UNIVERSITÀ DI PISA

MSc in Computer Engineering

Foundations of Cybersecurity

Cloud Storage Project

TEAM MEMBERS:

Fabrizio Lanzillo

Federico Montini

Niko Salamini

https://github.com/FabrizioLanzillo/Cybersecurity-Project-of-Cloud-Storage

Academic Year 2020/2021

Summary

1. Introduction

Our project is developed in C++14 language for Linux systems and consists of 2 different programs, **client.exe** and **server.exe.**
We recommend the use of the commands inside the makefiles, for the correct compilation of the programs.

Both programs **need the server port number** as an argument at execution time.

The client is configured during the compilation process with the IP address of the server.
In our case the system works in localhost, so the IP address is 127.0.0.1.

In order to test the application then client and server must reside on the same machine.

Even though the project is tested on the same machine, it was designed and developed with the consideration that client and server could be on different machines and even with different **Endianness**.

Indeed, **all packets** traversing the network **were properly serialized**.

We use **TCP** instead of UDP as the protocol for communication over the network.

TCP is reliable because it ensures the reliability of packet delivery, provides also the flow and the error control features and guarantees that a packet arrives at its destination without any duplication and with the same data order.

1.1 The Client

The client provides **functions to interact with the user** and to **manage communication with the server**.

The client when executed, shows all the operations that a user can perform, through the `help()` function, and then waits for the user to enter one of the commands described below.

1.1.1 Login

The login is the **first operation** that is performed by each user, because the **user's identity must be verified** before allowing them to access the data on the server. The private key of the client, generated by simple authority tool, is locked by a password. During the login phase is unlocked if the password is correct.
After this step, the station-to-station protocol runs for the **establishment of the shared session keys** between client and server.

1.1.2 Upload

This operation **uploads a particular file on the server**.

The file is saved on the server with the same name specified by the user.

The maximum size of the file is 4 Gb and during sending, the **file is divided into chunks** of predetermined size, which in our case is 4096 Bytes.

1.1.3 Download

This operation allows to **download a file from the server**.
The server sends the file requested by the user, and then it is saved locally with the same name that is on the server.
Also here there is the **division into chunks** during the sending, as in the case of the upload operation

1.1.4 List

The list operation **lists all the user's files** that are on the server.
The list is printed on the screen.

1.1.5 Rename

This operation **renames a specific file** on the server, chosen by the user.
If the operation is not possible the file is not renamed.

1.1.6 Delete

This operation **deletes a specific file** on the server, chosen by the user.
If the operation is not possible the file is not deleted from the server.

1.1.7 Logout

In the logout, **the connection** between client and server **is closed** and all session keys are deleted.

1.2 The Server

We choose to design the server as a **multi-threaded program**.

In this way **we can handle multiple clients at the same time**.

When the program is launched, the server listens on the main socket, and waits for new connection requests from clients.

When a new request arrives, the main thread, ***server***, **creates** a socket and a **new thread**, ***worker***, that handles communications with the client and packet exchange.

1.3 Used protocols and algorithms

To encrypt the session, we used the AES-128-CBC algorithm to have resistance against brute-force attacks and to avoid traffic analysis, for every new message a new initialization vector is generated.

We used a keyed hash (HMAC) to check the authenticity and integrity of the messages in the session and a encrypted counter to avoid replay attacks.

To establish the AES and HMAC keys we have used the station-to-station protocol that guarantees the source authentication of both server and client.

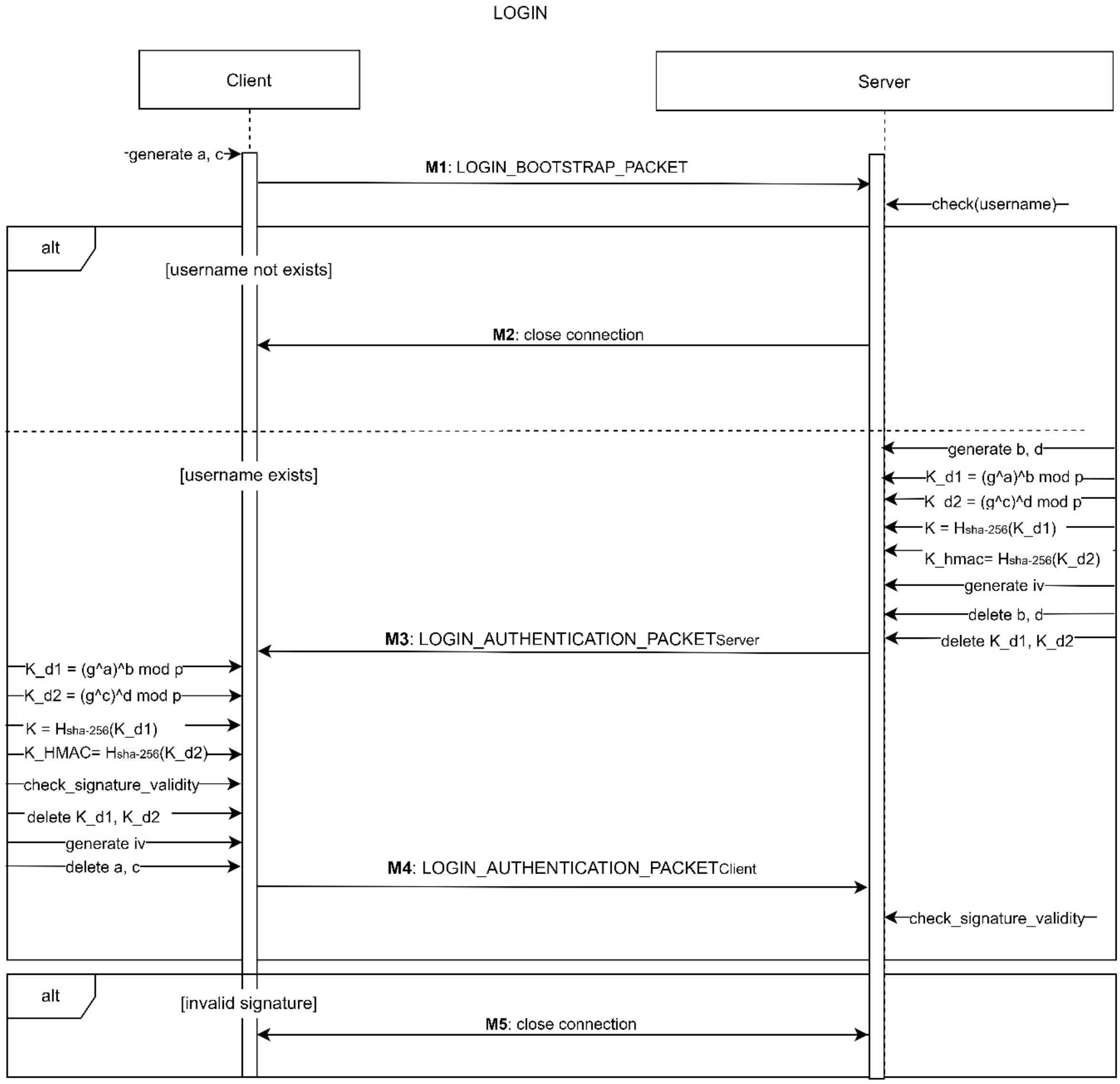
2. Implemented Protocols

From now on we consider that a packet is formed by two different messages sent one after the other from both client/server, those messages are:

- The length of the subsequent message
- The actual packet that contains the information

2.1 Protocol for Authentication and Key Establishment

LOGIN COMMAND



check_signature_validity: get ca certificate, check if server/client certificate is valid (signed by ca), use server/client public key to verify the corresponding signature. If signature is valid continue, otherwise close the connection.

M1 send a LOGIN_BOOTSTRAP_PKT type that will have the following fields:

code	**username_len**	**username**	**dh_key_len**	**dh_key_len**	**symmetric_key_dh**	**hmac_key_dh**
To differentiate packets	Explicit the length of the username	Username of the session user	Length of the serialized DH key	Length of the serialized DH key	DH to derive symmetric key	DH to derive hmac key

M2-M3 send a LOGIN_AUTHENTICATION_PKT type that will have the following fields cert, iv_cbc, g^b, g^d, encrypted_signing:

code	**cert_len**	**dh_key_len**	**dh_key_len**	**symmetric_key_dh**	**hmac_key_dh**	**Iv_cbc**	**cert**
To differentiate packets	Length of the serialized certificate	Length of the serialized DH key	Length of the serialized DH key	DH to derive symmetric key	DH to derive symmetric key	Initialization vector	certificate

Encrypted signing sent as a LOGIN_AUTHENTICATION_PKT encrypted field {<lenghts, g^a, g^b, g^c, g^d>Server}K:

dh_key_lenghts	**dh_keys**
Contains the length of the keys to be signed	Contains the dh keys to be signed

The encrypted signing will be verified on client and on server side.

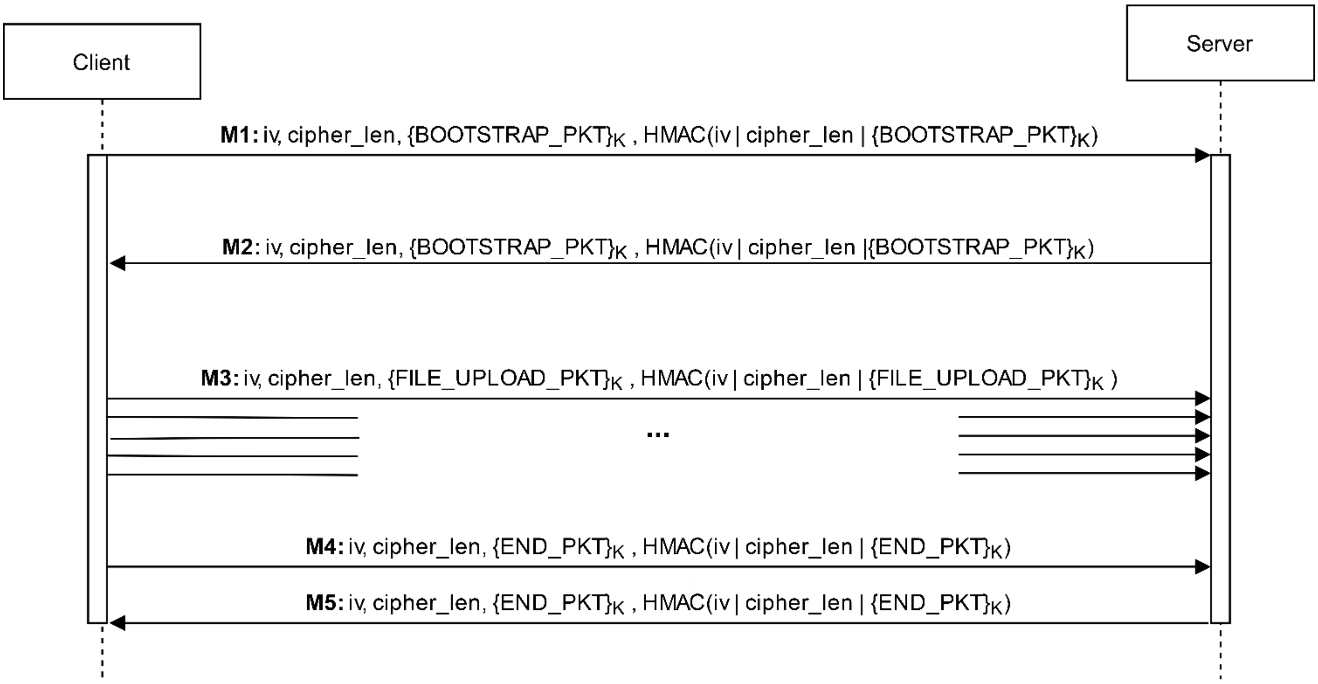
All the structure are serialized using PEM format.

2.1 Client’s Commands

The client might perform the following operations:

2.1.1 UPLOAD COMMAND

The command will follow this protocol scheme:



M1 and M2 send a BOOTSTRAP_PKT type that will have the following encrypted fields:

code	filename_len	filename	response	counter	size
Needed to differentiate between different kinds of packets	Explicit the length of the filename for the file to upload	Name of the file to upload	Contain the response of the server	Counter needed for replay attack	Size of the file that the user wants to upload

If the response is positive the client will proceed to manage the file upload to the server.

M3 is the packet type that is used to transfer the file and have the following fields:

code	counter	msg_len	msg
Needed to differentiate between different kinds of packets	Counter needed for replay attack	Length of the uploaded file chunk	Chunk of the file

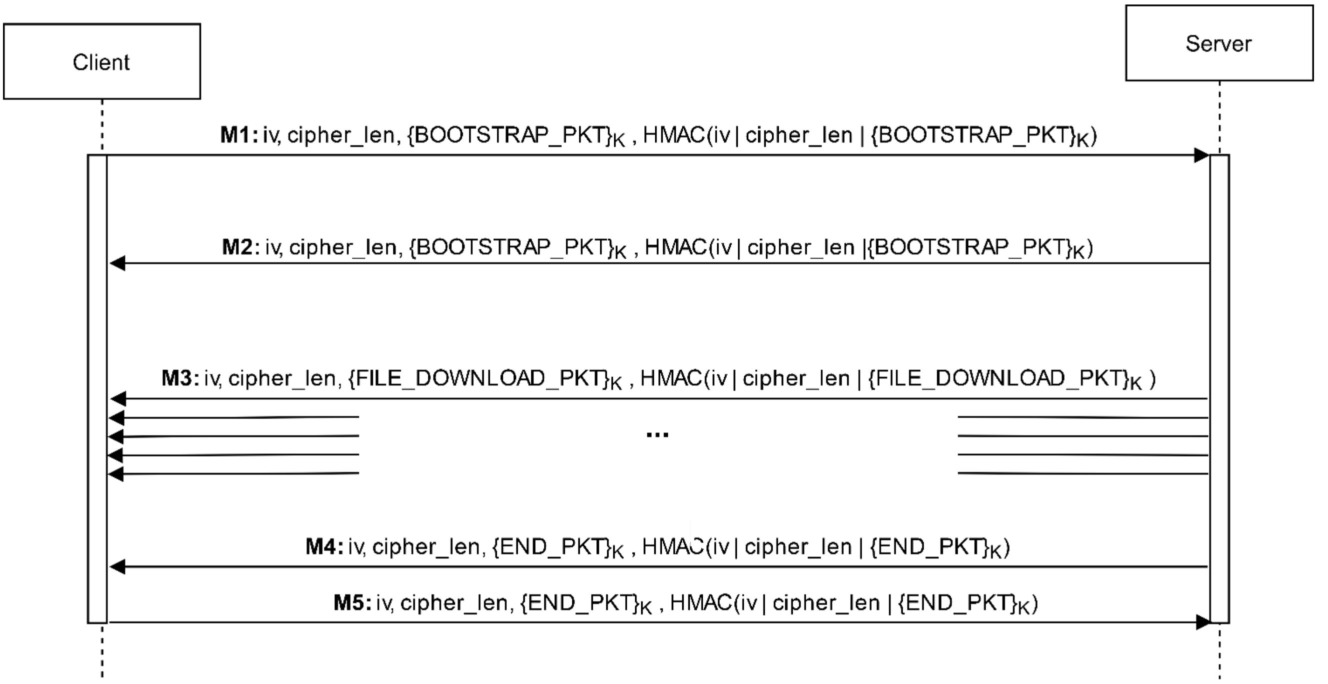
After the forwarding is completed, the client will proceed to send a packet to confirm the transfer completion.

M4 and M5 are composed of the END_PKT format used to finalize the upload with a handshake:

code	counter	response
Needed to differentiate between different kinds of packets	Counter needed for replay attack	String needed to notify if everything was correct

2.1.2 DOWNLOAD COMMAND

The command will follow this protocol scheme:



The packets used to perform this operation have the same fields of the packets used during the upload, the only difference is the code, which variate depending on the operation that the client want to perform.

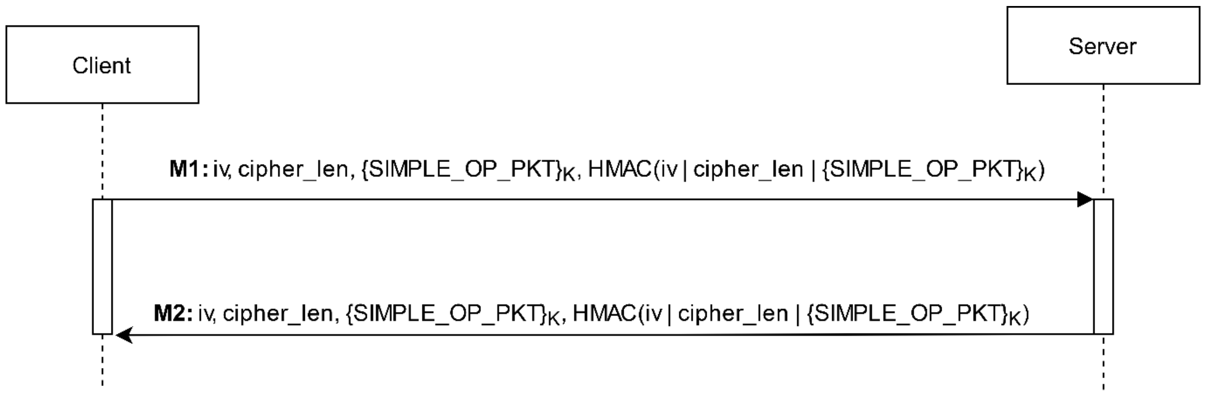
2.1.3 SIMPLE OPERATIONS

For the other simpler operations, we organized the code in order to manage all the operations with a single packet type, changing the packet code and filling the fields depending on the operation to perform. The packet has the following fields:

<table>
<tr><th>code</th><th>simple_op_code</th><th>response</th><th>filename</th></tr>
<tr><td>Needed to differentiate between different kinds of packets</td><td>Needed to recognize the simple_operation</td><td>Needed to feedback the user</td><td>Name of the file on which execute the operation</td></tr>
<tr><th>renamed_filename</th><th>Response_output</th><th colspan="2">counter</th></tr>
<tr><td>Not set if delete or list</td><td>Filled with list response if the operation is a list</td><td colspan="2">Counter needed for replay attack</td></tr>
</table>

2.1.3.1 RENAME COMMAND

The command will follow this protocol scheme:



M1:

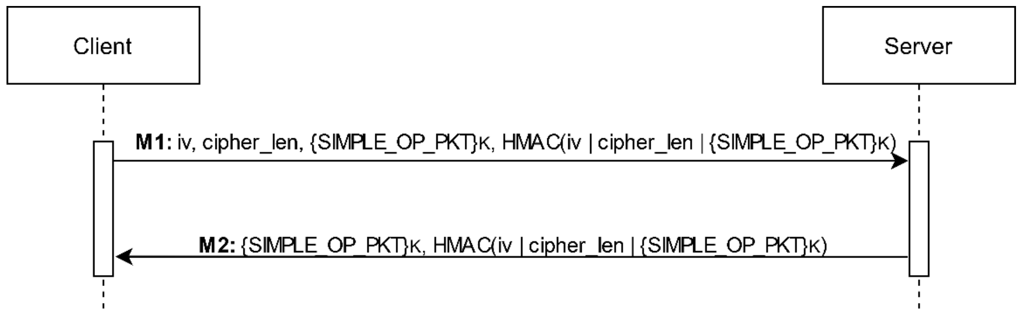
code	**simple_op_code**	**response**	**filename**
Needed to differentiate between different kinds of packets	Needed to recognize the simple_operation	Set to 0	Name of the file to rename
renamed_filename	**Response_output**	**counter**	
New file name	"- - "	Counter needed for replay attack	

M2:

code	**simple_op_code**	**response**	**filename**
Needed to differentiate between different kinds of packets	Needed to recognize the simple_operation	Set to 1 if the operation is successful	Name of the file to rename
renamed_filename	**Response_output**	**counter**	
New file name	"- - "	Counter needed for replay attack	

2.1.3.2 LIST COMMAND

The command will follow this protocol scheme:



M1:

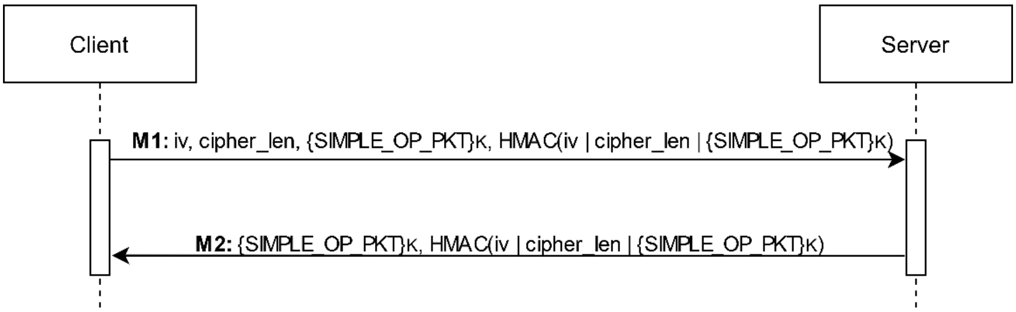
code	**simple_op_code**	**response**	**filename**
Needed to differentiate between different kinds of packets	Needed to recognize the simple_operation	Set to 0	“- -"
renamed_filename	**Response_output**	**counter**	
New file name	“- - ”	Counter needed for replay attack	

M2:

code	**simple_op_code**	**response**	**filename**
Needed to differentiate between different kinds of packets	Needed to recognize the simple_operation	Set to 0 if the operation wrong, set to 2 if the operation is correct	“- -“
renamed_filename	**Response_output**	**counter**	
“- -”	A string filled with the file present in the cloud separated from ‘\n’ char	Counter needed for replay attack	

2.1.3.3 DELETE COMMAND

The command will follow this protocol scheme:



M1:

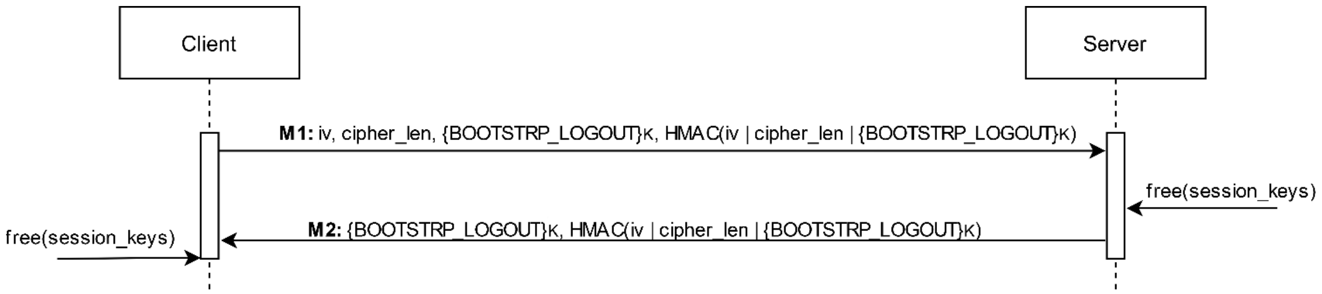
code	simple_op_code	response	filename
Needed to differentiate between different kinds of packets	Needed to recognize the simple_operation	Set to 0	Name of the file to delete
renamed_filename	**Response_output**	**counter**	
Not used	Not used	Counter needed for replay attack	

M2:

code	simple_op_code	response	filename
Needed to differentiate between different kinds of packets	Needed to recognize the simple_operation	Set to 0 if the operation wrong, set to 1 if the operation is correct	"- -"
renamed_filename	**Response_output**	**counter**	
Not used	Not used	Counter needed for replay attack	

2.1.4 LOGOUT COMMAND

The command will follow this protocol scheme:



M1-M2:

code	response	counter
Needed to differentiate between different kinds of packets	To feedback the user about the logout	Counter needed for replay attack

The response is 1 if the logout is done correctly.