Alejandro García Gutiérrez

CODIGO CRIPTOGRAFIA

1 –

# XOR de datos binarios

def xor\_data(binary\_data\_1, binary\_data\_2):

    return bytes([b1 ^ b2 for b1, b2 in zip(binary\_data\_1, binary\_data\_2)])

# Claves definidas como cadenas hexadecimales

key\_f\_c = "B1EF2ACFE2BAEEFF"

key\_f\_m= "91BA13BA21AABB12"

# Convertimos las claves a bytes

key\_f\_c = bytes.fromhex(key\_f\_c)

key\_f\_m = bytes.fromhex(key\_f\_m)

# XOR entre las claves

key\_dinamic = xor\_data(key\_f\_c, key\_f\_m)

# Resultado

print(f"Resultado XOR (hex): {key\_dinamic.hex()}")

2 –

from Crypto.Cipher import AES

from Crypto.Util.Padding import unpad

# Mensaje cifrado (en bytes)

msg\_encryt = bytes.fromhex(

    "4d0f5238c29ce9a152f5297185f2bdc13d7c517a4f09de74e577f927fe672c8ece7ffa3440a2165e0de7bb5451cf84161257b376b2c00f92cee144addda07f8b9d27bef25b0621be95ed5986947ce282"

)

# Clave y IV

key\_d = bytes.fromhex("A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72")

iv\_d = bytes.fromhex("00000000000000000000000000000000")  # IV de 16 bytes

# Descifrado

try:

    cipher\_d = AES.new(key\_d, AES.MODE\_CBC, iv\_d)

    msg\_claro\_con\_padding = cipher\_d.decrypt(msg\_encryt)

    # Mostrar el padding añadido

    padding\_length = msg\_claro\_con\_padding[-1]

    print(f"Cantidad de padding añadido: {padding\_length} bytes")

    print("Bytes de padding:", msg\_claro\_con\_padding[-padding\_length:])

    # Quitar el padding y mostrar el mensaje

    msg\_claro = unpad(msg\_claro\_con\_padding, AES.block\_size, style='pkcs7')

    print("Mensaje descifrado (hex):", msg\_claro.hex())

    print("Mensaje descifrado (texto):", msg\_claro.decode("utf-8"))

except ValueError as e:

    print("Error durante el descifrado:", e)

except UnicodeDecodeError:

    print("El mensaje descifrado no es texto UTF-8 válido.")

3 -

from Crypto.Cipher import ChaCha20\_Poly1305

from base64 import b64decode, b64encode

import json

try:

    # Texto en claro (convertido a bytes)

    texto\_plano = 'KeepCoding te enseña a codificar y a cifrar'.encode('utf-8')  # Convertir a bytes

    # Clave de 32 bytes

    clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')

    # Nonce específico (decodificado desde Base64)

    nonce\_mensaje = b64decode('9Yccn/f5nJJhAt2S')  # Decodificar Base64 a bytes

    # Datos asociados (pueden estar vacíos, pero deben ser consistentes entre cifrado y descifrado)

    datos\_asociados = b''

    # Cifrado

    cipher = ChaCha20\_Poly1305.new(key=clave, nonce=nonce\_mensaje)

    cipher.update(datos\_asociados)

    texto\_cifrado, tag = cipher.encrypt\_and\_digest(texto\_plano)

    # Crear mensaje en JSON con todos los elementos codificados en Base64

    mensaje\_enviado = {

        "nonce": b64encode(nonce\_mensaje).decode('utf-8'),

        "datos\_asociados": b64encode(datos\_asociados).decode('utf-8'),

        "texto\_cifrado": b64encode(texto\_cifrado).decode('utf-8'),

        "tag": b64encode(tag).decode('utf-8')

    }

    json\_mensaje = json.dumps(mensaje\_enviado)

    # Mostrar información del cifrado

    print("Mensaje enviado (JSON):", json\_mensaje)

    print("Nonce (Base64):", mensaje\_enviado["nonce"])

    print("Texto cifrado (Base64):", mensaje\_enviado["texto\_cifrado"])

    print("Tag (Base64):", mensaje\_enviado["tag"])

    # Descifrado

    mensaje\_recibido = json.loads(json\_mensaje)  # Simular recepción del JSON

    # Configurar el descifrador

    decipher = ChaCha20\_Poly1305.new(

        key=clave,

        nonce=b64decode(mensaje\_recibido["nonce"])

    )

    decipher.update(b64decode(mensaje\_recibido["datos\_asociados"]))

    # Descifrar y verificar el mensaje

    texto\_descifrado = decipher.decrypt\_and\_verify(

        b64decode(mensaje\_recibido["texto\_cifrado"]),

        b64decode(mensaje\_recibido["tag"])

    )

    # Mostrar texto descifrado

    print("Texto descifrado:", texto\_descifrado.decode('utf-8'))

except (ValueError, KeyError, TypeError) as error:

    print("Problemas al descifrar...")

    print("El motivo del error es:", error)

**ChaCha20-Poly1305**

from Crypto.Cipher import ChaCha20\_Poly1305

from Crypto.Random import get\_random\_bytes

from base64 import b64encode, b64decode

import json

import time

def cifrar\_datos(texto\_plano, clave, datos\_asociados):

    """Cifra el texto plano garantizando confidencialidad e integridad."""

    # Convertir texto plano a bytes si es necesario

    if isinstance(texto\_plano, str):

        texto\_plano = texto\_plano.encode('utf-8')

    # Generar un nonce único

    nonce = get\_random\_bytes(12)

    # Crear el cifrador ChaCha20-Poly1305

    cipher = ChaCha20\_Poly1305.new(key=clave, nonce=nonce)

    cipher.update(datos\_asociados)

    texto\_cifrado, tag = cipher.encrypt\_and\_digest(texto\_plano)

    # Crear un mensaje JSON codificado en Base64

    mensaje\_enviado = {

        "nonce": b64encode(nonce).decode('utf-8'),

        "datos\_asociados": b64encode(datos\_asociados).decode('utf-8'),

        "texto\_cifrado": b64encode(texto\_cifrado).decode('utf-8'),

        "tag": b64encode(tag).decode('utf-8')

    }

    return json.dumps(mensaje\_enviado)

def descifrar\_datos(mensaje\_json, clave):

    """Descifra y verifica el mensaje recibido."""

    mensaje\_recibido = json.loads(mensaje\_json)

    # Decodificar valores de Base64

    nonce = b64decode(mensaje\_recibido["nonce"])

    datos\_asociados = b64decode(mensaje\_recibido["datos\_asociados"])

    texto\_cifrado = b64decode(mensaje\_recibido["texto\_cifrado"])

    tag = b64decode(mensaje\_recibido["tag"])

    # Configurar el descifrador ChaCha20-Poly1305

    cipher = ChaCha20\_Poly1305.new(key=clave, nonce=nonce)

    cipher.update(datos\_asociados)

    # Descifrar y verificar el texto cifrado

    texto\_descifrado = cipher.decrypt\_and\_verify(texto\_cifrado, tag)

    return texto\_descifrado.decode('utf-8')

# ----------------------------

# Ejemplo de uso:

try:

    # Clave de 32 bytes

    clave = bytes.fromhex('AF9DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120')

    # Datos asociados: incluir contexto adicional

    datos\_asociados = f"UsuarioID:1234;Fecha:{time.time()}".encode('utf-8')

    # Texto a cifrar

    texto\_plano = "KeepCoding te enseña a codificar y a cifrar."

    # Cifrar datos

    mensaje\_cifrado = cifrar\_datos(texto\_plano, clave, datos\_asociados)

    print("Mensaje cifrado (JSON):", mensaje\_cifrado)

    # Descifrar datos

    texto\_descifrado = descifrar\_datos(mensaje\_cifrado, clave)

    print("Texto descifrado:", texto\_descifrado)

except (ValueError, KeyError) as error:

    print("Error:", error)

4 –

**Verificacion de PyJWT**

import jwt

# Clave secreta original

clave = "Con KeepCoding aprendemos"

# JWT original

jwt\_original = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwiaWF0IjoxNjY3OTMzNTMzfQ.gfhw0dDxp6oixMLXXRP97W4TDTrv0y7B5YjD0U8ixrE"

# JWT manipulado

jwt\_manipulado = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzQWRtaW4iLCJpYXQiOjE2Njc5MzM1MzN9.krgBkzCBQ5WZ8JnZHuRvmnAZdg4ZMeRNv2CIAODlHRI"

try:

# Validación del JWT manipulado

datos = jwt.decode(jwt\_manipulado, clave, algorithms=["HS256"])

print("JWT válido:", datos)

except jwt.InvalidSignatureError:

print("La firma no coincide. ¡Token manipulado!")

except Exception as e:

print("Error al validar el token:", e)

5 –

import sys

sys.stdout.reconfigure(encoding='utf-8')

import hashlib

m1 = hashlib.sha3\_256()

texto\_hashear\_bytes=bytes("En KeepCoding aprendemos cómo protegernos con criptografía.", "utf8")

m1.update(texto\_hashear\_bytes)

print("sha3-256: " + m1.digest().hex())

6 –

from Crypto.Hash import HMAC, SHA256

# Generamos MAC

key\_hex = "A212A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB"

key\_bytes = bytes.fromhex(key\_hex)

# Mensaje a hashear

mensaje = "Siempre existe más de una forma de hacerlo, y más de una solución válida."

texto\_hashear\_bytes = mensaje.encode("utf-8")  # Codificamos directamente

# Generamos el HMAC con SHA-256

hmac256 = HMAC.new(key\_bytes, msg=texto\_hashear\_bytes, digestmod=SHA256)

hmac\_result = hmac256.hexdigest()  # Resultado hexadecimal del HMAC

# Mostramos el HMAC generado

print("HMAC generado:", hmac\_result)

# Verificamos el HMAC

try:

    hmac256\_verificar = HMAC.new(key\_bytes, msg=texto\_hashear\_bytes, digestmod=SHA256)

    hmac256\_verificar.hexverify(hmac\_result)  # Verificamos

    print("Verificación: OK")  # Si no lanza excepción, está todo bien

except ValueError:

    print("Verificación: KO")  # Si hay excepción, el HMAC no coincide

9 –

from Crypto.Cipher import AES

from hashlib import sha256

# Clave AES proporcionada (hexadecimal)

clave\_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"

# Convertimos la clave a bytes

clave\_bytes = bytes.fromhex(clave\_hex)

# ---- KCV(SHA-256) ----

# Calculamos el hash SHA-256 de la clave

hash\_sha256 = sha256(clave\_bytes).digest()

# Los 3 primeros bytes son el KCV(SHA-256)

kcv\_sha256 = hash\_sha256[:3]

# ---- KCV(AES) ----

# Creamos un bloque de ceros (16 bytes) y el IV de ceros

bloque\_ceros = bytes(16)  # 16 bytes llenos de 0x00

# Configuramos AES en modo ECB

cipher = AES.new(clave\_bytes, AES.MODE\_ECB)

# Ciframos el bloque de ceros

bloque\_cifrado = cipher.encrypt(bloque\_ceros)

# Los 3 primeros bytes del bloque cifrado son el KCV(AES)

kcv\_aes = bloque\_cifrado[:3]

# ---- Resultados ----

print("KCV(SHA-256):", kcv\_sha256.hex().upper())

print("KCV(AES):", kcv\_aes.hex().upper())

10 –

C:\Users\nj\_al\ejercicio10>gpg --armor --sign --default-key "RRHH" mensaje\_a\_firmar.txt

gpg: usando "RRHH" como clave secreta predeterminada para firmar

C:\Users\nj\_al\ejercicio10>gpg --armor --encrypt --recipient "RRHH" --recipient "Pedro" mensaje\_firm\_PRRHH.txt

gpg: 25D6D0294035B650: No hay seguridad de que esta clave pertenezca realmente

al usuario que se nombra

sub cv25519/25D6D0294035B650 2022-06-26 Pedro Pedrito Pedro <pedro.pedrito.pedro@empresa.com>

Huella clave primaria: 1BDE 635E 4EAE 6E68 DFAD 2F7C D730 BE19 6E46 6101

Huella de subclave: 8E8C 6669 AC44 3271 42BC C244 25D6 D029 4035 B650

No es seguro que la clave pertenezca a la persona que se nombra en el

identificador de usuario. Si \*realmente\* sabe lo que está haciendo,

puede contestar sí a la siguiente pregunta.

¿Usar esta clave de todas formas? (s/N) s

gpg: 7C1A46EA20B0546F: No hay seguridad de que esta clave pertenezca realmente

al usuario que se nombra

sub cv25519/7C1A46EA20B0546F 2022-06-26 RRHH <RRHH@RRHH>

Huella clave primaria: F2B1 D0E8 958D F2D3 BDB6 A105 3869 803C 684D 287B

Huella de subclave: 811D 89A3 6199 A7C9 0BFE 69D6 7C1A 46EA 20B0 546F

No es seguro que la clave pertenezca a la persona que se nombra en el

identificador de usuario. Si \*realmente\* sabe lo que está haciendo,

puede contestar sí a la siguiente pregunta.

¿Usar esta clave de todas formas? (s/N) s

11 –

import os

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.Hash import SHA256

# Define el directorio base donde se encuentran las claves

base\_dir = "C:\\Users\\nj\_al\\OneDrive\\Documentos\\Criptografia\\Proyecto\\Ejercicio 11"

# Rutas de los archivos de claves

private\_key\_path = os.path.join(base\_dir, "clave-rsa-oaep-priv.pem")

public\_key\_path = os.path.join(base\_dir, "clave-rsa-oaep-publ.pem")

# Lee las claves desde los archivos

with open(private\_key\_path, "rb") as f\_privada:

    keyPair = RSA.import\_key(f\_privada.read())

with open(public\_key\_path, "rb") as f\_publica:

    pubKey = RSA.import\_key(f\_publica.read())

# Mensaje cifrado en hexadecimal

mensaje = bytes.fromhex(

    ""

)

# Descifra el mensaje usando la clave privada

decryptor = PKCS1\_OAEP.new(keyPair, hashAlgo=SHA256)

text\_en\_claro = decryptor.decrypt(mensaje)

print("Mensaje descifrado:", text\_en\_claro.hex())

# Cifra nuevamente el mensaje usando la clave pública

cipher = PKCS1\_OAEP.new(pubKey, hashAlgo=SHA256)

text\_cifrado = cipher.encrypt(text\_en\_claro)

print("Mensaje cifrado nuevamente:", text\_cifrado.hex())

13 –

import sys

sys.stdout.reconfigure(encoding='utf-8')

import json

from base64 import b64encode, b64decode

from Crypto.Cipher import AES

# Cifrado

texto\_a\_cifrar\_bytes = bytes('He descubierto el error y no volveré a hacerlo mal', 'UTF-8')

clave = bytes.fromhex('E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74')

# Nonce

nonce = b'9Yccn/f5nJJh'

# Inicializar el cifrador en modo GCM

cipher = AES.new(clave, AES.MODE\_GCM, nonce=nonce)

# Cifrar y generar el tag

texto\_cifrado\_bytes, tag = cipher.encrypt\_and\_digest(texto\_a\_cifrar\_bytes)

# Codificar todo en Base64 para guardarlo en JSON

nonce\_b64 = b64encode(nonce).decode('utf-8')

tag\_b64 = b64encode(tag).decode('utf-8')

texto\_cifrado\_b64 = b64encode(texto\_cifrado\_bytes).decode('utf-8')

# Crear el mensaje en formato JSON

mensaje\_json = json.dumps({

    'nonce': nonce\_b64,

    'tag': tag\_b64,

    'texto\_cifrado': texto\_cifrado\_b64

})

print("Mensaje JSON:", mensaje\_json)

# Descifrado

try:

    # Cargar y decodificar el JSON

    b64 = json.loads(mensaje\_json)

    nonce\_desc\_bytes = b64decode(b64['nonce'])

    texto\_cifrado\_bytes = b64decode(b64['texto\_cifrado'])

    tag\_desc\_bytes = b64decode(b64['tag'])

    # Inicializar el descifrador en modo GCM

    cipher = AES.new(clave, AES.MODE\_GCM, nonce=nonce\_desc\_bytes)

    # Descifrar y verificar el mensaje

    mensaje\_des\_bytes = cipher.decrypt\_and\_verify(texto\_cifrado\_bytes, tag\_desc\_bytes)

    print("El texto en claro es:", mensaje\_des\_bytes.decode("utf-8"))

except (ValueError, KeyError) as error:

    print('Problemas para descifrar...')

    print("El motivo del error es:", error)

import ed25519

# Base directory donde están las claves

base\_dir = "C:\\Users\\nj\_al\\OneDrive\\Documentos\\Criptografia\\Proyecto\\Ejercicio 13"

try:

    # Leer clave privada en binario crudo (32 bytes)

    with open(f"{base\_dir}\\ed25519-priv", "rb") as f:

        privatekey = f.read()[:32]  # Usar los primeros 32 bytes

    # Generar la clave de firma y derivar la clave pública

    signedKey = ed25519.SigningKey(privatekey)

    publickey = signedKey.get\_verifying\_key()  # Derivar clave pública desde la privada

    # Mensaje a firmar

    msg = bytes('El equipo está preparado para seguir con el proceso, necesitaremos más recursos.' , 'utf-8')

    # Firmar el mensaje

    signature = signedKey.sign(msg, encoding='hex')

    print("Firma Generada (64 bytes):", signature)

    # Verificar la firma usando la clave pública derivada

    publickey.verify(signature, msg, encoding='hex')

    print("La firma es válida")

except Exception as e:

    print(f"Error al procesar: {e}")

14 –

from cryptography.hazmat.primitives.kdf.hkdf import HKDF

from cryptography.hazmat.primitives.hashes import SHA512

from cryptography.hazmat.backends import default\_backend

import os

# Clave maestra y identificador de dispositivo en formato hexadecimal

clave\_maestra\_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"

identificador\_dispositivo\_hex = "e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3"

# Convertir a bytes

clave\_maestra = bytes.fromhex(clave\_maestra\_hex)

identificador\_dispositivo = bytes.fromhex(identificador\_dispositivo\_hex)

# Generar un salt aleatorio (16 bytes)

salt = os.urandom(16)

# Generar clave derivada con HKDF

hkdf = HKDF(

    algorithm=SHA512(),

    length=32,  # Longitud de la clave AES (256 bits = 32 bytes)

    salt=salt,  # Salt aleatorio

    info=identificador\_dispositivo,  # Información adicional (identificador del dispositivo)

    backend=default\_backend()

)

clave\_aes = hkdf.derive(clave\_maestra)

# Mostrar resultados

print("Salt utilizado (hex):", salt.hex())

print("Clave AES derivada (hex):", clave\_aes.hex())

15 –

from psec import tr31

header, key = tr31.unwrap( kbpk=bytes.fromhex("A1A10101010101010101010101010102"), key\_block="D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E03CD857FD37018E111B")

print("Clave=", key.hex())

print("Key Version ID: " + header.version\_id )

print("Algoritmo: " + header.algorithm)

print("Modo de uso: " + header.mode\_of\_use)

print("Uso de la clave: " + header.key\_usage)

print("Exportabilidad: " + header.exportability)