HTB - Challenge: Hidden Handshake

Dificultad:	**Very Easy**
Release:	**02/05/2025**

Skills Required

- Understanding of AES in CTR mode and its vulnerabilities
- Familiarity with keystream recovery through chosen plaintext attacks

Skills Learned

- Leveraging partial plaintext knowledge for full message recovery

El desafío *Hidden Handshake*, perteneciente a la categoría de criptografía en HackTheBox, plantea el análisis y explotación de un servicio remoto que implementa un mecanismo de cifrado aparentemente robusto basado en AES en modo CTR. A primera vista, el servicio parece adherirse a prácticas criptográficas estándar: deriva claves mediante SHA-256, utiliza un *nonce* explícito y encapsula la *flag* dentro de un mensaje cifrado. Sin embargo, un examen minucioso del código revela una concatenación de decisiones de diseño que, en conjunto, erosionan por completo las garantías de seguridad del esquema.

El objetivo de este write-up es exponer, con rigor técnico y claridad conceptual, el proceso de identificación de la vulnerabilidad, su fundamentación criptográfica y la construcción de un exploit funcional que permite recuperar la *flag* sin necesidad de conocer la clave secreta del servidor ni realizar ataques de fuerza bruta. Para ello, se analiza en profundidad el comportamiento del modo CTR, se formaliza matemáticamente el impacto de la reutilización del *keystream* y se demuestra cómo un atacante puede manipular el espacio de entrada controlado para reconstruir el texto en claro íntegro mediante operaciones XOR.

El resultado es una explotación elegante y determinista que ilustra cómo incluso implementaciones basadas en primitivas criptográficas sólidas pueden volverse completamente inseguras cuando se violan sus invariantes fundamentales. Este análisis no solo resuelve el reto, sino que también pone de manifiesto la importancia de comprender los modos de operación y sus propiedades algebraicas para evaluar la seguridad real de un sistema.

Analyzing the source code

En este desafío se nos proporciona un único artefacto: el fichero *server.py*, que contiene el código fuente del servicio remoto responsable de cifrar y exfiltrar un mensaje que incorpora la *flag*. El análisis preliminar del script revela que la aplicación implementa un esquema criptográfico basado en AES en modo CTR, empleando un flujo de cifrado derivado de un secreto estático del servidor (*server_secret*) y de un parámetro controlado por el usuario (*Agent Codename*).

Para contextualizar adecuadamente la vulnerabilidad, conviene detenerse brevemente en la naturaleza del modo CTR (*Counter Mode*). A diferencia de los modos de operación por bloques tradicionales, CTR convierte el cifrado simétrico en un generador de flujo pseudoaleatorio: en lugar de procesar directamente el texto en claro, AES se aplica sobre una secuencia de contadores combinados con un *nonce*, produciendo un *keystream* que posteriormente se mezcla con el mensaje mediante una operación XOR. Esta arquitectura aporta ventajas notables —paralelización, ausencia de padding, simetría entre cifrado y descifrado—, pero impone una condición criptográfica estricta: la unicidad del par *(clave, nonce)*. La reutilización de dicho par, especialmente si el *nonce* es predecible o manipulable por un atacante, compromete la seguridad del flujo de clave y permite recuperar o alterar el texto en claro.

En este punto resulta pertinente precisar qué entendemos por *keystream*. En los modos de cifrado en flujo, el *keystream* es la secuencia de bytes pseudoaleatorios generada a partir de la clave, el *nonce* y el contador interno del modo CTR. El texto en claro no se cifra directamente: se combina con este flujo mediante XOR. De esta propiedad se deriva una consecuencia crítica para la seguridad del esquema: si un atacante obtiene cualquier fragmento del texto en claro correspondiente a un ciphertext generado con un mismo *keystream*, puede recuperar dicho flujo por XOR y reutilizarlo para descifrar cualquier otro mensaje cifrado bajo el mismo par *(clave, nonce)*. La seguridad del modo CTR depende, por tanto, de que ese flujo de clave jamás se repita.

En el servicio analizado, esta propiedad se vulnera de forma explícita. El *nonce* del modo CTR no solo carece de aleatoriedad, sino que coincide exactamente con el valor *pass2* introducido por el usuario, convirtiendo un parámetro crítico del esquema criptográfico en un vector de manipulación directa. A ello se suma que la clave se deriva mediante SHA-256 sobre la concatenación de *server_secret* y *pass2*, de modo que cualquier reutilización o elección estratégica de *pass2* afecta simultáneamente al *nonce* y a la clave, generando una correlación artificial entre ambos elementos que degrada severamente la robustez del cifrado. Esta confluencia de decisiones de diseño —reutilización del *nonce*, dependencia directa del usuario y ausencia de mecanismos de autenticación— abre la puerta a un ataque de recuperación del mensaje mediante análisis del flujo de clave y explotación del espacio de entrada controlado.

El flujo de cifrado se articula a través de la función encrypt(), que instancia un objeto AES-CTR con la clave derivada y el *nonce* suministrado, y posteriormente cifra un mensaje construido dinámicamente. Dicho mensaje incorpora tanto el *Agent Codename* proporcionado por el usuario como la *flag* almacenada en el servidor, lo que convierte al *ciphertext* resultante en un objetivo de alto valor para un criptoanálisis basado en la reutilización del flujo de clave (*keystream reuse*) o en la manipulación del prefijo controlado por el atacante.

La estructura del servicio, reproducida a continuación, permite observar con claridad la concatenación de responsabilidades criptográficas, la ausencia de separación entre parámetros de autenticación y parámetros de cifrado, y la dependencia directa del usuario para la generación del *nonce*, factores que en conjunto habilitan un ataque de recuperación íntegra del mensaje en texto claro.

Finding the vulnerability

Una aproximación ingenua podría consistir en intentar recuperar la clave secreta mediante fuerza bruta. Sin embargo, esta vía resulta completamente inviable: el valor *server_secret* es una cadena aleatoria de ocho caracteres seleccionados de un alfabeto relativamente amplio, lo que genera un espacio de búsqueda astronómico y, por tanto, inabordable desde un punto de vista práctico.

No obstante, un examen más atento del código revela una debilidad estructural mucho más explotable. El parámetro *pass2*, proporcionado por el usuario, no solo interviene en la derivación de la clave mediante SHA-256, sino que además se reutiliza directamente como *nonce* del modo CTR.

Esta decisión de diseño provoca que, siempre que enviemos el mismo *pass2*, el servicio genere exactamente el mismo flujo de clave. En otras palabras, AES-CTR deja de comportarse como un cifrado seguro y pasa a equivaler a una simple operación XOR con un *keystream* constante pero desconocido.

Este hecho es crucial: si logramos obtener cualquier segmento del texto en claro correspondiente a un ciphertext generado con ese mismo *keystream*, podemos recuperar el flujo de clave por XOR y, a partir de ahí, descifrar cualquier otra salida producida bajo el mismo *pass2*. Aunque no conocemos el mensaje completo —pues incluye la *flag*— sí controlamos una parte significativa del texto en claro: el *Agent Codename*. Esto nos permite manipular la longitud del mensaje inicial para forzar al servidor a cifrar grandes cantidades de datos completamente controlados por nosotros.

La estrategia de explotación consiste, por tanto, en enviar un *Agent Codename* extremadamente largo, de modo que el ciphertext resultante contenga un fragmento sustancial del *keystream*. Una vez recuperado ese flujo de clave parcial, basta con repetir la operación utilizando el mismo *pass2*, pero esta vez enviando un nombre de usuario muy corto. Con ello conseguimos que la porción del ciphertext correspondiente a la *flag* se alinee exactamente con posiciones del *keystream* que ya hemos recuperado, permitiendo descifrarla de forma directa.

Exploitation

Una vez identificada la reutilización del *keystream* como núcleo de la vulnerabilidad, resulta pertinente formalizar matemáticamente las implicaciones de este fenómeno. Aunque el cifrado mediante XOR pueda parecer trivial, su estructura algebraica permite derivar relaciones muy potentes cuando el flujo de clave permanece constante entre dos cifrados distintos. Para clarificar este punto y justificar rigurosamente la estrategia de explotación, es útil enunciar y demostrar la propiedad fundamental que gobierna la recuperación del texto en claro en escenarios de reutilización de *keystream*. Esta propiedad, que puede formularse como un pequeño teorema dentro del marco del criptoanálisis clásico, establece la relación exacta entre los textos en claro y los ciphertexts afectados por un mismo flujo de clave.

Enunciado

Sea un esquema de cifrado en flujo basado en XOR, donde un mensaje P se cifra como

$$C = P \oplus K,$$

siendo Kel *keystream* generado a partir de una misma pareja ("clave","nonce"). Supongamos que se cifran dos mensajes P_1y P_2reutilizando exactamente el mismo *keystream* K, obteniéndose los correspondientes ciphertexts C_1y C_2. Entonces, si se conoce íntegramente P_1, se puede recuperar P_2mediante la relación:

$$P_2 = P_1 \oplus C_1 \oplus C_2$$

Demostración

Por definición del esquema de cifrado en flujo basado en XOR, tenemos:

$$C_1 = P_1 \oplus K,$$
$$C_2 = P_2 \oplus K,$$

donde en ambos casos el *keystream* Kes idéntico, al haberse reutilizado el mismo par ("clave","nonce").

Consideremos ahora la XOR de ambos ciphertexts:

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

Por asociatividad y conmutatividad de la XOR, podemos reagrupar términos:

$$C_1 \oplus C_2 = P_1 \oplus P_2 \oplus K \oplus K.$$

Dado que para cualquier valor x se cumple $x \oplus x = 0$, obtenemos:

$$K \oplus K = 0,$$

y, por tanto,

$$C_1 \oplus C_2 = P_1 \oplus P_2.$$

Si ahora conocemos P_1, basta con aplicar de nuevo XOR a ambos lados con P_1:

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

Aplicando de nuevo la propiedad $x \oplus x = 0$:

$$P_1 \oplus P_1 = 0,$$

obtenemos:

$$P_1 \oplus (C_1 \oplus C_2) = 0 \oplus P_2 = P_2.$$

Reordenando los términos, llegamos a la expresión final:

$$P_2 = P_1 \oplus C_1 \oplus C_2.$$

Lo que demuestra que, en presencia de reutilización de *keystream* y conocimiento de un texto en claro P_1, es posible recuperar el segundo texto en claro P_2 únicamente a partir de los ciphertexts C_1, C_2 y del propio P_1.

Solución

Una vez establecida la relación algebraica

$$P_2 = P_1 \oplus C_1 \oplus C_2,$$

la explotación del servicio se reduce a construir dos peticiones cuidadosamente diseñadas que permitan obtener los ciphertexts C_1 y C_2 bajo un mismo *keystream*, así como un texto en claro P_1 suficientemente extenso para recuperar una porción amplia del flujo de clave.

El objetivo es forzar al servidor a cifrar, con el mismo valor *pass2*, dos mensajes cuya estructura difiera únicamente en la longitud del prefijo controlado por el usuario. En la primera petición, se envía un *Agent Codename* extremadamente largo, de forma que el ciphertext resultante contenga una gran cantidad de bytes cuyo texto en claro conocemos íntegramente. Esto nos permite recuperar un segmento sustancial del *keystream* mediante XOR. En la segunda petición, se reutiliza exactamente el mismo *pass2*, pero esta vez se envía un nombre de usuario muy corto, desplazando así la parte del mensaje que contiene la *flag* hacia posiciones del *keystream* previamente recuperadas. La combinación de ambos ciphertexts, junto con el texto en claro conocido, permite aplicar directamente la igualdad demostrada en el teorema y obtener la *flag* en texto claro.

El siguiente script implementa esta estrategia de forma programática:

```python
#!/usr/bin/python3
from pwn import process, remote, xor
from argparse import ArgumentParser
import sys, re

def send_message(objetivo, pass2:bytes, username:bytes):
    objetivo.sendlineafter(b'key: ', pass2)
    objetivo.sendlineafter(b'Codename: ', username)
    objetivo.recvuntil(b'transmission: ')
    return bytes.fromhex(objetivo.recvline().strip().decode())

def main(ip):
    print(ip)
    if ip == "127.0.0.1":
        objetivo = process(['python3', 'server.py'])
    else:
        host, port = ip.split(':')
        objetivo = remote(host, port)

    pass2 = b'A' * 8
    username = b'B' * 1000

    known = f'Agent {username}, your clearance for Operation Blackout is: '.encode()
    ct1 = send_message(objetivo, pass2, username)

    username = b'B' * 5
    ct2 = send_message(objetivo, pass2, username)
    pt = xor(known, ct1, ct2)

    print(re.search(rb'(HTB{.*})\.', pt).groups(1)[0].decode())

if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument("-i", "--ip", help="ip objetivo, formato host:port", required=True)
    args = parser.parse_args()
    main(args.ip)
```

El exploit se articula en tres fases claramente diferenciadas:

1. **Fijación del *nonce* y del *keystream*:** Se selecciona un valor de *pass2* completamente controlado (b'A' * 8). Dado que el servidor reutiliza este valor como *nonce* y como parte de la derivación de la clave, cualquier petición posterior con el mismo *pass2* generará exactamente el mismo *keystream*.

2. **Extracción del *keystream*:** Se envía un *Agent Codename* de gran longitud (b'B' * 1000). El mensaje generado por el servidor comienza con un prefijo totalmente conocido por el atacante.

   Al conocer este prefijo completo, basta con aplicar XOR entre el ciphertext obtenido y el texto en claro para recuperar un segmento extenso del *keystream*.

3. **Descifrado de la flag:** Se envía una segunda petición con el mismo *pass2*, pero con un nombre de usuario muy corto. Esto provoca que la parte del mensaje que contiene la *flag* se cifre utilizando posiciones del *keystream* ya recuperadas. Aplicando la igualdad

   $$P_2 = P_1 \oplus C_1 \oplus C_2,$$

   se obtiene la *flag* en texto claro, que se extrae mediante una expresión regular.

Este procedimiento demuestra cómo una debilidad aparentemente sutil puede degradar completamente la seguridad del cifrado, permitiendo la recuperación íntegra de información sensible sin necesidad de conocer la clave ni de realizar ataques de fuerza bruta.

```
┌──(isolated)─(usuario㉿kali)-[~/HTB]
└─$ python3 solver.py -i 154.57.164.71:30725
154.57.164.71:30725
[+] Opening connection to 154.57.164.71 on port 30725: Done
HTB{                                  }
[*] Closed connection to 154.57.164.71 port 30725
```