Projeto: Programação Utilizando Sockets

Caio Buarque de Gusmão , Diego Henrique Vilaca Calixto, José Gabriel Santos Nascimento e Julia Arnaud de Melo Fragoso

Centro de Informática UFPE

Universidade Federal de Pernambuco

Conteúdo

- Introdução ao Projeto
- Detalhamento da Implementação
- Pendências
- Dificuldades
- Conclusão do Projeto

Introdução ao Projeto

Objetivos

Servidor de Autenticação

Servidor Web

Planejamento

Reuniões Semanais

Divisão de Tarefas

Detalhamento da Implementação

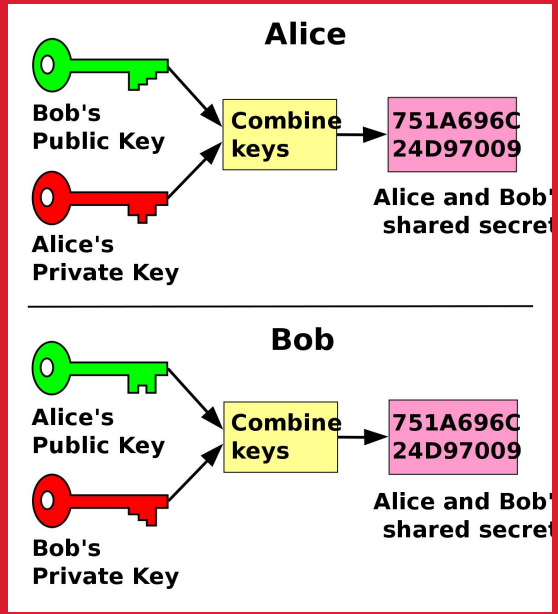
Bibliotecas Utilizadas:

- Sockets
- Cryptocode
- uuid
- Pickle
- Threading
- Cryptography.fernet
- Random
- Rsa
- Email.utils, Datetime e Time

Detalhamento da Implementação

Algoritmo de Criptografia

Para a primeira parte do projeto, referente ao servidor de autenticação, desenvolvemos o algoritmo de criptografia baseado na troca de chaves do algoritmo de **Diffie-Hellman** onde o cliente e o servidor compartilharão informações através do Handshake de maneira pública e através delas irão gerar uma chave secreta.

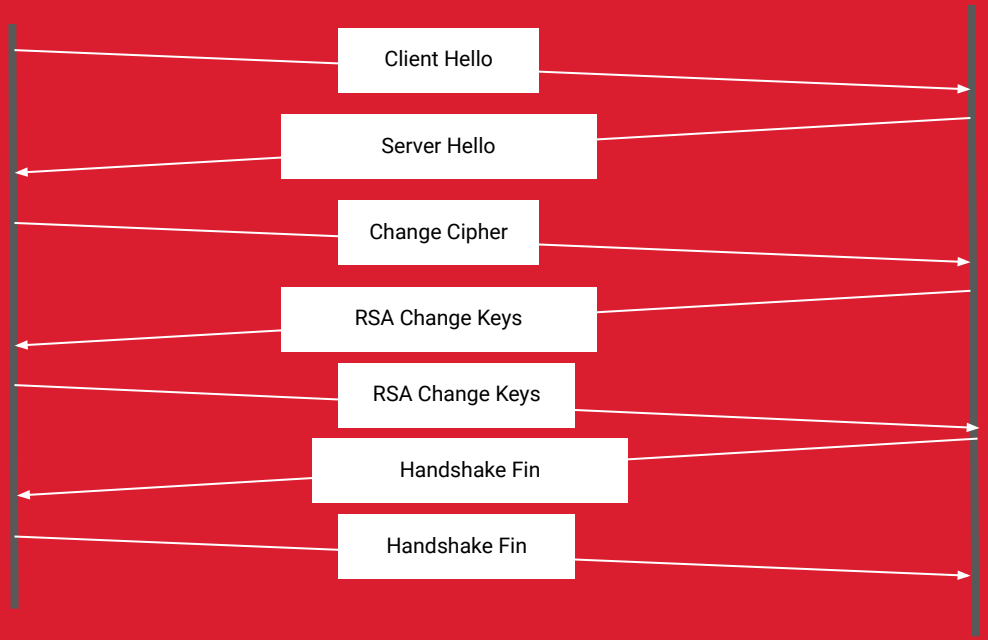


Detalhamento da Implementação

Handshake

Cliente

- recebimento de chaves públicas
- Cálculo da chave modular
- Criação e envio da chave de assinatura digital
- Cálculo da chave secreta



Servidor

- envio de chaves públicas
- Cálculo da chave modular
- Recebimento da chave pública de assinatura digital
- Cálculo da chave secreta

Detalhamento da Implementação

Cryptocode e o GET

1 - Cliente escreve o pedido no terminal.
2 - A mensagem do cliente é criptografada usando a chave secreta.
3 - A mensagem agora criptografada é transformada em bits e enviada ao servidor.
4 - O servidor recebe essa mensagem em bits e transforma ela para string.
5 - Para finalizar o Servidor descriptografa a mensagem, tendo em posse pôr fim do pedido feito pelo cliente

Vamos analisar isso na implementação....

Detalhamento da Implementação

Cliente:

```
# função que lida com as requisições get
def GET(mClientSocket, req):
    req = cryptocode.encrypt(req, chave_secreta_cliente)  # criptografia da requisição do ge
    req = req.encode()  # transformando em bytes para mandar pro servidor
    mClientSocket.send(req)
```

Servidor:

```
req = req.decode()  # da decode nos bytes para por na função de descriptografia em str
req = cryptocode.decrypt(req, chave_secreta_servidor)  # decriptografado utilziando a
```

Detalhamento da Implementação

Handle Request

Esta é a principal função do nosso código pois, a partir dela, as outras funções são chamadas

```python
# Essa função é uma thread que lida com a comunicação via sockets de cada cliente
def HandleRequest(mClientSocket, mClientAddr, P_ChavePubServ, G_ChavePubServ, B_ChavePrivServ, banco_de_dados):
    # inicialmente vamos ver se o cliente ja se comunicou antes com o servidor atraves de um indentificador unico
    # se ele não possuir um indentificador ou não for achado um vai ser criado um novo
    c1, existe_chave = Acharindentificador(mClientSocket, banco_de_dados, mClientAddr)
    # essa função retorna c1, que é a estrutura de dados do cliente contendo informações sobre ele
    # aqui por exemplo foi printado o endereço apenas para testar
    # e também retorna existe chave que verifica se o cliente ja possui as chaves

    print(f"Cliente {clientAddr} com indentificador : {c1.indentificador}")
    # caso não exista chaves é necessário cria-las através do handshake
    if not existe_chave:
        # em seguida é feito o handshake para adquirir a chave secreta
        chave_secreta_servidor, rsa_chave_pub_cliente = Handshake(mClientSocket, P_ChavePubServ, G_ChavePubServ, B_ChavePrivServ)
        # é salvo as chaves de acordo com o cliente
        c1.chave_secreta = chave_secreta_servidor
        c1.rsa_chave_secreta = rsa_chave_pub_cliente

    # A próxima etapa consiste em atender as requisições GET do cliente
    GetHandler(mClientSocket, chave_secreta_servidor, rsa_chave_pub_cliente)
```

Detalhamento da Implementação

Thread

Para que o nosso servidor tivesse a capacidade de atender múltiplos clientes em execução nós aplicamos o conceito de thread no código.

```
# Colocar o servidor para escutar as solicitações de conexão
mSocketServer.listen()
while True:
    # Este loop foi colocado para que o servidor conseguisse se conectar com vários cliente;
    clientSocket, clientAddr = mSocketServer.accept()
    print(f'\nO servidor aceitou a conexão do Cliente: {clientAddr}')

    # Criação de múltiplas threads para que o servidor consiga responder mais de um cliente por vez.
    Thread(target=HandleRequest, args=(
        clientSocket,
        clientAddr,
        P_ChavePubServ,
        G_ChavePubServ,
        B_ChavePrivServ,
        banco_de_dados)).start()
```

Detalhamento da Implementação

Identificador

Para cada cliente ser identificado e referenciado no servidor foi necessário criar um identificador único.

```python
# essa função é reponsável por criar um novo indentificador e salvar como novo cliente no banco de dados
def NovoIndentificador(banco_de_dados, mClientAddr):
    # é utilizado a função uuid1 da biblioteca uuid para criar um conjunto de caracteres que vão servir
    # para como indentificador único do cliente
    indentificador = uuid1()
    # a biblioteca cria o identificador numa estrutura de dados dela e a gente passa para string para melhor manipulação
    indentificador = str(indentificador)

    # criação da estrutura de dados cliente
    c1 = Cliente(indentificador, mClientAddr)  # é passado como parametro o indentificador e o endenreço do seu acesso
    # setattr(banco_de_dados, indentificador, c1)
    banco_de_dados[indentificador] = c1  # salvando no banco de dados o cliente com o seu id

    return indentificador, c1  # retornamos o id e o cliente
```

Detalhamento da Implementação

Cliente

Criamos uma estrutura de dados chamada cliente que armazena o identificador, endereço, chave secreta para criptografia e chave secreta para assinatura digital de cada cliente.

```
class Cliente:
    def __init__(self, indentificador, endereço):
        self.indentificador = indentificador
        self.endereço = endereço
        self.chave_secreta = None
        self.rsa_chave_secreta = None
```

Detalhamento da Implementação

Banco de Dados

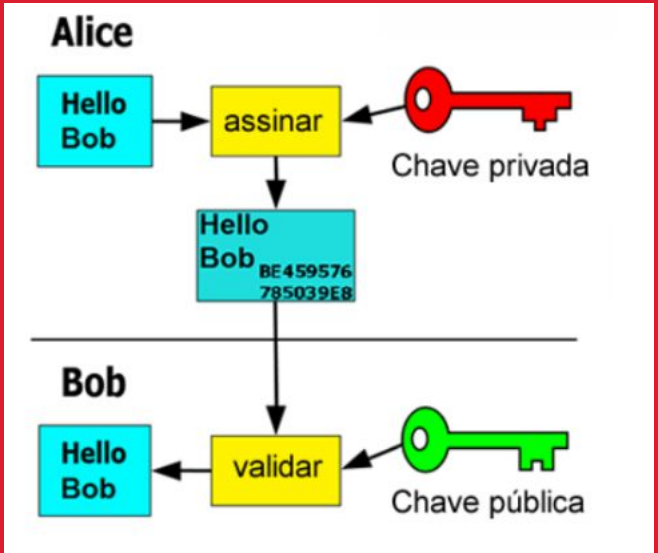
O banco de dados está organizado em um dicionário onde a chave é o identificador e o objeto cliente é seu par.

```
{'3fb6d834-5596-11ed-b4d3-00e04c05b934': <DadosCliente.Cliente object at 0x00000129FB7C79A0>,
 '46c8f9c8-5596-11ed-a4da-00e04c05b934': <DadosCliente.Cliente object at 0x00000129FBB2C040>}
```

Detalhamento da Implementação

Assinatura Digital

Assinar a mensagem GET do cliente, fazendo um hash com a sua chave privada, e verificar essa assinatura do lado do servidor, com a chave pública, que é compartilhada entre eles.



Como isso fica na implementação? →

Detalhamento da Implementação

Abaixo é possível ver a função, do lado do cliente, que cria as chaves e compartilha a chave pública com o servidor, utilizando a biblioteca RSA.

```
if req == "RSA CHANGE KEY ":  # faz a troca de chaves da biblioteca de assinatura digital
    resp = "RSA CHANGE KEY "
    mClientSocket.send(resp.encode())  # reponde, notificando para o servidor que irá ocorrera a troca de chaves

    (rsa_chave_pub_cliente, rsa_chave_priv_cliente) = rsa.newkeys(2048)

    rsa_chave_pub_cliente = rsa_chave_pub_cliente.save_pkcs1(format="DER")  # serializa a chave publica do cliente para bytes
    mClientSocket.send(rsa_chave_pub_cliente)  # envia a chave publica do cliente para o servidor
```

Detalhamento da Implementação

Nessa imagem, observa-se que o GET é assinado através de um hash que usa a chave privada do cliente.

```python
def GET(mClientSocket, req, rsa_chave_priv_cliente):

    # assinatura digital
    req = cryptocode.encrypt(req, chave_secreta_cliente)  # criptografia da requisição do get
    req = req.encode()  # transformando em bytes para mandar pro servidor
    mClientSocket.send(req)

    req_assinado = rsa.sign(req, rsa_chave_priv_cliente, 'SHA-512')  # assinatura digital
    mClientSocket.send(req_assinado)  # envio da assinatura digital para o servidor

    req = req.decode()
    req = cryptocode.decrypt(req, chave_secreta_cliente)
```

Detalhamento da Implementação

Por fim, podemos ver que, do lado do servidor, a assinatura digital é verificada com a função verify, autenticando a mensagem.

```
try:
    rsa.verify(req, req_assinado, rsa_chave_pub_cliente)  # verifica a assinatura digital
except rsa.pkcs1.VerificationError:  # tratamento de error caso a assinatura não seja válida
    print("verificação falhou")  # verificação falhou lidar com isso no tratamento de error
    # error 403
```

Detalhamento da Implementação

Cryptography.Fernet e os arquivos

A confidencialidade é necessária na entrega do arquivo requerido feito pelo servidor, e no recebimento pelo cliente. Para isso, utilizamos a biblioteca Fernet que criptografa os arquivos. Assim, seguimos os passos...

1 - A chave de criptografia de arquivo é enviada ao cliente.

2 - O servidor usa a chave para criptografar o arquivo e enviá-lo ao cliente

3 - O cliente recebe o arquivo criptografado e usa a chave para descriptografar

Detalhamento da Implementação

Criptografia do Arquivo

Cliente

Servidor

Faz o pedido

- Cria a chave

Envia a chave fernet

- Descriptografa a chave
- Torna ela usável

Envia o arquivo criptografado

- Criptografa o arquivo com ela
- Descriptografa o arquivo

Detalhamento da Implementação

Criptografia do Arquivo (Por dentro do código)

Cliente

Servidor

```python
with open("filekey.key", "wb") as key_file:
    crypt_keybits = mClientSocket.recv(2048)
    crypt_key_decode = crypt_keybits.decode()
    crypt_key = cryptocode.decrypt(crypt_key_decode, chave_secreta_cliente)

    crypt_key = crypt_key.encode()
    key_file.write(crypt_key)

with open('filekey.key', 'rb') as filekey:  # Lendo a chave recebida
    key = filekey.read()
```

```python
# Gerando a chave
key = Fernet.generate_key()
usable_key = Fernet(key)
decode_key = key.decode()
crypt_key = cryptocode.encrypt(decode_key, chave_secreta_servidor)
clientSocket.send(crypt_key.encode())
```

```python
with open(req, "rb") as crip_file:
    crypted = crip_file.read()

decrypt = usable_key.decrypt(crypted)

with open(req, 'wb') as final_file:
    final_file.write(decrypt)
```

```python
with open(req, 'rb') as file:  # Leitura do arquivo para variavel
    original = file.read()

Msg_200(mClientSocket, chave_secreta_servidor, formato)

encrypted = usable_key.encrypt(original)  # Criptografando a variavel que contem o arquiv

with open(req, 'wb') as encrypted_file:  # Escrita do Arquvio Criptografado
    encrypted_file.write(encrypted)

with open(req, 'rb') as file:  # Seleciona o arquvio pedido
    file = file.read()  # Leitura das linhas do arquivo
    clientSocket.send(file)  # Envio das linhas do arquivo para o cliente

with open(req, 'wb') as final_file:
    final_file.write(original)
```

Detalhamento da Implementação

4 Pilares de Segurança

Confidencialidade

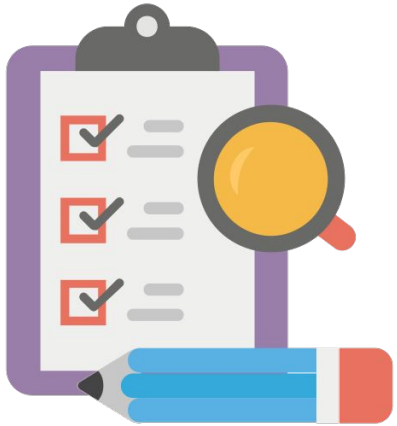
Autenticidade

Integridade

Disponibilidade

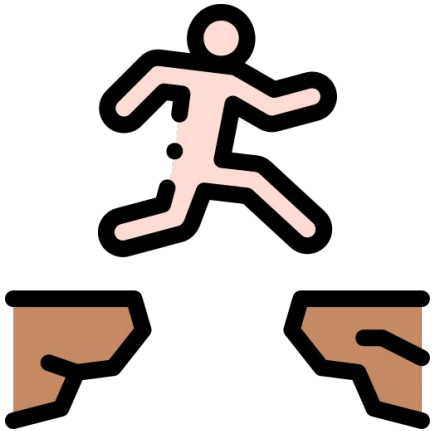
Pendências

- Problema na permanência da conexão do cliente com o servidor



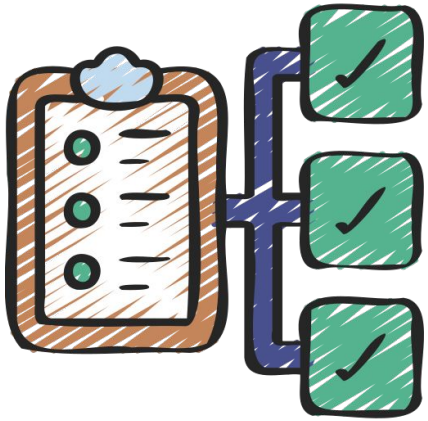
Dificuldades

- Entender os conceitos necessários para o desenvolvimento do projeto
- Saber como aplicar os conceitos que foram pesquisados e estudados
- Compreender o servidor de autenticação
- Dificuldade para integrar os códigos da equipe



Conclusão

Com o desenvolvimento deste projeto tivemos que lidar com uma série de dificuldades, principalmente em relação à autoridade certificadora, onde tivemos que pesquisar sobre o tema, ler e estudar, para conseguir aplicar esses processos no código. Apesar disso, o grupo conseguiu contornar essas dificuldades e evoluir, assim, nosso aprendizado sobre a parte mais **prática** da disciplina e sobre as **questões de segurança** foi muito ampliado com esse trabalho, uma vez que tivemos que lidar com sockets, algoritmos de criptografia, assinatura digital, servidor web, threads, e várias outras ferramentas.



Obrigado(a)!