

# 实验三



---

## EPO之指令Patch

# 在VC中观察两种形式的导入表使用

---

- ▶ 不包含头文件 windows.h, 自定义Sleep函数原型, 用以下两种不同形式实验:
- ▶ `void extern "C" _stdcall Sleep(int x);`
- ▶ `void extern "C" __declspec(dllimport) _stdcall Sleep(int x);`
- ▶ 前者没有明确告诉编译器函数是从DLL导入, 生成的是 **CALL...JMP [xxxx]** 的指令形式
- ▶ 而后者告诉编译器函数将从DLL导入, 因此生成的是 **CALL [xxxx]** 的指令形式

# 两种指令形式的关键机器码

观察两种指令形式的机器码：

CALL [xx xx xx xx] 对应的关键机器码是 **FF 15** xx xx xx xx

CALL ... JMP [xx xx xx xx] 对应的关键机器码是 **FF 25** xx xx xx xx



00401136 **FF 15** 4C A1 42 00 call dword ptr [\_\_imp\_\_CloseHandle@4 (0042a14c)]

00401145 E8 30 00 00 00 call Sleep (0040117a)

...  
0040117A **FF 25** 50 A1 42 00 jmp dword ptr [\_\_imp\_\_Sleep@4 (0042a150)]

因此，我们的病毒代码应该是要寻找具有FF 15和FF 25这种特殊形式的指令！！

# 对导入表调用指令进行patch感染

---

- ▶ 思路：如果**将CALL [xxxx] 或 JMP [xxx]指令修改替换指向我们的代码，将执行我们的代码**
- ▶ 选哪些函数的调用指令呢？选大多数程序都会去调用的函数
- ▶ 比如，对大多数程序而言，基本上在运行时库初始化代码中，编译器都会去调用GetCommandLineW或GetCommandLineA函数，该函数的作用是获取程序命令行参数传递给main函数
- ▶ 如果我们修改调用这个函数的CALL指令或JMP指令，则能规避入口检查的问题

# 以notepad.exe为例 说明CALL或JMP指令patch的原理

```
OllyDbg - notepad - 副本.exe - [CPU - main thread, module notepad_]
File View Debug Plugins Options Window Help
L E M T W C K B R ... S
010029A6 . 57 PUSH EDI
010029A7 . FF15 C8100001 Call virus GetCommandLineW>
010029AD . 68 D8130001 PUSH notepad_.010013D8
010029B2 . 6A 29 PUSH 29
010029B4 . 8BF8 MOV EDI,EAX
010029B6 . FF15 54120001 CALL DWORD PTR DS:[&USER32.GetSystemMetrics]
```

因为前面是CALL  
所以，病毒不管返回问题，API  
执行完毕，自然通过ret返回

被调API, 比如  
GetCommandLineW

# 间接跳转指令 Call [xx] 或 JMP [xx] 的Patch方法

▶ 1. 假定我们对调用某个函数A的CALL和JMP感兴趣，首先查找函数A的导入表项的地址xx xx xx xx（邮箱地址）

▶ 2. 得到地址后，查找代码段以找到间接跳转指令：

**FF 15 xx xx xx xx** (CALL [ xx xx xx xx ]) 或 **FF 25 xx xx xx xx** (JMP [xx xx xx xx])

▶ 3. 分别修改为到病毒的直接跳转指令：

**E8 yy yy yy yy 90** (CALL 偏移, NOP) 或 **E9 yy yy yy yy** (JMP 偏移)

虽然找到的指令可能不是调用A的指令，但基本没问题，病毒嘛，又不是正常程序....

思考以下问题：

1 为什么CALL [xx]要对应改成CALL yy，JMP [xx]要对应改成JMP yy？

2 为什么E8 yy yy yy yy后要加90？不加有问题么？

3 为什么E9 yy yy yy yy后可以不管？

010029A7 | . FF15 C8100001 | CALL DWORD

被patch的指令

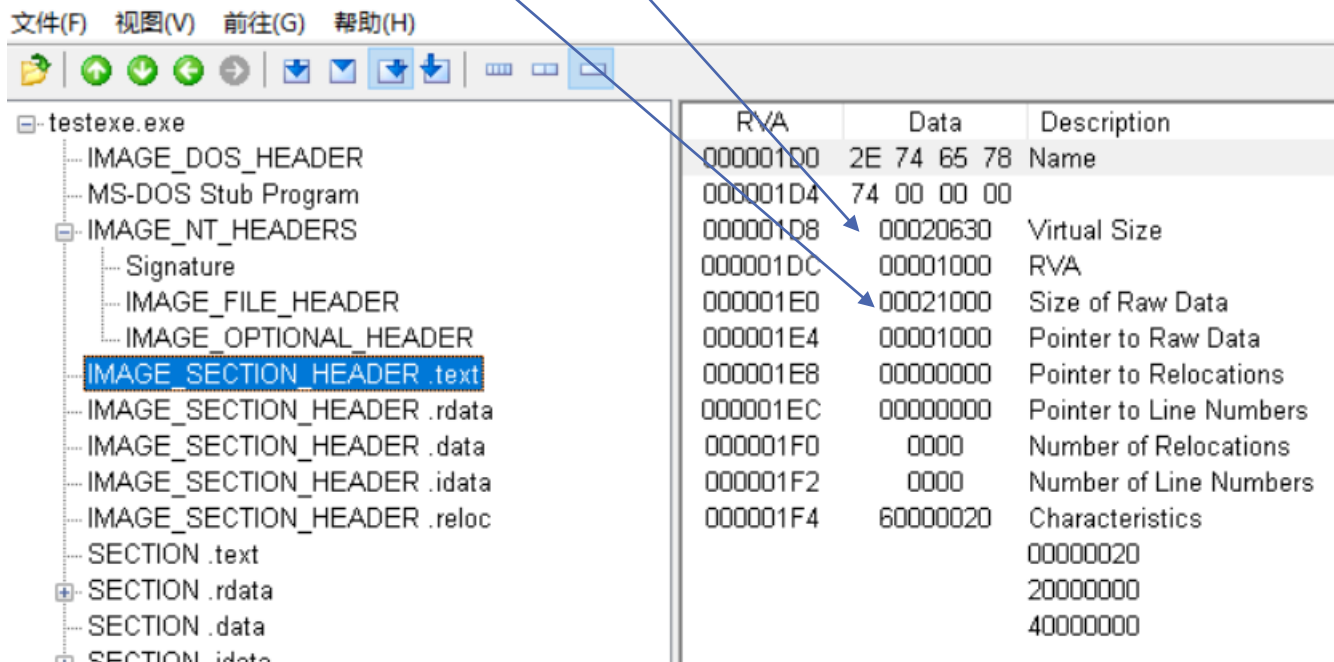
E8 xx xx xx xx 90

关键  
CALL和JMP  
的区别

# 实验步骤

在提供的testexe.exe文件进行病毒寄生

选取.text节寄生在文件的空洞



	RVA	Data	Description
IMAGE_DOS_HEADER	00000100	2E 74 65 78	Name
MS-DOS Stub Program	000001D4	74 00 00 00	
IMAGE_NT_HEADERS	000001D8	00020630	Virtual Size
Signature	000001DC	00001000	RVA
IMAGE_FILE_HEADER	000001E0	00021000	Size of Raw Data
IMAGE_OPTIONAL_HEADER	000001E4	00001000	Pointer to Raw Data
IMAGE_SECTION_HEADER .text	000001E8	00000000	Pointer to Relocations
IMAGE_SECTION_HEADER .rdata	000001EC	00000000	Pointer to Line Numbers
IMAGE_SECTION_HEADER .data	000001F0	0000	Number of Relocations
IMAGE_SECTION_HEADER .idata	000001F2	0000	Number of Line Numbers
IMAGE_SECTION_HEADER .reloc	000001F4	60000020	Characteristics
SECTION .text		00000020	
SECTION .rdata		20000000	
SECTION .data		40000000	
SECTION .idata			

# 生成病毒寄生代码

写一段病毒寄生代码，这段代码会将4个nop填充为学号后8位在调试模式下观察它的反汇编情况（注意更改为学号！）

为:E8 04 00 00 00, 90 90 90 90, 58, BB 11 11 11 11, 89 18,

```
int __tmain(int argc, _TCHAR* argv[])
{
    _asm{
        call code_start;
        nop;
        nop;
        nop;
        nop;
    }
    code_start:
        pop eax;
        mov ebx, 0x11111111;
        mov [eax], ebx;
        jmp code_start;
    }
    return 0;
}
```

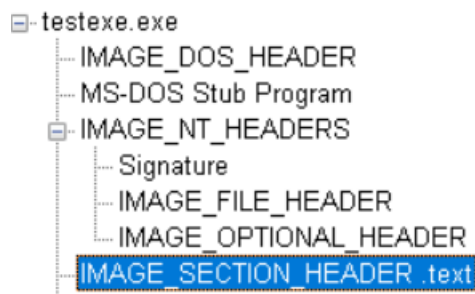
0003139E	E8 04 00 00 00	call	code_start (313A7h)
11:		nop;	
000313A3	90	nop	
12:		nop;	
000313A4	90	nop	
13:		nop;	
000313A5	90	nop	
14:		nop;	
000313A6	90	nop	
15:	code_start:		
16:	pop eax;		
000313A7	58	pop	eax
17:	mov ebx, 0x11111111;		
000313A8	BB 11 11 11 11	mov	ebx, 11111111h
18:	mov [eax], ebx;		
000313AD	89 18	mov	dword ptr [eax], ebx
19:	jmp code_start;		
000313AF	EB F6	jmp	code_start (313A7h)
20:			

最后一条JMP，我们需要替换成跳转回原API函数的入口点  
我们把指令Patch到这段病毒执行代码后，4个nop会被填充  
为0x11111111，从而可以查看我们Patch的指令是否成功



# 将病毒代码寄生到text节

Text节空洞位置为 $1000 + 20630 = 21630$



RVA	Data	Description
00000100	2E 74 65 78	Name
000001D4	74 00 00 00	
000001D8	00020630	Virtual Size
000001DC	00001000	RVA
000001E0	00021000	Size of Raw Data
000001E4	00001000	Pointer to Raw Data
000001E8	00000000	Pointer to Relocations

用UE/C32ASM打开文件，找到21630的位置，写入E8 04 00 00 00, 90 90 90 90, 58, BB 11 11 11 11, 89 18, **e9 00 00 00 00** (后续填充) 共22 (0x16) 个字节 (注意**11 11 11 11** 替换为学号后8位)

```
00021630h: E8 04 00 00 00 90 90 90 90 58 bb 11 11 11 11 89 ;
00021640h: 18 e9 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00021650h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
```

新的virtualSize为 $20630 + 16 = 0x20646$

# 添加text节的内存可写属性

病毒代码需要在4个NOP地方写数据，而text节没有内存写属性，所以需要修改属性为60000020|80000000 = E0000020

000001F4    60000020    Characteristics  
00000020  
20000000  
40000000

IMAGE\_SCN\_CNT\_CODE  
IMAGE\_SCN\_MEM\_EXECUTE  
IMAGE\_SCN\_MEM\_READ

testexe.exe  
... IMAGE\_DOS\_HEADER  
... MS-DOS Stub Program  
+ IMAGE\_NT\_HEADERS  
... IMAGE\_SECTION\_HEADER .text  
... IMAGE\_SECTION\_HEADER .rdata  
... **IMAGE\_SECTION\_HEADER .data**  
... IMAGE\_SECTION\_HEADER .idata  
... IMAGE\_SECTION\_HEADER .reloc  
... SECTION .text  
+ SECTION .rdata  
... SECTION .data  
+ SECTION .idata  
+ SECTION .reloc  
... IMAGE\_DEBUG\_TYPE\_CODEVIEW

p-file	Data	Description	Value
00000220	2E 64 61 74	Name	.data
00000224	61 00 00 00		
00000228	00005618	Virtual Size	
0000022C	00024000	RVA	
00000230	00004000	Size of Raw Data	
00000234	00024000	Pointer to Raw Data	
00000238	00000000	Pointer to Relocations	
0000023C	00000000	Pointer to Line Numbers	
00000240	0000	Number of Relocations	
00000242	0000	Number of Line Numbers	
00000244	C0000040	Characteristics	
	00000040		IMAGE_SCN_CNT_INITIALIZED_DATA
	40000000		IMAGE_SCN_MEM_READ
	80000000		IMAGE_SCN_MEM_WRITE

# 更新相关字段，使病毒代码可以加载到内存

pFile	Data	Description	Value
000001D0	2E 74 65 78	Name	.text
000001D4	74 00 00 00		
000001D8	00020630	Virtual Size	
000001DC	00001000	RVA	
000001E0	00021000	Size of Raw Data	
000001E4	00001000	Pointer to Raw Data	
000001E8	00000000	Pointer to Relocations	
000001EC	00000000	Pointer to Line Numbers	
000001F0	0000	Number of Relocations	
000001F2	0000	Number of Line Numbers	
000001F4	60000020	Characteristics	

**注意：请按实验三中原理进行修改，切勿机械照搬！**

在1D8修改virtualSize为 $20630 + 16 = 0x20646$

testexe.exe*																
000001d0h:	2E	74	65	78	74	00	00	00	46	06	02	00	00	10	00	00
000001e0h:	00	10	02	00	00	10	00	00	00	00	00	00	00	00	00	00
000001f0h:	00	00	00	00	20	00	00	60	2E	72	64	61	74	61	00	00
00000200h:	80	13	00	00	00	20	02	00	00	20	00	00	00	20	02	00

在1F4修改属性为~~E0000020~~

testexe.exe*																
000001f0h:	00	00	00	00	20	00	00	E0	2E	72	64	61	74	61	00	00
00000200h:	80	13	00	00	00	20	02	00	00	20	00	00	00	20	02	00
00000210h:	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	40
00000220h:	2E	64	61	74	61	00	00	00	18	56	00	00	00	40	02	00

# 分析正常的API调用指令

testexe.exe

- IMAGE\_DOS\_HEADER
- MS-DOS Stub Program
- IMAGE\_NT\_HEADERS
  - Signature
  - IMAGE\_FILE\_HEADER
  - IMAGE\_OPTIONAL\_HEADER
  - IMAGE\_SECTION\_HEADER .text
  - IMAGE\_SECTION\_HEADER .rdata
  - IMAGE\_SECTION\_HEADER .data
  - IMAGE\_SECTION\_HEADER .idata
  - IMAGE\_SECTION\_HEADER .reloc
- SECTION .text
- SECTION .rdata
- SECTION .data
- SECTION .idata
- IMPORT Directory Table
- IMPORT Name Table
- IMPORT Address Table

RVA	Data	Description	Value
0002A14C	0002A270	Hint/Name RVA	0296 Sleep
0002A150	0002A278	Hint/Name RVA	001B CloseHandle
0002A154	0002A294	Hint/Name RVA	00CA GetCommandLineA
0002A158	0002A2A6	Hint/Name RVA	0174 GetVersion
0002A15C	0002A2B4	Hint/Name RVA	007D ExitProcess
0002A160	0002A2C2	Hint/Name RVA	0051 DebugBreak
0002A164	0002A2D0	Hint/Name RVA	0152 GetStdHandle
0002A168	0002A2E0	Hint/Name RVA	02DF WriteFile
0002A16C	0002A2EC	Hint/Name RVA	01AD InterlockedDecrement
0002A170	0002A304	Hint/Name RVA	01F5 OutputDebugStringA
0002A174	0002A31A	Hint/Name RVA	013E GetProcAddress
0002A178	0002A32C	Hint/Name RVA	01C2 LoadLibraryA
0002A17C	0002A33C	Hint/Name RVA	01B0 InterlockedIncrement
0002A180	0002A354	Hint/Name RVA	0124 GetModuleFileNameA
0002A184	0002A36A	Hint/Name RVA	029E TerminateProcess
0002A188	0002A37E	Hint/Name RVA	00F7 GetCurrentProcess
0002A18C	0002A392	Hint/Name RVA	02AD UnhandledExceptionFilter
0002A190	0002A3AE	Hint/Name RVA	00B2 FreeEnvironmentStringsA

RVA	Data	Description
000000FC	00000000	Size of Uninitialized Data
00000100	00001190	Address of Entry Point
00000104	00001000	Base of Code
00000108	00001000	Base of Data
0000010C	00400000	Image Base

去导入表IAT找GetCommandLineA的表项的RVA，如图所示，为0002A154  
而exe文件的imageBase为00400000，所以该RVA对应的VA为0042A154  
那么，exe文件调用该API的指令为FF 15 54 A1 42 00 或 FF 25 54 A1 42 00

# 分析正常的API调用指令

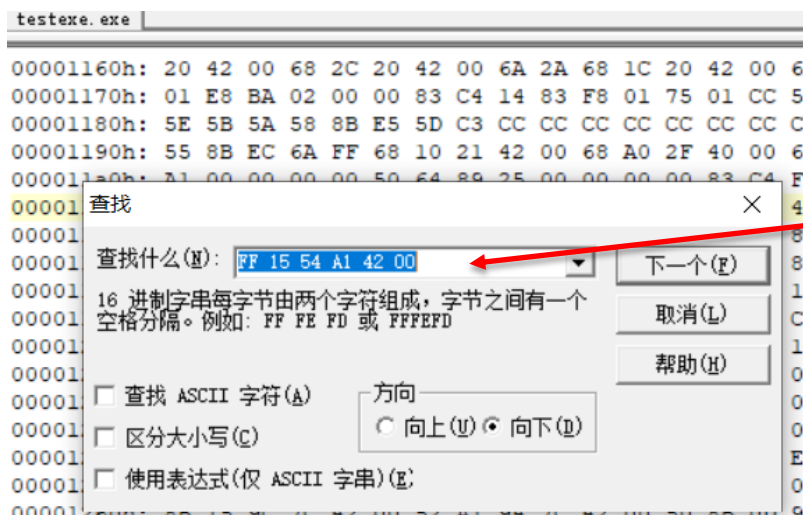
	RVA	Data	Description	Value
testexe.exe				
--IMAGE_DOS_HEADER	0002A14C	0002A270	Hint/Name RVA	0296 Sleep
--MS-DOS Stub Program	0002A150	0002A278	Hint/Name RVA	001B CloseHandle
--IMAGE_NT_HEADERS	0002A154	0002A294	Hint/Name RVA	00CA GetCommandLineA
--Signature	0002A158	0002A2A6	Hint/Name RVA	0174 GetVersion
--IMAGE_FILE_HEADER	0002A15C	0002A2B4	Hint/Name RVA	007D ExitProcess
--IMAGE_OPTIONAL_HEADER	0002A160	0002A2C2	Hint/Name RVA	0051 DebugBreak
--IMAGE_SECTION_HEADER .text	0002A164	0002A2D0	Hint/Name RVA	0152 GetStdHandle
--IMAGE_SECTION_HEADER .rdata	0002A168	0002A2E0	Hint/Name RVA	02DF WriteFile
--IMAGE_SECTION_HEADER .data	0002A16C	0002A2EC	Hint/Name RVA	01AD InterlockedDecrement
--IMAGE_SECTION_HEADER .idata	0002A170	0002A304	Hint/Name RVA	01F5 OutputDebugStringA
--IMAGE_SECTION_HEADER .reloc	0002A174	0002A31A	Hint/Name RVA	013E GetProcAddress
--SECTION .text	0002A178	0002A32C	Hint/Name RVA	01C2 LoadLibraryA
--SECTION .rdata	0002A17C	0002A33C	Hint/Name RVA	01B0 InterlockedIncrement
--SECTION .data	0002A180	0002A354	Hint/Name RVA	0124 GetModuleFileNameA
--SECTION .idata	0002A184	0002A36A	Hint/Name RVA	029E TerminateProcess
--IMPORT Directory Table	0002A188	0002A37E	Hint/Name RVA	00F7 GetCurrentProcess
--IMPORT Name Table	0002A18C	0002A392	Hint/Name RVA	02AD UnhandledExceptionFilter
--IMPORT Address Table	0002A190	0002A3AE	Hint/Name RVA	00B2 FreeEnvironmentStringsA

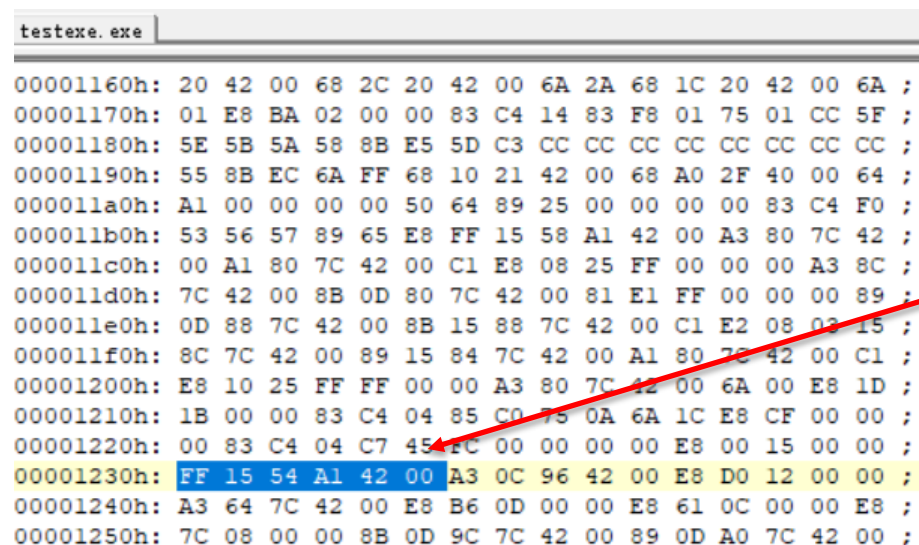
--IMAGE_OPTIONAL_HEADER	000000FC	00000000	Size of Uninitialized Data
--IMAGE_SECTION_HEADER .text	00000100	00001190	Address of Entry Point
--IMAGE_SECTION_HEADER .rdata	00000104	00001000	Base of Code
--IMAGE_SECTION_HEADER .data	00000108	00001000	Base of Data
--IMAGE_SECTION_HEADER .idata	0000010C	00400000	Image Base

**鼓励选用非课件列出的API，为可选加分项，如进行了此步骤，请在实验报告中列出你是如何找到该API的**

# 在文件中查找需要Patch的API调用指令



用UE/C32ASM查找该字符串



找到后, 该字符串的文件位置为 0x1230h

# Patch该API调用指令到病毒寄生代码

-testexe.exe				
- IMAGE_DOS_HEADER				
- MS-DOS Stub Program				
- IMAGE_NT_HEADERS				
- Signature				
- IMAGE_FILE_HEADER				
- IMAGE_OPTIONAL_HEADER				
- IMAGE_SECTION_HEADER .text				
- IMAGE_SECTION_HEADER .rdata				

	RVA	Data	Description	Value
	000001D0	2E 74 65 78	Name	.text
	000001D4	74 00 00 00		
	000001D8	00020646	Virtual Size	
	000001DC	00001000	RVA	
	000001E0	00021000	Size of Raw Data	
	000001E4	00001000	Pointer to Raw Data	
	000001E8	00000000	Pointer to Relocations	
	000001EC	00000000	Pointer to Line Numbers	

由于text节的RVA和PointerToRawData一致，所以API调用指令的RVA就是0x1230h

我们要跳转的病毒寄生位置为1000+20630(原virtualSize)，计算一下所要Patch的跳转指令（跳到病毒开始处）吧

$21630h - 1230h - 5$ （跳转指令自身的指令长度）= 000203FB

因此，我们需要将API调用指令FF 15 54 A1 42 00

改为E8 FB 03 02 00 90

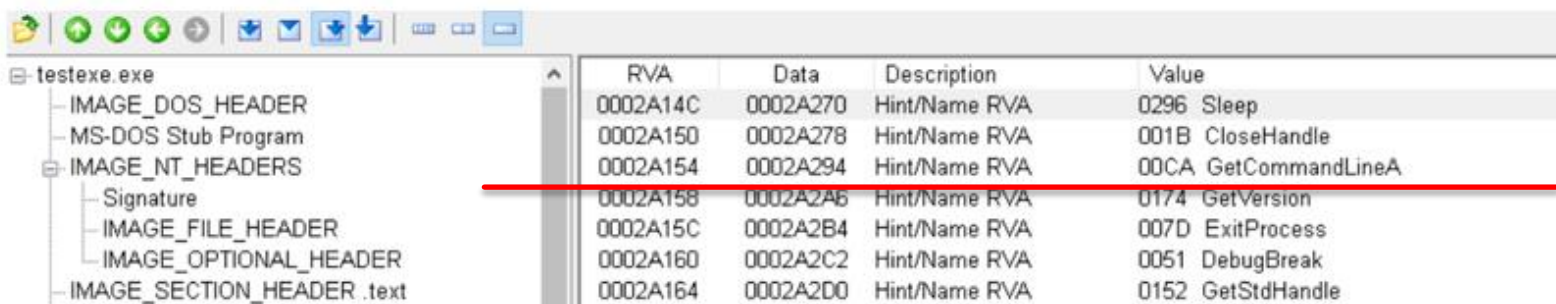


# Patch指令修改

```
-----  
000011f0h: 8C 7C 42 00 89 15 84 7C 42 00 A1 80 7C 42 00 C1 ;  
00001200h: E8 10 25 FF FF 00 00 A3 80 7C 42 00 6A 00 E8 1D ;  
00001210h: 1B 00 00 83 C4 04 85 C0 75 0A 6A 1C E8 CF 00 00 ;  
00001220h: 00 83 C4 04 C7 45 FC 00 00 00 00 E8 00 15 00 00 ;  
00001230h: E8 FB 03 02 00 90 A3 0C 96 42 00 E8 D0 12 00 00 ;  
00001240h: A3 64 7C 42 00 E8 B6 0D 00 00 E8 61 0C 00 00 E8 ;
```

在文件中修改之

如何让病毒尾部的JMP指令跳回原API函数呢？  
我们需要查看加载到内存后的导入表情况



	RVA	Data	Description	Value
testexe.exe				
- IMAGE_DOS_HEADER				
- MS-DOS Stub Program				
- IMAGE_NT_HEADERS				
- Signature				
- IMAGE_FILE_HEADER				
- IMAGE_OPTIONAL_HEADER				
- IMAGE_SECTION_HEADER .text				
	0002A14C	0002A270	Hint/Name RVA	0296 Sleep
	0002A150	0002A278	Hint/Name RVA	001B CloseHandle
	0002A154	0002A294	Hint/Name RVA	00CA GetCommandLineA
	0002A158	0002A2A6	Hint/Name RVA	0174 GetVersion
	0002A15C	0002A2B4	Hint/Name RVA	007D ExitProcess
	0002A160	0002A2C2	Hint/Name RVA	0051 DebugBreak
	0002A164	0002A2D0	Hint/Name RVA	0152 GetStdHandle

GetCommandLineA函数的RVA为2A154



# 在病毒尾部生成跳回原API函数的JMP指令

testexe.exe	RVA	Data	Description	Value
IMAGE_DOS_HEADER	000000F0	010B	Magic	IMAGE_NT_OPTIONAL_HDR32_MAGIC
MS-DOS Stub Program	000000F2	06	Major Linker Version	
IMAGE_NT_HEADERS	000000F3	00	Minor Linker Version	
Signature	000000F4	00021000	Size of Code	
IMAGE_FILE_HEADER	000000F8	0000A000	Size of Initialized Data	
IMAGE_OPTIONAL_HEADER	000000FC	00000000	Size of Uninitialized Data	
IMAGE_SECTION_HEADER .text	00000100	00001190	Address of Entry Point	
IMAGE_SECTION_HEADER .rdata	00000104	00001000	Base of Code	
IMAGE_SECTION_HEADER .data	00000108	00001000	Base of Data	
IMAGE_SECTION_HEADER .idata	0000010C	00400000	Image Base	

查看AddressOfEntryPoint为1190，ImageBase为400000，所以预期加载地址为401190，用OD查看实际入口点就是401190，所以没有重定位

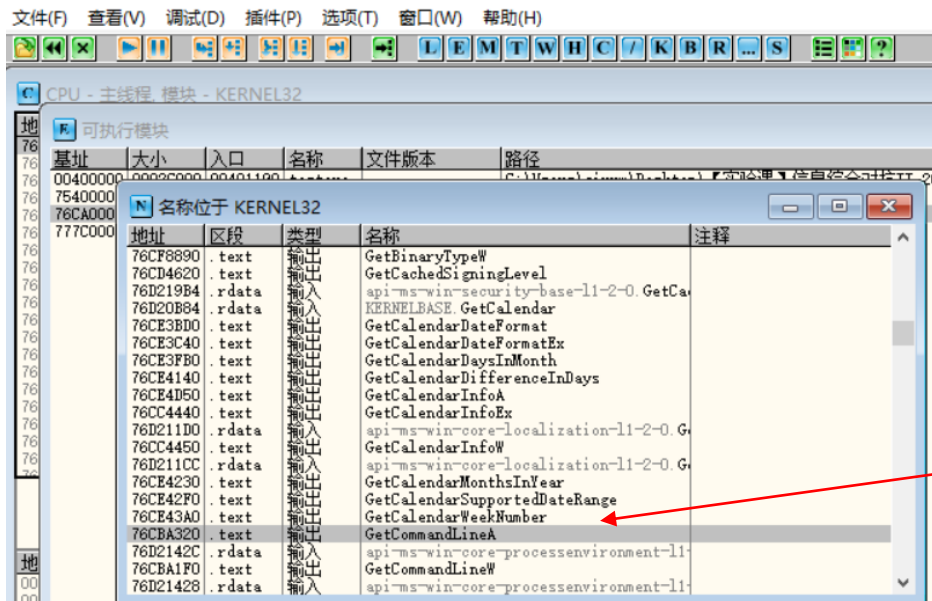
地址	数据	反汇编	注释
00401190	55	PUSH EBP	
00401191	8BEC	MOV EBP, ESP	
00401193	6A FF	PUSH -1	
00401195	68 10214200	PUSH testexe.00422110	
0040119A	68 A02F4000	PUSH testexe.00402FA0	SE 处理程序安装
0040119F	64:A1 00000000	MOV EAX, DWORD PTR FS:[0]	
004011A5	50	PUSH EAX	
004011A6	64:8925 00000000	MOV DWORD PTR FS:[0], ESP	
004011AD	83C4 F0	ADD ESP, -10	
004011B0	53	PUSH EBX	
004011B1	56	PUSH ESI	
004011B2	57	PUSH EDI	
004011B3	8965 E8	MOV DWORD PTR SS:[EBP-18], ESP	
004011B6	FF15 58A14200	CALL DWORD PTR DS:[<&KERNEL32.GetVersion	KERNEL32.GetVersion
004011BC	A3 807C4200	MOV DWORD PTR DS:[427C80], EAX	
004011C1	A1 807C4200	MOV EAX, DWORD PTR DS:[427C80]	
004011C6	C1E8 08	SHR EAX, 8	
004011C9	25 FF000000	AND EAX, 0	
004011CE	A3 8C7C4200	MOV DWORD PTR DS:[427C8C], EAX	
004011D3	8B0B 807C4200	MOV ECX, DWORD PTR DS:[427C80]	

因此，导入表项的内存地址就是  
 $\text{imageBase} + 2A154$   
即42A154

# 在病毒尾部生成跳回原API函数的JMP指令

地址	HEX 数据
0042A154	20 A3 CB 76 60 2D CC 76 90 59 CC 76 50 42 CD
0042A164	E0 9C CB 76 C0 43 CC 76 30 1C CC 76 80 F0 CB
0042A174	20 5F CB 76 40 2A CC 76 40 1B CC 76 D0 97 CB
0042A184	D0 F4 CB 76 90 3C CC 76 10 62 CD 76 40 45 CD
0042A194	90 A0 CB 76 50 46 CB 76 A0 47 CD 76 30 A3 CB
0042A1A4	50 2F CC 76 A0 41 CC 76 B0 20 CC 76 00 90 CB
0042A1B4	40 A3 CB 76 80 9D CB 76 80 2A CC 76 00 9A CB
0042A1C4	70 1A CB 76 F0 5E CB 76 F0 7D CB 76 70 46 CB
0042A1D4	70 47 CC 76 F0 A6 CB 76 70 41 CB 76 90 C4 CB
0042A1E4	00 9B CB 76 00 7D CB 76 30 30 CC 76 30 F9 80

用x64dbg/OD定位到内存42A154的位置，其数据76CBA320就是API函数的入口地址（不同系统该地址也不同）



定位到kernel32.dll，查看名称找API函数到GetCommandLineA，其入口地址就是76CBA320（不同系统该地址也不同）

# 在病毒尾部生成跳回原API函数的JMP指令

寄生病毒代码的最后一条跳回API的JMP指令的偏移量计算如下：

-testexe.exe				
-- IMAGE_DOS_HEADER				
-- MS-DOS Stub Program				
-- IMAGE_NT_HEADERS				
-- Signature				
-- IMAGE_FILE_HEADER				
-- IMAGE_OPTIONAL_HEADER				
-- <b>IMAGE_SECTION_HEADER .text</b>				
-- IMAGE_SECTION_HEADER .rdata				

	RVA	Data	Description	Value
	000001D0	2E 74 65 78	Name	.text
	000001D4	74 00 00 00		
	000001D8	00020646	Virtual Size	
	000001DC	00001000	RVA	
	000001E0	00021000	Size of Raw Data	
	000001E4	00001000	Pointer to Raw Data	
	000001E8	00000000	Pointer to Relocations	
	000001EC	00000000	Pointer to Line Numbers	

跳转的目的地址为API函数的入口VA： 76CBA320

跳转的源地址为病毒寄生代码的尾部，即为新的VirtualSize + 节RVA + ImageBase = 421646

跳转的偏移是 = 76CBA320 - 421646 = 76 89 8C DA

所以，病毒代码中最后一条跳转回API函数的指令为：  
E9 DA 8C 89 76

# 在病毒尾部生成跳回原API函数的JMP指令

寄生病毒代码的最后一条跳回API的JMP指令的偏移量计算如下：

```
00021630h: E8 04 00 00 00 90 90 90 90 58 bb 11 11 11 11 89 ;
00021640h: 18 e9 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
00021650h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ;
```

找到文件的位置，修改之

testexe.exe\*

```
00021620h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00021630h: E8 04 00 00 00 90 90 90 90 58 BB 11 11 11 11 89
00021640h: 18 E9 DA 8C 89 76 00 00 00 00 00 00 00 00 00 00
```

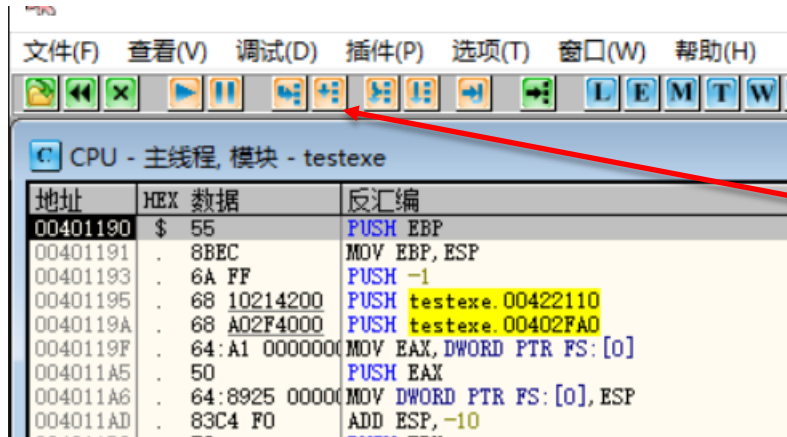
# 调试病毒 观察病毒加载情况

CPU - 主线程, 模块 - testexe		
地址	HEX 数据	反汇编
00401190	55	PUSH EBP
00401191	8BEC	MOV EBP, ESP
00401193	6A FF	PUSH -1
00401195	68 10214200	PUSH testexe.00422110
0040119A	68 A02F4000	PUSH testexe.00402FA0
0040119F	64:A1 00000000	MOV EAX, DWORD PTR FS:[0]
004011A5	50	PUSH EAX
004011A6	64:8925 00000000	MOV DWORD PTR FS:[0], ESP
004011AD	83C4 F0	ADD ESP, -10
004011B0	53	PUSH EBX
004011B1	56	PUSH ESI
004011B2	57	PUSH EDI
004011B3	8965 E8	MOV DWORD PTR SS:[EBP-18], ESP
004011B6	FF15 58A14200	CALL DWORD PTR DS:[<&KERNEL32.G
004011BC	A3 807C4200	MOV DWORD PTR DS:[427C80], EAX
004011C1	A1 807C4200	MOV EAX, DWORD PTR DS:[427C80]
004011C6	C1E8 08	SHR EAX, 8
004011C9	25 FF000000	AND EAX, 0FF
004011CE	A3 8C7C4200	MOV DWORD PTR DS:[427C8C], EAX
004011D3	8B0B 807C4200	MOV ECX, DWORD PTR DS:[427C80]
EBP=0019FF80		
地址	HEX 数据	
00421630	E8 04 00 00	00 90 90 90 90 58 DD 11 11 11 11 89
00421640	18 E9 DA 8C	89 76 00 00 00 00 00 00 00 00 00 00
00421650	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00
00421660	00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00

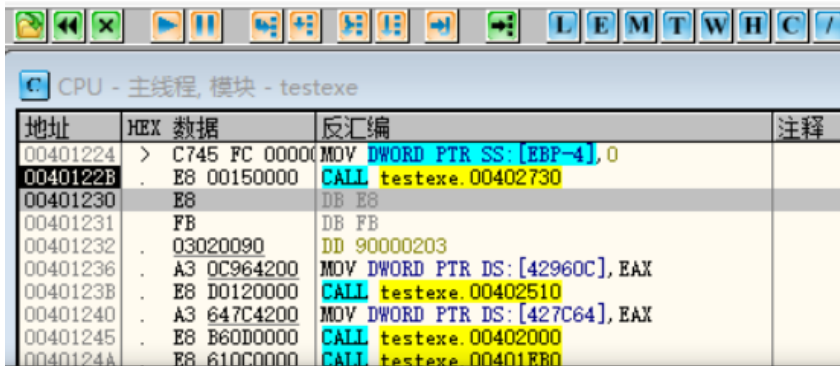
用x64dbg/OD启动程序。  
先定位到病毒寄生的位置421630，查看下我们病毒代码的加载情况

如果病毒执行成功，将会把4个90修改为4个11（注意替换为学号后8位）

# 定位到Patch的指令之前



让OD/x64dbg单步步过的方式执行到401230我们篡改的指令



原程序的调试提示，选择是

的断点



看上去您想在数据上设置断点.如果真是这样的话, 这些断点将不会执行并可能严重影响调试的程序. 您真的希望在此设置断点吗?

是(Y)

否(N)



# 执行被Patch的指令

文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H)

CPU - 主线程, 模块 - testexe

地址	HEX 数据	反汇编
00401224	> C745 FC 0000	MOV DWORD PTR SS:[EBP-4], 0
0040122B	. E8 00150000	CALL testexe.00402730
00401230	. E8	DB E8
00401231	. FB	DB FB
00401232	. 03020090	DD 90000203
00401236	. A3 0C964200	MOV DWORD PTR DS:[42960C], EAX
0040123B	. E8 D0120000	CALL testexe.00402510
00401240	. A3 647C4200	MOV DWORD PTR DS:[427C64], EAX
00401245	. E8 B60D0000	CALL testexe.00402000
0040124A	. E8 610C0000	CALL testexe.00401EB0
0040124F	. E8 7C080000	CALL testexe.00401AD0
00401254	. 8B0D 9C7C4200	MOV ECX, DWORD PTR DS:[427C9C]
0040125A	. 890D A07C4200	MOV DWORD PTR DS:[427CA0], ECX
00401260	. 8B15 9C7C4200	MOV EDX, DWORD PTR DS:[427C9C]
00401266	. 52	PUSH EDX

这里继续单步步过，但是实际已经跳转到我们寄生的病毒代码了！

CPU - 主线程, 模块 - testexe

地址	HEX 数据	反汇编	注释
00401235	? 90	NOP	
00401236	. A3 0C964200	MOV DWORD PTR DS:[42960C], EAX	
0040123B	. E8 D0120000	CALL testexe.00402510	
00401240	. A3 647C4200	MOV DWORD PTR DS:[427C64], EAX	
00401245	. E8 B60D0000	CALL testexe.00402000	
0040124A	. E8 610C0000	CALL testexe.00401EB0	
0040124F	. E8 7C080000	CALL testexe.00401AD0	
00401254	. 8B0D 9C7C4200	MOV ECX, DWORD PTR DS:[427C9C]	
0040125A	. 890D A07C4200	MOV DWORD PTR DS:[427CA0], ECX	
00401260	. 8B15 9C7C4200	MOV EDX, DWORD PTR DS:[427C9C]	
00401266	. 52	PUSH EDX	
00401267	. A1 947C4200	MOV EAX, DWORD PTR DS:[427C94]	
0040126C	. 50	PUSH EAX	

单步步过后，返回了401235（Patch指令的最后一个字节NOP

说明已成功执行了病毒和原API函数，然后通过原API函数返回了

## 观察病毒代码的执行情况

地址	HEX 数据	A
00421630	E8 04 00 00 00 11 11 11 11 58 BB 11 11 11 11 89	?
00421640	18 E9 DA 8C 89 76 00 00 00 00 00 00 00 00 00	C
00421650	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.
00421660	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.
00421670	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.
00421680	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.
00421690	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.
004216A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.
004216B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.

最后定位到寄生的病毒代码处，查看下病毒代码执行的情况，可看见4个90已经被你的学号后8位所替代了（演示时使用八个1）

实验要求：复现以上实验过程（注意学号）