

题目描述

首先，你得让它跑起来。（提示：PE结构）

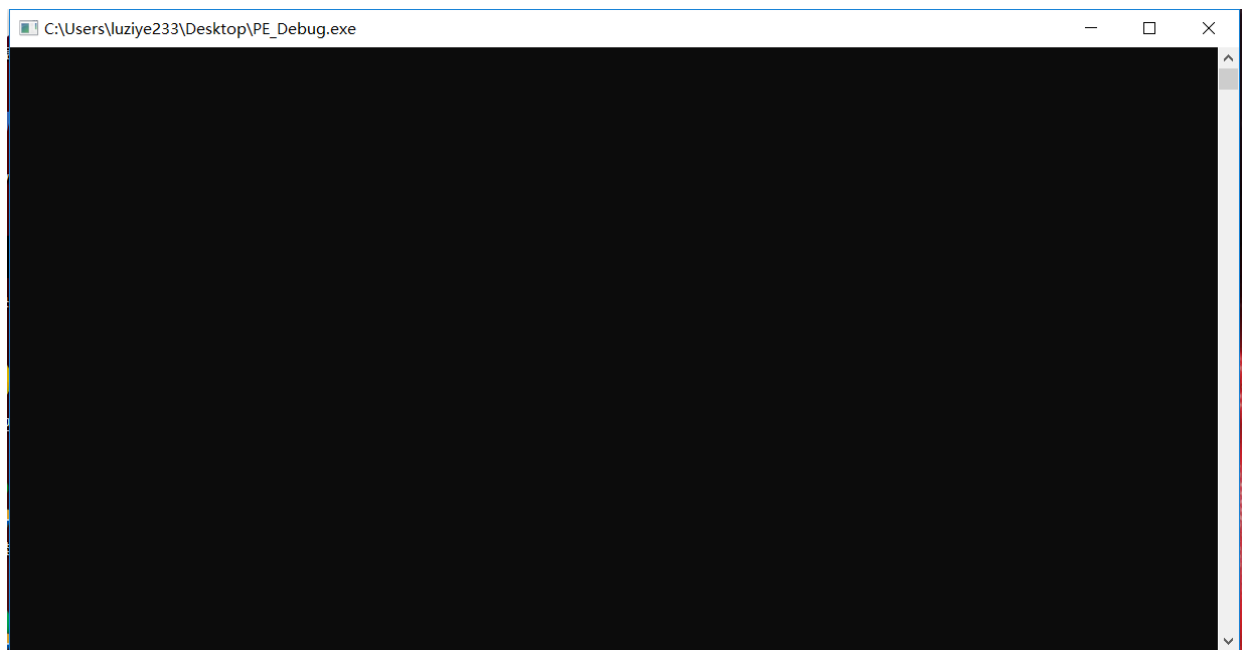
考点

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

PE 细节

基本信息

入口点:	000018DD	子系统:	0003
镜像基址:	00400000	节数目:	0005
镜像大小:	00007000	时间日期标志:	5BDC1D97
代码基址:	00001000	头部大小:	00000400
数据基址:	00003000	特征值:	0102
节对齐粒度:	00001000	校验和:	00000000
文件对齐粒度:	00000200	可选头部大小:	00E0
标志字:	010B	RVA 数目及大小:	00000099

目录信息

	RVA	大小	
输出表:	00000000	00000000	
输入表:	0000368C	000000B4	.. >
资源:	00005000	000001E0	.. >
TLS 表:	00000000	00000000	
调试:	00003280	00000070	.. >

关闭(C)

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，程序将无法加载 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 ; mode....$.
00000080h: 17 20 36 EE 53 41 58 BD 53 41 58 BD 53 41 58 BD ; . 6頤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 ; 歪發RAX?) [ 糝AX?
000000b0h: 01 29 5D BC 41 41 58 BD 01 29 5C BC 5F 41 58 BD ; .) ] 糝AX?) \ 糝AX?
000000c0h: 3C 25 59 BC 50 41 58 BD 53 41 59 BD 6F 41 58 BD ; <%Y 糝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?) bIRAX?
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 ; .....
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 ; ?躡.....?..
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

节查看器

名称	V. 偏移	V. 大小	R. 偏移	R. 大小	标志
.text	00001000	00001263	00000400	00001400	60000020
.rdata	00003000	00000D6A	00001800	<u>00000000</u>	40000040
.data	00004000	00000388	00002600	00000200	C0000040
.rsrc	00005000	000001E0	00002800	00000200	40000040
.reloc	00006000	000001B0	00002A00	00000200	42000040

关闭(C)

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

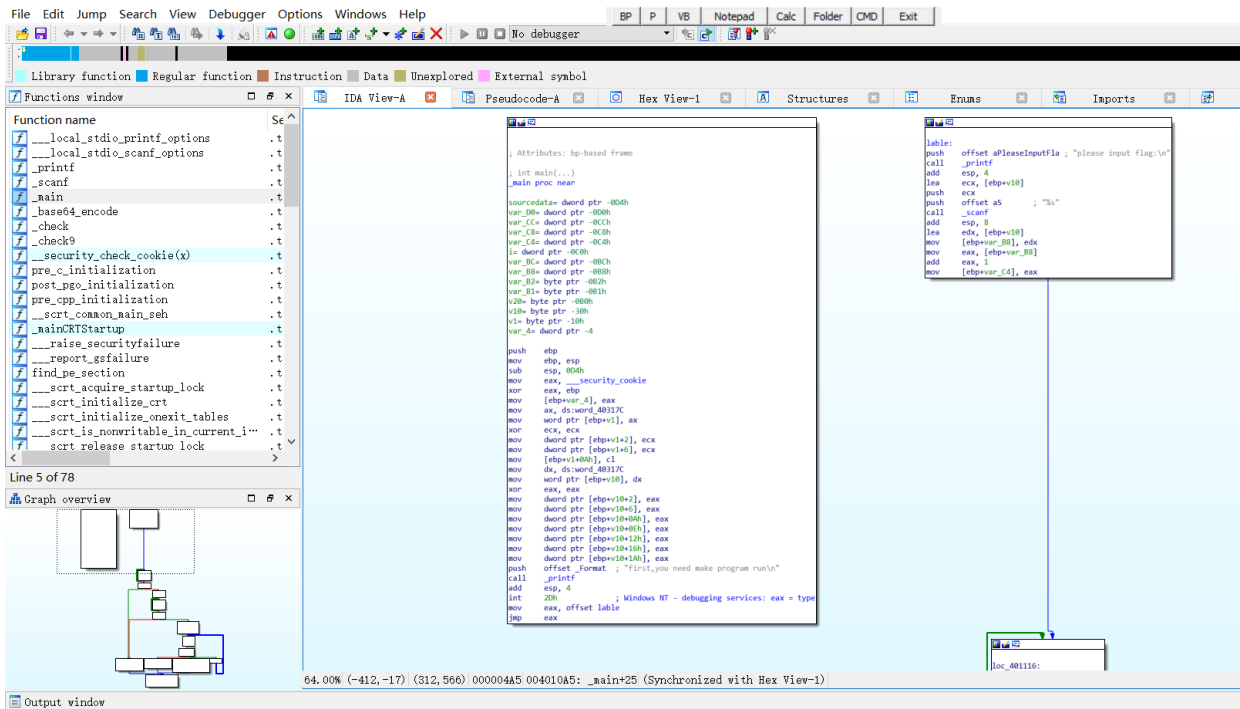
```
PE_Debug - 副本.exe x
0 1 2 3 4 5 6 7 8 9 a b c d e f
00000150h: 06 00 00 00 00 00 00 00 00 70 00 00 00 04 00 00 ; .....p.....
00000160h: 00 00 00 00 03 00 40 81 00 00 10 00 00 10 00 00 ; .....@?.....
00000170h: 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 ; .....
00000180h: 00 00 00 00 00 00 00 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.....
000001c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
000001d0h: F0 32 00 00 40 00 00 00 00 00 00 00 00 00 00 00 ; ?..@.....
000001e0h: 00 30 00 00 FC 00 00 00 00 00 00 00 00 00 00 00 00 ; .0..?.....
000001f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000200h: 2E 74 65 78 74 00 00 00 63 12 00 00 00 10 00 00 ; .text...c.....
00000210h: 00 14 00 00 00 04 00 00 00 00 00 00 00 00 00 00 ; .....
00000220h: 00 00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00 ; .... ..`rdata..
00000230h: 6A 0D 00 00 00 30 00 00 00 0E 00 00 00 18 00 00 ; j....0.....
00000240h: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 ; .....@..@
00000250h: 2E 64 61 74 61 00 00 00 88 03 00 00 00 40 00 00 ; .data...?...@..
00000260h: 00 02 00 00 00 26 00 00 00 00 00 00 00 00 00 00 ; .....&.....
00000270h: 00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00 ; ....@..?rsrc...
00000280h: E0 01 00 00 00 50 00 00 00 02 00 00 00 28 00 00 ; ?...P.....(..
00000290h: 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 ; .....@..@
000002a0h: 2E 72 65 6C 6F 63 00 00 B0 01 00 00 00 60 00 00 ; .reloc..?...`..
```

修改完成后，保存，运行程序

```
C:\Users\luziye233\Desktop\PE_Debug - 副本.exe
first,you need make program run
```

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

002D10C4	. 8945 E6	mov dword ptr ss:[ebp-0x1A],eax	
002D10C7	. 8945 EA	mov dword ptr ss:[ebp-0x16],eax	
002D10CA	. 68 8C312D00	push PE_Debug.002D318C	
002D10CF	. E8 4CFFFFFF	call PE_Debug.printfTStartupr_initialize	PE_Debug.01371020
002D10D4	. 83C4 04	add esp,0x4	
002D10D7	. B8 DE102D00	mov eax,PE_Debug.002D10DE	
002D10DC	?- FFE0	jmp eax	
002D10DE	. CD 2D	int 0x2D	
002D10E0	. 68 B0312D00	push PE_Debug.002D3180	
002D10E5	. E8 36FFFFFF	call PE_Debug.printfTStartupr_initialize	PE_Debug.01371020

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

G.P.U - main thread, module PE_Debug			
002D10AF	. 66:8955 D0	mov word ptr ss:[ebp-0x30],dx	
002D10B3	. 33C0	xor eax,eax	
002D10B5	. 8945 D2	mov dword ptr ss:[ebp-0x2E],eax	
002D10B8	. 8945 D6	mov dword ptr ss:[ebp-0x2A],eax	
002D10BB	. 8945 DA		
002D10BE	. 8945 DE		
002D10C1	. 8945 E2		
002D10C4	. 8945 E6		
002D10C7	. 8945 EA		
002D10CA	. 68 8C312D00	push PE_Debug.002D318C	
002D10CF	. E8 4CFFFFFF	call PE_Debug.printfTStartupr_initialize	PE_Debug.01371020
002D10D4	. 83C4 04	add esp,0x4	
002D10D7	. B8 DE102D00	mov eax,PE_Debug.002D10DE	
002D10DC	?- FFE0	jmp eax	
002D10DE	. CD 2D	int 0x2D	
002D10E0	. 68 B0312D00	push PE_Debug.002D3180	
002D10E5	. E8 36FFFFFF	call PE_Debug.printfTStartupr_initialize	PE_Debug.01371020
002D10EA	. 83C4 04	add esp,0x4	
002D10ED	. 8D4D D0	lea ecx,dword ptr ss:[ebp-0x30]	
002D10F0	. 51	push ecx	
002D10F1	. 68 C4312D00	push PE_Debug.002D318C	
002D10F6	. E8 55FFFFFF	call PE_Debug.scanfInitializetls_dtor_c	PE_Debug.01371050
002D10FB	. 83C4 08	add esp,0x8	
002D10FE	. 8D55 D0	lea edx,dword ptr ss:[ebp-0x30]	
002D1101	. 8995 44FFFFFF	mov dword ptr ss:[ebp-0xBC],edx	
002D1107	. 8B85 44FFFFFF	mov eax,dword ptr ss:[ebp-0xBC]	
002D110D	. 83C0 01	add eax,0x1	

再次运行：

C:\Users\luziye233\Desktop\PE_Debug_1.exe

```
first,you need make program run
please input flag:
asdqwda
sry,u are wrong :(
请按任意键继续. . .
```

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

C:\Users\luziye233\Desktop\PE_Debug_1.exe

```
first,you need make program run
please input flag:
3aSy_Ant1_De6ug
Congratulation, flag is:
D0g3{3aSy_Ant1_De6ug}
请按任意键继续. . .
```

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