Università di Pisa

Foundations of Cybersecurity
Project: Secure Bank Application

Fabio Piras, Giacomo Volpi

Academic Year: 2022/2023

Contents

Chapter 1

Introduction

The objective of the project was to design a secure bank application in C++ operated by CLI, both server and client use 127.0.0.1 as their IP, meanwhile the port on which to listen is passed as input through CLI, the client also takes the server port. For simplicity, the server and client are single process application, they both use I/O multiplexing for handling local and remote inputs. For simplicity purposes only two users are registered into the app and there is no register feature installed.

The client can perform the operations listed below

- login - perform the login to the secure bank application
- logout - log out from the application
- balance - show the balance of the user account
- transfer - perform a transfer to another user
- history - show the last bank movements
- exit - log out and end the program
- help - show the various commands

1 Code organization

To make code more modular we organized cpp files as follows. Server and Client are C++ classes that includes the following files

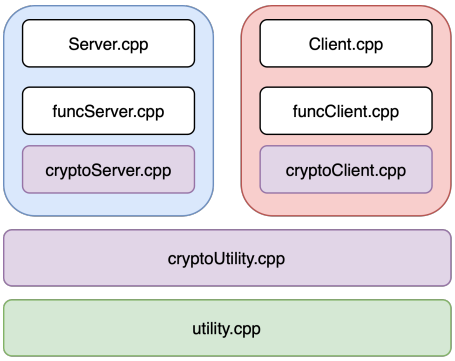

Figure 1.1: Code organization

- utility.cpp - contains utility function such as read/write binary files
- cryptoUtility.cpp - contains the most common used functions to encrypt/decrypt, for computing hash and digital signatures
- cryptoClient/cryptoServer.cpp - contain functions used in the login phase
- funcServer/funcClient.cpp - contain functions that implement all operation requested

Chapter 2

Protocol

As specified in the project guidelines the public key and private key for asymmetric encryption/decryption were already known to all parties, so we generated them with commands:

- openssl genrsa -aes128 -out rsaPrivKey.pem 3072
- openssl rsa -pubout -in rsaPrivKey.pem -out rsaPubKey.pem

For simplicity purposes the password of the private keys are the user-name for the client and "server" for the server.

1 Handshake

To achieve the perfect forward secrecy the station to station protocol was used as foundation to build upon our protocol. Since as already said the public keys were already known, there is no need for certificate in the exchanged messages, direct authentication is also achieved. Before explaining the actual protocol is important to notice that with derivation of the shared secret we summarize the generation of two keys, the first for encrypting-decrypting with AES128-CBC, the second is used for the HMAC in mode SHA256. The process is the result of the hash of the Diffie Hellman shared secret and the nonces to ensure freshness.

In regards of Fig.2.2 the first action is triggered by the user typing the login command. The first message contains the command NEW which is used by the server to differentiate from session messages to login request, the Diffie-Hellman parameter, a random nonce and the user-name. According to the station to station protocol, the message is sent in clear. The server then checks if the user exists, if it does the server continue operating regularly otherwise responds with ERR in clear.

The server creates its own Diffie Hellman parameters and nonce and then derives the shared secret. After that it sends to the client a message composed by the initialization vector IV, its Diffie-Hellman parameter and nonce and a cipher text encrypted with AES128-CBC. The encrypted text are the previous Diffie-Hellman parameters, the nonces and the IV digitally signed by the server.

Upon receiving the server answer, the client gets the various parts of the message and after deriving the shared secret, decrypts the ciphertext and then verifies the signature of the messages. If everything checks out, the client can answer with a message composed by the initialization vector IV and a ciphertext containing the password for completing the login procedure and the previous Diffie-Hellman parameters, the nonces, the IV and the password digitally signed. If something during the process result in an error, for example the signature is not valid, the client responds with ERR in clear to the server.

Finally the server can decrypt the ciphertext and verify the signature and, if the check is passed, verify the correctness of the password. If this final control is passed the server answers with a final message that close the handshake procedure composed by a sesssion id ID, the initialization vector IV, the HMAC of the ID, IV and the ciphertext and finally the ciphertext itself containing the OKLOG status command and the ID.

Upon receiving the final message the client is logged in and it can continue to perform other operations.

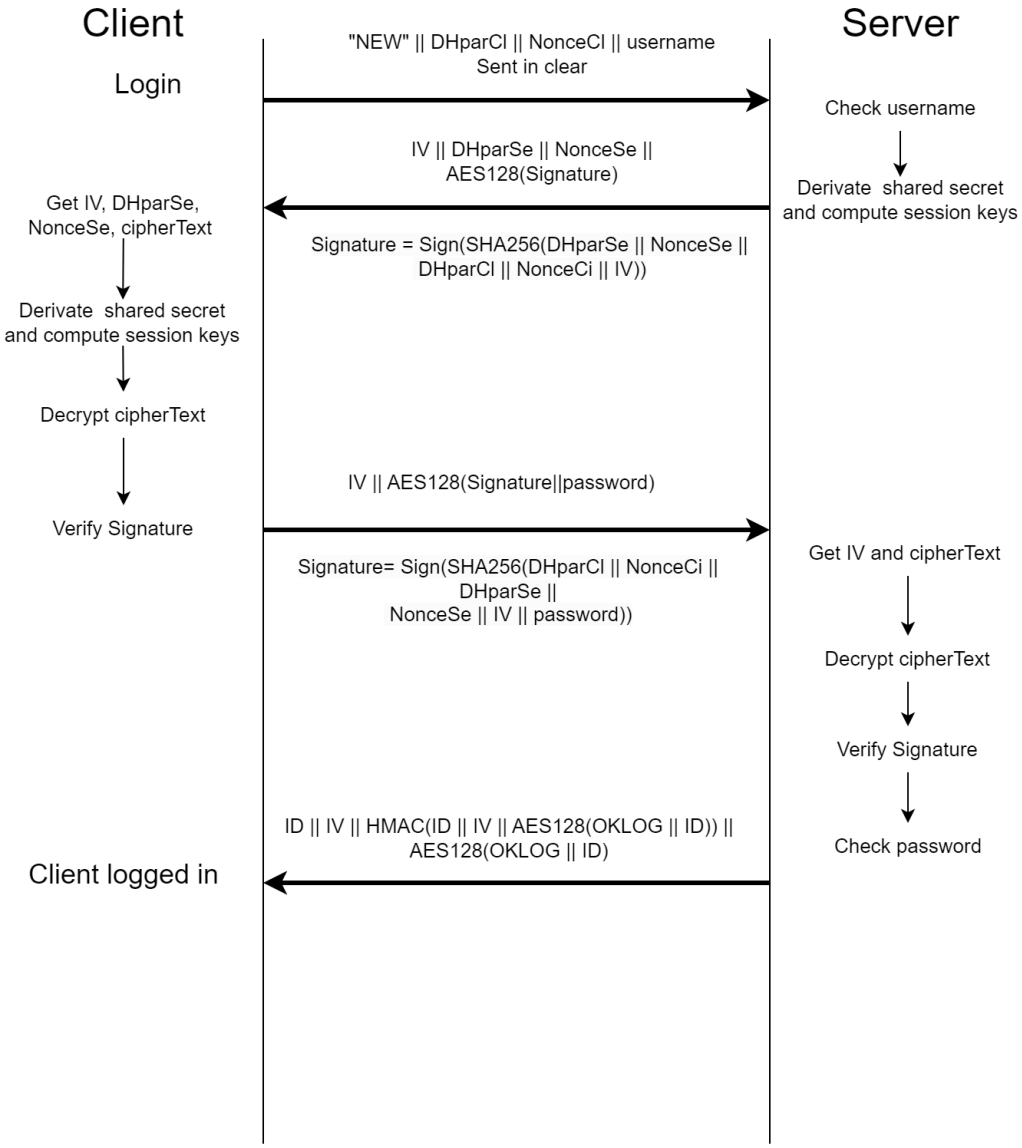

Figure 2.1: Initial handshake

2 Session message exchange

Everytime client queries the server for a request proceeds as follows.

- generates request and increments operation counter
- creates a fresh IV and encrypts the request
- generates MAC with encrypted and clear data

- allocates the sender buffer, appending : identifier, IV, MAC and ciphertext
- sends all the above elements to server

The server then does the following operations

- receives the message
- checks the TTL of the session keys
- computes the MAC and checks if it is equal to the one sent by client
- decrypts the cipher text and obtains the request
- checks the operation counter
- performs the request
- if there are no errors then it responds to client in the same manner, otherwise an encrypted error command is sent to client.

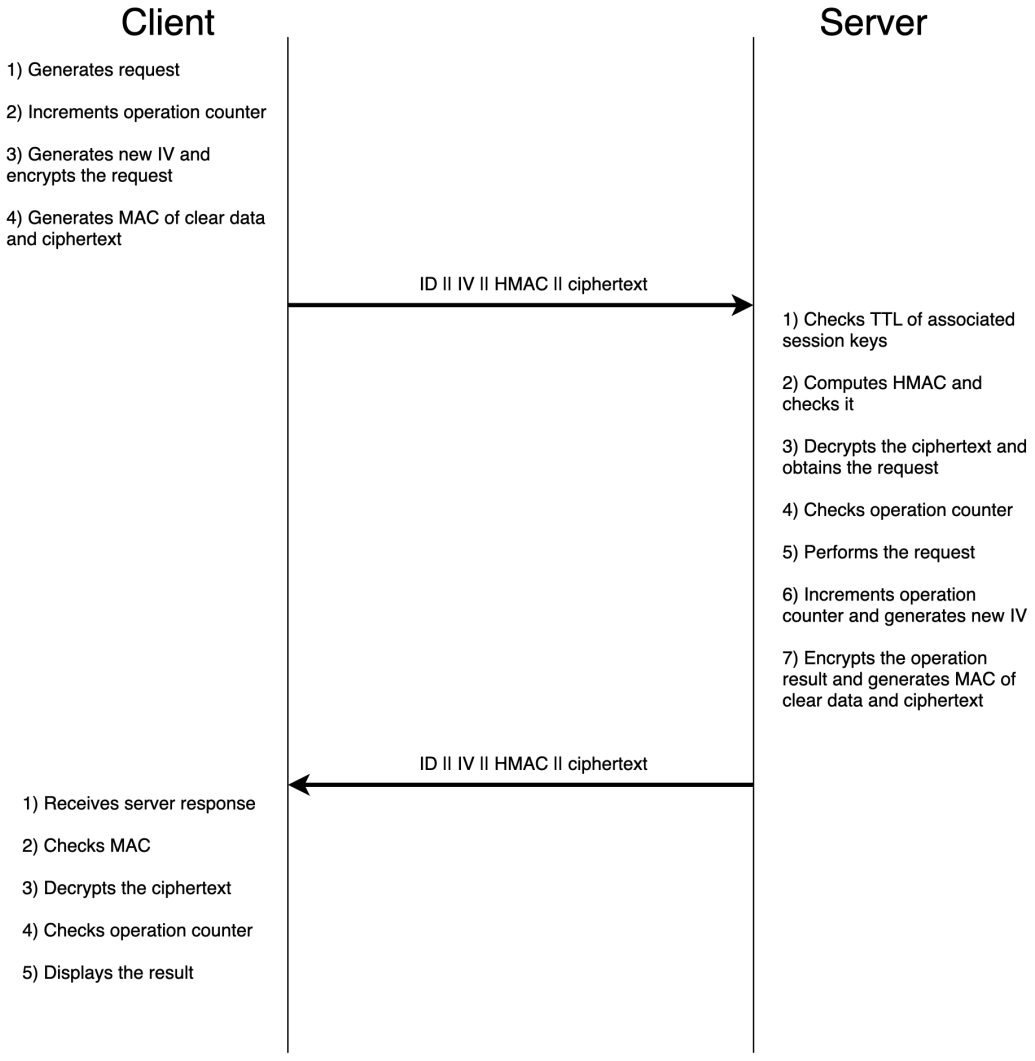

Figure 2.2: Session protocol

For the session, the Encrypt then MAC (EtM) mode was chosen since it provides integrity without compromising security.

Chapter 3

Data format

1 Message format

The message format is the same for all operation messages, once the client is authenticated, and both session and MAC keys have been established.

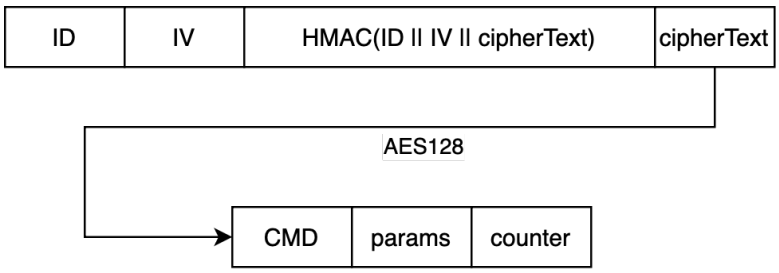

Figure 3.1: Format of the operation messages

ID is the session unique identifier, IV is the initialization vector used in encryption/decryption, then we have the MAC that protects the integrity of ID, IV and ciphertext, following the EtM scheme. Finally the ciphertext is the result of the encryption via AES-128-CBC of CMD (the command of the operation that is requested), params (parameters used in the request, for example amount of money to transfer) and a counter, that is used for preventing replay attacks.

2 Data structure

We defined the LoggedUser data structure on the server-side, it is used for handling different clients and their information at the same time. Everytime the login phase is successfully concluded, an entry is created. When the client perform an exit/logout operation the entry is removed and the keys are destroyed. In order to avoid the client to use for long period of time the same session keys, we implemented also a TTL (time to live) and checking it during the operations phase.

```
struct LoggedUser {
    string username;
    int id;
    unsigned char * session_key;
    unsigned char * HMAC_key;
    time_t session_key_generation_ts;
    int operation_counter_client;
    int operation_counter_server;
```

```
    uint16_t port;
    bool logged;
};
```

3 On file encryption

Since no cleartext files are allowed for security reason we decided to use the digital envelope to encrypt a file and store the result on the server. This method is used for password, balance and history of transfers files. We encrypt the key of the symmetric encryption with the public key of the server.