University of Pisa
MSc Computer Engineering
Foundations Of Cybersecurity

Secure File Transfer

Andrea Di Marco
Edoardo Loni
Lorenzo Mirabella

Academic Year 2021/2022

Contents

1 Introduction

The application is a secure implementation of a file transfer program that supports multiple users. Each user has a “dedicated storage” on the server, and User A cannot access User B “dedicated storage”.
Each user can Upload or Download a file to/from the server but each file size cannot exceed the 4GB limit. These files can also be renamed or deleted them if they are already been stored.
In order to visualize his/her own folder, every user can also request a list of every files stored on the server.
Every communication performed during the application usage are confidential, authenticated and reply protected, moreover the application is protected against any possible kind of path traversal attacks.

2 Handshake

In the handshake it has been used Elliptic Curve Diffie Hellman Ephemeral Key Exchange and RSA Signature scheme. ECDHEKE was chosen for perfect forward secrecy and performance reasons. The NONCE is cryptographically secure 8 bytes long random number, in order to ensure high probable freshness. The functions used to generate it are RAND_Poll and RAND_bytes of OpenSSL library.

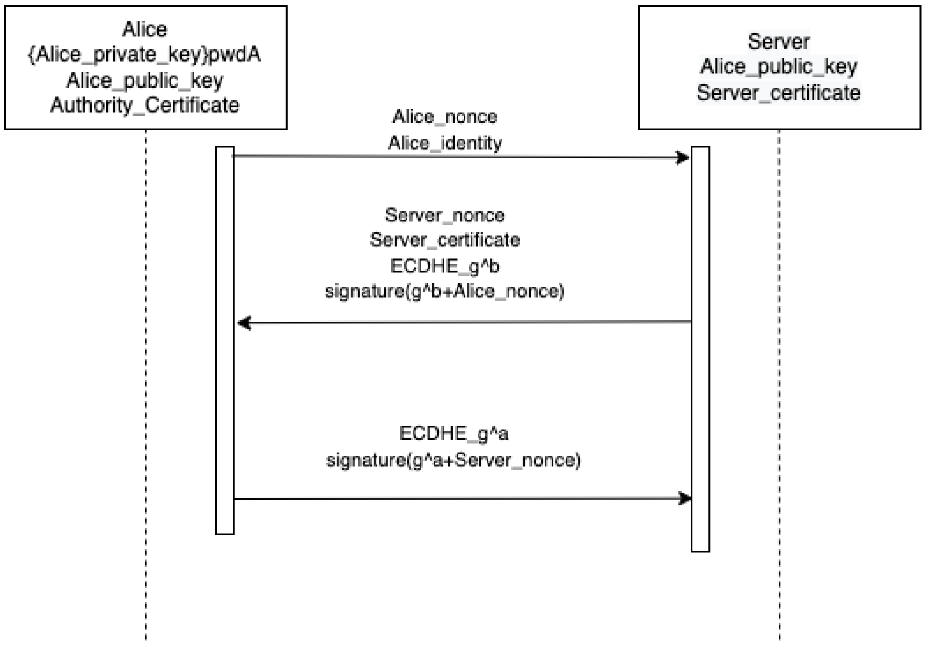

Figure 1: Handshake

2.1 Client Hello

In order to use the application, there is the need to establish a session key used to encrypt the messages. A user sends his ID and a fresh nonce to server.

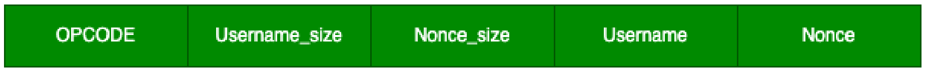

Figure 2: Client Hello

2.2 Server Hello

The server generates the DH private key (b), the public key g^b and a fresh nonce. Subsequently, it sends to the client the server nonce, his certificate, the public key and a signature of the public key plus the client nonce. The latter is necessary to prove to the client that the signature was just (client nonce) generated by the server (using his secret private key).

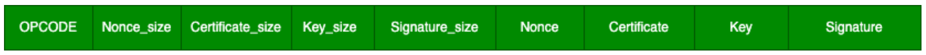

Figure 3: Server Hello

2.3 Client Authentication

At this point, the client has to firstly verify the server certificate using the CA certificate and the CRL. If the verification is successful, the signature can be verified. Now the client can generate his DH private key (a) and public key $g^{(ab)}$. The session key can be derived hashing with SHA-256 the shared key and taking the first 128 bits. The client sends to the server the public key and the signature of the latter plus the server nonce for the same previous reasons. The user private key is protected by a password, it follows that he has to insert it in the middle of the handshake.

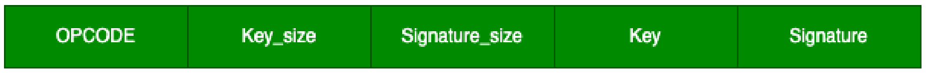

Figure 4: Client Authentication

2.4 End

Finally, the server can generate the same session key after verifying the signature with the public key of the user. It sends to the client a confirmation ACK with an already encrypted message.

In this way a Man-in-The-Middle cannot do anything to break this scheme. A single session can be broken only if the adversary can guess the DH private key.

3 Session

Once the handshake is finished, a session between the client and the server is established. All messages are built by using the format that is shown in figure 5:

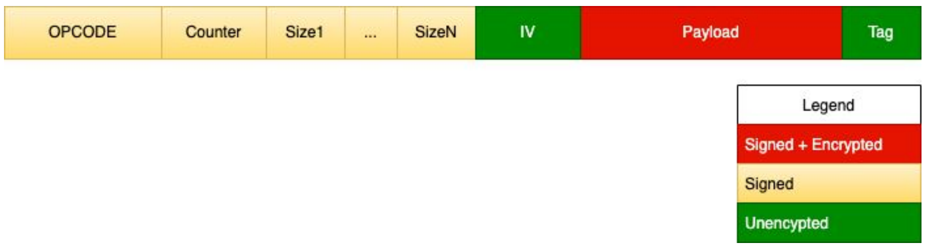

Figure 5: Generic format of a session message

In particular:

- The OPCODE field is used so that client and server can decide which actions they have to perform according to it after the reception of a message.
- Both the client and the server maintain a local counter that is initialized to 0 at session beginning; its value is incremented by each end point immediately before sending or receiving a new message.
- When a received message is processed, the value that has been read in the COUNTER field is compared with the locally maintained one. If they are different, there is the possibility that an adversary is trying to perform an in-session replay attack. In this case, an exception is thrown since the message has to be discarded.
- The dimension in bytes of each payload element is written in a different SIZE field.

AES_128_gcm is used as authenticated encryption protocol. This choice has been made since offers a good trade-off between performances and security.
OPCODE, COUNTER and SIZE fields are part of the AAD, since their integrity needs to be protected even if they don't carry sensible information from the point of view of confidentiality.
In both upload and download operations, the existence of the files within their respective folders is checked. In particular:

- the download operation fails immediately if a file is already present inside the client's file folder or if the file is not present inside the server's folder.

- in the upload operation the server sends an error message to the client before the actual file transfer is initiated if the file is already present inside the server's folder.

All files available in the cloud storage are located by the following relative path:
server_file/client/client_username/file/filename.
Every time that a file name inserted by the user is processed by the client or received by the server, a check is performed in order to verify that it is expressed in the format {....}.{....}, where {....} is an alphanumeric string. In this way, no path traversal is allowed.

In the following sections, the sequence diagrams and message formats are reported in relation to the message exchange that is required for each application operation.

3.1 List

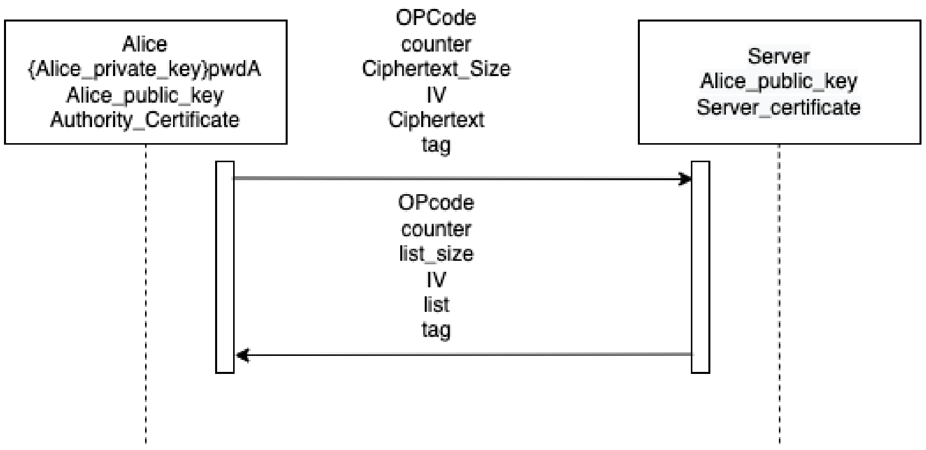

Figure 6: Sequence diagram of the !list message exchange

Message formats:

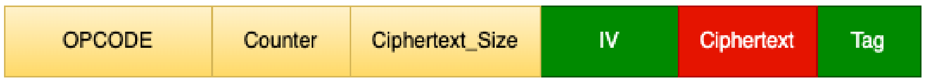

Figure 7: List request packet format

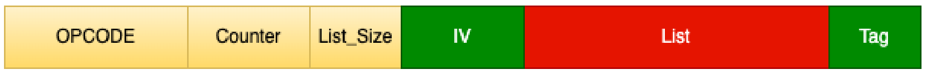

Figure 8: List packet format

3.2 Download

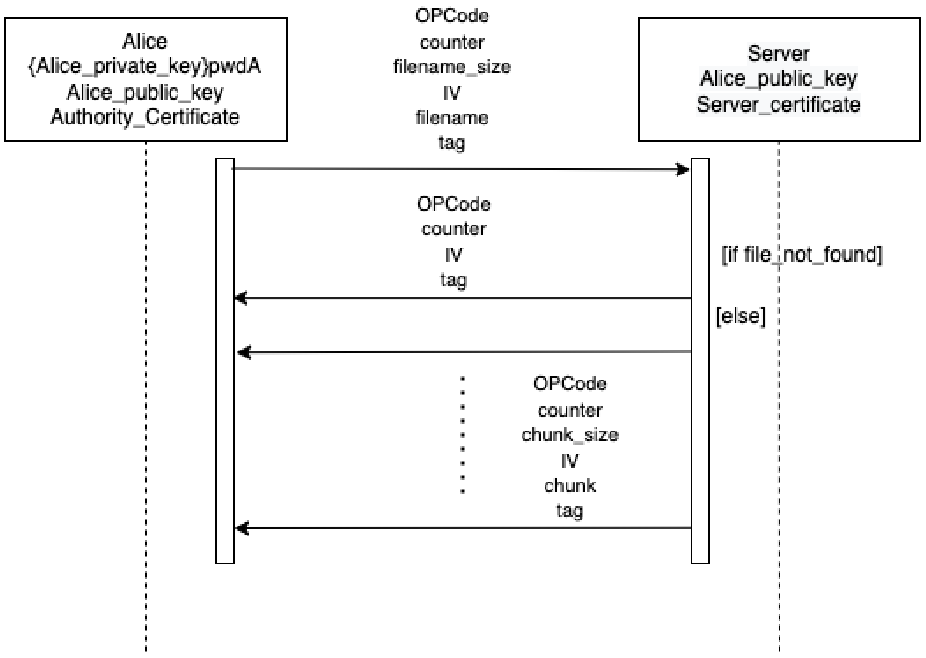

Figure 9: Sequence diagram of the !download message exchange

Message formats:

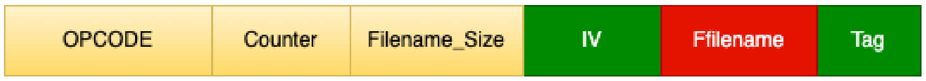

Figure 10: Download request packet format

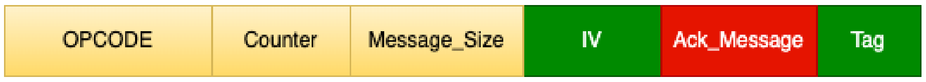

Figure 11: Download ack packet format

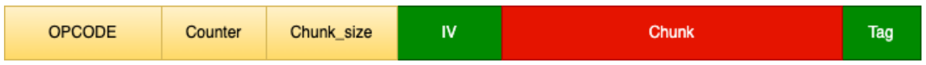

Figure 12: Downloaded chunk packet format

3.3 Upload

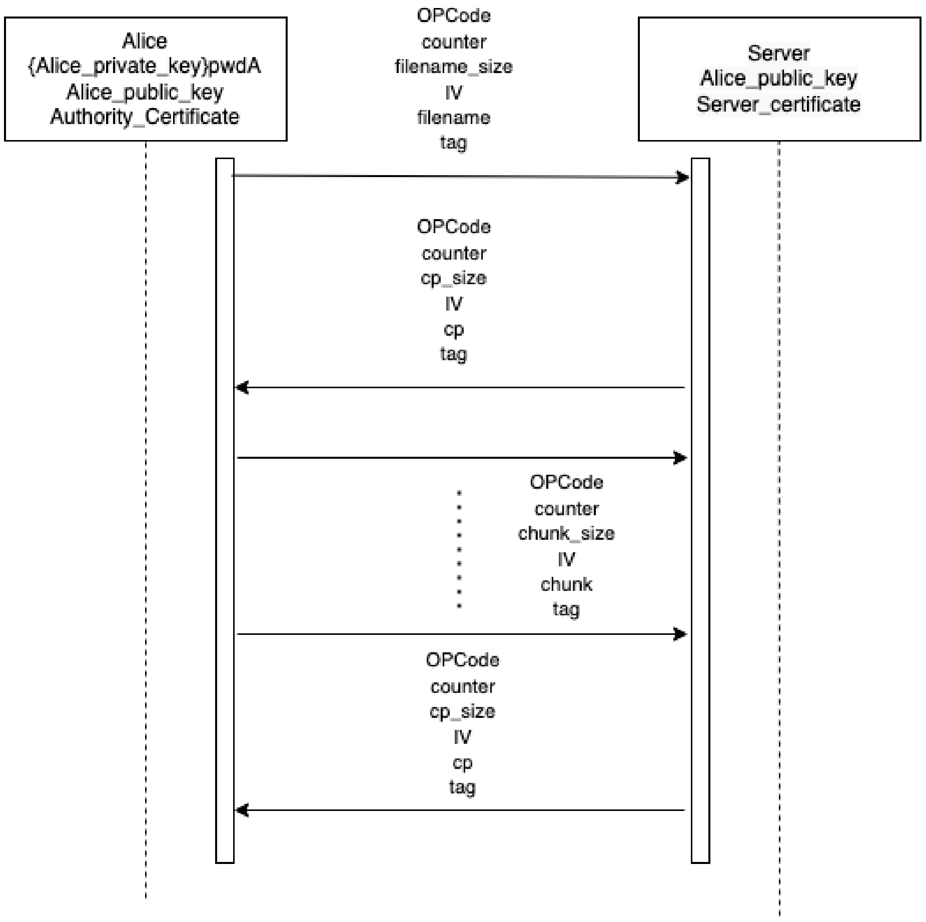

Figure 13: Sequence diagram of the !upload message exchange

Message formats:

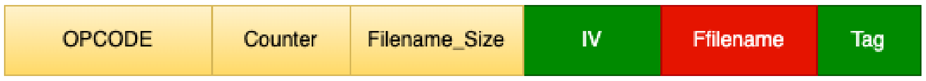

Figure 14: Upload request packet format

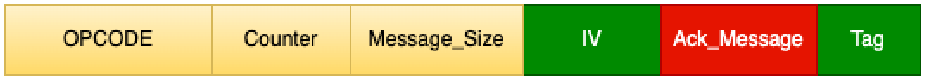

Figure 15: Upload ack packet format

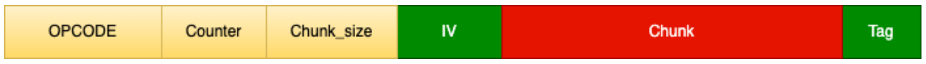

Figure 16: Uploaded chunk packet format

3.4 Rename

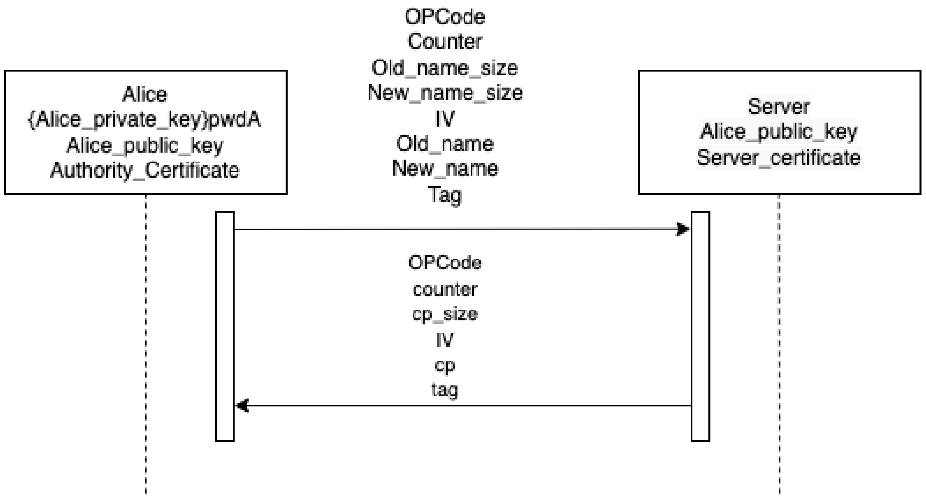

Figure 17: Sequence diagram of the !rename message exchange

Message formats:

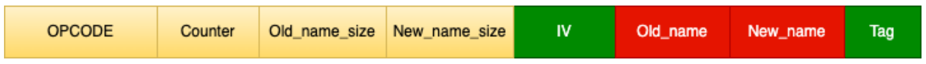

Figure 18: Rename request packet format

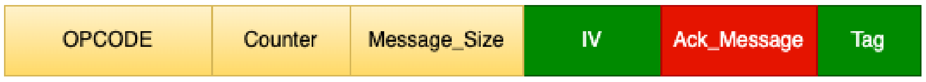

Figure 19: Rename ack packet format

3.5 Delete

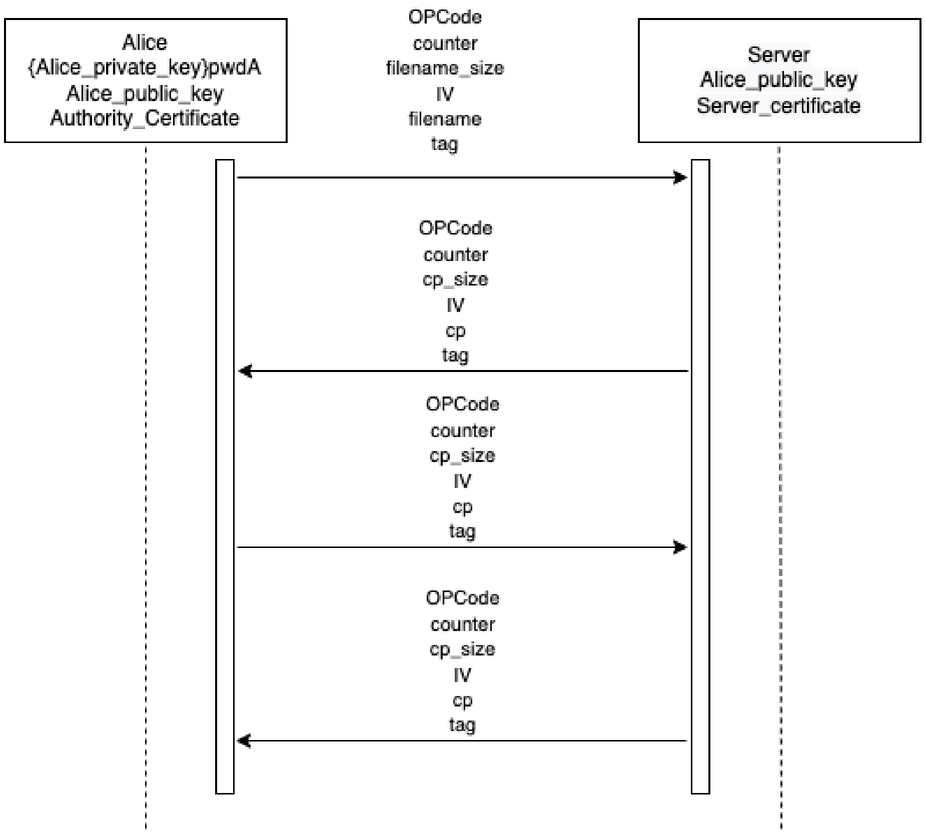

Figure 20: Sequence diagram of the !delete message exchange

Message formats:

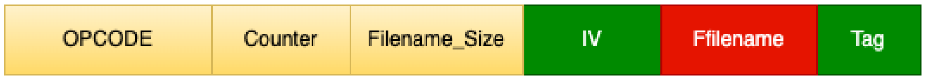

Figure 21: Delete request packet format

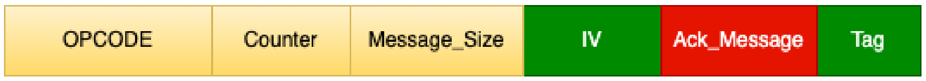

Figure 22: Delete confirm-ack packet format

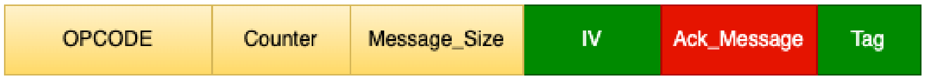

Figure 23: Delete confirm packet format

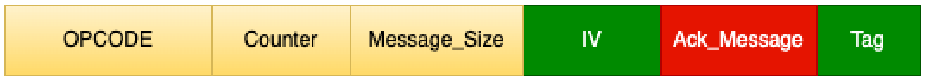

Figure 24: Delete ack packet format