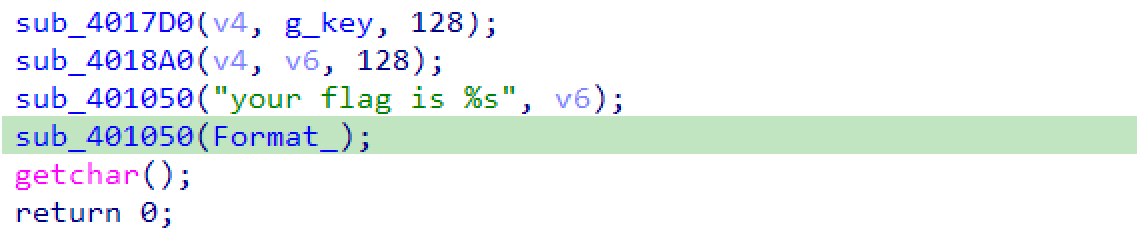
RE题 Chaos2

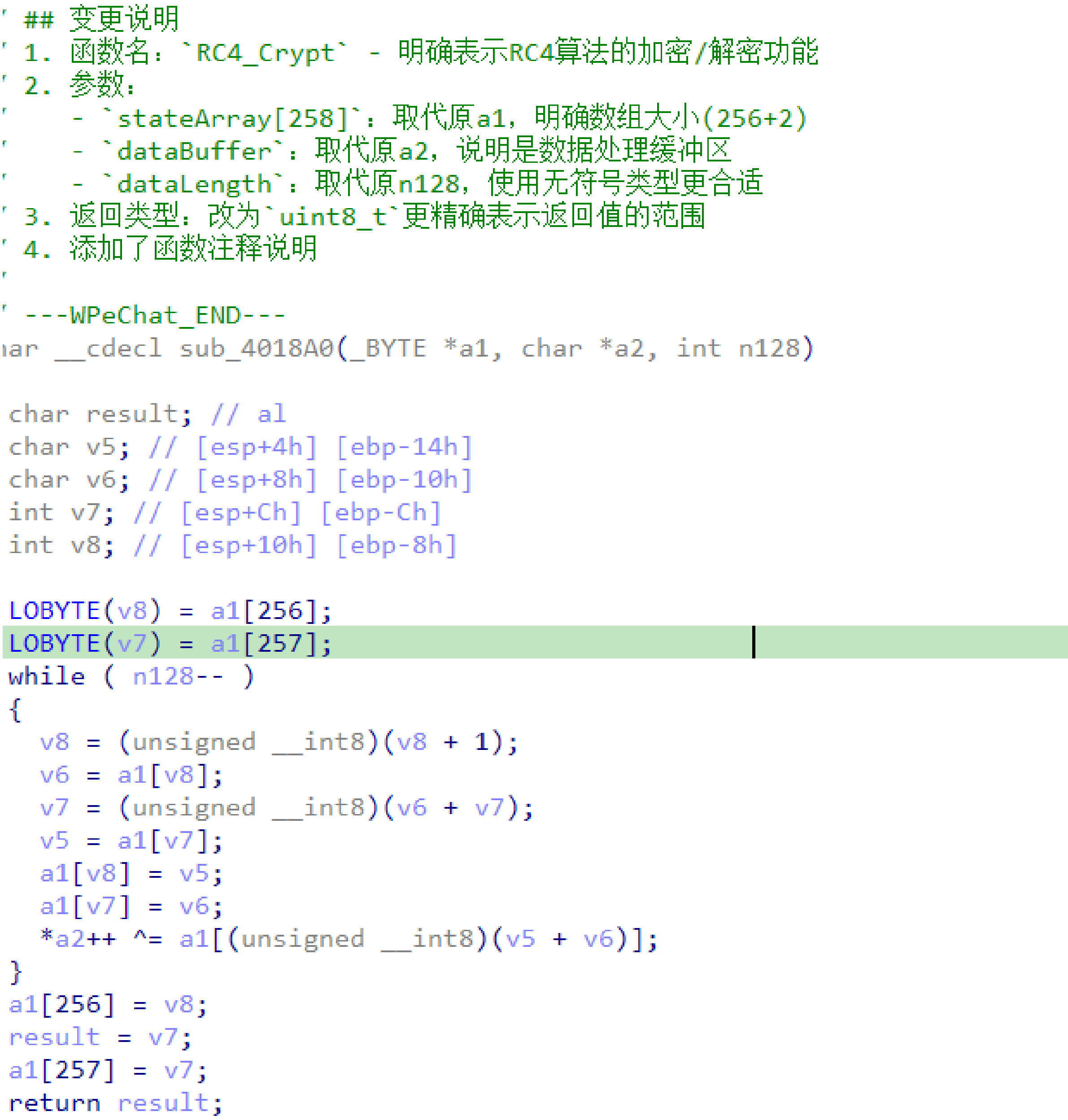
解题思路:

IDA打开main函数里面有花指令的,发现都是一些无意义的跳转使用Nop大法即可,之后就几个函数,下面是关键函数.



```
  sub_4017D0(v4, g_key, 128);
  sub_4018A0(v4, v6, 128);
  sub_401050("your flag is %s", v6);
  sub_401050(Format_);
  getchar();
  return 0;
```

使用AI插件分析一下:

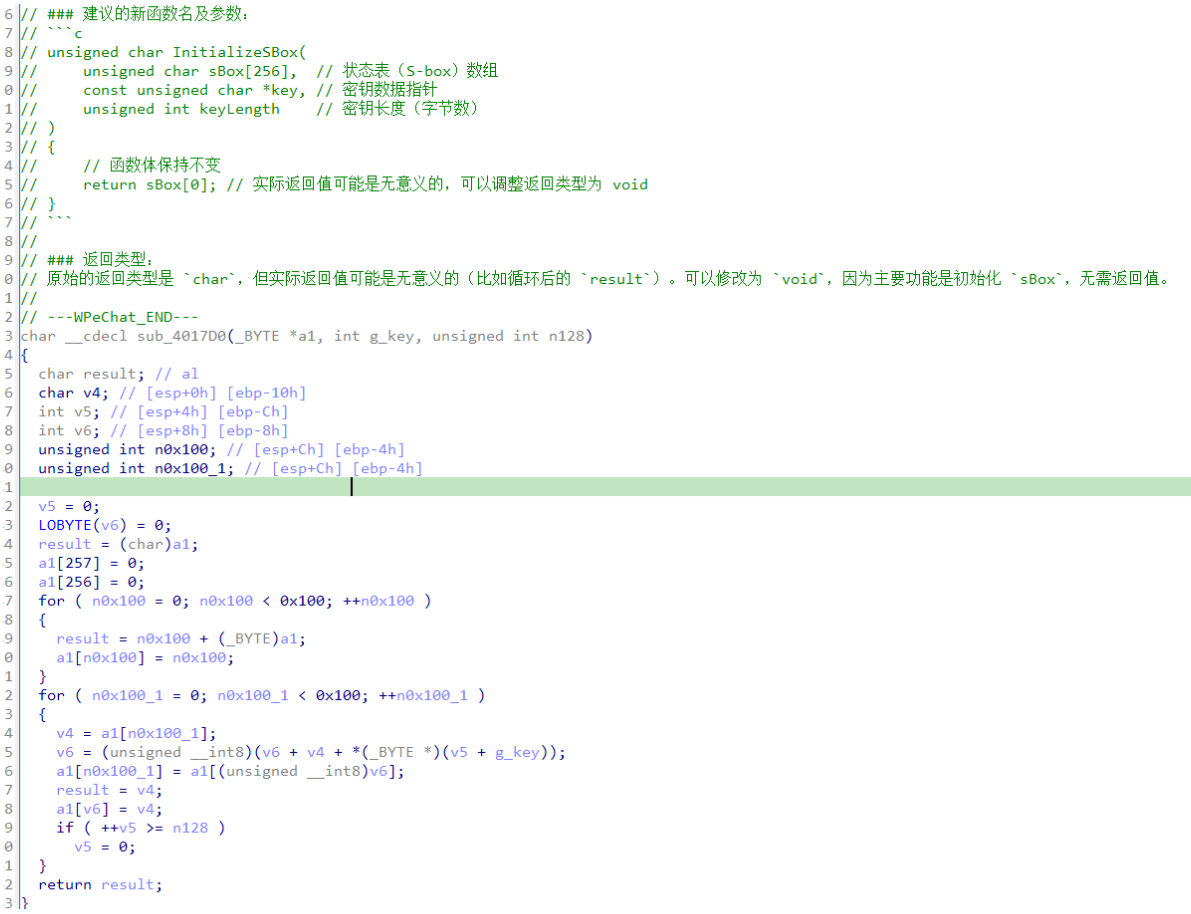


```
' ## 变更说明
' 1. 函数名：`RC4_Crypt` - 明确表示RC4算法的加密/解密功能
' 2. 参数：
'    - `stateArray[258]`：取代原a1，明确数组大小(256+2)
'    - `dataBuffer`：取代原a2，说明是数据处理缓冲区
'    - `dataLength`：取代原n128，使用无符号类型更合适
' 3. 返回类型：改为`uint8_t`更精确表示返回值的范围
' 4. 添加了函数注释说明
'
' ---WPeChat_END---
ıar __cdecl sub_4018A0(_BYTE *a1, char *a2, int n128)

 char result; // al
 char v5; // [esp+4h] [ebp-14h]
 char v6; // [esp+8h] [ebp-10h]
 int v7; // [esp+Ch] [ebp-Ch]
 int v8; // [esp+10h] [ebp-8h]

 LOBYTE(v8) = a1[256];
 LOBYTE(v7) = a1[257];
 while ( n128-- )
 {
   v8 = (unsigned __int8)(v8 + 1);
   v6 = a1[v8];
   v7 = (unsigned __int8)(v6 + v7);
   v5 = a1[v7];
   a1[v8] = v5;
   a1[v7] = v6;
   *a2++ ^= a1[(unsigned __int8)(v5 + v6)];
 }
 a1[256] = v8;
 result = v7;
 a1[257] = v7;
 return result;
```

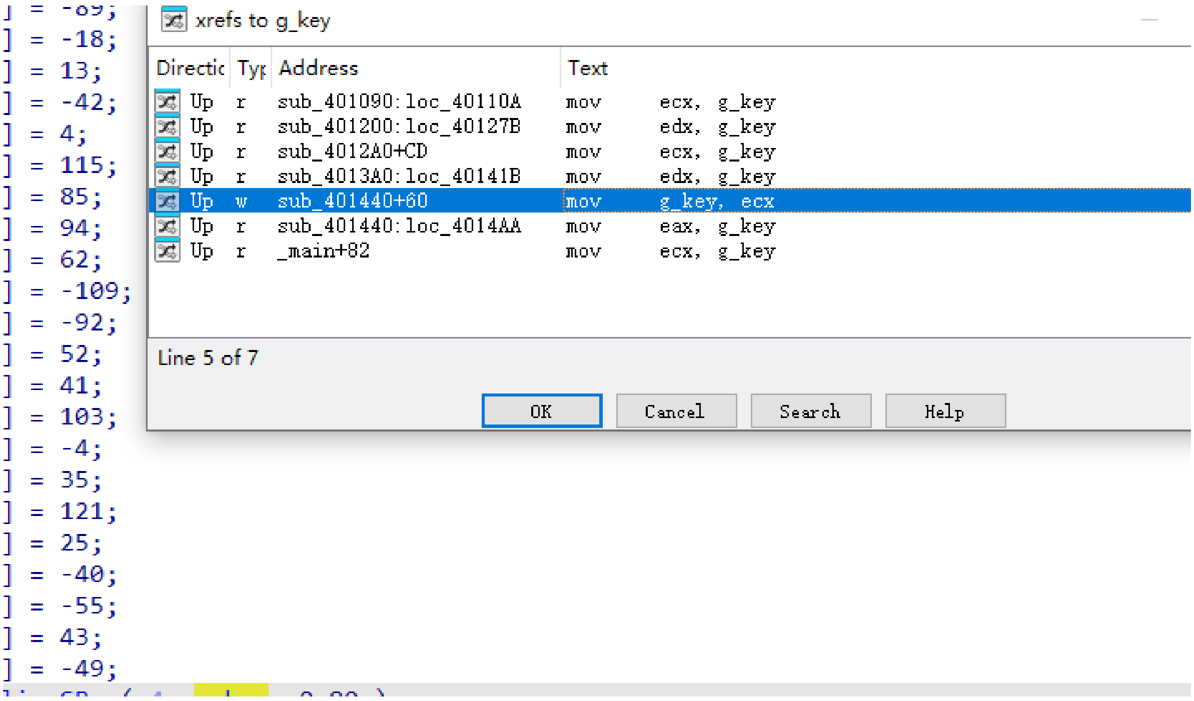
发现是RC4加密算,再看其他几个函数.



```
// ### 建议的新函数名及参数：
// ```c
// unsigned char InitializeSBox(
//     unsigned char sBox[256],  // 状态表（S-box）数组
//     const unsigned char *key, // 密钥数据指针
//     unsigned int keyLength    // 密钥长度（字节数）
// )
// {
//     // 函数体保持不变
//     return sBox[0]; // 实际返回值可能是无意义的，可以调整返回类型为 void
// }
// ```
//
// ### 返回类型：
// 原始的返回类型是 `char`，但实际返回值可能是无意义的（比如循环后的 `result`）。可以修改为 `void`，因为主要功能是初始化 `sBox`，无需返回值。
//
// ---WPeChat_END---
char __cdecl sub_4017D0(_BYTE *a1, int g_key, unsigned int n128)
{
  char result; // al
  char v4; // [esp+0h] [ebp-10h]
  int v5; // [esp+4h] [ebp-Ch]
  int v6; // [esp+8h] [ebp-8h]
  unsigned int n0x100; // [esp+Ch] [ebp-4h]
  unsigned int n0x100_1; // [esp+Ch] [ebp-4h]

  v5 = 0;
  LOBYTE(v6) = 0;
  result = (char)a1;
  a1[257] = 0;
  a1[256] = 0;
  for ( n0x100 = 0; n0x100 < 0x100; ++n0x100 )
  {
    result = n0x100 + (_BYTE)a1;
    a1[n0x100] = n0x100;
  }
  for ( n0x100_1 = 0; n0x100_1 < 0x100; ++n0x100_1 )
  {
    v4 = a1[n0x100_1];
    v6 = (unsigned __int8)(v6 + v4 + *(_BYTE *)(v5 + g_key));
    a1[n0x100_1] = a1[(unsigned __int8)v6];
    result = v4;
    a1[v6] = v4;
    if ( ++v5 >= n128 )
      v5 = 0;
  }
  return result;
}
```

这个就是初始化密钥了,那剩下的找到Key就行了.



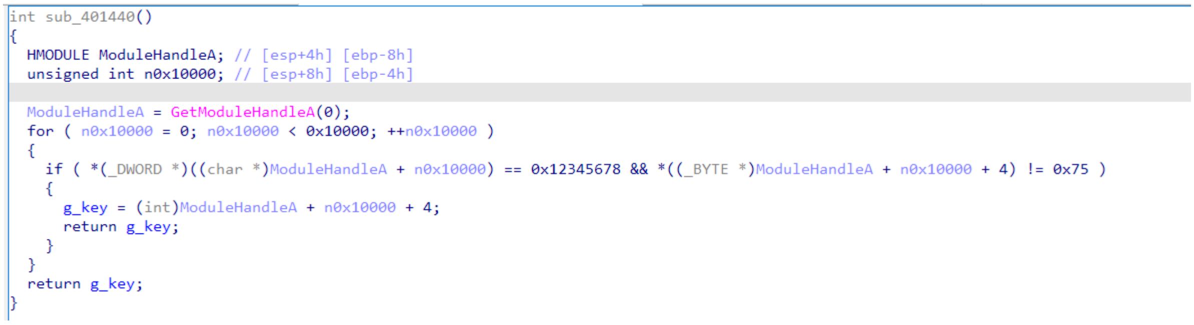
```
] = -69;
] = -18;
] = 13;
] = -42;
] = 4;
] = 115;
] = 85;
] = 94;
] = 62;
] = -109;
] = -92;
] = 52;
] = 41;
] = 103;
] = -4;
] = 35;
] = 121;
] = 25;
] = -40;
] = -55;
] = 43;
] = -49;
```

xrefs to g_key

Directic	Typ	Address	Text
Up	r	sub_401090:loc_40110A	mov ecx, g_key
Up	r	sub_401200:loc_40127B	mov edx, g_key
Up	r	sub_4012A0+CD	mov ecx, g_key
Up	r	sub_4013A0:loc_40141B	mov edx, g_key
Up	w	sub_401440+60	mov g_key, ecx
Up	r	sub_401440:loc_4014AA	mov eax, g_key
Up	r	_main+82	mov ecx, g_key

Line 5 of 7

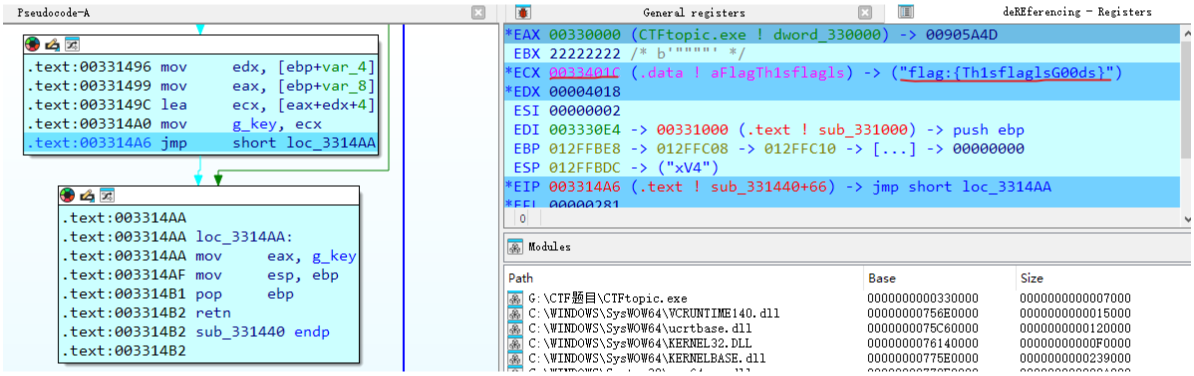
对着g_key按x查找引用,发现来自sub_401440+60,进去看下



```
int sub_401440()
{
  HMODULE ModuleHandleA; // [esp+4h] [ebp-8h]
  unsigned int n0x10000; // [esp+8h] [ebp-4h]

  ModuleHandleA = GetModuleHandleA(0);
  for ( n0x10000 = 0; n0x10000 < 0x10000; ++n0x10000 )
  {
    if ( *(_DWORD *)((char *)ModuleHandleA + n0x10000) == 0x12345678 && *((_BYTE *)ModuleHandleA + n0x10000 + 4) != 0x75 )
    {
      g_key = (int)ModuleHandleA + n0x10000 + 4;
      return g_key;
    }
  }
  return g_key;
}
```

发现来自一个基址加偏移的位置,动态跟踪一下.



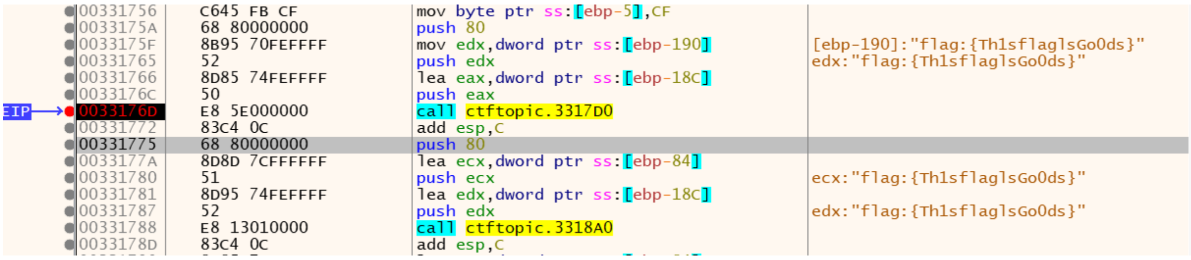
```
.text:00331496 mov     edx, [ebp+var_4]
.text:00331499 mov     eax, [ebp+var_8]
.text:0033149C lea     ecx, [eax+edx+4]
.text:003314A0 mov     g_key, ecx
.text:003314A6 jmp     short loc_3314AA

.text:003314AA
.text:003314AA loc_3314AA:
.text:003314AA mov     eax, g_key
.text:003314AF mov     esp, ebp
.text:003314B1 pop     ebp
.text:003314B2 retn
.text:003314B2 sub_331440 endp
.text:003314B2
```

```
*EAX 00330000 (CTFtopic.exe ! dword_330000) -> 00905A4D
 EBX 22222222 /* b'""""' */
*ECX 0033401C (.data ! aFlagTh1sflagls) -> ("flag:{Th1sflaglsG00ds}")
*EDX 00004018
 ESI 00000002
 EDI 003330E4 -> 00331000 (.text ! sub_331000) -> push ebp
 EBP 012FFBE8 -> 012FFC08 -> 012FFC10 -> [...] -> 00000000
 ESP 012FFBDC -> ("xV4")
*EIP 003314A6 (.text ! sub_331440+66) -> jmp short loc_3314AA
*EFL 00000281
```

Path	Base	Size
G:\CTF题目\CTFtopic.exe	0000000000330000	0000000000007000
C:\WINDOWS\SysWOW64\VCRUNTIME140.dll	00000000756E0000	0000000000015000
C:\WINDOWS\SysWOW64\ucrtbase.dll	0000000075C60000	0000000000120000
C:\WINDOWS\SysWOW64\KERNEL32.DLL	0000000076140000	00000000000F0000
C:\WINDOWS\SysWOW64\KERNELBASE.dll	00000000775E0000	0000000000239000

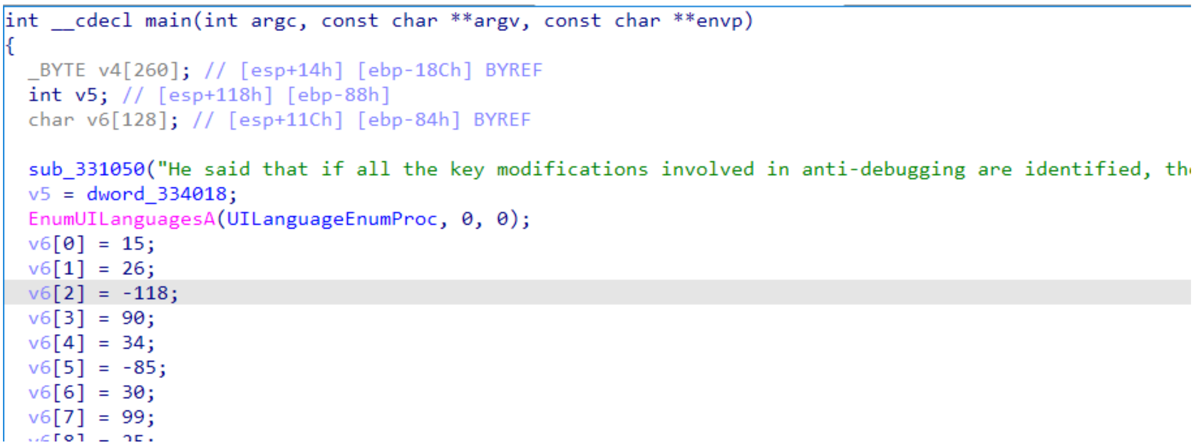
发现一个字符串,然后继续运行发现不会断在初始化算法的地方,使用xdbg试一下.



```
    00331756   C645 FB CF       mov byte ptr ss:[ebp-5],CF
    0033175A   68 80000000      push 80
    0033175F   8B95 70FEFFFF    mov edx,dword ptr ss:[ebp-190]   [ebp-190]:"flag:{Th1sflaglsGo0ds}"
    00331765   52               push edx                         edx:"flag:{Th1sflaglsGo0ds}"
    00331766   8D85 74FEFFFF    lea eax,dword ptr ss:[ebp-18C]
    0033176C   50               push eax
EIP 0033176D   E8 5E000000      call ctftopic.3317D0
    00331772   83C4 0C          add esp,C
    00331775   68 80000000      push 80
    0033177A   8D8D 7CFFFFFF    lea ecx,dword ptr ss:[ebp-84]
    00331780   51               push ecx                         ecx:"flag:{Th1sflaglsGo0ds}"
    00331781   8D95 74FEFFFF    lea edx,dword ptr ss:[ebp-18C]
    00331787   52               push edx                         edx:"flag:{Th1sflaglsGo0ds}"
    00331788   E8 13010000      call ctftopic.3318A0
    0033178D   83C4 0C          add esp,C
```

发现也有类似字符串,仔细看可以发现两者有点不一样,edx=0033401C "flag:{Th1sflagIsGo0ds}",运行之后发现是乱的,感觉是key不对,根据提示这是一个关于反调试的题目,到目前为止没感觉到又反调试.看看有哪些地方修改了0x0033401C地址,我在OEP断下后对内存下访问断点看看.测试发现就赋值的地方会断下.

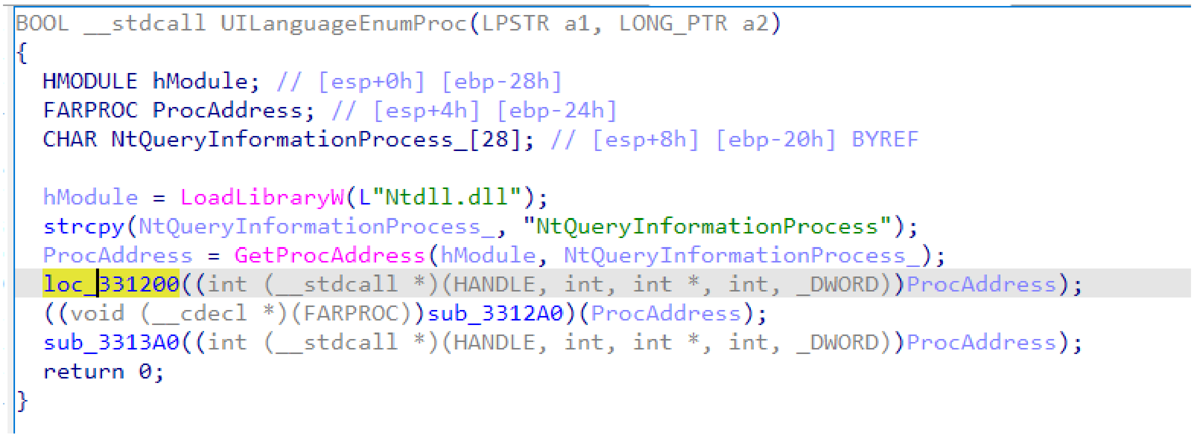
回到ida中main函数反汇编可以看到有个回调函数.



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  _BYTE v4[260]; // [esp+14h] [ebp-18Ch] BYREF
  int v5; // [esp+118h] [ebp-88h]
  char v6[128]; // [esp+11Ch] [ebp-84h] BYREF

  sub_331050("He said that if all the key modifications involved in anti-debugging are identified, th
  v5 = dword_334018;
  EnumUILanguagesA(UILanguageEnumProc, 0, 0);
  v6[0] = 15;
  v6[1] = 26;
  v6[2] = -118;
  v6[3] = 90;
  v6[4] = 34;
  v6[5] = -85;
  v6[6] = 30;
  v6[7] = 99;
  v6[8] = 25;
```

下面图中就是回调函数实现,里有个

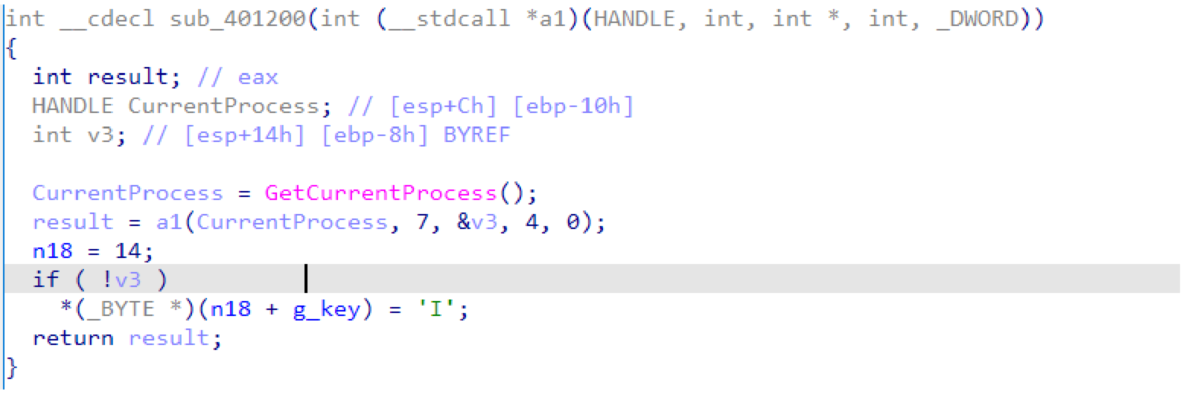


```
BOOL __stdcall UILanguageEnumProc(LPSTR a1, LONG_PTR a2)
{
  HMODULE hModule; // [esp+0h] [ebp-28h]
  FARPROC ProcAddress; // [esp+4h] [ebp-24h]
  CHAR NtQueryInformationProcess_[28]; // [esp+8h] [ebp-20h] BYREF

  hModule = LoadLibraryW(L"Ntdll.dll");
  strcpy(NtQueryInformationProcess_, "NtQueryInformationProcess");
  ProcAddress = GetProcAddress(hModule, NtQueryInformationProcess_);
  loc_331200((int (__stdcall *)(HANDLE, int, int *, int, _DWORD))ProcAddress);
  ((void (__cdecl *)(FARPROC))sub_3312A0)(ProcAddress);
  sub_3313A0((int (__stdcall *)(HANDLE, int, int *, int, _DWORD))ProcAddress);
  return 0;
}
```

进去这几个函数看下,里面也有花指令,使用nop大法之后f5. 发现了4处修改字符串的操作.

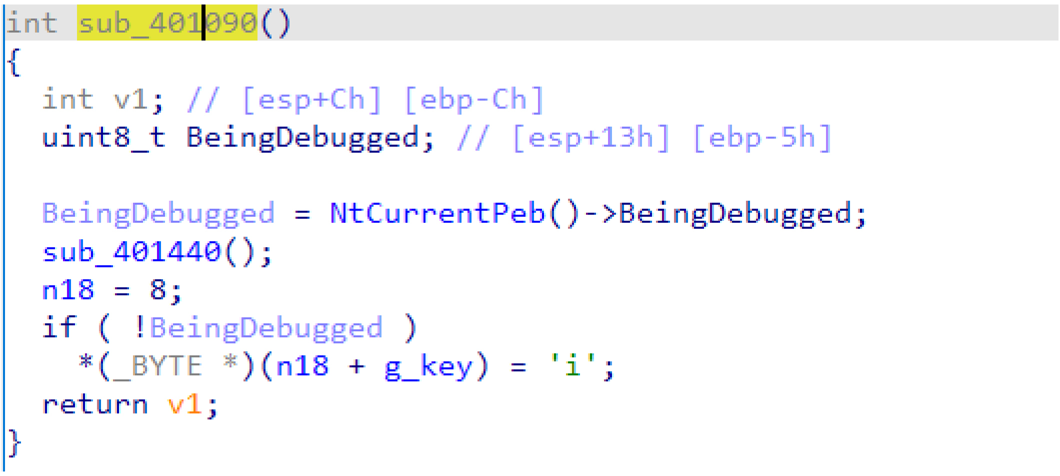
g_key[14]='l';



```
int __cdecl sub_401200(int (__stdcall *a1)(HANDLE, int, int *, int, _DWORD))
{
  int result; // eax
  HANDLE CurrentProcess; // [esp+Ch] [ebp-10h]
  int v3; // [esp+14h] [ebp-8h] BYREF

  CurrentProcess = GetCurrentProcess();
  result = a1(CurrentProcess, 7, &v3, 4, 0);
  n18 = 14;
  if ( !v3 )
    *(_BYTE *)(n18 + g_key) = 'I';
  return result;
}
```

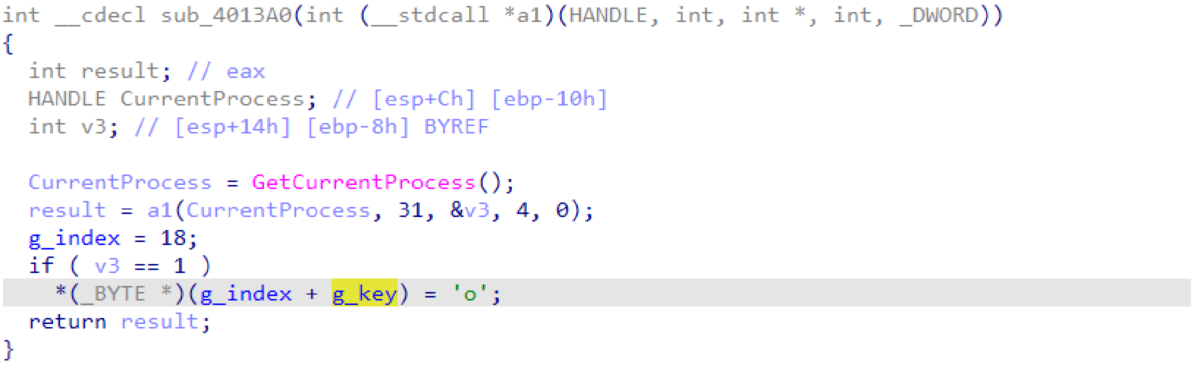
g_key[8]='i';



```
int sub_401090()
{
  int v1; // [esp+Ch] [ebp-Ch]
  uint8_t BeingDebugged; // [esp+13h] [ebp-5h]

  BeingDebugged = NtCurrentPeb()->BeingDebugged;
  sub_401440();
  n18 = 8;
  if ( !BeingDebugged )
    *(_BYTE *)(n18 + g_key) = 'i';
  return v1;
}
```

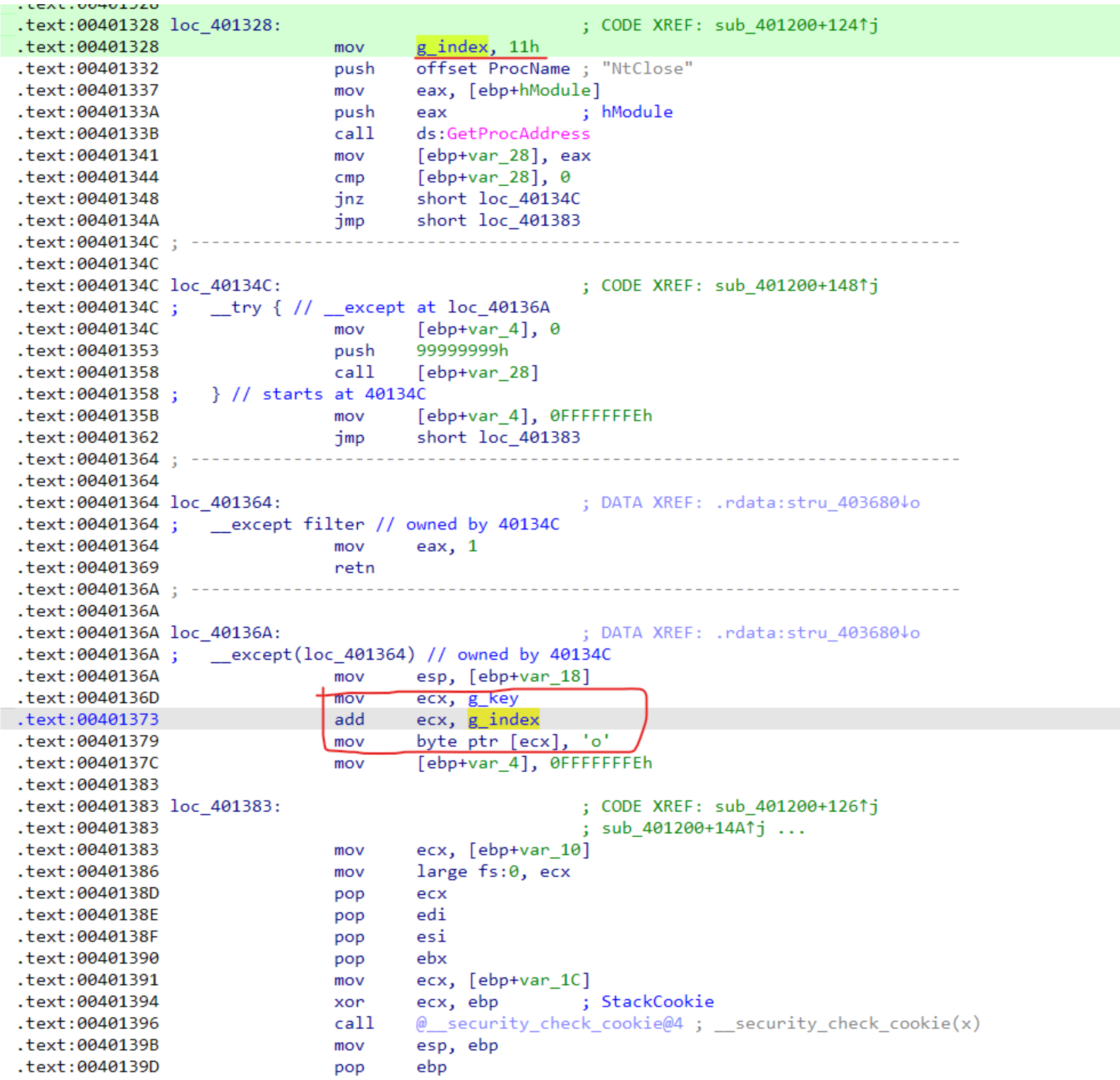
g_key[18]='o';



```
int __cdecl sub_4013A0(int (__stdcall *a1)(HANDLE, int, int *, int, _DWORD))
{
  int result; // eax
  HANDLE CurrentProcess; // [esp+Ch] [ebp-10h]
  int v3; // [esp+14h] [ebp-8h] BYREF

  CurrentProcess = GetCurrentProcess();
  result = a1(CurrentProcess, 31, &v3, 4, 0);
  g_index = 18;
  if ( v3 == 1 )
    *(_BYTE *)(g_index + g_key) = 'o';
  return result;
}
```

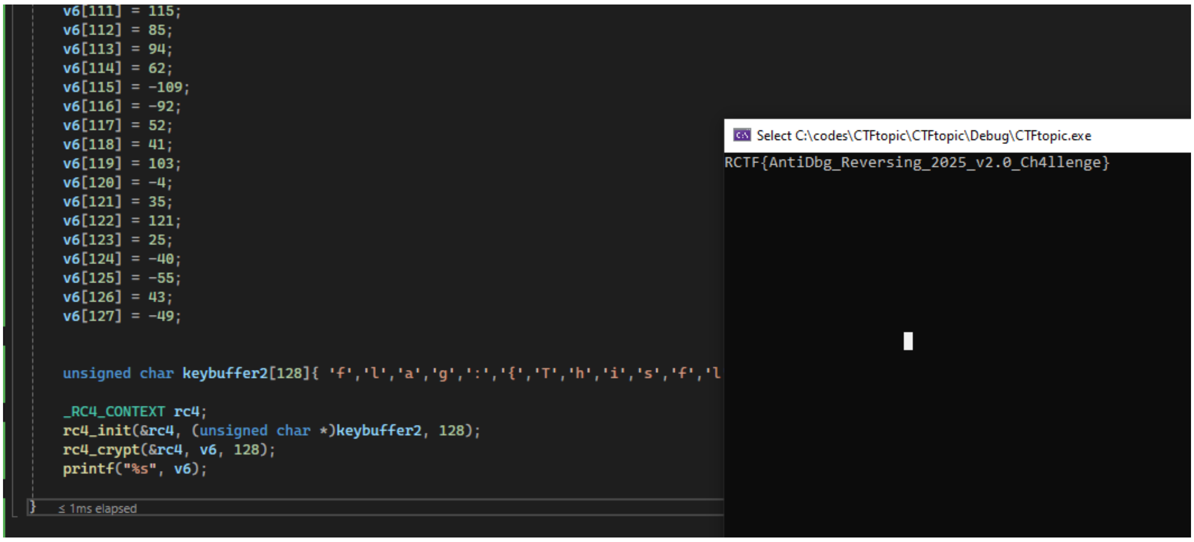
g_key[17]='o';



```
.text:00401328
.text:00401328 loc_401328:                             ; CODE XREF: sub_401200+124↑j
.text:00401328                 mov     g_index, 11h
.text:00401332                 push    offset ProcName ; "NtClose"
.text:00401337                 mov     eax, [ebp+hModule]
.text:0040133A                 push    eax             ; hModule
.text:0040133B                 call    ds:GetProcAddress
.text:00401341                 mov     [ebp+var_28], eax
.text:00401344                 cmp     [ebp+var_28], 0
.text:00401348                 jnz     short loc_40134C
.text:0040134A                 jmp     short loc_401383
.text:0040134C ; ---------------------------------------------------------------------------
.text:0040134C
.text:0040134C loc_40134C:                             ; CODE XREF: sub_401200+148↑j
.text:0040134C ;   __try { // __except at loc_40136A
.text:0040134C                 mov     [ebp+var_4], 0
.text:00401353                 push    99999999h
.text:00401358                 call    [ebp+var_28]
.text:00401358 ;   } // starts at 40134C
.text:0040135B                 mov     [ebp+var_4], 0FFFFFFFEh
.text:00401362                 jmp     short loc_401383
.text:00401364 ; ---------------------------------------------------------------------------
.text:00401364
.text:00401364 loc_401364:                             ; DATA XREF: .rdata:stru_403680↓o
.text:00401364 ;   __except filter // owned by 40134C
.text:00401364                 mov     eax, 1
.text:00401369                 retn
.text:0040136A ; ---------------------------------------------------------------------------
.text:0040136A
.text:0040136A loc_40136A:                             ; DATA XREF: .rdata:stru_403680↓o
.text:0040136A ;   __except(loc_401364) // owned by 40134C
.text:0040136A                 mov     esp, [ebp+var_18]
.text:0040136D                 mov     ecx, g_key
.text:00401373                 add     ecx, g_index
.text:00401379                 mov     byte ptr [ecx], 'o'
.text:0040137C                 mov     [ebp+var_4], 0FFFFFFFEh
.text:00401383
.text:00401383 loc_401383:                             ; CODE XREF: sub_401200+126↑j
.text:00401383                                         ; sub_401200+14A↑j ...
.text:00401383                 mov     ecx, [ebp+var_10]
.text:00401386                 mov     large fs:0, ecx
.text:0040138D                 pop     ecx
.text:0040138E                 pop     edi
.text:0040138F                 pop     esi
.text:00401390                 pop     ebx
.text:00401391                 mov     ecx, [ebp+var_1C]
.text:00401394                 xor     ecx, ebp        ; StackCookie
.text:00401396                 call    @__security_check_cookie@4 ; __security_check_cookie(x)
.text:0040139B                 mov     esp, ebp
.text:0040139D                 pop     ebp
```

上面几个函数都是反调试检测，如果没用检测到调试就会把RC4算法的密钥还原，其中横线标记的函数逻辑有点特殊，他是检测到了才会还原，以此来对抗使用工具无脑使用反反调试工具的人。

密钥为 “flag:{ThisflagIsGoods}”。



```
    v6[111] = 115;
    v6[112] = 85;
    v6[113] = 94;
    v6[114] = 62;
    v6[115] = -109;
    v6[116] = -92;
    v6[117] = 52;
    v6[118] = 41;
    v6[119] = 103;
    v6[120] = -4;
    v6[121] = 35;
    v6[122] = 121;
    v6[123] = 25;
    v6[124] = -40;
    v6[125] = -55;
    v6[126] = 43;
    v6[127] = -49;


    unsigned char keybuffer2[128]{ 'f','l','a','g',':','{','T','h','i','s','f','l

    _RC4_CONTEXT rc4;
    rc4_init(&rc4, (unsigned char *)keybuffer2, 128);
    rc4_crypt(&rc4, v6, 128);
    printf("%s", v6);

}   ≤ 1ms elapsed
```

Select C:\codes\CTFtopic\CTFtopic\Debug\CTFtopic.exe

```
RCTF{AntiDbg_Reversing_2025_v2.0_Ch4llenge}
```

之后通过rc4_crypt函数输出真正flag。

RCTF{AntiDbg_Reversing_2025_v2.0_Ch4llenge}