3. *Level 03*

On revient sur un binaire 32 bits ici. On reconstruit le code source suivant :

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


void    decrypt(unsigned int r)
{
    int     n_2 = 0x757c7d51;       // [ebp-0x1d]
    int     n_3 = 0x67667360;       // [ebp-0x19]
    int     n_4 = 0x7b66737e;       // [ebp-0x15]
    int     n_5 = 0x33617c7d;       // [ebp-0x11]
                                    // "Q}|u`sfg~sf{}|a3"

    char     c = 0x0;               // [ebp-0xd]
    int     size = 0;               // [ebp-0x24]
    int     n_7 = 0;                // [ebp-0x28]
    char     *correct = "Congratulations!";

    while (*((char *)&n_2 + size) != 0)
        size++;
    while (n_7 < size)
    {
        *((char *)&n_2 + n_7) = r ^ *((char *)&n_2 + n_7);
        n_7++;
    }

    int i = 0;
    while (*((char *)&n_2 + i) == correct[i] && i < 16)
        i++;
    if (i == 16)
    {
        system("/bin/sh");
        return ;
    }
    else
        puts("\nInvalid Password");
    return ;
}


void    test(int n, int hex)
{
    unsigned int r;                 // [ebp-0xc]

    r = hex - n;
    if (r < 0x15)
```

```
    {
        unsigned int tmp = r*4;
        if (tmp == 0)
            goto RAND;
        if (tmp == 4)
            goto G46;
        if (tmp == 8)
            goto G62;
        if (tmp == 12)
            goto G78;
        if (tmp == 16)
            goto G94;
        if (tmp == 20)
            goto G110;
        if (tmp == 24)
            goto G126;
        if (tmp == 28)
            goto G142;
        if (tmp == 32)
            goto G155;
        if (tmp == 36)
            goto G168;
        if (tmp == 40 || tmp == 44 || tmp == 48 || tmp == 52 || tmp == 56 || tmp == 60)
            goto RAND;
        if (tmp == 64)
            goto G181;
        if (tmp == 68)
            goto G194;
        if (tmp == 72)
            goto G207;
        if (tmp == 76)
            goto G220;
        if (tmp == 80)
            goto G233;
        if (tmp == 84)
            goto G246;
    }
    else
    {
    RAND:
        decrypt(rand());
        return ;
    }

G46:
    decrypt(r);
    return ;
G62:
    decrypt(r);
    return ;
G78:
```

```
    decrypt(r);
    return ;
G94:
    decrypt(r);
    return ;
G110:
    decrypt(r);
    return ;
G126:
    decrypt(r);
    return ;
G142:
    decrypt(r);
    return ;
G155:
    decrypt(r);
    return ;
G168:
    decrypt(r);
    return ;
G181:
    decrypt(r);
    return ;
G194:
    decrypt(r);
    return ;
G207:
    decrypt(r);
    return ;
G220:
    decrypt(r);
    return ;
G233:
    decrypt(r);
    return ;
G246:
    decrypt(r);
    return ;

    return ;
}


int     main(void)
{
    int     n;                      // [esp+0x1c]

    srand(time(NULL));
    puts("**********************************");
    puts("        level03        **");
    puts("**********************************");
```

```
    printf("Password:");
    scanf("%d", &n);
    test(n, 0x1337d00d);
    return (0);
}
```

On avait plein de sauts inconditionnels **jmp** dans la fonction **test**, qui produit cet ensemble de **goto** aussi moche qu'inutile. Les deux fonctions réellement importantes étaient **main** et **decrypt**.

Pour résumer le déroulé du programme :

> On demande à l'utilisateur un **entier n**.
> On transmet cet entier, ainsi que `0x1337d00d` (--> 322424845) à la fonction **test**.
> On stock le résultat de la différence `0x1337d00d - n` dans un `unsigned int r`.
> Si cet `unsigned int r` est inférieur à `0x15` (21), on appellera (probablement) `decrypt` avec en argument cet `unsigned int r` (sauf si r*4 est égal à 40 – 44 – 48 – 52 – 56 – 60).

> La fonction **decrypt** a comme argument local la chaîne encryptée "`Q}|u`sfg~sf{}|a3`" (sous la forme de 4 ints consécutifs, suivis d'un char 0x0 pour le NULL BYTE).
> Pour chaque caractère de cette chaîne, la fonction effectue un **xor** sur le caractère avec le **unsigned int r**.
> Si, après les opérations **xor**, la chaîne résultante est "`Congratulations!`", on nous donne une **shell system**. Sinon, on affiche "`Invalid Password`".

Au vu de ce mécanisme, on peut bruteforce la clé de manière assez aisée. Il est certain que le **unsigned int r** doit être entre 0 et 0x15 (0 et 21) pour que la clé soit la bonne (sinon, **decrypt** est appelée avec un chiffre aléatoire, sans prendre en aucun cas en compte l'input utilisateur dont le mot de passe ne sera pas validé).

On va donc simplement tester toutes les valeurs de mot de passe telles que **unsigned int r** adopte toutes les valeurs entre 0 et 21.

Puisque **unsigned int r** est calculé à partir de `322424845 - <input utilisateur>`, on va simplement essayer avec les valeurs suivantes :

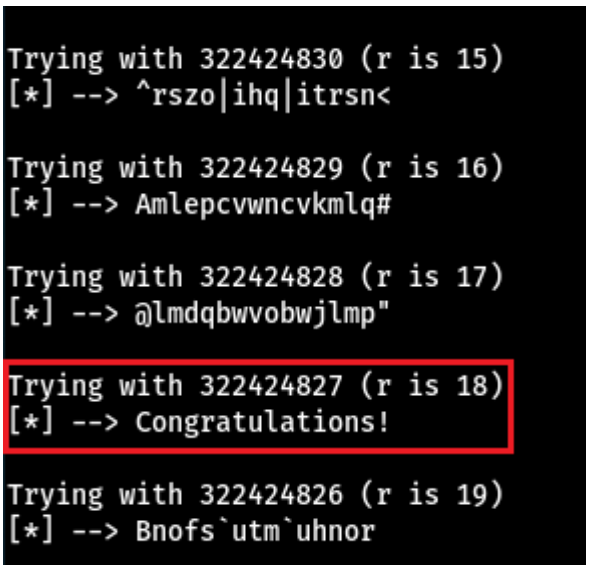
- 322424845 – **322424845** (r = 0)
- 322424845 – **322424844** (r = 1)
- 322424845 – **322424843** (r = 2)
- etc...

Jusqu'à ce que la chaîne encryptée nous retourne "Congratulations!". On le fait simplement en C :

```c
#include <stdio.h>

void decrypt(unsigned int r)
{
    char *enc = "Q}|u`sfg~sf{}|a3";
    int    size = 0;
    int    i = 0;
    char result[64];

    while (enc[size])
        size++;
    while (i < size)
    {
        result[i] = (char)(r ^ (*(enc + i)));
        i++;
    }
    result[i] = '\0';
    printf("[*] --> %s\n", result);
    return ;

}

int    main(void)
{
    int d00d    = 322424845;
    int guess    = 322424845;

    int i = 0;
    while (guess + i < d00d + 0x15)
    {
        unsigned int r = d00d - (guess - i);
        printf("\nTrying with %d (r is %u)\n", guess - i, r);
        decrypt(r);
        i++;
    }
    return (0);
}
```

On diminue progressivement la valeur de notre input (**guess**), ce qui fait adopter à **r** toutes les valeurs entre 0 et 21. Dans la fonction **decrypt**, on prend la chaîne encryptée, et on effectue un **xor** à partir de la valeur de **r** ; on affiche ensuite le résultat.

Cela finit par fonctionner pour l'input **322424827** (produisant une valeur de 18 pour r).



On donne en input au programme cette valeur, et on obtient une shell.

>> Exploitation manuelle

```
$ ./level03
Password:322424827
```

>> Exploit automatique

```
from pwn import *

s = ssh(host="192.168.1.3", port=4242, user="level03",
password="Hh74RPnuQ9sa5JAEXgNWCqz7sXGnh5J5M9KfPg3H")
p = s.process("/home/users/level03/./level03")

p.recvline()
p.recvline()
p.recvline()
p.sendline(b"322424827")
p.interactive()
```