Università di Pisa

Master Of Science in Computer Engineering

Foundations of Cybersecurity

Digital signature server

Francesco De Lucchini

Academic Year 2024/2025

Contents

1 Introduction

1.1 Problem description

A digital signature server is a **trusted third party that** creates private-public key pairs, stores them and **generates digital signatures on the behalf of the users**. This setup is particularly valuable in organizational environments where end users, such as employees, may not be trusted to securely manage their own cryptographic keys.

1.2 Security requirements

- **Users are registered off-line**. At registration, users receive the server's public key and a temporary password that must be changed at the first login.
- Users authenticate the server by means of the server's public key
- The server authenticates the users by means of their password
- The server stores users' private keys in encrypted form
- Users interact with the server through a **secure channel** that must be established before issuing operations. The secure channel must fulfill **confidentiality**, **integrity** (non-malleability), **perfect forward secrecy** and must be **resistant to replay attacks**

1.3 Software requirements

After securely connecting to the server, a user may invoke the following operations:

- **CreateKeys**: creates and stores a pair of private-public keys on behalf of the invoking user. If a key pair is already present, the operation has no effect
- **SignDocument**: digitally signs the specified document on behalf of the invoking user
- **VerifySignature**: given a document, a signature, and a public key, checks whether the document was digitally signed by the owner of the public key (computations are performed only on the client side)
- **GetPublicKey**: returns the public key of a specified user
- **DeleteKeys**: deletes the key pair of the invoking user. After a key pair has been deleted, an user can not create a new one unless it is (off-line) registered again

1.4 Technical implementation

The server (and a demo of the client) is implemented in `C` (in particular, `C11`), with most[1] of the security features being provided by the `OpenSSL` library (version `3.5.0`).

[1]The only exception is the hashing (and salting) of stored passwords, which is provided by the `Argon2id` algorithm implementation of `libsodium` (version `1.0.20`)

The objective of this project is not to build a production-ready digital signature server but rather to design an authentication protocol from scratch and implement it in a low-level language, which forces you to consider all the security details. For this reason, several simplifying design choices were made:

- **Blocking sockets**: the communication between a client and the server is implemented using blocking TCP sockets (messages must arrive with no errors and in the correct order)
- `Select` **syscall**: instead of using multi-threading, the server handles multiple clients at the same time by performing I/O multiplexing (`select` syscall) on a single thread
- **Perfect `send` and `recv` syscalls**: it is assumed that each `send` and `recv` syscall sends or receives the entire buffer (in a real-word scenario, a while loop should be used)
- **In-memory database**: the database of users is just an array of structures populated at the server's startup with hard-coded default values. In particular, there are two default users, 0 and 1, both with `password` as temporary password
- **On-disk private key**: the private key of the server is stored as an `AES-128` encrypted `PEM` file on the server's filesystem. The password is `server` and is required at boot

Finally, all the messages are sent through sockets in two steps:

1. First, the length of the message is sent as 4-byte big-endian integer
2. Then, the actual message is sent

2 Authentication protocol

2.1 Server authentication

2.1.1 Sequence diagram

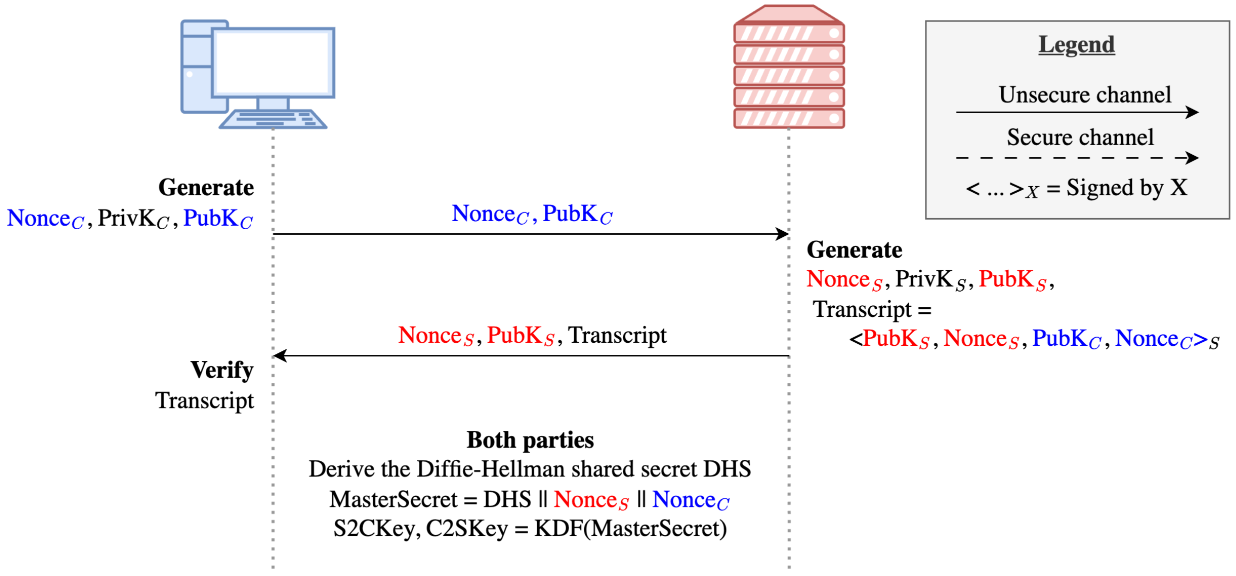

Legend
Unsecure channel
Secure channel
< ... >$_X$ = Signed by X
Generate
Nonce$_C$, PrivK$_C$, PubK$_C$
Nonce$_C$, PubK$_C$
Generate
Nonce$_S$, PrivK$_S$, PubK$_S$,
Transcript =
<PubK$_S$, Nonce$_S$, PubK$_C$, Nonce$_C$>$_S$
Nonce$_S$, PubK$_S$, Transcript
Verify
Transcript
Both parties
Derive the Diffie-Hellman shared secret DHS
MasterSecret = DHS || Nonce$_S$ || Nonce$_C$
S2CKey, C2SKey = KDF(MasterSecret)

Figure 1: Sequence diagram of the server authentication protocol

2.1.2 Design choices

The secure channel is established by using **ephemeral Diffie-Hellman** key exchange:

- Why Diffie-Hellman? Because the client and the server must share a secret in order to implement a secure channel (symmetric encryption)
- Why ephemeral? To satisfy the perfect forward secrecy requirement (Section 1.2): if, somehow, one of the two private keys gets compromised, only the messages exchanged in that session are at risk. To mitigate this issue even further, **sessions are dropped by the server every** 5 **minutes**, forcing a key refresh. This also makes sure that not too much data is encrypted with the same shared key, which is usually good practice (the less information an attacker has, the better)

After the client's hello message, the server replies with the **signature of the handshake transcript**. This proves to the client that:

1. Nonce$_C$ and PubK$_C$ were not modified during transit (Client $\rightarrow$ Server path)
2. Nonce$_S$ and PubK$_S$ were actually generated by the server and were not modified during transit (Server $\rightarrow$ Client path)

3. The response of the server is not being replayed, as the signature links both the server's response and the client's nonce

One might be wondering:

- Why is there a client's nonce (Nonce_C)? Since PubK_C is chosen randomly and is included in the server's signed response, isn't that sufficient to prove the freshness of the messages? Yes, it should be sufficient. However, the role of the Diffie-Hellman public key is to achieve a shared secret (hence, provide security), not to prove freshness. I find it better not to mix roles and have each component of the messages add a specific security feature to the protocol
- Why is there a server's nonce (Nonce_S)? It's usually good practice to have a nonce coming from both parties (imagine a single party with a defective/compromised RNG), moreover, along with the client's nonce, it adds randomness to the master secret

Finally, both parties compute the Diffie-Hellman shared secret, append it to the nonces, and use it derive the symmetric keys needed for the secure channel. The latter is implemented using **authenticated encryption**, as it checks all the requirements listed in Section 1.2. Another option could have been to perform the `encrypt-then-MAC` process manually, but the added complexity of the implementation was not worth the security benefit of using separate keys for encryption and authentication, since the lifetime of the keys is very short anyways.

One might be wondering: why are two keys (`S2CKey`, `C2SKey`) derived from the master secret? With a bi-directional key, an attacker could take a message going from the client to the server and bounce it back in the opposite direction, and, with no additional precautions (see the last point of Section 2.1.3), it would still be a valid message. Moreover, if, somehow, one of the keys gets compromised, only the traffic flowing in a single direction is at risk. Finally, having mono-directional keys allows each key to encrypt less data overall (as stated before, the less information an attacker has, the better).

2.1.3 Implementation details

- The Diffie-Hellman key exchange is implemented with the `X25519` elliptic curve (one of the curves recommended in `TLS 1.3`, which provides 128-bit security)[2] and 16-byte nonces. Public keys are sent through the socket in raw format (32-bytes fixed length)
- All signatures are made using `ECDSA` on the `ANSI X9.62 Prime 256v1` (a.k.a. `NIST P-256`) elliptic curve, which, again, targets the 128-bit security level
    - Why elliptic curves? They (allegedly[3]) enable strong security with smaller key sizes compared to traditional algorithms like `RSA`
- The key derivation function is `HKDF` based on `SHA-256`

[2]RFC 8446 - Section 4.2.7

[3]Should we trust the NIST recommended parameters?

- The authenticated encryption scheme is implemented using `AES-128-GCM`, as shown in Figure 2. In particular, the `AAD` field is always a 4-byte counter, incremented and checked by both the client and the server on each message. Without this authenticated counter, an adversary could reply a message within the same session (and the same direction)

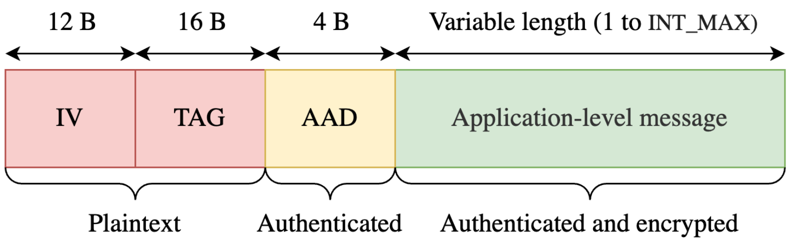

12 B
16 B
4 B
Variable length (1 to INT_MAX)
IV
TAG
AAD
Application-level message
Plaintext
Authenticated
Authenticated and encrypted

Figure 2: Composition of a message sent over the secure channel

2.2 Client authentication

2.2.1 Sequence diagram

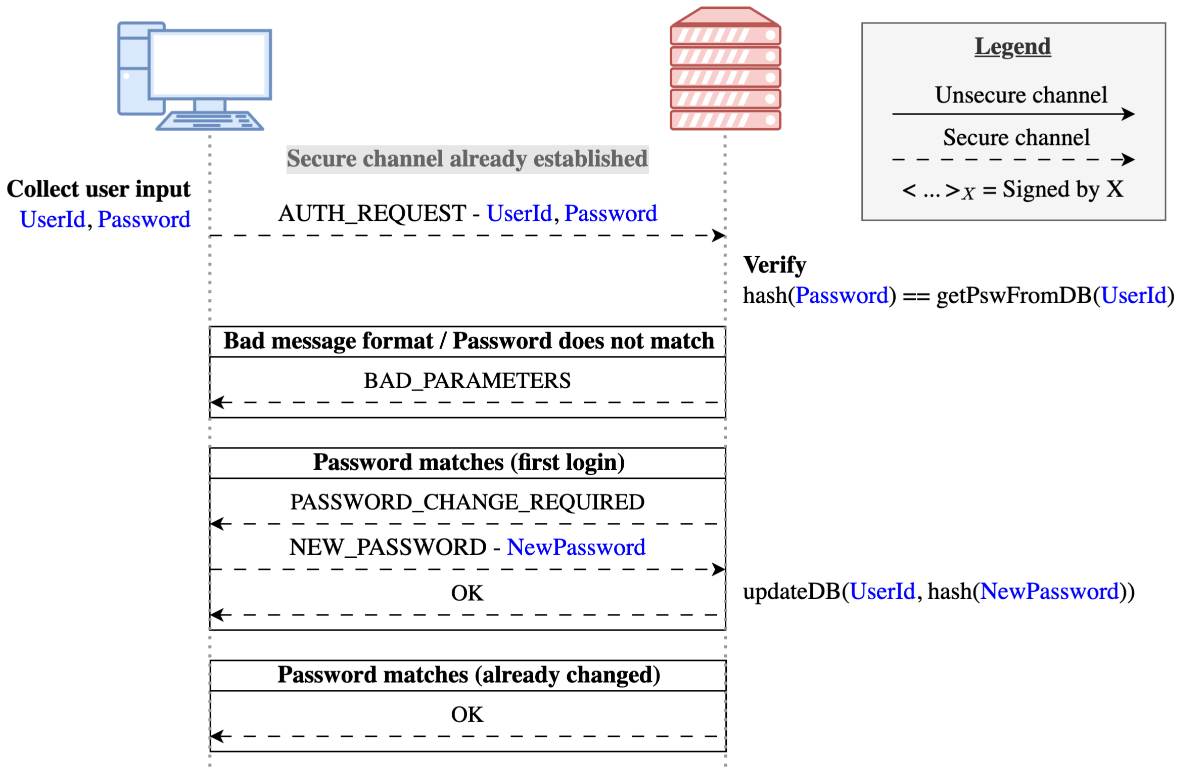

Legend
Unsecure channel
Secure channel
< ... >$_X$ = Signed by X
Secure channel already established
Collect user input
UserId, Password
AUTH_REQUEST - UserId, Password
Verify
hash(Password) == getPswFromDB(UserId)
Bad message format / Password does not match
BAD_PARAMETERS
Password matches (first login)
PASSWORD_CHANGE_REQUIRED
NEW_PASSWORD - NewPassword
OK
updateDB(UserId, hash(NewPassword))
Password matches (already changed)
OK

Figure 3: Sequence diagram of the client authentication protocol

2.2.2 Design choices

Once the secure channel is established, the client believes it is communicating with the real server. The server, however, does not yet know the identity of the client. To fix this, a **simple username/password authentication is implemented**: the client sends its (username, password) pair, and the server checks it against the ones stored in the database.

Once the authentication protocol finishes, the server has all the information needed to create a `Session` object, which links the communication socket with the authenticated client. In particular, **the `Session` object** (temporarily) **stores the plain-text password of the client**, as it is required to encrypt/decrypt the private key (as stated in Section 1.2, one of the requirements is that the private keys of the users are stored in encrypted form).

If, for any reason, one wants to avoid storing, even temporarily, the plain-text password of the client on the server, a possible alternative is to encrypt the private key with the hash of the password. It's crucial that the hashing algorithm used here is different from the one used for storage: if the same hashing algorithm is used, a database leak compromises all the private keys.

2.2.3 Implementation details

- As stated in Section 1.4, the hashing (and salting) algorithm for password storage is provided by the `Argon2id` implementation of `libsodium` (version `1.0.20`)
- With `AES-GCM`, the length of the cipher-text is equal to the length of the plaint-text. Without any precautions, this can leak the length of the users password to an eavesdropper. For this reason, **the client always sends a fixed amount of bytes** (32) **as a password**. The actual end of the password is marked by the null terminator (only visible once the message is decrypted)

3 Application logic

3.1 Messages

Every application-level message is composed by two fields:

- **Action**: encoded in the first byte, is either:
  - A COMMAND, issued by the client

    ```
    enum COMMAND
    {
        AUTH_REQUEST,
        NEW_PASSWORD,
        CMD_CREATE_KEYS,
        CMD_SIGN_DOC,
        CMD_GET_PUBKEY,
        CMD_DELETE_KEYS
    };
    ```

  - A RESPONSE, issued by the server

    ```
    enum RESPONSE
    {
        OK,
        PASSWORD_CHANGE_REQUIRED,
        BAD_CMD,
        BAD_PARAMETERS,
        NO_KEYS,
        KEYS_ALREADY_EXIST,
        TIMED_OUT
    };
    ```

- **Parameters**: optional, encoded from the second byte onwards

3.2 Operations

In this section, the implementation of each operation listed in the software requirements (Section 1.3) is briefly described:

- **CreateKeys**
  - Action: `CMD_CREATE_KEYS`
  - Parameters: None
  - Implementation details: private `ECDSA` keys are serialized as `AES-128` encrypted `PEM` files and stored in the database (memory) as strings
- **SignDocument**
  - Action: `CMD_SIGN_DOC`

  - Parameters: The **hash** of the document to sign
  - Implementation details: to minimize bandwidth usage, users do not actually send the whole documents but their hash. A consequence of this is that the server does not know the content of the signed documents. This is not a problem since the server signs on behalf of the invoking user. The signature computed by the server is saved on the client's filesystem as `{document}_signed.bin`

- **VerifySignature**
  - Action: -
  - Parameters: -
  - Implementation details: computations are performed only on the client side

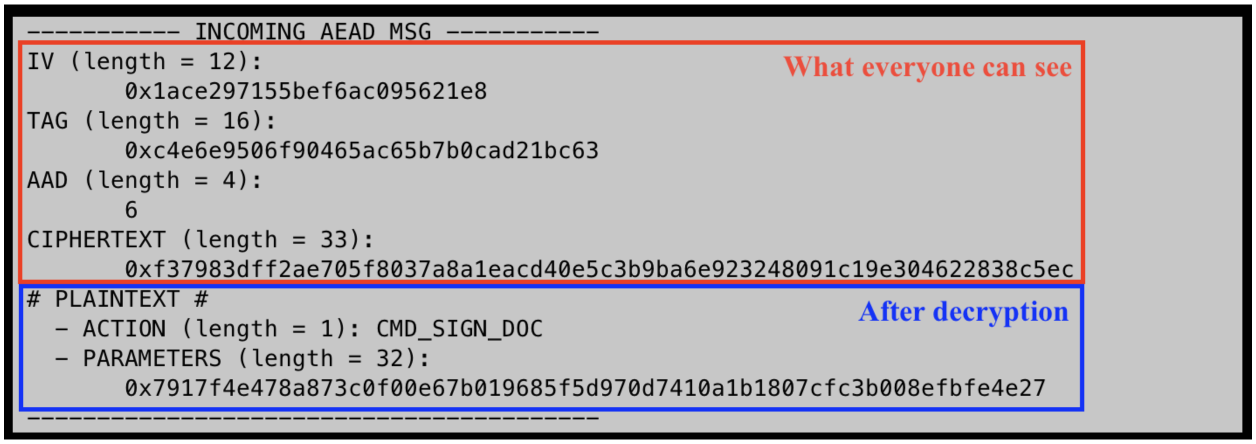
- **GetPublicKey**
  - Action: `CMD_GET_PUBKEY`
  - Parameters: `userId`
  - Implementation details: The received public key is saved on the client's filesystem as `public_key_{userId}.pem`

- **DeleteKeys**
  - Action: `CMD_DELETE_KEYS`
  - Parameters: None
  - Implementation details: Nothing particular

4 Traffic analysis

In this section, I briefly analyze the traffic between the client and the server to check that everything works as expected. As an example, the `SignDocument` operation is analyzed:

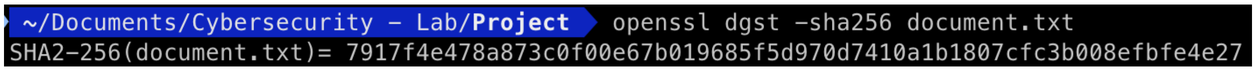
- Client request: as expected, after decryption, the server sees the `CMD_SIGN_DOC` command with a 256-bit wide parameter



```
----------- INCOMING AEAD MSG -----------
IV (length = 12):
       0x1ace297155bef6ac095621e8
TAG (length = 16):
       0xc4e6e9506f90465ac65b7b0cad21bc63
AAD (length = 4):
       6
CIPHERTEXT (length = 33):
       0xf37983dff2ae705f8037a8a1eacd40e5c3b9ba6e923248091c19e304622838c5ec
# PLAINTEXT #
  - ACTION (length = 1): CMD_SIGN_DOC
  - PARAMETERS (length = 32):
       0x7917f4e478a873c0f00e67b019685f5d970d7410a1b1807cfc3b008efbfe4e27
-----------------------------------------
```

Figure 4: Sample client request for the `SignDocument` operation

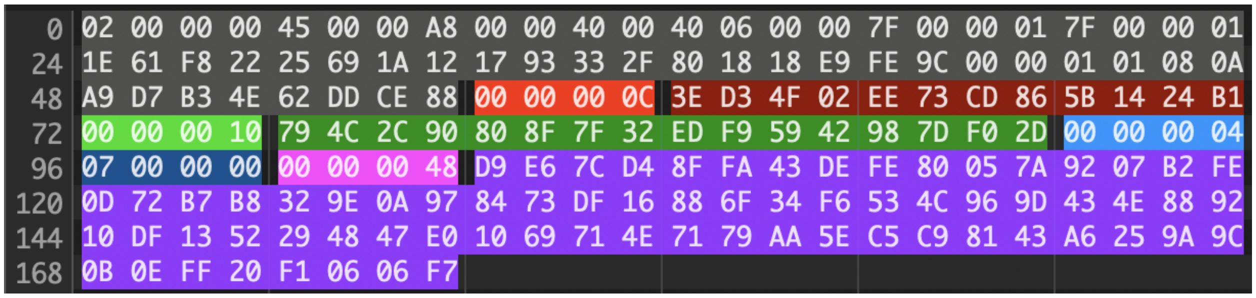
- The 256-bit wide parameter sent by the client actually corresponds to the `SHA-256` of the document to sign



```
~/Documents/Cybersecurity - Lab/Project  openssl dgst -sha256 document.txt
SHA2-256(document.txt)= 7917f4e478a873c0f00e67b019685f5d970d7410a1b1807cfc3b008efbfe4e27
```

Figure 5: SHA-256 hash of the document used in Figure 4

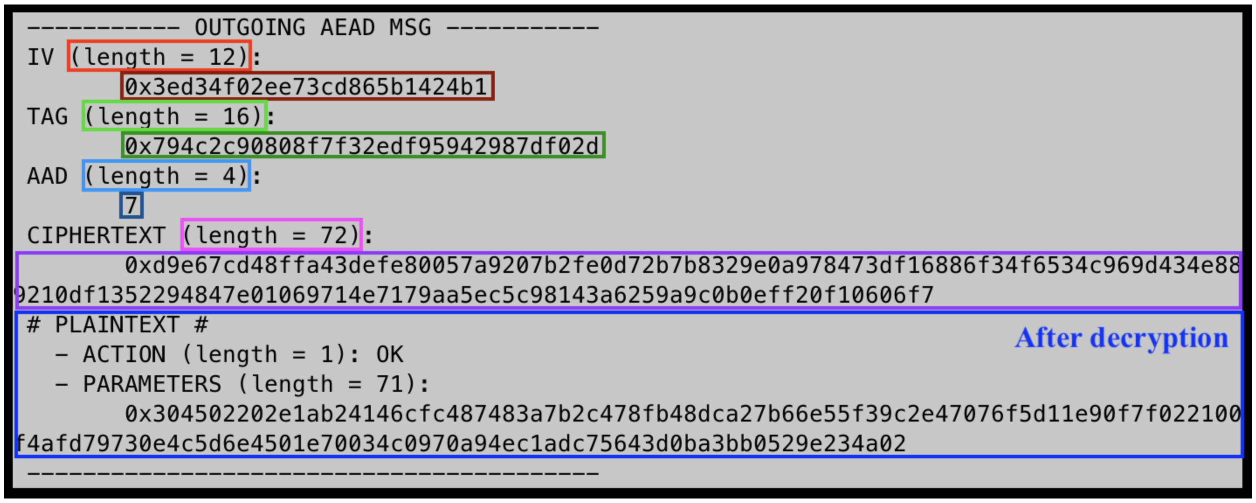
- Server's response, captured by **Wireshark** (grey bytes are TCP headers)



```
  0 02 00 00 00 45 00 00 A8 00 00 40 00 40 06 00 00 7F 00 00 01 7F 00 00 01
 24 1E 61 F8 22 25 69 1A 12 17 93 33 2F 80 18 18 E9 FE 9C 00 00 01 01 08 0A
 48 A9 D7 B3 4E 62 DD CE 88 00 00 00 0C 3E D3 4F 02 EE 73 CD 86 5B 14 24 B1
 72 00 00 00 10 79 4C 2C 90 80 8F 7F 32 ED F9 59 42 98 7D F0 2D 00 00 00 04
 96 07 00 00 00 00 00 00 48 D9 E6 7C D4 8F FA 43 DE FE 80 05 7A 92 07 B2 FE
120 0D 72 B7 B8 32 9E 0A 97 84 73 DF 16 88 6F 34 F6 53 4C 96 9D 43 4E 88 92
144 10 DF 13 52 29 48 47 E0 10 69 71 4E 71 79 AA 5E C5 C9 81 43 A6 25 9A 9C
168 0B 0E FF 20 F1 06 06 F7
```

Figure 6: Wireshark capture of the message described in Figure 7

- Server's response, logged by the server: as expected, the captured packet and the logs match. Moreover, the `AAD` (sequence number) is correctly incremented with respect to the client's request



```
----------- OUTGOING AEAD MSG -----------
IV (length = 12):
       0x3ed34f02ee73cd865b1424b1
TAG (length = 16):
       0x794c2c90808f7f32edf95942987df02d
AAD (length = 4):
       7
CIPHERTEXT (length = 72):
       0xd9e67cd48ffa43defe80057a9207b2fe0d72b7b8329e0a978473df16886f34f6534c969d434e88
9210df1352294847e01069714e7179aa5ec5c98143a6259a9c0b0eff20f10606f7
# PLAINTEXT #
  - ACTION (length = 1): OK
  - PARAMETERS (length = 71):
       0x304502202e1ab24146cfc487483a7b2c478fb48dca27b66e55f39c2e47076f5d11e90f7f022100
f4afd79730e4c5d6e4501e70034c0970a94ec1adc75643d0ba3bb0529e234a02
-----------------------------------------
```

Figure 7: Server's response to the `SignDocument` operation issued in Figure 4