Universidade de Aveiro

Segurança Informática nas Organizações

Sistema de Leilões

Relatório Final

Autores :
84793 Daniel Nunes
84921 Rafael Direito

Professores :
João Paulo Barraca
Vitor Cunha

28 de Janeiro de 2019

Conteúdo

Lista de Figuras

Lista de Códigos

1 Introdução

Inserido no plano curricular da disciplina de Segurança Informática e Nas Organizações, do curso de Engenharia Informática, da Universidade de Aveiro e lecionada pelos professores João Paulo Barraca e Vítor Cunha, este relatório é proveniente da execução do projeto final da cadeira.

Este projeto consiste num sistema de leilões onde um cliente pode criar leilões, participar em leilões e no final observar o resultado do leilão criado. Todas as comunicações neste sistema são seguras e todas as licitações podem ser validadas no final do leilão, através da comparação com os recibos de cada cliente.

2 Processos

2.1 Proteção (Encriptação, Autenticação e Integridade) das Mensagens Trocadas

Todas as mensagens são trocadas no formato json utilizando uma chave 'type', que indica o propósito da mensagem, sendo que para o envio das mensagens são utilizadas sockets.

Relativamente à proteção das mensagens podemos distinguir 2 cenários diferentes:

- Mensagens entre os clientes e o repositório.
- Mensagens entre o repositório e o leiloeiro.

2.1.1 Mensagens entre os clientes e o repositório

Relativamente ao primeiro cenário, utilizamos uma cifra híbrida, de forma a partilhar as chaves da nova sessão a estabelecer. Desta forma, iremos trocar uma chave AES-CCM que, para além de encriptar as mensagens, garante a integridade das mesmas.

Entre o cliente e o repositório existe uma mensagem inicial onde vai ser trocada a AES-CCM (ver Chaves Simétricas) para comunicação entre os mesmos.
A partir desta mensagem, as mensagens seguintes passam todas a ser encriptadas pela chave trocada.

Procesos e Interações:

1. O cliente, aquando da ligação ao repositório, gera uma *key* e um *nonce* AES-CCM .
2. O cliente encripta a *key* e o *nonce* com a chave pública do repositório obtida através do certificado do repositório. Desta forma garantimos que a mensagem apenas poderá ser decifrada pelo repositório.
3. A *key* e o *nonce* são assinados através do Cartão de Cidadão garantindo a autenticidade dos mesmos.
4. O certificado do Cartão de Cidadão deverá ser enviado para validação das assinaturas no lado do repositório.

5. O repositório valida o certificado (com a construção da cadeia de certificados) assim como da assinatura feita na *key* e no *nonce*.
6. O Repositório decifra a *key* e o *nonce* com a sua chave privada.
7. O Repositório gera um *session id*
8. O Repositório encripta o *session id* com uma cifra AES-CCM (cuja *key* foi recebida previamente e o *nonce* gerado aleatoriamente) e envia este *id* para o cliente.

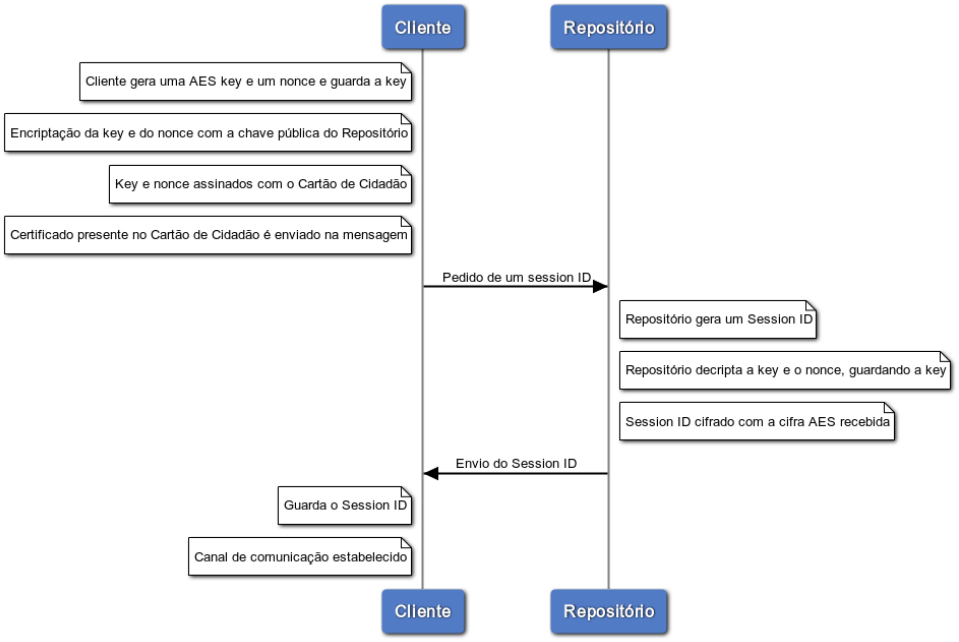

Figura 1: Estabelecimento de uma Sessão Segura entre o Cliente e o Repositório

A partir deste momento, todas as mensagens entre o cliente e o repositório serão encriptadas com recurso à *key* trocada e a um *nonce* aleatório enviado em todas as mensagens. Este processo aumenta a dificuldade de descobrir a *session key*.

2.1.2 Mensagens entre o repositório e o leiloeiro

A segurança da comunicação entre o repositório e o leiloeiro é garantida através de um canal seguro TLS.

Para tal, implementámos uma autenticação mútua com certificados gerados com *openssl*. Estes são assinados por uma Entidade Certificadora, garantido a validade de ambos. Sempre que se inicia a comunicação entre o repositório e o leiloeiro os certificados são trocados e é avaliada a sua validade, tendo em conta a sua cadeia entidades certificadoras - neste caso, apenas uma.

O canal de TLS, para além de garantir um canal seguro de troca de mensagens, permite também a autenticidade das mensagens, bem como a sua integridade.

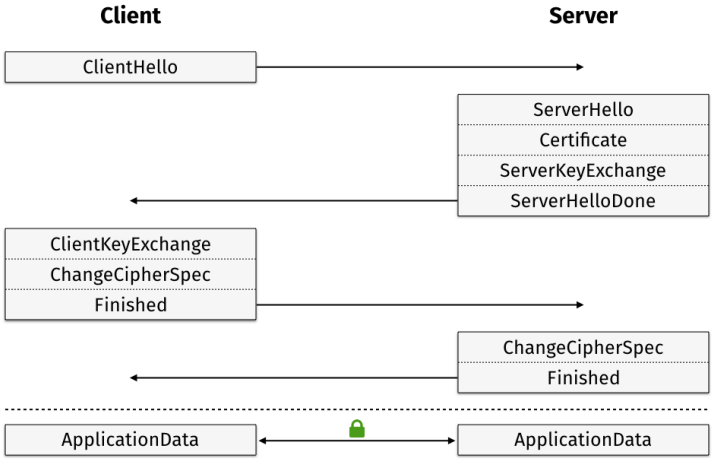

Figura 2: Protocolo TLS

2.2 Proteção das *bids* até ao fim do leilão

Relativamente a um leilão, todos as *bids* se encontram protegidas através do uso de cifra híbrida.

Dinamicamente, o criador de um dado leilão pode definir quais os campos de uma bid que pretende que sejam encriptados até ao final deste. Assim sendo, aquando da criação de um leilão, o criador vai gerar um par de chaves assimétricas (RSA). A chave pública irá ser inserida no bloco inicial da *blockchain* e será utilizada por todos os participantes do leilão para encriptar os campos definidos no leilão. Estes devem, sempre que inserem uma nova *bid*, consultar o bloco inicial e obter esta chave. Já a chave privada irá ser guardada pelo criador (esta será utilizada no final do leilão, como será posteriormente descrito).

Para que o criador do leilão não possa decriptar as *bids*, tendo assim acesso privilegiado, o leiloeiro, aquando da validação das bids, irá encriptar os campos com

uma cifra AES-CCM gerada para o efeito. A chave associada a esta cifra irá ser guardada pelo manager, contudo, sempre que este cifrar uma nova bid, irá adicionar o *nonce* ao novo bloco, para que posteriormente o possa utilizar e desencriptar os campos encriptados.

2.3 Identificação do autor de cada *bid* com o Cartão de Cidadão

De forma a autenticar as *bids*, o valor destas será sempre assinado pelo cliente, com recurso ao seu Cartão de Cidadão. Juntamente com a *bid* assinada, será ainda enviado o certificado que contém a chave pública associada ao cliente em questão. Este será encriptado com recurso a uma cifra AES-CCM.

Isto acontece uma vez que, devido ao seu elevado tamanho, o certificado não poderá ser assinado com uma cifra assimétrica RSA.

Tanto a *key* como o *nonce* gerados para a cifra AES serão posteriormente encriptados com a chave pública associado ao leilão (que pode ser encontrada no primeiro bloco da *blockchain* do mesmo) e, já na validação da *bid* pelo leiloeiro, encriptados com a sua cifra AES-CCM.

Relativamente às assinaturas utilizámos *PCKS#11*, assinando as bids com a chave contida no Cartão de Cidadão Português. Desta forma, validando a assinatura presente em cada bloco, através da chave pública contida no certificado do Cartão de Cidadão, podemos verificar qual o utilizador que submeteu cada bid.

Para utilizar o certificado acima referido como forma de verificar uma assinatura será necessário validar toda a sua cadeia de entidades certificadoras (CAs).

```
mechanism = PyKCS11.Mechanism(PyKCS11.CKM_SHA1_RSA_PKCS, None)
signature = None
    try:
        signature = bytes(session.sign(private_key, message, mechanism))
    except:
        # error solving code
```

Código 1: Assinatura com Recurso ao Cartão de Cidadão

2.4 Revelar as *bids* no final do leilão

Aquando o final do leilão, o criador do mesmo irá ter que colocar a chave privada RSA que criou para o mesmo no último bloco da *blockchain*. Contudo, esta chave por si só não é suficiente para decriptar todas as bids do leilão, uma vez que estas foram também encriptadas pelo manager. Desta forma, também este irá ter de colocar a sua chave simétrica (AES) no último bloco da *blockchain*, fechando-a.

Cada utilizador terá, então, que utilizar estas 2 chaves para desencriptar toda a blockchain e verificar a validade da mesma, podendo verificar se os seu recibos estão na blockchain, bem como saber quem foi o vencedor do leilão.

2.5 Validação das *bids* com recurso a código dinâmico

Sempre que se inicia um leilão, o seu criador terá de enviar um ficheiro em *python* com o código de validação das bids efetuadas. Esta metodologia permite que as bids sejam validadas dinamicamente, de acordo com a preferência do criador do leilão.

Posteriormente, o código inserido pelo criador será enviado para o leiloeiro, através do repositório. O leiloeiro reconstrói o código num ficheiro *python* e irá executá-lo com recurso ao **exec(...)** do *python*. Neste processo, é essencial definir que variáveis e *imports* podem ser utilizados dentro do código *python* a ser executado. Assim, não se deve permitir o acesso a, por exemplo, funções do módulo *os*.

Desta forma, implementámos a nossa solução respeitando o descrito acima.

Ao executarmos o ficheiro de validação fornecemos-lhe variáveis globais e locais, consoante a sua utilização.

```
exec(open("./validation_"+str(message['auction_id'])+".py").read(),
    variables4validation, script_locals)
```

Código 2: Execução de Código Dinâmico

2.6 Modificação das *bids* com recurso a código dinâmico

Após a validação das *bids*, o criador do leilão poderá também executar algumas modificações nas mesmas, através de código dinâmico. A sua utilização segue as mesmas premissas descritas acima.

No caso da nossa solução, geralmente optamos por enviar código dinâmico que defina a dificuldade de realização do *cryptopuzzle* associado a cada bloco, permitindo aumentar a dificuldade de inserção de blocos consoante o número de *bids* existentes.

Contudo, o utilizador poderá optar por enviar código que realize outra função.

```
block = None
blockchain = None

def change():
   l = len(blockchain.content)
   difficulty = (1/(-(l+ ((4 * blockchain.content[0].puzzle_difficulty) +
       1)))*5) + 4
   difficulty = round(difficulty)
   block.data["puzzle_difficulty"] = difficulty
   return block, difficulty

block, difficulty = change()
```

Código 3: Exemplo de uma Função de Modificação Dinâmica

2.7 Construção da *blockchain* associada a um leilão

Cada leilão está implementado com recurso a uma *blockchain*. Esta permite que cada utilizador valide a autenticidade e a validade de cada bloco da mesma.

Nesta, encontramos diversos blocos, cada um com informação acerca de uma *bid*. Estes contêm referências para os seu bloco pai (como se tratasse de uma *linked list*), o que permite verificar a ordem pelos quais foram inseridos. Em última instância, caso um bloco tenha sido alterado após a sua inserção, um utilizador conseguirá verificar a ocorrência deste evento.

Sempre que é inserido um novo bloco, o utilizador poderá verificar a integridade de toda a cadeia, que, caso tenha sido quebrada em algum momento, torna-se inválida, sendo que os dados contidos nesta deixarão de ser confiáveis.

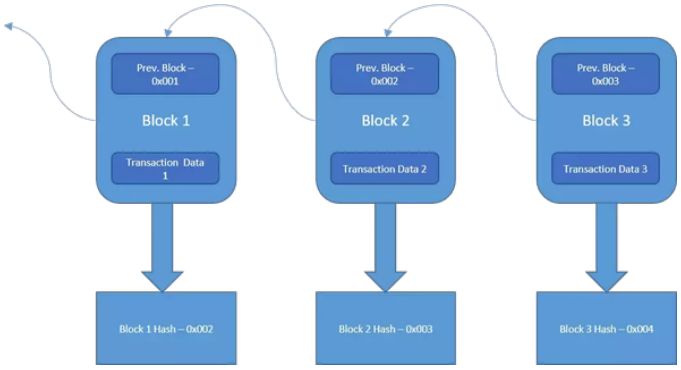

Figura 3: *Blockchain*

Segue-se o código associado a cada bloco da cadeia:

```
class Block:
    def __init__(self, prev_index, data, parent_hash):
        self.index = prev_index + 1
        # contains the key pair user-bid
        self.data = data
        self.puzzle_solution = None
        self.puzzle_difficulty = 1
        self.parent_hash = parent_hash
        self.nonce = int.from_bytes(os.urandom(8), byteorder="big")
        self.timestamp = date.datetime.now()

    def __setattr__(self, name, value):
        # records every time an element has been touched
        if name != "timestamp":
           self.timestamp = date.datetime.now()
        self.__dict__[name] = value
```

Código 4: Classe Block

2.8 Deployment dos *cryptopuzzles*

No contexto da nossa solução, podemos definir um *crypopuzzle* como sendo um desafio enviado para o cliente, pelo servidor. O cliente, terá que resolver este desafio (*proof of work*) e incluir a sua solução na resposta ao servidor, que posteriormente verificará se esta está correta.

Este mecanismo permite, também, diminuir a carga temporal de *requests* do servidor, uma vez que um *crypopuzzle* poderá demorar alguns segundos a ser resolvido, consoante a dificuldade exigida pelo servidor. Mais ainda, este poderá aumentar a dificuldade dos *crypopuzzle*, de forma a acompanhar o número crescente de *requests*.

Na solução implementada, aquando a inserção de um novo bloco na *blockchain* o cliente terá que resolver um *crypopuzzle* e incluir a sua solução no mesmo, que posteriormente será verificada, sendo este um fator importante para a validade da *blockchain*. Assim, o utilizador recebe um bloco com um *nonce random*, que irá incrementar até que a síntese do bloco se inicie por um conjunto de *bits* pré-definido.

Quando se verificar a condição descrita, a solução é colocada no bloco e enviada ao repositório que avaliará a sua veracidade, com base num processo bastante menos moroso, uma vez que não tem necessidade de calcular *nonces*.

```
def solve(difficulty, block_json):
   solution = hash_function(json.dumps(block_json))
   while solution[:difficulty] != "1"*difficulty:
       block_json["Nonce"] += 1
       block_json["Timestamp"] = str(datetime.now())
       solution = hash_function(json.dumps(block_json))
   block_json["Solution"] = solution
   return block_json
```

Código 5: Função Utilizada Para Resolver um *Cryptopuzzle*

2.9 Produção e validação dos recibos

Quando o repositório recebe o bloco com a solução do *crypopuzzle*, este valida-a e, caso esta esteja correta, insere o bloco na *blockchain* guardando neste o seu local de inserção, o seu bloco pai e o momento em que foi inserido na *chain*.

Posteriormente, como recibo, o repositório irá enviar o bloco inserido ao cliente. Este bloco contém informações encriptadas pelo leiloeiro e será assinado pelo repositório, de forma a garantir a autenticidade do mesmo. A assinatura do repositório assegura, também, a integridade do recibo.

Posteriormente, o recibo será recebido pelo cliente, que validará a assinatura do repositório e confirmará se os dado enviados por si, são aqueles que constam no recibo.

Se estas condições se verificarem, o recibo é considerado válido e é guardado em disco.

Posteriormente, quando a *blockchain* for decifrada, o utilizador poderá verificar todos os seus recibos.

```
# get blockchain and insert new block, validating it
bc = myAuction.auctions[auction_id][0]
ret = bc.insert_solved_block(tmp_block)

signature = myAuction.repository_pvk.sign(
   bytes(ret.puzzle_solution, 'utf-8'),
   padding.PSS(
       mgf=padding.MGF1(hashes.SHA256()),
       salt_length=padding.PSS.MAX_LENGTH
   ),
   hashes.SHA256()
)

# sign block and return it as a receipt
b = ret.to_json()
b["Solution"] = ret.puzzle_solution

return {"block" : json.dumps(b), "solution": ret.puzzle_solution,
    "signature" : base64.b64encode(signature).decode()}
```

Código 6: Inserção de um Bloco na *Blockchain* e Posterior Envio do Recibo do Mesmo

2.10 Validação de um Leilão Fechado (Por um Cliente)

No fim de um leilão, o criador do leilão e o leiloeiro inserem as suas chaves do leilão na *blockchain* (como descrito anteriormente), o que permite que um cliente consiga decifrar a totalidade da *blockchain*.

Após este processo, o cliente poderá validar a *blockchain* tendo em conta:

- **Autenticidade de cada *bid*:** cada cliente poderá verificar se a assinatura de uma *bid* é válida, através do uso da chave pública contida no certificado associado à *bid*.
- **Soluções dos *cryptopuzzles*:** o cliente poderá resolver eficientemente o

cryptopuzzle associado a cada bloco e verificar se a solução do mesmo se encontra associada a este.

- **Validação da ordem das *bids*:** Através das ligações estabelecidas entre os blocos é possível validar a ordem pela qual foram inseridos estes blocos.

Se conseguirmos validar todos os aspectos referidos acima, para cada bloco, então podemos considerar a *blockchain* como válida.

```
flag = True
deciphered_bids = []
for i in range(len(chain)):
   if "owner_key" in last_block[1]["Data"]:
       validation = validate_block(chain,i, AESCCM(manager_key),
           owner_key, cipher_fields)
       if i != 0 and i != len(chain) -1 :
           block = copy.copy(chain[i][1])
           if len(validation) > 2:
               block["Data"]["user"] = validation[2][0]
               block["Data"]["bid"] = validation[2][1]
               deciphered_bids.append(block)
       else:
           deciphered_bids.append(copy.copy(chain[i][1]))
   else:
       validation = validate_block(chain,i,None, None, None)
   flag = flag and validation[0]
return flag
```

Código 7: Validação da *Blockchain*

3 Extras

3.1 Autenticação de 2 fatores

De forma a garantir maior segurança na autenticação do cliente no servidor, este poderá enviar ao cliente uma *OTP*, via *SMS*.

Esta funcionalidade está implementada com uma api externa - Sinch. De momento, apenas estamos a providenciar o envio de *OTPs* para um número de telefone pré-definido, uma vez que o Cartão de Cidadão Português não tem nenhum número de telemóvel associado a um cidadão.

Contudo, esta solução poderia ser implementada com *smartcards* que guardassem o contacto do seu *owner*.

Relativamente à sua implementação, para que o repositório peça ao utilizar uma autenticação dupla, temos de iniciar este servidor com:

```
python3 auction_repository two_factor
```

Assim que o cliente iniciar uma conexão ao repositório este vai avaliar se irá ser necessária uma *OTP* e envia esta informação ao cliente. Ao mesmo tempo, envia um *SMS* com a *OTP* escolhida aleatoriamente.

```
if message["type"] == "otp_needed":
    if len(sys.argv) > 1 and sys.argv[1] == "two_factor":
        # define session OTP
        otp = ""
        while len(otp) < 6:
            n = secrets.randbelow(10)
            otp += str(n)

        myAuction.otp = otp

        # send sms to client
        number = '+351911111111'
        m = "OTP to access repository: " + otp
        client = SinchSMS("41da1648-fe23-49d2-9ff6-77d85ab9fe03",
            "fZmQ4p54RE6oDiklITnS6w==")

        print("Sending '%s' to %s" % (m, number))
```

```
        r = client.send_message(number, m)

        # enviar resposta
        resp = {'response' : 'yes'}
        resp = json.dumps(resp)
        conn.send(bytes(resp, 'utf-8'))
    else:
        # enviar resposta
        resp = {'response' : 'no'}
        resp = json.dumps(resp)
        conn.send(bytes(resp, 'utf-8'))
```

Código 8: Verificação e Envio da *OTP* via *SMS*

3.2 *TLS* entre repositório e leiloeiro

A segurança da comunicação entre o repositório e o leiloeiro é garantida através de um canal seguro *TLS*.

Para tal, implementámos uma autenticação mútua com certificados gerados com *openssl*. Estes são assinados por uma Entidade Certificadora, garantido a validade de ambos. Sempre que se inicia a comunicação entre o repositório e o leiloeiro, os certificados são trocados e é avaliada a sua validade, tendo em conta a sua cadeia de entidades certificadoras - neste caso, apenas uma.

O canal de *TLS*, para além de garantir um canal seguro de troca de mensagens, permite também a autenticidade das mensagens, bem como a sua integridade.

```
def connect_manager():
    man_addr = '127.0.0.1'
    man_port = 8080
    server_hostname = 'localhost'
    server_ca = 'sioCACertificate.pem'
    client_key = 'repositoryKey.key'
    client_cert = 'repositoryCertificate.pem'
    context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
    context.verify_mode = ssl.CERT_REQUIRED
    context.load_cert_chain(certfile=client_cert, keyfile=client_key)
    context.load_verify_locations(cafile=server_ca)
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    man_conn = context.wrap_socket(s, server_side=False,
        server_hostname=server_hostname)
    man_conn.connect((man_addr, man_port))
    return man_conn

man_conn = connect_manager()
```

Código 9: Autenticação Mútua entre o Repositório e o Leiloeiro com Recurso a *TLS*

3.3 Encriptação de Ficheiros

Relativamente aos recibos das *bids* de cada cliente, estes estão a ser guardados localmente, em disco, num diretório comum. Assim, de forma a que os clientes não consigam ver os recibos de outros clientes, desenvolvemos uma solução que encripta os ficheiros dos recibos de cada clientes.

Desta forma, usando o *7zip* encriptam-se, não só os recibos das *bids*, mas também as chaves que cada cliente cria para os leilões que este inicia.

Sempre que um cliente se autentica, é lhe pedida a *password* associada aos ficheiros, sendo que estes serão posteriormente comprimidos num zip, que será encriptado e desencriptado consoante a necessidade de escrever nos ficheiros que este contém.

```
import subprocess, shutil
from os import listdir, remove, _exists
import os
from os.path import isfile, join


def unzip(serial_number, pw):
    cmd = ['7z', 'x', serial_number + '.zip', '-o'+serial_number, '-p'+
        pw]

    p = subprocess.Popen(cmd, stderr=subprocess.DEVNULL,
        stdout=subprocess.DEVNULL)
    p.communicate()
    folder = serial_number
    onlyfiles = [f for f in listdir(folder) if isfile(join(folder, f))]
    for f in onlyfiles:
        cmd = ['mv', folder+'/'+f, '.']
```

```
        p = subprocess.Popen(cmd, stderr=subprocess.DEVNULL,
            stdout=subprocess.DEVNULL)
        p.communicate()
    shutil.rmtree(folder)
    os.remove(os.path.join('.', serial_number + '.zip'))

def zzip(serial_number, pw):
    cmd = ['7z', 'a', serial_number + '.zip', '-p' + pw, '-y',
        serial_number+'*']
    p = subprocess.Popen(cmd, stderr=subprocess.DEVNULL,
        stdout=subprocess.DEVNULL)
    p.communicate()
    for fname in os.listdir('.'):
        if fname.startswith(serial_number) and not fname.endswith('.zip'):
            os.remove(os.path.join('.', fname))
```

Código 10: Encriptação e Decriptação de Ficheiros

3.4 Guardar leilões em disco

De forma a prevenir que todos os leilões efetuados se percam no caso do repositório *crashar*, estes são guardados num ficheiro local.

Assim, utilizámos a biblioteca *pickle* para serializar a classe *python* que guarda todos os leilões, guardando esta informação num ficheiro local. Sempre que o repositório é iniciado, este tem em atenção os leilões previamente existentes.

Uma vez que há classes que não são serializáveis com o *pickle* - como é o caso de certificados do Cartão de Cidadão -, será necessário serializá-las "manualmente".

```
def load_pickle(f):
   # Load Pickle File
   my_file = Path(f)
   if my_file.is_file():
       myAuction = pickle.load( open(f,"rb") )
       # reconvert
       myAuction.repository_pvk = serialization.load_pem_private_key(
           myAuction.repository_pvk,
           password=None,
           backend=default_backend()
       )
```

```
    else:
        myAuction = AuctionInfo()

    return myAuction

def save_pickle(f):
    pem = myAuction.repository_pvk.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    )
    tmpkey = copy.copy(myAuction.repository_pvk)

    myAuction.repository_pvk = pem
    pickle_file = open("auction.pickle","wb")
    pickle.dump(myAuction, pickle_file)
    pickle_file.close()
    myAuction.repository_pvk = tmpkey
```

Código 11: Escrita e Leitura de Ficheiros com a Biblioteca *Pickle*

3.5 Leilão *single bid*

De forma a aumentar o conjunto de leilões disponíveis, criámos código para a realização/validação de um leilão *single bid*.

Neste tipo de leilão cada utilizador apenas poderá submeter uma bid.

4 Esquemas de apoio

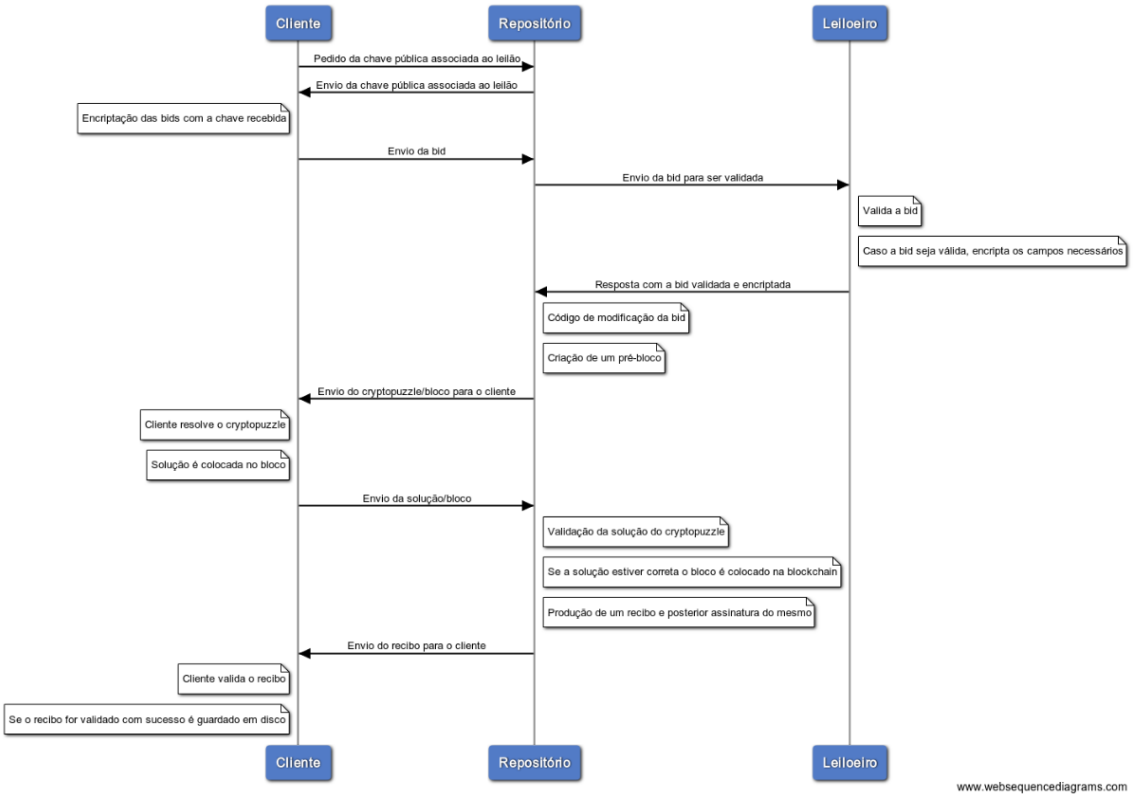

Figura 4: Submissão de uma *bid*

5 Informação adicional

Para poder utilizar a nossa solução é necessário abrir 3 terminais diferentes, um para cada actor do sistema, de acordo com a seguinte ordem:

1. Leiloeiro - deve ser executado com ***python3 auction_manager.py***
2. Repositório - deve ser executado com ***python3 auction_repository.py*** ou com ***python3 auction_repository two_factor.py***, caso se deseje utilizar autenticação de 2 fatores ao nível do repositório.
3. Cliente - deve ser executado com ***python3 client.py***

Todo o código pode ser encontrado num repositório do CodeUA. É possível fazer *clone* do repositório com o seguinte comando: ***git clone https://code.ua.pt/git/sio2018-p1g21***

6 Conclusão

Olhando em retrospectiva, consideramos que todos os objetivos relativos a este projeto foram atingidos. Sendo assim, e através de uma análise cuidada ao entregável final, pensamos que conseguimos implementar todas as funcionalidades que inicialmente projetámos.

Em suma, concluímos que, com este projeto, os conhecimentos na área de segurança, quer no seu planeamento, quer na sua efetiva implementação foram, sem qualquer dúvida, melhorados e que a capacidade de pesquisa e autonomia foram, também elas, bem desenvolvidas, na tentativa de obter soluções face aos problemas encontrados.

Referências

[1] ZÚQUETE, André Ventura. **Segurança em Redes Informáticas**. 4ª Edição. FCA - Editora de Informática

[2] https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_7.1.0/com.ibm.mq.doc/sy10670_.htm

[3] https://docs.python.org/3/library/ssl.html#ssl-security

[4] https://deliciousbrains.com/ssl-certificate-authority-for-local-https-development/

[5] https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/?highlight=rsa

[6] https://cryptography.io/en/latest/hazmat/primitives/padding/

[7] https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/

[8] https://medium.com/coinmonks/a-laymans-explanation-of-public-key-cryptography-and-digital-signatures-1090d4bd072e

[9] https://towardsdatascience.com/blockchain-explained-in-7-python-functions-c49c84f34ba5

[10] https://www.pythonsheets.com/notes/python-security.html