

Projeto 2

Universidade de Aveiro

Ana Miguel Monteiro, Florentino Sánchez,
Gonçalo Aguiar, Tiago Bastos



universidade de aveiro
theoria poiesis praxis

Projeto 2

Departamento de Electrónica, Telecomunicações e
Informática (DETI)
Universidade de Aveiro

Ana Miguel Monteiro, Florentino Sánchez,
Gonçalo Aguiar, Tiago Bastos
(98314) anamiguel24@ua.pt, (98181) florentino@ua.pt
(97590) tiagovilar07@ua.pt, (98266) gonc.soares12@ua.pt

2 de janeiro de 2022

Conteúdo

1	Introdução	2
2	Alterações feitas no projeto anterior	3
3	User authentication application (UAP)	5
4	Encriptação	6
4.0.1	Encriptação das contas do autenticador	6
4.0.2	Encriptação das contas incluídas em cada conta do autenticador	6
4.0.3	Geração da Key	9
5	Autenticação	11
5.0.1	Processo de autenticação	11
5.0.2	Protocolo	11
5.0.3	Esquema do protocolo implementado	16
6	Conclusão	18

Capítulo 1

Introdução

Neste projeto utilizámos o website simples desenvolvido no projeto anterior para desenvolver este novo trabalho. Desenvolvemos uma web app clara e específica a partir do '0' com o propósito de ser uma aplicação web de autenticação.

O objetivo do trabalho é a implementação de protocolos de autenticação robustos. Para tal, o nosso grupo desenvolveu um novo protocolo baseado no protocolo MS-CHAP-V2 e uma aplicação Web relacionada para explorá-lo.

Depois desta introdução, segue-se as explicações do desenvolvimento da web app de autenticação e das técnicas de encriptação e autenticação utilizadas, com informações relevantes e demonstrações de como essas técnicas são exploradas e o seu impacto.

No Capítulo 2, Capítulo 3, Capítulo 4 e Capítulo 5 é feita a abordagem às alterações feitas no projeto anterior, à aplicação User authentication application (UAP), às técnicas de encriptação e autenticação, respetivamente.

Depois de explicitado o desenvolvimento de cada parte segue-se no Capítulo 6 as conclusões gerais do trabalho.

Capítulo 2

Alterações feitas no projeto anterior

Fornecemos uma versão da nossa aplicação Web desenvolvida no projeto anterior, mas agora explorando o nosso protocolo baseado no protocolo MS-CHAP-V2, de forma a comunicar com a aplicação de autenticação sem ter que enviar segredos partilhados (passwords).

Acrescentamos também botões na página inicial da loja (projeto anterior), que redirecionam para o autenticador, de forma a que seja possível o utilizador autenticar-se no site, iniciar sessão ou criar conta no autenticador.

Foi acrescentada também uma página cujo objetivo é pedir as credenciais do autenticador e comunicar, através de POST requests, com o servidor do Autenticador, e sucessivamente, fazer a autenticação ou não.

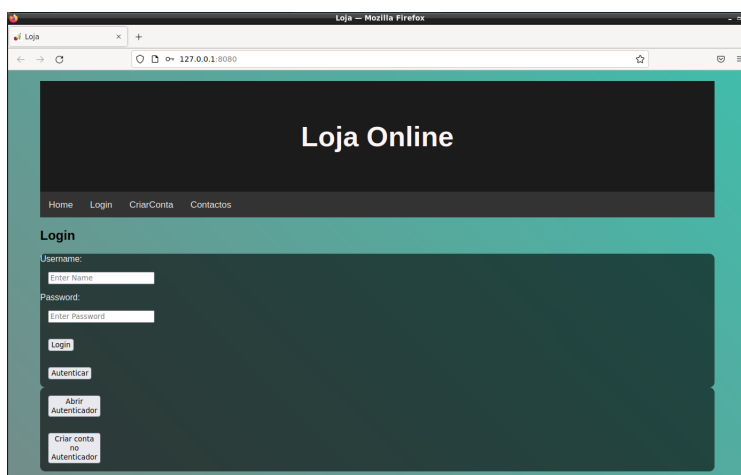


Figura 2.1: Página da loja

Alteramos também código no ficheiro 'cherry.py' de modo a que seja possível a comunicação com o servidor da UAP e para que seja feita a autenticação. Essas alterações são explicitadas no Capítulo 5.

Capítulo 3

User authentication application (UAP)

A aplicação da loja (trabalho anterior), chamará a aplicação de autenticação de usuário, que expõe uma API REST, recebe solicitações HTTP de um navegador (local) para realizar uma autenticação com um determinado servidor (nome DNS) e executa um protocolo de autenticação com esse servidor. Após uma conclusão bem-sucedida, o controle deve retornar ao navegador e ao serviço que exigiu a autenticação.

A nossa UAP tem as seguintes funcionalidades:

- Usa redirecionamentos HTTP por meio do navegador.
- Usa o navegador do usuário como interface.
- Permite que o usuário armazene vários pares de nome de usuário e senha para cada nome DNS de serviço.
- Informa o usuário sobre o protocolo de autenticação que será executado em seu nome.
- Em relação às execuções de E-CHAP, informa ao usuário sobre um erro numa resposta do serviço, incluindo a fase do protocolo em que ocorreu. No entanto, o E-CHAP não deve ser abortado por causa desse erro.
- Deve ser iniciado de forma autônoma, pois não faz parte do navegador. Ao iniciar, ele deve solicitar uma senha que será usada para criptografar o banco de dados de credenciais do usuário (pares de nome de usuário-senha)
- Todos os dados armazenados são criptografados. Por motivos de privacidade, os nomes de serviço também são criptografados e nomes de usuário ou senhas iguais não são detectados. Finalmente, todos os valores criptografados incluem algum tipo de 'sal' aleatório, para evitar o vazamento de informações de seu tamanho.

Capítulo 4

Encriptação

4.0.1 Encriptação das contas do autenticador

Com o objetivo de tornar a uap mais segura fizemos a encriptação de todos os pares de username-password de cada conta do autenticador aquando da criação/adição da mesma.

Em relação à encriptação do username utilizámos a HASH "SHA512", que sucessivamente irá ser guardada na base de dados, uma vez que é um processo irreversível e não irá ser necessário obter o username original a partir da base de dados. Caso queirámos ir buscar dados associados a um determinado username dado, basta fazer a mesma HASH, e comparármos o valor obtido com os existentes na base de dados (Isto foi feito por ex. na parte da autenticação/comunicação(cliente/servidor)).

Em relação à encriptação da password a cifra que usámos para o efeito foi a "AES-128" em que utilizámos para gerar a key uma função denominada "generate_key" que tem como argumentos, a password do utilizador atual do autenticador, um salt com um valor aleatório de 16 bytes e o algoritmo "AES-128".

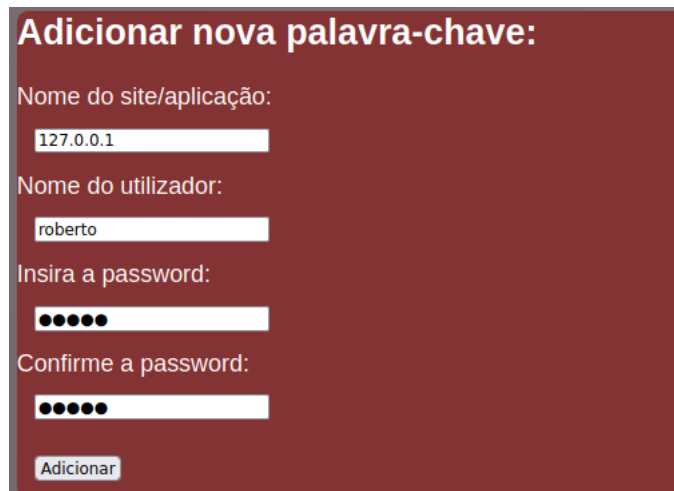
A descrição relativa à função "generate_key" está descrita no subtópico 4.0.3 e a imagem da função "encrypt", que é usada para encriptação dado uns dados (neste caso password), uma key, um algoritmo ("AES-128") e um iv (valor aleatório de 16 bytes) está descrita na figura 4.2 mais abaixo.

4.0.2 Encriptação das contas incluídas em cada conta do autenticador

Fizemos também a encriptação de todos os pares de username-password que cada conta que o autenticador armazena.

A cifra que usámos para o efeito foi a "AES-128" em que utilizámos para gerar a key uma função denominada "generate_key" que tem como argumentos, a password do utilizador atual do autenticador, um salt com um valor aleatório de 16 bytes e o algoritmo "AES-128".

Ou seja, sempre que um utilizador da uap adiciona um par de username-password referente a um dado dns(como se vê na figura em baixo) é feita a encriptação do username e da password através da seguinte função encrypt:



The image shows a web form titled "Adicionar nova palavra-chave:" on a dark red background. It contains four input fields: "Nome do site/aplicação:" with the value "127.0.0.1", "Nome do utilizador:" with the value "roberto", "Insira a password:" with five black dots, and "Confirme a password:" with five black dots. At the bottom is a button labeled "Adicionar".

Figura 4.1: Adição de um par de username-password

```
def encrypt(data, key, algorithm, iv):
    if algorithm == "AES-128":
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    elif algorithm == "ChaCha20":
        cipher = Cipher(algorithms.ChaCha20(key, iv), mode=None)

    encryptor = cipher.encryptor()

    if algorithm == "AES-128":
        padder = padding.PKCS7(128).padder()
        padded_data = padder.update(data) + padder.finalize()
        encrypted_data = encryptor.update(padded_data) + encryptor.finalize()
    elif algorithm == "ChaCha20":
        encrypted_data = encryptor.update(data) + encryptor.finalize()

    return encrypted_data
```

Figura 4.2: Função de encriptação

Depois de ser feita a adição do par de username-password (neste caso roberto-ola12) podemos ver o resultado da encriptação na imagem que se segue:



Figura 4.3: Pares de username-password encriptados

A descriptação dos pares de username-password é feita apenas quando o utilizador da uap carrega no botão "desencriptar contas e senhas dos domínios" e quando é feita a autenticação de um dado site sendo para o efeito utilizada a seguinte função decrypt:

```
def decrypt(data, key, algorithm_name, iv):
    if algorithm_name == "AES-128":
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    elif algorithm_name == "ChaCha20":
        cipher = Cipher(algorithms.ChaCha20(key, iv), mode=None)

    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(data) + decryptor.finalize()
    return decrypted_data
```

Figura 4.4: Função de descriptação

De forma a tornar a uap um pouco mais segura quando o utilizador da uap carrega em "desencriptar contas e senhas dos domínios" é pedida a senha da uap mais uma vez servindo a mesma para gerar outra vez a key. Se o utilizador colocar a senha correta são mostrados os pares de username-password de acordo com a imagem seguinte:

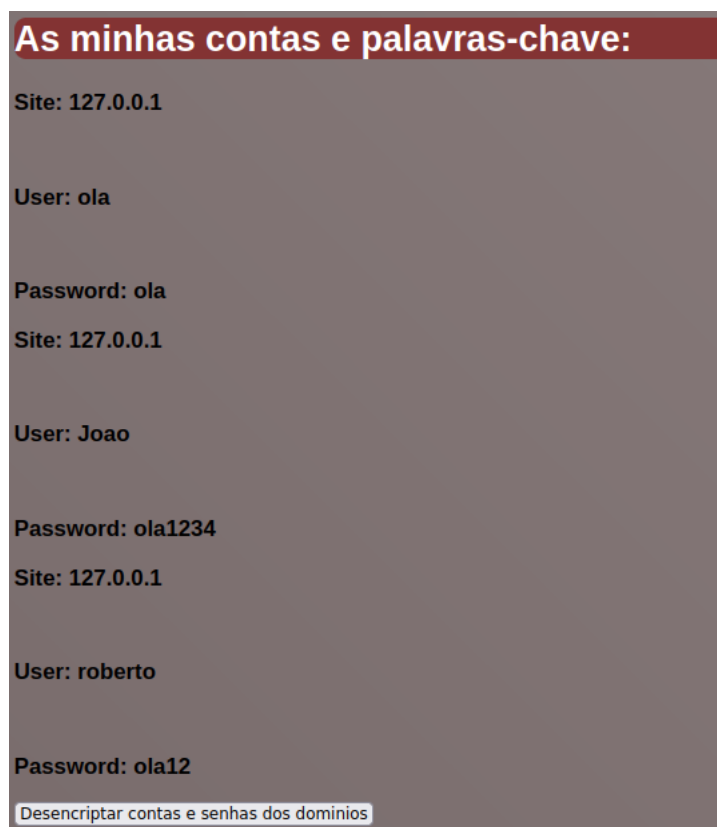


Figura 4.5: Pares de username-password desencriptados

4.0.3 Geração da Key

Em relação às gerações das keys que foram utilizadas na parte da encriptação e sucessivamente na desencriptação (quando necessária), utilizámos uma função denominada "generate_key", que tem como argumentos uma "password", um "algoritmo", e um "salt".

A password utilizada para esta função, tanto na encriptação das contas do autenticador como das contas que estão incluídas dentro de cada conta do mesmo, é a password do utilizador atual do autenticador. O salt é um valor aleatório de 16 bytes, gerado no código que irá usar esta função, e como algoritmo

usámos o "AES-128" sempre que precisámos de gerar uma key.

Dentro da função usamos uma "Key derivation function" denominada por "PBKDF2" que tem como argumentos um algoritmo "hashes.SHA256()", um length(32), o salt dado e um número de iterações(100000).

Segue-se a seguir uma imagem com a função que foi descrita e respetivo código:

```
def generate_key(password, algorithm, salt):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000
    )

    key = kdf.derive(password.encode())

    if algorithm == "AES-128":
        key = key[:16]
    elif algorithm == "ChaCha20":
        key = key[:64]

    return key
```

Figura 4.6: Generate Key

Capítulo 5

Autenticação

5.0.1 Processo de autenticação

O processo de autenticação implementado foi baseado no MS-CHAP-V2, onde são enviados desafios de 16 bytes por parte do servidor e do cliente.

Neste processo são gerados dois digests (tanto do lado do servidor como do cliente) usando a função `pdkdf2` com auxílio à função criptográfica SHA-256. Estes códigos são gerados com a palavra-passe do utilizador no autenticador, com 10000 iterações e num tamanho de 16 bytes.

O sal usado nestes digests (`h` e `h2`), é criado da seguinte forma:

- $\text{salth} = \text{ServerChallenge} + \text{ClientChallenge}$
- $\text{salth2} = \text{ClientChallenge} + \text{ServerChallenge}$

Com o intuito de evitar ataques por dicionários, foi usada a versão melhorada do CHAP, o Enhanced Challenge-Handshake Authentication Protocol (E-CHAP). Nesta versão, informação delicada como os digests (que incluem passwords) é enviada bit a bit e o processo não termina se algum dos lados detetar alguma incoerência. Em vez disso, é enviado um bit aleatório, dando continuidade ao processo até ao fim.

5.0.2 Protocolo

A autenticação é constituída pelos seguintes passos:

1. O cliente envia o username para o servidor.

```
@cherry.py.expose
def entrar(self,user,password):
    global current_auth_user
    global current_auth_pass
    global pos
    pos=0
    current_auth_user = user
    current_auth_pass = password

    payload = {"user": user}
    url = "http://127.0.0.1:10010/receive"
    r = requests.post(url, data=payload)

    return open('index.html','r').read()
```

Figura 5.1: Função que envia o username para o servidor e dá início ao processo de autenticação

2. O servidor gera o Server Challenge (SC) de 16 bytes e envia para o cliente.

```
@cherry.py.expose
def receive2(self,user):

    global challenge
    global current_username
    global current_password
    global current_dns
    current_username = user

    username_hash = get_hash(user.encode("utf-8"), "SHA512")
    current_dns = cherry.py.request.headers['Remote-Addr']

    with sqlite3.connect(DB_STRING) as conn:
        command = conn.execute("SELECT * FROM users WHERE user=?", [username_hash])
        data = command.fetchall()
        x = popup()
        if data!=[]:
            key = generate_key(aux, "AES-128",data[0][5])
            password2 = data[0][2]
            iv = data[0][4]
            decrypted_password = decrypt(password2, key, "AES-128", iv)
            unpadded_password = unpadder(decrypted_password, "AES-128")
            current_password = unpadded_password.decode("utf-8")
        else:
            print("ERRO NA AUTENTICAÇÃO")
            return

    challenge = generate_challenge()
    payload = {"Serverchallenge": challenge}
    url = "http://127.0.0.1:8080/response2"
    r = requests.post(url, data=payload)
    return
```

Figura 5.2: Função que recebe o username, gera e envia o challenge para o cliente e guarda a direção da página web que fez o pedido de autenticação

```
def generate_challenge():
    random_s = ''.join(random.choices(string.ascii_lowercase +
                                     string.ascii_uppercase +
                                     string.digits,
                                     k = 16)) #gera uma string aleatória com 30 caracteres
    return random_s
```

Figura 5.3: Função que gera o challenge

Com o objetivo de descriptar o username e a password guardados nas bases de dados, foi utilizado um pop-up, a correr do lado do servidor, para obter a palavra-passe do utilizador que irá ser usada para gerar a respetiva key.

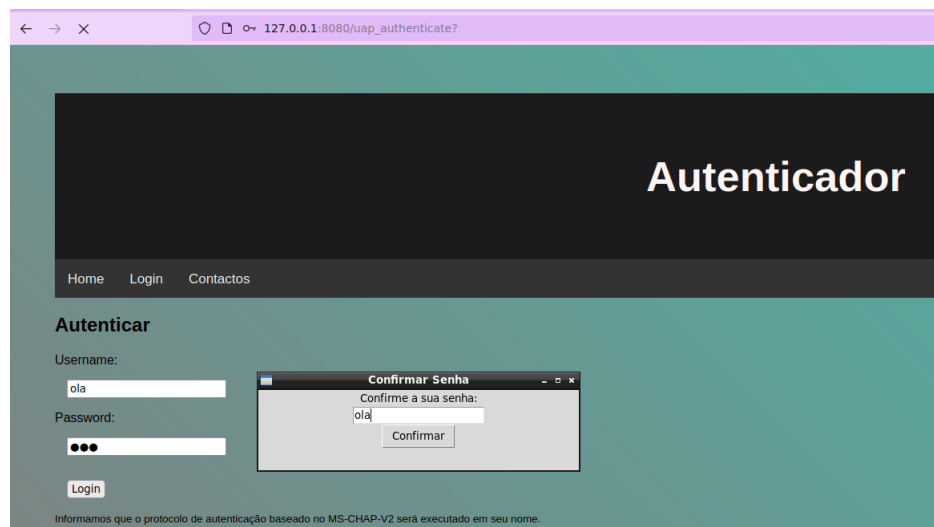


Figura 5.4: Pop-up criado com a biblioteca tkinter do Python

3. O cliente recebe o SC, gera o Client Challenge (CC), gera os dois digests h e h2 e envia o CC para o servidor.

```
@cherry.py.expose
def response2(self, Serverchallenge):

    global h_bin, h2_bin
    Clientchallenge= ''.join(random.choices(STRING.ascii_lowercase +
                                            STRING.ascii_uppercase +
                                            STRING.digits,
                                            k = 16))

    h = pbkdf2_hmac("sha256",
                    current_auth_pass.encode('utf-8'),
                    (Serverchallenge + Clientchallenge).encode('utf-8'),
                    100000,
                    16)

    h2 = pbkdf2_hmac("sha256",
                     current_auth_pass.encode('utf-8'),
                     (Clientchallenge + Serverchallenge).encode('utf-8'),
                     100000,
                     16)

    h_bin = bin(int.from_bytes(h, byteorder=sys.byteorder))
    h2_bin = bin(int.from_bytes(h2, byteorder=sys.byteorder))
    h_bin=h_bin[2:]
    h2_bin=h2_bin[2:]

    payload = {"Clientchallenge": Clientchallenge}
    url = "http://127.0.0.1:18010/authenticate2"
    r = requests.post(url, data=payload)
    return
```

Figura 5.5: Função que recebe o SC, gera o CC, gera os digests e envia o CC para o servidor

4. O servidor recebe o CC, gera os dois digests h e h2 e envia começa a enviar os bits do h2, um de cada vez, para o cliente

```
@cherry.py.expose
def authenticate2(self, Clientchallenge):
    global h_bin
    h = pbkdf2_hmac("sha256",
                    current_password.encode('utf-8'),
                    (challenge + Clientchallenge).encode('utf-8'),
                    100000,
                    16)
    h2 = pbkdf2_hmac("sha256",
                     current_password.encode('utf-8'),
                     (Clientchallenge + challenge).encode('utf-8'),
                     100000,
                     16)
    h2_bin = bin(int.from_bytes(h2, byteorder=sys.byteorder))
    h2_bin=h2_bin[2:]
    h_bin = bin(int.from_bytes(h, byteorder=sys.byteorder))
    h_bin=h_bin[2:]
    for bit in h2_bin:
        if flag == 1:
            payload = {"b": str(bit)}
            url = "http://127.0.0.1:18010/validate"
            r = requests.post(url, data=payload)
        else:
            b = random.randint(0, 1)
            payload = {"b": str(b)}
            url = "http://127.0.0.1:18010/validate"
            r = requests.post(url, data=payload)
        if flag == 2:
            print("AUTENTICADO")
            with sqlite3.connect(current_username + ".db") as conn:
                r = conn.execute("SELECT * FROM accounts WHERE dbuser=?", [current_dns])
                s = r.fetchall()
                if s[0]:
                    user = s[0][1]
                    password = s[0][2]
                    iv = s[0][4]
                    salt = s[0][5]
                    payload = ("{" + b64encode(user).decode('utf-8') + "," + b64encode(password).decode('utf-8') + "," + b64encode(iv).decode('utf-8') + "," + b64encode(salt).decode('utf-8') + "}")
                    url = "http://127.0.0.1:18010/pets"
                    r = requests.post(url, data=payload)
            else:
                print("ERRO NA AUTENTICACAO")
            return
```

Figura 5.6: Função que recebe o CC, gera os digests, valida o bit enviado pelo cliente e eventualmente envia as credenciais da conta para o cliente

5. O servidor envia o primeiro bit do h2 para o cliente
6. O cliente recebe o bitN do servidor e compara com o bitN do h2 calculado. Se forem iguais e o flag for igual a 1, envia o bitN do h para o servidor.

Se não, envia um bit aleatório e atribui o valor '0' à flag, sinalizando que o processo de autenticação falhou.

```
@cherry.py.expose
def validate(self,b):
    global pos
    global h2_bin
    global flag
    bit = bin(int(b))
    if bit[2] == h2_bin[pos] and flag==1:
        print("OK client")
        b = h_bin[pos]
        payload = {"b": str(b)}
        url = "http://127.0.0.1:10010/validate"
        r = requests.post(url, data=payload)
    else:
        print("NOT OK")
        b = random.randint(0, 1)
        payload = {"b": str(b)}
        url = "http://127.0.0.1:10010/validate"
        r = requests.post(url, data=payload)
        flag=0
    pos = pos +1
    return
```

Figura 5.7: Função que valida o bit enviado pelo servidor

7. O servidor recebe o bitN do cliente e compara com o bitN do h calculado. Se forem iguais e o flag for igual a 1, envia o bitN do h2 para o servidor. Se não, envia um bit aleatório e atribui o valor '0' à flag, sinalizando que o processo de autenticação falhou.
8. 4 e 5 ocorrem até serem enviados os N bits.
9. O servidor verifica se a flag é ainda igual a '1', se for, envia para o cliente as credenciais (codificadas em b64) para aceder ao site que fez o pedido de autenticação.

10. O cliente verifica se a flag é igual a '1', se for, descripta as credenciais recebidas pelo servidor e faz login.

```
@cherryypy.expose
def getc(self,c1,c2,iv,salt):
    global check
    global current_username
    global current_password

    print(b64decode(c1.encode('utf-8')))
    print(b64decode(c2.encode('utf-8')))
    print(b64decode(iv.encode('utf-8')))
    print(b64decode(salt.encode('utf-8')))

    key = generate_key(current_auth_pass, "AES-128", b64decode(salt.encode('utf-8')))

    decrypted_user = decrypt(b64decode(c1.encode('utf-8')), key, "AES-128", b64decode(iv.encode('utf-8')))
    unpadded_user = unpadder(decrypted_user, "AES-128")

    decrypted_senha = decrypt(b64decode(c2.encode('utf-8')), key, "AES-128", b64decode(iv.encode('utf-8')))
    unpadded_senha = unpadder(decrypted_senha, "AES-128")

    current_username = unpadded_user.decode('utf-8')
    current_password = unpadded_senha.decode('utf-8')
    return
```

Figura 5.8: Função que descripta e guarda as credenciais enviadas pelo servidor

5.0.3 Esquema do protocolo implementado

De modo a resumir e facilitar a compreensão do protocolo utilizado neste trabalho, construímos a seguinte figura:

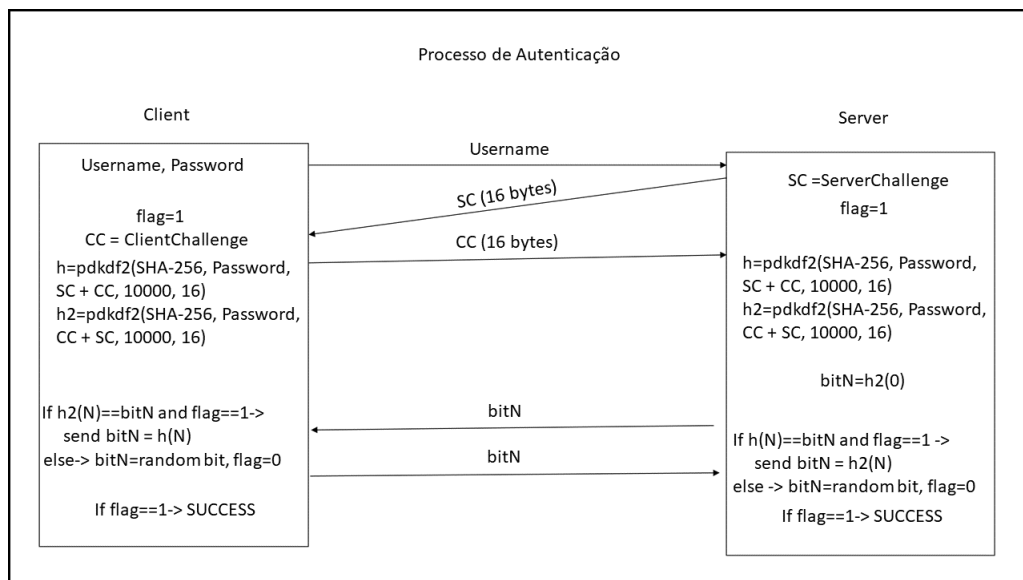


Figura 5.9: Esquema do protocolo

Capítulo 6

Conclusão

Após a realização deste trabalho em que tivemos de implementar uma webapp de autenticação. Percebemos que o "mundo" na internet pode ser bastante inseguro, daí usarmos algumas técnicas de encriptação para tornar o processo de autenticação mais seguro para o utilizador.

Este trabalho deu-nos uma visão geral dos perigos existentes e como nos podemos proteger a nós e às aplicações em relação aos mesmos. Achamos que o objetivo do trabalho foi atingido, usámos os conhecimentos adquiridos nas aulas para desenvolver a UAP através de técnicas de autenticação e encriptação.

Contribuições dos autores

A percentagem que deve ser atribuída a cada um dos autores é 25% à Ana Miguel Monteiro (AM), 25% ao Florentino Sánchez (FS), 25% ao Gonçalo Aguiar (GA) e 25% Tiago Bastos (TB).

Acrónimos

AM Ana Miguel Monteiro

FS Florentino Sánchez

GA Gonçalo Aguiar

TB Tiago Bastos

DETI Departamento de Electrónica, Telecomunicações e Informática

E-CHAP Enhanced Challenge-Handshake Authentication Protocol

UAP User authentication application

SC Server Challenge

CC Client Challenge