UNIVERSITY OF PISA

MSc in COMPUTER ENGINEERING

Foundations of Cybersecurity

Bulletin Board System

Professor:

Prof. Gianluca Dini

Student:

Carlo Pio Pace

ACADEMIC YEAR 2024/2025

Contents

1 System overview

The proposed system is a modern implementation of a Bulletin Board System (**BBS**), a distributed service engineered to foster community engagement through focused, public discussions. Drawing inspiration from classic BBS architecture while incorporating contemporary security standards, this platform provides a dedicated space for users to post messages and participate in conversations, similar in function to modern internet forums. The core user functionalities are:

- **Post a Message**: Users can create new topics or reply to existing ones, contributing their thoughts and content to the public discussion boards.
- **List Messages**: The system provides an intuitive interface to list all available messages, allowing users to easily browse the posted messages.
- **Read a Message**: Users can select any message from the list to view its full content.

The afore mentioned functionalities are provided only to authenticated user, that already successfully performed a login in the system. The essential account and session management functions include:

- **Sign-up**: A simple and secure registration process allows new individuals to create an account and join the community.
- **Login/Logout**: Registered users can log in to access the system's features and securely log out to end their session, protecting their account from unauthorized access

2 Tools

The system has been developed using the following tools:

- Oracle VM Virtual Box (7.0.14).
- Kali linux (2024.1).
- Redis Database (7.0.15), high-performance, in-memory data structure store, Redis serves as primary database. Its speed is ideal for the real-time nature of a bulletin board system, handling message posts and retrievals with minimal latency.
- hiredis library (1.2.0), It is a minimalistic yet powerful C client that enables efficient and stable communication with the Redis server.
- OpenSSL (3.3.1), robust, industry-standard toolkit ensures that all data in transit is protected against eavesdropping and tampering. All cryptographic operations, including the establishment of secure, encrypted connections, are handled using OpenSSL.

3 Design

This section begins by introducing the system requirements, then defining the core elements of the Bulletin Board System (BBS), including its primary components, involved actors, underlying assumptions, and technical requirements. The discussion then proceeds to the mechanism for establishing a secure client-server connection via our custom-developed protocol.

3.1 Requirements

The requirements for the system are:

- Never store or transmit passwords in the clear
- Confidentiality, integrity, No-replay and Non-malleability in communications
- Perfect forward secrecy

3.2 Server

The system utilizes a centralized server to manage user authentication and data persistence. Clients initiate connections to the server through a predefined and known combination of IP address and port number. The server possesses an RSA key pair, and its public key ($pubK_{RSA,server}$) the key assumption is that this key is already known to all the clients. This architecture simplifies the secure channel establishment by removing the requirement for a Certification Authority (CA) to validate the server's identity.

3.3 Client

Users interact with the BBS through a client application. Upon launch, the application immediately establishes a secure connection to the server. The user is then presented with three options: sign up for a new account, log in to an existing one or close the application. The sign-up process requires a nickname, email address, and password. To finalize the registration, the user must then verify their account with a One-Time Password (OTP) sent to their email address. Once the account is created, they can log in using their chosen nickname and password. After successfully logging in, the user can browse message titles, read full messages, and post new messages. They can also log out to return to the main screen or close the application entirely.

3.4 Message

Each message is a tuple containing four distinct fields: a unique identifier (message ID or **MID**), an author (the user's nickname), a title, and a body.

3.5 Secure connection

The system uses TCP for reliable packet transmission, but this does not provide security. To protect the connection, a custom security protocol is layered on top, using RSA, the Diffie-Hellman exchange, digital signatures, and Authenticated Encryption. The secure connection is established through a five-phase handshake:

1. **Client Key Exchange** (Phase 1): The client initiates the handshake by sending its public RSA key ($pubK_{RSA,Client}$) to the server. This key will be used by the server to verify the client's identity in a later phase.

2. **Server Authentication & Key Exchange** (Phase 2): The server generates a Diffie-Hellman (DH) key pair. It then sends its DH public key ($pubK_{DH,Server}$) to the client, along with a digital signature of that key. This signature is created using the server's private RSA key ($prvK_{RSA,Server}$). The client receives the data, uses the server's known public key ($pubK_{RSA,Server}$) to verify the signature, and thus authenticates the server.

3. **Client Authentication & Key Exchange** (Phase 3): The client generates its own DH key pair. It sends its DH public key ($pubK_{DH,Client}$) to the server, signed with its private RSA key ($prvK_{RSA,Client}$). The server uses the client's public RSA key (received in Phase 1) to verify this signature, thereby authenticating the client.

4. **Shared Secret & Session Keys Derivation** (Phase 4): With a mutually authenticated DH exchange complete, both the client and server can now independently compute an identical shared secret. From this secret, two session keys are derived using the HMAC algorithm.

5. **Secure channel check** (Phase 5): To verify that the secure channel is working, the client performs a challenge-response test. It generates a random nonce, encrypts it, and sends it to the server. The server decrypts the nonce, immediately re-encrypts it, and sends it back. If the nonce received by the client matches the one it originally sent, the secure connection is confirmed and ready for use.

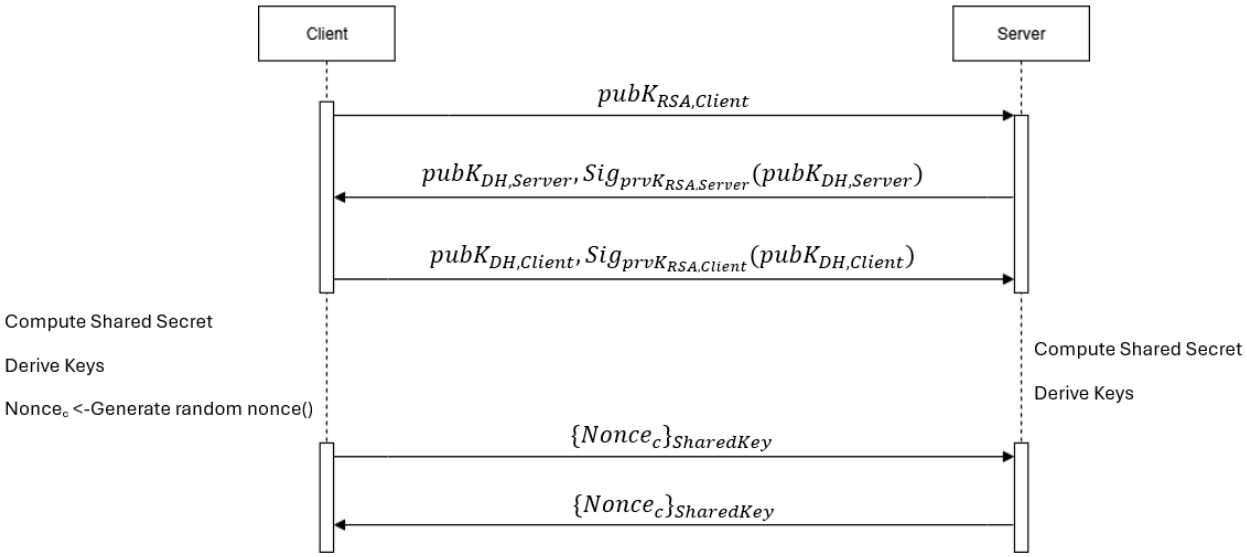

Image 1: Custom protocol sequence diagram

3.6 Secure Communications

All client-server communication is secured using **Authenticated Encryption** (AE), a method that simultaneously guarantees data confidentiality and authenticity. AE is a form of encryption that ensures that messages are not only private (confidentiality) but also that they come from the stated source and have not been tampered with (authenticity and integrity). The process involves these key elements:

- **Key**, A secret key shared between the client and server.
- **Plaintext** , the original, readable message.
- **Additional Authenticated Data** (AAD): Optional information (like packet headers or metadata) that we want to protect from tampering but does not need to be encrypted.
- **Ciphertext**: The encrypted, unreadable version of the plaintext.
- **Authentication Tag**: A cryptographic checksum generated during the encryption process. The receiver uses this tag to verify the authenticity and integrity of both the ciphertext and the AAD.

The operation can be summarized as follows:

$$AE(Key, AAD, Plaintext) = (Ciphertext, Tag)$$

During the secure connection handshake (Phase 5), non-secret but critical data like the Initialization Vector (IV) is included in the AAD. While the IV doesn't need to be encrypted, placing it in the AAD ensures that the final authentication tag protects it from being modified in transit. This prevents a variety of potential attacks.

4 Implementation

This section outlines the implementation of the BBS system, detailing its architecture, modular components, and the flow of client-server operations. Additionally, it describes the packet structure used for secure communication between the client and server, emphasizing the secure exchange of messages

4.1 System Modules

The system's architecture is organized into several distinct modules, each handling a specific set of functions. The core application logic:

- Server.cpp: This module implements the main multi-threaded server-side logic, responsible for managing client connections, processing requests, and maintaining application state
- Client.cpp : Contains the core logic for the client application, including handling user input and interacting with the server.

Data models:

- User (User.h, User.cpp): Defines the User class, which encapsulates all data and methods related to a user's profile and state.
- Message (message.h, message.cpp): Defines the Message class, representing the data structure and associated operations for all messages within the system.

Utilities:

- Communications.h: This file specifies the client-server communication protocol. It defines the structure of all exchanged messages and contains the logic for serializing and processing them.
- Query.h: Manages all interactions with the Redis database, providing a centralized location for all data storage and retrieval queries.
- CryptoUtils.cpp: A dedicated utility module that centralizes all cryptographic operations. It handles the secure connection establishment, as well as the encryption and decryption of all communications.

4.2 Secure connection

The custom protocol is implemented with strong, standardized cryptographic parameters and includes a final verification step to ensure the integrity of the secure channel. The protocol's security relies on the following industry-standard cryptographic components:

- **RSA Keys**: The handshake process uses a 2048-bit RSA key pair for authenticating the parties via digital signatures.
- **Diffie-Hellman** (DH) Parameters: To ensure a secure and efficient key exchange, the protocol uses pre-computed and vetted "safe" prime groups provided by OpenSSL, as recommended in standards like RFC 5114 and RFC 7919. The specific parameters are :
  - **Modulus Prime Length** : 2048 bits
  - **Subgroup Generator Size** : 256 bits

After the shared keys are computed, a final challenge-response exchange is performed to confirm that both parties can correctly encrypt and decrypt data. This confirms the secure channel is operational. The client generates a random nonce (a single-use number). It encrypts this nonce using the newly derived session keys and Authenticated Encryption. A separate, unique nonce is used as an Initialization Vector (IV) and included in the Additional Authenticated Data (AAD). The server receives the ciphertext, decrypts it, and extracts the original nonce. It then re-encrypts the same nonce and sends it back to the client, again using a fresh IV for its own response. The secure connection is considered successfully established only when the client decrypts the server's message and verifies that the returned nonce perfectly matches the one it sent in the initial challenge. This confirms that both ends of the channel are secure and synchronized.

4.3 Communication sequence numbers

To prevent replay attacks, the client and server use synchronized counters, for all communication. Each message is tagged with an incremental number, and any message received with an old or out-of-order number is automatically discarded. After establishing a secure connection, both the client and server employ a sequencing mechanism to ensure the integrity and timeliness of every message exchanged. This system is specifically designed to thwart replay attacks. Upon successful connection, both parties initialize two unsigned integer counters to zero.

These are called **Communication Sequence Numbers**:

- *Client Sequence Number*: Tracks messages originating from the client
- *Server Sequence Number*: Tracks messages originating from the server.

The process is synchronized. When the client sends a message, it increments its Client Sequence Number and attaches this value to the message. When the server receives the message, it verifies the sequence number and then increments its own copy of the Client Sequence Number. This exact same mechanism is mirrored for communication flowing from the server to the client using the Server Sequence Number. This synchronized counting guarantees that each message has a unique and predictable sequence number. For example, if the server has just processed a message with Client Sequence Number 6, it will expect the next message to have the number 7. An attacker attempting a replay attack would capture and resend an old message (e.g., the one with sequence number 6). The server, expecting 7, would immediately identify the sequence number as invalid and discard the message. This ensures that an attacker cannot reuse old, intercepted messages to maliciously repeat any action.

4.4 Keys derivation

Two distinct keys are used during a secure session:

- *AES KEY*, the encryption key used, for the authenticated encryption.
- *NONCE KEY*, used in combination with communication sequence number, to generate initialization vectors used for the authenticated encryption.

These key are derived using a Key Derivation Function (KDF), which take initial, potentially weak key material (in this case the master secret) and generate one or more strong, cryptographically secure keys from it. The key derivation function used is the HKDF (HMAC-based Key Derivation Function) (RFC 5869), ensuring they are cryptographically strong and independent. To guarantee that the AES KEY and NONCE KEY are computationally independent, the HKDF is provided with different context information (e.g., the text "aes-key" and "nonce-key") for each key it generates. This, combined with a cryptographic salt, ensures that the derived keys are unique and secure for their specific roles.

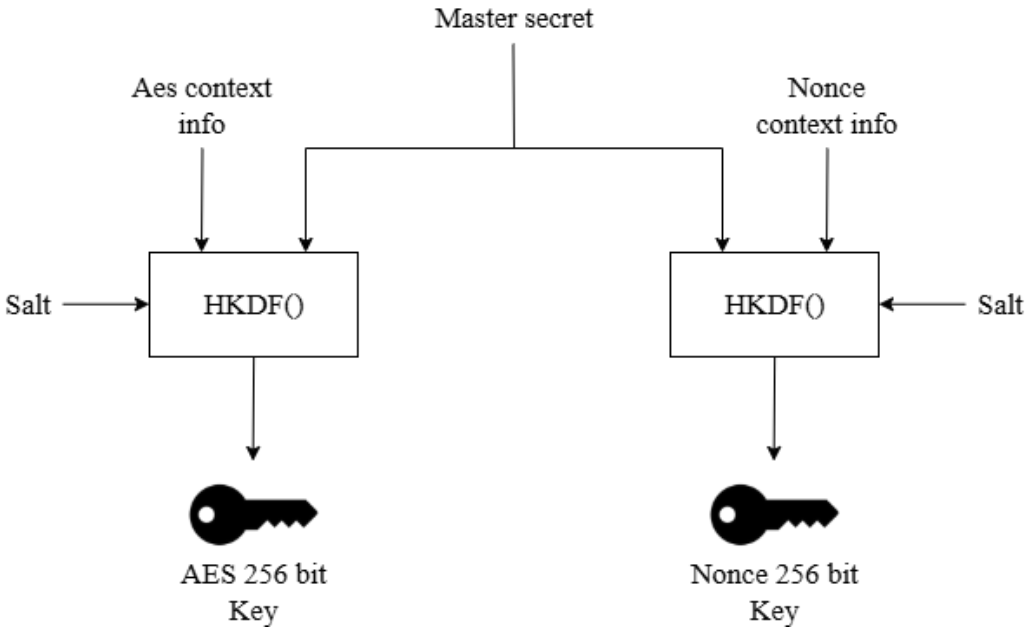

Image 2: Keys derivation mechanism

4.5 Encryption/Decryption Mechanism

All communication between the client and server is secured using AES-256 in Galois/Counter Mode (GCM). This modern cryptographic mode provides both data confidentiality (encryption) and authenticity (protection against tampering) in a single, efficient operation.

4.5.1 Initialization Vector (IV) generation

A critical requirement for AES-GCM is that every encrypted message must use a unique Initialization Vector (IV). In this protocol, the IV (also called a communication nonce) is deterministically generated for each message using the HMAC-SHA256 algorithm. The inputs are the shared *NONCE KEY* and the current Communication Sequence Number

$$HMAC_{sha256}(CommunicationSequenceNumber, NonceKey) = 32\,byte\ Digest$$

The first 12 bytes of the resulting 32-byte digest are then used as the unique 12-byte IV for the AES-GCM encryption process.

4.5.2 Encryption and Message Structure

When a message is sent, the AES-GCM algorithm takes the plaintext, the AES Key, the generated IV, and any Additional Authenticated Data (AAD) as input. The AAD contains metadata that needs to be authenticated but not encrypted, such as the IV itself. The process produces 2 outputs:

- The Ciphertext
- A 16-byte Authentication Tag

The final communication that is transmitted consists of the IV, the Ciphertext, the Authentication Tag and the communication length.

$$Communication\,length = Cipher\,length + Tag\,length + IV\,length$$

In the AAD field are included the communication length, and the IV.

4.5.3 Decryption and Verification

Upon receiving a message, the recipient performs a rigorous two-step verification:

1. **IV Verification**: First, the receiver independently computes the expected IV using its own Nonce Key and the expected sequence number. It compares this computed IV with the one received in the message. If they don't match, the packet is immediately rejected as invalid.
2. **Authentication Check**: If the IV is correct, the receiver proceeds with the AES-GCM decryption. This operation will only succeed if the Authentication Tag is valid, proving that the message has not been modified in any way since it was sent. If the tag is invalid, the message is discarded.

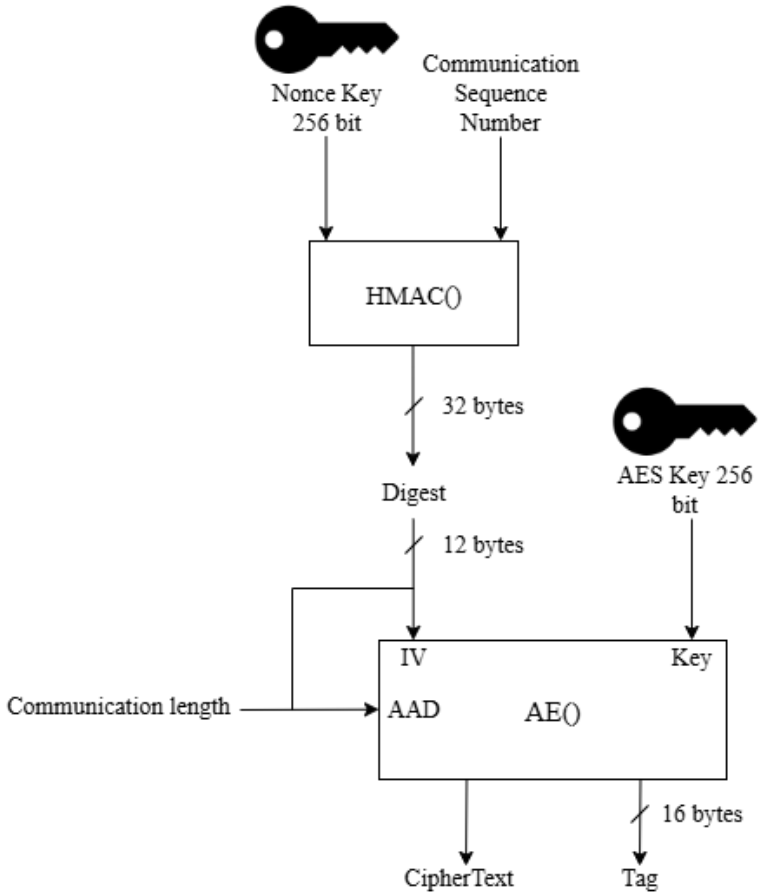

Image 3: Encryption flow

4.6 Key renegotiation

To guarantee security, the protocol ensures all cryptographic keys are "fresh" and bound to a single, temporary session. A new shared secret is generated via a full handshake every time the user starts the client application. This shared secret, and the keys derived from it, are valid for the duration of one session. A session is defined as the period from the initial secure connection until a user either successfully completes a sign-up or explicitly logs out. Upon a successful sign-up or logout, the session is terminated, and all associated cryptographic material is securely destroyed. This ensures that the keys used for registration or a previous login are completely separate from any future logged-in session. When a session ends (either through a logout or a successful sign-up), a key renegotiation must take place before the user can log in. This process is an optimized version of the initial handshake. It follows the same procedure but skips Phase 1, meaning the client does not resend its public RSA key. The server already has this key from the initial application connection, making the establishment of a new session for login more efficient.

4.7 Evaluation of the protocol

The system's design incorporates several robust security principles to satisfy its requirements and ensure the comprehensive protection of user data and communications. Regarding password protection, users passwords are never stored or transmitted in cleartext. Instead, only their cryptographic hashes are handled by the system. This critical measure ensures that even in the unlikely event of a database breach, the original passwords remain secure and cannot be recovered. For which concern communications integrity, confidentiality and non-malleability, all data exchanged between the client and server is protected by Authenticated Encryption (AE) using AES-GCM. This single, powerful mechanism provides three critical guarantees for every message, Prevents eavesdroppers from reading the content of messages, ensures that messages cannot be altered in transit without immediate detection and prevents an attacker from modifying encrypted messages to produce a different, but still valid, message. To prevent replay attacks, the protocol use communication sequence numbers, which ensure that messages can only be processed in the correct, incremental order. Any message with an old or out-of-order sequence number is automatically discarded, preventing an attacker from maliciously resending previously captured communications. Concerning perfect forward secrecy, even if an attacker were to compromise the server's long-term private keys, they could not decrypt previously recorded sessions. This is accomplished by using the Diffie-Hellman key exchange to generate unique, temporary session keys that are never transmitted. Furthermore, the protocol enforces a strict key lifecycle: session keys are discarded and renegotiated after every session (which terminates upon logout or sign-up). This practice ensures key freshness and minimizes the time window in which any single key is used, further strengthening the overall security.

5 Conclusions

This project culminated in the development of a secure and reliable Bulletin Board System (BBS), providing a robust platform for user communication. The final system effectively meets all its design goals by adhering to modern best practices in both system architecture and cryptography. The foundation of the system's security is its custom communication protocol. By implementing a Diffie-Hellman key exchange, it achieves Perfect Forward Secrecy, rendering past communications secure even if long-term keys are compromised. Furthermore, all data is protected with Authenticated Encryption, which simultaneously guarantees the confidentiality, integrity, and authenticity of every message. Architecturally, the multi-threaded server design is key to the system's performance, allowing it to handle numerous concurrent client connections efficiently without sacrificing security. The BBS was built on a robust, modular design, which is critical for its long-term viability. This approach simplifies maintenance and, more importantly, makes the system highly adaptable. Future requirements, whether they are new user features or upgrades to stronger cryptographic standards, can be integrated with minimal disruption. This ensures the platform not only meets today's security standards but is also prepared for the challenges of tomorrow.