Zeste de savoir

Programmez en langage d'assemblage sous Linux !

12 août 2019

Table des matières

Cet article se veut la suite logique de l'introduction à la rétroingénierie des binaires et de l'introduction aux ”buffer overflows” . Au travers de ces écrits, nous souhaitons faire écrire au lecteur ses premiers programmes écrits en langage machine.

Le fait qu'il soit la suite logique de l'introduction au *reverse engineering* est qu'il se veut être une suite pratique à cet article qui montrait comment fonctionnait un programme écrit à l'origine en C, notamment avec l'utilisation de la pile d'exécution.

De même, le fait qu'il soit la suite de l'introduction aux ”buffer overflows” est qu'il se veut **aussi** être une suite pratique pour l'exploitation de vulnérabilités plus ”critiques”. Parce que s'il s'avère nécessaire dans la plupart des cas de savoir comprendre un binaire désassemblé pour débusquer des vulnérabilités liées à des défauts de programmation (manque de rigueur, notamment), il peut être intéressant d'injecter son propre code au sein d'un processus. Ce code sera naturellement une suite d'instructions binaires. D'où la nécessité de savoir programmer en langage d'assemblage dans ces cas-là.

De fait, nous verrons trois exemples au travers de cet article. Le premier exemple illustrera un programme destiné aux architectures 32-bit basique qui se contentera de faire un ”Hello world” en utilisant la fonction `printf` de la bibliothèque standard C. Le deuxième exemple illustrera la même fonctionnalité que le premier, mais en utilisant cette fois-ci les **appels systèmes**. Enfin, le troisième et dernier exemple illustrera le même cas que son prédécesseur, mais pour les architectures 64-bit.

Vous êtes prêts ? Attachez vos ceintures, ça va secouer !

1. Avant de commencer

Avant de commencer à écrire nos programmes en langage d'assemblage, je me dois d'éclaircir certains points.

Vous aurez remarqué que je parle 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, je m'engagerai à faire 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 bien mieux que moi les possibilités de cet assembleur. La source officielle est en anglais, mais cela ne doit pas vous arrêter !

L'assembleur `nasm` est sûrement disponible dans les paquets de votre distribution. Étant utilisateur régulier de Debian, je vous communiquerai simplement la ligne magique pour installer l'assembleur :

```
sudo apt-get install nasm
```

Vous êtes prêt à écrire des programmes en langage d'assemblage ! Commençons par notre premier programme : un Hello World en utilisant la bibliothèque standard C (c'est possible, ça ?!).

2. Programmons en binaire !

2.1. Exemple 1 : "Hello world", Intel x86, en utilisant la libc

Ouvrez votre éditeur préféré et inscrivez-y le contenu suivant que nous décortiquerons 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 :

```
1 bits 32
2 global main
3 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 :

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

La première ligne indique que nous déclarons le contenu de la "section .data"...

> *i* Question aux lecteurs de l'article sur la rétroingénierie de binaires référencé dans l'introduction : vous vous souvenez du "format ELF" auquel je faisais référence ? Eh bien nous parlons ici des sections contenues dans un binaire structuré dans ce format ! Pour rappel, une section a des propriétés (son adresse mémoire de base, sa taille physique dans le fichier, la taille qu'elle prendra en mémoire, des droits d'accès...) et un contenu. C'est tout ce que vous avez à savoir pour ce qui va suivre.

Par convention, la section nommée ".data" contient... Des données ! Rien de plus que des données accessibles par votre programme, le plus souvent en lecture et parfois en écriture également. Mais jamais (ô grand 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 :

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

Celle-ci 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 :

```
1 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"). Les possibilités sont grandes si vous souhaitez représenter votre information d'une certaine manière.

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 "à la C". Le fameux `\0` si vous préférez.

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 !

```
1 section .text
2
3 main:
4     ; ... code
```

On déclare 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. Pourquoi .text et pas .code ? Aucune idée. Sachez juste qu'il s'agit d'une convention, et que l'assembleur saura quels droits affecter à ladite section s'il reconnaît son nom, à savoir les droits d'exécution !

Enfin, on déclare une **étiquette**, un **libellé** que nous appellerons... `main`. Vous reconnaissez là qu'il s'agit du début de notre traditionnelle fonction `main` en C ?

Ici, c'est pareil. Au sens de l'assembleur, il s'agit juste d'une simple étiquette pour permettre de référencer plus facilement une destination que 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
```

C'est là que je ne veux pas perdre énormément de temps sur les explications du code et que j'aimerais vous renvoyer à l'article sur la rétro-ingénierie des binaires. Je vous dirai juste qu'après compréhension des tenants et des aboutissants, il est aisé de repérer :

— 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.

Et si l'on construisait tout ça ? Enregistrez le code que je vous ai donné 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. Au risque de me répéter, nous avons ici fait de l'assemblage et non de la compilation. Il en ressort juste 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 :

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

Ôte toute ambiguïté quant au contenu de notre fichier. Pour faire l'édition de liens, utilisez votre compilateur C favori. Pour ma part, j'utilise gcc, mais cela dépend de tout un chacun !

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

N'oubliez pas l'option -m32 si vous êtes sous un système 64-bit, évidemment. 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
```

Et j'espère que vous en voulez encore, car ce n'est pas fini ! Nous allons voir comment écrire un programme qui n'utilise ni `printf`, ni la fonction `main`, mais qui se contente tout de même d'afficher les mêmes informations à l'écran, ceci à l'aide d'**appels systèmes** !

2.2. Exemple 2 : "Hello world", Intel x86, en utilisant les appels systèmes

Puisqu'on parle d'assembleur, de bas niveau, de binaire, il y a une question qu'il nous semble légitime de nous poser :

> *i* 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.

Et 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 et j'en passe !

On traduit en anglais le terme appel système par "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 ceux qui ne l'auraient pas sur leur machine, strace est sûrement disponible dans les dépôts de votre distribution linux.

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 directory)
...
write(1, "Hello world!\n", 13Hello world!
)           = 13
exit_group(0)                           = ?
```

On voit qu'il y a pléthore d'appels systèmes appelés par notre programme !

Je ne vais pas vous demander de comprendre tous les appels systèmes qui ont été 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, vous l'aurez deviné, 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. Y compris le prototype C :

```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, en utilisant `printf`, a appelé `write` de la sorte :

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

On arrive à mettre en évidence la valeur de nos deux derniers arguments - la 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.

> *i* Je sais ! Il suffit d'utiliser la directive `extern write` et de faire un `call write` en ayant inséré tous les arguments qui vont bien sur la pile d'exécution !

Et la réponse est... Oui, mais non !

En effet, la libc met à disposition ce qu'on appelle un "wrapper" autour de l'appel système `write`. Une interface d'appel, si vous préférez. Mais nous, on veut **vraiment** faire un appel système en pur et dur !

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 standard suivant 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. Vous savez ? Le fameux `0` qui indique que tout s'est bien passé !

Le numéro d'appel système d`exit` est le... ?

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

C'est le numéro 1 !

OUF ! Cela en fait, des explications pour simple hello world au plus bas niveau possible !

Nous pouvons faire notre Hello world sans avoir besoin de la libc désormais. 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
```

Ici, nul besoin de prologue et d'épilogue de fonction. On va 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, on fait **ce qu'on veut**.

L'instruction `int`, vous l'aurez deviné, permet de déclencher une interruption au niveau logiciel. Ici, on appelle l'interruption 0x80 avec nos registres correctement initialisés, comme je l'ai expliqué plus haut.

On n'a plus qu'à assembler et lier notre programme ! Cette fois-ci, ce sera un poil différent :

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

On utilisera en effet 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, bien entendu !

Le binaire sera tout autant fonctionnel que dans le premier exemple :

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

Et si nous voyions un troisième et dernier exemple, toujours sur les appels systèmes, mais sur une architecture 64-bit désormais ? Vous verrez que peu de choses changent et que l'exemple sera présent surtout pour la postérité, afin que vous puissiez rapidement vous faire la main sur le langage d'assemblage intel x64 si le coeur vous en dit !

2.3. Exemple 3 : "Hello world", Intel x64, en utilisant les appels systèmes

La partie de cet article sera plutôt rapide par rapport à la précédente, puisque nous nous passerons d'explications redondantes cette fois-ci.

Il y a juste deux-trois choses à savoir : vous vous douterez que le numéro des appels systèmes est sujet à changer d'une part, et que l'**ABI** sera différente 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.

Pour les curieux de la chose, vous pouvez consulter le document traitant de l'**ABI** de Linux x64 ici : http ://www.x86-64.org/documentation/abi.pdf

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

```
 % 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.

i

En résumé, préférez exécuter des binaires 64 bits sur votre machine que des binaires 32 bits. C'est 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
```

```
14
15      mov rax, 60 ; sys_exit
16      xor rdi, rdi ; exit(0)
17      syscall
```

On assemble et lie ?

```
1 % nasm -f elf64 hello3.asm -o hello3.o
2 % ld -o hello3 hello3.o
```

Aucun problème : vérifions le fichier et exécutons-le.

```
1  % file hello3
2  % ./hello3
3 Hello world!
```

Félicitations, lecteurs. Vous venez de fouler les terres inconnues d'un monde passionnant, dont l'étendue nous donne le vertige !

Alors, pas perdus ?

Vous savez désormais écrire des petits programmes basiques en langage d'assemblage sous Linux. Appelons un chat un chat : vous savez programmer en **binaire** !

L'utilisation d'un assembleur et de mnémoniques nous est bénéfique puisque la dernière étape serait carrément d'ouvrir un éditeur hexadécimal pour taper nos octets nous-mêmes. Que je vous rassure si l'idée vous traverse l'esprit : vous n'êtes pas forcément masochistes puisque cette technique est utilisée pour **patcher** à froid certains binaires. Notamment par les casseurs de logiciels qui font en sorte que votre version de Photoshop valide n'importe quelle clef-cd, par exemple. Mais programmer de but en blanc une routine en hexadécimal... Là, je vous souhaite bonne chance !

Nous n'avons vraiment vu que les bases, puisqu'il ne s'agit pas d'un tutoriel à proprement parler, mais d'un article. Si vous tenez vraiment à écrire des programmes avec plus de fonctionnalités, des conditions et j'en passe, renseignez-vous sur les instructions de branchement et autre. Il y a de quoi faire. Ou... Compilez un programme écrit en C et désassemblez-le ? C'est un excellent moyen d'apprendre, n'est-ce pas ?

Les possibilités sont énormes. De l'apprentissage et l'amusement au développement de routines sophistiquées au sein de programmes originellement écrits en C ou C++, le langage d'assemblage se défend encore assez et garde une place intéressante dans le milieu de l'informatique.