Universidade de Aveiro

Segurança

Blockchain-based auction management Final Report

Inês Lopes, Nº Mec:76769
João Pedro Fonseca, Nº Mec:73779

January 3 2019

Contents

1 Introduction

The main objective of this project is to implement a secure auction management system, using the principles of a Blockchain. This system, enables users to create and participate in auctions.

1.1 Security Rules of the system

The blockchain-based auction management system we purpose, follows 4 rules:

- Bid confidentiality, integrity and authentication
- Bid acceptance control and confirmation
- Bid author identity and anonymity
- Honesty assurance

1.2 Security Mechanisms implemented

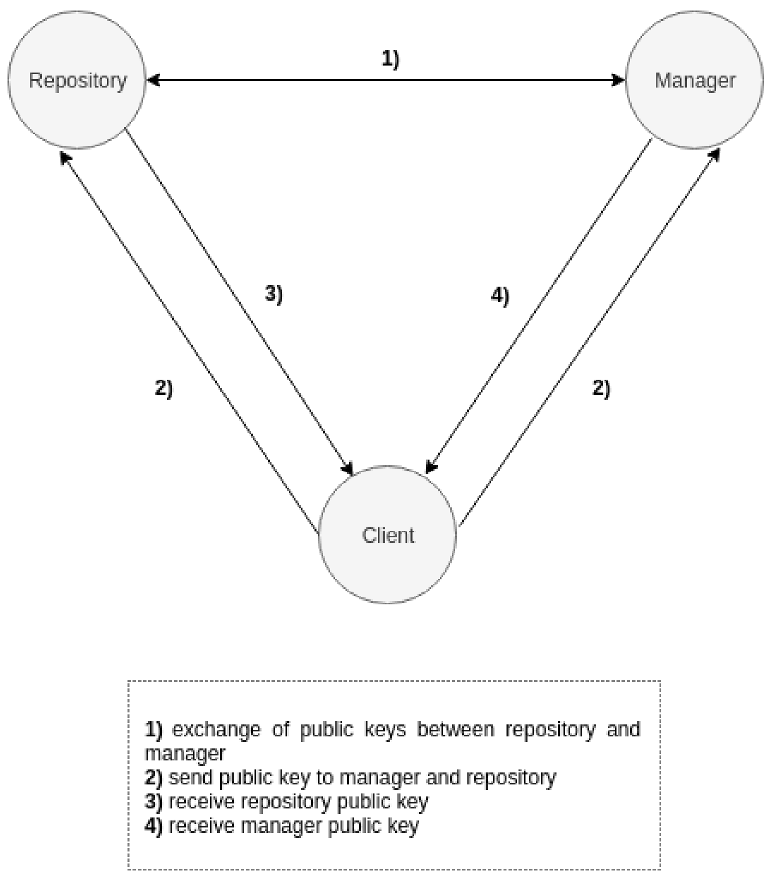
The rules stated before are complemented with security mechanisms that will allow:

- Confidentiality of sensitive information;
- Privacy of the clients;
- Protection of information transmitted in insecure channels;
- Verification of the authenticity of the messages exchanged between the entities of the system.

This report presents and explains all the security measures and features implemented on our system, to make the auction management system safe and confidential for all participants.

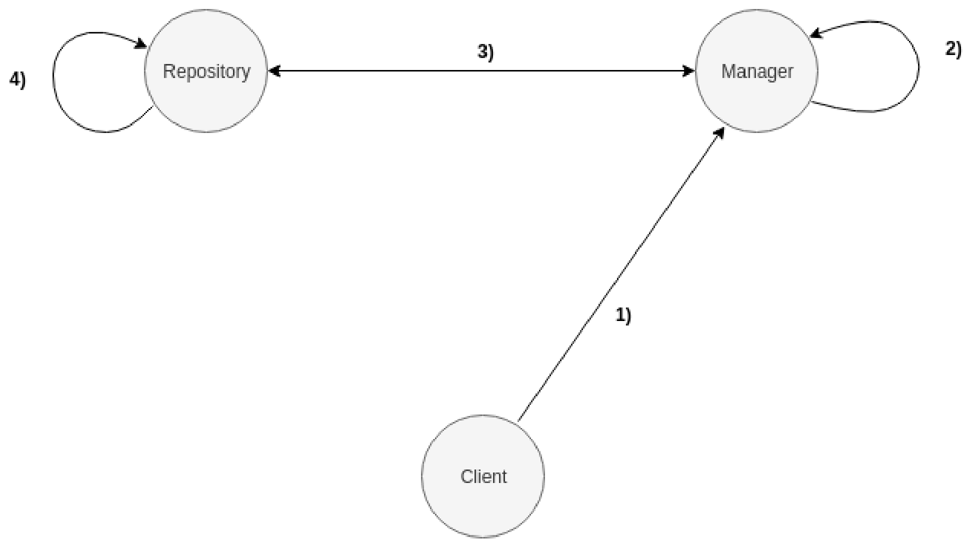
2 Architecture of the project

Below are 4 diagrams which demonstrate, sequentially, the normal flow of the execution of our system.



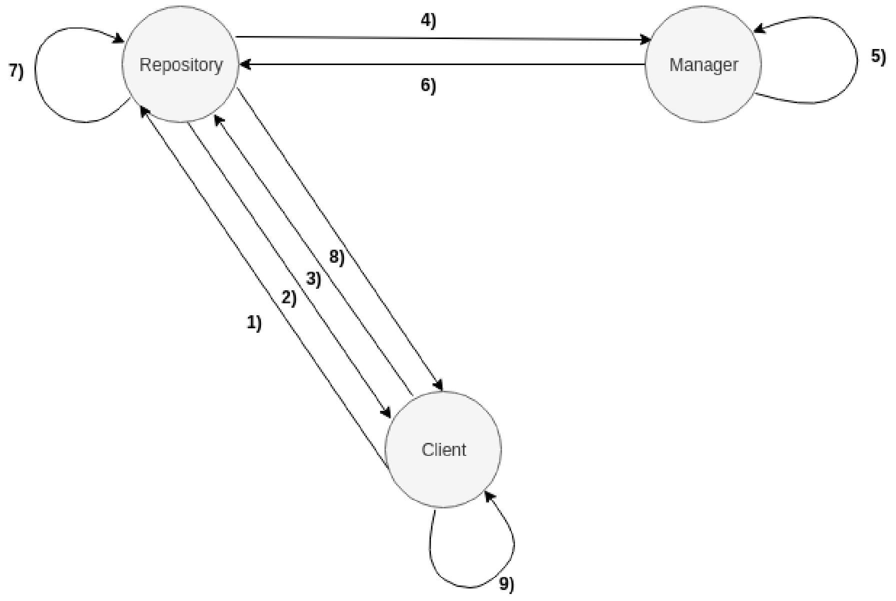
1) exchange of public keys between repository and manager
2) send public key to manager and repository
3) receive repository public key
4) receive manager public key

Figure 1: phase 1 - exchange of public keys



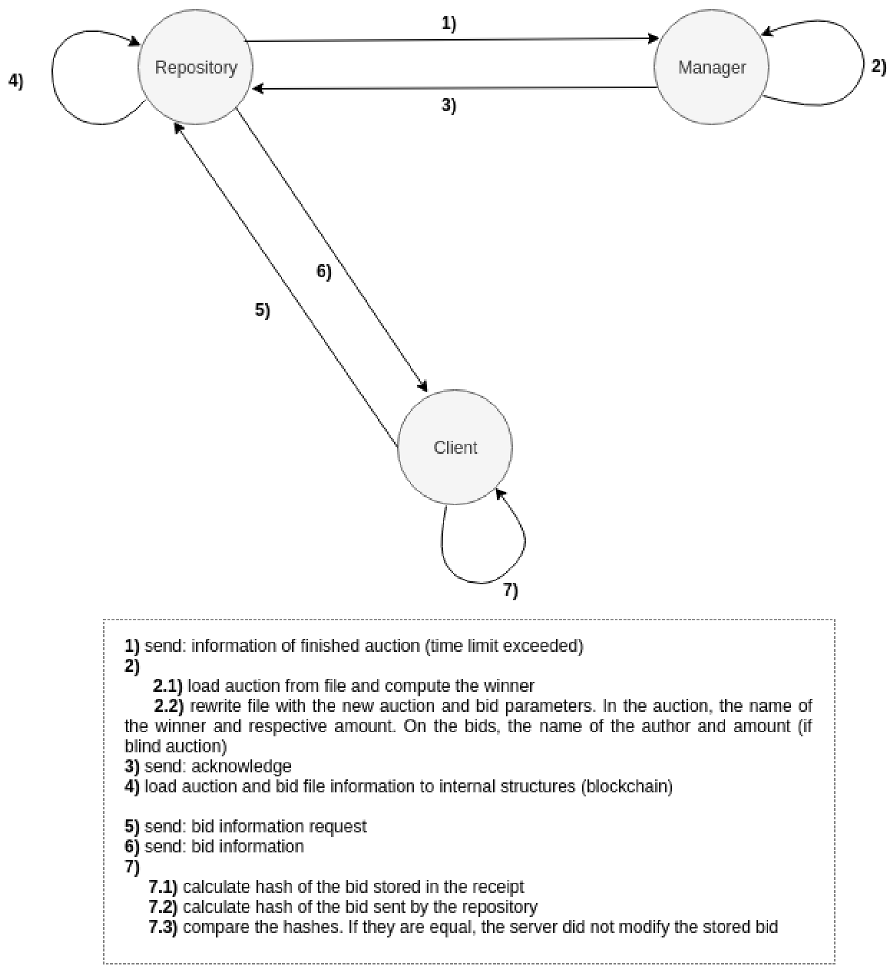
1) send: encrypted symmetric key + encrypted certificate + auction
2)
 2.1) decrypt symmetric key
 2.2) decrypt certificate
 2.3) verify certificate's chain
3) send auction + validity of the certificate
4) if certificate is valid, store auction

Figure 2: phase 2 - creation of an auction



1) send: bid request
2) send: proof-of-work parameters
3) send: encrypted symmetric key + encrypted certificate + bid + signed receipt
4) send: encrypted symmetric key + encrypted certificate + bid
5)
 5.1) decrypt symmetric key
 5.2) decrypt certificate
 5.3) verifidy certificate's chain and run dynamic code for bid validation
 5.4) sign receipt, previously signed by client
6) send: bid validity and receipt
7) if bid is valid, store it and sign receipt
8) send: acknowledge + signed receipt
9) validate receipt signatures and store it

Figure 3: phase 3 - placing a bid



1) send: information of finished auction (time limit exceeded)
2)
  2.1) load auction from file and compute the winner
  2.2) rewrite file with the new auction and bid parameters. In the auction, the name of the winner and respective amount. On the bids, the name of the author and amount (if blind auction)
3) send: acknowledge
4) load auction and bid file information to internal structures (blockchain)

5) send: bid information request
6) send: bid information
7)
  7.1) calculate hash of the bid stored in the receipt
  7.2) calculate hash of the bid sent by the repository
  7.3) compare the hashes. If they are equal, the server did not modify the stored bid

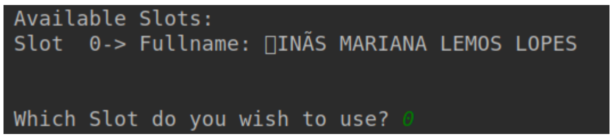
Figure 4: phase 4 - calculating the winner (manager) and checking the receipt (client)

3 Implementation

3.1 Functionalities of the system

The user's interface for the system consists of a menu, in which the user may choose an option associated with the action s/he wants to perform.

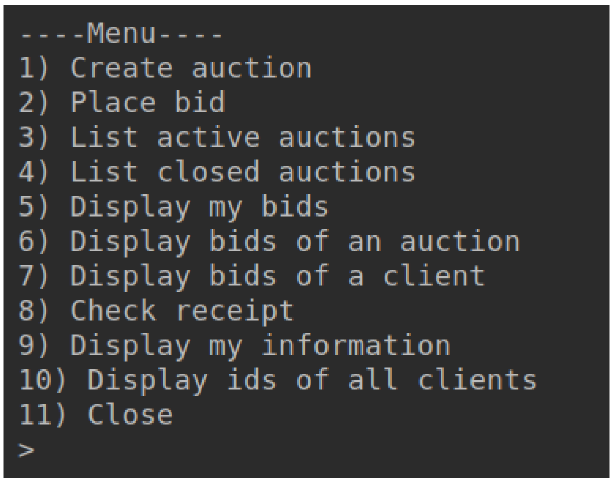
Before the display of the menu (see figure 6), the user is asked which slot s/he wants to use. The number of slots (see figure 5) is equal to the number of citizen cards inserted on card readers (connected to the computer).



```
Available Slots:
Slot  0-> Fullname: INÃS MARIANA LEMOS LOPES


Which Slot do you wish to use? 0
```

Figure 5: Initial setup of the application



```
----Menu----
1) Create auction
2) Place bid
3) List active auctions
4) List closed auctions
5) Display my bids
6) Display bids of an auction
7) Display bids of a client
8) Check receipt
9) Display my information
10) Display ids of all clients
11) Close
>
```

Figure 6: Menu of the client's application

The functionalities are the following:

1. Create auction: ask client for auction parameters (name, time limit, description, type and dynamic code file). The auction is then sent to the manager server, which will validate the client's authentication certificate and afterwards send the auction to the repository, which will store it.

2. Place bid: ask client for bid parameters (serial number of the auction and amount). The request is sent to the repository which answers with a proof-of-work (see Cryptopuzzle section). The client computes the proof-of-work and sends the answer to the repository. If it is a valid answer, the client will send the bid parameters to the repository. Then, the certificate will be validated by the manager and if valid, the repository will store a new block (bid) in the blockchain (auction).

3. List active auctions: lists the currently active auctions of the system.

4. List close auctions: lists the close auctions of the system.

5. Display bids of an auction: ask client for a serial number of an auction and displays all its bids.

6. Display bids of a client: ask client for an ID of a client and display all his/her bids.

7. Check receipt: ask for a serial number of an auction and an hash of a bid (the hash is the identification of the bid). The request of information is sent to the repository, which sends it back to the client. The hash of the bid information on the client's receipt is compared against the calculated hash of the information of the bid retrieved from the repository and if they are equal, the information stored on the server was not modified.
8. Display my information: displays the information of the current user (name and ID).
9. Display ids of all clients: displays the IDs of the clients currently active on the system.
10. Close: the client is shutdown and a message is sent to the servers for them to shutdown too.

3.2 Identification of the clients

The identity of the clients (located inside the authentication certificate) must be hidden until the end of the auction. The certificates are only decrypted for verifying the chain and when there is the need to extract the names of the bidders or decrypt the amount and write them back into their respective bids or auction. In any other case, the certificates are kept encrypted on the .txt files of the blockchain, so that no other entity (besides the manager) might deduct the identity of a new bid from closed auctions.

We also created another identification of the client (ID) in the system, as a way to identify the author of a bid without compromising their true identity. This ID is equivalent to the numbers given to bidders on real auctions and it is a MD5 digest of the number of the Citizen Card of the client.

3.3 Messages format

All messages are exchanged in JSON format. The encrypted fields of the messages, certificates and symmetric keys are transmitted in Base64 format.

All messages exchanged have the following format and are sent through sockets using UDP:

```
{
    'payload' : payload ,
    'signature': signature
}
```

The payload field contains the actual information and the signature is the signature of the payload field, done by the entity transmitting the message. All messages exchanged on the system are signed and therefore, have the signature field.

The auction and bid parameters are transmitted on the payload field (and stored) also with a dictionary format:

auction:

```
{
    "key": key, "cert": cert, "serial": serial, "id": id,
    "timestamp": timestamp, "name": name, "time-limit": time-limit,
    "description": description, "type": type, "state": state, "winner": winner,
    "winner_amount": winner_amount
}
```

bid:

```
{
    "key": key, "cert": cert, "serial": serial, "hash": hash,
    "hash_prev": hash_prev, "amount": amount, "name": name,
    "id": id, "timestamp": timestamp
}
```

Besides the auction and bid payloads, the user may also send 'command' messages to the repository, to execute the functionalities described on the menu:

```
{
    "payload": {
                    "command": command, (additional fields)
                },
    'signature': signature
}
```

The command field may be one of the following:

- bid_request: request the creation of a bid, which the repository answers with proof-of-work parameters.
- list_open: send list of active auctions.
- list_closed: send list of closed auctions.
- bid_auction: send list of all bids of an auction.
- bid_client: send list of all bids of a client.
- check_receipt: send bid information to the clients for them to check against their own receipt.
- list_ids: send list of IDs of active clients of the system.

3.4 Types of auctions

The types of auction implemented were the two mandatory:

English auction: requested on the system as type 'e', the english auction implies that the next bid must overcome the amount of the previous bid. Also, identities must be encrypted, so the certificates associated with each bid are encrypted with a symmetric key (which is encrypted with the manager's public key). All other values are stored in cleartext.

Blind auction: requested on the system as type 'b', the blind auction (in our solution) also enforces the encryption of the identity (certificate), but the amount is also encrypted. The amount is encrypted with the same symmetric key that was used to encrypt the certificate of the client.

3.5 Blockchain

Each auction is stored in the repository server has an ordered linked list, like a blockchain, where the nodes/blocks are the sequential inserted bids (see file *blockchain.py*). Each bid is identified by an hash, which is the MD5 digest of the contents of the bid: encrypted symmetric key, encrypted certificate, serial number of auction, hash of the previous block, amount, id, timestamp. The name of the author is not used on the hash because, in the receipt, the name is not present (the repository does not know the author of the bid at the time of the receipt signing).

A Blockchain has the parameters stated on section 3.3 and also a reference to the head block and the tail block, being the head the first bid added to the blockchain and the tail the most recent one.

3.6 Receipts

When the repository stores a bid, it must send a receipt to the client who made it. The receipt is a file with the bid parameters and the sequential signatures of the 2 servers and the client. The scheme of the receipt is seen below:

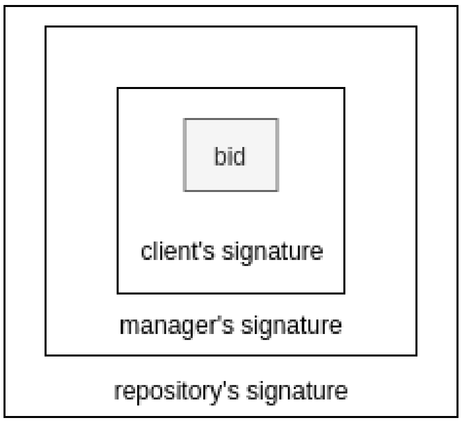


Figure 7: Scheme of the receipt

3.6.1 Creation

The flow of the receipt creation is as follows:

1. The client sends the bid parameters to the repository and signs the bid, proving s/he was the author of the bid.

   Note: sig_c is the client's signature.

```
{
   'payload': {
                 'bid': parameters,
                 'sig_c': sig_c
                 },
   'signature': signature
}
```

2. The repository will forward the bid to the manager, for validation. If it is valid, the manager will sign the receipt (signing the data['payload'] of the previous JSON).

   Note: sig_m is the manager's signature.

```
{
   "payload": {
                "valid": True,
                "receipt": {
                            "bid": parameters,
                            "sig_c": sig_c,
                            "sig_m": sig_m
                            },
   "signature": signature
}
```

3. The bid validation result is sent to the repository, along with the receipt, which will now be signed by the repository. The repository stores the bid and sends an acknowledge and the receipt to the client.

   Note: sig_r is the repository's signature.

```
{
   "payload": {
                "ack": "ok",
                "receipt": {
                            "bid": parameters,
                            "sig_c": sig_c,
                            "sig_m": sig_m,
                            "sig_r": sig_r
                            }
                }
   "signature": signature
}
```

3.6.2 Validation

Before storing the receipt in a .txt file, the client will have to validate the signatures in it. The process of validation is the following:

1. Validate the repository's signature, with its public key, the "sig_r" value and the data:

```
{
   "bid": parameters,
   "sig_c": sig_c,
   "sig_m": sig_m
}
```

2. Validate the manager's signature, with its public key, the "sig_m" value and the data:

```
{
   "bid": parameters,
   "sig_c": sig_c
}
```

3. Validate client's own signature, with one's public key, the "sig_c" value and the data:

```
{
   "bid": parameters
}
```

3.7 Cryptopuzzle

To control the rate at which the bids are sent to the repository, it will answer each bid request with a cryptopuzzle, which must be solved by the client and the result accepted by the repository for the bid to be accepted for further validation. The mechanism cryptopuzzle or proof-of-work is a mechanism of relatively high complexity (hard to perform), whose result is easy to be validated (in this case, by the repository).

On bitcoin, there is also the need for a cryptopuzzle. The average time to "mine" a block, in other words, the time that takes to compute a mathematical problem (cryptopuzzle) is in average 10 minutes [1].

Of all the existing proof-of-work algorithms, we decided to use the hash cash algorithm. In this algorithm, a digest of a string is computed in a while loop, changing in each iteration with the use of a concatenated counter value, until the string starts with a specified number of zeros. When this happens, the string used and the digest are sent to the entity that first sent the proof-of-work for further validation.

In hash cash, used for limiting e-mail spam and denial-of-service attacks, the digest is computed using a string with the following contents: version, timestamp, email address, random seed [2]. A counter may also be added.

Example: 1:190203:ines@ua.pt:casjhcb45362ddf88

In our version of hash cash, the repository server sends a random string (unique for each bid request) and a number of zeros (5 in our project - arbitrary number). It is important that the string is not reused on later bid requests, because if it was found by an attacker, it would be easier to calculate the proof-of-work. The client will then concatenate the received string with a counter, incremented in each loop iteration, and compute a 256 bit digest (SHA256) of the string. When the digest starts with "00000", the result was found and it is sent with the string (random string + counter) to the repository server.

When the repository receives the answer, it will calculate the digest of the string and check if it is equal to the client's digest. If they are equal, the client is allowed to send the bid parameters to the server.

3.8 Dynamic Code

One of the key aspects of the project, was the availability of a Client that made an auction to validate the bids performed by other Clients. In Python, this can be done by using the Built_In functions[14] compile(), eval() and exec(). All of them enable compilation of foreign code with restrains set by a developer. The existence of these functions made easy for us to verify the bids dynamically. On the manager we have exposed a set of variables that can be used /changed on the exec() sandbox. They are:

- 'id_client' - variable that holds the current bidder id
- 'num_bids'- variable that holds the number of bids a client can make
- 'valid' - variable that will hold the result of the execution of the dynamic code

Whenever a client sends a piece of code for validating an auction, the manager will save that code. Afterwards, when another client makes a bid, the saved code will be compiled using the compile() function. If the compilation goes well the bytecode generated will be executed on the exec() function. After the execution of the exec() function the valid variable will have one of these values : None, True or False.

[1] `https://data.bitcoinity.org/bitcoin/block_time/5y?f=m10&t=l`
[2] `http://www.hashcash.org/faq/`

4 Security Mechanisms-Overview

After introducing how our Blockchain-based auction management application works, we will now explain how we made our application more reliable, abiding to the rules, in subsection 1.1, and security mechanisms, in subsection 1.2, needed to make the application "secure".

One of the essential requirements of the system was the possibility of the authentication of Portuguese Citizens using this service, with their Portuguese Citizen Card. Thus, we have implemented a Python Wrapper called: "cc_interface.py", which was based on a existent Wrapper[1], built by a student which had taken the Security course, before us.
We will explain key functions of the Wrapper we built and how we used it in our service.

Knowing beforehand we need to make sure the client app needs to be connected to 2 servers (Repository and Manager), and make the communication between them trusty, we have developed another Wrapper called: "security.py", which holds the classes (GenerateCertificates, CertificateOperations, CryptoUtils) responsible for the operations using certificates and public/private keys.

4.1 Citizen Card Wrapper

Our first step was to read the manual for development using the Citizen Card(CC)[15]. After doing so, we came to a simple conclusion: it needs to be updated. It doesn't mention the programming languages which have an wrapper of the offline middle-ware. Right now the official support for application development using the Portuguese Citizen Card(PTC) in Python is none. We have checked the `https://svn.gov.pt/projects/ccidadao/`, in the hope of finding a old Wrapper but we weren't lucky.Thus we searched for the alternatives to get the information out of the CC smartcard.

The Portuguese Citizen Card, has a PKCS#11 interface, which offers support for non-Microsoft applications. The existence of this interface enabled us, to use the PyKCS11 Wrapper [11], which enables us to obtain information from the CC. Most of the functions o present on the PyKCS11 Wrapper make easy the development of a Wrapper that can get information from the Certificates present on the CC and make use of the cryptographic tokens present on this smartcard. As we have stated before, due to the nonexistence of a official Wrapper for interacting with the Citizen Card, we needed to make our own. We made our Wrapper,"citizen_interface.py", using the previous work of a colleague, which was only supported in Python version 2.

To use the CC Wrapper one needs to be using a Linux environment with the applications for making use of the smartcard. In particular, it is needed the library "libpteidpkcs11.so", which is installed in the environment.
The first thing our module does upon being called externally is to verify the existence of this library and then initialize the PyKCS11 environment. The PyKCS11 environment will gives the program access to all the sessions available. Each session will correspond to a Portuguese Citizen Card. Afterwards one is able to use the PyKCS11 functions.

Having the PyKCS11 module loaded, the next thing our Wrapper does is to load all the certificates and CRLs that are needed from the folders : "certLists/", "crlLists/". All the certificates associated with the citizen card authentication and respective certificate revocation lists need to be downloaded to this folders and can be obtained at the Portuguese Citizen Card Public Key Infrastructure(PKI) [13].

The certification chain of the Registry Authorities of the Portuguese State(ER) is shown in figure 8. As it can be seen the ER is certified by the Portuguese Certification Authorities, this means we need their level 2 certificate [7]. After that, we need to verify the level 2 certificate, this is done by comparing it to the "ECRaizdeEstado.crt", which is a level 1 certificate in figure 8.

Even the Portuguese Certification Authorities need to be certified. This fact made us use 2 new certificates issued by the only Portuguese Certification Authority, Multicert:

- Multicert Certification Authority Certificate [9]

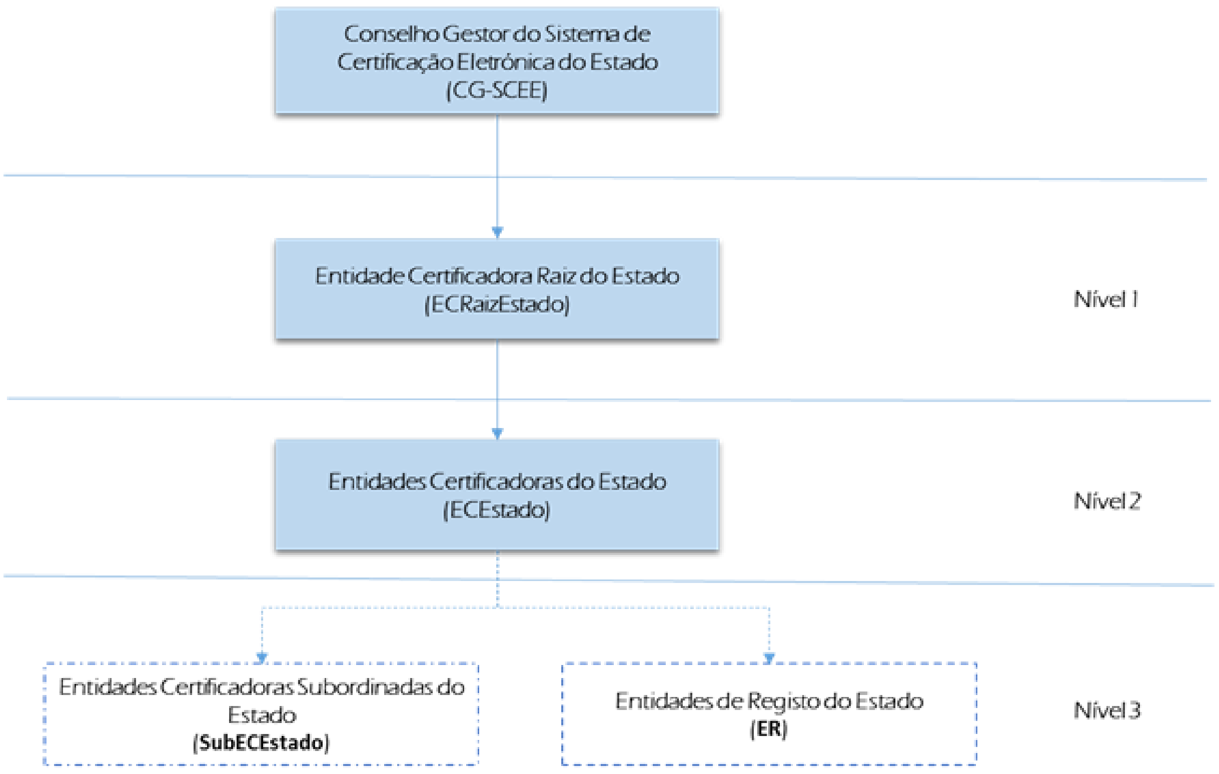

Conselho Gestor do Sistema de Certificação Eletrónica do Estado (CG-SCEE)
Entidade Certificadora Raiz do Estado (ECRaizEstado)
Nível 1
Entidades Certificadoras do Estado (ECEstado)
Nível 2
Entidades Certificadoras Subordinadas do Estado (SubECEstado)
Entidades de Registo do Estado (ER)
Nível 3

Figure 8: Funcional Architecture of the SCEE [6]

- Multicert Root Certification Authority Certificate [10]

Both of these certificates need to be added to the "certLists/" folder.

Having loaded all the information, we can use all the certificates, certificate revocation lists and the PyKCS11 module.

4.1.1 _ccStoreContext(rootCerts, trustedCerts, crlList)

This method creates a X509StoreContext Description that can be used to validate a given Portuguese Citizen Card. To create the X509StoreContext Description, one needs to import this module from the "OpenSSL.crypto" package available in the PyOpenSSL lib [3]. The X509Store is physical store where certificates are persisted and manage, this may include a set of certificates to trust, a set of certificate revocation lists, verification flags and more.

4.1.2 PTEID_GetID()

This method gets the name of the owner (Subject) of the CC by fetching it from the "CKA_SUBJECT" field on the present session of the PTC.

4.1.3 PTEID_GetBI()

This method gets the number of the BI of the owner (Subject) of the CC by fetching it from the "CKA_SUBJECT" field on the present session of the PTC.

4.1.4 certGetSerial()

This method is used to return the CC authentication certificate serial number.

4.1.5 PTEID_GetCertificate(slot)

This method will return a CITIZEN AUTHENTICATION CERTIFICATE from a connected CC smartcard on a given slot.

4.1.6 verifyChainOfTrust(certificate)

This method verifies if the given certificate(cert) is valid under the Authority of the Portuguese Citizen Card Authority and under the root of state Authority. This verification can be done by verifying the X509 Object Certificate under the context we have created before by running the verify_certificate() function implemented on the Object X509Context created before.

4.1.7 sign_data(slot,data)

This method signs a string using the Private Key of the Portuguese Citizen Card. The data will be signed using the RSA signature with the SHA-256 hash function(CKM_SHA256_RSA_PKCS).

4.1.8 verifySignature(certificate, data, signature)

This method is used to verify the signature of a document/string signed using a certificate's public key, which is extracted from the provided certificate. The original data that was signed must be also provided. The certificate should pass the test of trust beforehand by verifying the Chain of Trust of the Portuguese Citizen Card. The verify function will be applied over a rsa.RSAPublicKey object which is provided on the cryptography lib [4].

4.1.9 GetNameFromCERT(certificate)

This method will fetch the name of the subject of the CITIZEN AUTHENTICATION CERTIFICATE of the certificate provided

4.2 Generate Certificates

This class is responsible to create all x.509 certificates (not implemented) and the Private/Public Keys of the servers (Repository and Manager). This class will make use of the RSA package which is provided on the cryptography library [4]. The generation of RSA private key is done by running the method generate_private_key and specifying the key size. If a server has already an associated private key, there will not be generated a new private key. The existence of a private key is tested by verifying over the folder "serverCerts" if the server has already a privateKey. The Private keys need to be stored in a ".pem" file which can be encrypted or not. The next step done when this class is called is the generation if a public key(_RSAPublicKey Object) from a _RSAPrivateKey Object, by running the method public_key() over this Object. Having the public key created, the generation of a Certificate would be the next step, but was not implemented due to the lack of time. Still, it would be simple to do, after having a root authority for the servers which would sign the public keys of both servers.

4.2.1 writePrivateKeyToFile(name, password)

This method will write the private key stored in a variable into the file with PEM encoding and using a PKCS8 padding. This method takes the name of the server and a password and writes the Private key to a file. The password parameter can be a string or nothing. This will generate a encrypted private key or a private key.

4.2.2 signData(data)

This method signs a string of data using the Private Key of the given Server. The data will be signed using the RSA signature with the SHA-256 hash function(CKM_SHA256_RSA_PKCS_PSS).

4.3 Certificate Operations

The class Certifacate Operations also makes use of the cryptography library [4] and loads all the certificates present on the folder "userCerts/" to memory and will return them in a dictionary where the keys are the user id and the values is the certificate of the user.

4.3.1 loadFromFile(id, certPath=None)

This method will load a certificate with a given id, present on a given path(certPath) into bytes.

4.3.2 writeToFile(cert, id)

This method will load the bytes of a certificate into a file with the id on the filename.

4.3.3 getCertfromPem(cert):

This method will turn the bytes of a certicate into a X509 Object using the load_pem_x509_certificate() inherited from the X509 Object's of the cryptography library [4].

4.3.4 rsaPubkToPem(pubk)

This method will convert a rsa.RSAPublicKey Object into bytes using the serialization module of the cryptography library[4]·

4.4 Crypto Utils

The Crypto Utils class has a collection of methods to perform :

- verifications over the signatures of the data that was signed
- encryption of the data using a RSA Public Key
- decryption of the data using a RSA Private Key

4.4.1 verifySignatureCC(pubk, data, signature)

This method will receive a _RSAPublicKey object obtained from the Citizen Card Certificate and test if the signature provided (bytes) corresponds to the owner of that public key

4.4.2 verifySignatureServers(pubk, data, signature)

This method will receive a _RSAPublicKey object obtained from the private keys of a server and test if the signature provided (bytes) corresponds to the owner of that public key

4.4.3 RSAEncryptData(pubK, data)

This method will receive a string and encrypt using the method inherited from the _RSAPublicKey object

4.4.4 RSADecryptData(privK, encryptedData)

This method will receive a bytes and decrypt them with the _RSAPrivateKey provided.

5 Security Mechanisms - Message Exchange

After explaining the modules we have created to make the messages exchange secure, we will now present the behaviour of the Servers and Clients when they perform a message exchange. As we have seen on figure 1, the public key exchange starts when a entity (server or client) sends its public key to another in a message which doesn't have any security. Given that only the Client has a certificate (provided by a User), on the first transaction, we are trusting that both servers are trustful. On the next messages exchanges, all the entities will need to verify each other, considering the public keys that were received on the first connection. If the servers had a certificate signed by an authority which certified both of them, all the entities were expected to verify each others certificate's chain of trust. As the client is the only entity that has a certificate, that needs to be verified, in auction and bid request messages it sends to the servers, there will be a verification of the certificate's chain of trust. If the certificate is not valid the server will send that information to the client. Unlike phase 1, the phases 2, 3 and 4, presented in section 2, have a layer of security, which makes all the messages non-repudiable.

5.1 Non-repudiaton of Messages

We need to make sure the Client/Server which sent a message cannot/will not repudiate the message sent. Thus, it is essential that all the messages are signed by the sender of the message. The message signature, will be made on the plain text of the payload. The client messages will be signed using the CKM_SHA256_RSA_PKCS algorithm, while the servers will sign their messages with a CKM_SHA256_RSA_PKCS_PSS algorithm.

5.2 Message confidentiality

Our system makes sure that the sensitive information on messages is hidden, by performing a hybrid public key encryption of the fields of payload of the messages. As encryption of a message will encrypt the sensitive data using a symmetric encryption method called Fernet[5] which is available in the cryptography library[4], that will be fed a symmetric key that can also be generated using this method Object. After encrypting the payload of the message, we will encrypt the key used in the symmetric encryption with a RSA public key encryption that makes use of the SHA256 hashing function and the MGF1 padding. The result of the 2 steps of the hybrid pubic key encryption are a symmetric encrypted payload(equation 1) and an assymetric encrypted Fernet Key(equation 2).

$$encrypted\_payload = Fernet(Payload, FernetKey) \tag{1}$$

$$encrypted\_FernetKey = RSA\_256\_MGF\_Encrypt(FernetKey) \tag{2}$$

During phase 2, figure2, whenever a Client creates a new Auction, it will use the same Fernet Key for all auctions he creates.

Unlike phase 2, whenever a Client places a bid, phase 3, figure 3, a new Fernet Key will be used. This makes the confusion higher but won't increase security since the public key that will be used on the assymetric encryption is always the same.

6 Deploying the project

The project must be executed using python 3.

It is advised to run the project inside a python virtual environment, which can be installed and activated as follows. The commands should be executed inside the root folder of the project.

```
$ python3 -m pip install --user virtualenv
$ python3 -m  virtualenv venv
$ source venv/bin/activate
```

The virtual environment is currently activated. Next, run the following commands for installation of required packages, in Ubuntu:

```
$ sudo apt-get install swig
$ pip3 install -r requirements.txt
```

Finally, the project must be run in the following sequence (repo.py, then man.py, then client.py):

```
$ cd src/
$ python3 repo.py && python3 man.py && python3 client.py
```

Note: The explanation of the contents of each file and folder of the project is described in the README.md file of the root folder.

7 Issues of the system

We have currently 2 known issues in our system.

First problem:

The repository server is blocked in a recvfrom() function (socket module) most of the time. This function is located on its loop and is responsible for listening for client requests, both from normal clients and the manager server.

We did not use threads on our project and so we calculate the timelapse of an auction in the beginning of the loop, where it is verified if the delta (current time - timestamp of the auction) is higher or lower than the time-limit of the auction. This verification occurs before the blocking function.

This means that, even if the time-limit as been exceeded on an auction, that verification will only be done if the server receives a "trigger" message from the client, so to unblock the recvfrom() function and initiate a new iteration on the loop. The trigger may be a simple command, like "list active auctions".

Second problem:

The verification of the chain of the authentication certificates can only be done in one of the tested computers. In the other, the outcome of the verification is always "non valid certificate".

We then decided to run the code in a python virtual environment, with the installed packages on the same version on both computers. We also used the same folder with the certificates and CRLs needed. However, this did not solve the problem and we could not find the root of it, leaving the system with that same fault.

8 Conclusions

Apart from the problems encountered on our project, it was implemented with almost all functionalities proposed on the project description.

The receive functions of the socket module had to be handled carefully to prevent blocking on the system and send functions had to be executed only when a receive function was being executed in another entity, so not to loose information.

The hash cash is a good algorithm for preventing denial of service attacks and we were able to create our own (more simplistic) version and still get a reasonable computation time (above 2 seconds and below 6). It is a computation time far low than the bitcoin.

In this project we explored the concept of object security transmitted in insecure communication channels. Further research about the theme lead us to understand how much that concept is important on the information security area.

Having encrypted objects instead of a secure channel means that only the destination containing the key is able to decrypt the information. If the channel is compromised, the information will still be safe.

Also, this concept is particularly interesting in the Internet of Things area. In IoT, there are use cases where more than one client might try to access the same resources, each with different access rights. Also, two endpoints might not be able to setup a secure channel. In this situation, secure channels might not be enough and object security would be a key mechanism. Refer to [1] for further reading on the theme.

[1] `https://www.w3.org/2014/02/wot/papers/mattsson.pdf`

Bibliography

[1] G. Cardoso. Pkcs11_wrapper, 2019. URL https://github.com/luminoso/chatsecure/blob/master/M2/PKCS11_Wrapper.py. [Online; accessed 2-January-2019].

[2] W. Commons. Transport layer security, 2019. URL https://en.wikipedia.org/wiki/Transport_Layer_Security#Forward_secrecy. [Online; accessed 2-January-2019].

[3] I. Contributors. Welcome to pyopenssl's documentation!, 2009. URL https://pyopenssl.org/en/stable/. [Online; accessed 2-January-2019].

[4] I. Contributors. Welcome to pyca/cryptography, 2019. URL https://cryptography.io/en/latest/. [Online; accessed 2-January-2019].

[5] I. Contributors. Fernet (symmetric encryption), 2019. URL https://cryptography.io/en/latest/fernet/. [Online; accessed 2-January-2019].

[6] S. de Certificação Electrónica do Estado. Arquitectura funcional do scee, 2019. URL https://www.scee.gov.pt/media/6832/ecee-arquitetura-funcional.png. [Online; accessed 2-January-2019].

[7] S. de Certificação Electrónica do Estado. Sistema de certificação electrónica do estado, 2019. URL https://www.scee.gov.pt/rep/certificados/. [Online; accessed 2-January-2019].

[8] E. C. C. do Estado. Entidade certificadora comum do estado, 2019. URL https://www.ecce.gov.pt/certificados/. [Online; accessed 2-January-2019].

[9] M. S. D. C. ELECTRÓNICA. Certificado da entidade de certificação da multicert, 2019. URL https://pki.multicert.com/cert/MULTICERT_CA/certificado.html. [Online; accessed 2-January-2019].

[10] M. S. D. C. ELECTRÓNICA. Certificado da entidade raíz de certificação da multicert, 2019. URL https://pki.multicert.com/cert/MULTICERT_CA/certificado.html. [Online; accessed 2-January-2019].

[11] L. R. Giuseppe Amato (Midori). Welcome to pykcs11's documentation!, 2019. URL https://pkcs11wrap.sourceforge.io/api/. [Online; accessed 2-January-2019].

[12] Microsoft. Asymmetric keys, 2019. URL https://docs.microsoft.com/pt-pt/windows/desktop/SecCrypto/public-private-key-pairs. [Online; accessed 2-January-2019].

[13] I. D. R. E. D. NOTARIADO. Cartão de cidadão, 2019. URL https://pki.cartaodecidadao.pt/. [Online; accessed 2-January-2019].

[14] Python. Built-in functions, 2019. URL https://docs.python.org/3/library/functions.html. [Online; accessed 2-January-2019].

[15] V. Silva. Manual técnico do middleware cartão de cidadão, 2019. URL https://www.autenticacao.gov.pt/documents/10179/11463/Manual+T\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{e\global\mathchardef\accent@spacefactor\spacefactor}\accent19e\egroup\spacefactor\accent@

spacefactorcnico+do+Middleware+do+Cart~ao+de+Cidad~ao/. [Online; accessed 2-January-2019].

[16] B. Wiki. Proof of work, November 2018 (last edited). URL https://en.bitcoin.it/wiki/Proof_of_work#List_of_algorithms. [Online; accessed 23-January-2019].

[17] A. Zúquete. *Segurança em redes informáticas.* FCA - Editora de Informática, 2013. ISBN 978-972-722-767-9.