Trabajo Final - Criptografía Aplicada

Autor: Sebastián Emilio Mayo

Asignatura: Criptografía Aplicada

Contenido: Resolución de desafíos seleccionados y material reproducible. Este Incluye código para ejecutar en local o Colab, explicaciones y recomendaciones de mitigación.

Este trabajo es con ayuda de lectura, busqueda en la web y mucha de IA, no es excusa pero no soy programador y muchas cosas no las alcanzo a comprender.

Índice

---

1. Introducción general

En este trabajo se resuelven y analizan **tres desafíos** representativos de debilidades criptográficas: malas fuentes de entropía, reuso de keystream en cifrados de flujo y manipulación de mensajes cifrados en modo CBC mediante flips de bits. Para cada desafío se incluye:

- Descripción mínima del desafío
- Análisis del problema y razones técnicas
- Desarrollo de la solución con código reproducible
- Prevención / formas de evitar el problema
- Herramientas y referencias

---

2. Desafío 1 — Clave generada a partir de fecha y hora (Aleatoriedad)

2.1 Descripción

Una aplicación genera claves o contraseñas derivadas directamente del timestamp del sistema (fecha y hora). Se dispone del hash resultante y el objetivo es recuperar la clave original o demostrar que es predecible.

2.2 Análisis del problema

La hora del sistema es un valor con baja entropía y predecible si se conoce la ventana temporal en la que se generó la clave. Un atacante puede realizar fuerza bruta en esa ventana temporal y comparar hashes hasta encontrar coincidencias.

2.3 Desarrollo de la solución (ataque por búsqueda temporal)

Script de ejemplo (descubrimiento por brute-force acotado por tiempo):

```python
# find_by_time.py
import hashlib
import datetime

objetivo = "REEMPLAZAR_CON_HASH_OBJETIVO"
# ejemplo: 2025-10-14 10:00:00 a 2025-10-14 10:05:00
inicio = datetime.datetime(2025,10,14,10,0,0)
fin = datetime.datetime(2025,10,14,10,5,0)

ts_inicio = int(inicio.timestamp())
ts_fin = int(fin.timestamp())

for ts in range(ts_inicio, ts_fin+1):
    fecha = str(ts)
    h = hashlib.md5(fecha.encode()).hexdigest()
    if h == objetivo:
        print("Encontrado:", ts, datetime.datetime.fromtimestamp(ts))
        break
else:
    print("No encontrado en el rango especificado")
```

Explicación: el script itera por los segundos del rango y calcula el hash (MD5 en el ejemplo) sobre el string del timestamp. Si coincide con el objetivo, se recupera el timestamp original.

2.4 Prevención

- No usar timestamps como semilla directa para claves.
- Usar generadores criptográficamente seguros: `os.urandom()`, `secrets` (Python), `SecureRandom` (Java).

- Si se necesita derivar una clave de algo temporal, combinar con sal (salt) aleatoria y un KDF seguro (p. ej. PBKDF2, Argon2).

2.5 Entregables/Archivos

- `find_by_time.py`
- Instrucciones de ejecución incluidas en la sección 5.

---

3. Desafío 2 — Secuencia cifrante repetida (Cifrado de flujo)

3.1 Descripción

Se usan dos o más mensajes cifrados con la **misma keystream** (reuso del keystream). Se tienen los ciphertexts y el objetivo es recuperar los plaintexts.

3.2 Análisis del problema

Si $C1 = P1 \oplus K$ y $C2 = P2 \oplus K$, entonces $C1 \oplus C2 = P1 \oplus P2$. La XOR de los dos ciphertexts elimina la keystream y devuelve la XOR entre los plaintexts, lo que permite ataques como crib-dragging para recuperar texto legible.

3.3 Desarrollo de la solución (simulación y crib-dragging)

Simulación (generación que reusa keystream):

```python
# sim_encrypt.py
import os

def xor_bytes(a: bytes, b: bytes) -> bytes:
    return bytes(x ^ y for x, y in zip(a, b))

P1 = b"Hola profesor, esto es un mensaje secreto 1."
P2 = b"Estimado profesor, este mensaje secreto 2 es similar."

keystream = os.urandom(max(len(P1), len(P2)))
C1 = xor_bytes(P1, keystream[:len(P1)])
C2 = xor_bytes(P2, keystream[:len(P2)])

print(C1.hex())
print(C2.hex())
```

Crib-dragging (ataque asistido):

```python
# crib_drag.py
import binascii

# Pegar los hex de C1 y C2
C1_hex = "REEMPLAZAR_C1_HEX"
C2_hex = "REEMPLAZAR_C2_HEX"
```

```
C1 = binascii.unhexlify(C1_hex)
C2 = binascii.unhexlify(C2_hex)

L = min(len(C1), len(C2))
X = bytes(a ^ b for a, b in zip(C1[:L], C2[:L]))  # P1 xor P2

def apply_crib(xor_p1p2: bytes, crib: bytes):
    results = []
    for i in range(0, len(xor_p1p2) - len(crib) + 1):
        frag = bytes(a ^ b for a, b in zip(crib, xor_p1p2[i:i+len(crib)])
)
        printable = ''.join(chr(c) if 32 <= c < 127 else '?' for c in fra
g)
        results.append((i, printable))
    return results

if __name__ == '__main__':
    crib = input("Ingrese crib: ").encode()
    for pos, out in apply_crib(X, crib):
        print(f"pos {pos}: {out}")
```

Uso práctico: probar cribs comunes (“profesor”, “mensaje”, “secreto”, etc.) para identificar posiciones y reconstruir trozos de texto.

3.4 Prevención

- No reutilizar keystreams: OTP y stream ciphers requieren uso único.
- Usar modos autenticados y modernos (ChaCha20-Poly1305, AES-GCM) y gestionar nonces correctamente.
- Rotación de claves y generación de nonces/IVs con entropía suficiente.

3.5 Entregables

- `sim_encrypt.py`, `crib_drag.py`
- Capturas de ejecución y resultados en anexos.

---

4. Desafío 3 — Cambio de bits en modo CBC (Bit-flipping en CBC)

4.1 Descripción

Un sistema utiliza AES-CBC (u otro cifrado por bloques en CBC) y no valida la integridad/autenticidad del mensaje. Un atacante puede modificar bits en el ciphertext para provocar cambios controlados en el plaintext del bloque siguiente tras la descifrado.

4.2 Análisis del problema

En CBC:

- Cifrado: $C_i = E_K(P_i \oplus C_{i-1})$ (con $C_0 = IV$)
- Descifrado: $P_i = D_K(C_i) \oplus C_{i-1}$

Si un atacante modifica C_{i-1} (por ejemplo, $C_{i-1}' = C_{i-1} \oplus delta$), entonces el bloque descifrado $P_i' = D_K(C_i) \oplus C_{i-1}' = P_i \oplus delta$. Es decir, la modificación de bits en un bloque del ciphertext afecta de forma XOR al plaintext del bloque siguiente. Esto permite ataques tipo *bit-flipping* para cambiar campos controlables (p. ej. privilegios, valores en un token) si no existe autenticación.

4.3 Desarrollo de la solución (demostración y exploit controlado)

Script de ejemplo que muestra el efecto de flip:

```
# cbc_bitflip_demo.py
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes

BLOCK = AES.block_size

key = get_random_bytes(16)
iv = get_random_bytes(BLOCK)

# Mensaje original (de longitud múltiplo de bloque o con padding PKCS7)
from Crypto.Util.Padding import pad, unpad

plaintext = b"user=guest;uid=1000;role=guest;"
cipher = AES.new(key, AES.MODE_CBC, iv)
ct = iv + cipher.encrypt(pad(plaintext, BLOCK))

# Separar iv y ciphertext blocks
iv = ct[:BLOCK]
C1 = ct[BLOCK:BLOCK*2]
C2 = ct[BLOCK*2:BLOCK*3]

# Supongamos que queremos cambiar 'role=guest' a 'role=admin ' (nota: mis
mo largo)
# Encontrar offset donde aparece 'guest' en el plaintext y calcular delta

# Para demo: recuperamos plaintext real para saber la posición (en ataque
real se usaría inferencia)
cipher2 = AES.new(key, AES.MODE_CBC, ct[:BLOCK])
pt = unpad(AES.new(key, AES.MODE_CBC, ct[:BLOCK]).decrypt(ct[BLOCK:BLOCK*
2]) if False else b"", BLOCK)
# En esta demo vamos a operar directamente:

print("Ciphertext original (hex):", ct.hex())

# Identificamos el bloque que contiene 'guest' y calculamos delta
```

```
old = b"guest"
new = b"admin "  # 6 bytes; ajustar si es necesario

# Suponiendo que 'guest' empieza en un bloque en posición conocida, calcu
lamos delta
# delta = old xor new; aplicamos delta al bloque anterior correspondiente
en el ciphertext

delta = bytes([a ^ b for a,b in zip(old, new)])

# Para simplificar la demo: buscamos la ocurrencia de 'guest' en el plain
text
from Crypto.Cipher import AES
pt_full = unpad(AES.new(key, AES.MODE_CBC, ct[:BLOCK]).decrypt(ct[BLOCK:]
), BLOCK)
idx = pt_full.find(old)
print("Posicion de 'guest' en plaintext:", idx)

# Ajustar índice para encontrar en qué bloque está y aplicar delta al blo
que anterior
block_idx = idx // BLOCK
byte_offset = idx % BLOCK

ct_list = bytearray(ct)
# aplicar delta al bloque anterior (IV si block_idx == 0)
for i in range(len(delta)):
    ct_list[BLOCK*(block_idx) + byte_offset + i] ^= delta[i]

ct_modified = bytes(ct_list)
print("Ciphertext modificado (hex):", ct_modified.hex())

# Descifrar con la misma clave para ver el resultado
cipher_dec = AES.new(key, AES.MODE_CBC, ct_modified[:BLOCK])
pt_mod = unpad(cipher_dec.decrypt(ct_modified[BLOCK:]), BLOCK)
print("Plaintext modificado:", pt_mod)
```

Nota: el script demuestra cómo una modificación en el bloque anterior cambia bytes en el plaintext del bloque siguiente. En implementaciones reales, la posición exacta se deduce por conocimiento del formato o por ensayo/error.

4.4 Prevención

- **Autenticación de los mensajes:** usar AEAD (AES-GCM, ChaCha20-Poly1305) o combinar cifrado con MAC (Encrypt-then-MAC).
- **Validar integridad antes de procesar campos sensibles.**
- **No confiar en que el ciphertext sea inalterable.**

4.5 Entregables

- `cbc_bitflip_demo.py`

- Explicación paso a paso y resultados (capturas) en anexos.

---

5. Herramientas y cómo ejecutar los scripts

Recomendado: ejecutar en entorno virtual Python 3.8+.

Instalación básica:

```
python -m venv venv
source venv/bin/activate   # Linux / Mac
venv\Scripts\activate.bat  # Windows
pip install pycryptodome
```

Archivos incluidos (sugeridos para el ZIP): - `find_by_time.py` - `sim_encrypt.py` - `crib_drag.py` - `cbc_bitflip_demo.py` - `README.md` (instrucciones)

Ejecutar en Colab: subir los `.py` o copiar el código a un notebook; ejecutar celdas con outputs.

---

6. Conclusiones generales

Los tres desafíos escogidos muestran fallos típicos en implementaciones criptográficas: - Fuentes de entropía predecibles (timestamps) comprometen claves. - Reusar keystream en cifrados de flujo permite reconstrucción cruzada de mensajes. - La falta de autenticación en cifrados por bloques permite manipulación del plaintext mediante bit-flipping.

La solución práctica en todos los casos converge hacia dos principios: **usar primitivas seguras y modernas** (AEAD) y **gestionar correctamente entropía, nonces y claves**.

---

7. Anexos (código fuente completo)

(Incluir los archivos `.py` mencionados en la sección 5, cada uno con comentarios y ejemplos de ejecución. Se sugiere adjuntar capturas de ejecución y un README que explique cómo generar las salidas que se muestran en el informe.)

Fin del documento.

---

8. Desafío 4 — Padding Oracle (Cifrado de bloques)

8.1 Descripción

Un Padding Oracle ocurre cuando un servicio indica (directa o indirectamente) si el padding de un ciphertext es válido tras el descifrado. Un atacante puede aprovechar esa información (oracle) para descifrar el ciphertext completo sin conocer la clave, mediante múltiples consultas.

8.2 Análisis del problema

En esquemas como AES-CBC con padding PKCS#7, si el servidor devuelve mensajes distintos (errores de padding vs. errores de autenticación o contenido) según si el padding es correcto, esto constituye un oráculo que permite, mediante un ataque de tipo “padding oracle”, descubrir byte a byte el plaintext.

8.3 Desarrollo de la solución (esquema de ataque)

El ataque básico usa manipulación del bloque anterior (similar al bit-flipping) y consultas al servidor: 1. Se toma el ciphertext `C = IV || C1 || C2 || ...`. 2. Para recuperar `P_n` (último bloque), se modifica `C_{n-1}` prueba por prueba hasta que el servidor responda que el padding es válido. Esto permite deducir bytes del `P_n`. 3. Repetir byte a byte hasta recuperar el bloque.

8.4 Script de demostración (esqueleto para pruebas locales)

```
# padding_oracle_demo.py (esqueleto)
# Nota: este script no ataca servidores reales sin autorización.

import os
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

KEY = os.urandom(16)

# Funciones auxiliares para simular un oracle local

def encrypt(plaintext: bytes):
    iv = os.urandom(16)
    cipher = AES.new(KEY, AES.MODE_CBC, iv)
    return iv + cipher.encrypt(pad(plaintext, 16))

def padding_oracle(ciphertext: bytes) -> bool:
    """Devuelve True si el padding es válido, False si no."""
    iv = ciphertext[:16]
    body = ciphertext[16:]
    cipher = AES.new(KEY, AES.MODE_CBC, iv)
    try:
        pt = unpad(cipher.decrypt(body), 16)
        return True
```

```
    except ValueError:
        return False

# El ataque usa el oracle padding_oracle() para recuperar plaintext sin c
onocer KEY.
```

8.5 Prevención

- **No exponer información diferenciada en errores**: retornar un mensaje genérico y el mismo código de estado ante fallos de padding o de autenticación.
- **Usar esquemas autenticados AEAD** (AES-GCM, ChaCha20-Poly1305) que no requieren padding y protegen integridad y confidencialidad.
- **Validar integridad antes de procesar** y usar Encrypt-then-MAC si no hay AEAD.

(He añadido este desafío al documento; podés revisarlo y editarlo en el lienzo.)