argument to get and show the list of online users.
10- Threads for Messaging: Creation of two threads in:-
    - t_send: This handles sending messages to the server via send_message.
    - t_recv: Receives messages from the server by calling recv_message.
11- Start and join threads Explanation This starts the threads and then joins them, meaning the program waits for them to finish

```
244  def main():
245      global client_socket, client_private_key
246      client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
247
248      server_address = ('127.0.0.1', 10000)
249
250      try:
251          print("Attempting to connect to the server...")
252          client_socket.connect(server_address)
253          print("Connected to the server.")
254      except ConnectionRefusedError:
255          print("Unable to connect to the server. Please make sure the server is running.")
256          sys.exit()
257      except Exception as e:
258          print(f"Error occurred while connecting: {e}")
259          sys.exit()
260
261      signal.signal(signal.SIGINT, catch_ctrl_c)
262
263      print(colors[5] + "\n\t  ====== Welcome to the chat-room ======   " + def_col)
264
```

```
    # Receive the server's public key
    receive_server_public_key()

    # Generate client's RSA key pair
    client_private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    client_public_key = client_private_key.public_key()

    # Send the client's public key to the server
    send_client_public_key(client_public_key)

    choice = int(input("Choose an option:\n1. Sign up\n2. Log in\nYour choice: "))

    login_success = False
    if choice == 1:
        login_success = signup_user(client_socket)
    elif choice == 2:
        login_success = login_user(client_socket)
    else:
        print("Invalid choice. Exiting...")
        client_socket.close()
        sys.exit()

    if not login_success:
        print("Operation failed. Please check your credentials or try again.")
        client_socket.close()
        sys.exit()

    display_users(client_socket)

    t_send = threading.Thread(target=send_message, args=(client_socket,))
    t_recv = threading.Thread(target=recv_message, args=(client_socket,))

    t_send.start()
    t_recv.start()

    t_send.join()
    t_recv.join()

if __name__ == "__main__":
    main()
```

The Server code

```
import socket
import threading
import signal
import sys
import hashlib
import base64
import os
import uuid
import struct
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding as sym_padding, hashes, serialization
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.backends import default_backend
from datetime import datetime
from cryptography.hazmat.primitives.asymmetric import rsa, padding as asym_padding

# Server Configuration
MAX_LEN = 200
NUM_COLORS = 6
ENCRYPTION_PASSWORD = "PrivateConvo123"
ENCRYPTION_SALT = b"UniqueSaltHere"  # Ensure this is unique and consistent
AES_KEY_LENGTH = 32  # 256-bit AES key

# Global Variables
clients = []
conversation = []  # Stores all conversation messages
session_id = str(uuid.uuid4())  # Generate a unique session ID for this server instance
cout_lock = threading.Lock()
clients_lock = threading.Lock()
```

I will begin by explaining the server code and how our application performs all the encryption techniques and comes. The first thing is to import the necessary libraries that we are going to use their functions to perform all the functionalities of the application. The use of threading supports the concurrent sending and receiving of messages, with sockets for network interactions. Os gives input validation and file system interfaces, while signal and sys handle system-level operations such as signal handling and system controls. Base64 and hashlib would be in use for secure encoding and ensuring integrity, respectively; similarly, struct handles binary data.

It also imports, from cryptography.hazmat, cryptographic libraries of ciphers, and padding, thus enabling the use of state-of-the-art encryption algorithms such as AES and RSA for secure encryption and decryption of data. These libraries ensure that data is properly padded for encryption, which is a very important security layer. The hashes module provides several cryptographic hash functions, such as SHA-256, which are very helpful in keeping data integrity and creating digital signatures.

It uses PBKDF2HMAC to safely derive encryption keys from passwords. This adds a layer of security because it will not use the plain password. It serializes handling and parsing keys, such as receiving and using a server's public key for encryption. The default_backend ensures that all cryptographic operations are performed with a secure standard cryptography backend. Lastly, the DateTime functionality enables the addition of timestamps to messages or logging events; in this way, all network activity can be tracked and time-stamped easily for audit purposes.

Then we defined some important things for the generation of the AES key such as the password and the unique salt, which will used together to generate the AES key, as well as assign the key length, and declare global variables for the conversation that gets stored in and variable for the clients to get stored in. Lastly, the code generates a unique session ID for the server and creates locks to safely manage concurrent access to shared resources, such as console output and the list of connected clients, in a multithreaded environment.

```
34  # Generate RSA key pair for the server
35  private_key = rsa.generate_private_key(
36      public_exponent=65537,
37      key_size=2048,
38      backend=default_backend()
39  )
40  public_key = private_key.public_key()
```

```python
class Terminal:
    """Represents a connected client."""
    def __init__(self, id, name, socket):
        self.id = id
        self.name = name
        self.socket = socket
        self.public_key = None  # Will be set after receiving the client's public key

# Utility Functions
def color(code):
    return colors[code % NUM_COLORS]

def shared_print(message, end_line=True):
    with cout_lock:
        print(message, end="\n" if end_line else "", flush=True)
```

The first part of the code In the images above generates the RSA key pair for the server, both the server's private and public key, and then gets stored in private_key, and the corresponding public key is extracted and stored in public_key. Then we must create a terminal class, which will hold information about the connected client and store their information like their unique ID, name, socket connection, and their public key, this helps because we can have multiple clients, so this keeps track and stores their things in one place. The last function to define is color(code) returns the selected color from the colors list by the given code and ensures that the code is in the range of available colors. Shared_print(message), end_line=True/ prints messages thread-safe but offers the possibility to print messages without a new line at the end, too.

```python
def broadcast_message(message, sender_id):
    global conversation
    disconnected_clients = []

    # Add timestamp to the message
    timestamp = datetime.now().strftime("[%Y-%m-%d %H:%M:%S]")
    formatted_message = f"{timestamp} {message}"

    with clients_lock:
        for client in clients:
            if client.id != sender_id:
                try:
                    # Encrypt and send the message to the client
                    send_encrypted_message_to_client(client.socket, client.public_key, formatted_message.encode())
                except Exception as e:
                    shared_print(f"Error sending message to client {client.name}: {e}")
                    disconnected_clients.append(client)

    # Remove disconnected clients
    with clients_lock:
        for client in disconnected_clients:
            clients.remove(client)

    # Save the formatted message to the conversation log
    conversation.append(formatted_message)
```

The broadcast message function is responsible for sending the messages that other clients send to the chatroom in other clients' chatrooms. For example, if client 1 sends a message to the server that has two clients connected to it, the message that client 1 sent will be visible in the chatroom of client 2. In summary, this function broadcasts a message to all connected clients except the sender. This happens by first adding a timestamp to the message, and then it iterates through the list of clients and attempts to send the encrypted message to each one of them except the sender. If sending to a client fails (due to disconnection), the client is added to a list of disconnected clients. After sending the message, it removes any disconnected clients from the list and finally saves the messages in the conversation logs.

```
84  def send_encrypted_message_to_client(client_socket, client_public_key, message):
85      try:
86          # Encrypt the message using the client's public key
87          encrypted = client_public_key.encrypt(
88              message,
89              asym_padding.OAEP(
90                  mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
91                  algorithm=hashes.SHA256(),
92                  label=None
93              )
94          )
95          # Encode the encrypted message in Base64
96          encrypted_base64 = base64.b64encode(encrypted)
97          # Send the length of the message first
98          msg_length = struct.pack('>I', len(encrypted_base64))
99          client_socket.sendall(msg_length + encrypted_base64)
100     except Exception as e:
101         shared_print(f"Error sending encrypted message to client: {e}")
102         raise
```

This function is the same as the one on the client's side, it encrypts the message it sends to the client using the client's public key with RSA and OAEP padding. After the encryption process is finished, it encodes the message by bas64 encoding to be read for transmission, and lastly, it sends the length of the encrypted message followed by the actual encrypted message to the client to let the client first know the length of the message he is about to receive and then receive the actual encrypted message. This function does exactly as the one On the client side.

```
104 def end_connection(id):
105     with clients_lock:
106         for client in clients:
107             if client.id == id:
108                 client.socket.close()
109                 clients.remove(client)
110                 shared_print(f"Connection closed for {client.name}.")
111                 break
112
113 # AES Encryption/Decryption Utilities
114 def derive_aes_key(password, salt, length=32):
115     """Derive a valid AES key of specific length (default: 32 bytes)."""
116     kdf = PBKDF2HMAC(
117         algorithm=hashes.SHA256(),
118         length=length,
119         salt=salt,
```

```
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    return kdf.derive(password.encode())

def pad_data(data):
    padder = sym_padding.PKCS7(algorithms.AES.block_size).padder()
    return padder.update(data) + padder.finalize()

def unpad_data(data):
    unpadder = sym_padding.PKCS7(algorithms.AES.block_size).unpadder()
    return unpadder.update(data) + unpadder.finalize()
```

The first function In the images above is the end connection function, which is responsible for ending the connection to the specified client, this works by searching for the client with the specified ID in the “clients” list, once the client is found, it closes their socket and removes this client from the list. The last thing is that it prints a message indicating that the connection for the specified client has been closed. This function will be called later In the code when we close the connection for a client.

Then we have the three functions that are responsible for generating the AES key and adding padding to the data to ensure that the AES block size is a multiple of 128 bits. The first one uses PBKDF2 with HMAC (SHA-256) to derive a secure AES key from a password and a salt defined at the top of the code. The derived key length defaults to 32 bytes. Then we have 2 padding functions, one adds padding to the data, and the other one removes the padding added to the data during encryption to ensure it returns to its original form.

```
def encrypt_conversation(conversation, filename):
    try:
        key = derive_aes_key(ENCRYPTION_PASSWORD, ENCRYPTION_SALT)
        validate_key_and_iv(key)

        # Pad the conversation
        padded_data = pad_data("\n".join(conversation).encode())

        # Create AES cipher
        cipher = Cipher(algorithms.AES(key), modes.CBC(key[:16]), backend=default_backend())
        encryptor = cipher.encryptor()

        # Encrypt the conversation
        encrypted_data = encryptor.update(padded_data) + encryptor.finalize()

        # Encode the encrypted data to Base64
        encrypted_base64 = base64.b64encode(encrypted_data).decode('utf-8')

        # Write Base64-encoded encrypted data to file
        with open(filename, "w") as f:
            f.write(encrypted_base64)

        return True
    except Exception as e:
        shared_print(f"Error encrypting conversation: {e}")
        return False
```

```python
def decrypt_conversation(filename, output_filename):
    try:
        key = derive_aes_key(ENCRYPTION_PASSWORD, ENCRYPTION_SALT)
        validate_key_and_iv(key)

        # Read Base64-encoded encrypted data from file
        with open(filename, "r") as f:
            encrypted_base64 = f.read().strip()

        # Decode Base64 to get the encrypted data
        encrypted_data = base64.b64decode(encrypted_base64)

        # Create AES cipher
        cipher = Cipher(algorithms.AES(key), modes.CBC(key[:16]), backend=default_backend())
        decryptor = cipher.decryptor()

        # Decrypt the data
        padded_data = decryptor.update(encrypted_data) + decryptor.finalize()
        decrypted_data = unpad_data(padded_data).decode()

        # Write decrypted data to file
        with open(output_filename, "w") as f:
            f.write(decrypted_data)

        return True
    except Exception as e:
        shared_print(f"Error decrypting conversation: {e}")
        return False
```

The two functions in the images above are responsible for encrypting and decrypting the conversation that contains the chat between the clients and the interaction of the clients to the server, such as the client has joined and left. First, the encrypt_conversation function that first generates the AES key by calling the password and unique salt we defied earlier to put them in the function that generates the key and generates a key in 32-bit. The code after generating the key, begins encrypting by first padding the conversation to ensure it fits the AES clock size, then it creates an AES cipher using the key and a 16-byte initialization vector derived from the key. After getting everything ready, it encrypts the conversation data and encodes it in Base64, and lastly, it saves the Base64-encoded encrypted data into the provided file (filename) using write mode “w” to write the encoded conversation in the file.

The second function, which is the decrypt_conversation is the same exactly as the one above it, but the key difference is that it starts from bottom to top, whatever we finish in the encryption we will start with in the decryption. Derives the AES encryption key using the same password and salt then reads the Base64 encoded encrypted data from the file decodes the Base64 data for getting encrypted bytes creates an AES cipher using the derived key and the 16-byte IV then decrypts the encrypted data and removes the padding then saves the decrypted conversation to a new file (output_filename).

```python
def validate_key_and_iv(key, iv_length=16):
    """Validate AES key and IV length."""
    if len(key) not in [16, 24, 32]:
        raise ValueError(f"Invalid key size: {len(key)} bytes. Must be 16, 24, or 32 bytes.")
    if len(key[:iv_length]) != iv_length:
        raise ValueError(f"Invalid IV size: {len(key[:iv_length])} bytes. Must be {iv_length} bytes.")

def save_conversation():
    global conversation, session_id

    if not conversation:
        shared_print("No conversation to save.")
        return

    encrypted_filename = f"encrypted_conversation_{session_id}.txt"
    decrypted_filename = f"decrypted_conversation_{session_id}.txt"

    # Save encrypted conversation
    if encrypt_conversation(conversation, encrypted_filename):
        shared_print(f"Encrypted conversation saved as: {encrypted_filename}")

    # Save decrypted conversation
    try:
        with open(decrypted_filename, "w") as f:
            f.write("\n".join(conversation))
        shared_print(f"Decrypted conversation saved as: {decrypted_filename}")
    except Exception as e:
        shared_print(f"Error saving decrypted conversation: {e}")
```

The first function In the images above is responsible for ensuring and validating that the AES key and the initialization vector (IV) are the correct lengths. It ensures the key is one of 16, 24, or 32 bytes long, and that the IV (which is derived from the key) is exactly 16 bytes. If either condition fails, it raises an exception with a descriptive error message. The second function (save_conversation), is responsible for saving both the encrypted and decrypted conversation, it encrypts the conversation by calling the encrypt_conversation function and assigning its parameters including the conversation itself and the encrypted filename, which is saved by the session ID like this “encrypted_conversation_{session_id}.txt”. then the same thing happens in the decrypted version, it stores the conversation as plain text in a file named “decrypted_conversation_{session_id}.txt”.

```python
# Signal Handling with user prompt
def handle_ctrl_c(signal_received, frame):
    shared_print("\nCTRL+C pressed. Do you want to end the server?")
    shared_print("Type 'yes' to end the server or 'no' to return to the chat.")

    choice = input("Your choice: ").strip().lower()

    if choice == 'yes':
        shared_print("Shutting down the server...")
        save_conversation()  # Save the conversation before shutting down
        with clients_lock:
            for client in clients:
                client.socket.close()  # Close all client connections
            clients.clear()

        # Ensure a clean exit
        shared_print("Server shut down successfully.")
        sys.exit(0)  # Exit the program after shutting down
    else:
        shared_print("Returning to the chat...")
        return
```

```python
def decrypt_message(encrypted_message_base64):
    try:
        # Decode the Base64-encoded data
        encrypted_message = base64.b64decode(encrypted_message_base64)

        decrypted = private_key.decrypt(
            encrypted_message,
            asym_padding.OAEP(
                mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return decrypted.decode()
    except Exception as e:
        shared_print(f"Error decrypting message: {e}")
        return None
```

The handle_ctrl_c function as illustrated in the first picture above will handle the CTRL+C signal, which usually oversees process interruptions and stops, such that it will give the user an option to exit the server neatly or go back to the chat without exiting it. In case of detecting a CTRL+C, it will ask if the user wants to shut it down or keep going. That is, on termination, he calls a function save_conversation() to save the conversation, closes all the client connections, clears the list containing clients, and then sys.exit(0) kicks the server out. In the case of returning to the chat, the server keeps running without interference.

The second function is decrypting a message coming from the client that has been encrypted using RSA by the server public ley, the first thing that will happen is taking the encrypted message in Base64 format as input, and then decode the Base64 data to get the encrypted byte sequence and decrypts the message using the RSA private key and OAEP padding (with SHA-256) and returning the decrypted message as a string.

```python
def recvall(sock, n):
    """Helper function to receive n bytes or return None if EOF is hit."""
    data = b''
    while len(data) < n:
        packet = sock.recv(n - len(data))
        if not packet:
            return None
        data += packet
    return data

def receive_client_public_key(client_socket):
    try:
        # Receive the client's public key
        pem_public_key = client_socket.recv(2048)
        client_public_key = serialization.load_pem_public_key(pem_public_key)
        return client_public_key
    except Exception as e:
        shared_print(f"Failed to receive client public key: {e}")
        return None
```

The recvall function is like a helper that keeps collecting data from a connection until it gets exactly the amount it expects. If the connection drops or stops sending data, it gives up and returns None. The receive_client_public_key function is there to grab the client's public key from

the connection, decode it into a usable format, and return it. If something goes wrong during this process, it politely reports the issue without crashing everything.

Then we have the handle_client function, which manages all interactions between the server and a client, ensuring secure communication, user authentication, and real-time messaging.

```python
# Client Handler
def handle_client(client_socket, id, terminal):
    rsa_successful = True  # Initialize the RSA success flag to True
    try:
        # Send the server's public key to the client
        pem_public_key = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        client_socket.send(pem_public_key)

        # Receive the client's public key
        client_public_key = receive_client_public_key(client_socket)
        if client_public_key is None:
            shared_print(f"Failed to receive client public key from client {id}.")
            client_socket.close()
            return

        # Store the client's public key in the terminal object
        terminal.public_key = client_public_key

        # Receive the action indicator
        raw_msglen = recvall(client_socket, 4)
        if not raw_msglen:
            shared_print(f"Failed to receive action indicator from client {id}.")
            client_socket.close()
            return
        msglen = struct.unpack('>I', raw_msglen)[0]
        encrypted_action = recvall(client_socket, msglen)
        action = decrypt_message(encrypted_action)
        if action not in ['LOGIN', 'SIGNUP']:
            shared_print(f"Invalid action from client {id}: {action}")
            # Encrypt and send failure response
            send_encrypted_message_to_client(client_socket, terminal.public_key, b'0')
            return

        # Receive credentials from the client
        raw_msglen = recvall(client_socket, 4)
        if not raw_msglen:
            shared_print(f"Failed to receive credentials from client {id}.")
            client_socket.close()
            return
        msglen = struct.unpack('>I', raw_msglen)[0]
        encrypted_credentials = recvall(client_socket, msglen)
        credentials = decrypt_message(encrypted_credentials)
        if not credentials:
            shared_print(f"Received empty credentials from client {id}.")
            # Encrypt and send failure response
            send_encrypted_message_to_client(client_socket, terminal.public_key, b'0')
            return

        # Parse credentials
        try:
            name, password = credentials.split()
        except ValueError:
            shared_print(f"Invalid credentials format from client {id}.")
            # Encrypt and send failure response
            send_encrypted_message_to_client(client_socket, terminal.public_key, b'0')
            return

        shared_print(f"Received name: {name}")
```

```
        if action == 'LOGIN':
            # Authenticate user
            authenticated = authenticate_user(name, password)
            if not authenticated:
                shared_print(f"Authentication failed for {name}.")
                # Encrypt and send failure response
                send_encrypted_message_to_client(client_socket, terminal.public_key, b'0')
                return
            else:
                # Encrypt and send success response
                send_encrypted_message_to_client(client_socket, terminal.public_key, b'1')
        elif action == 'SIGNUP':
            # Attempt to sign up the user
            if not signup_user(name, password):
                # Encrypt and send failure response
                send_encrypted_message_to_client(client_socket, terminal.public_key, b'0')
                return
            else:
                # Encrypt and send success response
                send_encrypted_message_to_client(client_socket, terminal.public_key, b'1')

        # Update the client's name in the Terminal object
        terminal.name = name  # Update the name to the authenticated username

        # Announce user join
        broadcast_message(f"{name} has joined", id)
        shared_print(color(id) + f"{name} has joined" + def_col)

        while True:
            # Read message length (4 bytes)
            raw_msglen = recvall(client_socket, 4)
            if not raw_msglen:
                break
            msglen = struct.unpack('>I', raw_msglen)[0]
            # Read the message data
            encrypted_message_base64 = recvall(client_socket, msglen)
            if not encrypted_message_base64:
                break

            # Decrypt the message
            message = decrypt_message(encrypted_message_base64)
            if not message:
                rsa_successful = False  # Set the flag to False if decryption fails
                continue  # Skip if decryption failed

            if message == "#exit":
                broadcast_message(f"{name} has left", id)
                shared_print(color(id) + f"{name} has left" + def_col)
                break  # Exit the loop to close the connection

            if message == 'GET_USERS':
                with clients_lock:
                    user_list = ', '.join(client.name for client in clients if client.name != "Anonymous")
                user_list = "People entered the chatroom: " + user_list
                try:
                    # Encrypt and send the user list to the client
                    send_encrypted_message_to_client(client_socket, terminal.public_key, user_list.encode())
                except Exception as e:
                    shared_print(f"Error sending user list to client {name}: {e}")
                continue

            broadcast_message(f"{name}: {message}", id)
            shared_print(color(id) + f"{name}: " + def_col + message)

```

```
    except ConnectionResetError:
        shared_print(f"Connection was reset by client {name}.")
    except Exception as e:
        shared_print(f"An error occurred while handling client {name}: {e}")
    finally:
        # Ensure the connection is closed
        end_connection(id)
        # Check the RSA success flag and print the message
        if rsa_successful:
            shared_print(f"This conversation with {name} was successfully encrypted using RSA Encryption.")

```

The explanation of the handle client function and how it functions together to handle the connection between the server and the client will be explained in the form of steps:-

1) Start Communication with Public Key Exchange

- It starts with the server sending over its RSA public key to the client enabling a secure connection between both the client and the server
- It then waits for the client to return the public key.
- It then makes sure that, upon reception, the server stores the client's public key to ensure that all further messages will be encrypted and decrypted securely.

2) Understand the client's actions.

- It sends from the client a secret encrypted message to the server with any one of the two messages, saying: LOGIN or SIGNUP.
- The server decrypts this message to figure out the client's intention.
- It would, in turn, return an error message if anything goes wrong and it doesn't understand the message and then gracefully close the connection.

3) Processing User Credentials

The client safely transmits its username and password across (all encrypted, of course).

Once the server has decrypted the credentials, the next action is performed based on what the client does.

LOGIN:

- It checks with the server whether the username and password match its records.
- In case everything checks out, it greets the client with a success message; otherwise, it will provide information to the client about what went wrong.

SIGNUP:

- The server will, after receiving all the above-mentioned information, create an account with the provided details. If successful, it gives a success confirmation message, or it lets the client know if something is wrong.

4) Adding users to the chatroom

- Once the client is authenticated, the server assigns them a unique name, announcing their arrival to all the clients currently present in the chatroom.
- Others would then see, for example, "John has joined," which will notify the other clients that there is a user that has entered the chat.

5) Keep the conversation going by handling incoming messages

Now the server enters a loop to handle incoming messages from the client:

Regular Chat Messages:

- The client sends their messages securely encrypted, of course.
- These messages are decrypted by the server and relayed to other participants of the chat room.

Special Commands:

- The client, after entering #exit, broadcasts that he has left, and the socket communication is closed.
- If the client inputs GET_USERS, then the server compiles a list of all the active participants and sends the compiled list securely back.

6) Manage disconnections and errors

- It's left to the server to clean up in case the client goes away unexpectedly, or an error occurs:-
  - This will remove the client from the list of currently connected users.
  - This will notify every member in the chatroom that the client has disconnected.

7) Ensure everything stays secure

- During the session, the server also verifies RSA encryption to ensure that all affected interactions are safe.
- Finally, after the completion of the chat, it logs the session details to confirm that everything went well with encryption and that the chat was fully encrypted and secure.

```python
# Authentication Functions
def authenticate_user(username, password):
    try:
        with open("user_credentials.txt", "r") as file:
            for line in file:
                if not line.strip():
                    continue
                user, stored_hash = line.strip().split()
                if user == username and verify_sha256(password, stored_hash):
                    return True
    except FileNotFoundError:
        shared_print("Error: User credentials file not found.")
    return False

def verify_sha256(password, stored_hash):
    """Verify if the SHA-256 hash of the password matches the stored hash."""
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    return hashed_password == stored_hash

def signup_user(username, password):
    try:
        # Check if username already exists
        if os.path.exists("user_credentials.txt"):
            with open("user_credentials.txt", "r") as file:
                for line in file:
                    if not line.strip():
                        continue
                    existing_user = line.strip().split()[0]
                    if username == existing_user:
                        shared_print(f"Signup failed for {username}: Username already exists.")
                        return False

        # Hash the password and store credentials
        hashed_password = hashlib.sha256(password.encode()).hexdigest()  # SHA-256 hash
        with open("user_credentials.txt", "a") as file:
            file.write(f"{username} {hashed_password}\n")
            shared_print(f"User {username} signed up successfully.")
            return True

    except IOError as e:
        shared_print(f"Error: Unable to open file: {e}")
        return False

```

These are two authentication functions that handle the user login process as well as the registration process securely. The first function first checks if the user exists and if their password matches one that is stored in the file (user_credentials.txt) after it is being hashed to then be compared with other hashes from the file, and if the file is missing credentials don't match, it returns false.

The next function is a helper function that hashes the given password during the login process to hashed using SHA-256 to check if it matches the stored hash. It's the backbone of password verification for secure comparison. The last function is the signup function, which is responsible for allowing the addition of new users. It checks user_credentials.txt to see whether the given username does not already exist, hashes the given password via SHA-256, adds the credentials of a new user in the credentials file, and acknowledges success. If any errors occur such as file issues, it handles them gracefully.

```python
def main():
    global session_id

    # Print the welcome message
    print(colors[5] + "\n\t  ====== Welcome to the chat-room ======   " + def_col)

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind(('0.0.0.0', 10000))
    server_socket.listen(8)

    signal.signal(signal.SIGINT, handle_ctrl_c)

    print(f"Server is running. Session ID: {session_id}")

    while True:
        try:
            client_socket, client_addr = server_socket.accept()
            id = len(clients)
            client_name = f"Client_{id}"

            shared_print(f"Connection accepted from {client_addr}.")

            terminal = Terminal(id, client_name, client_socket)
            with clients_lock:
                clients.append(terminal)

            # Pass the terminal object to handle_client
            threading.Thread(target=handle_client, args=(client_socket, id, terminal), daemon=True).start()
        except Exception as e:
            shared_print(f"Error accepting connection: {e}")
            break

if __name__ == "__main__":
    main()
```

The main() function sets up and runs the chat server. Here is a simple explanation of its workflow:

- It will display a greeting message along with creating a server socket to listen on port 10000 for any incoming connections.
- Signal Handling: The script binds the signal of CTRL+C to the function handle_ctrl_c for a cleanup shutdown of this server upon interrupt.
- Listen For Clients: The server is put into a loop where it accepts clients connecting. Upon each new connection, it performs an action of assigning a unique ID, and a default name, logs the connection, and then joins in the list of clients.
- Client Handling: A new thread for each client is created; this thread sends the socket of the client, the client's ID, and the terminal object to the handle_client function for further communication.

Security concerns during the software development lifecycle

Throughout the SDLC(software development lifecycle), several major security concerns required our serious attention and strategic mitigation. One of the major attacks that we tried to keep at bay included brute forcing and dictionary attacks. To be sure, strong encryption techniques and several security protocols were employed to reinforce our barriers against unauthorized access. As this application deals with the credentials of the users, like passwords and other details, due care had to be taken so that these remain safe. We have used strong encryption for this information so that in the case of leakage, data will not be readable to any third-party unauthorized person.

Another major concern of our security strategy was safely sending information and messages from various clients to our server. We did this by encrypting the messages that are usually sent interchangeably between the different clients and the server by intercepting them from any other party. This also includes protection of login information or signup credentials whenever there is a user authentication, thus enhancing security for the user by a notch. Our application can also save data to files, other than handling live data transactions. For the protection of the information stored in these files, we have applied advanced encryption techniques(AES) so that whatever interactions get archived remains confidential. To this end, encryption reduces the risk of the attacker having access to those files and interpreting old conversations.

These are very multi-factorial, and we tried to address them through three quite robust encryption methodologies: RSA, more formally Rivest-Shamir-Adleman; Advanced Encryption Standard (AES); and by using a strong hashing algorithm, identified as SHA-256. A holistic approach was waged in this aspect to address all possible shortcomings in encrypting user data and communications effectively. We then could safely keep our fingers crossed that no attacker would foul our application or get unauthorized access to sensitive messages and user credentials.