RE专项赛

0基础入门题

```
Buf1[1] = 0xF6CCE8C6;
Buf1[2] = 0xCA9680DA;
Buf1[3] = 0xECCAA4BE;
Buf1[4] = 0x66E6A4CA;
Buf1[5] = 0x72DCCABE;
Buf1[6] = 0x8ACA9C62;
Buf1[7] = 0xCEDC62A4;
Buf1[8] = 0x66A48EBE;
Buf1[9] = 0x80BE6EC2;
Buf1[10] = 0xDC6282CE;
v10 = 0xFA;
Buf2[0] = 0i64;
v12 = 0i64;
Buf2[1] = 0i64;
v13 = 0;
v14 = 0;
sub_140001010("%45s");
v3 = 0i64;
v4 = -1i64;
do
  ++v4;
while ( *((_BYTE *)Buf2 + v4) );
if ( v4 )
{
  do
  {
    *((_BYTE *)Buf2 + v3) = __ROL1__(*((_BYTE *)Buf2 + v3), 1);
    ++v3;
    v5 = -1i64;
    do
      ++v5;
    while ( *((_BYTE *)Buf2 + v5) );
  }
  while ( v3 < v5 );
}
v6 = memcmp(Buf1, Buf2, 0x2Dui64);
v7 = "WELCOME TO THE WORLD OF REVERSE ENGINEERING!!!";
if ( v6 )
  v7 = "PLEASE, DO NOT GIVE UP! GIVE IT ANOTHER TRY.";
```

内存比较

经过分析后，发现只进行了ROL1这一处的运算，用过的人可能会知道这是对参数1循环左移参数2个位数，不知道的话可以看汇编

```
0001123                nop     dword ptr [rax+00h]
0001127                nop     word ptr [rax+rax+00000000h]
0001130
0001130 loc_140001130:                         ; CODE XREF: main+EC↓
0001130                movzx   eax, byte ptr [rbp+rcx+57h+Buf2]
0001135                lea     rdx, [rbp+57h+Buf2]
0001139                rol     al, 1
000113B                mov     byte ptr [rbp+rcx+57h+Buf2], al
000113F                inc     rcx
0001142                mov     rax, 0FFFFFFFFFFFFFFFFh
```

```
38  if ( v4 )
39  {
40    do
41    {
42      *((_BYTE *)Buf2 + v3) = __ROL1__(*((_BYTE *)Buf2 + v3), 1);
43      ++v3;
44      v5 = -1i64;
45      do
46        ++v5;
47      while ( *(( BYTE *)Buf2 + v5) );
```

很清楚是循环左移1位

#因为都是偶数，可知最后一位都为0，所以可以不进行逆循环的操作

```
aa = [0xE8DAEAC6, 0xF6CCE8C6, 0xCA9680DA, 0xECCAA4BE,  0x66E6A4CA, 0x72DCCABE,
0x8ACA9C62, 0xCEDC62A4, 0x66A48EBE,
      0x80BE6EC2, 0xDC6282CE, 0xFA]
flag = ''
for a in aa:
    for i in range(4):
        if a == 0:
            continue
        tmp = a & 0xff
        a >>= 8
        tmp >>= 1
        flag += chr(tmp)

print(flag)
```

字节码

python字节码，分析得到

```
n = [-83, -96, -78, -21, -3, -17, 58, 31, 58]
    ff = input()
    c = ''
    with open("1", "r", encoding='utf-8') as f:
        s = f.read()
        f.close()
    with open("2", "rb") as f:
        b = f.read(9)
        f.close()
    for i in range(len(s)):
        tmp = ord(s[i]) ^ b[i]
        tmp += n[i]
        c += chr(tmp)
    print('Right! Please add cumtctf{}') if (c == ff) else print('Try again!')
```

逆向脚本

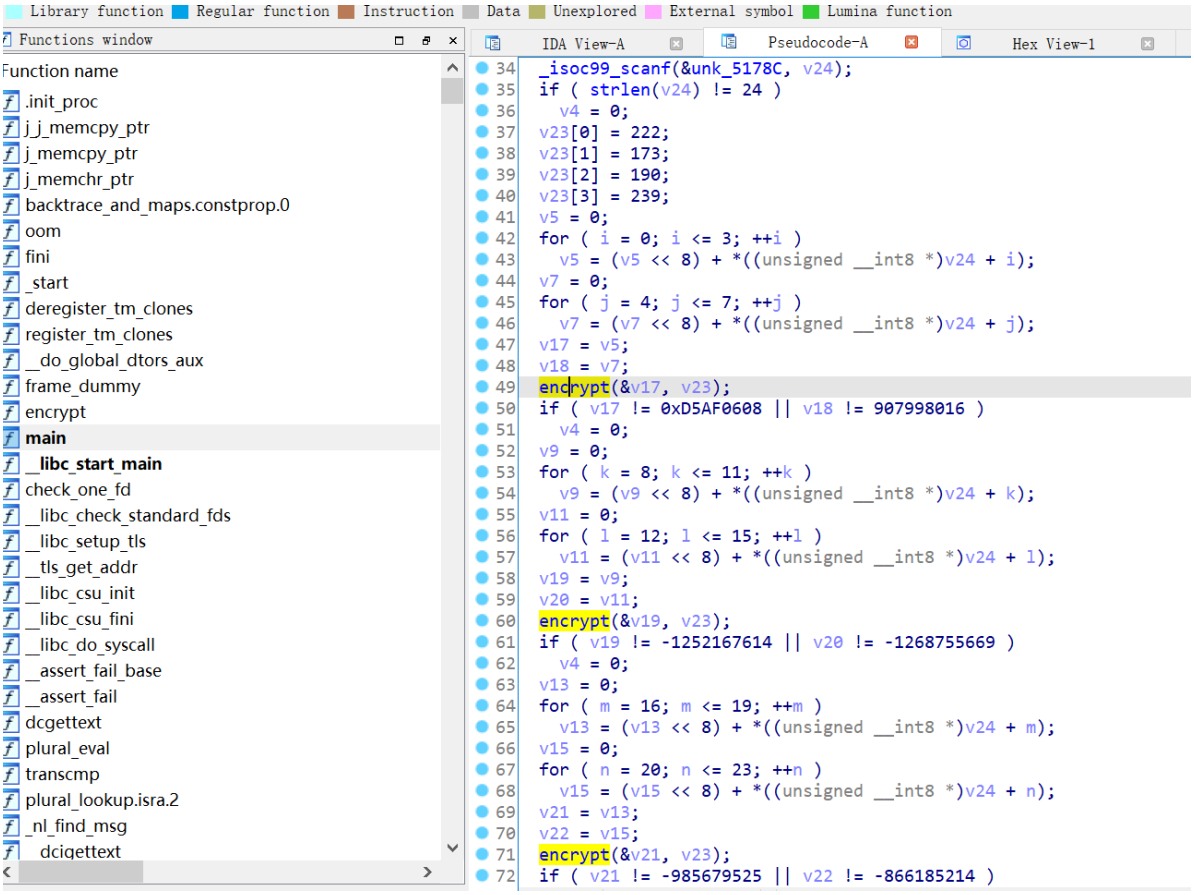
```
c = ''
n = [-83, -96, -78, -21, -3, -17, 58, 31, 58]
with open("1", "r", encoding='utf-8') as f:
    s = f.read()
    f.close()
with open("2", "rb") as f:
    b = f.read(9)
    f.close()
for i in range(len(s)):
    tmp = ord(s[i]) ^ b[i]
    tmp += n[i]
    c += chr(tmp)
print(c)
```

my cloth

文件拖入DIE工具，发现加了UPX壳

脱壳`upx -d {filename}`

脱下来的文件拖入ida



符号没去，函数原名很明显

这里有的人说没法跑起来，是因为这是arm架构的，想要运行，需要在arm的虚拟环境下进行，但是这题，也不用动调

```
unsigned int *__fastcall encrypt(unsigned int *result, _DWORD *a2)
{
  unsigned int v2; // [sp+Ch] [bp+Ch]
  unsigned int v3; // [sp+10h] [bp+10h]
  int v4; // [sp+14h] [bp+14h]
  unsigned int i; // [sp+18h] [bp+18h]

  v2 = *result;
  v3 = result[1];
  v4 = 0;
  for ( i = 0; i <= 0x3F; ++i )
  {
    v4 -= 559038737;
    v2 += (a2[1] + (v3 >> 5)) ^ (16 * v3 + *a2) ^ (v4 + v3);
    v3 += (a2[3] + (v2 >> 5)) ^ (16 * v2 + a2[2]) ^ (v4 + v2);
  }
  *result = v2;
  result[1] = v3;
  return result;
}
```

熟悉的人会知道这是tea加密算法

轮数64，delta=0xdeadbeef，找网上脚本改参进行tea解密即可

weak

代码中有少部分的花指令，其功能是给返回地址加上5，因此只需要将call与紧接其后的五个字节的指令即可。

后面的逻辑是检测有无行与列的重复，那么可以推断出来，是构造了一个简单的六阶数独游戏。

由于数独中给的已知数字较多，数独较为简单，可以直接解出。

```
"""
0 0 1 2 4 6
2 0 5 0 3 1
0 0 0 3 0 0
3 5 0 1 6 4
6 0 0 5 1 2
0 2 4 6 0 0
"""
```

数独大致如上。输入要求依次输入初始值为零的值，填入数独的数字即可得到答案。

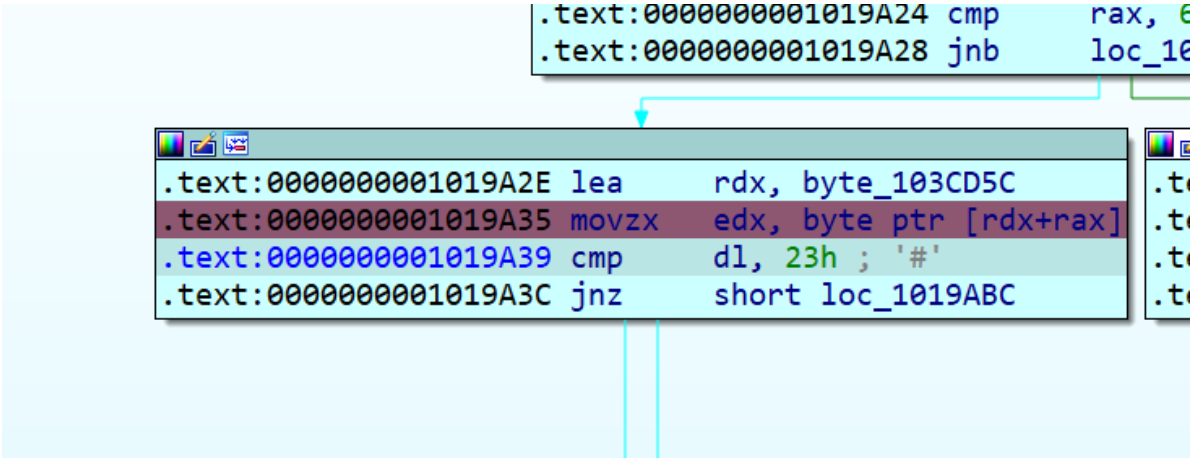
gogogo

go语言的迷宫题。降低难度考虑，本题保留了所有的方法名，Maze_advance的方法用于迷宫的前进操作，在此进行验证路径是否可行。

不过这里有个地方：

```
        v6[0] = &off_4D6200;
        fmt_Fprintln();
      }
    }
    if ( byte_4BCD5C[11] == 35 )
      fmt_Fprintln();
    fmt_Fprintln();
  }
}
```

很具有迷惑性。这里似乎是ida的反编译错误，误认为是值为11的静态值，但其实是一个动态的值



可以通过动调发现。输入值不同时，该处的rax会不一样。

接下来只需要分析Maze_advance函数

```
if ( v1 > 0x76u )
{
  if ( v1 == 'x' )
  {
    v7 = (*v0)--;
    v8 = 5LL * v0[1];
    v9 = v7 + 10LL * v0[1] - 1;
    v10 = v7 + 2 * v8;
    if ( v9 >= 0x64 )
      runtime_panicIndex();
    if ( byte_103CD5C[v10 - 1] == 49 )
      return 0LL;
  }
  else if ( v1 == 'z' )
  {
    v11 = (*v0)++;
    v12 = 5LL * v0[1];
    v13 = v11 + 10LL * v0[1] + 1;
    v14 = v11 + 2 * v12;
    if ( v13 >= 0x64 )
      runtime_panicIndex();
    if ( byte_103CD5C[v14 + 1] == 49 )
      return 0LL;
  }
}
else if ( v1 == 'c' )
{
  v2 = v0[1];
  v0[1] = v2 + 1;
  v3 = *v0 + 10 * v2;
  if ( (unsigned __int64)(v3 + 10) >= 0x64 )
    runtime_panicIndex();
  if ( byte_103CD5C[v3 + 10] == 49 )
    return 0LL;
}
else if ( v1 == 'v' )
{
  v5 = v0[1];
```

可以结合动调，发现vcxz分别对应着上下左右

这样，手动走一遍迷宫即可得到flag（当然，根据提示还需要进行md5的哈希运算）

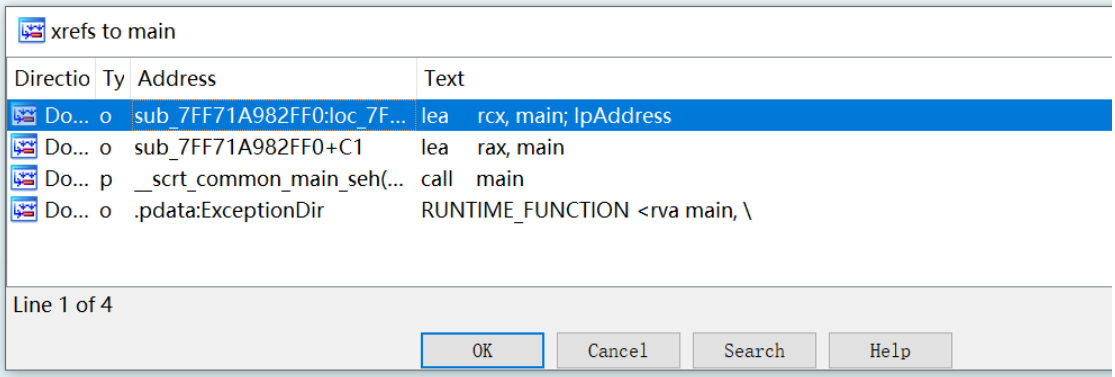
SimpleSMC

反调+花指令+SMC+blowfish

进入main是两个毫无意义的指令。其中iretq用于从中断中返回用户态地址，是一个特权指令，当然不会出现在用户态的程序中。因此，足以判断此题使用了smc技术，会动态的修改main函数的代码。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
main proc near
; __unwind { // __GSHandlerCheck
lea     ebp, cs:10F6DDF13h
iretq
main endp
```

经过调试，发现main并没有被修改，那么可能是做了一些对调试器的检测。首先尝试对main函数地址的xref操作。



可以看到有两个地方引用了这个地址。（题外话：写这题的时候，我在测试中由于花指令的存在，ida甚至都无法找到任何xref，这对此题加大了难度，不过可以通过import表查找敏感的api函数调用）

顺着这两个地方查找，可以看到一个函数，将花指令去掉（patch无效的字节），可以看到这样的函数

IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums | Imports

```
1 void sub_7FF71A982FF0()
2 {
3   ULONG v0; // [rsp+34h] [rbp+4h] BYREF
4   NTSTATUS v1; // [rsp+38h] [rbp+8h]
5   DWORD flOldProtect[2]; // [rsp+3Ch] [rbp+Ch] BYREF
6   __int64 ProcessInformation; // [rsp+48h] [rbp+18h] BYREF
7
8   v1 = NtQueryInformationProcess((HANDLE)0xFFFFFFFFFFFFFFFFi64, ProcessDebugObjectHandle, &ProcessInformation, 8u, &v0);
9   if ( v1 == 0xC0000353 || !ProcessInformation )
10  {
11    VirtualProtect(NtQueryInformationProcess, 0x10ui64, 0x20u, flOldProtect);
12    flOldProtect[1] = (unsigned int)NtQueryInformationProcess;
13    if ( (_DWORD)NtQueryInformationProcess == 0xB8D18B4C )
14      sub_7FF71A983110();
15  }
16 }
```

对其入口下断点，可以看出它在main函数开始运行之前就已经运行。

后面跟着的是三个反调试的技术。可以通过patch进行绕过。

(这三个技术分别是检测调试对象，检测hook，以及使用eflag检测调试器)

但这些函数与main似乎并没有关系，仔细检查，可以发现代码隐藏在了seh的处理函数里

```
1081 ; ---------------------------------------------------------------------------
1082 ; __unwind { // __C_specific_handler
1082 ;   __except(loc_1400030F4) // owned by 140003073
1082 word_140003082  dw 0B848h, 2 dup(0FFFFh)
1082                                         ; DATA XREF: .rdata:0000000140008E2C↓o
1082                                         ; .pdata:000000014000D03C↓o ...
1088 ; ---------------------------------------------------------------------------
1088                 jmp     short loc_140003092
1088 ; ---------------------------------------------------------------------------
108A                 dw 0FFFFh, 3348h, 0EBC0h
1090 ; ---------------------------------------------------------------------------
1090                 imul    eax
1092
1092 loc_140003092:                          ; CODE XREF: sub_140002FF0+98↑j
1092                 lea     rcx, main       ; lpAddress
1099                 lea     r9, [rbp+30h+flOldProtect] ; lpflOldProtect
109D                 mov     edx, 1F50h      ; dwSize
10A2                 mov     r8d, 40h ; '@'  ; flNewProtect
10A8                 call    cs:VirtualProtect
10AE                 mov     [rbp+30h+var_30], eax
10B1                 lea     rax, main
10B8                 mov     [rbp+30h+var_10], rax
10BC                 mov     [rbp+30h+var_1C], 0
10C3
```

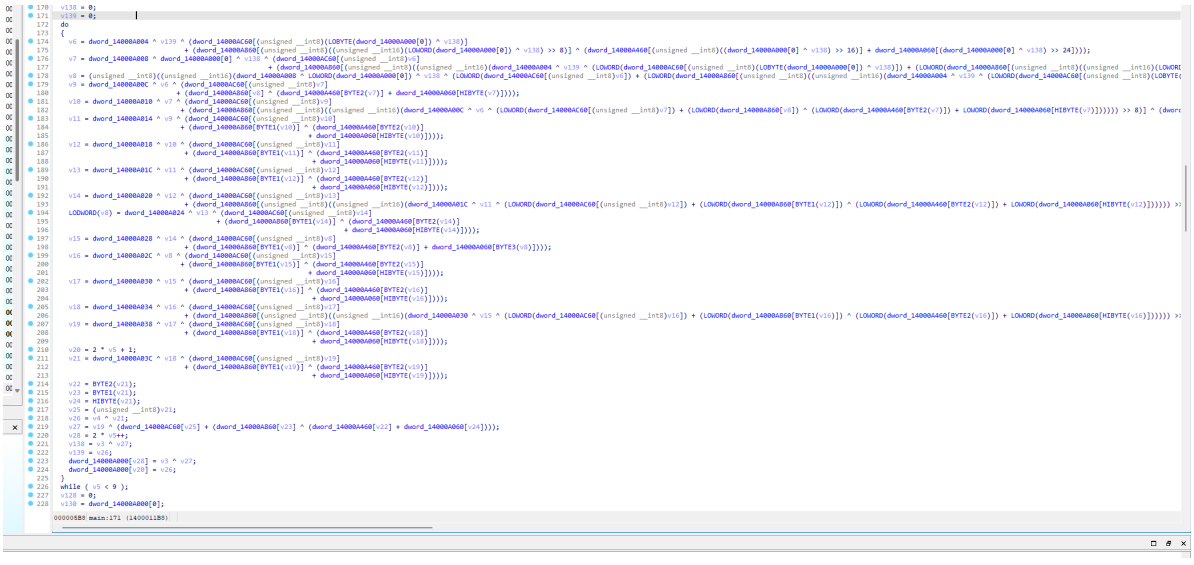
在此处，是真正的smc代码。（异常由前面的0000000140003110函数触发）

```
1110 var_8           = qword ptr -8
1110
1110                 pushfq
1111                 or      [rsp+8+var_8], 100h
1119                 popfq
111A                 retn
111A sub_140003110   endp
111A
```

通过分析汇编，可以看出大致算法为循环左移三位，再与0x3f进行异或

这样，可以写出脚本patch改变main的代码，再将前面整个函数nop掉，以方便调试。

解密出的部分main函数大致如下



使用了blowfish算法，不过由于msvc编译器优化的缘故，加密函数都是以内联的形式，而且较为混乱，分析有难度，因此也是直接给了提示。

大致逻辑，是将flag的第一个4字节与最后的4字节进行blowfish加密，再将flag的第二个4字节与倒数第二的4字节进行blowfish加密，以此类推。可以根据此写出解密的代码。s盒与p盒均与原始的blowfish不同，不过加密过程是一样的。

加解密代码大致如下（与wiki中的相同）

```
static uint32_t f(uint32_t x) {
    uint32_t h = S[0][x >> 24] + S[1][x >> 16 & 0xff];
    return (h ^ S[2][x >> 8 & 0xff]) + S[3][x & 0xff];
}

void blowfish_encrypt(uint32_t* L, uint32_t* R) {
    for (short r = 0; r < 16; r++) {
        *L = *L ^ P[r];
        *R = f(*L) ^ *R;
        swap(L, R);
    }
    swap(L, R);
    *R = *R ^ P[16];
    *L = *L ^ P[17];
}

void blowfish_decrypt(uint32_t* L, uint32_t* R) {
    for (short r = 17; r > 1; r--) {
        *L = *L ^ P[r];
        *R = f(*L) ^ *R;
        swap(L, R);
    }
    swap(L, R);
    *R = *R ^ P[1];
    *L = *L ^ P[0];
}

// ...
// initializing the P-array and S-boxes with values derived from pi; omitted in
the example (you can find them below)
// ...
void blowfish_init(const unsigned char* key, int key_len)
{
    /* initialize P box w/ key*/
    uint32_t k;
    for (short i = 0, p = 0; i < 18; i++) {
        k = 0x00;
        for (short j = 0; j < 4; j++) {
            k = (k << 8) | (uint8_t)key[p];
            p = (p + 1) % key_len;
        }
        P[i] ^= k;
    }

    /* blowfish key expansion (521 iterations) */
    uint32_t l = 0x00, r = 0x00;
    for (short i = 0; i < 18; i += 2) {
        blowfish_encrypt(&l, &r);
        P[i] = l;
        P[i + 1] = r;
    }
    for (short i = 0; i < 4; i++) {
        for (short j = 0; j < 256; j += 2) {
            blowfish_encrypt(&l, &r);
            S[i][j] = l;
            S[i][j + 1] = r;
        }
    }
```

```
}
```

此处的p盒与s盒被修改，需要从程序中转储。由于数据较多，这里只列举一小部分。

```
uint32_t P[18] = {
    0x190d1124, 0xf1d75a61, 0xdc405868, 0x8ef837c9, 0x08d588e3, 0x0985b9e8,
0x63f6b790, 0x51418fd6, 0x3e131760, 0x7a3d7317, 0x6d295e9f, 0xf48b4c9e,
0xcb8c6aa6, 0xfd40f0aa, 0xa9b562f3, 0x88f25293, 0xba9cddc8, 0xf4d4b55a
};


uint32_t S[4][256] =
{{
0xe133f0fc, 0x5bd41e14, 0x4ba6a473, 0x9f33d405,
//...
```

调用解密函数：

```
static char result[]{
0x38, 0xC9, 0x87, 0x71, 0xC1, 0x38, 0xE1, 0xCD, 0x8C, 0x6F,
0xBA, 0x3D, 0x2A, 0xD1, 0x68, 0x4E, 0xEE, 0x22, 0xFB, 0xA7,
0x49, 0x3F, 0xE7, 0x52, 0x85, 0x64, 0xE1, 0x81, 0xD7, 0x87,
0x3D, 0x75
};

// ...main
blowfish_decrypt((uint32_t*)(result + 28), (uint32_t*)result);
blowfish_decrypt((uint32_t*)(result + 24), (uint32_t*)(result + 4));
blowfish_decrypt((uint32_t*)(result + 20), (uint32_t*)(result + 8));
blowfish_decrypt((uint32_t*)(result + 16), (uint32_t*)(result + 12));
```

即可得到flag