Università di Pisa

Master Of Science in Computer Engineering

Foundations of Cybersecurity

Secure Bank Application

Project members:
Maksymilian Leder
Niccolò Mulè

Contents

1 Introduction

This document outlines the design and implementation of a cryptographically secure bank application project developed in C++. Utilizing the OpenSSL library, the project aims to provide solid security measures. The application follows a protocol for a socket-based client-server communication, ensuring the integrity, confidentiality and authenticity of data. Additionally, the application implements several security measures to protect sensitive data and prevent an array of potential attacks.

1.1 Assumptions

In order for our application to be considered secure, some assumptions are necessary:

1. Our app uses a SSL-like approach. During the execution of the handshake protocol, the user authenticates the server but not viceversa. The user is authenticated later on at application level using a password.
2. For this reason, we assume all users already underwent a registration process through an authenticated and secure channel before using the app. Doing so, the password can be used as a shared secret for authentication purposes.
3. As for certificates, we adopt a simple Centralized Trust model. We assume both server and client trust a third-party certification authority. Certificates for the server and certification authority were generated using OpenSSL command-line tool. Certificates are stored under the `/cert` directory.
4. To mitigate replay attacks, our app makes use of timestamps assuming the clocks of client's and server's host machines are loosely synchronized.

1.2 Building the app

To build our application, the user has to compile the source code using the `make` command. It creates a `/bin` directory with two folders - `Server` and `Client`. Each folder contains an executable file called `main`. We can run both server and client with command `./main`.

2 Handshake protocol

During the handshake, the server and client establish a session that can be used to encrypt and authenticate the upcoming connections. The handshake protocol is executed immediately upon client execution. The protocol we designed is based on Ephemeral RSA in order to ensure compliance with the Perfect Forward Secrecy requirement. Furthermore, our protocol also provides key confirmation.

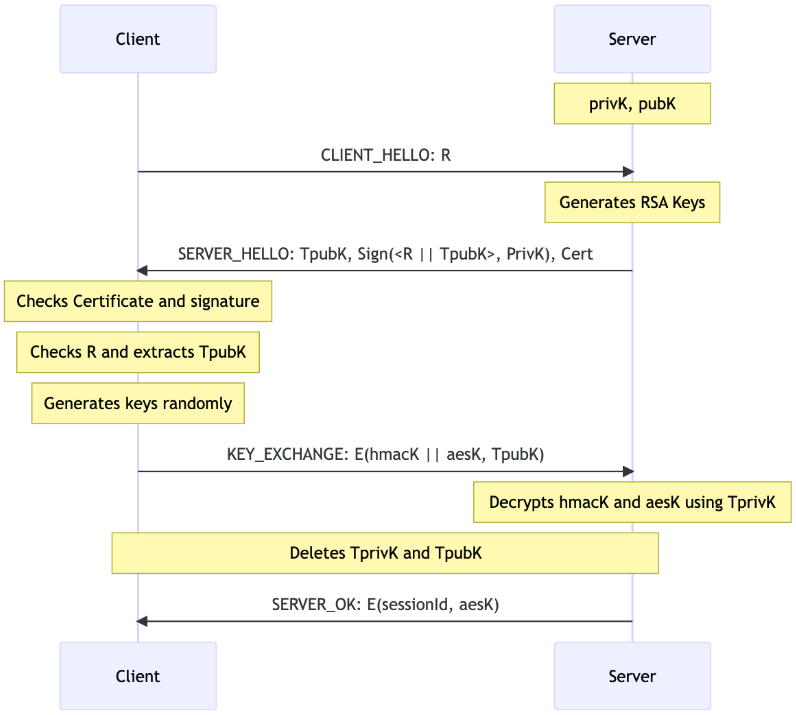
Client
Server
privK, pubK
CLIENT_HELLO: R
Generates RSA Keys
SERVER_HELLO: TpubK, Sign(<R || TpubK>, PrivK), Cert
Checks Certificate and signature
Checks R and extracts TpubK
Generates keys randomly
KEY_EXCHANGE: E(hmacK || aesK, TpubK)
Decrypts hmacK and aesK using TprivK
Deletes TprivK and TpubK
SERVER_OK: E(sessionId, aesK)
Client
Server

Figure 1: Handshake protocol

The handshake protocol can be described as follows:

1. The server maintains a pair of RSA keys: $privK, pubK$
2. Upon execution, the client initiates the handshake process by transmitting a `CLIENT_HELLO` message, which includes a random value, R.
3. Upon receiving R, the server generates a pair of ephemeral 2048-bit RSA keys. The server then replies by sending a `SERVER_HELLO` message, that contains:
   (a) $TpubK$, the newly-generated ephemeral public key.
   (b) $< R||TpubK >_{privK}$, the server's signature of R and $TpubK$ concatenated together. The signature is based on RSA and uses SHA256 as hashing algorithm.
   (c) $Cert$, the server's certificate.
4. Upon receiving the server's response, the client performs the following security operations:
   (a) Verifies the server's certificate validity using the certification authority's certificate.
   (b) Checks that the certificate actually belongs to the server by checking the owner of the certificate.
   (c) Checks for potential Man-In-The-Middle attacks by verifying the validity of the signature and checking that R is the same one as it sent in the first message.

After doing this, the client generates a pair of keys, one for MAC and one for symmetric encryption. The client then builds a `KEY_EXCHANGE` message which contains the keys encrypted with the ephemeral public key $TpubK$.

5. The server decrypts the message by using $TprivK$, thus obtaining both keys.

6. For security purposes, both client and server delete the ephemeral RSA key pair.

7. The server concludes the protocol by transmitting a `SERVER_OK` message. This message includes a *sessionId* encrypted with the AES key, thereby confirming the key to the client. The *sessionId* is a random stream of 4 bytes, stored as `uint32_t` and can be used by the client as identifier for the session for future connections.

3 Client-Server communication

In the following section, we dig into the structure of the messages exchanged within our banking application as well as the protocol we follow for application-level communication between server and clients.

3.1 Message structure

Messages are made up of 4 parts:

1. Command: contains information about type of message, the operation user wants to perform and, if it's a server response, weather the operation was successful or what type of error occurred.

2. Timestamp: This contains the timestamp created when client message was sent. Server response contains timestamp of message to which it responds.

3. Data: contains all the information for a specific command type.

4. HMAC: contains HMAC created from command, timestamp and data components. The HMAC is based on SHA256.

Before transmission, the message is encrypted using AES-128-CBC, and a header containing the sender's ID and the length of the message is appended.

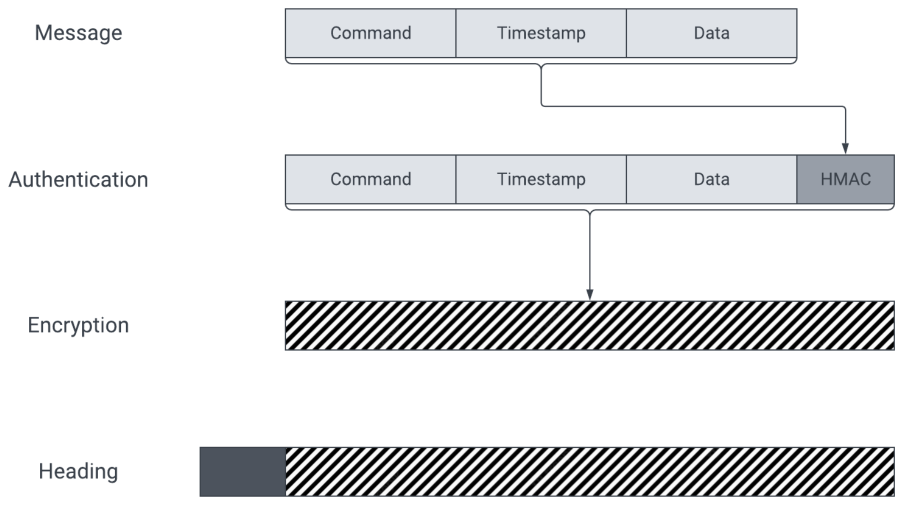

Figure 2: Message structure

3.2 Protocol

In this section, we describe the protocol used for the messages that serve the application-level functions of the app:

1. Client creates message that includes command, timestamp, data and HMAC.
2. Message is encrypted using AES-128-CBC and sent with its header.
3. Server receives the message from client and chooses the key pair according to the session ID.
4. Server decrypts the message with appropriate key.
5. Server checks validity of HMAC and correctness of the timestamp (i.e., whether the timestamp is ahead of the last received message timestamp and within the acceptable window).
6. If the verification procedure is successful, server performs the given operations.
7. The client receives the server's response, verifies the HMAC, and checks if the timestamp matches that of the client's original message.

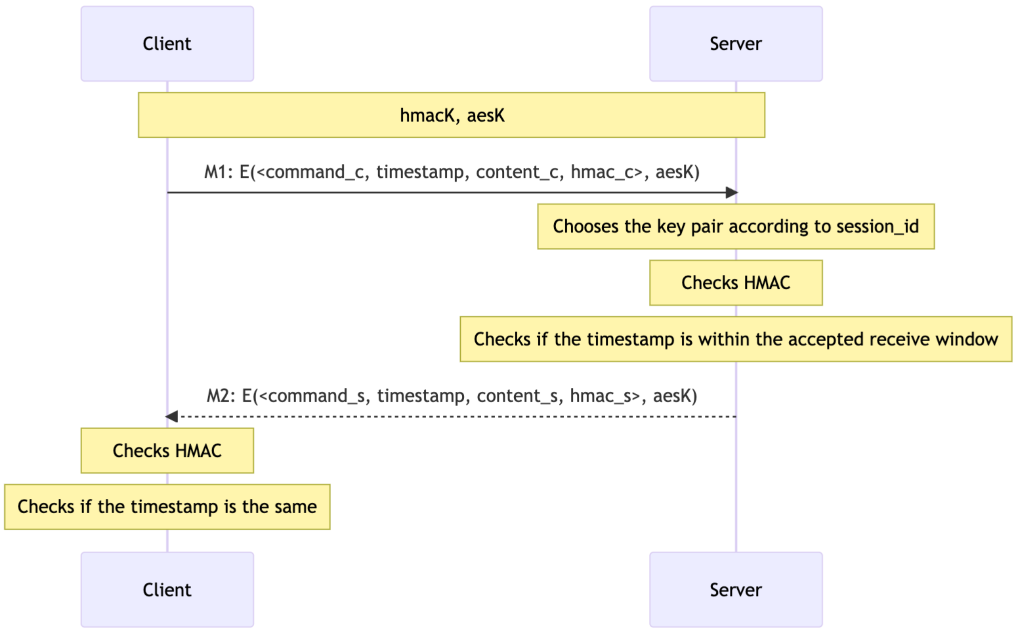
Client
Server
hmacK, aesK
M1: E(<command_c, timestamp, content_c, hmac_c>, aesK)
Chooses the key pair according to session_id
Checks HMAC
Checks if the timestamp is within the accepted receive window
M2: E(<command_s, timestamp, content_s, hmac_s>, aesK)
Checks HMAC
Checks if the timestamp is the same
Client
Server

Figure 3: Application-level communication protocol

4 Handling timeouts

The client can safely terminate the session with the server by issuing a `CLOSE` command when necessary. The server will erase the corresponding session keys and return a confirmation message to the client, which can therefore delete the session keys and exit the app. This, however, does not address scenarios where the client runs into network issues or experiences a power outage. This would cause the server to store the session keys in the memory potentially for ever. For this reason, we decided to structure our server in order to run two parallel tasks:

- Serving client requests
- Erasing expired sessions

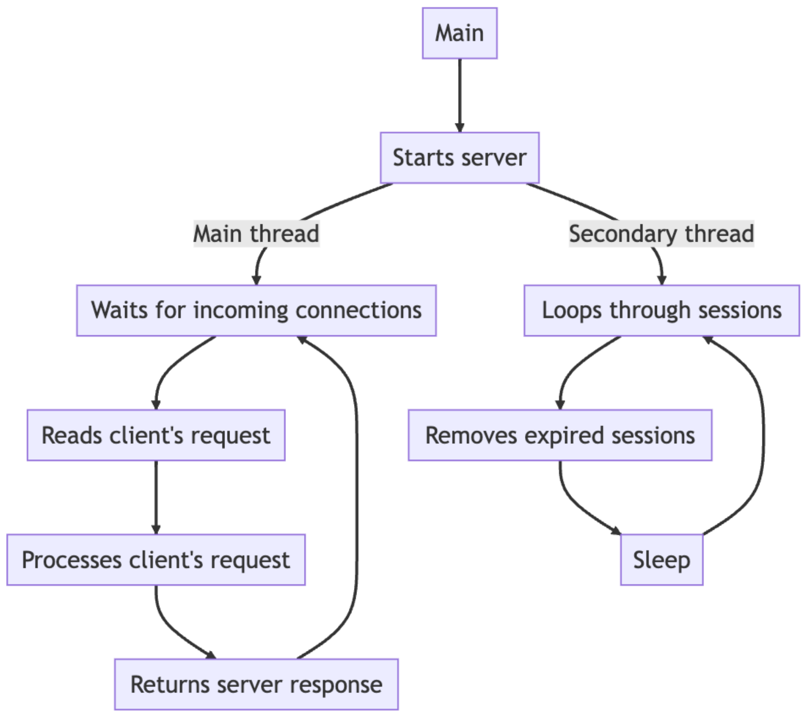

Figure 4: Threads structure

More precisely:

- For every user, the server stores the timestamp of the last received message.
- In its session loop, the server verifies if the difference between the current time and the last timestamp exceeds the timeout window. If this is true, the server deletes the session keys and erases the session. For our app, the timeout window is 5 minutes.

If the client didn't timeout but was simply idle, any new request from the client will result in an error message indicating an invalid session. The client automatically executes the handshake procedure again in order to refresh session keys.

5 Other security measures

Our application also incorporates several general security measures designed to protect sensitive data and prevent potential attacks.

- Password storage: as per project requirement, all the passwords are SHA256-hashed with a 128-bit salt, thus making offline attacks such as rainbow attack unfeasible.
- Encrypted transfer data: all the data of transfers is AES-encrypted before being stored offline.
- Timing attacks prevention: we actively prevent timing attacks on the password hashes by using functions such as `CRYPTO_memcmp`, which enable constant-time comparison of memory buffers.
- Enhanced User Privacy: in order to guarantee the best privacy, the client disables terminal echoing when the user is requested to type in the password.
- All sensitive data is promptly erased from memory using `memset` when it's no longer necessary.