Projeto 3 - Autenticação

Trabalho Realizado por: Vasco Ramos - nº mec 88931
Diogo Silva - nº mec 89348

Data de Preparação: Aveiro, 13 de Dezembro de 2019

Cadeira: Segurança Informática e nas Organizações

Corpo Docente: Professor João Paulo Barraca
Professor Vítor Cunha

universidade de aveiro deti departamento de electrónica, telecomunicações e informática

Índice

Introdução **3**

1. Planeamento **4**

2. Desenho **5**

3. Implementação **6**

3.1 Autenticação do Servidor 6

3.2 Autenticação do Cliente - Password 11

3.3 Autenticação do Cliente - Cartão de Cidadão 14

3.4 Controlo de Acesso 16

4. Execução **17**

Exemplo de Execução - rick.txt - Using CC 17

Exemplo de Execução - rick.txt - Using Password 19

5. Conclusão **21**

6. Bibliografia e Referências **21**

Introdução

No seguimento do plano curricular da disciplina de Segurança Informática e nas Organizações, da Licenciatura em Engenharia Informática, da Universidade de Aveiro, este relatório é o resultado da execução do terceiro trabalho prático e tem como principal objetivo explorar os conceitos relacionados com o estabelecimento de uma sessão segura entre dois interlocutores e com a autenticação dos intervenientes na comunicação e o controlo de acesso dos clientes.

Ao longo deste documento vamos apresentar:

- O **planeamento** da solução;
- A **descrição** do protocolo;
- A **implementação** do sistema.

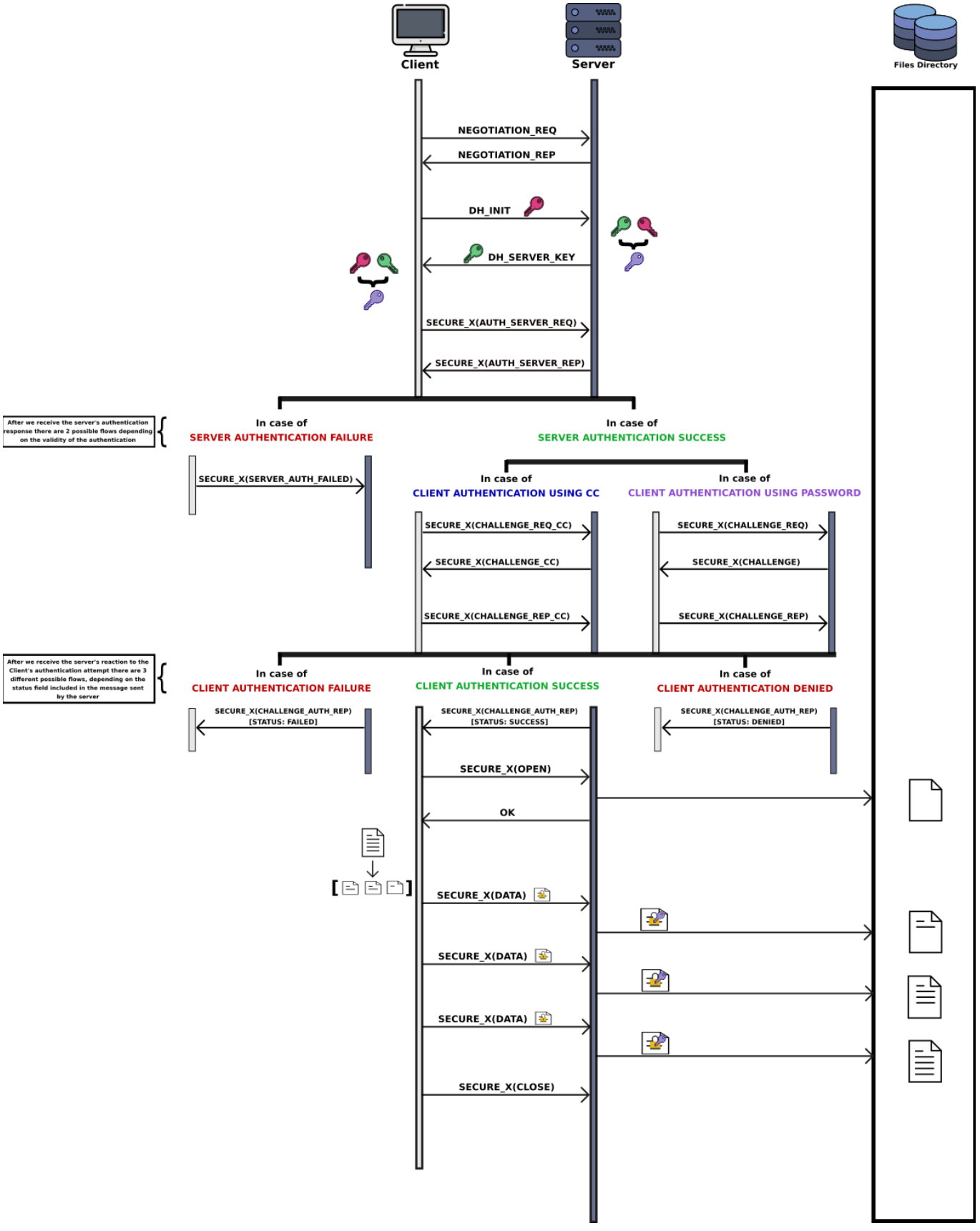
1. Planeamento

Numa primeira fase, abordámos o problema de um ângulo mais teórico de forma a planearmos, o melhor possível, o que seria necessário fazer.

Assim, focámo-nos nos tópicos descritos no guião:

1. **Desenhar um protocolo** para o estabelecimento de uma **sessão segura** entre cliente e servidor, suportando:
    a. Autenticação do cliente através do cartão de cidadão;
    b. Autenticação do cliente através de desafio resposta;
    c. Controlo de acesso;
    d. Autenticação do servidor utilizando certificados X.509.
2. **Implementar a autenticação do cliente**, através da apresentação do cartão de cidadão .
3. **Implementar a autenticação do cliente**, através da apresentação de senhas (desafio-resposta) .
4. **Implementar o suporte para o controlo de acesso**.
5. **Implementar a autenticação do servidor**, através de certificados X.509.
6. **Implementar funções genéricas** de validação da cadeia de certificados, criação de assinaturas e validação das mesmas, entre outras.

2. Desenho


Client
Server
Files Directory
NEGOTIATION_REQ
NEGOTIATION_REP
DH_INIT
DH_SERVER_KEY
SECURE_X(AUTH_SERVER_REQ)
SECURE_X(AUTH_SERVER_REP)
After we receive the server's authentication response there are 2 possible flows depending on the validity of the authentication
In case of
SERVER AUTHENTICATION FAILURE
In case of
SERVER AUTHENTICATION SUCCESS
SECURE_X(SERVER_AUTH_FAILED)
In case of
CLIENT AUTHENTICATION USING CC
In case of
CLIENT AUTHENTICATION USING PASSWORD
SECURE_X(CHALLENGE_REQ_CC)
SECURE_X(CHALLENGE_CC)
SECURE_X(CHALLENGE_REP_CC)
SECURE_X(CHALLENGE_REQ)
SECURE_X(CHALLENGE)
SECURE_X(CHALLENGE_REP)
After we receive the server's reaction to the Client's authentication attempt there are 3 different possible flows, depending on the status field included in the message sent by the server
In case of
CLIENT AUTHENTICATION FAILURE
In case of
CLIENT AUTHENTICATION SUCCESS
In case of
CLIENT AUTHENTICATION DENIED
SECURE_X(CHALLENGE_AUTH_REP)
[STATUS: FAILED]
SECURE_X(CHALLENGE_AUTH_REP)
[STATUS: SUCCESS]
SECURE_X(CHALLENGE_AUTH_REP)
[STATUS: DENIED]
SECURE_X(OPEN)
OK
SECURE_X(DATA)
SECURE_X(DATA)
SECURE_X(DATA)
SECURE_X(CLOSE)

3. Implementação

3.1 Autenticação do Servidor

A autenticação do servidor é realizada, utilizando certificados X.509. Este processo começa com a geração de um **NONCE**, do lado do cliente. A mensagem que inicia esta fase contém:

- Nonce;
- Chave pública do cliente.

A chave pública do cliente será usada para validar as assinaturas do cliente, mais à frente.

Para tal, o cliente faz o pedido ao servidor através de uma mensagem do tipo **AUTH_SERVER_REQ**:

```
self.nonce = os.urandom(16)

req = {
    "type": "AUTH_SERVER_REQ",
    "nonce": base64.b64encode(self.nonce).decode(),
    "public_pem": base64.b64encode(self.rsa_public_pem).decode(),
}

message = crypto_funcs.create_secure_message(
    req,
    self.shared_key,
    self.used_symetric_cipher,
    self.used_cipher_mode,
    self.used_digest_algorithm,
)

self.state = STATE_SERVER_AUTH
self._send(message)
```

Ao receber esta mensagem, o servidor processa a informação, guarda o pem da chave pública do cliente para usar mais tarde e carrega o seu certificado, o certificado da sua raíz e a chave privada associada ao seu certificado. Com a sua chave privada, assina o **NONCE** enviado pelo cliente e envia essa assinatura, o seu certificado e o certificado da sua raíz, através de uma mensagem do tipo **AUTH_SERVER_REP**:

```
self.server_cert = crypto_funcs.load_certificate(
    "server_certs/secure_server.pem")

self.server_ca_cert = crypto_funcs.load_certificate(
    "server_roots/Secure_Server_CA.pem")

self.rsa_public_key = self.server_cert.public_key()
self.rsa_private_key = crypto_funcs.load_private_from_pem(
    "server_certs/server_key.pem")

nonce = message["nonce"]
signature = crypto_funcs.rsa_signing(
    base64.b64decode(nonce), self.rsa_private_key
)

message = {
    "type": "AUTH_SERVER_REP",
    "signature": base64.b64encode(signature).decode(),
    "server_cert": crypto_funcs.get_certificate_bytes(
        self.server_cert
    ).decode(),
    "server_root": crypto_funcs.get_certificate_bytes(
        self.server_ca_cert
    ).decode(),
}

sec_message = crypto_funcs.create_secure_message(
    message,
    self.shared_key,
    self.used_symetric_cipher,
    self.used_cipher_mode,
    self.used_digest_algorithm,
)

self._send(sec_message)
```

O cliente, após receber e processar esta mensagem, valida a assinatura que veio do servidor, utilizando a chave pública deste:

```
val_signature = crypto_funcs.validate_rsa_signature(
    signature, self.nonce, self.server_public_key
)
logger.info(f"Server signature validation: {val_signature}")
if not val_signature:
    return False
```

De seguida, valida se o *common name* do certificado do servidor e o nome do servidor que é suposto estar a ser contactado são iguais:

```
val_common_name = self.host_name == crypto_funcs.get_common_name(
    self.server_cert
)
logger.info(f"Server common_name validation: {val_common_name}")
if not val_common_name:
    return False
```

O passo final deste processo é criação da cadeia de certificados e consequente validação de cada um dos certificados presentes na mesma.

1. Validar a data de expiração

```
today = datetime.now().timestamp()
return (cert.not_valid_before.timestamp() <= today <=
     cert.not_valid_after.timestamp()
)
```

2. Validar o *purpose*

```
if indx == 0:
    for c in cert.extensions.get_extension_for_class(x509.ExtendedKeyUsage).value:
        if c.dotted_string == "1.3.6.1.5.5.7.3.1":
            return True
    return False
else:
    return cert.extensions.get_extension_for_class(
        x509.KeyUsage
    ).value.key_cert_sign
```

Ao validar o *purpose* dos certificados, o primeiro certificado tem uma validação diferente dos demais.

Neste, por ser o certificado do servidor, é necessário garantir que inclui a *KeyUsage* **SERVER_AUTH**.

Nos restantes certificados temos de garantir que incluem a KeyUsage **KEY_CERT_SIGN**.

3. Validar a assinatura do certificado

```
cert_signature = cert.signature
issuer_public_key = issuer.public_key()

try:
    issuer_public_key.verify(
        cert_signature,
        cert.tbs_certificate_bytes,
        padding.PKCS1v15(),
        cert.signature_hash_algorithm,
    )
except:
    return False
return True
```

4. Validar o *common name* do certificado

```
return get_issuer_common_name(cert) == get_common_name(issuer)
```

5. Validar o se o certificado está ou não revogado

```
try:
    builder = ocsp.OCSPRequestBuilder()

    builder = builder.add_certificate(cert, issuer, SHA1())
    req = builder.build()

        for ext in cert.extensions.get_extension_for_class(
            x509.AuthorityInformationAccess
        ).value:
            if ext.access_method.dotted_string == "1.3.6.1.5.5.7.48.1":
                data = req.public_bytes(serialization.Encoding.DER)

                ocsp_url = ext.access_location.value
                request = requests.post(
                    ocsp_url,
                    headers={"Content-Type": "application/ocsp-request"},
                    data=data,
                )

                ocsp_resp = ocsp.load_der_ocsp_response(request.content)
                logger.warning(f"OCSP CERT STATUS: {ocsp_resp.certificate_status}")
                if ocsp_resp.certificate_status == ocsp.OCSPCertStatus.GOOD:
                    return False
                else:
                    return True
except:
    logger.debug("OCSP is not available for this certificate!")
```

```
try:
    for ext in cert.extensions.get_extension_for_class(
        x509.CRLDistributionPoints
    ).value:
        for name in ext.full_name:
            file_name = wget.download(name.value)

            revocation_list = load_certificate_crl(file_name)

            if revocation_list is None:
                return False

            cert_is_revoked = cert.serial_number in [
                l.serial_number for l in revocation_list
            ]


    for ext in issuer.extensions.get_extension_for_class(
        x509.CRLDistributionPoints
    ).value:
        for b in ext.full_name:
            file_name = wget.download(b.value)

            revocation_list = load_certificate_crl(file_name)

            if revocation_list is None:
                return False

            isser_is_revoked = issuer.serial_number in [
                l.serial_number for l in revocation_list
            ]

    return cert_is_revoked or isser_is_revoked
except:
    logger.debug("CRL is not available for this certificate!")

return True
```

NOTA: Na validação do estado de revogação, inicialmente, tentamos OCSP, e só depois, tentamos CRL. Implementámos também para analisar as DELTA CRL, mas ocultámos do relatório para o manter o menos extenso possível.

Caso todos os certificados passem este conjunto de validações, então a cadeia é validada e o cliente transita para a fase da sua própria autenticação.

NOTA 2: Para criarmos os certificados utilizados na representação do servidor, usámos o software **XCA** e exportámos os mesmos para o formato PEM. Criámos também uma CRL que disponibilizámos num link para download online, de forma a esta ser incluída nos nossos certificados, permitindo assim a validação do estado de revogação do certificado.

3.2 Autenticação do Cliente - Password

Uma das opções de autenticação do Cliente é usando um sistema de credenciação de senhas. No nosso projeto pode ser encontrada uma pasta chamada **credentials_bd** que possui um dois ficheiros:

- **users.py** : Script python que gera um ficheiro csv com vários users aleatórios com Username, Password e Permissions
- **users.csv** : O ficheiro gerado pelo users.py que possui várias contas de clientes que poderão ser utilizadas para autenticação.

```
Username;Password;;Permissions
leonorramos0@ua.pt;lBYe3u(DPHpCG&~qau/<;;-
mariaramos1@ua.pt;Q%VsaRmBtOcqGmG8=!%g;;-
matildebastos2@ua.pt;PE,-LOhSUL9y^R~c-rOs;;t
sofiaandrade3@ua.pt;I!vsbZeLQ6}]=%y:l8ti;;t
franciscobastos4@ua.pt;wdM1q?1-JZZ%u`^UEn0f;;-
rodrigosilva5@ua.pt;E]!(R=gA8In5/{5EyK-);;t
mariaoliveira6@ua.pt;)LP9):KM3d1hCZGHFSU1;;t
rodrigovasconcelos7@ua.pt;-<t1*ka+p`u@J9<M@W}<;;-
vascoalmeida8@ua.pt;U}7)gTSB,+[HgV%KjJ#g;;t
sofiaandrade9@ua.pt;s>@D#Z5F!@Fx5&0#nXnx;;t
OOF;OOF;;t
```

Em termos de processo, o Cliente começa por enviar uma mensagem de **CHALLENGE_REQ.** Do lado do servidor este vai receber esta mensagem e gerar um **nonce**, iniciando assim um Challenge Protocol. Este nonce é incluído numa mensagem **CHALLENGE** e enviado para o Cliente.

Quando o Cliente receber a mensagem **CHALLENGE**, vai ser pedido ao utilizador que insira um **username** e uma **password**. A resposta ao desafio do servidor será composta pela concatenação do **nonce do servidor** à **password inserida** encriptada utilizando a **RSA**

Key Privada do Cliente. Tanto a resposta encriptada como o username serão então incluídos numa mensagem **CHALLENGE_REP** e enviada para o Server:

```
username = input("Username: ")
password = getpass.getpass("Password: ")

answer = crypto_funcs.rsa_signing(
       (str(nonce) + password).encode("utf-8"), self.rsa_private
)
rep_message["answer"] = base64.b64encode(answer).decode()
rep_message["username"] = base64.b64encode(username.encode("utf-8")).decode()

message = crypto_funcs.create_secure_message(
        rep_message,
        self.shared_key,
        self.used_symetric_cipher,
        self.used_cipher_mode,
        self.used_digest_algorithm,
)
```

De novo do lado do Server, este vai pegar na resposta e no username recebidos na mensagem **CHALLENGE_REP** e vai:

1. Verificar se o username existe no ficheiro Users.csv
    a. Se não existir a validação falha
2. Pega na password associada ao username e constrói a **resposta correta** (concatenando o mesmo nonce que enviou ao cliente à password que está no ficheiro)
3. Utilizando a **RSA Publica** do Cliente, o Server vai chamar a função de validação de assinaturas do RSA do CryptographyIO para verificar se a resposta recebida pelo cliente está correta
    a. Caso esteja correta, é enviada uma mensagem **CHALLENGE_AUTH_REP** com o **STATUS: SUCCESS**
    b. Caso contrario é enviada na mesma uma **CHALLENGE_AUTH_REP** mas desta vez com o **STATUS: FAILED**

```
with open("credentials_db/users.csv") as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=";")

    line_count = 0
    user_password = None
    user_permissions = None
    for row in csv_reader:
        logger.debug("EMAIL: " + row[0] + " RECEIVED: " + username
            + " EQUALITY: " + str(row[0] == username))
        if line_count == 0:  # Ignore header row
            line_count += 1
            continue

        else:
            if row[0] == username:  # We found our username
                user_password = row[1]
                user_permissions = row[3]
                break

        line_count += 1

    if user_password is not None:  # If we found our user
        correct_answer = (str(self.nonce) + user_password).encode("utf-8")

        public_pem = crypto_funcs.load_public_from_pem(self.client_public_pem)

        if not crypto_funcs.validate_rsa_signature(
            answer, correct_answer, public_pem
        ):
            rep = {"type": "CHALLENGE_AUTH_REP", "status": "FAILED"}

            message = crypto_funcs.create_secure_message(
                rep,
                self.shared_key,
                self.used_symetric_cipher,
                self.used_cipher_mode,
                self.used_digest_algorithm,
            )

            self._send(message)
            return False
```

3.3 Autenticação do Cliente - Cartão de Cidadão

A outra opção de autenticação do Cliente é usando o cartão de cidadão. Tal como no protocolo por senha, começamos por enviar uma mensagem para o servidor, após este ter sido validado, através de uma mensagem encriptada **CHALLENGE_REQ_CC**.

Após receber esta mensagem, o servidor gera um **NONCE** como desafio para enviar ao cliente através da mensagem **CHALLENGE**:

```
self.nonce = os.urandom(16)
challenge_message = {
    "type": "CHALLENGE",
    "nonce": base64.b64encode(self.nonce).decode(),
}

message = crypto_funcs.create_secure_message(
    challenge_message,
    self.shared_key,
    self.used_symetric_cipher,
    self.used_cipher_mode,
    self.used_digest_algorithm,
)

self.state = STATE_CLIENT_AUTH
self._send(message)
```

Novamente no cliente, após receber a mensagem, este irá introduzir o seu **username** para o servidor validar o controlo de acesso (explicado no ponto 3.4) e assinar o **NONCE** recebido com o seu cartão de cidadão. Esta assinatura, o username e o certificado serão enviados para o servidor através da mensagem **CHALLENGE_REP**:

```
logger.info("Replying to Challenge Authentication")

username = input("Username: ")

rep_message = {"type": "CHALLENGE_REP_CC", "answer": None, "cert": None}
answer, cert = crypto_funcs.sign_with_cc(nonce)
rep_message["answer"] = base64.b64encode(answer).decode()
```

```
rep_message["cert"] = base64.b64encode(cert).decode()
rep_message["username"] = base64.b64encode(username.encode("utf-8")).decode()

message = crypto_funcs.create_secure_message(
    rep_message,
    self.shared_key,
    self.used_symetric_cipher,
    self.used_cipher_mode,
    self.used_digest_algorithm,
)
```

Ao receber e processar esta mensagem, o servidor irá verificar se o *username* existe no sistema e, caso exista, irá validar a assinatura enviada pelo cliente, utilizando a chave pública do certificado recebido.

Por fim, à semelhança do que acontece na validação da cadeia de certificação do servidor, o servidor irá construir a cadeia de certificados associada ao cartão de cidadão do cliente, validando cada um dos certificados quanto à data de expiração, *purpose*, assinatura do certificado, *common name* e revogação do mesmo.

NOTA: É importante mencionar, que, aquando da validação se o certificado da raiz do estado está revogado através de OCSP, o estado que recebemos é **REVOKED**. Após pesquisa externa e diálogo com os docentes, acabámos por não conseguir encontrar solução ideal. Deste modo, acabámos por usar uma aproximação simplificada de uma sugestão do docente Vítor Cunha que se traduziu em considerar que, neste caso específico, o certificado está validado.

3.4 Controlo de Acesso

Para além da autenticação dos Clientes, foi também implementado um sistema de controlo de acessos que previne clientes de carregar ficheiros para o servidor caso não possuam permissão suficiente para tal.

Este controlo foi implementado tanto para a autenticação por Password como por CC da seguinte formas:

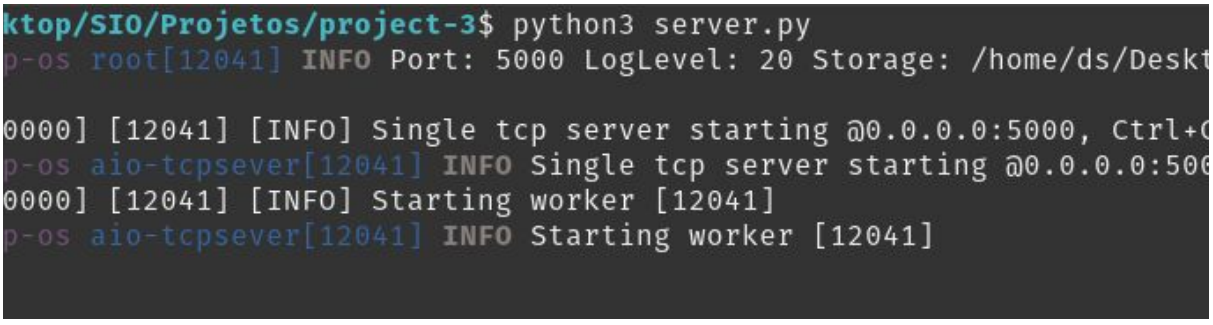
- Um dos campos no **Users.csv** é o de permissões, sendo que este tem o valor "**t**" caso o utilizador possua permissões ou "**-**" caso contrario.
- Após a verificação da autenticidade do user (quer por CC ou por Password) é verificado o valor deste campo na row do username especificado e mandada uma mensagem de **CHALLENGE_AUTH_REP** com o valor **STATUS: DENIED** caso este não tenha permissão.

```
if user_permissions != "t":
    rep = {"type": "CHALLENGE_AUTH_REP", "status": "DENIED"}

    message = crypto_funcs.create_secure_message(
        rep,
        self.shared_key,
        self.used_symetric_cipher,
        self.used_cipher_mode,
        self.used_digest_algorithm,
    )

    self._send(message)
    return False
```

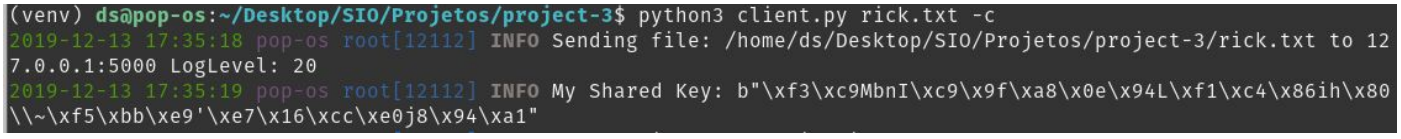
4. Execução

Exemplo de Execução - rick.txt - Using CC

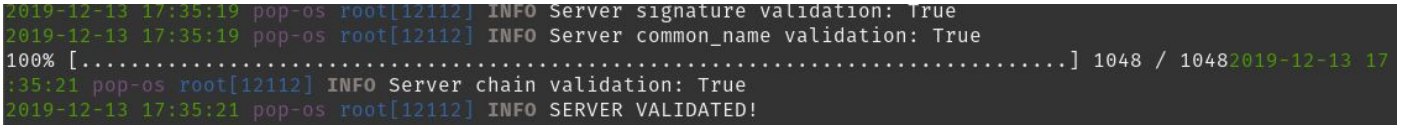
Neste exemplo vamos explicar de forma explícita todo o processo da transferência do ficheiro *rick.txt* do cliente para o servidor utilizando autenticação do Cliente com o CC:



Img 4.1 - Inicialização do Servidor.



Img 4.2 - Inicialização do Cliente, especificando o ficheiro rick.txt e a opção -c que especifica que queremos utilizar o CC para autenticação do cliente.



Img 4.3 - O Cliente pede ao servidor para que se autentique enviando-lhe um desafio e validando a resposta através da verificação da chain.

```
2019-12-13 17:35:21 pop-os root[12112] INFO SERVER VALIDATED!
2019-12-13 17:35:21 pop-os root[12112] INFO REQUEST LOGIN WITH CC
2019-12-13 17:35:21 pop-os root[12112] INFO PROCESING REQUEST CHALLENGE CC
2019-12-13 17:35:21 pop-os root[12112] INFO Replying to Challenge Authentication
Username: OOF
```

Img 4.4 - O Cliente pede ao servidor para que este lhe mande um desafio para se poder autenticar (o cliente) usando o CC. O servidor responde com o desafio, o cliente responde-lhe e o servidor verifica se a resposta está correta validando (ou não) o cliente.

```
2019-12-13 17:50:29 pop-os root[12721] INFO VALIDATING CC CHALLENGE REPLY
2019-12-13 17:50:29 pop-os root[12721] INFO VALIDATING REPLY
2019-12-13 17:50:29 pop-os root[12721] WARNING Not pem!
2019-12-13 17:50:29 pop-os root[12721] WARNING Not pem!
2019-12-13 17:50:30 pop-os root[12721] WARNING OCSP CERT STATUS: OCSPCertStatus.GOOD
2019-12-13 17:50:30 pop-os root[12721] WARNING OCSP CERT STATUS: OCSPCertStatus.GOOD
2019-12-13 17:50:30 pop-os root[12721] WARNING OCSP CERT STATUS: OCSPCertStatus.GOOD
2019-12-13 17:50:30 pop-os root[12721] WARNING OCSP CERT STATUS: OCSPCertStatus.REVOKED
2019-12-13 17:50:30 pop-os root[12721] WARNING Validation2: True
2019-12-13 17:50:30 pop-os root[12721] INFO ACCEPTING VALIDATION
```

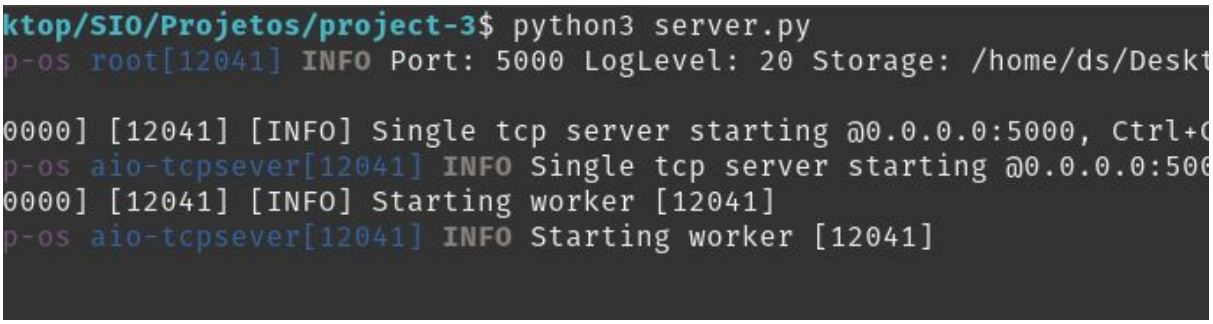
Img 4.5 - O Servidor verifica se o Cliente é válido (i.e se a autenticação for feita com sucesso e este não for denied) e responde com uma mensagem de sucesso, dizendo ao cliente que pode iniciar a transferência do ficheiro.

```
2019-12-13 17:50:30 pop-os root[12724] INFO User authentication sucessful.
2019-12-13 17:50:30 pop-os root[12724] INFO Channel open
2019-12-13 17:50:30 pop-os root[12724] INFO Transfering Chunk
2019-12-13 17:50:30 pop-os root[12724] INFO Transfering Chunk
2019-12-13 17:50:30 pop-os root[12724] INFO File transferred. Closing transport
2019-12-13 17:50:30 pop-os root[12724] INFO The server closed the connection
```

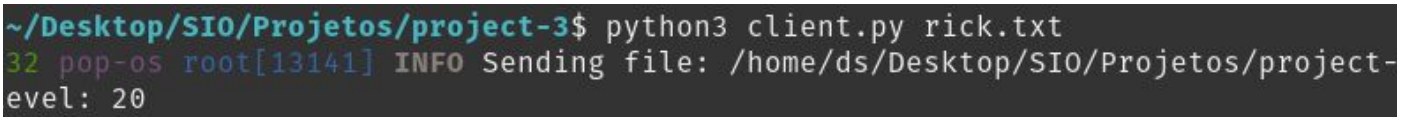
Img 4.6 - O Cliente recebe do servidor a confirmação que se autenticou com sucesso e procede com o envio do ficheiro.

Exemplo de Execução - rick.txt - Using Password

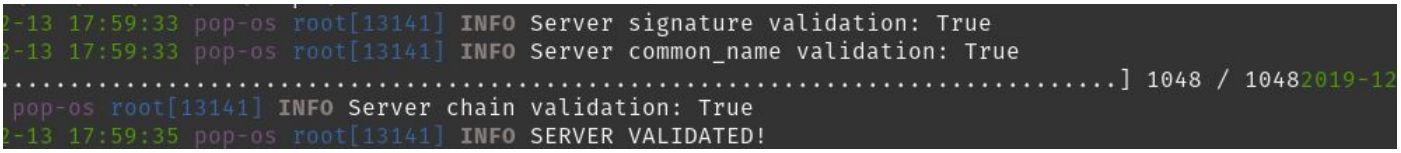
Neste exemplo vamos explicar de forma explícita todo o processo da transferência do ficheiro *rick.txt* do cliente para o servidor utilizando autenticação do Cliente com o CC:



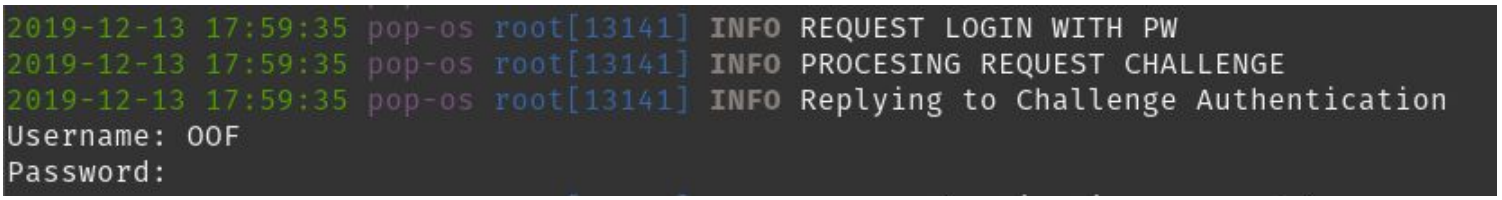
Img 4.1 - Inicialização do Servidor.



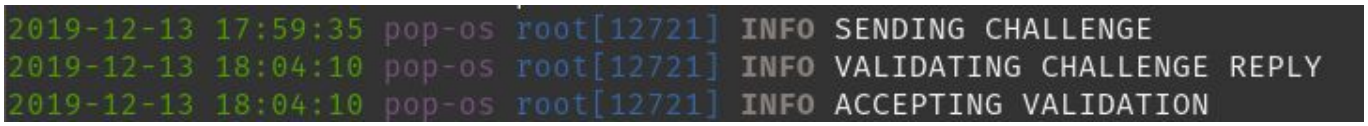
Img 4.2 - Inicialização do Cliente, especificando o ficheiro rick.txt. Como não especificamos a opção -c, por default o programa vai assumir que queremos utilizar password authentication



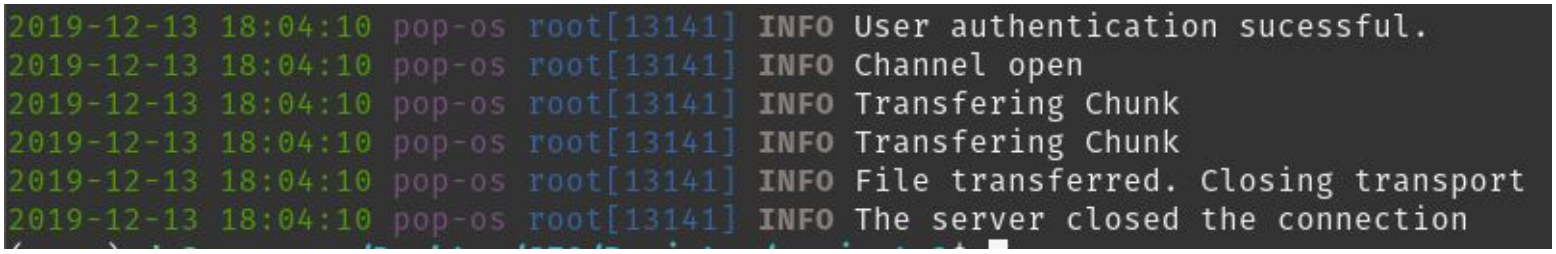
Img 4.3 - O Cliente pede ao servidor para que se autentique enviando-lhe um desafio e validando a resposta através da verificação da chain.



Img 4.4 - O Cliente pede ao servidor para que este lhe mande um desafio para se poder autenticar (o cliente) usando o CC. O servidor responde com o desafio, o cliente responde-lhe e o servidor verifica se a resposta está correta validando (ou não) o cliente.



Img 4.5 - O Servidor verifica se o Cliente é válido (i.e se a autenticação for feita com sucesso e este não for denied) e responde com uma mensagem de sucesso, dizendo ao cliente que pode iniciar a transferência do ficheiro.



Img 4.6 - O Cliente recebe do servidor a confirmação que se autenticou com sucesso e procede com o envio do ficheiro.

5. Conclusão

Chegando então ao término deste trabalho, podemos concluir que fomos capazes de cumprir todos os objetivos estabelecidos pelos docentes da disciplina, tendo aprofundado os conhecimentos relativos a autenticação de clientes usando um mecanismo de desafio-resposta, controlos de acessos, autenticação usando o cartão de cidadão e autenticação de servidores usando certificados X.509.

6. Bibliografia e Referências

Para a realização deste trabalho, foi-nos crucial as informações fornecidas pelas seguintes fontes:

- Conjunto de Slides 4, 5 e 6 da disciplina de Segurança Informática e nas Organizações
- Fichas das aulas práticas 4 e 5
- Documentação Oficial do CryptographyIO
    - https://cryptography.io/en/latest/
- Documentação Oficial da PKCS#11
    - https://python-pkcs11.readthedocs.io/en/latest/