UNIVERSITÀ DI PISA

Computer Engineering

Foundations of Cybersecurity

Secure Messaging Service

Project Report

Student:
Ayoub El Ourrak

Academic Year: 2020/2021

Contents

1 — Specifications

The main goal of the activity described in this report is the following: implementing a secure messaging service for many clients with a server as intermediate.

Server:

- authenticate itself using a certificate.
- authenticate the clients after their access using their, pre-shared, public key.
- gives to the clients the public key of a client whom they want to chat.
- relays the messages from one client to another.

Client:

- authenticate itself using its public key.
- authenticate the server using a certificate.
- use these commands:
  - **!help** to get the list of commands that the client can use.
  - **!online** to get the list of the users who are online.
  - **!chat** to chat with an online user.
  - **!stop** to finish the chat with someone.
  - **!exit** to exit the service.

2 — Design choices

2.1 Implementation

The client side gets the input of the user from the terminal and communicates with the server via socket, in fact as soon as the client execution starts, it connect to a standard port of the server, which is listening, then after the authentication it's redirected to a dedicated port.
The server creates a process for each user connected and it replies to the commands sent by the client. After a constant interval of time the server process dedicated to a user reads a message from a shared incoming queue and sends it to the client, there are one incoming messages queue for each user and it allows the server to relay the messages from a user to another.

2.2 Protocols

2.2.1 Client-Server Handshake

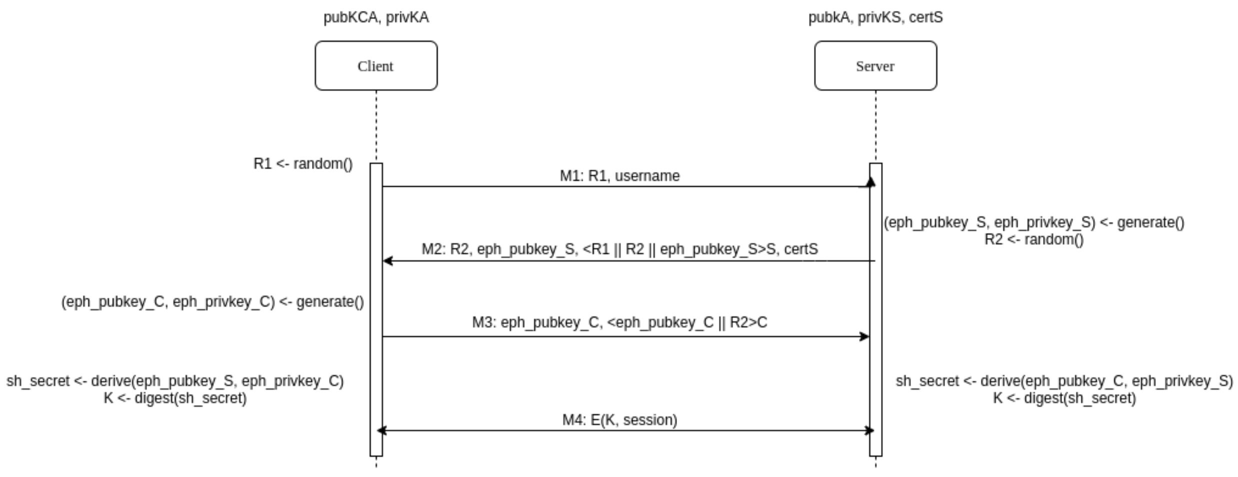
pubKCA, privKA
pubkA, privKS, certS
Client
Server
R1 <- random()
M1: R1, username
(eph_pubkey_S, eph_privkey_S) <- generate()
R2 <- random()
M2: R2, eph_pubkey_S, <R1 || R2 || eph_pubkey_S>S, certS
(eph_pubkey_C, eph_privkey_C) <- generate()
M3: eph_pubkey_C, <eph_pubkey_C || R2>C
sh_secret <- derive(eph_pubkey_S, eph_privkey_C)
K <- digest(sh_secret)
sh_secret <- derive(eph_pubkey_C, eph_privkey_S)
K <- digest(sh_secret)
M4: E(K, session)

Figure 1: Handshake Schema

The two nonces, R1 and R2, ensure protection against the man in the middle attack; the client authenticates the server using its certificate and the ephemeral Diffie-Hellman keys are used to generate the session key and then they are destroyed. This procedure ensure the Perfect Forward Secrecy for the session exchanges.

2.2.2 Client-Client Handshake

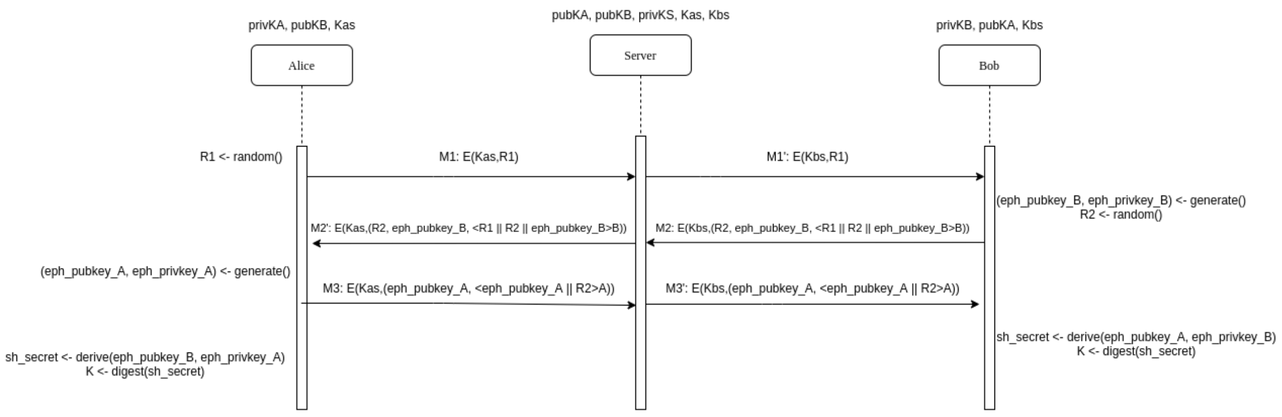
privKA, pubKB, Kas
pubKA, pubKB, privKS, Kas, Kbs
privKB, pubKA, Kbs
Alice
Server
Bob
R1 <- random()
M1: E(Kas,R1)
M1': E(Kbs,R1)
(eph_pubkey_B, eph_privkey_B) <- generate()
R2 <- random()
M2': E(Kas,(R2, eph_pubkey_B, <R1 || R2 || eph_pubkey_B>B))
M2: E(Kbs,(R2, eph_pubkey_B, <R1 || R2 || eph_pubkey_B>B))
(eph_pubkey_A, eph_privkey_A) <- generate()
M3: E(Kas,(eph_pubkey_A, <eph_pubkey_A || R2>A))
M3': E(Kbs,(eph_pubkey_A, <eph_pubkey_A || R2>A))
sh_secret <- derive(eph_pubkey_A, eph_privkey_B)
K <- digest(sh_secret)
sh_secret <- derive(eph_pubkey_B, eph_privkey_A)
K <- digest(sh_secret)

Figure 2: Handshake Schema

This handshake differs from the previous in how the clients authenticate each other, in fact they use the public key of the other provided previously by the server.

2.2.3 Client-Client Chat Request

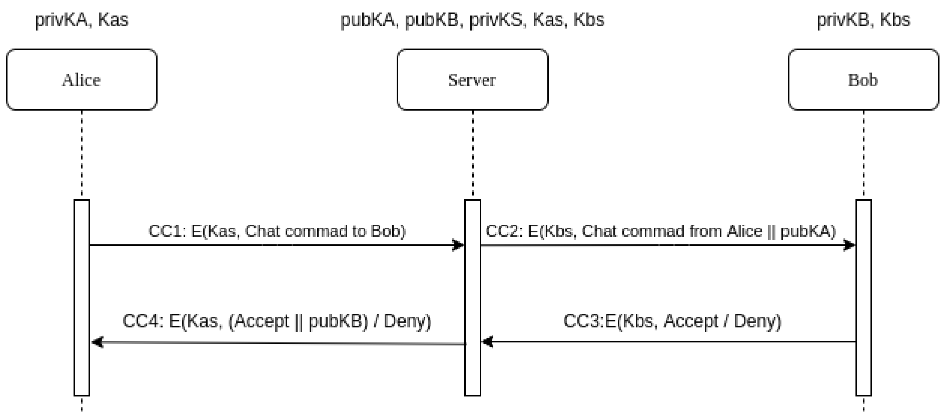
privKA, Kas
pubKA, pubKB, privKS, Kas, Kbs
privKB, Kbs
Alice
Server
Bob
CC1: E(Kas, Chat commad to Bob)
CC2: E(Kbs, Chat commad from Alice || pubKA)
CC4: E(Kas, (Accept || pubKB) / Deny)
CC3:E(Kbs, Accept / Deny)

Figure 3: Request Schema

This protocol allows the clients to have a secure exchange of public keys before the client-client handshake.

2.3 Algorithms

2.3.1 Long Term Keys

The private and public keys of the server and of the users are generated using OpenSSL from the terminal:

```
openssl genrsa -aes256 -passout file:password.txt -out
    alice_privkey.pem 2048
openssl rsa -pubout -in alice_privkey.pem -out alice_pubkey.
    pem
```

Then a certificate request is created with:

```
openssl req -new -key alice_privkey.pem -out alice_req.pem
```

Finally the program SimpleAuthority generates the certificates. The certification Authority (CA) has a self-signed certificate, which includes its public key.
The server has its private key and certificate, the public key of the users, the CA's certificate and the revocation list. Each user has its own private and public keys.

2.3.2 Short Term Keys

For the ephemeral DH keys, I've used *NID_X9_62_prime256v1* elliptic-curve parameters (256-bit curve, =128 bit security strength).
Meanwhile for the session key I've used the *SHA-256* digest of the shared secret.

2.3.3 Encryption

After the handshake I use *AES-256 GCM* authenticated encryption protocol with the session key obtained. To ensure protection against reply attack the messages are numbered with a 32 bits number, different for the communication in client-client and client-server, and after it reaches the maximum value the session is automatically closed.

3 — Messages Format

- The server header and the client header are 16 byte long, they have the same format and they are sent in clear, the rest is encrypted.
- All the lengths are integer of 4 bytes.
- The nonces are 2 bytes.

3.1 Client-Server Handshake

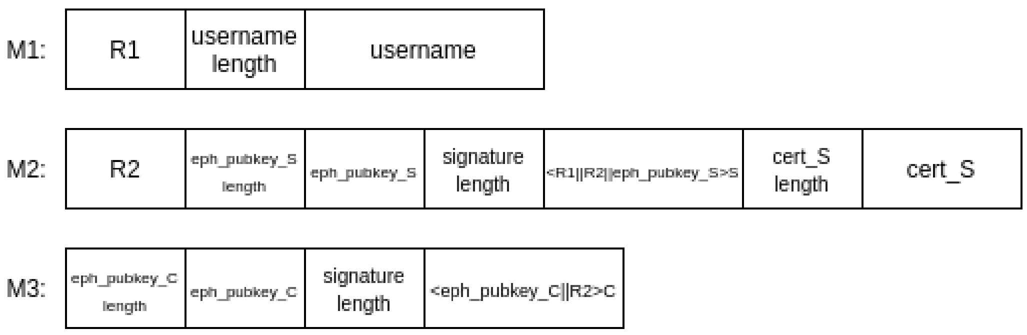

Figure 4: Handshake format

3.2 Header

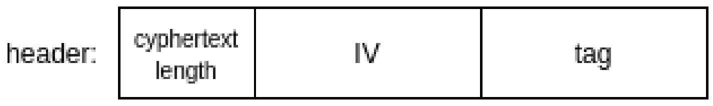

Figure 5: Header format

3.3 Commad

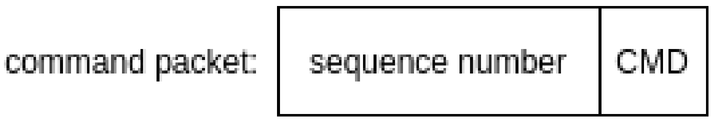

Figure 6: Command format

3.4 Client-Client Handshake

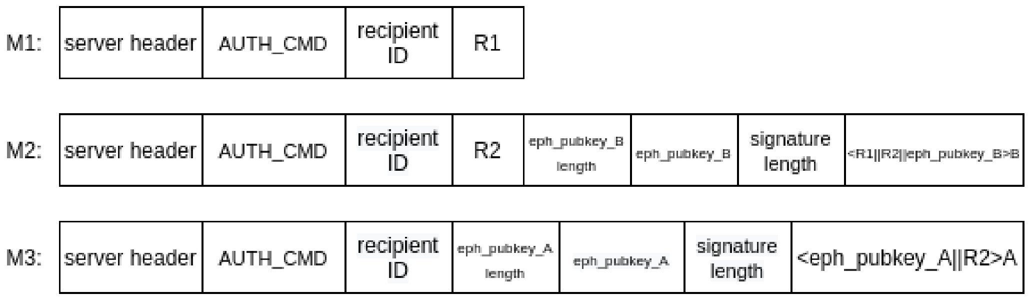

Figure 7: Handshake format

3.5 Chat Request

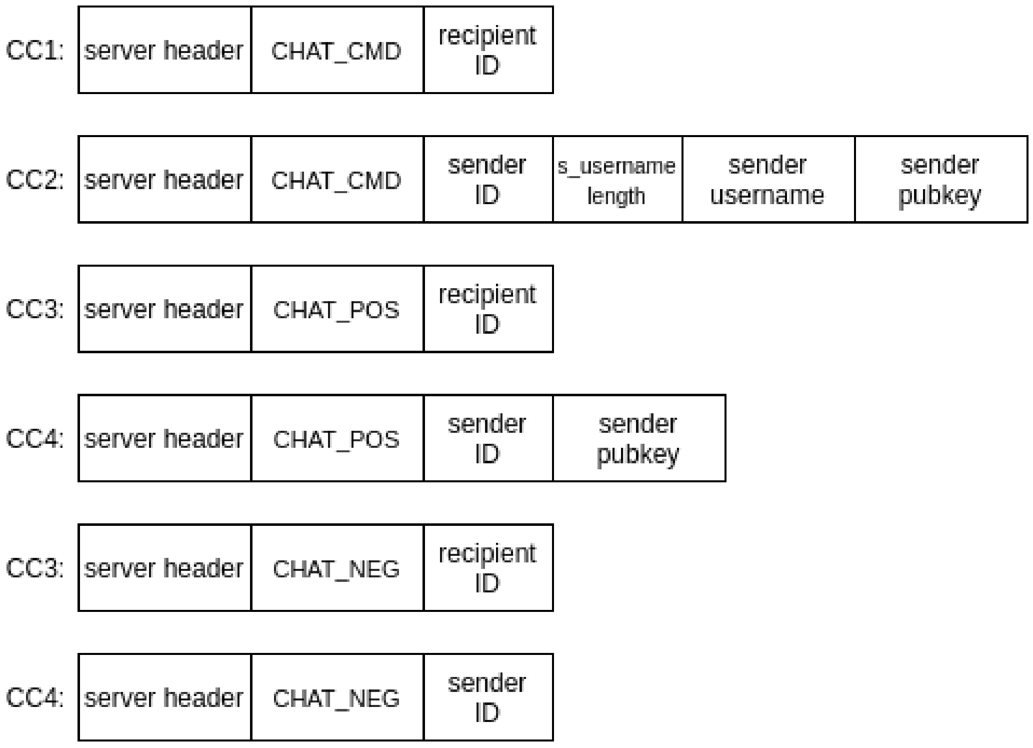

Figure 8: Request format

3.6 First message

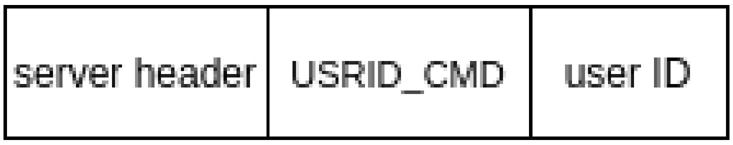

Figure 9: user ID command

3.7 Online Command

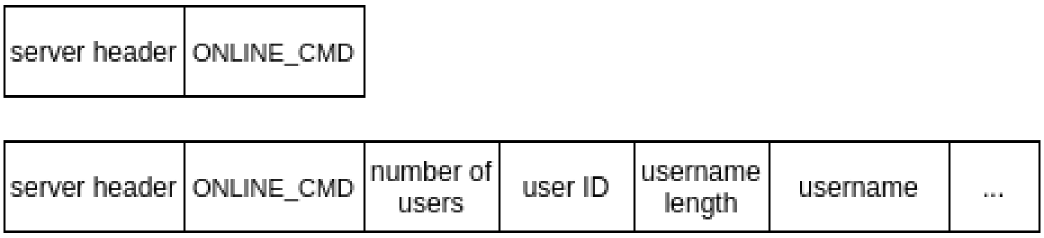

Figure 10: online format

3.8 Client-Client Messages

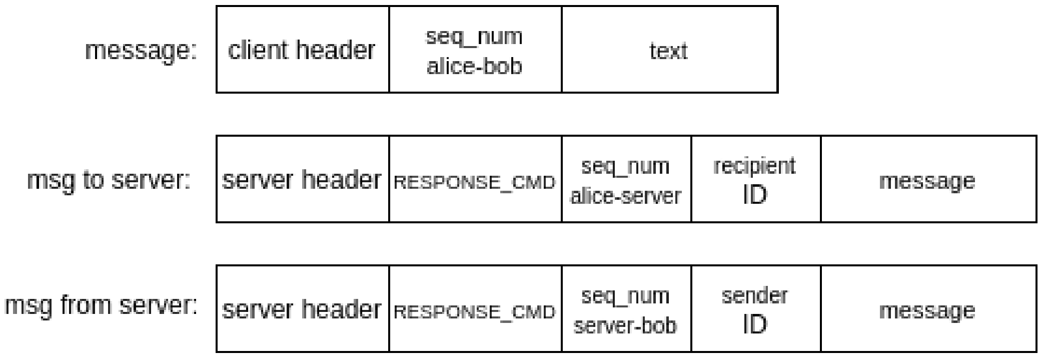

Figure 11: Chat messages

3.9 Stop Command

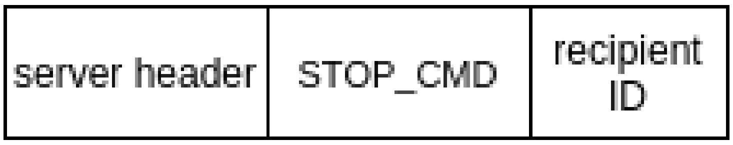

Figure 12: Stop format

3.10 Exit Command

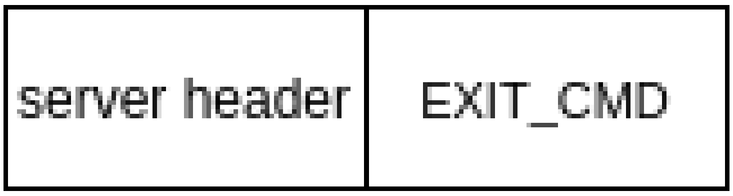

Figure 13: Exit format