汇编语言与逆向技术

第8章 PE文件结构

王志
zwang@nankai.edu.cn

南开大学 网络空间安全学院

2024-2025学年

本章知识点

1. 可执行文件
2. PE文件基本概念
   - 难点：虚拟地址空间、相对虚拟地址
3. DOS文件头
   - 重点：MZ头、定位PE头
4. PE文件头
   - 重点：AddressOfEntryPoint、ImageBase、数据目录
5. 节
   - 难点：文件偏移RAW到内存地址RVA的转换方法
6. 导入表
   - 难点：INT（Import Name Table）与IAT（Import Address Table）
7. 导出表
   - 难点：AddressOfNames，AddressOfNameOrdinals，AddressOfFunctions

可执行文件

主观题 10分

设置

exe可执行程序是否仅包含汇编语言对应的CPU指令序列？exe程序中除了CPU指令，还需要哪些信息。

作答

单选题 10分

设置

Windows、Linux上的可执行文件格式是一样的吗？ （使用相同的CPU，Windows上的可执行程序是否可以在Linux上执行。）

A 是

B 不是

提交

可执行文件

- 可执行文件 (executable file)
  - 可以由**操作系统**进行加载、执行的文件
  - 在不同的操作系统环境下，可执行文件的格式不一样
    - Windows、Linux、MacOS
  - 二进制文件，不同于txt、word、excel等文本文件

填空题 10分

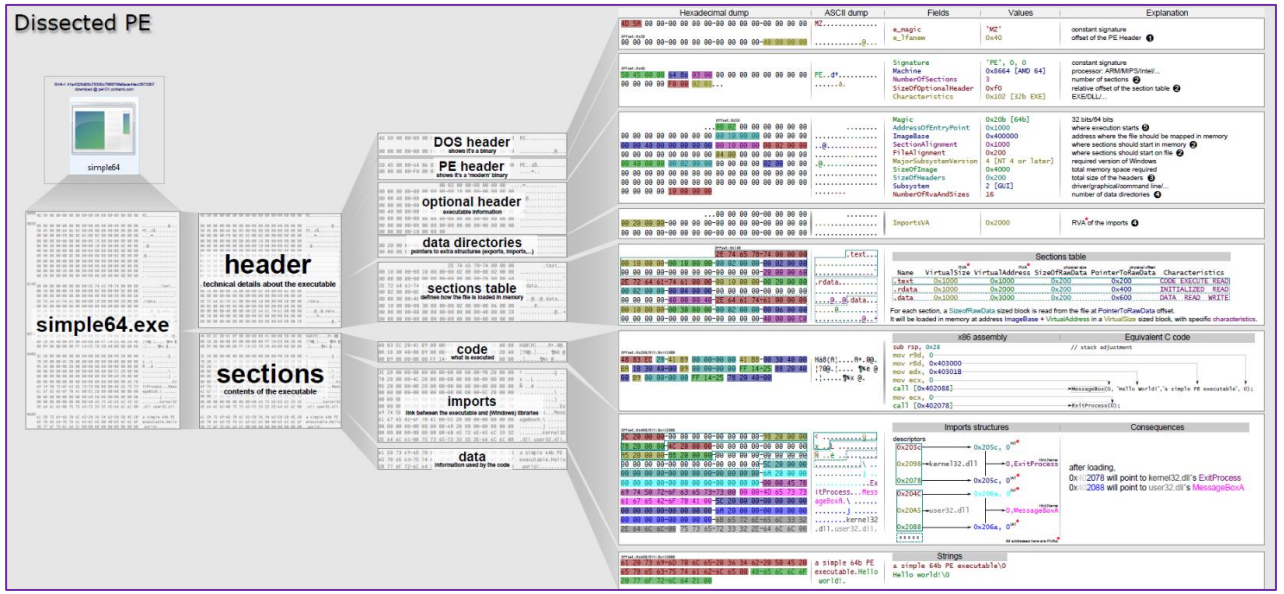
设置

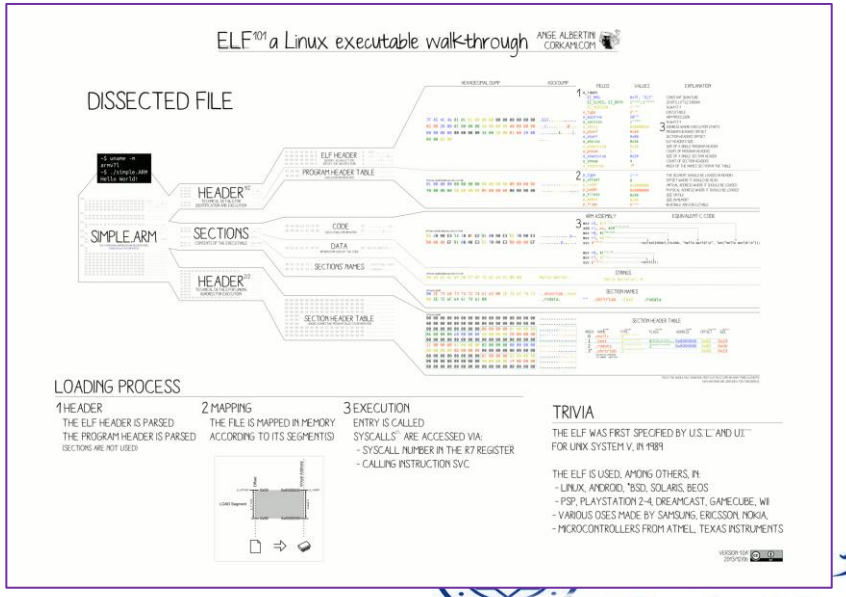
Windows系统可执行文件的格式是 [填空1]，Linux系统可执行文件的格式是 [填空2]

作答

可执行文件格式

- Windows系统可执行文件使用PE文件格式（Portable Executable）
- Linux系统可执行文件使用ELF文件格式（Executable and Linkable Format）





主观题 10分

设置

除了exe，Windows上有哪些可执行文件的后缀?

作答

Windows系统可执行文件

- 在Windows操作系统下，可执行程序可以是.com文件、 .exe文件、 .sys文件、 .dll文件、.scr文件等类型文件

.com文件

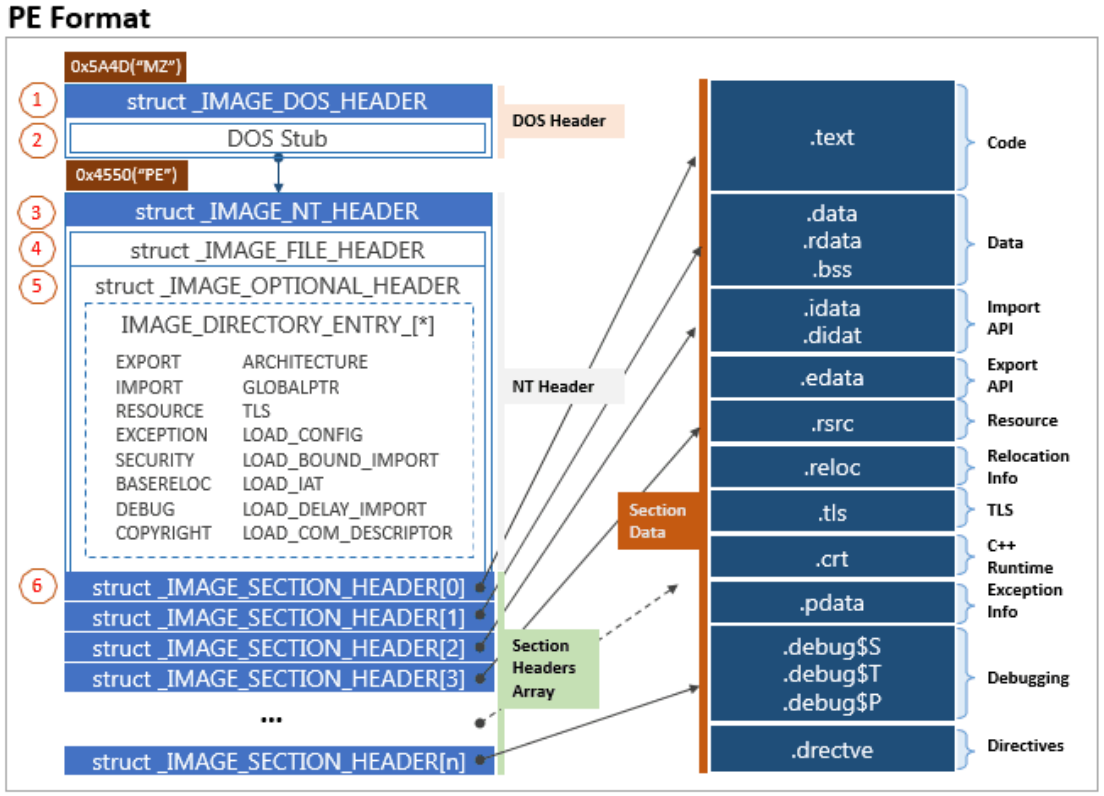
- .com文件在IBM PC早期出现，格式主要用于命令行应用程序、最大65,280字节
- 与MS-DOS操作系统的可执行文件兼容

.exe、.dll、.sys可执行文件

- .exe,.dll,.sys文件使用的是PE文件结构
- PE（Portable Executable File Format）可移植可执行文件结构
- 理解PE文件结构是逆向技术的基础

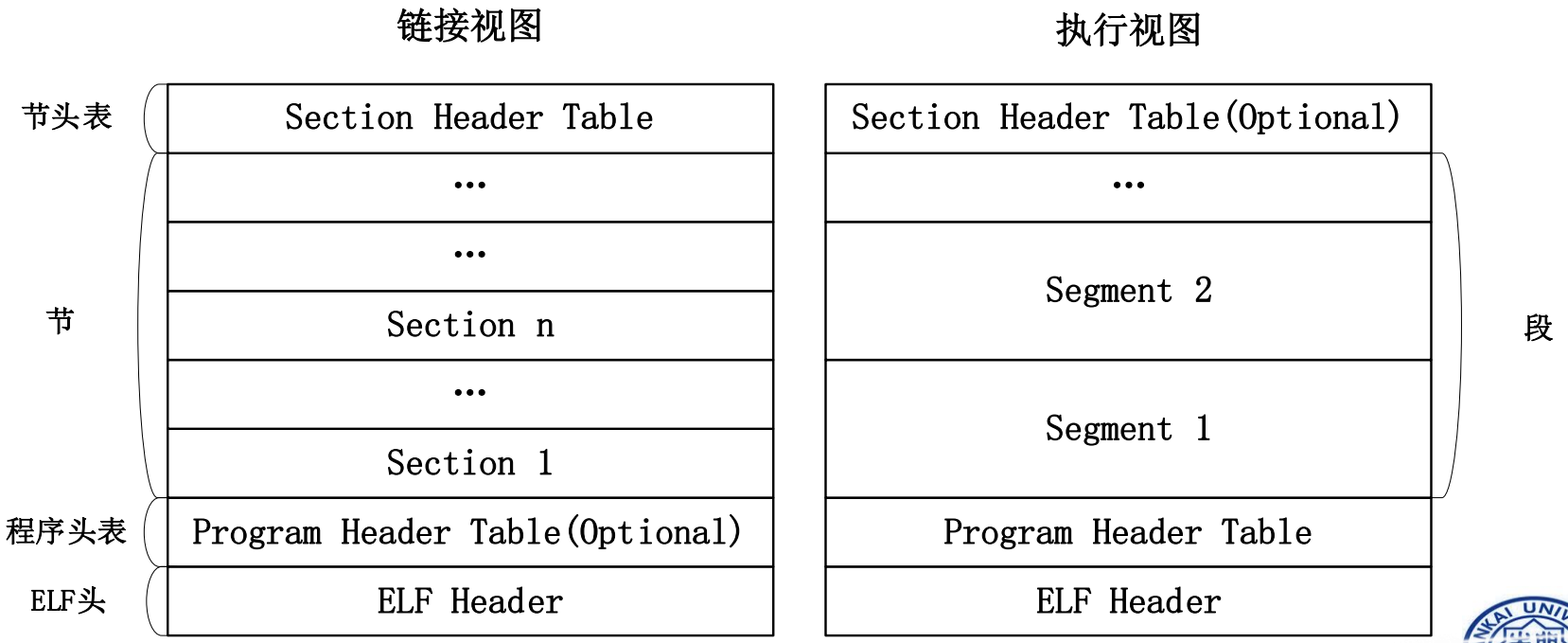
PE结构的二进制文件

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..............
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	........@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	........!..L.!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~..............
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	........}?......
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich............
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE..L......^....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	....
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	....
000000E0	00	20	00	00	00	00	40	00	00	10	00	00	00	02	00	00	. ....@.........
000000F0	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	....
00000100	00	40	00	00	00	04	00	00	00	00	00	00	03	00	00	00	.@..............
00000110	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	....
00000120	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	....
00000130	10	20	00	00	28	00	00	00	00	00	00	00	00	00	00	00	. ..(...........
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....



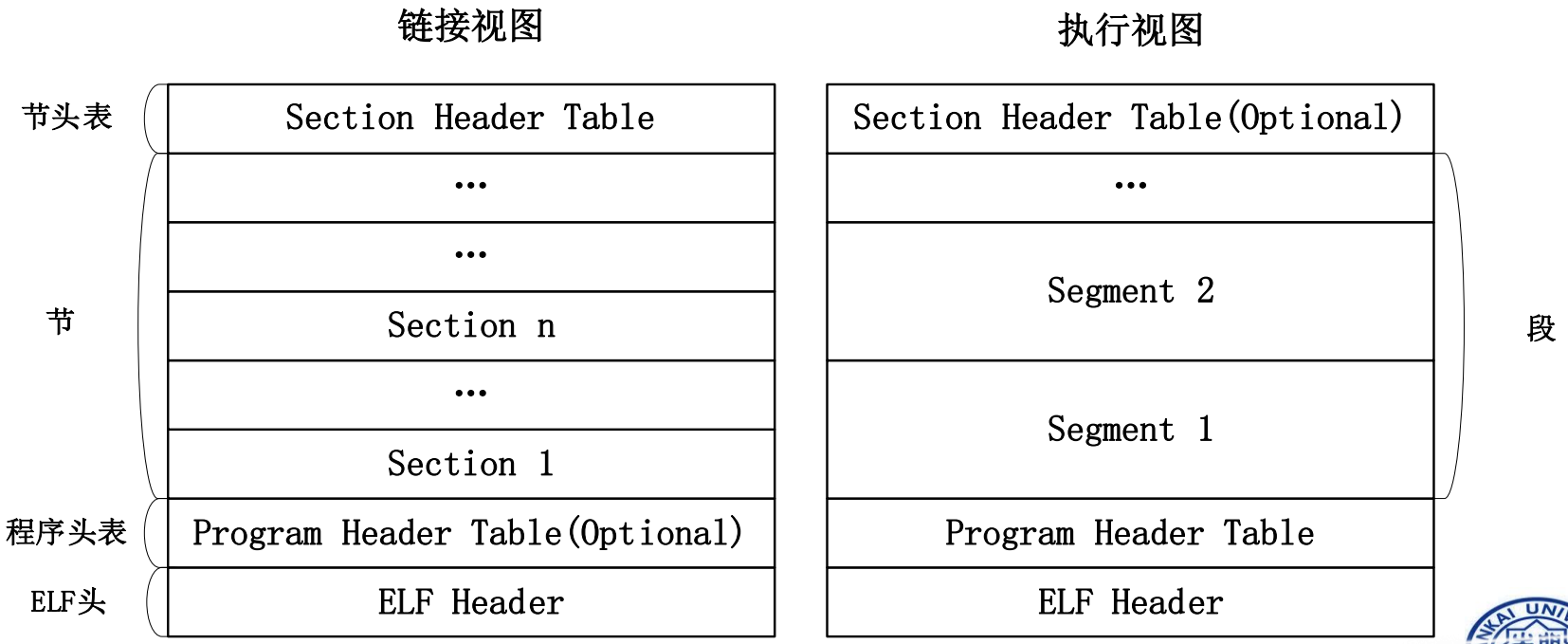
Linux系统可执行文件

- 在Linux系统中，最常见的可执行文件格式是ELF（Executable and Linkable Format）



Linux系统可执行文件

- 静态链接器会以链接视图解析ELF文件，动态链接器会以执行视图解析ELF文件并动态链接。



多选题 2分

设置

以下哪些是Windows可执行文件?

A .exe文件

B .dll文件

C .doc文件

D .txt文件

提交

南開大學 Nankai University

单选题 1分

设置

Windows可执行文件的格式是?

A PE

B ELF

C APK

D ZIP

提交

单选题 1分

设置

Linux系统可执行文件的格式是?

A PE

B ELF

C APK

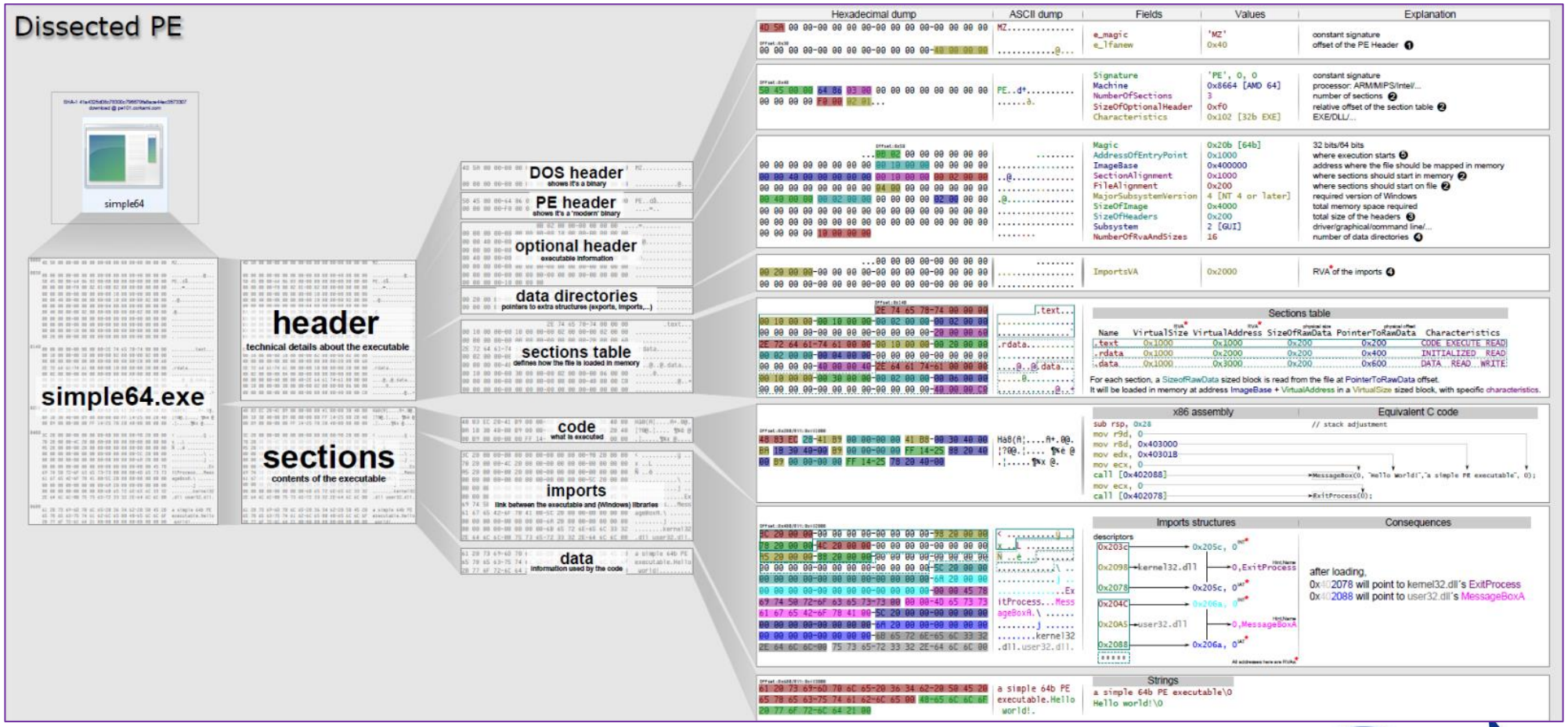
D ZIP

提交

南開大學 Nankai University

PE文件基本概念

PE文件结构

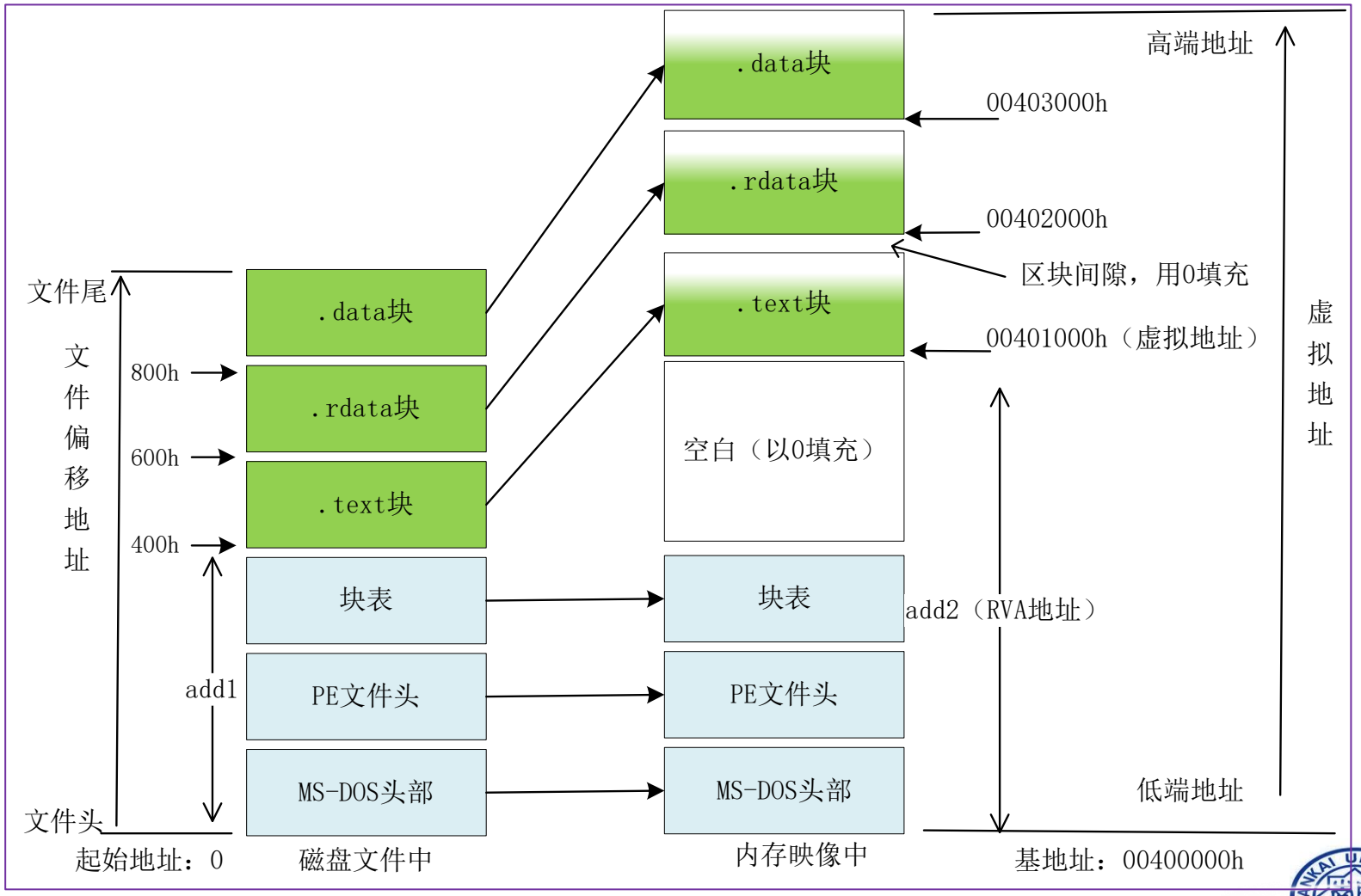

Dissected PE
simple64
simple64.exe
header
technical details about the executable
sections
contents of the executable
DOS header
shows it's a binary
PE header
shows it's a 'modern' binary
optional header
executable information
data directories
pointers to extra structures (exports, imports,...)
sections table
defines how the file is loaded in memory
code
what is executed
imports
link between the executable and (Windows) libraries
data
information used by the code
Hexadecimal dump
ASCII dump
Fields
Values
Explanation
e_magic
e_lfanew
Signature
Machine
NumberOfSections
SizeOfOptionalHeader
Characteristics
Magic
AddressOfEntryPoint
ImageBase
SectionAlignment
FileAlignment
MajorSubsystemVersion
SizeOfImage
SizeOfHeaders
Subsystem
NumberOfRvaAndSizes
ImportsVA
Sections table
Name VirtualSize VirtualAddress SizeOfRawData PointerToRawData Characteristics
.text 0x1000 0x1000 0x200 0x200 CODE EXECUTE READ
.rdata 0x1000 0x2000 0x200 0x400 INITIALIZED READ
.data 0x1000 0x3000 0x200 0x600 DATA READ WRITE
For each section, a SizeofRawData sized block is read from the file at PointerToRawData offset.
It will be loaded in memory at address ImageBase + VirtualAddress in a VirtualSize sized block, with specific characteristics.
x86 assembly
Equivalent C code
// stack adjustment
MessageBox(0, "Hello World!","a simple PE executable", 0);
ExitProcess(0);
Imports structures
Consequences
descriptors
kernel32.dll
user32.dll
0,ExitProcess
0,MessageBoxA
after loading,
0x402078 will point to kernel32.dll's ExitProcess
0x402088 will point to user32.dll's MessageBoxA
Strings
a simple 64b PE executable\0
Hello world!\0

PE文件结构

- PE文件使用的是线性地址空间
  - 所有代码和数据都在一个地址空间，组成PE文件
- 文件内容被分割为不同的节(Section，也叫做块、区块、段等)

节

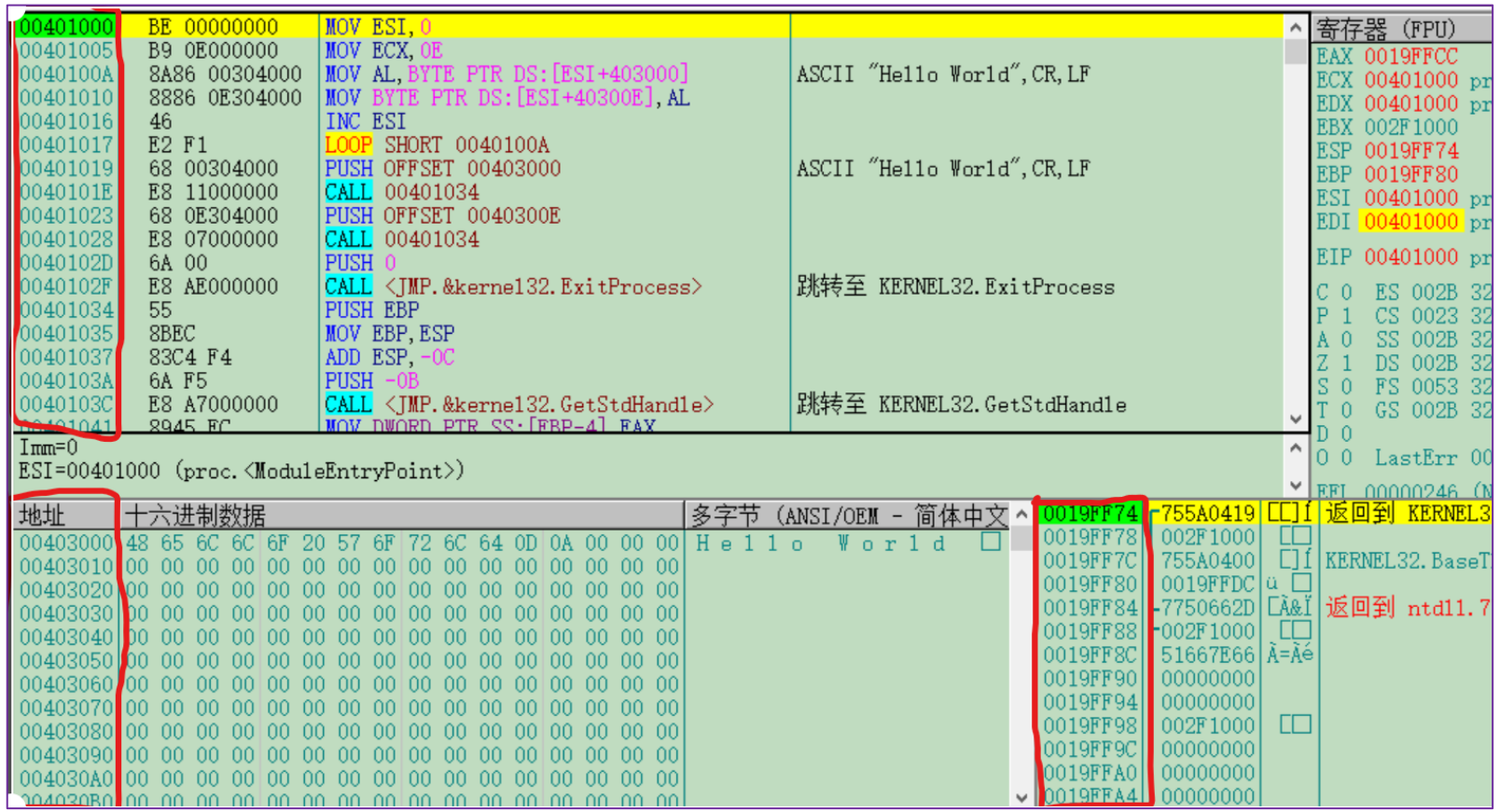
- 代码节、数据节
- 各个节按页边界对齐（4096Bytes）
- 节是一个连续结构，没有大小限制
- 每个节都有自己的内存属性
  - 内存有哪些属性？


高端地址
.data块
00403000h
.rdata块
00402000h
区块间隙，用0填充
.text块
00401000h（虚拟地址）
空白（以0填充）
虚拟地址
文件尾
文件偏移地址
800h
.data块
.rdata块
600h
.text块
400h
块表
块表
add2（RVA地址）
add1
PE文件头
PE文件头
MS-DOS头部
MS-DOS头部
低端地址
文件头
起始地址：0
磁盘文件中
内存映像中
基地址：00400000h

虚拟内存地址空间

- 每个程序都有自己的虚拟内存地址空间，虚拟空间的内存地址称为虚拟地址(Virtual Address，VA)
  - 4GB
  - 不同进程的虚拟地址空间是相互隔离的

虚拟地址


00401000 BE 00000000 MOV ESI,0
00401005 B9 0E000000 MOV ECX,0E
0040100A 8A86 00304000 MOV AL,BYTE PTR DS:[ESI+403000] ASCII "Hello World",CR,LF
00401010 8886 0E304000 MOV BYTE PTR DS:[ESI+40300E],AL
00401016 46 INC ESI
00401017 E2 F1 LOOP SHORT 0040100A
00401019 68 00304000 PUSH OFFSET 00403000 ASCII "Hello World",CR,LF
0040101E E8 11000000 CALL 00401034
00401023 68 0E304000 PUSH OFFSET 0040300E
00401028 E8 07000000 CALL 00401034
0040102D 6A 00 PUSH 0
0040102F E8 AE000000 CALL <JMP.&kernel32.ExitProcess> 跳转至 KERNEL32.ExitProcess
00401034 55 PUSH EBP
00401035 8BEC MOV EBP,ESP
00401037 83C4 F4 ADD ESP,-0C
0040103A 6A F5 PUSH -0B
0040103C E8 A7000000 CALL <JMP.&kernel32.GetStdHandle> 跳转至 KERNEL32.GetStdHandle
Imm=0
ESI=00401000 (proc.<ModuleEntryPoint>)
寄存器 (FPU)
EAX 0019FFCC
ECX 00401000
EDX 00401000
EBX 002F1000
ESP 0019FF74
EBP 0019FF80
ESI 00401000
EDI 00401000
EIP 00401000
C 0 ES 002B 32
P 1 CS 0023 32
A 0 SS 002B 32
Z 1 DS 002B 32
S 0 FS 0053 32
T 0 GS 002B 32
D 0
O 0 LastErr 00
地址 十六进制数据 多字节 (ANSI/OEM - 简体中文
00403000 48 65 6C 6C 6F 20 57 6F 72 6C 64 0D 0A 00 00 00 Hello World
0019FF74 755A0419 返回到 KERNEL3
0019FF78 002F1000
0019FF7C 755A0400 KERNEL32.BaseT
0019FF80 0019FFDC
0019FF84 7750662D 返回到 ntdll.7
0019FF88 002F1000
0019FF8C 51667E66
0019FF90 00000000
0019FF94 00000000
0019FF98 002F1000
0019FF9C 00000000
0019FFA0 00000000
0019FFA4 00000000

主观题 10分

设置

一个进程的虚拟地址空间中只有一个PE文件结构吗?

作答

模块

00400000	00001000	proc		PE 文件头
00401000	00001000	proc	.text	ã码
00402000	00001000	proc	.rdata	口入表
00403000	00001000	proc	.data	P据
00410000	000C5000			
004E0000	00006000			
006E0000	00005000			
75580000	00001000	KERNEL32		PE 文件头
75590000	00064000	KERNEL32	.text	ã码
75600000	0002F000	KERNEL32	.rdata	口入表,输出表
75630000	00001000	KERNEL32	.data	P据
75640000	00001000	KERNEL32	.rsrc	D源
75650000	00005000	KERNEL32	.reloc	f定位
76830000	00001000	KERNELBASE		PE 文件头
76831000	001C3000	KERNELBASE	.text	ã码,输出表
769F4000	00004000	KERNELBASE	.data	P据
769F8000	00006000	KERNELBASE	.idata	口入表
769FE000	00001000	KERNELBASE	.didat	
769FF000	00001000	KERNELBASE	.rsrc	D源
76A00000	0002A000	KERNELBASE	.reloc	f定位

模块

- 当PE文件通过Windows加载器载入内存后，内存中的版本称为模块(Module)
  - 映射文件的起始地址称为模块句柄(hModule)
  - 初始内存地址也称为基地址(ImageBase)

模块

- 内存中的模块代表进程将这个可执行文件所需要的代码、数据、资源、输入表、输出表及其它有用的数据结构都放在一个连续的内存节中

模块句柄

- Windows将Module的基地址作为Module的实例句柄(Instance Handle，即Hinstance)。
  - GetModuleHandle获得DLL句柄，通过句柄访问DLL Module的内容

```
HMODULE GetModuleHandle(LPCTSTR lpModuleName);
```

主观题 10分

设置

思考题：进程的内存空间中有多个模块。模块在内存中的位置是固定的吗？字符串、全局变量、函数等内存地址是固定的吗？

作答

相对虚拟地址

- 模块的地址冲突问题
  - 模块的加载顺序和加载地址是不确定的。
- 在可执行文件中，有许多地方需要内存地址
  - 例如，引用全局变量时需要指定它的地址
- PE文件有一个首选的载入地址(基地址)
- PE文件可以载入到进程空间任何地方

相对虚拟地址

- 为了避免绝对内存地址，引入了相对虚拟地址(Relative Virtual Address，RVA)的概念
  - RVA是相对于PE文件载入地址的偏移位置

相对虚拟地址

- 假设一个EXE文件从00400000h处载入内存
- 代码节起始地址为401000h，代码节起始地址的RVA计算方法如下:
  - 目标地址401000h - 载入地址400000h = RVA 1000h

相对虚拟地址

- 将RVA转换成虚拟地址VA的过程
  - 用实际的载入地址ImageBase加相对虚拟地址RVA
  - 虚拟地址(VA)=基地址(ImageBase)+相对虚拟地址(RVA)

文件偏移地址

- PE文件储存在磁盘中，某个数据的位置相对于文件头的偏移量称为文件偏移地址(File Offset)或物理地址(RAW Offset)。
  - 文件偏移地址从PE文件的第1个字节开始计数，起始值为0。

PE文件

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..............
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	........@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	........!..L.!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~..............
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	........}?......
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich............
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE..L......^....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	....
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	....
000000E0	00	20	00	00	00	00	40	00	00	10	00	00	00	02	00	00	. ....@.........
000000F0	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	....
00000100	00	40	00	00	00	04	00	00	00	00	00	00	03	00	00	00	.@..............
00000110	00	00	10	00	00	10	00	00	00	00	10	00	00	10	00	00	....
00000120	00	00	00	00	10	00	00	00	00	00	00	00	00	00	00	00	....
00000130	10	20	00	00	28	00	00	00	00	00	00	00	00	00	00	00	. ..(...........
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

主观题 10分

设置

PE文件结构为什么要使用相对虚拟地址?

作答

单选题 10分

设置

PE文件的偏移地址（RAW Offset）是否等于相对虚拟地址（RVA）？

A 是

B 不是

提交

主观题 10分

设置

hello.exe在内存中的基地址（ImageBase）是00400000h，入口点的相对虚拟地址RVA（hello.exe执行的第一条CPU指令的相对虚拟地址）是00001000h，hello.exe的入口点虚拟地址VA是？

正常使用主观题需2.0以上版本雨课堂

作答

DOS文件头

主观题 10分

设置

为什么Windows系统要用一个16位的DOS程序作为32位PE文件的头部?

作答

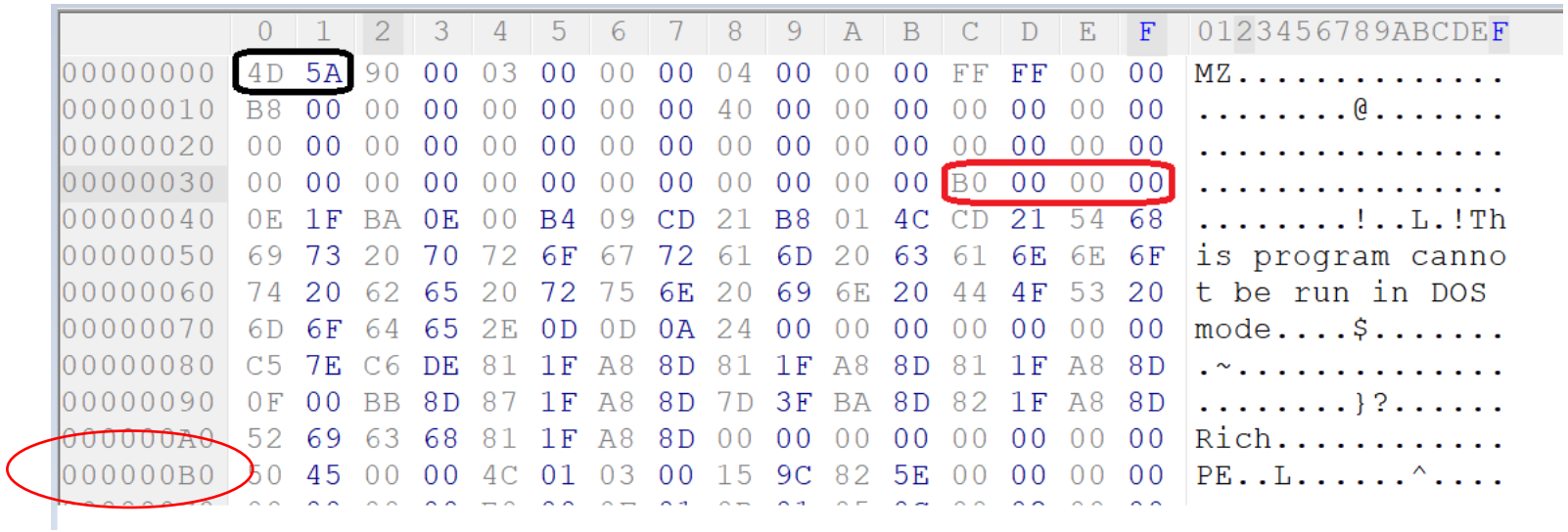
DOS文件头

- 每个PE文件都是以一个16位的DOS程序开始的
- DOS MZ头与DOS stub合称为DOS文件头

IMAGE_ DOS _HEADER

```
IMAGE_DOS_HEADER_STRUCT{
  +0h    e_magic      WORD  ?          ;DOS 可执行文件标记 "MZ"
  +2h    e_cblp       WORD  ?
  +4h    e_cp         WORD  ?
  +6h    e_crlc       WORD  ?
  +8h    e_cparhdr    WORD  ?
  +0ah   e_minalloc   WORD  ?
  +0ch   e_maxalloc   WORD  ?
  +0eh   e_ss         WORD  ?
  +10h   e_sp         WORD  ?
  +12h   e_csum       WORD  ?
  +14h   e_ip         WORD  ?          ;DOS 代码入口 IP
  +16h   e_cs         WORD  ?          ;DOS 代码入口 CS
  +18h   e_lfarlc     WORD  ?
  +1ah   e_ovno       WORD  ?
  +1ch   e_res        WORD  4 dup(?)
  +24h   e_oemid      WORD  ?
  +26h   e_oeminfo    WORD  ?
  +28h   e_res2       WORD 10 dup(?)
  +3ch   e_lfanew     DWORD ?          ;指向 PE 文件头"PE",0,0
} IMAGE_DOS_HEADER_ENDS
```

DOS文件头



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..............
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	........@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	........!..L.!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~..............
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	........}?......
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich............
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE..L......^....

单选题 10分

设置

所有PE文件的前两个字节是确定的吗?

A 是

B 不是

提交

填空题 1分

设置

PE文件开始的两个字节是 [填空1]

作答

主观题 10分

设置

如何通过DOS Stub结构找到PE头在文件中的位置?

作答

PE文件头

PE文件头

- PE文件头(PE Header)紧跟在DOS stub的后面
- IMAGE_DOS_HEADER结构的e_lfanew字段定位PE Header的起始偏移量，加上基址，得到PE文件头的指针
  - PNTHeader = ImageBase+DosHeader->e_lfanew

PE文件头

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..............
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	........@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	........!..L.!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~..............
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	........}?......
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich............
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE..L......^....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	....
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	....

PE文件头

- Signature
- FileHeader
- OptionalHeader

IMAGE_NT_HEADERS32 structure (winnt.h)

04/02/2021 • 2 minutes to read

Represents the PE header format.

Syntax

C++ Copy

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_nt_headers32

Signature

- 在一个有效的PE文件，Signature字段的值是00004550h
  - ASCII字符是“PE\0\0”

```
#define IMAGE_NT_SIGNATURE                  0x00004550
```

FileHeader

- IMAGE_FILE_HEADER结构中记录了PE文件的一些基本信息，并指出了IMAGE_OPTIONAL_HEADER的大小

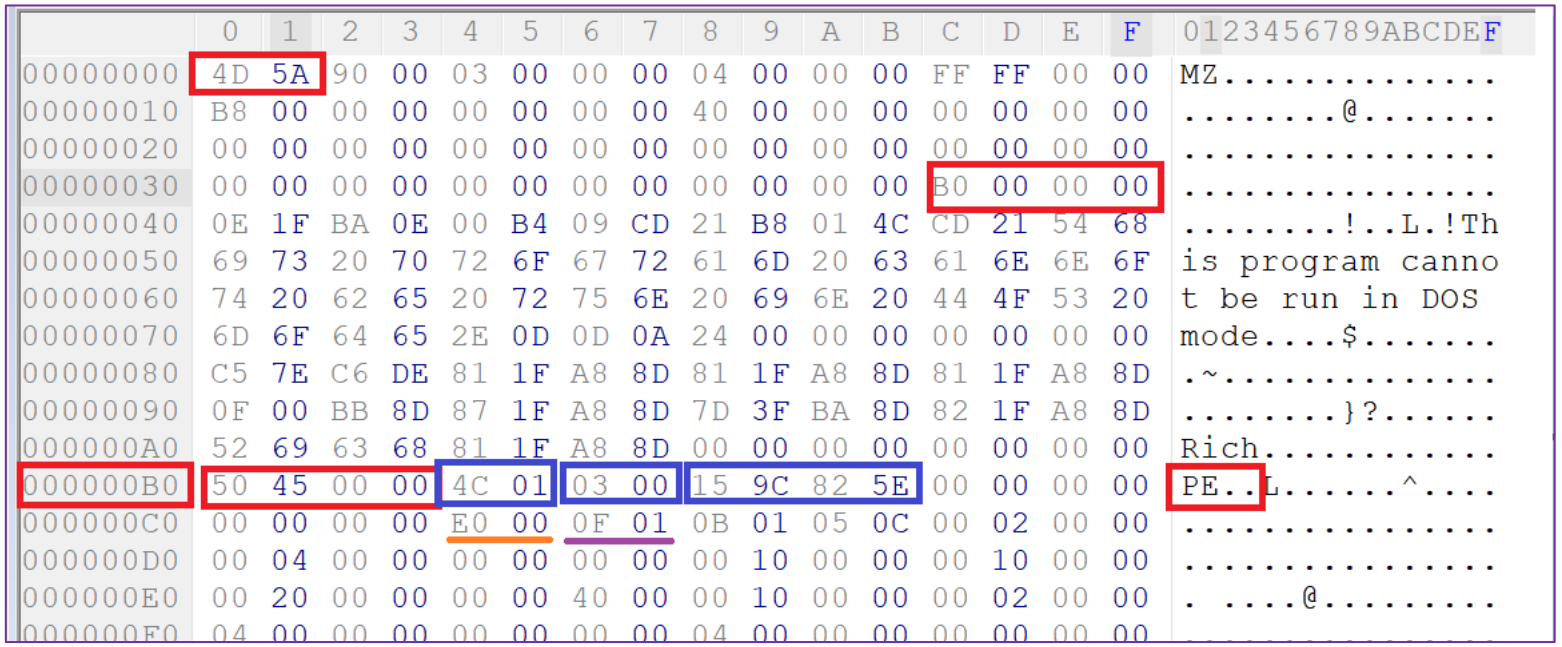
Syntax

C++

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

```
IMAGE_FILE_HEADER STRUCT
  +04h Machine                 WORD    ?    ;运行平台
  +06h NumberOfSections        WORD    ?    ;文件的区块数
  +08h TimeDateStamp           DWORD   ?    ;文件创建日期和时间
  +0Ch PointerToSymbolTable    DWORD   ?    ;指向符号表（用于调试）
  +10h NumberOfSymbols         DWORD   ?    ;符号表中符号的个数（用于调试）
  +14h SizeOfOptionalHeader    WORD    ?    ;IMAGE_OPTIONAL_HEADER32 结构的大小
  +16h Characteristics         WORD    ?    ;文件属性
IMAGE_FILE_HEADER ENDS
```

FileHeader



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..............
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	........@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	........!..L.!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~..............
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	........}?......
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich............
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE..L......^....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	....
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	....
000000E0	00	20	00	00	00	00	40	00	00	10	00	00	00	02	00	00	. ....@.........
000000F0	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	....

Machine

- Machine
  - 可执行文件的目标CPU类型
  - PE文件可以在多种CPU机器上使用，不同平台上指令的机器码不同。

Value	Meaning
IMAGE_FILE_MACHINE_I386 0x014c	x86
IMAGE_FILE_MACHINE_IA64 0x0200	Intel Itanium
IMAGE_FILE_MACHINE_AMD64 0x8664	x64

https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_file_header

NumberOfSections

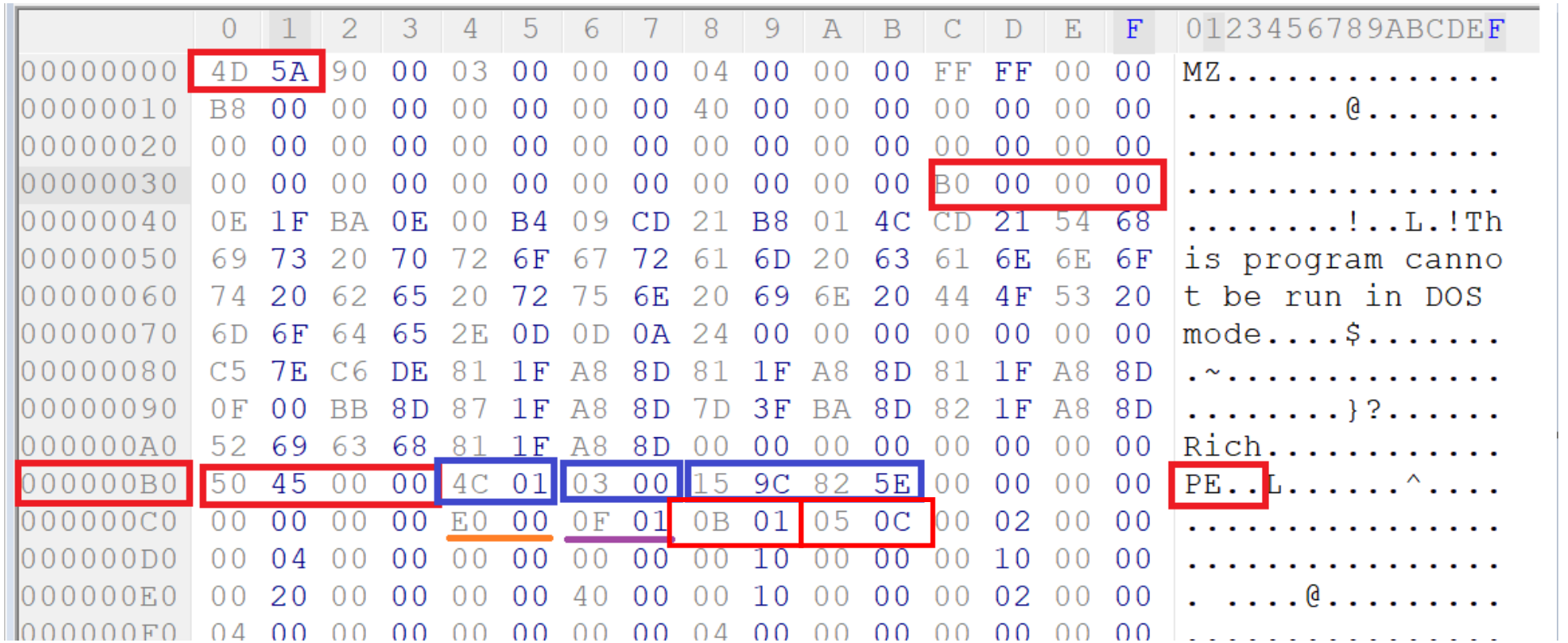
- NumberOfSections记录节(Section)的数量
  - 节表紧跟在IMAGE_NT_HEADERS后面定义

TimeDateStamp

- TimeDateStamp
  - 文件的创建时间
  - 1970年1月1日到创建该文件的所有的秒数

Characteristics

特 征 值	含 义
0001h	文件中不存在重定位信息
0002h	文件可执行。如果为 0，一般是链接时出问题了
0004h	行号信息被移去
0008h	符号信息被移去
0020h	应用程序可以处理超过 2GB 的地址。该功能是从 NT SP3 开始被支持的。因为大部分数据库服务器需要很大的内存，而 NT 仅提供 2GB 给应用程序，所以从 NT SP3 开始，通过加载 /3GB 参数，可以使应用程序被分配 2～3GB 区域的地址，而该处原来属于系统内存区
0080h	处理机的低位字节是相反的
0100h	目标平台为 32 位机器
0200h	.DBG 文件的调试信息被移去
0400h	如果映像文件在可移动介质中，则先复制到交换文件中再运行
0800h	如果映像文件在网络中，则复制到交换文件后才运行
1000h	系统文件
2000h	文件是 DLL 文件
4000h	文件只能运行在单处理器上
8000h	处理机的高位字节是相反的



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ..............
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	........@.......
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	B0	00	00	00	....
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	........!..L.!Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....$.......
00000080	C5	7E	C6	DE	81	1F	A8	8D	81	1F	A8	8D	81	1F	A8	8D	.~..............
00000090	0F	00	BB	8D	87	1F	A8	8D	7D	3F	BA	8D	82	1F	A8	8D	........}?......
000000A0	52	69	63	68	81	1F	A8	8D	00	00	00	00	00	00	00	00	Rich............
000000B0	50	45	00	00	4C	01	03	00	15	9C	82	5E	00	00	00	00	PE..L......^....
000000C0	00	00	00	00	E0	00	0F	01	0B	01	05	0C	00	02	00	00	....
000000D0	00	04	00	00	00	00	00	00	00	10	00	00	00	10	00	00	....
000000E0	00	20	00	00	00	00	40	00	00	10	00	00	00	02	00	00	. ....@.........
000000F0	04	00	00	00	00	00	00	00	04	00	00	00	00	00	00	00	....

主观题 10分

设置

hello.exe的PE结构中Character值是010Fh，说明hello.exe的文件属性是什么？

正常使用主观题需2.0以上版本雨课堂

作答

IMAGE _OPTIONAL_HEADER

- 可选映像头(IMAGE_ OPTIONAL _HEADER)结构中定义了更多的PE文件数据
- IMAGE_FILE_HEADER和IMAGE_OPTIONAL_HEADER合在一起构成PE文件头

IMAGE_OPTIONAL_HEADER

```
//IMAGE_OPTIONAL_HEADER结构(可选映像头)
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //
    WORD    Magic;  //幻数，一般为10BH
    BYTE    MajorLinkerVersion;  //链接程序的主版本号
    BYTE    MinorLinkerVersion;  //链接程序的次版本号
    DWORD   SizeOfCode;  //代码段大小
    DWORD   SizeOfInitializedData;  //已初始化数据块的大小
    DWORD   SizeOfUninitializedData;  //未初始化数据库的大小
    DWORD   AddressOfEntryPoint;  //程序开始执行的入口地址,这是一个RVA（相对虚拟地址）
    DWORD   BaseOfCode; //代码段的起始RVA 一般来说 是 1000h
    DWORD   BaseOfData; //数据段的起始RVA
    //
```

https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header32

IMAGE_OPTIONAL_HEADER

```
    //
    // NT additional fields.
    //
    DWORD   ImageBase; //可执行文件默认装入的基地址
    DWORD   SectionAlignment; //内存中块的对齐值（默认的块对齐值为1000H，4KB个字节）
    DWORD   FileAlignment;//文件中块的对齐值（默认值为200H字节，为了保证块总是从磁盘的扇区开始的）
    WORD    MajorOperatingSystemVersion;//要求操作系统的最低版本号的主版本号
    WORD    MinorOperatingSystemVersion;//要求操作系统的最低版本号的次版本号
    WORD    MajorImageVersion;//该可执行文件的主版本号
    WORD    MinorImageVersion;//该可执行文件的次版本号
    WORD    MajorSubsystemVersion;//要求最低之子系统版本的主版本号 默认 0004
    WORD    MinorSubsystemVersion;//要求最低之子系统版本的次版本号 默认 0000
    DWORD   Win32VersionValue;//保留字                                    默认00000000
    DWORD   SizeOfImage;//映像装入内存后的总尺寸                          一般为00004000 映射到内存中一个块1000内存
    DWORD   SizeOfHeaders;//部首及块表的大小
    DWORD   CheckSum;//CRC检验和                                          一般为00000000
    WORD    Subsystem;//程序使用的用户接口子系统
    WORD    DllCharacteristics;//DLLmain函数何时被调用，默认为0
    DWORD   SizeOfStackReserve;//初始化时堆栈大小
    DWORD   SizeOfStackCommit;//初始化时实际提交的堆栈大小
    DWORD   SizeOfHeapReserve;//初始化时保留的堆大小
    DWORD   SizeOfHeapCommit;//初始化时实际提交的对大小
    DWORD   LoaderFlags;//与调试有关，默认为0
    DWORD   NumberOfRvaAndSizes;//数据目录结构的数目
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];//数据目录表
}
```

IMAGE_OPTIONAL_HEADER

- Magic（WORD）:这是一个标记字，说明文件是ROM映像(0107h)还是普通可执行的映像(010Bh)
  - 010Bh
- MajorLinkerVersion（BYTE）:链接程序的主版本号
  - 05h
- MinorLinkerVersion(BYTE):链接程序的次版本号
  - 0Ch

IMAGE_OPTIONAL_HEADER

- SizeOfCode(DWORD)：代码段的大小
  - 00000200h
- SizeOflnitializedData(DWORD): 已初始化数据块的大小：

  00000400h
- SizeOfUninitializedData(DWORD):未初始化数据块的大小：

  00000000h

IMAGE_OPTIONAL_HEADER

- AddressOfEntryPoint（DWORD）程序执行的入口RVA：00001000h
- BaseOfCode （DWORD）代码段的起始RVA：00001000h
- BaseOfData （DWORD）数据段的起始RVA：00002000h
- ImageBase （DWORD）文件在内存中的载入地址
  - 00400000h

IMAGE_OPTIONAL_HEADER

- SectionAlignment(DWORD)： 内存上区块间的对齐大小
  - 00001000h
  - 多少字节？
- FileAlignment(DWORD):硬盘上区块间的对齐大小
  - 00000200h
  - 多少字节？

IMAGE_OPTIONAL_HEADER

- MajorOperatingSystemVersion（WORD）
- MinorOperatingSystemVersion（WORD）
  - PE文件执行所需要的Windows系统最低版本号
  - 0004、0000
  - Windows NT 的内部版本号是 4.0
  - Win2000是5.0、XP是5.1、Vista是6.0、Win7是6.1、Win8是6.2、Win10是10.0、Win11是10.0

IMAGE_OPTIONAL_HEADER

- MajorImageVersion(WORD)
- MinorImageVersion(WORD)
  - 程序的主版本号和次版本号
  - 由程序员自己定义
  - 0000 0000

IMAGE_OPTIONAL_HEADER

- MajorSubsystemVersion(WORD)
- MinorSubsystemVersion(WORD)
  - PE文件所要求的子系统的版本号
  - 0004 0000
- Win32VersionValue(DWORD)：通常被设置为0

IMAGE_OPTIONAL_HEADER

- SizeOfImage（DWORD）：映像载入内存后的总大小，是指载入文件从ImageBase到最后一个块的大小
  - 00004000h
- SizeOfHeaders（DWORD）：MS-DOS头部、PE文件头、区块表的总尺寸
  - 00000400h

IMAGE_OPTIONAL_HEADER

- SizeOfStackReserve （DWORD）
- SizeOfStackCommit （DWORD）
- SizeOfHeapReserve （DWORD）
- SizeOfHeapCommit （DWORD）
  - 栈、堆的设置信息
- LoaderFlags （DWORD） :与调试有关

IMAGE_DATA_DIRECTORY 数据目录

- NumberOfRvaAndSizes（DWORD）：数据目录的项数
- DataDirectory[16]:数据目录表

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0   // 导出表
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1   // 导入表
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2   // 资源
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3   // 异常
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4   // 安全
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5   // 重定位表
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6   // 调试信息
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT       7   // (X86 usage)
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7   // 版权信息
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8   // RVA of GP
#define IMAGE_DIRECTORY_ENTRY_TLS             9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // 导入函数地址表
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT   13   // Delay Load Import Descriptors
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14   // COM Runtime descriptor
```

IMAGE_DATA_DIRECTORY 数据目录

Syntax

C++

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

- VirtualAddress
  - 该目录的相对虚拟地址
  - RVA
- Size
  - 该目录的大小

https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_data_directory

数据目录

- 定位PE文件的导入表、导出表、资源的时候， 需要用到数据目录

单选题 10分

设置

以下哪个字段存储了PE文件头在文件中的偏移地址？

A e_magic

B e_ip

C e_cs

D e_lfanew

提交

南開大學 Nankai University

多选题 10分

设置

PE文件头包括哪几部分？

A Signature

B FileHeader

C OptionalHeader

D DOS Header

提交

南開大學 Nankai University

多选题 10分

设置

以下哪些数据项可以计算出程序入口点的虚拟地址VA？

A ImageBase

B BaseOfCode

C AddressOfEntryPoint

D SizeOfCode

提交

南開大學 Nankai University

节

节表

- 保证程序的安全性
  - 把code和data放在同一个内存节中相互纠缠，很容易引发安全问题
  - code有可能被data覆盖，导致崩溃
  - PE文件格式将内存属性相同的数据统一保存在一个被称为“节”（Section）的地方

主观题 10分

设置

PE文件格式将内存属性相同的数据统一保存在节中。节表中应该记录哪些节的相关信息?

作答

IMAGE_SECTION_HEADER(winnt.h)

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_section_header

节表

Offset	0	1	2	3	4	5	6	7	8	9	A①	B	C	D	E	F	
000001A0	00	00	00	00	00③	00	00	00	2E	74	65	78	74	00⑤	01	59	.....text..Y
000001B0②	9A	01	00	00	00	10	00	00④	00	02	00	00	00	04	00	00	?.....
000001C0⑥	00	00	00	00	00	00	00	00	00	00	00	00	20	00	00	60	....`
000001D0	2E	72	64	61	74⑦	61	00	00	C2⑧	01	00⑨	00	00	20⑩	00	00	.rdata..?... ..
000001E0	00	02	00	00	00	06	00	00	00	00	00	00	00	00	00	00	.....
000001F0	00	00	00	00	40	00	00	40	2E	64	61	74	61	00	00	00	....@..@.data...
00000200	38	00	00	00	00	30	00	00	00	02	00	00	00	08	00	00	8....0.....
00000210	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	C0	.....@..@

图 11.9　十六进制工具中的块表

节表

- Name（8 BYTE）：块名
  - An 8-byte, null-padded UTF-8 string.
  - There is no terminating null character if the string is exactly eight characters long.
- VirtualSize（DWORD）：在内存空间中，节的大小
  - The total size of the section when loaded into memory, in bytes.
- VirtualAddress（DWORD）：节在内存空间中的起始RVA
  - The address of the first byte of the section when loaded into memory, relative to the image base.

节表

- SizeOfRawData（DWORD）：该节在硬盘中所占的空间
  - The size of the initialized data on disk, in bytes.
- PointerToRawData（DWORD）：该节在硬盘中的偏移
  - A file pointer to the first page within the COFF file.
  - This value must be a multiple of the FileAlignment member of the IMAGE_OPTIONAL_HEADER structure.

单选题 1分

设置

VirtualSize是否需要与SizeOfRawData一致?

A 是

B 不是

提交

节表

- PointerToReLocations （DWORD）：在EXE文件中无意义
- PointerToLinenumbers （DWORD）：行号表在文件中的偏移量
- NumberOfReLocations （WORD）：在EXE文件中无意义
- NumberOfLinenumbers （WORD）：该块在行号表中的行号数目

节的内存属性

- Characteristics（DWORD）：块属性
  - IMAGE_SCN_MEM_EXECUTE
    - 20000000h，可执行
  - IMAGE_SCN_MEM_READ
    - 40000000h，可读
  - IMAGE_SCN_MEM_WRITE
    - 80000000h，可写

多选题 10分

设置

.text节的属性值是60000020h，则.text节的内存属性是？

A 可执行

B 可读

C 可写

提交

南開大學 Nankai University

多选题 10分

设置

.rdata节的属性值是40000040h，则.rdata节的内存属性是？

A 可执行

B 可读

C 可写

提交

南開大學 Nankai University

多选题 10分

设置

.data节的属性值是0C0000040h，则.data节的内存属性是？

A 可执行

B 可读

C 可写

提交

南開大學 Nankai University

节的内容

- IMAGE_SCN_CNT_CODE
  - 00000020h，包含可执行代码
- IMAGE_SCN_CNT_INITIALIZED_DATA
  - 00000040h，包含已初始化数据
- IMAGE_SCN_CNT_UNINITIALIZED_DATA
  - 00000080h，包含未初始化数据

多选题 10分

设置

.text节的属性值是60000020h，则.text节的内容是？

A 代码

B 已初始化数据

C 未初始化数据

提交

多选题 10分

设置

.rdata节的属性值是40000040h，则.rdata节的内容是？

A 代码

B 已初始化数据

C 未初始化数据

提交

多选题 10分

设置

.data节的属性值是0C0000040h，则.data节的内存属性是？

A 代码

B 已初始化数据

C 未初始化数据

提交

常见的节

- .text，代码节，链接器把所有目标文件的.text节连接成一个大的.text节
- .data，读、写数据节，全局标量
- .rdata，只读数据节，调试目录、字符串等

常见的节

- .idata，导入表
- .edata，导出表
- .rsrc，资源数据，菜单、图标、位图等
- .bss，未初始化数据，被.data取代，增加VirtualSize到足够放下未初始化数据

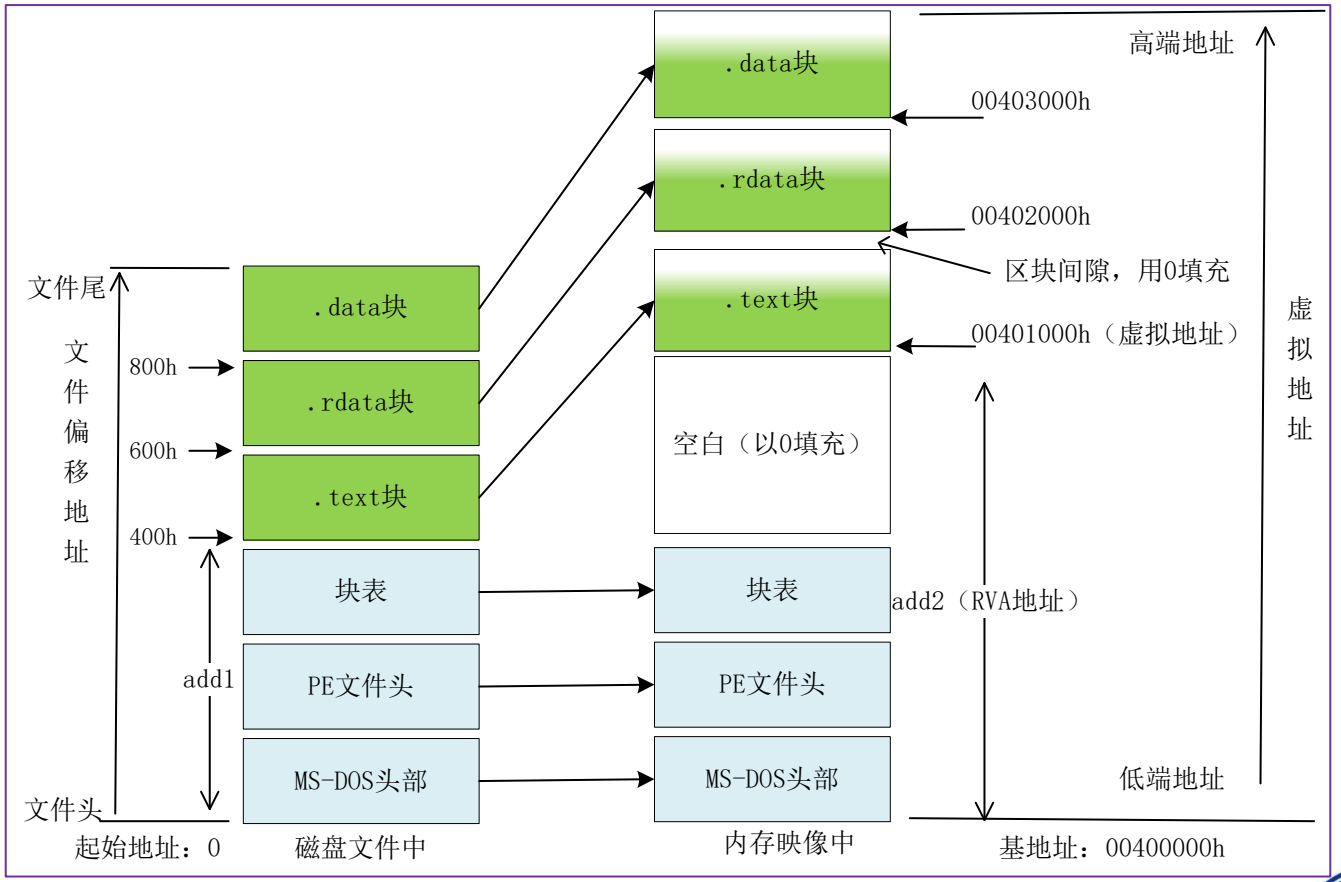
节的对齐

- 硬盘上的对齐
  - FileAlignment
  - 200h，扇区对齐，节间隙
- 内存上的对齐
  - SectionAlignment
  - 1000h，内存页对齐（64位系统，8KB内存页）

文件偏移与虚拟内存地址转换

- Image（映像）
- PE文件加载到内存时，不会原封不动地加载
  - 根据节表的定义加载
- PE文件与内存中的Image具有不同的形态

文件偏移与虚拟内存地址转换



文件偏移与虚拟内存地址转换

- 文件被映射到内存中时，MS-DOS头部、PE文件头和节表的偏移位置与大小均没有变化
- 各节被映射到内存中后，其偏移位置就发生了变化

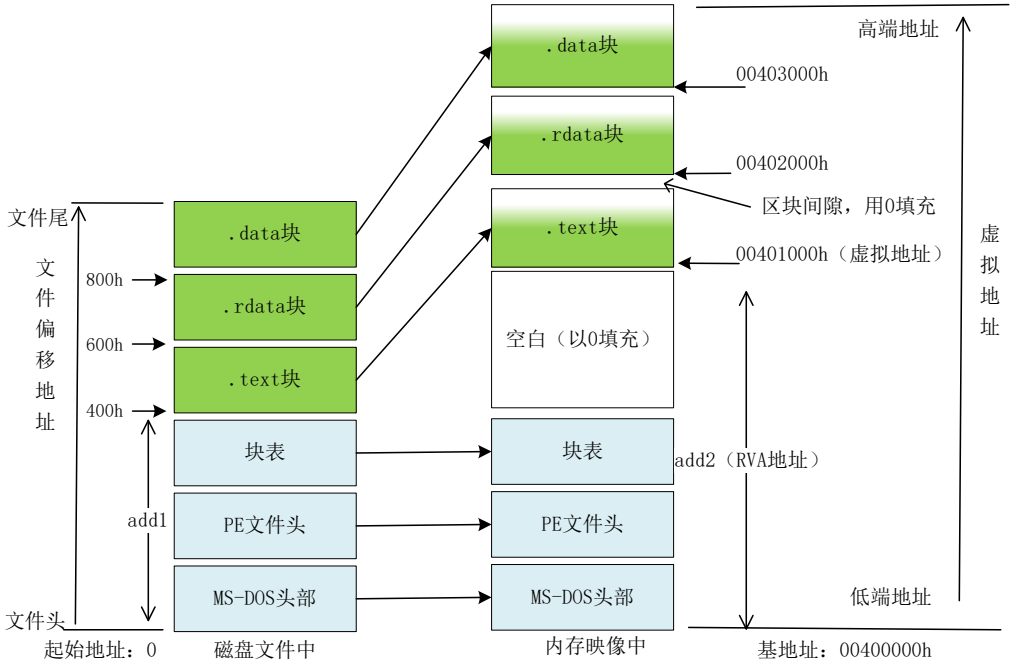
文件偏移与虚拟内存地址转换

- RVA to RAW
  - RAW – PointerToRawData = RVA – VirtualAddress
  - RAW = RVA – VirtualAddress + PointerToRawData

主观题 10分

设置

PE文件在内存中的映射未知如图所示， 相对内存地址RVA=2123h的数据在PE文件中的Offset是？



正常使用主观题需2.0以上版本雨课堂

作答

南開大學 Nankai University

RVA to RAW

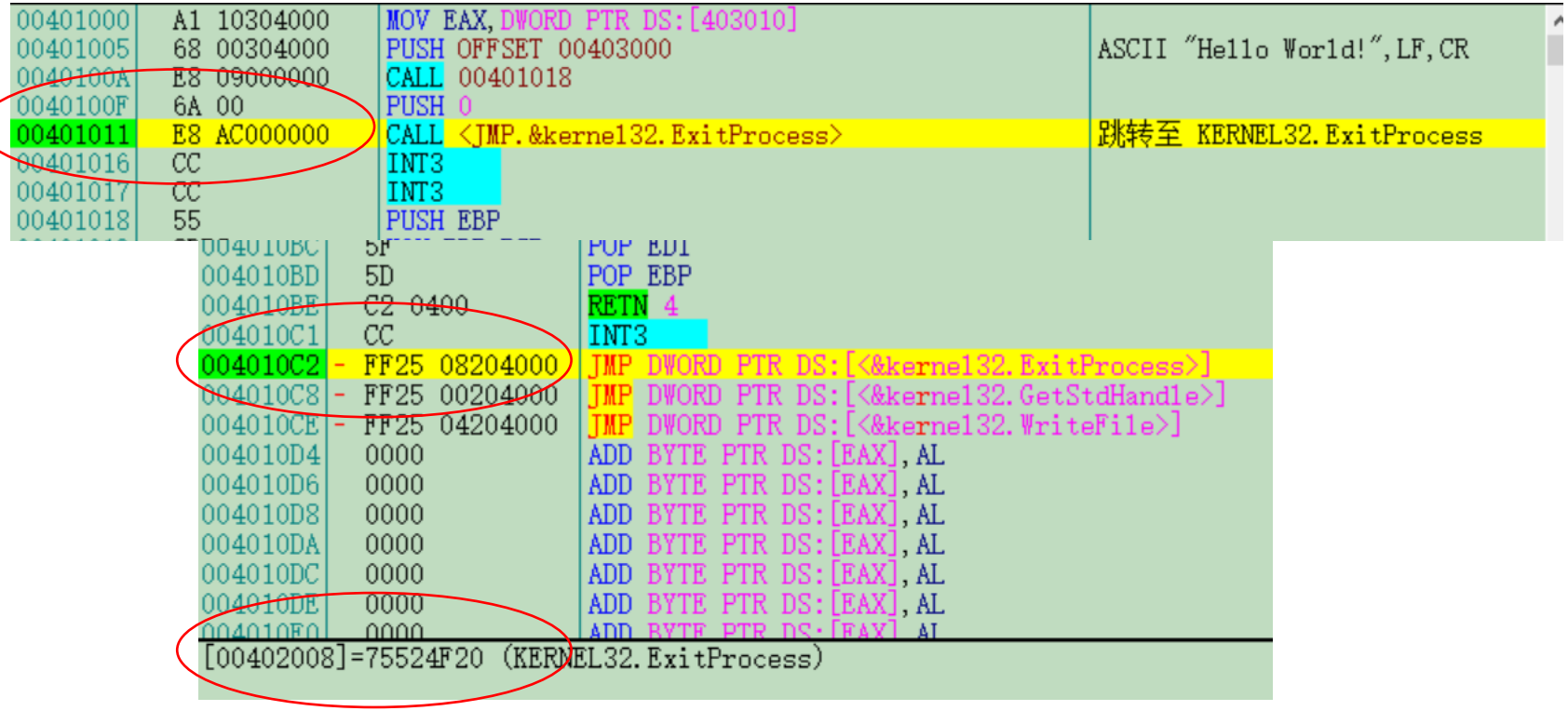
- RVA=2123h在.rdata节
  - .rdata节的相对内存地址范围是2000h到3000h
- VirtualAddress = 2000h
- PointerToRawData=600h
- RAW=2123h（RVA）- 2000h（VirtualAddress）+600h（PointerToRawData）=**723**h

RVA to RAW

- RVA与RAW（文件偏移）间的相互变换时PE头的最基本内容，需要熟悉并掌握。

导入表 IAT

hello.exe



为什么不直接使用call 75524F20调用函数？

JMP-jump

- **近跳转（Near jump）**
  - A jump to an instruction within the current code segment (the segment currently pointed to by the CS register)
  - E9 → JMP rel32/rel16 （长度5字节）
- **短跳转 （Short jump）**
  - A near jump where the jump range is limited to –128 to +127 from the current EIP value.
  - EB → JMP rel8 （长度2字节）
- **远跳转（Far jump）**
  - A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
  - FF 25 → JMP m16:32 （长度6字节）
    JMP DWORD PTR **DS**:[<&kernel32.ExitProcess>]

导入表IAT

- 操作系统的版本不同
- kernel32.dll的版本不同
- DLL重定位
  - ImageBase值在不同程序内存空间中是不一样的

主观题 10分

设置

为什么需要导入表（Import Table），系统函数、库函数的内存地址在编译和链接的时候是不是已经确定了？

作答

导入表 IAT

- 导入表 Import Address Table（IAT）
- 学习PE文件结构，最难过的一关就是IAT
- IAT中的内容与Windows操作系统的核心进程、内存、DLL结构等有关
  - “理解了IAT，就掌握了Windows操作系统的根基”

导入表IAT

- IAT是一种表结构
- 标记程序需要使用哪些库中的哪些函数
- IMAGE_IMPORT_DESCRIPTOR（IID）
- IMAGE_IMPORT_BY_NAME

IMAGE_IMPORT_DESCRIPTOR（IID）

IMAGE_IMPORT_DESCRIPTOR结构体中记录PE文件要导入哪些库文件

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;
    } DUMMYUNIONNAME;
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
    DWORD   Name;
    DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

IMAGE_IMPORT_DESCRIPTOR

- PE程序往往需要导入多个库
- 导入多少个库，就存在多少个IMAGE_IMPORT_DESCRIPTOR结构体
- 多个结构体组成数组，以NULL结构体结束

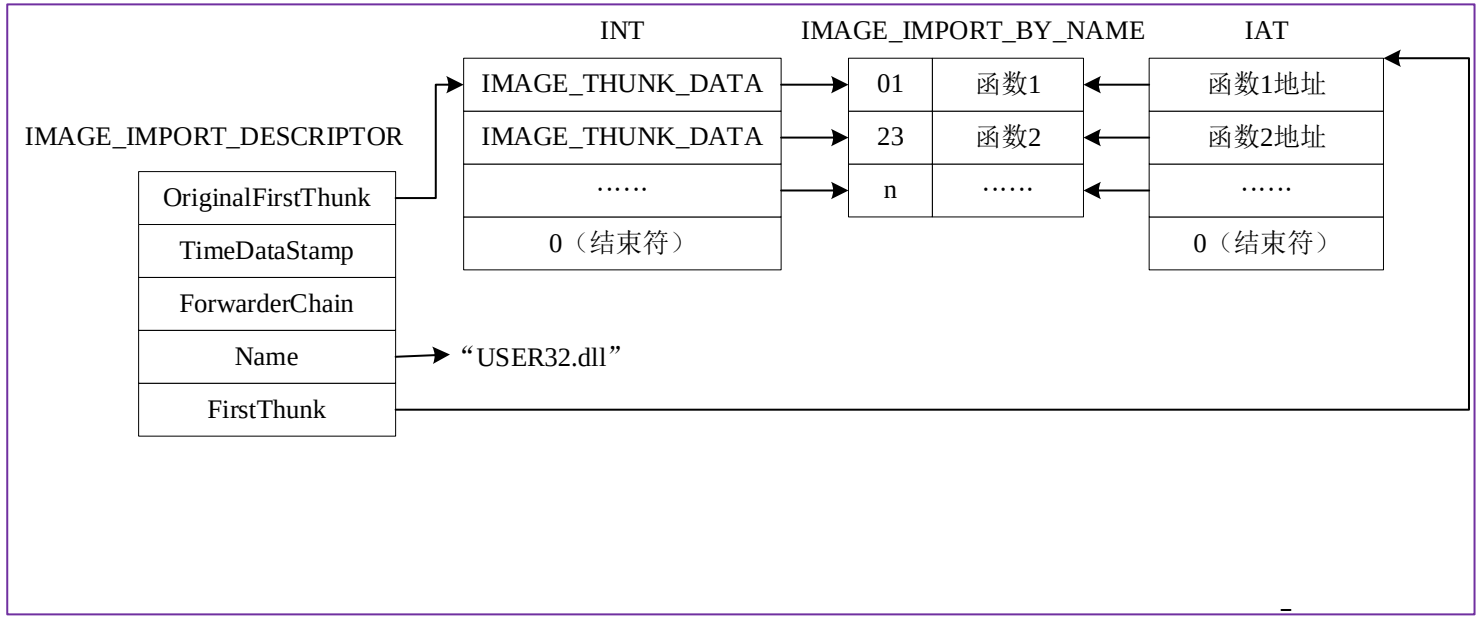
IMAGE_IMPORT_DESCRIPTOR

- Name （DWORD），库文件名字符串的地址（RVA）
- OriginalFirstThunk（DWORD），INT(Import Name Table)的地址（RVA）
- FirstThunk （DWORD），IAT(Import Address Table)的地址（RVA）

IMAGE_IMPORT_DESCRIPTOR

- Table
  - 在PE头中，Table即指数组
- INT与IAT是DWORD数组，以NULL结束
- INT与IAT的各元素指向相同地址

INT（Import Name Table），IAT（Import Address Table）



IMAGE_IMPORT_BY_NAME

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    CHAR   Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

- Hint 是函数在DLL引出表中的索引号。
  - PE装载器用Hint在DLL的引出表里快速查询函数
  - 该值不是必须的，一些连接器将此值设为0。
- Name 引入函数的函数名
  - 函数名是一个ASCII字符串。
  - **是可变尺寸域**，以NULL结尾

PE装载器

- 1. 读取IID的name成员，获取库名称字符串，例如“kernel32.dll”
- 2. 装载相应的库，类似LoadLibrary（“kernel32.dll”）

PE装载器

- 3. 读取IID的OriginalFirstThunk成员，获取INT地址
- 4. 读取INT，逐一获得IMAGE_IMPORT_BY_NAME的地址（RVA）

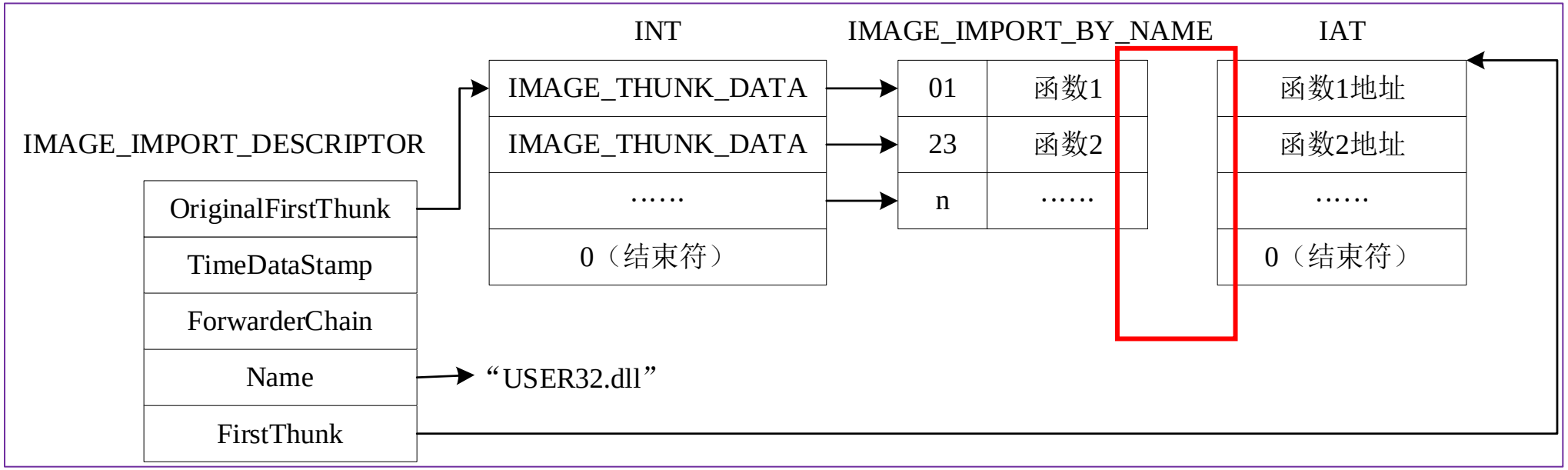
PE装载器

- 5. 使用IMAGE_IMPORT_BY_NAME的Hint或者Name项，获得函数的起始地址
  - 类似GetProcAddress（“ExitProcess”）
- 6. 读取IID的FirstThunk成员，获得IAT地址

PE装载器

- 7. 将第5步获得的函数地址写入IAT数组相应位置
- 8. 重复步骤4到7，直到INT结束

PE文件加载后的IAT



单选题 10分

设置

下面哪个结构的数据，在PE文件装载到内存中会产生变化？

A Import Name Table(INT)

B Import Address Table(IAT)

C IMAGE_IMPORT_DESCRIPTOR(IID)

D IMAGE_IMPORT_BY_NAME

提交

单选题 10分

设置

库函数的内存地址存储在哪个结构中？

A INT

B IAT

C IMAGE_IMPORT_DESCRIPTOR(IID)

D IMAGE_IMPORT_BY_NAME

提交

南開大學 Nankai University

单选题 10分

设置

函数名字符串存储在哪个数据结构中？

A IAT

B INT

C IMAGE_IMPORT_DESCRIPTOR(IID)

D IMAGE_IMPORT_BY_NAME

提交

南開大學 Nankai University

填空题 20分

设置

Windows装载PE文件时，将库函数的内存地址动态地写入PE文件的导入表。 下面4个数据结构的访问顺序是：[填空1] [填空2] [填空3] [填空4]（填写A、B、C、D）

A IAT

B INT

C IMAGE_IMPORT_DESCRIPTOR(IID)

D IMAGE_IMPORT_BY_NAME

正常使用填空题需3.0以上版本雨课堂

作答

主观题 10分

设置

导入表有哪些安全问题？如何进行安全加固？

正常使用主观题需2.0以上版本雨课堂

作答

南開大學 Nankai University

导出表 EAT

DLL

- Windows操作系统提供了数量庞大的库函数
  - 进程、内存、窗口、消息、文件、网络等
- 同时运行多个程序时，每个进程都包含相同的库，严重浪费内存
- Dynamic Link Libary(DLL)，内存映射

IMAGE_EXPORT_DIRECTORY

- 导出表是DLL的核心机制
  - 不同程序可以调用库文件中提供的函数
  - 通过EAT，得到库文件导出函数的入口地址

IMAGE_EXPORT_DIRECTORY

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of image
    DWORD   AddressOfNames;         // RVA from base of image
    DWORD   AddressOfNameOrdinals;  // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

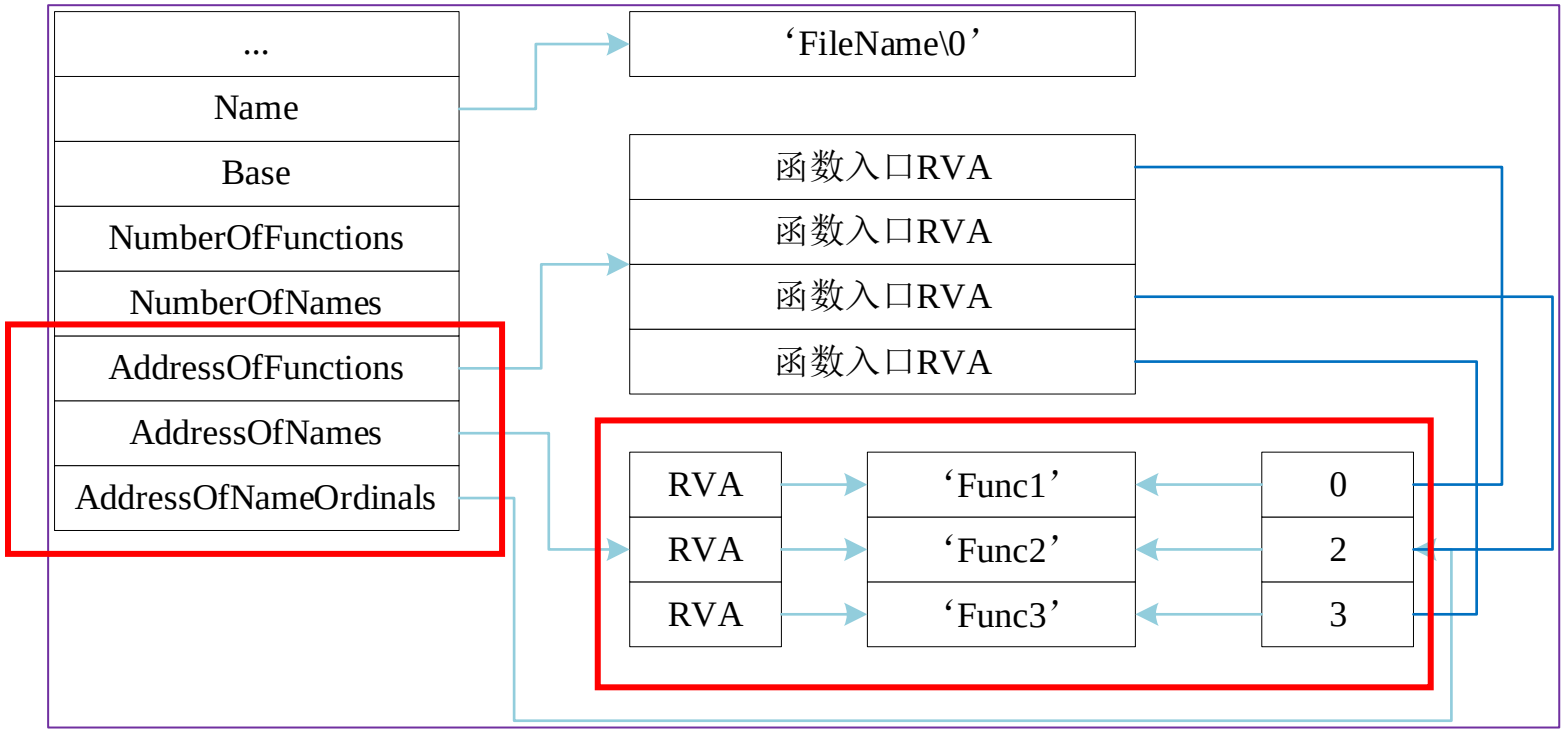
IMAGE_EXPORT_DIRECTORY

- Name
  - 库文件名字符串地址
- NumberOfFunctions
  - 实际Export函数的个数
- NumberOfNames
  - Export函数中具有名字的函数个数

IMAGE_EXPORT_DIRECTORY

- AddressOfFunctions
  - Export函数地址数组
- AddressOfNames
  - 函数名称地址数组
- AddressOfNameOrdinals
  - Ordinal地址数组

导出表


...
Name
Base
NumberOfFunctions
NumberOfNames
AddressOfFunctions
AddressOfNames
AddressOfNameOrdinals
‘FileName\0’
函数入口RVA
函数入口RVA
函数入口RVA
函数入口RVA
RVA
RVA
RVA
‘Func1’
‘Func2’
‘Func3’
0
2
3

GetProcAddress()操作原理

- 从库中获得函数地址的API为GetProcAddress()函数
  - 如何通过EAT获得函数地址？

GetProcAddress()操作原理

- 1. AddressOfName定位“函数名称数组”
- 2. 在“函数名称数组”中，通过比较字符串（strcmp），查找指定的函数名称
  - 此时的数组索引称为name_index
- 3. 利用AddressOfNameOridinals成员，定位ordinal数组

GetProcAddress()操作原理

- 4. 在ordinal数组中，通过name_index查找相应的ordinal值
- 5. AddressOfFunctions，定位“函数地址数组”（EAT）
- 在“函数地址数组”中，利用ordinal值作为索引，获得指定函数的起始地址

导出表

- 如果函数是以序号导出的，那么查找的时候直接用序号减去Base，得到的值就是函数在AddressOfFunctions中的下标

单选题 10分

设置

IMAGE_EXPORT_DIRECTORY结构的哪个成员指向DLL的“函数名称数组”？

A AddressOfNames

B AddressOfFunctions

C AddressOfNameOrdinals

D NumberOfNames

提交

南開大學 Nankai University

单选题 10分

设置

IMAGE_EXPORT_DIRECTORY结构的哪个成员指向DLL的“函数地址数组”（EAT）？

A AddressOfNames

B AddressOfFunctions

C AddressOfNameOrdinals

D NumberOfNames

提交

南開大學 Nankai University

填空题 30分

设置

给定库函数的函数名，定位该函数的起始地址的过程中，访问IMAGE_EXPORT_DIRECTORY结构成员的顺序是 [填空1] [填空2] [填空3] (直接填写A、B、C)

A. AddressOfFunctions

B. AddressOfNames

C. AddressOfOrdinals

正常使用填空题需3.0以上版本雨课堂

作答

本章知识点

1. 可执行文件
2. PE文件基本概念
   - 难点：虚拟地址空间、相对虚拟地址
3. DOS文件头
   - 重点：MZ头、定位PE头
4. PE文件头
   - 重点：AddressOfEntryPoint、ImageBase、数据目录
5. 节
   - 难点：文件偏移RAW到内存地址RVA的转换方法
6. 导入表
   - 难点：INT（Import Name Table）与IAT（Import Address Table）
7. 导出表
   - 难点：AddressOfNames，AddressOfNameOrdinals，AddressOfFunctions

汇编语言与逆向技术

第8章 PE文件结构

王志
zwang@nankai.edu.cn

南开大学 网络空间安全学院

2024-2025学年