# Travaux dirigés Système n°5 : assembleur

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

## 1. Avant de commencer

Avant de commencer à écrire nos programmes en langage d'assemblage, il est nécessaire d'éclaircir certains points.
Vous aurez remarqué que nous parlons ici de langage d'assemblage et d'assembleur de manière distincte. En effet, l'assembleur est le programme qui va convertir le langage d'assemblage en binaire. C'est exactement comme le rôle du compilateur qui va traduire le langage C en binaire ; sauf qu'ici, on parle d'assembleur.
Beaucoup disent qu'ils « programment en assembleur »pour faire un raccourci. Il s'agit d'un amalgame communément accepté, mais qui ne plait pas aux puristes. Ainsi, dans cet article, nous ferons la part des choses entre le langage d'assemblage et l'assembleur.

Un fichier contenant du langage d'assemblage comporte généralement l'extension .asm. Pour assembler ces fichiers, nous utiliserons nasm. Son acronyme signifie « Netwide ASseMbler », il s'agit de l'assembleur de référence pour les architectures x86-64 d'Intel/AMD. Son site officiel est au goût du jour et présentera les possibilités de cet assembleur.
L'assembleur nasm est disponible dans les paquets de nombreuses distributions. Sous système Debian ou Ubuntu, la commande suivante permet d'installer les packages nécessaires :

```
sudo apt install nasm gcc-multilib
```

Commençons par un premier programme : un Hello World en utilisant la bibliothèque standard C).

## 2. « Hello world », Intel x86, en utilisant la libc

Ouvrez votre éditeur préféré et inscrivez-y le contenu suivant que nous allons étudier pas-à-pas.

```
bits 32
extern printf
global main

section .data
Hello db "Hello world!", 10, 0


section .text

main:
```

```
    push ebp
    mov ebp, esp
    sub esp, 4
    mov dword [esp], Hello
    call printf
    xor eax, eax
    leave
    ret
```

Commençons par nous attarder sur les trois premières lignes :

```
bits 32
global main
extern printf
```

La première ligne indique que le code que nous écrirons par la suite sera du code 32-bit. Cette directive n'est pas indispensable dans le cas présent puisqu'il n'y a aucune ambiguïté sur le code qui suivra, mais il fait toujours bon de l'insérer à titre informatif.

Les deux lignes suivantes sont intéressantes et vont servir à l'éditeur de liens, ou *linker*.
La seconde directive, **global main**, indique que nous avons une étiquette **main** au sein de notre programme qui devra être considérée comme un symbole public. Sans ça, l'éditeur de lien ne pourra pas trouver où se situe la fameuse « fonction main »que vous écririez en C.
Enfin, la troisième ligne **extern printf** indique à l'assembleur qu'il doit faire fi de ce symbole et que celui-ci sera résolu au moment de l'édition de liens. En effet, le code de la fonction *printf* se situe dans la libc. Notre binaire final se chargera d'appeler ladite fonction et d'exécuter son code au moment de l'appel. Nous verrons plus tard comment appeler une directive plus bas niveau pour afficher du texte à l'écran.

Attardons-nous sur la suite du programme :

```
section .data
Hello db "Hello world!", 10, 0
```

La première ligne indique que nous déclarons le contenu de la « section .data ».
Par convention, la section nommée « .data »contient des données accessibles par votre programme, le plus souvent en lecture et parfois en écriture également. Mais jamais en exécution, à moins qu'il s'agisse d'un binaire réécrit à la main ou autre. En résumé, à partir de cette ligne, tout ce que nous déclarerons sera stocké dans la section .data.

Analysons de près la seconde ligne :

```
Hello db "Hello world!", 10, 0
```

Elle va nous permettre de déclarer des données dans notre section .data. Ces données seront référencées au moyen d'un symbole nommé « Hello ». La directive db signifie littéralement *data byte*. Elle permet de spécifier que l'unité des données décrites par la suite sera huit bits (« one byte », « un octet »...). La syntaxe générale est la suivante :

```
Symbol db 0xde, 0037, 137, 0b0110, "Some String"
```

Il est en effet possible de déclarer des octets en hexadécimal (0xde ou 0deh avec le suffixe « h »), en octal (0037), en décimal (137), en binaire (0b0110) et en chaîne de caractères (« Some String »).

Pour en revenir à notre ligne, nous avons déclaré « Hello world », suivi des octets 10 et 0. Les habitués du langage C remarqueront que la valeur décimale 10 représente dans le code ASCII le fameux « Line Feed »qui permet de revenir à la ligne, et que le 0 n'est autre que le fameux null-byte de terminaison de chaîne de caractère, soit \0 si cela vous convient peu.

En passant rapidement les autres directives, pour pourrez déclarer des mots de 16 bits via **dw** (declare word), des doubles mots de 32 bits via **dd** (declare double (word)) ou des quadruples mots de 64 bits via **dq** (declare quadruple (word)).

Nous avons déclaré toutes les données dont nous avons besoin, à savoir notre chaîne à afficher à l'écran. Attaquons-nous maintenant à la section .text. Les explications vont être rapides :

```
section .text

main:
    ; ... code
```

Nous déclarons en premier lieu que nous ne sommes plus dans la section .data, mais bien dans la section .text. Il faut savoir que celle-ci contient, par convention, le code machine destiné à être exécuté. L'assembleur saura quels droits affecter à ladite section s'il reconnaît son nom, à savoir les droits d'exécution.

Enfin, nous déclarons une étiquette, un libellé que nous appellerons « main ». Vous reconnaissez ici qu'il s'agit du début de notre traditionnelle fonction main en C. Au sens de l'assembleur, il s'agit d'une simple étiquette pour permettre de référencer plus facilement une destination par rapport à une adresse mémoire brute. Mais le fait d'avoir considéré cette étiquette comme « globale »comme nous l'avons fait plus haut au sein de notre code permet de spécifier à l'éditeur de lien qu'il s'agira d'un symbole exporté.

Vient ensuite le code du programme en lui-même :

```
    push ebp
    mov ebp, esp
    sub esp, 4
    mov dword [esp], Hello
    call printf
    xor eax, eax
    leave
    ret
```

Nous repérons ici :

- Le prologue de la fonction qui va mettre en place le cadre de pile et allouer son propre espace mémoire au sein de la pile d'exécution, ceci afin de préserver les données utilisées par la fonction parente ;
- L'instruction **mov dword [esp], Hello** qui dépose au sommet de la pile l'adresse mémoire qui pointe sur nos données que la fonction printf va afficher. Il est possible de réaliser cette instruction au lieu d'un **push Hello** puisque nous avons précédemment fait un **sub esp, 4**. L'intérêt de faire un « sub »au lieu de « push »successifs permet d'allouer de la place dans la pile d'exécution d'une seule traite ;

- L'appel à **printf** et le **xor eax, eax** qui met à zéro le registre éponyme pour mettre à jour le code de retour de notre fonction ;
- L'épilogue de la fonction.

Enregistrez le code sous un fichier hello.asm et exécutez la ligne de commande suivante :

```
nasm -f elf32 hello.asm -o hello.o
```

Cette ligne de commande va produire le fameux fichier .o que les programmeurs C reconnaîtront. Nous avons ici fait de l'assemblage et non de la compilation. Il en ressort que nous avons un fichier .o prêt à être lié à d'autres afin de produire un binaire exécutable, au format ELF. En effet, la commande suivante ôte toute ambiguïté quant au contenu de notre fichier :

```
file hello.o

hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV),
not stripped
```

Pour faire l'édition de liens, utilisez votre compilateur C favori GCC :

```
gcc -o hello hello.o -m32
```

N'oubliez pas l'option -m32 si vous êtes sous un système 64-bit car nous voulons produire un binaire 32-bit.

À la fin de l'exécution de ces deux commandes, nous avons notre binaire fonctionnel :

```
./hello
Hello world!

file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.26,
BuildID[sha1]=0x5f5a4592997bc9380054ebe8836b01f7d86ef24b,
not stripped
```

Nous allons voir comment écrire un programme qui n'utilise ni printf, ni fonction main, mais qui se contente tout de même d'afficher les mêmes informations à l'écran, ceci à l'aide d'appels systèmes.

## 3. « Hello world », Intel x86, en utilisant les appels systèmes

Puisque nous parlons d'assembleur, de bas niveau, de binaire, il y a une question qu'il nous semble légitime de nous poser : À quoi ressemble le code de printf ? Existe-t-il une ou des instructions miracles qui permettent d'afficher nos données à l'écran ? La réponse est oui !

En effet, les systèmes d'exploitation récents ont des noyaux qui mettent à disposition ce qu'on appelle des appels systèmes. Pour faire simple, il s'agit d'une interface que le noyau met à disposition pour les programmes de l'utilisateur. Il faut savoir que la gestion des fichiers, des entrées/sorties au sens large, des processus, de l'affichage graphique, etc... reposent sur des mécanismes compliqués. Ceux-ci sont gérés par votre système d'exploitation et votre noyau sans que vous ayez à vous soucier d'accéder, par exemple, au disque dur pour écrire sur un fichier, à la mémoire pour écrire une phrase à l'écran, etc...

Donc, grâce à ces appels systèmes, qui ressemblent fortement à des fonctions, il est possible d'accéder à des fichiers, de manipuler des sockets, des mutex, des sémaphores, etc... Traduit en anglais, le terme appel système donne « syscall ». Chaque appel système possède un numéro et des arguments.

Le programme que nous avons construit dans l'exemple 1 fait appel à printf. La fonction printf va elle-même utiliser un appel système pour afficher des données à l'écran. Pour connaître les appels systèmes employés par un programme, l'outil strace est parfaitement indiqué. Pour « stracer »le binaire « ./hello »que nous avons construit, rien de plus simple :

```
strace ./hello

execve("./hello", ["./hello"], [/* 17 vars */]) = 0
[ Process PID=13792 runs in 32 bit mode. ]
brk(0)                                  = 0x878d000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directo
...
write(1, "Hello world!\n", 13Hello world!
)           = 13
exit_group(0)                           = ?
```

Nous voyons qu'il y a pléthore d'appels systèmes appelés par notre programme ! Celui qui nous intéresse le plus, c'est celui-là :

```
write(1, "Hello world!\n", 13Hello world!
)           = 13
```

Il s'agit de l'appel système « write ». Pour avoir des informations sur cet appel système, la commande *man 2 write* vous donnera toutes les informations dont vous aurez besoin, compris le prototype C.

```
#include <unistd.h>

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

En résumé, **write** va écrire *count* octets pointés par *buf* dans le descripteur *fd*. C'est bien plus compliqué que si nous appelions printf...

Notre programme précedent, en utilisant printf, a appelé write de la sorte :

```
write(1, "Hello world!\n", 13);
```

Nous identifions ici la valeur de nos deux derniers arguments : a chaîne à afficher et sa taille. Quant au premier argument, il correspond tout simplement à *stdout*, la « sortie standard »dont le descripteur vaut 1 sur les systèmes Linux. Il ne nous reste plus qu'à savoir comment appeler cet appel système en langage d'assemblage.

Pour cela, nous allons utiliser ce qu'on appelle une interruption logicielle. Le noyau Linux met en effet à disposition l'interruption 0x80 - 128 en décimal - qui permet d'effectuer un appel système au niveau des logiciels utilisateurs. Cette information est décrite et disponible dans ce qu'on appelle l'**ABI** - *Application Binary Interface*. C'est sensiblement le même principe d'une API, sauf qu'ici, on se met d'accord sur des conventions au niveau binaire.

L'ABI x86 sur système Linux impose les standards suivants lors d'un appel système :

- Le registre **eax** contiendra le numéro de l'appel système ;
- Le registre **ebx** contiendra le premier argument de l'appel système ;
- Le registre **ecx** contiendra le second/deuxième argument de l'appel système ;
- Le registre **edx** contiendra le troisième argument de l'appel système ;
- Le registre **esi** contiendra le quatrième argument de l'appel système ;
- Le registre **edi** contiendra le cinquième argument de l'appel système.

Ceci sous réserve qu'un appel système ait besoin d'autant d'arguments, bien sûr. Dans notre cas, l'appel système *write* ne prend que trois arguments. On aura donc le schéma suivant :

- Le registre **eax** contiendra le numéro d'appel système de write ;
- Le registre **ebx** contiendra la valeur du descripteur de l'entrée sortie, soit 1 ;
- Le registre **ecx** contiendra l'adresse vers notre mémoire à écrire dans le descripteur (notre chaîne « Hello World\n ») ;
- Le registre edx contiendra le nombre de caractères à écrire dans le descripteur (13).

Il ne nous reste plus qu'à récupérer le numéro de l'appel système *write*. De nombreuses ressources sur internet vous permettront de l'identifier. Par exemple, sur le lien suivant : http ://-docs.cs.up.ac.za/programming/asm/derick _tut/syscalls.html , on nous informe que le numéro d'appel système de *write* est **4**. Sous système 64-bit, il est aisé de vérifier cette affirmation :

```
cat /usr/include/asm/unistd_32.h | grep write

#define __NR_write                4
#define __NR_writev             146
#define __NR_pwrite64           181
#define __NR_pwritev            334
#define __NR_process_vm_writev  348
```

Il s'agit bien du numéro 4 !

Il nous faut aussi pouvoir quitter le programme proprement, grâce à l'appel système *exit*. Celui-ci prend en unique argument la valeur de retour du programme. Le numéro d'appel système d'*exit* est le 1 comme le montre la commande ci-dessous :

```
cat /usr/include/asm/unistd_32.h | grep exit
#define __NR_exit                 1
#define __NR_exit_group         252
```

Nous pouvons désormais faire notre Hello world sans avoir besoin de la libc. Ouvrons notre fichier hello2.asm et inscrivons-y le code suivant :

```
bits 32

global _start

section .data
Hello db 'Hello world', 10

section .text
_start:
```

```
    push    ebp
    mov     ebp, esp
    mov     eax, 4 ; sys_write
    mov     ebx, 1 ; stdout
    mov     ecx, Hello
    mov     edx, 13
    int     0x80

    mov     eax, 1
    xor     ebx, ebx ; ebx = 0
    int     0x80
```

Nul besoin de prologue et d'épilogue de fonction. Nous allons exécuter du code situé directement au point d'entrée du programme, c'est-à-dire à partir du symbole *_start*. Dans un programme écrit en C, le code présent à partir de *_start* se chargerait de faire des opérations diverses et variées avant d'appeler main avec les arguments qui vont bien. Ici, onous faisons comme nous le souhaitons.
L'instruction *int* permet de déclencher une interruption au niveau logiciel. Ici, nous appelons l'interruption 0x80 avec nos registres correctement initialisés, comme expliqué plus haut.

Il ne nous reste qu'à assembler et lier notre programme, avec quelques différences avec l'exemple précédent :

```
nasm -f elf32 hello2.asm -o hello2.o
ld -m elf_i386 -o hello2 hello2.o
```

Nous utiliserons ici l'outil ld pour lier notre programme, en lui précisant que nous aurons un binaire 32-bit « émulé »sur système 64-bit. Toujours un ELF.

```
file hello2
hello2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, not stripped

./hello2
Hello world
```

Voyions enfin un troisième et dernier exemple, toujours sur les appels systèmes, mais sur une architecture 64-bit.

## 4. « Hello world », Intel x64, en utilisant les appels systèmes

En architecture 64 bits, le numéro des appels systèmes est sujet à changer d'une part, et **l'ABI** sera différente du 32 bits d'autre part. En effet, pour effectuer un appel système sur un Linux 64-bit, il faut avoir le schéma suivant :

- le registre **rax** doit contenir le numéro d'appel système ;
- le registre **rdi** doit contenir le premier argument ;
- le registre **rsi** doit contenir le deuxième/second argument ;
- le registre **rdx** doit contenir le troisième argument ;
- le registre **r10** doit contenir le quatrième argument ;
- le registre *r8* doit contenir le cinquième argument ;

- le registre *r9* doit contenir le sixième argument.

Vous pouvez consulter le document traitant de l'ABI de Linux x64 ici : http ://www.x86-64.org/documentation/a

Les numéros d'appels systèmes sont probablement différents. Il peut ne plus s'agir de 4 pour *write* ni de 1 pour *exit*. Vérifions :

```
cat /usr/include/asm/unistd_64.h | grep write
#define __NR_write                                1
__SYSCALL(__NR_write, sys_write)
[...]

cat /usr/include/asm/unistd_64.h | grep exit
#define __NR_exit                                 60
__SYSCALL(__NR_exit, sys_exit)
[...]
```

Il s'agit respectivement de 1 et 60.

Un dernier détail qui a toute son importance ; sur architecture x64, nous n'utiliserons plus l'interruption logicielle 0x80 pour effectuer un appel système, mais nous utiliserons à la place une instruction dédiée : l'instruction *syscall*. En plus d'être plus parlante, elle a l'avantage d'être plus rapide. En effet, l'instruction *int* provoque une interruption. Dans le domaine de l'architecture matérielle des ordinateurs, les interruptions sont réputées pour être lentes, car il y a tout un tas d'opérations « lourdes »à effectuer - aller chercher la routine à appeler pour l'interruption 0x80, sauvegarder le contexte d'exécution actuel... Or, notre instruction *syscall* s'occupera de faire un appel système sans passer par le mécanisme coûteux de l'interruption. Préférez donc exécuter des binaires 64 bits sur votre machine que des binaires 32 bits. C'est bien plus rapide !

Écrivons notre troisième et dernier programme une fois pour toutes :

```
bits 64
global _start

section .data
Hello db "Hello world!", 10

section .text
_start:
    mov rax, 1 ; sys_write
    mov rdi, 1 ; stdout
    mov rsi, Hello
    mov rdx, 13
    syscall

    mov rax, 60 ; sys_exit
    xor rdi, rdi ; exit(0)
    syscall
```

Nous assemblons et lions :

```
nasm -f elf64 hello3.asm -o hello3.o
ld -o hello3 hello3.o
```

Vérifions le fichier et exécutons-le :

```
file hello3

./hello3
Hello world!
```

Vous savez désormais écrire des petits programmes basiques en langage d'assemblage sous Linux.

*Hexagone 2023 - Ce TD est largement inspiré des cours de A. Michelizza (ANSSI) - Vous pouvez retrouver son travail en assembleur sur le site http ://a.michelizza.free.fr/pmwiki.php ?n=TutoOS.TutoOS*