./level03

```
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE        No RPATH   No RUNPATH   /home/user/level03/level03

level03@OverRide:~$
```

Decompiled file with ***Ghidra***:

```
void decrypt(int key)
{
    char cipher[21] = "Q}|u`sfg~sf{}|a3";
    size_t len = strlen(cipher);

    for (size_t i = 0; i < len; i++)
        cipher[i] ^= key;

    if (!strcmp(cipher, "Congratulations!"))
        system("/bin/sh");
    else
        puts("Invalid Password!");
}

void test(int arg1, int arg2)
{
    int diff = arg2 - arg1;

    if ((diff > 0 && diff < 22))
        decrypt(diff);
    else
    {
        int randomValue = rand();
        decrypt(randomValue);
    }
}

int main(void)
{
    int userInput;
    srand((unsigned)time(NULL));

    puts("***********************************");
    puts("*               level03         **");
    puts("***********************************");
    printf("\nPassword:");
    scanf("%d", &userInput);
    test(userInput, 0x1337d00d);
    return EXIT_SUCCESS;
}
```

This **C program** is a simple **password checker** that uses a cryptographic **XOR operation** for validation. It begins by asking for an integer password from the user. Internally, it takes the user input and calculates the difference from the **hexadecimal** constant 0x1337d00d. This difference is then used as a **key** to decrypt a hardcoded cipher text.

The valid range for the **key** is limited, as indicated by the **conditional checks** in the program: it must be between 1 and 21, inclusive.
If the difference doesn't fall within these ranges, the program will use a **random value** as the **key**, which typically results in decryption failure and an **Invalid Password!** message.

The decryption process involves a **bitwise XOR operation** (exclusive OR), a simple bitwise operation that gives 0 if the bits are the same, and it gives 1 if the bits are different.

The encrypted string in the program is **Q}|u`sfg~sf{}|a3**. If, after being XORed with the **key**, it matches **Congratulations!**, the program opens a system **shell**.

To **crack** the program, we need to *reverse-engineer* the **key** from the known **plaintext** and the **encrypted** string. By **XOR**ing these two strings, we obtain the **key**:

```
Q        }        |        u        ...  }        |        a        3
01010001 01111101 01111100 01110101      01111101 01111100 01100001 00110011
C        o        n        g        ...  o        n        s        !
01000011 01101111 01101110 01100111      01101111 01101110 01110011 00100001
-------------------------------------------------------------------------------
00010010 00010010 00010010 00010010      00010010 00010010 00010010 00010010
```

The key is 10010_2 (12_{16}) and can then be used to find the correct password: it's the number that, when subtracted from 0x1337d00d, yields the key.

```
level03@OverRide:~$ {
    python -c 'print str(0x1337d00d - 0x12)';
    echo "cd ../level04 && cat .pass";
} | ./level03

***********************************
*               level03         **
***********************************
kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf

level03@OverRide:~$ su level04
Password: kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf

level04@OverRide:~$
```