Voyage au coeur d'un programme exécutable
EPISODE 2

Thierry Vaira
LaSalle Avignon BTS SN IR

v0.1 28/05/2020

Sommaire

Pré-requis : gdb

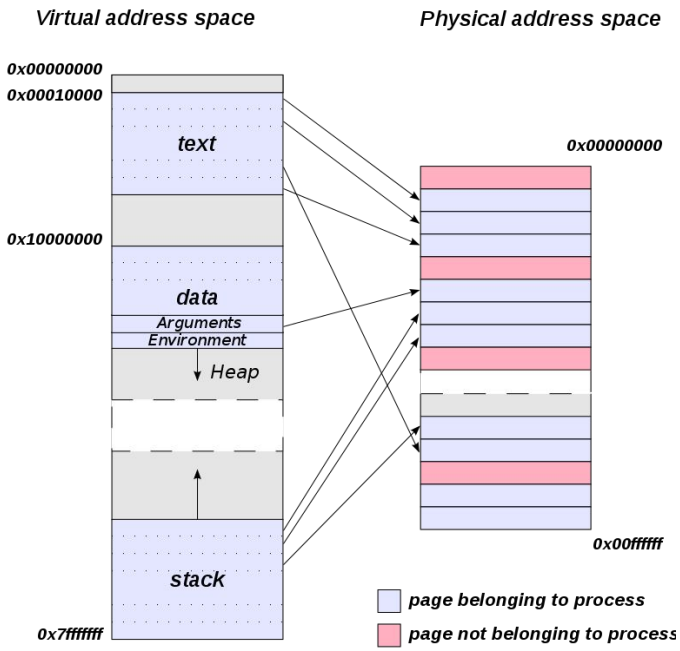
I. Notions de base : mémoire virtuelle

Chaque processus (un programme en exécution) a “droit” à un espace mémoire entièrement isolé (mémoire virtuelle).

Dans une architecture 32 bits, cette mémoire est adressée par mots (4 octets) et a une taille de 4 Giga octets dans l'espace d'adresse de **`0x00000000 - 0xffffffff`**.

Dans une architecture 64 bits, l’espace mémoire adressable est en théorie de 16 ExaOctets (environ 4 milliards de fois la taille de l’espace 32 bits). Pour l’instant, seuls les 48 bits d'une adresse virtuelle sont effectivement utilisés (notamment l’espace de **`0`** à **`0x00007fffffffffff`**) pour un total de 256 To d'espace d'adressage virtuel utilisable. Ceci est encore 65 536 fois plus grand que l'espace d'adressage de 4 Go des machines 32 bits.

I. Notions de base : mémoire virtuelle



Lien : https://wiki.deimos.fr/L%27adressage_m%C3%A9moire_et_son_allocation.html

Exemple n°0 : mémoire virtuelle

```
$ vim exemple0.c
...
int main(int argc, char **argv)
{
    printf("\n@argc       : %p\n", &argc);
    printf("@argv        : %p\n", &argv);
    printf("@main        : %p\n", main);
    system("sleep 20");
    return 0;
}
```

Manipulations : mémoire virtuelle

```
$ gcc exemple0.c -o exemple0

$ ./exemple0 &
@argc          : 0x7fffffffde5c
@argv          : 0x7fffffffde50
@main          : 0x555555554694

$ ./exemple0 &
@argc          : 0x7fffffffde5c
@argv          : 0x7fffffffde50
@main          : 0x555555554694
```

Exemple n°1 : les variables

```
$ vim exemple1.c
#include <stdlib.h>
#include <string.h>
int var1;
char var2[] = "buf1";
int main(int argc, char **argv) {
    int var3 = 3;
    static int var4;
    static char var5[] = "buf2";
    char * var6;
    var6 = malloc(512);
    strcpy(var6, "hello");
    return 0;
}
```

Variables globales

Variables locales

Où en mémoire ???

Manipulations : compilation

```
$ gcc exemple1.c

$ ls -l
-rwxr-xr-x  1 tv tv 8,3K juin   7 06:25 a.out
-rw-rw-r--  1 tv tv  265 juin   7 06:22 exemple1.c

$ gcc -g exemple1.c

$ ls -l
-rwxr-xr-x  1 tv tv 9,6K juin   7 06:25 a.out
-rw-rw-r--  1 tv tv  265 juin   7 06:22 exemple1.c
```

II. Notions de base : format ELF

Le système d'exploitation Linux utilise pour les programmes exécutables, le format **ELF** (*Executable Linking Format*) qui est composé de plusieurs sections.

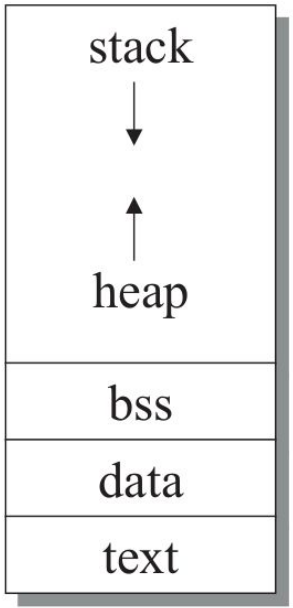
Un exécutable ELF est transformé en une image processus par le program **`loader`** (utilisation de l'appel système **`mmap()`** pour le mappage en mémoire virtuelle).

Manipulations : format ELF

```
$ hexdump -C a.out | more
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00
|.ELF............|

$ file a.out
a.out: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
GNU/Linux 3.2.0,
BuildID[sha1]=bce88b8a3b29692a51f79de58ef4d409b7d096fc, not stripped

$ objdump -d a.out
$ objdump -D a.out
```

III. Notions de base : les sections

Les sections principales d'un programme en mémoire sont :



les variables locales, les arguments (**`argc`** et **`argv`**), ainsi que les variables d'environnement

Les variables allouées dynamiquement par la fonction **`malloc()`** ou l'opérateur **`new`**

les données globales et statiques non-initialisées

les données globales et statiques initialisées (cad connues à la compilation)

Le code du programme, c'est-à-dire les instructions.

Manipulations les sections

```
$ size -A -x -t --common a.out

# Voir aussi
$ readelf -e a.out
$ objdump -h a.out

# Taille max de la pile
$ ulimit -a | grep stack
stack size              (kbytes, -s) 8192
# Espace du tas et de la pile pour un processus
$ cat /proc/{pid}/maps | grep -E "stack|heap"
$ cat /proc/self/maps | grep -E "stack|heap"
55555f0ae000-55555f2ea000 rw-p 00000000 00:00 0                          [heap]
7fffffdd000-7ffffffff000 rw-p 00000000 00:00 0                           [stack]
$ pmap <pid>
```

Manipulations : les sections

```
$ gcc -g exemple1.c

$ gdb -q ./a.out
(gdb) list
(gdb) break 13
(gdb) run

(gdb) print &var1
$1 = (int *) 0x555555755024 <var1>
(gdb) info symbol 0x555555755024
var1 in section .bss of a.out
(gdb) info address var1
Symbol "var1" is static storage at address 0x555555755024.
```

Manipulations : les sections

```
(gdb) print &var3
Can't take address of "var3" which isn't an lvalue.
```

```
int var3;
```

```
(gdb) print &var3
$1 = (int *) 0x7fffffffddd4
```

```
int var3 = 3;
```

```
(gdb) print &var6
$2 = (char **) 0x7fffffffddd8
(gdb) print var6
$3 = 0x555555756260 "hello"
(gdb) print /x*0x555555756260@2
$8 = {0x6c6c6568, 0x6f}
(gdb) x/2x 0x555555756260
0x555555756260:    0x6c6c6568    0x0000006f
```

little endian

Bilan : Variables et sections

```
$ cat exemple1.c
#include <stdlib.h>

int var1;                                   // bss
char var2[] = "buf1";                       // data

int main(int argc, char **argv)             // stack
{
    int var3 = 3;                           // stack
    static int var4;                        // bss
    static char var5[] = "buf2";            // data
    char * var6;                            // stack
    var6 = malloc(512);                     // heap
    strcpy(var6, "hello");
    return 0;
}
```

Bilan : Variables et sections

```
$ cat exemple1p.c
...
    printf("@var1 = %p (%ld octets) bss\n",   &var1, sizeof(var1));
    printf("@var2 = %p (%ld octets) data\n",  &var2, sizeof(&var2));
    printf("@var3 = %p (%ld octets) stack\n", &var3, sizeof(var3));
    printf("@var4 = %p (%ld octets) bss\n",   &var4, sizeof(var4));
    printf("@var5 = %p (%ld octets) data\n",  &var5, sizeof(&var5));
    printf("@var6 = %p (%ld octets) stack\n", &var6, sizeof(var6));
    printf("@argc = %p (%ld octets) stack\n", &argc, sizeof(argc));
    printf("@argv = %p (%ld octets) stack\n", &argv, sizeof(argv));

    return 0;
}
```

Manipulations : Variables et sections

```
$ gdb -q ./exemple1p
(gdb) start
Temporary breakpoint 1, main (argc=3, argv=0x7fffffffdd78) at exemple1p.c:9
9       {
(gdb) next
10          int var3 = 3;
...
17          printf("@var1 = %p (%ld octets) bss\n", &var1, sizeof(var1));
(gdb) next
@var1 = 0x555555755024 (4 octets) bss
(gdb) print &var1
$1 = (int *) 0x555555755024 <var1>
...
```

IV. Notions de base : les registres

Un **registre** est un emplacement de mémoire interne à un processeur.

Un processeur peut contenir plusieurs centaines de registres, à titre d'exemple, un processeur Intel 32 bits en contient 16.

Chaque registre a une capacité de 8, 16, 32 ou 64 bits (couramment la taille d'un bus).

Ils sont accessibles par le jeu d'instructions du processeur (très souvent avec l'instruction **`MOV`** pour un processeur Intel).

IV. Notions de base : les registres

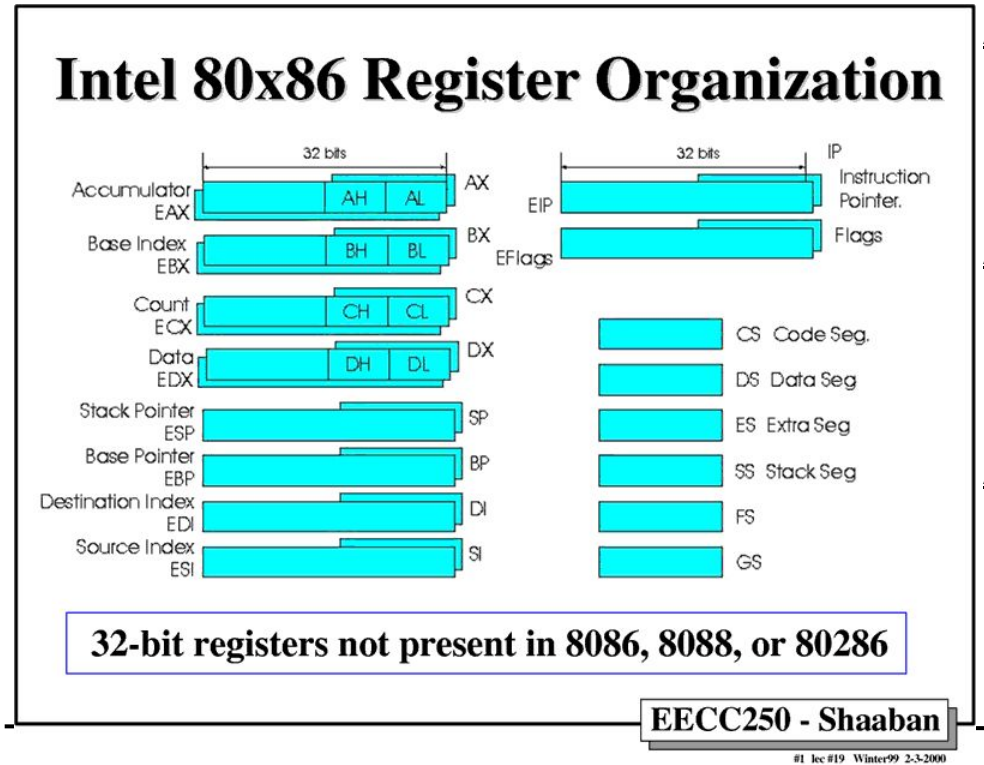
Principe :

- ➔ Les programmes transfèrent d'abord des données de la mémoire centrale vers des registres,
- ➔ puis effectuent des opérations sur ces registres (résultat dans un registre),
- ➔ et enfin transfèrent le résultat en mémoire centrale.

IV. Notions de base : les registres

- **Registres spécialisés** :
  - Sur de nombreux processeurs, les registres sont spécialisés et ne peuvent contenir qu'un type bien précis de données (entiers, flottants, adresses).
  - Certains registres spécialisés ont un rôle précis et unique : **pointeur d'instruction** (indique l'emplacement de la prochaine instruction à être exécutée), **registre d'état** (chaque bit représente un drapeau) et **pointeur de pile** (indique la position du prochain emplacement disponible dans la pile)
- **Registres généraux** : Ceux-ci peuvent stocker indifféremment adresses, entiers, flottants, etc. (parfois notés R0, R1, etc.)

IV. Notions de base : les registres



Architecture 16 bits :

Registres généraux : %ax, %bx, %cx et %dx

(gdb) print /x $ax

$1 = 0x6260

Architecture 32 bits :

Registres généraux : %eax, %ebx, %ecx et %edx

(gdb) print /x $eax

$2 = 0x55756260

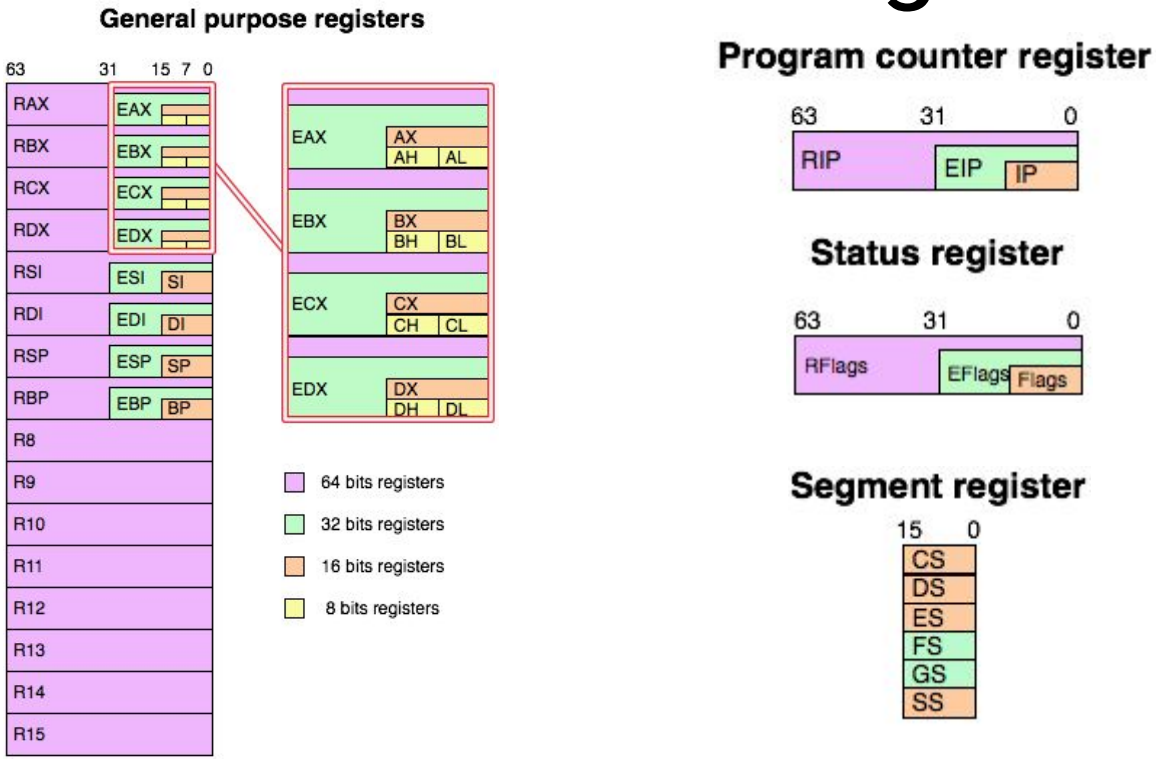
Architecture 64 bits :

Registres généraux : %rax, %rbx, %rcx et %rdx

(gdb) print /x $rax

$3 = 0x555555756260

IV. Notions de base : les registres



Manipulations : les registres

```
(gdb) info registers
rax            0x555555756260 93824994337376
rbx            0x00
rcx            0x555555756260 93824994337376
rdx            0x555555756260 93824994337376
rsi            0x00
rdi            0x555555756460 93824994337888
rbp            0x7fffffffddb0 0x7fffffffddb0
rsp            0x7fffffffdd90 0x7fffffffdd90
r8             0x22
...
r15            0x00
rip            0x55555555466e 0x55555555466e <main+36>
eflags         0x206    [ PF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x00
es             0x00
fs             0x00
gs             0x00
```

Manipulations : les registres

```
(gdb) print $rsp
$1 = (void *) 0x7fffffffdde0

(gdb) print $rbp
$2 = (void *) 0x7fffffffdde0

(gdb) print $rip
$3 = (void (*)()) 0x555555554604 <foo+10>

(gdb) print $pc
$4 = (void (*)()) 0x555555554604 <foo+10>

(gdb) x/5i $pc-6
   0x5555555545fe <foo+4>:    mov    %edi,-0x14(%rbp)
   0x555555554601 <foo+7>:    mov    %esi,-0x18(%rbp)
=> 0x555555554604 <foo+10>:   mov    -0x14(%rbp),%eax
   0x555555554607 <foo+13>:   mov    %eax,-0xc(%rbp)
   0x55555555460a <foo+16>:   mov    -0x18(%rbp),%eax
```

Manipulations : les registres

```
(gdb) info stack
#0  main (argc=3, argv=0x7fffffffde98) at exemple1.c:14

(gdb) info frame
Stack level 0, frame at 0x7fffffffddc0:
 rip = 0x55555555466e in main (exemple1.c:14); saved rip = 0x7ffff7a05b97
 source language c.
 Arglist at 0x7fffffffddb0, args: argc=3, argv=0x7fffffffde98
 Locals at 0x7fffffffddb0, Previous frame's sp is 0x7fffffffddc0
 Saved registers:
  rbp at 0x7fffffffddb0, rip at 0x7fffffffddb8

(gdb) info program
     Using the running image of child process 21982.
Program stopped at 0x55555555466e.
It stopped at breakpoint 1.
```

V. Notions de base : pile d'exécution

La zone de la pile d'exécution est utilisée par les fonctions (stockage des variables locales et passage des paramètres). Elle se comporte comme une pile, c'est-à-dire dernier entré, premier sorti.

L'espace mémoire de la pile est géré à partir de deux registres :

- **SP** (*Stack Pointer*) qui pointe sur le sommet de la pile et se met à jour automatiquement par les instructions d'empilement (**`push`**) et de dépilement (**`pop`**).
- **BP** (*Base Pointer*) qui pointe sur la base de la région de la pile contenant les données accessibles (variables locales, paramètres, ...).

VI. Notions de base : appel de fonction

A chaque fois qu'une fonction est appelée, il faut créer un nouvel environnement dans la mémoire pour les variables locales et les paramètres de cette fonction.

Le registre **`%rsp`** (*stack pointer*) contiendra l'adresse du sommet de la pile.

Le registre **`%rbp`** (*base pointer*) contiendra l'adresse du début de l'environnement de la fonction en cours.

Pour accéder aux paramètres ou variables locales de la fonction, il suffira simplement d'exprimer un décalage (*offset*) par rapport au registre **`%rbp`**.

VI. Notions de base : appel de fonction

L'appel d'une fonction se déroule en 3 étapes :

1. le **prologue** : à l'entrée d'une fonction, on sauvegarde l'état de la pile tel qu'il était avant d'entrer dans la fonction puis en réservant la mémoire nécessaire au bon déroulement de la fonction (par exemple ici 32 octets soit 0x20) :

```
push   %rbp
mov    %rsp,%rbp
sub    $0x20,%rsp
```

VI. Notions de base : appel de fonction

2. l'**appel** de la fonction : quand une fonction est appelée, ses paramètre sont mis dans la pile puis le pointeur d'instructions (IP) est sauvegardé pour que l'exécution des instructions reprenne au bon endroit après la fonction.

C'est l'instruction d'appel de la fonction (**`call`**) qui enregistre l'adresse de retour dans la pile d'exécution par un appel implicite à **`push %rip`**.

On aura alors :

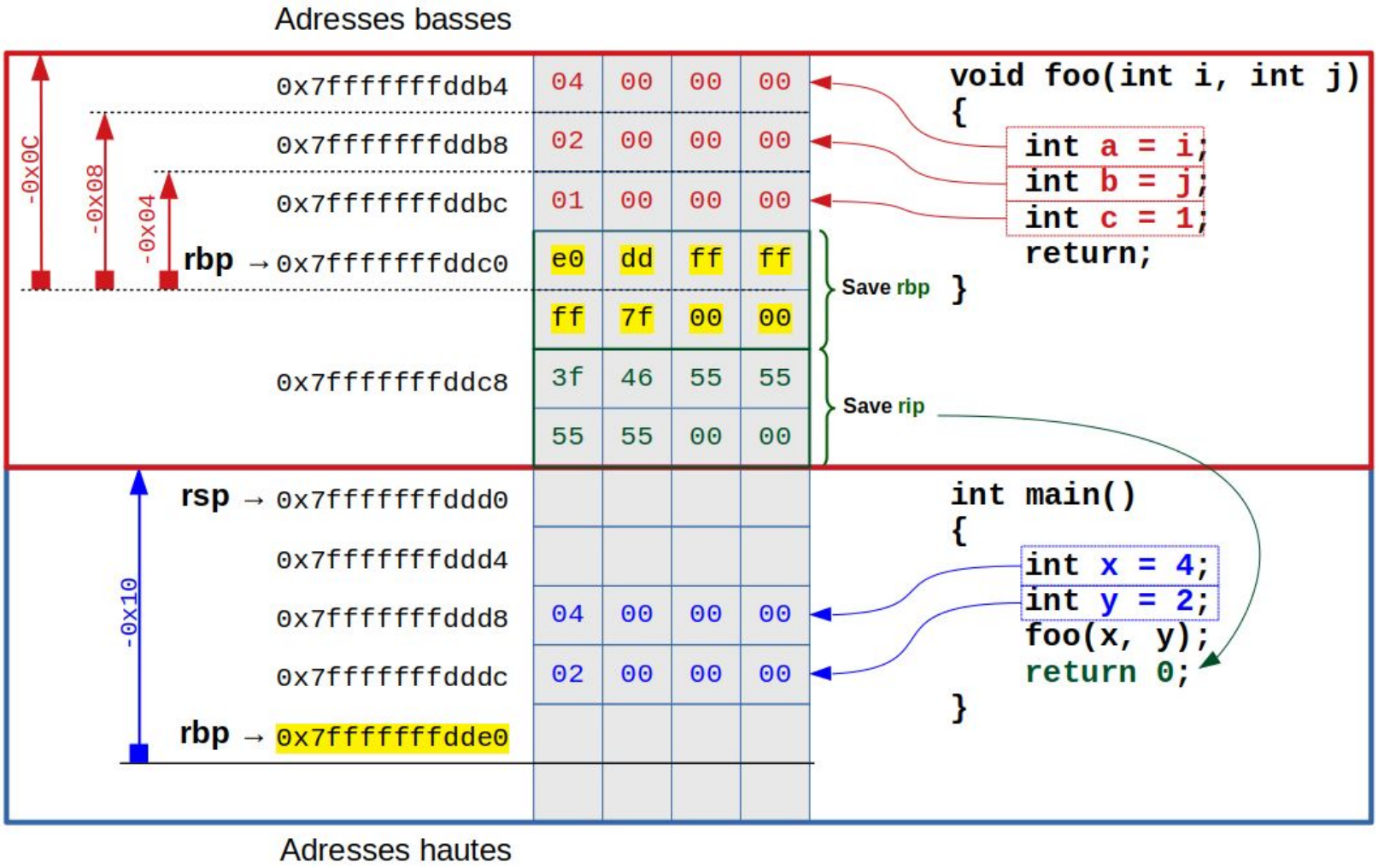
- à **`%rbp + 0`** : l'adresse de la trame de pile précédente
- à **`%rbp + 8`** : l'adresse de retour de la fonction

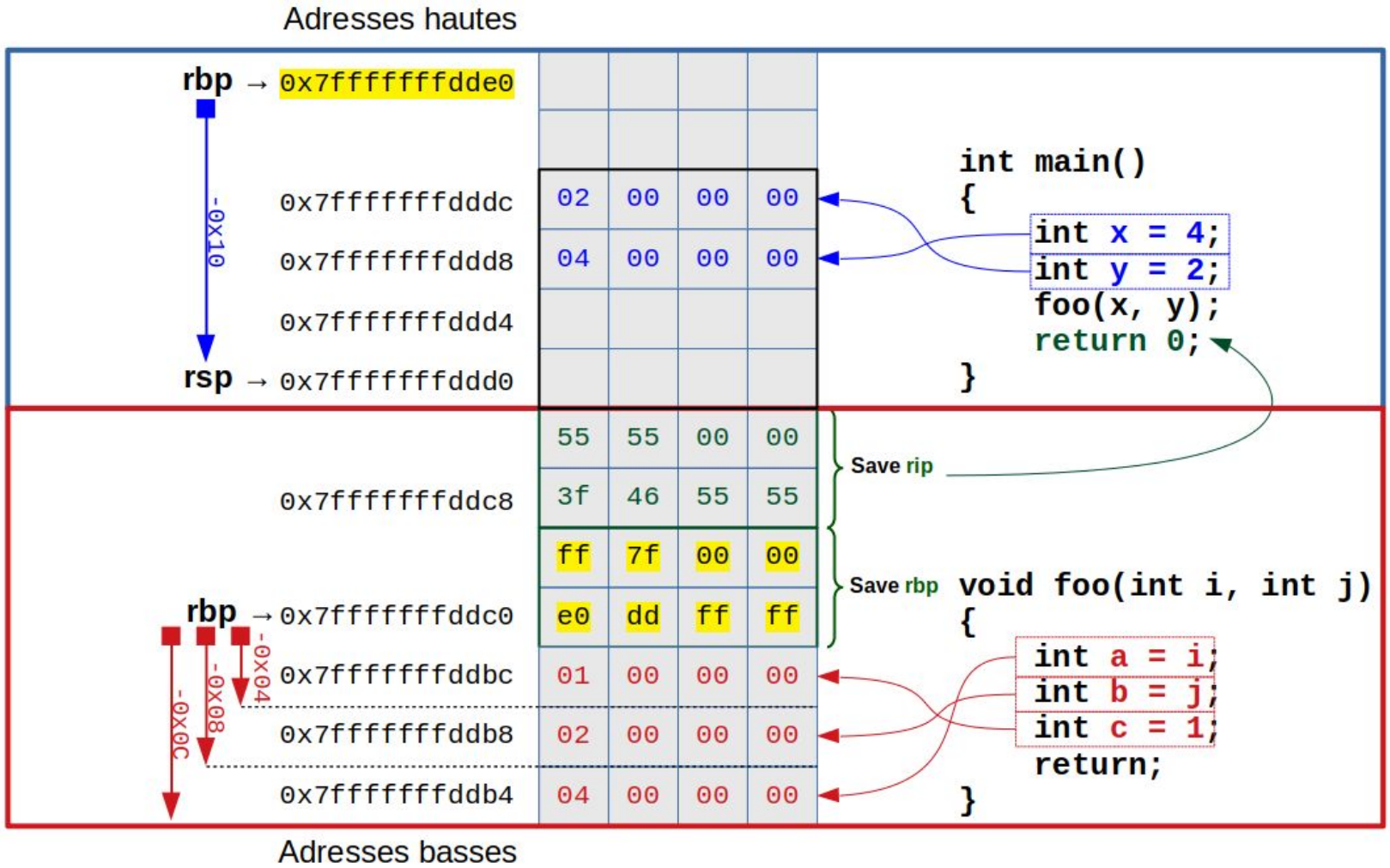
VI. Notions de base : appel de fonction

3. le **retour** de la fonction : il s'agit de remettre les choses telles qu'elles étaient avant l'appel de la fonction : c'est-à-dire remettre les registres **`%rbp`** et **`%rip`** dans leur état d'avant l'appel.

Tout d'abord, le registre **`%rbp`** est restauré simplement avec l'instruction **`pop %rbp`** ou avec **`leave`** pour remettre **`%rbp`** et **`%rsp`** dans leur état d'origine.

Lors de l'exécution de l'instruction **`ret`** qui marque la fin de la fonction, le processeur récupère l'adresse de retour (pour la restaurer dans **`%rip`**) qu'il a précédemment stockée dans la pile d'exécution et le processus pourra continuer son exécution à cette adresse.


Adresses basses
0x7fffffffddb4
0x7fffffffddb8
0x7fffffffddbc
rbp → 0x7fffffffddc0
0x7fffffffddc8
04 00 00 00
02 00 00 00
01 00 00 00
e0 dd ff ff
ff 7f 00 00
3f 46 55 55
55 55 00 00
-0x0C
-0x08
-0x04
Save rbp
Save rip
void foo(int i, int j)
{
int a = i;
int b = j;
int c = 1;
return;
}
rsp → 0x7fffffffddd0
0x7fffffffddd4
0x7fffffffddd8
0x7fffffffdddc
rbp → 0x7fffffffdde0
-0x10
04 00 00 00
02 00 00 00
int main()
{
int x = 4;
int y = 2;
foo(x, y);
return 0;
}
Adresses hautes


Adresses hautes
rbp → 0x7fffffffdde0
-0X10
0x7fffffffdddc
0x7fffffffddd8
0x7fffffffddd4
rsp → 0x7fffffffddd0
02 00 00 00
04 00 00 00
int main()
{
int x = 4;
int y = 2;
foo(x, y);
return 0;
}
55 55 00 00
3f 46 55 55
ff 7f 00 00
e0 dd ff ff
Save rip
Save rbp
0x7fffffffddc8
rbp → 0x7fffffffddc0
-0X04
-0X08
-0X0C
0x7fffffffddbc
0x7fffffffddb8
0x7fffffffddb4
01 00 00 00
02 00 00 00
04 00 00 00
void foo(int i, int j)
{
int a = i;
int b = j;
int c = 1;
return;
}
Adresses basses

Exemple n°2 : appel de fonction

```
$ vim exemple2.c
```

```
void foo(int i, int j)
{
    int a = i;
    int b = j;
    int c = 1;
    return;
}

int main()
{
    foo(4, 2);
    return 0;
}
```

```
$ gcc -g exemple2.c
```

Manipulations : appel de fonction

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000055555555461a <+0>:   push   %rbp
   0x000055555555461b <+1>:   mov    %rsp,%rbp
=> 0x000055555555461e <+4>:   mov    $0x2,%esi
   0x0000555555554623 <+9>:   mov    $0x4,%edi
   0x0000555555554628 <+14>:  callq  0x5555555545fa <foo>
   0x000055555555462d <+19>:  mov    $0x0,%eax
   0x0000555555554632 <+24>:  pop    %rbp
   0x0000555555554633 <+25>:  retq
End of assembler dump.
```

Manipulations : appel de fonction

```
(gdb) disassemble foo
Dump of assembler code for function foo:
   0x00005555555545fa <+0>:    push   %rbp
   0x00005555555545fb <+1>:    mov    %rsp,%rbp
   0x00005555555545fe <+4>:    mov    %edi,-0x14(%rbp)
   0x0000555555554601 <+7>:    mov    %esi,-0x18(%rbp)
   0x0000555555554604 <+10>:   mov    -0x14(%rbp),%eax
   0x0000555555554607 <+13>:   mov    %eax,-0xc(%rbp)
   0x000055555555460a <+16>:   mov    -0x18(%rbp),%eax
   0x000055555555460d <+19>:   mov    %eax,-0x8(%rbp)
   0x0000555555554610 <+22>:   movl   $0x1,-0x4(%rbp)
   0x0000555555554617 <+29>:   nop
   0x0000555555554618 <+30>:   pop    %rbp
   0x0000555555554619 <+31>:   retq
End of assembler dump.
```

Manipulations : appel de fonction

```
(gdb) print $rbp
$6 = (void *) 0x7fffffffddd0
(gdb) print $rsp
$7 = (void *) 0x7fffffffddd0
(gdb) print i
$8 = 4
(gdb) print &i
$9 = (int *) 0x7fffffffddbc
(gdb) print j
$10 = 2
(gdb) print &j
$11 = (int *) 0x7fffffffddb8
(gdb) print &a
$13 = (int *) 0x7fffffffddc4
(gdb) print &b
$14 = (int *) 0x7fffffffddc8
(gdb) print &c
$15 = (int *) 0x7fffffffddcc
```

Exemple : les arguments

```
$ vim exemple1.c
#include <stdlib.h>
#include <string.h>
int var1;
char var2[] = "buf1";
int main(int argc, char **argv) {
    int var3 = 3;
    static int var4;
    static char var5[] = "buf2";
    char * var6;
    var6 = malloc(512);
    strcpy(var6, "hello");
    return 0;
}
```

Manipulations : les arguments

```
$ gcc -g exemple1.c

$ gdb -q --args ./a.out azertyuiop qsdfghjklm
# Ou :
(gdb) set args azertyuiop qsdfghjklm
(gdb) break 13
(gdb) run

(gdb) info args
argc = 3
argv = 0x7fffffffde98

(gdb) print *argv@argc
$1 = {0x7fffffffe1fa "/home/tv/Téléchargements/buffer-overflow/a.out",
0x7fffffffe22b "azertyuiop", 0x7fffffffe236 "qsdfghjklm"}
```

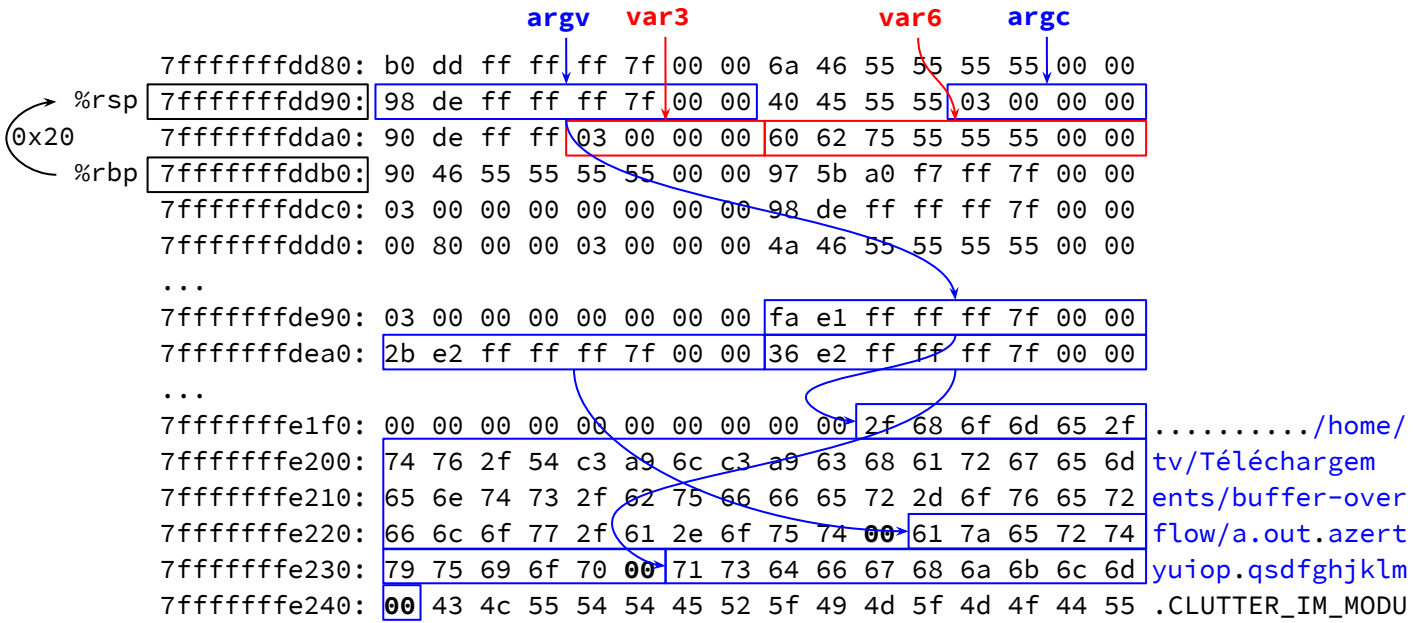
Manipulations : les arguments

```
(gdb) print &argv
$27 = (char ***) 0x7fffffffdd90

(gdb) print &argv[0]
$28 = (char **) 0x7fffffffde98
(gdb) print &argv[1]
$29 = (char **) 0x7fffffffdea0
(gdb) print &argv[2]
$30 = (char **) 0x7fffffffdea8

(gdb) print argv[0] // ou print *argv
$32 = 0x7fffffffe1fa "/home/tv/Téléchargements/buffer-overflow/a.out"
(gdb) print argv[1] // ou print *(argv+1)
$33 = 0x7fffffffe22b "azertyuiop"
(gdb) print argv[2]
$34 = 0x7fffffffe236 "qsdfghjklm"
```

Manipulations : les arguments

```
(gdb) disassemble
Dump of assembler code for function main:
   0x000055555554464a <+0>:     push   %rbp
   0x000055555554464b <+1>:     mov    %rsp,%rbp
   0x000055555554464e <+4>:     sub    $0x20,%rsp
   0x0000555555554652 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000555555554655 <+11>:    mov    %rsi,-0x20(%rbp)
=> 0x0000555555554659 <+15>:    movl   $0x3,-0xc(%rbp)
   0x0000555555554660 <+22>:    mov    $0x200,%edi
   0x0000555555554665 <+27>:    callq  0x555555554520 <malloc@plt>
   0x000055555554466a <+32>:    mov    %rax,-0x8(%rbp)
   0x000055555554466e <+36>:    mov    -0x8(%rbp),%rax
   0x0000555555554672 <+40>:    movl   $0x6c6c6568,(%rax)
   0x0000555555554678 <+46>:    movw   $0x6f,0x4(%rax)
   0x000055555554467e <+52>:    mov    $0x0,%eax
   0x0000555555554683 <+57>:    leaveq
   0x0000555555554684 <+58>:    retq
End of assembler dump.
```

La pile


argv
var3
var6
argc
%rsp
0x20
%rbp
7fffffffdd80: b0 dd ff ff ff 7f 00 00 6a 46 55 55 55 55 00 00
7fffffffdd90: 98 de ff ff ff 7f 00 00 40 45 55 55 03 00 00 00
7fffffffdda0: 90 de ff ff 03 00 00 00 60 62 75 55 55 55 00 00
7fffffffddb0: 90 46 55 55 55 55 00 00 97 5b a0 f7 ff 7f 00 00
7fffffffddc0: 03 00 00 00 00 00 00 00 98 de ff ff ff 7f 00 00
7fffffffddd0: 00 80 00 00 03 00 00 00 4a 46 55 55 55 55 00 00
...
7fffffffde90: 03 00 00 00 00 00 00 00 fa e1 ff ff ff 7f 00 00
7fffffffdea0: 2b e2 ff ff ff 7f 00 00 36 e2 ff ff ff 7f 00 00
...
7fffffffe1f0: 00 00 00 00 00 00 00 00 00 00 2f 68 6f 6d 65 2f ..../home/
7fffffffe200: 74 76 2f 54 c3 a9 6c c3 a9 63 68 61 72 67 65 6d tv/Téléchargem
7fffffffe210: 65 6e 74 73 2f 62 75 66 66 65 72 2d 6f 76 65 72 ents/buffer-over
7fffffffe220: 66 6c 6f 77 2f 61 2e 6f 75 74 00 61 7a 65 72 74 flow/a.out.azert
7fffffffe230: 79 75 69 6f 70 00 71 73 64 66 67 68 6a 6b 6c 6d yuiop.qsdfghjklm
7fffffffe240: 00 43 4c 55 54 54 45 52 5f 49 4d 5f 4d 4f 44 55 .CLUTTER_IM_MODU

Manipulations : des informations

```
(gdb) info locals
var3 = 3
var4 = 0
var5 = <optimized out>
var6 = 0x555555756260 ""
(gdb) info stack
#0  main (argc=3, argv=0x7fffffffde98) at exemple1.c:14
(gdb) info frame
Stack level 0, frame at 0x7fffffffddc0:
 rip = 0x55555555466e in main (exemple1.c:14); saved rip = 0x7ffff7a05b97
 source language c.
 Arglist at 0x7fffffffddb0, args: argc=3, argv=0x7fffffffde98
 Locals at 0x7fffffffddb0, Previous frame's sp is 0x7fffffffddc0
 Saved registers:
  rbp at 0x7fffffffddb0, rip at 0x7fffffffddb8
```

Exemple n°3 : le côté obscur

```
$ vim exemple3.c
```

```
void bar() {
    printf("bar()\n");
    return;
}

void foo(int i, int j) {
    int a = i;
    int b = j;
    int c = 1;
    printf("foo()\n");
    return;
}

int main() {
    printf("@bar = %p\n", &bar);
    printf("@foo = %p\n", &foo);
    foo(4, 2);
    return 0;
}
```

Manipulations : le côté obscur

```
Dump of assembler code for function main:
   0x0000555555554697 <+0>:push   %rbp
   0x0000555555554698 <+1>:mov    %rsp,%rbp
   0x000055555555469b <+4>:lea    -0x58(%rip),%rsi        # 0x55555555464a <bar>
   0x00005555555546a2 <+11>:    lea    0xf7(%rip),%rdi        # 0x5555555547a0
   0x00005555555546a9 <+18>:    mov    $0x0,%eax
   0x00005555555546ae <+23>:    callq  0x555555554520 <printf@plt>
   0x00005555555546b3 <+28>:    lea    -0x58(%rip),%rsi        # 0x555555554662 <foo>
   0x00005555555546ba <+35>:    lea    0xea(%rip),%rdi        # 0x5555555547ab
   0x00005555555546c1 <+42>:    mov    $0x0,%eax
   0x00005555555546c6 <+47>:    callq  0x555555554520 <printf@plt>
   0x00005555555546cb <+52>:    mov    $0x2,%esi
   0x00005555555546d0 <+57>:    mov    $0x4,%edi
   0x00005555555546d5 <+62>:    callq  0x555555554662 <foo>
   0x00005555555546da <+67>:    mov    $0x0,%eax
   0x00005555555546df <+72>:    pop    %rbp
   0x00005555555546e0 <+73>:    retq
End of assembler dump.
```

Manipulations : le côté obscur

```
(gdb) x/16x 0x7fffffffddd0
0x7fffffffddd0:     0xe0 0xdd 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffddd8:     0xda 0x46 0x55 0x55 0x55 0x55 0x00 0x00
(gdb) print *0x7fffffffddd8=0x5555464a
$3 = 1431651914
(gdb) x/16x 0x7fffffffddd0
0x7fffffffddd0:     0xe0 0xdd 0xff 0xff 0xff 0x7f 0x00 0x00
0x7fffffffddd8:     0x4a 0x46 0x55 0x55 0x55 0x55 0x00 0x00
(gdb) step
foo (i=4, j=2) at exemple3.c:15
15          return;
(gdb) step
bar () at exemple3.c:4
(gdb) step
5           printf("bar()\n");
```

La suite au prochain épisode ...

Voyage au coeur d'un programme exécutable

Episode 3 - buffer overflow