UNIVERSITÀ DI PISA

Master's Degree in Cybersecurity

Project Report
Bulletin Board System

Student:
Giuseppe Conte

February 2025

Contents

Introduction

The project involves the development of a **Bulletin Board System (BBS)**, a distributed system[1] that allows users to read messages and add their own. Each user is identified through a unique nickname, chosen during registration along with a password.

Messages within the BBS are represented as tuples consisting of the following fields: *unique identifier*, *title*, *author* (which corresponds to the nickname of the user who added the message), and *body*. The BBS offers several features to registered users:

- `List(n)`: shows the last *n* messages available in the system.
- `Get(mid)`: allows downloading a specific message identified by the ID `mid`.
- `Add(title, author, body)`: allows adding a new message to the system by specifying its title, author, and body.

To use these features, users must be registered and have successfully logged in. Each operation is executed through a secure channel, ensuring the confidentiality of communications. Users cannot interact with the system until they log in again after disconnecting.

The BBS is implemented in a centralized manner through a **BBS server**, accessible via a well-known pair of *IP address and port*. The server has a public-private cryptographic key pair: the public component (`pubKbbs`) is shared with users to ensure the security of communications.

Users interact with the system through two main phases:

- **Registration**: the user connects securely to the BBS server and provides an email address, a nickname, and a password. The server generates a challenge, prints it on the terminal, and waits for the client to enter the correct value. If the challenge is passed, the server saves the information and completes the registration.[2]
- **Login**: a registered user connects securely to the BBS server and provides their nickname and password. The server verifies the credentials and grants access. Once logged in, a secure session is established, which remains active until the user disconnects.

[1] The implemented project is closer to a centralized system than a distributed one.

[2] For simplicity, the challenge is not sent via email as required by the specification, but is printed on the server's terminal to be entered by the client.

Compilation and Execution on the Terminal

To compile and run the project, some preliminary steps are necessary to install the dependencies and correctly configure the development environment. In particular, the project uses the OpenSSL library, which must be installed on your system.

Installation of OpenSSL

Before compiling the project, it is necessary to make sure that the OpenSSL library is installed on your system. You can install it using the package manager of your operating system. For example, on a Ubuntu-based system, you can use the command:

```
sudo apt install openssl libssl-dev
```

For other operating systems, consult the official OpenSSL documentation for installation instructions.

Folder Structure

The project is organized into three main folders:

- `client_folder/`: Contains the source code related to the client.
- `server_folder/`: Contains the source code related to the server, some server-specific libraries, and the files with user data, posts, and the private key.
- `shared_folder/`: Contains code and libraries shared between the client and the server, and the public key.

Project Compilation

To compile the project, you need to open the terminal and navigate to the folder where the provided `Makefile` is located, which includes a series of useful commands for managing compilation, creating executables, and managing data. The main commands available are described below:

- `make all`: Compiles all the necessary files for the project, generating the executables `client`, `server` and `generate_rsa_keys`. This command performs all the operations necessary for the creation of the executables.

- `make client`: Compiles the file `client.c` inside the `client_folder` folder and generates the executable `client`.
- `make server`: Compiles the file `server.c` inside the `server_folder` folder and generates the executable `server`.
- `make keys`: Compiles the file `Generate_RSA_Keys.c` in the `server_folder` folder and generates the executable `generate_rsa_keys`, useful for generating RSA keys.
- `make clean`: Removes the generated executables (`client`, `server`, `generate_rsa_keys`). This command is useful for cleaning the directory of temporary files and executables.
- `make reset`: Deletes all data related to the project, including the key files (`public_key.pem`, `private_key.pem`) and the data files (`counter.txt`, `posts.txt`, `users.txt`). It is useful to completely reset the execution environment and remove any test data.
- `make debug`: Compiles the project including the debug flags. This command activates the `#ifdef DEBUG` block in the code, enabling the printing of debug messages. Both `client` and `server` are compiled, with the debug option activated.

These commands provide an efficient workflow for compiling and running the project, with specific options for cleaning, key generation, and enabling debug mode.

Project Execution

After successfully compiling the project, it will be possible to run the server and the client on different terminals. To start the server, execute the command:

```
./server
```

To start the client, run the following command on one or more terminals:

```
./client
```

Make sure that the server is running before starting the client, as the client must connect to the server to be able to interact with the system.

RSA Key Generation

To generate RSA keys, you can execute the command:

```
./generate_rsa_keys
```

However, it is not necessary to run this command manually. Every time the server is started, the RSA key generation operation is performed automatically in a separate process. In this way, new RSA keys are generated every time the server is started.

Common Errors

If errors occur during compilation, check that the OpenSSL library is correctly installed and that the system is correctly configured for the `gcc` compiler. Any compilation errors could also derive from changes to the source code or from the lack of some shared libraries.

Project Overview

The project was developed in the C language and designed to be executed on a Linux operating system. The system structure follows a **client-server** architecture, where the **server** is the central component that manages requests from multiple clients simultaneously.

The server is designed to **handle multiple connections concurrently** using **threads**. Each time a client connects to the server, a new dedicated **thread** is created for that connection, ensuring that the server can manage multiple clients simultaneously without blocking other operations and ensuring isolation and fluidity in request management. This approach allows the server to remain responsive, responding in parallel to multiple clients without the need to handle each request sequentially.

A fundamental element for system security is the use of **encryption**. Every client that connects to the server generates a unique **AES** key for that session. This key is then **encrypted** with the server's **public key** using the **RSA** algorithm, sent through the connection, and finally **decrypted** by the server using its **private key**. Once the AES key has been decrypted by the server, all subsequent message exchanges between the client and the server occur using **AES** encryption, ensuring the confidentiality and security of the exchanged information.

Furthermore, the system includes secure management of user credentials. During the **registration** of a new user, the **password** is **encrypted** using a secure **hashing** function. The password, once hashed, is never stored in clear text in the system, but only the hash of the password is saved. This approach increases security and prevents potential vulnerabilities in case of data breaches.

The server is designed to be easily scalable and versatile, allowing the addition of new features and improvements based on the specific needs of the context. To ensure the security of communications, the system makes use of external libraries such as **OpenSSL**, which provides secure implementations for the management of public and private keys, and **pthread**, for thread management.

The project stands out for the adoption of a solid multithreaded architecture and for advanced security measures, such as the management of encryption keys and secure password hashing, thus offering a robust and secure system for communication between clients and servers.

Message Formatting

The project does not include a graphical interface. Therefore, all interactions with the system must be performed through the terminal. The user can send commands directly in the terminal window to perform the desired operations, such as registration, adding posts, or managing the session.

The communication protocol between client and server is based on predefined commands and

structured messages and uses the delimiter `::` to separate the different fields of the messages. Each command is sent from the client to the server along with the necessary parameters, and the server returns a response message, structured or simple, depending on the requested operation. The messages exchanged between client and server follow a well-defined format:

- From client to server: `command::parameter1::parameter2:...`, where each command is followed by the related parameters, separated by the delimiter `::`.
- From server to client: the format of the messages varies according to the context. Error and success responses are simple and do not contain the delimiter `::`, while more complex responses, such as the list of posts, follow a structured format with separate fields.

The main supported commands are:

- `list` and `get`: to request the list of the latest posts or a specific post through its ID.
- `add`: to add a new post.
- `register`: for the registration of a new user.
- `login`: for user authentication.
- `logout`: to log out.
- `close`: to close the program.

For example, a message sent to add a post could be structured as follows:

`add::author::title::text`

where `add` is the predefined command, while the other parameters are in text format.
A response from the server could be:

```
The post has been added to the BBS
```

or

```
Error during post saving
```

which will simply be printed on the Client's terminal. The details of each command, including their syntax and operation, will be described in the section dedicated to the *Client*.

Server

Server Operation

The project server is started using the command:

```
./server
```

Upon startup, the server performs two main operations. The first is the generation of a pair of public and private keys, necessary exclusively to encrypt the AES symmetric key transmitted by the client, ensuring security in communication. To do this, the server starts a separate process that creates these keys. Once the key pair has been generated, the process terminates, while the server continues with other operations.

The second part of the server deals with managing client connections and all subsequent operations. First of all, the server masks system signals such as `SIGINT`, `SIGTERM`, `SIGQUIT` and `SIGHUP`, to prevent these signals from interrupting the server's operation at inappropriate times. Subsequently, a special thread is created, called **sigHandler**, which waits for one of these signals. If, for example, the user sends the `SIGINT` signal (by pressing `Ctrl+C` in the server terminal), the server will stop in a controlled manner, performing the closing operations.

The server then prepares to handle incoming connections from clients, opening a listening socket on a known address. While waiting for a connection, the server remains ready to receive requests. Every time a client connects, a new thread is started dedicated to handling their request. In this way, the server is able to manage multiple clients simultaneously without blocking the execution of other operations, since each client has its own separate thread. Once the client's thread has completed its operation, the server returns to waiting for a new connection.

In the event that the server receives a signal from the **sigHandler**, which indicates the intention to stop the server, the following actions are taken: first of all, the server stops accepting new connections. Then, the server terminates all active connections, performs the `join` on all active threads (including the threads that manage the clients and the signal handler thread), closes the socket and finally terminates execution. This process ensures that the server shuts down in an orderly manner, without leaving incomplete operations or unreleased resources.

Regarding concurrency management, each client has its own dedicated buffer. This means that messages between different clients do not mix, and each client can communicate securely without interfering with others. This approach allows concurrency to be managed without conflicts, keeping each client's data separate.

In summary, the server offers robust management of client connections, stops safely by listening

to system signals, and effectively manages concurrency through separate threads and independent buffers for each client.

Data Saving

The server is responsible for managing and saving all data, which is stored in files. The public key, stored in the `public_key.pem` file, is saved in the `shared_folder` folder, while all other data is kept in the `server_folder` folder. Among these files are the private key, stored in the `private_key.pem` file, the `posts.txt` file containing the posts, the `users.txt` file with user credentials, and the `counter.txt` file which stores the last ID assigned to a post.

Posts are stored in the following format:

```
id::author::post_title::post_text
```

Each post includes a unique ID, the author's name, the title and the content of the post. Posts are saved in the file in chronological order, from the oldest to the most recent. Therefore, to read the most recent posts, it is necessary to scroll through the entire file.
The **ID** is a 4-character alphanumeric string, composed of uppercase letters, lowercase letters and numbers. IDs are generated sequentially, starting from `AAAA`, `AAAB`, and so on. To speed up the creation of the next ID, the last assigned ID is stored in the `counter.txt` file.
It is possible to generate up to 62^4 different IDs.

User credentials are stored in the following format:

```
email::nickname::salt::hash_password
```

The file stores the email and nickname of each user, along with the password hash and the related salt. The use of the salt ensures that users with the same password have different hashes, thus improving the security of the system.

Client

Client Operation

The project client is started using the command:

```
./client
```

Upon startup, the client performs a series of initial operations: it creates the socket and connects to the server, generates the AES key and the IV, then starts the `handle_communication` function, which handles communication with the server.
Inside `handle_communication`, two threads are created: one for sending messages and one for receiving them. The function waits for their termination with a *join*, after which control returns to `main`, which closes the socket and finally terminates the program.
The message sending thread manages the inputs that the user types and monitors a timer that resets every time the user sends a message to the server.
The execution of the client terminates in one of the following cases:

- The user types the command `close`.
- The timer expires, indicating that the user has not sent any messages within the time limit.
- The server closes.

Thread Management in the Client

In the project, the client is designed to operate asynchronously, without waiting for a response from the server after each command sent. To achieve this behavior, the client uses two independent threads that handle sending and receiving messages, respectively.

- **Sending Thread**: deals exclusively with sending messages to the server. Each command is sent independently, without the client stopping to wait for a response. In this way, the client can continue to send commands without interruptions.
- **Receiving Thread**: manages listening for messages sent by the server. Every time the server sends a message, it is received and printed to the screen by the client. The receiving thread is in continuous execution, ready to print any message that arrives from the server.

This multithreaded architecture allows the client to be always responsive: this allows to immediately print the messages sent by the server, such as the server closing message, and to display all

received posts in sequence. If instead a single-threaded model had been adopted, in which each request necessarily waits for a response before proceeding, it would have been complex to effectively manage these scenarios, compromising the fluidity of communication.

Some Useful Functions

In the program there are several utility functions, including the following two, fundamental for managing input and encrypted communication:

`myscanf(string, min, max)`
The `myscanf` function verifies that the string typed by the user has a length between `min` and `max`. Furthermore, it checks that the string does not contain the special character `":::"`, used to separate the fields in the messages sent to the server.

`send_encrypted_message(socket, key, iv, message)`
The `send_encrypted_message` function automatically handles the encryption and sending of a message to the server. It uses the `key` and the initialization vector `iv` to encrypt the `message` with the AES algorithm before transmitting it through the `socket`.

Message Sending Thread

The `send_thread` function starts by encrypting the AES key with the public key using the RSA algorithm and sending it to the server. Subsequently, it sends the IV value in clear text.

After this initial phase, the function enters a loop in which a timer is set: if the client does not send commands within a time limit, the client closes automatically.

The client then waits for a command from the user and, through a series of `if-elseif` structures, performs the appropriate operation based on the received command. If the command is not recognized, the client does not perform any operation.

The exit from the loop occurs when the user types the `close` command, which causes the client to close.

The Commands

In the code there is an atomic variable `loggedin` that indicates whether there is a logged in user or not. The commands behave differently depending on the value of `loggedin`.

Command `register`

```
register
```

The `register` command is used to register a user. If `loggedin == 0`, then the user can register. The command invokes the `register` function to compile the message, during which the user will be asked to enter an email, a username and a password. However, for simplicity, no checks are performed on the validity of the entered values (for example, it is not verified if the email field actually contains a valid address).
The three values are concatenated using the character `"::"` as a separator. Subsequently, the prefix `"register::"` is added to the beginning of the message before sending it to the server.
Consequently, the sent message has the following format:

```
register::email::nickname::password
```

Finally, the user is asked to enter a numerical value related to the challenge, which is subsequently sent to the server.

Command `login`

```
login
```

If `loggedin == 0`, the `login` command invokes the `login` function to compile the authentication message. During the execution of the command, the user will be asked to enter their nickname and password.
The final message will be formatted as follows:

```
login::nickname::password
```

Command `logout`

```
logout
```

If `loggedin == 1`, the `logout` command sends the corresponding message to the server to log out the user, and the value of `loggedin` is set to 0.

Command `add`

```
add
```

The `add` command allows you to write a post. If `loggedin == 1`, the `write_post` function is invoked to formalize the message. During this phase, the user will be asked for the title and text of the post.
However, as with every message written by the user, there is a maximum character limit beyond which the post is not considered valid. After entering the content, the user is asked: `Are you sure you want to add this post?` If the user types `"Y"`, the message is formatted as follows:

```
add::author::title::text
```

and subsequently sent to the server.

Command `list`

```
list
```

The `list` command is used to request the latest posts published in the BBS. If `loggedin == 1`, the `flist` function is invoked, which asks how many posts to load. The message sent to the server will be formatted as follows:

```
list::num
```

Command `get`

```
get
```

The `get` command allows you to request a specific post. If `loggedin == 1`, the `fget` function is invoked, which asks the user to enter the ID of the desired post. The message sent to the server will be formatted as follows:

```
get::id
```

Command `close`

```
close
```

The `close` command is used to close the client. It does not send any message to the server, but sets the loop control variable to `1`, thus causing the exit from the main program loop.

Command `hack`

```
hack
```

If the program was compiled in **debug** mode, it is possible to type this ”secret” command to facilitate debugging.
Specifically, writing the next line bypasses the character check performed by the `myscanf` function. The message is then encrypted and sent to the server exactly as it was written, without any verification.
This allows to test the behavior of the server when it receives *crafted messages*, simulating possible attack attempts or errors in input management.

Message Receiving Thread

The message receiving thread remains running in a loop until the client terminates, as described previously. This thread is always waiting for messages from the server.
When it receives a message, the thread performs the following operations:

- Before any other operation, the thread decrypts the received message.
- If the received message is `"LOGIN_OK"`, it sets the value of the `loggedin` variable to 1, indicating that the user is correctly logged in.
- If the message contains a post, and therefore starts with `"LIST::"` or `"GET::"`, the thread performs a `split` operation to get the following fields: `id`, `author`, `title` and `text`. Subsequently, it prints each field on a separate line.
- Otherwise, the thread prints the decrypted message as is, without any modification.

System Security

Protection of User Credentials

To ensure that user passwords are never stored or transmitted in clear text:

- On the server side, passwords are not saved directly, but only their **hash** accompanied by a random **salt**.
- This ensures that two users with the same password have different hashes, making attacks based on precomputed tables (*rainbow tables*) ineffective.

Communication Security

All communications between client and server are encrypted to guarantee **confidentiality, integrity and protection against replay attacks**:

- When the client connects to the server, it generates an **AES** key and an **IV** for symmetric encryption.
- The AES key is encrypted with the server's **RSA** public key and sent to the server, thus ensuring a **secure key exchange**.
- After this first exchange, all subsequent communications are encrypted with AES, including the sending of the user's password.
- Each connection between client and server uses different AES keys, making **replay attacks impossible**, since a reused message cannot be decrypted in a different session.

Non-malleability and protection against malicious input

The server was designed to ignore malformed or intentionally manipulated messages:

- If a received message is not well formed or lacks the required parameters, the server **does not perform any operation**.
- For example, if a client tries to send an `add` command without specifying title and text, the server will refuse the operation.
- If the specified author does not match the nickname of the logged in user, the server will not accept the post.

- During the registration phase, the server verifies that the challenge code sent by the client has the correct format (six numeric digits).
- The client itself performs checks on the inputs before sending the messages, so if the server receives incorrect inputs, it is most likely an attack attempt.

To facilitate debugging and test the server behavior with anomalous inputs, the `hack` command was added (active only in debug mode). This command allows to send arbitrary messages to the server, bypassing the client checks.

RSA key management and long-term security

At each server startup, a new pair of **RSA (public-private)** keys is generated, with the previous pair being deleted. This improves security for several reasons:

- If an attacker compromised the private key, they could only decrypt the communications **of the current session**, but not future ones.
- Even if an attacker recorded the encrypted traffic for long periods, they **could not decrypt past sessions in the future**, because each server startup uses a different key.
- This reduces the effectiveness of persistent attacks and limits the time window in which a stolen key is useful.

Although this strategy **does not guarantee Perfect Forward Secrecy (PFS) in the strictest sense**, it is still an important security measure. In fact, PFS would require that not even during the current session an attacker with access to the private key could decrypt previous messages. In our case, if an attacker managed to steal the key **while the server is running**, they could still read the traffic of the current session.

Test and Debugging

An automated test suite has not been implemented, but the system's behavior has been manually verified:

- The server is started and then the client is started.
- Registration, login, post addition, and post retrieval operations are performed to verify the expected behavior.
- It is possible to compile the program in debug mode with the command `make debug`, which enables code sections conditioned by `#ifdef DEBUG`. In this mode, the values of some variables are printed to the screen to monitor their contents.
- Furthermore, in debug mode, it is possible to use the secret command `hack`, which disables the checks on the inputs typed by the user, allowing the sending of arbitrary messages to the server. This allows testing the server's behavior in the presence of anomalous or malformed inputs.

Conclusions

The project implements a secure communication system between client and server, adopting advanced encryption techniques to guarantee the confidentiality, integrity, and authenticity of the transmitted data. The combined use of RSA for the secure exchange of the AES key and AES for symmetric encryption ensures that each session is isolated and protected from interception or replay attacks.

From a security point of view, the system protects user credentials by hashing passwords with random *salts*, thus preventing attacks based on precomputed tables. Furthermore, the management of RSA keys rotated for each server startup reduces the possibility of long-term compromises, making each session independent of previous ones.

The multithreaded architecture of the client guarantees a fluid and responsive experience, allowing messages to be received and sent asynchronously without operational blocks. At the same time, the input validation mechanisms and the server-side error management system prevent the occurrence of exploits based on malicious input or attack attempts for data manipulation.

The integration of modern cryptographic techniques and good credential management practices gives the system a high level of security, reducing the risks arising from cyber attacks and ensuring the protection of transmitted data.

Notes and Possible Improvements

Perfect Forward Secrecy (PFS)

Currently, the symmetric key exchange occurs by encrypting the AES key with the server's RSA public key. This system guarantees secure transmission, but does not offer Perfect Forward Secrecy in the strictest sense. If an attacker were able to obtain the server's private key, they could decrypt all past intercepted communications. A possible improvement would be to use Diffie-Hellman for the symmetric key exchange, which would guarantee that even knowing the server's private key, an attacker could not recover the keys from past sessions.

Note on Challenge Management during Registration

During the registration process, as initially described in the *Introduction* section, the user securely connects to the BBS server and provides an email address, a nickname, and a password. The server generates a challenge, prints it on the terminal, and waits for the client to enter the correct value. Once the challenge is passed, the server saves the information and completes the registration.

It is worth noting that, although the specification required the challenge to be sent via email, for simplicity, in my project, the challenge is only printed on the server's terminal.

Memory Management

In the message exchange system between the client and server, the buffer has a maximum size defined by the constant `BUFFER_SIZE`, and therefore the maximum size of each individual message is limited to `BUFFER_SIZE`. Each parameter within the message has a predefined maximum size, and the sum of all parameter sizes never exceeds `BUFFER_SIZE`, thus preventing the risk of overflow. However, this may lead to a small amount of unused memory, due to the predefined size of the parameters, which could result in a slight memory waste.

File Access Management

Access to the files `test.txt`, `post.txt` and `counter.txt` is protected by locks, preventing multiple threads from accessing them simultaneously in an uncontrolled manner. This ensures data consistency and avoids corruption or unwanted concurrent access.

Security in Client Threads

In the client, the control variable that regulates the execution of the main loop is declared atomic. This ensures that, when the client needs to terminate, the modification of the variable is correctly seen by the other thread, avoiding race conditions.

The variable that stores the nickname of the connected user is also atomic. This ensures that the modification of the nickname, by one thread, is correctly synchronized and visible to the other thread, avoiding conflicts in its update during the execution of the program.

Limitation of Simultaneous Users and Overload Prevention

The server is designed to handle a limited number of simultaneous connections. Beyond this threshold, the server refuses new connections. Furthermore, the client automatically closes after a certain period of inactivity, freeing up resources if a user leaves the connection open without interacting. However, despite security measures being implemented, the system is not sufficient to prevent Flooding attacks, in which a large number of simultaneous requests are sent to the server, saturating its resources.

Concurrency Management and Use of `sleep()`

The use of `sleep()` has been employed in certain critical parts of the code to avoid errors related to concurrency between threads. Despite the implementation of synchronization mechanisms on the server, there were situations where it became necessary to introduce an explicit delay to ensure the correct functioning of the system.

A specific example of this issue occurred during the execution of the `List` command, when the client requests multiple posts from the server in sequence. The server sends the posts one at a time, but due to latency between the messages sent via sockets, the posts sometimes overlap with each other. This means that the data of one post may be partially or completely mixed with the data of another, causing confusion on the client side as it receives the data.

To mitigate this race condition, a delay was introduced using `sleep()` between operations. Although this solution reduced the problem, message overlap still occasionally occurs, suggesting that more robust synchronization between the client and server may be needed.

A possible solution to this problem could be the adoption of a message acknowledgment system, similar to a SYN-ACK mechanism, where the client receives confirmation of the correct receipt of one post before moving on to the next. This would prevent posts sent by the server from overlapping with each other, improving synchronization between the two.