# Compilation : librairies, sécurité

[L3 Informatique] Théorie des langages et compilation

Gaétan Richard (avec des éléments de Florent Madelaine)

2024-2025

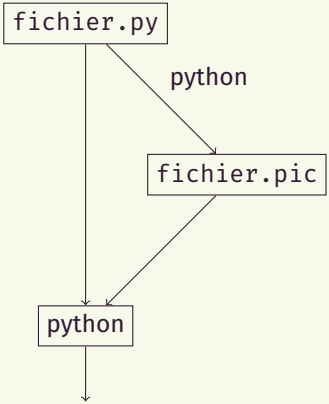


# Rappels

# Du source à l'exécution

Cas des langages interprétés :

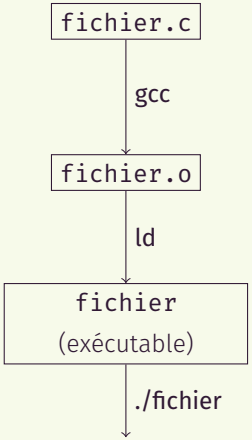
- le fichier source contient le code du programme ;
- on convertit le langage vers un langage intermédiaire ;
- un interpréteur se charge alors d'exécuter le source et de gérer les fonctions de bibliothèques.



# Du source à l'exécution

Cas des langages compilés :

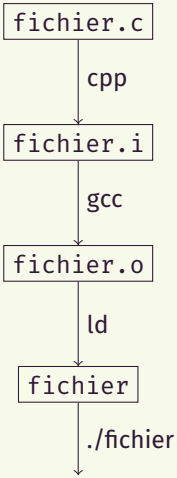
- le fichier source contient le code du programme ;
- on compile le fichier pour produire du code exécutable ;
- on fait le liens avec les fonctions de bibliothèques (linkage) ;
- on lance le programme.



# Du source à l'exécution

## Cas des langages compilés avec précompilation

- le fichier source contient le code du programme ;
- on compile le fichier pour produire du code exécutable ;
- on fait le liens avec les fonctions de bibliothèques (linkage) ;
- on lance le programme.



# Format exécutable

# C’est quoi exactement un binaire exécutable ?

Tout d’abord il y a différent formats standardisés selon le système d’exploitation :

- Mach-O (Mach Object) sous `macOS`, `iOS`;
- ELF (Executable and Linkable Format) pour `Unix`
- PE (portable executable) sous `Windows`

# ELF : format exécutable et liable

> *ELF [...] est un format de fichier binaire utilisé pour l'enregistrement de code compilé (objets, exécutables, bibliothèques de fonctions). Il a été développé par l'USL (Unix System Laboratories) pour remplacer les anciens formats* `a.out` *et* `COFF` *qui avaient atteint leurs limites. Aujourd'hui, ce format est utilisé dans la plupart des systèmes d'exploitation de type Unix (GNU/Linux, Solaris, IRIX, System V, BSD), à l'exception de Mac OS X.*

(Wikipedia)

# Exemple

```
    .file  "tiny.c"
  .text
  .globl  main
  .type  main, @function
main:
.LFB0:
  .cfi_startproc
  pushl  %ebp
  .cfi_def_cfa_offset 8
  .cfi_offset 5, -8
  movl  %esp, %ebp
  .cfi_def_cfa_register 5
  movl  $42, %eax
  popl  %ebp
  .cfi_def_cfa 4, 4
  .cfi_restore 5
  ret
  .cfi_endproc
.LFE0:
  .size  main, .-main
  .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
  .section  .note.GNU-stack,"",@progbits
```

# Mais ce n'est pas du binaire ?

En pratique, ce fichier est vraiment codé en binaire.

On peut toutefois à l'aide d'utilitaires passer du format textuel au format binaire et inversement.

# Que veut dire format liable ?

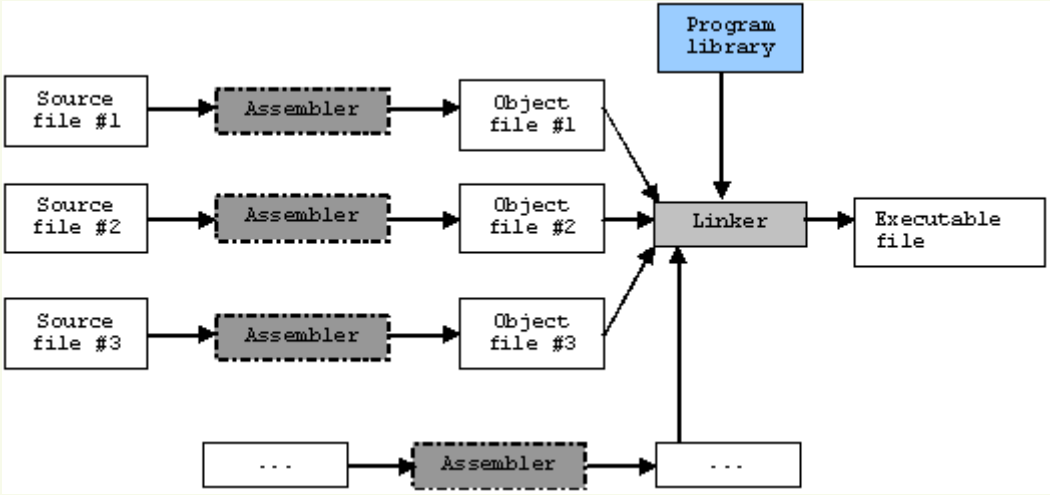
### Grand principe

on ne va pas recompiler le même code plusieurs fois.

### Idée

au lieu de recopier du code de librairie dans notre code puis de compiler le tout ensemble, on va pré-assembler et lier le tout au niveau du binaire (avec des fichiers ELF bas-niveau).

# Illustration



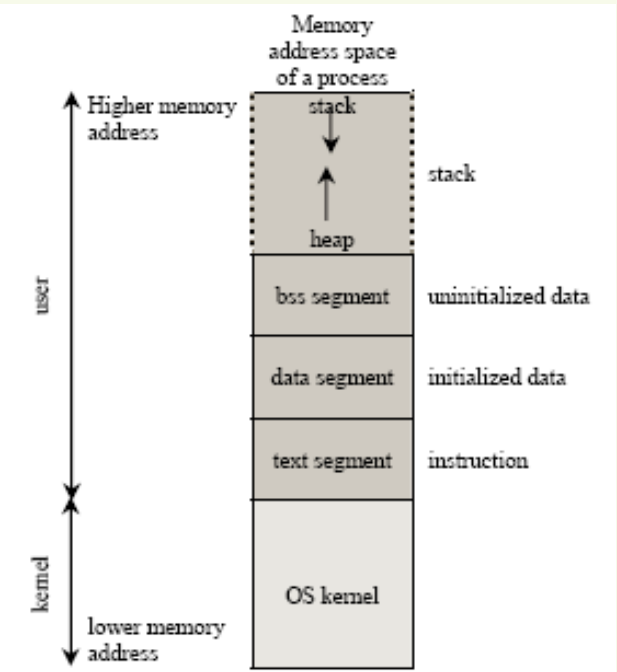
### Validation de la mémoire et de l'accès

Le noyau lit les informations dans l'en-tête du programme et valide ou pas le droit d'accès, la demande de mémoire, etc puis calcule les besoins en mémoire.

### Mise en place du process

allocation de mémoire, recopie les sections `.text` (le code) et `.data` (données initialisées) en mémoire, initialise les registres, saute à l'instruction de départ.

# Illustration



# Bibliothèques

# Les bibliothèques

### Grand principe

ne pas réinventer la roue.

### Idée

mise en commun et normalisation / standardisation de code.

### Exemple

librairie standard C.

# Comment charger une librairie ?

Il y a plusieurs façons :

- statiquement (mais sans recompiler) en assemblant les binaires (static library).
- partagée en assemblant leur binaires au moment de l'exécution (shared library).
- dynamiquement en assemblant leur binaires au besoin durant l'exécution (dynamic library).

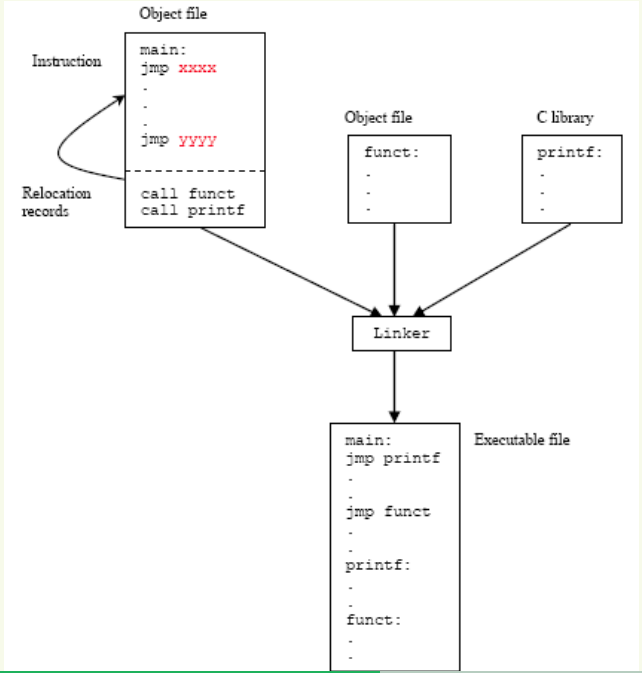
# Librairie statique

- Collection de fichiers objets
- historiquement premier type de librairies
- création avec `ar` (archiver)`$ ar rcs my_library.a file1.o file2.o`

## Avantages/Inconvénients

- + liens vers (binaire de) progs existants sans recompiler (à la préhistoire : compiler prenait beaucoup de temps).
- + Théorie : 1 à 5% plus rapide qu'autres librairies. Pratique pas forcément.
- + Utilisation pour dev qui ne veulent pas donner leurs sources.
- - Pour programmeur utilisant la librairie.
- - Prend de la place.
- - Nécessite d'être maintenu à jour.

# Illustration



# Librairie partagées

- Collection de fichiers objets
- chargée par le programme au démarrage

## Avantages/Inconvénients

- + Permet de faire une mise à jour des libraries tout en conservant le support pour des librairies plus anciennes et incompatibles
- + Cacher/remplacer une librairie lors de l'exécution d'un programme spécifique.
- + Exécutable moins gros
- - Il faut installer la librairie partagée (et dans la bonne version)!

## Plusieurs noms

- soname est un lien symbolique vers le real name (ex : `libnetpbm.so.10 -> libnetpbm.so.10.0` )
- real name est un fichier avec le code assemblé (ex : `libnetpbm.so.10.0`)
- linker name (soname sans la versions) pointe vers le dernier soname (ex : `libnetpbm.so`). Dans les paquages debian, ce fichier est dans le paquet libX-dev avec les `.h` et `.a`.

## Connaître les dependances d'un programme

En général on en a au moins 2 : + `/lib/ld-linux.so.N` chargement de librairies; + `/libc.so.N` libc.

```
$ ldd /bin/chmod
  linux-gate.so.1 =>  (0xb76eb000)
  libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7525000)
  /lib/ld-linux.so.2 (0xb76ec000)
```

ATTENTION

Ne pas utiliser ldd sur un programme en lequel vous n'avez pas confiance ! (voir man ldd)

### Changement de soname / version

Un changement de numéro majeur soname indique une possible incompatibilité au niveau de l'ABI (Application Binary Interface).

# Librairies chargée dynamiquement

- Collection de fichiers objets
- Chargement pas nécessairement au démarrage du programme. e.g. chargement *plugin* lorsque nécessaire.

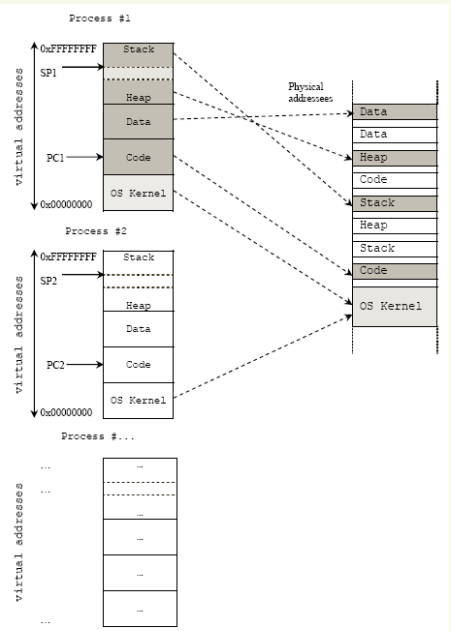
## Sous linux

- format identique : `.o` ou `.so`
- différence : utilisation d'une API spéciale via lib `<dlfcn.h>` [ dlopen, dlclose, dlsym (qui va chercher symboles dans librairies) et dlerror ]

# Exemple

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <dlfcn.h>
4
5  int main(int argc, char \textcolor{gahelp}{}argv) {
6    void *handle;
7    double (*cosine)(double);
8    char *error;
9
10   handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
11   if (!handle) { fputs (dlerror(), stderr);  exit(1); }
12
13   cosine = dlsym(handle, "cos");
14   if ((error = dlerror()) != NULL)  { fputs(error, stderr);  exit(1);}
15
16   printf ("%f\n", (*cosine)(2.0));
17   dlclose(handle);
18 }
```

# Illustration



# Compilation séparée en C (par l'exemple)

# Outils

- le préprocesseur (cpp);
- le compilateur (gcc);
- l'assembleur (as);
- le linker (ld).

# Exemple (1/3)

Fichier `Exf.c`

```c
int f (int x) {
  return 2*x;
}
```

Fichier `Exf.h`

```c
int f (int i);
```

# Exemple (2/3)

Fichier Exg.c

```c
#include "Exf.h"

int g (int i,int j) {
  while (i < j+1) i=f(i);
  return j;
}
```

Fichier Exg.h

```c
#include "Exf.h"

int g (int i,int j);
```

# Exemple (3/3)

Fichier Exmain.c

```c
#include <stdio.h>
#include "Exg.h"

int main ( void ) {
  int i;
  i = g(3,4);
  printf("%d",i);
}
```

### Commandes

```
cpp Exg.c -o Exg.i; cpp Exmain.c -o Exmain.i
```

### Résultat

Interprétation des directives

# Génération du code assembleur

## Commandes

`gcc -S Exf.c`, `gcc -S Exg.i`, `gcc -S Exmain.i`

## Résultat

```
      .file  "Exf.c"
        .text
        .globl  f
        .type  f, @function
f:
.LFB0:
        .cfi_startproc
        pushq  %rbp
<...>
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size  f, .-f
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section  .note.GNU-stack,"",@progbits
```

### Commande

```
gcc -c Exf.s
```

### Résultat

Un fichier binaire Exf.o que l'on peut analyser avec **objdump** :

- `objdump -f Exf.o` donne la table des symbole ;
- `objdump -d Exf.o` visualise le code binaire.

# Création d'une librairie statique

### Commande

```
ar rcs libEx.a Exf.o Exg.o
```

### Résultat

Une archive ar contenant les deux fichiers. Cette archive peut être analysée par objdump

### Programme

The GNU linker ld (via `collect2`)

### Commande (simplifiée)

```
ld --build-id -m elf_x86_64 --hash-style=gnu --as-needed -static -z
relro /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o
-L/usr/lib/gcc/x86_64-linux-gnu/9 Exmain.o libEx.a --start-group -lgcc
-lgcc_eh -lc --end-group /usr/lib/x86_64-linux-gnu/crtn.o
```

### Résultat

l'exécutable (avec tout le code).

```
$ ls -lh a.out
-rwxr-xr-x 1 richardg admin 852K févr. 23 11:17 a.out
```

# Buffer overflows

### Principe général

profiter que le programme ne vérifie pas que la longueur des données saisies dans le tampon pour écraser des données dans l'espace d'adressage du processus et lui faire exécuter par exemple du code malicieux qu'on injectera.

### Plusieurs approches

- stack overflow (débordement dans la pile)
- heap overflow (débordement dans le tas)
- integer overflow (index en dehors d'un tableau, pas seulement trop grand, peut être aussi un entier négatif là où on attend normalement un entier positif)

Note : Les pages suivantes (en particulier les figures) sont tirées du site http://www.tenouk.com/Bufferoverflowc/.

# Dans quelles conditions peut-on avoir ce problème ?

- Pas de sûreté du typage

  Exemple : en C et C++ il y a des fonctions de la librairie standard (gestion mémoire, manipulation de chaînes de caractères) qui ne vérifient ni la taille des tableaux, ni les types (c'est au programmeur d'utiliser les versions sûres de ces fonctions ou bien de faire les vérifications).

- Accès ou copie d'un tampon sur la pile de manière non sûre

  Exemple : déclaration de 100 bytes, mais recopie de 150 bytes.

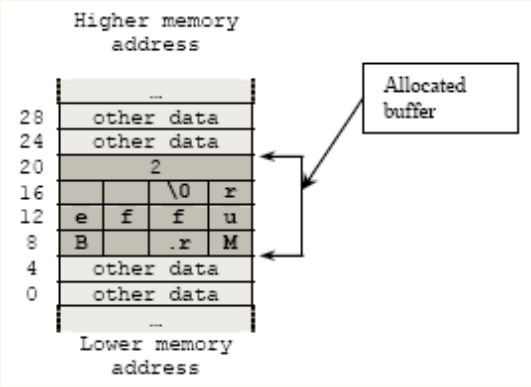
- Emplacement du tampon proche de section critique sur la pile

  Exemple : l'emplacement réservé pour le buffer est proche du bloc d'activation d'un appel de fonction.

Code

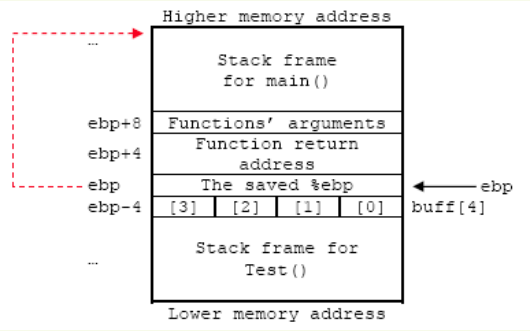
```
// in C, for string array
// it is NULL (\0) terminated
char Name[12] = "Mr. Buffer";
int num = 2;
```

Sur la pile



# Exemple

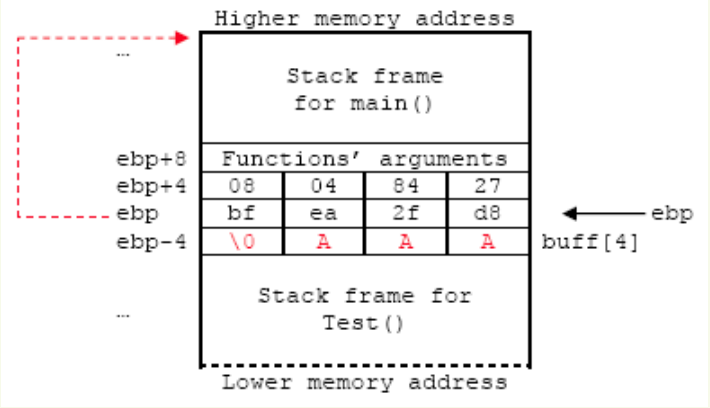
```
1  // test buffer program
2  #include <unistd.h>
3  void Test(){
4    char buff[4];
5    printf("Some input: ");
6    gets(buff);
7    puts(buff);
8  }
9
10 int main(int argc, char *argv[]){
11   Test();
12   return 0;
13 }
```



# Exemple : lorsque le tampon déborde

Saisie :

AAA



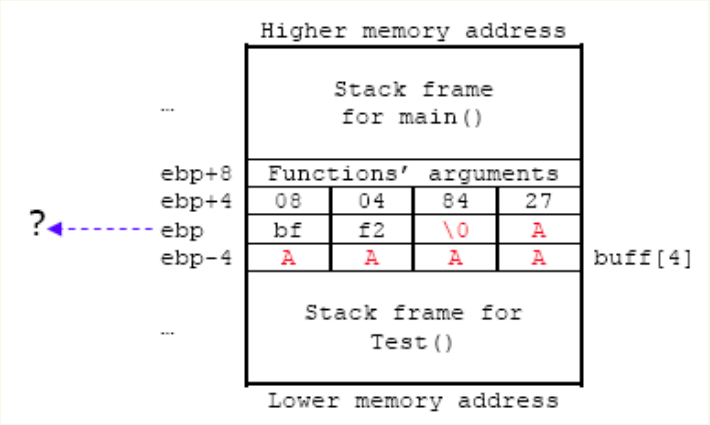
# Exemple : lorsque le tampon déborde

Saisie :

AAAAA

L'ancienne valeur du registre pile de fonction est perdue (notre notation en MVàP : `fp`, ici `ebp`).

Le tampon a débordé dans la zone critique (bloc d'activation de la fonction appelante).



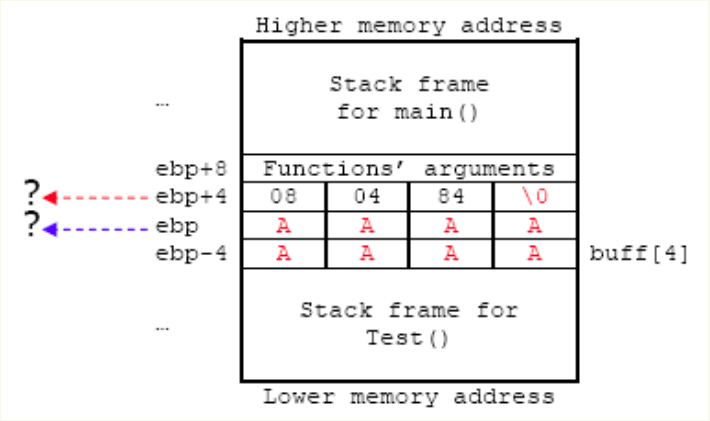
# Exemple : lorsque le tampon déborde

Saisie :

AAAAAAAA

L'adresse de retour dans le code (paramètre du RETURN pour la MVàP) est compromise.

Le tampon a débordé encore plus loin dans la zone critique (bloc d'activation de la fonction appelante).

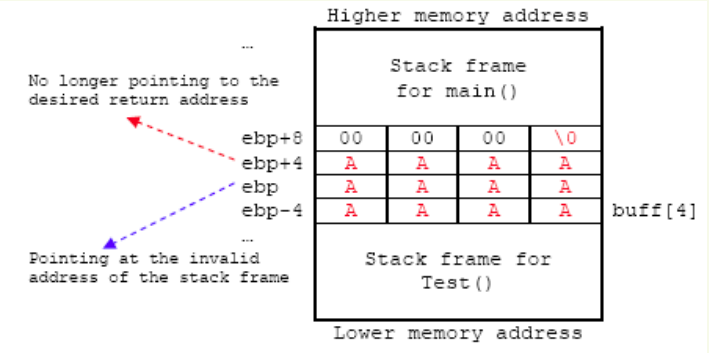


# Exemple : lorsque le tampon déborde

Saisie :

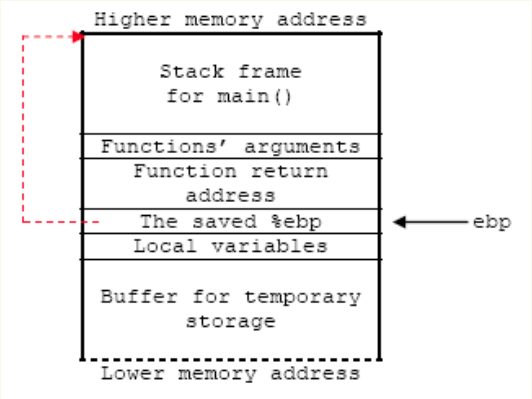
AAAAAAAAAAAA

L'adresse de retour est complètement corrompue.

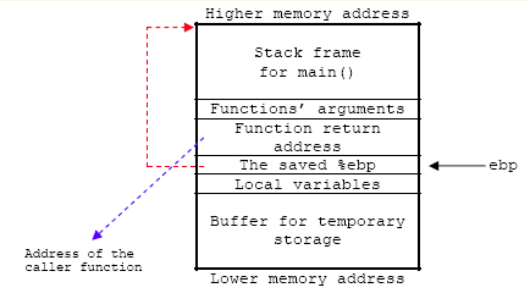


# Exemple : lorsque le tampon déborde (état de la mémoire)

Normal :

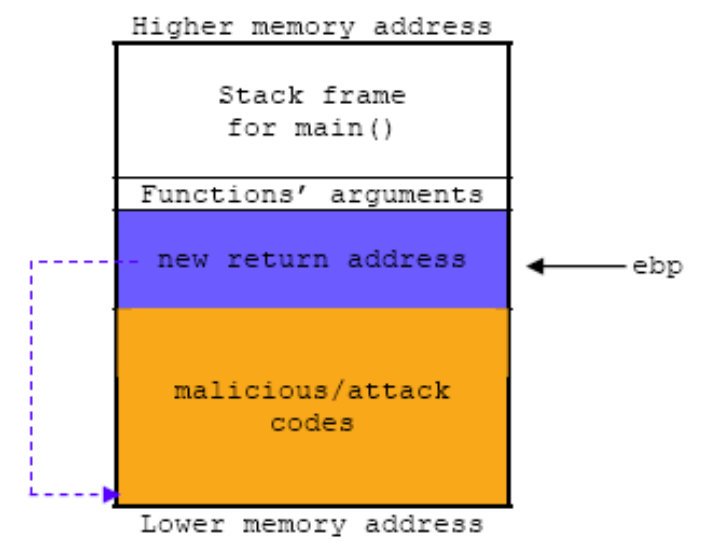



Altéré :



**Attaque par débordement de pile**

corruption adresse du code de retour redirigé vers code injecté dans buffer.



En pratique, il existe des protections et on ne peut pas forcément exécuter n'importe quelle code en mémoire.

NX bit ( *Never eXecute* )

Dissociation entre zones de mémoire contenant des instructions, donc exécutables, et zones contenant des données.

Le nom précis diffère selon le fabriquant de processeur Intel XD bit (*eXecute Disable*), AMD Enhanced Virus Protection, ARM XN (*eXecute Never*).

En présence du NX bit, on peut exécuter du code déjà présent (et donc forcément exécutable) par exemple celui de la fonction `System()` de la libc, qui permet d'exécuter un programme arbitraire.

Nom de l'attaque : return-to-libc

### Contre-mesure

address space layout randomisation (*ASLR*) qui charge les fonctions en mémoires à des positions aléatoires. Cette méthode n'est pas forcément suffisante sur un système 32-bits (seuls 16 bits sont aléatoires, une attaque par force brute est possible).

### Comment se protéger ?

Le plus simple reste toutefois de n'utiliser que des librairies sûres de C et de faire des contrôles dans les programmes.