Foundations Of Cybersecurity:

PROJECT REPORT

Project Group:

Riccardo Sagramoni Giovanni Menghini

INTRODUCTION

Our project is structured in two different programs: **server.exe** and **client.exe**. We strongly suggest using the provided makefiles for a correct compilation process.

Both programs require the number of the server's port as an argument when launched. Client.exe is configured at compilation time with the server's IP address. In our case, it's set to the default value of **127.0.0.1**. Both server.exe and client.exe are then required to run on the same computer for the test execution.

Even if the project has been tested on the same machine, we kept in mind that client and server could run on computers with different **endianness**, so every integer is converted from host to network format before being sent through the network.

For network communications, we chose to use the **TCP** protocol instead of UDP. TCP provides reliable, in-order and error-checked delivery of packets, which are imperative properties to guarantee the required grade of security.

THE SERVER

We have chosen to structure our server as a **multithreaded** program, using the C++11 <thread> library. In this way, we were able to serve multiple clients at the same time and we could easily share client's related data between different threads. We used the C++11 <mutex>, C++11 <condition_variable> and C++14 <shared_mutex> to achieve the thread's synchronization.

When the program is launched, the main thread initializes the fundamental data structures for its execution and start listening on the main socket for any connection request from clients. When a new request arrives, the main thread will create a socket and a thread that will handle the messages sent by the new client.

Every thread, apart from the main, serves a different client. Connection data of each client (socket descriptor, username, session key, socket's mutex...) are stored in a shared hashed map.

THE REQUEST TO TALK

In our program, the mechanism behind a "**request to talk**" is considerably complex. Let us suppose that user Alice, who is served by Thread A, sends a "request to talk" to user Bob, who is served by Thread B.

Thread A receives the request, checks if the request is valid (e.g., both parties exist and are online), sets Alice as unavailable, locks both I/O streams of her socket, locks Bob's output stream and forward the request to him.

In the base case, the input stream of Bob's socket is locked by Thread B, which is waiting for a new command from Bob. Thread A is then not able to directly receive Bob's answer to Alice's request. It must stop its execution and wait for a signal from Thread B since it will receive Bob's answer.

When Thread B receives Bob's answer, it will notify its content to Thread A. In case of a positive response, the talk is starting, so Thread B must stop its execution and cede control of Bob's socket to Thread A, which will continue the TALK protocol as described in a later chapter.

After the execution of the Diffie-Hellman protocol between Alice and Bob, Thread A creates another thread. While Thread A will forward Alice's message to Bob, its children thread will forward Bob's message to Alice. In this way, the server will not stop its execution on waiting for a new message from one of the parties. This allows both parties to send multiple messages without having to wait for an answer.

When the talk ends, Thread A will give the control of Bob's socket back to Thread B and wake it up.

THE CLIENT

Like the server, also the client is **multithreaded**. A thread_bridge structure, allocated as a member variable of class Client, provides functions for thread synchronization.

After the authentication and the key negotiation phase with the server, the master/main thread is used for the **execute_user_command** function, which executes the user's commands. The main thread will execute the control flow of the application and it will handle the output stream of the socket.

A slave thread, on the other hand, will control the input stream of the socket. It will cyclically wait for new messages from the server and notify them to the master thread according to the following algorithm:

- If the message is a request to talk, the thread will check its validity, get the name of the user who sent the request and notify a new "*request to talk*" through the thread_bridge structure.
- If the message is not a "*request to talk*", it will be stored into the thread_bridge structure and then notified to the master thread.

COMMAND SHOW

The show function is used to print out the list of names of all online clients. The client will send a **TYPE_SHOW** message to the server, which will answer with the serialize list of usernames.

COMMAND EXIT

Exit is used to stop the connection with the server and stop the application. The client program will send a **TYPE_EXIT** message to the server, will shut the socket down and will terminate its execution.

THE REQUEST TO TALK

When the client gets a new "*request to talk*", the program will ask the user if they want to accept or refuse the request and it will notify the response to the server.

In the case of a positive answer from the user, a symmetric client-to-client key will be negotiated and then the chat will start.

To be able to both send and receive messages at the same time, the main thread will create a child thread. The main thread will wait for messages from the standard input and it will send them through the socket. On the other hand, the child thread will wait for new packets received by the "input slave thread" (as previously described) and it will print them on the standard output.

If the user inserts the **!exit** command, the "exit from talk" protocol will be executed, as described in the following chapter.

IMPLEMENTED PROTOCOLS

In the following protocols, a "packet" is formed by two different messages sent by a party to another:

1. The length of the second message (4 bytes)
2. The actual packet

PROTOCOL FOR AUTHENTICATION AND KEY ESTABLISHMENT

In our project, we have chosen to merge the authentication and the key establishment phases in one sing phase, using the **STS protocol** (*Station-to-Station protocol*). In this way, the requirement for *Forward Secrecy* is also fulfilled.

AES 128 bit in GCM mode is used to provide authenticated encryption.

SHA 256 bit is used to hash the shared secret derived from the Diffie-Hellman protocol and for signing the messages.

CLIENT	SERVER
M1.a: client's **username** (*cleartext*) M1.b: **g^a** (*cleartext*), i.e. the client's part for the Diffie-Hellman key. *g* is public knowledge, *a* is a cryptographically secure nonce.	
	Check if username received exists. If false, close the connection. If true, generate **g^b** and calculate shared secret *k_DH*. Then hash *k*_DH and extract the first to get the **symmetric session key between client and server**. THE SERVER OWNS THE SESSION KEY M2.a: **g^b** (*cleartext*) M2.b: **IV** (*authenticated*) M2.c: **sign** of <g^b, g^a> with server's private key (*authenticated and encrypted with session key*) M2.d: **MAC** of M2.b+M2.c M2.e: server's **certificate** (*cleartext*)
Generate the session key from hashing the result of the DH protocol (like how the server did).	

Check if server's certificate is valid and it has been signed by the CA. Decrypt M2.c, check if the MAC is correct and then check if the signature is correct. If any of this fail, close the connection. Otherwise, THE SERVER IS NOW AUTHENTICATED AND THE CLIENTS OWNS THE SESSION KEY M3.a: **IV** (*authenticated*) M3.b: **sign** of <g^a, g^b> with client's private key (*authenticated and encrypted with session key*) M3.c: **MAC** of M3.a+M3.b	
	Decrypt M3.b, check MAC's correctness with session key and signature's correctness with client's public key (already installed in the server). If it fails, close the connection. Otherwise, THE CLIENT IS NOW AUTHENTICATED

CLIENT'S COMMANDS

The client can send three kinds of commands to the server:

1. **SHOW**, which makes the server send back a list of every online user who is available to talk
2. **TALK**, which makes the server start the process for establishing a secure chat between two clients.
3. **EXIT**, which makes the server delete user's session information and makes the client log out from the server.

In any case, every message (which could contain a command or not) has the following format:

4 bytes	1 byte	*Variable length (>=0 bytes)*
COUNTER	**TYPE OF MESSAGE**	**PAYLOAD**

The *counter* field is a **uint32_t** counter which is used to prevent replay attacks. There are two kinds of counters: one for messages sent by the server, one for messages sent by the client. Every time a message is sent, the corresponding counter is increased. If the counter overflows, the connection with the client will be closed.

Messages generated by client/server are encrypted and authenticated with **AES-128 GCM-mode**, using the previously established session key. Every message is sent in three different packets:

1. **IV** (*authenticated*)
2. **ENCRYPTED MESSAGE** (*authenticated*)
3. **TAG OF IV+MESSAGE**

In case of recoverable error, the server sends a message of only the type byte (SERVER_ERR). In case of unrecoverable failure, the server or the client close the socket connection.

TYPE OF MESSAGE:

- Client to Server
  - TYPE_SHOW 0x00
  - TYPE_TALK 0x01
  - TYPE_EXIT 0x02
  - ACCEPT_TALK 0x03
  - REFUSE_TALK 0x13
  - TALKING 0x04
  - END_TALK 0x05
  - CLIENT_ERROR 0xFF
- Server to Client
  - SERVER_OK 0x00
  - SERVER_ERR 0xFF
  - SERVER_REQUEST_TO_TALK 0x01
  - SERVER_END_TALK 0X02

SHOW command

1. CLIENT->SERVER

4 bytes	1 byte
COUNTER	**TYPE_SHOW**

2. SERVER->CLIENT

4 bytes	1 byte	???
COUNTER	**SERVER_OK**	List of usernames serialized as (length || username)

EXIT command

1. CLIENT->SERVER

4 bytes	1 byte
COUNTER	**TYPE_EXIT**

2. CLIENT: closes the socket connection and terminates
3. SERVER: closes the socket connection and remove client's data from the server's RAM

TALK command

PHASE 1: Client **ALICE** send a request to talk to Client **BOB**

1. Alice->Server (encrypted with Alice-Server key)

4 bytes	1 byte	4 bytes (uint32_t)	??? (null terminated string)
COUNTER	**TYPE_TALK**	Length of username	Username of the second client (*Bob*)

2. Server->Bob (encrypted with Bob-Server key)

4 bytes	1 byte	4 bytes (uint32_t)	??? (null terminated string)
COUNTER	**SERVER_REQUEST_TO_TALK**	Length of username	Username of the client who sent the request (*Alice*)

PHASE 2.a: **BOB** accepts ALICE's request

1. Bob->Server (encrypted with Bob-Server key)

4 bytes	1 byte	4 bytes (uint32_t)	??? (null terminated string)
COUNTER	**ACCEPT_TALK**	Length of username	Username of the client who sent the request (*Alice*)

2. Server->Alice (encrypted with Alice-Server key)

4 bytes	1 byte	
COUNTER	**SERVER_OK**	**BOB's pub key**

3. Server->Bob (encrypted with Bob-Server key)

4 bytes	1 byte	
COUNTER	**SERVER_OK**	**Alice's pub key**

PHASE 2.b: **BOB** refuses ALICE's request

1. Bob->Server (encrypted with Bob-Server key)

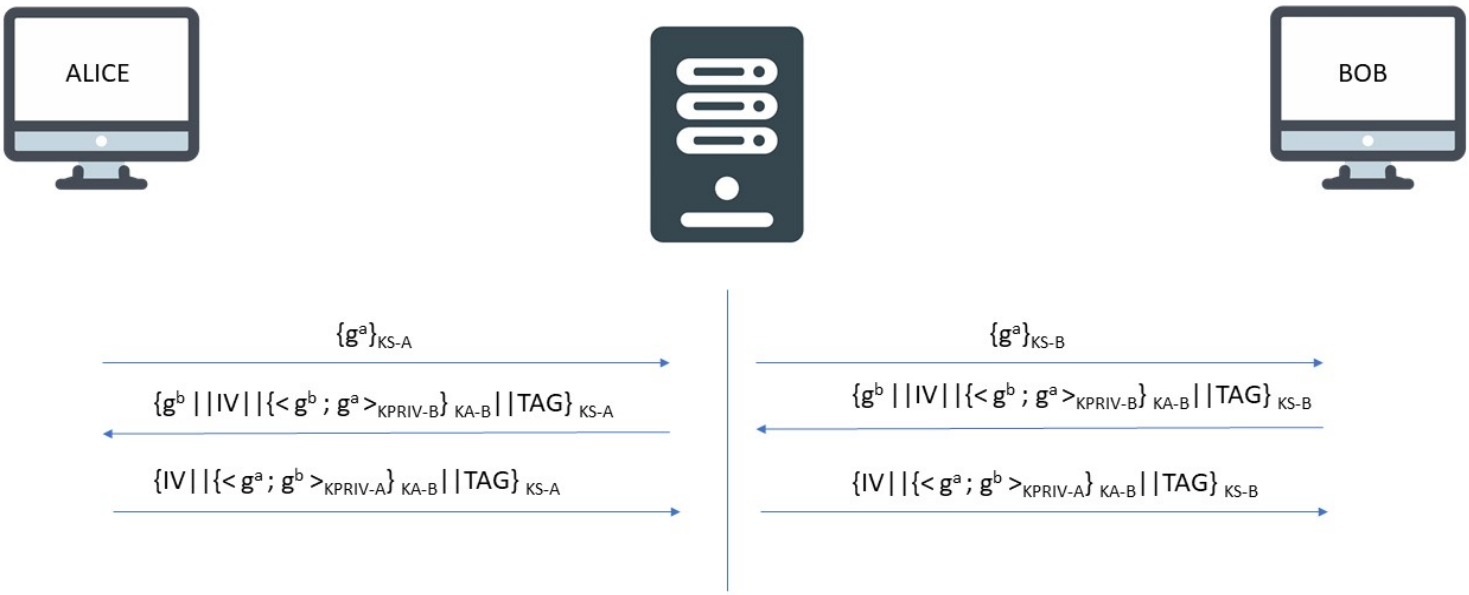
4 bytes	1 byte	4 bytes (uint32_t)	??? (null terminated string)
COUNTER	**REFUSE_TALK**	Length of username	Username of the client who sent the request (*Alice*)

2. Server->Alice (encrypted with Alice-Server key)

4 bytes	1 byte
COUNTER	**SERVER_ERR**

PHASE 3 (continues from 2.a): **ALICE** AND **BOB** NEGOTIATE A SESSION KEY

Now, **STS protocol** is used for the authentication and negotiation of shred secret key Client-to-Client:



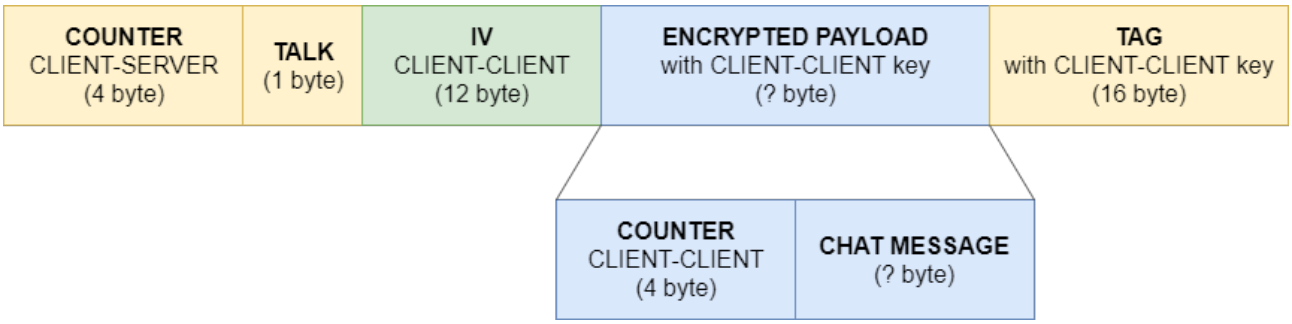
- KS-A : shared key server-alice
- KS-B : shared key server-bob
- KA-B : shared key alice-bob
- KPRIV-A : private key alice
- KPRIV-B : private key bob

1. Alice (user who has sent the request) generates $\mathbf{g^a}$, serializes it, encrypts it with Alice-Server key and sends it to Server.
2. Server decrypts the message with Alice-Server key, encrypts it with Bob-Server key and sends it to Bob.
3. Bob decrypts the message with Bob-Server key and obtains $\mathbf{g^a}$. Bob generate $\mathbf{g^b}$, serializes it and concatenate it with $\mathbf{g^a}$. Then, sign concatenated keys with its private key, encrypt it with encrypts it with Bob-Server key and sends it to Server. Bob derives the Alice-Bob session key from and $\mathbf{g^b}$.
4. Server decrypts the message with Bob-Server key, encrypts it with Alice-Server key and sends it to Alice.
5. Alice decrypts the message with Alice-Server key and obtains $\mathbf{g^b}$ and crypted signed keys. First, she can derive the Alice-Bob session key from $\mathbf{g^a}$ and $\mathbf{g^b}$. Then she can decrypt signed keys and verifying it with bobs public key. If everything goes well, Alice concatenates $\mathbf{g^a}$ and $\mathbf{g^b}$, signs them with its private key and encrypt them with alice-bob shared key, that has derived before. Then encrypt all with alice-server shared key and send it to the server.
6. Here, server decrypts the message with Alice-Server key, encrypts it with Bob-Server key and sends it to Bob.
7. Then, bob decrypt the message with server-bob key, decrypt encrypted sign with alice-bob key and verify it. If all it is ok, the chat will start.

PHASE 4: CHAT BETWEEN ALICE AND BOB

Both two parties own a shared secret session key, so now they can communicate to each other. Client-to-Client messages are first authenticated and encrypted with the Client-Client key, then authenticated and encrypted with Client-Server key. Since counter is present in all messages of the session, a replay attack is not possible.

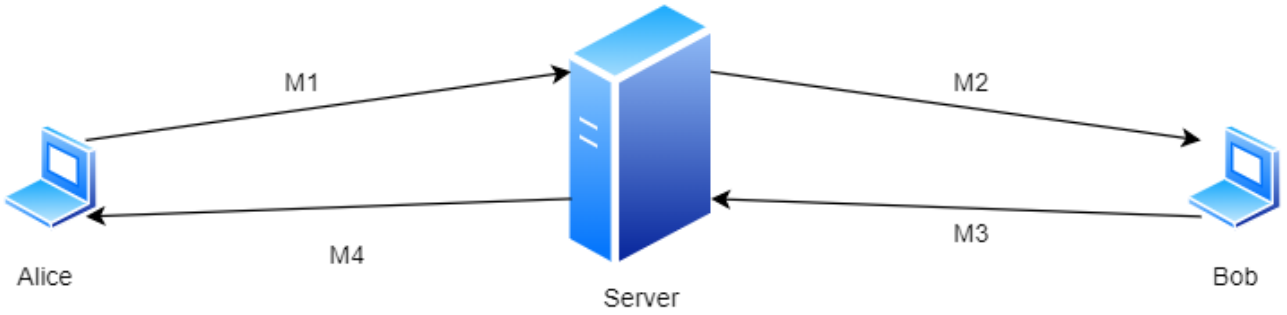
The packets are sent in the following manner:

1. **IV** for Client-Server encryption (*authenticated with Client-Server key*)
2. 

3. **TAG** generated with Client-Server key.

The clients will send the three packets every time they want to send a chat message. The server will decrypt the second packet with the Sender-Server key (after checking its integrity) and will then encrypt it again with Receiver-Server key, sending it to the Receiver.

PHASE 5: EXIT THE CHAT



M1: COUNTER (4 bytes) || END_TALK (1 byte)

M2: COUNTER (4 bytes) || SERVER_END_TALK (1 byte)

M3: COUNTER (4 bytes) || END_TALK (1 byte) →*Bob acknowledges that he is exiting the chat*

M4: COUNTER (4 bytes) || SERVER_END_TALK (1 byte) →*Alice exits the chat*

M1 and M4 are authenticated and encrypted with Alice-Server key.

M2 and M3 are authenticated and encrypted with Bob-Server key.

If any error happens during the talk, the server closes the connection with the faulty party and execute the second and third part of the exit protocol with the non-faulty party (e.g., server sends M2 to Bob if Alice fails and wait for M3 before ending the talk and making Bob available).