

# Fake code 题目说明

## 初步分析

用64位IDA打开这个程序并且F5，你会看到主函数长这个样子：

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      int result; // eax
4      int i; // [rsp+20h] [rbp-B8h]
5      int v5; // [rsp+24h] [rbp-B4h]
6      __int64 v6; // [rsp+30h] [rbp-A8h]
7      char v7[112]; // [rsp+50h] [rbp-88h] BYREF
8
9      v5 = 0;
10     puts("Can you read my assembly in exception?");
11     puts("Give me your flag:");
12     sub_140001290("%s", v7);
13     v6 = -1i64;
14     do
15         ++v6;
16     while ( v7[v6] );
17     if ( v6 == 51 )
18     {
19         for ( i = 0; i < 51; ++i )
20         {
21             v5 = (127 * v5 + 102) % 255;
22             v7[i] ^= byte_140005010[dword_140005000];
23         }
24         if ( (unsigned int)sub_140001020(&unk_140005110, v7) )
25             puts("\nTTTTTTTTTTQQQQQQQQQQQQLLLLLLLLLL!!!!");
26         else
27             puts("\nQwQ, please try again.");
28         result = 0;
29     }
30     else
31     {
32         puts("\nQwQ, please try again.");
33         result = 0;
34     }
35     return result;
36 }
```

显然主要的加密逻辑就是19到23行的for循环。

这个for循环里，第21行是使用一个式子来不停地更新v5的值，而14行是用 byte\_140005010[dword\_140005000] 来与你的输入（即v7）异或。双击 byte\_140005010，可以在IDA里看到，这是一个长度为255的box。双击 dword\_140005000 则可以看到，这是一个值为0x19的常量。

你也许会奇怪，这里用来异或的box和索引都是固定的值，而且后面也并没有用到v5，那么前面更新v5的值的意义在哪里？

出题人当然不会在这里无缘无故写一个没用的变量，之所以会产生这种情况，是因为有时候IDA会欺骗你，就像现在这样。

IDA反编译（按F5）产生的毕竟只是伪代码，不能完全信任，所以当伪代码晦涩难懂或者显然存在一些问题时，我们经常需要去查看汇编代码。

对于这个题而言，点进main函数的汇编窗口，如果你的IDA是下图这种流程图的样子，按空格切换到汇编代码。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near

var_A8= dword ptr -0A8h
var_A4= dword ptr -0A4h
var_A0= dword ptr -0A0h
var_88= byte ptr -88h
var_18= qword ptr -18h

; FUNCTION CHUNK AT .text:000000140002080 SIZE 0000002B BYTES

; __unwind { // __GSHandlerCheck_SEH
sub     rsp, 0C8h
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rsp+0C8h+var_18], rax
mov     [rsp+0C8h+var_A4], 0
lea     rcx, Buffer ; "Can you read my assembly in exception?"
call    cs:puts
lea     rcx, aGiveMeYourFlag ; "Give me your flag:"
call    cs:puts
mov     r8d, 64h ; 'd'
lea     rdx, [rsp+0C8h+var_88]
lea     rcx, aS ; "%s"
call    sub_140001240
mov     [rsp+0C8h+var_A8], 0
jmp     short loc_140001166

loc_140001166:
cmp     [rsp+0C8h+var_A8], 33h ; '3'
jge     loc_1400011f2

loc_140001171:
__try { // __except at loc_1400011A2
    eax, [rsp+0C8h+var_A4], 7Fh
    eax, 66h ; 'f'

loc_1400011f2:
lea     rdx, [rsp+0C8h+var_88]
lea     rcx, unk_140004110
call    sub_140001020
test    eax, eax
```

# 结构化异常处理——SEH

main函数往下翻一点，你会看到这样的结构：

```
1 | .text:00000001400011B8 loc_1400011B8: ; DATA XREF: .rdata:0000000140003880↓o
2 | .text:00000001400011B8 ; __try { // __except at loc_1400011E9
3 | .text:00000001400011B8 imul eax, [rsp+0D8h+var_B4], 7Fh
4 | .text:00000001400011BD add eax, 66h ; 'f'
5 | .text:00000001400011C0 cdq
6 | .text:00000001400011C1 mov ecx, 0FFh
7 | .text:00000001400011C6 idiv ecx
8 | .text:00000001400011C8 mov eax, edx
9 | .text:00000001400011CA mov [rsp+0D8h+var_B4], eax
10 | .text:00000001400011CE mov eax, [rsp+0D8h+var_B4]
11 | .text:00000001400011D2 sar eax, 7
12 | .text:00000001400011D5 mov [rsp+0D8h+var_B0], eax
13 | .text:00000001400011D9 mov eax, 1
14 | .text:00000001400011DE cdq
15 | .text:00000001400011DF idiv [rsp+0D8h+var_B0]
16 | .text:00000001400011E3 mov [rsp+0D8h+var_B0], eax
17 | .text:00000001400011E7 jmp short loc_140001212
18 | .text:00000001400011E7 ; } // starts at 1400011B8
19 | .text:00000001400011E9 ; -----
20 | .text:00000001400011E9
21 | .text:00000001400011E9 loc_1400011E9: ; DATA XREF: .rdata:0000000140003880↓o
22 | .text:00000001400011E9 ; __except(loc_1400020D0) // owned by 1400011B8
23 | .text:00000001400011E9 imul eax, cs:dword_140005000, 61h ; 'a'
24 | .text:00000001400011F0 add eax, 65h ; 'e'
25 | .text:00000001400011F3 cdq
26 | .text:00000001400011F4 mov ecx, 0E9h
27 | .text:00000001400011F9 idiv ecx
28 | .text:00000001400011FB mov eax, edx
29 | .text:00000001400011FD mov cs:dword_140005000, eax
30 | .text:0000000140001203 mov eax, cs:dword_140005000
31 | .text:0000000140001209 xor eax, 29h
32 | .text:000000014000120C mov cs:dword_140005000, eax
```

这种 `__try{...} __except(filter){...}` 的形式是SEH，**Windows的一种异常处理机制**。

由于SEH中的很多代码不会被IDA反编译出来，所以它常常被用来反静态分析：程序员在try块中的某些情况下故意触发一些异常，来执行except块中他们想要隐藏起来的代码。

当然，SEH也可以反动态调试，由于跟本题关系不大，此处就不展开了。

如果想详细了解SEH可以康[出题人的博客](#)

简而言之，就像它们的名字一样，程序先执行try块中的逻辑，如果try中产生了一些异常的情况（可以简单理解为程序发生了一些意外），就会执行except中的代码来处理异常；反之，如果try块中没有产生异

常，except块就不会被执行。

当然了，except块中的内容是由程序员编写的，至于里面具体执行了什么东西，谁知道呢。除了处理异常之外，自然也可以干一些其他的事情，比如偷偷改掉一些关键数据之类的。

## try块

1400011B8~1400011E7处是try块的逻辑。

显然，IDA的伪代码关于v5的部分只翻译出了前7句，即  $v5 = (127 * v5 + 102) \% 255;$ ，此后还进行了一些运算。

前面说过：程序先执行try块中的逻辑，如果try中产生了一些异常的情况，就会执行except中的代码来处理异常。显然这里try块中可能产生异常的地方，就是IDA没有反编译出来的剩下的几句代码。

## except块

再来看except块，except块后面先是一个小括号，里面是 loc\_1400020D0，然后是一些汇编代码，1400011E9~14000120C，也就是except块中的代码逻辑。

双击 loc\_1400020D0，会跳转到这里：

```
1 | .text:00000001400020D0 ; __except filter // owned by 1400011B8
2 | .text:00000001400020D0          push   rbp
3 | .text:00000001400020D2          sub    rsp, 20h
4 | .text:00000001400020D6          mov    rbp, rdx
5 | .text:00000001400020D9          mov    [rbp+48h], rcx
6 | .text:00000001400020DD          mov    rax, [rbp+48h]
7 | .text:00000001400020E1          mov    rax, [rax]
8 | .text:00000001400020E4          mov    eax, [rax]
9 | .text:00000001400020E6          mov    [rbp+38h], eax
10 | .text:00000001400020E9          mov    eax, [rbp+38h]
11 | .text:00000001400020EC          mov    ecx, eax
12 | .text:00000001400020EE          call  sub_140001000
13 | .text:00000001400020F3          nop
14 | .text:00000001400020F4          add    rsp, 20h
15 | .text:00000001400020F8          pop    rbp
16 | .text:00000001400020F9          retn
```

这里是except的一个过滤器（即第一行提示的 \_\_except filter）。

过滤器又是什么？顾名思义，过滤器是用来过滤异常的。

try块中会产生很多异常，比如除数为0、非法内存访问、触发断点异常等等。当我们只希望except来处理某些特定的异常时，就会用到异常过滤器。

举个🔥：假如我在过滤器中规定了except只能处理除数为0的异常，那么它只能在发生除数为0的异常时才会执行，而在其他异常（非法内存访问、触发断点异常等等）发生时就不能处理。

这里看不懂也没关系，本题过滤器就是限制except块处理除数为0的异常（当然了，try块中也并没有发生其他的异常）。你只需要关注前面的try块和except块即可。

## 最后

那么这个题的考点是什么？

~~当然不能是SEH啦，不然我给你分析出来干什么（逃~~

其实就是阅读汇编代码，分析try块和except块中的代码逻辑：

- **try块中执行了什么？怎样触发除零异常？**
- **触发异常后，except块改变了哪里的数据？**

汇编功底对于后面的逆向工程学习还是很重要的，加油吧 :~)

如果对SEH部分还有什么不理解的欢迎来  出题人云之君