



第2讲. 软件安全基础知识

网络空间安全学院 付才

Mail: fucai@hust.edu.cn

QQ:5146279

提纲

2.1 系统引导与控制权

2.2 80X86处理器的工作模式

2.3 Windows内存结构与管理

2.4 磁盘的物理与逻辑结构

2.5 FAT32文件系统

2.6 PE文件格式

2.7 破解实践

2.1 系统引导与控制权

- 系统引导与恶意软件有何关系？

恶意软件如何再次获得控制权？

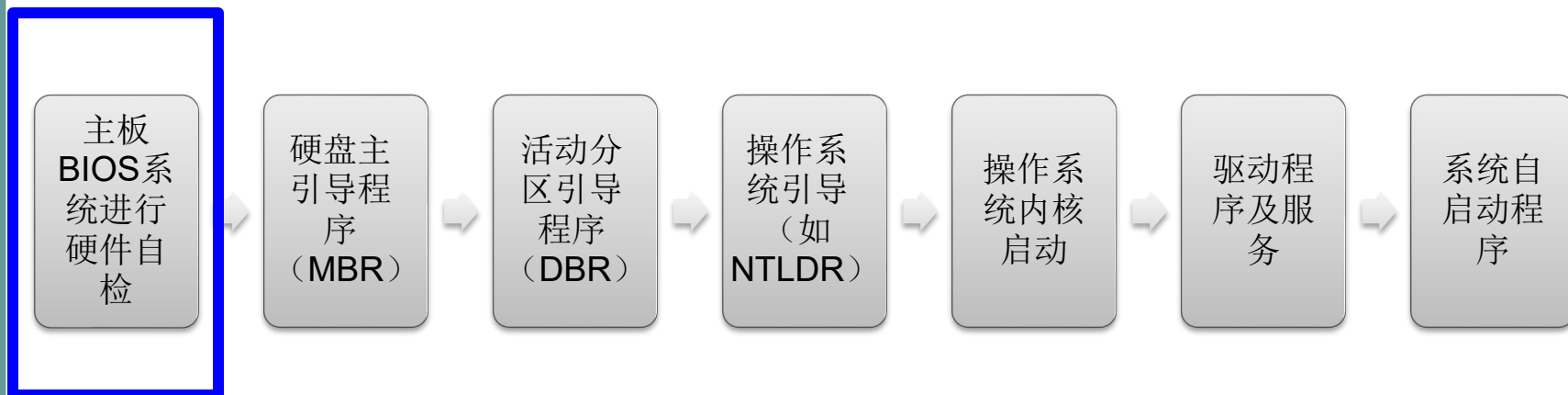
自身被结
束之后

操作系统
重启之后

操作系统
重装之后

硬盘更换
之后。。。

2.1.1 计算机系统引导过程



BIOS: Basic Input and Output System

- “基本输入输出系统”，存储在主板**BIOS Flash**（或**ROM**）芯片。
- 为计算机提供最底层的、最直接的硬件设置和控制。



BIOS的自检与初始化工作

- 任务：检测系统中的一些关键设备（如内存和显卡等）是否存在和能否正常工作，进行初始化，并将控制权交给后续引导程序。
 - 显卡及其他相关设备初始化。
 - 显示系统**BIOS**启动画面，其中包括有系统**BIOS**的类型、序列号和版本号等内容。
 - 检测**CPU**的类型和工作频率，内存容量、并将检测结果显示在屏幕上。
 - 检测系统中安装的一些标准硬件设备及即插即用设备，这些设备包括：硬盘、**CD-ROM**、软驱、串行接口和并行接口等。
 - 根据用户指定的启动顺序从软盘、硬盘或光驱启动。
 - 如果从硬盘启动，则将控制权交给硬盘主引导程序。

系统自检



硬盘主引导程序

- 所在位置：
 - **MBR, Master Boot Record**, 硬盘第一个扇区。
- 主要功能：
 - 通过主分区表中定位活动分区
 - 装载活动分区的引导程序，并移交控制权。

活动分区引导程序

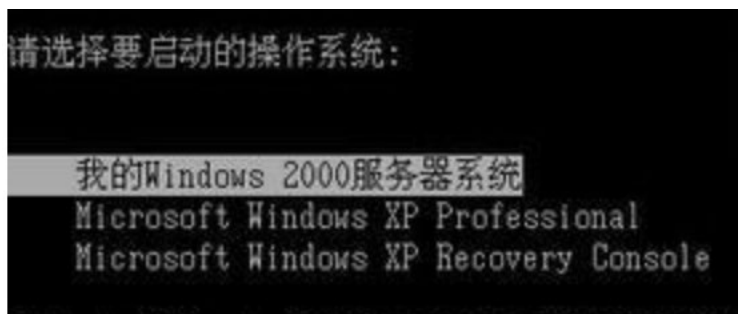
- 所在位置：
 - **DBR (DOS Boot Record)**，或称**OBR (OS Boot Record)**，或称分区引导记录 (**PBR, Partition Boot Record**)
 - 分区的第一个扇区
- 功能：
 - 加载操作系统引导程序
 - 如**Windows XP**系统的**NTLDR**
 - **Windows10**系统的**bootmgr**



```
NTLDR is missing  
Press Ctrl+Alt+Del to restart  
-
```

操作系统引导—以Windows NTLDR为例

- 将处理器从**16位**内存模式拓展为**32位**（**64位**）内存模式
- 启动小型文件系统驱动，以识别**FAT32**和**NTFS**文件系统
- 读取**boot.ini**，进行多操作系统选择（或**hiberfil.sys**恢复休眠）
- 检测和配置硬件（**NT**或**XP**系统，则运行**NTDETECT.COM**，其将硬件信息提交给**NTLDR**，写入“**HKEY_LOCAL_MACHINE**”中的**Hardware**中）



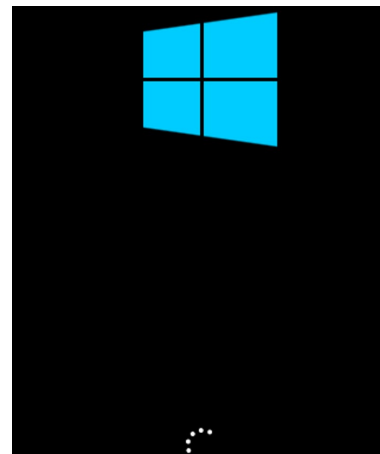
系统内核加载

- **NTLDR**加载内核程序**NTOSKRNL.EXE**以及硬件抽象层**HAL.dll**等。
- 读取并加载**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet** 下指定的驱动程序。
- **NTLDR**将把控制权传递给**NTOSKRNL.EXE**，至此引导过程将结束。

Windows系统装载

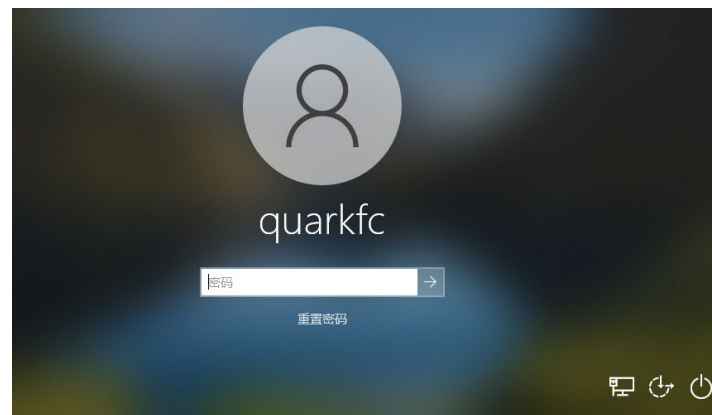
1. 创建系统环境变量
2. 启动win32.sys（Windows子系统的内核模式部分）。
3. 启动csrss.exe（Windows子系统的用户模式部分）。
4. 启动winlogon.exe等

屏幕显示：Windows logo 界面和进度条



Windows系统装载—登陆阶段

1. 启动需要自动启动的**Windows**服务
2. 启动本地安全认证**Lsass.exe**
3. 显示登录界面等



Windows登陆之后

- 系统启动当前用户环境下的自启动项程序
 - 注册表特定键值
 - 特定目录（如**startup**）等
- 用户触发和执行各类应用程序
 - 如**IE**、**QQ**、**Office**等

Windows系统引导过程 (win7和win10部分步骤不同)

1. 加电, 主板**BIOS**自检程序开始运行
2. 硬盘主引导记录被装入内存, 主引导程序开始执行
3. 活动分区的引导扇区被装入内存并执行, **NTLDR**从引导扇区被装入并初始化
4. **NTLDR**将处理器的从**16位**实模式改为**32位平滑内存模式**
5. **NTLDR**加载小文件系统驱动程序。
6. **NTLDR**读**boot.ini**文件, 用户选择操作系统。
7. **NTLDR**装载所选操作系统
8. **Ntdetect.com** 搜索计算机硬件并将列表传送给**NTLDR**, 以便将这些信息写进 **\HKEY_LOCAL_MACHINE\HARDWARE**中。
9. **NTLDR**装载**Ntoskrnl.exe**, **Hal.dll**和系统信息集合。
10. **Ntldr**搜索系统信息集合, 并装载设备驱动。
11. **Ntldr**把控制权交给**Ntoskrnl.exe**, 这时,启动程序结束
12. **Windows**开始装载
13. 执行驱动程序及服务
14. 系统执行自启动程序
15. 用户触发执行程序

2.1.2 系统引导与恶意软件的关联

- 系统引导与恶意软件有何关系？
 - 恶意软件在植入系统之后，如何再次获得控制权？
 - 在计算机系统引导阶段获得控制权
 - **Bootkit**: BIOS木马、MBR木马等，可用于长期驻留在系统；早期的DOS引导区病毒等。
 - **CIH病毒**
 - 在操作系统启动阶段获得控制权
 - 最常见的恶意软件启动方法，多见于独立的恶意软件程序。
 - 在应用程序执行阶段获得控制权
 - 最常见的文件感染型病毒启动方法。

提纲

2.1 系统引导与控制权

2.2 80X86处理器的工作模式

2.3 Windows内存结构与管理

2.4 磁盘的物理与逻辑结构

2.5 FAT32文件系统

2.6 PE文件格式

2.7 破解实践

2.2 80X86处理器的工作模式

- 80X86处理器支持**3**种工作模式：**实模式**、**保护模式**和**虚拟8086模式**。
 - 实模式和虚拟8086模式是为了向下兼容8086处理器的程序而设计。

实模式

- **80X86处理器在复位或加电时**是以实模式启动的。
- **寻址方式：20位寻址（段+偏移），1M空间。**
- **不能对内存进行分页管理。**
- **不支持优先级**，所有的指令相当于工作在特权级(优先级0)。
- **切换到保护模式：**通过在实模式下初始化控制寄存器，**GDTR**，**LDTR**等管理寄存器以及页表，然后再置位**CR0**寄存器的保护模式使能位（**PE: Protected-Mode Enable**，第0位）。

保护模式

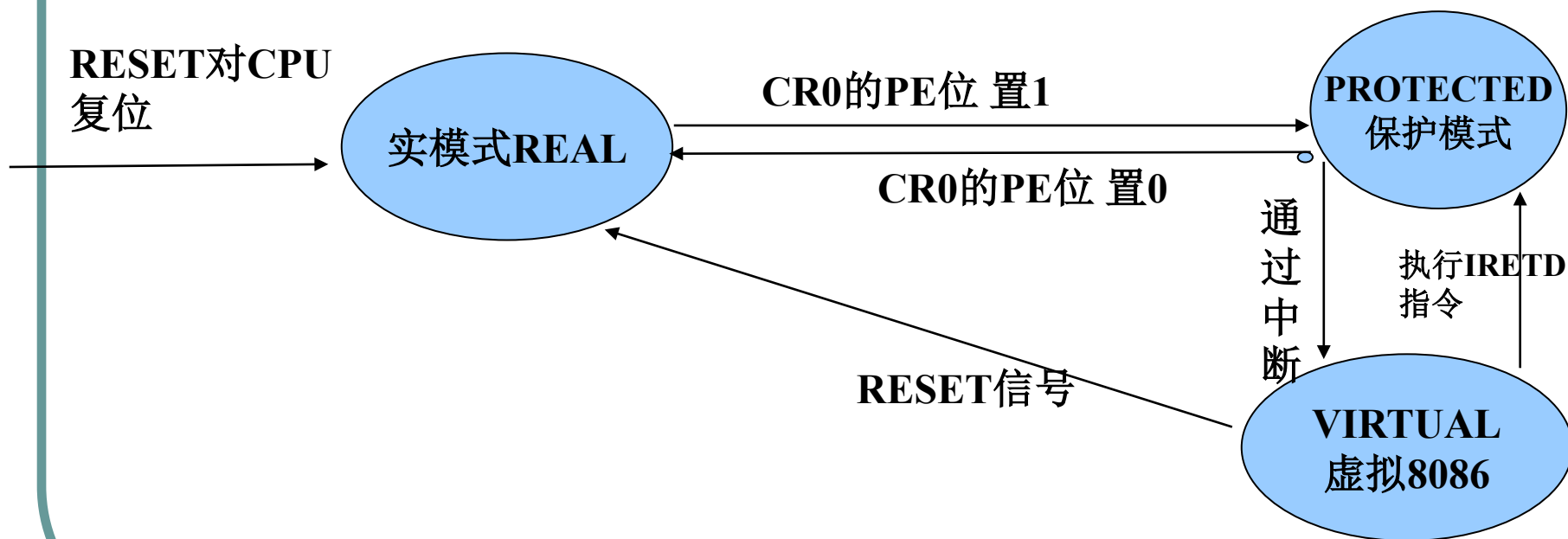
- 是80X86处理器的**常态工作模式**；
- 32位处理器**支持32位寻址**，物理寻址空间达**4G**。
- **支持内存分页机制**，提供了对虚拟内存的良好支持；
- **支持优先级机制**，根据任务特性进行了**运行环境隔离**；
- **切换到实模式**：通过修改控制寄存器**CR0**的**PE**位（**Protected-Mode Enable**，第0位），切换到实模式。

虚拟8086模式

- 为了在保护模式下兼容8086程序而设置的。
- 虚拟8086模式是以任务的形式在保护模式上执行的，在80X86上可以同时支持多个真正的80X86任务和虚拟8086模式构成的任务。
- 支持任务切换和内存分页。
 - 操作系统用分页机制将不同的虚拟8086任务的地址空间映射到不同的物理地址上面去，使得每个虚拟8086任务看来都认为自己在使用0~1MB的地址空间。

Intel80X86处理器三种工作模式关系：

实模式、保护模式和虚拟86模式



提纲

2.1 系统引导与控制权

2.2 80X86处理器的工作模式

2.3 Windows内存结构与管理

2.4 磁盘的物理与逻辑结构

2.5 FAT32文件系统

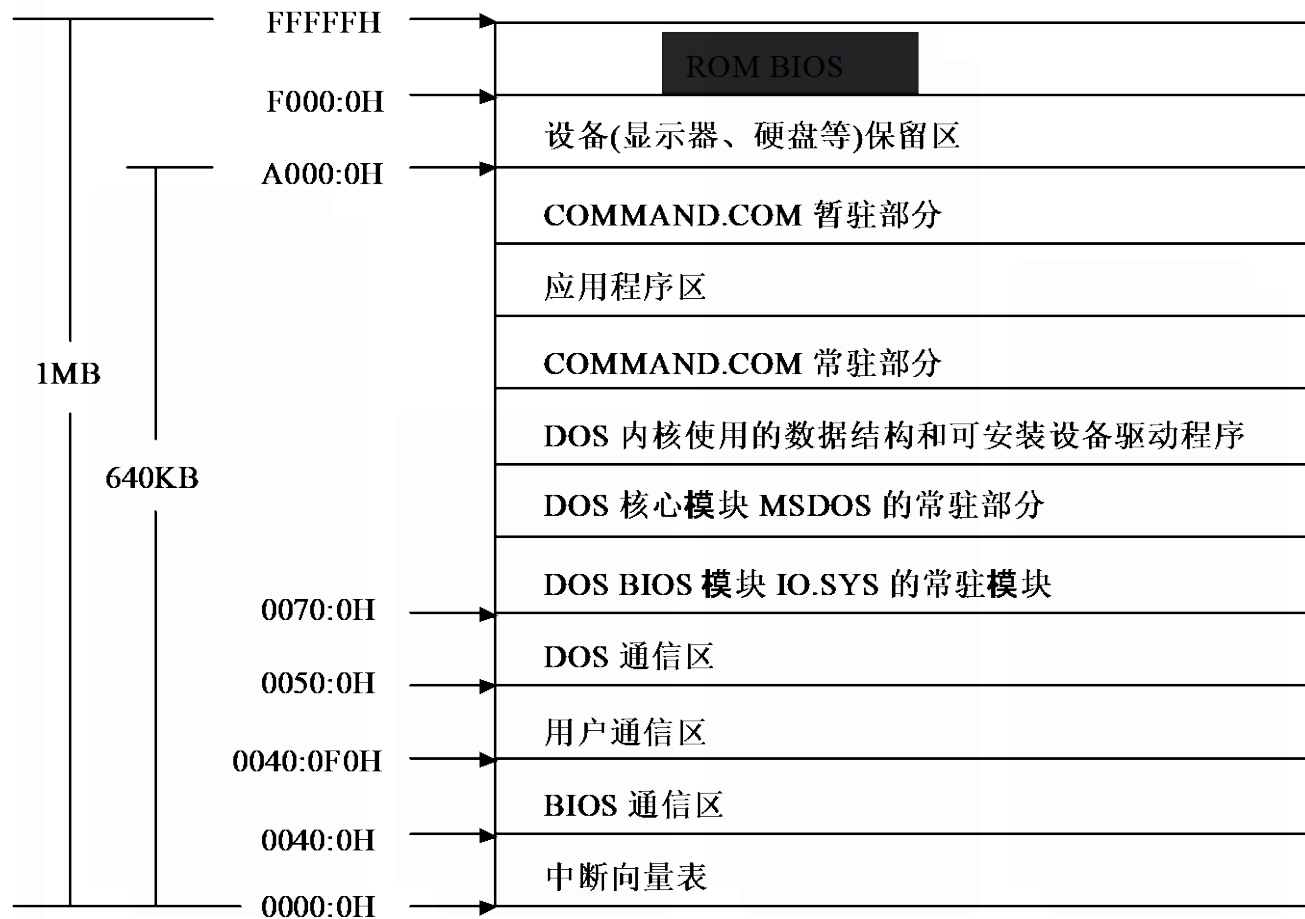
2.6 PE文件格式

2.7 破解实践

2.3 Windows内存结构与管理

- **DOS实模式下的内存布局**
- **Windows下的虚拟地址空间布局**
- **虚拟地址与物理地址的转换**

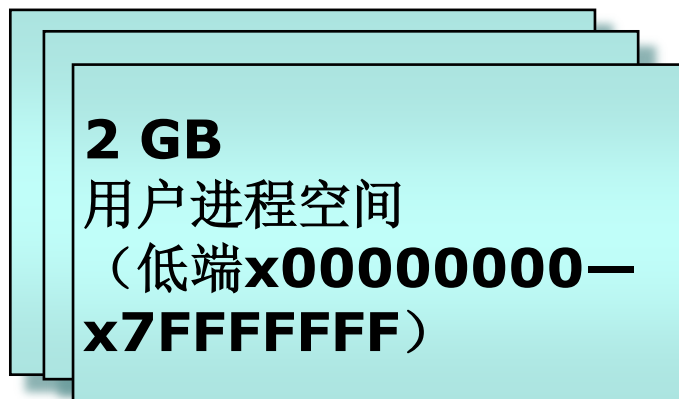
DOS实模式下的内存布局



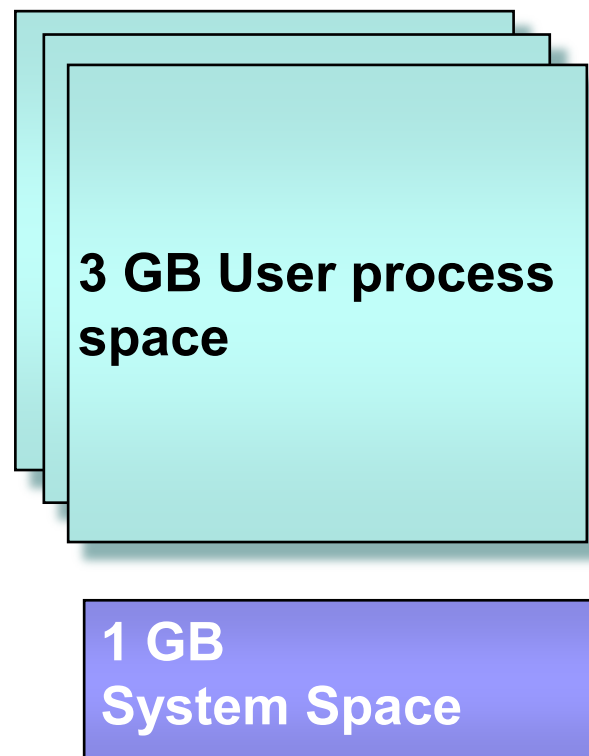
Windows虚拟地址空间

(32-bit x86 虚拟地址空间最大为4GB)

Default



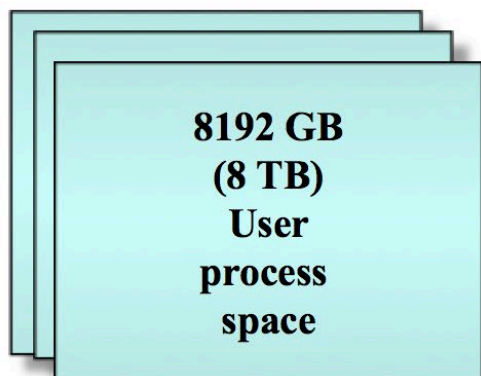
3 GB user space



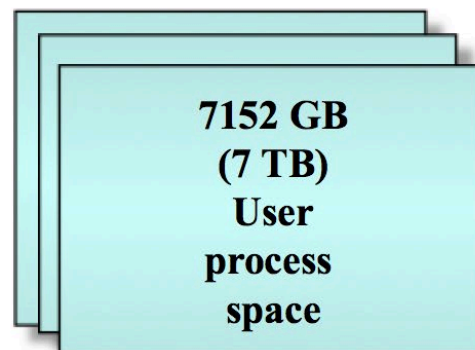
Windows 虚拟地址空间（64-bit 处理器）

64-bit Address Spaces

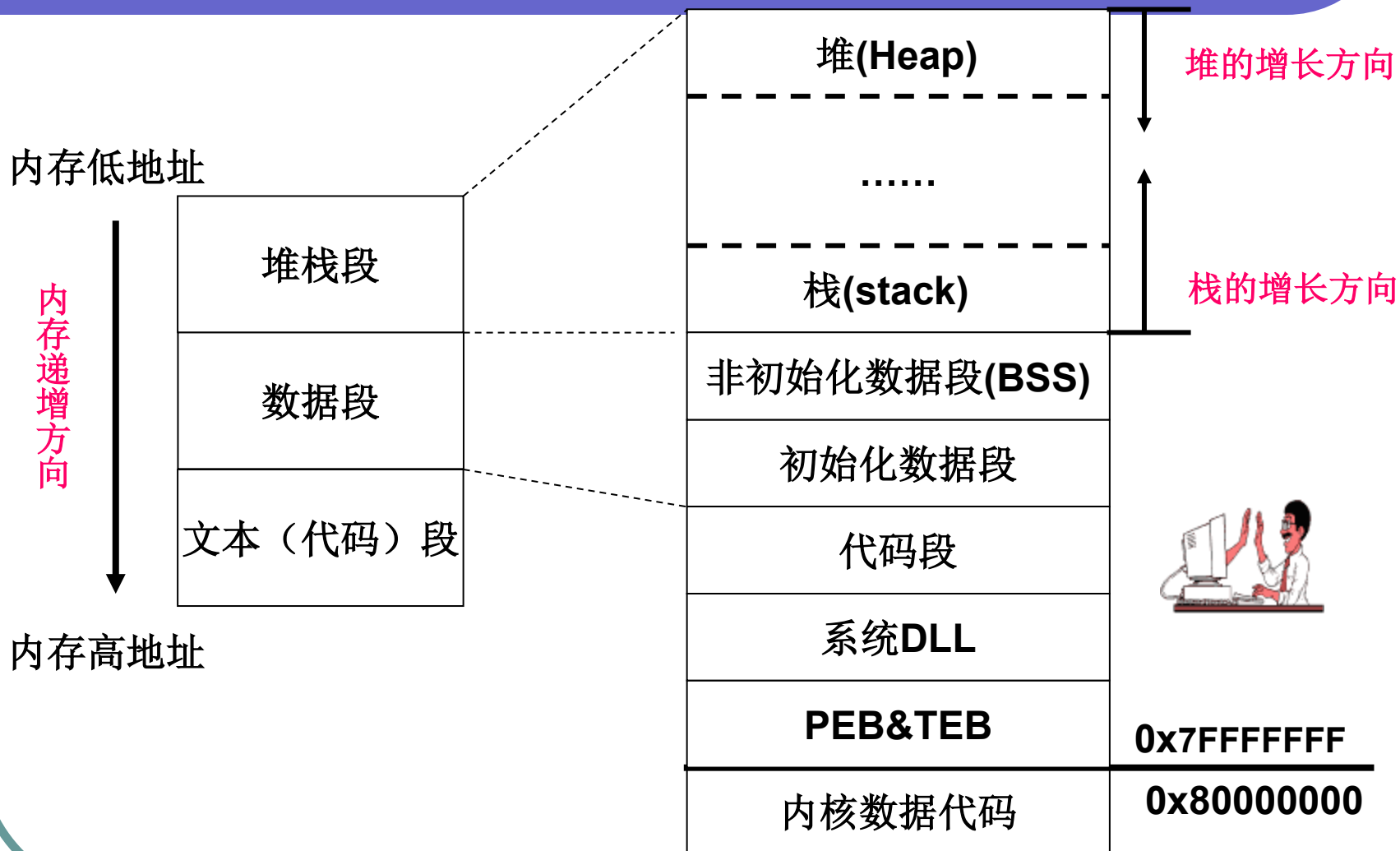
x64



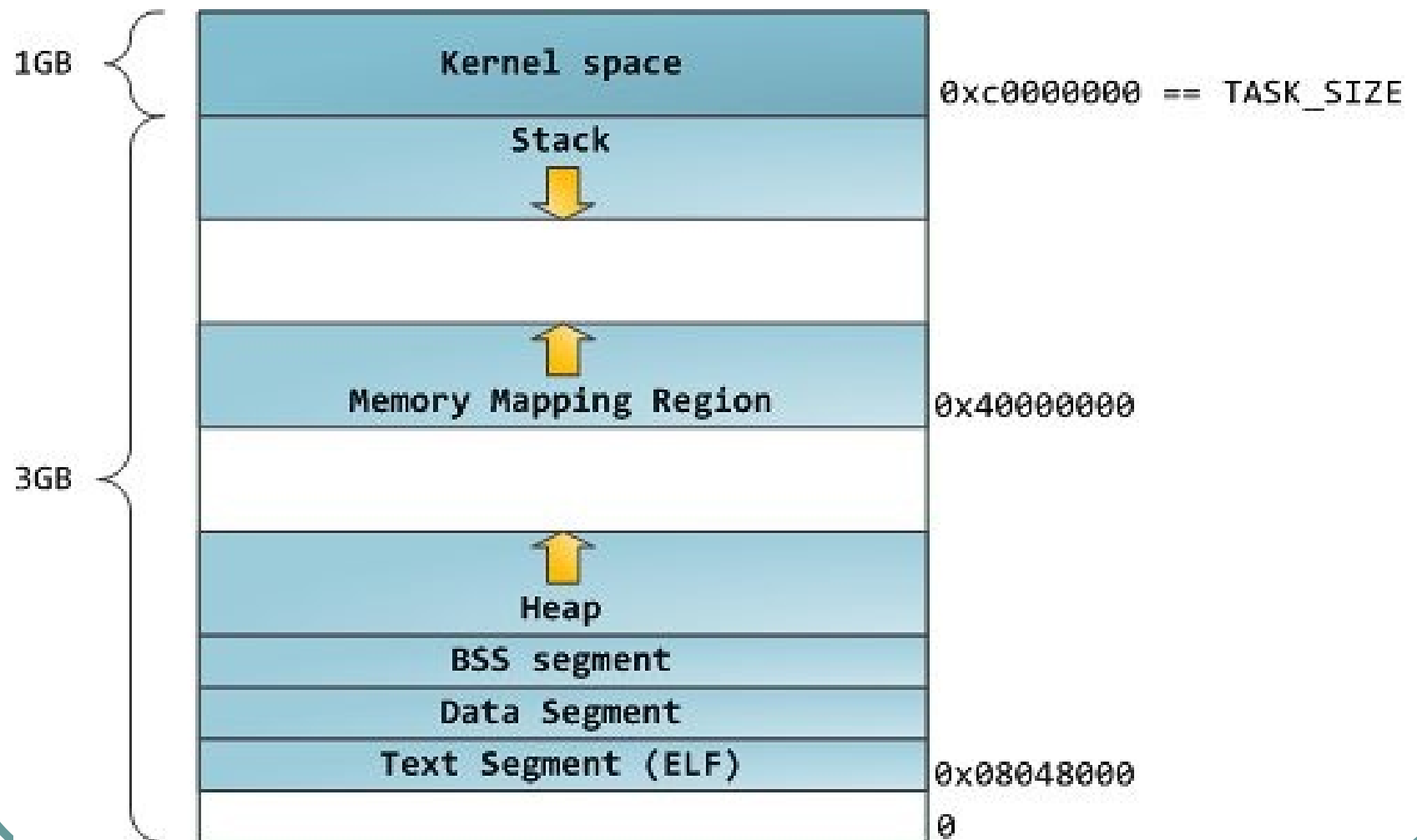
Itanium



程序在内存中的映像整体情况(32位)



程序在内存中的映像 (Linux)



CPU特权级与内存访问

- 与进程虚拟内存中用户模式区和内核模式区相对应，**Windows**为了确保系统的稳定性，将处理器存取模式划分为用户模式（**Ring 3**）和内核模式（**Ring 0**）。
 - **用户应用程序**一般运行在用户模式。
 - 其访问空间局限于用户区；
 - **操作系统内核代码**（如系统服务和设备驱动程序等）运行在内核模式。
 - 可以访问所有的内存空间（包括用户模式分区）和硬件，可使用所有处理器指令。

用户区内存

- 用户区是每个进程真正独立的可用内存空间，进程中的绝大部分数据都保存在这一区域。
 - 主要包括：应用程序代码、全局变量、所有线程的线程栈以及加载的DLL代码等
- 每个进程的用户区的虚拟内存空间相互独立，一般不可以直接跨进程访问，这使得一个程序直接破坏另一个程序的可能性非常小。

Notepad进程内存的用户区

地址	大小	属主	区段	包含	类型	访问
00010000	00001000				Priv 00021004	RW
00020000	00001000				Priv 00021004	RW
00030000	00005000				Priv 00021004	RW
00060000	00001000				Priv 00021104	RW
0006F000	00011000				Priv 00021104	RW
00080000	00003000				Map 00041002	R
00090000	00002000				Map 00041002	R
000A0000	00009000				Priv 00021004	RW
001A0000	00006000				Priv 00021004	RW
001B0000	00003000				Map 00041004	RW
001C0000	00016000				Map 00041002	R
001E0000	00041000				Map 00041002	R
00230000	00041000				Map 00041002	R
00280000	00006000				Map 00041002	R
00290000	00041000				Map 00041002	R
002E0000	0000E000				Map 00041020	R E
003A0000	00002000				Map 00041020	R E
003B0000	00008000				Priv 00021004	RW
003C0000	00001000				Priv 00021004	RW
003D0000	00001000				Priv 00021004	RW
003E0000	00002000				Map 00041002	R
003F0000	00002000				Map 00041002	R
00400000	00003000				Priv 00021004	RW
00410000	00008000				Priv 00021004	RW
00420000	00004000				Priv 00021004	RW
00430000	00003000				Map 00041002	R
00440000	00003000				Priv 00021004	RW
00480000	00103000				Map 00041002	R
00590000	00123000				Map 00041020	R E
00890000	00001000				Priv 00021004	RW
01000000	00001000	notepad		PE 文件头	Imag 01001002	R
01001000	00008000	notepad	.text	代码, 输入表, 输出表	Imag 01001002	R
01009000	00002000	notepad	.data	数据	Imag 01001002	R
0100B000	00008000	notepad	.rsrc	资源	Imag 01001002	R
58FB0000	00001000	AcGenral		PE 文件头	Imag 01001002	R
58FB1000	00032000	AcGenral	.text	代码, 输入表, 输出表	Imag 01001002	R
58FE3000	00009000	AcGenral	.data	数据	Imag 01001002	R
58FEC000	00188000	AcGenral	.rsrc	资源	Imag 01001002	R
59174000	00006000	AcGenral	.reloc	重定位	Imag 01001002	R
5ADC0000	00001000	UxTheme		PE 文件头	Imag 01001002	R
5ADC1000	00030000	UxTheme	.text	代码, 输入表, 输出表	Imag 01001002	R
5ADF1000	00001000	UxTheme	.data	数据	Imag 01001002	R
5ADF2000	00003000	UxTheme	.rsrc	资源	Imag 01001002	R
5ADF5000	00002000	UxTheme	.reloc	重定位	Imag 01001002	R
5CC30000	00001000	ShimEng		PE 文件头	Imag 01001002	R
5CC31000	0000E000	ShimEng	.text	代码, 输入表, 输出表	Imag 01001002	R
5CC3F000	00014000	ShimEng	.data	数据	Imag 01001002	R
5CC53000	00001000	ShimEng	.rsrc	资源	Imag 01001002	R
5CC54000	00002000	ShimEng	.reloc	重定位	Imag 01001002	R
62C20000	00001000	LPK		PE 文件头	Imag 01001002	R
62C21000	00005000	LPK	.text	代码, 输入表, 输出表	Imag 01001002	R
62C26000	00001000	LPK	.data	数据	Imag 01001002	R
62C27000	00001000	LPK	.rsrc	资源	Imag 01001002	R

地址	大小	属主	区段	包含	类型	访问
77C35000	00003000	msvcrt	.reloc	重定位	Imag 01001002	R
77D10000	00001000	USER32		PE 文件头	Imag 01001002	R
77D11000	00060000	USER32	.text	代码, 输入表, 输出表	Imag 01001002	R
77D71000	00002000	USER32	.data	数据	Imag 01001002	R
77D73000	0002A000	USER32	.rsrc	资源	Imag 01001002	R
77D9D000	00003000	USER32	.reloc	重定位	Imag 01001002	R
77DA0000	00001000	ADVAPI32		PE 文件头	Imag 01001002	R
77DA1000	00075000	ADVAPI32	.text	代码, 输入表, 输出表	Imag 01001002	R
77E16000	00005000	ADVAPI32	.data	数据	Imag 01001002	R
77E1B000	00029000	ADVAPI32	.rsrc	资源	Imag 01001002	R
77E44000	00005000	ADVAPI32	.reloc	重定位	Imag 01001002	R
77E50000	00001000	RPCRT4		PE 文件头	Imag 01001002	R
77E51000	00084000	RPCRT4	.text	代码, 输入表, 输出表	Imag 01001002	R
77ED5000	00007000	RPCRT4	.orpc	代码	Imag 01001002	R
77EDC000	00001000	RPCRT4	.data	数据	Imag 01001002	R
77EDD000	00001000	RPCRT4	.rsrc	资源	Imag 01001002	R
77EDE000	00005000	RPCRT4	.reloc	重定位	Imag 01001002	R
77EF0000	00001000	GDI32		PE 文件头	Imag 01001002	R
77EF1000	00043000	GDI32	.text	代码, 输入表, 输出表	Imag 01001002	R
77F34000	00002000	GDI32	.data	数据	Imag 01001002	R
77F36000	00001000	GDI32	.rsrc	资源	Imag 01001002	R
77F37000	00002000	GDI32	.reloc	重定位	Imag 01001002	R
77F40000	00001000	SHLWAPI		PE 文件头	Imag 01001002	R
77F41000	0006C000	SHLWAPI	.text	代码, 输入表, 输出表	Imag 01001002	R
77FAD000	00001000	SHLWAPI	.data	数据	Imag 01001002	R
77FAE000	00002000	SHLWAPI	.rsrc	资源	Imag 01001002	R
77FB0000	00006000	SHLWAPI	.reloc	重定位	Imag 01001002	R
77FC0000	00001000	Secur32		PE 文件头	Imag 01001002	R
77FC1000	0000D000	Secur32	.text	代码, 输入表, 输出表	Imag 01001002	R
77FCE000	00001000	Secur32	.data	数据	Imag 01001002	R
77FCF000	00001000	Secur32	.rsrc	资源	Imag 01001002	R
77FD0000	00001000	Secur32	.reloc	重定位	Imag 01001002	R
7C800000	00001000	kernel32		PE 文件头	Imag 01001002	R
7C801000	00084000	kernel32	.text	代码, 输入表, 输出表	Imag 01001002	R
7C850000	00005000	kernel32	.data	数据	Imag 01001002	R
7C88A000	0008E000	kernel32	.rsrc	资源	Imag 01001002	R
7C918000	00006000	kernel32	.reloc	重定位	Imag 01001002	R
7C920000	00001000	ntdll		PE 文件头	Imag 01001002	R
7C921000	0007D000	ntdll	.text	代码, 输出表	Imag 01001002	R
7C99E000	00005000	ntdll	.data	数据	Imag 01001002	R
7C9A3000	00010000	ntdll	.rsrc	资源	Imag 01001002	R
7C9B3000	00003000	ntdll	.reloc	重定位	Imag 01001002	R
7D590000	00001000	SHELL32		PE 文件头	Imag 01001002	R
7D591000	001FF000	SHELL32	.text	代码, 输入表, 输出表	Imag 01001002	R
7D790000	0001D000	SHELL32	.data	数据	Imag 01001002	R
7D7AD000	005BD000	SHELL32	.rsrc	资源	Imag 01001002	R
7DD6A000	0001B000	SHELL32	.reloc	重定位	Imag 01001002	R
7FF6F0000	00006000				Map 00041020	R E
7FFA0000	00033000				Map 00041002	R
7FFDA000	00001000				Priv 00021004	RW
7FFDF000	00001000			数据块 于 主线程	Priv 00021004	RW
7FFE0000	00001000				Priv 00021002	R

内核区的内存

- 内存内核区中的所有数据是**所用进程共享的**，是操作系统代码的驻地。
 - 其中包括：操作系统内核代码，以及与线程调度、内存管理、文件系统支持、网络支持、设备驱动程序相关的代码。
- 该分区中所有代码和数据都被操作系统保护。
 - 用户模式代码无法直接访问和操作：如果应用程序直接对该内存空间内的地址访问，将会发生地址访问违规。

Windows系统下的内存布局

每个进程可用4GB内存空间？但我的电脑内存才2G！



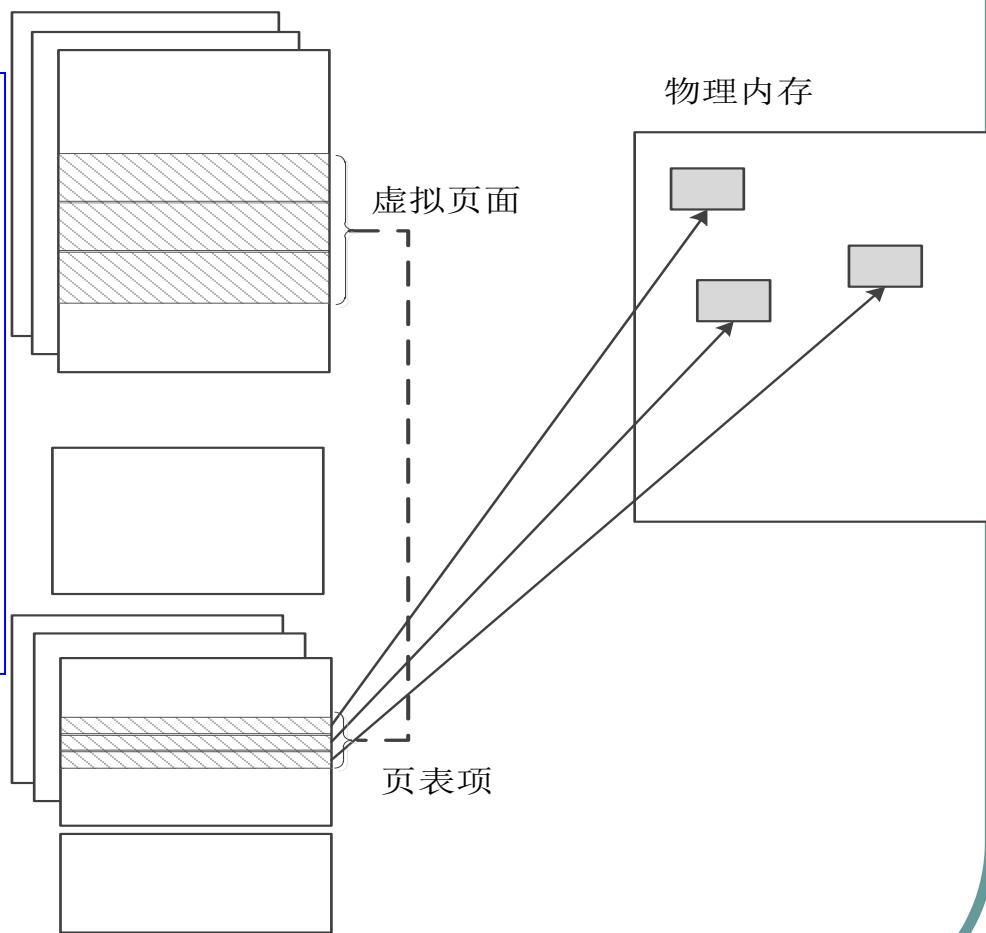
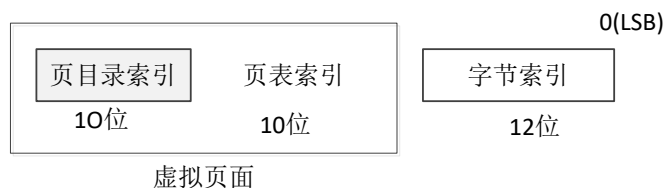
两个进程的可执行程序映像加载地址都是00400000H，但同一地址对应的代码却不一样，为什么？

• ?

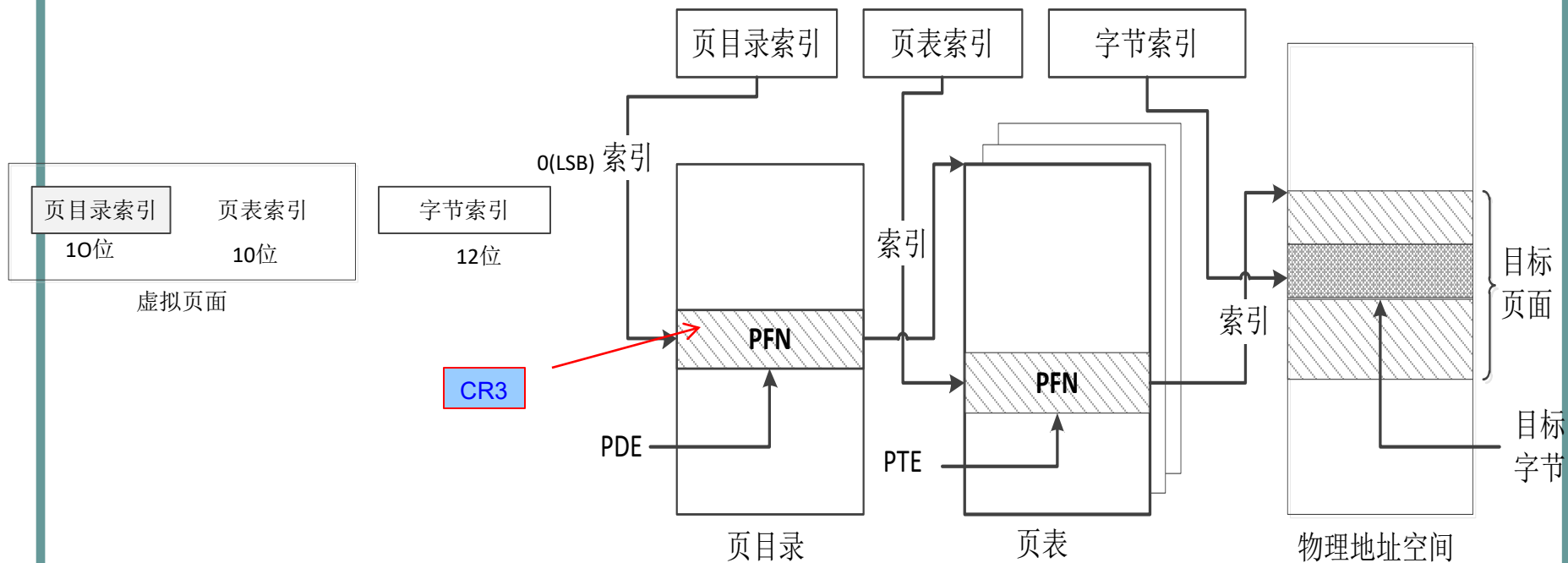
• ?

Windows 虚拟地址空间与物理地址空间

- **X86 Windows**默认使用**二级页表**来把虚拟地址转译为物理地址。
 - 一个**32**位地址被划分为三个单独部分：**页目录索引**、**页表索引**和**字节索引**。
- 在**x86**系统上默认页面大小为**4K**，故页内字节索引宽度为**12**位。



Windows: 虚拟页面通过二级页表项映射到物理内存



页目录 (**Page Directory**): 通过**CR3**寄存器获得页目录基地址。

PDE: Page Directory Entry, 页目录项。

PTE: Page Table Entry, 页表项, 指向虚拟页面所映射的物理地址。

PFN: Page Frame Number, 为页帧号。

思考题

(以32位系统为例):

1.windows编程中malloc实际上能够支持多大的内存呢?

2.不断增加物理内存, 能够增加malloc的内存大小吗?

3.为什么增加物理内存, 能够使得系统跑得更流畅呢?

体会一下: **64bit CPU**优势

提纲

2.1 系统引导与控制权

2.2 80X86处理器的工作模式

2.3 Windows内存结构与管理

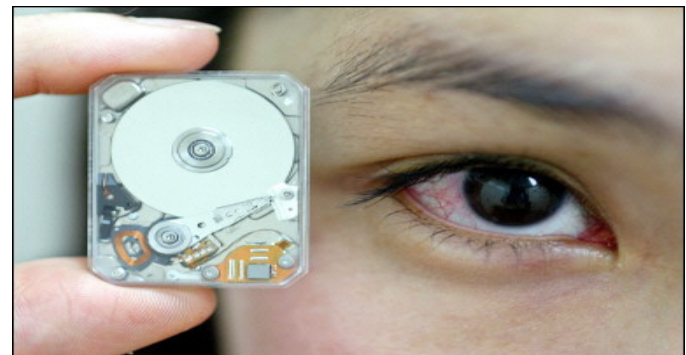
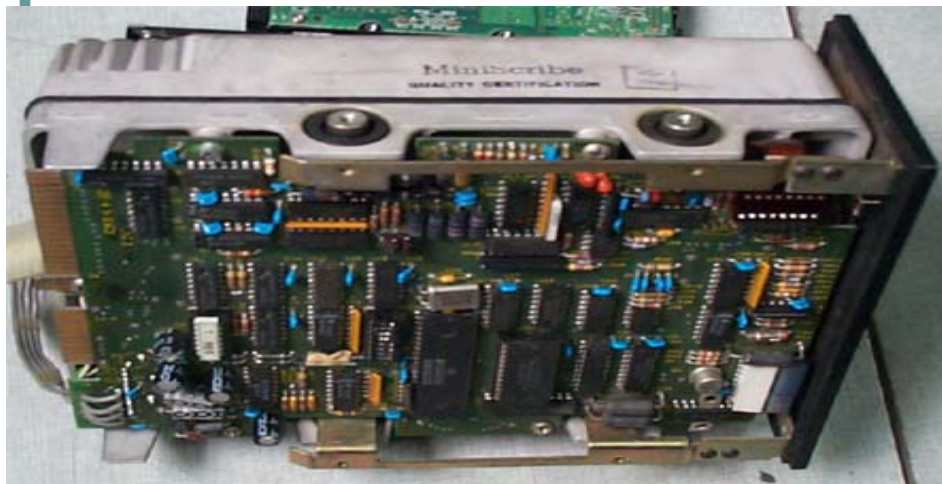
2.4 磁盘的物理与逻辑结构

2.5 FAT32文件系统

2.6 PE文件格式

2.7 破解实践

2.4 磁盘的物理与逻辑结构



硬盘与控制权

- 硬盘是
 - 控制代码的静态存储仓库
 - 系统引导代码
 - 各类程序与数据等
 - 恶意软件进行控制权争夺的中心

2.4.1 硬盘物理结构

- 硬盘外部结构

- 接口（电源接口+数据接口）
- 硬盘控制电路
- 固定面板

- 硬盘内部结构

- 盘片、磁头、盘片主轴、控制电机、磁头控制器、数据转换器...

(1) 硬盘外部结构

- 接口
 - 并口 (**PATA**)
 - 串口 (**SATA**)



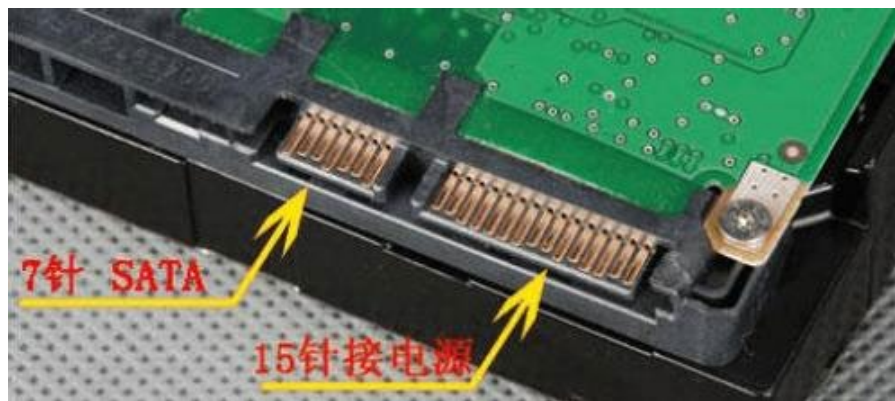
并口 (PATA—Parallel Advanced Technology Attachment, ATA/33、66、100、133等)

- 速度慢: **<133MB/s**
- 电源线: **4针**
- 数据线: **80/40/39针**



串口 (SATA—Serial Advanced Technology Attachment, SATA1.0-3.0)

- 速度快: **150/300/600MB/s**
- 电源线: **15针**
- 数据线: **7针**



硬盘控制电路

硬盘控制电路近照



拆下硬盘控制电路后



固定面板

- **固定面板**：保证硬盘盘片和机构的稳定运行。
 - **产品标签**：产品型号、产品序列号、产品、生产日期等。



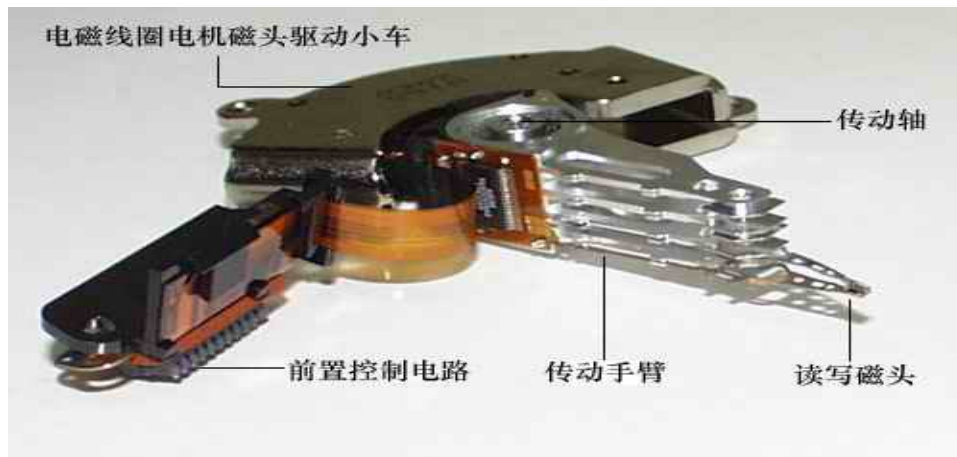
(2) 硬盘内部结构

- 硬盘内部组件主要包括：

- 磁盘盘片、读写磁头、盘片主轴、控制电机、磁头控制装置（传动手臂、传动轴、弹簧装置等）等。

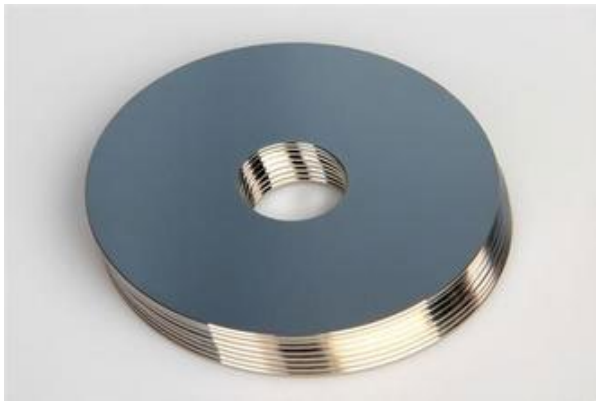


磁头组件



磁盘片

- 金属或玻璃

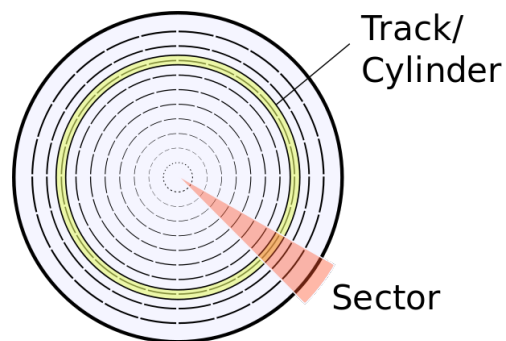
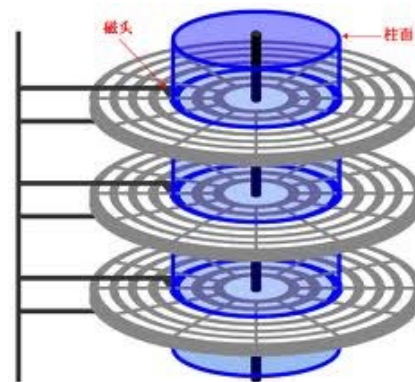


2.4.2 逻辑结构

2.4.2.1 寻址方式

● CHS参数寻址

- 柱面 (Cylinders) : 每个盘片的半径均为相同值 R 的同心圆 (磁道)
- 磁头 (Headers) : 每个盘片有两个面, 每个面有一个读写磁头。
- 扇区 (Sector) : 每个磁道被划分为几十个扇区



查看思考题。

2.4.2 逻辑结构

2.4.2.1 寻址方式

- **CHS(Cylinder/Head/Sector)参数取值范围**
 - 磁头数(Heads): 0-255 (8位)
 - 柱面数(Cylinders): 0-1023 (10位)
 - 扇区数(Sectors): 1-63 (6位); 通常每扇区512个字节。
- **CHS参数可以寻址的磁盘最大容量?**
 - $256 * 1024 * 63 * 512 / 1048576 = 8064 \text{ MB (1M = 1048576 Bytes)}$

2.4.2 逻辑结构

2.4.2.1 寻址方式

- 老式硬盘：每个磁道的扇区数相等。
- 当前硬盘：
 - 采用“等密度结构”
 - 寻址方式采用线性逻辑块寻址（LBA, Logical Block Address），即以扇区为单位进行线性寻址。
- 兼容问题如何解决？
 - 地址翻译器：负责将CHS参数翻译成线性参数。

2.4.2 逻辑结构

2.4.2.2—总体结构

- 硬盘的分区格式
 - MBR分区
 - 主引导扇区
 - 基本分区
 - 扩展分区
 - 逻辑驱动器
 - GPT分区 (GUID Partition Table)
- Windows常见分区类型
 - FAT32
 - NTFS等

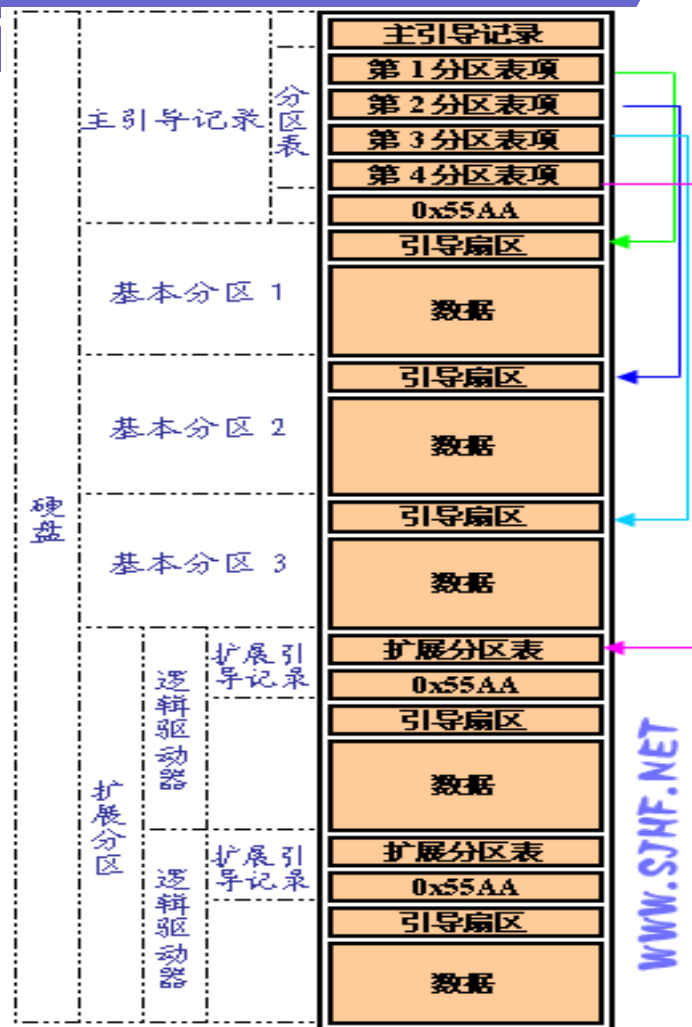


图5 一个4分区的基本磁盘

2.4.2.3 MBR分区

1.主引导扇区-MBR分区格式

- 位于整个硬盘的0柱面0磁头1扇区(硬盘的第一个扇区)
 - **MBR引导程序**：占了其中的前446个字节(偏移0H~偏移1BDH)
 - **DPT（硬盘分区表）**：Disk Partition Table，随后的64个字节
 - **结束标志**：最后的两个字节“55 AA” (偏移1FEH~偏移1FFH)。

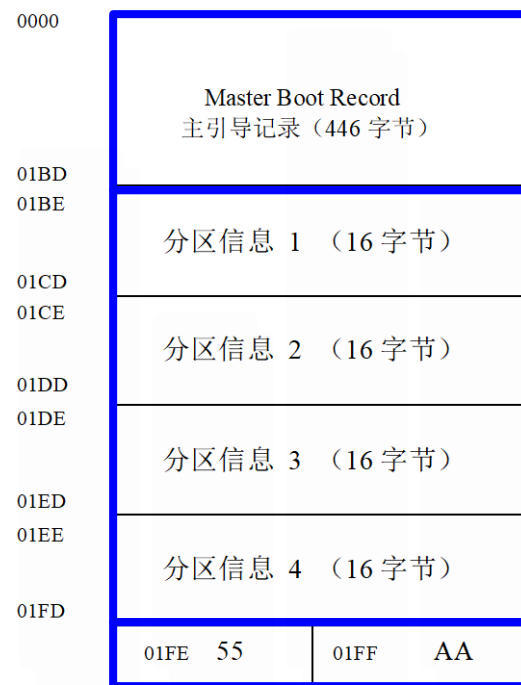


图 2-1 主引导扇区的结构

GPT（硬盘分区表）

- **4个分区项，64个字节。**
 - 描述各分区基本信息：
 - 分区开始位置、总扇区数
 - 分区类型等

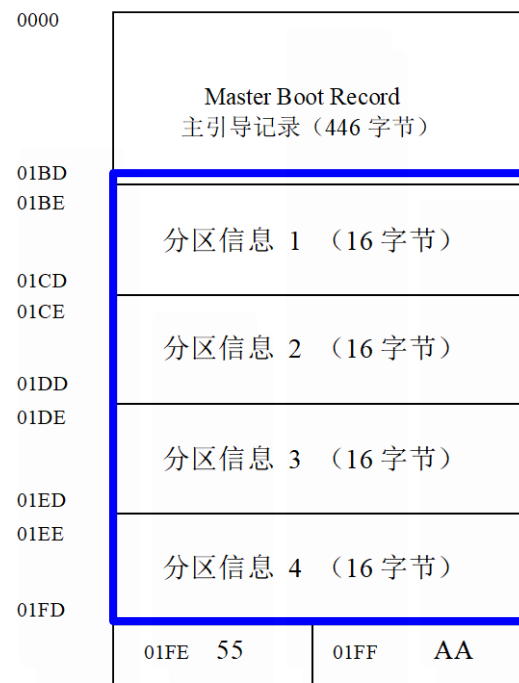


图 2-1 主引导扇区的结构

2.4.2.3 MBR分区 扩展分区

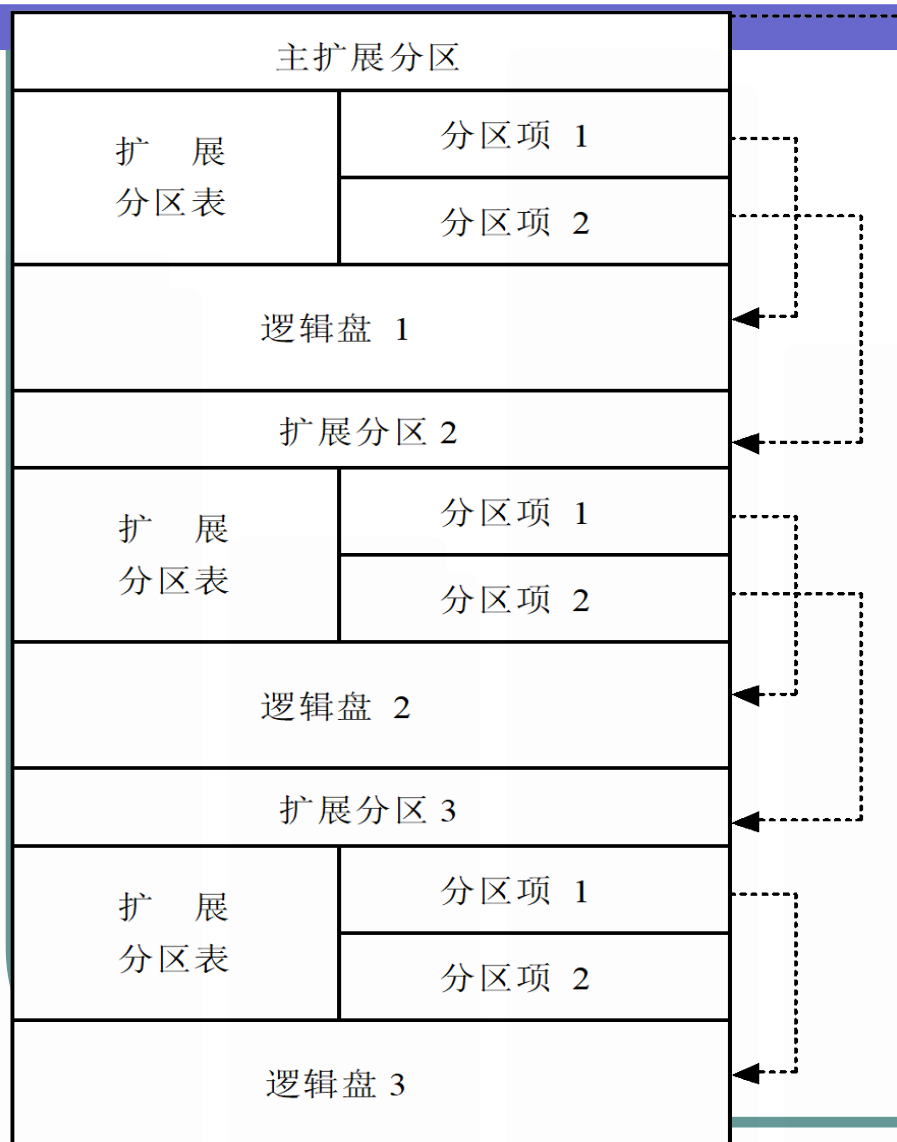


图 2-2 扩展分区和逻辑盘的示意图

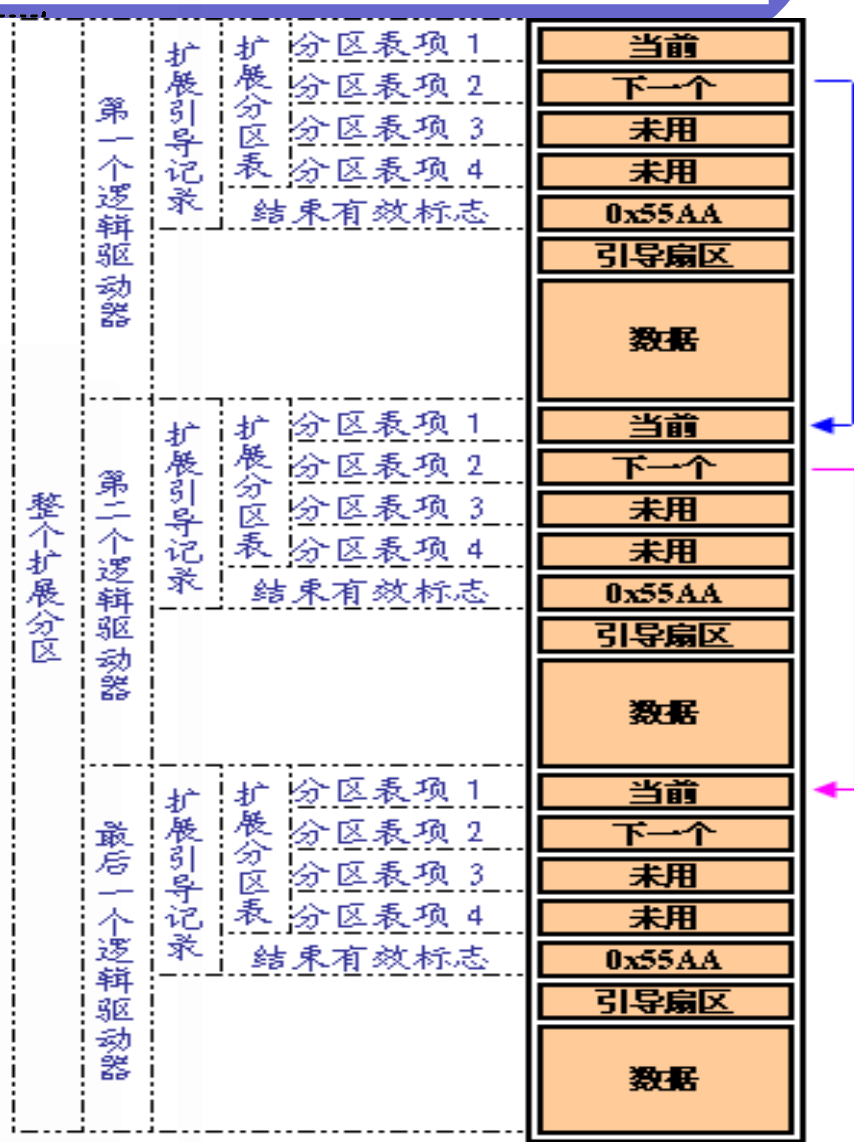


图6 分区表链接图示

提纲

2.1 系统引导与控制权

2.2 80X86处理器的工作模式

2.3 Windows内存结构与管理

2.4 磁盘的物理与逻辑结构

2.5 FAT32文件系统

2.6 PE文件格式

2.7 破解实践

2.5 FAT32文件系统

引导扇区



描述分区属性:

- 1.分区大小；
- 2.簇的大小
- 3.FAT表个数与大小
- 4.分区引导程序等

FAT (File
Allocation Table,
文件分配表)



两个功能:

- 1.记录数据存储区每一个簇的使用情况（是否被使用，或坏簇）；
- 2.形成每个文件的簇链表

数据存储区
(以簇为单位，每簇包含多个扇区，以簇号进行标示)



功能: 存储两类数据

- 目录项（目录和文件的属性信息，如文件名，大小，文件存储首簇号，时间等）-文件档案
- 文件数据

概念1：簇

- 文件系统将磁盘空间以一定数目（ 2^n , n 为整数）的扇区为单位进行划分，这样的单位称为簇。
 - 每扇区大小为512字节。
 - 簇的大小一般是512B、1KB、2KB、4KB、8KB、16KB、32KB、64KB等。

簇是进行文件空间分配的最小单位。

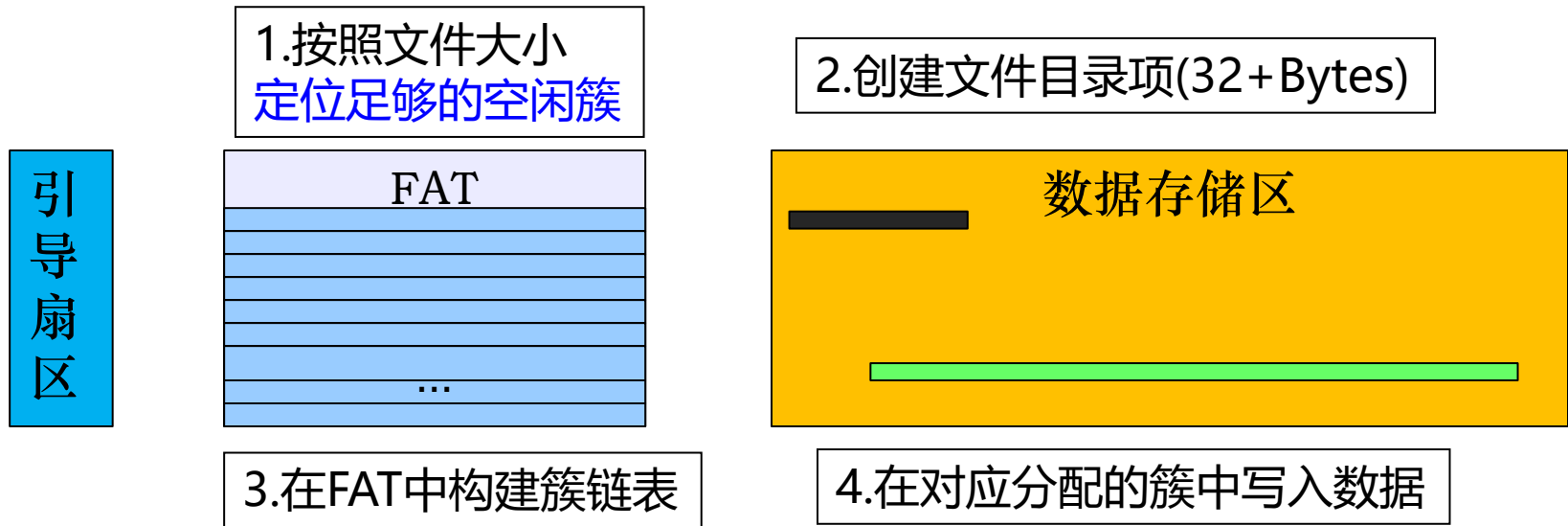
概念2: FAT表

- **FAT表**(File Allocation Table 文件分配表)是Microsoft在FAT文件系统中用于磁盘数据(文件)索引和定位引进的一种单向链式结构。
 - FAT区用每一个FAT项来记录每一个簇的占用情况
 - FAT表中表项的个数=簇的个数
 - 如果为0, 则表示对应簇为空闲, 可存储数据。
 - 每个表项有多大?
 - **FAT32: 32位, 4字节**
 - 可表达的最大簇号空间为4G
 - **FAT16**的最大簇号空间为64K

概念3：簇链

- 一个文件所占用簇的序号形成的单向链表。
- 实现方法：
 - 在文件占用簇的对应簇号的**FAT**项，填写下一个簇的簇号，如果为最后一簇，则输入结束标识“FFFFFFFF0F”

文件的存储



操作示意图

被删除文件的恢复机理

差异

- 目录项：
 - 文件名首字节被修改为E5
 - 首簇高位被清零
- FAT表簇链：
 - 被全部清空
- 文件内容：
 - 无变化

可否恢复？

- 目录项
 - 文件名首位是否可还原？
 - 如何确定高位？
- FAT表簇链如何修复？
 - 连续存储（默认）
 - 总簇数（文件大小）

提纲

2.1 系统引导与控制权

2.2 80X86处理器的工作模式

2.3 Windows内存结构与管理

2.4 磁盘的物理与逻辑结构

2.5 FAT32文件系统

2.6 PE文件格式

2.7 破解实践

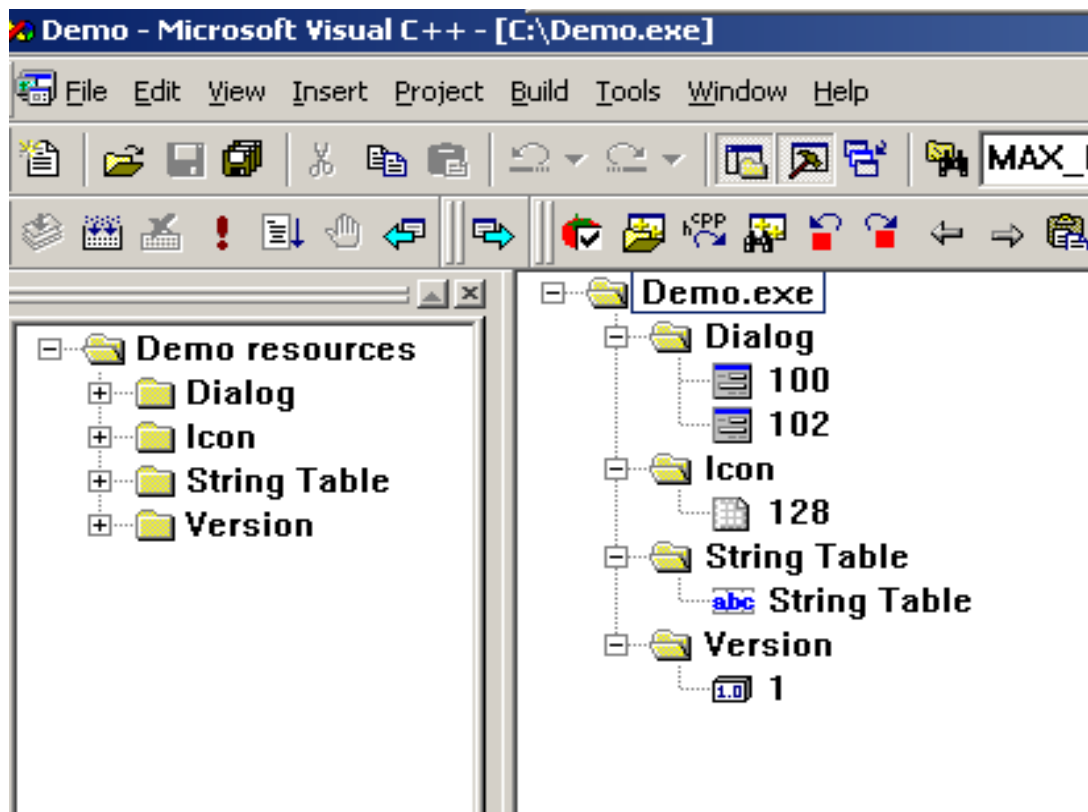
PE 文件格式

- 可移植的执行体 **PE** (**P**ortable **E**xecutable) 是 **Win32** 平台下可执行文件格式, 常见的 **exe**、**dll**、**ocx**、**sys**、**com** 都是 **PE** 文件, 其可移植可执行体现在跨 **Win32** 平台
- **PE** 文件格式规定了代码、菜单、图标、位图、字符串等信息在可执行文件中如何组织
- **PE** 文件格式将可执行文件分成若干节 (**section**), 一个 **WinNT** 应用程序典型地拥有 9 个预定义节: **.text**、**.bss**、**.rdata**、**.data**、**.pdata**、**.rsrc**、**.edata**、**.idata** 和 **.debug**
 - **.text** 由编译器产生, 存放二进制的机器代码
 - **.data** 初始化的数据块, 如全局变量、静态变量等
 - **.idata** 可执行文件所使用的动态链接库等外来函数与文件信息
 - **.rsrc** 存放程序的资源, 如图标、菜单等
 - **.rdata** 表示只读数据区
- 在 **VC** 中可用 **#pragma data_seg()** 将代码中的任意部分编译到 **PE** 的任意节, 且节名可以自定义



PE 文件格式(续)

- VC编写程序的资源段可以修改

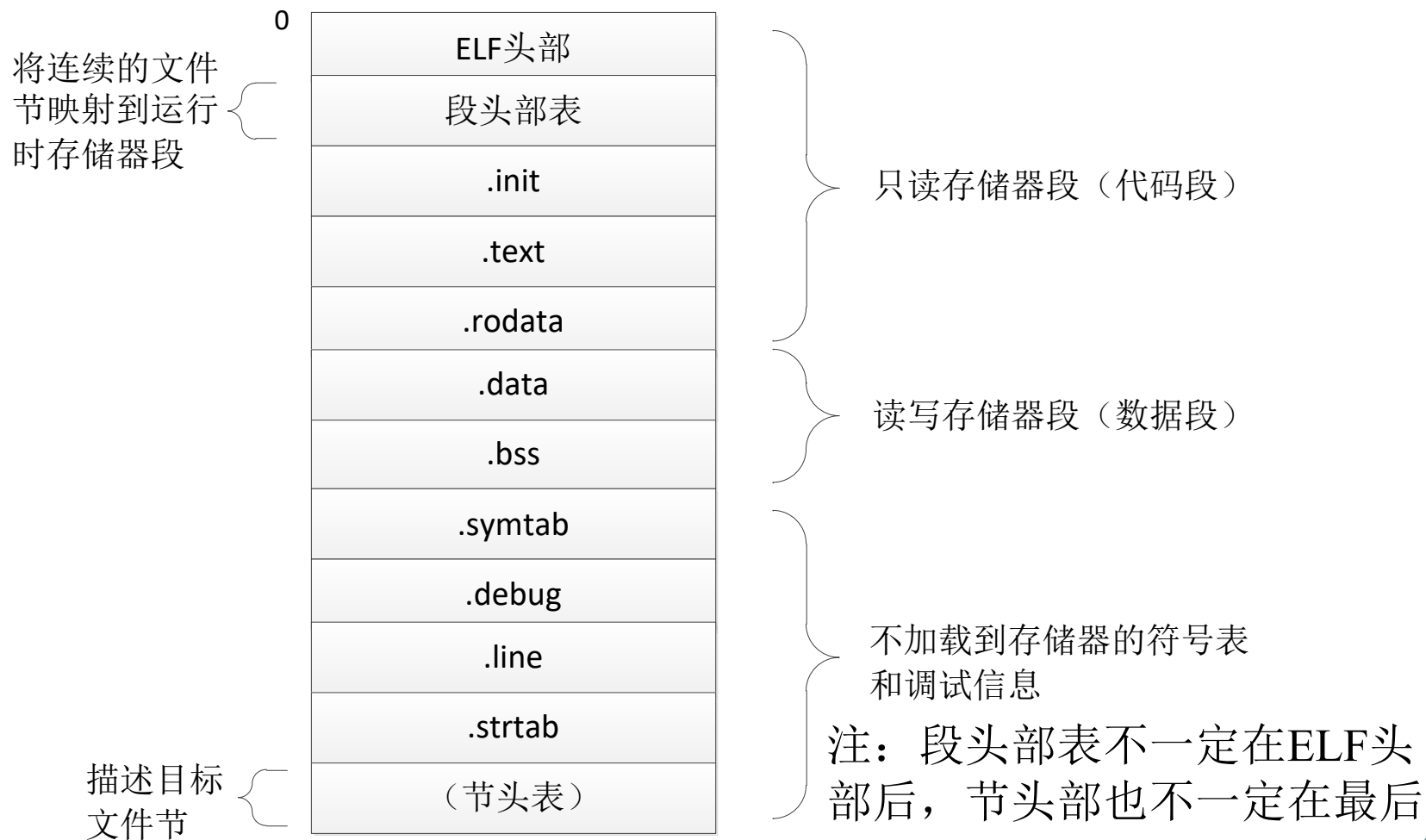


PE 文件格式(续)

MS-DOS MZ 头部
MS-DOS 实模式残余程序
PE 文件标志
PE 文件头
PE 文件 可选头部
.text 节头部
.bss 节头部
.rdata 节头部
.....
. debug 节头部
.text 节
.bss 节
.rdata 节
.....
. debug 节

- 整个格式的组成：一个 **MS-DOS** 的 **MZ** 头部，之后是一个实模式的残余程序、**PE**文件标志、**PE**文件头部、**PE**可选头部、所有的节头部，最后所有的节实体
- 可选头部的末尾是数据目录入口的数组，这些相对虚拟地址指向节实体之中的数据目录。每个数据目录都表示了一个特定的段实体数据是如何组织的。
- **PE** 文件格式有 **9** 个预定义节，这对所有的**WinNT**应用程序通用的，但是每个应用程序可以为它自己的代码以及数据定义它自己独特的节。
- **.debug** 预定义节也可以分离为一个单独的调试文件。如果这样的话，就会有一个特定的调试头部来用于解析这个调试文件，**PE** 文件中也会有一个标志来表示调试数据被分离了出去。

Linux ELF可执行文件



PE 文件与虚拟内存的映射

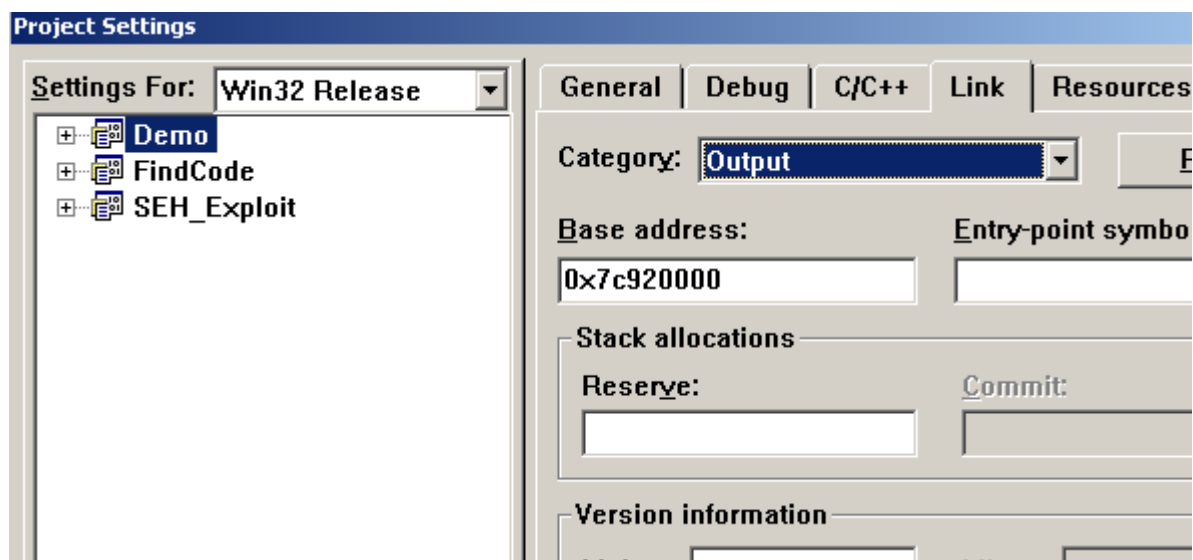
- 文件偏移地址 (**File Offset, FO or Roffset or RA**)
 - PE 文件在硬盘上存放时相对于文件头的偏移
- 装载基址 (**Image Base, IB**)
 - PE 装入内存时的基地址，默认情况下，**exe** 的装载基址为 **0x00400000**，**dll** 的装载基址 为 **0x10000000**
- 虚拟内存地址 (**Virtual Address, VA**)
 - PE 文件中的指令被装入内存后的地址
- 相对虚拟地址 (**Relative Virtual Address, RVA**)
 - 指令的虚拟内存地址相对于装载基址的偏移量

$$VA = \text{Image Base} + RVA$$



PE 文件与虚拟内存的映射

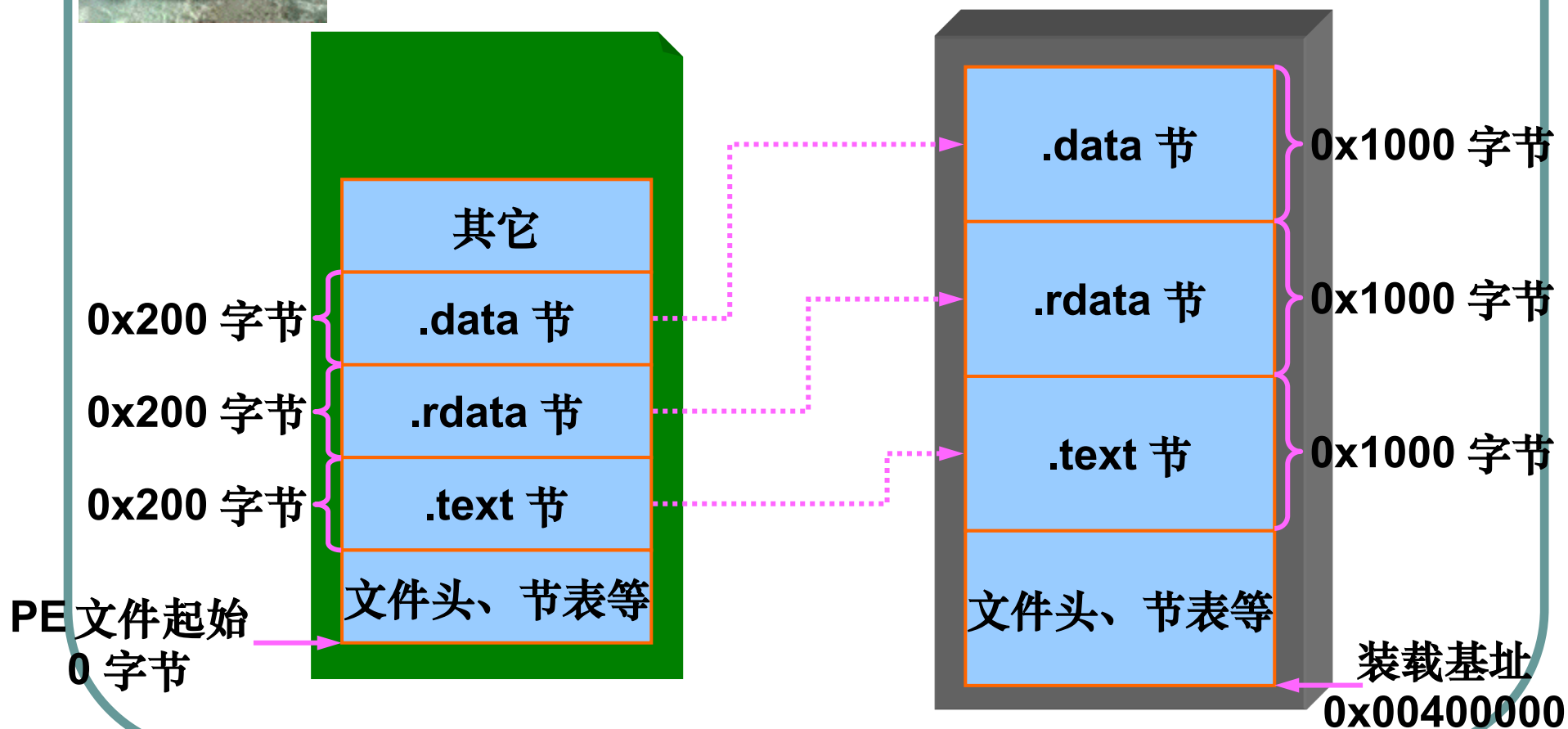
- VC如何定制基址



$$VA = \text{Image Base} + RVA$$



PE 文件与虚拟内存的映射(续)



PE 文件与虚拟内存的映射(续)

- **PE** 文件中的数据按磁盘数据标准存放，以 **0x200** 字节为基本单位进行组织，**PE** 节的大小是 **0x200** 的整数倍，不足用 **0x00** 填充
- **PE** 文件装入内存后，将按内存数据标准存放，以 **0x1000** 为基本单位进行组织，在内存中 **PE** 节的大小是 **0x1000** 的整数倍，不足用 **0x00** 填充
- 节偏移：由于磁盘和内存存储分配单位的差异引起的节基址之差

文件偏移地址 $FOA = VA - \text{Image Base} - VSO(\text{虚拟内存节偏移}) + FSO(\text{文件节偏移})$
 $= RVA - VSO(\text{虚拟内存节偏移}) + FSO(\text{文件节偏移})$

PE 文件与虚拟内存的映射(续)

节	相对虚拟偏移地址 RVA/VSO/VOffset	文件偏移量 RA/FSO/ROffset
CODE	0x00001000	0x00000400
DATA	0x001D1000	0x001D0200
BSS	0x001D5000	0x001D3C00
.idata	0x001D7000	0x001D3C00
.edata	0x001DB000	0x001D7200
.tls	0x001DC000	0x001D7400
.rdata	0x001DD000	0x001D7400
.reloc	0x001DE000	0x00000000
.rsrc	0x001FB000	0x001D7600

CODE 虚拟内存节偏移**VSO=0x1000**,文件节偏移**FSO=0x400**

虚拟内存地址: **0x0049FBC4** 处有一条指令 **mov edi, eax (8B F8)**

文件偏移地址 = ?(在磁盘文件中的地址是多少) (设基址**BA=0x400000**)

逆向分析实例-相关工具简介

- **PE 文件修改工具：Lord PE**
 - **PE 文件与虚拟内存的地址转换**
 - **查看 PE 文件的节信息、装载基址、镜像大小**
 - **修改 PE 文件头信息，以重建 PE，主要用于脱壳**



相关工具简介(续)

● 二进制编辑器

● UltraEdit

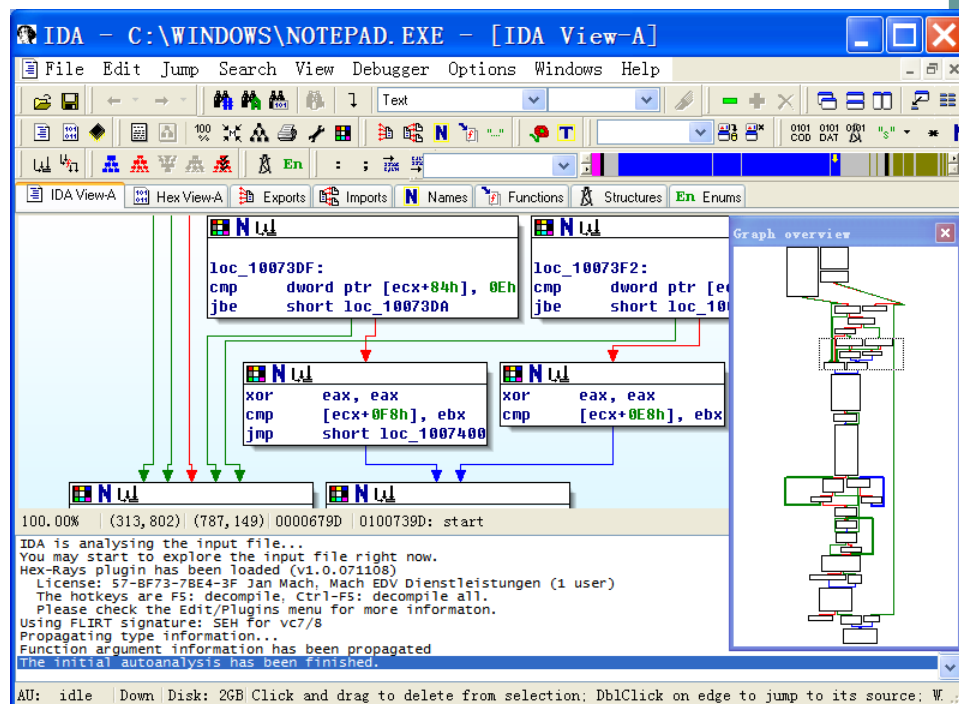
- **alt + c**: 列模式与行模式编辑状态切换
- **ctrl + h**: **16** 进制与 **ASCII** 编辑模式间切换
- **ctrl + f**: 查找（可统计待查找串出现的次数）
- **F3 (ctrl + F3)**: 查找下（上）一个串
- **ctrl + r**: 替换
- **ctrl + g**: 跳转到特定行或特定偏移地址
- **ctrl + d**: **16** 进制插入或删除



- **Hex Workshop**: 专注于十六进制编辑，附带转换器与计算器
- **WinHex**: 能够透过文件系统直接对磁盘的扇区、簇进行操作，因此用于数据恢复非常方便（也很危险）
- **H-View**: 运行于黑底白字的命令行方式

相关工具简介(续)

- 静态反汇编工具：**IDA Pro**
 - 当前最强大的静态反汇编软件（也能进行简单的动态调试）
 - 能将庞大的汇编指令序列分成不同层次的单元、模块、函数，并给予标注和注解，以便交叉引用
 - 能自动识别和标注 **VC**、**BC**、**TC**、**Delphi** 等常用编译器的标准库函数
 - 能以图形化的方式显示函数内部的执行流程
 - 可以将标注好的函数名、注释等信息导出为 **map** 文件，供 **OllyDbg** 动态调试时使用



相关工具简介(续)

- 动态调试工具

- **SoftICE (Soft In Circuit Emulator)**

- 工作在操作系统的 **Ring 0**，以软件的方式实现了监视 **CPU** 的所有动作，可以调试驱动等内核对象，也可使用**RCP/IP**连接进行远程调试
- 暴力中断所有进程，不如 **OllyDbg** 使用方便
- 所有功能通过调试命令完成，并且有可能无意间修改系统很底层的东西，无经验者使用经常出现死机、蓝屏
- **ctrl + d**: 呼出调试界面（但不易对界面截图）

- **WinDbg**

- 一款介于 **OllyDbg** 和 **SoftICE** 之间的较为“温和”的调试器
- 可以调试内核，但不如 **SoftICE** 那样“毫不讲理”地中断操作系统也可以仅在 **Ring 3** 级进行调试
- 可以设置异常复杂的断点条件逻辑
- 主要功能由调试命令完成，且与 **SoftICE** 的调试命令类似

相关工具简介(续)

● OllyDbg

- 工作于 **Ring 3** 级，界面友好，使用方便，是目前主流的动态调试器
- 绿色软件，无须安装
- 扩展性极强，现已有 **100** 多个特殊用途的插件，如果还不满足你的需要，可以自己开发专用的插件

● 基本功能快捷键

- **F2**: 断点设置与取消之间切换
- **F4**: 运行到当前光标所在处
- **F7**: 单步步入，遇到函数调用则跟进去
- **F8**: 单步步过，遇到函数调用不跟进
- **F9**: 运行程序直到遇到断点而中断下来
- **ctrl + F9**: 执行到函数返回前的指令
- **Alt + F9**: 从系统空间返回到用户空间
- **Ctrl + g**: 查看任意位置的数据
- **Ctrl + e**: 编辑任意位置的数据
- **Enter**: 转到指定位置
- **Space**: 在任意位置处反汇编
- **Ctrl + A**: 分析当前模块的代码节
- **;**: 添加自定义注释
- **:**: 添加自定义标签

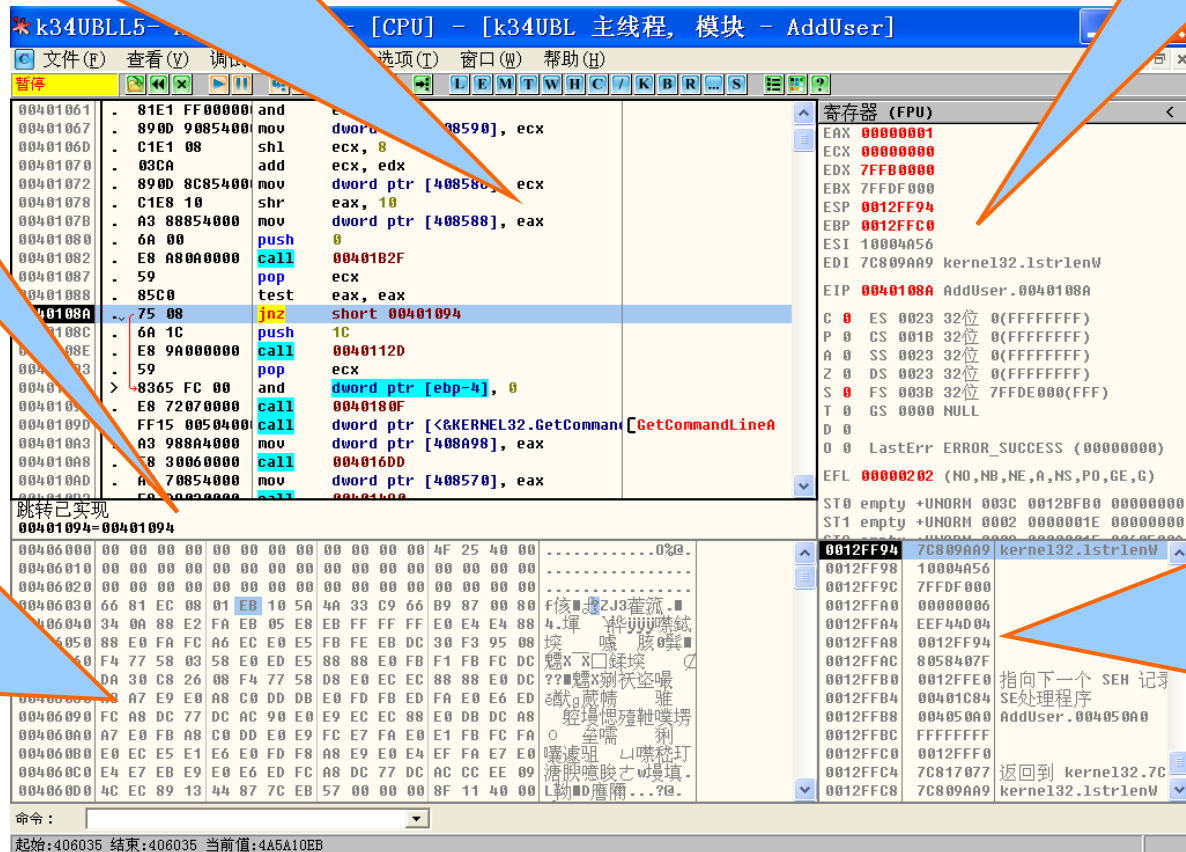
相关工具简介(续)

代码区：显示指令地址、机器码、汇编指令、注释，对于常用函数的调用，可直接显示函数名

寄存器区：实时查看寄存器的变化

预执行区：提前计算当前指令的运算结果

内存区：可以方便地查看与修改任意位置的数据，注释区可对十六进制数据进行译码并显示



栈区：显示栈地址及内容，自动标注返回地址、SHE句柄、字符串指针所指向的内容、完整的栈帧

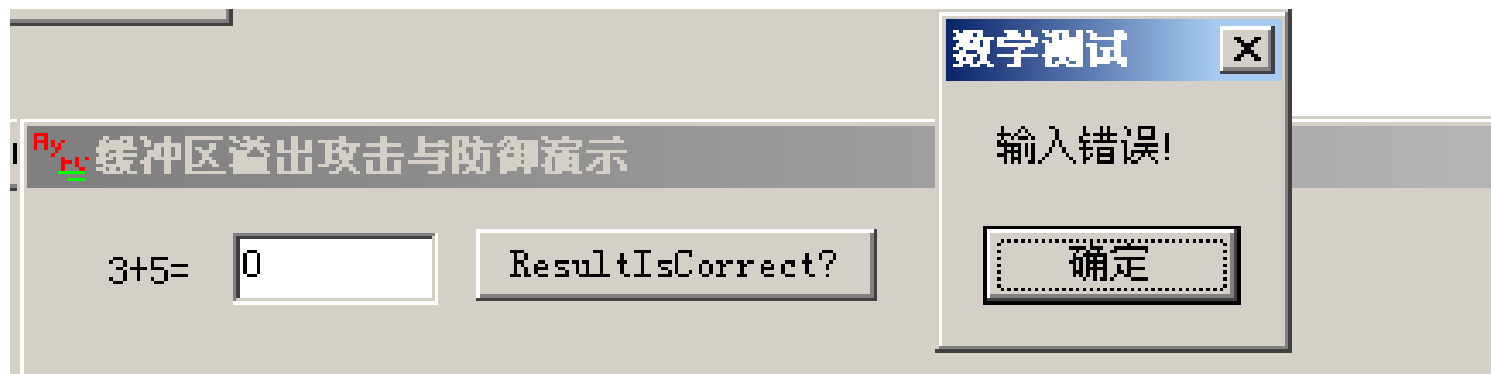
Crack 二进制文件(1)

- 以下用一个简单的破解小实验，综合运用上述各种工具，使大家对上手操作有一个感性认识



Crack 二进制文件(2)

- 先根据外在特点进行分析，找出关键函数位置... ..



Crack 二进制文件(3)

- 用调试工具设置断点，找到关键代码... ..

(1) MessageBox

N 全部名称				
地址	模	区段	类型	名称
77E70261		.text	导出	MesHandleFree
77EB3EAA		.text	导出	MesIncrementalHandleReset
77EB4AF2		.text	导出	MesInqProcEncodingId
62C21104		.text	导入	USER32.MessageBeep
73DCA42C		.rdata	导入	USER32.MessageBeep
763011F4		.text	导入	USER32.MessageBeep
77D31F7B		.text	导出	MessageBeep
00403204		.rdata	导入	USER32.MessageBoxA
73DCA5A0		.rdata	导入	USER32.MessageBoxA
77D507EA		.text	导出	MessageBoxA
77D5085C		.text	导出	MessageBoxExA
77D50838		.text	导出	MessageBoxExW
77D3A082		.text	导出	MessageBoxIndirectA
77D664D5		.text	导出	MessageBoxIndirectW
77D66406		.text	导出	MessageBoxTimeoutA
77D66383		.text	导出	MessageBoxTimeoutW
77D66534		.text	导出	MessageBoxW
77EB042A		.text	导出	MIDL_wchar_strcpy
77EB040A		.text	导出	MIDL_wchar_strlen
77D11150		.text	导入	GDI32.MirrorRgn



Crack 二进制文件(4)

- 用调试工具设置断点，找到关键代码...

(2)回朔

CPU - 主要线程，模块 - USER32		
地址	十六进制	反汇编
77D507EA USER32.MessageBoxA	8BFF	mov edi,edi
77D507EC	55	push ebp
77D507ED	8BEC	mov ebp,esp
77D507EF	833D BC14D777	cmp dword ptr ds:[77D714BC],0
77D507F6	74 24	je short USER32.77D5081C
77D507F8	64:A1 18000000	mov eax,dword ptr fs:[18]
77D507FE	6A 00	push 0
77D50800	FF70 24	push dword ptr ds:[eax+24]
77D50803	68 241BD777	push USER32.77D71B24
77D50808	FF15 C412D177	call dword ptr ds:[<&KERNEL32.Ini
77D5080E	85C0	test eax,eax
77D50810	75 0A	jnz short USER32.77D5081C
77D50812	C705 201BD777	mov dword ptr ds:[77D71B20],1
77D5081C	6A 00	push 0
77D5081E	FF75 14	push dword ptr ss:[ebp+14]
77D50821	FF75 10	push dword ptr ss:[ebp+10]
77D50824	FF75 0C	push dword ptr ss:[ebp+C]
77D50827	FF75 08	push dword ptr ss:[ebp+8]
77D5082A	E8 2D000000	call USER32.MessageBoxExA
77D5082F	5D	pop ebp
77D50830	C2 1000	ret 10
77D50833	90	nop



Crack 二进制文件(5)

- 用调试工具设置断点，找到关键代码...

(3)继续回朔...

OllyDbg - Demo.exe - [CPU - 主要线程, 模块 - Demo]

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

暂停

地址	十六进制	反汇编	注释
0040128B	90	nop	
0040128C	90	nop	
0040128D	90	nop	
0040128E	90	nop	
0040128F	90	nop	
00401290	56	push esi	
00401291	6A 01	push 1	
00401293	8BF1	mov esi,ecx	
00401295	E8 760E0000	call <jmp.&MFC42.#6334>	
0040129A	8B46 60	mov eax,dword ptr ds:[esi+60]	
0040129D	5E	pop esi	
0040129E	6A 00	push 0	
004012A0	83F8 08	cmp eax,8	
004012A3	68 40424000	push Demo.00404240	
004012A8	75 0E	jnz short Demo.004012B8	
004012AA	68 2C424000	push Demo.0040422C	
004012AF	6A 00	push 0	
004012B1	FF15 04324000	call dword ptr ds:[<&USER32.MessageBoxA>	USER32.MessageBoxA
004012B7	C3	retn	
004012B8	68 20424000	push Demo.00404220	
004012BD	6A 00	push 0	
004012BF	FF15 04324000	call dword ptr ds:[<&USER32.MessageBoxA>	USER32.MessageBoxA
004012C5	C3	retn	
004012C6	90	nop	

Crack 二进制文件(6)

- 用调试工具设置断点，找到关键代码...

(4)重新设置断点

OllyDbg - Demo.exe - [CPU - 主要线程, 模块 - Demo]			
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H)			
暂停 [Icons] L E M T W H C / K B R ... S [Icons] ?			
地址	十六进制	反汇编	注释
0040128E	90	nop	
0040128F	90	nop	
00401290	56	push esi	
00401291	6A 01	push 1	
00401293	8BF1	mov esi,ecx	
00401295	E8 760E0000	call <jmp.&MFC42.#6334>	
0040129A	8B46 60	mov eax,dword ptr ds:[esi+60]	
0040129D	5E	pop esi	
0040129E	6A 00	push 0	
004012A0	83F8 08	cmp eax,8	
004012A3	68 40424000	push Demo.00404240	
004012A8	75 0E	jnz short Demo.004012B8	
004012AA	68 2C424000	push Demo.0040422C	
004012AF	6A 00	push 0	
004012B1	FF15 04324000	call dword ptr ds:[<&USER32.MessageBoxA>	USER32.MessageBoxA
004012B7	C3	retn	
004012B8	68 20424000	push Demo.00404220	
004012BD	6A 00	push 0	
004012BF	FF15 04324000	call dword ptr ds:[<&USER32.MessageBoxA>	USER32.MessageBoxA
004012C5	C3	retn	
004012C6	90	nop	
004012C7	00	non	

Crack 二进制文件(7)

● 破解...

004012A3	68 40424000	push Demo.00404240
004012A8	75 0E	jnz short Demo.004012B8
004012AA	68 2C424000	push Demo.0040422C
004012AF	6A 00	push 0
004012B1	FF15 04324000	call dword ptr ds:[<&USER32.MessageBoxA>]
	--	



004012A0	83F8 08	cmp eax,8
004012A3	68 40424000	push Demo.00404240
004012A8	90	nop
004012A9	90	nop
004012AA	68 2C424000	push Demo.0040422C
004012AF	6A 00	push 0
004012B1	FF15 04324000	call dword ptr ds:[<&USER32.MessageBoxA>]
004012B7	C3	ret



Crack 二进制文件(8)

- 看看结果

>X

.&MFC42.#6334>

缓冲区溢出攻击与防御演示

3+5=

6

ResultIsCorrect?

数字测试

您输入为8,输入正确!

确定



Crack 二进制文件(9)

● 还未完，固化破解

虚拟地址基址

[PE Editor] - c:\demo.exe [READ ONLY]

Basic PE Header Information

EntryPoint: 00002190 Subsystem: 0002 OK Save

ImageBase: 00400000

SizeOfImage: 00006000

BaseOfCode: 00001000

BaseOfData: 00003000

SectionAlignment: 00001000

FileAlignment: 00001000

Magic: 010B

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00001422	00001000	00002000	60000020
.rdata	00003000	00000CC8	00003000	00001000	40000040
.data	00004000	00000704	00004000	00001000	C0000040
.rsrc	00005000	00000FC8	00005000	00001000	40000040

虚拟地址节偏移,VSO 文件节偏移(File Section Offset,FSO)

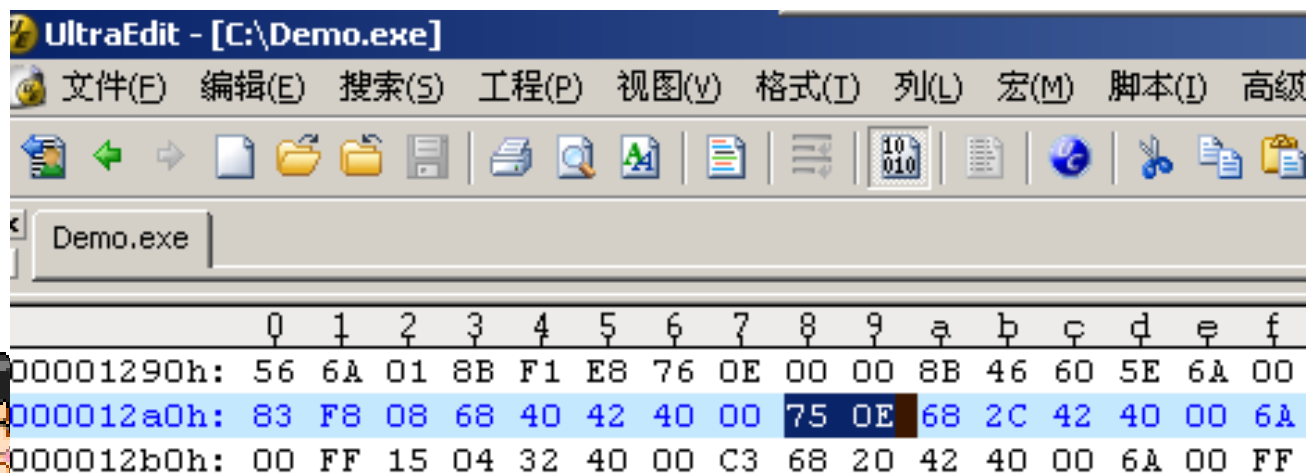


Crack 二进制文件(10)

- 计算文件中指令所在的物理地址...

004012A3	68 40424000	push Demo.00404240
004012A8	75 0E	jnz short Demo.004012B8
004012AA	68 2C424000	push Demo.0040422C

RAddress=? 0x004012A8-0x00400000-0x00001000+0x00001000
= 0x000012A8



UltraEdit - [C:\Demo.exe]

文件(F) 编辑(E) 搜索(S) 工程(P) 视图(V) 格式(T) 列(L) 宏(M) 脚本(I) 高级

Demo.exe

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00001290h:	56	6A	01	8B	F1	E8	76	0E	00	00	8B	46	60	5E	6A	00
000012a0h:	83	F8	08	68	40	42	40	00	75	0E	68	2C	42	40	00	6A
000012b0h:	00	FF	15	04	32	40	00	C3	68	20	42	40	00	6A	00	FF



Crack 二进制文件(11)

- 直接修改文件...

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00001290h:	56	6A	01	8B	F1	E8	76	0E	00	00	8B	46	60	5E	6A	00
000012a0h:	83	F8	08	68	40	42	40	00	90	90	68	2C	42	40	00	6A
000012b0h:	00	FF	15	04	32	40	00	C3	68	20	42	40	00	6A	00	FF
000012c0h:	15	04	32	40	00	C3	90	90	90	90	90	90	90	90	90	90



Crack 二进制文件(12)

- 查看运行效果



破解成功!

缓冲区溢出分析基础

- 看实际演练...

练习

1. 为什么文件删除后还可以恢复？
2. 请根据Windows进程空间内存分布，列举可能存在的的安全的问题。
3. 为什么不需要源代码可以直接修改PE文件获得不同界面语言的版本？
4. 对于有多个盘面的硬盘，在存储数据时候，既可以将数据连续存储在一个盘面上，也可以将数据存储在不同盘面上，试分析怎么存储合理？
5. 请自己练习逆向破解课堂例程.

作业交1-4题。

精彩内容下章继续...

❖ 下堂课见

