Prática 04: Assinatura Digital com função Hash

Disciplina: Redes de Computadores

28/04/24

1 O que é uma Função Hash?

Uma função hash é um algoritmo que mapeia dados de qualquer tamanho para um valor fixo, geralmente menor, conhecido como "hash"ou "resumo". Essa função foi desenvolvida para ser eficiente em termos de tempo de execução e criar hashes distintos para vários tipos de dados.

As funções hash são frequentemente usadas na criptografia, bancos de dados, segurança da informação e outras áreas da computação.

2 SHA-256 (Secure Hash Algorithm 256 bits)

O SHA-256 é uma versão mais segura do algoritmo SHA, que produz um resumo de 256 bits (32 bytes). É amplamente utilizado em aplicações de segurança da informação e criptografia.

Exemplo em Python:

```
// python script to show an example for sha256
import hashlib

// Dados de entrada
data = "Hello, world!"

// Calculando o hash SHA-256
sha256_hash = hashlib.sha256(data.encode()).hexdigest()

print("SHA-256:", sha256_hash)

// output:
//SHA-256:315f5bdb76d078c43b8ac0064e4a0164612b1fce77c869345bfc94c75894edd3
```

Listing 1: SHA-256.

3 Assinatura Digital

A assinatura digital é uma maneira de garantir que as informações digitais sejam verdadeiras e seguras. Ela é criada a partir de um hash calculado do conteúdo da mensagem, que foi criptografado usando a chave privada do remetente. A chave pública do remetente pode ser usada para realizar uma verificação matemática dessa assinatura. A integridade e autenticidade das informações são confirmadas se o hash decifrado da assinatura for igual ao hash calculado da mensagem original. Caso contrário, a assinatura ou a mensagem mostram que houve uma modificação.

4 PyCryptodome

O PyCryptodome é uma biblioteca escrita em Python que ajuda a implementar algoritmos de hash e criptografia. É uma extensão do PyCrypto que oferece uma API mais fácil de usar e maior segurança.

Em esta introdução, discutiremos os recursos do PyCryptodome, incluindo algoritmos de hash, criptografia simétrica e assimétrica e assinatura digital.

5 Criptografia Assimétrica

Uma chave privada e uma chave pública são usados na criptografia assimétrica. O PyCryptodome pode cifrar, decifrar e criar chaves usando algoritmos como RSA.

6 Funcionalidades Principais:

1. Os principais recursos incluem algoritmos de criptografia simétrica e assimétrica: O PyCryptodome suporta vários algoritmos de criptografia, como AES, DES, RSA e ECC.

2. Algoritmos de Hash: Disponibiliza a execução de vários algoritmos de hash, incluindo SHA-256, SHA-512 e MD5, que são usados para gerar resumos criptográficos de dados.

3. Geração de Números Aleatórios Seguros Criptograficamente: O PyCryptodome oferece um gerador de números aleatórios seguro, que é essencial para vários processos criptográficos.

4. Gerenciamento de Chaves: facilita a criação e manipulação de chaves criptográficas.

5. Assinatura Digital: Facilita a criação e verificação de assinaturas digitais, garantindo que os dados sejam autenticos e seguros.

7 Por que usar o PyCryptodome?

- **Segurança**: Implementações de algoritmos criptográficos confiáveis.
- **Facilidade de uso**: API fácil de usar e documentação detalhada para desenvolvimento.
- **Flexibilidade**: Para atender às necessidades de segurança específicas de cada aplicação, é suportado um grande número de algoritmos.
- **Código Aberto**: Disponível sob a Licença de Código Aberto Apache 2.0, pode ser usado em projetos comerciais e não comerciais.

8 Exemplo de Uso:

```
// biblioteca
from Crypto.Hash import SHA256

// Mensagem de texto
texto = 'Exemplo de mensagem para hash'

// Criando um objeto hash SHA256
hash_obj = SHA256.new()

// Atualizando o hash com a mensagem de texto
hash_obj.update(texto.encode())

// Obtendo o hash em formato hexadecimal
hash_resultado = hash_obj.hexdigest()

print("Hash SHA256 da mensagem:", hash_resultado)
```

Listing 2: SHA-256.

9 Atividade

Faça uma aplicação cliente-servidor (continuação da aula de criptografia) para demonstrar a programação de socket, função Hash e assinatura digital, da seguinte maneira:

1. O cliente e o servidor utilizam assinatura com chave pública RSA e Hash com SHA256;
2. A chave pública do servidor foi previamente compartilhada para o cliente.
3. O servidor inicializa e fica aguardando conexão.
4. Um cliente envia para o servidor um texto(chamado de desafio);
5. O servidor recebe o desafio, calcula o hash, assina o hash com sua chave privada e envia para o cliente.
6. O cliente recebe a resposta, calcula o hash do desafio e compara com a decriptografia (verificação) da mensagem ddo servidor, com a chave pública do servidor.
7. Rode o Wireshark e veja o funcionamento do seu programa na rede.

A configuração em que o servidor e o cliente usam assinaturas digitais com uma chave pública RSA e um hash SHA256. Para enviar uma mensagem ao servidor de forma segura e garantir sua autenticidade e integridade, o cliente deve seguir os passos a seguir:

1. **Criação da Assinatura Digital**:
   - O cliente calcula o hash SHA256 da mensagem.
   - Em seguida, ele assina o hash usando sua chave privada RSA.
   - O resultado é a assinatura digital da mensagem.
2. **Envio da Mensagem e da Assinatura ao Servidor**:
   - O cliente envia a mensagem e sua assinatura digital ao servidor.

Por sua vez, o servidor recebe a mensagem e a assinatura digital e executa os seguintes passos:

1. **Verificação da Assinatura Digital**:
   - O servidor calcula o hash SHA256 da mensagem recebida.
   - Ele usa a chave pública RSA do cliente para verificar a assinatura digital recebida.
   - Se a verificação for bem-sucedida, isso confirma que a mensagem foi realmente enviada pelo cliente e não foi alterada desde então.
2. **Processamento da Mensagem**:

- Se a assinatura digital for válida, o servidor processa a mensagem.

Ao permitir que apenas o cliente use sua chave privada para criar a assinatura digital, essa técnica garante que a mensagem seja verdadeira e confiável. Além disso, como a assinatura é baseada no hash SHA256 da mensagem, qualquer mudança na mensagem seria detectada durante a verificação da assinatura. Agora, vamos criar a chave pública-privada com um script python:

```
from Crypto.PublicKey import RSA
from Crypto import Random

// Tamanho da chave RSA em bits
tamanho_chave = 1024

// Gerando um objeto Random
rand_gen = Random.new().read

// Gerando as chaves RSA
chave = RSA.generate(tamanho_chave, rand_gen)

// Separando as chaves privada e pública
chave_privada = chave.export_key()
chave_publica = chave.publickey().export_key()

// Salvando as chaves em arquivos
with open("chave_privada.pem", "wb") as f:
    f.write(chave_privada)

with open("chave_publica.pem", "wb") as f:
    f.write(chave_publica)

print("Chaves geradas e salvas com sucesso!")
```

Listing 3: Chaves geradas - publica-privada.

```
// -----BEGIN PUBLIC KEY-----
// MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDIwVpFBwp83YsLBpZen+j5bP/d
// io/guJA1YtdyRIkERqjkeySaoRNCqSnhuZfLN26gdyxGxuarCtJ4o02a/aRCeBgX
// gyFjf/HIqKu/gZnd2csSLR8BPKpRMc91iLou1utZ2o6vSdesEfQR5NczovBAKUx6
// 7kLjXGdF/Vhtfi1lAQIDAQAB
// -----END PUBLIC KEY-----%

// -----BEGIN RSA PRIVATE KEY-----
// MIICXAIBAAKBgQDIwVpFBwp83YsLBpZen+j5bP/dio/guJA1YtdyRIkERqjkeySa
// oRNCqSnhuZfLN26gdyxGxuarCtJ4o02a/aRCeBgXgyFjf/HIqKu/gZnd2csSLR8B
// PKpRMc91iLou1utZ2o6vSdesEfQR5NczovBAKUx67kLjXGdF/Vhtfi1lAQIDAQAB
// AoGAPCa2+ezIpy4oTabtIjAOucF/jq1IO+CBEQXrIOlJFpdnXoJJLu2pXDVcf65A
// vZp/0qOyiAhrr/8fnhbsF079SodRc+qgJR/6odo2bnVaFWf4H0sDjNeN38rEbETv
// IktoXOcei9dKUcwIQvzFIHI8cz/VuAgcO6MT9LKvjMRJKDsCQQDcOZZZi4OCwndF
// vxMXICekCwlTkvBglginmpIbbh9X5ifAXxqZJ5dHi9aT35NH7oLMxJkNOSM89+FA
// 4SL23g6jAkEA6V4WajcegY1XssWogErO17OEA7rvgNgHUhlrR8IsGu8AtkjOzfQj
// s0kNuBngzjCHjDhpLcAkAgDFe0WA7xJsCwJABqdGv5XTd1Pgvp6zOPOjvvUGZxv9
// Xy2pPUcSOvnswH8XnFxDNXVYwLSc2wLaNEYkdYNLDHc5dVIX4BntMIAs+QJAB5Ea
// bvU8kvzPRCeukAJc9JeIh0pva6EVk67pUJlWLsVjI4X21qy835pVzItiQ61FJ+HI
// XOhkon/95OJYrOfPAwJBAJEB9r9jiCYm/rloJOcHksmAXcX1E3DOsR4Wa5eq/FCg
// 7D+4eKHUepnUnz9/Gd6+XjxPsri/BrXjd/PUqFAbWa4=
// -----END RSA PRIVATE KEY-----%
```

Listing 4: Chaves geradas - publica-privada.

- A chave pública do servidor foi previamente compartilhada para o cliente.
- O servidor inicializa e fica aguardando conexão.

Agora, vamos escrever o código para o servidor:

```
import socket
from Crypto.Hash import SHA256
from Crypto.Signature import pkcs1_15
from Crypto.PublicKey import RSA

// Chave pública do servidor (previamente compartilhada)
chave_publica_servidor = None  // Insira a chave pública aqui

// Inicializando o socket do servidor
host = "localhost"
porta = 12346
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((host, porta))
server_socket.listen(1)
```

```
// Carregando a chave privada do arquivo
with open("chave_privada.pem", "rb") as f:
    chave_privada_servidor = RSA.import_key(f.read())

print("Servidor aguardando conexão...")

while True:
    // Aceitando a conexão
    conn, addr = server_socket.accept()
    print("Conectado com", addr)

    // Recebendo o desafio do cliente
    desafio = conn.recv(1024).decode('utf-8')
    print(f"desafio: {desafio}")

    // Calculando o hash do desafio
    hash_desafio = SHA256.new(desafio.encode('utf-8'))
    print(f"hash(desafio) >>> {hash_desafio}")

    // Assinando o hash com a chave privada do servidor
    assinatura = pkcs1_15.new(chave_privada_servidor).sign(hash_desafio)
    print(f"server(assinatura) >>> {assinatura}")

    // Enviando a assinatura para o cliente
    conn.send(assinatura)
```

Listing 5: Chaves geradas - publica-privada.

Executando o código acima, temos:

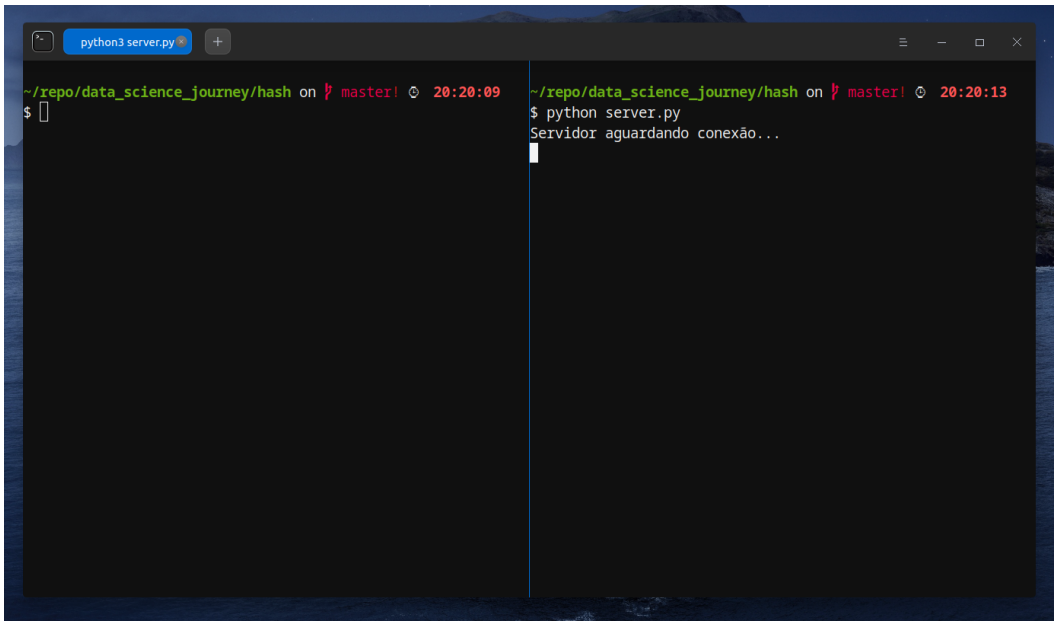

Figura 1: Servidor - Aguardando a conexão.

Como visto acima, o servidor esta aguardando a conexão do cliente. Agora, vamos escrever o código para o cliente:

```
import socket
from Crypto.Hash import SHA256
from Crypto.Signature import pkcs1_15
from Crypto.PublicKey import RSA

// Carregando a chave pública do arquivo
with open("chave_publica.pem", "rb") as f:
    chave_publica_servidor = RSA.import_key(f.read())

// Inicializando o socket do cliente
host = "localhost"
porta = 12345
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((host, porta))

// Enviando o desafio para o servidor
desafio = "Desafio para o servidor"
print(f"client: {desafio}")
client_socket.send(desafio.encode("utf-8"))

// Recebendo a assinatura do servidor
assinatura = client_socket.recv(1024)
print(f"assinatura for client from server: {assinatura}")

// Calculando o hash do desafio
hash_desafio = SHA256.new(desafio.encode("utf-8"))
print(f"hash_desafio(client): {hash_desafio}")

// Verificando a assinatura com a chave pública do servidor
try:
    pkcs1_15.new(chave_publica_servidor).verify(hash_desafio, assinatura)
    print("Assinatura válida.")
except (ValueError, TypeError):
    print("Assinatura inválida.")

// Fechando o socket do cliente
client_socket.close()
```

Abaixo podemos ver que após o cliente enviar a mensagem para o servidor em seguida o servidor envia a assinatura para o cliente.

Com o desafio é necessário calcular a hash do desafio que será utilizado para verificar a assinatura com a chave pública do servidor.

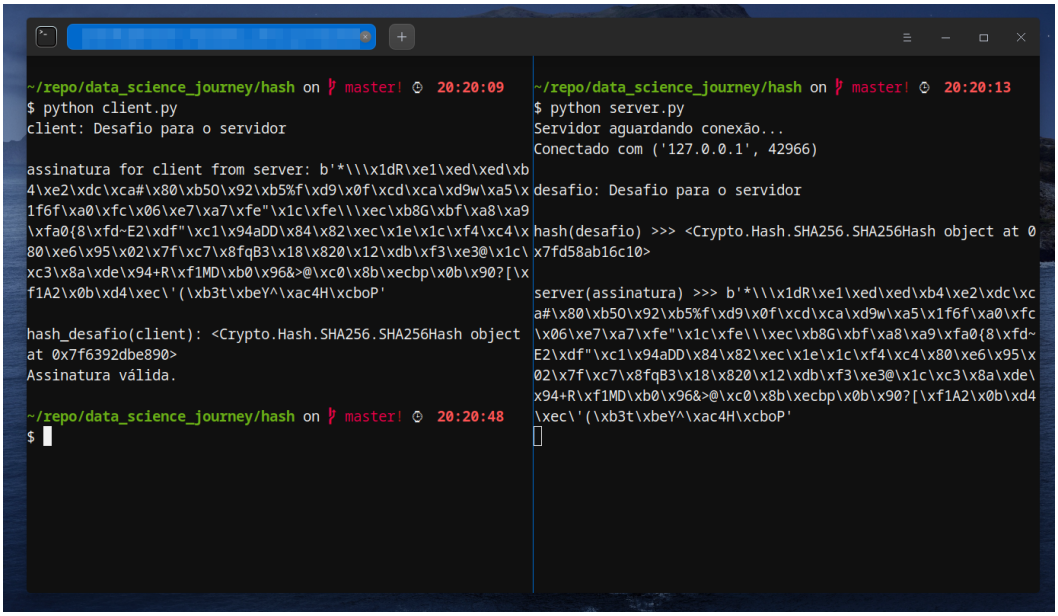

Figura 2: Cliente - Enviando a mensagem para o servidor.

Por fim, um método eficaz de garantir a autenticidade e a integridade das mensagens trocadas é o uso de assinaturas digitais com chave pública RSA e hash com SHA256 entre cliente e servidor.

Ao assinar uma mensagem usando sua chave privada, o cliente garante que apenas ele poderia ter gerado a assinatura usando a chave privada correspondente. Além disso, como o hash SHA256 da mensagem serve como base para a assinatura, qualquer alteração na mensagem seria identificada pelo servidor durante a verificação da assinatura.

Essa técnica é confiável, robusta e adiciona segurança à comunicação cliente-servidor. O PyCryptodome fornece as ferramentas que você precisa para implementar esse sistema em Python de forma eficiente e segura.

```
// Below I will organize in ordem of events
// Server:
Servidor aguardando conexão...
Conectado com ('127.0.0.1', 42966)

// Server:
// mensagem from Client:
desafio: Desafio para o servidor

// Server:
hash(desafio) >>> <Crypto.Hash.SHA256.SHA256Hash object at 0x7fd58ab16c10>

// Server:
server(assinatura) >>> b'*\\\x1dR\xe1\xed\xed\xb4\xe2\xdc\xca#
\x80\xb5O\x92\xb5%f\xd9\x0f\xcd\xca\xd9w\xa5\x1f6f\xa0\xfc\x06
\xe7\xa7\xfe"\x1c\xfe\\\xec\xb8G\xbf\xa8\xa9\xfa0{8\xfd~E2\xdf"
\xc1\x94aDD\x84\x82\xec\x1e\x1c\xf4\xc4\x80\xe6\x95\x02\x7f\xc7
\x8fqB3\x18\x820\x12\xdb\xf3\xe3@\x1c\xc3\x8a\xde\x94+R\xf1MD\xb0
\x96&>@\xc0\x8b\xecbp\x0b\x90?[\xf1A2\x0b\xd4\xec\'(\xb3t\xbe
Y^\xac4H\xcboP'

// Client:
assinatura for client from server: b'*\\\x1dR\xe1\xed\xed\xb4\xe2
\xdc\xca#\x80\xb5O\x92\xb5%f\xd9\x0f\xcd\xca\xd9w\xa5\x1f6f\xa0\
xfc\x06\xe7\xa7\xfe"\x1c\xfe\\\xec\xb8G\xbf\xa8\xa9\xfa0{8\xfd~E2
\xdf"\xc1\x94aDD\x84\x82\xec\x1e\x1c\xf4\xc4\x80\xe6\x95\x02\x7f\
xc7\x8fqB3\x18\x820\x12\xdb\xf3\xe3@\x1c\xc3\x8a\xde\x94+R\xf1MD
\xb0\x96&>@\xc0\x8b\xecbp\x0b\x90?[\xf1A2\x0b\xd4\xec\'(\xb3t\xb
eY^\xac4H\xcboP'

//Client:
hash_desafio(client): <Crypto.Hash.SHA256.SHA256Hash object at 0x7f6392dbe890>
Assinatura válida.
```