UNIVERSITÀ DI PISA

Computer engineering

Computer Architecture

FoC project documentation

Francesco Berti

Alessandro Versari

ANNO ACCADEMICO 2022/2023

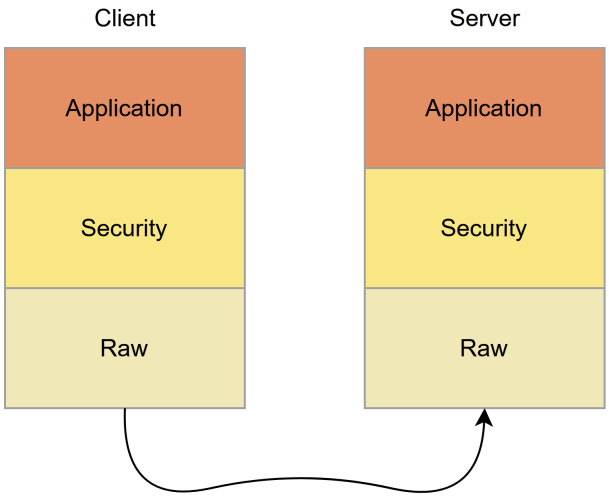
Contents

1 Design choices

- Confidentiality: is guaranteed by using RSA with a modulus of 4096 bits at the beginning of the handshake and AES 256 CBC for the rest of the communication. In particular, for every message encrypted with AES a new IV is created and appended at the beginning of the message.
- Integrity: in the last step of the handshake protocol the client sends to the server the encrypted MAC of all the messages exchanged during the handshake. 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: the first message of the handshake contains a timestamp which has to be inside an acceptance window, if any of the handshake's messages are being replicated the final MAC will be different and the handshake will fail. The server also checks if a user is logged in before processing a request.
- Non-malleability: both RSA and AES use padding, for every session message a new IV and the relative MAC are generated.

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:

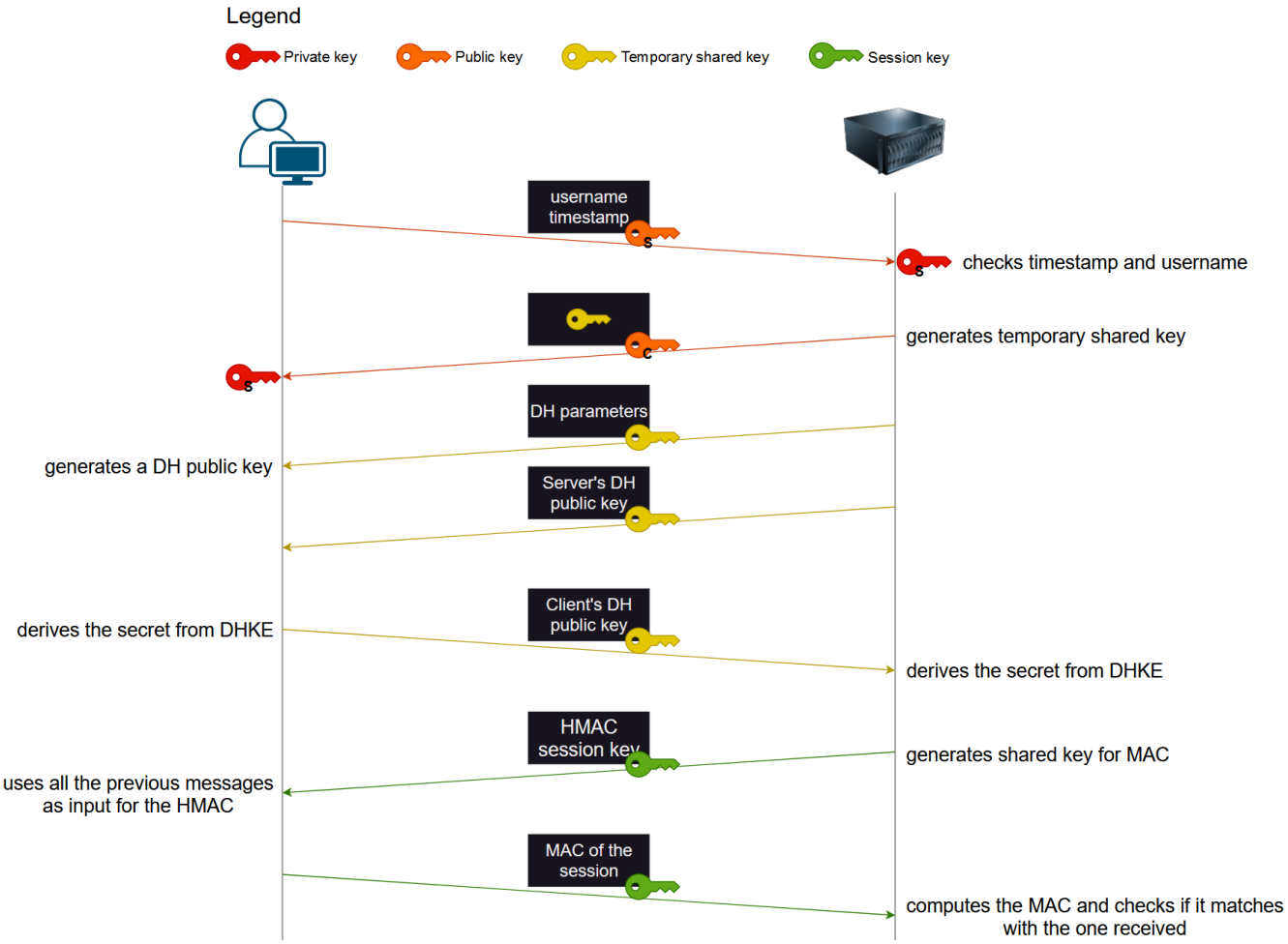
```
{
    "status" : 200|400,
}
```

- 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 and a timestamp is encrypted through an AsymCrypt instance and sent to the server. The server decrypt the message with his private key, checks if the timestamp is inside the acceptance window and eventually loads the public key of the user.

The server generates a random symmetric key to use temporarily for a SymCrypt instance. This temporary AES is used to overcome the max length of the message that can be encrypted using RSA. The random key is encrypted by means of the user RSA public key and sent to him/her.

The client decrypts using its private key and generate a temporary instance of SymCrypt using the key just received.

The server now generates some DH parameters (g, p) and a DH public key (Ys) using the Security module's APIs, serialize theme, encrypt them by means of temporary AES and send them to the client.

The client uses the DH parameters to generate a DH public key (Yc), which will be sent to the client by encrypting it using the temporary AES, and uses the dedicated API to derive the secret and the session key.

The server decrypts and also derives secret and session key then generates a random shared key to be used with a MAC instance, this key is encrypted using the newly created SymCrypt and sent to the client.

The client uses the received key to create an instance of MAC, then uses all the message exchanged during the handshake as an input for the MAC and sends it to the client by encrypting it using the session SymCrypt.

The server builds the MAC in the same way and checks if it matches with what it received.

2.2.2 Symmetric Encryption

The session key created in the handshake protocol is then used to encrypt messages exchanged in a session between client and server.

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, a sanitization method 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 cipher instace:

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

  both the encryption and decryption 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

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)