

## SMC技术

SMC，即Self Modifying Code，动态代码加密技术，指通过修改代码或数据，阻止别人直接静态分析，然后在动态运行程序时对代码进行解密，达到程序正常运行的效果。

**VirtualProtect** 函数通常用于代码自加密的场景。代码自加密是一种保护代码不被轻易逆向分析的技术，通过在程序运行时动态地修改代码的内存保护属性，使得代码在执行时可以被修改和执行，但在不执行时则不能被读取或修改。

在Linux系统中，可以通过**mprotect**函数修改目标内存的权限  
在Windows系统中，**VirtualProtect**函数实现内存权限的修改  
因此也可以观察是否有这两个函数来判断是否进行了SMC

SMC一般有俩种破解方法，第一种是找到对代码或数据加密的函数后通过**idapython**写解密脚本。第二种是动态调试到SMC解密结束的地方dump出来。

## 堆栈平衡

// 堆栈不平衡问题出现原因

**one:** 一般是程序代码有一些干扰代码，让IDA的反汇编分析出现错误。比如用**push + n**条指令 + **retn**来实际跳转，而IDA会以为**retn**是函数要结束，结果它分析后发现调用栈不平衡，因此就提示**sp analysis failed**。

**two:** 还有一些比如编译器优化，因为**ida**是用**retn**指令来识别函数结束的，如果函数不是以这种方式结束，IDA就会分析为栈不平衡。也就是IDA找不到函数结束的位置。

// 堆栈平衡原理

### 1、概念解释

函数返回时，堆栈需要恢复到调用前的状态

### 2、平衡的重要性

避免内存访问错误和程序崩溃

保证程序的正常执行逻辑

### 3、平栈的方式

外平栈（**cdecl**）

在 **cdecl** 调用约定中，参数也是从右往左一次压入堆栈，但堆栈平衡由调用者在函数返回后负责清理

内平栈（**stdcall**）

`stdcall` 是内平栈，既有被调用函数在返回前负责清理堆栈，通过调整栈指针来移除压入栈中的参数

## 下面简单举个例子，让大家更深刻的了解堆栈平衡的概念

### [汇编中的函数调用中栈的工作过程](#)

```
// 下面以汇编的角度来解释函数调用的过程，这里就以上面视频中的例子为例
A_Func(5,6); // 这里main函数要调用A_Func函数，参数为5和6，此时假设esp的值为0x1000
push 6 // 参数先入栈，顺序为从右往左
push 5 // 此时 esp 因为加入了两个int类型参数，所以变成了0x1000-8
call A_Func // call指令做的操作相当于把当前的eip值压入栈中（也就是call指令的下一条指令地址），并跳转到A_Func函的入口地址
add esp, 8 // 这一步做的就是平栈操作，把刚才压入栈中的参数弹出，恢复esp的原值，此时esp的值为0x1000

// 以上就是堆栈平衡的过程，简单来说就是调用函数前后堆栈的状态要保持一致。
// 在调用函数时，会先将参数从右到左依次压入栈中，然后 call 跳转到被调用函数（压入call指令的
//下一个指令的地址），在被调用函数中进行 push ebp; 等操作。我们可以 F9 步过call指令，会发
// 现 ESP又恢复了参数压入之前的数值，这就是堆栈平衡。
```

## 网鼎杯2020jocker



```
// 利用Exeinfo查看文件的PE信息，可以知道文件是 32为无壳程序，利用IDA Pro打开
Shift+F12 进入字符串窗口 -> 看到关键字"please input you flag:", 双击进入 ->
Ctrl+x查看交叉引用 -> 从而定位到main函数
// 利用IDA打开，出现 positive sp value has been detected, the output may be
wrong!
```

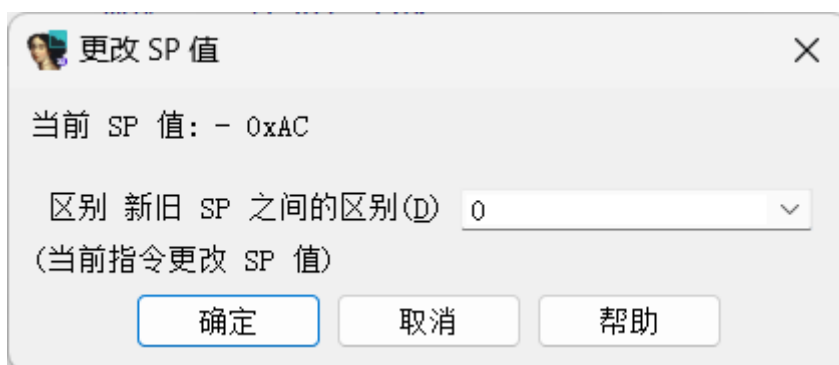
// 出现了栈不平衡的问题，导致ida无法生成代码。

选项 -> 常规 -> 堆栈指针（查看栈帧变换）

```
.text:0040180A 0AC      add     eax, offset __Z7encryptPc ; encrypt(char *)
.text:0040180F 0AC      mov     edx, [ebp+var_C]
.text:00401812 0AC      add     edx, offset __Z7encryptPc ; encrypt(char *)
.text:00401818 0AC      movzx   edx, byte ptr [edx]
.text:0040181B 0AC      xor     edx, 41h
.text:0040181E 0AC      mov     [eax], dl
.text:00401820 0AC      add     [ebp+var_C], 1
.text:00401824      loc_401824:                                ; CODE XREF: _main+DF↑j
.text:00401824 0AC      cmp     [ebp+var_C], 0BAh
.text:0040182B 0AC      jle     short loc_401807
.text:0040182D 0AC      lea     eax, [ebp+Destination]
.text:00401830 0AC      mov     [esp], eax
.text:00401833 0AC      call    near ptr __Z7encryptPc ; encrypt(char *)
.text:00401838 -485C    test    eax, eax
.text:0040183A -485C    setnz   al
.text:0040183D -485C    test    al, al
.text:0040183F -485C    jz      short loc_40184C
.text:00401841 -485C    lea     eax, [ebp+Destination]
.text:00401844 -485C    mov     [esp], eax ; char *
.text:00401847 -485C    call    __Z7finallyPc ; finally(char *)
.text:0040184C      loc_40184C:                                ; CODE XREF: _main+119↑j
.text:0040184C -B209    mov     eax, 0
.text:00401851 -B209    mov     ecx, [ebp+var_4]
.text:00401854 -B209    leave
.text:00401855 000      lea     esp, [ecx-4]
.text:00401858 008      retn
.text:00401858      main      endp ; sp-analysis failed
.text:00401858      ; -----
.text:00401859      align 10h
00000C2B: 0040182B: _main+105
```

可以看到，箭头指的两个地方栈偏移都出了问题，应该在call完之后都会平栈，也就是应该都是0AC才对，

我们有快捷键 alt+k 将偏移改为0



然后F5返回main函数可以发现报错没了

```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    char Str[50]; // [esp+12h] [ebp-96h] BYREF
    char Destination[80]; // [esp+44h] [ebp-64h] BYREF
    DWORD flOldProtect; // [esp+94h] [ebp-14h] BYREF
    size_t v7; // [esp+98h] [ebp-10h]
    int i; // [esp+9Ch] [ebp-Ch]

    __main();
    puts("please input you flag:");
    if ( !VirtualProtect(encrypt, 0xC8u, 4u, &flOldProtect) )
        exit(1);
    scanf("%40s", Str);
    v7 = strlen(Str);
    if ( v7 != 24 )
    {
        puts("Wrong!");
        exit(0);
    }
    strcpy(Destination, Str);
    wrong(Str);
    omg(Str);
    for ( i = 0; i <= 186; ++i )
        *((_BYTE *)encrypt + i) ^= 0x41u;
    if ( encrypt(Destination) )
        finally(Destination);
    return 0;
}

```

00000C47 | \_\_main:26 (401847) |

先简单分析代码，可以知道输入字符串长度为24，这里还有 `VirtualProtect` 函数，可以猜测用到了SMC技术，

加密了`encrypt()`函数，并在倒数第6行动态解密`encrypt()`函数。中间还有一个将输入的`Str`字符串赋值给了

`Destination`变量。

然后就是，`Str`经过`wrong()`函数和`omg()`函数，接下来我们进入看看，具体做了那些操作。

```
char *__cdecl wrong(char *a1)
{
    char *result; // eax
    int i; // [esp+Ch] [ebp-4h]

    for ( i = 0; i <= 23; ++i )
    {
        result = &a1[i];
        if ( (i & 1) != 0 )
            a1[i] -= i;
        else
            a1[i] ^= i;
    }
    return result;
}

int __cdecl omg(char *a1)
{
    int v2[24]; // [esp+18h] [ebp-80h] BYREF
    int i; // [esp+78h] [ebp-20h]
    int v4; // [esp+7Ch] [ebp-1Ch]

    v4 = 1;
    qmemcpy(v2, &unk_4030C0, sizeof(v2));
    for ( i = 0; i <= 23; ++i )
    {
        if ( a1[i] != v2[i] )
            v4 = 0;
    }
    if ( v4 == 1 )
        return puts("hahahaha_do_you_find_me?");
    else
        return puts("wrong ~~ But seems a little program");
}
```

00000AC7 \_\_Z5wrongPc:1 (4016C7) | 00000A89 \_\_Z3omgPc:13 (401689)

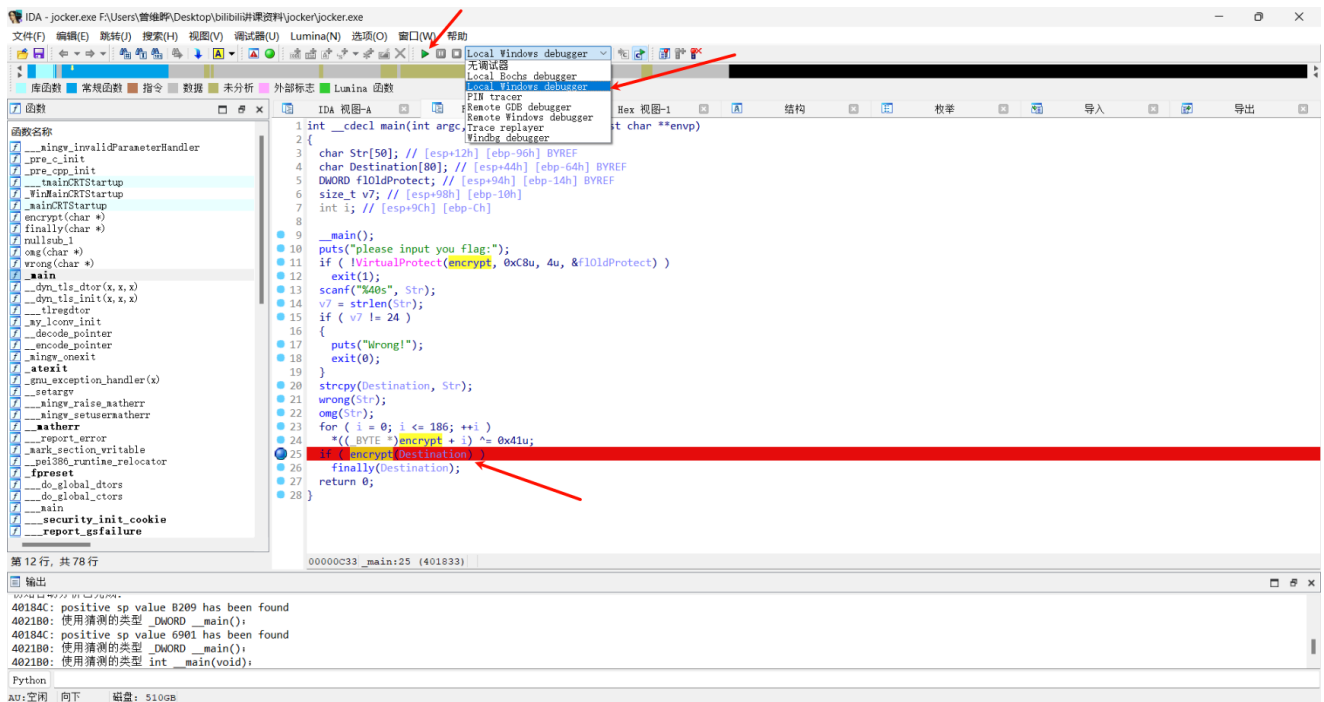
首先，**wrong()**函数的主要操作就是，遍历输入的**Str**字符串，若下标**i**为奇数，该元素就与下标做减法。相反若是偶数，则该元素与下标做异或。

最后**omg()**函数就是比较，经过处理后的**Str**字符串与**&unk\_4030c0**的字符串进行比较，若相同则输入正确，反之错误。

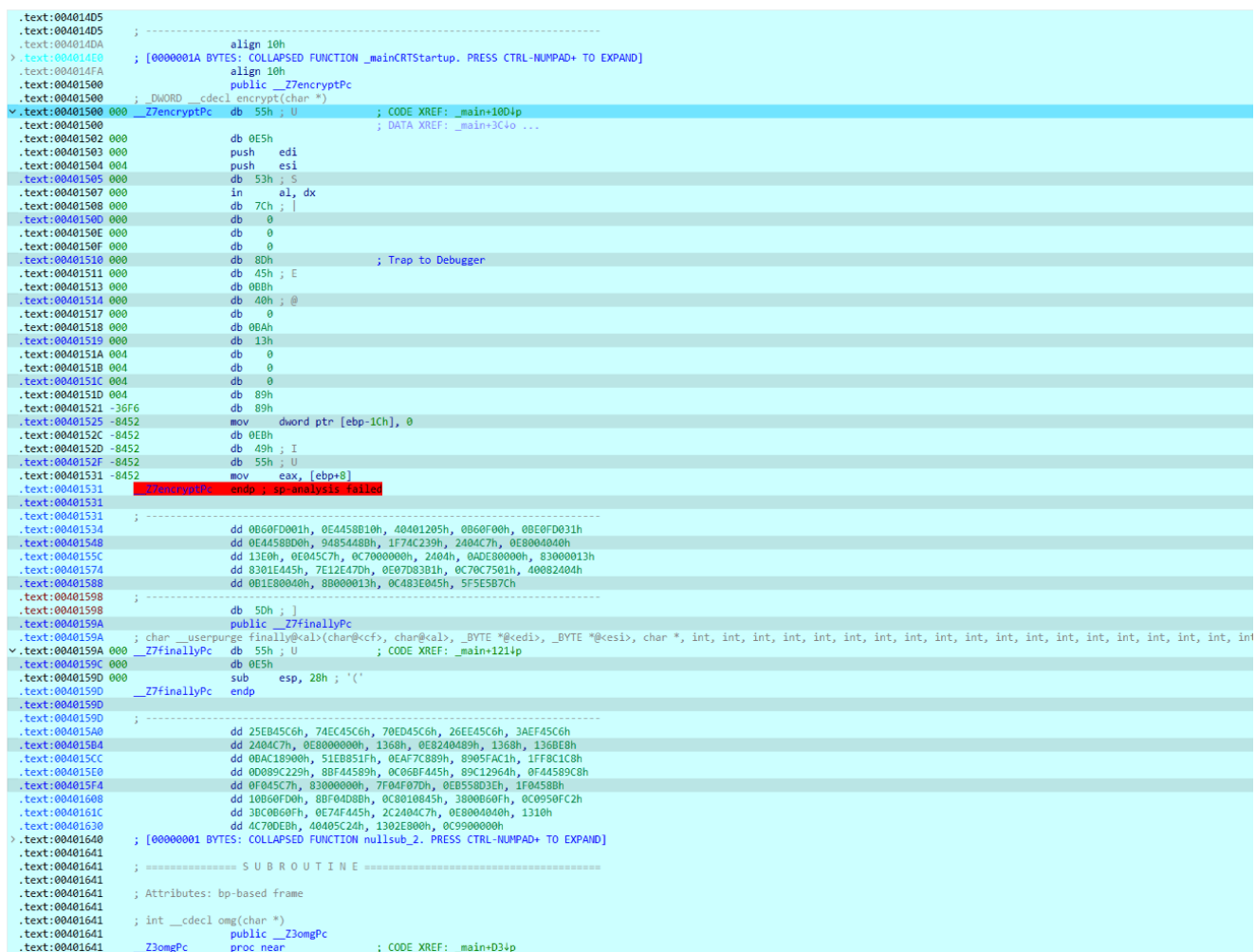
但是得出来的**flag**是错误的，真正的加密在**encrypt()**函数中，我们需要找到这个函数。但是**encrypt()**函数

由于堆栈不平衡反汇编不出来，且是用到**SMC**技术的，所以我们这里用动态调试的方式来解决这些问题。

## 正片开始



根据上图下好断点后，进入动态调试之后，随便输入24位字符串aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa。点击上方的视图 -> 选择反汇编窗口 -> 然后按 F7 单步执行进入encrypt()函数。可以看到红线下面还有一段代码它不属于任何函数中，应该就是ida识别错误。



点击\_\_Z7encryptPc 先用U将其设为无定义

```
>.text:004014C9 ; [0000001A BYTES: COLLAPSED FUNCTION _mainCRTStartup. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:004014FA align 10h
.text:00401500 public __Z7encryptPc
.text:00401500 ; DuORD cdecl encrypt(char *)
.text:00401500 __Z7encryptPc db 55h ; U ; CODE XREF: _main+1004ip
.text:00401500 ; DATA XREF: _main+3C40 ...
.text:00401501 db 89h
.text:00401502 db 0E5h
.text:00401503 db 57h ; W
.text:00401504 db 56h ; V
.text:00401505 db 53h ; S
.text:00401506 db 83h
.text:00401507 db 0ECh
.text:00401508 db 7Ch ; J
.text:00401509 db 0C7h
.text:0040150A db 45h ; E
.text:0040150B db 0E0h
.text:0040150C db 1
.text:0040150D db 0
.text:0040150E db 0
.text:0040150F db 0
.text:00401510 db 8Dh
.text:00401511 db 45h ; E
.text:00401512 db 94h
.text:00401513 db 0BBh
.text:00401514 db 40h ; @
.text:00401515 db 30h ; 0
.text:00401516 db 40h ; @
.text:00401517 db 0
.text:00401518 db 0BAh
.text:00401519 db 13h
.text:0040151A db 0
.text:0040151B db 0
.text:0040151C db 0
.text:0040151D db 89h
.text:0040151E db 0C7h
.text:0040151F db 89h
.text:00401520 db 0ECh
.text:00401521 db 89h
.text:00401522 db 001h
.text:00401523 db 0F3h
.text:00401524 db 0A5h
.text:00401525 db 0C7h
.text:00401526 db 45h ; E
.text:00401527 db 0E4h
.text:00401528 db 0
.text:00401529 db 0
.text:0040152A db 0
.text:0040152B db 0
.text:0040152C db 0EBh
.text:0040152D db 49h ; I
.text:0040152E db 8Bh
.text:0040152F db 55h ; U
.text:00401530 db 0E4h
.text:00401531 db 8Bh
.text:00401532 db 45h ; E
.text:00401533 db 8
.text:00401534 dd 0B60FD001h, 0E4458B10h, 40401205h, 0B60F00h, 0BE0FD031h
.text:00401535 dd 0E4458B00h, 94854488h, 1F74C239h, 2404C7h, 0E8004040h
.text:00401536 dd 13E8h, 0E045C7h, 0C7000000h, 2484h, 0ADE80000h, 83000013h
.text:00401537 dd 8301E445h, 7E12E47Dh, 0E07D0381h, 0C70C7501h, 4082404h
.text:00401538 dd 0B1E80040h, 8B000013h, 0C483E045h, 5F5E5B7Ch
.text:00401539 ; -----
.text:00401539 db 5Dh ; J
.text:00401539 public __Z7finallyPc
.text:00401539 ; char __userpurge finally@cal>(char@cal>, char@cal>, _BYTE *@cedi>, _BYTE *@cesis>, char *, int, int, int, int, int, int, int, int, int, int, int, int, int, int, int, int)
.text:00401539 000 __Z7finallyPc db 55h ; U ; CODE XREF: _main+1214ip
.text:00401539 000 db 0E5h
.text:00401539 000 sub esp, 28h ; '('
.text:00401539 __Z7finallyPc endp
.text:00401539 ; -----
.text:00401539 ; -----
.text:00401539 dd 25EB45C6h, 74EC45C6h, 70ED45C6h, 26EE45C6h, 3AEF45C6h
.text:0040153A dd 2404C7h, 0E8000000h, 1368h, 0E8240489h, 1368h, 136BE8h
.text:0040153B dd 0BAC18900h, 51EB851Fh, 0EAF7C889h, 8905FAC1h, 1FF8C1C8h
.text:0040153C dd 00089C229h, 8BF44589h, 0C068F445h, 89C12964h, 0F44589C8h
.text:0040153D dd 0F045C7h, 83000000h, 7F04F07Dh, 0EB558D3Eh, 1F0458Bh
.text:0040153E dd 10B60F00h, 8BF04D0Bh, 0C8010B45h, 3B00060Fh, 0C0950FC2h
.text:0040153F dd 3BC0B60Fh, 0E74F445h, 2C2A04C7h, 0E8004040h, 1310h
.text:00401540 dd 4C70DEBh, 40405C24h, 1302E800h, 0C9900000h
.text:00401640 ; [00000001 BYTES: COLLAPSED FUNCTION nullsub_2. PRESS CTRL-NUMPAD+ TO EXPAND]
.text:00401641 ; ===== SUBROUTINE =====
.text:00401641 ; Attributes: bp-based frame
.text:00401641 ; int __cdecl omg(char *)
.text:00401641 public __Z3omgPc
```

再按 **c** 声明为代码段（对黄色线条之内的数据），点击 **F(强制) + yes**，最后返回点击 **\_\_Z7encryptPc** 用 **P** 将其设置为函数，然后 **F5 / Tab** 键来到反汇编界面，**encrypted**函数就被修复了，最后就是 **finally**函数，直接对 **loc\_40159A**按 **P** 即可

//sub\_40159A:

含义: **sub\_40159A** 通常表示一个完整的函数（**subroutine**），位于地址 **0x40159A**。

命名习惯: **sub\_** 前缀加上地址的形式通常用于自动命名 **IDA** 识别的函数。例如 **sub\_40159A** 表示 **IDA** 认为从

**0x40159A** 地址开始是一段独立的代码，符合函数的结构。

//loc\_40159A:

含义: **loc\_40159A** 表示一个代码标签（**location**），通常用于跳转或引用的位置，位于地址 **0x40159A**。



命名习惯：**loc\_** 前缀加上地址表示 IDA 识别的某个特定位置，作为代码段内的标签或局部跳转目标。

```
.text:00401599      retn
.text:00401599      __Z7encryptPc      endp
.text:00401599 ;-----
.text:0040159A      ; _DWORD __cdecl loc_40159A(char *)
.text:0040159A      loc_40159A:                                ; CODE XREF: __main+1214p
.text:0040159A      push    ebp
.text:0040159A      mov     ebp, esp
.text:0040159A      sub     esp, 20h
.text:004015A0      mov     byte ptr [ebp-15h], 25h ; '%'
.text:004015A4      mov     byte ptr [ebp-14h], 74h ; 't'
.text:004015A8      mov     byte ptr [ebp-13h], 70h ; 'p'
.text:004015AC      mov     byte ptr [ebp-12h], 26h ; '&'
.text:004015B0      mov     byte ptr [ebp-11h], 3Ah ; ':'
.text:004015B4      mov     dword ptr [esp], 0
.text:004015B8      call    _time
.text:004015C0      mov     [esp], eax
.text:004015C3      call    _srand
.text:004015C8      call    _rand
.text:004015CD      mov     ecx, eax
.text:004015CF      mov     edx, 51E0851Fh
.text:004015D4      mov     eax, ecx
.text:004015D6      imul    edx
.text:004015D8      sar     edx, 5
.text:004015DB      mov     eax, ecx
.text:004015DD      sar     eax, 1Fh
.text:004015E0      sub     edx, eax
.text:004015E2      mov     eax, edx
.text:004015E4      mov     [ebp-0Ch], eax
.text:004015E7      mov     eax, [ebp-0Ch]
.text:004015EA      imul    eax, 64h ; 'd'
.text:004015ED      sub     ecx, eax
.text:004015EF      mov     eax, ecx
.text:004015F1      mov     [ebp-9Ch], eax
.text:004015F4      mov     dword ptr [ebp-10h], 0
.text:004015F8      cmp     dword ptr [ebp-10h], 4
.text:004015FF      jg      short locret_40163F
.text:00401601      lea     edx, [ebp-15h]
.text:00401604      mov     eax, [ebp-10h]
.text:00401607      add     eax, edx
.text:00401609      movzx   edx, byte ptr [eax]
.text:0040160C      mov     ecx, [ebp-10h]
.text:0040160F      mov     eax, [ebp+8]
.text:00401612      add     eax, ecx
.text:00401614      movzx   eax, byte ptr [eax]
.text:00401617      cmp     dl, al
.text:00401619      setnz   al
.text:0040161C      movzx   eax, al
.text:0040161F      cmp     eax, [ebp-0Ch]
.text:00401622      jz      short loc_401632
.text:00401624      mov     dword ptr [esp], offset aIHideTheLastPa ; "I hide the last part, you will not succ"...
.text:00401628      call    _puts
.text:00401630      jmp     short locret_40163F
.text:00401632 ;-----
.text:00401632      loc_401632:                                ; CODE XREF: .text:00401622j
.text:00401632      mov     dword ptr [esp], offset aReallyDidYouFi ; "Really??? Did you find it?OMG!!!"
.text:00401639      call    _puts
.text:0040163E      nop
.text:0040163F      locret_40163F:                            ; CODE XREF: .text:004015FFtj
.text:0040163F      leave   ; .text:00401630tj
.text:00401640      retn
.text:00401641 ;-----
.text:00401641      ;===== S U B R O U T I N E =====
.text:00401641      ; Attributes: bp-based frame
.text:00401641      ; _DWORD __cdecl ong(char *)
.text:00401641      public __Z3ongPc
.text:00401641      __Z3ongPc      proc near                ; CODE XREF: __main+D34p
```

完成以上步骤，**encrypt()**函数和**finally()**函数就被修复了。函数代码如下：

```
int __cdecl encrypt(char *a1)
{
    int v2[19]; // [esp+1Ch] [ebp-6Ch] BYREF
    int v3; // [esp+68h] [ebp-20h]
    int i; // [esp+6Ch] [ebp-1Ch]

    v3 = 1;
    qmemcpy(v2, &unk_403040, sizeof(v2));
    for ( i = 0; i <= 18; ++i )
    {
        if ( (a1[i] ^ Buffer[i]) != v2[i] )
        {
            puts("wrong ~");
            v3 = 0;
            exit(0);
        }
    }
    puts("come here");
    return v3;
}

1 int __cdecl sub_40159A(char *a1)
2 {
3     unsigned int v1; // eax
4     char v3[9]; // [esp+13h] [ebp-15h] BYREF
5     int v4; // [esp+1Ch] [ebp-Ch]
6
7     strcpy(v3, "%tp&:");
8     v1 = time(0);
9     srand(v1);
10    v4 = rand() % 100;
11    v3[6] = 0;
12    *&v3[7] = 0;
13    if ( (v3[v3[5]] != a1[v3[5]]) == v4 )
14        return puts("Really??? Did you find it?OMG!!!");
15    else
16        return puts("I hide the last part, you will not succeed!!!");
17 }
```

00000900 \_\_Z7encryptPc:1 (401500)

0000099A sub\_40159A:1 (40159A)



**encrypt()**函数:

可以看到就是将输入字符串的前19位与Buffer数组进行一对一异或，异或后的结果与v2数组一对一比较。

也就是说v2数组就是加密后的前19位字符串。

**finally()**函数:

不知道他在干嘛，但是根据网上资料知道，输入字符串后面5位应该是与某一个随机数进行异或得到v3。

# 解密脚本

```
hh = 'hahahaha_do_you_find_me?'
```

```
v2 = [0x0E, 0x0D, 0x9, 0x6, 0x13, 0x5, 0x58, 0x56, 0x3E, 0x6, 0x0C, 0x3C,  
0x1F, 0x57, 0x14, 0x6B, 0x57, 0x59, 0x0D]
```

```
flag = []
```

```
for i in range(19):  
    flag.append(chr(v2[i] ^ ord(hh[i])))
```

```
v3 = [37, 116, 112, 38, 58]
```

```
key = ord('}') ^ 58
```

```
for i in range(5):  
    flag.append(chr(v3[i] ^ key))
```

```
print(''.join(flag))
```

```
# flag{d07abccf8a410cb37a}
```