UNIVERSITÀ DI PISA

Master degree in Computer Engineering

880II Foundation of Cybersecurity

# Bulletin Board System

Designer:

**Gabriele Suma**

ACADEMIC YEAR 2023/2024

# Contents

# Chapter 1

# Introduction

In this documentation, we will discuss the implementation choices made to realize the functionalities required by the specifications for a **Bulletin Board System (BBS)**, a distributed service where users can read and add messages.

## 1.1 Tools

The development environment for the project was based on **WSL** (Windows Subsystem for Linux), running **Ubuntu 20.04 LTS**. The programming language used is **C++**, utilizing the lcrypto library provided by **OpenSSL**. For the demo, both the server and client are hosted on the **localhost** address, using port **4242** by default. The server employs an **I/O multiplexing** technique to handle multiple sockets sequentially without using additional threads. The client operates on a single thread and communicates with the server via the **TCP** protocol to enhance system security and to avoid any kind of vulnerability introduced by the UDP protocol. At the start, the server listens on the main socket and waits for new connection from the clients. When a client is connected successfully, it is able to perform the functionalities defined by the specifications.

## 1.2 Client

A client can perform:

- **register** [Email] [Username] [Password]: securely connect to the BBS server and specifies an email,username and password.
- **login** [Username] [Password]: securely connect to the BBS server and logs in by means of his/here username and password.
- **logout**: logs out and ends the session with the server.
- **list** [N° of latest posts]: lists the latests **n** posts available in the BBS.
- **get** [Id]: downloads from the BBS the message specified by the post identifier **Id**.
- **add** [Title] [Author] [Body]: adds a new post to the BBS.
- **exit**: closes the client.

## 1.3 Server

According to the specifications, the BBS server is centralized and operates at a well-known (IP, port) pair. The server is equipped with a private-public key pair, where the public key component, $\mathbf{K_{PUB_{BBS}}}$, is known to users. This key pair has been generated using the OpenSSL library through the command line. Then the server uses the key pair ($\mathbf{K_{PRV_{BBS}}}$, $\mathbf{K_{PUB_{BBS}}}$) and an additional key, $\mathbf{K_{AES_{BBS}}}$, for post encryption and decryption.

# Chapter 2

# Implementation

## 2.1 Registration

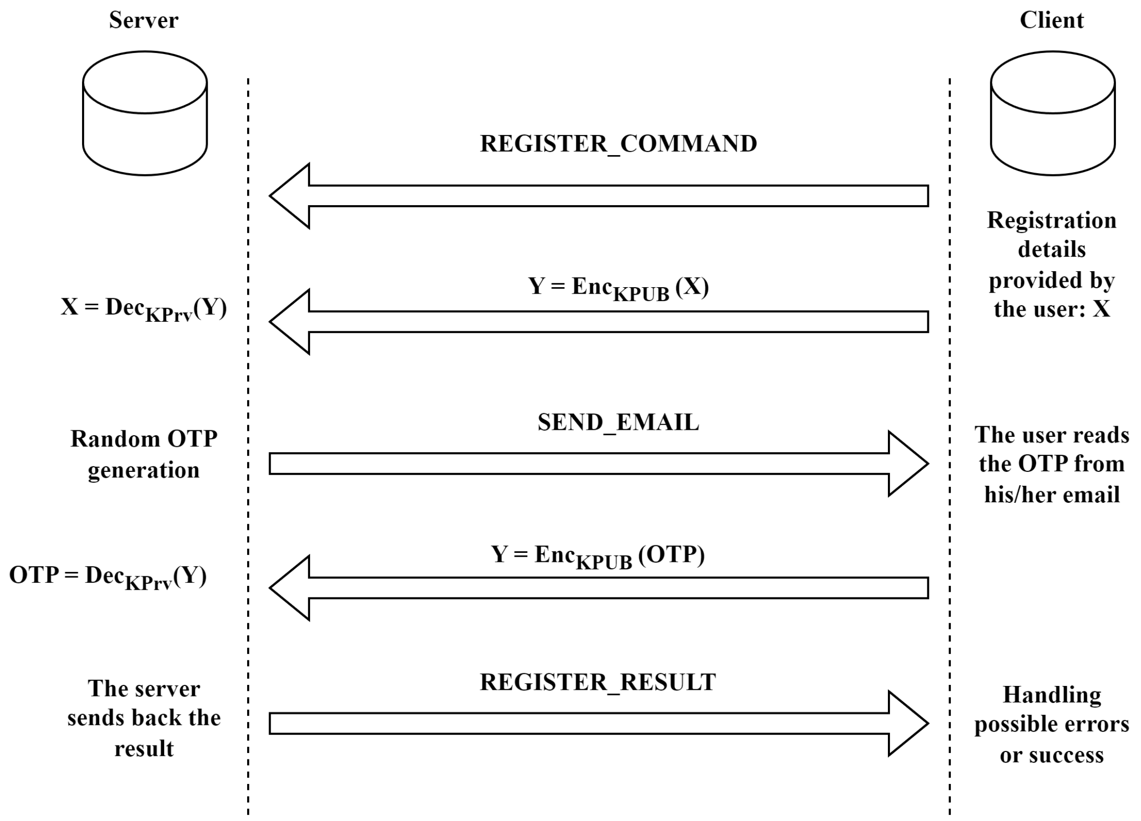

Figure 2.1: Registration protocol.

With the server's public key being publicly known, the choice was straightforward: the **RSA cryptosystem**. The implementation specifically uses a **digital envelope**. The registration flow is as follows:

1. The client sends a **REGISTER_COMMAND** in clear text to the server.

2. The information, submitted by the user, is encrypted using the **server's public key**.
3. The server **decrypts** the received message using **its private key** and sends an email with a random OTP number to ensure **two-factor** authentication.
4. If the user correctly delivers the **OTP** message (encrypted using the server's public key), the registration is successfully completed.

### Remarks

The **password** provided by the user is hashed using **SHA-256** and will be stored by the server as a **long-term shared secret**. This is crucial for the implementation of the users functionalities described later. **For simplicity and demo purposes**, the server's challenge to authenticate the client has been implemented as a write and read operation from a **shared file**. The requests and responses between the client and server, which are in plain text, are actually identifiers represented by **4-byte** values that are well-known to the distributed system.

## 2.2 Login

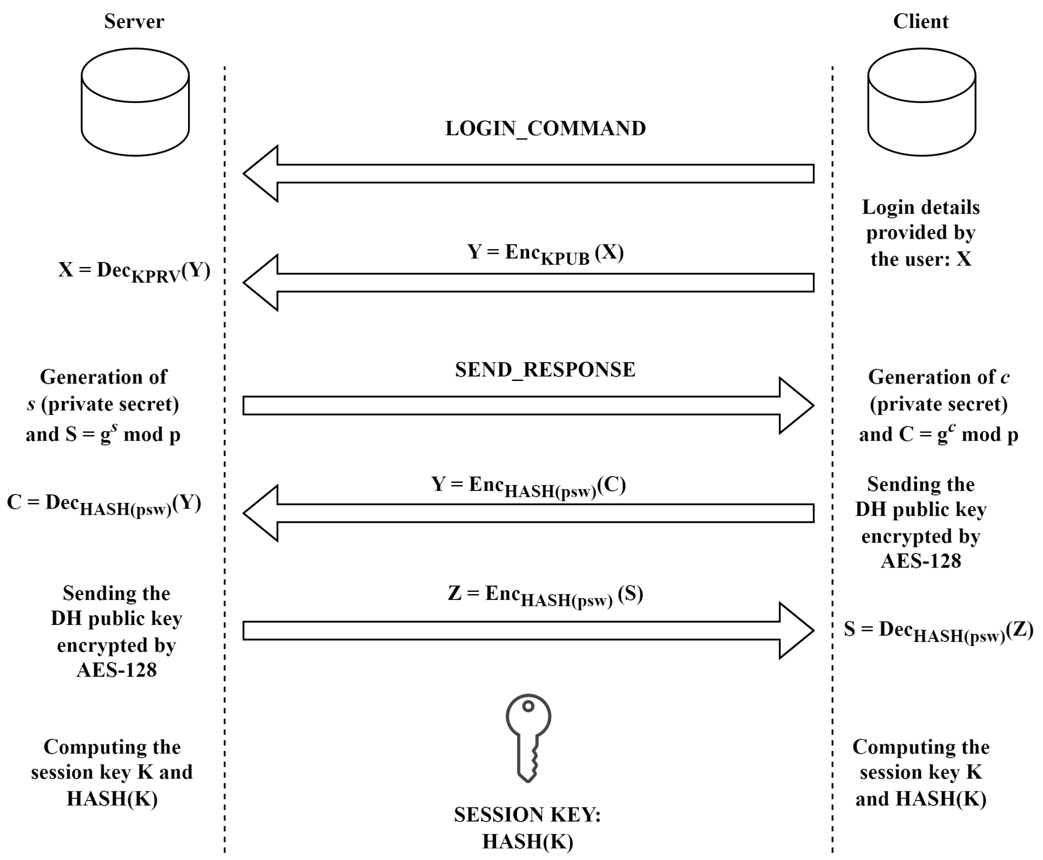
Server
Client
LOGIN_COMMAND
Login details provided by the user: X
$Y = Enc_{KPUB}(X)$
$X = Dec_{KPRV}(Y)$
Generation of $s$ (private secret) and $S = g^s \mod p$
SEND_RESPONSE
Generation of $c$ (private secret) and $C = g^c \mod p$
$C = Dec_{HASH(psw)}(Y)$
$Y = Enc_{HASH(psw)}(C)$
Sending the DH public key encrypted by AES-128
Sending the DH public key encrypted by AES-128
$Z = Enc_{HASH(psw)}(S)$
$S = Dec_{HASH(psw)}(Z)$
Computing the session key K and HASH(K)
SESSION KEY: HASH(K)
Computing the session key K and HASH(K)

Figure 2.2: Login protocol.

The login flow is as follows:

1. The client sends the **LOGIN_COMMAND** request and sends the login details **encrypted** by means of the RSA server's public key.
2. The server **decrypts the credentials** received and **verifies them**. If the verification is successful, the server generates a pair of **Diffie-Hellman private and public keys**.
3. Once received the successful response from the server, the client generates its own pair of **Diffie-Hellman private and public keys**.
4. Subsequently, the client sends its **DH public key** encrypted by means of the long-term secret (Hashed password) to the server.
5. Dually, the server sends its **DH public key** encrypted by means of the long-term secret (Hashed password) to the client.
6. Server and client now shares a common secret in a **not-interactive way**.

### Perfect Forward Secrecy

During the login phase as shown in the [Figure 2.2], the Ephemeral Diffie-Hellman protocol has been implemented. Due to the project specification we needed to guarantee **Perfect Forward Secrecy**. Under this property, even if the long-term shared secret gets disclosed, the secrecy of the exchanged keys from earlier runs are not compromised. The messages **C** and **S** are encrypted by means of the long-term secret shared during the registration phase. Now the long-term secret is used for authentication. In order to obtain the session key **K**, an attacker needs to solve the discrete logarithm problem (DLP). The requirements needed to ensure **PFS** is to use $\boldsymbol{s}$ and $\boldsymbol{c}$ as ephemeral and one-time for each session and once the session has been established, they must be deleted.

## 2.3 List,get,add and logout

To satisfy the constraints about **no-replay attack** and **authentication**, once the session has been enstablished, the client and server communicates using the **AES-128-GCM**. For the actual communication, the distributed system adopts a structured message named **network_message**.

| **uint16_t** | **uint8_t** | **string** | **uint16_t** |
|---|---|---|---|
| Nonce | Request | Content | ContentLength |

Table 2.1: Structure of messages exchanged between client and server.

As the name of the fields suggests:

- **Nonce**: is used to guarantee freshness of the message.
- **Request**: is used as a field in order to identify the requests and the possible information regarding the outcome of a request.
- **Content**: is used to store the actual content.
- **ContentLength**: is used for the length of the content string.

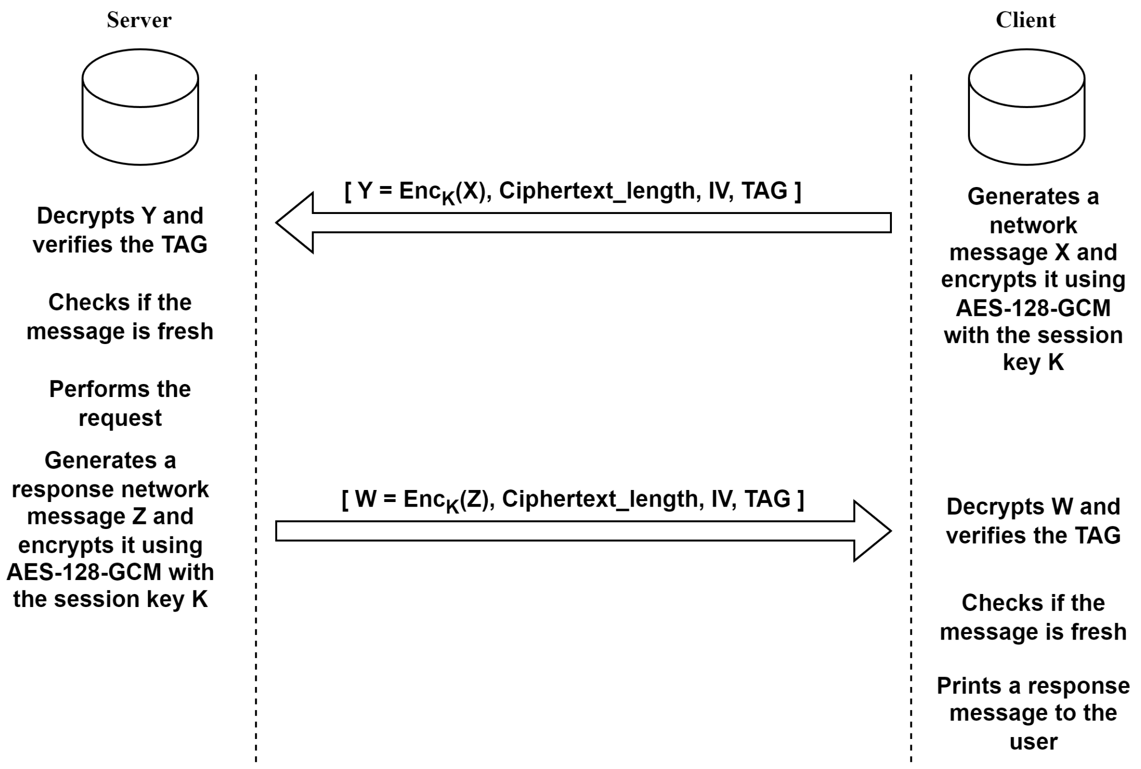
Server
Client
[ Y = Enc$_K$(X), Ciphertext_length, IV, TAG ]
Generates a network message X and encrypts it using AES-128-GCM with the session key K
Decrypts Y and verifies the TAG
Checks if the message is fresh
Performs the request
Generates a response network message Z and encrypts it using AES-128-GCM with the session key K
[ W = Enc$_K$(Z), Ciphertext_length, IV, TAG ]
Decrypts W and verifies the TAG
Checks if the message is fresh
Prints a response message to the user

Figure 2.3: Generic schema followed by List, Get, Add and Logout commands.

**Immunity against replay and dictionary attacks**

For simplicity and demonstration purposes, the nonce is generated using a **counter**. With this mechanism, it is possible to avoid any kind of **replay attack** in the distributed system. Both the server and the client check if the nonce is **fresh**, and the server also checks for a possible **nonce overflow**. In this case the session becomes **expired** and the clients need to establish a new session with the server to still use the functionalities.

## 2.4 Post encryption and decryption

For an additional layer of secrecy, the posts added to the BBS are encrypted using **AES-128**. Every time the server **starts**, it loads all the posts previously added by users, **decrypting** them from the file **post.txt**. The server uses the $\mathbf{K_{AES_{BBS}}}$ and **IV**, which are retrieved from a file during startup. When the server shuts down, instead, it generates a new $\mathbf{K_{AES_{BBS}}}$ and a new **IV**, storing them in the server storage for use in the next startup.

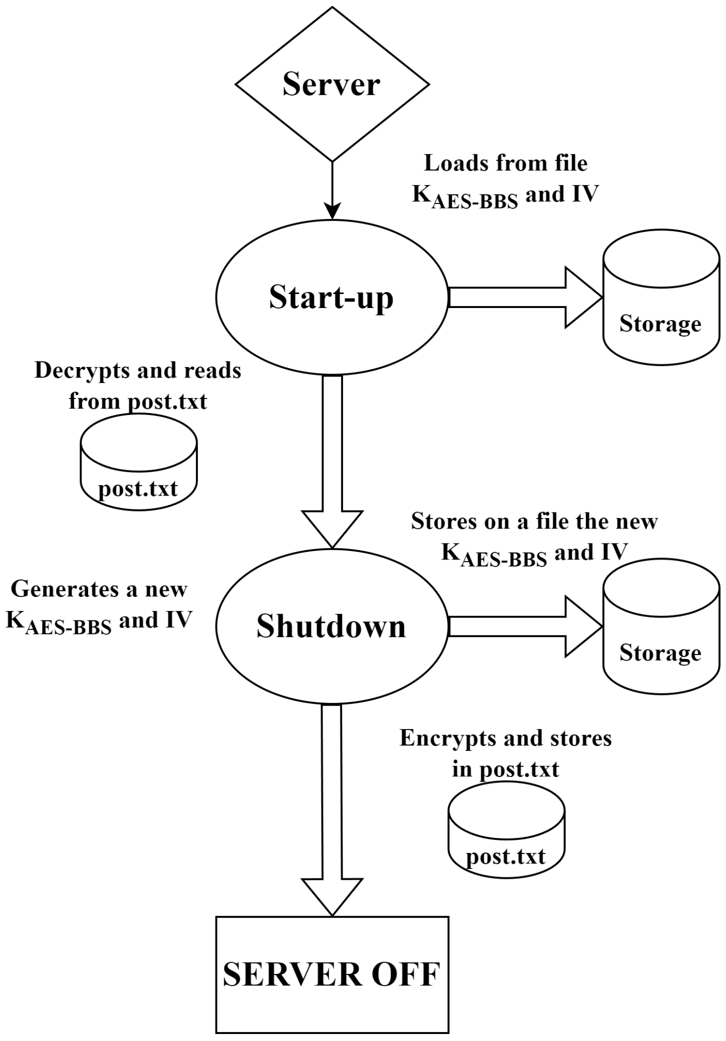
Server
Loads from file
K_AES-BBS and IV
Start-up
Storage
Decrypts and reads from post.txt
post.txt
Stores on a file the new K_AES-BBS and IV
Generates a new K_AES-BBS and IV
Shutdown
Storage
Encrypts and stores in post.txt
post.txt
SERVER OFF

Figure 2.4: Schema of post encryption and decryption.