Secure file transfer application

2021. spring

—

SHAjt

Simon Csongor

Németh Krisztián

1. Application overview

The developed application is a secure file transfer application. It consists of a client program and a server program that communicate with each other securely. The client interacts with the user via a console interface. The user can type in commands, which are sent to the server by the client. The server executes the received commands, and provides feedback to the user about the result of the operation.

Using these commands, users can manage their own secure remote directory. The commands are related to file operations such as uploading files to and downloading files from the server, creating folders, listing the content of a folder, etc.

The server is able to handle the folders and files of multiple users. Users have access only to their own folders and files. The server is able to handle multiple parallel connections with different clients at a time.

The client is able to authenticate the server through a digital certificate. The users are authenticated by their username and password. A session lasts until the user logs out, or until the server closes the session due to inactivity.

1.1. Functional requirements

- The file server is able to store user data.
- The file server can authenticate users, and perform access control.
- The file server is able to handle the folders and files of multiple users.
- The file server can handle multiple active sessions at once.
- The client can initiate connection to the file server.
- After the authentication procedure, the client can access its own files on the file server.
- The client is a console application, which can be controlled using a set of commands.
- The server executes the commands and returns the results to the client.
- The results of the commands are displayed on the client's console window.
- The application must support the commands listed in section 1.2.

1.2. Supported commands

- 0x00 MKD [path] – creating a folder on the server
- 0x01 RMD [path] – removing a folder from the server
- 0x02 GWD – asking for the name of the current folder (working directory) on the server
- 0x03 CWD [path] – changing the current folder on the server
- 0x04 LST – listing the content of a folder on the server
- 0x05 UPL [path] – uploading a file to the server
- 0x06 DNL [path] – downloading a file from the server
- 0x07 RMF [path] – removing a file from a folder on the server
- 0x08 LGT - invalidating the current session and logging out the user

2. Attacker models

2.1. Authenticated user

Goals:

- To access other users' folders and files.
- To have the server execute malicious code (e.g. through memory corruption).
- To perform a denial of service attack on the server.

Capabilities:

- Has an active, authenticated connection with the server.
- Able to send commands to the server.

2.2. Outsider

Goals:

- To steal user data.
- To impersonate legitimate users.
- To distribute malicious software to users (e.g. by impersonating the server).
- To perform a denial of service attack on the server.

Capabilities:

- Eavesdropping on client-server communications
- Man-in-the-middle attacks (e.g. impersonate the server)
- Able to replay captured messages
- Able to send session initiation messages to the server

2.3. Application server (trusted but curious)

Goals:

- To read the stored user data

Capabilities:

- Has physical access to the user files

3. Security requirements

3.1. Requirements

Based on our attacker models and our general goals for this project, we derive the following security requirements:

Confidentiality:

1. The server must not store user passwords in plaintext, only as salted hashes.
2. The server must not store its private key in plaintext.
3. The client and the server must establish a secure channel at the beginning of the communication (through the exchange of cryptographic keys).
4. All messages must travel encrypted, through the secure channel.
5. All user files must be encrypted before they are sent to the server.
6. The server must not have access to the decryption keys for the user files.
7. Perfect forward secrecy should be ensured.

Integrity:

8. Sequence numbering must be implemented for replay protection.
9. Both parties must be able to verify that the received messages have not been altered in transit.

Access control, Identification, Authentication, Authorization:

10. User authentication must be implemented for the login process.
11. The client must be able to authenticate the server.
12. All messages must use a proper MAC to authenticate the sender.
13. Proper access control must be implemented for the users' files and folders.

Other security requirements:

14. Command inputs must be properly sanitized.
15. Per-user command-rate limiting must be implemented.
16. Key freshness must be ensured.
17. The pseudo-random generation of nonces and sequence numbers must be implemented correctly, and the numbers must not be predictable.
18. Edge-cases, like reaching the message limit for a given encryption key (limited to avoid IV reuse for the given key) should be handled correctly.

3.2. Non-requirements and limitations

- Providing protection against denial-of-service attacks, outside the ones addressed by implementing per-user command-rate limiting, is outside the scope of this application and its security protocol.
- Our protocol does not take into consideration scenarios, where the long term private key of the server or the certification authority is compromised.
- Compliance with existing RFC standards is not a requirement.
- The maximum size of plaintext to encrypt in messages is 64 GB (GCM mode block counter limit).
- The user registration procedure is not detailed in this document, and the implementation of the registration process is not a requirement.

4. Protocol specifications

4.1. General approach

Our design decisions were based on the attacker models and security requirements we previously defined. We also aimed to minimize the number of necessary messages in the session establishment phase of the protocol, to make the process faster. We also chose to encrypt all user files with a key unknown to the server before uploading them, resulting in end-to-end encryption. Consequently, the server has no ability to read any of the user data stored on the server.

The following is a short overview of the designed protocol, all parts of which, including the specific cryptography algorithms used are detailed and illustrated in the following sections. The exact algorithms and protocols we use can be found in section 4.2.

As our initial state, the server has a long term private key - public key pair, and a X.509 certificate containing the server's public key, which has been signed by a (theoretical) trusted certification authority's public key. In practice, the private key would be stored on a tamper resistant hardware token.

The first step is to create a secure channel between the client and the server. To do this, the client and the server must agree on a shared symmetric key. They perform the Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol using freshly generated ephemeral keys. We describe the details of the key agreement protocol in section 4.3. After the key agreement protocol, both parties will possess a fresh shared session key.

During the key agreement process, the client authenticates the server. The server sends the client its signed certificate containing its public key, and a previously received, client-chosen random number, which the server signs with its private key, as a proof of ownership. The client can verify the signature using the public key inside the signed certificate.

After the shared session key has been established, the client sends the username and password to the server, encrypted with the session key, using a client-generated random nonce as the IV. The message also includes the client sequence number,

which must later be incremented with every message sent by the server to provide replay protection. Similarly, the client-generated random nonce is used as a counter value, and must be incremented with each message sent or received, and never reused during the session. The server derives a hash from the password (and the stored salt), compares it to the one stored in the server's user database, and notifies the user about the success or failure of the authentication. If the authentication was successful, the server will note down the session with the client as active, and will send a response message containing the server sequence number. The details of the authentication process can be found in section 4.4.

After the session has been established, the client can begin to send messages to the server, and upload or download files. We provide a detailed description of these messages in section 4.5.

4.2. Applied cryptography algorithms and protocols

Purpose	**Algorithm or protocol**
Encryption of user files	AES-256 CBC, padded w/ ANSI X.923
Encryption of single block SessionID	AES-256 ECB, padded w/ ANSI X.923
Digital certificate for the server	X.509, NIST P-521 public key, DER format
Digital signature	ECDSA w/ SHA512
Public key cryptography	NIST P-521 ECC
Symmetric key cryptography	AES-256 GCM
Message Auhentication Code (MAC)	AES-256 GCM GHASH
Pseudo-random number generation	os.urandom
Password hashing	24 byte Argon2id hash w/ 16 byte salt
Session key derivation	32 byte HKDF key w/ HMAC-SHA512
Key agreement protocol	Ephemeral ECDH
Pre-session integrity check	HMAC-SHA512

Most algorithms are imported from the cryptography 3.4.6 library, with the exception of Argon2, which is imported from the argon2-cff library.

4.3. Server authentication and key agreement

The first part of our protocol is the key exchange process, during which the client and the server agree on a shared secret, from which they can both derive a symmetric session key to use during the rest of the communication process.

Simultaneously, the client verifies the authenticity of the server by challenging it to sign a client-chosen proof with its private key, the authenticity of which can be verified using the signed certificate of the server.

Initially, the server possesses K_{pub} and K_{priv} NIST P-521 keypair, and an X.509 certificate containing K_{pub} , which was signed using $K_{CA\text{-}priv}$, the NIST P-521 private key of our theoretical certification authority.

When the client decides to start a session, it generates the $DHpub_{client}$ and $DHpriv_{client}$ keypair using a NIST P-521 elliptic curve. The client also generates 32 bytes of random data, which will later be signed by the server using its private key as a proof of ownership of the private key. The client sends a message to the server containing a 16 byte header, the generated 32 byte random data, and the generated keypair's public key in DER format.

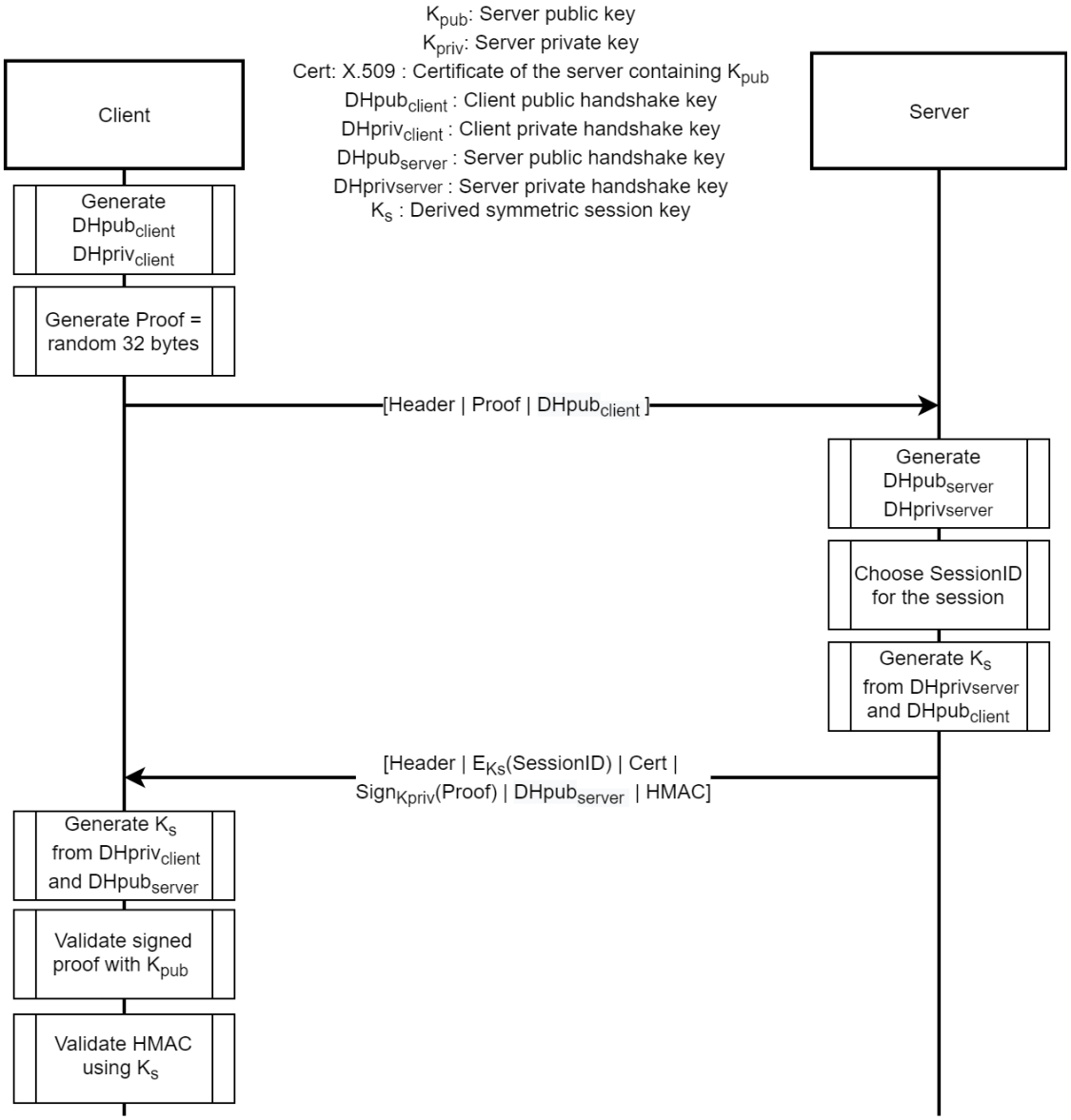
The server generates its own keypair using the same method, and derives the session key from $DHpub_{client}$ and $DHpriv_{server}$. This can be done through scalar multiplication, due to the properties of the elliptic curve:

Given $\mathbf{E: y^2 = x^3 + ax + b}$ elliptic curve with agreed upon parameters, the client can generate $DHpriv_{client}$ integer and $DHpub_{client}$ point, where $\mathbf{DHpub_{client} = DHpriv_{client} * G}$ generator point. The server does the same, and generates the $DHpub_{server}$ and $DHpriv_{server}$ keypair. The public keys can be shared through an insecure channel. After receiving the public key of the client, the server can compute $\mathbf{(X, Y) = DHpriv_{server} * DHpub_{client}}$ point, which will be the same as the $\mathbf{(X, Y) = DHpriv_{client} * DHpub_{server}}$ point calculated by the client. This point is now a shared secret, and can be used to derive K_s session key using the HKDF key derivation function. For the parameters of the HKDF function, we use the bytes of X as our master secret, Y as our salt, and HMAC-SHA512 as our hash function. The final session key produced is 32 bytes long, which will later be used for the AES-256 encryption of messages.

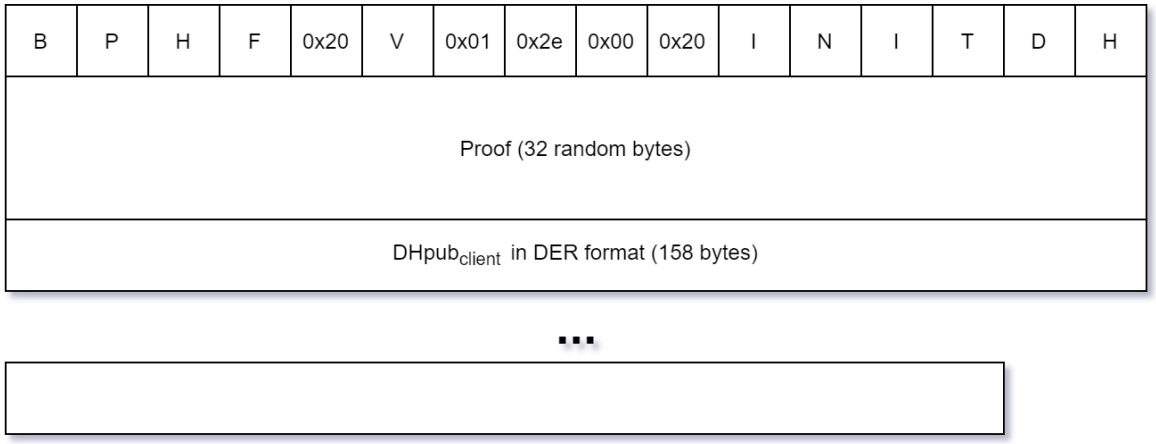
After deriving the session key, the server sends the client a message containing the same message header, a randomly chosen 8 byte SessionID padded to one 16 byte block (using ANSI X.923 padding) and encrypted using AES-256 ECB with K_s (not colliding with the IDs of existing active sessions!), the X.509 certificate of the server in DER format, the signed client random (ECDSA signed using K_{priv}) padded to 144 bytes, the public DH key of the server ($DHpub_{server}$), and a HMAC-SHA512 MAC of the message, created using K_s session key.

The client derives the same K_s session key as the server using $DHpriv_{client}$ and $DHpub_{server}$ and the same HKDF parameters, verifies the HMAC of the server message using K_s, and verifies the signature on the client random using K_{pub} (contained inside the certificate). After this step, the key exchange is concluded, with the server authenticated to the client, and K_s session key established.

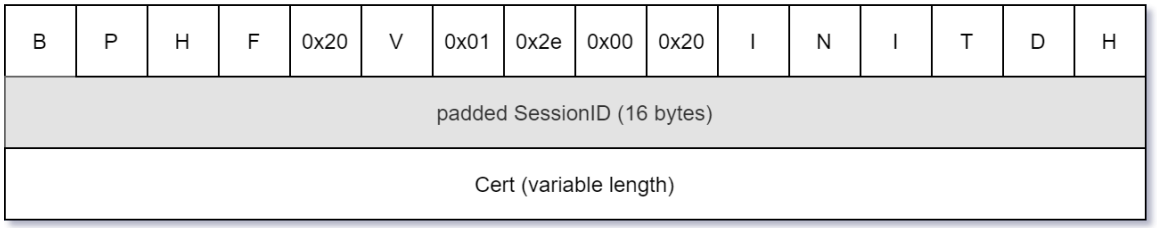
Communication sequence:



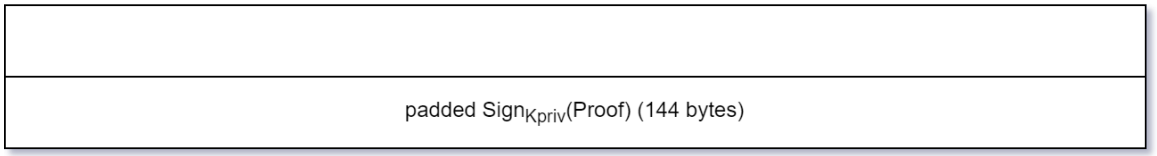
Message bodies:



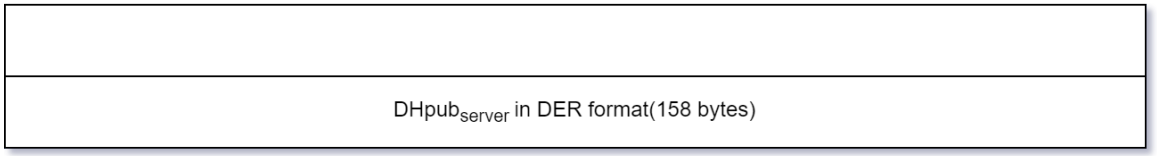
...



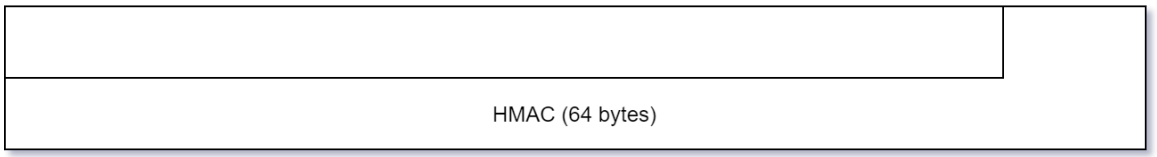
...



...



...



...

4.4. Client authentication

With the symmetric session key constructed and the server authenticated, the password based authentication of the client can be started. The client first generates a 16 byte random nonce, which will be used as the initialization vector for the AES GCM block cipher used during the communication. The nonce is incremented with each message between the client and the server in the session. This does not pose a risk on the security of the GCM mode encryption, since the initial value of the nonce is not predictable, and the nonces are never reused.

The client generates a client sequence number, which the server will later increment with each message sent to the client. The sequence number is 16 bytes long, with the upper 8 bytes being initially randomly generated, and the lower 8 bytes initially being 0. During the later phases of the communication, the sequence number serves to protect against replay attacks, and the sequence number of the received message must always be larger than the sequence number in the last received message.

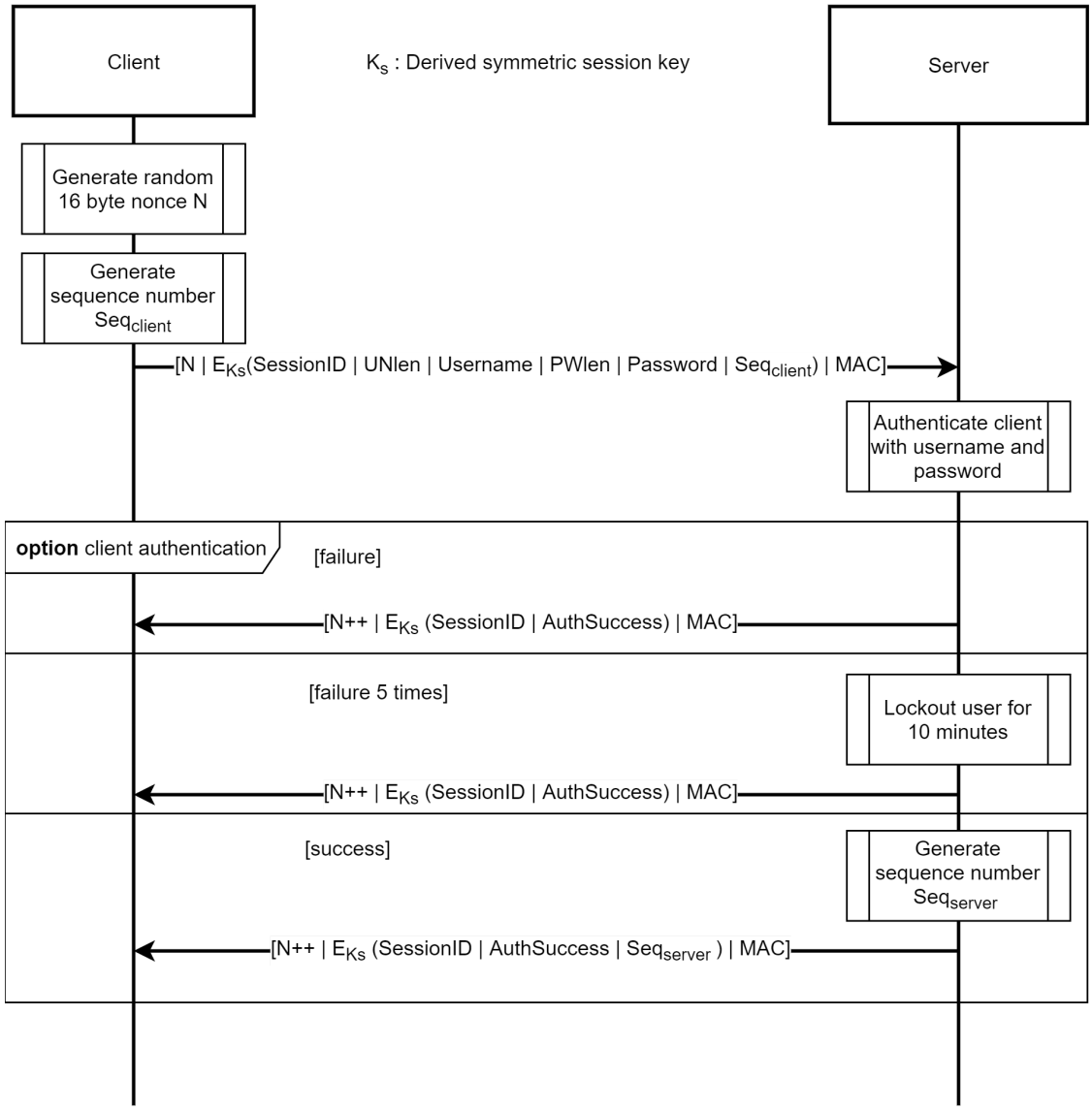
The client sends a message to the server, which will contain the generated nonce, an AES-256 GCM encrypted payload, and a MAC generated by the GCM mode cipher. The payload is encrypted with the session key and the generated nonce. The payload contains the previously received SessionID, the username and password of the user, and the client sequence number.

The server verifies the MAC, decrypts the payload, and derives a key from the password using the Argon2id algorithm, with the salt stored for the user in the user database of the server. If the derived key matches the one stored in the database, the client authentication is successful. The server notifies the client about the success/or failure of the login process using a 1 byte variable in the next message.

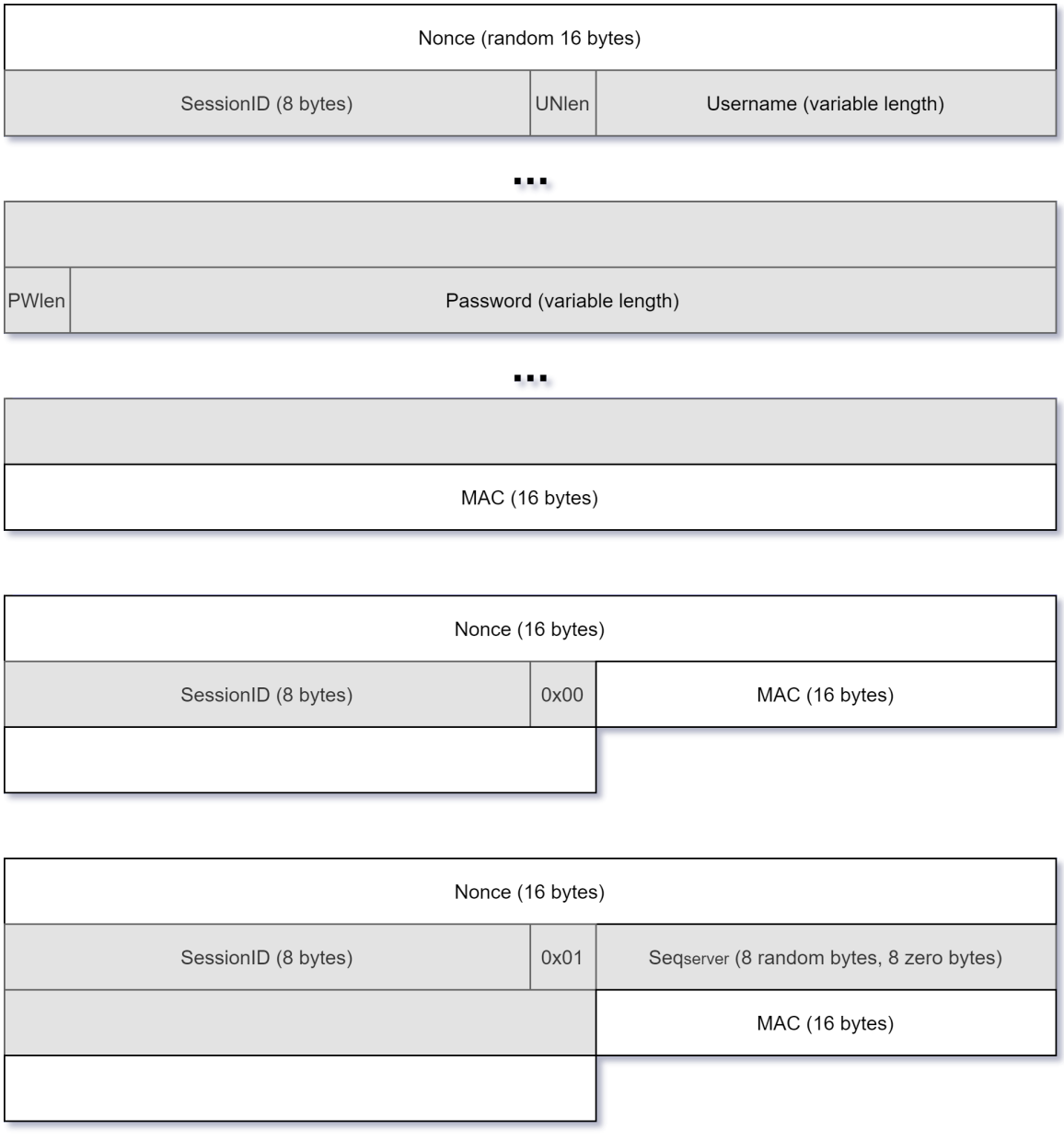
If the authentication fails 5 times in a 1 minute window, the server will invalidate the session, and block authentication attempts for the username for 10 minutes.

If the authentication is successful, the server will also include the initial server sequence number in the message, the format of which is the same as the client sequence number.

Communication sequence:



Message bodies:



4.5. Default command behaviour

After both members of the communication, the client and the server, have been authenticated, the client can begin to send commands to the server. The message sent by the client begins with the previously described 16 byte nonce, which has been incremented to be unique. The AES-256 GCM cipher encrypted payload follows the nonce in the message. The payload consists of the sessionID, the type of the command which was sent, the length of the parameters for the command (as a 2 byte integer), the the bytes of the command parameters, and the server's next sequence number. The sessionID is used by the server to distinguish which authenticated user sent the command, and if it should be executed. The command types are associated with a number so they can be sent on one byte. The command parameters are the necessary information for the execution of some commands, f.e it could be the name of the folder which should be created. The MKD, RMD, CWD, RMF, UPL, and DNL commands cannot be executed without it. The message ends with the 16 byte MAC from the GCM mode cipher.

After the server verifies the MAC which was generated by the GCM mode cipher, and deciphers the message payload with the using the received nonce and the shared session key, it checks if the user associated with that session has the right to execute the sent command, for example, the user isn't trying to delete a different user's folder. If no access violation was detected, the server executes the command and creates the response message.

The response message also begins with the incremented nonce. The Payload is, again, encrypted with an AES-256 GCM cipher. The payload consists of the sessionID, a success indicator, the executed command's response and the client's next sequence number. The message ends with the 16 byte MAC from the GCM mode cipher.

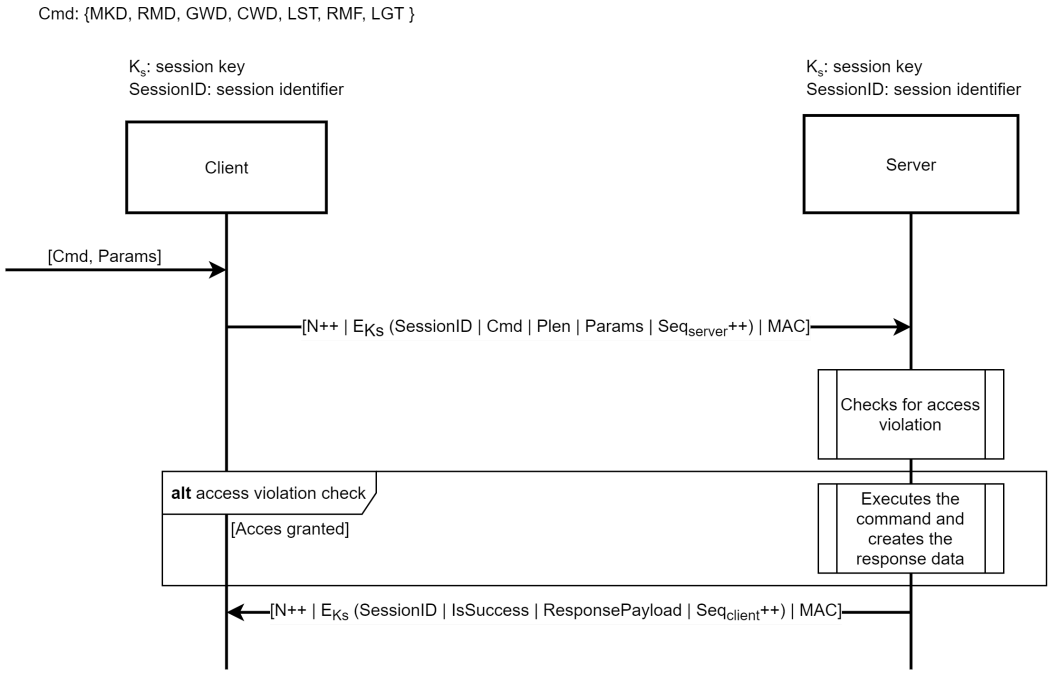
The possible command responses are: 0x00 (failure), 0x01 (success), 0x02 (access violation), 0x03 (path not found).

The client decrypts the payload with the sent nonce and the session key, then prints the command response to the console for the user.

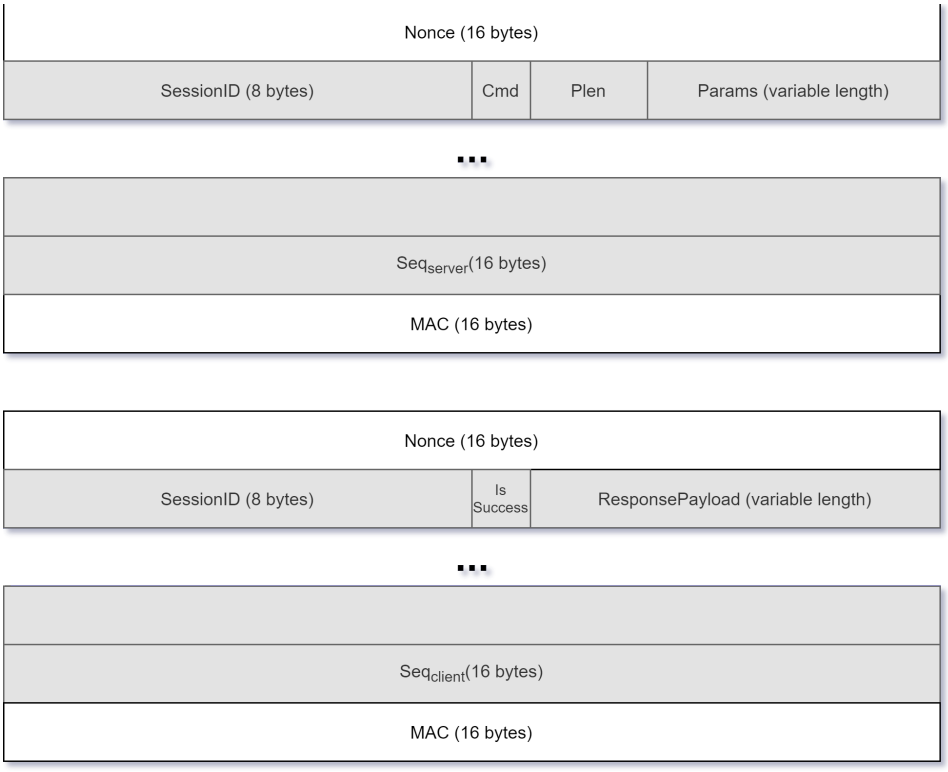
One unique message to mention is the LGT command. When this command is executed by the server, the server invalidates the SessionID contained in the

message, and deletes the corresponding session key. If the client received a "success" response to the LGT command, the client also deletes the session key.

Communication sequence:



Message bodies:



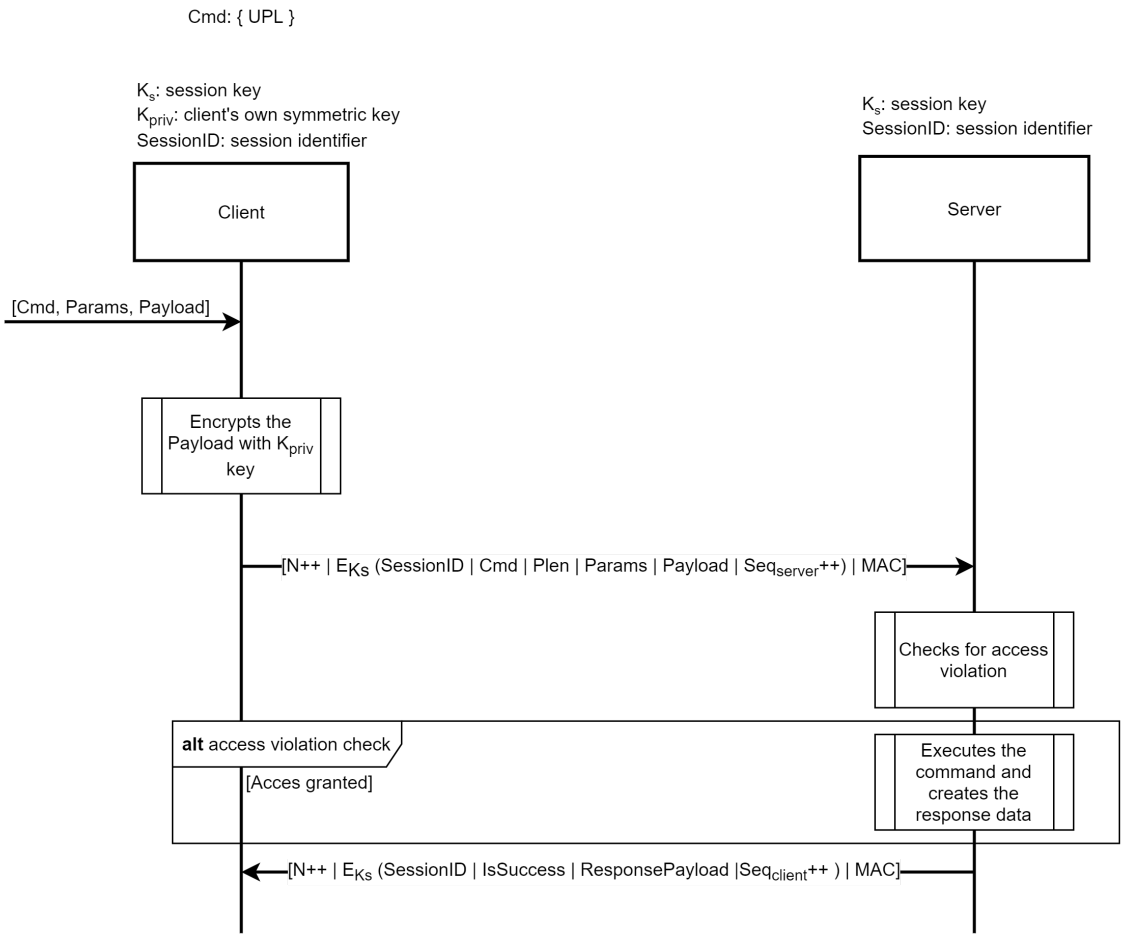
4.6. Upload / download command behaviour

In case of the upload command the file which is going to be uploaded is encrypted by the client beforehand with a key which is only known to the user. The server can't decrypt the file's actual content!

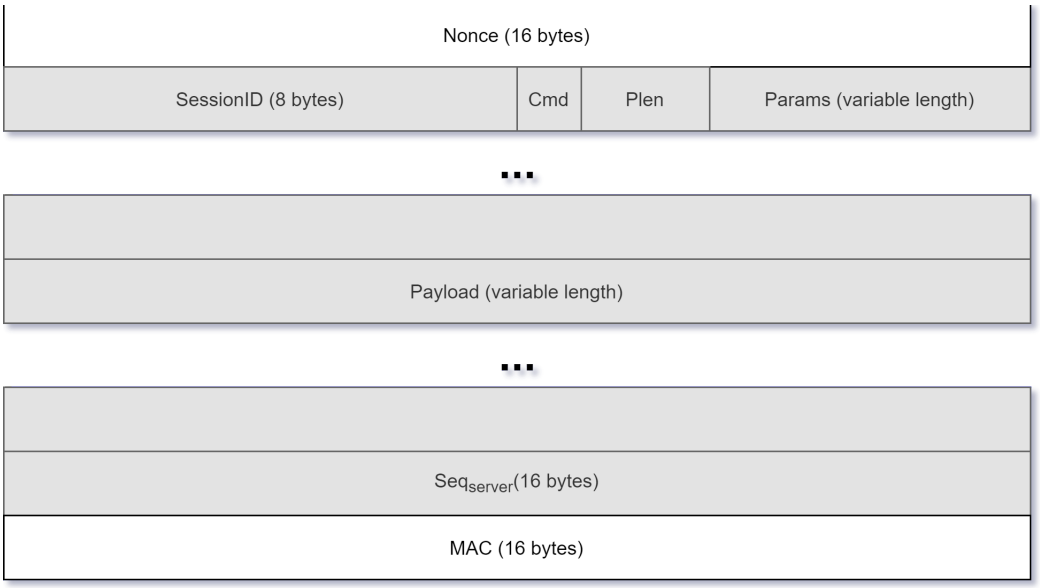
When the user wants to upload a file, he is required to give a password via the console interface. From this password the client generates a key with the Argon2id function. The key is then used with a random 16 byte IV to encrypt the user file using an AES-256 CBC cipher. The Argon2 parameters used for the key derivation, and the IV are stored in a file on the local file system for transportability. This file contains a list of SHA-512 hashes of the files encrypted on the client, and alongside each entry, the Argon2 parameters and the random IV used during the encryption of the files. The user is always required to give the password for the key generation, it is not stored in file!

The encrypted file is sent in the Payload section of the message, also encrypted with AES-256 GCM using the session key. The rest of the message follows the default command message format described previously.

Communication sequence:



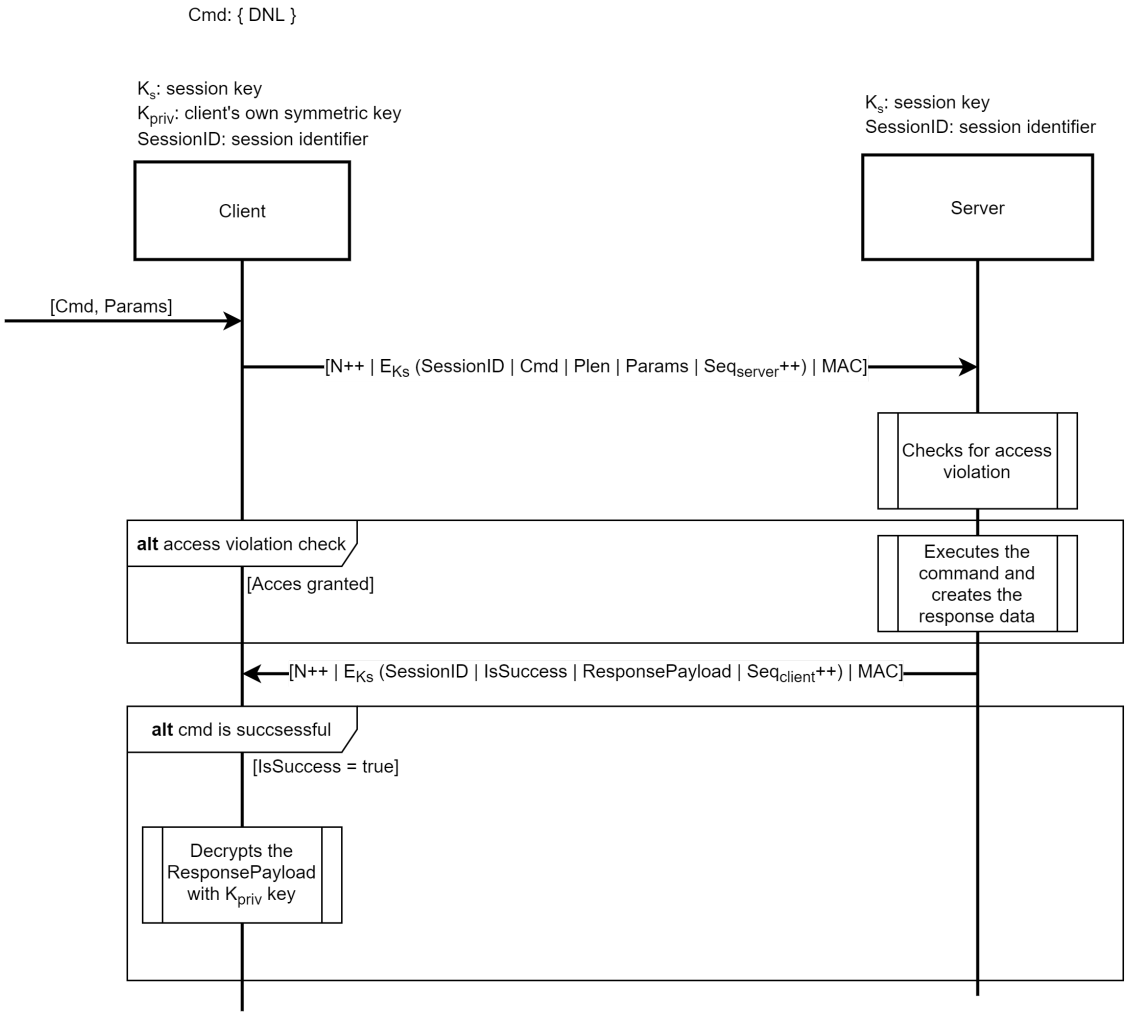
Message body:



When the server executes a download command, the selected file is sent back to the client in the message ResponsePayload section. To decrypt the file itself, the user is required to give the password which was used for the encryption of the file via the console interface, and the SHA-512 hash of the encrypted file has to be in the file storing the encryption information.

If the user gives the right password, the file is decrypted using the Argon2 key derived from the password and the parameters stored on the file system, and saved to the user's file folder on the client side, where it can be opened.

Communication sequence:



5. Evaluation

In this section, we aim to confirm that our protocol uses up-to-date techniques, fulfills the previously defined security requirements, and has no known vulnerabilities based on the attacker models we considered. We also discuss some ways to improve the security of our protocol, and cover its limitations.

5.1. Applied algorithms and techniques

Our protocol was in many ways inspired by the TLS 1.3 protocol. This newest iteration of TLS removed support for some older cryptography algorithms and key exchange protocols, so we looked at what protocols are still supported in this version. We wanted to use the most up-to-date techniques possible, but also wanted to keep the protocol simple, and implementable with only one or two different python libraries.

We chose to use elliptic-curve cryptography instead of traditional RSA because of its equivalent security with smaller key size, and because we wanted to learn more about its implementation details, and see how it performs in practice. An added benefit we realized while researching ECC techniques was the simplicity of the ECDHE protocol, making the key exchange process very easy to implement.

For the authenticated encryption of our main communication, we considered using ChaCha20-Poly1305, but ultimately chose AES-256 GCM.

We considered several possibilities for storing password hashes on the server, including scrypt, bcrypt, or 1 million rounds of PBKDF2, but we chose Argon2 due to it being a current state-of-the-art solution.

Overall, it can be said that we use some of the most up-to-date cryptography algorithms, but also carefully thought out where and how we will implement them, and what purpose they serve, since most of the time, it is the incorrect implementation and bad system design that causes security incidents, not the applied protocols being weak or outdated.

5.2. Security requirements

Our designed protocol fulfills all of the security requirements we set for this application.

Confidentiality:

1. The server must not store user passwords in plaintext, only as salted hashes.
    - **We store only the Argon2 hashes and salts of the passwords.**
2. The server must not store its private key in plaintext.
    - **The private key is encrypted with AES-256 CBC.**
3. The client and the server must establish a secure channel at the beginning of the communication (through the exchange of cryptographic keys).
    - **We use an Ephemeral ECDH key exchange.**
4. All messages must travel encrypted, through the secure channel.
    - **All sensitive information is encrypted with AES-256 GCM.**
5. All user files must be encrypted before they are sent to the server.
    - **User files are end-to-end encrypted with AES-256 CBC.**
6. The server must not have access to the decryption keys for the user files.
    - **The files' encryption password is never written to a file, and the KDF parameters are stored client side.**
7. Perfect forward secrecy should be ensured.
    - **This is ensured by using ECDH with ephemeral keys.**

Integrity:

8. Sequence numbering must be implemented for replay protection.
    - **We use both client and server side sequence numbers.**
9. Both parties must be able to verify that the received messages have not been altered in transit.
    - **All messages contain a MAC.**

Access control, Identification, Authentication, Authorization:

10. User authentication must be implemented for the login process.
    - **We implement password based authentication.**
11. The client must be able to authenticate the server.
    - **The server is authenticated using its digital certificate and by providing a signed, client-chosen proof.**
12. All messages must use a proper MAC to authenticate the sender.

- **All messages contain a MAC, a SessionID, and are encrypted using a key derived from a shared secret.**

13. Proper access control must be implemented for the users' files and folders.
    - **This is an implementation requirement, but everything is given to implement proper access control (Server/Client authentication, SessionID, MAC, ephemeral keys ...).**

Other security requirements:

14. Command inputs must be properly sanitized.
    - **This is an implementation requirement.**
15. Per-user command-rate limiting must be implemented.
    - **This is an implementation requirement.**
16. Key freshness must be ensured.
    - **This is ensured by using ECDH with ephemeral keys.**
17. The pseudo-random generation of nonces and sequence numbers must be implemented correctly, and the numbers must not be predictable.
    - **We use os.urandom, which is considered cryptographically secure, and does not require to be seeded by the developer.**
18. Edge-cases, like reaching the message limit for a given encryption key (limited to avoid IV reuse for the given key) should be handled correctly.
    - **This is an implementation requirement.**

5.3. Discussion and limitations

One important assumption we made while designing the protocol was that the communication channel has good quality of service, and packet loss and reordering is rare or non-existent. Essentially, we assumed the protocol to work above a TCP layer. If the channel would not have this quality, our sequence numbering system would need to be redesigned to use a sliding window for accepting packets.

Another relevant issue is the possible reuse of nonces. The client and the server increment the same nonce when exchanging messages during a session. If the client were to send messages rapidly, and not wait for the server response in between, one of the messages sent by the client would likely end up using the same nonce as the response sent by the server. For this reason, we assume our protocol to be heavily request-response-based, and we limit the sending of commands client-side until a response message arrives for the previous command. This issue could be

circumvented by using a different counter for our nonces on the client and on the server, which have no overlapping values (for example, one counting forward, one counting backwards from a given starting value).

A topic we discussed a lot during designing the protocol was the topic of client authentication. We considered implementing some kind of zero-knowledge-proof system, where the client does not have to send the password to the server, but such protocols were not supported by popular crypto libraries, and seemed too complex to implement for the purpose of this application.

Something else that could have improved security would have been the implementation of a two factor client authentication system, but this was outside our current scope.