STACK-OVERFLOWS

BAUCHIERO Cédric & DUCHESNE Sylvain

4 Décembre 2007

Table des matières

1 Introducion

Ces dernières années, la très grande majorité des failles découvertes et exploitées provient de Buffer-Overflows soit un débordement de tampon dans la pile d'exécution. Ce dernier résulte d'erreur de programmation très fréquentes (dues à l'augmentation de la longueur des programmes), ce qui produit des failles de sécurité dans lesquelles les pirates s'engouffrent pour prendre le contrôle du système.

Ces problèmes n'existent pas dans les langages tels que Pascal, Java, C#... on ne les rencontre que dans les langages « bas niveau » tel que C ou C++, car ceux-ci ne font pas de vérifications ce qui par ailleurs contribue à leur rapidité. Et ces langages sont très utilisés.

A travers cette synthèse, nous allons analyser dans un premier temps le fonctionnement de la Stack dans son état normal de fonctionnement puis nous verrons à travers de multiples exemples comment sont provoquer diverses Buffer-Overflows et ce que l'on peut mettre en oeuvre pour prévenir ce genre d'attaques.

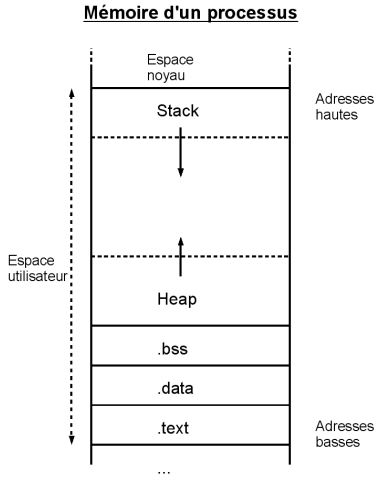
2 Fonctionnement de la Stack

2.1 Espace mémoire d'un processus

Chaque processus dispose de son propre espace de mémoire pour gérer ses variables que l'on peut diviser en sections dont les plus importantes pour ce sujet sont :

- .text : stocke les instructions, le code du programme.
- .data : stocke les données globales initialisées.
- .bss : stocke les données globales non initialisées.
- Heap : stocke les variables allouées dynamiquement ex : malloc()...).
- Stack : stocke les données temporaires, locales.

La mémoire est adressée par mots et couvre l'espace d'adresse de 0x00000000 à 0xbfffffff pour l'utilisateur et de 0xbfffffff à 0xffffffff pour le noyau (remarque : on travail en hexadécimal et non en décimal donc 0..15 nous donnera 0..9,a,b,c,d,e,f).



2.2 Principe de la Stack

Nous allons nous intéresser à la section de la Stack (Pile) et comprendre son fonctionnement avant de nous intéresser au mécanisme de Stack Overflows (Dépassement de tampon) proprement dit. La pile ne manipule que des mots donc chaque variable allouée occupera un certain nombre de mots. Dans notre cas, avec un processeur 32bits(4octets) AT&T, un mot correspondra donc à 4 octets. L'espace mémoire de la pile est donc de taille multiples de 4 (ex : int prend 4octets donc 1 mot, char buffer[6] prend 6octets donc 2 mots(1mot étant insuffisant) , ...).

La pile est une zone mémoire basée sur le principe «dernier arrivé, premier sortie» ou LIFO «Last In, First Out». Il s'agit d'un moyen d'accéder à des données en les empilant, telle une pile de livres, puis en les dépilant pour les utiliser. Nous ne pouvons dépiler que le dernier élément empilé, il est donc nécessaire de dépiler les valeurs stockées au sommet pour pouvoir accéder aux valeurs situées à la base de la pile.

La zone mémoire correspondante à la pile est déterminée par deux pointeurs (Stack pointers) : un pointeur de haut de pile (%esp : extended stack pointer) et un pointeur vers la base de la pile (%ebp : extended base pointer). %esp est donc modifié à chaque appel de Push ou Pop, en effet à chaque appel de push, %esp diminue d'un mot et on stocke la valeur donnée en argument à cette adresse et à chaque appel de pop on enlève la valeur sur laquelle %esp pointe que l'on met dans le registre donné en argument puis on incrémente %esp d'un mot (car l'augmentation de la pile se fait des adresses mémoires les plus hautes vers les adresses les plus basses comme on a pu le voir dans le schéma précédent). Il existe aussi un pointeur %eip qui pointe toujours sur la prochaine instruction à exécuter.

Les principales actions que l'on peut réaliser sur les piles sont donc :
- Empiler : qui ajoute un élément au dessus de la pile. («Push»)
- Dépiler : qui enlève l'élément du dessus de la pile pour le stocker dans un registre. («Pop»)
- Savoir si la pile est vide ou pas.
- Connaître le nombre d'éléments de la pile.

Les registres : Ce sont des emplacements mémoires d'un mot situé à l'intérieur du processeur. Toute nouvelle valeur entrée dans un registre supprime automatiquement l'ancienne. Ils permettent une communication étroite entre la mémoire et le processeur. Ils existent 4 catégories de registres, nous n'utiliserons que les registres généraux qui servent à la manipulation des données (%eax, %ebx, %ecx, %edx) ainsi que certains registres d'offset qui indiquent un décalage (%esp, %ebp, %eip).

Voici un exemple de fonctionnement de pile qui permet l'inversion des valeurs des registres eax et edx:

(Base de la pile)

125	0x110	125	0x110	125	0x110	125	0x110	125	0x110
23	0x10e	23	0x10e	23	0x10e	23	0x10e	23	0x10e
		8	0x10b	8	0x10b	8	0x10b	*S*	0x10b
				112	0x107	*112*	0x107	*112*	0x107

(haut de la pile)

Initialisation	**Pushl %eax**	**Pushl %edx**	**Popl %eax**	**Popl %edx**
%eax=8	%eax=8	%eax=8	%eax=112	%eax=112
%edx=112	%edx=112	%edx=112	%edx=112	%edx=8
%esp=0x10e	%esp=0x10b	%esp=0x107	%esp=0x10b	%esp=0x10e
%ebp=0x110	%ebp=0x110	%ebp=0x110	%ebp=0x110	%ebp=0x110

2.3 Appel de fonction au sein de la Stack

Nous allons prendre un exemple simple pour comprendre à travers de l'assembleur comment marche la pile et les registres lors d'un appel de fonction. Etudions le cas du déroulement d'un appel de fonction normal pour comprendre comment certaines attaques arrivent à détourner se déroulement à leur insu.

```
#include <stdio.h>

 int foo(int a, int b) {
      char buffer1[5]="hello" ;
      int k=7 ;
      return 0 ;
}

int main() {
      int a=3 ;
      foo(a,5) ;
      return 0 ;
}
```

Nous allons désassembler ce programme par l'intermédiaire du débugger gdb, pour comprendre ce qui se passe dans la pile au moment de l'appel à la fonction foo().

```
(gdb) disas main
Dump of assembler code for function main:
0x0804837d <main+0>:    lea    0x4(%esp),%ecx
0x08048381 <main+4>:    and    $0xfffffff0,%esp
0x08048384 <main+7>:    pushl  0xfffffffc(%ecx)
0x08048387 <main+10>:   push   %ebp                          //Prolog main()
0x08048388 <main+11>:   mov    %esp,%ebp
0x0804838a <main+13>:   push   %ecx
0x0804838b <main+14>:   sub    $0x4,%esp
0x0804838e <main+17>:   movl   $0x3,0xfffffff8(%ebp)        //appel foo()
0x08048395 <main+24>:   movl   $0x5,0x4(%esp)
0x0804839d <main+32>:   mov    0xfffffff8(%ebp),%eax
0x080483a0 <main+35>:   mov    %eax,(%esp)
0x080483a3 <main+38>:   call   0x8048344 <foo>
0x080483a8 <main+43>:   mov    $0x0,%eax
0x080483ad <main+48>:   add    $0x8,%esp                     //retour de foo()
0x080483b0 <main+51>:   pop    %ecx
0x080483b1 <main+52>:   pop    %ebp                          //Epilog main()
0x080483b2 <main+53>:   lea    0xfffffffc(%ecx),%esp
0x080483b5 <main+56>:   ret
End of assembler dump.
```

```
(gdb) disas foo
Dump of assembler code for function foo:
0x08048344 <foo+0>:     push   %ebp                   //Prolog foo()
0x08048345 <foo+1>:     mov    %esp,%ebp
0x08048347 <foo+3>:     sub    $0xc,%esp
0x0804834a <foo+6>:     mov    0x8048470,%eax
0x0804834f <foo+11>:    mov    %eax,0xfffffff7(%ebp)
0x08048352 <foo+14>:    movzbl 0x8048474,%eax
0x08048359 <foo+21>:    mov    %al,0xfffffffb(%ebp)
0x0804835c <foo+24>:    movl   $0x7,0xfffffffc(%ebp)
0x08048363 <foo+31>:    mov    $0x0,%eax
0x08048368 <foo+36>:    leave                         //Epilog foo()
0x08048369 <foo+37>:    ret
End of assembler dump.
```

Voici donc les fonctions main() et foo() desassemblées. Nous remarquons qu'à chaque appel de fonction, nous avons le même bloc d'initialisation, appelé Prolog, et de terminaison, appelé Epilog, puis des instructions qui constituent l'appel de la fonction foo() dans la fonction main().

Le Prolog correspond à la sauvegarde de la pile en entrée de fonction pour que lorsque l'on en sort on puisse remettre tout dans l'ordre où on l'avait trouvé. Il réserve aussi un espace mémoire pour le bon fonctionnement de la fonction.

L'appel de fonction met les arguments de la fonction dans la pile et sauvegarde le registre %eip qui permet de revenir au bon endroit après l'exécution de la fonction.

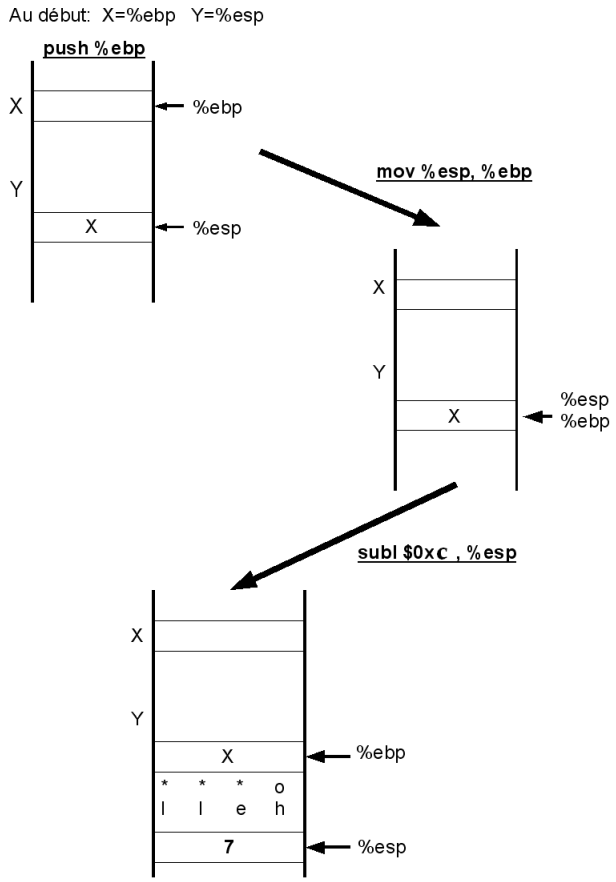
L'Epilog correspond à remettre les éléments de la pile tels qu'on les avait trouvé avant l'appel de fonction.

- Le Prolog :

```
 push %ebp
mov %esp, %ebp
sub $0x10, %esp
```

Le processeur sauvegarde dabord le pointeur de base de pile(%ebp). Ainsi quand nous sortirons de la fonction le pointeur de base de pile pourra être remis à sa valeur initiale. Puis il copie le contenu de %esp dans %ebp ce qui correspond à déplacer le pointeur de la base de la pile au niveau du pointeur de haut de pile. Puis un nouveau pointeur de haut de pile est crée pour réserver de la place pour les variables locales, nous remarquons que %esp est décrémenté car comme nous l'avons dit la pile augmente vers les adresses basses.

Cela revient donc à créer à l'intérieur de la pile une autre pile temporaire pour la fonction.



- L'appel :

```
 movl $0x3,0xfffffff8(%ebp)
movl $0x5,0x4(%esp)
call 0x8048344 <foo>
```

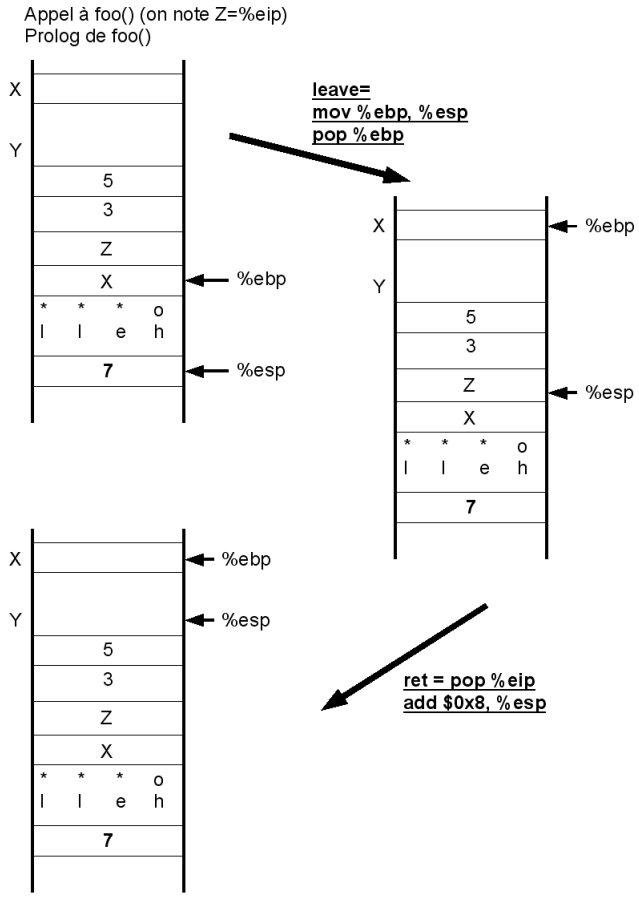
Cette étape correspond à la mise en pile des arguments nécessaires à la fonction ici 3 et 5 ainsi qu'à l'enregistrement du registre %eip dans la pile après les arguments. Cette seconde étape est faite de manière implicite par « call ». Puis il donne la main à la fonction en mettant le registre %eip égal à l'adresse passé en argument soit la première étape du Prolog de foo().

- L'Epilog :

```
 leave ou ( mov %ebp, %esp puis pop %ebp)
ret
```

C'est l'opération inverse de celle du Prolog c'est à dire on déplace le pointeur de haut de pile au niveau du pointeur de base de la pile puis on remet la valeur que l'on avait sauvegardé dans %ebp. Ainsi %ebp pointe au même endroit qu'avant l'appel de fonction et %esp pointe sur la valeur de %eip sauvegardée pendant l'appel de fonction.

L'instruction ret, qui revient à faire un Pop %eip, nous renvoie juste après l'appel à la fonction foo() dans le main() grâce à la valeur de retour stockée dans la pile au moment de l'appel. La pile est presque revenue dans l'état dans lequel elle était avant l'appel de fonction mis à part %esp, qui pointe encore sur les arguments de la fonction. On le replace grâce au add $0x8 , %esp présent dans la fonction main().


Appel à foo() (on note Z=%eip)
Prolog de foo()
X
Y
5
3
Z
X
%ebp
* * * o
l l e h
7
%esp
leave=
mov %ebp, %esp
pop %ebp
X
%ebp
Y
5
3
Z
%esp
X
* * * o
l l e h
7
X
%ebp
Y
%esp
5
3
Z
X
* * * o
l l e h
7
ret = pop %eip
add $0x8, %esp

3 Principales méthodes de Stack Overflows et leurs contre-mesures

L'idée général d'un Stack Overflow est de remplir un buffer avec plus de données qu'il ne peut en contenir (de par sa déclaration) de manière à réécrire une donnée importante. On distingue deux grands types d'attaques selon que cette données est une valeur liée au fonctionnement de la pile (le rip ou le sfp), dans ce cas on parle d'Activation Record Hijacking ou bien un pointeur (de fonction, de données...) dans ce cas on parle de Pointer Subterfuge. Nous allons maintenant détaillés ces deux courants en étudiant quelques unes des méthodes qui leur appartiennent.

3.1 Activation Record Hijacking

Il existe essentiellement trois techniques utilisant ce principe, comme nous allons le voir maintenant.

3.1.1 Réécriture du rip-Shellcode

Principe Comme nous l'avons vu précédemment, le sfp et le rip sont stockés en mémoire au dessus du buffer (qui est une variable locale). Donc en copiant dans ce buffer un nombre de caractères supérieur à sa taille, on peut réécrire ces deux valeurs qui sont essentielles au bon déroulement du programme.

Dans le cas présent, cette réécriture va nous permettre d'exécuter un code que nous auront au préalable copié dans le buffer. Ce code, que l'on appelle Shellcode, à pour but de lancer un shell, avec éventuellement les droits d'administrateur. Ce shellcode est un mini-programme, codé directement en langage machine. En voici un exemple :

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

Pour détourner le flot d'exécution, on va réécrire le rip (l'adresse de l'instruction que le programme doit lire après avoir quitté la fonction courante) : on va le remplacer par l'adresse de début du buffer, contenant le shellcode. Ainsi, à la sortie de la fonction, c'est le shellcode qui sera exécuté.

Exemple `vulnerable.c`

```
-------------------------

void copie (char *args){
         char buffer[200] ;
         strcpy(buffer, args) ;
   }
```

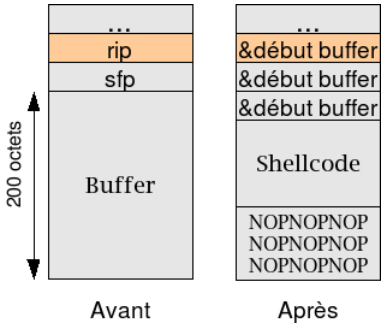
```
int main (int argc, char *argv[]){
      if (argc > 1)
      copie(argv[1]) ;
      return 0 ;
}
```

Ce programme est très simple, il copie l'argument reçu dans le buffer de taille 200. Il utilise pour cela la fonction strcpy.

L'attaque consiste à appeler vulnerable.c avec en argument la concaténation de plusieurs éléments :

- un certain nombre de NOP . Un NOP est une non-instruction, il dit à la machine de ne rien faire et de passer à l'instruction suivante. Nous ne connaissons pas l'adresse exacte de début du buffer, on peut la calculer mais il est plus simple d'utiliser cette suite de non-instructions qui fait que quel que soit le NOP sur lequel pointe l'adresse qui se trouve à la fin du buffer, on est sûr que le shellcode sera exécuté.
- le shellcode qui doit se trouver à peu près au milieu.
- et l'adresse estimée de début du buffer, répétée plusieurs fois pour augmenter les chances de réécrire le rip avec cette adresse.

Les deux schémas suivants représentent l'état de la pile avant et après le Stack Overflow :



Ainsi lorsque l'on quittera la fonction copie(), quand la machine cherchera où elle doit aller lire les instructions suivantes, elle ira lire celles contenues dans le buffer et un shell sera alors ouvert sur la machine hôte.

Limites de la méthode Pour que cette méthode fonctionne, les conditions suivantes doivent être remplies :

- Le programme que l'on attaque doit contenir un buffer
- celui-ci doit être remplit par des données qui viennent de l'utilisateur, c'est-à-dire que l'attaquant doit pouvoir y insérer les données qu'il souhaite
- ce remplissage doit se faire à l'aide d'une fonction non sécurisée, c'est-à-dire, qui ne contrôle pas si la chaîne qu'elle copie est plus grande ou non que la destination, exemple : strcpy, strcat, sprintf, vsprintf, scanf (dans certains cas), getc....
- il faut, en outre, que le buffer soit suffisamment grand pour accueillir au minimum le shellcode et l'adresse de retour. Le shellcode que nous avons donné en exemple pèse 46 octets.

Contre-mesures Il existe plusieurs manières de parer ce genre d'attaque :

- Utilisation des fonctions « en n » :
  Au lieu d'utiliser la fonction strcpy qui n'effectue aucun contrôle sur la longueur de la chaîne source et celle de la chaîne de destination, on peut utiliser la fonction strncpy qui copie la première dans la deuxième jusqu'à ce qu'une certaine longueur soit atteinte (longueur donnée en troisième argument). Il est nécessaire ensuite d'ajouter le « \0 » terminal.
  Exemple :
  ```
  void copie2 (char *args)
  { char buffer[200] ;
  strncpy(buffer, args, sizeof(buffer)-1) ;
  buffer[sizeof(buffer)-1] = '\0' ;
  }
  ```
  Dans le cas présent, si args est plus grand que buffer, il sera tronqué de manière à laisser un caractère vide dans buffer : le « \0 ». Si la taille de args est inférieur au troisième argument, le remplissage de buffer sera complété par des caractères nuls jusqu'à arriver à la limite (fixé par le troisième argument).
- Rendre la pile non exécutable :
  Il est possible sous certaines architectures de rendre la pile non exécutable, c'est à dire de faire en sorte qu'aucune donnée lu dans la pile ne puisse être exécutée. Ainsi l'utilisation d'un shellcode est inutile. PaX, un patch de sécurité pour Linux permet cela [11]

3.1.2 Réécriture du sfp - *return-into-libc*

Principe Pour la méthode suivante, il n'est pas nécessaire que le buffer soit particulièrement grand et surtout elle permet de contourner le fait que la pile ne soit pas exécutable. Elle s'appelle return-into-libc car elle consiste à établir un lien vers une fonction chargée dynamiquement, cette fonction se trouve généralement dans la libc, d'où ce nom. Dans notre cas cette fonction sera system() qui prend deux paramètres :

- une adresse de retour, à laquelle se rendra le programme une fois que system() et le programme lancé par system() auront terminés ; dans notre cas il s'agira de la fonction exit(), cela permet au programme de se terminer proprement
- l'adresse d'une chaîne de caractères contenant le nom complet du programme à appeler, dans notre cas « /bin/sh »

Le principe est un peu le même que pour l'attaque précédente : il faut récrire le rip avec l'adresse de system() , et il faut qu'il y ait au dessus les paramètres de system(), soit : l'adresse de exit() et l'adresse de la chaîne « /bin/sh ».
Nous obtiendrons ces trois adresses en utilisant gdb et le programme mem-dump.c qui a été écrit par Clad Strife (et dont on peut trouver le code en [14]). Le premier nous permettra de trouver les adresses de system() et exit() et le deuxième nous cherchera la chaîne « /bin/sh » dans la libc.

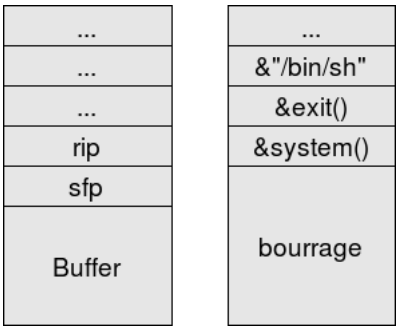
Exemple
```
$ gdb vulnerable
(gdb) r
Starting program : vulnerable

Program exited normally.
(gdb) x/x system
0x49c24ef0 <system> : 0x890cec82
(gdb) x/x exit
0x49c1a4c0 <exit> : 0x57e58956
```
Nous utilisons le même programme que précédemment (vulnerable.c) car il se prête bien à ce type d'attaque même si comme nous l'avons vu, il n'est pas nécessaire que le buffer soit si grand.
Nous avons désormais les adresses de system() et exit(). Il ne reste plus que « /bin/sh » :

```
   $ ps
PID TTY TIME CMD
7210 pts/1 00 :00 :00 bash
7902 pts/1 00 :00 :00 ps
$ cat /proc/7210/maps | grep libc
49bed000-49d31000 r-xp 00000000 08 :12 711057 /lib/tls/i686/cmov/libc-2.6.1.so
```

```
49d31000-49d32000 r-p 00143000 08 :12 711057 /lib/tls/i686/cmov/libc-2.6.1.so
49d32000-49d34000 rw-p 00144000 08 :12 711057 /lib/tls/i686/cmov/libc-2.6.1.so
$ ./memdump /bin/sh 7210 49bed000
Searching...
[/bin/sh] found in processus 7210 at : 0x49d150ae
```

On utilise le pid du bash utilisé car l'adresse recherché ne varie pas d'un programme à l'autre dans le cas présent.
Nous avons donc nos trois adresses, il suffit donc de remplir un buffer comme ceci : [bourrage de 204 octets] [&system] [&exit] [&"/bin/sh"]. Voici l'état de la pile avant et après l'attaque :



Le bourrage doit être de la taille du buffer plus un mot pour que l'adresse de system() se trouve à la place du rip et non du sfp.

Limites de la méthode Ceux sont essentiellement les mêmes que celles de la condition précédente, avec une de moins cependant, il n'est plus nécessaire que le buffer soit grand.

Contre-mesures

- La principale contre-mesure est PaX, que nous avons déjà évoqué et plus précisément il s'agit de l'address space layout randomization qu'il intègre. Celle-ci va rendre aléatoire les adresses auxquelles seront chargées les fonctions dynamiques.
- Là encore, l'utilisation des fonctions « en n » est efficace car elle ne permet plus le dépassement de buffer.

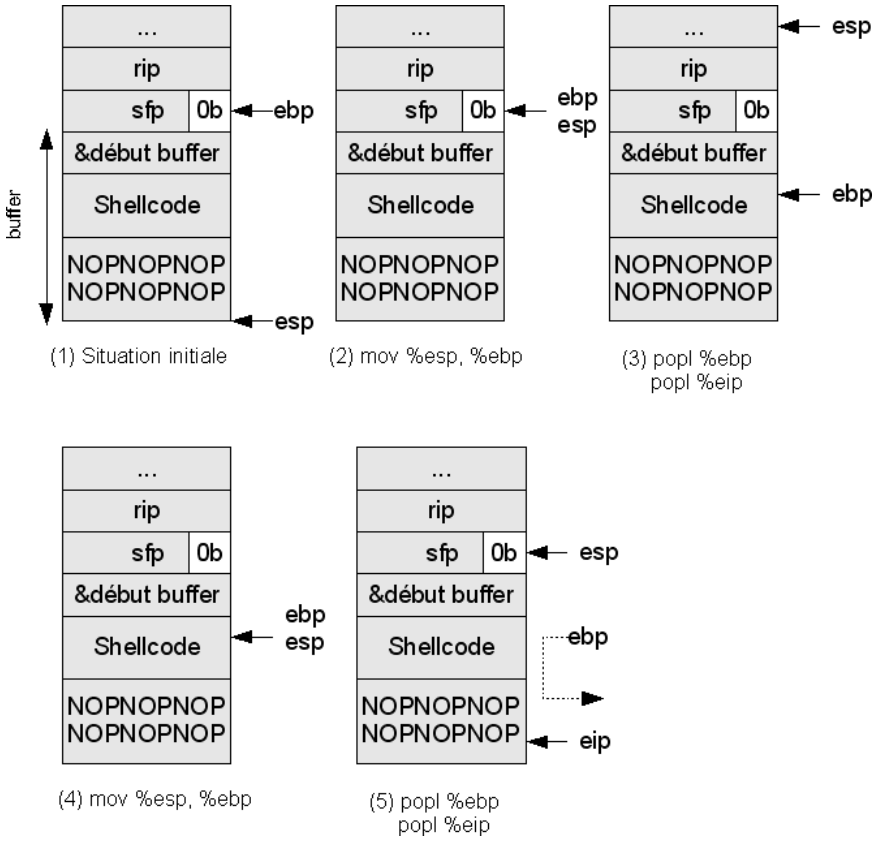
3.1.3 Réécriture du sfp - erreurs de type off-by-one

Principe Une erreur de type off-by-one est une erreur « de un pas », elle survient en général quand on utilise « <= » à la place de « < » (ou inversement), en voici un exemple : `void copie2 (char *args)`
`{ char[] tab = char[200] ;`
`int i ;`
`for(i = 0 ; i <= 20 ; i++)`
`tab[i] = args[i] ;`
`}` args[200] ne sera pas écrit dans tab[200] car l'indice maximum de tab est 199, il sera donc écrit sur ce qui est juste avant dans la stack. Dans le cas présent, la pile ressemble à ceci :

```
  i         tab                                      sfp         eip
[         ][                  ...                  ][         ][         ]
```

Donc args[200] va réécrire le premier octet de sfp. Or si nous nous trouvons sur une architecture little-endian (c'est le cas de x86 qui est l'architecture que l'on retrouve dans les PC), le premier octet est celui de poids faible. Un attaquant peut donc, tirer partie de cette erreur de programmation pour modifier l'octet de poids faible du sfp. Exemple : si l'on considère l'adresse 0xa0b70708, il peut en modifier les deux derniers chiffres hexadécimaux.
Voyons maintenant, sur un exemple, comment cet attaquant peut profiter de cette opportunité pour lancer un shell. L'attaque sera du même type que la première que nous avons vu : le buffer contiendra des NOP, le shellcode et l'adresse de début du buffer.

Exemple Le schéma suivant illustre le déroulement de l'attaque :



- en (1), on réécrit le premier octet du sfp, pour des raisons de lisibilité nous l'avons mis à la fin alors qu'il se trouve au début : comme nous l'avons dit précédemment il s'agit de l'octet de poids le plus faible donc des des deux chiffres hexadécimaux les plus à droite lorsque l'on écrit l'adresse en hexadécimal. Ce nouvel sfp doit pointer sur le dernier mot du shellcode.
- (2) et (3) correspondent au leave de copie2() ; esp pointe désormais sur le dernier mot du shellcode, donc pour la machine, le rip se trouve dans la case précédente : celle qui contient l'adresse de début du buffer
- (4) et (5) correspondent au leave de la fonction main() ; lors du « popl %ebp », la machine interprète le dernier mot du shellcode comme une adresse et la copie dans ebp, celui-ci pointe vers une zone quelconque de la mémoire mais cela est sans importance : eip pointe désormais sur le début du buffer et donc, la machine, après avoir lu tous les NOP, va exécuter le shellcode.

Limites de la méthode Trois conditions doivent être remplies pour que cette attaque fonctionne :

- il faut que l'octet de poids faible du sfp soit suffisamment élevé : si sfp est du type 0x******00, on ne pourra pas décaler vers le bas. De manière plus précise, deux chiffres hexadécimaux permettent de représenter un décalage de 0 à 255 octets, donc de 0 à 63 mots. Mais on ne peut décaler de 63 mots que si l'adresse original se termine par « ff ». Il faut donc que ces deux chiffres originaux permettent ce décalage de n+2 mots où n est le décalage entre le sfp et le mot sur lequel il pointe, car on souhaite descendre 2 mots en dessous du mot contenant le sfp.
- corollaire de la condition précédente : s'il y a trop de variables entre le sfp et le buffer, l'attaque n'est plus possible
- il est aussi nécessaire qu'il n'y ait pas d'écriture sur la pile entre le leave de la fonction copie2() et celui de la fonction main() : le sfp pointe sur une case se trouvant au dessus du shellcode, il pourrait donc réécrire dessus.

Contre-mesures Ce sont les mêmes que pour la première technique présentée :

- utilisation des fonctions « en n »
- rendre la pile non-exécutable

4 Pointer Subterfuge

Ce type d'attaque peut avoir lieu lorsque le programme cible contient un buffer dont la déclaration précède celle d'un pointeur. On peut alors modifier la valeur de ce pointeur et utiliser cela pour prendre le contrôle de la machine hôte. Ces attaques ont vu le jour pour contrer les mécanismes de protection de la stack dont nous avons parlé. Nous allons aborder deux cas de figure : le pointeur de fonction et le pointeur de données.

4.1 Modification d'un pointeur de fonction

4.1.1 Principe

Nous avons donc un programme contenant la déclaration d'un buffer, immédiatement suivie de celle d'un pointeur de fonction, ce buffer est rempli par des données venant de l'utilisateur et cette fonction est appelée quelque part dans la suite du programme.
L'idée est donc de réécrire ce pointeur de fonction pour qu'il pointe vers un code que l'on aura apporté, par exemple, un shellcode qui se trouvera dans le buffer, comme nous l'avons déjà fait plusieurs fois.

4.1.2 Exemple

Le programme que nous attaquons contient la fonction suivante :

```
void vuln(void *arg, size_t len)
  {
        char buffer[256] ;
        void (*f)() = ... ;

        memcpy(buffer, arg, len) ;
        ...
        f() ;

        return 0 ;
  }
```

Ainsi en dépassant le buffer, on peut réécrire le pointeur *f (c'est pour cela que sa réel déclaration importe peu) de manière à le faire pointer sur le début du buffer qui contient notre shellcode (éventuellement précédé d'une série de NOPs).

4.1.3 Limites de la méthode

Si la fonction est déclarée *extern*, il n'est plus possible de la modifier.

4.1.4 Contre-mesures

Vérifier le code source pour ne jamais présenter ce genre de faille, donc dans l'exemple précédent, limiter la copie à la taille du buffer et non à len ou bien vérifier que len est inférieur à la taille du buffer.

4.2 Modification d'un pointeur de données

4.2.1 Principe

Nous nous trouvons maintenant dans la situation suivante : nous avons, juste après la déclaration du buffer, celles d'un pointeur puis d'une variable. De plus, il devra y avoir, après que le buffer ait été rempli, une affectation de la valeur de la variable dans la case sur laquelle pointe le pointeur.

Cette affectation va nous permettre de modifier une valeur où l'on veut dans la mémoire. S'il s'agit d'un pointeur de type *long* comme dans l'exemple qui suit, cela nous permet de modifier un mot (4 octets) de mémoire et nous allons voir comment on peut en tirer partie, en utilisant les deux méthodes que l'on vient d'aborder.

4.2.2 Exemple

Considérons la fonction suivante :

```
void vuln2(void *arg, size_t len)
  {
        char buffer[256] ;
        long val = ... ;
        long *ptr = ... ;
        extern void (*f)() ;

        memcpy(buffer, arg, len) ;
        ...
        *ptr = val ;
        ...
        f() ;

        return 0 ;
  }
```

On voit bien ici qu'en dépassant le buffer, on va pouvoir réécrire les deux variables *val* et **ptr*. Et l'affectation « *ptr = val » nous permet ainsi de modifier la portion de mémoire que l'on veut. Comme il s'agit de long, nous pouvons comme nous l'avons vu modifier un mot, ce qui est suffisant pour détourner la fonction *f()* vers notre shellcode qui se trouve comme précédemment dans le buffer.

4.2.3 Limites de la méthode

Aucune limite connue, si ce n'est la présence de ce buffer avant les déclarations du pointeur et de la variable.

4.2.4 Contre-mesures

Comme dans le cas du pointeur de fonction, la principale parade consiste à vérifier que l'on déborde pas des bornes du buffer et à ne pas laisser un code dans une configuration aussi favorable à une attaque.

5 Conclusion

Comme nous l'avons vu, il existe de nombreuses façons de mener à bien une attaque de type Stack Overflow. Il s'agit en général d'exploiter d'une configuration un peu particulière d'un programme ou d'une fonction.

Mais il existe des parades : la première d'entre elle consiste à programmer de manière « sécurisée », c'est-à-dire en utilisant les fonctions qui effectuent des tests sur les longueurs de chaînes qu'elles copient. Il n'est pas évident de demander à l'ensemble des programmeurs de tenir compte de cela, car hélas beaucoup ne savent pas comment fonctionne la stack et ne sont pas sensibilisés à ce problème. Cependant il existe des logiciels de tests et d'audit de programmes qui peuvent être utilisés pour réduire considérablement le nombre de failles.

Et pour les programmes dont le code n'est pas accessible, il existe des solutions telles que PaX qui permettent d'augmenter la sécurité en réduisant considérablement les possibilités des attaquants.

Références

[1] Aleph One, Smashing The Stack For Fun and Profit, Phrack 49, 1996

[2] Klog, The Frame Pointer Overwrite, Phrack 55, 1999

[3] Frédéric Raynal, Christopher Blaess, Christopher Grenier, Éviter les failles de sécurité dès le développement d'une application : mémoire, pile et fonctions, shellcode, http ://www.cgsecurity.org/Articles/SecProg/Art2/index-fr.html, 2001

[4] Frédéric Raynal, Christopher Blaess, Christopher Grenier, Éviter les failles de sécurité dès le développement d'une application : débordements de buffer, http ://www.cgsecurity.org/Articles/SecProg/Art3/index-fr.html, 2001

[5] Matthias Vallentin, On the Evolution of Buffer Overflows, 2007

[6] Stéphane Zampelli, Benoit Georges, Emmanuel Koch, Gauthier van den Hove, Construction et détection d′une attaque par "buffer overflows", http ://www.zone-h.fr/files/7/Construction_detection_BOF.pdf, 2002

[7] Denis Ducamp, Introduction à l'exploitation des débordements de tampons, http ://www.hsc.fr/ressources/breves/stackoverflow-exploit.html.fr, 2001

[8] Exploitation Avancée de Stack Overflow Vulnerabilities, http ://ouah.org/BO-RedKod.htm, 2002

[9] Buffer overflow,
http ://asphalos.net/xoops/modules/wiwimod/index.php ?page=BufferOverflow& back=ExPloit, 2007

[10] Commentcamarche, Attaques : Débordement de tampon (Buffer overflow), http ://www.commentcamarche.net/attaques/buffer-overflow.php3

[11] Wikipedia, PaX, en.wikipedia.org/wiki/PaX

[12] Wikipedia, Dépassement de tampon,
http ://fr.wikipedia.org/wiki/D%C3%A9passement_de_tampon

[13] tolwin, Introductions aux débordements de tampon, http ://www.blackclowns.org/articles/WinBufferOverflow.pdf, 2003

[14] Clad Strife, Ret Onto Ret Into Vsyscalls, http ://www.blackclowns.org/articles/RetOntoRetIntoVsyscalls, 2005