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 de código 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 Crisis de Entropía y Claves Débiles

La debilidad fundamental de generar claves a partir de la fecha y hora (*timestamp*) radica en la **aleatoriedad insuficiente (baja entropía)**. Un generador de números pseudoaleatorios (PRNG) utiliza una **semilla** como punto de partida; si esta semilla es predecible, como el time() del sistema, los números generados son predecibles.

Un atacante, conociendo el momento aproximado de la generación de la clave, solo necesita probar un rango de *timestamps* muy pequeño (ej. 5 segundos antes y después) para reproducir todas las posibles claves, reduciendo el ataque a **fuerza bruta** a un tiempo trivial.

Caso Real de Alto Impacto (La Crisis de Debian OpenSSL, 2006-2008): La importancia de una fuente de entropía robusta fue dramáticamente demostrada por una vulnerabilidad en la biblioteca **OpenSSL** de Debian y sus derivados (Ubuntu). Un desarrollador modificó accidentalmente el código para eliminar la recolección de entropía externa, basando la generación de claves en una cantidad muy limitada de *pids* (identificadores de proceso). Esto causó que millones de claves criptográficas (para **SSH, SSL/TLS, DNSSEC**) fueran generadas a partir de un **conjunto reducido y predecible de valores**. Un atacante podía precalcular estas claves débiles y utilizarlas para descifrar el tráfico o autenticarse en sistemas sin permiso. La solución fue la implementación de **RNGs criptográficamente seguros (CSPRNGs)** que obligan al sistema a recolectar entropía de fuentes físicas impredecibles (movimientos del mouse, latencia de E/S, ruido del sistema).

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):

```
# 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`

---

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

Reúso de *Keystream* y Fallo en la Confidencialidad

El **Cifrado de Flujo** utiliza una secuencia cifrante (*keystream*) de bits que se combina con el texto claro mediante la operación **OR Exclusivo (XOR)**: $C = P \oplus K$.

La regla criptográfica cardinal de los cifrados de flujo es que **la secuencia cifrante (K) nunca debe ser reutilizada**. El **reúso de *keystream*** ocurre cuando se utiliza la misma clave y el mismo **IV (Vector de Inicialización)** para cifrar dos mensajes distintos (P_1 y P_2).

Si el atacante obtiene dos textos cifrados (C_1 y C_2) cifrados con la misma secuencia K, puede realizar la siguiente operación sin conocer la clave:

$$C_1 \oplus C_2 = (P_1 \oplus K) \oplus (P_2 \oplus K) = P_1 \oplus P_2$$

El resultado ($P_1 \oplus P_2$) es la diferencia entre los dos textos claros. Si el atacante tiene conocimiento parcial de uno de los mensajes, puede **deducir el contenido del otro mensaje** y, eventualmente, la secuencia cifrante K. Esto anula completamente la **Confidencialidad** que el cifrado estaba destinado a ofrecer.

Caso Real de Alto Impacto (Fallo del WEP):

El protocolo de seguridad WEP (Wired Equivalent Privacy) para redes WiFi colapsó debido precisamente al reúso de keystream. Utilizaba el algoritmo RC4 y un IV de solo 24 bits, que se agotaba rápidamente, forzando la repetición de la secuencia cifrante. Los ataques FMS y KoreK se basaban en la captura masiva de paquetes para explotar esta debilidad y recuperar la clave maestra del AP en minutos.

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

Simulación (generación que reusa keystream):

```
# 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):

```
# 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`

---

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

Manipulación de Bits en CBC y Ataques *Padding Oracle*

El modo de operación **Cipher Block Chaining (CBC)** encadena el cifrado de cada bloque (Cn) con el bloque cifrado anterior (C_{n-1}), utilizando la operación **XOR** antes del descifrado: $P_n = D_K(C_n) \oplus C_{n-1}$.

El problema surge porque esta estructura permite al atacante realizar un **ataque de manipulación (*Bit-Flipping*)**. Un cambio de un bit en el bloque cifrado anterior (C_{n-1}) resulta en un **cambio predecible en el mismo bit de la posición del texto claro** (P_n). Esto rompe la **Integridad** y permite inyectar contenido malicioso al mensaje.

Caso Real de Alto Impacto (Padding Oracle y POODLE):

Esta manipulación se vuelve crítica cuando se combina con un Oráculo de Padding. Un Oráculo es una debilidad en el servidor que revela información (ej. un mensaje de error o un tiempo de respuesta diferente) si el relleno (padding) del mensaje descifrado es válido o inválido.

- El atacante manipula sistemáticamente el último bloque cifrado.
- Observa la respuesta del servidor (el "oráculo").
- Si la respuesta indica que el *padding* es válido, el atacante deduce el valor de un byte del texto claro.

Al repetir este proceso byte a byte, el atacante puede **descifrar completamente el mensaje (confidencialidad)** e incluso **alterar su contenido (integridad)** sin conocer la clave. Este vector de ataque afectó protocolos como **TLS/SSL** (ej. vulnerabilidad POODLE) y numerosos *frameworks* de desarrollo web.

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, ChaCha20Poly1305) 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+.

---

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. Desafío 4 — Padding Oracle (Cifrado de bloques)

7.1 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.

7.2 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.

7.3 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.

7.4 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

Prevención y Mitigaciones

Contra Aleatoriedad Débil (Desafío 1):

- **Utilizar CSPRNgs:** Emplear únicamente Generadores de Números Pseudoaleatorios Criptográficamente Seguros (CSPRNGs), como os.urandom o secrets en Python, o APIs de sistema operativo diseñadas para recolectar entropía de fuentes físicas (ej. /dev/urandom en Linux o la API de CryptGenRandom de Windows).
- **Nunca usar *timestamp* como única semilla:** La semilla de un PRNG debe ser una fuente de entropía de alta calidad, nunca valores predecibles como la fecha y hora.

Contra Reúso de *Keystream* (Desafío 2):

- **Garantizar la unicidad del IV:** Asegurar que el **Vector de Inicialización (IV)** sea un valor aleatorio y único para **cada mensaje cifrado** con la misma clave. El IV puede ser público, pero **no debe repetirse jamás**.
- **Migrar a AES/Cifrado de Bloques:** Reemplazar algoritmos de flujo obsoletos (como RC4) por cifrados de bloque modernos (como AES) en modos seguros (ej. GCM).

Contra Manipulación de Bits en CBC (Desafío 3):

- **Usar Cifrado Autenticado (AEAD):** Migrar a modos de operación que proporcionan **Integridad y Confidencialidad simultáneamente**, como **AES-GCM (Galois/Counter Mode)** o **ChaCha20-Poly1305**. Estos modos añaden una etiqueta de autenticación (MAC) al mensaje, lo que obliga al receptor a **verificar la integridad** antes de descifrar.
- **Mitigación de Oráculos:** Si se debe usar CBC, **nunca exponer información diferenciada** en caso de errores de *padding* o autenticación. Todos los errores deben retornar un mensaje genérico y el mismo código de estado para no darle información al atacante.
