UNIVERSITÀ DI PISA

Cybersecurity

Foundations of Cybersecurity

COURSE PROJECT

Secure implementation of Bulletin Board System

STUDENTS:
Lepore Nicola
Copelli Francesco

Academic Year 2023/2024

Abstract

This project has been developed in C++ as a workgroup during the classes of FOUNDATIONS OF CYBERSECURITY. It details the design and implementation of a secure Bulletin Board System (BBS) adhering to best practices in Applied Cryptography. The BBS allows users to register, log in, post messages, and retrieve existing messages. Security is paramount, with features like secure communication channels, password protection, and perfect forward secrecy (PFS) implemented using the OPENSSL library (excluding the TLS API).

The system utilizes a centralized server architecture with user authentication through username and passwords. Messages are identified uniquely and contains title, author, and body fields. Users can list recent messages, retrieve specific messages by ID, and add new messages (upon successful login).

The document outlines the registration and login phases, emphasizing secure user authentication and challenge-response mechanisms.

Contents

1 | Specifications

The application developed implements a Bulletin Board System. It allows the users, registered through the server, to add and read the tickets posted on the online bulletin board.

The application allows the users, already registered on the server, to log-in using *public key authentication*. Server and client authenticates each other verifying that their public keys are valid.

After the log-in, a user can get a list of tickets on the bulletin board and download one by one with the specific command.

The communication between the client and the server and during the session(s) are secured using the OpenSSL library. In particular, when a client connects to the server, a *key negotiation algorithm*, based on the *Diffie-Hellman key exchange algorithm*, is executed. Each message exchanged during this negotiation is authenticated using *public key authentication*. When the two parties have agreed on the session key, they encrypt each subsequent message using the Shared Secret calculated based on the key negotiation.

The application is developed using the C++ language and offer a command-line interface (CLI) to the user. The user makes actions in the application when asked to perform something.

For details on how to run and use the application, see Appendix A.1 "Compile and run the application".

2 | Application design

The application is divided into three components:

Client The client application that is used by the user to log-in with the server and make the request to it. The code is available in the `client` folder;

Common A library that contains modules that are used both by the server and the client. This includes the protocol implementation used to exchange commands between the parts, the Message composition and the cryptographic algorithm used to encrypt the communication and for the hashing. The code is available in the `common` folder;

Server The server application which handles the communication with the sockets representing the registered users allowing them to log-in and to request for note(s) on the Bulletin Board System. The code is available in the `server` folder.

Each components is divided into a number of modules, each module provides functions for a specific functionality of the application and it's compound by *CPP source file*.

Figure 2.1 and Figure 2.2 shows all the modules of the developed application along with their connections with other modules.
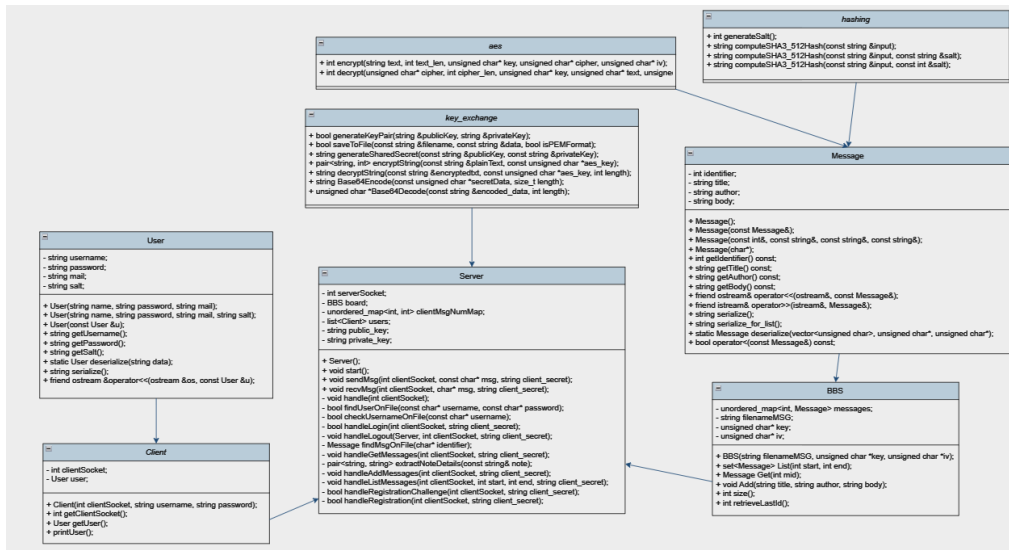
Figure 2.1 (Server package) — class diagram

aes
- + int encrypt(string text, int text_len, unsigned char* key, unsigned char* cipher, unsigned char* iv);
- + int decrypt(unsigned char* cipher, int cipher_len, unsigned char* key, unsigned char* text, unsigne...

hashing
- + int generateSalt();
- + string computeSHA3_512Hash(const string &input);
- + string computeSHA3_512Hash(const string &input, const string &salt);
- + string computeSHA3_512Hash(const string &input, const int &salt);

key_exchange
- + bool generateKeyPair(string &publicKey, string &privateKey);
- + bool saveToFile(const string &filename, const string &data, bool isPEMFormat);
- + string generateSharedSecret(const string &publicKey, const string &privateKey);
- + pair<string, int> encryptString(const string &plainText, const unsigned char *aes_key);
- + string decryptString(const string &encryptedtxt, const unsigned char *aes_key, int length);
- + string Base64Encode(const unsigned char *secretData, size_t length);
- + unsigned char *Base64Decode(const string &encoded_data, int length);

Message
- - int identifier;
- - string title;
- - string author;
- - string body;
- + Message();
- + Message(const Message&);
- + Message(const int&, const string&, const string&, const string&);
- + Message(char*);
- + int getIdentifier() const;
- + string getTitle() const;
- + string getAuthor() const;
- + string getBody() const;
- + friend ostream& operator<<(ostream&, const Message&);
- + friend istream& operator>>(istream&, Message&);
- + string serialize();
- + string serialize_for_list();
- + static Message deserialize(vector<unsigned char>, unsigned char*, unsigned char*);
- + bool operator<(const Message&) const;

User
- - string username;
- - string password;
- - string mail;
- - string salt;
- + User(string name, string password, string mail);
- + User(string name, string password, string mail, string salt);
- + User(const User &u);
- + string getUsername();
- + string getPassword();
- + string getSalt();
- + static User deserialize(string data);
- + string serialize();
- + friend ostream &operator<<(ostream &os, const User &u);

Server
- - int serverSocket;
- - BBS board;
- - unordered_map<int, int> clientMsgNumMap;
- - list<Client> users;
- - string public_key;
- - string private_key;
- + Server();
- + void start();
- + void sendMsg(int clientSocket, const char* msg, string client_secret);
- + void recvMsg(int clientSocket, char* msg, string client_secret);
- - void handle(int clientSocket);
- - bool findUserOnFile(const char* username, const char* password);
- - bool checkUsernameOnFile(const char* username);
- - bool handleLogin(int clientSocket, string client_secret);
- - void handleLogout(Server, int clientSocket, string client_secret);
- - Message findMsgOnFile(char* identifier);
- - void handleGetMessages(int clientSocket, string client_secret);
- - pair<string, string> extractNoteDetails(const string& note);
- - void handleAddMessages(int clientSocket, string client_secret);
- - void handleListMessages(int clientSocket, int start, int end, string client_secret);
- - bool handleRegistrationChallenge(int clientSocket, string client_secret);
- - bool handleRegistration(int clientSocket, string client_secret);

Client
- - int clientSocket;
- - User user;
- + Client(int clientSocket, string username, string password);
- + int getClientSocket();
- + User getUser();
- + printUser();

BBS
- - unordered_map<int, Message> messages;
- - string filenameMSG;
- - unsigned char* key;
- - unsigned char* iv;
- + BBS(string filenameMSG, unsigned char *key, unsigned char *iv);
- + set<Message> List(int start, int end);
- + Message Get(int mid);
- + void Add(string title, string author, string body);
- + int size();
- + int retrieveLastId();

Figure 2.1: Server package

Figure 2.2 (Client package) — class diagram

key_exchange
- + bool generateKeyPair(string &publicKey, string &privateKey);
- + bool saveToFile(const string &filename, const string &data, bool isPEMFormat);
- + string generateSharedSecret(const string &publicKey, const string &privateKey);
- + pair<string, int> encryptString(const string &plainText, const unsigned char *aes_key);
- + string decryptString(const string &encryptedtxt, const unsigned char *aes_key, int length);
- + string Base64Encode(const unsigned char *secretData, size_t length);
- + unsigned char *Base64Decode(const string &encoded_data, int length);

Message
- - int identifier;
- - string title;
- - string author;
- - string body;
- + Message();
- + Message(const Message&);
- + Message(const int&, const string&, const string&, const string&);
- + Message(char*);
- + int getIdentifier() const;
- + string getTitle() const;
- + string getAuthor() const;
- + string getBody() const;
- + friend ostream& operator<<(ostream&, const Message&);
- + friend istream& operator>>(istream&, Message&);
- + string serialize();
- + string serialize_for_list();
- + static Message deserialize(vector<unsigned char>, unsigned char*, unsigned char*);
- + bool operator<(const Message&) const;

hashing
- + int generateSalt();
- + string computeSHA3_512Hash(const string &input);
- + string computeSHA3_512Hash(const string &input, const string &salt);
- + string computeSHA3_512Hash(const string &input, const int &salt);

ClientHandler
- - int clientSocket;
- - bool isLogged;
- - string username;
- - string public_key;
- - string private_key;
- - string server_secret;
- - int msg_num;
- - void handle();
- - void sendMsg(int clientSocket, const char* msg);
- - void recvMsg(int clientSocket, char* msg, bool decrypt = true);
- - bool handleRegistration();
- - bool handleRegistrationChallenge(int clientSocket);
- - bool handleLogin(int clientSocket);
- - void handleAdd(int clientSocket);
- - void handleList(int clientSocket);
- - void handleGet(int clientSocket);
- - void handleLogout();
- - bool handleExit(int clientSocket);
- - bool handleQuit(int clientSocket, bool isLogged);
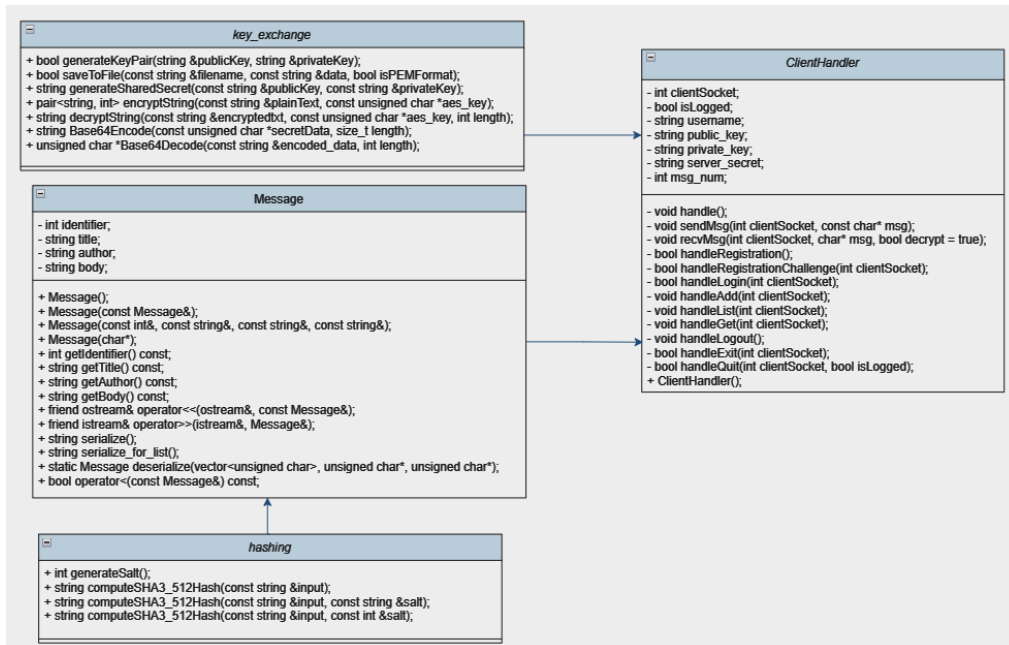- + ClientHandler();

Figure 2.2: Client package

4

2.1 Modules description

This section describe the modules that are part of the application.

2.1.1 Client module

This section describes the modules of the common library.
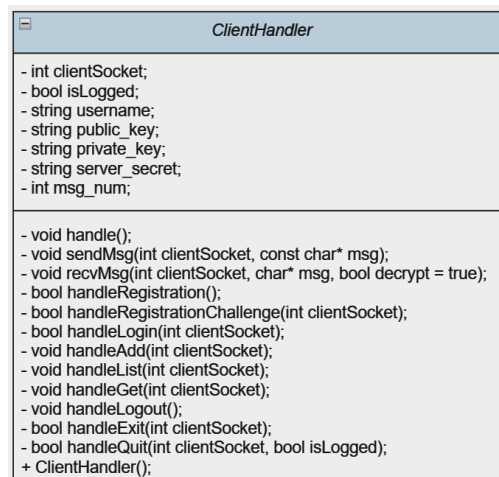
ClientHandler.cpp

The Figure 2.3 shows the method defined in the file. The **ClientHandler** class represents the client-side functionalities for the bulletin board system. The class is composed by the following attributes:

- **clientSocket**: An integer representing the socket used for communication with the server.

- **isLogged**: A boolean flag indicating if the client is logged in.

- **username**: A string storing the username of the client.

- **public_key**, **private_key**: Strings containing the client's public and private keys used for cryptography.

- **server_secret**: A string holding the shared secret generated during the key exchange with the server.

- **msg_num**: An integer used for message numbering for client-side tracking.

The class contains the following methods:

- Public methods:

  - **ClientHandler**: The constructor that establishes a connection with the server, generates key pairs, and performs initial handshakes.

- Private methods:

  - **handle**: The main loop for handling user interaction and communication with the server.
  - **sendMsg**: Sends a message to the server with encryption if the server secret is available.
  - **recvMsg**: Receives a message from the server with decryption if required.
  - **handleRegistration**: Handles the user registration process on the server.

– **handleRegistrationChallenge**: Handles the registration challenge sent by the server.

– **handleLogin**: Handles the user login process by sending credentials to the server.

– **handleAdd**: Handles adding a new message to the bulletin board on the server.

– **handleList**: Handles listing messages available on the bulletin board.

– **handleGet**: Handles retrieving a specific message from the bulletin board.

– **handleLogout**: Handles logging out the user from the server.

– **handleExit**: Handles closing the client socket and exiting the program.

– **handleQuit**: Handles quitting the program, performing logout if the user is logged in.

```
┌──────────────────────────────────────────────────────┐
│ ⊟                  ClientHandler                        │
├──────────────────────────────────────────────────────┤
│ - int clientSocket;                                     │
│ - bool isLogged;                                        │
│ - string username;                                      │
│ - string public_key;                                    │
│ - string private_key;                                   │
│ - string server_secret;                                 │
│ - int msg_num;                                          │
├──────────────────────────────────────────────────────┤
│ - void handle();                                        │
│ - void sendMsg(int clientSocket, const char* msg);      │
│ - void recvMsg(int clientSocket, char* msg, bool decrypt = true); │
│ - bool handleRegistration();                            │
│ - bool handleRegistrationChallenge(int clientSocket);   │
│ - bool handleLogin(int clientSocket);                   │
│ - void handleAdd(int clientSocket);                     │
│ - void handleList(int clientSocket);                    │
│ - void handleGet(int clientSocket);                     │
│ - void handleLogout();                                  │
│ - bool handleExit(int clientSocket);                    │
│ - bool handleQuit(int clientSocket, bool isLogged);     │
│ + ClientHandler();                                      │
└──────────────────────────────────────────────────────┘
```

Figure 2.3: Client.cpp module

2.1.2 Common

This package provides support for both client and server about common files and classes.

Crypto/aes.cpp

It uses functions for encryption and decryption using the OPENSSL library, providing the following functions:

- `encrypt(string, int, unsigned char*, unsigned char*, unsigned char*)`: Encrypts a string using AES-256 CBC mode with a provided key and initialization vector (IV).

- `decrypt(unsigned char*, int, unsigned char*, unsigned char*, unsigned char*)`: Decrypts a ciphertext using AES-256 CBC mode with a provided key and IV.

Refer to Figure 2.4

aes
+ int encrypt(string text, int text_len, unsigned char* key, unsigned char* cipher, unsigned char* iv); + int decrypt(unsigned char* cipher, int cipher_len, unsigned char* key, unsigned char* text, unsigne
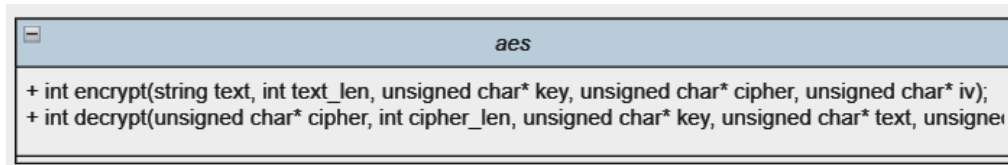
Figure 2.4: aes.cpp module

Crypto/Hashing

This file uses functions for various SHA3-512 hashing functionalities. The implemented methods, shown in Figure 2.5 are:

- **generateSalt()**: Generates a random integer to be used as a salt value.

- **computeSHA3_512Hash(const string&)**: Computes the SHA3-512 hash of a given string.

- **computeSHA3_512Hash(const string&, const string&)**: Computes the SHA3-512 hash of a string concatenated with a provided salt string.

- **computeSHA3_512Hash(const string&, const int&)**: Computes the SHA3-512 hash of a string concatenated with a provided salt integer (converted to string first).

hashing
+ int generateSalt(); + string computeSHA3_512Hash(const string &input); + string computeSHA3_512Hash(const string &input, const string &salt); + string computeSHA3_512Hash(const string &input, const int &salt);
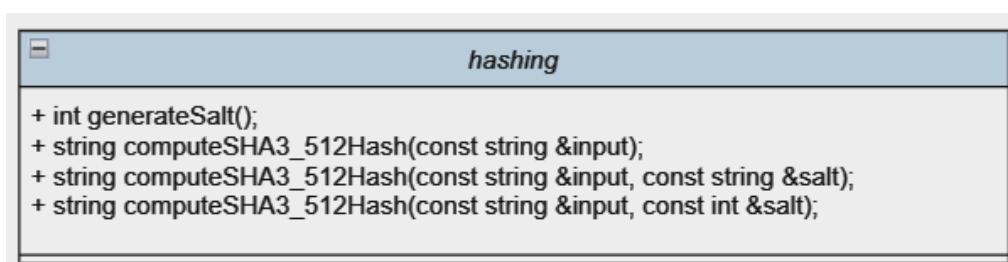
Figure 2.5: Hashing.cpp class diagram

7

Crypto/KeyExchange

This, defines several functions for key generation, shared secret generation, encryption, and decryption, providing the following:

- **generateKeyPair(string&, string&)**: Generates a public-private key pair.

- **saveToFile(const string&, const string&, bool)**: Saves a key to a file in PEM format.

- **generateSharedSecret(const string&, const string&)**: Generates a shared secret key using Diffie-Hellman.

- **encryptString(const string&, const unsigned char*)**: Encrypts a string using AES-256 CBC with a provided key, and returns the encrypted text and its length. Before encrypting the given string, PKCS7 padding will be applied to ensure the string's length is a multiple of 16 bytes. PKCS7 padding works by adding a series of bytes to the end of the plaintext, where each added byte is the same and represents the number of bytes added.

- **decryptString(const string&, const unsigned char, int*)**: Decrypts a string using AES-256 CBC with a provided key and the length of the encrypted text. Before performing the decryption, the padding added (of type `EVP_PADDING_PKCS7`) is removed from the string, allowing the algorithm to work correctly.

- **handle_errors()**: Prints error messages from OpenSSL and exits the program.

- **Base64Encode(const unsigned char, size_t*)**: Encodes an input unsigned char of size size_t into Base64 string

- **Base64Decode(const string&, int)**: Decodes an input string of fixed a string

See diagram in Figure 2.6


```
┌─────────────────────────────────────────────────────────────────────┐
│ ⊟                          key_exchange                              │
├─────────────────────────────────────────────────────────────────────┤
│ + bool generateKeyPair(string &publicKey, string &privateKey);      │
│ + bool saveToFile(const string &filename, const string &data, bool isPEMFormat); │
│ + string generateSharedSecret(const string &publicKey, const string &privateKey); │
│ + pair<string, int> encryptString(const string &plainText, const unsigned char *aes_key); │
│ + string decryptString(const string &encryptedtxt, const unsigned char *aes_key, int length); │
│ + string Base64Encode(const unsigned char *secretData, size_t length); │
│ + unsigned char *Base64Decode(const string &encoded_data, int length); │
└─────────────────────────────────────────────────────────────────────┘
```

Figure 2.6: Key Exchange modules

Message

This class provides a struct of the Message that has to be inserted in the Board Bulleting System, see Figure 2.7. It contains:

- Private attributes:

  - **Identifier**: Unique identifier for the message (int).
  - **Title**: Title of the message (string).
  - **Author**: Author of the message (string).
  - **Body**: Body of the message (string).

- Public methods:

  - **Message()**: Default constructor (initializes attributes to default values).
  - **Message(const Message&)**: Copy constructor to create a copy of an existing message.
  - **Message(const int&, const string&, const string&, const string&)**: Constructor to create a message with provided arguments.
  - **Message(char*)**: Constructor to deserialize a message object from a formatted string.
  - **getIdentifier() const**: Returns the message identifier.
  - **getTitle() const**: Returns the message title.
  - **getAuthor() const**: Returns the message author.
  - **getBody() const**: Returns the message body.
  - **friend ostream& operator«(ostream&, const Message&)**: Allows printing a message object using « operator.
  - **friend istream& operator»(istream&, Message&)**: Allows reading message data using » operator (interactive input).
  - **serialize()**: Serializes the message object into a formatted string.
  - **serialize_for_list()**: Serializes the message object for a list representation (it only includes identifier, title, and author).
  - **static Message deserialize(vector<unsigned char>, unsigned char*, unsigned char*)**: Deserializes a message object from encrypted data using a key and initialization vector.
  - **bool operator<(const Message&) const**: Allows sorting messages based on Message identifier.
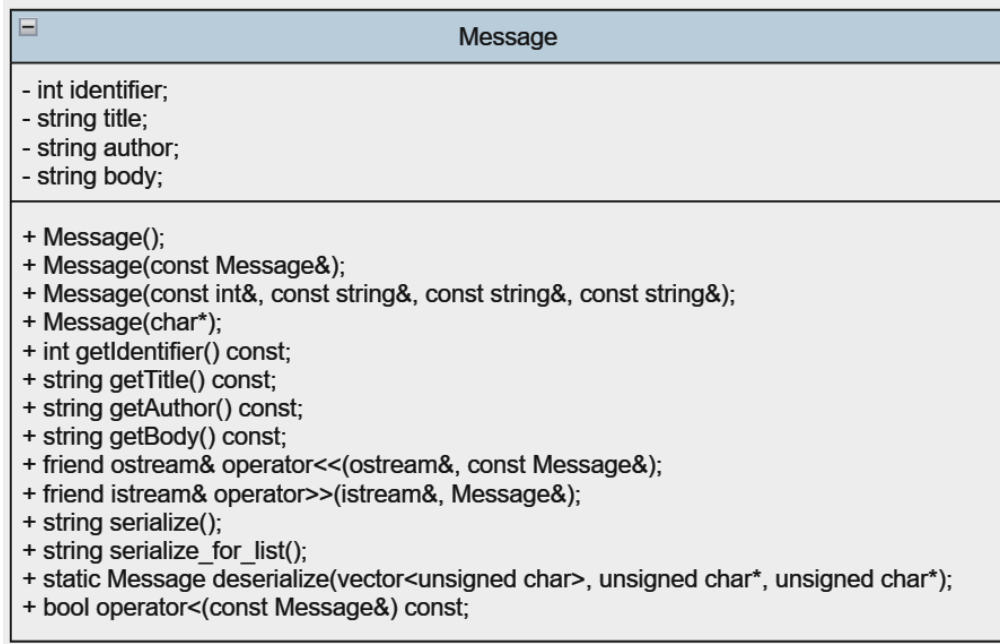
Figure 2.7: Message class diagram

2.1.3 Server

User.cpp

This class represent the author of a message in the Bulletin Board System, as shown in Figure 2.8. It's composed by the following:

- Attributes:

  - **username**: Username of the user (string).

  - **password**: Hashed password of the user (string).

  - **mail**: Email address of the user (string).

  - **salt**: Salt used for password hashing (string).

- Public methods:

  - **User(string, string, string)**: Creates a new user with the provided username, password, and email.

  - **User(string, string, string, string)**: Creates a new user with the provided username, password, email, and salt.

  - **User(const User&)**: Copy constructor to create a copy of an existing user.

  - **getUsername() const**: Returns the username.

  - **getPassword() const**: Returns the hashed password.

  - **getSalt() const**: Returns the salt used for password hashing.

– **static User deserialize(string)**: Deserializes a user object from a formatted string.

– **serialize() const**: Serializes the user object into a formatted string.

– **friend ostream& operator«(ostream&, const User&)**: Allows printing a User object using « operator.
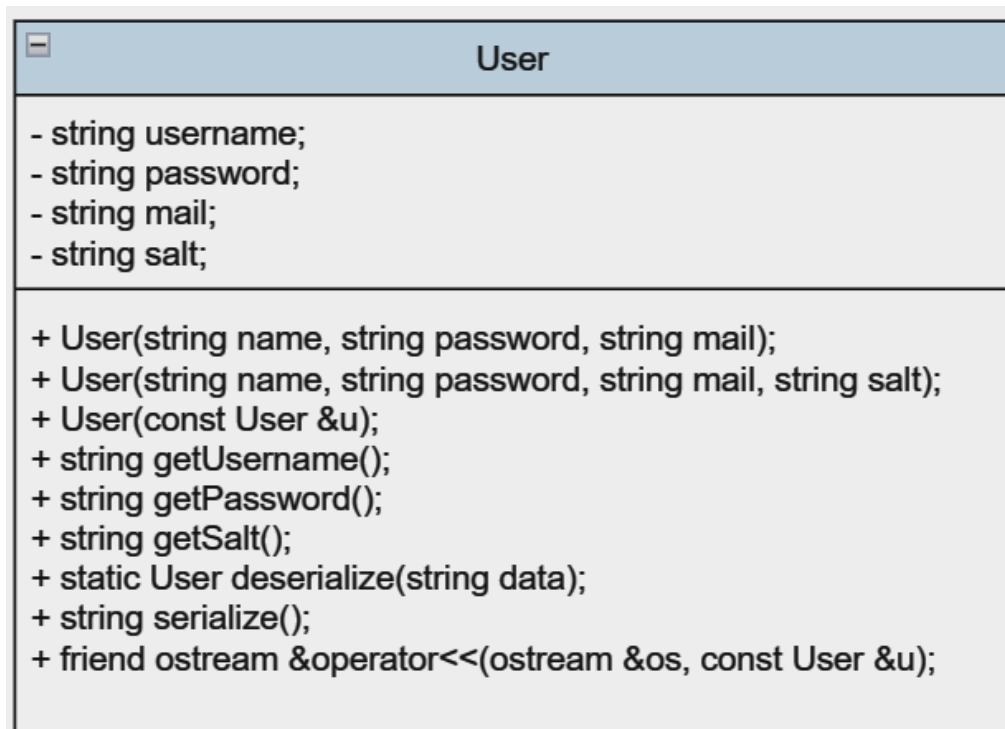
Figure 2.8: User.cpp class diagram

Client.cpp

This class represent the user that is trying to connect to the server to perform operations, see Figure 2.9. It's composed by:

- Attributes:

    – **clientSocket**: An integer representing the socket used for communication with the server.

    – **user**: A User object containing the client's username, password, email and salt.

- Public Methods:

    – **Client(int, string, string)**: The constructor that likely initializes the clientSocket and creates a User object with the provided information.

– **getClientSocket()**: Returns the value of clientSocket.

– **getUser()**: Returns the User object associated with the client.

– **printUser()**: Prints the user information stored in the User object.


```
┌─────────────────────────────────────────────────────────┐
│ ⊟                        Client                          │
├─────────────────────────────────────────────────────────┤
│ - int clientSocket;                                      │
│ - User user;                                             │
│                                                          │
├─────────────────────────────────────────────────────────┤
│ + Client(int clientSocket, string username, string password); │
│ + int getClientSocket();                                 │
│ + User getUser();                                        │
│ + printUser();                                           │
└─────────────────────────────────────────────────────────┘
```
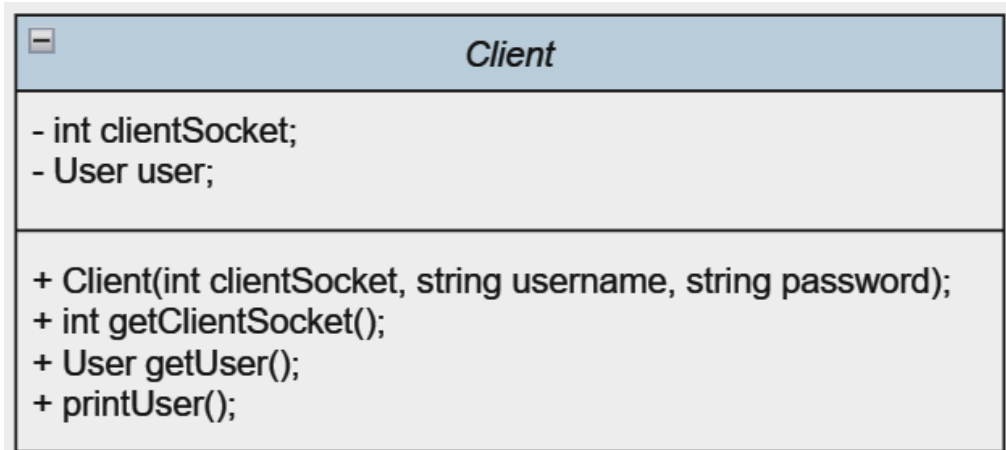
Figure 2.9: Client.cpp class diagram

BBS.cpp

This classes represents Bulletin Board System, see Figure 2.10. It's composed by:

- Attributes:

  – **messages**: An unordered map that stores messages by their unique identifiers (integers) as keys and the corresponding Message objects as values.

  – **filenameMSG**: A string containing the filename used to store and load messages.

  – **key**: A pointer to an unsigned char array representing the encryption key.

  – **iv**: A pointer to an unsigned char array representing the initialization vector for encryption.

- Public Methods:

  – **BBS(string, unsigned char*, unsigned char*)**: The constructor that initializes the filenameMSG, key, and iv attributes, and loads messages from the specified file into the messages map. In this method, the program is loading the Board Message from the file. The content of the message is encrypted using the AES CBC mode 256.

– **List(int, int)**: This method takes a starting and ending message ID and returns a set of Message objects within that range.

– **Get(int)**: This method takes a message ID and returns the corresponding Message object if found, otherwise it returns a default Message object with an invalid ID.

– **Add(string, string, string)**: This method adds a new message with the provided title, author, and body to the BBS. It generates a unique ID, creates a Message object, stores it in the messages map, and encrypts the message before writing it to the message file.

– **size()**: This method returns the number of messages currently stored in the messages map.

– **retrieveLastId()**: This method iterates through the messages map and returns the highest message ID used.
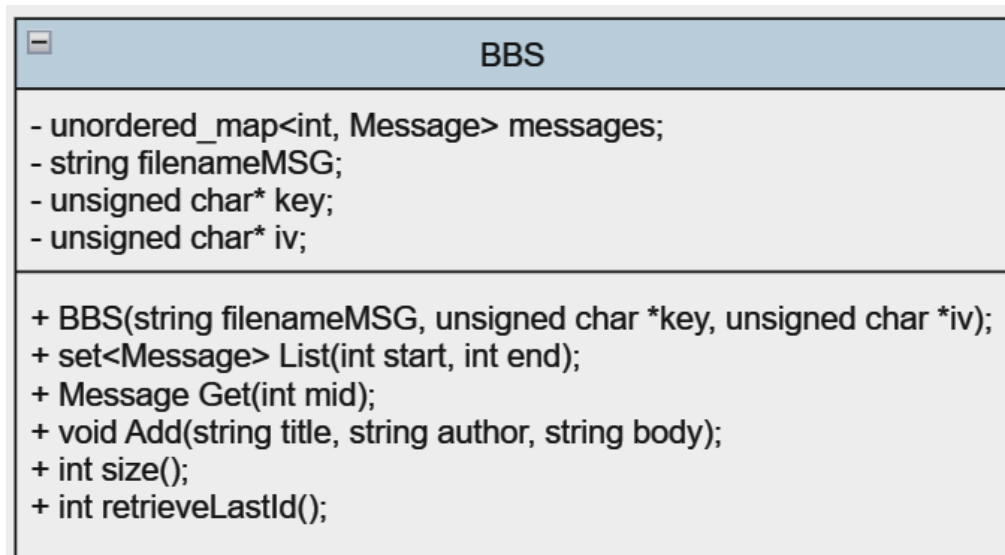
Figure 2.10: Bulletin Board System class diagram

Server.cpp

These classes represent the core of the server side, providing support for multiple connections with up to 10 clients. The class diagram shown in Figure 2.11 is composed of the following components:

- Attributes:

  – **serverSocket**: An integer representing the socket used by the server to listen for incoming client connections.

  – **board**: A BBS object that manages the message board functionalities like storing, retrieving, and listing messages.

- **clientMsgNumMap**: An unordered map that stores information about connected clients. The key is the client socket number (integer), and the value is likely an integer representing a message counter or sequence number for each client.
- **users**: A list of Client objects representing all the currently connected clients.
- **public_key**: A string containing the server's public key used for encryption.
- **private_key**: A string containing the server's private key used for decryption.

- Public Methods:
  - **Server()**: The constructor, likely responsible for initialization tasks.
  - **start()**: Starts the server by listening on the specified socket and accepting client connections.
  - **sendMsg(int, const char*, string)**: Sends a message (likely encrypted) to a specific client using their socket and using a shared secret for additional security.
  - **recvMsg(int, char*, string)**: Receives an encrypted message from a specific client, using a shared secret for decryption.
  - **handle(int)**: The main loop for handling communication with a specific client. This likely involves receiving messages, processing them based on their content, and sending responses back to the client.
  - **findUserOnFile(const char*, const char*)**: Checks if a user with the provided username and password exists in a user file which is encrypted using AES CBC mode 256.
  - **checkUsernameOnFile(const char*)**: Checks if a username already exists in a user file.
  - **handleLogin(int, string)**: Handles the login process for a client, likely by verifying credentials and establishing a shared secret for secure communication.
  - **handleLogout(Server, int, string)**: Handles the logout process for a client, removing it from the connected user list and invalidating their shared secret.
  - **findMsgOnFile(char*)**: Searches for a specific message on the message board based on an identifier (the message ID) and returns the corresponding Message object if found.
  - **handleGetMessages(int, string)**: Handles a client request to retrieve messages from the message board.

- **extractNoteDetails(const string&)**: Helper function for extracting details like title and body from a message represented as a string.

- **handleAddMessages(int, string)**: Handles a client request to add a new message to the message board.

- **handleListMessages(int, int, int, string)**: Handles a client request to list messages on the board, potentially specifying a range of messages by start and end positions.

- **handleRegistrationChallenge(int, string)**: Handles a challenge sent by the server during the user registration process for verification.

- **handleRegistration(int, string)**: Handles the user registration process, involving creating a new user account and establishing a shared secret.

Server
- int serverSocket; - BBS board; - unordered_map<int, int> clientMsgNumMap; - list<Client> users; - string public_key; - string private_key;
+ Server(); + void start(); + void sendMsg(int clientSocket, const char* msg, string client_secret); + void recvMsg(int clientSocket, char* msg, string client_secret); - void handle(int clientSocket); - bool findUserOnFile(const char* username, const char* password); - bool checkUsernameOnFile(const char* username); - bool handleLogin(int clientSocket, string client_secret); - void handleLogout(Server, int clientSocket, string client_secret); - Message findMsgOnFile(char* identifier); - void handleGetMessages(int clientSocket, string client_secret); - pair<string, string> extractNoteDetails(const string& note); - void handleAddMessages(int clientSocket, string client_secret); - void handleListMessages(int clientSocket, int start, int end, string client_secret); - bool handleRegistrationChallenge(int clientSocket, string client_secret); - bool handleRegistration(int clientSocket, string client_secret);

Figure 2.11: Server.cpp class diagram

3 | Protocol design

Client(s) and Server, can send and receive two different type of messages:

1. **Plain message**: a plaintext message. These messages are used only by the server to send its public key to the client and viceversa;

2. **Encrypted message**: A ciphertext message encrypted using the Shared Secret key, which is calculated by both the client and server through the Diffie-Hellman Key Exchange. The keypair (public key and private key) is generated using OPENSSL EVP_Keygen, and these keys are then used to calculate the shared key as described in Section 3.2 "Key exchange protocol".

All the messages exchanged by the application, are composed as follows:

Figure 3.1: Message formatting

Here, we describe each field of the header and their use:

message The message to be send to the other part, can be encrypted or in plaintext;

delimiter *$$$* Delimiter symbols useful when parsing to remove out the checking parameter when displaying the message to the client;

counter An incremental message counter. It starts from 0 and it's incremented for each message sent. It is used to avoid replay attacks;

delimiter *@@@* Delimiter symbols used to separate the counter from the next information;

hash The computed Hashing of the message, the $$$ delimiter and the counter. This is due to guarantee the integrity of the message.

3.1 Security of the Protocol

Signed and CBC-encrypted messages require careful consideration to avoid security flaws.

CBC (Cipher Block Chaining) needs to be initialized with a key and an initialization vector (IV). To ensure security, the same IV must not be reused with the same key. In our case, the IV is an unsigned char array of length `EVP_MAX_IV_LENGTH`, which is 16 by default; this array is filled using the `RAND_bytes` function provided by the OpenSSL library.

Regarding encrypted messages, we need to implement additional protection against replay attacks. An attacker may record the messages exchanged between two parties (e.g. the client and the server) and resend some of the recorded messages to one party to impersonate the other. In this case, the receiver may consider the message valid since it comes with a valid signature.

The `counter` field is used to avoid replay attacks within a single session: it cannot prevent replays between different sessions since, when the application is restarted, the counter is reset to 0.

3.2 Key exchange protocol

Figure 3.2 shows the sequence diagram of a session with two parties: Client and the Server.
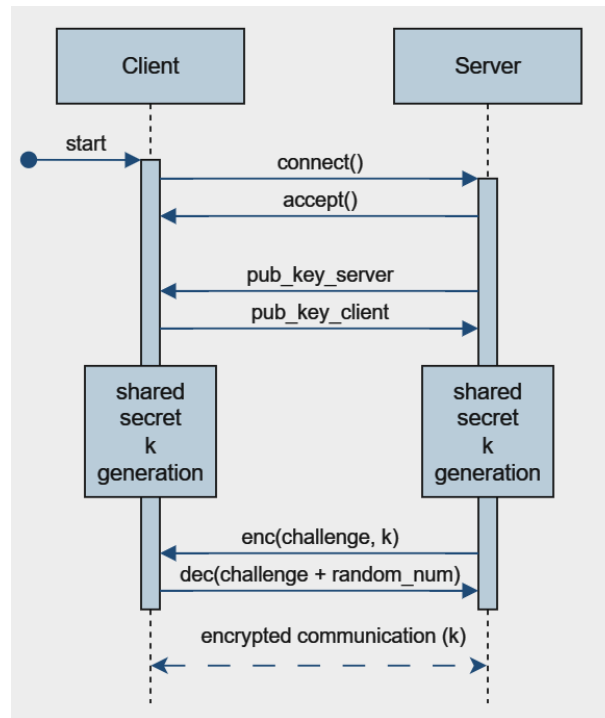
Figure 3.2: Key Exchange Protocol between the server and a client

3.2.1   Formal description

1. **Server Availability Check**

   (a) Client verifies server is available at the specified IP address and port.

2. **Client Initiates Connection**

   (a) Client sends a connection request to the server at the specified IP address and port.

3. **Server Accepts Connection**

   (a) Server accepts the connection request from the client.

   (b) Server establishes a client socket for communication.

4. **Key Generation**

   (a) Client generates a key pair (public key, private key).

   (b) Server generates a key pair (public key, private key).

5. **Server Public Key Exchange**

   (a) Server sends its public key to the client in plain text through the client socket.

6. **Client Processing**

   (a) Client receives the server's public key.

   (b) Client generates a shared secret key using the received server's public key and its own private key.

   (c) Client sends its public key to the server through the client socket.

7. **Server Processing**

   (a) Server receives the client's public key.

   (b) Server generates a shared secret key using the received client's public key and its own private key.

   (c) Server encrypts a challenge message (a string) using the shared secret key derived from the client's public key.

   (d) Server sends the encrypted challenge message to the client.

8. **Client Challenge Response**

   (a) Client receives the encrypted challenge message from the server.

   (b) Client decrypts the challenge message using the shared secret key derived from the server's public key.

(c) Client sends an acknowledgment message back to the server.

(d) Client encrypts its own challenge message using the shared secret key.

(e) Client transmits the encrypted challenge message to the server.

9. **Server Challenge Verification**

(a) Server receives the acknowledgment message from the client.

(b) Server receives the encrypted challenge message from the client.

(c) Server attempts to decrypt the client's challenge message using the shared secret key established earlier.

Following this secure key exchange, subsequent data exchanged between the client and the server can be encrypted using the established shared secret key, ensuring confidentiality and data integrity, see Figure 3.2

4 | Exchanged messages

In this section we describe the different types of messages which are exchanged in the system. The definition of each message can be found in `common/protocol.h`.

4.1 Client-Server Communication Protocol

This section details the communication protocol between the client and server applications. The messages exchanged utilize encryption to ensure data security, particularly for sensitive information like usernames and passwords.

- **LOGIN REQUEST** (*Encrypted*): The client initiates login using this command. The server's 'handleLogin' function processes it. The detailed process is the following:

  1. The client sends the `CMD_LOGIN` command to the server.
  2. The server prompts the client for username and password credentials.
  3. The client transmits the username and password to the server.
  4. The server verifies the credentials using the `findUserOnFile(const char *, const char *)` function located in 'server.cpp'.
  5. Upon successful authentication, the server sends a success message to the client, enabling a logged-in state.

- **REGISTRATION REQUEST** (*Encrypted*): The client sends a registration request using this command. The server's 'handleRegistration' function handles it. The process is the following:

  1. The client sends the `CMD_REGISTRATION` command to the server.
  2. The server prompts the client for username, password, and email address.
  3. The client transmits the username, password, and email address to the server.
  4. The server verifies username availability using the `checkUsernameOnFile(const char *)` function located in 'server.cpp'.

5. If the username is available, the server generates a one-time password (OTP) for verification, which is then prompted to the client.

6. Upon successful verification of the OTP entered by the client, a new **User** object is created and saved to a file.

- **ADD MESSAGE REQUEST** (*Encrypted*) {`CMD_ADD`}: Enables logged-in clients to submit new messages. The server's 'handleAddMessages' function (located in 'server/server.cpp') processes this request. The process is as follows:

  1. The client sends the `CMD_ADD` command to the server.

  2. The server prompts the client for the message title, author, and body.

  3. The client transmits the message title, author, and body to the server.

  4. The server creates a new **Message** object and adds it to the **BBS** object (representing the main message board).

- **LIST MESSAGES REQUEST** (*Encrypted*) {`CMD_LIST`}: Enables logged-in clients to request a list of available messages. The server's 'handleListMes

  1. The client sends the `CMD_LIST` command to the server.

  2. The server retrieves the message list from the **BBS** object using the 'List(int start, int end)' method.

  3. The server transmits the message list to the client.

- **GET MESSAGE REQUEST** (*Encrypted*) {`CMD_GET`}: Enables logged-in clients to retrieve a specific message by its ID. The server's 'handleGetMessages' function (located in 'server/server.cpp') processes this request. The process is as follows:

  1. The client sends the `CMD_GET` command along with the message ID to the server.

  2. The server retrieves the message with the corresponding ID from the **BBS** object using the 'Get(int mid)' method.

  3. The server transmits the retrieved message to the client.

- **Logout Request** (*Encrypted*) {`CMD_LOGOUT`}: Logged-in clients can use this command to terminate their session. The server's 'handleLogout' function processes this request.

- **(Unrecognized) Command** (*Encrypted*) {`INVALID_COMMAND`}: The server responds with this message if it receives an unrecognized command from the client.

4.1.1 Server Responses

The server responds to client commands with messages indicating success, failure, or providing requested information.

- **Error Message** (*Plain*) {`ERROR`}: The server sends this message if an error occurs during command processing.

  Successful Command Execution (*Plain*) {`OK`}: The server sends this message to acknowledge successful execution of a client command.

Appendix A

Compile and Run

Here we illustrate how to compile and run the application.

A.1   Compile and run the application

We need to compile both the server and client. To do so, we compile first the server with the following command:

```
$ cd server && make clean && make server
```

This will enter the server folder, will clean the previous binaries, compile the 'server.cpp' and all related library into an executable file and it will automatically run it. Similarly, the same can be performed to compile and execute the client. The command to execute are:

```
$ cd client && make clean && make client
```

The following pairs of usernames and (very secure) passwords can be used for testing:

```
qwerty:1234567890
qwerty2:1234567890
```

The project structure is the following:

```
/
├── client
│   ├── clientHandler.cpp
│   ├── Makefile
│   └── otp.txt
├── common
│   ├── crypto
│   │   ├── aes.cpp
│   │   ├── hashing.cpp
│   │   └── key_exchange.cpp
│   ├── Message.cpp
│   └── protocol.h
├── README.md
├── server
│   ├── BBS.cpp
│   ├── Client.cpp
│   ├── database
│   │   ├── BBS.txt
│   │   └── users.txt
│   ├── Makefile
│   ├── server.cpp
│   └── User.cpp
├── Report.pdf
└── proj_guidelines_2024-Bullettin_Board_System.pdf
```