Blockchain-Based Auction Management
Relatório

Alunos:
Rafael Teixeira Nº 84746
Pedro Ferreira Nº 84735

Disciplina: Segurança Informática e Nas Organizações
Professores:
João Paulo Barraca
Vitor Cunha

Índice

Proteção das mensagens trocadas

Cliente-Servidor

Usamos as chaves assimétricas para fazer a negociação da chave simétrica a ser usada na sessão.

A chave assimétrica que usamos para cifrar a chave simétrica é a chave pública do AucionManager/AuctionRepository disponibilizada e distribuída durante a instalação do client.

Para autenticarmos ambas as partes a primeira mensagem do cliente é assinada com o seu CC, este envia juntamente o certificado que permite validar a sua autenticidade.

O certificado é validado [Validação de um Certificado] e depois o AuctionManager/ AuctionRepository encripta a sua próxima mensagem, um ACK, com a chave simétrica enviada pelo client, que comprava que este tem na sua posse a chave privada respetiva e assim autentica-o.

O client após esta interação envia o seu pedido para o AuctionManager/Repository encriptado com a mesma Chave.

A chave é um valor aleatório de 256 bits.

Todas as mensagens usam um nonce diferente que é um valor aleatório de 16 bytes e é transmitido entre as entidades em texto simples.

Na primeira mensagem a assinatura do client garante a integridade da mensagem e nas sequenciais o mecanismo usa AES-GCM continuando assim garantida a integridade.

```
simKey = AESGCM.generate_key(bit_length=256)
aesgcm = AESGCM(simKey)
nonce = secrets.token_bytes(16)
```

Image 1 [Instalação da chave do lado do cliente]

```
owner = cert.subject.get_attributes_for_oid(NameOID.COMMON_NAME)[0].value
simKey = privKey.decrypt(base64.b64decode(message['Key']), encrptPadd)
aesgcm = AESGCM(simKey)
nonce = secrets.token_bytes(16)
```

Image 2 [Instalação da chave do lado do servidor]

Servidor-Servidor

Comunicação TLS sobre sockets TCP, com recurso à biblioteca ssl de python.

Usam os certificados para servidor / cliente de cada um respetivamente à comunicação que querem realizar.

As opções usadas são:

```
context.options = ssl.OP_NO_SSLv2 | ssl.OP_NO_SSLv3 | ssl.OP_NO_TLSv1 | ssl.OP_SINGLE_DH_USE | ssl.OP_SINGLE_ECDH_USE
```

Image 3 [Opções usadas na definição do TLS]

evitando assim protocolos de comunicação vulneráveis, e o reuso de chaves em curvas elípticas evitando ataques men-in-the-middle.

A cifra escolhida é ECDHE-ECDSA-AES256-GCM-SHA384, evitando assim cifras fracas e obtendo um controlo de integridade sobre a mensagem trocada.

Os certificados trocados são verficados, sendo que têm de ter como objetivo de utilização (https server/client) e têm de ter como CA assinante o CA Auction_Signer criado para a aplicação e considerado a root destes certificados, é também verficado o nome do servidor, comparando com o que vem no certificado.

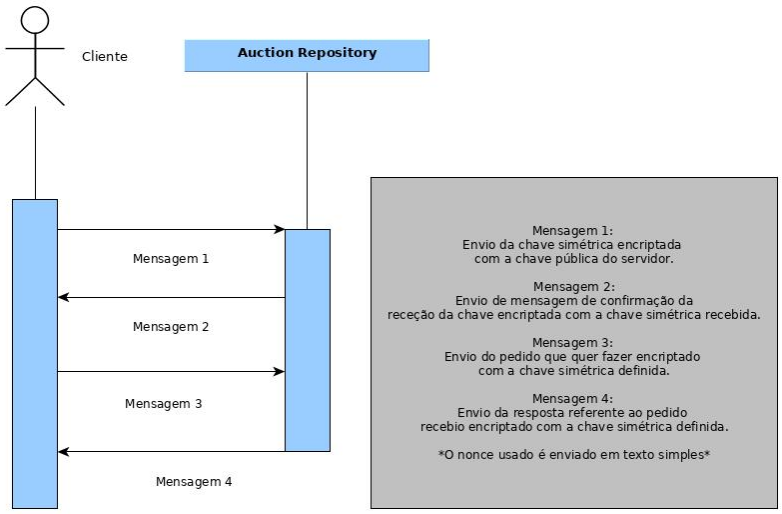

Image 4 [Comunicação entre o cliente e o servidor com troca de chaves e encriptação do canal]

Validação de um Certificado

Os certificados a validar são os certificados que o client coloca nas bids e o que usa para se autenticar perante um dos servidores.

Estes certificados são verificados da seguite forma:

- Período de Validade Correto
- Construção de uma cadeia completa até à entidade certificadora.
- Verificação das assinaturas feitas pelos certificados que emitiram os outros certificados
- Verificação dos porpósitos dos certificados (cert_sign ou digital_signature)
- Verificação da existência de algum dos certificados da cadeia formada em CRLs

A localização das CRLs é obtida através da extensão CRL_DISTRIBUTION_POINT, presente nos certificados do CC, e são obtidas através da biblioteca urllib que comunica com os respetivos servidores.

```
def getCRLs(certs):
    CRLs = []
    for cert in certs:
        for i in cert.extensions.get_extension_for_oid(ExtensionOID.CRL_DISTRIBUTION_POINTS).value:
            for dist_point in i.full_name:
                if dist_point.value not in CRLs:
                    CRLs.append(dist_point.value)
    crls = []
    for crl in CRLs:
        crls.append(x509.load_der_x509_crl(request.urlopen(crl).read(), default_backend()))
    return crls
```

Image 5 [Download das CRLs e carregamento das mesmas]

```
    for crl in crls:
        serial = chain[cert].serial_number
        if False:#crl.get_revoked_certificate_by_serial_number(serial) != None:
            return False
```

Image 6 [Verificação da existencia de certificados revogados na cadeia]

Proteção das bids até ao fim do seu auction

Cifra por parte do cliente:

Os campos das bids que são necessários proteger são cifrados pelo cliente antes de este as enviar ao repositório usando uma chave simétrica AES com uma chave aleatória de 32 bytes e um modo de cifra [CBC] e um nonce de 16 bytes diferente para cada campo, a chave simétrica usada é encriptada por sua vez com recurso a uma chave pública disponibilizada pelo Owner do auction que no fim do mesmo disponibiliza a chave privada e possibilita a desencriptação dos campos e verficação dos mesmos.

```
fields = ['Value', 'Cert', 'Signature'] #Fields of the bid to encrypt
key = secrets.token_bytes(32)
backend = default_backend()
algorithm = algorithms.AES(key)
iv_list = []

for field in fields:
    if field != 'Value':
        field_value = base64.b64decode(bid[field])
    else:
        field_value = bytes(str(bid[field]), 'utf-8')
    iv = secrets.token_bytes(16)
    iv_list.append(base64.b64encode(iv).decode('utf-8'))
    mode = modes.CBC(iv)
    cipher = Cipher(algorithm, mode, backend)
    encryptor = cipher.encryptor()
    padder = syPadding.PKCS7(128).padder()
    padded_data = padder.update(field_value) + padder.finalize()
    ct = encryptor.update(padded_data) + encryptor.finalize()
    ct = base64.b64encode(ct).decode('utf-8')
    bid[field] = ct

if self.pubKey != b'':
    ct = self.pubKey.encrypt(key, asyPadding.OAEP(mgf=asyPadding.MGF1(hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
    bid['Key'] = base64.b64encode(ct).decode('utf-8')
    bid['IV_list'] = iv_list
```

Image 7 [Encriptação dos campos da bid e encriptação da chave simétrica usada para isso]

Cifra por parte do auction manager:

Os mesmo campos que o client cifra são cifrados pelo auction manager com uma chave simétrica AES usando o mesmo método que o client, contudo as chaves usadas pelo auction manager são mantidas com ele até ao fim do auction sendo disponibilizadas ao mesmo tempo que a chave privada do owner, isto evita que haja um ponto de falha único, ou seja que o owner graças à sua chave privada consiga desencriptar o auction todo antes de este acabar, ou que o auction manager caso comprometido disponibilize as chaves e consequentemente o auction.

```
def encrypt(self, auctionId, bid):
    exec(self.auctions[auctionId][0], locals(), globals())
    self.auction_keys[auctionId].append((key, iv_list,))
    return bid
```

Image 8 [Encriptação e armazenamento da chave e dos ivs usados]

As cifras executadas são realizadas com código dinâmico possibilitando o uso de diferentes tipos e métodos de cifras, possibilitando ainda proteção de campos costumizada e liberdade para criação de novos auctions.

Para as implementações usadas decidimos usar AES com chaves de 32 bytes uma vez que é bastante popular e também robusta, juntamente com um modo de cifra que permite o reuso de chave para cifras diferentes, esconde padrões, dificulta a alteração do resultado e cria confusão no input da cifra

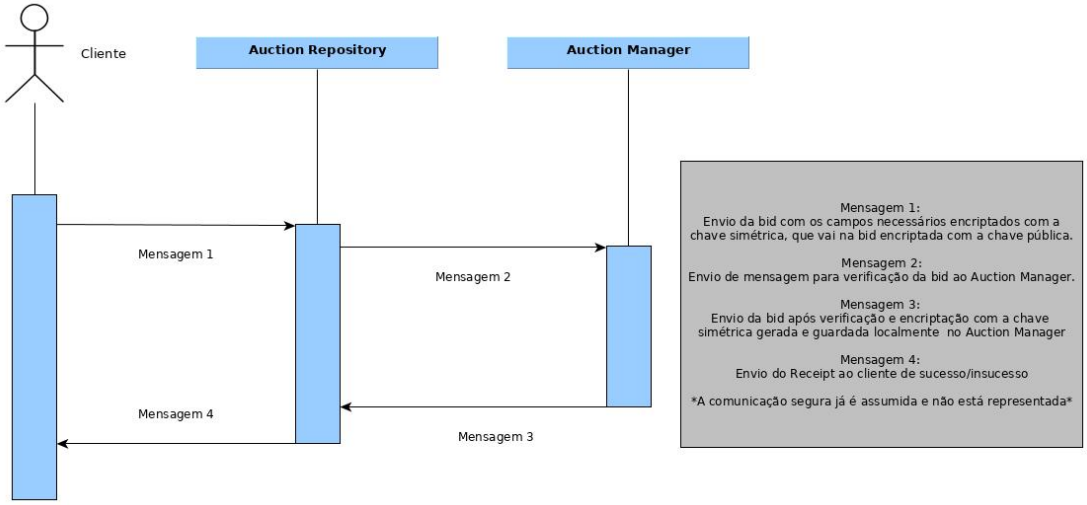

Image 9 [Troca de mensagens no processo de encriptação e validação das bids]

Identificação da Bid com o CC

As bids realizadas por cada cliente são autenticadas pelo mesmo pois estas são assinadas com recurso ao seu CC, usando a chave privada que este tem no mesmo. O certificado público respetivo é cifrado e colocado também na bid para que no fim do auction todos os participantes possam verificar a assinatura deste e também para que o AuctionRepository identifique quem é o vencedor.

Todos os campos da bid são assinados ainda em clear text, garantido assim a integridade da mesma face a possíveis alterações.

Exposição das bids necessárias no fim do auction

No fim do auction o owner disponibiliza a sua chave privada, assim como o auction manager disponibiliza as simétricas usadas para aquele auction. O auction repository ao recebe-las coloca-as como um novo e último bloco do auction e revela o vencedor ao mesmo tempo.

A verficiação do vencedor é também ela feita com recurso a código dinâmico.

O client depois se necessitar de verificar todo o auction este pode pedir a blockchain completa, retirar do primeiro bloco o código executado na validação, encriptação/desencriptação e na escolha do vencedor, do auction do último bloco as chaves para desencriptar as bids e depois de obter esta informação verificar toda as bids da chain [Validação de um Auction] comparando também com os seus recibos se necessário.

```
#Cria o link para o previous block
previousLink = self.finishedAuctions[auctionId].getLastBlock().getLink() + self.finishedAuctions[auctionId].getLastBlock().getRepSign()
digest.update(previousLink)
link =  digest.finalize()
text_to_sign = base64.b64decode(clientKey) + json.dumps(auctionManagerKeys).encode() + link
assin = self.key.sign(text_to_sign, self.padding, hashes.SHA256())
#Cria o bloco e adiciona à blockchain
block = Block({'ClientKey' : clientKey, 'AuctManKeys': auctionManagerKeys} , None, link, None, assin)
self.finishedAuctions[auctionId].addToBlockChain(block)
```

Image 10 [Construção do último bloco da chain]

```
padd = padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),algorithm=hashes.SHA256(),label=None)
keys = chain[-1].getContent()
manKeys = keys['AuctManKeys']
privKey = keys['clientKey']
key, iv_list = (manKeys[-1][0], manKeys[-1][1])
best = (None,None)
keyPriv = serialization.load_pem_private_key(base64.b64decode(privKey) , password = None, backend=default_backend())
decrypt = open('BlindDecrypt.py').read()

for i in range(1,len(chain)-2):
    bid = chain[i].getContent().getJson()

    exec(decrypt, locals(), globals())

    keys= keyPriv.decrypt(base64.b64decode(bid['Key']), padd)
    key = base64.b64decode(keys['Key'])

    iv_list = []
    for i in keys['Iv_list']
        iv_list.append(base64.b64decode(i))

    exec(decrypt, locals(), globals())
    if best[1] == None or best[1] < bid['Value']:
        best = (bid['Author'], bid['Value'])

winner = best[0]
```

Image 11 [Código de atribuição do vencedor]

Validação das Bids com recurso a código dinâmico

Como já foi referido anteriormente é de facto isto que acontece. Na criação do auction o owner define que código deve ser utilizado (existem funções pré-definidas para os auctions que o professor forneceu) o auction Manager guarda esse código (para uso futuro) e envia-o também juntamente com o resto da informação para o Auction Repository que o coloca no primeiro bloco da chain ficando disponível para todos.

Quando as bids chegam ao AuctionManager para serem validadas o AuctionManager executa o código que tinha guardado para o respetivo Auction validando-as.

```
bid_value = bid['Value']
if self.last_bid != {} and bid_value > self.last_bid['Value'] and (bid_value - self.last_bid['Value']) >= self.min_value:
    validBid = True
elif self.last_bid == {} and self.bid_value >= self.min_value:
    validBid = True
```

Image 12 [Código da execução de uma verificação]

Modificação das Bids com recurso a código dinâmico

Como vimos na proteção das bids esta é feita com recurso a código dinâmico.

Quanto ao processo de distribuição este é análogo ao anterior sendo o owner a disponibilizar o código (existindo já funções pré-definidas para os auctions que o professor forneceu) o auction manager guarda-o localmente e envia-o para o auction repository juntamente com o código de validação dinâmico e coloca-o no primeiro bloco.

```
fields = ['Value', 'Cert', 'Signature'] #Fields of the bid to encrypt
key = secrets.token_bytes(32)
backend = default_backend()
algorithm = algorithms.AES(key)
iv_list = []

for field in fields:
    if field != 'Value':
        field_value = base64.b64decode(bid[field])
    else:
        field_value = bytes(str(bid[field]), 'utf-8')
    iv = secrets.token_bytes(16)
    iv_list.append(base64.b64encode(iv).decode('utf-8'))
    mode = modes.CBC(iv)
    cipher = Cipher(algorithm, mode, backend)
    encryptor = cipher.encryptor()
    padder = syPadding.PKCS7(128).padder()
    padded_data = padder.update(field_value) + padder.finalize()
    ct = encryptor.update(padded_data) + encryptor.finalize()
    ct = base64.b64encode(ct).decode('utf-8')
    bid[field] = ct

if self.pubKey != b'':
    ct = self.pubKey.encrypt(key, asyPadding.OAEP(mgf=asyPadding.MGF1(hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
    bid['Key'] = base64.b64encode(ct).decode('utf-8')
    bid['IV_list'] = iv_list
```

Image 13 [código da encriptação]

Construção de uma blockchain por leilão

A blockchain que usamos é composta por blocos que contém para além do conteúdo suposto (Bid ou informações) um timestamp de colocação na chain, um link (digest do (link + assinatura do rep) do bloco anterior) a assinatura do rep de todos os campos entre outros.

A blockchain é garantidamente inalterada pois todos os campos do bloco anterior foram assinados e essa assinatura, juntamente com o link do bloco anterior ao bloco anterior a esse, são usados para criar a ligação a este, garantido assim não só a ligação ao bloco anterior como a inalteração do mesmo.

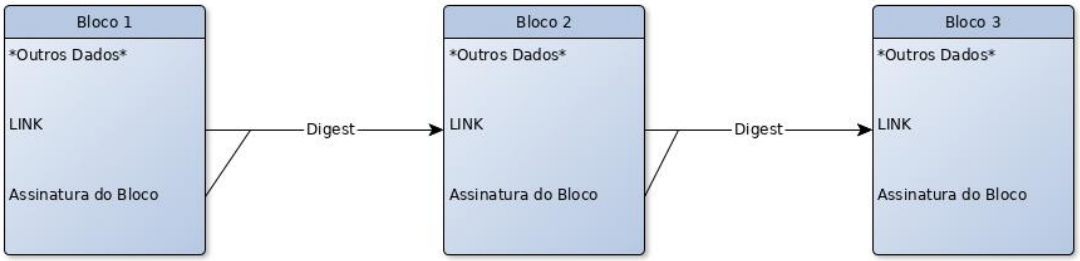

Image 14 [Ligação dos blocos na blockchain]

A hash usada para obter o digest é a SHA256 sendo uma hash que não tem ataques de colisão conhecidos, e produz um digest com um tamanho grande o suficiente para que o input mais parecido possível gere duas hashes diferentes.

Implatação de Criptopuzzles

Antes de um client colocar uma bid este tem que responder a um criptopuzzle, este é composto por um challenge (link do último bloco), uma dificuldade aleatória entre 1 e 3 (dificulta sniping) e um método de hash. O Client tem que gerar um nonce com 8 bytes e juntá-lo ao challenge que recebe até que o início do digest resultante tenha um número de zeros igual à dificuldade. Depois de encontrar a solução este envia a resposta, o nonce e a dificuldade, para que o auction repository verifique se este fez o desafio correto e se o resultado é verdadeiro.

```
def doChallenge(self, difficulty, link):
    nonce = secrets.token_bytes(8)
    hashF = hashes.Hash(hashes.SHA256(), backend=default_backend())
    hashF.update(nonce + link)
    digest = hashF.finalize()
    while(not digest[0:difficulty] == b'0'*difficulty):
        nonce = secrets.token_bytes(8)
        hashF = hashes.Hash(hashes.SHA256(), backend=default_backend())
        hashF.update(nonce + link)
        digest = hashF.finalize()
    return { 'Nonce' : nonce , 'Response' : digest, 'Difficulty' : difficulty }
```

Image 15 [código da implatação do criptopuzzle]

Produção e validação de recibos

Sempre que um cliente realiza uma bid ele recebe um recibo que indica se a mesma teve sucesso ou não, este recibo é assinado pelo auction repository e caso a bid tenha sucesso ele indica a posição em qua a bid ficou na chain e caso não tenha indica o motivo.

A verificação dos recibos é feita em dois passos, o primeiro é a verificação da assinatura do auction repository e o segundo é no fim do auction onde os recibos de sucesso são verificados (ver se o dono da bid na posição indicada pelo recibo é ele).

```
   def saveAndValidReceipt(self, rec):
       #Cria o Receipt a ser Guardado
       receipt = json.dumps({ 'TimestampRec' : rec['TimestampRec'], 'TimestampEnv' : rec['TimestampEnv'], 'Success' : rec['Success'], 'Pos' : rec['Pos'], 'Sign' : rec[
'Sign'] })
       #Carrega a Chave pública do AuctionRepository
       padd = asyPadding.PSS(mgf=asyPadding.MGF1(hashes.SHA256()), salt_length=asyPadding.PSS.MAX_LENGTH)
       repKey = x509.load_pem_x509_certificate(open("{}/certs_servers/AuctionRepository.crt".format(path), "rb").read() , backend=default_backend()).public_key()
       #Cria o texto de verificação de Assinatura
       text_to_sign = (rec['TimestampRec'] + rec['TimestampEnv'] + rec['Success'] + str(rec['Pos'])).encode()
       #Verifica a Assinatura
       try:
           repKey.verify(base64.b64decode(rec['Sign']), text_to_sign, padd, hashes.SHA256())
       except Exception:
           print("Error Validating Receipt: " + "Auction{}_Receipt{}.receipt".format(str(rec['AuctionId']), str(rec['ReceiptId'])))
       #Guarda o Receipt
       f = open("{}/receipts/Auction{}_Receipt{}.receipt".format(path, str(rec['AuctionId']), str(rec['ReceiptId'])), "w+")
       f.write(receipt)
       f.close()
```

Image 16 [Validação e armazenamento dos recibos]

Validação de um auction a decorrer

A validação de um leilão que ainda está a decorrer faz apenas a validação dos links e das assinaturas feitas pelo auction repository

```
 def verifyOnChain(chain):
    #Load da Chave do Repository
    padd = padding.PSS(mgf =padding.MGF1(hashes.SHA256()), salt_length = padding.PSS.MAX_LENGTH)
    repKey = x509.load_pem_x509_certificate(open("../certs_servers/AuctionRepository.crt", "rb").read() , backend=default_backend()).public_key()
    for i in range(len(chain)):
        if i != 0:
            #Verificação dos links da blockchain
            link = chain[i].getLink()
            digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
            previousLink = chain[i-1].getLink() + chain[i-1].getRepSign()
            digest.update(previousLink)
            if link !=  digest.finalize():
                return False
            try:
                #Verificação da Assinatura do Repositório (para os blocos com bids)
                link = chain[i].getLink()
                bid =  chain[i].getContent()
                challenge = chain[i].getChallenge()
                time = chain[i].getTimestamp()
                repKey.verify(chain[i].getRepSign(), bid.getAuthor() + bid.getValue() + link +str(time).encode()+ json.dumps(challenge).encode(), padd, hashes.SHA256())
            except Exception as e:
                print(e)
                return False
        else:
            try:
                #Verificação da Assinatura do Repositório para o primeiro bloco (com as regras do auction)
                link = chain[i].getLink()
                cont =  chain[i].getContent()
                verDin = cont['VerDin']
                encDin = cont['EncDin']
                repKey.verify(chain[i].getRepSign(), json.dumps(verDin).encode() + json.dumps(encDin).encode() + link, padd, hashes.SHA256())
            except Exception:
                return False

    return True
```

Image 17 [Verificação de um leilão ainda a decorrer]

Validação de um auction fechado

A validação de um auction fechado consistem em algumas tarefas que passo a enumerar:

- Verificar todas as assinaturas feitas pelo auction repository nos blocos
- Verificar todos os links entre os blocos
- Verificar todas as assinaturas e certificados nas bids
- Verificar a validade de todas as bids
- Verificar a validade do criptopuzzle
- Verificar os receipts do cliente que executa a verificação
- Verificar o vencedor do Auction

```
        return False
else:
    try:
        #Verificação da Assinatura do Repositório (para o bloco final com as chaves)
        link = chain[i].getLink()
        bid =  chain[i].getContent()
        challenge = chain[i].getChallenge()
        time = chain[i].getTimestamp()
        repKey.verify(chain[i].getRepSign(), bid.getAuthor() + bid.getValue() + link +str(time).encode()+ json.dumps(challenge).encode(), padd, hashes.SHA256())

        bid = bid.getJson()
        key = base64.b64decode(manKeys[i-1][0])

        iv_list = []
        for i in manKeys[i-1][1]:
            iv_list.append(base64.b64decode(i))

        exec(decrypt, locals(), globals())

        exec(valDin, locals(), globals())

        if not validBid:
            return validBid

        keys = keyPriv.decrypt(base64.b64decode(bid['Key']), padd)
        key = base64.b64decode(keys['Key'])

        iv_list = []
        for i in keys['IV_list']:
            iv_list.append(base64.b64decode(i))

        cert = x509.load_pem_x509_certificate(bid.getCert(), backend=default_backend())
        cert_chain =  build_chain([], cert, [], certs)
        if not checkChain(chain, crls):
            return False

        #Verificação da Assinatura da Bid
        cliKey = x509.load_pem_x509_certificate(bid.getCert(), backend=default_backend()).public_key()
        cliKey.verify(bid.getSignature(), bid.getAuthor() + bid.getValue() +str(bid.getTimestamp()).encode()+ str(bid.getCriptAnswer()).encode() + bid.getCert() + bid.getKey(), padd, hashes.SHA256())

        #Verificação do criptopuzzle
        digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
        response = bid['CriptAnswer']
        nonce = response['Nonce']
        digest.update(nonce.encode() + base64.b64decode(challenge['Challenge']))
        result =  digest.finalize()
        if result[0:challenge['Difficulty']] != b'0'*challenge['Difficulty'] or result != response['Response']:
            return False

        #Verificação dos seus receipts
        if i in pos and bid.getAuthor() != user:
            return False
                                                                                                                                    235,38        93%
```

Image 18 [Verificações feitas para as bids (Não inclui o 1º e o último bloco)]