Course in Foundations of Cybersecurity

Project: Cloud Storage

Alessio Vivani Marta Sanguinetti

A.Y. 2021/2022

Table of contents

1. Introduction

The project consisted in implementing a Client-Server application that resembles a Cloud Storage, where each user has a “dedicated storage” on the server, and User A cannot access User B “dedicated storage”.

Users can Upload, Download, Rename, or Delete data to/from the Cloud storage in a safe manner.

Project was developed using C++ as programming language on Ubuntu 20.04, with the library OpenSSL for all the various cryptographic APIs.

The main goal was to develop the application ensuring a secure communication protocol.

For what concerns the communication medium we have used sockets TCP.

Moreover, we have implemented the server so that it creates a process whenever it receives a new connection from a client. That process will handle all the operations requested by that client, until it logs out or an error occurs.

2. Authentication

2.1 Pre-shared crypto material

Users have already the CA certificate and a long-term RSA key-pair, where the long-term private key is password-protected.

Server has its own certificate signed by the CA, knows the username of every registered user and their RSA public keys. Moreover "dedicated storage" is already allocated.

We have generated the certificates of server and CA using SimpleAuthority, while RSA private and public key have been generated through the OpenSSL command-line tool, encrypting the private keys with a passphrase.

2.2 Protocol

When the application starts, Server and Client must authenticate and negotiate a symmetric session key.

In order to guarantee Perfect Forward Secrecy for the negotiation we decided to use the **Station-to-Station protocol**, that also provides direct authentication. Since Station-to-Station uses Diffie-Hellman protocol to establish the session key, we decided to use ECDH which provides the same level of security with less bits.

Here we show the message exchange during the protocol, where <> represent digital signature, while {} the encryption.

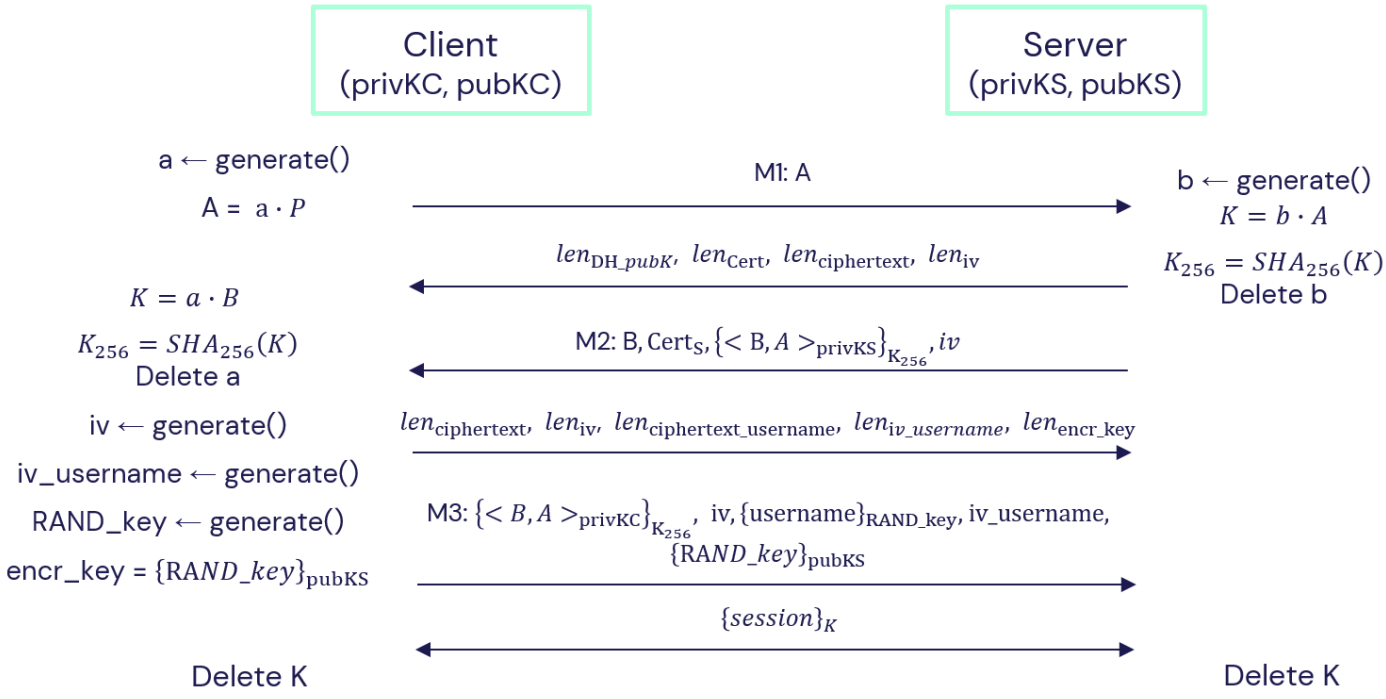

Figure 1: message exchange during authentication

Before starting to describe all the messages exchanged during the protocol, let's clarify which algorithms have been used for encryption and signature:

- Symmetric encryption done with K_{256}: the algorithm that we used is **AES-CBC-256**. It is CPA secure but as drawbacks we have that block reordering affects decryption and that it suffers

from error propagation. Anyway, these problems don't affect our system since one of these would make it impossible to verify the signature, making the other part realize that a problem occurred and stopping the process of authentication.

- Digital signature: as a digital signature technique we have used **hash-and-sign** paradigm, that means we have generated the hash of the message using SHA256 and then we have encrypted it using the RSA private key.
- Asymmetric encryption: we have used asymmetric encryption to communicate the username to the server, as it will be shown later. Instead of encrypting the whole plaintext with RSA public key of the recipient, we have used the technique of **digital envelop** generating a random symmetric key (AES-CBC on 128 bits), using it to encrypt the message and then encrypting the randomly generated symmetric key with the RSA public key.

M1: First of all the client generates its Elliptic-Curve Diffie-Hellman (ECDH) parameters relative to the prime256v1 elliptic curves, and then computes its pair of DH keys. Once obtained, it sends to the server its DH public key.

M2: When the server receives client's DH public key, it computes its own DH key pair and then calculates the shared secret on which it calculates the hash using SHA-256 algorithm. This is done in order to have a key of a fixed size and to increase randomicity.
Then it sends:

- Its DH public key;
- Its certificate, signed by the certification authority;
- The pair of DH public keys signed using its private key and encrypted using symmetric encryption with the key K_256;
- The iv used for symmetric encryption.

This is the legend from now one whenever we show the format of the messages:

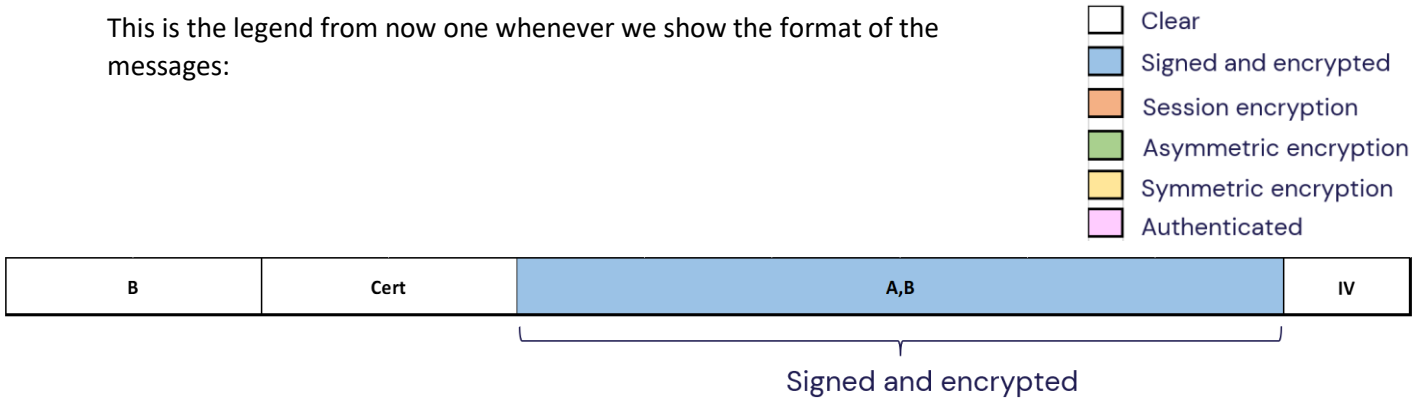

Figure 2: format of M2

Before sending all this information, it is necessary to send a message specifying the length of all the different fields, so that the client can derive them correctly.

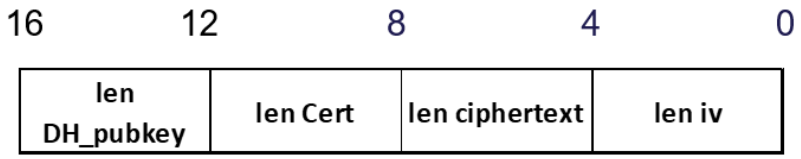

Figure 3: message format for lengths

M3: once the client has received M2, it can compute the shared secret as well and then it produces the iv needed for symmetric encryption and the iv and random key needed for the digital envelop of the username. Once it has all these parameters it signs and encrypts the pair of DH public keys and produces the digital envelop of the username, then it sends:

- The pair of DH public keys signed using its private key and encrypted using symmetric encryption with the key K_256;
- The iv for symmetric encryption;
- Its username encrypted with the randomly generated symmetric key;
- Randomly generated symmetric key encrypted with RSA public key of the server;
- Iv used for symmetric encryption of the random key.

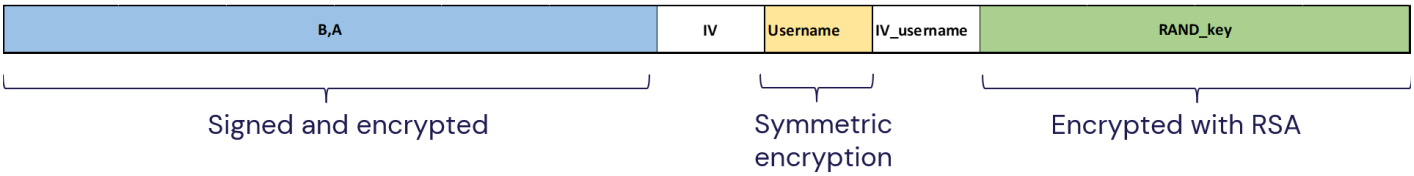

Figure 4: format of message M3

Also in this case, the client sends before a message containing the length of all these fields.

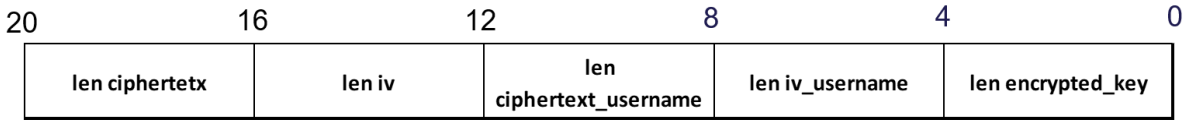

Figure 5: format of message for lengths

At this point, if everything went well, the authentication can be considered as completed.

It is necessary to underline that in case of an error in any step of the authentication, the connection between client and server is closed and client terminates while server side the process related to that client is terminated.

3. Session

The messages sent during the session are all encrypted using **AES_GCM_128**, which also provides authentication. GCM allows to specify also additional authenticated data, used to compute the TAG. In our implementation we have, during each session, a counter that starts from 0 and increments on each message. By using this counter to compute the TAG of every message sent during the session we can be safe from replay attacks. Note that, if the counter is incremented too many times it may roll over to 0 again. To avoid this issue, that would lead to a replay attack vulnerability, we have a countermeasure. When we compute the session key we use a SHA-256 hash, but during our encryption we use a key of 128 bits, so we split the hash value in two kays, call them K1 and K2. At the start of the session we use K1 up until the counter goes to UINT_MAX, so 2^{32} messages, and then when we can simply change K1 to K2 and restart with the counter to 0, allowing us to send 2^{32} new messages. Whenever we are using K2 and the counter reaching it's upper limit, we immediately close the session to avoid vulnerabilities.

3.1 Secure coding

In order to avoid tainted inputs we used some techniques of secure coding. In particular it was necessary to check usernames and filenames so that these wouldn't cause a path traversal or some other problems. The technique that we have used is **whitelisting**, accepting only the following characters: "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890.- àéèòù".

We couldn't use the function realpath() to do canonicalization because it would have been impossible to check the filename of a file to be uploaded since it didn't exist yet.

3.2 Behaviour during errors

We defined a couple of possible errors during the session, and then the behaviour of client and server when facing them. The possible errors are:

- **TBIG** : it means that the file specified is too big, in our case, bigger than 4GB. This check is done during the upload operation server side by checking the received values of "number_of_blocks" and "last_block_size". In our implementation we divide the file to upload in blocks of 4MB, so if the server sees an integer overflow over those two values, or simply the obtained file size is higher than 4GB it stops the operation sending the TBIG error state before even receiving the first block of the file. Both client and server keep the connection active after such error because it may depend on a mistake by the user.

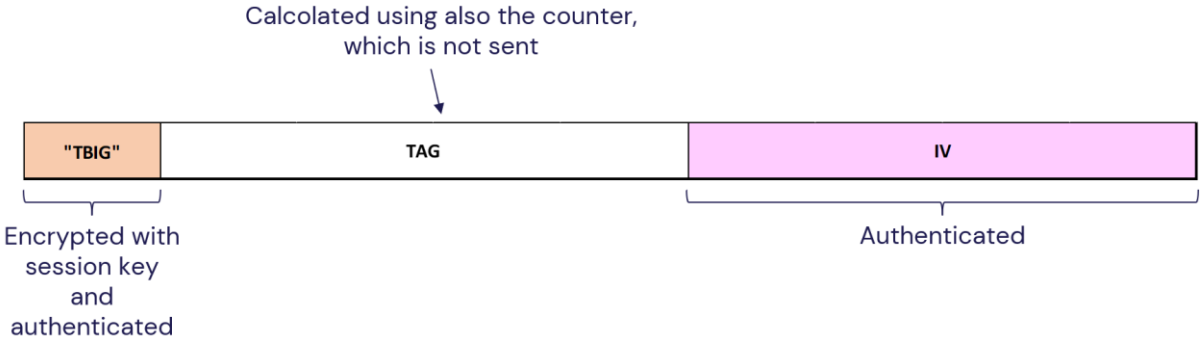

Figure 6: format for the message "file too big"

- **FILE**: the specified filename is not valid. We have a whitelist for the filename, we admit only characters that can't lead to path traversal between user directories, so the error could be cause by an user trying to send a file with a name that would lead to path traversal. Another possible reason is that, for example, an user is trying to download a file that doesn't exists.

After this error both keep the connection alive, again because the error may depend on a non malicious mistake user side.

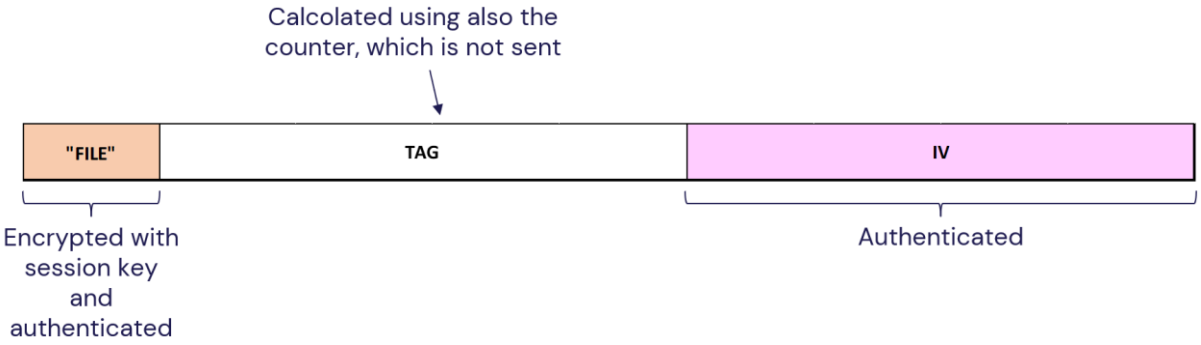

Figure 7: format of the message "incorrect filename"

- **COPY** : if an user tries to upload a file with a filename that already exists on his dedicated directory on the server, he will receive this error message, blocking the operation. This mean that you can't overwrite files, you first have to delete the old one. Once again, the connection is kept alive again.

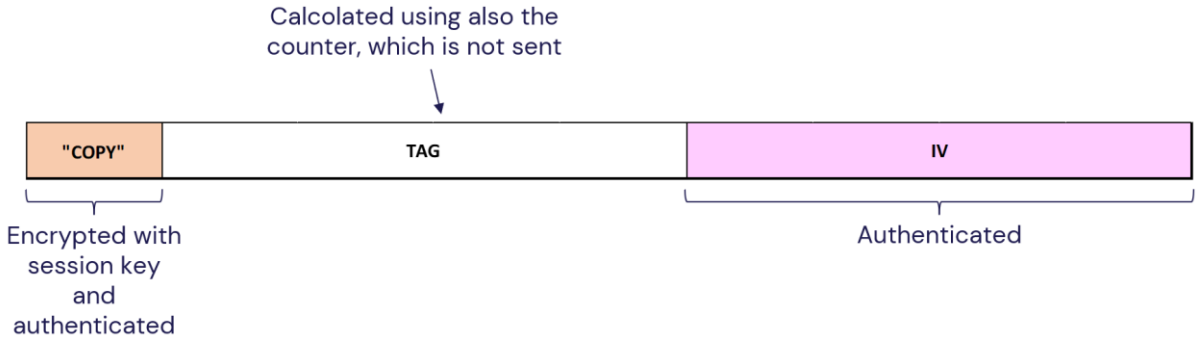

Figure 8: format for the message "file already exists"

- If an error during decryption occurs the connection between server and client is immediately closed, meaning that they would have to start again the authentication before creating a new session. This is done because, since we use TCP, if we receive a message with errors in it, it means that an attacker tried to modify such message.
- If the decryption has been done correctly and none of the above errors occurred, the server will, when required, send an "**OKAY**" message, this is done because for example if the client sends a filename for the upload operation, before starting to send file data he needs to be sure that the server has accepted the operation, the communication hasn't had any issue and that the chosen filename has not been refused.

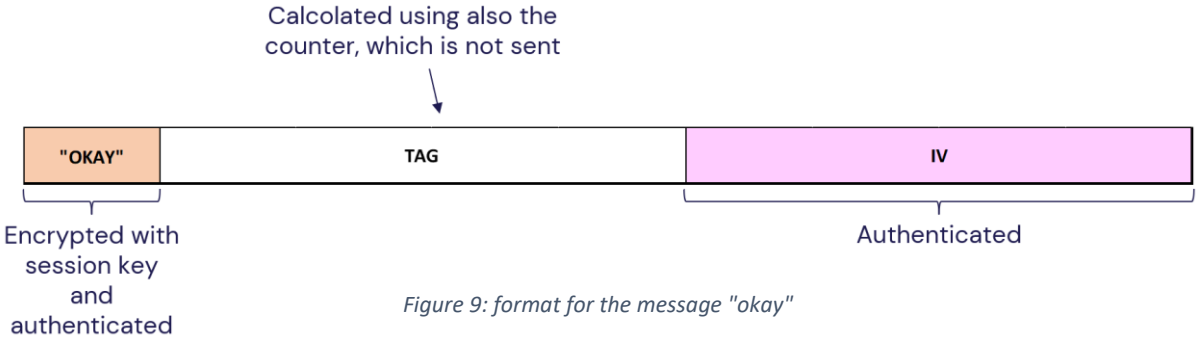

Figure 9: format for the message "okay"

4. Operations

After the authentication, the session starts and a client can ask the server to provide 6 different services: UPLOAD, DOWNLOAD, LIST, RENAME, DELETE and LOGOUT. Each operation has it's own starting message, that is standardized on a message of 12 bytes, added to the 16 byte tag and 16 byte IV. The choice of 12 bytes depends on the fact that the rename function will require exactly 12 bytes on the first message, and that is the maximum number of bytes that an operation will need to use for the first message because for the latter the request will have the first 4 bytes equal to "RENA", then 4 bytes for the length of the filename of the file to rename and the last 4 bytes containing the length of the new name defined by the client. This mean that for other operations we will pad the non used bytes with random bytes that will be ignored by the receiver, as they are used only to define a fixed size first message.

4.1 Upload

This is the operation needed to upload a local file to the cloud storage.

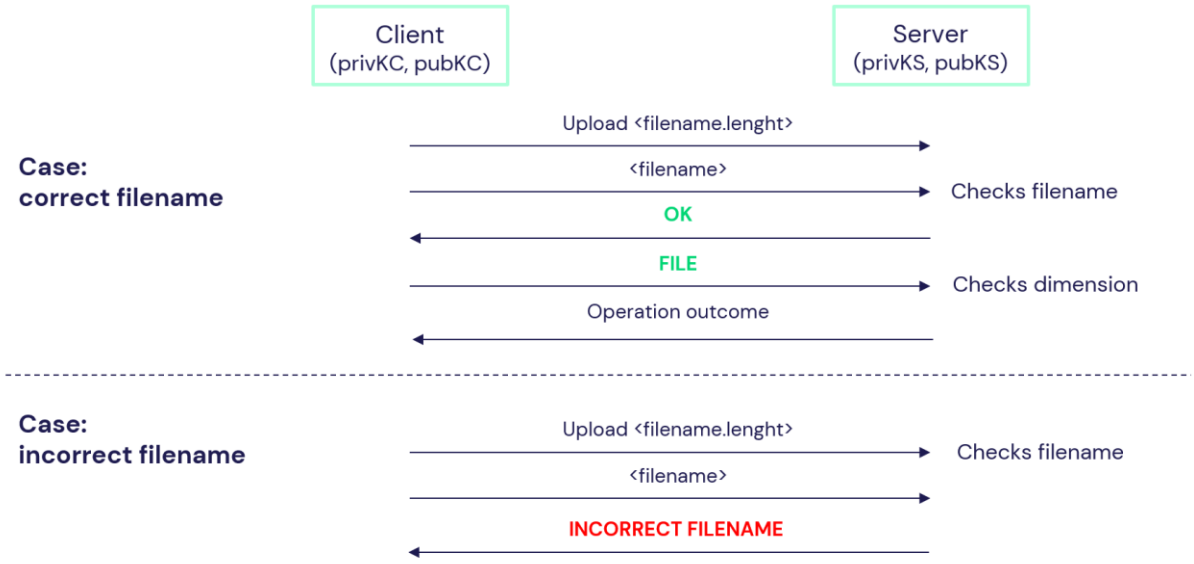

Figure 10: upload operation

The first message will have 4 bytes equal to "UPLO", then 4 bytes referring to the length of the filename that we want to upload and 4 random bytes.

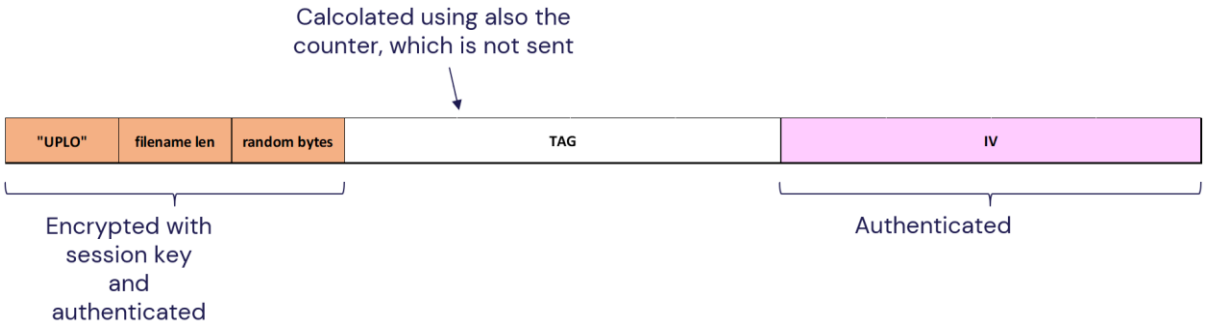

Figure 11: format of the message for the request of upload

The second message will be the filename, whose length will be the value sent in the previous message.

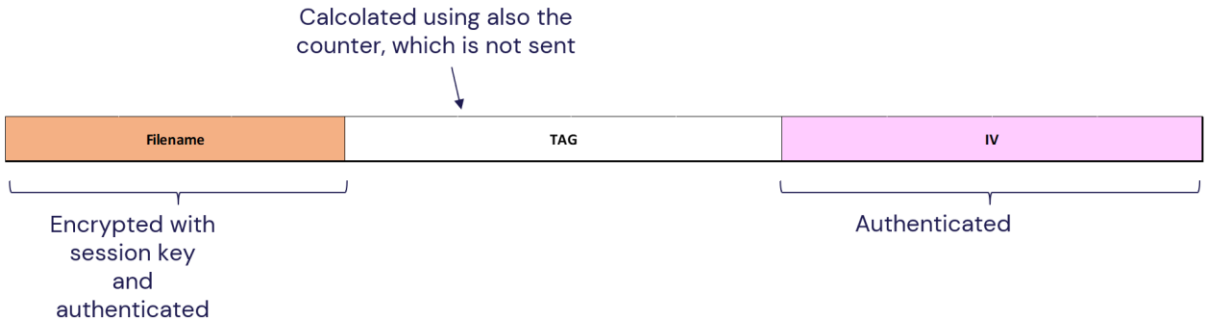

Figure 12: format of the message containing the filename

After that the server will check if the name is correct and if it's not already present in the dedicated directory of that user (of course this is done if the decryption has been successful and the tag is correct) and if everything is fine will send to the user an "OKAY" message. After this the user will send 4 bytes containing the number of blocks in which the file has been divided and 4 bytes referring to the size of the last block, in case the file size isn't a multiple of the BLOCK_SIZE, that is 4MB. Once again the server will check the values, to be sure that the file is not bigger than 4GB and then send "OKAY" (or "TBIG", if so the operation is stopped as explained in the error message section). Finally there will be the data transmission, in particular the client will send every block without waiting for and answer by the server up until he sends the last block.

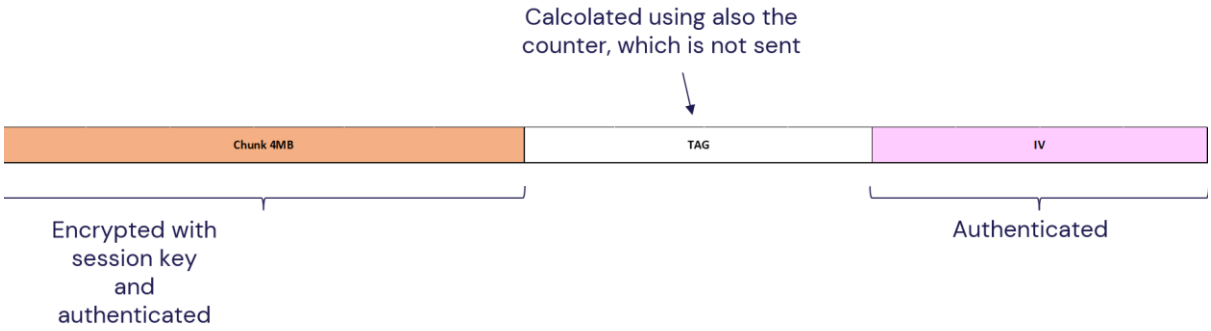

Figure 13: format of the message for the transmission of the chunck

Note that if the server finds an error during the data transmission, the error will be a decryption one, meaning that he will close the connection (and delete the file that he has created during such operation) and the client will eventually do the same thing. Instead if everything went well, the server will send "OKAY" to the client and this will conclude the service. This last message is important because now the client will be sure that the file is correctly stored on his dedicated directory.

4.2 Download

With the operation download the user is able to download a file from the cloud storage to its local storage.

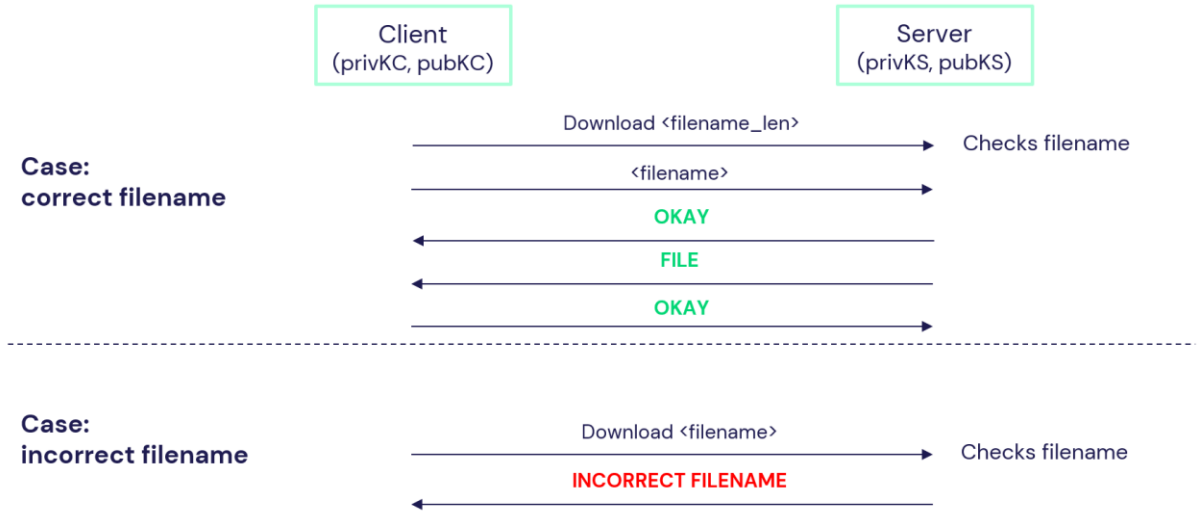

Figure 14: download operation

The first message has 4 bytes equal to “DOWN”, then 4 bytes with the length of the filename and the last 4 will be random.

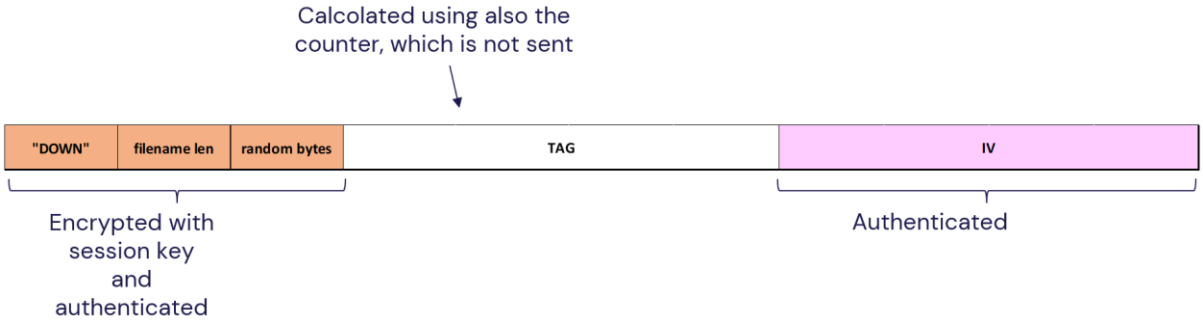

Figure 15: format of the message for download operation

After this one the client will provide the filename to the server and he will check the validity of the latter, and also if such file exists or not, sending the result of those checks back to the user. Now the server will have to send a message in which 4 bytes tell the number of blocks to send and 4 referring to the last block size. In this case the user will not send “TBIG” if the file size is above 4GB, instead it will immediately close the connection, this because since the server will only accept files of which size is below 4GB, it’s impossible that he has one bigger than that, meaning that there must have been an error or an alteration of the message. After this check, the server will send the blocks in the same way the client does for the upload, and he will wait for the “OKAY” response from the user.

4.3 List

With this operation the user is able to see the list of the files that are present inside its cloud storage.

Figure 16: list operation

The service begin by sending 4 bytes equal to “LIST” and 8 random bytes.

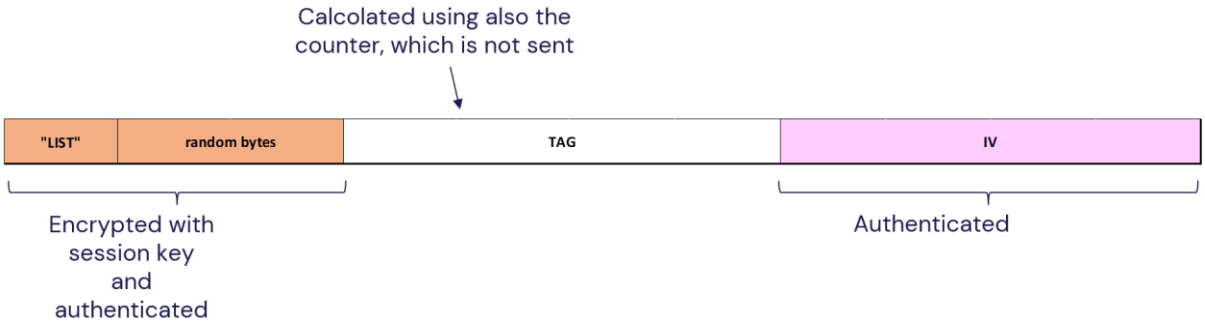

Figure 17: format of the message for the list operation

After that the server will respond with 4 bytes containing the size of the list and then sends the list of files contained in his dedicated storage. There is no need to wait for an “OKAY” message since the only error that could occur here is a decryption error, that will lead to the shot down of the session.

4.4 Rename

This is the operation needed to rename a file on the cloud storage.

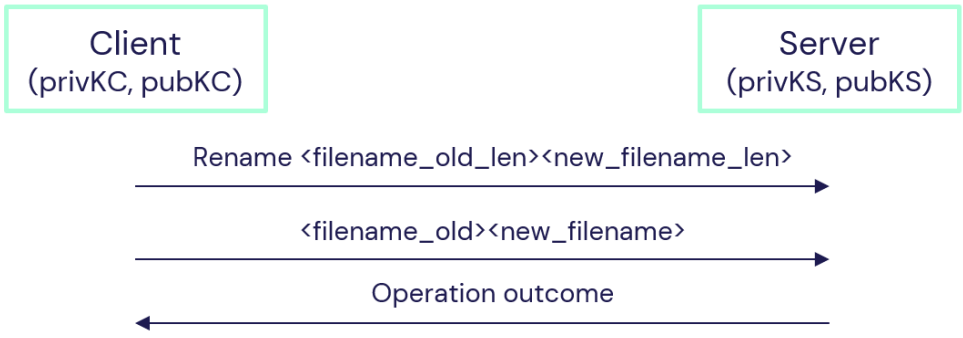

Figure 18: rename operation

The first message has 4 bytes equal to “RENA”, then 4 bytes containing the length of the filename to change and the last 4 hold the length of the new file name. Once again we’ll have to send the filenames to the server, that will check their validity and return the status. If everything went well until now, the server will simply rename the file.

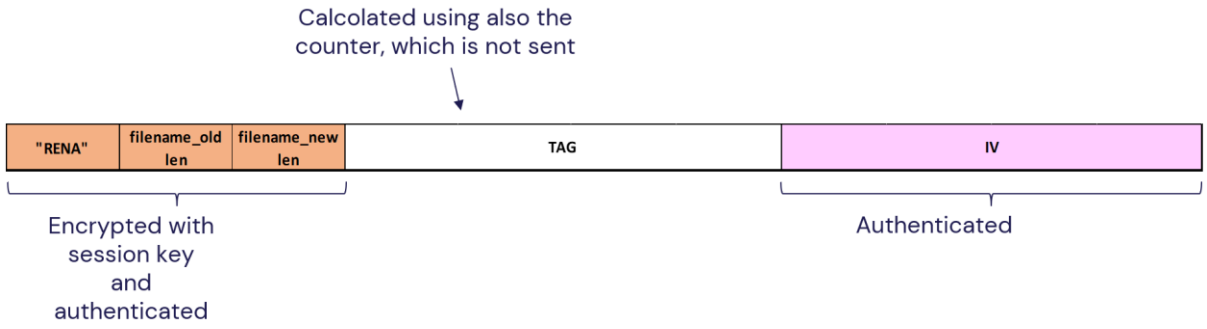

Figure 19: format of the message for rename operation

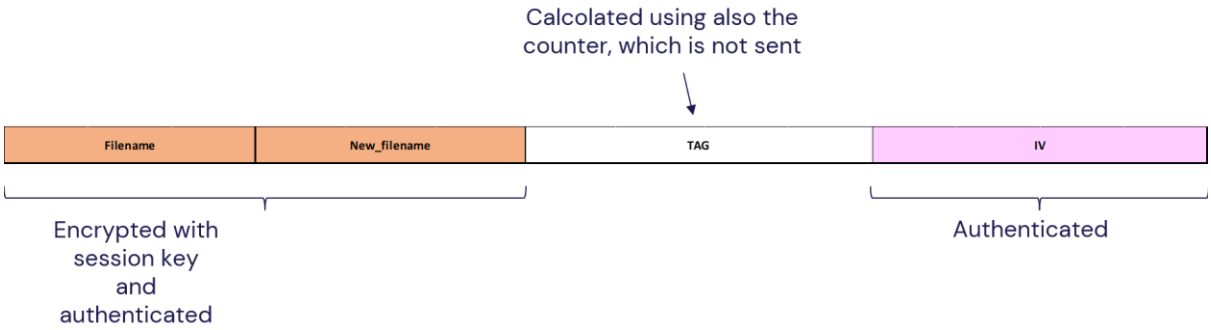

Figure 20: format of the message sent to communicate filenames

4.5 Delete

With this operation a user can delete a file from its cloud storage.

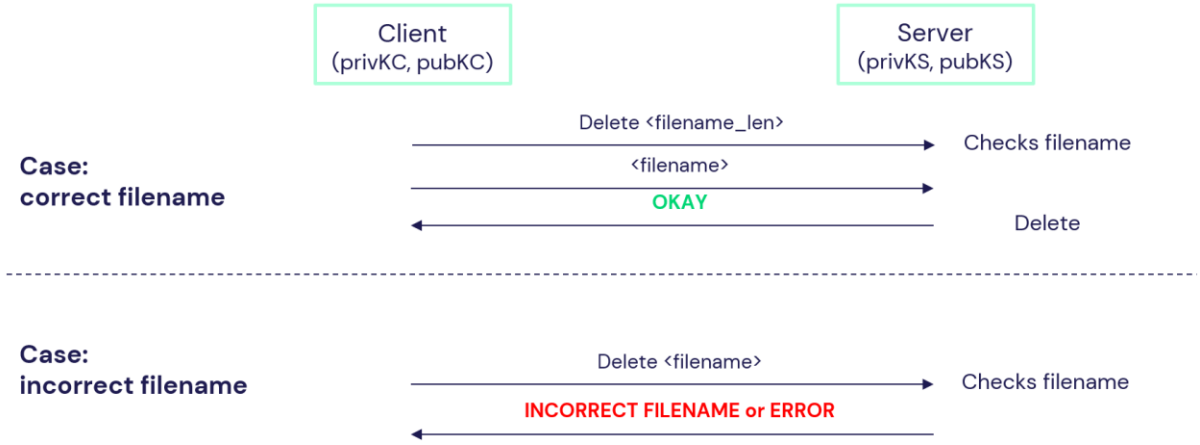

Figure 21: message exchange for delete operation

Here we will have, for the first package, 4 bytes equal to “DELE” and 4 referring to the filename length, so we can then send the filename to delete to the server. If the name is valid, the server will delete it and notify it with the “OKAY” status, otherwise will send an error message.

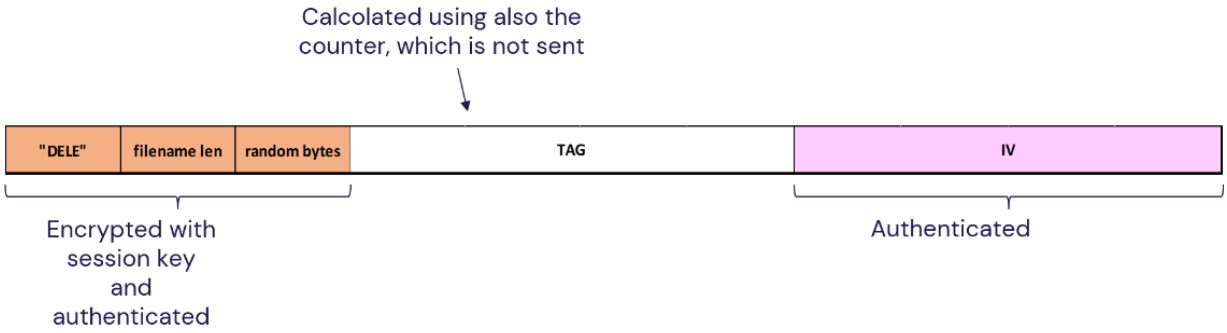

Figure 22: message for delete operation

4.6 Logout

When the user logs out sends a message to the server to close the connection and then shuts down.

Here we will have only one message to send, that is one containing 4 bytes equal to “LOGO” and 8 random bytes. The client will then close the connection with the server and it will do the same when reading that.

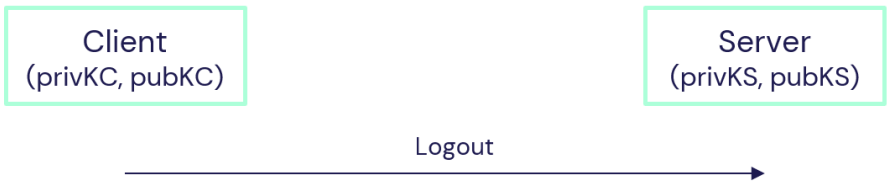

Figure 23: message exchange for logout

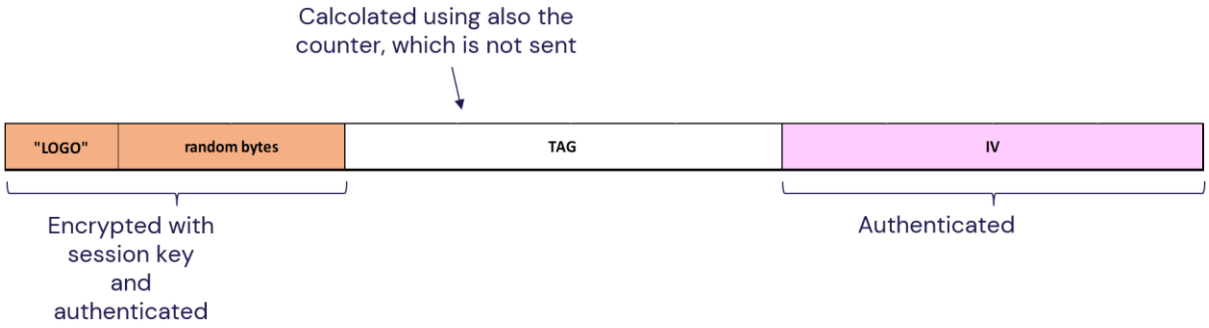

Figure 24: message for logout operation