UNIVERSITY OF PISA

MSc in Computer Engineering

Foundations Of Cybersecurity

Digital Signature Server

Professors:

Prof. Gianluca Dini

Students:

Antonio Andrea Salvalaggio (646238)

Andrea Di Matteo (641388)

ACADEMIC YEAR 2024/2025

Contents

1 Introduction 3
- 1.1 Security requirements . . . 3
- 1.2 Session establishment and user authentication . . . 3
- 1.3 Available actions . . . 4
  - 1.3.1 CreateKeys . . . 4
  - 1.3.2 SignDoc . . . 4
  - 1.3.3 GetPublicKey . . . 4
  - 1.3.4 DeleteKeys . . . 4
  - 1.3.5 Quit . . . 4

2 Cryptographic Protocols 5
- 2.1 Key Exchange Protocol . . . 5
- 2.2 Application Level Protocol . . . 7
- 2.3 User Authentication . . . 8
- 2.4 Key creation and document signing . . . 9

3 Exchanged Messages 9
- 3.1 Key Exchange . . . 9
- 3.2 Application Messages . . . 9
- 3.3 Size Limits . . . 10

1 Introduction

This project implements a Digital Signature Service (DSS) for an organization's employees: a trusted third party that creates public-private key pairs, stores them and generates digital signatures on behalf of the organization's employees.

1.1 Security requirements

Clients communicate with the DSS server using a secure channel that fulfills the following requirements:

- Perfect Forward Secrecy: session keys are not derived form long term shared secrets, but are established with Ephemeral DH;
- Server authentication: the server is authenticated during channel establishment using a pre-dristributed public key;
- Client authentication: users are authenticated at application level using passwords;
- Integrity and non-malleability: every message exchanged is encrypted and authenticated using AES-256-GCM (encrypt then authenticate);
- No replay attacks: a new symmetric key is generated for each session and messages within the same session are encrypted using an incremental IV that guarantees freshness.

1.2 Session establishment and user authentication

At startup each client establishes a secure channel with the server using Ephemeral ECDHKE to derive a shared secret. During the exchange the server authenticates itself by sending an RSA digital signature. The server's public key is distributed offline to all employees upon registration. The server's public and private keys can be generated using the following `openssl` commands:

- `openssl genrsa -out priv_server.pem`
- `openssl rsa -pubout -in priv_server.pem -out pub_server.pem`

A shared key is derived from the session's shared secret and all further communications are made using a secure channel that uses symmetric encryption (AES-256-GCM).

Before running any command the client needs to authenticate itself by sending a `username-password` pair. The server stores user passwords salted and hashed with SHA-256. Employees receive a password upon registration, but at first login they are required to change it.

The DSS server is designed to handle multiple clients concurrently using multiple threads and mutexes to synchronize access to stored files and shared data structures.

1.3 Available actions

After logging in, and if needed changing password, employees can use the following commands.

1.3.1 CreateKeys

Requires a passphrase as an argument. The service creates and stores a pair of private and public keys on behalf of the invoking employee. If a key pair for the employee already exists or it has been deleted, the command has no effect. The passphrase is used to encrypt the private key and is not stored anywhere on the server, so it will be necessary to provide it again when signing documents.

1.3.2 SignDoc

Requires two arguments: the employee's passphrase and the path to the file to sign. The file is uploaded to the server. The service digitally signs the uploaded document on the invoking user's behalf and returns the resulting digital signature. The digital signature is saved in a file in the same location as the original one.

1.3.3 GetPublicKey

Requires a username as an argument. Returns the public key for the corresponding employee if it is stored on the DSS.

1.3.4 DeleteKeys

Deletes the key pair of the invoking employee. After a key pair has been deleted, an employee cannot create a new one unless the account is eliminated and (off-line) registered again.

1.3.5 Quit

Ends the session and closes the application.

2 Cryptographic Protocols

2.1 Key Exchange Protocol

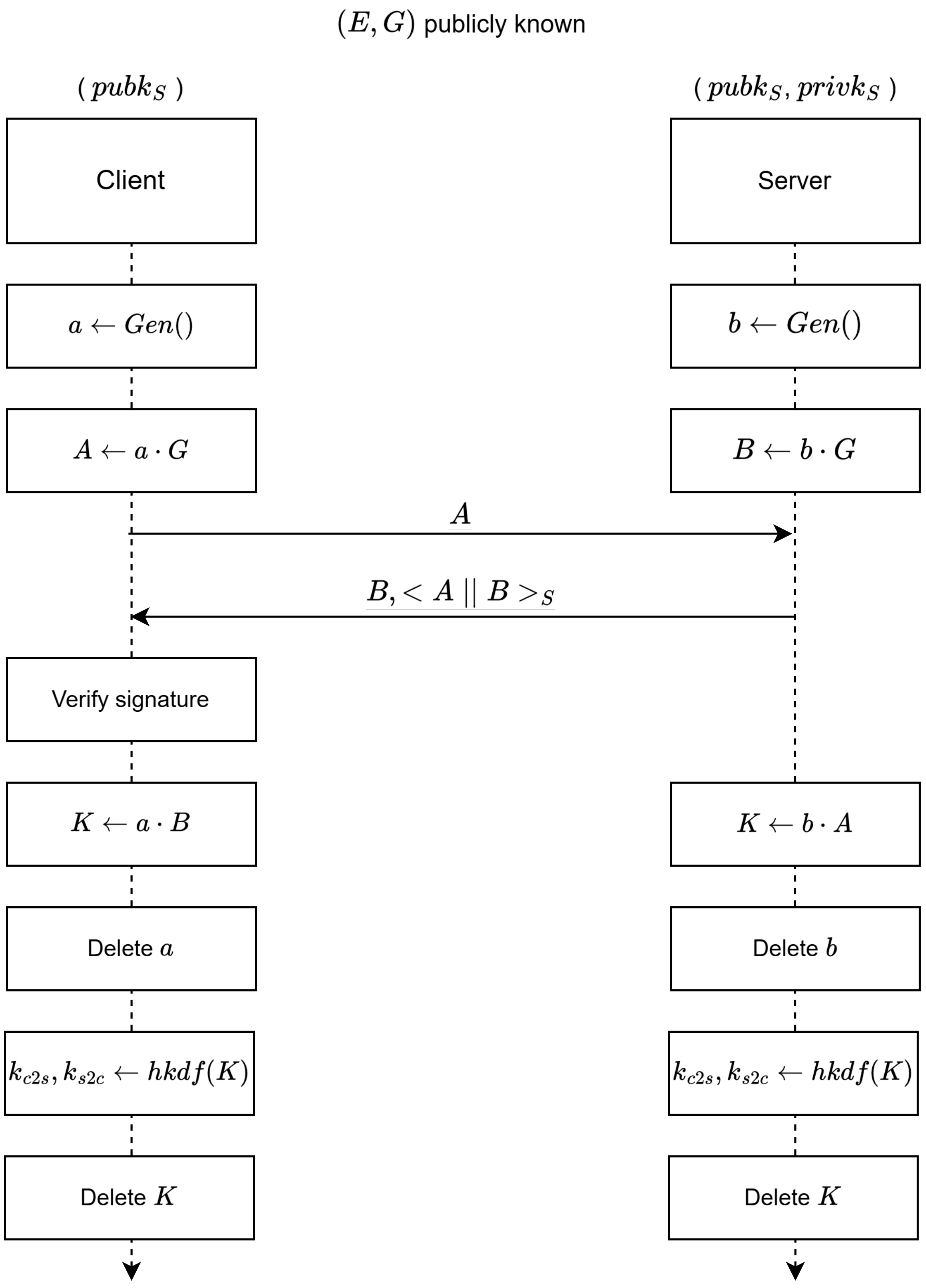

Figure 1: Ephemeral Diffie-Hellman Key Exchange Protocol

The key exchange between the client and the server, shown in fig. 1, follows these steps:

1. Both parties use the same public parameters: an elliptic curve E (in our case `prime256v1`) and generator G;
2. Each party randomly generates an ephemeral private key before each new session: the client generates scalar a, the server generates scalar b;
3. Each party computes their public key: point $A = a \cdot G$ for the client, and point $B = b \cdot G$ for the server;
4. The client sends A to the server;
5. The server computes the signature of $A||B$ using its private key $privk_S$;
6. The server answers sending B and the signature $< A||B >_S$;
7. The client verifies the signature using $pubk_S$ (off-line distributed);
8. Each party computes the shared secret using their private key and the other party's public key: for the client $K = a \cdot B = a \cdot b \cdot G$, while for the server: $K = b \cdot A = b \cdot a \cdot G$;
9. Both parties delete their ephemeral private key (a and b respectively);
10. Both parties derive symmetric encryption keys using HKDF (HMAC-based Key Derivation Function) from the shared secret.

The derived encryption keys (which include the IV prefix) are then used with AES-256-GCM to provide confidentiality, integrity and non-malleability on the channel.

The protocol guarantees perfect forward secrecy as a and b are randomly generated for each session and are not stored on any device after the handshake has been completed.

The signature verification allows the client to authenticate the server. Both A and B are included in the signature in order to prevent man-in-the-middle attacks. As A is a fresh quantity this also prevents replay attacks.

2.2 Application Level Protocol

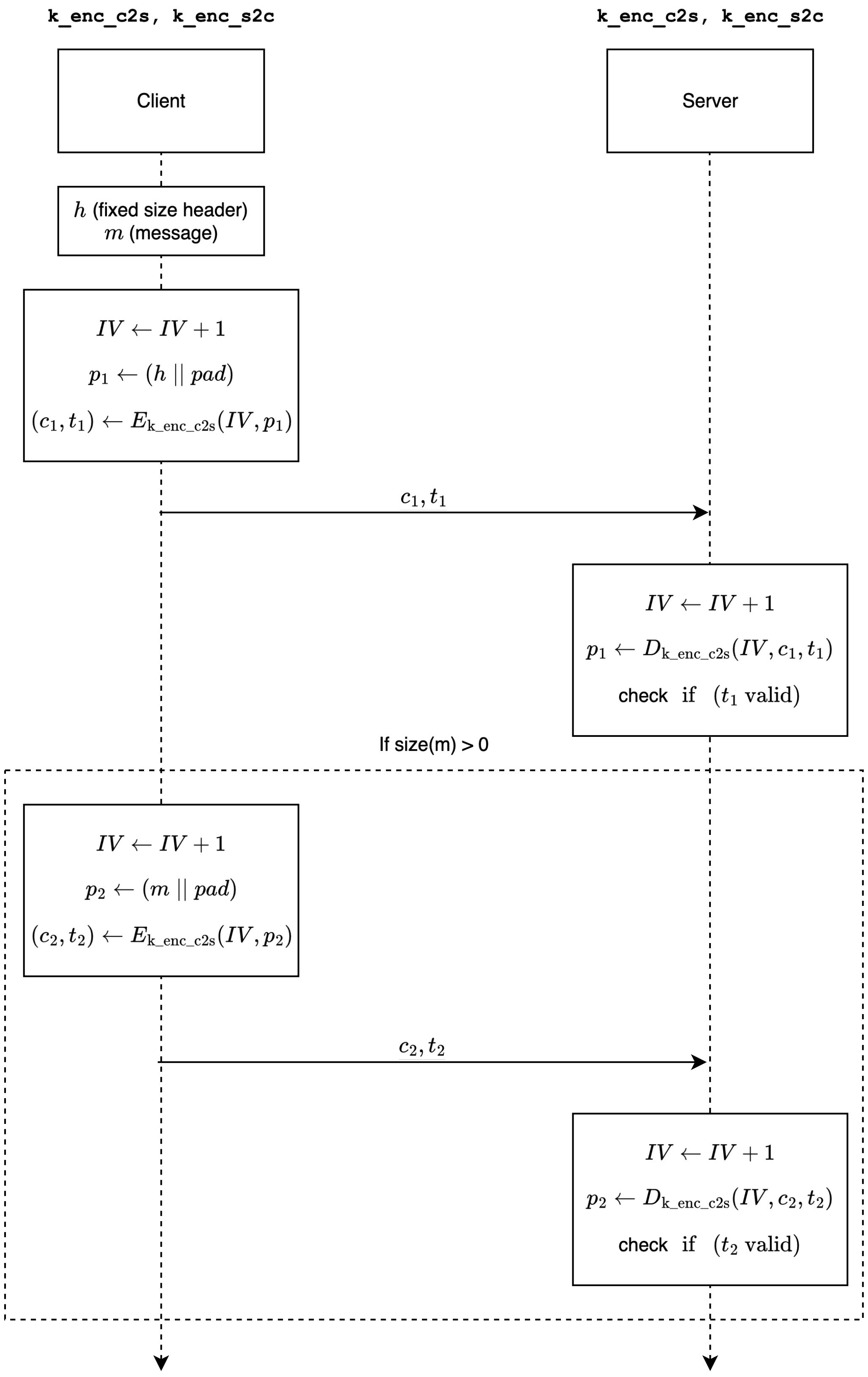
k_enc_c2s, k_enc_s2c
k_enc_c2s, k_enc_s2c
Client
Server
h (fixed size header)
m (message)
$IV \leftarrow IV + 1$
$p_1 \leftarrow (h \,||\, pad)$
$(c_1, t_1) \leftarrow E_{\text{k\_enc\_c2s}}(IV, p_1)$
c_1, t_1
$IV \leftarrow IV + 1$
$p_1 \leftarrow D_{\text{k\_enc\_c2s}}(IV, c_1, t_1)$
check if $(t_1$ valid$)$
If size(m) > 0
$IV \leftarrow IV + 1$
$p_2 \leftarrow (m \,||\, pad)$
$(c_2, t_2) \leftarrow E_{\text{k\_enc\_c2s}}(IV, p_2)$
c_2, t_2
$IV \leftarrow IV + 1$
$p_2 \leftarrow D_{\text{k\_enc\_c2s}}(IV, c_2, t_2)$
check if $(t_2$ valid$)$

Figure 2: Client-to-server message protocol on the secure channel.

All client-to-server communication follows the following protocol:

1. The client increments the value of the counter (IV).
2. The clients prepares the message p_1 to send: $p_1 = h||padding$. The header h is of fixed size and contains the payload length. Details about all the contents of the header can be found in section 3.2. `PKCS#7` padding is added at the end.
3. The client uses AES-256-GCM with the encryption key for client-to-server communication to encrypt p_1 and generate the MAC tag.
4. The server increments the stored IV.
5. The server verifies the MAC tag, deciphers the message and verifies the correct format of the padding.
6. OPTIONAL: if the length of the payload is not zero points 1 to 5 are repeated using the variable-sized payload instead of the fixed-size header.

After receiving the message header and, optionally, its payload the server can process the request accordingly and send an answer back to the client. The same protocol is applied in reverse for server-to-client communication.

The IV is composed of a 4 byte random prefix derived from the shared secret at the start of the session and an 8 byte counter that gets incremented with each message. The IV is not sent together with the message, but its independently computed by client and server. It thus can be used to guarantee non-replayability within the session itself (replayability across sessions is prevented by regenerating the symmetric keys).

2.3 User Authentication

After the secure channel has been established, the server expects the first application message received each session to contain the user's login information.

Both usernames and passwords can be up to 32 characters long and can contain letters, numbers and, only for passwords, a limited set of symbols.

Usernames and passwords (hashed and salted) for all clients are saved along with the salt used and wether or not the user has already changed its password.

Until the client sends a valid username-password pair, no other operation is allowed on the session. If a session has already been started for a given user, no new logins will be allowed on that account until the session is closed. After the login, if the password has not yet been changed, the only available operation is the change password one (with only one argument: the new password).

2.4 Key creation and document signing

The key creation command is only possible if no key files (either valid or empty) for the current user are present. The generated keys are saved in a pair of `.pem` files for the current user. The signature algorithm used, and for which keys are generated, is 2048-bit RSA.

The key creation operation as the only argument requires a passphrase with the same maximum size and other restriction as the user's password. The passphrase is used to encrypt the private key using AES-256-CBC.

For document signing the client must send the server both the passphrase to decipher its own private key and the file itself. The signature is computed using 2048-bit RSA on a SHA-256 digest of the file and then sent back to the client.

Key deletion overwrites the key files of the current user saved on the server with empty files.

3 Exchanged Messages

3.1 Key Exchange

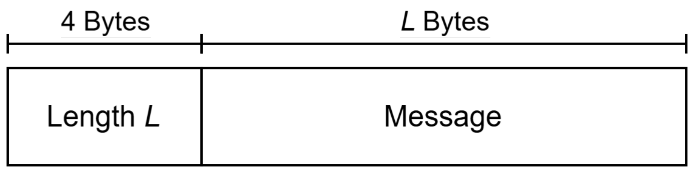

Figure 3: Format of the unencrypted messages sent during secure channel establishment.

Messages exchanged during the secure channel establishment are unencrypted and unauthenticated. They follow the format displayed in fig. 3. The length cannot be longer than the maximum message length allowed by the application.

3.2 Application Messages

Application messages are divided into two parts:

- the header which contains information about the operation that is being performed, error codes and payload length;
- the payload which contains request arguments and responses.

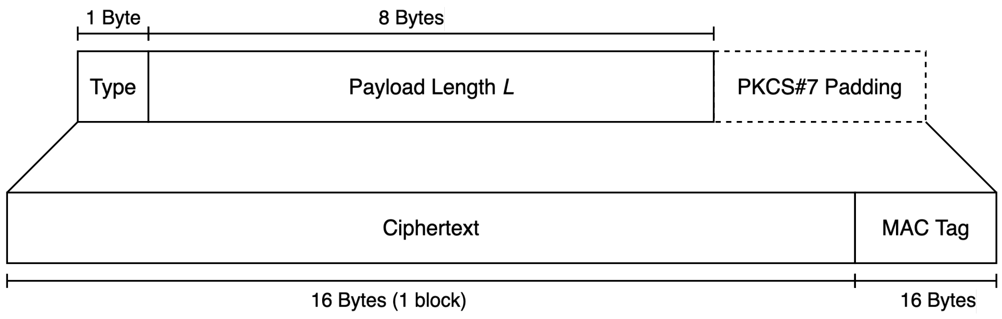

Figure 4: Format of the header message used for all application communications.

Figure 4 shows the format of the fixed-size header message. The type byte encodes the following information:

- Bits 0-1: codes to identify the four possible operations (00: CreateKeys, 01: SignDoc, 10: GetPubKey, 11: DeleteKeys). Login and change password operations do not require an operation code as they can, and must, be performed only at startup.
- Bit 2: Error flag: in a response if it is set to 0 signifies a successful operation or an error otherwise.
- Bits 3-7: Error code that specifies the cause of the error when the error flag is equal to 1. Should be equal to 0 unless the error flag is raised.

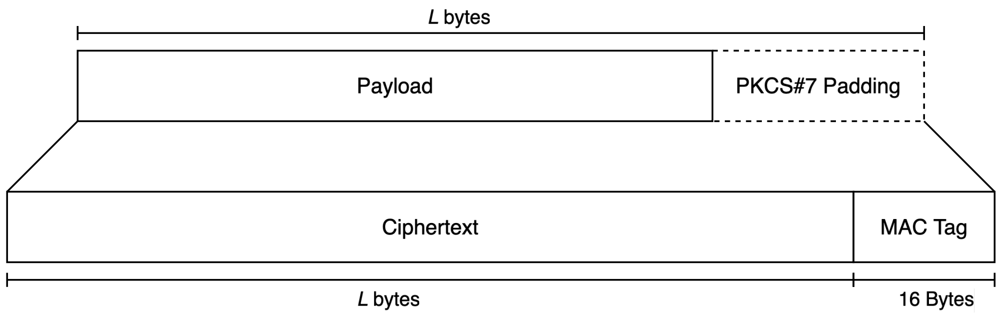

Figure 5: Format of the optional payload message that follows the header message in application communications.

Figure 4 shows the format of the optional payload message. Arguments inside the payload are always either of fixed size (usernames and passwords on 32 bytes) or as long as the payload itself (documents and responses), thus no additional information to distinguish them is necessary.

3.3 Size Limits

At application level, the service imposes limitations on the input parameters:

- `MAX_FIELD_SIZE`: Username and Password fields cannot exceed 32 bytes each;
- `MAX_FILE_SIZE`: the document can never exceed 10MB size.

The `createKeys` request has only a 32 bytes passphrase payload. The `deleteKeys` request and response have no payload, whereas `getPublicKyes` request sends a username as payload (64 bytes total) and gets a public key (2048 bits) as response payload. Finally the `signDoc` request sends a document (limited by `MAX_FILE_SIZE`) and gets a fixed size digital signature (2048 bit). For this reason, no additional parameters are needed.