题目描述

首先，你得让它跑起来。（提示：PE结构）

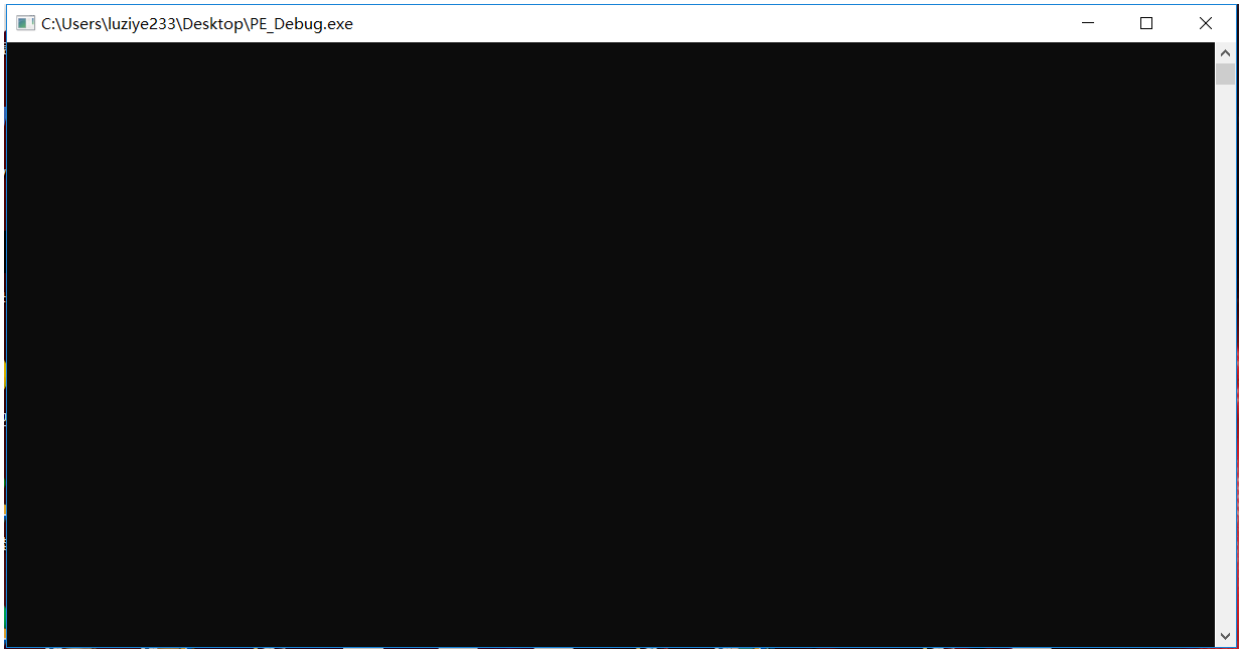
考点

PE结构、反调试

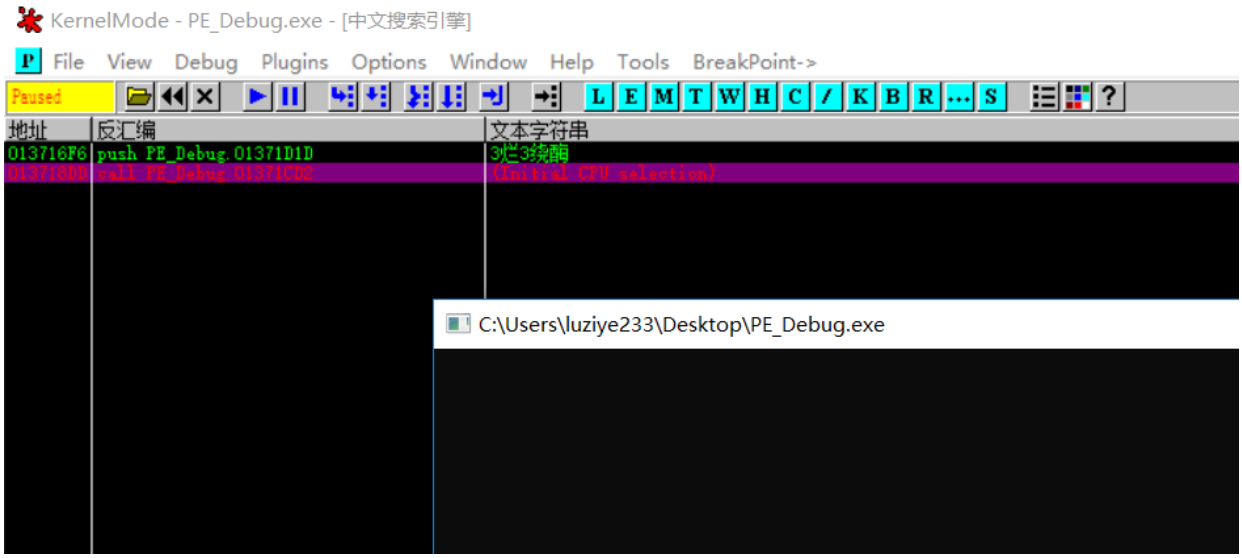
解题思路

0x00 准备

- 1 Peid/PE View
- 2 OD、ida
- 3 c32 asm/winhex/uedit32



直接打开程序，什么都没有的




```
10  _DWORD *v9; // esi
11  int v10; // edi
12  _DWORD *v11; // esi
13  int v12; // eax
14  int v13; // et1
15
16  while ( 1 )
17  {
18    sub_401CD2();
19    if ( !(unsigned __int8)sub_401A7E(1)
20      || (v2 = 0, *(_BYTE *)(a1 - 25) = 0,
21                  *(_DWORD *)(a1 - 4) = 0,
22                  *(_BYTE *)(a1 - 36) = sub_401A4C(),
23                  dword_404334 == 1) )
24    {
25      sub_401D90(7);
26      goto LABEL_20;
27    }
28    if ( dword_404334 )
29    {
30      v2 = 1;
31      *(_BYTE *)(a1 - 25) = 1;
32    }
33    else
```

在od和ida中也无法正常呈现

0x01 分析

程序能在od、ida里运行，是没有加壳的 但是如果仔细看一下文件的pe结构，会发现两个奇怪的地方

- 1 DataDirectory

我们知道在OllyDbg非常严格地遵循了微软对PE文件头部的规定。在PE文件的头部，通常存在一个叫作IMAGE_OPTIONAL_HEADER的结构。 因为DataDirectory数组不足以容纳超过0x10个目录项，所以当NumberOfRvaAndSizes大于0x10时，Windows加载器将会忽略NumberOfRvaAndSizes。老版od会遵循这个规则，程序会无法加载 我们可以把这里改回10h（具体位置可能不太一样）

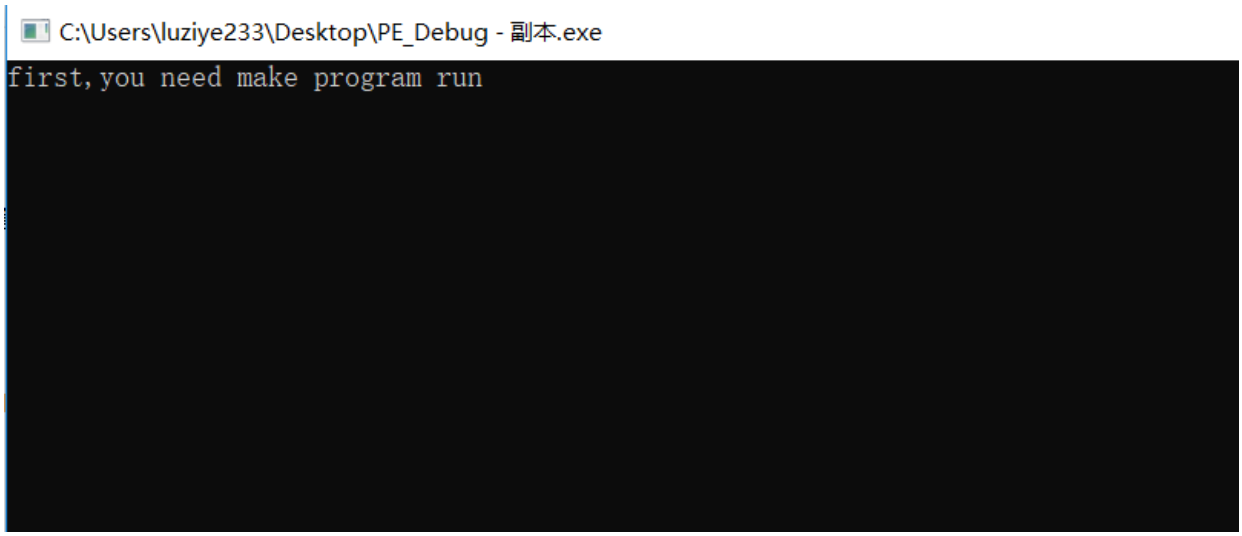
```
00000060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 ; t be run in DOS
00000070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 ; mode....$.......
00000080h: 17 20 36 EE 53 41 58 BD 53 41 58 BD 53 41 58 BD ; . 6頡AX絀AX絀AX?
00000090h: 5A 39 CB BD 5F 41 58 BD 01 29 59 BC 51 41 58 BD ; Z9私_AX?)Y糧AX?
000000a0h: CD E1 9F BD 52 41 58 BD 01 29 5B BC 52 41 58 BD ; 歪赍RAX?)[糝AX?
000000b0h: 01 29 5D BC 41 41 58 BD 01 29 5C BC 5F 41 58 BD ; .)]糀AX?)\耬AX?
000000c0h: 3C 25 59 BC 50 41 58 BD 53 41 59 BD 6F 41 58 BD ; <%Y糙AX絀AY給AX?
000000d0h: 35 29 50 BC 52 41 58 BD 35 29 A7 BD 52 41 58 BD ; 5)P糝AX?)bRAX?
000000e0h: 35 29 5A BC 52 41 58 BD 52 69 63 68 53 41 58 BD ; 5)Z糝AX絀ichSAX?
000000f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000100h: 00 00 00 00 00 00 00 00 50 45 00 00 4C 01 05 00 ; ........PE..L...
00000110h: 97 1D DC 5B 00 00 00 00 00 00 00 00 E0 00 02 01 ; ?躪........?..
00000120h: 0B 01 0E 0F 00 14 00 00 00 16 00 00 00 00 00 00 ; ................
00000130h: DD 18 00 00 00 10 00 00 00 30 00 00 00 00 40 00 ; ?.......0....@.
00000140h: 00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00 ; ................
00000150h: 06 00 00 00 00 00 00 00 00 70 00 00 00 04 00 00 ; .........p......
00000160h: 00 00 00 00 03 00 40 81 00 00 10 00 00 10 00 00 ; ......@?........
00000170h: 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 ; ................
00000180h: 00 00 00 00 00 00 00 00 8C 36 00 00 B4 00 00 00 ; ........?..?..
00000190h: 00 50 00 00 E0 01 00 00 00 00 00 00 00 00 00 00 ; .P..?...........
000001a0h: 00 00 00 00 00 00 00 00 00 60 00 00 B0 01 00 00 ; .........`..?..
000001b0h: 80 32 00 00 70 00 00 00 00 00 00 00 00 00 00 00 ; €2..p...........
```

- ## 2 节区头

另一种PE头的欺骗与节头部有关。文件内容中包含的节包括代码节、数据节、资源节，以及一些其他信息节。每个节都拥有一个IMAGE_SECTION_HEADER结构的头部。 VirtualSize和SizeOfRawData是其中两个比较重要的属性。根据微软对PE的规定，VirtualSize应该包 含载入到内存的节大小，SizeOfRawData应该包含节在硬盘中的大小。Windows加载器使用VirtualSize和SizeOfRawData中的最小值将节数据映射到内存。如果SizeOfRawData大于VirtualSize，则仅将VirtualSize大小的数据复制入内存，忽略其余数据。

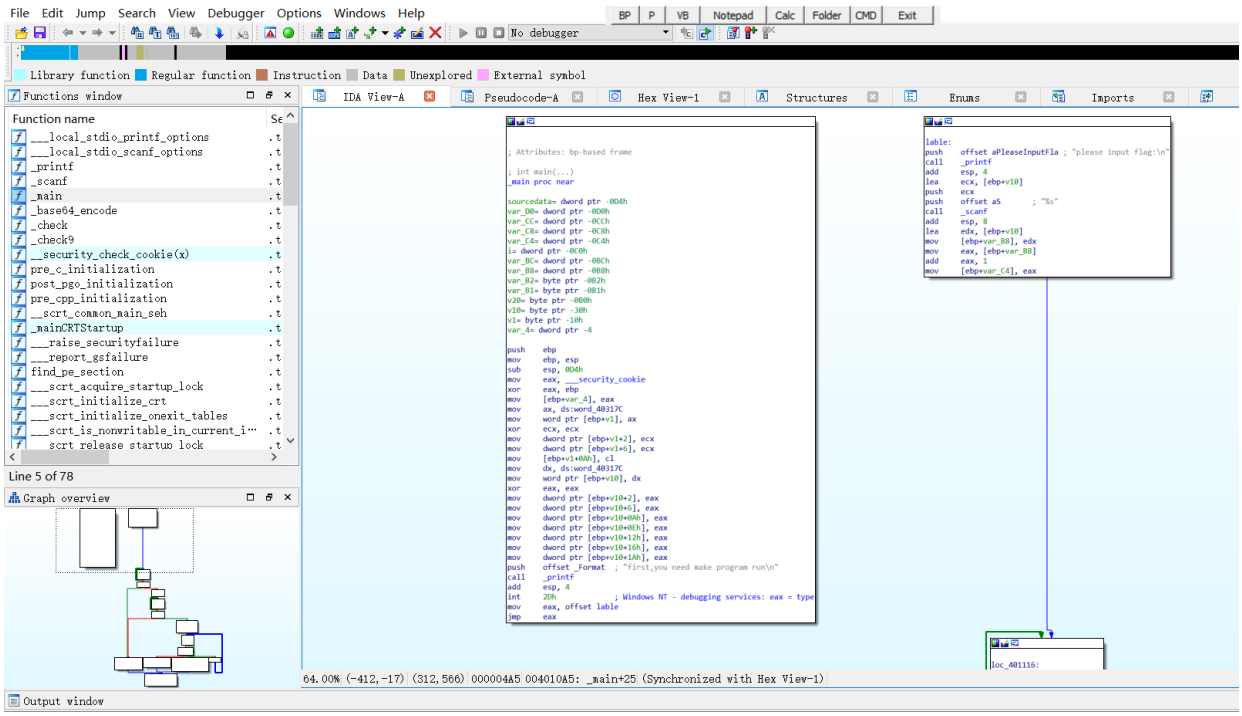
如果程序有压缩壳，SizeOfRawData是可能比VirtualSize小的，但是为0就太过异常，我们把.rdata节区的SizeOfRawData大小修改一下，让它比虚拟大小稍大

修改完成后，保存，运行程序



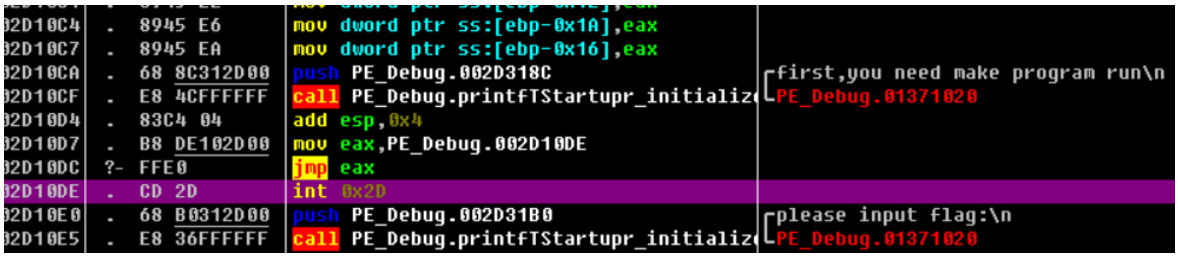
现在程序能够运行出文字了，可还是闪退

这时我们再丢入od、ida里看看

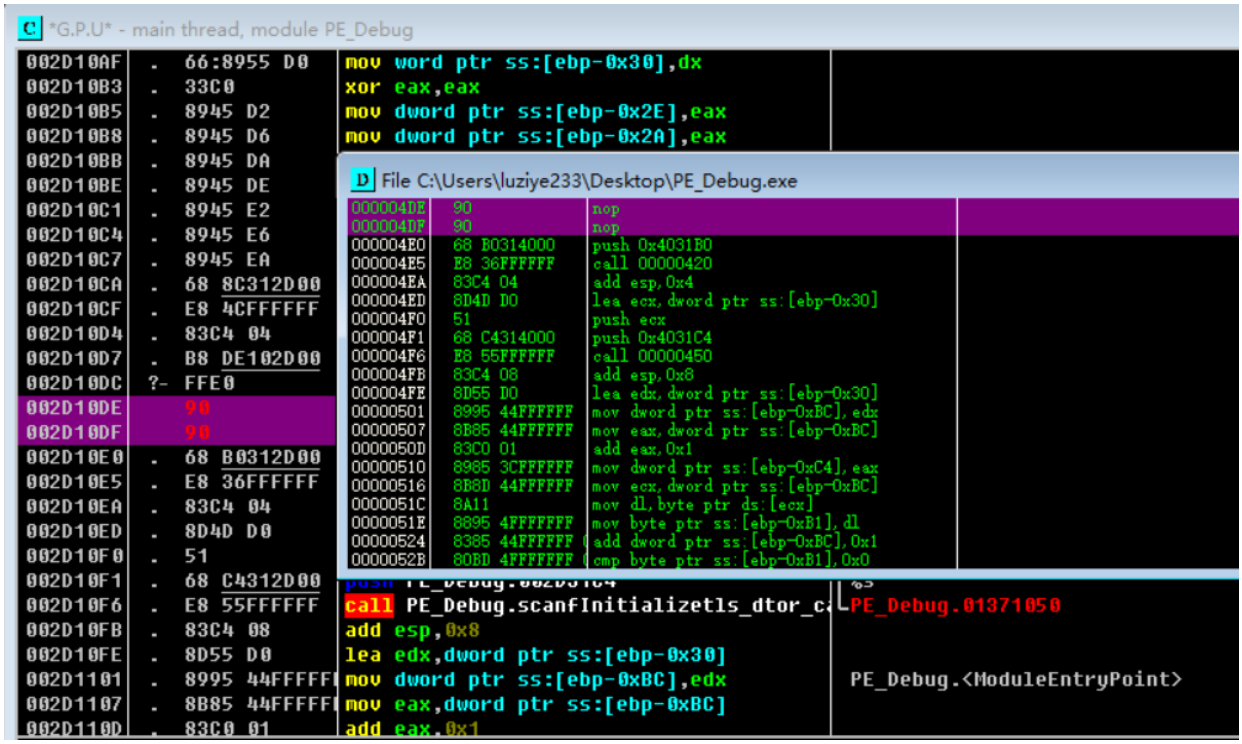


在ida中和之前已经截然不同，甚至出现了之前刚才运行的文字“first,you need make program run”

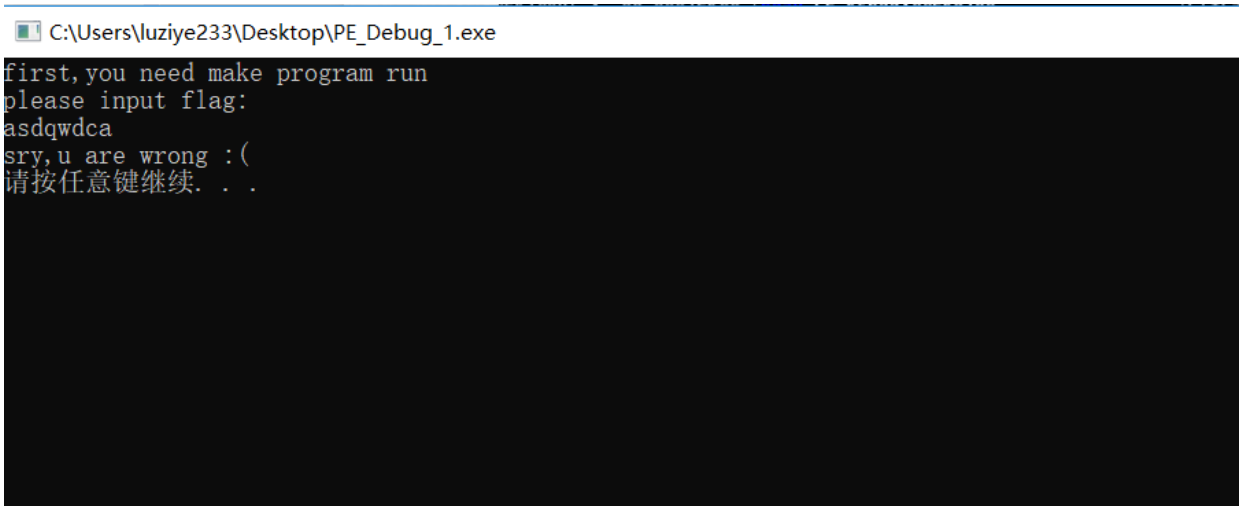
- **3 Int 2D中断** 在ida里看不出什么异常，在od里看看



这里有一个int 2d中断 int 2d有一个有的意思的特性，它会忽略下条指令的第一个字节。也就是说，因为int 2d，后面的代码在实际运行中其实已经被打乱了，所以程序会运行失败，我们把它nop掉就行。 当然不用nop也是可以分析的，只是可能会比较麻烦



再次运行：

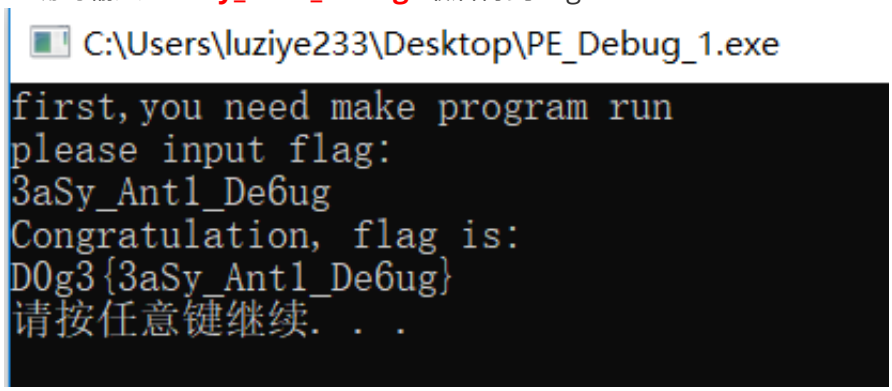


至此程序已经可以正常运行了

- **4 简单的算法** 有10个简单的反调试检测，每一个检测通过会返回一个字符 得到串字符是：2TVBnx0lnn

  然后我们输入经过base64加密后，分别和一串特定字符串：LKd8gPYWS[，还有过掉所有反调试得到的字符串做比较： for(int i=9; i>=0; i--) *(str1+i) == * (base64 +2 * i +1) && (*(base64 + i * 2) + 2) == (str2[i] ^ 3); 逆推我就不写了，自己做吧

正确的输入：**3aSy_Ant1_De6ug** 最后得到flag



0x02 总结

这道题总体上没有什么难度，可能在一开始的pe头和int 2d会让人摸不着头脑，实际上也只是想让大家更多的了解到一些反调试的方法，如果实在脑洞太大，那只能认罚了。。。

这些反调试的简单机制在《逆向工程核心原理》或多或少都有所涉及 最后附上一篇写得很不错的反调试技术的文章，希望有空能看一看

https://blog.csdn.net/qq_32400847/article/details/52798050