UNIVERSITÀ DI PISA

Computer engineering

Computer Architecture

FoC project documentation

Francesco Berti

Alessandro Versari

ANNO ACCADEMICO 2022/2023

Contents

1 Design choices **1**

2 Protocol **1**
2.1 Application level 2
2.2 Security level 4
2.2.1 Handshake 4
2.2.2 Session 5

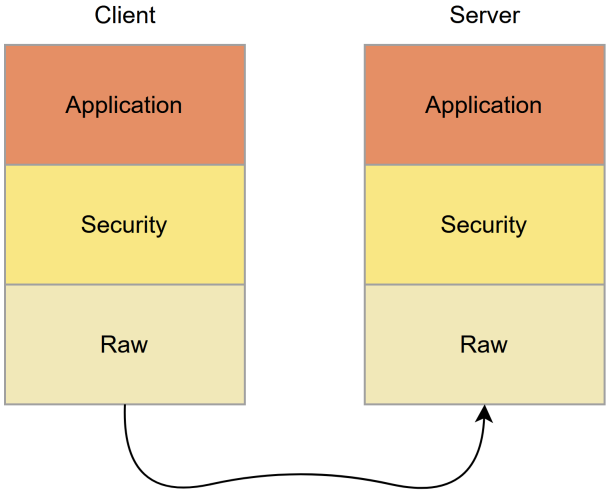
3 Security module **5**
3.1 AsymCrypt 5
3.2 SymCrypt 5
3.3 Hmac 6
3.4 Key handling methods 6

1 Design choices

- Confidentiality: is guaranteed by using AES 256 CBC for every message exchanged in the session. In particular, for every message encrypted with AES a new IV is created and appended at the beginning of the message as well as a progressive counter to avoid potential replay attacks.
- Integrity: in the last step of the handshake protocol the server sends to the client the encrypted MAC of all the messages exchanged during the handshake. The client then sends back to the server the MAC of all handshake's messages, including the MAC of the server. Both MACs are encrypted by means of the session key to achieve key authentication and verification. For every message of the session the "Encrypt than MAC" scheme is used, guaranteeing integrity and avoiding traffic analysis (since the ciphertext will change even if the message is the same).
- No-replay: thanks to the use of DHKE the handshake protocol cannot be replayed. Every message of the session has a progressive counter, furthermore the server checks if a user is logged in before processing a request.
- Non-malleability: for every session message a new IV and the relative MAC are generated so even if the message is manipulated the receiver can detect that
- PFS: achieved by means of the STS protocol
- Encrypted DB: the database of the users, including their passwords, and the history of transfers are stored in encrypted json files using AES 256 CBC. Passwords are stored inside the encrypted json in "hash and salt" form using SHA3-512.

2 Protocol

The following image illustrates a look alike OSI model for our application, it is composed by three layers:



- Application: at this level messages are encoded as a JSON string which makes the protocol human readable, easier to implement and to modify. The client makes request to the server which contain a *route* and a non mandatory *content*, every response contains a custom status code and a non mandatory *content*.
- Security: this level is responsible for implementing the handshake and symmetrical encryption. Both client and server receive this level as dependency in order to be interchangeable with other security implementations. The programmer can also opt out the security module and use a "fake" security module, this is useful to test the application during development.

- Raw: this level is responsible to send and receive data (encrypted or not).

Everyone of this protocol are stateful, they use the same TCP connection over multiple requests and the user is identified via the socket descriptor.

2.1 Application level

These are the possible requests the client can issue:

- Login:

```
{
    "route" : "login",
    "content" : {
        "username" : "name",
        "password" : "password"
    }
}
```

- Balance:

```
{
    "route" : "balance"
}
```

- Transfer:

```
{
    "route" : "transfer",
    "content" : {
        "beneficiary" : "name",
        "amount" : 10.20
    }
}
```

- History:

```
{
    "route" : "history"
}
```

Inside the server there is a router which uses the route field inside every requests to decide handler to start. In case of bad request a status code 400 is returned. Every response has a status code, in case of error the response are formatted like this:

- ```
  {
      "status" : 400,
      "message" : "error␣message"
  }
  ```

The following are the possible responses the server can send to the client:

- Login:

  ```
  {
      "status" : 200|401,
  }
  ```

- Balance:

  ```
  {
      "status" : 200,
      "balance" : 10.10,
      "accountID" : "efjkb32ui",
  }
  ```

- Transfer:

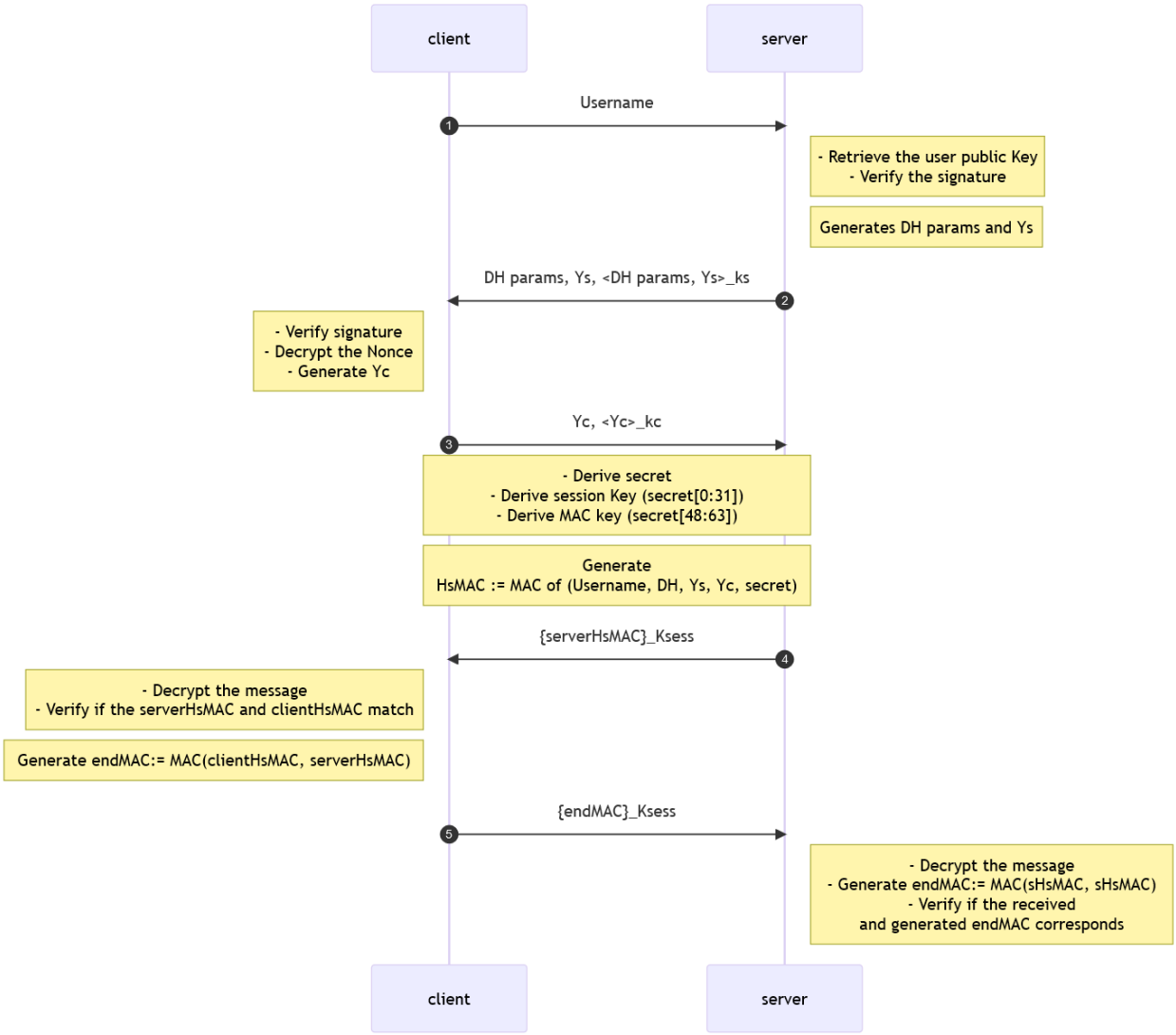
  ```
  {
      "status" : 200|400,
  }
  ```

- History:

  ```
  {
      "status" : 200,
      "history" : [
          {
              "amount" : 10.20,
              "from" : "user1",
              "to" : "user2"
          }
      ]
  }
  ```
  ```

2.2 Security level

2.2.1 Handshake



The client starts the interaction at the login, a message containing the username of the user is sent to the server. The server generates DH parameters (g and p) and a DH public key (Ys), then proceeds to use his private key to sign both parameters and send them to the client.
The client verifies the signature and uses the parameters to create a DH public key (Yc) which is signed and sent to the server. The server will verify the signature of the client, at this point each peer will have authenticated the other.
Both client and server use the public DH keys to derive the secret and then proceed to extract the session and MAC keys using the dedicated API.

The server generates the MAC of all the messages exchanged in the handshake, including the derived secret, and encrypts the MAC by means of the session key. The client decrypts and checks if his/her computed MAC is equal to the one received, otherwise closes the connection.
The client then generates a MAC using as input all the message exchanged until now, including the MAC of the server, then encrypts it by means of the session key and sends it to the server. The latter will decrypt and verify if the MAC matches. By doing this both peers will have:

- authenticated and verified the keys
- verified the integrity of the handshake

2.2.2 Session

Each message of the session includes a progressive counter, a random IV, the ciphertext and the MAC of all the previous fields. The progressive counter is used to avoid replay attacks, the random IV ensures different ciphertexts for the same messages (avoid traffic analysis), the MAC ensures the integrity.

3 Security module

The security module implements all the APIs used to ensure the confidentiality, integrity and authenticity. It's composed of:

- Three different classes: AsymCrypt, SymCrypt and Hmac which implement RSA, AES 256 CBC and MAC respectively
- A method to encode and one to decode EVP_PKEY objects to be transmitted over a socket, a method to encode DH parameters, sanitization methods to use with the user's inputs

The following sections will explain each class and method in detail:

3.1 AsymCrypt

This class implements a RSA digital signature instance:

- `AsymCrypt(string path_private_key, string path_public_key_peer, string password);`

  the constructor takes in input the path of the *.pem* file where the encrypted private key is stored, the path of the *.pem* file where the public key of the peer is stored and the password used to encrypt the private key.

- `void setPeerKey(string path_public_key_peer);`

  used to set the path of the *.pem* file containing the peer public key

- ```
  tuple<vector<uint8_t>, entity::Error> sign(vector<uint8_t> mess);
  tuple<vector<uint8_t>, entity::Error> verify(vector<uint8_t> msg,
      vector<uint8_t> signature);
  ```

  both the sign and verify methods take in input and return a vector of *uint8_t*, in case of error a custom error code is returned

3.2 SymCrypt

This class implements an AES 256 CBC cipher instance:

- `SymCrypt(unsigned char *key);`

  the constructor takes in input the shared key to be used in this instance

- `tuple<vector<uint8_t>, entity::Error> encrypt(vector<uint8_t> plaintext);`
  ```

the encrypt method generates a new IV and encrypts the plaintext, returns the concatenation of IV and ciphertext

- `tuple<vector<uint8_t>, entity::Error> decrypt(vector<uint8_t> ciphertext);`

  the decrypt method uses the first 16 bytes of ciphertext as the IV and the remaining bytes as the actual ciphertext, then decrypts and returns the plaintext

3.3 Hmac

This class is used to generate MAC:

- `Hmac();`
  `Hmac(vector<uint8_t> key);`

  the empty constructor automatically generates a symmetric key, the second constructor set the key to the one passed as argument

- `vector<uint8_t> getKey();`

  method used to return the key of the Hmac instance

- `tuple<vector<uint8_t>, entity::Error> MAC(vector<uint8_t> data);`

  generates the MAC for data

3.4 Key handling methods

The following methods can be used to generate, encode and decode keys and parameters:

- `entity::Error generateRSAkeys(string path, string password, unsigned int bits);`

  generates a pair of private and public key with modulus size of *bits* and save them to path, uses *password* to encrypt the private key. *path* needs to have the following syntax "/path/username".

- `tuple<vector<uint8_t>, entity::Error>encodePeerKey(EVP_PKEY *keyToEncode);`

  function to encode an EVP_PKEY into a vector of uint8_t to be sent via sockets

- `tuple<EVP_PKEY*, entity::Error> decodePeerKey(vector<uint8_t> encodedKey);`

  function to decode an EVP_PKEY from a vector of uint8_t

- `tuple<vector<uint8_t>, entity::Error> genDHparam(EVP_PKEY *&params);`

  populates *params* with Diffie - Hellman parameters, returns the encoded version of the parameters.

- `entity::Error genDH(EVP_PKEY *&dhkey, EVP_PKEY *params);`

  uses *params* as Diffie - Hellman parameters from which *dhkey* is populated with a public DH key

- `tuple<vector<uint8_t>, entity::Error>encodeDH(DH *dh);`

  function to encode a DH object into a vector of uint8_t to be sent via sockets

- `tuple<EVP_PKEY*, entity::Error> retrieveDHparam(vector<uint8_t> DHserialized)`

  returns the DH parameters retrieved from *DHserialized*

- `tuple<vector<uint8_t>, entity::Error> derivateDH(EVP_PKEY *your_dhkey, EVP_PKEY *pe`

function to derive the secret from the DH keys

- `tuple<vector<uint8_t>, entity::Error> keyFromSecret(vector<uint8_t> secret);`

  function to derive the session key from a secret generated with DHKE

- ```
  bool isCurrency(const string& str);
  bool isUsername(const string& str);
  bool isPassword(const string& str);
  ```

  functions to sanitize accordingly to the nature of *str*:

  - *isCurrency*: returns true only if *str* contains numeric characters and "." (for decimal point) and if its length is > 0
  - *isUsername*: returns true only if *str* contains alphabetic characters and its length is between 0 (excluded) and *MAX_SANITIZATION_LEN* (included)
  - *isPassword*: returns true only if *str* contains alphanumeric characters and some symbols and if its length is between 0 (excluded) and *MAX_SANITIZATION_LEN* (included)
  ```