Ricardo Antão NºMEC: 73237
Ana Ferrolho NºMEC: 88822

SIO - Segurança Informática nas Organizações

Projeto 3: Autenticação

Introdução

No âmbito da unidade curricular Segurança Informática e nas Organizações, foi elaborado um projeto com semelhanças em relação ao projeto anterior, pois ambos estão relacionados com o estabelecimento de comunicações seguras entre o cliente e o servidor. Contudo, neste trabalho, o foco do projeto residirá mais na parte da autenticação e controlo do acesso dos clientes ao servidor.

Especificamente, pretende-se planear, implementar e validar um sistema de autenticação, sendo necessário definir um protocolo que estabelece uma comunicação segura entre duas entidades, ambas autenticadas. Esse sistema deve ser capaz de transmitir um ficheiro entre o servidor/autenticador e o cliente/autenticado, com a condição de o servidor dar essas permissões ao cliente. O utilizador pode fazer autenticação com o cartão de cidadão ou através de senhas, e o servidor deve conseguir provar a sua entidade sem realizar ataques de impersonação, nomeadamente o MitM (*man in the middle*).

Este projeto pode ser dividido em várias tarefas, sendo essas a criação de protocolos e mecanismos para os seguintes sistemas:

- Autenticação tipo desafio-resposta, também conhecido por CHAP (*CHallenge-response Authentication Protocol*), recorrendo ao uso de senhas;
- Controlo de acesso;
- Autenticação de utentes através do cartão de cidadão;
- Autenticação do servidor com o auxílio de certificados X.509.

Preparação

O código-base usado neste projeto foi exatamente o mesmo usado no projeto anterior, ou seja, os familiares ficheiros client.py e server.py. Era possível continuar com o código desenvolvido no projeto anterior, mas optámos por desenvolver a partir do código-raíz que nos fora fornecido neste projeto. Por isso, a preparação exigida neste projeto é idêntica à do anterior, ou seja, volta a ser necessário criar um ambiente virtual onde se vão executar os programas e instalar um conjunto de requisitos incluídos em requirements.txt. Também foi desenvolvido um novo ficheiro, generate_file.py.

Desenho dos protocolos

1.Desenho de um protocolo (planeamento e descrição) para a autenticação de utentes através de um mecanismo de desafio resposta. Pode-se considerar a existência de uma ferramenta para aprovisionamento dos clientes do lado do servidor. Ou seja: não é necessário considerar o registo *online* dos clientes.

Um protocolo de autenticação tipo desafio-resposta dita a forma como o utente apresenta a prova de autenticação ao servidor, de modo a ser autenticado com sucesso. Existe um outro protocolo, o de tipo direto, que é mais simples de implementar e desempenha uma função idêntica à do desafio-resposta. No entanto, o protocolo desafio-resposta representa um sistema mais seguro. Isto porque, ao contrário do protocolo direto, que corresponde a uma apresentação explícita da prova, o protocolo desafio-resposta está associado a uma apresentação implícita. Numa apresentação explícita, o cliente simplesmente exibe o seu elemento de prova ao servidor. Já numa apresentação implícita, o cliente prova ao servidor que conhece ou possui o elemento de prova, mas sem o mostrar explicitamente. Ao não revelar o conteúdo num modo explícito, impede-se a suas captura por entidades externas, reduzindo as vulnerabilidades e os riscos de serem realizados ataques. Seguem-se alguns exemplos de provas de autenticação:

- Explícita/direta: Biometria (algo que o utilizador é); senha memorizada (algo que o utilizador sabe); senhas descartáveis (algo que o utilizador tem);
- Implícita/desafio-resposta: Segredos partilhados, tais como uma senha/password (algo que o utilizador sabe); dispositivos (algo que o utilizador tem); *smartcards,* nomeadamente o cartão de cidadão (algo que o utilizador tem).

Quando o cliente tenta aceder a um servidor através de um pedido de autenticação/*log attempt*, o servidor gera um desafio/*challenge*, por exemplo um *nonce* (um número que só é usado uma vez, de modo a garantir que a comunicação não é reutilizada para repetir mensagens), guarda-o e envia uma cópia ao cliente. O cliente, ao receber o desafio, recorre a um algoritmo de encriptação, como por exemplo uma função *hash* (função recursiva chamada várias vezes, cujo parâmetro inicial é, neste contexto, o desafio. Exemplo: func(func(func(desafio)))), para codificar esse desafio. O resultado obtido nessa encriptação é enviado como resposta/*response* ao servidor. De seguida, o servidor encripta também o desafio (como o servidor usa o mesmo algoritmo que o cliente, trata-se de um protocolo desafio-resposta com funções não invertíveis), o que tinha guardado inicialmente antes de enviar uma cópia ao cliente, com a sua chave privada e o mesmo algoritmo, para poder comparar o resultado obtido com a resposta do cliente. Se os valores forem iguais, o servidor autentica o cliente; senão, a autenticação simplesmente não é realizada e termina o protocolo de autenticação.

Com a exceção do algoritmo considerado no exemplo (optou-se por um mais complexo), apresenta-se a seguinte exemplificação do protocolo que foi mais tarde implementado:

1) O Bob é responsável pelo controlo de acesso, ou seja, desempenha a função de servidor.
2) A Alice, uma cliente, pretende ter acesso a esse servidor.
3) O Bob, sabendo que a Alice quer aceder ao seu servidor, envia um desafio à Alice, por exemplo, “s1a40s”.
4) A Alice, ao receber o desafio, deve devolver ao servidor a resposta correspondente, por exemplo, “u3c62u”, de modo a aceder ao servidor.
5) O algoritmo de encriptação (pode incluir o uso de uma chave privada, senha, entre outros) que gera a correspondência entre o desafio e a resposta são conhecidos tanto pela Alice como pelo Bob. No caso dos exemplos dados anteriormente, o algoritmo seria somar dois dígitos a cada dígito e “saltar” duas letras por cada letra.
6) Perante a resposta recebida e o algoritmo previamente conhecido, Bob autoriza e autentica, ou não, o acesso da Alice ao servidor.

Segue-se um diagrama, representado na figura 1, que ilustra o protocolo desenhado.

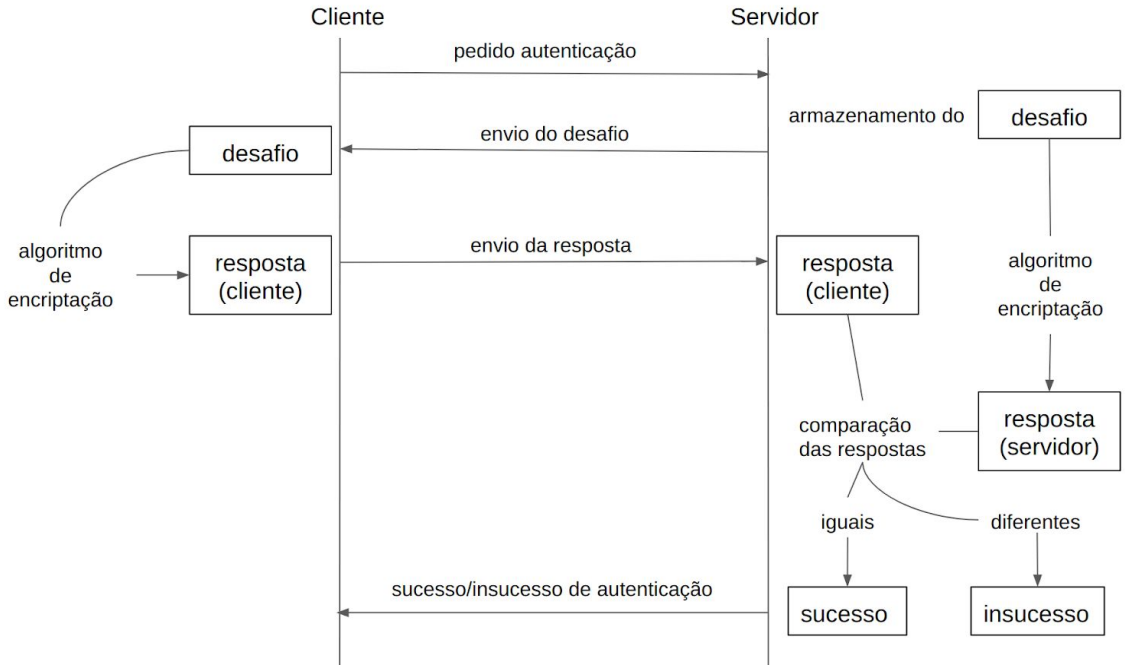

Fig.1 - Protocolo de autenticação tipo desafio-resposta com funções não invertíveis.

É de notar que existem várias formas de implementar um protocolo de autenticação do tipo desafio-resposta. Por exemplo, em vez de se guardar uma cópia do desafio no servidor, o cliente podia enviar, em conjunto com a resposta, o desafio de volta ao servidor, permitindo-lhe

realizar o mesmo procedimento de encriptação do desafio recebido e comparar o resultado com a resposta recebida, como mostra a figura 2.

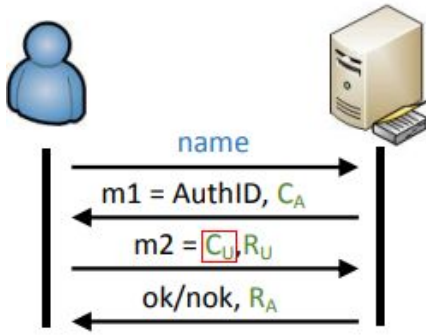

Fig. 2 - Etapa em que o cliente devolve o desafio (C_u) de volta para o servidor, que usa funções não invertíveis.

Outra alternativa seria o servidor, ao receber somente a resposta do cliente, mas conhecendo o desafio, realizar um algoritmo de decriptação para descodificar a resposta recebida (como o servidor usa um algoritmo que desempenha a função inversa do algoritmo do lado do cliente, trata-se de um protocolo desafio-resposta com funções invertíveis), e comparar o desafio obtido com o que já conhecia anteriormente, como apresenta a figura 3. A etapa na qual o servidor decripta a resposta recebida (destacado a vermelho) é a que distingue este procedimento dos que foram previamente descritos.

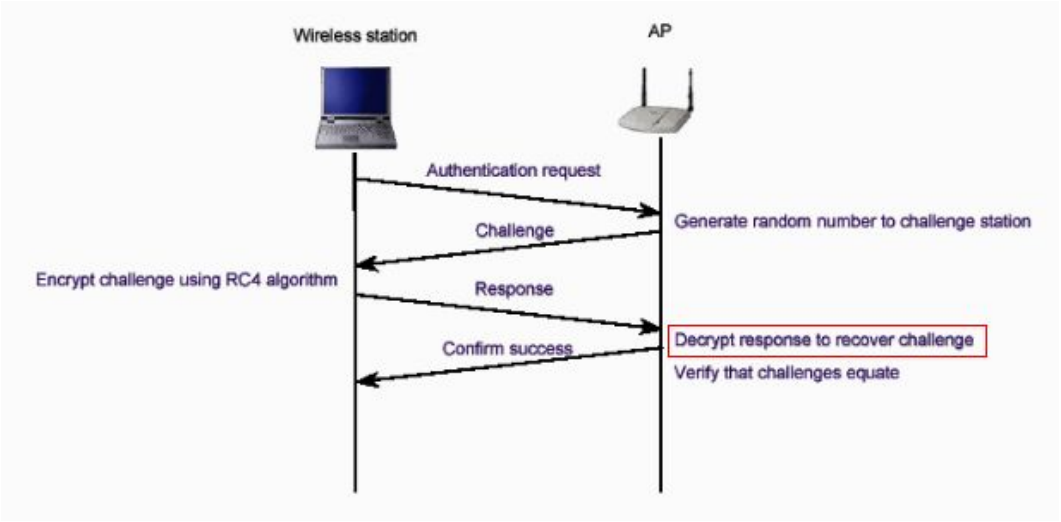

Fig. 3 - Wireless station corresponde ao cliente e AP ao servidor, que usa funções invertíveis.

Também seria possível realizar uma combinação destas duas alternativas. O servidor não guarda o desafio, mas recebe-o em conjunto com a resposta gerada pelo cliente. A seguir, o servidor recorre a um algoritmo de decriptação que é aplicado à resposta recebida. Por fim, compara o resultado obtido com o desafio diretamente adquirido através do cliente, e, conforme o resultado, autentica ou não o cliente.

Um outro fator que não foi explorado na implementação deste sistema de autenticação foi o uso de chaves secretas partilhadas nos algoritmos de encriptação e/ou desencriptação.

Para evitar interferências de atacantes, em particular a repetição de mensagens por uma terceira entidade despercebida pelo cliente e servidor (*replay attack*), decidiu-se implementar um protocolo que armazena o desafio no servidor, pois assim o cliente não precisa de enviar o desafio de volta, não se expondo assim o desafio a vulnerabilidades. Devido a questões de simplicidade, optou-se por usar um algoritmo de encriptação em ambas as entidades. Assim, basta implementar um algoritmo de encriptação, deixando de ser necessário desenvolver um outro algoritmo, de decriptação, no lado do servidor. Restou-nos, portanto, a alternativa descrita inicialmente, que consiste em armazenar o desafio no servidor e encriptá-lo para poder comparar com a resposta do cliente.

2.Desenho de um mecanismo para controlo de acesso, que permita indicar explicitamente se um utente pode ou não transferir ficheiros.

Para um mecanismo de controlo de acesso que restringe os utilizadores que podem ou não transferir ficheiros, será implementado através de um ficheiro que contém uma lista de utilizadores e as suas respectivas passwords, somente utilizadores incluídos neste ficheiro podem transferir ficheiros entre o cliente e o servidor.

3.Desenho de um protocolo (planeamento e descrição) para a autenticação de utentes através do cartão de cidadão.

Um cartão de cidadão é um tipo de *smartcard*, aparelhos eletrónicos de autenticação usados para restringir acesso a informação, no caso do cartão de cidadão este consiste num sistema de autenticação da identidade do proprietário, de tal forma que pode ser usado para autenticar ligações e assinar certificados informáticos. Possíveis funções do cartão de cidadão:

- Autenticação do dono do cartão;
- Distribuição do certificado para verificação da identidade;
- Autenticar outras pessoas com cartões semelhantes;
- Cadeia de certificação presente no cartão;
- Autenticação de clientes com certificados semelhantes;
- Pedidos com certificados “especiais” validados pelo cartão.

O utilizador começa por pedir acesso ao servidor. Depois, o servidor gera um pedido SAML (*security assertion markup language*), conhecido por ser SSO (*web browser single sign-on*), e redirecciona o utilizador a uma terceira entidade, neste caso, ao *plugin* “Autenticação.gov”, com um pedido de autenticação. Nessa aplicação, é realizada a validação desse pedido e gerada uma resposta a confirmar, ou não, a autenticação do utilizador. A seguir, essa resposta é

redirecionada até ao servidor através do cliente. Quando o servidor recebe a resposta, faz a validação, extrai os atributos de SAML e envia ao cliente uma mensagem de sucesso/insucesso de autenticação.

Este protocolo planeado apresenta semelhanças com o do desafio-resposta. Contudo, a principal diferença reside, para o caso da autenticação com *smartcards,* na existência de uma terceira identidade, e nos consequentes redirecionamentos do utilizador.

A figura 4 demonstra o procedimento descrito. *User* e *Service Provider* correspondem respectivamente ao cliente e servidor.

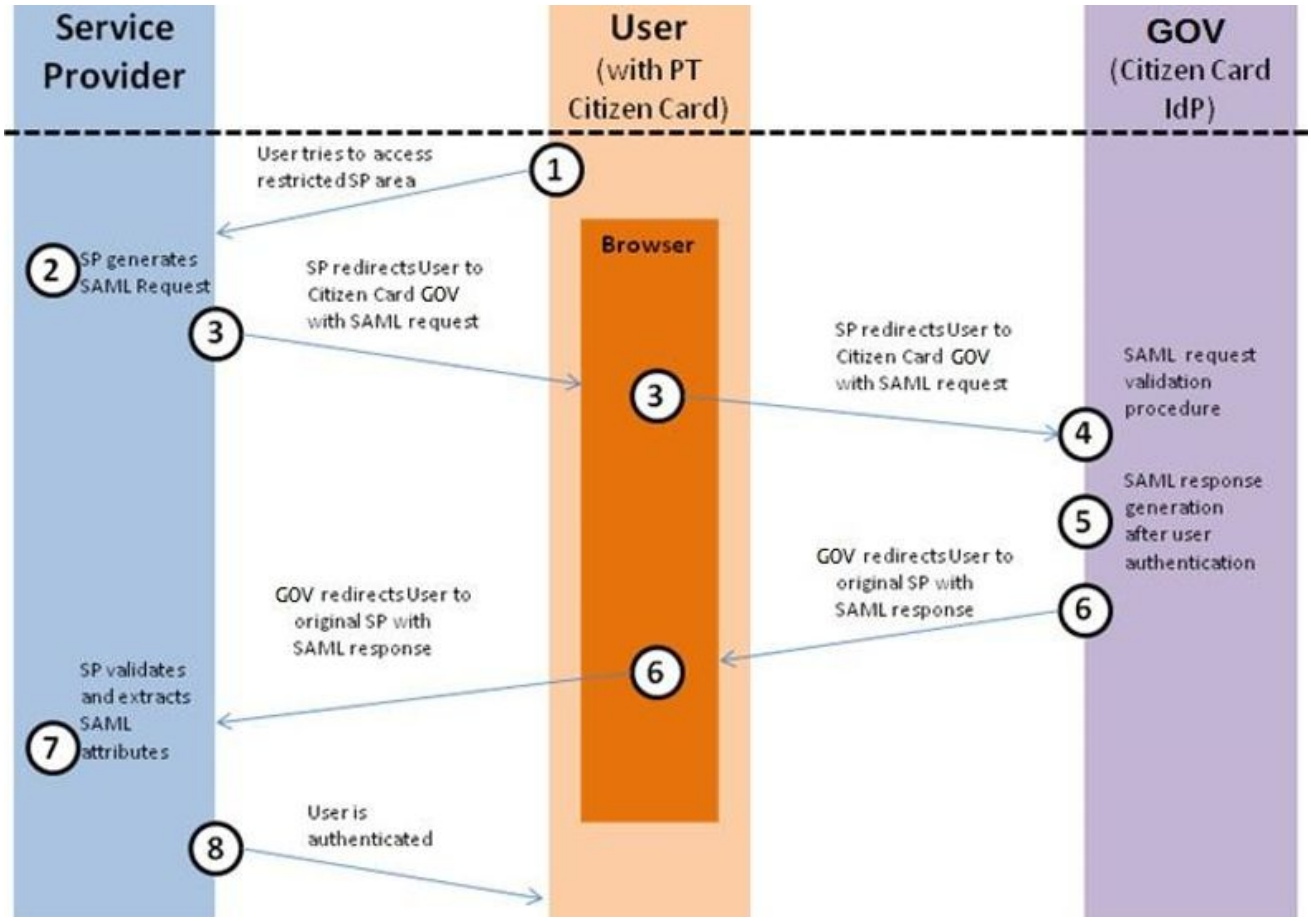

Fig 4 - Protocolo de autenticação do cliente através do cartão de cidadão.

4. Desenho de um protocolo (planeamento e descrição) para a autenticação do servidor utilizando certificados X.509.

Para compreendermos o certificado X.509, é necessário entendermos o significado de certificado digital. Um certificado digital é um arquivo que averigua se uma entidade, tipicamente um cliente ou servidor, é ela mesma. Para isso, é preciso garantir que uma chave

pública pertence a essa entidade. Resumindo, é um arquivo usado para autenticar entidades. Já o certificado X.509 define-se por um certificado digital que especifica as informações necessárias para autenticar o cliente, e o formato no qual devem ser enviadas.

Num certificado X.509 destacam-se as seguintes componentes:

- Versão: Identifica a versão do formato do certificado (a versão 3 é a mais recente);
- Número de série: número sequencial único para cada certificado emitido por uma CA (Autoridade de Certificação);
- Identificador/*signature*: algoritmo usado para assinar o certificado (por exemplo: RSA - *Rivest, Shamir, and Adelman*);
- DN (*distinguished name*): nome único da CA;
- Período de validade: data de início e data de término da validade;
- Nome do sujeito: Nome da entidade cuja chave pública foi assinada;
- Informações da chave pública: algoritmo, parâmetros e atributos da chave pública do dono do certificado;
- Assinatura Digital: Assinatura gerada usando a chave pública da CA sobre as informações acima;
- Extensões.

Este certificado é usado no protocolo SSL (*Secure Sockets Layer*). É um protocolo que assegura comunicações seguras entre o cliente e servidor através de transportes de ligação, nomeadamente TCP. É composto pelos subprotocolos *SSL Handshake*, que gera e garante sessões seguras, e *SSL Transport Protocol,* que recorre a parâmetros e algoritmos criptográficos para tornar o transporte de dados mais seguro. Este protocolo não é capaz de autenticar somente o cliente: ou autentica apenas o servidor, ou ambas as entidades, ou ainda nenhuma das entidades.

O funcionamento do protocolo desenhado inicia-se por um pedido realizado pelo cliente, com informações criptográficas, ao servidor. O servidor retribui obrigatoriamente com o seu certificado e uma *cipherSuite*. O cliente, ao receber estas informações, realiza a autenticação do servidor, ou seja, verifica se recebeu o certificado do servidor e os parâmetros criptográficos recebidos. De seguida, o cliente envia, para além da sua chave secreta, encriptada com a chave pública do servidor, o seu certificado ao servidor. Posteriormente, o servidor averigua o certificado do cliente, realizando assim a autenticação do cliente. A seguir, o cliente e o servidor trocam informação entre si no intuito de saberem se foram ou não autenticados com sucesso. O protocolo termina com a troca de mensagens encriptadas com a mesma chave secreta partilhada.

A figura 5 ilustra o protocolo de autenticação com uso de certificados X.509, acabado de ser descrito.

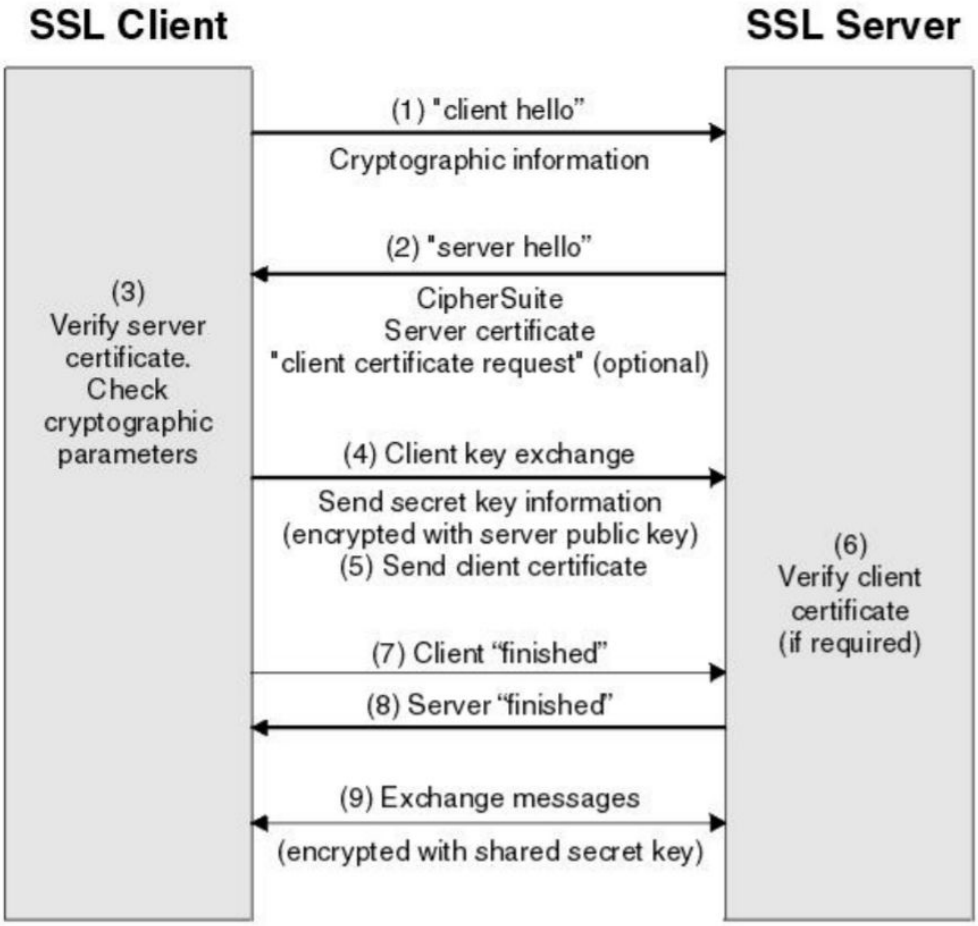

Fig 5 - Protocolo de autenticação com uso de certificados X.509.

Implementação dos mecanismos

Para além do código previamente adquirido como base no projeto anterior, foram desenvolvidos os seguintes atributos e métodos, para o cliente e servidor:

Lado cliente
- Atributos: username, password, private_key;
- Métodos: challenge_answer(message), cc_function(message), certificate_function(message).

Lado servidor
- Atributos: username, password, private_key;
- Métodos: processUser(message), processChallenge(message), cc_function(), process_cc_authentication(message), create_certificate(), process_certificate(message).

Foram ainda importadas as seguintes livrarias nos ficheiros client.py e server.py:
- random;
- cryptography:
  - x509;
  - x509.oid.NameOID;
  - hazmat.primitives.hashes.
- PyKCS11.

Para o generate_file.py foram importadas as livrarias string e random.

5. Implementação do protocolo para autenticação de utentes através da apresentação de senhas.

Um exemplo de implementação de um protocolo tipo desafio-resposta, como já foi visto anteriormente, na tarefa 1, consiste em recorrer a uma senha ou password conhecida pelo utilizador. Portanto, nesta tarefa implementou-se o protocolo desafio-resposta desenhado na primeira tarefa, aplicando-o ao exemplo de autenticação através de uma senha/password.

Ignorando por uns momentos o protocolo de desafio-resposta, um registo com a password do utilizador funciona do seguinte modo:
1) Cliente envia nome de utilizador ao servidor.
2) Servidor averigua se esse nome está registado.
3) Servidor pede password ao cliente.
4) Cliente envia password ao servidor.
5) Servidor confirma se a password corresponde à do utilizador.
6) Servidor autentica o utilizador com sucesso ou não, conforme a password dada pelo cliente.

Segue-se o código desenvolvido para esta tarefa, tendo por isso em consideração o protocolo desafio-resposta.

```
##Challenge-Answer
self.username = raw_input("Enter Username:")
self.password = raw_input("Enter Password")
self._send({'type': 'USER', 'username': self.username})
```

Fig 6 - Cliente, introdução do username e password e envio do primeiro.

```
############################################################
#Function to give the password as answer to the challenge from the server
def challenge_answer(self, message):
    self._send({'type': 'CHALLENGE', 'answer': self.password})
```

Fig 7 - Cliente, função para enviar a password como resposta ao pedido do Servidor.

```
############################################################
#Sending Password Challenge
def processUser(self, message: str):
    lista_ficheiro =open('users.txt', 'r')
    users = [u for u in lista_ficheiro if u%2 == 0]
    passwords = [p for p in lista_ficheiro if p%2 == 0]
    if message['username'] in users:
        self.user = message['username']
        self.password = passwords[ ((users.index(message['username'])+1))+1]
        self._send({'type': 'CHALLENGE', 'challenge': 'PASSWORD'})
        return True
    else:
        print("Failed to process User/Password")
        return False
```

Fig.8 - Servidor, função que confirma se o username existe e se sim envia o pedido da password.

```
############################################################
#Processing message Challenge from client, if equal OK if not close the communication
def processChallenge(self,message: str) -> bool:
    if message['answer'] == self.password:
        return True
    print("Failed Challenge-Answer")
    return False
```

Fig.9 - Servidor, função para verificar se a password corresponde à password do username introduzido.

6. Implementação do mecanismo para controlo de acesso.

Para o mecanismo de controlo de acesso implementou-se a criação de um ficheiro “users.txt” que contém os *usernames* e as respectivas passwords, somente aos *usernames* registados no ficheiro são permitidos a troca de ficheiros entre o cliente e o servidor.

Na figura 10 ilustra-se o código desenvolvido, num ficheiro aparte, generate_file.py, para implementar o mecanismo.

A função principal, como se pode deduzir, é generate_file(). Nesta função, começa-se por abrir/criar o ficheiro users.txt no modo escrita. Realiza-se de seguida uma iteração tipo for entre uma a 1000 vezes, conforme for o valor gerado pela função importada random. Por cada iteração é escrito no ficheiro o nome do utilizador gerado pela função user() e, na linha seguinte do ficheiro, a password, gerada pela função password(). Resumindo, são introduzidas informações de um utilizador em cada iteração. Após as iterações, executa-se a boa prática de fechar o ficheiro.

Na função user(), define-se uma variável, *letters_digits*, com uma string que inclui todas as letras do alfabeto maiúsculas e minúsculas, *string.ascii_letters*, e dígitos, *string.digits*. É devolvida uma string que contém entre três a dez caracteres, sendo os caracteres escolhidos aleatoriamente a partir de *letters_digits*.

A função password() é comparável com a anterior, exceto que os caracteres contidos na string devolvida são escolhidos aleatoriamente não só a partir de letras e números, como também de caracteres especiais. Além disso, a string devolvida tem comprimento entre cinco a vinte caracteres.

```
#####################################
# Criar users e suas passwords
def user(self):
    letters_digits = string.ascii_letters + string.digits  # nao inclui " "
    return ''.join(random.choice(letters_digits) for i in range(int(random.random() * 7 + 3)))

def password(self):
    password_characters = string.ascii_letters + string.digits + string.punctuation  # tmb nao inclui " "
    return ''.join(random.choice(password_characters) for i in range(int(random.random() * 15 + 5)))

def generate_file(self):
    file = open('users.txt', 'w')
    for data in range(int(random.random() * 999 + 1)):
        file.write(str(user()) + '\n')
        file.write(str(password()) + '\n')
    file.close()
```

Fig.10 - Funções para gerar random usernames e passwords e colocá-los no ficheiro “users.txt” para controlo de acesso

Para aceder aos dados do ficheiro gerado, converte-se cada linha do ficheiro num elemento de uma lista. Os nomes dos utilizadores correspondem aos elementos de índice par e as passwords aos índices ímpares. Para aceder à password respetiva de uma certo utilizador, ”anda-se” um índice para a frente.

7.Implementação do protocolo para autenticação de utentes através do cartão de cidadão.

Para a autenticação pelo cartão de cidadão foi implementado o seguinte código.
A figura 11 descreve a criação do certificado para autenticação por cartão de cidadão do lado do servidor. Ao gerar a chave privada e o certificado, seguida da assinatura do certificado com a chave privada para se enviar para o cliente.

```
#####################################################
#CC Authentication
def cc_function(self):
    self.private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend())
    csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
        # Provide various details about who we are.
        x509.NameAttribute(NameOID.COUNTRY_NAME, u"PT"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"AVEIRO"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, u"AVEIRO"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"UA"),
        x509.NameAttribute(NameOID.COMMON_NAME, u"Server"),
    ])).add_extension(
        x509.SubjectAlternativeName([
            # Describe what sites we want this certificate for.
            x509.DNSName(u"mysite.com"),
            x509.DNSName(u"www.mysite.com"),
            x509.DNSName(u"subdomain.mysite.com"),
        ]),
        critical=False,
        # Sign the CSR with our private key.
    ).sign(self.private_key, hashes.SHA256(), default_backend())

    self._send({'type': 'CC', 'certificate': csr})
```

Fig. 11 - Função usada para implementar o mecanismo de autenticação com o auxílio do cartão de cidadão, desenvolvida no lado do servidor

A figura 12 demonstra o código do cliente para autenticar o certificado recebido do servidor. Primeiro criando a sua chave privada com a livraria PyKCS11 e usando a mesma para assinar o certificado e reenviar para o servidor.

```
#####################################################
#Function to respond to CC authentication
def cc_function(self, message):
    session = pkcs11.openSession(slot)
    self.private_key = session.findObjects(
        [(PyKCS11.CKA_CLASS, PyKCS11.CKO_PRIVATE_KEY), (PyKCS11.CKA_LABEL, 'CITIZEN AUTHENTICATION KEY')])[0]
    mechanism = PyKCS11.Mechanism(PyKCS11.CKM_SHA1_RSA_PKCS, None)
    text = message['certificate']
    signature = bytes(session.sign(self.private_key, text, mechanism))
    self._send({'type':'CC', 'authentication': signature})
```

Fig. 12 - Função usada pelo cliente para responder ao pedido de autenticação com o cartão de cidadão enviado pelo servidor

8.Implementação do protocolo para autenticação do servidor através de certificados X.509.

Para a autenticação de certificados X.509 foi criado o seguinte código ilustrado na figura 13.
O código é semelhante ao apresentado na figura 11 pois segue a mesma lógica no servidor. Gera uma chave privada e um certificado, este certificado é então assinado com o uso da chave e enviado para o cliente para que este possa fazer autenticação do certificado.

```
#########################################################
#Create and send certificate to be authenticated by client
def create_certificate(self):
    self.private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend())
    csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
        # Provide various details about who we are.
            x509.NameAttribute(NameOID.COUNTRY_NAME, u"PT"),
            x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"AVEIRO"),
            x509.NameAttribute(NameOID.LOCALITY_NAME, u"AVEIRO"),
            x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"UA"),
            x509.NameAttribute(NameOID.COMMON_NAME, u"Server"),
        ])).add_extension(
    x509.SubjectAlternativeName([
        # Describe what sites we want this certificate for.
            x509.DNSName(u"mysite.com"),
            x509.DNSName(u"www.mysite.com"),
            x509.DNSName(u"subdomain.mysite.com"),
        ]),
    critical = False,
            # Sign the CSR with our private key.
            ).sign(self.private_key, hashes.SHA256(), default_backend())

    self.send({'type': 'CERTIFICATE', 'certificate': csr})
```

Fig. 13 - Função usada pelo servidor para gerar um certificado x509 para autenticação

Bibliografia e Ferramentas

- Noções gerais de autenticação
  https://en.wikipedia.org/wiki/Authentication

- Challenge–response authentication
  https://en.wikipedia.org/wiki/Challenge%E2%80%93response_authentication
  https://searchsecurity.techtarget.com/definition/challenge-response-system

- *Nonce*
  https://en.wikipedia.org/wiki/Cryptographic_nonce

- *Replay attack*
  https://en.wikipedia.org/wiki/Replay_attack

- Certificados x.509
  https://en.wikipedia.org/wiki/X.509
  https://www.ssl.com/faqs/what-is-an-x-509-certificate/
  https://www.ibm.com/support/knowledgecenter/pt-br/SSFKSJ_8.0.0/com.ibm.mq.sec.doc/q009830_.htm
  https://www.programcreek.com/python/example/102811/cryptography.x509.Certificate
  https://cryptography.io/en/latest/x509/tutorial/

- Segurança em Redes Informáticas, A. Zúquete: Secs. 5.3, 5.4.1, 8.6.3, 8.9.2, Cap. 10

- Biblioteca Cryptography.io

- PyKCS11