Débuter l'assembleur avec MASM32

Eh oui, ça y est, je me décide *enfin* à écrire un tutoriel sur le langage assembleur avec MASM32. Je me suis souvent dis que ça ne tiendrait pas la route, en raison de mes vagues connaissances à ce sujet. Mais c'est le fait d'écrire ce tutoriel (je ne sais pas s'il sera grand, ça, on verra plus tard) qui consolidera peut-être ce savoir. Et, bien évidemment, j'espère que ce "savoir" se diffusera aussi aux lecteurs, c'est-à-dire vous !

Pour ceux qui ne connaissent pas le codage de l'information et d'autres pré-requis, absolument pas de panique. Ayant envie d'entrer dans le vif du sujet dès le début, je mettrai les pré-requis en annexes. Car normalement, pour commencer l'assembleur, il faut être patient. Les tutoriels vous demandent de maîtriser le codage de l'information en bases 2, 10 et 16 ; c'est pas bien dur, hein, on s'y fait. Ils vous listent également l'histoire de l'architecture des processeurs INTEL, qui sont le cerveau de l'ordinateur. Ce qu'il faut savoir - je pense - c'est que le langage assembleur est en fait un intermédiaire entre le langage machine compris par le processeur, et l'humain. Le langage assembleur est donc, après le langage machine, le langage le plus bas niveau dans la programmation. Un schéma, pour résumer :

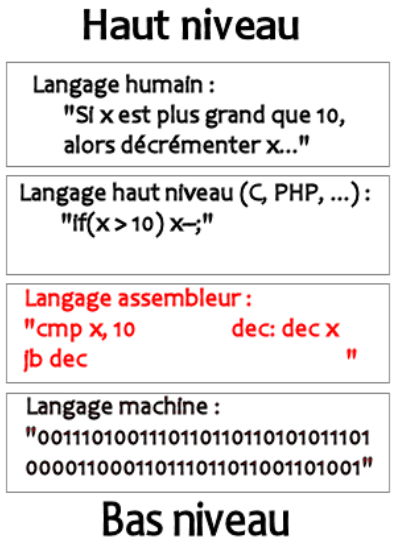


Schéma hiérarchique des langages

Je vous préviens, j'ai écrit n'importe quoi dans "assembleur" et "binaire" ; c'est juste pour vous schématiser la tronche des langages, quoi.

En résumé, ce tutoriel va vous apprendre à "parler" le langage du processeur. Ca serait bête de vous apprendre à écrire une suite d'octets en hexadécimal (même si ça se fait, pour concevoir des shellcodes ou taper dans les fichiers binaires via un éditeur hexadécimal) alors qu'on a **le langage assembleur** pour faire ça.

Rhaaa, je vous mets en appétit, mais je dois tout de même vous mettre en garde sur certains pré-requis à avoir.

- Faire de l'assembleur, je ne vous le cache pas, c'est très compliqué. En plus, on va en faire sous Windows, et je vous demande d'avoir des bases en C. C'est **impératif**, et vous comprendrez plus tard pourquoi.
- Tourner sur un windows 32 bits. Si vous êtes sur une architecture 64 bits, j'ai le regret de vous annoncer que vous ne pourrez vous servir de ce tutoriel qu'à des fins théoriques.
- J'allais oublier *the most important* : avoir windows.

Enfin, je tiens à préciser que ce tutoriel s'adresse aux curieux comme moi, et à ceux qui veulent un peu comprendre comment ça fonctionne. Je ne ferai jamais de vous des programmeurs en assembleur. Ce langage est dépassé et ne vous servira pas à écrire une application complète (quel intérêt, quand on a des langages évolués ?). C'est juste pour l'amusement.

Petits trucs "de base" à savoir

Dans notre ordinateur, on a plusieurs types de mémoire :

- RAM (Random Access Memory) : la mémoire vive, donc tous les logiciels se servent pour stocker des informations dedans. Cette mémoire est grande, rapide, et se vide lorsqu'il n'y a plus de courant qui l'alimente. Néanmoins, on peut en trouver certaines qui sont constamment alimentées avec une pile de lithium ;
- ROM (Read-Only Mémory) : la mémoire morte, ou "en lecture seule" comme on le traduit de l'anglais. Cette mémoire sert à stocker des informations permanentes, notamment sur le système d'exploitation de votre ordinateur, de votre BIOS, bref, des informations qui restent écrites et effaçable par réécriture. Pour réécrire la mémoire, il y a différents moyens - à titre informatif - comme exposer la mémoire aux rayons UV, par exemple (ouais, j'ai étudié ça en électronique, c'était marrant). Vous comprenez, par ailleurs, pourquoi on parle de CD-ROM. Parce qu'une fois gravés, on ne peut pas modifier les données, à condition que le CD soit "RW", mais là, il faut réécrire par dessus. ;
- Disque dur : Grosse capacité de mémoire. Permet de stocker des fichiers à long terme. La mémoire des disques durs actuels va jusqu'à se mesurer en Terra-Octets, soit 1000 Go. Cette mémoire est lente, si on compare la rapidité d'accès de la RAM ;
- Les registres : Certes, pour les non-connaisseurs, je vous prends en traitre. Voyons plus bas ce que sont les registres !

Les registres

Les registres sont de petites mémoires dont la taille dépend de notre architecture logicielle. Je m'explique : vous avez un système d'exploitation 32 bits, cela signifie donc que la taille de vos registres fait... 32 bits ! C'est très petit, me direz-vous, et "quel intérêt ?", vous me demandez donc. Patience, j'y viens.

Les registres sont de mémoires minuscules et très rapides. Elles sont, pour certaines, internes au processeur, et ce dernier s'en sert constamment pour faire les opérations de bases, que ça soit logiques - AND, OR, XOR... - et arithmétiques - addition, soustraction, multiplication, division, ... et donc pour traiter l'info. Il faut le rappeler : un ordinateur, c'est con ; on va pas lui demander en langage machine "Affiche-moi coucou à l'écran" en une seule instruction. Il y a plusieurs milliers d'instructions assembleur derrière. Enfin, je vous rassure, on n'ira pas écrire ces milliers d'instructions pour faire un bête "Hello world". C'est juste que c'est pas comme le C, où vous n'avez qu'à faire ça :
Code : c

```c
#include <stdio.h>
int main() {
        printf("Hello, world!\n");
        return 0;
}
```

Mais enfin, je m'écarte du sujet, là. Voyons les quatre registres de base (eh oui, ils ont un nom !) :

- Registre accumulateur : on s'en sert conventionnellement pour stocker des valeurs de retour de fonctions. On le note "eax" lorsqu'il s'agit de manipuler toute sa taille
- Registre de base : conventionnellement... je sais pas à quoi ça peut servir. Un registre de base, quoi. Ca va pas vous empêcher de l'utiliser. On le note "ebx"
- Registre de compteur : conventionnellement, ça sert pour faire des boucles. Noté "ecx"
- Registre de données : "edx". On s'en sert - probablement - pour manipuler des... données !

Chaque registre a une taille de 32 bits, mais il est possible de les manipuler par plus petite taille. Prenons par exemple eax. On peut manipuler sa partie basse sur 16 bits, à savoir "ax". Cette même partie, ax, contient deux sous-parties. La partie haute est notée ah, la partie basse est notée al.

Un schéma pour résumer le tout :

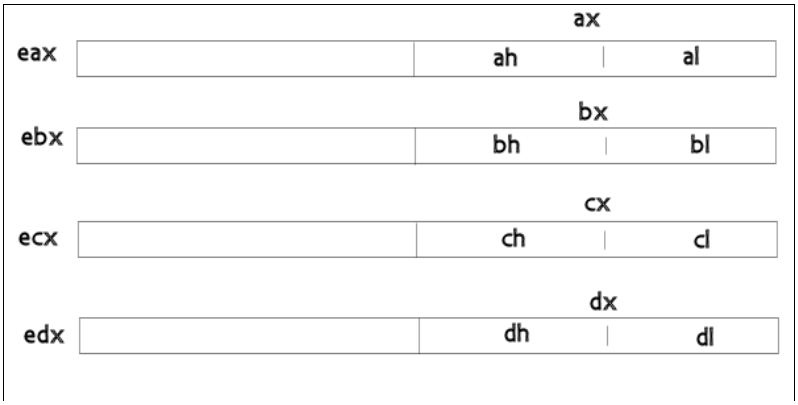


Schéma de la structure des registres de base

Rassurez-vous, ça n'est pas bien grave si vous ne retenez pas tout ce remue-ménage. Tout ce qu'il faut savoir, c'est que les registres peuvent êtres manipulés par tranche de 32, 16 et 8 bits.

Notez qu'il existe d'autres registres que nous verrons dans la suite du tutoriel ; ça serait dommage de vous barber dès le début avec ça. Voyons maintenant la notion d'adresses :

Les adresses

Il faut impérativement comprendre cette notion pour continuer. En assembleur, quand on programme, on adressera nos données et nos sections de code par des labels, ou étiquettes. Sauf qu'une fois le code compilé, le tout est accessible par des adresses mémoires. Ces adresses sont codées sur..... 32 bits ! (ce qui explique pourquoi on parle d'*OS 32 bits*).

On représente souvent ses adresses en hexadécimal. Exemple : **0x7f3d215a**, le 0x devant pour préciser que ce qui suit est de l'hexadécimal. La RAM contient des adresses mémoires, qui contiennent elle-même des données. Si vous avez fait du C et que vous êtes familiarisé avec la notion de "pointeur", vous devriez savoir de quoi je parle.

Et bien, sachez qu'en plus des valeurs de variables, les intructions assembleur sont aussi adressées. Une fois notre programme chargé en mémoire, les octets qui disent au processeur de mettre à zéro un registre - par exemple, hein - se trouvent à une adresse mémoire. Mais enfin, nous utiliserons OllyDbg - outil de debuging puissant - pour mieux comprendre comment ça fonctionne.

Ce qu'il nous faudra étudier, maintenant, c'est le concept de pile !

La pile

Les gens l'appellent souvent "stack". La pile est une zone de mémoire située dans la mémoire accessible en écriture allouée par le programme. Le principe de fonctionnement se résume par ce sigle : *LIFO* => Last In, First Out, ou "Dernier entré, premier sorti".

Prenez par exemple une pile d'assiettes :

```
|_____________________|
```

```
|_____________________|
|_____________________|
|_____________________|
```

On a quatre assiettes empilées. Si on veut retirer la deuxième assiette :

```
|_____________________|
|_____________________|
|_____________________|
|_____________________|
```

En partant du haut, il va falloir dépiler la première assiette. On se retrouve donc avec l'assiette rouge au sommet :

```
|_____________________|
|_____________________|
|_____________________|
```

Pour enfin la dépiler. Elle a disparu :

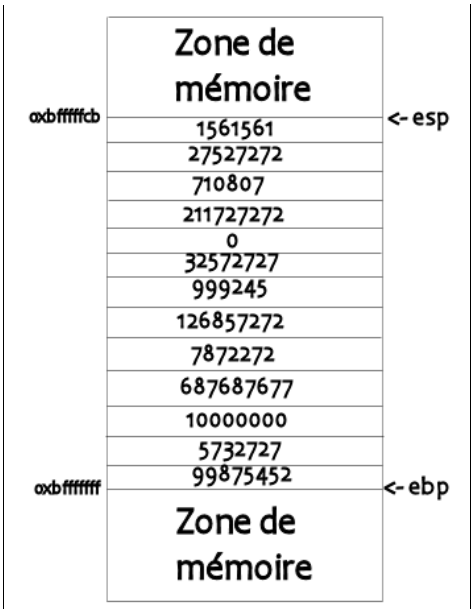
```
|_____________________|
|_____________________|
```

Je vais prendre un exemple qui se rapproche plus de la situation réelle. La pile est une zone de mémoire, comme je vous l'ai dit. Elle a donc ses adresses mémoires. Elle est délimitée par deux registres qui font office de pointeurs :

- esp : "Extended Stack Pointer", ou "pointeur étendu de pile". Ce registre contient et contiendra toujours, en tant que valeur, l'adresse mémoire qui pointe sur le sommet de la pile. Par exemple, si esp contient 0x00405632, alors le sommet de la pile pointera sur l'adresse mémoire 0x00405632, qui contiendra une valeur spécifique ;
- ebp : "Extended Base Pointer", ou "Pointeur étendu de base". A ne pas confondre avec ebx. Le registre ebp pointe sur le bas de la pile.

Enfin, quelque chose à savoir : esp sera toujours plus petit qu'ebp. Plus l'on va vers le haut de la pile, plus les adresses seront basses. A l'inverse, plus l'on ira bas dans la pile, plus les adresses seront hautes.

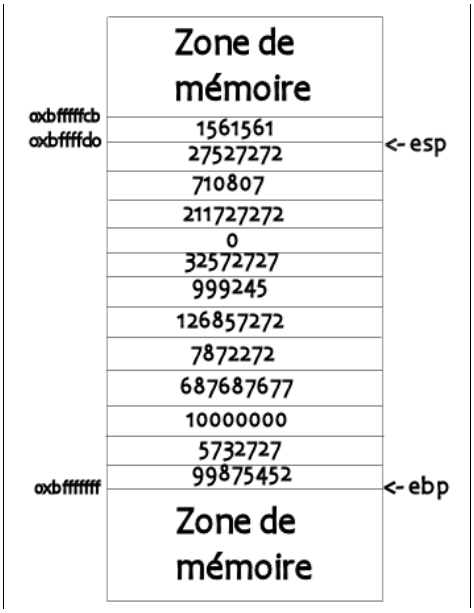
Allez, encore un schéma qui vaudra mieux qu'un long discours :



La pile au beau milieu de la mémoire

Notez que j'ai volontairement mis des valeurs aléatoire dans la pile. C'est pour les représenter au beau milieu de notre zone de mémoire. On voit que le registre esp pointe sur le sommet de la pile, qui est adressée à 0xbfffffcb et la valeur à ce sommet est 1561561 (adresse que j'ai mise au pif). Cette adresse est basse, comparée à 0xbfffffff qui est pointée par ebp.

On remarque que la valeur 1561561 est au sommet de la pile. Si on la dépile, au final, on aura :



La pile au beau milieu de la mémoire, après dépilage de 1561561

Vous l'aurez compris, le registre esp s'est mis à pointer sur la valeur du dessous : 27527272. Notez qu'on peut aussi modifier une valeur au milieu de la pile - qui n'est donc pas nécessairement au sommet - mais on ne pourra pas la dépiler comme ça.

> Quel est l'intérêt d'utiliser la pile ?

La pile est utilisée pour beaucoup de choses. Notamment pour sauvegarder la valeur des registres lorsqu'on appelle une routine (ou fonction) en assembleur. On se sert également de la pile pour empiler nos arguments avant d'appeler une fonction qui va elle-même les dépiler.

Je pense avoir dit le nécessaire pour vous faire *enfin* programmer !

All you need is...

Nous allons enfin passer à la pratique. Téléchargez Masm32 à cette adresse : http://www.masm32.com/masmdl.htm. Installez-le ensuite. Il se trouvera normalement dans **C:\masm32**. Je vous demanderai de créer un dossier nommé "asm_sources" également à la racine du site. Ça facilitera votre organisation. C'est dans ce dossier que nous ferons des sous-dossiers pour chaque programme compilé. Une fois le dossier asm_sources créé, allez dedans, et créez un nouveau dossier nommé hello. Dedans, créez le fichier **hello.asm**, et inserez-y le code suivant :
Code : asm

```
.386
.model flat, stdcall
option casemap :none

include \masm32\include\msvcrt.inc
include \masm32\include\kernel32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\msvcrt.lib

.data
    Phrase db "Hello !",10
    Taille dd 8

.code
start:
    mov eax, Taille
    push eax
    push offset Phrase
    push 1
    call crt__write

    push 0
    call ExitProcess
end start
```

Tout ce code permet de faire :

```
Hello !
```

Pour le compiler, faites un fichier "make.bat" dans le répertoire du fichier, et mettez-y les instructions suivantes :
Code : dos

```
@echo off
```

```
\masm32\bin\ml /c /Zd /coff hello.asm
\masm32\bin\Link /SUBSYSTEM:CONSOLE hello.obj
pause
```

Il vous suffira simplement de double-cliquer sur "make.bat" pour voir, normalement, apparaître :

```
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: hello.asm
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Appuyez sur une touche pour continuer...
```

La ligne 2 du fichier bat réalise la transformation du code source en code objet ; du code machine directement compréhensible par le processeur. Enfin, la troisième ligne réalise l'édition de liens grâce au fichier objet **hello.obj** qui a été généré lors de l'instruction en ligne 2. Au final, vous avez un fichier exécutable pondu dans le même répertoire que vos autres fichiers. Pour le lancer :

```
Microsoft Windows XP [version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Geoffrey>cd ../../asm_sources/hello

C:\asm_sources\hello>hello
Hello !

C:\asm_sources\hello>
```

Sans trop de difficulté, on a fait notre premier programme en assembleur. Voyons maintenant pas à pas la signification des lignes du code source (quand même !)

Les directives

En assembleur masm32, on utilise des directives pour indiquer certaines options au compilateur. Ici, nous avons :
Code : asm

```
.386
.model flat, stdcall
option casemap :none

include \masm32\include\msvcrt.inc
include \masm32\include\kernel32.inc

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\msvcrt.lib
```

Première ligne : **.386**. Cette directive indique que l'on va utiliser le jeu d'instructions d'un processeur x86 32 bits. Ne m'en demandez pas plus à ce sujet, je sais juste qu'il est important de mettre cette ligne.
Seconde ligne : **.model**. Cette directive spécifie le modèle de mémoire du programme. "flat" signifie qu'on va travailler avec des adresses mémoires "lointaines" ou "prêtes". "stdcall" indique que, lors d'appels aux fonctions, les arguments devront être empilés de droite à gauche.

> Qu'est-ce que ça signifie ?

Et bien, reprenons le code assembleur, où je fais appel à la fonction write. Sa signature en C est la suivante :
Code : C

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

La fonction write prend trois arguments :

- fildes : "FILe DEScriptor", ou descripteur de fichier. La fonction write peut, en effet, écrire dans des fichiers, ou dans les flux standards tel que stdout (la sortie, l'écran) ;
- buf : "BUFfer", ou "tampon". Il s'agit d'un pointeur vers les données à écrire ; il s'agit généralement d'une chaîne de caractères.
- nbyte : "Number of BYTE", ou "nombre d'octets". Indique le nombre d'octets à écrire.

Il y aura trois arguments à empiler, et comme on fonctionne de "droite à gauche", on empilera d'abord nbyte, puis *buf, et enfin fdes. Enfin, continuons après cette brève parenthèse.

option casemap :none indique au compilateur que les noms d'étiquettes seront sensibles à la casse. C'est-à-dire que l'étiquette hello - qui pointera sur un bloc de données - et Hello - qui pointera sur un autre bloc de données - seront différentes. C'est comme les noms de fonction en C : on vous a normalement appris que Main() et main(), c'est pas pareil !

Passons aux include et includelib (je récapitule) :
Code : asm

```
include \masm32\include\msvcrt.inc
include \masm32\include\kernel32.inc
```

```
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\msvcrt.lib
```

la directive "include" en masm32 correspond à un "#include" en C. Les fichiers .inc contiennent les signatures des fonctions à utiliser. Eh oui, sous Windows, on doit se servir des fonctions "mappées" dans les fichiers DLL. Les fichiers inc contiennent les définitions des fonctions.

Enfin, la directive includelib indique au compilateur qu'on veut se servir des fichiers lib qui, eux, contiennent carrément le code des fonctions.

Vous avez vu que je me sers de kernel32.lib/inc et msvcrt.lib/inc. Kernel32 contient la fonction ExitProcess(), qui consiste à quitter proprement le programme pour rendre la main au système d'exploitation (il y a du code qui s'en charge, et dont on ne doit pas se soucier) et msvcrt contient le code de la fonction write(), que je vous ai décrite plus haut.

Les sections

Dans notre code, nous avons déclaré deux sections. Voyons la première :
Code : asm

```
.data
    Phrase db "Hello !",10
    Taille dd 8
```

La directive ".data" indique le début de notre section data, qui contient des données initialisées. C'est là ça devient intéressant. En fait, cette section sert à déclarer des données dont nous nous servirons dans le code.

Code : asm

```
Phrase db "Hello !",10
```

Cette instruction signifie que je déclare une chaîne de caractères qui contient le mot "Hello !", suivi de l'octet 10, qui correspond à un retour à la ligne. **Phrase** désigne un label afin de pointer aisément sur notre chaîne de caractères. enfin, l'inscruction db signifie "declare byte", afin de préciser que les unités ont pour taille 1 octet (8 bits).

Code : asm

```
Taille dd 8
```

Même principe, sauf que dd signifie "declare double". Un "double", ici, correspond à quatre octets (32 bits). Le label se nomme Taille, et la donnée est de '8' ; étant codée sur quatre octets, on aura donc, en hexadécimal, 0x00000008. Cette donnée correspond en fait à la taille de la chaîne de caractère "Hello !",10. Voici, après compilation, l'organisation de la section data :
Code :

```
+---+---+---+---+---+---+---+---+----+----+----+----+----+-------------
|'H'|'e'|'l'|'l'|'o'|' '|'!'|0x0A|0x08|0x00|0x00|0x00| ..........
+---+---+---+---+---+---+---+---+----+----+----+----+----+-------------
  ^                           ^
  |                           |
  |                           |
Phrase                      Taille
```

Vous verrez que j'ai volontairement représenté certains octets sous leur forme hexadécimale. 0x0A correspond, en décimal, à la valeur 10, donc à notre retour à la ligne. Enfin, vous verrez que Taille pointe sur 0x00000008, qui est une valeur écrite à l'envers en mémoire. Elle est écrite à l'envers car on ne veut pas lire "quatre octets à la suite", "mais un entier à la fois". Dès lors, pour représenter, par exemple, le nombre 0x65231489 en mémoire, ça sera :
Code :

```
+----+----+----+----+----------
|0x89|0x14|0x23|0x65|..........
+----+----+----+----+----------
```

Mais revenons-en à notre schéma plus au-dessus, où j'ai représenté les données du programme. J'ai montré que Phrase et Taille pointaient sur ces données. Ces labels correspondront en fait, après la compilations, à des adresses mémoires.

Si Phrase correspond à l'adresse mémoire 0x00401000, alors Taille correspondra à l'adresse mémoire 0x40100008, car Phrase pointe sur huit octets. Enfin, vous verrez que si vous vous mettez à pointer sur Phrase+8, vous retomberez sur Taille.

> Ce qu'il faut savoir, en résumé, c'est que le fait de déclarer des données de façon séquentielle reviens à les écrire de la même façon dans la section data de notre exécutable.

Étudions à présent la section .code, qui correspond à la section exécutable de notre programme. Cela signifie que les octets - qui seront en fait représentés par des instructions en assembleur - écrits dans cette section seront lus par le processeur et exécutés.
Code : asm

```
.code
start:
```

La directive "start:", elle, indique le point d'entrée du programme. C'est comme la fonction main() en C. C'est après cette directive que les choses deviennent intéressantes, mais intéressons-nous à l'algorithme logique du programme.

Comme nous l'avons vu, le programme s'est contenté d'afficher bêtement "Hello !", et de rendre la main au système d'exploitation. Il a donc fallu appeler **write()**,

et **ExitProcess()**. En premier lieu, il faut donc empiler les arguments de **write()**, appeler la fonction, puis empiler les arguments de **ExitProcess()** et enfin appeler cette fonction.

Code : asm
```
    mov eax, Taille
    push eax
    push offset Phrase
    push 1
    call crt__write

    push 0
    call ExitProcess
```

La première ligne indique au processeur de mettre dans le registre eax la valeur pointée par le label Taille, à savoir 0x00000008. Le registre eax, après cette instruction, contiendra - sur 32 bits, soit 4 octets - 0x00000008. En assembleur intel, on a la synthaxe suivante :

mov destination, source

Et "destination", par l'intruction "mov", recevra en nouvelle valeur "source" et aura son ancienne valeur écrasée.

Code : asm
```
push eax
```

L'instruction push, en assembleur, consiste à mettre une donnée au sommet de la pile (à l'empiler, quoi... Je me complique la vie !). Ici, on a empiler la valeur d'eax, à savoir, 8. On a empilé notre troisième argument de write() ; vous vous souvenez quand je vous ai dit qu'il fallait empiler les arguments de droite à gauche, donc dans l'ordre inverse ?

Code : asm
```
push offset Phrase
```

On va encore empiler quelque chose, sauf qu'ici, il ne s'agit pas d'empiler une chaîne entière, mais un pointeur vers celle-ci. L'instruction offset indique que l'on va empiler l'adresse mémoire de notre chaîne. En C, on passe des pointeurs en paramètres. Et que contiennent les pointeurs ? Des adresses mémoires. C'est la même, vous dis-je !

Code : asm
```
push 1
```

Enfin, on empile la valeur 1. Cette valeur est représentée par défaut sur 32 bits (4 octets) puisque nous sommes sur un système d'exploitation 32 bits. Le nombre 1 correspond à la sortie standard, soit l'écran. Cela indique que l'on va écrire nos octets à l'écran. Et enfin, on appelle la fonction :

Code : asm
```
call crt__write
```

> Pourquoi "crt__write" et pas "write" ?

Je ne sais pas ce que signifie crt (à vrai dire, j'ai pas vraiment cherché ; c'est pas bien, je sais) mais j'ai mené une petite recherche dans les fichiers .inc, et j'ai vu que la fonction write() était déclarée en "crt__write".

L'instruction call correspond à un saut très très lointain sur une autre adresse mémoire. Lors de la compilation, le compilateur se débrouillera pour remplacer crt__write par une adresse mémoire sur laquelle sauter. Oui, les fonctions qui se situent dans les DLL se situent à des adresses, **toute donnée est adressée**. Le call se charge aussi d'empiler l'adresse de l'instruction qui se situe juste derrière ; cela permettra au processeur de revenir où il en était après l'exécution de la fonction.

On vient donc d'exécuter cet équivalent en C :
Code : C
```
write(1,"Hello !\n",8);
```

Il nous faut enfin quitter le programme proprement. En C, cela reviendra à utiliser :
Code : C
```
ExitProcess(0);
```

la fonction prend un unique argument ; sa valeur, 0, indique que le déroulement du programme a été normal, et qu'on le quitte proprement. On peut utiliser d'autres codes en fonction d'éventuelles erreurs produites pendant le déroulement du programme, que ça soit 1, -1... 1 reste la valeur conventionnelle aussi bien sous Windows que Linux (je connais pas Mac, par contre. J'aime pas les pommes).

De ce fait :
Code : asm
```
    push 0
    call ExitProcess
```

Consiste à mettre la valeur 0x00000000 sur la pile et d'appeler la fonction ExitProcess(), qui va terminer proprement le processus en rendant la main au système

d'exploitation (qui, lui, aura récupéré le 0 passé à la fonction. On peut traiter ça avec des algos, mais nous n'expliciterons pas la chose).

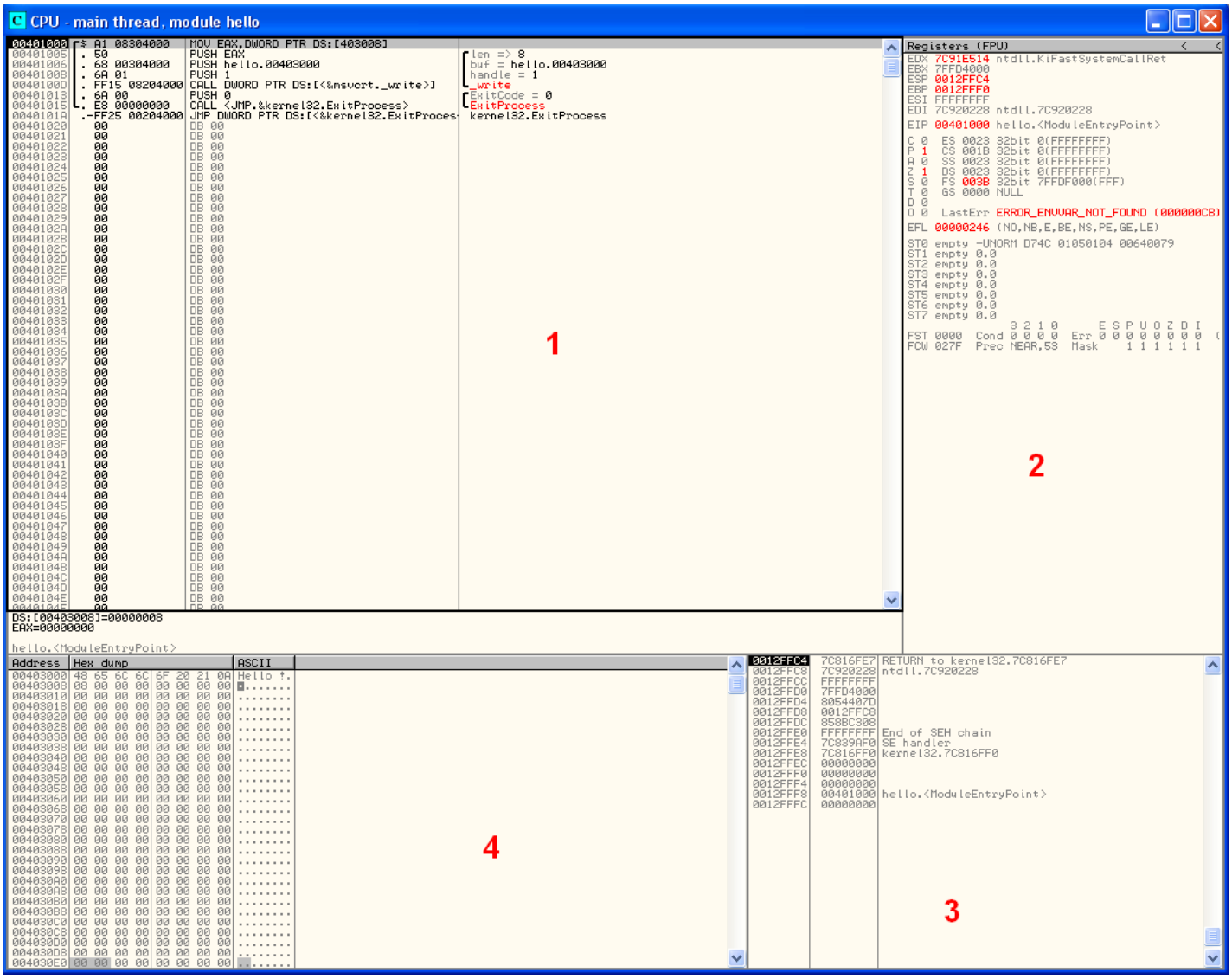
Et enfin, on indique la fin de la routine principale :
Code : asm

```
end start
```

Dites-moi, sérieusement... Il y avait quoi de dur ?

Désassembler le programme

Je vous avais promis une session de debug avec OllyDbg. Commencez par le télécharger à cette adresse : http://www.ollydbg.de/odbg110.zip. Décompressez-le dans un dossier quelconque, que ça soit sur le Bureau ou dans "Mes Documents", ça n'a pas d'importances. Ouvrez la bête, et faites File > Open, enfin, cherchez votre fameux hello.exe que l'on a compilé ensemble. Voici une sous-fenêtre que vous devriez normalement obtenir :



Sous-fenêtre d'Olly Dbg

Pour ceux qui n'ont jamais utilisé cet outil, ne vous en faites pas, j'ai numéroté les parties pour vous expliquer en détail.

1. La section code de notre programme. Il y a, tout en haut, la première instruction qui sera exécutée. La "scrollbar" s'est, en effet, positionnée au point d'entrée. Dans cette fenêtre, on a quatre colonnes qui représente respectivement : l'adresse mémoire de l'instruction, l'équivalent hexadécimal de l'instruction (son équivalent en langage machine pur et dur, quoi), l'instruction assembleur, et, enfin, des informations supplémentaires sur l'instruction.
2. Cette frame donne l'état des registres. Vous remarquez qu'il y a EAX, EBX, ECX, EDX, EBP, ESP et d'autres que je n'ai pas explicités. C'est le moment, si on veut. ESI (Extended Source Index) et EDI (Extended Data Index) sont des registres qui sont manipulables dans le code. A quoi ils servent... Je ne saurais pas vous dire. EIP, lui, signifie Extended Instruction Pointer. Ce registre contient l'adresse de l'instruction suivante. Ici, EIP contient 00401000, ce qui signifie que l'instruction qui va être exécutée sera celle à l'adresse 00401000, soit **MOV EAX,DWORD PTR DS:[403008]**, qui correspond, dans notre code, à "mov eax, Taille".
3. Cette fenêtre montre la pile. En haut de la frame, on a le sommet de la pile. Pour être fixé, regardez la valeur d'ESP - 0012ffc4 - et l'adresse en haut de la frame : ce sont les mêmes.
4. La section data. On remarque, par ailleurs, que nos données sont bien écrites les unes à la suite des autres, comme je vous l'avais schématisé.

Pour exécuter une instruction à la fois, appuyez sur F7. Hop, vous venez d'exécuter **MOV EAX,DWORD PTR DS:[403008]**. regardez à présent la valeur d'EAX : le registre contient 0x00000008. Ensuite, tracez jusqu'à **CALL DWORD PTR DS:[402008]**. Là, vous avez deux options :

- Soit vous appuyez sur F7, et vous tracez pas à pas le code assembleur de la fonction write(), qui correspond à un gros bordel que moi-même je ne connais pas.
- Sois vous appuyez sur F8, et vous tracez le CALL sans entrer dedans. C'est comme si vous traciez une instruction normale.

Que dire d'autre à propos d'Ollydbg ? Il est pratique si vous avez fait un programme et qu'il y a un bug que vous avez du mal à résoudre. Ca vous permet de debugger facilement vos futurs programmes écrits en MASM32, en tout cas.

Conclusion

J'ai explicité pas mal de trucs dans cette introduction, tout comme j'en ai oubliés certains. Il existe pas mal de tutoriels qui vous explique bien plus de fondamental que ça, à savoir, les instructions de saut conditionnels/inconditionnels (pour implémenter les boucles ou les conditions, par exemple), le pop qui permet de dépiler une valeur (on l'a pas vu, mais c'est pas bien grave), les registres d'état, etc. Des liens pour vous guider :

http://win32assembly.online.fr/. Bien que le site soit en anglais, c'est pour moi une mine d'or pour les curieux.
http://jeanfrancoisdelnero.free.fr/prog.htm Contient quelques progs en assembleur, et une petite section lien en bas de page.

Ce que j'ai à dire pour ma part : l'assembleur, c'est pas méchant. C'est sympa si on veut comprendre comment fonctionne la programmation au plus bas niveau. Sous windows, on fait des appels de fonction. Sous linux, le noyau fournit directement des appels noyau grâce à des interruptions, ce qui rend la programmation encore plus simple puisqu'il ne faut pas nécessairement faire de CALL. Enfin, ça, c'est une autre affaire.

Enfin, je tiens à dire que je suis assez amateur, et que ce tutoriel n'est pas exhaustif et contient de très nombreux défauts. J'espère juste que ceux qui l'ont lu ne s'en sortiront pas sans n'avoir rien appris ! Pour toute question : geo [point] 669 [at] gmail [point] com.
J'oubliais : énorme merci à Overcl0k, sans qui la rédaction de ce tutoriel n'aurait probablement pas eu lieu.

Geo