

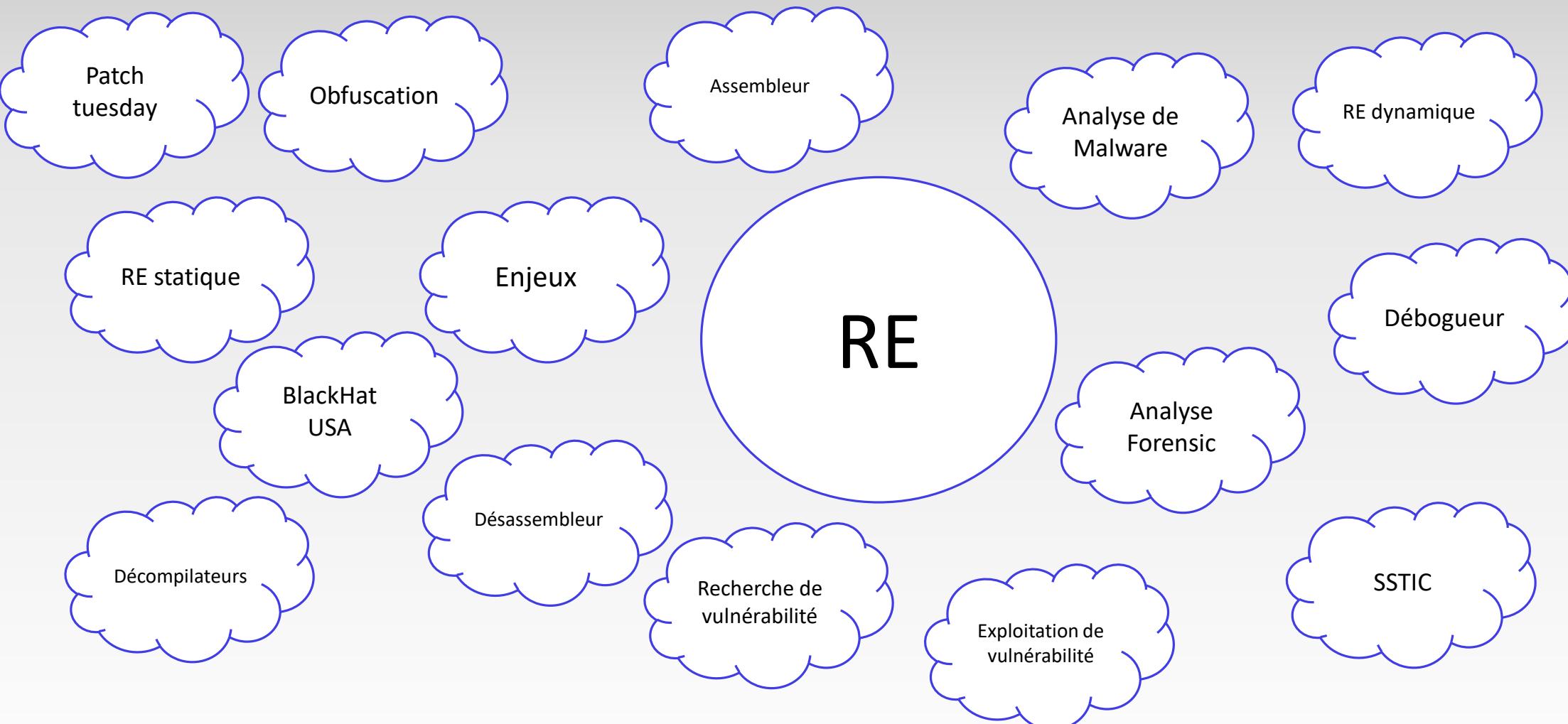
La rétro-ingénierie

COURS D'INTRODUCTION

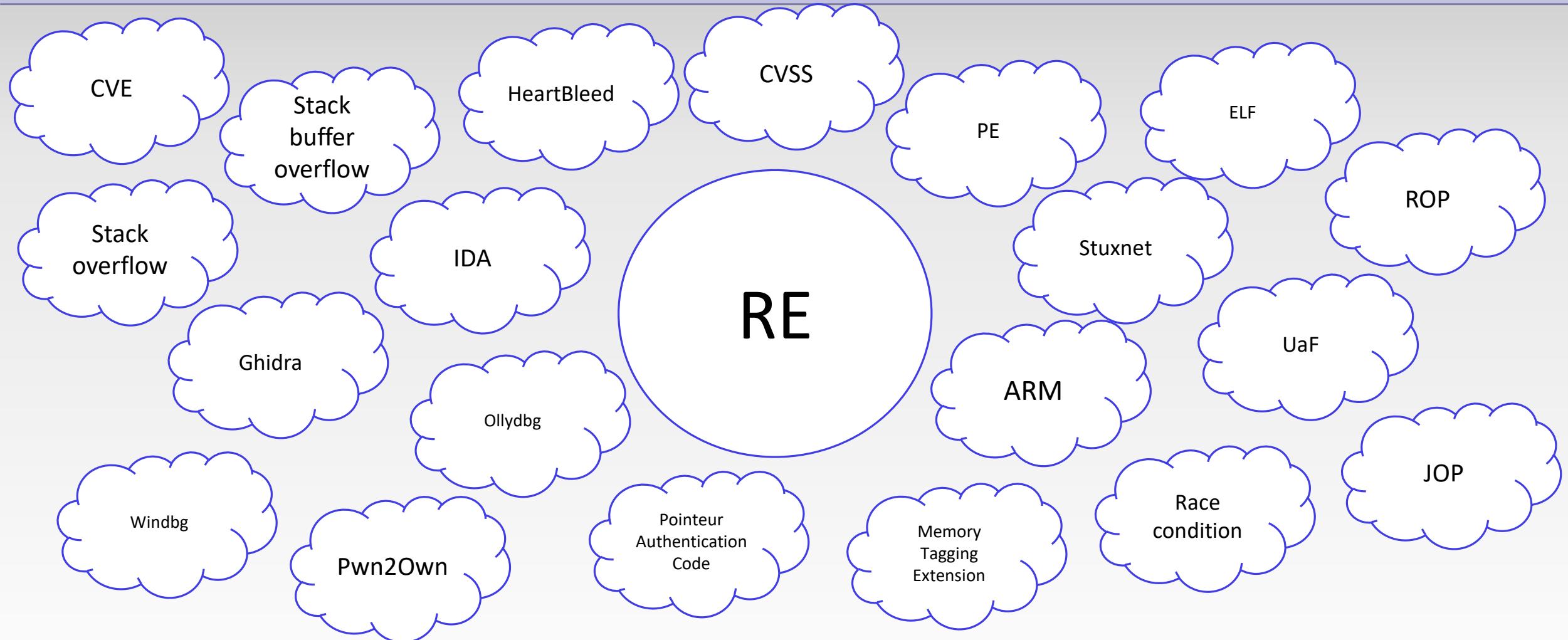


Merci de ne pas enregistrer ni diffuser le contenu du cours à l'extérieur de la classe.

La rétro-ingénierie



La rétro-ingénierie



Les objectifs macro et les applications

- Comprendre les enjeux de la rétro-ingénierie
- Connaître la technique et manipuler les outils
 - Pour la rétro-ingénierie statique
 - Pour la rétro-ingénierie dynamique
- Savoir faire des missions simples de rétro-ingénierie
 - Application sur des exercices en TP
 - Application Exploitation
 - Application Forensics
 - Application Analyse de Malware

Les objectifs macro et les applications

- Comprendre les enjeux de la rétro-ingénierie
- Connaître la technique et manipuler les outils
 - Pour la rétro-ingénierie statique
 - Pour la rétro-ingénierie dynamique
- Savoir faire des missions simples de rétro-ingénierie
 - Application sur des exercices en TP
 - Application Exploitation
 - Application Forensics
 - Application Analyse de Malware

Généralités

DE QUOI PARLE T-ON ?

Première définition

- Selon le Larousse:

Étude d'un produit ou d'un système existant dans le but de déterminer son fonctionnement et la manière dont il a été conçu.

- Beaucoup de cas d'application (même au-delà de l'informatique)
- Les questions sous-jacentes en informatique pour un programme donné :
 - A quoi sert ce programme ?
 - Avec qui interagit-il ?
 - Comment est-il architecturé ?
 - Est-il sécurisé / malveillant ?
- Différence sécurité offensive / défensive ?

La sécurité offensive et défensive

- La sécurité (plutôt) offensive
 - Rechercher des vulnérabilités informatiques pour du pentest
 - Coder des 1-day (diffing party après un correctif)
 - Red team
- La sécurité (plutôt) défensive
 - Recherche et Analyse de bug via une stratégie établie (patch cycle / fuzzing)
 - Analyser des malwares pour comprendre les axes d'attaques
 - Mettre en place des mécanismes de protections
 - Faire des analyses forensics (via un dump mémoire par exemple)
 - Blue Team

Les vulnérabilités zero-days

0-day : La vulnérabilité n'est pas connue de l'éditeur (zéro jour depuis le correctif)

1-day : La vulnérabilité est connue de l'éditeur (Il a émis un correctif depuis au moins 1 jour)

Questions: Quelle est la signification d'une vulnérabilité 0.5 day ?

Les acteurs

- Les chercheurs indépendants
 - Bug bounty
- Les éditeurs
 - Besoin de sécuriser un produit
- Les SSII (prestations de services)
 - Fournissent des services de RE
- Organismes publics
 - Missions variables
- Les CESTI (Centre Evaluation de la Sécurité des Technologies de l'Information)
 - Peuvent effectuer des missions CSPN (premier niveau) pour sécuriser un produit
- Les CERTs (Réponse à Incident)
 - La liste est disponible sur internet. En France: certfr (anssi) / cert-renater

Les applications – analyse forensics / malware

- Votre chef a récupéré un dump mémoire d'un poste infecté
- Vous devez trouver et comprendre le fonctionnement du programme malveillant
 - Quels étaient les programmes en activité
 - Quel est le système de chiffrement employé (si applicable)?
 - Comment le programme survit-il à un reboot ?
 - Quelles sont les traces laissées par l'attaque / le mode opératoire ?
 - Comment se latéralise-t-il dans le système ?
 - Quel est le patient zéro et le premier vecteur d'infection ?
- La rétro-ingénierie permet de donner des éléments de réponse à ces questions

Les applications – Exploitation de vulnérabilités

- Un correctif de sécurité sort sur un composant critique
- Votre mission est de coder un 1-day pour le prochain pentest chez un client
 - Comment fonctionne la vulnérabilité
 - Quels sont les prérequis pour la déclencher
 - Quels sont les droits ou les primitives d'arrivées
 - Quels est le vecteur d'initial
 - Comment établir une stratégie d'exploitation (dépend de la vulnérabilité)
- La rétro-ingénierie permet de donner des éléments de réponse à ces questions

Le Périmètre

QUELLES SONT LES TECHNOLOGIES SOUS-JACENTES

Le périmètre – software / hardware

Il peut s'agir du logiciel (software) ou bien du matériel (hardware). Dans notre cas, on se limitera au logiciel.

N'importe quel programme informatique peut s'observer par des techniques de rétro-ingénierie
MAIS

- Cela peut s'avérer une perte de temps dans le cas d'un binaire protégé par de l'obfuscation
- Cela peut demander des ressources plus ou moins faciles à avoir:
 - Board de dev (interface jtag)
 - Faille de sécurité (mécanisme de codesigning par exemple)
 - Le programme adéquat (si instructions inconnues)
 - Dumper le code d'une flash

Le périmètre

Exemple de programmes :

- Le noyau windows
- Le navigateur Safari iOS
- Le système de gestion de l'eau ou des centrales nucléaires (SCADA)
- Un firmware d'imprimante
- Un service linux qui gère la file d'impression (cups)
- Une Application Mobile

Le périmètre

Exemple de programmes :

- Le noyau windows [format PE / architecture x86-64]
- Le navigateur Safari iOS [format mach-O: architecture arm64e]
- Le système de gestion de l'eau ou des centrales nucléaires (SCADA) [multiple]
- Un firmware d'imprimante [multiple]
- Un service linux qui gère la file d'impression (cups) [ELF / multiple]
- Une Application Mobile [multiple]

Exemple le plus célèbre : hello world

```
j@shell > cat main.c
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 2;
}
j@shell >
```

Exemple le plus célèbre : hello world x86 ELF

```
j@shell > cat main.c
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 2;
}
j@shell >
```

```
00000000001049 <main>:
1149: f3 0f 1e fa
114d: 55
114e: 48 89 e5
1151: 48 8d 3d ac 0e 00 00
1158: e8 f3 fe ff ff
115d: b8 02 00 00 00
1162: 5d
1163: c3
endbr64
push rbp
mov rbp,rsp
lea rdi,[rip+0xeac]
call 1050 <puts@plt>
mov eax,0x2
pop rbp
ret
```

00001140	f3 0f 1e fa f3 e9 77 ff ff ff f3 0f 1e fa ff ee 18 90W.....UH.
00001150	e5 48 8d 3d ac 0e 00 00 e8 f3 fe ff ff b8 02 00	.H.=.....
00001160	00 00 5d c3 66 2e 0f 1f 84 00 00 00 00 00 66 90	..].F.....f.
00001170	f3 0f 1e fa 41 5f 4c 8a 5a 3d 2c 00 00 41 50 49AWL.=;,..AVI
00001180	89 d6 41 55 49 89 f5 41 54 41 89 fc 55 48 8d 2d	..AUI..ATA..UH.-
00001190	2c 2c 00 00 53 4c 29 fd 48 83 ec 08 e8 5f fe ff	,,.SL).H...._..
000011a0	ff 48 c1 fd 03 74 1f 31 db 0f 1f 80 00 00 00 00	.H...t.1.....
000011b0	4c 89 f2 4c 89 ee 44 89 e7 41 ff 14 df 48 83 c3	L..L..D..A..H..

Exemple le plus célèbre : hello world x86-64 PE

```
j@shell > cat main.c
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 2;
}
j@shell >
```

48 83 ec 28 48 8d 0d f5 af 01 00 e8 70 00 00 00 b8 02 00 00 00 48 83 c4 28 c3	sub rsp,0x28 lea rcx,[rip+0x1aff5] call 0x140001080 mov eax,0x2 add rsp,0x28 ret
--	---

```
j@shell> hexdump -C main.exe -s 0x400 -n 40
00000400 48 83 ec 28 48 8d 0d f5 af 01 00 e8 70 00 00 00 |H..(H.....p....|
00000410 b8 02 00 00 00 48 83 c4 28 c3 cc cc cc cc cc cc |.....H..(.....|
00000420 48 8d 05 d9 cb 01 00 c3 H.....|
00000428 j@shell>
```

Exemple le plus célèbre : hello world ARM ELF

```
j@shell > cat main.c
#include <stdio.h>

int main(void)
{
    printf("hello world\n");
    return 2;
}
j@shell >
```

```
0000000000000007c4 <main>:
7c4: fd 7b bf a9      stp    x29, x30, [sp, #-16]!
7c8: fd 03 00 91      mov    x29, sp
7cc: 00 00 00 90      adrp   x0, #0
7d0: 00 e0 1f 91      add    x0, x0, #2040
7d4: 93 ff ff 97      b1    0x620 <$x+0x20>
7d8: 40 00 80 52      mov    w0, #2
7dc: fd 7b c1 a8      ldp    x29, x30, [sp], #16
7e0: c0 03 5f d6      ret
```

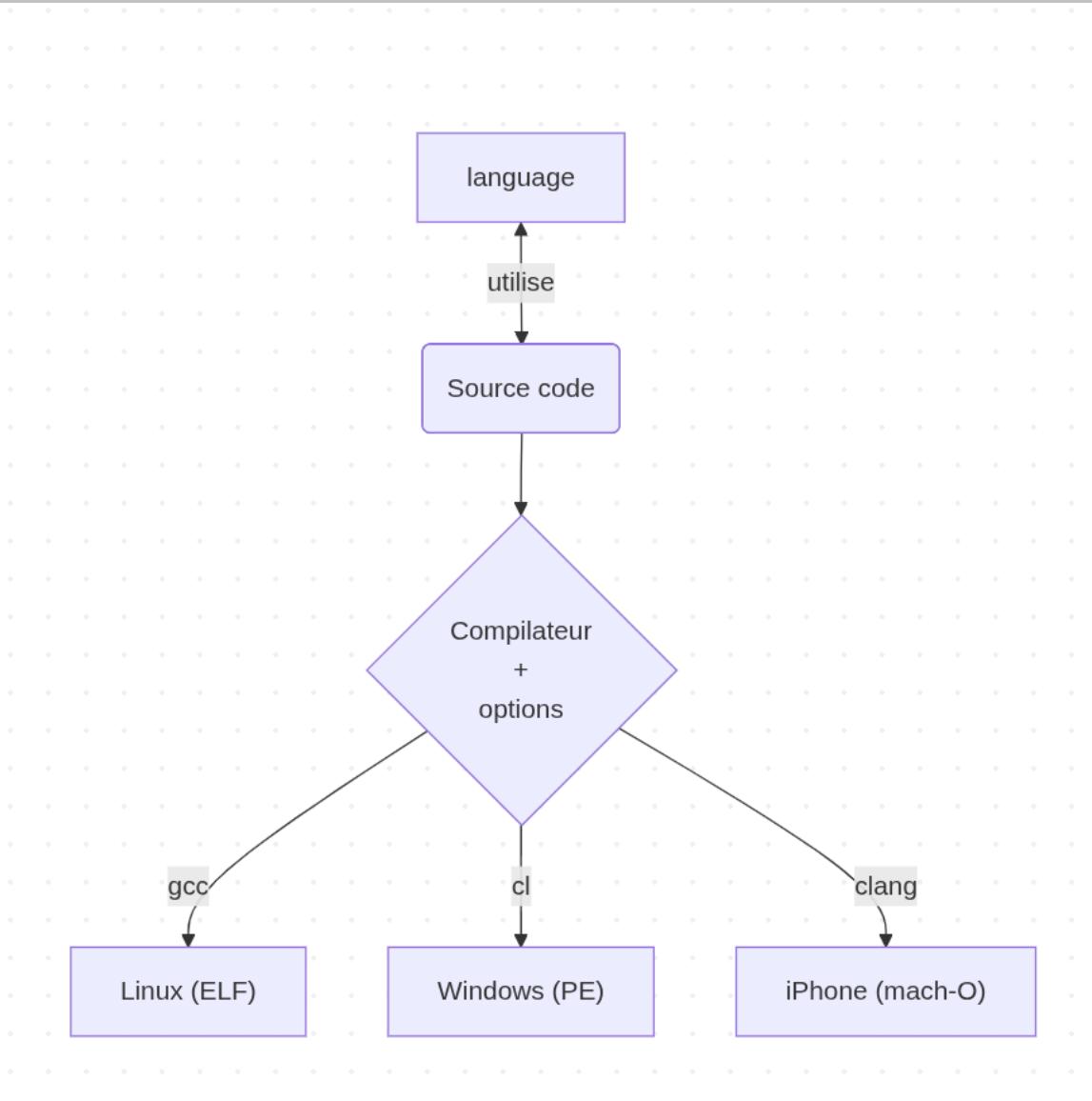
```
j@shell> hexdump -C a.out -s 0x7a0 -n 0x50
000007a0  01 01 00 90 00 00 00 90  fd 03 00 91 21 40 00 91
000007b0  00 e0 20 91 ab ff ff 97  fd 7b c1 a8 cd ff ff 17
000007c0  cc ff ff 17 fd 7b bf a9  fd 03 00 91 00 00 00 90
000007d0  00 e0 1f 91 93 ff ff 97  40 00 80 52 fd 7b c1 a8
000007e0  c0 03 5f d6 fd 7b bf a9  fd 03 00 91 fd 7b c1 a8
```

Exemple le plus célèbre : hello world

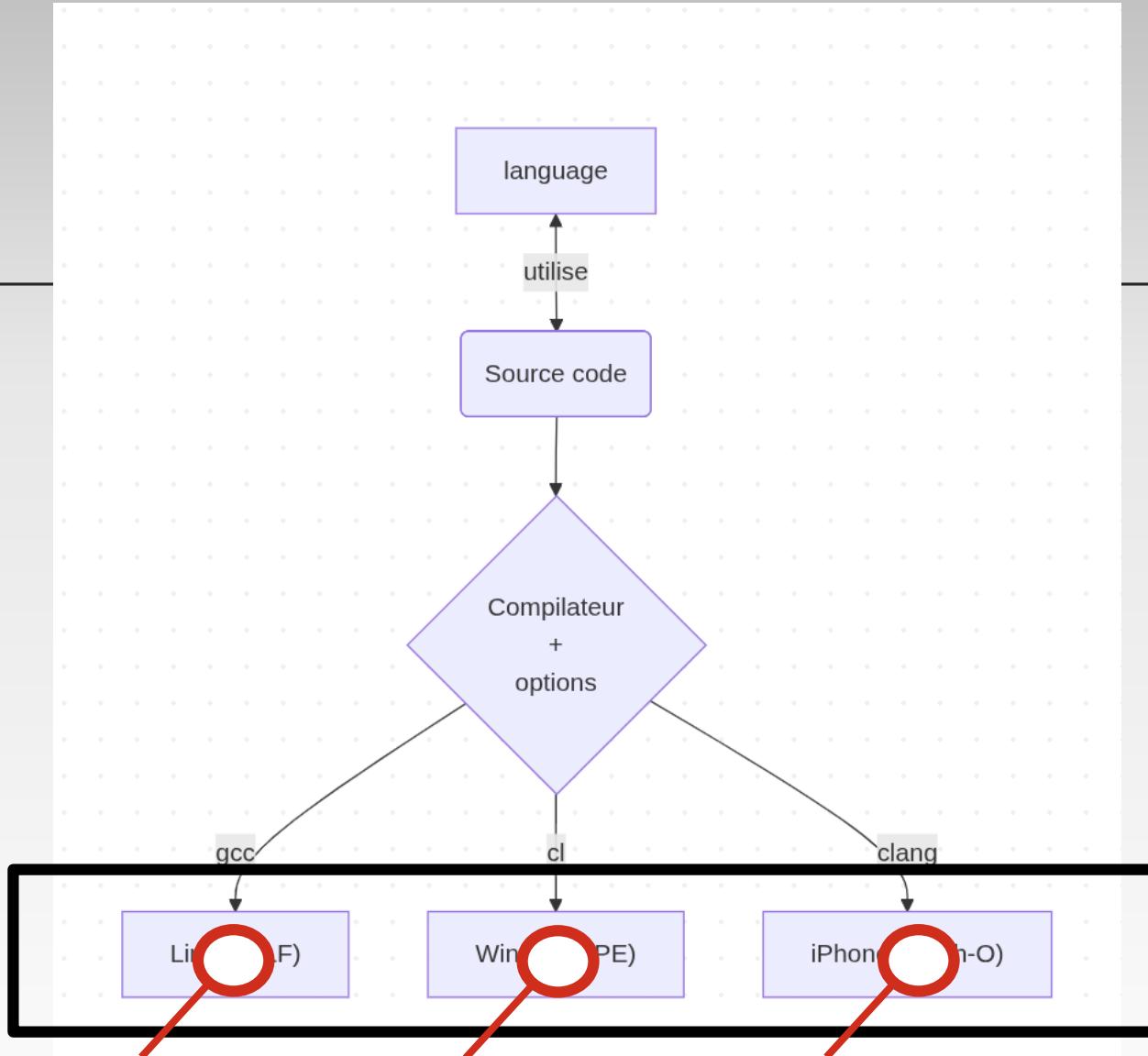
Moralité:

- Le même code source sera très différent en binaire selon:
 - L'architecture
 - Le format
- Mais aussi:
 - Les options de compilation
 - Le langage
- La liste est non exhaustive

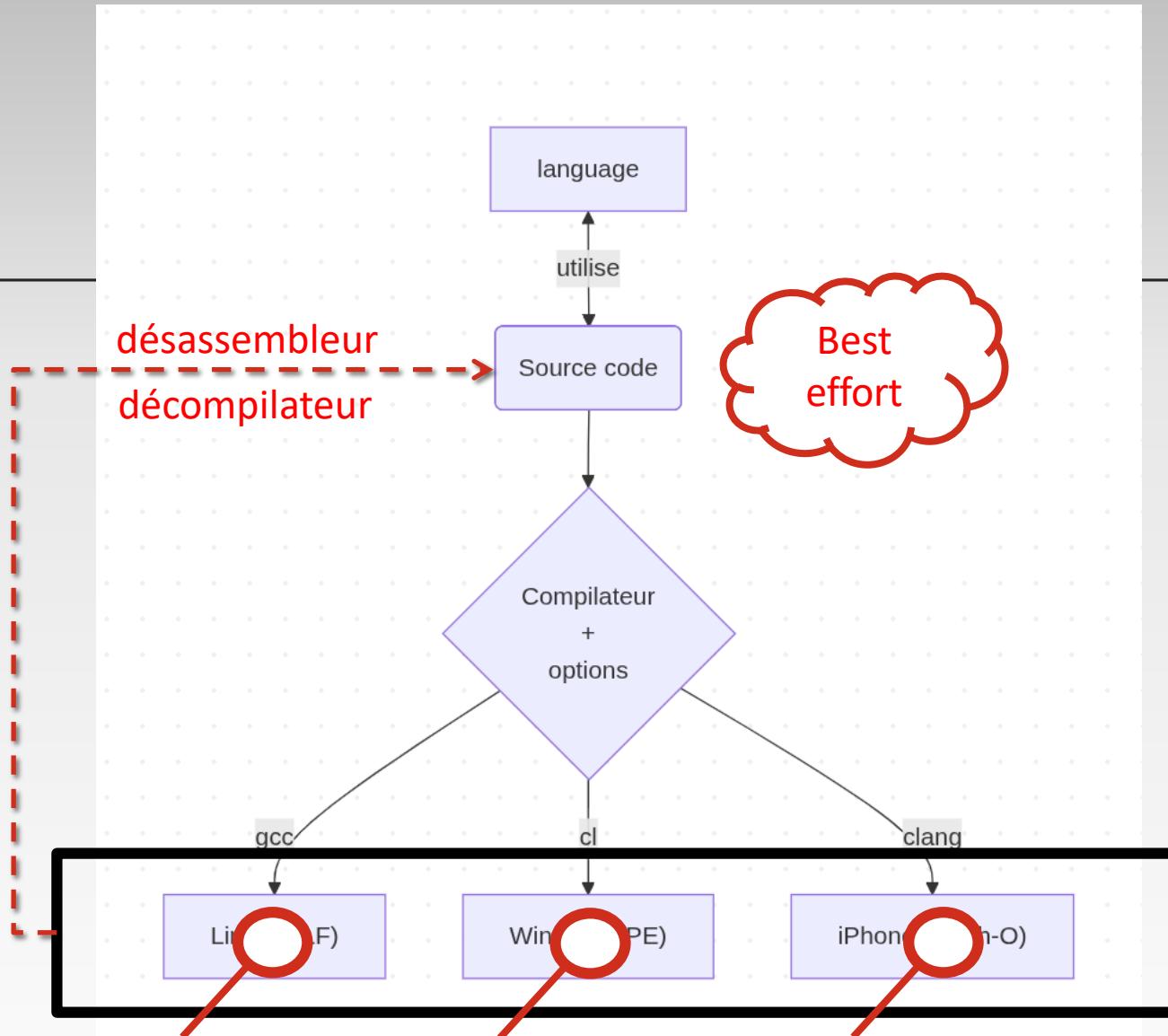
MAP



MAP



MAP



Attention : la syntaxe de l'assembleur

Il existe principalement deux syntaxes pour l'architecture x86:

- la syntaxe intel (utilisée dans ce cours)

```
114d: 55          push rbp
114e: 48 89 e5    mov rbp,rsp
1151: 48 8d 3d ac 0e 00 00 lea rdi,[rip+0xeac]
1158: e8 f3 fe ff ff call 1050 <puts@plt>
```

- la syntaxe AT&T

```
114d: 55          push %rbp
114e: 48 89 e5    mov %rsp,%rbp
1151: 48 8d 3d ac 0e 00 00 lea 0xeac(%rip),%rdi
1158: e8 f3 fe ff ff callq 1050 <puts@plt>
```

Attention : la syntaxe de l'assembleur

Il existe principalement deux syntaxes pour l'architecture x86:

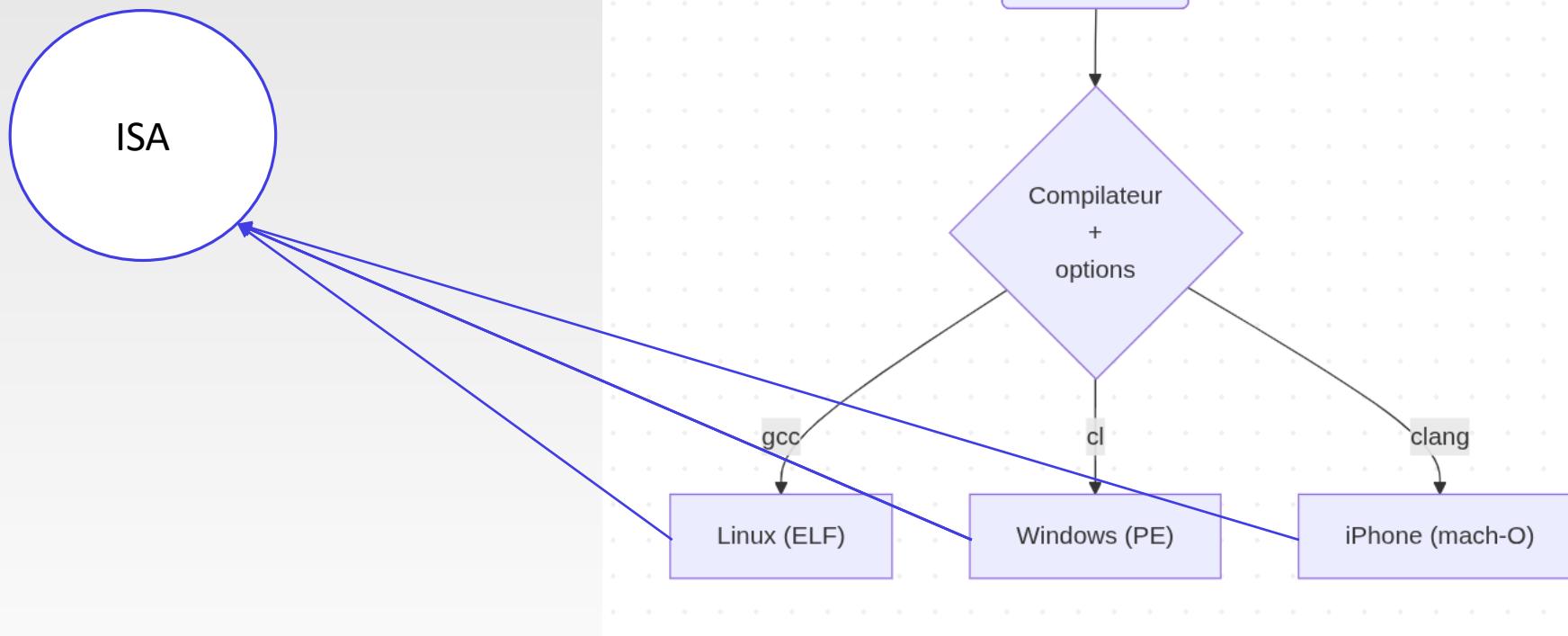
- la syntaxe intel (utilisée dans ce cours)

```
114d: 55          push rbp
114e: 48 89 e5    mov rbp,rsp
1151: 48 8d 3d ac 0e 00 00 lea rdi,[rip+0xeac]
1158: e8 f3 fe ff ff call 1050 <puts@plt>
```

- la syntaxe AT&T

```
114d: 55          push %rbp
114e: 48 89 e5    mov %rsp,%rbp
1151: 48 8d 3d ac 0e 00 00 lea 0xeac(%rip),%rdi
1158: e8 f3 fe ff ff callq 1050 <puts@plt>
```

MAP



Les jeux d'instructions - définition

- In computer science, an **instruction set architecture** (ISA) is an abstract model that generally defines how software controls the CPU in a computer or a family of computers. A device or program that executes instructions described by that ISA, such as a central processing unit (CPU), is called an implementation of that ISA. (Source:Wikipédia)
- Le processeur est fabriqué dans une fonderie et contient des circuits électroniques, sa fonction est de gérer **le jeu d'instruction qu'il implémente**. (lire et exécuter les instructions)
- Celui-ci est composé de:
 - Registres Généraux
 - Registres Spécifiques
 - Registres Flottants
- Flags
- Pipelines d'instructions
- Caches
- Protection hardwares
- Cette liste n'est pas exhaustive

Les jeux d'instructions - types

Deux types principaux de processeurs implémentant chacun un jeu d'instruction se dégagent:

- Les processeurs RISC
- Les processeurs CISC

Les jeux d'instructions – types de processeur

Deux types principaux de processeurs implémentant chacun un jeu d'instruction se dégagent:

- Les processeurs **RISC** : Reduced Instruction Set Computer
- Les processeurs **CISC** :

Les jeux d'instructions – types de processeur

Deux types principaux de processeurs implémentant chacun un jeu d'instruction se dégagent:

- Les processeurs **RISC** : **Reduced** Instruction Set Computer
- Les processeurs **CISC** : **Complexe** Instruction Set Computer

Les jeux d'instructions – ARM

- Utilisé en général dans le domaine de l'embarqué car les processeurs consomment peu d'énergie.
- Actuellement on assiste à un changement de paradigme où ces processeurs arrivent sur le marché des ordinateurs portables.
- La société ARM a fait le design de ces processeurs et des instructions associées. Tout constructeur utilisant un processeur qui utilise le jeu d'instruction ARM doit avoir une licence ARM. (C'est-à-dire la plupart des constructeurs de téléphone portable par exemple)

Les jeux d'instructions – x86

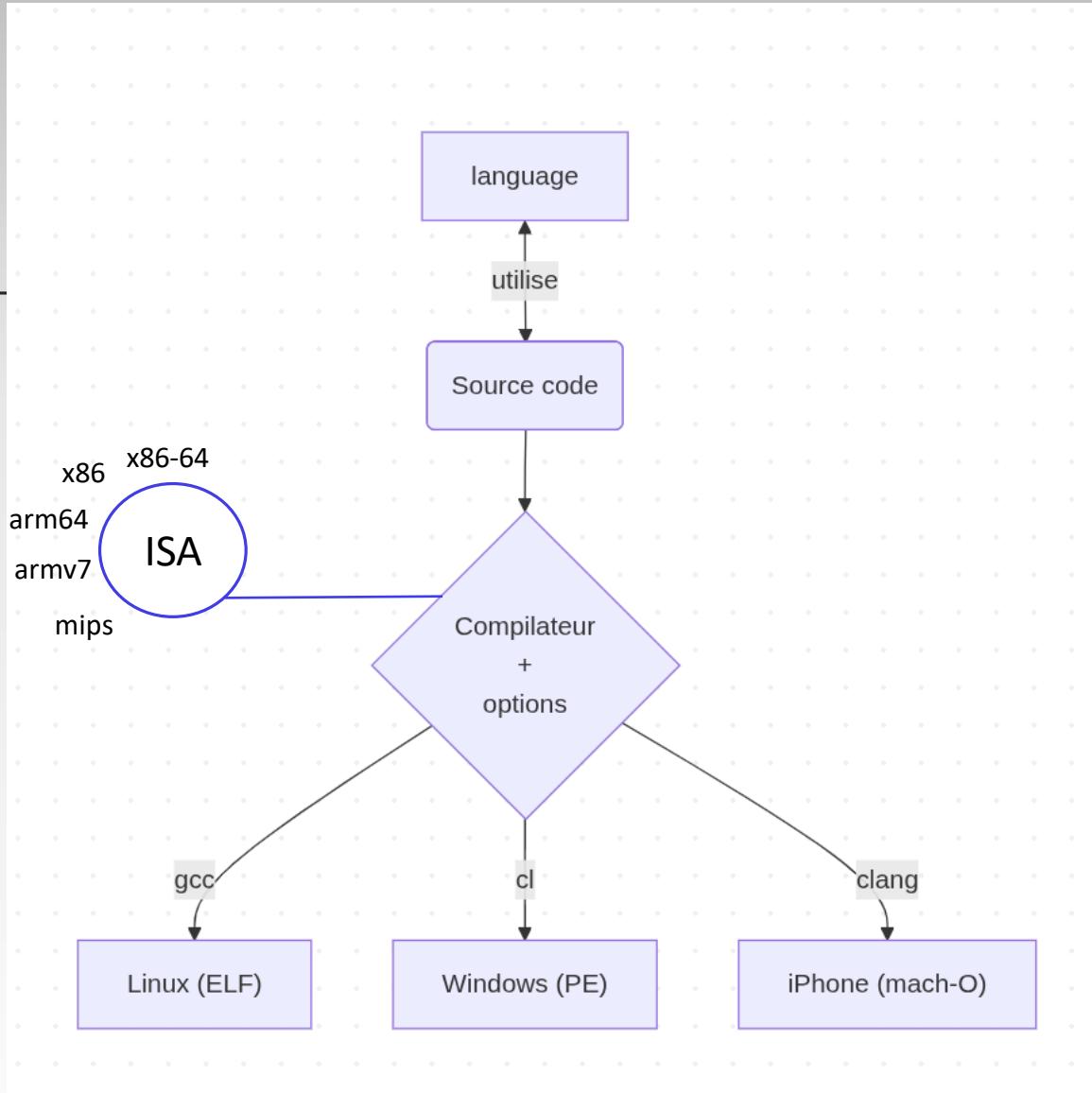
- Les processeurs x86 sont utilisés en général dans les autres cas que l'embarqué
- La société Intel a été à la base de l'architecture x86 en 16 puis 32 bits, ensuite AMD a pris le relai pour les architectures 64bits. (L'histoire est un peu plus complexe – et intéressante mais cela suffit à fixer les idées)

x86-64 – Rappels assembleur pour le reverse

Attention:

Les prochaines slides seront un peu indigestes

MAP



x86 assembly

QUE DIT LE MANUEL ?

x86-64 – Rappels assembleur pour le reverse

- Création : AMD (et non ce n'est pas intel)
 - Année : ~2000
 - Nombre d'instructions: 1000+
 - Il existe des extensions d'instruction (par exemple des instructions de crypto ou bien de virtualisation)
 - Les instructions ont une taille variable
- Exemple d'instruction simple :
 - XOR RAX, RAX (Que fait cette instruction?)
 - PUSH RBP
 - RETN 8
 - Exemple d'instruction compliquée :
 - GF2P8AFFINEINVQB (?)

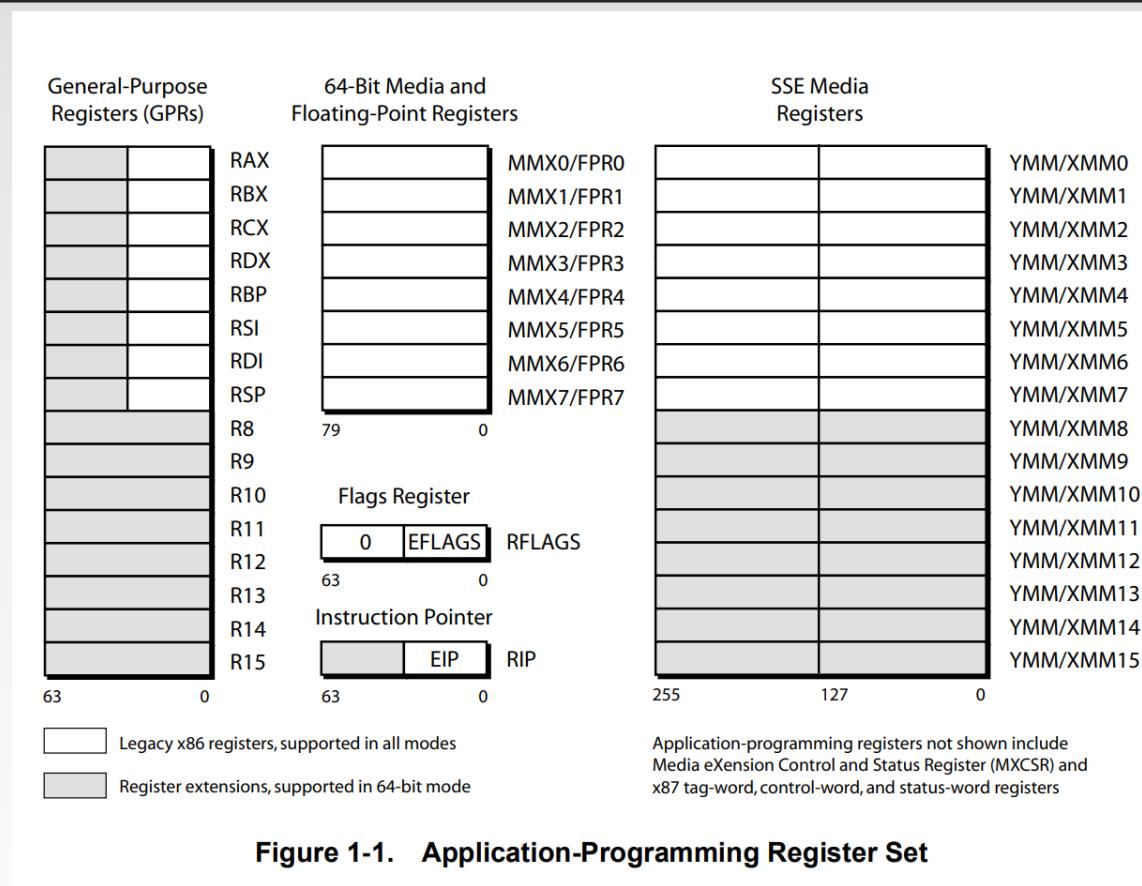
x86-64 – Rappels assembleur pour le reverse

En règle générale, on peut bien comprendre l'assembleur x86 avec environ 50 instructions.

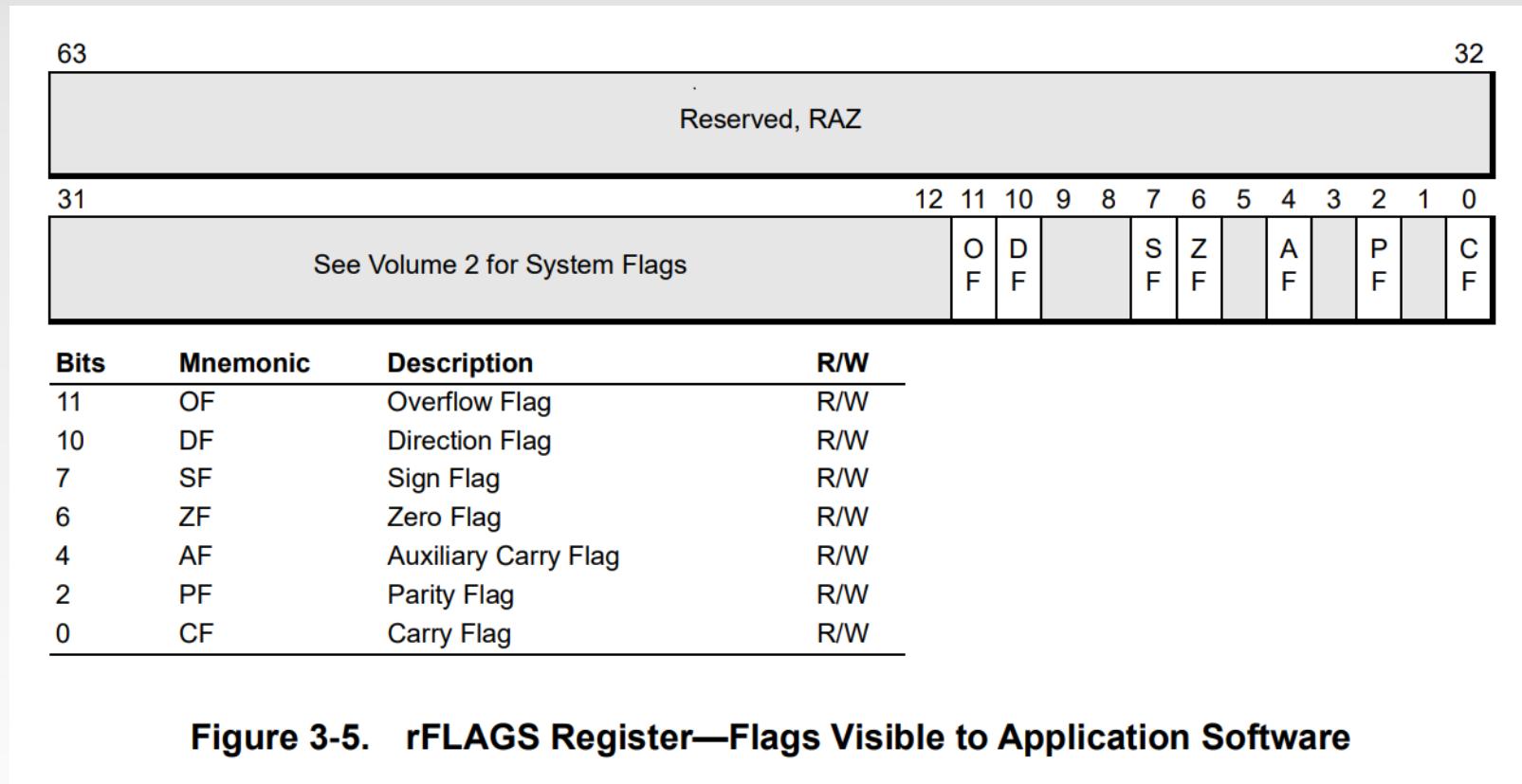
x86-64 – Rappels assembleur pour le reverse

- Les registres
- Source: “AMD64 Architecture Programmer’s Manual Volume 1: Application Programming

x86-64 – Rappels assembleur pour le reverse



x86-64 – Rappels assembleur pour le reverse



x86-64 – Prerequis assembleur pour le reverse

- Les instructions usuelles en x86_64: CALL/JMP/MOV/ADD/XOR/RET/PUSH/POP
- La taille des registres (ex: RAX:64bits/EAX:32bits/AX:16bits/AL:8bits)
- Tous les sauts conditionnels (voir slide suivante)
- Dans ce cours, on ne traitera pas les flottants

x86-64 – Rappels assembleur pour le reverse

- Les sauts conditionnels

Table 3-6. rFLAGS for Jcc Instructions

Mnemonic	Required Flag State	Description
JO	OF = 1	Jump near if overflow
JNO	OF = 0	Jump near if not overflow
JB JC JNAE	CF = 1	Jump near if below Jump near if carry Jump near if not above or equal
JNB JNC JAE	CF = 0	Jump near if not below Jump near if not carry Jump near if above or equal
JZ JE	ZF = 1	Jump near if 0 Jump near if equal
JNZ JNE	ZF = 0	Jump near if not zero Jump near if not equal
JNA JBE	CF = 1 or ZF = 1	Jump near if not above Jump near if below or equal
JNBE JA	CF = 0 and ZF = 0	Jump near if not below or equal Jump near if above
JS	SF = 1	Jump near if sign
JNS	SF = 0	Jump near if not sign
JP JPE	PF = 1	Jump near if parity Jump near if parity even
JNP JPO	PF = 0	Jump near if not parity Jump near if parity odd
JL JNGE	SF <> OF	Jump near if less Jump near if not greater or equal

x86-64 – Rappels assembleur pour le reverse

- Les sauts conditionnels

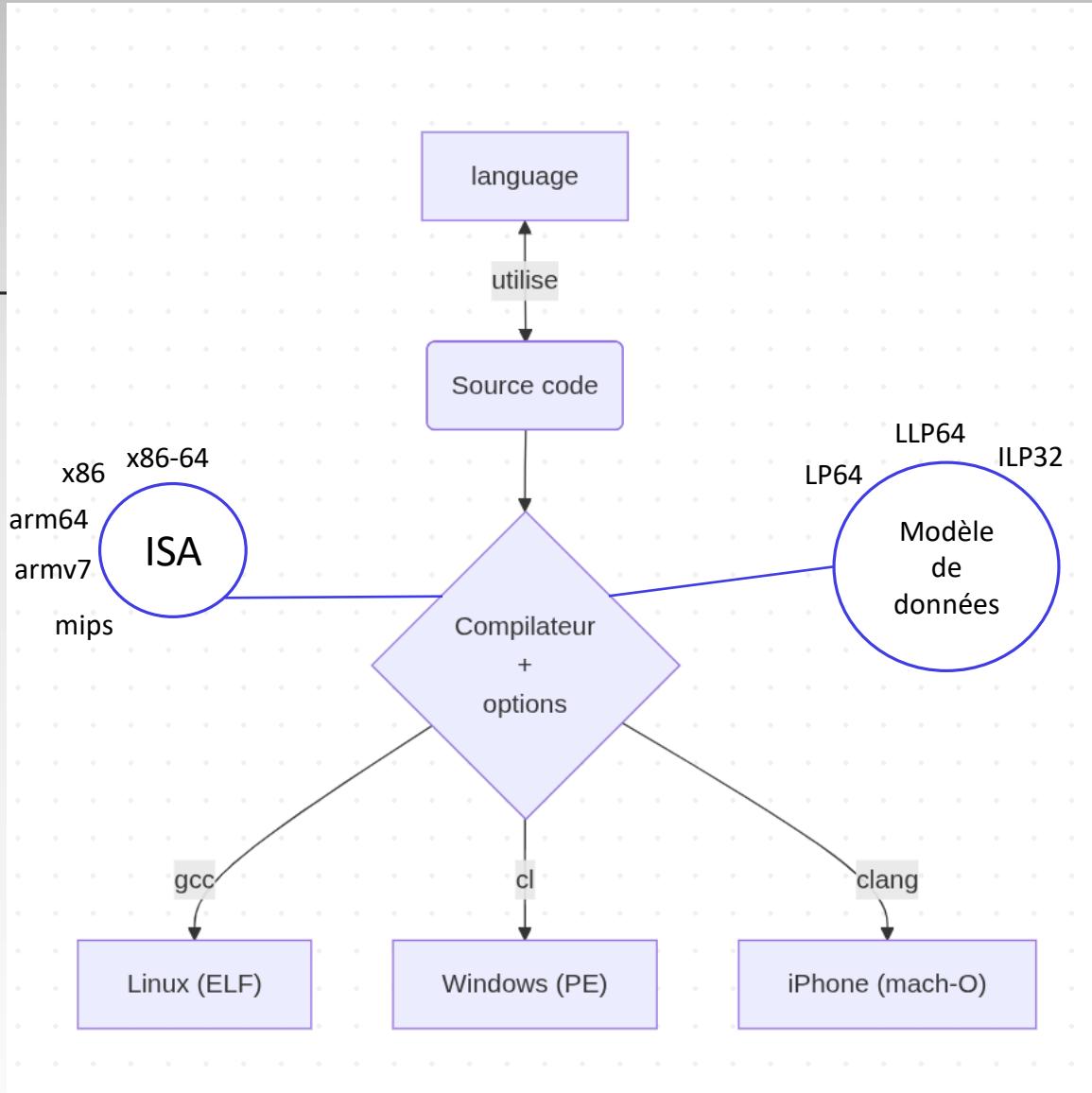
Table 3-6. rFLAGS for Jcc Instructions (continued)

Mnemonic	Required Flag State	Description
JGE JNL	SF = OF	Jump near if greater or equal Jump near if not less
JNG JLE	ZF = 1 or SF <> OF	Jump near if not greater Jump near if less or equal
JNLE JG	ZF = 0 and SF = OF	Jump near if not less or equal Jump near if greater

x86-64 – Assembleur pour le reverse (Signed/Unsigned)

- Exemple
- **unsigned** int i = 4 ; if (i <3) printf("ok")
 - Le compilateur génère un **JA**
- **int** i = 4 ; if (i < 3) printf("ok");
 - Le compilateur génère un **JG**
- En regardant l'assembleur, on peut *parfois* savoir quel type a été utilisé par le programmeur.
- Question: Que se passe-t-il pour: **unsigned int i = 4; if (i<0); printf("ok")** ?

MAP



Les types de données

101 ASSEMBLEUR POUR LE REVERSE

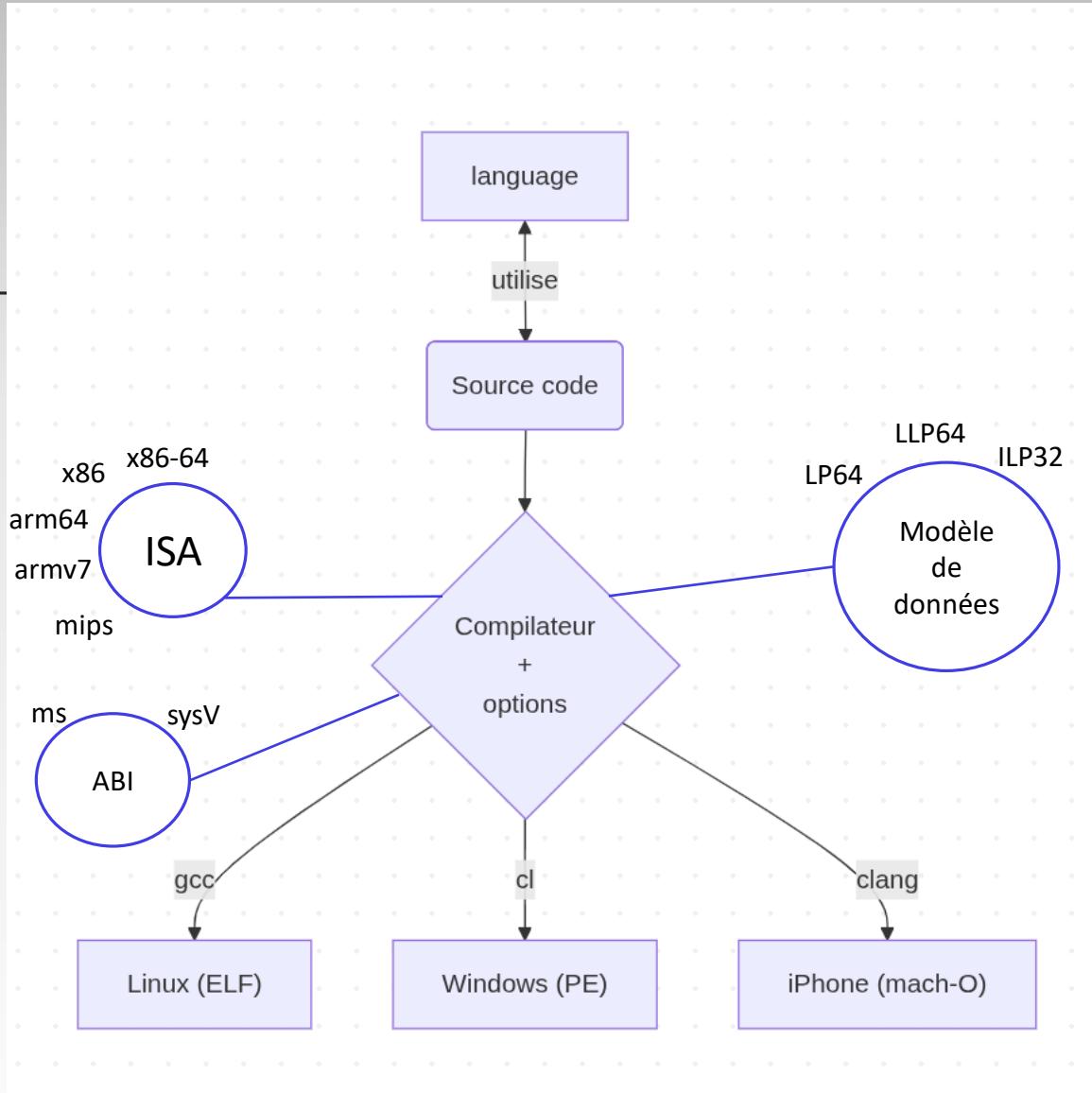
x86-64 – Rappels assembleur pour le reverse

- Les data types
- Le processeur reconnaît des types de données:
 - Signed Integer (sur 8 / 16 / 32 / 64bits)
 - Unsigned Integer (sur 8 / 16 / 32 / 64 bits)
 - ~~Doubles~~ : Non traités dans ce cours
- Représentation des signed via le complément à 2: (Pourquoi ?)
- Pour trouver la représentation négative, on flip les bits et on ajoute 1.
- Ex: sur 16bit, -17: (-) 0000 0000 0001 0001 -> 1111 1111 1110 1110 (+1) -> FFEF

x86-64 – Rappels assembleur pour le reverse

- Les types de données sont également gérés plus finement par l'environnement de compilation et les usages: on parle des short / int / long / long long
- Exemple: Un long vaut 8 octets sous linux
- Question : Que vaut-il sous windows par défaut?

MAP



ABI

101 ASSEMBLEUR POUR LE REVERSE

x86-64 – Les appels de fonctions sous linux

Si une fonction A appelle une fonction B, alors on dit que :

- A est le “caller”
- B est la “callee”
- La façon dont les arguments sont passés du caller au callee est définie par **l'Application Binary Interface** ou (ABI).
- L'ABI diffère selon les systèmes (via les compilateurs associés) : Il s'agit d'un ensemble de conventions qui permet d'interfacer des unités d'exécution binaires entre elles (par exemple une fonction et une sous-fonction). Ces conventions sont implémentées dans le compilateur.

x86-64 – Les appels de fonctions – le cas sysV (linux)

Les paramètres sont passés par les registres suivants :

- \$rdi
- \$rsi
- \$rdx
- \$rcx
- \$r8
- \$r9
- le reste sur la stack

x86-64 – Les appels de fonctions – le cas sysV (linux)

```
4 int main( void )
5 {
6     printf("[+] I'm the caller. I will call the function callee: \n");
7     int res = callee( 0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888 );
8     printf("[+] I just returned from callee: %d\n", res);
9     return res;
10 }
```

Appel de fonction: cas linux x86-64

```
Dump of assembler code for function main:
0x000000000000011f3 <+0>:    endbr64
0x000000000000011f7 <+4>:    push   rbp
0x000000000000011f8 <+5>:    mov    rbp,rs
0x000000000000011fb <+8>:    sub    rs,0x10
0x000000000000011ff <+12>:   lea    rdi,[rip+0xe0a]
0x00000000000001206 <+19>:   call   0x1060 <puts@plt>
0x0000000000000120b <+24>:   push   0x8888
0x00000000000001210 <+29>:   push   0x7777
0x00000000000001215 <+34>:   mov    r9d,0x6666
0x0000000000000121b <+40>:   mov    r8d,0x5555
0x00000000000001221 <+46>:   mov    ecx,0x4444
0x00000000000001226 <+51>:   mov    edx,0x3333
0x0000000000000122b <+56>:   mov    esi,0x2222
0x00000000000001230 <+61>:   mov    edi,0x1111
0x00000000000001235 <+66>:   call   0x1169 <callee>
0x0000000000000123a <+71>:   add    rsp,0x10
```

x86-64 – Les appels de fonctions – le cas sysV (linux)

```
4 int main( void )
5 {
6     printf("[+] I'm the caller. I will call the function callee: \n");
7     int res = callee( 0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888 );
8     printf("[+] I just returned from callee: %d\n", res);
9     return res;
10 }
```

Appel de fonction: cas linux x86-64

```
Dump of assembler code for function main:
0x000000000000011f3 <+0>:    endbr64
0x000000000000011f7 <+4>:    push   rbp
0x000000000000011f8 <+5>:    mov    rbp,rs
0x000000000000011fb <+8>:    sub    rs,0x10
0x000000000000011ff <+12>:   lea    rdi,[rip+0xe0a]
0x00000000000001206 <+19>:   call   0x1060 <puts@plt>
0x0000000000000120b <+24>:   push   0x8888
0x00000000000001210 <+29>:   push   0x7777
0x00000000000001215 <+34>:   mov    r9d,0x6666
0x0000000000000121b <+40>:   mov    r8d,0x5555
0x00000000000001221 <+46>:   mov    ecx,0x4444
0x00000000000001226 <+51>:   mov    edx,0x3333
0x0000000000000122b <+56>:   mov    esi,0x2222
0x00000000000001230 <+61>:   mov    edi,0x1111
0x00000000000001235 <+66>:   call   0x1169 <callee>
0x0000000000000123a <+71>:   add    rsp,0x10
```

x86-64 – Les appels de fonctions – le cas sysV (linux)

```
4 int main( void )
5 {
6     printf("[+] I'm the caller. I will call the function callee: \n");
7     int res = callee( 0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888 );
8     printf("[+] I just returned from callee: %d\n", res);
9     return res;
10 }
```

Appel de fonction: cas linux x86-64

Fonction: « callee »

Dump of assembler code for function main:

```
0x00000000000001f3 <+0>:    endbr64
0x00000000000001f7 <+4>:    push   rbp
0x00000000000001f8 <+5>:    mov    rbp,rs
0x00000000000001fb <+8>:    sub    rs,0x10
0x00000000000001ff <+12>:   lea    rdi,[rip+0xe0a]
0x00000000000001206 <+19>:   call   0x1060 <puts@plt>
0x0000000000000120b <+24>:   push   0x8888
0x00000000000001210 <+29>:   push   0x7777
0x00000000000001215 <+34>:   mov    r9d,0x6666
0x0000000000000121b <+40>:   mov    r8d,0x5555
0x00000000000001221 <+46>:   mov    ecx,0x4444
0x00000000000001226 <+51>:   mov    edx,0x3333
0x0000000000000122b <+56>:   mov    esi,0x2222
0x00000000000001230 <+61>:   mov    edi,0x1111
0x00000000000001235 <+66>:   call   0x1169 <callee>
```

x86-64 – Les appels de fonctions – le cas sysV (linux)

```
4 int main( void )
5 {
6     printf("[+] I'm the caller. I will call the function callee: \n");
7     int res = callee( 0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888 );
8     printf("[+] I just returned from callee: %d\n", res);
9     return res;
10 }
```

Appel de fonction: cas linux x86-64

Paramètre 1 : dans \$rdi

Dump of assembler code for function main:

```
0x00000000000001f3 <+0>:    endbr64
0x00000000000001f7 <+4>:    push   rbp
0x00000000000001f8 <+5>:    mov    rbp,rs
0x00000000000001fb <+8>:    sub    rs,0x10
0x00000000000001ff <+12>:   lea    rdi,[rip+0xe0a]
0x00000000000001206 <+19>:   call   0x1060 <puts@plt>
0x0000000000000120b <+24>:   push   0x8888
0x00000000000001210 <+29>:   push   0x7777
0x00000000000001215 <+34>:   mov    r9d,0x6666
0x0000000000000121b <+40>:   mov    r8d,0x5555
0x00000000000001221 <+46>:   mov    ecx,0x4444
0x00000000000001226 <+51>:   mov    edx,0x3333
0x0000000000000122b <+56>:   mov    esi,0x2222
0x00000000000001230 <+61>:   mov    edi,0x1111
0x00000000000001235 <+66>:   call   0x1160 <callee>
0x0000000000000123a <+71>:   add    rsp,0x10
```

x86-64 – Les appels de fonctions – le cas sysV (linux)

```
4 int main( void )
5 {
6     printf("[+] I'm the caller. I will call the function callee: \n");
7     int res = callee( 0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888 );
8     printf("[+] I just returned from callee: %d\n", res);
9     return res;
10 }
```

Appel de fonction: cas linux x86-64

Paramètre 7 sur la stack [\$rsp]

Dump of assembler code for function main:

```
0x00000000000001f3 <+0>:    endbr64
0x00000000000001f7 <+4>:    push   rbp
0x00000000000001f8 <+5>:    mov    rbp,rs
0x00000000000001fb <+8>:    sub    rs,0x10
0x00000000000001ff <+12>:   lea    rdi,[rip+0xe0a]
0x00000000000001206 <+19>:   call   0x1060 <puts@plt>
0x0000000000000120b <+24>:   push   0x0000
0x00000000000001210 <+29>:   push   0x7777
0x00000000000001215 <+34>:   mov    r9d,0x6666
0x0000000000000121b <+40>:   mov    r8d,0x5555
0x00000000000001221 <+46>:   mov    ecx,0x4444
0x00000000000001226 <+51>:   mov    edx,0x3333
0x0000000000000122b <+56>:   mov    esi,0x2222
0x00000000000001230 <+61>:   mov    edi,0x1111
0x00000000000001235 <+66>:   call   0x1169 <callee>
0x0000000000000123a <+71>:   add    rsp,0x10
```

x86-64 – Les appels de fonctions – le cas ms (windows)

Les paramètres sont passés par les registres suivants:

- \$rcx
- \$rdx
- \$r8
- \$r9
- le reste sur la stack

x86-64 – Les appels de fonctions – le cas ms (windows)

```
int main(void)
{
    int j = 0x1337;
    callee(0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888);
    return 2;
}
```

Appel de fonction: cas « ms »

```
sub    rsp, 58h
mov    [rsp+58h+var_18], 1337h
mov    [rsp+58h+var_20], 8888h
mov    [rsp+58h+var_28], 7777h
mov    [rsp+58h+var_30], 6666h
mov    [rsp+58h+var_38], 5555h
mov    r9d, 4444h
mov    r8d, 3333h
mov    edx, 2222h
mov    ecx, 1111h
call   sub_140001000
mov    eax, 2
add    rsp, 58h
retn
main endp
```

x86-64 – Les appels de fonctions – le cas ms (windows)

```
int main(void)
{
    int j = 0x1337;
    callee(0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888);
    return 2;
}
```

Appel de fonction: cas « ms »
Fonction « callee »

```
sub    rsp, 58h
mov    [rsp+58h+var_18], 1337h
mov    [rsp+58h+var_20], 8888h
mov    [rsp+58h+var_28], 7777h
mov    [rsp+58h+var_30], 6666h
mov    [rsp+58h+var_38], 5555h
mov    r9d, 4444h
mov    r8d, 3333h
mov    edx, 2222h
mov    ecx, 1111h
call   sub_14000100
mov    eax, 2
add    rsp, 58h
retn
main endp
```

x86-64 – Les appels de fonctions – le cas ms (windows)

```
int main(void)
{
    int i = 0x1337;
    callee(0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888);
    return 2;
}
```

Appel de fonction: cas « ms »
Paramètre 1 dans \$rcx

```
sub    rsp, 58h
mov    [rsp+58h+var_18], 1337h
mov    [rsp+58h+var_20], 8888h
mov    [rsp+58h+var_28], 7777h
mov    [rsp+58h+var_30], 6666h
mov    [rsp+58h+var_38], 5555h
mov    r9d, 4444h
mov    r8d, 3333h
mov    edx, 2222h
mov    ecx, 1111h
call   sub_140001000
mov    eax, 2
add    rsp, 58h
retn
main endp
```

x86-64 – Les appels de fonctions – le cas ms (windows)

```
int main(void)
{
    int j = 0x1337;
    callee(0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888);
    return 2;
}
```

Appel de fonction: cas « ms »
Paramètre 2 dans \$rdx

```
sub    rsp, 58h
mov    [rsp+58h+var_18], 1337h
mov    [rsp+58h+var_20], 8888h
mov    [rsp+58h+var_28], 7777h
mov    [rsp+58h+var_30], 6666h
mov    [rsp+58h+var_38], 5555h
mov    r9d, 4444h
mov    r8d, 3333h
mov    edx, 2222h
mov    ecx, 1111h
call   sub_14000100
mov    eax, 2
add    rsp, 58h
ret
main endp
```

x86-64 – Les appels de fonctions – le cas ms (windows)

```
int main(void)
{
    int j = 0x1337;
    callee(0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888);
    return 2;
}
```

Appel de fonction: cas « ms »
Paramètre 5 sur la stack

```
sub    rsp, 58h
mov    [rsp+58h+var_18], 1337h
mov    [rsp+58h+var_20], 8888h
mov    [rsp+58h+var_28], 7777h
mov    [rsp+58h+var_30], 6666h
mov    [rsp+58h+var_38], 5555h
mov    r9d, 4444h
mov    r8d, 3333h
mov    edx, 2222h
mov    ecx, 1111h
call   sub_140001000
mov    eax, 2
add    rsp, 58h
retn
main endp
```

arm64

ASSEMBLEUR POUR LE REVERSE

arm64 – Rappels assembleur pour le reverse

Attention:

Les prochaines slides seront un peu indigestes

arm64 – Rappels assembleur pour le reverse

- Création : ARM
 - Nombre d'instructions: ~200
 - Les instructions ont une taille de 4 bytes
 - Exemple d'instruction compliquée
 - (?)
-
- Exemple d'instruction simple
 - PUSH X0

arm64 – Rappels assembleur pour le reverse

Les registres

B1.2 Registers in AArch64 Execution state

The following registers are visible at EL0 using AArch64:

R0-R30 31 general-purpose registers, R0 to R30. Each can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

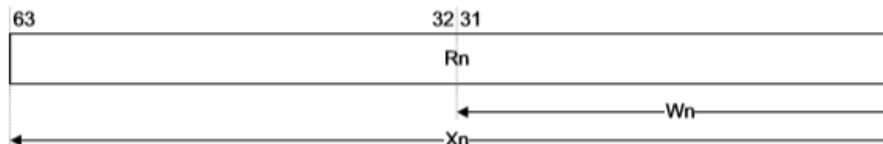


Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

————— Note ————

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

arm64 – Rappels assembleur pour le reverse

B1.3 Process state, PSTATE

Process state, or PSTATE, is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

For the system level view of PSTATE, see *Process state, PSTATE* in Chapter D1.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

arm64 – Les appels de fonctions –

Les paramètres sont passés par les registres suivants:

- \$x0
- \$x1
- \$x2
- \$x3
- \$x4

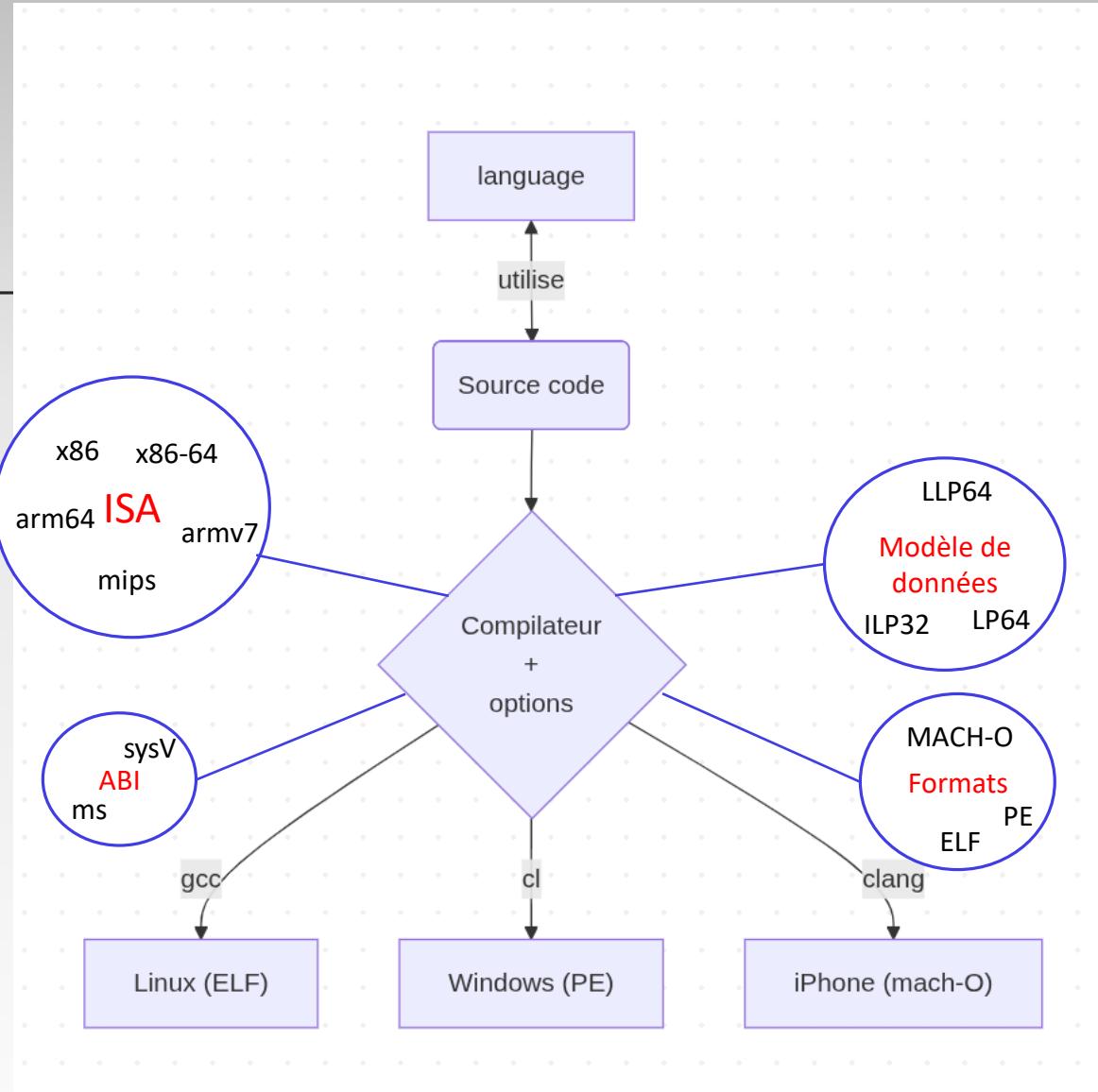
...

arm64 – Les appels de fonctions –

```
int main( void )
{
    printf("[+] I'm the caller. I will call the function callee: \n");
    int res = callee( 0x1111, 0x2222, 0x3333, 0x4444, 0x5555, 0x6666, 0x7777, 0x8888 );
    printf("[+] I just returned from callee: %d\n", res);
    return res;
}
```

```
43   main:
44       stp    x29, x30, [sp, -32]!
45       mov    x29, sp
46       adrp   x0, .LC1
47       add    x0, x0, :lo12:.LC1
48       bl     puts
49       mov    w7, 34952
50       mov    x6, 30583
51       mov    w5, 26214
52       mov    x4, 21845
53       mov    w3, 17476
54       mov    x2, 13107
55       mov    w1, 8738
56       mov    x0, 4369
57       bl     callee
58       str    w0, [sp, 28]
```

MAP



Les formats – Introduction

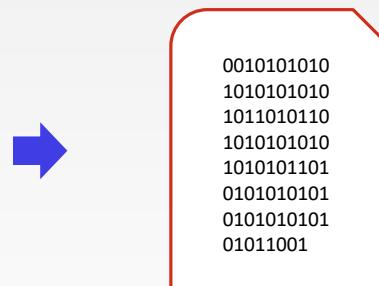
- Un format est un packaging structuré d'un binaire.
- Il est issu d'un compilateur et d'un linker
- Il est toujours compréhensible par un loader
- Exemples de format :
 - PE (portable executable) utilisé dans le monde de Microsoft
 - ELF (Executable et Linked File) utilisé dans le monde Unix
 - Mach-O : utilisé dans le monde Apple
 - Liste non exhaustive

Les formats : Généralités

Les formats : Généralités

Un format binaire est une spécification qui définit la structure d'un programme afin qu'il soit bien compris par le loader correspondant.

Il est nécessairement lié à un loader qui va prendre en entrée le fichier brute sur le disque et le transformer en une unité d'exécution du système.

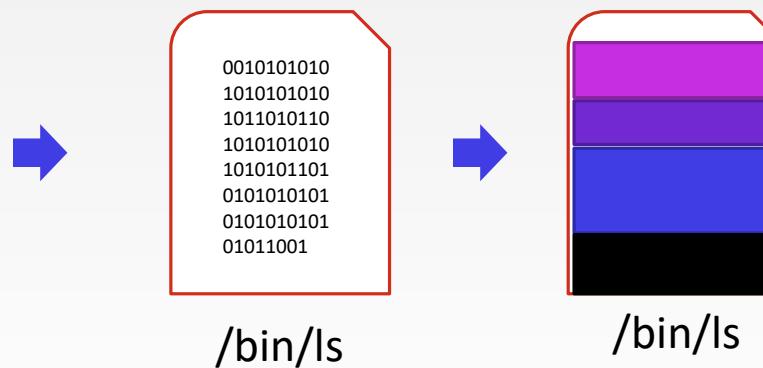


/bin/ls

Les formats : Généralités

Un format binaire est une spécification qui définit la structure d'un programme afin qu'il soit bien compris par le loader correspondant.

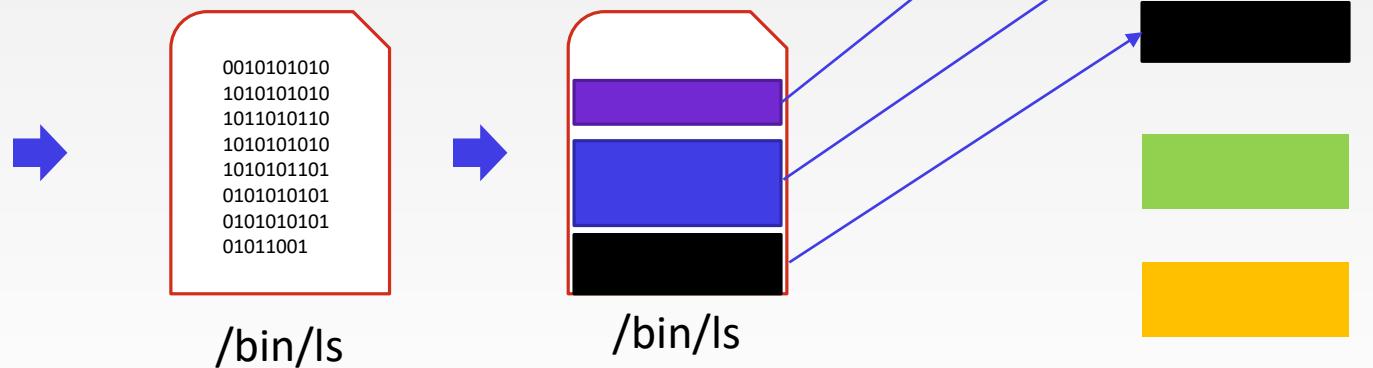
Il est nécessairement lié à un loader qui va prendre en entrée le fichier brute sur le disque et le transformer en une unité d'exécution du système.



Les formats : Généralités

Un format binaire est une spécification qui définit la structure d'un programme afin qu'il soit bien compris par le loader correspondant.

Il est nécessairement lié à un loader qui va prendre en entrée le fichier brute sur le disque et le transformer en une unité d'exécution du système.



0x0000000000000000
bibliothèques

partie 1

partie 2

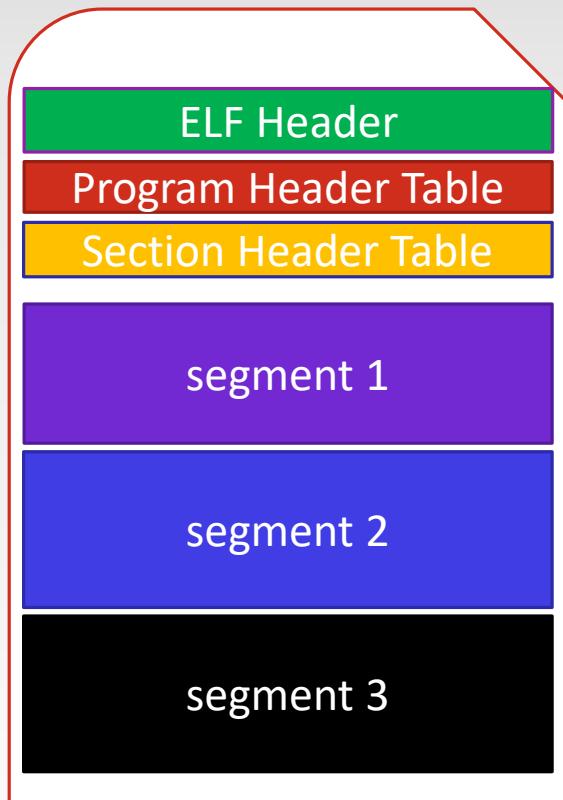
partie 3

tas

stack

0x00007FFFFFFFFF

Le format ELF : les bases



	0h	608h	struct	
> elf_header	0h	40h	struct	The main e
program_header_table	40h	2D8h	struct	Text
> program_table_element[13]	40h	2D8h	struct program...	Program he
section_header_table	3910h	740h	struct	
section_table_element[29]	3910h	740h	struct section_t...	
> section_table_element[0] SHN_UNDEF	3910h	40h	struct section_t...	
> section_table_element[1] .interp	3950h	40h	struct section_t...	
> section_table_element[2] .note.gnu.property	3990h	40h	struct section_t...	
> section_table_element[3] .note.gnu.build-id	39D0h	40h	struct section_t...	
> section_table_element[4] .note.ABI-tag	3A10h	40h	struct section_t...	
> section_table_element[5] .gnu.hash	3A50h	40h	struct section_t...	
> section_table_element[6] .dynsym	3A90h	40h	struct section_t...	
> section_table_element[7] .dynstr	3AD0h	40h	struct section_t...	
> section_table_element[8] .gnu.version	3B10h	40h	struct section_t...	
> section_table_element[9] .gnu.version_r	3B50h	40h	struct section_t...	
> section_table_element[10] .rela.dyn	3B90h	40h	struct section_t...	
> section_table_element[11] .init	3BD0h	40h	struct section_t...	
> section_table_element[12] .plt	3C10h	40h	struct section_t...	
> section_table_element[13] .plt.got	3C50h	40h	struct section_t...	
> section_table_element[14] .text	3C90h	40h	struct section_t...	
> section_table_element[15] .fini	3CD0h	40h	struct section_t...	
> section_table_element[16] .rodata	3D10h	40h	struct section_t...	
> section_table_element[17] .eh_frame_hdr	3D50h	40h	struct section_t...	
> section_table_element[18] .eh_frame	3D90h	40h	struct section_t...	
> section_table_element[19] .init_array	3DD0h	40h	struct section_t...	
> section_table_element[20] .fini_array	3E10h	40h	struct section_t...	
> section_table_element[21] .dynamic	3E50h	40h	struct section_t...	
> section_table_element[22] .got	3E90h	40h	struct section_t...	
> section_table_element[23] .data	3ED0h	40h	struct section_t...	
> section_table_element[24] .bss	3F10h	40h	struct section_t...	
> section_table_element[25] .comment	3F50h	40h	struct section_t...	
> section_table_element[26] .symtab	3F90h	40h	struct section_t...	
> section_table_element[27] .strtab	3FD0h	40h	struct section_t...	
> section_table_element[28] .shstrtab	4010h	40h	struct section_t...	
> symbol_table	3040h	5D0h	struct	Text
> symtab[62]	3040h	5D0h	struct Elf64_Sym	Text
dynamic_symbol_table	3C8h	90h	struct	Text
pt_dynamic_symbol_table	3C8h	240h	struct	Text
> symtab[24]	3C8h	240h	struct Elf64_Sym	Text

Le format ELF – les bases

Les sections importantes:

- .text : contient les instructions du programme
- .rodata : contient des données en Read Only
- .data : contient des données R/W
- .plt / .got : utilisées pour faire l'édition de lien dynamique avec des fonctions externes

Le format PE – les bases

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF	
0:0000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..	
0:0010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	,.....@.....	
0:0020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@.....	
0:0030	00	00	00	00	00	00	00	00	00	00	00	00	F8	00	00	00@.....	
0:0040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..@..!i!.Li!Th	
0:0050	69	73	20	70	72	6F	67	72	61	60	20	63	61	6E	6E	6F	is program canno	
0:0060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS	
0:0070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode...\$.....	
0:0080	18	A1	CA	3B	5C	00	A4	68	5C	00	A4	68	.jE\À�h\À�h\À�h					
0:0090	17	B8	A7	69	56	00	A4	68	17	B8	A1	69	D1	00	A4	68	..,SiVÀ�h..,iNÀ�h	
0:00A0	17	B8	A0	69	48	00	A4	68	17	B8	A5	69	5F	00	A4	68	..,iHÀ�h..,yìÀ�h	
0:00B0	5C	00	A5	68	00	00	A4	68	46	44	A7	69	48	00	A4	68	\Avh\À�hD5iHÀ�h	
0:00C0	4C	44	A0	69	4E	00	A4	68	4C	44	A1	69	7A	00	A4	68	LD iNÀ�hD1zÀ�h	
0:00D0	17	A5	A0	69	5D	00	A4	68	17	A5	69	5D	00	A4	68	.E ijÀ�h.E ij]À�h		
0:00E0	52	69	63	68	5C	00	A4	68	00	00	00	00	00	00	00	00	Rich\À�h.....	
0:00F0	00	00	00	00	00	00	00	00	50	45	00	00	AC	01	04	00	.PE.L...	
0:0100	77	C9	3B	67	00	00	00	00	00	00	E0	00	02	01	00	00	WÉ8g.....à..	
0:0110	0B	01	29	00	D2	00	00	00	84	00	00	00	00	00	00	00	...)ò.....@..	
0:0120	85	12	00	00	00	00	10	00	00	00	F0	00	00	00	00	00@..	
0:0130	00	10	00	00	00	02	00	00	06	00	00	00	00	00	00	00@..	
0:0140	06	00	00	00	00	00	00	00	80	01	00	00	04	00	00	00@..	
0:0150	00	00	00	00	03	00	40	81	00	00	10	00	00	10	00	00@..	
0:0160	00	00	00	10	00	00	10	00	00	00	00	00	00	10	00	00@..	
0:0170	00	00	00	00	00	00	00	00	34	48	01	00	28	00	00	00@..	
0:0180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:0190	00	00	00	00	00	00	00	00	00	70	01	00	38	00	00	00@..	
0:01A0	B0	3D	01	01	1C	00	00	00	00	00	00	00	00	00	00	00@..	
0:01B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:01C0	FO	3C	01	01	40	00	00	00	00	00	00	00	00	00	00	00	δ<..@..	
0:01D0	00	ED	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:01E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:01F0	2E	74	65	78	74	00	00	00	B3	00	00	00	10	00	00	00	.text..^D.....	
0:0200	00	D2	00	00	00	04	00	00	00	00	00	00	00	00	00	00@..	
0:0210	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00@..rdata..	
0:0220	40	5E	00	00	00	F0	00	00	00	60	00	00	D6	00	00	00	e^..@..@..0..	
0:0230	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40@..@..@..@..	
0:0240	2E	64	61	74	61	00	00	00	80	13	00	00	00	50	01	00	.data..€..P..	
0:0250	00	0A	00	00	00	00	36	01	00	00	00	00	00	00	00	006..	
0:0260	00	00	00	00	40	00	00	00	C0	2E	72	65	6C	6F	63	00	00@..reloc..
0:0270	38	0F	00	00	00	70	01	00	00	10	00	00	00	40	01	00	8....p.....@..	
0:0280	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	42@..@..B..	
0:0290	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:02A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:02B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:02C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	
0:02D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00@..	

Name	Value	Start	Size	Type	Color
> DosHeader		0h	40h	struct IMAGE_D...	
> DosStub		40h	A8h	struct IMAGE_D...	
> NtHeader		F8h	F8h	struct IMAGE_N...	
> SectionHeaders[4]		1F0h	A0h	struct IMAGE_S...	
> Section[0]	.text	400h	D200h	struct IMAGE_S...	
> Section[1]	.rdata	D600h	6000h	struct IMAGE_S...	
> Section[2]	.data	13600h	A00h	struct IMAGE_S...	
> Section[3]	.reloc	14000h	1000h	struct IMAGE_S...	
> ImportDescriptor	KERNEL32.dll	12E34h	14h	struct IMAGE_I...	
> RelocTable		14000h	F38h	struct BASE_RE...	21
> DebugDirectory[1]		123B0h	1Ch	struct IMAGE_D...	

Le format PE – les bases

Les sections importantes:

- .text
- .rodata
- .data
- .reloc pour la gestion des relocations

La table des imports est utilisée pour l'édition de lien vers des fonctions externes

Un exemple pour finir

CRACKME 101

```
00000000000011a9 <main>:
11a9: f3 0f 1e fa        endbr64
11ad: 55                 push    rbp
11ae: 48 89 e5           mov     rbp,rs
11b1: 48 83 ec 20       sub    rsp,0x20
11b5: 89 7d ec           mov     DWORD PTR [rbp-0x14],edi
11b8: 48 89 75 e0       mov     QWORD PTR [rbp-0x20],rsi
11bc: 48 8d 05 41 0e 00 00 lea    rax,[rip+0xe41]      # 2004 <_IO_stdin_used+0x4>
11c3: 48 89 45 f8       mov     QWORD PTR [rbp-0x8],rax
11c7: 48 8b 45 f8       mov     rax,QWORD PTR [rbp-0x8]
11cb: 48 89 c7           mov     rdi,rax
11ce: e8 bd fe ff ff    call   1090 <strlen@plt>
11d3: 89 45 f4           mov     DWORD PTR [rbp-0xc],eax
11d6: 83 7d ec 02       cmp    DWORD PTR [rbp-0x14],0x2
11da: 74 0a               je    11e6 <main+0x3d>
11dc: bf 02 00 00 00       mov     edi,0x2
11e1: e8 ca fe ff ff    call   10b0 <exit@plt>
11e6: 48 8b 45 e0       mov     rax,QWORD PTR [rbp-0x20]
11ea: 48 83 c0 08       add    rax,0x8
11ee: 48 8b 00           mov     rax,QWORD PTR [rax]
11f1: 48 89 c7           mov     rdi,rax
11f4: e8 97 fe ff ff    call   1090 <strlen@plt>
11f9: 8b 55 f4           mov     edx,DWORD PTR [rbp-0xc]
11fc: 48 39 d0           cmp    rax,rdx
11ff: 74 0a               je    120b <main+0x62>
1201: bf 02 00 00 00       mov     edi,0x2
1206: e8 a5 fe ff ff    call   10b0 <exit@plt>
120b: 8b 55 f4           mov     edx,DWORD PTR [rbp-0xc]
120e: 48 8b 45 e0       mov     rax,QWORD PTR [rbp-0x20]
1212: 48 83 c0 08       add    rax,0x8
1216: 48 8b 00           mov     rax,QWORD PTR [rax]
1219: 48 8b 4d f8       mov     rcx,QWORD PTR [rbp-0x8]
121d: 48 89 ce           mov     rsi,rcx
1220: 48 89 c7           mov     rdi,rax
1223: e8 58 fe ff ff    call   1080 <strcmp@plt>
1228: 85 c0               test   eax,eax
122a: 74 0a               je    1236 <main+0x8d>
122c: bf 02 00 00 00       mov     edi,0x2
1231: e8 7a fe ff ff    call   10b0 <exit@plt>
1236: 48 8d 3d d3 0d 00 00 lea    rdi,[rip+0xdd3]      # 2010 <_IO_stdin_used+0x10>
123d: b8 00 00 00 00       mov     eax,0x0
1242: e8 59 fe ff ff    call   10a0 <printf@plt>
1247: b8 00 00 00 00       mov     eax,0x0
124c: c9                 leave 
124d: c3                 ret
124e: 66 90               xchg   ax,ax
```

```
h@h:~/BachelorCyber$ python3 -c "print(hex(0x11bc+0xe41+7))"
0x2004
h@h:~/BachelorCyber$ hexdump -C -s "0x2004" -n 20 a.out
000002004  4d 6f 6e 50 61 73 73 77  6f 72 64 00 4f 6b 21 00  |MonPassword.Ok!.|
000002014  01 1b 03 3b                                     |...;|
000002018
```

Questions ?

MERCI POUR VOTRE ATTENTION