**APLICAÇÃO DE ENVIO DE MENSAGEM CRIPTOGRAFADA E DESCRIPTOGRAFAR NO DESTINO USANDO CHAVE SIMÉTRICA**

Para essa aplicação, usamos o algoritmo AES (Advanced Encryption Standard), como já descrito no título, ele é um algoritmo de criptografia simétrica, o que significa que as chaves usadas para criptografar e descriptografar são as mesmas. Como o AES é uma cifra de bloco, os dados são divididos em blocos de 128 bits antes de criptografá-los com a chave de até 256 bits.

Código:

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.backends import default\_backend

import base64

def generate\_key(password, salt):

    kdf = PBKDF2HMAC(

        algorithm=hashes.SHA256(),

        length=32,

        salt=salt,

        iterations=100000,

        backend=default\_backend()

    )

    return kdf.derive(password.encode())

def encrypt\_message(message, key):

    iv = b'1234567890123456' # Vetor de inicialização de 16 bytes

    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend()) # Objeto 'cipher' que será utilizado para criptografar

    encryptor = cipher.encryptor() # Objeto 'encryptor' a partir do 'cipher', que será usado para criptografar a mensagem

    ciphertext = encryptor.update(message.encode()) + encryptor.finalize() # Criptografamos a mensagem convertida em bytes e concatenamos o resultado com IV

    return base64.b64encode(iv + ciphertext) # Retornamos o texto cifrado codificado em base64 para facilitar o envio pela rede

def decrypt\_message(encrypted\_message, key):

    encrypted\_message = base64.b64decode(encrypted\_message)

    iv = encrypted\_message[:16]  # Obtém o IV do início da mensagem

    ciphertext = encrypted\_message[16:]  # Obtém o texto cifrado

    cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default\_backend()) # Seguindo a mesma lógica da função de cima, só que para descriptografar

    decryptor = cipher.decryptor()

    decrypted\_message = decryptor.update(ciphertext) + decryptor.finalize()

    return decrypted\_message

if \_\_name\_\_ == "\_\_main\_\_":

    password = "senha\_super\_secreta"

    salt = b'salt\_aleatorio'

    message = input("Qual mensagem deseja enviar e criptografar? ")

    key = generate\_key(password, salt)

    encrypted\_message = encrypt\_message(message, key)

    print("Mensagem criptografada:", encrypted\_message)

    decrypted\_message = decrypt\_message(encrypted\_message, key)

    print("Mensagem descriptografada:", decrypted\_message.decode('utf-8'))

**EXPLICAÇÕES SOBRE A LÓGICA DA IMPLEMENTAÇÃO E ALGUMAS FUNÇÕES E CLASSES:**

Bom, pensando em um cenário real onde poderia ocorrer ataques, utilizamos algumas funções e classes que deixam o algoritmo mais resistente.

Na função “generate\_key” geramos uma chave a partir de uma senha e um salt, quando uma senha é usada diretamente para derivar uma chave, pode ser meio problemático porque duas pessoas com a mesma senha terão a mesma chave derivada. Isso significa que se um atacante souber a chave derivada através, por exemplo, de um vazamento de dados, ele poderá usar essa chave para descriptografar as mensagens de qualquer pessoa que tenha a mesma senha.

Ao adicionar um salt único (que é uma sequência de bytes aleatórios) a cada senha antes de derivar a chave, mesmo que duas pessoas tenham a mesma senha, as chaves derivadas serão diferentes devido ao salt diferente, tornando inútil a chave que o atacante tenha.

A geração de chaves ter 100,000 iterações é uma técnica de segurança chamada “key\_stretching” (estiramento de chave), quando um atacante tenta quebrar a senha através de força bruta, ele testa diferente combinações até encontrar a correta. Se o processo de derivação de chave for muito rápido, um atacante pode testar muitas senhas em pouco tempo, o que torna mais fácil encontrar a senha correta.

Ao adicionar múltiplas iterações, o processo de derivação se torna mais lento para todos os envolvidos, porque o atacante vai levar mais tempo testar as senhas.

A função “default\_backend” é bem simples, quando chamamos ela, a biblioteca verifica o ambiente em que o código está sendo executado, e seleciona o backend mais adequado a aquele ambiente. O uso dessa função é conveniente porque permite seja portável para diferentes sistemas operacionais, sem precisar ficar alterando o código para especificar qual backend é mais bem adequado.

O IV (Vetor de Inicialização) é gerado como uma sequência de bytes aleatórios e é usado junto com a chave para criptografar a mensagem, isso garante que, mesmo que a mensagem seja criptografada várias vezes com a mesma chave, os textos cifrados resultantes serão diferentes devido ao uso de IVs diferentes.