



Rétro-ingénierie

Partie 1: Introduction

Planning

- Lundi: Introduction au reverse engineering
- Mardi: Recherche de vulnérabilités et exploitation
- Mercredi: Reverse Malware
- Jeudi & Vendredi: Exercices

Whoami

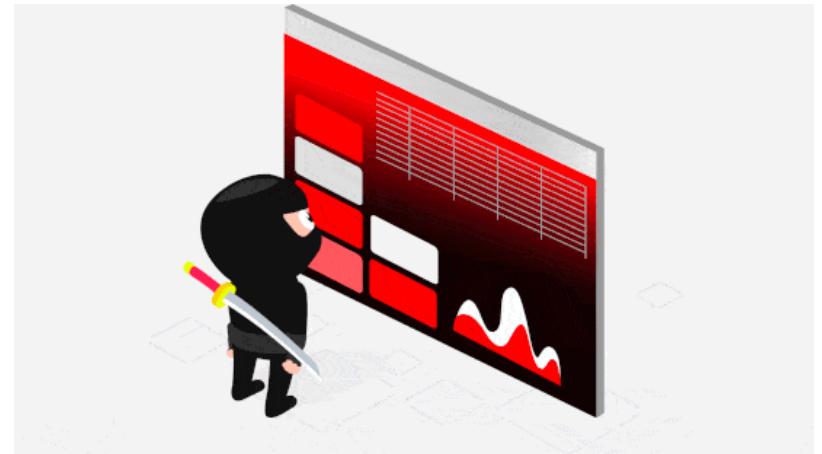


- Paul Viel (Dvorhack)
- Reverser at Synacktiv
- CTF Player Team France, HackUTT, PwnStars

SYNACKTIV



- Entreprise de cybersécurité offensive
- + de 160 ninjas
- Plusieurs pôles: Reverse, Pentest, Dev, CSIRT
- Sur 5 lieux: Paris, Rennes, Lyon, Toulouse et Lille



On Recrute !!

Sommaire

- Introduction à la rétro-ingénierie
- Architecture processeur
- Assembleur X86
- Compilation
- Architecture OS
- Format ELF

Introduction à la rétro-ingénierie

« *La rétro-ingénierie, ou ingénierie inversée, est l'activité qui consiste à étudier un objet pour en déterminer le fonctionnement interne* » (Cf.Wikipedia)

- **A des fins d'interopérabilité**
- **Analyse de code malveillant**
- **Evaluer la sécurité d'un logiciel**

A des fins d'intéropérabilité

L'interopérabilité désigne la capacité d'outils à communiquer entre eux

- **Dans les années 1980, Phoenix Technologies produit son propre BIOS compatible avec les PC IBM. Travaux issus de rétro-ingénierie du BIOS IBM.**
- **Driver Nouveau issu d'un travail de rétro-ingénierie sur les drivers NVIDIA Linux.**
- **Projet Samba sous Linux issu d'un travail de rétro-ingénierie sur le protocole (par observation des paquets réseaux)**

Analyse de logiciel malveillant

- **Identifier les IOC (Indicateurs de Compromission)**
 - Clef de registres
 - Noms de fichiers
 - Traffic réseau
 - ...
- **Créer des règles de détections basées sur le code, le comportement**
- **Corréler des attaques entre elles « le logiciel malveillant A utilise le même code que le logiciel malveillant B »**
- **Identifier les faiblesses des systèmes « le logiciel exploite une vulnérabilité inconnue à ce jour pour éléver ses privilèges »**
- **Remédiation : dans le cas d'un ransomware « est-il possible de déchiffrer les fichiers ? »**

Evaluer la sécurité d'un logiciel

- Recherche de vulnérabilité « L'application que j'ai installée fragilise-t-elle mon environnement ? »
- Vérifier les bonnes pratiques de sécurité
 - Utilisation d'algorithmes cryptographiques sûrs
 - Cloisonnement des processus sensibles
 - Sécurisation des données sensibles
 - ...



Techniques d'évaluation logiciel

Analyse comportementale

- Rapide
- Pas besoin de sources
- Peu de compétences et de moyen

Fuzzing en boite noire (envoi de données aléatoires)

- Plus ou moins rapide
- Parfois complexe à mettre en oeuvre
- Nécessite des compétences en développement

Analyse directe de binaire

- Avis éclairé
- Pas besoin de sources
- Nécessite des compétences en assembleur

Outils

- Désassembleurs / Décompilateurs

- IDA
- Ghidra
- Binary Ninja
- radare2



- Analyse périphérique

- Resources Hacker
- Wireshark
- Procmon
- Process Hacker



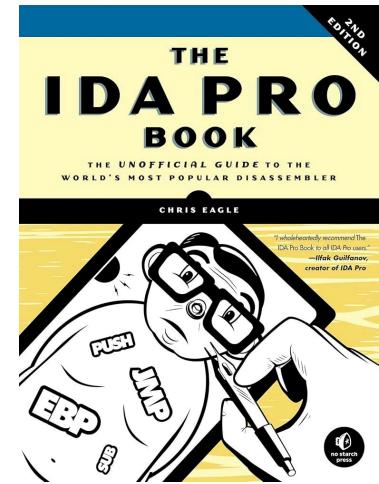
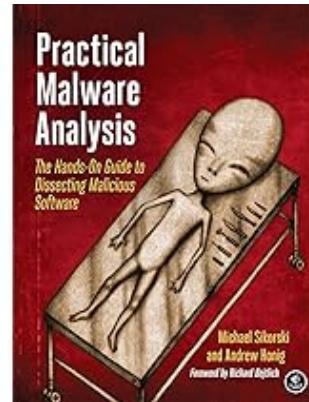
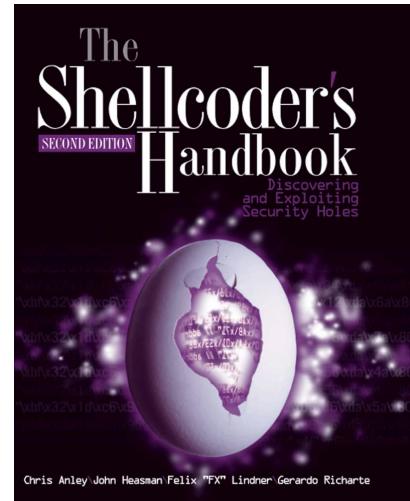
- Débuggeurs

- gdb (gef, pwndbg)
- frida

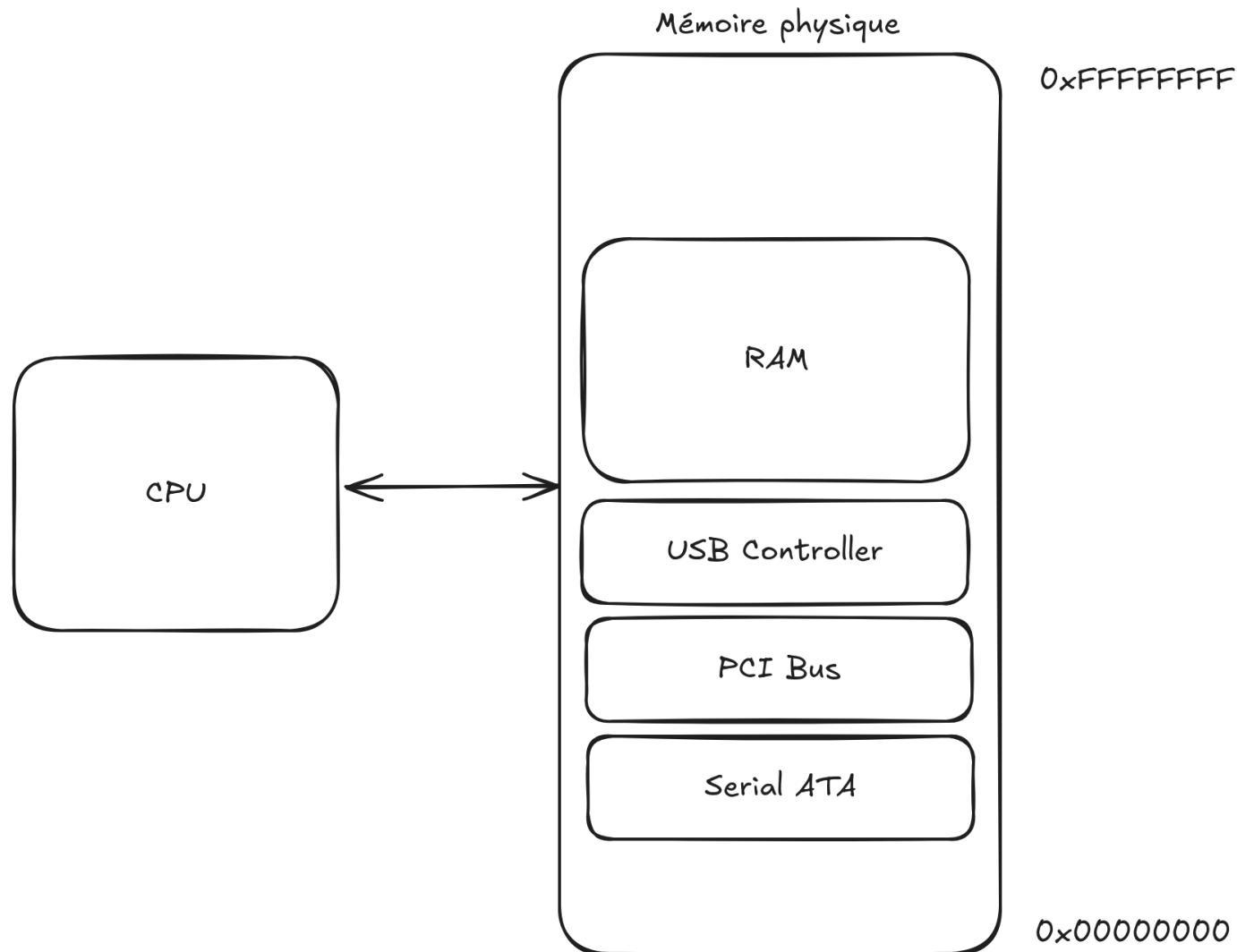


Littérature

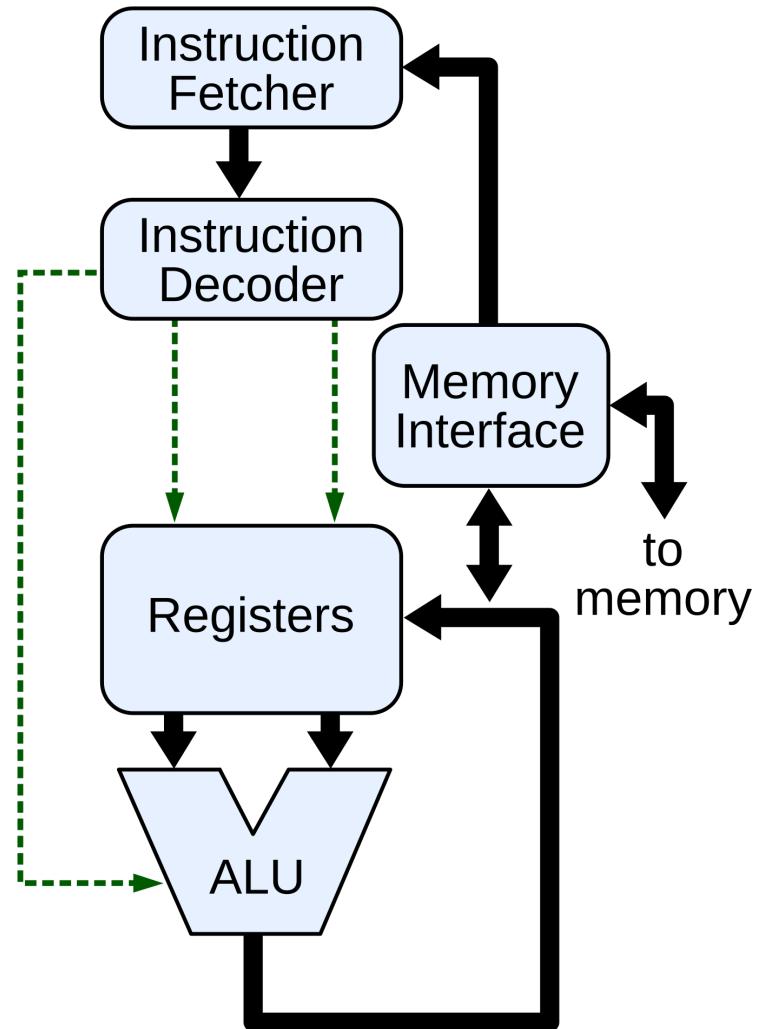
- Techniques de Hacking (Jon Ericsson)
- The IDA PRO book
- Practical Malware Analysis
- The shellcoder handbook



Architecture simplifiée d'un PC



Architecture simplifiée d'un CPU

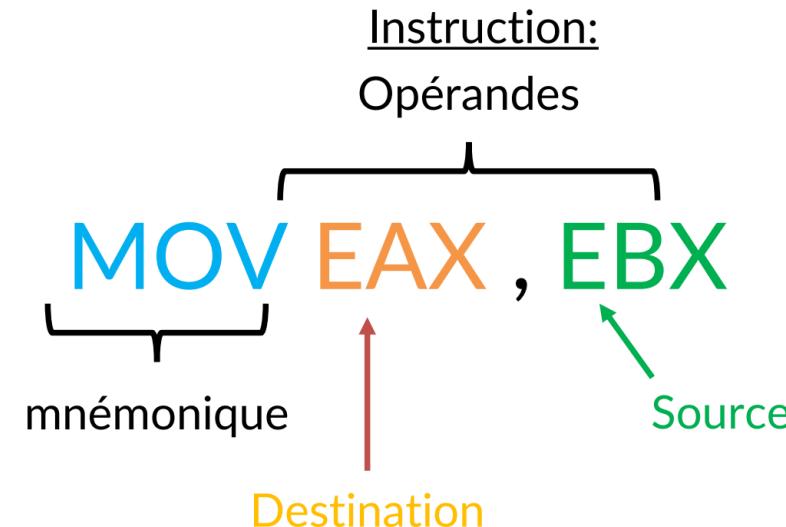


Langage assembleur

- Chaque architecture de processeur (x86, ARM, RISC-V, ...) a son propre jeu d'instructions
- Deux grandes catégories:
 - RISC: Reduced Instruction Set Computer (ex: ARM, MIPS)
 - CISC: Complex Instruction Set Computer (ex: x86)
- Le langage assembleur est un langage bas niveau qui représente le langage machine sous forme lisible par un humain

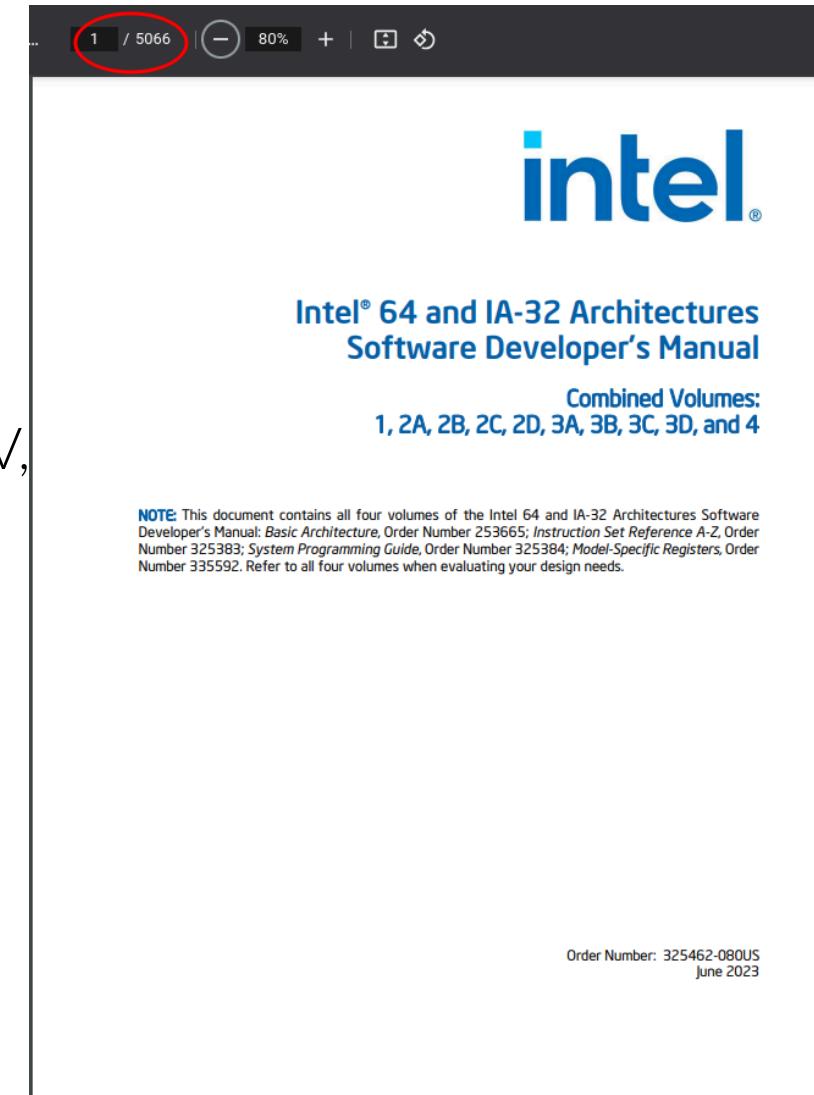
Code hexadécimal:

8B C3
 └─
 opcode



Types d'instructions

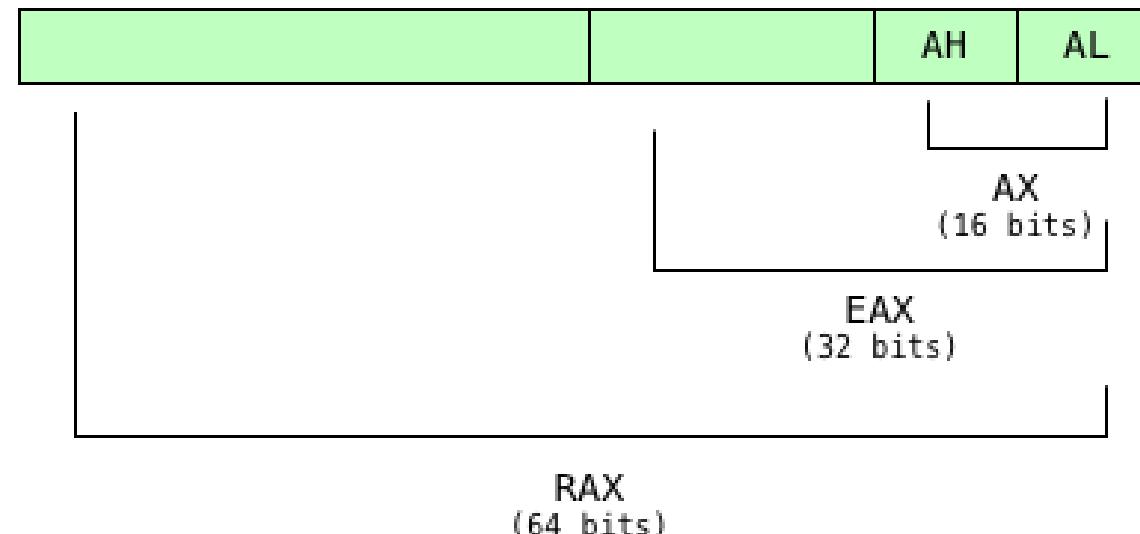
- Instructions arithmétiques (ADD, SUB)
- Instructions mouvement de données (MOV,
- Instructions logiques (AND, OR)
- Instructions branchement (JMP, CALL)
- Instructions conditionnelles (JE, JL)



Registres généraux

Un registre est un emplacement de mémoire interne à un processeur, il s'agit de la mémoire la plus rapide d'un processeur

- RAX
- RBX
- RCX
- RDX



Autres registres

- Registres d'offset
 - RBP (Base Pointer) : pointe le début de la pile
 - RSP (Stack Pointer) : pointe la fin de la pile
 - RSI
 - RDI
- Registres de segments
 - DS (Data Segment)
 - CS (Code Segment)
 - SS (Stack Segment)
 - ES, FS, GS
- Registres particuliers
 - RIP (Instruction Pointer) : pointe sur l'instruction courante
 - EFLAGS : registre indicateur d'état
 - CR0, CR2, CR3, CR4 : Registres de contrôle

Registres EFLAGS

- Constitué de drapeaux (flags)
- Chaque flag a une signification particulière
- chaque instruction peut modifier 0, 1 ou plusieurs flag

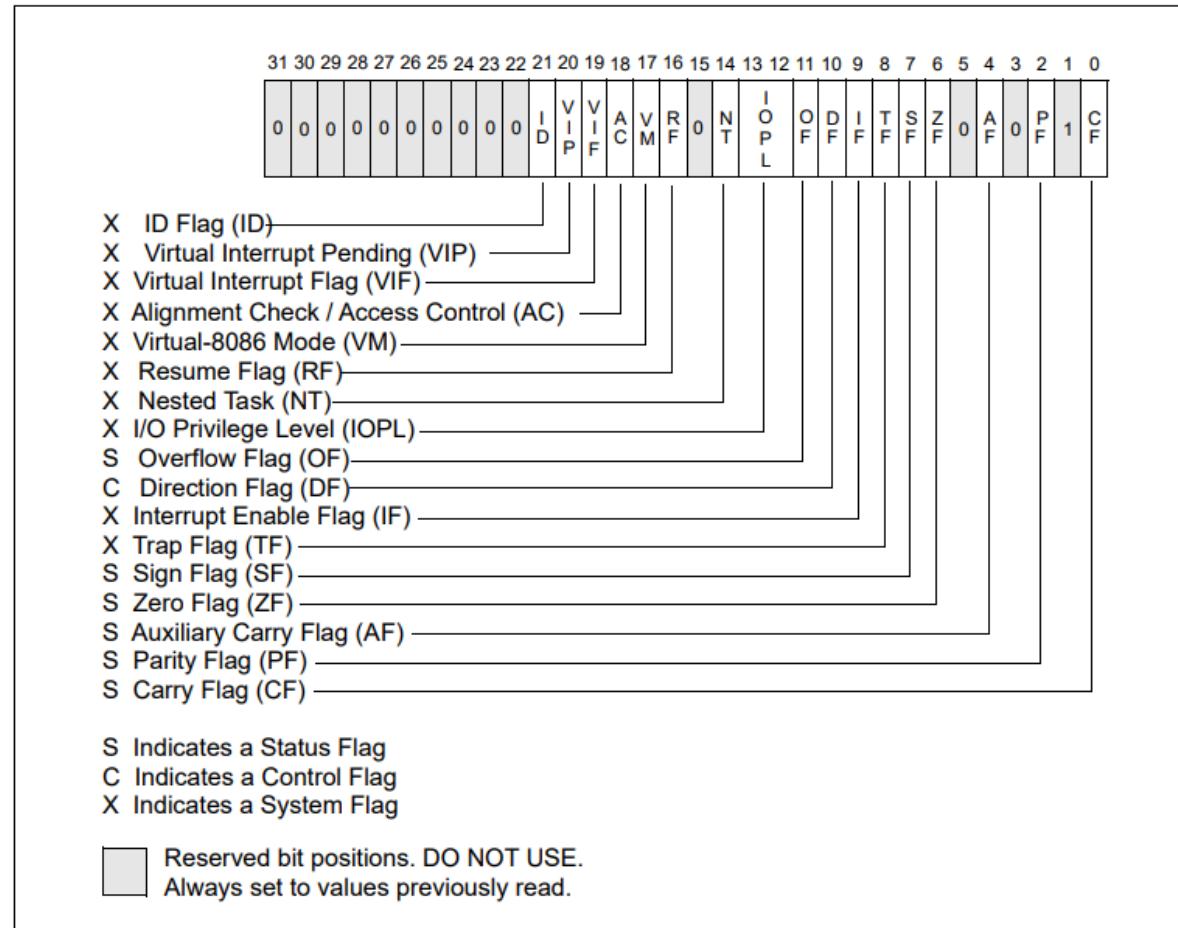


Figure 3-8. EFLAGS Register

Saut conditionnel et comparaison

- Selon les états des flags du registre EFLAGS le CPU peut décider de sauter ou non à une adresse
- Les instructions de saut ont la mnémonique suivante Jcc (cc: Condition Code)
- Example
 - JZ : Jump if Zero (ZF = 1)
 - JNZ : Jump if Not Zero (ZF = 0)
 - JA : Jump if Above (CF = 0, ZF = 0)
- CMP effectue une comparaison en soustrayant la première opérande à la seconde
- TEST effectue un ET logique

Exercice

- A la fin de ce code assembleur le registre DH vaut ?

```
XOR ECX, ECX
SUB EDX, EDX
MOV EAX, 0x0123
SUB ECX, EAX
NEG ECX
ADD EDX, ECX
```

La Pile

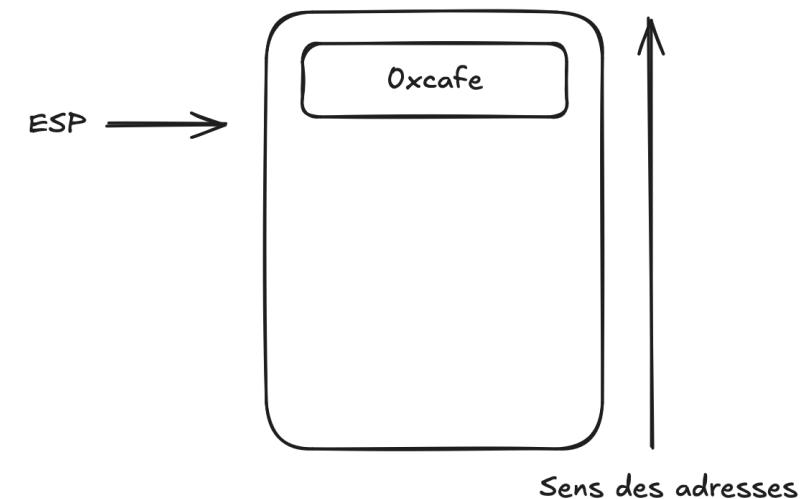
- Zone de données en mémoire vive dont l'adresse du sommet est stockée dans le registre RSP
- Cette zone est utilisée pour stocker des données temporaires (arguments passés à une fonction, variables locales d'une fonction)
- La pile grandit vers les adresses basses
- PUSH Reg/Imm : empile une valeur
- POP Reg/Imm : dépile une valeur
- CALL adresse : empile l'adresse de l'instruction suivante
- RET : dépile la donnée (adresse de retour) dans RIP

La Pile

```
/* Vous êtes ici */  
PUSH 0x1234  
POP  EAX
```

ESP = 0xffffCAF0

EAX = 0

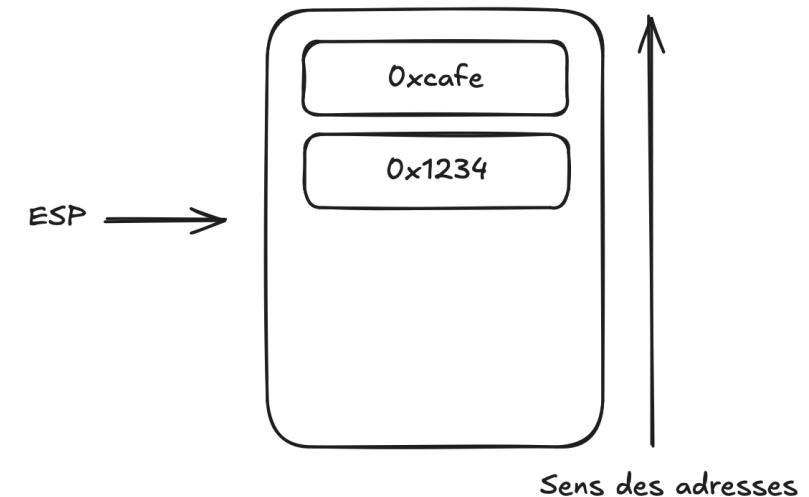


La Pile

```
PUSH 0x1234  
/* Vous êtes ici */  
POP  EAX
```

ESP = 0xffffCAEC

EAX = 0

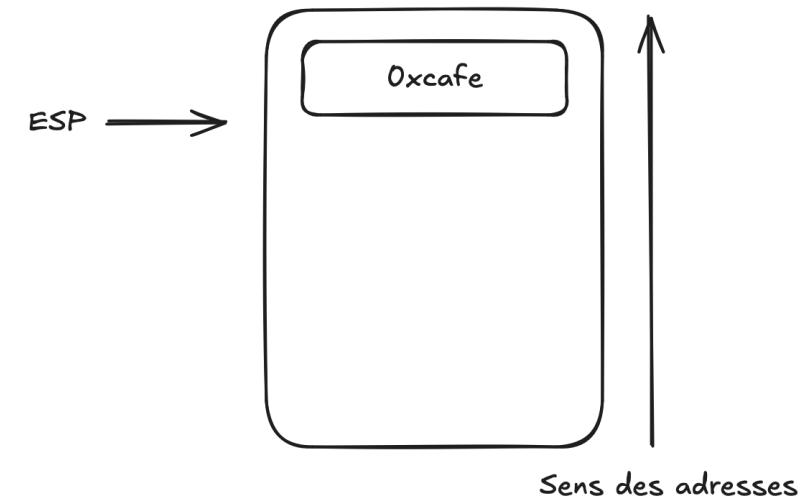


La Pile

```
PUSH 0x1234  
POP  EAX  
/* Vous êtes ici */
```

ESP = 0xffffCAFO

EAX = 0x1234



Exercice

A la fin de ce code assembleur combien vaut ECX ?

```
0: 31 c9          xor  ecx,ecx
2: ba 0b 00 00 00 mov  edx,0x9
7: e8 00 00 00 00 call 0xd
c: 90             nop
d: 58             pop  eax
e: 01 d0          add   eax,edx
10: 50            push  eax
11: c3             ret
12: 83 c1 01      add   ecx,0x1
15: 83 c1 02      add   ecx,0x2
18: 83 c1 03      add   ecx,0x3
1b: 83 c1 04      add   ecx,0x4
```

Calling convention

- Où sont les arguments lors qu'appels de fonctions ?
- 32 bits: Arguments sur la stack
 - Arg 1: [ESP]
 - Arg 2: [ESP+4]
 - Arg 3: [ESP+8]
- 64 bits: Arguments en registres
 - Arg 1: RDI
 - Arg 2: RSI
 - Arg 3: RDX

Calling convention 64 bits

- 64 bits: Arguments en registres
 - Arg 1: RDI
 - Arg 2: RSI
 - Arg 3: RDX

```
| lea      rdx, [rbp+user_input]
| mov      eax, [rbp+passwd]
| mov      rsi, rdx
| mov      edi, eax
| call    check
```

Calling convention 32 bits

- 32 bits: Arguments sur la stack
 - Arg 1: [ESP]
 - Arg 2: [ESP+4]
 - Arg 3: [ESP+8]

MOV	EAX,dword ptr [EBP + param2]
ADD	EAX,0x28
MOV	EAX,dword ptr [EAX]
PUSH	EAX
LEA	EAX=>buffer,[EBP + -0xc]
PUSH	EAX
CALL	<EXTERNAL>::strcpy

Fonction C

Code C

```
void foo(int a, int b){  
    puts(a);  
}  
  
void main(){  
    foo("test", 0xcafe);  
}
```

Code Assembleur

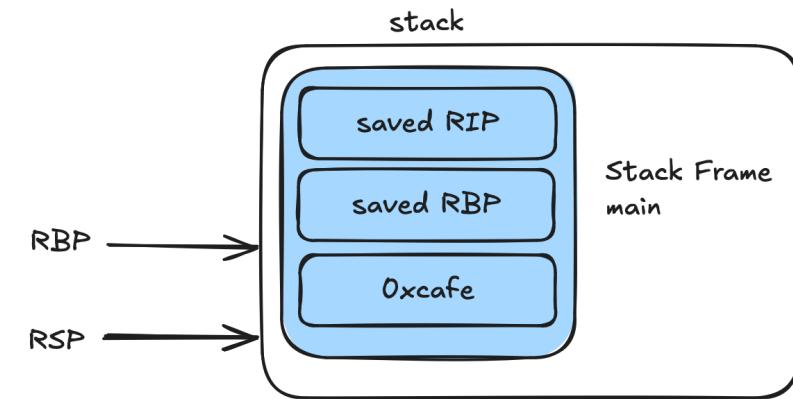
```
foo:  
    push    rbp  
    mov     rbp,rsp  
    sub     rsp,0x10  
    mov     QWORD PTR [rbp-0x8],rdi  
    mov     DWORD PTR [rbp-0xc],esi  
    mov     rax,QWORD PTR [rbp-0x8]  
    mov     rdi,rax  
    call    401030 <puts@plt>  
    nop  
    leave  
    ret  
  
main:  
    push    rbp  
    mov     rbp,rsp  
    mov     esi,0xcafe  
    lea     rax,[rip+0xeb0] // addr of "test"  
    mov     rdi,rax  
    call    401126 <foo>  
    nop  
    pop     rbp  
    ret
```

Stockage de variable

```
int global_variable = 0x1234; // in .data
int uninitialized_variable; // in .bss

void foo(){
    int a = 0xdead;
    int b = 0xbeef;
}

void main(){
    int local_variable = 0xcafe;
    /* Vous êtes ici */
    foo();
}
```

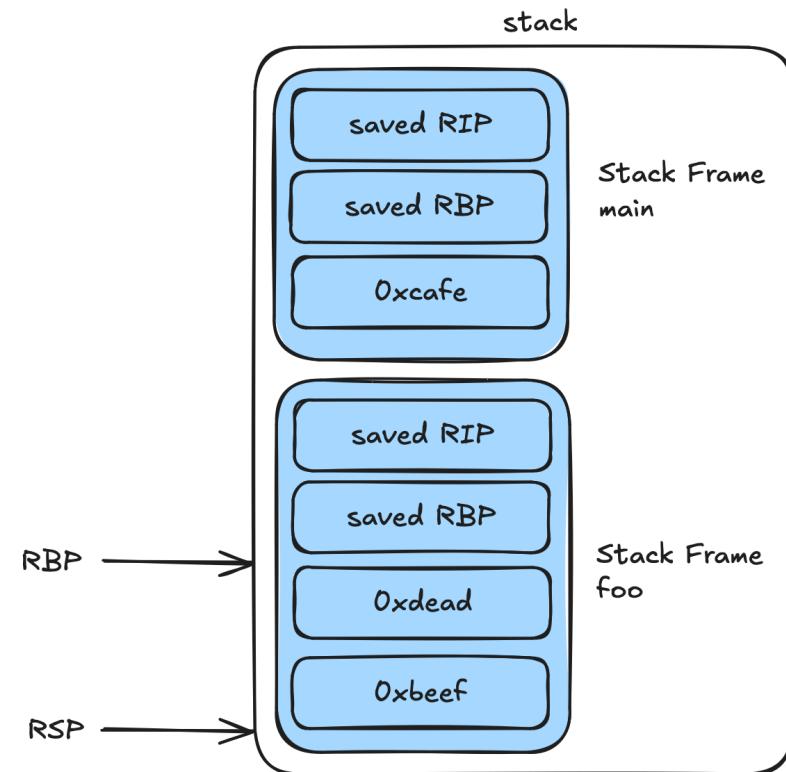


Stockage de variable

```
int global_variable = 0x1234; // in .data
int uninitialized_variable; // in .bss

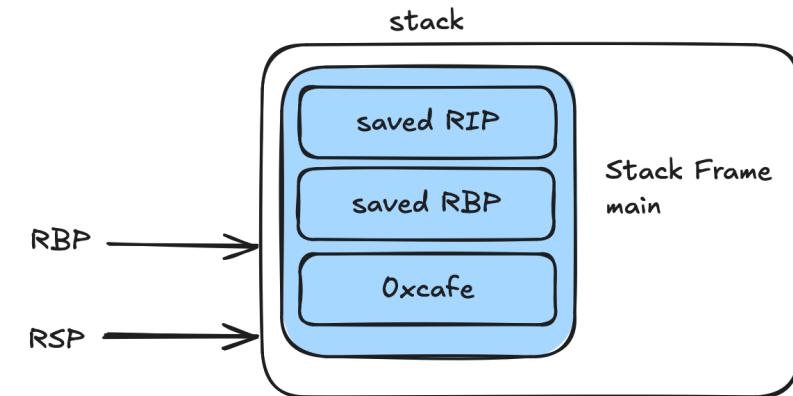
void foo(){
    int a = 0xdead;
    int b = 0xbeef;
    /* Vous êtes ici */
}

void main(){
    int local_variable = 0xcafe;
    foo();
}
```



Stockage de variable

```
0x401106 foo:  
    push    rbp  
    mov     rbp, rsp  
    mov     DWORD PTR [rbp-0x4], 0xdead  
    mov     DWORD PTR [rbp-0x8], 0xbeef  
    nop  
    pop     rbp  
    ret  
  
0x40111b main:  
    push    rbp  
    mov     rbp, rsp  
    sub     rsp, 0x10  
    mov     DWORD PTR [rbp-0x4], 0xcafe  
    mov     eax, DWORD PTR [rbp-0x4]  
    mov     edi, eax  
    mov     eax, 0x0  
    call    401106 <foo>  
    nop  
    leave  
    ret
```



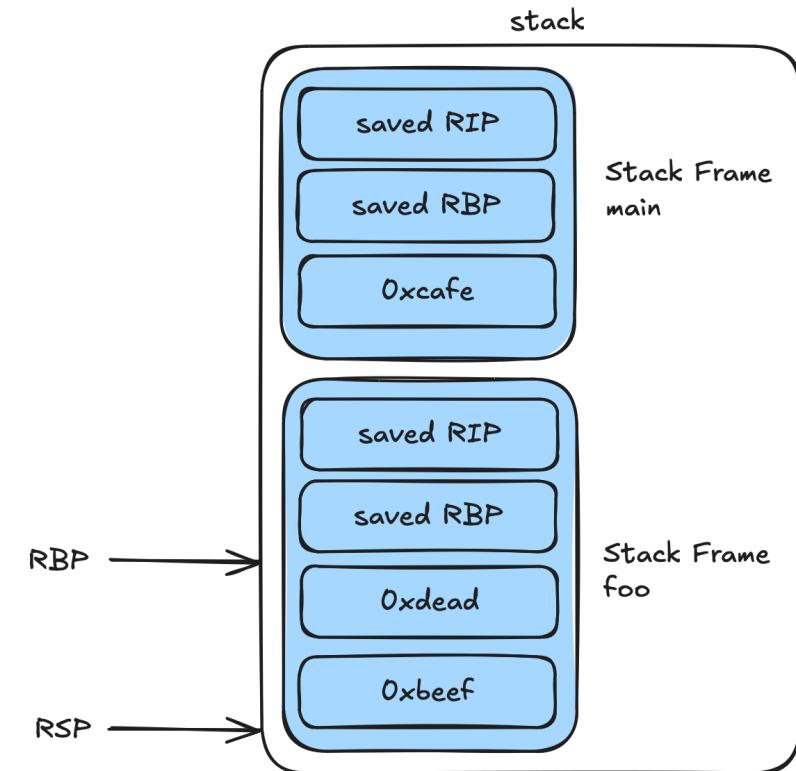
Stockage de variable

```

0x401106 foo:
push rbp
mov rbp, rsp
mov DWORD PTR [rbp-0x4], 0xdead
mov DWORD PTR [rbp-0x8], 0xbeef
nop
pop rbp
ret

0x40111b main:
push rbp
mov rbp, rsp
sub rsp, 0x10
mov DWORD PTR [rbp-0x4], 0xcafe
mov eax, DWORD PTR [rbp-0x4]
mov edi, eax
mov eax, 0x0
call 401106 <foo>
nop
leave
ret

```

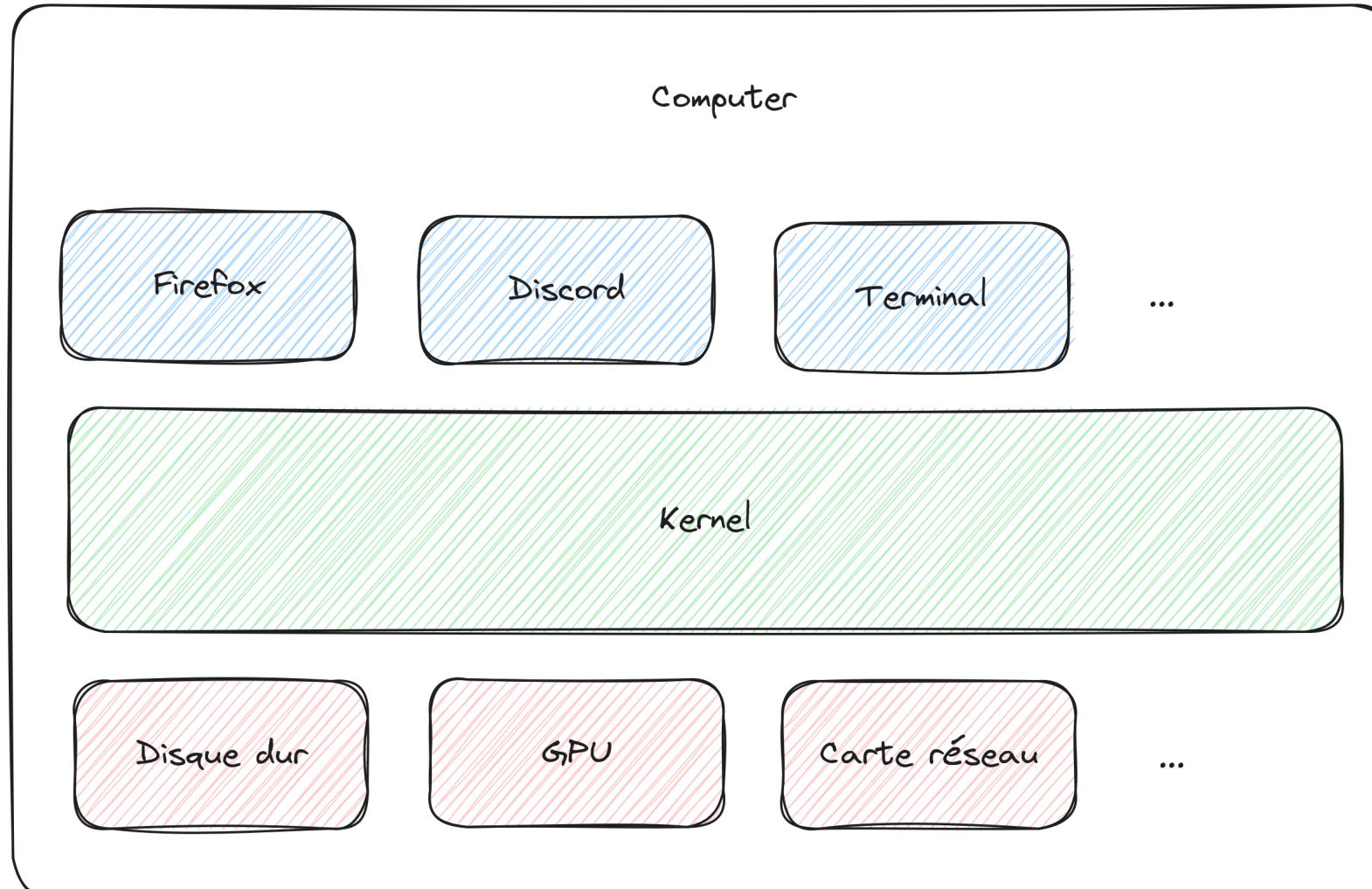


Théorème de la décompilation

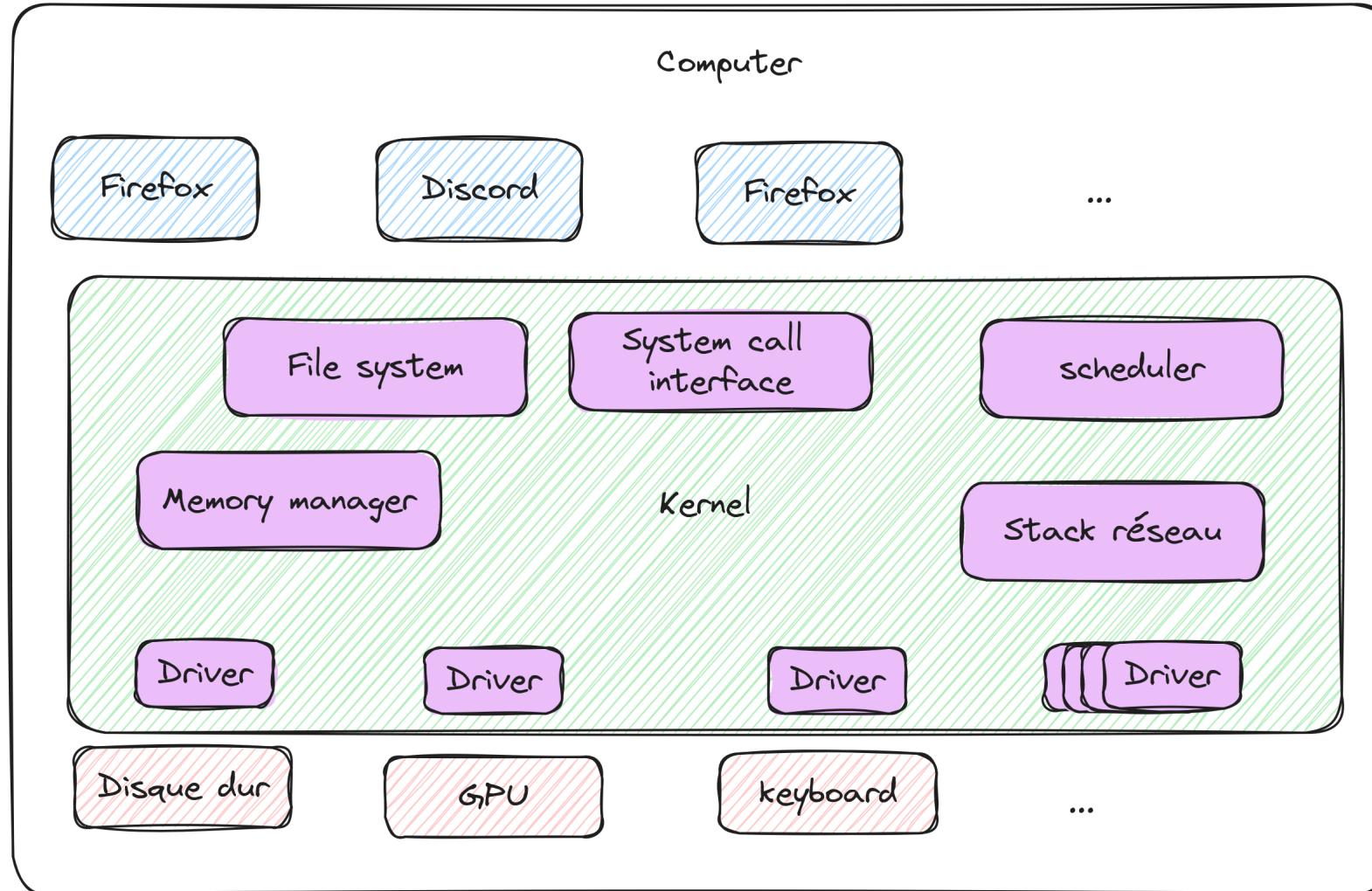
Pour être un bon reverser il faut bien connaître le langage de prog que l'on reverse

Example: structures rust, runtime go, ..

Système d'exploitation

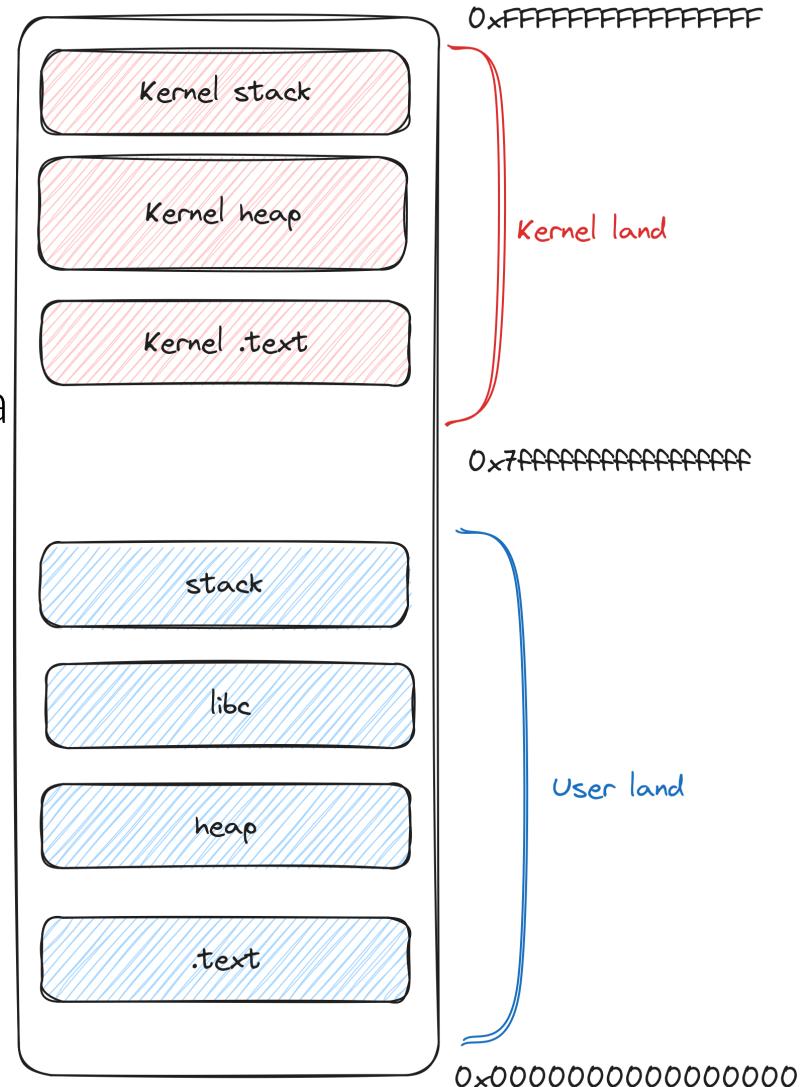


Système d'exploitation



Memory map

- Pour chaque processus, la mémoire ressemble à
- Chaque adresse mémoire stocke 1 octet
- .text: code du programme
- heap: zone de stockage de données



Syscall

```
section .data
    msg db      "hello, world!"

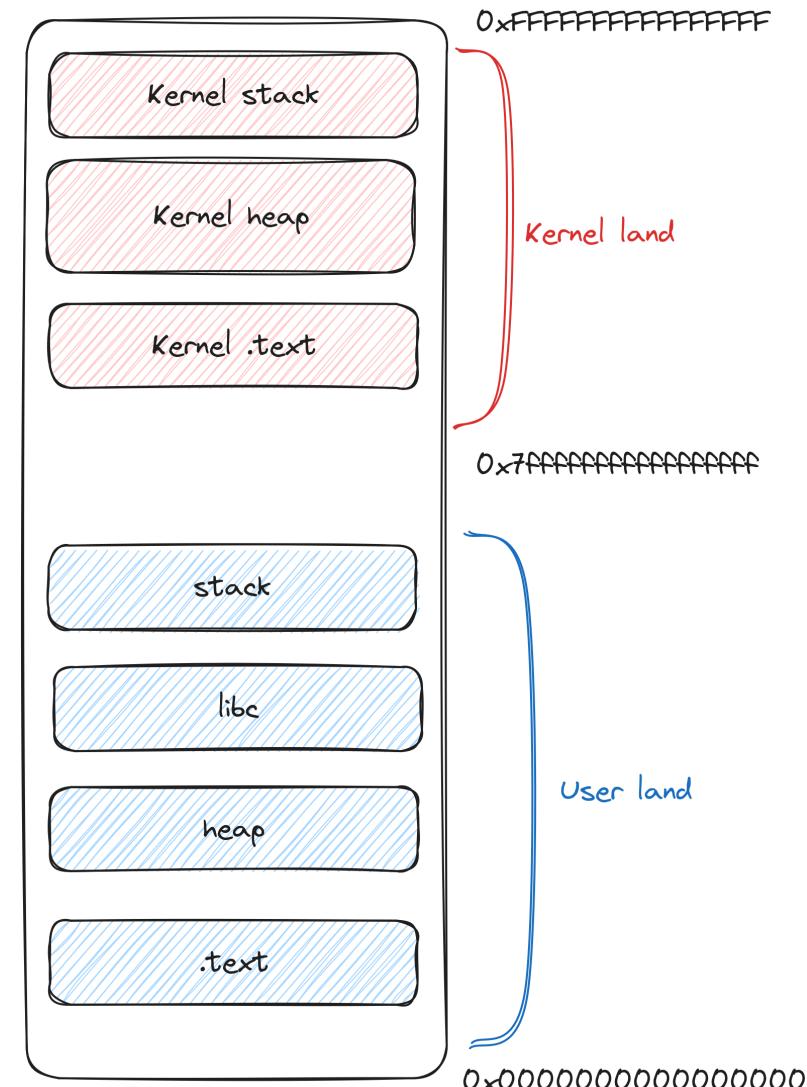
section .text
    global _start

_start:
    mov rdx, 16
    mov rsi, msg
    mov rdi, 1
    mov rax, 1
    syscall
```

Liste sur [ce lien](#)

Librairies partagées

- Libc
 - `printf` / `scanf`
 - `malloc` / `free`
- openssl
 - `RSA` / `AES`
- statique
 - évite les problèmes de version et d'installation
 - gros binaire
- dynamique
 - plus complexe
 - permet des mises à jour



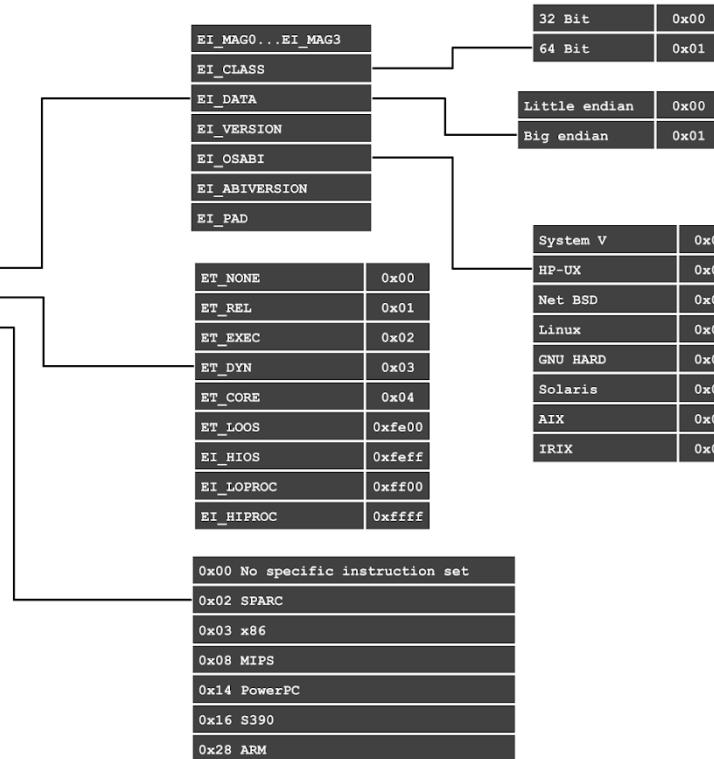
Format ELF

- Adresse virtuelle
- Point d'entrée
- Librairies obligatoires

`readelf` est votre meilleur ami

ELF 64 bit

ELF Identification Field	0x00	e_ident
ELF File Type	0x10	e_type
Machine Type	0x12	e_machine
ELF file format version	0x14	e_version
Program execution entry	0x18	e_entry
Program Header Table start	0x20	e_phoff
Section Header Table start	0x28	e_shoff
Flags	0x30	e_flags
Size of ELF header (0x40)	0x34	e_ehsize
Size of program header table entries	0x36	e_phentsize
Number of program headers	0x38	e_phnum
Size of section header table entries	0x3A	e_shentsize
Number of section header entries	0x3C	e_shnum
Index in section table containing .shstrtab	0x3E	e_shstrndx



Methodologie de reverse

- Analyse statique (ida)
- Analyse dynamique (gdb)

F5

The screenshot shows the IDA Pro interface with the following details:

- Functions View:** Shows a list of functions, many of which are highlighted in pink. Some visible entries include: _init_proc, sub_401020, _free, __errno_location, strcpy, puts, sigaction, mmap, setbuf, system, printf, memset, strcspn, read, fgets, strcmp, sigemptyset, access, perror, strtoul, exit, strdup, start, _dl_relocate_static_pie, deregister_tm_clones, register_tm_clones, and do_global_dtors_aux.
- IDA View-A (Pseudocode-A):** The central window displays the pseudocode for the main function. It starts with a subroutine block:

```
===== S U B R O U T I N E =====
```

Attributes: noreturn bp-based frame

```
int __fastcall main(int argc, const char **argv, const char **env)
```

It then defines a proc near _main with an unwind handler:

```
in _unwind {
```

and contains the following assembly-like pseudocode:

```
push rbp  
mov rbp, rsp  
mov eax, 0  
call init  
mov eax, 0  
call create_zoo  
lea rax, aQuelEstTomNom ; "Quel est tom nom ?"  
mov rdi, rax ; format  
mov eax, 0  
call _printf  
mov edx, 20h ; ' ' ; nbytes  
lea rax, nom ; "default"  
mov rsi, rax ; buf  
mov edi, 0 ; fd  
call _read
```

At address c_401A99, it branches to make_choice:

```
c_401A99: call make_choice ; CODE XREF: main+4A+j
```

The block ends with an endp instruction:

```
} // starts at 401A54  
in endp
```

Finally, it ends with an ends instruction:

```
ext ends
```

Segment type: Pure code

00001AA0 000000000401AA0: _term_| (Synchronized with Hex View)
- Pseudocode-A View:** A right-hand panel showing the pseudocode for the main function:

```
1 int __fastcall __noreturn main(int argc, const char **argv, const char **env)
2 {
3     init(argc, argv, envp);
4     create_zoo();
5     printf("Quel est tom nom ? ");
6     read(0, nom, 0x20uLL);
7     while ( 1 )
8         make_choice();
9 }
```

- Output View:** The bottom-left pane shows the output of the propagation process:

```
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
401933: using guessed type __int64 make_choice(void);
4019A8: using guessed type __int64 create_zoo(void);
401A07: using guessed type __int64 __fastcall init(_QWORD, _QWORD, _QWORD);
```

GDB Cheat sheet

- Désassembler une fonction: `disassemble func`
- Breakpoint: `b *0x1234` `b *main`
- Afficher la mémoire: `x/gx 0x1234` `x/gx $RSP`

GDB Cheat Sheet

Consignes

- Télécharger ida <https://hex-rays.com/ida-free>
- Installer gdb: `sudo apt install gdb`
- Exercices sur: <https://ctf.serviel.fr>
- Groupes de 2 ou 3
- Dernière heure: présentation d'une solution



<https://www.linkedin.com/company/synacktiv>



<https://twitter.com/synacktiv>



<https://synacktiv.com>