PRÁCTICA: CRIPTOGRAFÍA

Juan Gabriel Moviglia

Respuesta Ejercicio 1:

clave_fija XOR clave_properties =clave_final

entonces hay que hacer clave_fija XOR clave_final = clave_properties

criptografia > # EJERCICIO 1 - CASO DESARROLLO.py > ...

```python
# Resolución del ejercicio de disociación de claves usando XOR

# Clave fija que está en el código
clave_fija = bytes.fromhex("B1EF2ACFE2BAEEFF")

# --- CASO 1 (Desarrollo): Encontrar la clave del properties ---
clave_final = bytes.fromhex("91BA13BA21AABB12")
clave_properties = bytes(a ^ b for a, b in zip(clave_fija, clave_final))
print("1. Clave en properties:", clave_properties.hex().upper())

# --- CASO 2 (Producción): Calcular la clave final en memoria ---
clave_properties_prod = bytes.fromhex("B98A15BA31AEBB3F")
clave_final_produccion = bytes(a ^ b for a, b in zip(clave_fija, clave_properties_prod))
print("2. Clave final en memoria:", clave_final_produccion.hex().upper())
```

1. Clave en properties (desarrollo): 20553975C310451D
2. Clave final (producción): 08653F75D3145530

¿Cómo funciona?

1.bytes.fromhex() convierte una cadena hexadecimal (como "B1EF2ACF...") en datos binarios.

2.zip() toma dos listas de bytes y las empareja para operar con ellas juntas.

3.a ^ b aplica la operación XOR entre cada par de bytes emparejados.

4.hex().upper() convierte el resultado de vuelta a una cadena hexadecimal legible en mayúsculas.

Respuesta Ejercicio 2:

criptografia > # EJERCICIO 1 - CASO DESARROLLO.py > ...

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import binascii

# Datos del ejercicio (proporcionados en hexadecimal)
clave_hex = "E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
iv_hex = "00" * 16
cifrado_hex = "979DF30474898787A45605CCB9B936D33B780D03CABC81719D52383480DC3120"

# Conversión a bytes
clave = binascii.unhexlify(clave_hex)
iv = binascii.unhexlify(iv_hex)
cifrado = binascii.unhexlify(cifrado_hex)

# 1. Descifrado con AES-256 CBC y PKCS7
cipher = AES.new(clave, AES.MODE_CBC, iv)
descifrado = cipher.decrypt(cifrado)
texto_claro = unpad(descifrado, AES.block_size, style='pkcs7')

print("1. Texto claro (hex):", texto_claro.hex())
print("2. Padding añadido originalmente:", descifrado[-1], "bytes")

# 2. Intentar con padding X.923 (ANSI X.923)
try:
    texto_x923 = unpad(descifrado, AES.block_size, style='x923')
    print("3. Con X.923 funciona:", texto_x923.hex())
except ValueError as e:
    print("3. X.923 falla:", e)
```

1. Texto claro (hex): 50616464696e67206f6e20626c6f636b2063697068657273
2. Padding añadido originalmente: 16 bytes
3. X.923 falla: omo el padding real es PKCS7, intentar removerlo como X.923 genera error. Esto demuestra que el estándar de padding debe coincidir en cifrado y descifrado.

Respuesta Ejercicio 3:

```python
import base64
import hashlib
import hmac
from Crypto.Cipher import ChaCha20

# 1. DATOS PROPORCIONADOS
texto_plano = "KeepCoding te enseña a codificar y a cifrar"
texto_claro = texto_plano.encode('utf-8')  # Codificar a UTF-8

clave_hex = "E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
clave = bytes.fromhex(clave_hex)

nonce_b64 = "9Yccn/f5nJJhAt2S"
nonce = base64.b64decode(nonce_b64)

# 2. CIFRADO CHACHA20
cipher = ChaCha20.new(key=clave, nonce=nonce)
texto_cifrado = cipher.encrypt(texto_claro)

print("Texto cifrado (hex):", texto_cifrado.hex())
print("Nonce (hex):", nonce.hex())

# 3. MEJORA: INTEGRIDAD CON HMAC-SHA256
clave_hmac = hashlib.sha256(clave + b"INTEGRIDAD").digest()
hmac_calculado = hmac.new(clave_hmac, texto_cifrado, hashlib.sha256).digest()

print("HMAC (hex):", hmac_calculado.hex())
print("Clave HMAC derivada (hex):", clave_hmac.hex())

# 4. VERIFICACIÓN
hmac_verificar = hmac.new(clave_hmac, texto_cifrado, hashlib.sha256).digest()
if hmac.compare_digest(hmac_calculado, hmac_verificar):
    print("✅ HMAC válido. Integridad confirmada.")
    cipher_dec = ChaCha20.new(key=clave, nonce=nonce)
    texto_recuperado = cipher_dec.decrypt(texto_cifrado)
    print("✅ Texto recuperado:", texto_recuperado.decode('utf-8'))
else:
    print("❌ HMAC inválido. Mensaje alterado.")
```

Texto cifrado
(hex):3b20ea2bf9449c6acb9704573a6a6bf310eaf2c2787001d29a623a94d5e7ed718de7c9d69789b
ac83eb7b0c2
Nonce (hex): f5871c9ff7f99c926102dd92
HMAC (hex): ca3e1c11bee749137743c07aa2d73c811a5a9d25fa665249d29009bff9185ca9
Clave HMAC derivada (hex):
2154e314cf42b674c7806cd8ffdb7573be81335e618fbe73a159e7887ad91da9

Propuesta de mejora (Integridad):
Esquema: Cifrar con ChaCha20, luego aplicar HMAC-SHA256 al texto cifrado.
El HMAC detecta cualquier alteración del cifrado.

Respuesta Ejercicio 4: Tomamos la primera parte (header) del primer JWT:

```python
import base64
header = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9"
# Añadir padding si es necesario y decodificar
decoded = base64.urlsafe_b64decode(header + "==").decode('utf-8')
print(decoded)  # Resultado: {"typ":"JWT","alg":"HS256"}
```

Respuesta: El campo "alg":"HS256" indica que es HMAC con SHA-256 (un algoritmo simétrico que requiere una clave secreta).

Descodificamos la segunda parte (payload) del primer JWT:

```python
import base64
payload_b64 = "eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3MgcGFsb3RlcyIsInJvbCI6ImlzTm9ybWFsIiwia
decoded_payload = base64.urlsafe_b64decode(payload_b64 + "==").decode('utf-8')
print(decoded_payload)
```

Respuesta : "usuario": "Don Pepito de los palotes", "rol": "isNormal", "iat": 1667933533

¿Qué está intentando realizar el hacker?

Compara los payloads:
Original: "rol":"isNormal"
Hacker: "rol":"isAdmin" (en su JWT modificado).
Conclusión: Intenta elevar privilegios (pasando de usuario normal a administrador).

Validación con pyjwt: Falla (InvalidSignatureError), porque la firma no coincide con la clave secreta.

```python
import jwt
clave = "Con KeepCoding aprendemos"
jwt_hacker = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c3VhcmlvIjoiRG9uIFBlcGl0byBkZSBsb3Mgc
try:
    payload = jwt.decode(jwt_hacker, clave, algorithms=["HS256"])
    print("Válido:", payload)
except jwt.InvalidSignatureError:
    print("Firma inválida. El hacker no pudo falsificar la firma.")
```

Respuesta: Firma invalida. EL hacker no pudo falsificar la firma.

Respuesta Ejercicio 5:

1. Tipo de SHA3 Keccak generado (primer hash):SHA3-256 (64 caracteres hex = 256 bits)
2. Tipo de hash SHA2 (segundo resultado):SHA-512 (128 caracteres hex = 512 bits)

```python
import hashlib
# Hash SHA3-256 sin punto
texto1 = "En KeepCoding aprendemos cómo protegernos con criptografía"
hash1 = hashlib.sha3_256(texto1.encode()).hexdigest()
print("SHA3-256 sin punto:", hash1)
# Hash SHA3-256 con punto
texto2 = texto1 + "."
hash2 = hashlib.sha3_256(texto2.encode()).hexdigest()
print("SHA3-256 con punto:", hash2)
# Efecto avalancha: comparar primeros 8 caracteres
print("¿Primeros 8 chars iguales?", hash1[:8] == hash2[:8])
```

Respuesta:
SHA3-256 sin punto: bced1be95fbd85d2ffcce9c85434d79aa26f24ce82fbd4439517ea3f072d56fe
SHA3-256 con punto: 302be507113222694d8c63f9813727a85fef61a152176ca90edf1cfb952b19bf
¿Primeros 8 chars iguales? False
(añadir .) genera un hash completamente diferente.

Respuesta Ejercicio 6:

```python
import hmac, hashlib, binascii

# 1. CLAVE y TEXTO
clave_hex = "2712A51C997E14B4DF08D55967641B0677CA31E049E672A4B06861AA4D5826EB"
texto = "Siempre existe más de una forma de hacerlo, y más de una solución válida."

# 2. CONVERTIR Y CALCULAR
clave_bytes = binascii.unhexlify(clave_hex)
texto_bytes = texto.encode('utf-8')
hmac_resultado = hmac.new(clave_bytes, texto_bytes, hashlib.sha256).hexdigest().upper()

# 3. RESULTADO
print("HMAC-SHA256 (hex):", hmac_resultado)
```

Respuesta: HMAC-SHA256 (hex): D0B686743822793FB185263F1FA4DED09BD557BCD341AC53E06DAD76238C8E09

Respuesta Ejercicio 7:

¿Por qué SHA-1 es una mala opción para almacenar contraseñas?

La propuesta de usar SHA-1 es un riesgo de seguridad alto. Aunque fue un algoritmo válido en su día, hoy está criptográficamente roto: se han demostrado vulnerabilidades que permiten a un atacante generar colisiones, es decir, encontrar una contraseña diferente que produzca el mismo hash. Además, y esto es clave para contraseñas, es demasiado rápido. Los algoritmos hash genéricos como SHA-1 están diseñados para ser eficientes, lo que permite a un atacante con hardware moderno (como GPUs) probar miles de millones de contraseñas por segundo en un ataque de fuerza bruta. Por estos motivos, su uso está absolutamente desaconsejado.

¿Cómo fortaleceríamos una propuesta basada en SHA-256?

Mejorar una propuesta de SHA-256 es imprescindible. El paso crítico que falta es la “Salt”. Esto consiste en generar una cadena de bytes aleatoria única (la “sal”) para cada usuario, combinarla con su contraseña, y luego aplicar el SHA-256. Tanto el hash resultante como la sal se almacenan.

Esto soluciona dos problemas principales: primero, neutraliza los ataques con 'tablas arcoíris', que son bases de datos precomputadas de hashes para contraseñas comunes; y segundo, evita que dos usuarios con la misma contraseña tengan el mismo hash en la base de datos. Sin embargo, aunque el “Salt” es obligatorio, SHA-256 sigue siendo un algoritmo rápido, por lo que aún hay margen para una capa de protección más robusta.

¿Qué mejora final propondrías?

La mejora definitiva es abandonar el uso de funciones hash criptográficas genéricas (como SHA-256) y adoptar funciones de derivación de clave o 'hash' específicas para contraseñas. Las más recomendadas son bcrypt, scrypt o Argon2 (este último es ganador del 'Password Hashing Competition' y se considera el estado del arte).

Estos algoritmos están diseñados con un factor de trabajo o costo ajustable, lo que los hace intencionadamente lentos y costosos en recursos (como memoria). Esto significa que, mientras la verificación legítima de un login toma una fracción de segundo tolerable para el usuario, para un atacante que intenta probar miles de millones de contraseñas, el proceso se vuelve computacionalmente inviable. Esta es la mejor práctica actual para el almacenamiento seguro de contraseñas.

Respuesta Ejercicio 8:

Para redefinir la API garantizando confidencialidad e integridad sin TLS, propondría un sistema de cifrado híbrido:

Para el intercambio seguro inicial y la autenticación, usaría criptografía asimétrica (RSA). El cliente cifra una clave de sesión AES con la clave pública del servidor, asegurando que solo el servidor legítimo puede acceder a ella.

Para proteger los datos confidenciales en cada mensaje (como el número de tarjeta), usaría criptografía simétrica (AES-256 en modo GCM). Este modo proporciona tanto cifrado confidencial como una etiqueta de integridad, asegurando que los datos no sean alterados.

Como mejora adicional, se podrían implementar firmas digitales RSA para que el servidor pueda verificar de manera irrefutable la identidad del cliente que envió cada petición.

Respuesta Ejercicio 9:

```python
import hashlib, binascii
from Crypto.Cipher import AES

clave_hex = "A2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"
clave_bytes = binascii.unhexlify(clave_hex)

# 1. Calcular SHA-256 de la clave
hash_sha256 = hashlib.sha256(clave_bytes).digest()
# 2. KCV(SHA-256): primeros 3 bytes (6 caracteres hex)
kcv_sha256 = hash_sha256[:3]
print("KCV(SHA-256):", binascii.hexlify(kcv_sha256).decode().upper())

# 3. Preparar bloque de 16 bytes de ceros y IV de ceros
bloque_ceros = b'\x00' * 16
iv_ceros = b'\x00' * 16

# 4. Cifrar en modo CBC
cipher = AES.new(clave_bytes, AES.MODE_CBC, iv_ceros)
texto_cifrado = cipher.encrypt(bloque_ceros)

# 5. KCV(AES): primeros 3 bytes del resultado cifrado
kcv_aes = texto_cifrado[:3]
print("KCV(AES):", binascii.hexlify(kcv_aes).decode().upper())
```

Respuestas:

KCV(SHA-256): DB7DF2

KCV(AES): 5244DB

Respuesta Ejercicio 11:

```python
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend
import binascii

texto_cifrado_hex = ("b72e6fd48155f565dd2684df3ffa8746d649b11f0ed4637fc4c99d18283b32e1709b30c"
                     "96b4a8a20d5dbc639e9d83a53681e6d96f76a0e4c279f0dffa76a329d04e3d3d4ad629"
                     "793eb00cc76d10fc00475eb76bfbc1273303882609957c4c0ae2c4f5ba670a4126f2f14"
                     "a9f4b6f41aa2edba01b4bd586624659fca82f5b4970186502de8624071be78ccef573d"
                     "896b8eac86f5d43ca7b10b59be4acf8f8e0498a455da04f67d3f98b4cd907f27639f4b1"
                     "df3c50e05d5bf63768088226e2a9177485c54f72407fdf358fe64479677d8296ad38c6f"
                     "177ea7cb74927651cf24b01dee27895d4f05fb5c161957845cd1b5848ed64ed3b0372"
                     "2b21a526a6e447cb8ee")
texto_cifrado_bytes = binascii.unhexlify(texto_cifrado_hex)

with open('clave-rsa-oaep-priv.pem', 'rb') as f:
    priv_key = serialization.load_pem_private_key(f.read(), password=None, backend=default_backend())

clave_simetrica = priv_key.decrypt(texto_cifrado_bytes, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
print("Clave simétrica recuperada (hex):", clave_simetrica.hex())

with open('clave-rsa-oaep-publ.pem', 'rb') as f:
    pub_key = serialization.load_pem_public_key(f.read(), backend=default_backend())

nuevo_cifrado = pub_key.encrypt(clave_simetrica, padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
print("Nuevo texto cifrado (hex):", nuevo_cifrado.hex())
```

Respuesta:

Clave sim  trica recuperada (hex): e2cff885901a5449e9c448ba5b948a8c4ee377152b3f1acfa0148fb3a426db72

Nuevo texto cifrado (hex):
28fc1e1b3dac1ccfb96c6f29fd9060204aa565cbd38c3113fe449aac21b94364ff247a736663b450f637d836d05f82f29d
54b29248311100dfb574aeb0ac3763488b69b6ede4c70770433686b71c73bbdcd1d2d9f062808f375e58392bd40ff15
d4277e1895d68298cd567c38e4b3971fb11a4acfac7fbcbe6bebe041c3d915458ddbc6ba8ac4622414e2f0110caf9e13f
09720c206b0d3a41abac16d2bceb97447a1439274f552844070bc2bc146673432be514bcf94befdb3811b58fd6dfef99
5a307d346c76894d6d284f6c9c863ed237c46dc20c0f673f7b467c2afb4118928494d856fab34aaf6d7246c5d89e61c8
b5234fc0c44939b80dac53f5c9e337

Respuesta Ejercicio 10:

```python
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import padding, rsa
from cryptography.hazmat.backends import default_backend

with open('MensajeRespoDeRaulARRHH.txt', 'r', encoding='utf-8') as f:
    print("MENSAJE DE PEDRO:\n" + f.read())

mensaje = "Viendo su perfil en el mercado, hemos decidido ascenderle y mejorarle un 25% su salario.\nSaludos."
with open('clave-rsa-oaep-priv.pem', 'rb') as f:
    priv_key = serialization.load_pem_private_key(f.read(), password=None, backend=default_backend())
firma = priv_key.sign(mensaje.encode('utf-8'), padding.PSS(mgf=padding.MGF1(hashes.SHA256()), salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
with open('Respuesta_RRHH.sig', 'wb') as f:
    f.write(firma)
print("Firma guardada: Respuesta_RRHH.sig")

mensaje_secreto = b"Estamos todos de acuerdo, el ascenso sera el mes que viene, agosto, si no hay sorpresas."
with open('clave-rsa-oaep-publ.pem', 'rb') as f:
    pub_key = serialization.load_pem_public_key(f.read(), backend=default_backend())
cifrado_rrhh = pub_key.encrypt(mensaje_secreto, padding.OAEP(mgf=padding.MGF1(hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
with open('Cifrado_RRHH.bin', 'wb') as f:
    f.write(cifrado_rrhh)
print("Cifrado RRHH guardado: Cifrado_RRHH.bin")

pedro_key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
cifrado_pedro = pedro_key.public_key().encrypt(mensaje_secreto, padding.OAEP(mgf=padding.MGF1(hashes.SHA256()), algorithm=hashes.SHA256(), label=None))
with open('Cifrado_Pedro.bin', 'wb') as f:
    f.write(cifrado_pedro)
print("Cifrado Pedro guardado: Cifrado_Pedro.bin")
```

Respuesta: “MENSAJE DE PEDRO”:

Se debe ascender inmediatamente a Ra  l. Es necesario mejorarle sus condiciones econmicas un 20% para que se quede con nosotros.
Firma guardada: Respuesta_RRHH.sig

Cifrado RRHH guardado: Cifrado_RRHH.bin
Cifrado Pedro guardado: Cifrado_Pedro.bin

Respuesta Ejercicio 12:

```python
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
import base64, binascii

clave_hex = "E2CFF885901B3449E9C448BA5B948A8C4EE322152B3F1ACFA0148FB3A426DB74"
nonce_b64 = "9Yccn/f5nJJhAt2S"
texto_plano = b"He descubierto el error y no volvere a hacerlo mal"

clave = binascii.unhexlify(clave_hex)
nonce = base64.b64decode(nonce_b64)

aesgcm = AESGCM(clave)
texto_cifrado_completo = aesgcm.encrypt(nonce, texto_plano, None)

print("Texto cifrado completo (cifrado + tag de autenticacion):")
print("Hexadecimal:", texto_cifrado_completo.hex())
print("Base64:", base64.b64encode(texto_cifrado_completo).decode())
```

Respuesta: Texto cifrado completo (cifrado + tag de autenticacion):
Hexadecimal:
5dcbb6261d0fba29ce39431e9a013b34cbca2a4e04bb2d90149d61f4afd04d65e2abdd3b0dfae7a38a050f484e7bb4b11a28fc9001cb6c1ad290687f473367a17623
Base64: Xcu2Jh0PuinOOUMemgE7NMvKKk4Euy2QFJ1h9K/QTWXiq907Dfrno4oFD0hOe7SxGij8kAHLbBrSkGh/RzNnoXYj

En el ejercicio se identificó un error crítico de seguridad: la reutilización del mismo Nonce (IV) con la misma clave en el algoritmo AES-GCM. Esto anula la seguridad del modo, permitiendo potencialmente la falsificación de mensajes y el descifrado de datos.

A pesar de este error, como solicita el enunciado, se procedió a cifrar el texto "He descubierto el error y no volveré a hacerlo mal" utilizando la clave AES-256 y el nonce proporcionados (decodificando la clave desde hexadecimal y el nonce desde Base64). El cifrado con AES-GCM genera un resultado que combina el texto cifrado y un tag de autenticación.

Respuesta Ejercicio 13:

```python
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import padding, ed25519
from cryptography.hazmat.backends import default_backend
import base64

msg = b"El equipo esta preparado para seguir con el proceso, necesitaremos mas

# 1. Firma RSA (ya funciona)
with open('clave-rsa-oaep-priv.pem', 'rb') as f:
    rsa_key = serialization.load_pem_private_key(f.read(), password=None, backe
print("Firma RSA (hex):", rsa_key.sign(msg, padding.PKCS1v15(), hashes.SHA256()

# 2. Firma Ed25519 (con detección de formato)
with open('ed25519-priv', 'rb') as f:
    data = f.read()

# Intenta descifrar el formato
if b'-----BEGIN' in data:  # Es formato PEM
    # Extrae la parte base64 entre los encabezados
    lines = data.decode().split('\n')
    b64_data = ''.join([l for l in lines if l and not l.startswith('-----')])
    key_bytes = base64.b64decode(b64_data)
else:
    key_bytes = data  # Podría ser raw o base64

# Si después de procesar no son 32 bytes, intenta base64 decode
if len(key_bytes) != 32:
    try:
        key_bytes = base64.b64decode(key_bytes)
    except:
        pass

# Toma los primeros 32 bytes (la clave privada Ed25519)
priv_bytes = key_bytes[:32] if len(key_bytes) >= 32 else key_bytes
print(f"Clave privada leída: {len(priv_bytes)} bytes")

if len(priv_bytes) == 32:
    ed_key = ed25519.Ed25519PrivateKey.from_private_bytes(priv_bytes)
    print("Firma Ed25519 (hex):", ed_key.sign(msg).hex())
else:
    print(f"ERROR: Se necesitan 32 bytes, se obtuvieron {len(priv_bytes)}")
```

Respuesta:

Firma RSA (hex):

25afd3ee13d79adfef36a04eaf758efd1a0ad6306cf696c3bd57e7cd6af170d94dd1900d0622b5fce3adaad3304b356a2969327b35a6
ee0d89dcc4f6d8cca34b6b7caf27bca9b6bf91cddda693514f647300148353e0be3e0cf7c7e93bb90d0661494d7eb8bd506287ad316
b932dcf5a92a77e5667d58cad19a363b81497b45785e77e9c065e64e9c3d71fdd917edca1571f4e366ce31f64949c07b5fb73dc71ca8
28bd081059f15f410ab6ffc1b328b8a6514c8a978a2703cb5d84c892768ce2399ed261f5da4ef93771f37b0be0ad5c45d63ff06bc25c2
3feeea8054203058e2998f5561b762c4d404a1441def543de8933b7d89f4e721bf42490425b7b145

Clave privada le  da: 32 bytes

Firma Ed25519 (hex):

cbd641e969c88926c8aef44a9c5251c5f631725f9e7e2e1ffae51a65f22558dc4ab7bd5faa05a7a57804aead7488f48f85ce29f66cd4dc
51969ba06a61ce9903

1. Firma con RSA PKCS#1 v1.5

Qué hice: Usé la clave privada RSA contenida en el archivo clave-rsa-oaep-priv.pem.

Cómo lo hice: Apliqué el esquema de firma estándar PKCS#1 v1.5 al mensaje, utilizando la función hash SHA-256 para primero obtener un resumen único del texto.

Resultado: Obtuve un valor hexadecimal largo (512 caracteres), que es la firma digital. Esta firma solo puede ser generada con mi clave privada, pero cualquiera puede verificarla usando mi clave pública correspondiente.

2. Firma con la curva elíptica Ed25519:

Qué hice: Usé la clave privada Ed25519 contenida en el archivo ed25519-priv.

Cómo lo hice: El algoritmo Ed25519, basado en curvas elípticas, firma el mensaje directamente de una manera muy eficiente y segura.

Resultado: Obtuve un valor hexadecimal más corto (128 caracteres). Las firmas Ed25519 son conocidas por ser compactas, rápidas y ofrecer un alto nivel de seguridad.

Respuesta Ejercicio 14:

```python
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

MASTER_KEY_HEX = "E2CFF885901A5449E9C448BA5B948A8C4EE377152B3F1ACFA0148FB3A426DB72"

DEVICE_ID_HEX = "e43bb4067cbcfab3bec54437b84bef4623e345682d89de9948fbb0afedc461a3"

# Convertir hexadecimal a bytes
master_key = bytes.fromhex(MASTER_KEY_HEX)
device_id = bytes.fromhex(DEVICE_ID_HEX)

# Derivar clave AES-256 con HKDF-SHA512, sin salt, usando device_id como info
hkdf = HKDF(
    algorithm=hashes.SHA512(),
    length=32,
    salt=None,
    info=device_id,
)
derived_key = hkdf.derive(master_key)

print("Clave derivada (hex):", derived_key.hex())
```

Respuesta:

Clave derivada (hex): 39e449691537f307c875a2b3aeb01fd27803566bb4b8320c133b391dcdf6dc3e

Respuesta Ejercicio 15:

¿Con qué algoritmo se ha protegido el bloque de clave?

```python
bloque_tr31_hex = ("D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDB"
                   "E6A5626F79FA7B4071E9EE1423C6D7970FA2B965D18B23922B5B2E5657495E0"
                   "3CD857FD37018E111B")

cabecera = bloque_tr31_hex[:16]  # Primeros 8 bytes en hexadecimal
identificador_version = cabecera[0]  # Primer byte de la cabecera

mapa_algoritmo = {
    'B': 'TDEA (Triple DES)',
    'D': 'AES (con CMAC)',
}

algoritmo = mapa_algoritmo.get(identificador_version, f'Desconocido (Identificador: {identificador_version})')

print(f"Cabecera (hex): {cabecera}")
print(f"Identificador de Versión: {identificador_version}")
print(f"Algoritmo de protección: {algoritmo}")
```

Respuesta:

Cabecera (hex): D0144D0AB00S0000

Identificador de VersiOn: D

Algoritmo de protecciOn: AES (con CMAC)

¿Para qué algoritmo se ha definido la clave?

La clave se ha definido para el algoritmo Triple DES (3DES) de tres claves.

¿Para qué modo de uso se ha generado?

El modo de uso indicado por el código 4 es Verificación de PIN

¿Es exportable?

La exportabilidad de una clave en un bloque TR-31 se determina por el quinto carácter de la cabecera (posición 4, índice base 0), que es el campo Exportability. El valor 4 en el campo de exportabilidad indica que la clave es "Exportable bajo restricciones" o "Exportable solo bajo autorización". Esto significa que la clave puede ser exportada fuera del entorno seguro original, pero solo bajo condiciones

¿Para qué se puede usar la clave?

Esta clave está autorizada para ser utilizada exclusivamente en operaciones de verificación de PIN utilizando el algoritmo Triple DES (3DES) de tres claves.

¿Qué valor tiene la clave?

Respuesta: `0123456789ABCDEFFEDCBA9876543210`

```python
from cryptography.hazmat.primitives.keywrap import aes_key_unwrap
from cryptography.hazmat.backends import default_backend
import binascii

# Clave de transporte para desenvolver
clave_transporte = binascii.unhexlify('A1A10101010101010101010101010102')

# Bloque TR31 (corregir caracter 'S' a '5')
bloque_hex = 'D0144D0AB00S000042766B9265B2DF93AE6E29B58135B77A2F616C8D515ACDBE6A5626F79FA7B4071E9EE1423C6D7970FA2E
bloque_corregido = bloque_hex.replace('S', '5')

# Extraer parte cifrada (ignorar encabezado 16 chars y MAC 8 chars)
parte_cifrada = binascii.unhexlify(bloque_corregido[16:-8])

# Desenvolver clave
clave_desenvuelta = aes_key_unwrap(clave_transporte, parte_cifrada, default_backend())

# Primer byte indica longitud real
longitud_real = clave_desenvuelta[0]
clave_final = clave_desenvuelta[1:1+longitud_real]

print(binascii.hexlify(clave_final).decode())
```