6. *Level 06*

On reconstruit le code source suivant ; on est toujours en **x32** :

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

int     auth(char *buff, unsigned int serial)
{
    size_t              s;          // [ebp-0xc]
    unsigned int        key;        // [ebp-0x10]
    int                 n;          // [ebp-0x14]

    buff[strcspn(buff, "\n")] = '\0';
    s = strnlen(buff, 32);

    if (s <= 5)
        return (1);
    if (ptrace(PTRACE_TRACEME) == -1)
    {
        puts("\033[32m.-------------------------.");
        puts("\033[31m| !! TAMPERING DETECTED !!  |");
        puts("\033[32m'-------------------------'");
        return (1);
    }
    key = ((int)buff[3] ^ 0x1337) + 0x5eeded;
    n = 0;

    while (n < (int)s)
    {
        unsigned int v2, v3, v4;

        if (buff[n] <= 31)
            return (1);

        v2 = buff[n];
        v2 ^= key;
        v3 = (uint64_t)v2*0x88233b2b >> 32;
        v4 = v2 - v3;
        v4 = v4 >> 1;
        v4 += v3;
        v4 = v4 >> 10;
        v4 *= 0x539;
        v4 = v2 - v4;

        key += v4;
//      key = key + ((int)buff[n] ^ key) % 0x539;
```

```
        n++;
    }
    if (serial == key)
        return (0);
    return (1);

}

int    main(int argc, char **argv)
{
    char          *av_0;            // [esp+0x1c]
    int           n;                // [esp+0x4c]
    char          buff[32];         // [esp+0x2c]
    unsigned int  serial;           // [esp+0x28]

    av_0 = argv[0];
    n = 0x14;

    puts("***********************************");
    puts("*        level06        *");
    puts("***********************************");

    printf("-> Enter Login: ");
    fgets(buff, 32, stdin);

    puts("***********************************");
    puts("***** NEW ACCOUNT DETECTED ********");
    puts("***********************************");

    printf("-> Enter Serial: ");
    scanf("%u", &serial);

    if (auth(buff, serial) == 0)
    {
        puts("Authenticated!");
        system("/bin/sh");
        return (0);
    }
    return (1);
}
```

De manière générale, le programme effectue les actions suivantes :
> Demande un username.
> Demande un numéro de série (**serial**).
> Génère la clé correspondant au username renseigné.
> Vérifie que le numéro de série soit égal à la clé.
> Si oui, exécute un **system("/bin/sh")**.
> Si non, quitte le programme.

Une première remarque. On remarque le passage suivant :

```
if (ptrace(PTRACE_TRACEME) == -1)
{
      return (1);
}
```

Il s'agit simplement d'une méthode empêchant de parcourir le reste du programme avec un debugger comme **gdb**. En effet, un process **ne peut avoir qu'un seul "tracer"** (programme qui le trace). Or, lorsqu'on utilise **gdb**, **gdb** est déjà en train de tracer le programme qui s'exécute ; **ptrace** va renvoyer -1, et on va exit le programme.

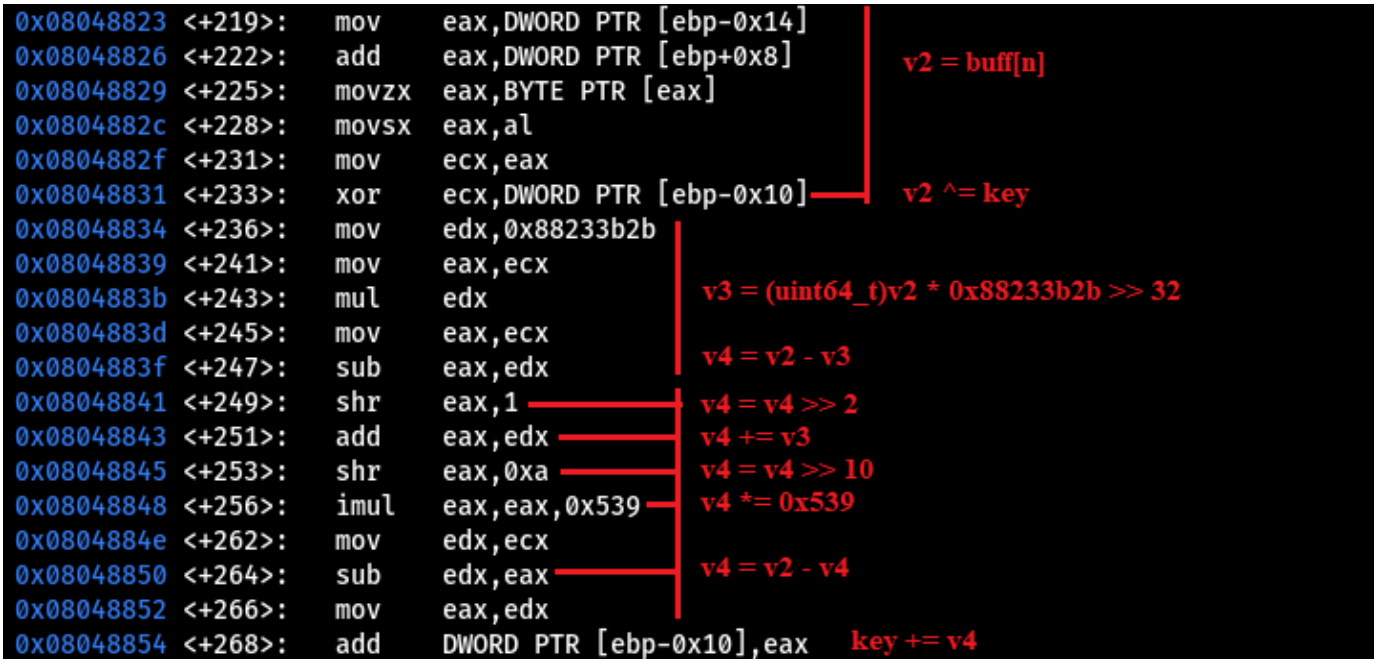
Il s'agit d'une protection vraiment simple à contourner, on place simplement un **breakpoint** dans **gdb** à l'endroit où la vérification de la valeur de retour de **ptrace** est effectuée, on change la valeur de **eax** pour la rendre différente de **-1**, et le programme continue son exécution dans le debugger.

A part ça, la partie un peu pénible à reconstruire à partir du code assembleur était celle-ci :

```
v2 = buff[n];
v2 ^= key;
v3 = (uint64_t)v2*0x88233b2b >> 32;
v4 = v2 - v3;
v4 = v4 >> 1;
v4 += v3;
v4 = v4 >> 10;
v4 *= 0x539;
v4 = v2 - v4;

key += v4;
```

La partie correspondante en assembly était la suivante :



La partie un peu difficile à comprendre était relative à l'instruction **mul**. On a trouvé une explication dans un writeup (qui ressemblait étrangement à notre exercice) :
https://devel0pment.de/?p=4

"The reason is that **mul** implicitly uses the eax register. The operand is multiplied by eax and the result is stored in eax **and** the operand register because the product of two 32-bit registers may exceed 32-bit. The **lower bits are stored in eax** and **the upper bits are stored in the operand register.**"

Ici, dans l'enchaînement d'instructions assembly, on voit que seul ce qui a été stocké dans l'operand (**edx**) est utilisé ; en d'autres termes, seuls les 32 **upper bits** sont utilisés.

Pour simuler cette action en assembleur, on stock le résultat de la multiplication dans un entier de **64 bits** (simulant eax **et** edx), puis on shift de 32 bits vers la droite afin de ne conserver que les **16 upper bits**.

On s'aperçoit que toute cette fonction / cette série d'instructions peut en réalité s'écrire sous la forme suivante (une décompilation avec **ghidra** nous l'a indiqué) :

```
key = key + ((int)buff[n] ^ key) % 0x539;
```

Bref. Quoi qu'il en soit, nous savons exactement comment le programme génère une clé associée à un nom d'utilisateur (**xor** le 3ème caractère, puis pour chaque caractère du nom d'utilisateur effectuer le modulo indiqué ci-dessus).

Il est donc trivial de reproduire la procédure de génération de la clé pour un nom d'utilisateur donné. On utilise exactement le même procédé ; au lieu de vérifier la clé générée contre le numéro de série fourni par l'utilisateur, on se contente de l'afficher :

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

void    keygen(char *buff)
{
    size_t        s;
    unsigned int    key;
    int        n;

    s = strnlen(buff, 32);
    if (s <= 5)
        return ;

    key = ((int)buff[3] ^ 0x1337) + 0x5eeded;
    n = 0;

    while (n < (int)s)
    {
        unsigned int v2, v3, v4;

        if (buff[n] <= 31)
            return ;

        v2 = buff[n];
        v2 ^= key;
```

```
        v3 = (uint64_t)v2*0x88233b2b >> 32;
        v4 = v2 - v3;
        v4 = v4 >> 1;
        v4 += v3;
        v4 = v4 >> 10;
        v4 *= 0x539;
        v4 = v2 - v4;

        key += v4;


        //        key = key + ((int)buff[n] ^ key) % 0x539;
        n++;
    }
    printf("[+] %u\n", key);
}


int     main(int argc, char **argv)
{
    keygen(argv[1]);
    return (0);
}
```

Ce programme génère la bonne clé pour un nom d'utilisateur donné, i.e. la clé qui sera comparée au numéro de série fourni par l'utilisateur dans le véritable programme afin de décider de l'exécution de **system("/bin/sh")**.

```
→ level06 gcc -m32 -no-pie keygen.c -o keygen
→ level06 ./keygen Quentin
[+] 6233795
→ level06 ./level06
***********************************
*               level06           *
***********************************
-> Enter Login: Quentin
***********************************
***** NEW ACCOUNT DETECTED ********
***********************************
-> Enter Serial: 6233795
Authenticated!
# id
uid=0(root) gid=0(root) groupes=0(root)
```

>> Exploitation manuelle

```
$ ./level06
[...]
-> Enter login: Quentin
[...]
-> Enter serial: 6233795
```

>> Exploit automatique

```python
from pwn import *

s = ssh(host='192.168.1.3', port=4242, user="level06",
password="h4GtNnaMs2kZFN92ymTr2DcJHAzMfzLW25Ep59mq")
p = s.process(["/home/users/level06/./level06"])

p.sendline(b"Quentin")
p.sendline(b"6233795")
p.interactive()
```