Introduction à l'assembleur x86 par rétro-ingénierie

Can Pouliquen

Décembre 2020

1 Introduction

L'assembleur est un langage de programmation très bas niveau, spécifique à une architecture de processeur, qui est ensuite traduit en langage machine (binaire) par un assembleur (l'assembleur traduit le langage assembleur en binaire, bonjour la confusion).
Les avantages de comprendre la programmation en assembleur sont multiples :

- Acquérir une compréhension très bas niveau des intéractions d'un programme avec l'OS, le processeur, la mémoire.
- Comprendre le fonctionnement interne d'un processeur.
- La rétro-ingénierie (à des fins légales, bien entendu).
- C'est stylé.

Pour toutes ces raisons, il peut être très intéressant, sans nécessairement être un développeur assembleur aguerri, d'en comprendre les principes fondamentaux.

Un ordinateur peut grossièrement être résumé en un processeur et de la mémoire. La mémoire contient un programme à être exécuté, ce qui est fait par le processseur.
Dans le but de comprendre l'x86, qui est l'architecture d'Intel, leader incontestable des processeurs de bureau, nous allons ici résoudre un problème de rétro-ingénierie (tiré d'un CTF[1] pour débutants). Les explications se feront au fur et à mesure de l'avancement de l'exercice et UNIQUEMENT sur les points nécessaires à sa résolution. J'inclurai des liens pour ceux qui voudront en savoir plus. La lecture de ce document seul ne permettra donc évidemment pas une connaissance exhaustive de l'x86. En annexe, j'ajouterai un autre exercice plus compliqué dont je mettrai la réponse mais pas la correction. Vous pourrez me contacter à son sujet si ça vous intéresse.

2 Prérequis

Représentations binaires et héxadécimales.
Notions de base du fonctionnement d'un CPU.
Notions de base de programmation en C.
Globalement, si vous avez suivi les tutos de Latorre, c'est bon.

[1] Capture The Flag. Compétition et/ou exercice de piratage informatique.

3 Le problème

Soit une fonction *int asm(int)*. Sa compilation produit le code suivant :
asm:

```
<+0>:   push    ebp
<+1>:   mov     ebp, esp
<+3>:   cmp     DWORD PTR [ebp+0x8], 0x3a2
<+10>:  jg      0x512 <asm1+37>
<+12>:  cmp     DWORD PTR [ebp+0x8], 0x358
<+19>:  jne     0x50a <asm1+29>
<+21>:  mov     eax, DWORD PTR [ebp+0x8]
<+24>:  add     eax, 0x12
<+27>:  jmp     0x529 <asm1+60>
<+29>:  mov     eax, DWORD PTR [ebp+0x8]
<+32>:  sub     eax, 0x12
<+35>:  jmp     0x529 <asm1+60>
<+37>:  cmp     DWORD PTR [ebp+0x8], 0x6fa
<+44>:  jne     0x523 <asm1+54>
<+46>:  mov     eax, DWORD PTR [ebp+0x8]
<+49>:  sub     eax, 0x12
<+52>:  jmp     0x529 <asm1+60>
<+54>:  mov     eax, DWORD PTR [ebp+0x8]
<+57>:  add     eax, 0x12
<+60>:  pop     ebp
<+61>:  ret
```

Que renvoie asm(0x6FA)?

4 La pile

La pile est un espace mémoire alloué qui suit une structure LIFO. Le point peut-être le plus important à retenir est que la pile empile ses éléments "vers le bas". En d'autres termes, vers les addresses mémoires descendantes, et non ascendantes comme on pourrait s'y attendre.
Lorsqu'un programme fait appel à une fonction, le processeur lui réserve une zone mémoire : la fonction a sa propre pile, rien qu'à elle, où sont chargés ses arguments (d'où l'importance en C de préciser le type des arguments, étant donné qu'il faudra leur allouer de la mémoire en adéquation à leur taille).
Le début de la pile est pointée par le registre EBP (qui contient donc l'adresse la plus élevée de la pile) et sa fin est pointée par le registre ESP (qui contient alors l'adresse la plus faible de la pile).
Pour ceux qui veulent en savoir plus sur le fonctionnement de la pile en mémoire :
https://en.wikibooks.org/wiki/X86_Disassembly/The_Stack

5 Les registres

- eax : Extended Accumulator Register. Principalement utilisée pour contenir la valeur de retour d'une fonction.
- ebp : Extended Base Pointer. Contient un pointeur vers l'adresse de la base de la pile.
- esp : Extended Stack Pointer. Contient un pointeur vers l'adresse de la tête de la pile.

Pour ceux qui veulent en savoir plus sur les registres de l'x86 :
https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture
https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

6 Le jeu d'instruction

- PUSH : Ajouter un élément à la pile.
- MOV : Effectuer un déplacement de données.
- CMP : Comparer deux opérandes.
- JG : Instruction conditionnelle >.
- JNE : Instruction conditionnelle !=.
- ADD : Opération arithmétique d'addition
- JMP : Instruction non-conditionnelle de saut (jump).
- SUB : Opération arithmétique de soustraction.
- POP : Enlever un élément de la pile (structure LIFO).
- RET : Retourner la fonction.

Pour ceux qui veulent en savoir plus sur le jeu d'instruction de l'x86 :
https://en.wikipedia.org/wiki/X86_instruction_listings

7 Méthode de résolution

L'ensemble des éléments fournis plus haut sont suffisants pour résoudre notre problème. Que savons-nous? La fonction a pour argument 0x6FA, sous forme d'entier de 32-bits. La première action accomplie lors de l'appel à la fonction est donc le chargement de l'argument dans la pile. On doit lui allouer 8 octets. L'adresse [ebp+0x8] contiendra donc 0x6FA.

<+0>: push ebp
<+1>: mov ebp, esp
<+3>: cmp DWORD PTR [ebp+0x8], 0x3a2 ; On veut comparer l'argument à 0x3A2
<+10>: jg 0x512 <asm1+37> ; Si 0x3A2 est plus grand, sauter à l'adresse 0x512, donc la ligne 37 de notre fonction. Ce n'est pas le cas ici, donc on passe à l'instruction suivante.
<+12>: cmp DWORD PTR [ebp+0x8], 0x358 ; On désire maintenant comparer l'argument à 0x358
<+19>: jne 0x50a <asm1+29> ; Si les deux entiers sont différents, sauter à la ligne 29.

C'est évidemment le cas ici.

```
<+21>:  mov     eax, DWORD PTR [ebp+0x8]
<+24>:  add     eax, 0x12
<+27>:  jmp     0x529 <asm1+60>
<+29>:  mov     eax, DWORD PTR [ebp+0x8] ; On déplace l'argument dans le registre EAX
<+32>:  sub     eax, 0x12 ; On soustrait 0x12 à EAX : 0x6FA-0x12 = 0x6E8
<+35>:  jmp     0x529 <asm1+60> ; On saute à la ligne 60
<+37>:  cmp     DWORD PTR [ebp+0x8], 0x6fa
<+44>:  jne     0x523 <asm1+54>
<+46>:  mov     eax, DWORD PTR [ebp+0x8]
<+49>:  sub     eax, 0x12
<+52>:  jmp     0x529 <asm1+60>
<+54>:  mov     eax, DWORD PTR [ebp+0x8]
<+57>:  add     eax, 0x12
<+60>:  pop     ebp ; On vide la pile
<+61>:  ret ; On retourne EAX
```

La fonction renvoie donc 0x6E8 !

Une traduction de l'x86 en C donnerait donc une fonction qui aurait une tête similaire à la suivante :

```
int asm(int arg)
{
    if(arg > 930)
    {
        if(arg != 856)
        {
            arg -= 18;
            return arg;
        }
        else
        {
            arg += 18;
            return arg;
        }
    }
    else if(arg != 1786)
    {
        arg += 18;
        return arg;
    }
    else
    {
        arg -= 18;
        return arg;
    }
}
```

8 Problème 2

Indication : faites des recherches sur l'endianisme et les registres.
asm3:

```
<+0>   push   ebp
<+1>   mov    ebp,esp
<+3>   xor    eax,eax
<+5>   mov    ah,BYTE PTR [ebp+0x9]
<+8>   shl    ax,0x10
<+12>  sub    al,BYTE PTR [ebp+0xe]
<+15>  add    ah,BYTE PTR [ebp+0xf]
<+18>  xor    ax,WORD PTR [ebp+0x12]
<+22>  nop
<+23>  pop    ebp
<+24>  ret
```

Fonction : int asm(int,int,int)
Appel : asm(0xd2c26416, 0xe6cf51f0, 0xe54409d5)
Réponse : 0x375