第 5 章

执行 PE 节中的 shellcode

“知己知彼，百战不殆。”对于逆向分析恶意代码，必须明白 shellcode 可能隐藏的位置，这样才能更好地分析，从而提取和分析 shellcode。本章将介绍 shellcode 可能存储在 PE 文件中的节位置，以及如何执行节中的 shellcode，最终能够提取并分析 shellcode 代码功能。

5.1 嵌入 PE 节的原理

PE 文件的结构是分节的，将文件分成若干节区(Section)，不同的资源放在不同的节区中，PE 文件常见节区如表 5-1 所示。

表 5-1 PE 文件常见节区和功能

节 名 称	功 能
.text	存放可执行的二进制机器码，例如局部变量
.data	存放初始化的数据，例如全局变量
.rsrc	存放程序的资源文件，例如图标

在 PE 文件的不同节区中保存 shellcode 代码，在一定程度上可以做到隐藏 shellcode，从而做到免杀的效果。

注意：免杀指程序防止杀毒软件的检测，即防止编码程序被杀毒软件作为计算机恶意程序删除。

5.1.1 内存中执行 shellcode 原理

在计算机操作系统中，无法直接执行 shellcode，但可以通过代码将 shellcode 加载到内存执行。内存中执行 shellcode 的流程可划分为申请内存空间、将 shellcode 复制到内存空间、将内存空间设置为可执行状态、执行内存空间中的 shellcode，如图 5-1 所示。

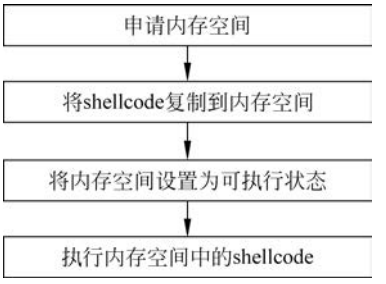

图 5-1 内存中执行 shellcode 流程

在 Windows 操作系统中，使用 C 语言调用 Windows

API 函数可实现在内存中执行 shellcode 二进制代码，其中每步操作都需要调用不同的函数。

5.1.2　常用 Windows API 函数介绍

Windows API 就是 Windows 应用程序接口，是针对 Microsoft Windows 操作系统家族的系统编程接口，其中 32 位 Windows 操作系统的编程接口常被称为 Win32 API。

对于程序员来讲，Windows API 就是一个应用程序接口。在这个接口中，Windows 操作系统提供给应用程序可调用的函数，使程序员无须考虑底层代码实现或理解内部原理，只考虑调用函数实现对应功能。

实现在内存执行 shellcode 二进制代码的过程中，需要分别考虑每步具体调用的 API 函数。

第 1 步，应用程序调用 VitualAlloc 函数从当前进程内存中申请可用空间，代码如下：

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,              //分配内存空间的起始地址,设置为 0 时系统自动分配
    SIZE_T dwSize,                 //分配内存空间的大小
    DWORD flAllocationType,        //分配内存的类型
    DWORD flProtect                //内存保护类型
);
```

注意：将 flAllocationType 设置为 MEM_COMMIT | MEM_RESERVE，用于保留和提交内存页面。将 flProtect 设置为 PAGE_READWRITE，用于保证申请到内存页面可读可写。函数成功申请到内存空间后，返回内存空间的起始地址。

第 2 步，应用程序调用 RtlMoveMemory 函数将 shellcode 二进制机器码复制到新申请的内存空间中，代码如下：

```
VOID RtlMoveMemory(
    VOID UNALIGNED * Destination,          //将字节复制到的目标地址
    const VOID UNALIGNED * Source,         //复制字节的源地址
    SIZE_T           Length                //将源地址复制到目标地址的字节数
);
```

注意：将 Destination 设置为申请的内存空间起始地址，将 Source 设置为 shellcode 二进制机器码起始地址，将 Length 设置为 shellcode 二进制机器码的字节数。函数没有返回值。

第 3 步，应用程序调用 VirtualProtect 函数并将当前进程中内存空间更改为可执行状态，代码如下：

```
BOOL VirtualProtect(
    LPVOID lpAddress,                    //内存空间的起始地址
    SIZE_T dwSize,                       //内存空间大小的字节数
    DWORD flNewProtect,                  //内存保护选项
    PDWORD lpflOldProtect                //原始内存保护选项,设置为 0 即可
);
```

注意：将 lpAddress 设置为申请的内存空间起始地址，将 dwSize 设置为申请的内存空间大小字节数，将 flNewProtect 设置为 PAGE_EXECUTE_READ，使内存空间页面可读可执行。将函数内存空间成功设置为可执行状态后，返回非零值。

第 4 步，应用程序调用 CreateThread 函数在当前进程下创建新线程，执行 shellcode 二进制机器码，代码如下：

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES     lpThreadAttributes,
    SIZE_T                    dwStackSize,
    LPTHREAD_START_ROUTINE    lpStartAddress, //线程的起始地址
    __drv_aliasesMem LPVOID   lpParameter,
    LPDWORD                   lpThreadId
);
```

注意：在 CreateThread 函数中，将 lpStartAddress 设置为分配的内存空间起始地址，将其他参数设置为 0。

在以上步骤中调用 Windows API 函数可在内存空间中执行 shellcode 二进制机器码。将 shellcode 二进制代码存储在数组变量后，将数组定义在代码的不同位置，使 shellcode 二进制代码存储在 PE 程序不同的节区。

5.1.3 scdbg 逆向分析 shellcode

虽然无法轻易识别 shellcode 二进制代码的功能，但是借助 scdbg 工具可以分析 shellcode 二进制代码调用的 Windows API 函数，从而理解 shellcode 二进制代码的作用。

scdbg 是一款多平台开源的 shellcode 模拟运行、分析工具。其基于 libemulibrary 搭建的虚拟环境，通过模拟 32 位处理器、内存和基本 Windows API 运行环境来虚拟执行 shellcode 以分析其行为。

无论是从互联网下载 shellcode，还是使用本地工具生成 shellcode，大多数情况下 shellcode 以数组的形式保存，代码如下：

```
//第 5 章/shellcode.txt
unsigned char shellcode[] =          #定义 shellcode 数组
    "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
```

```
"\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
"\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
"\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
"\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
"\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
"\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
"\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
"\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
"\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
"\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
"\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
"\x8E\x03\xD3\x52\x33\xFF\x57\x68\x61\x72"
"\x79\x41\x68\x4C\x69\x62\x72\x68\x4C\x6F"
"\x61\x64\x54\x53\xFF\xD2\x68\x33\x32\x01"
"\x01\x66\x89\x7C\x24\x02\x68\x75\x73\x65"
"\x72\x54\xFF\xD0\x68\x6F\x78\x41\x01\x8B"
"\xDF\x88\x5C\x24\x03\x68\x61\x67\x65\x42"
"\x68\x4D\x65\x73\x73\x54\x50\xFF\x54\x24"
"\x2C\x57\x68\x4F\x5F\x6F\x21\x8B\xDC\x57"
"\x53\x53\x57\xFF\xD0\x68\x65\x73\x73\x01"
"\x8B\xDF\x88\x5C\x24\x03\x68\x50\x72\x6F"
"\x63\x68\x45\x78\x69\x74\x54\xFF\x74\x24"
"\x40\xFF\x54\x24\x40\x57\xFF\xD0";
```

如果 shellcode 二进制代码中没有任何注释，则无法理解 shellcode 二进制代码的功能。此时可以在操作系统中执行 shellcode 二进制代码，根据代码执行后的变化来了解其功能，但这样会造成安全威胁，因此不建议通过以上方法分析 shellcode 二进制代码的功能。

使用 scdbg 工具建立虚拟环境分析 shellcode 二进制代码的前提，需要将数组形式的 shellcode 二进制代码转换为纯二进制格式。

首先使用字符替换网站对 shellcode 二进制代码数组进行处理，如图 5-2 所示。

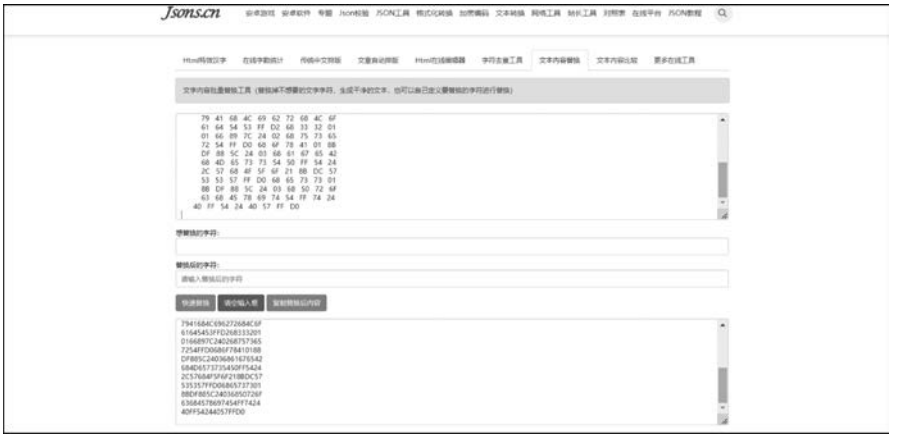

图 5-2　字符替换网站处理 shellcode 二进制代码

获取处理完毕的代码后，使用 Python 对结果代码再次进行处理，代码如下：

```
//第 5 章/test.py
shellcode = '''
FC33D2B23064FF325A8B
520C8B52148B722833C9
B11833FF33C0AC3C617C
022C20C1CF0D03F8E2F0
81FF5BBC4A6A8B5A108B
1275DA8B533C03D3FF72
348B527803D38B722003
F333C941AD03C3813847
65745075F4817804726F
634175EB817808646472
6575E2498B722403F366
8B0C4E8B721C03F38B14
8E03D35233FF57686172
7941684C696272684C6F
61645453FFD268333201
0166897C240268757365
7254FFD0686F7841018B
DF885C24036861676542
684D6573735450FF5424
2C57684F5F6F218BDC57
535357FFD06865737301
8BDF885C24036850726F
636845786974 54FF7424
40FF54244057FFD0
'''

shellcode = "".join(shellcode.split())      #删除空格和换行
print(shellcode.encode())

with open("shellcode.bin","wb") as f:       #将结果保存到 shellcode.bin 文件
    f.write(shellcode.encode())
```

在 cmd.exe 命令提示符窗口中执行 test.py，生成 shellcode.bin 文件，在此文件中保存着 shellcode 二进制代码，如图 5-3 所示。

图 5-3　生成 shellcode.bin 文件

使用 scdbg.exe 程序加载 shellcode.bin 文件，分析 shellcode 代码中调用的 Windows API 函数，命令如下：

```
scdbg.exe /f shellcode.bin      #/f 参数加载二进制文件并分析
```

如果 scdbg.exe 成功分析二进制文件，则会输出分析结果，如图 5-4 所示。

```
D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章>scdbg.exe /f shellcode.bin
Loaded 1dc bytes from file shellcode.bin
Detected straight hex encoding input format converting...
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

401092  GetProcAddress(LoadLibraryA)
4010a4  LoadLibraryA(user32)
4010bf  GetProcAddress(MessageBoxA)
4010cd  MessageBoxA(O_o!, O_o!)
4010eb  GetProcAddress(ExitProcess)
4010ee  ExitProcess(0)

Stepcount 2695
```

图 5-4　scdbg 分析 shellcode 结果

从结果可以得出，当前 shellcode 仅执行 MessageBoxA()函数输出“O_o!, O_o!”字符串，并没有调用其他可能存在安全威胁的函数。

5.2　嵌入 PE .text 节区的 shellcode

局部变量也称为内部变量，是指在一个函数内部或复合语句内部定义的变量，保存于 PE 文件结构的.text 节区，如图 5-5 所示。

使用 C 语言编写程序，将存储 shellcode 二进制代码的数组声明为 main 函数的局部变量。编译源代码生成可执行程序，此时会将局部变量的值保存到 PE 可执行程序的.text 节区。

首先，准备输出对话框提示信息的 shellcode 二进制代码，代码如下：

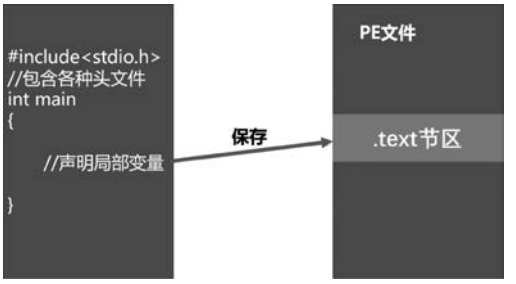

图 5-5　局部变量保存到 PE 文件的.text 节区

```
//第 5 章/shellcode.txt
unsigned chaR Shellcode[ ] =          //定义 shellcode 数组
    "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
    "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
    "\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
    "\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
    "\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
    "\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
    "\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
    "\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
```

```
    "\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
    "\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
    "\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
    "\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
    "\x8E\x03\xD3\x52\x33\xFF\x57\x68\x61\x72"
    "\x79\x41\x68\x4C\x69\x62\x72\x68\x4C\x6F"
    "\x61\x64\x54\x53\xFF\xD2\x68\x33\x32\x01"
    "\x01\x66\x89\x7C\x24\x02\x68\x75\x73\x65"
    "\x72\x54\xFF\xD0\x68\x6F\x78\x41\x01\x8B"
    "\xDF\x88\x5C\x24\x03\x68\x61\x67\x65\x42"
    "\x68\x4D\x65\x73\x73\x54\x50\xFF\x54\x24"
    "\x2C\x57\x68\x4F\x5F\x6F\x21\x8B\xDC\x57"
    "\x53\x53\x57\xFF\xD0\x68\x65\x73\x73\x01"
    "\x8B\xDF\x88\x5C\x24\x03\x68\x50\x72\x6F"
    "\x63\x68\x45\x78\x69\x74\x54\xFF\x74\x24"
    "\x40\xFF\x54\x24\x40\x57\xFF\xD0";
```

获取 shellcode 二进制机器码后，编写 C 语言程序加载执行 shellcode，代码如下：

```
//第 5 章/PEtext.cpp
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {

    void * alloc_mem;
    BOOL retval;
    HANDLE threadHandle;
DWORD oldprotect = 0;

    unsigned chaR Shellcode[] =          //定义 shellcode 数组
    "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
    "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
    "\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
    "\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
    "\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
    "\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
    "\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
    "\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
    "\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
    "\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
    "\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
    "\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
    "\x8E\x03\xD3\x52\x33\xFF\x57\x68\x61\x72"
    "\x79\x41\x68\x4C\x69\x62\x72\x68\x4C\x6F"
    "\x61\x64\x54\x53\xFF\xD2\x68\x33\x32\x01"
    "\x01\x66\x89\x7C\x24\x02\x68\x75\x73\x65"
```

```
    "\x72\x54\xFF\xD0\x68\x6F\x78\x41\x01\x8B"
    "\xDF\x88\x5C\x24\x03\x68\x61\x67\x65\x42"
    "\x68\x4D\x65\x73\x73\x54\x50\xFF\x54\x24"
    "\x2C\x57\x68\x4F\x5F\x6F\x21\x8B\xDC\x57"
    "\x53\x53\x57\xFF\xD0\x68\x65\x73\x73\x01"
    "\x8B\xDF\x88\x5C\x24\x03\x68\x50\x72\x6F"
    "\x63\x68\x45\x78\x69\x74\x54\xFF\x74\x24"
    "\x40\xFF\x54\x24\x40\x57\xFF\xD0";

    unsigned int lengthOfshellcodePayload = sizeof shellcode;

    //申请内存空间
    alloc_mem = VirtualAlloc(0, lengthOfshellcodePayload, MEM_COMMIT | MEM_RESERVE, PAGE_
READWRITE);

    //将 shellcode 复制到分配好的内存空间
    RtlMoveMemory(alloc_mem, shellcode, lengthOfshellcodePayload);

    //将内存空间设定为可执行状态
     retval = VirtualProtect(alloc_mem, lengthOfshellcodePayload, PAGE_EXECUTE_READ,
&oldprotect);

    printf("\nPress Enter to Create Thread!\n");
    getchar();

    //如果设定成功,则以线程的方式执行 shellcode 代码
    if ( retval != 0 )
      {
        threadHandle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) alloc_mem,0, 0, 0);
        WaitForSingleObject(threadHandle, -1);
    }

    return 0;
}
```

打开 Windows 操作系统中的 x64 Native Tools Command Prompt for VS 2022 命令提示符终端后,使用 cl.exe 应用程序编译 PEtext.cpp 文件,命令如下:

```
cl.exe /nologo /Ox /MT /W0 /GS- /DNDebug /TcPEtext.cpp /link /OUT:PEtext.exe /SUBSYSTEM:
CONSOLE /MACHINE:x64
```

如果成功编译源代码文件,则会在当前工作路径下生成 PEtext.exe 可执行程序,如图 5-6 所示。

双击运行 PEtext.exe 程序,此时会弹出提示对话框,如图 5-7 所示。

程序执行后,按 Enter 键会弹出提示对话框。

```
D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章>cl.exe /nologo /Ox /MT /W0 /GS- /DNDEBUG /Tc PEtext.cpp /link /OUT:PEtext.exe /SU
BSYSTEM:CONSOLE /MACHINE:x64
PEtext.cpp

D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章>dir
 驱动器 D 中的卷是 软件
 卷的序列号是 5817-9A34

 D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章 的目录

2022/10/19  16:45    <DIR>          .
2022/10/19  16:45    <DIR>          ..
2022/10/19  14:55    <DIR>          .idea
2022/10/19  16:38             2,226 PEtext.cpp
2022/10/19  16:45           128,512 PEtext.exe
2022/10/19  16:45             3,508 PEtext.obj
2022/10/19  14:26    <DIR>          scdbg
2019/10/17  18:46           853,504 scdbg.exe
2022/10/19  15:24               476 shellcode.bin
2022/10/19  14:37             1,225 shellcode.txt
2022/10/19  15:24               687 test.py
               7 个文件        990,138 字节
               4 个目录 29,880,553,472 可用字节

D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章>
```

编译成功,生成PEtext.exe可执行程序

图 5-6　cl. exe 编译 PEtext. cpp 源代码文件

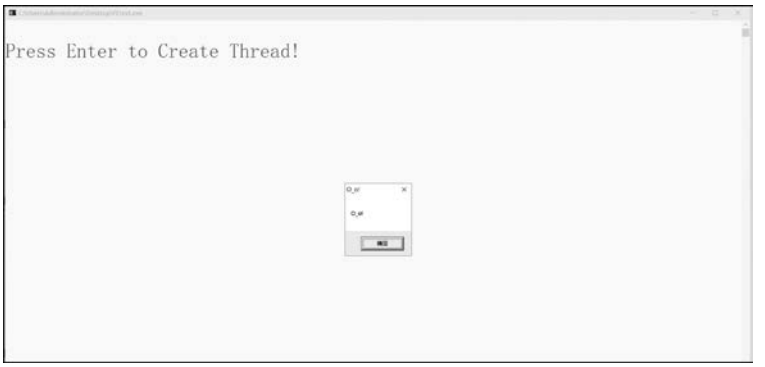

图 5-7　运行 PEText. exe 可执行程序

5.3　嵌入 PE . data 节区的 shellcode

全局变量也称为外部变量,是指定义在函数外部,可以在程序的任意位置使用的变量。全局变量保存于 PE 文件结构的. data 节区,如图 5-8 所示。

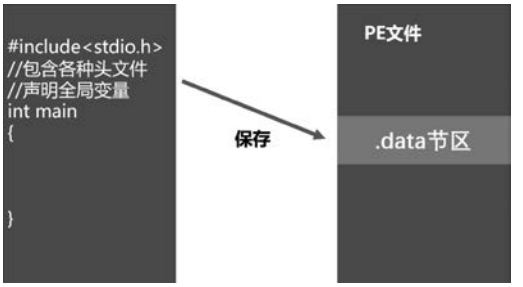

图 5-8　全局变量保存到 PE 文件的. data 节区

使用 C 语言编写程序,将存储 shellcode 二进制代码的数组声明为 main 函数外部变量。编译源代码生成可执行程序,此时会将全部变量的值保存到 PE 可执行程序的. data 节区。

首先，准备输出对话框提示信息的 shellcode 二进制代码，代码如下：

```
//第 5 章/shellcode.txt
unsigned chaR Shellcode[ ] =          //定义 shellcode 数组
    "\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
    "\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
    "\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
    "\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
    "\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
    "\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
    "\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
    "\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
    "\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
    "\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
    "\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
    "\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
    "\x8E\x03\xD3\x52\x33\xFF\x57\x68\x61\x72"
    "\x79\x41\x68\x4C\x69\x62\x72\x68\x4C\x6F"
    "\x61\x64\x54\x53\xFF\xD2\x68\x33\x32\x01"
    "\x01\x66\x89\x7C\x24\x02\x68\x75\x73\x65"
    "\x72\x54\xFF\xD0\x68\x6F\x78\x41\x01\x8B"
    "\xDF\x88\x5C\x24\x03\x68\x61\x67\x65\x42"
    "\x68\x4D\x65\x73\x73\x54\x50\xFF\x54\x24"
    "\x2C\x57\x68\x4F\x5F\x6F\x21\x8B\xDC\x57"
    "\x53\x53\x57\xFF\xD0\x68\x65\x73\x73\x01"
    "\x8B\xDF\x88\x5C\x24\x03\x68\x50\x72\x6F"
    "\x63\x68\x45\x78\x69\x74\x54\xFF\x74\x24"
    "\x40\xFF\x54\x24\x40\x57\xFF\xD0";
```

获取 shellcode 二进制机器码后，编写 C 语言程序加载执行 shellcode，代码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned chaR Shellcode[ ] =          //定义 shellcode 数组
"\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
"\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
"\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
"\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
"\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
"\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
"\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
"\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
"\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
"\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
"\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
"\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
```

```
"\x8E\x03\xD3\x52\x33\xFF\x57\x68\x61\x72"
"\x79\x41\x68\x4C\x69\x62\x72\x68\x4C\x6F"
"\x61\x64\x54\x53\xFF\xD2\x68\x33\x32\x01"
"\x01\x66\x89\x7C\x24\x02\x68\x75\x73\x65"
"\x72\x54\xFF\xD0\x68\x6F\x78\x41\x01\x8B"
"\xDF\x88\x5C\x24\x03\x68\x61\x67\x65\x42"
"\x68\x4D\x65\x73\x73\x54\x50\xFF\x54\x24"
"\x2C\x57\x68\x4F\x5F\x6F\x21\x8B\xDC\x57"
"\x53\x53\x57\xFF\xD0\x68\x65\x73\x73\x01"
"\x8B\xDF\x88\x5C\x24\x03\x68\x50\x72\x6F"
"\x63\x68\x45\x78\x69\x74\x54\xFF\x74\x24"
"\x40\xFF\x54\x24\x40\x57\xFF\xD0";

int main(void) {
unsigned int length = sizeof(buf);
//分配 shellcode 长度的内存空间
void * addr = VirtualAlloc(0, length, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
//复制 shellcode 到分配好的内存空间
RtlMoveMemory(addr, buf, length);
//设置内存空间保护模式为可执行、可读
BOOL retval = VirtualProtect(addr, length, PAGE_EXECUTE_READ, 0);
if ( retval != 0 )
{
//以线程的方式运行内存空间中的二进制机器码
HANDLE threadHandle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)addr, 0, 0, 0);
WaitForSingleObject(threadHandle, -1);
}
return 0;
}
```

打开 Windows 操作系统的中 x64 Native Tools Command Prompt for VS 2022 命令提示符终端后，使用 cl.exe 应用程序编译 PEdata.cpp 文件，命令如下：

```
cl.exe /nologo /Ox /MT /W0 /GS- /DNDebug /TcPEdata.cpp /link /OUT:PEdata.exe /SUBSYSTEM:
CONSOLE /MACHINE:x64
```

如果成功编译源代码文件，则会在当前工作路径下生成 PEdata.exe 可执行程序，如图 5-9 所示。

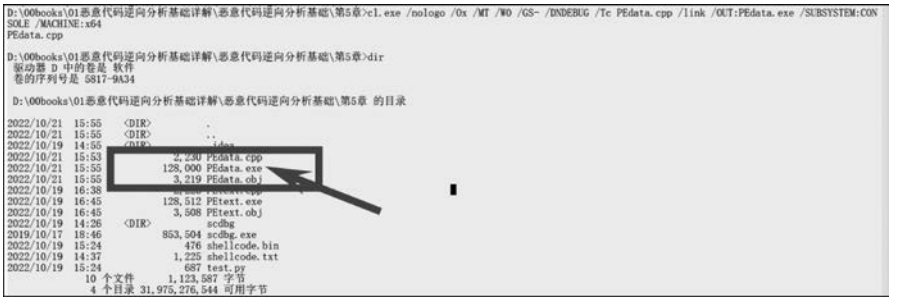

图 5-9　cl.exe 编译 PEtext.cpp 源代码文件

双击运行 PEdata.exe 程序，此时会弹出提示对话框，如图 5-10 所示。

图 5-10　运行 PEdata.exe 可执行程序

程序执行后，按 Enter 键会弹出提示对话框。

5.4　嵌入 PE .rsrc 节区的 shellcode

PE 文件结构的.rsrc 节区存放着程序的资源，如图标、菜单等。因为程序加载.rsrc 节区的数据时不会判断资源是否安全合法，所以恶意代码也可以使用.rsrc 节区存储 shellcode 二进制代码。

5.4.1　Windows 程序资源文件介绍

资源是二进制数据，可添加到 Windows 可执行文件中。资源可以是标准资源，也可以是自定义资源。标准资源涵盖图标、光标、菜单、对话框、位图、增强的图元文件、字体、快捷键表、消息表条目、字符串表条目或版本信息等，而自定义资源包含特定应用程序所需的任何数据。

如果在源代码中调用资源文件，则必须使用文本编译器编辑资源定义文件(.rc)，编译生成后缀名为.res 的二进制文件，最终通过链接器将 res 文件添加到可执行程序。应用程序保存资源数据的流程如图 5-11 所示。

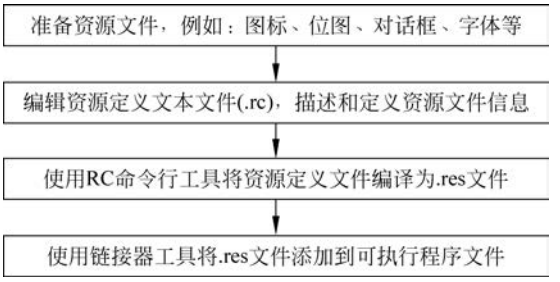

图 5-11　应用程序保存资源数据流程

资源定义脚本文件是后缀名为.rc的文本文件,文件内容支持的编码类型有单字节、多字节、Unicode等。RC命令行工具使用资源定义脚本,根据脚本内容检索资源文件,生成.res文件。可执行文件将读取.res文件,并保存到.rsrc节区。

5.4.2 查找与加载.rsrc节区相关函数介绍

可执行文件的资源数据保存在.rsrc节区,通过查找将资源数据加载到内存空间,从而引用资源数据。

虽然恶意程序的shellcode二进制代码保存在.rsrc节区,但是恶意程序必须调用Win32 API函数FindResourceA查找指定类型或名称的资源位置。这个函数定义在winbase.h头文件中,代码如下:

```
HRSRC FindResourceA(
  [in, optional] HMODULE hModule,
  [in]           LPCSTR lpName,
  [in]           LPCSTR lpType
);
```

参数hModule用于设定资源数据的查找位置,如果设置为NULL,则表示从当前进程中查找资源数据。

参数lpName用于设定目标资源的名称,根据名称会在指定位置查找对应资源数据。这个参数的值可以设定为MAKEINTRESOURCE(ID)的形式。

参数lpType用于设定目标资源的类型。这个参数的值可以设定为RT_RCDATA,表示类型为应用程序定义的原始资源数据。

如果成功执行FindResourceA函数,则会返回特定资源块的句柄。应用程序使用句柄可以引用资源数据。

虽然通过调用FindResourceA函数可以获取资源句柄,但是资源数据并没有加载到内存空间,因此恶意程序必须调用LoadResource函数将资源数据加载到内存空间。这个函数定义在libloaderapi.h头文件中,代码如下:

```
HGLOBAL LoadResource(
  [in, optional] HMODULE hModule,
  [in]         HRSRC     hResInfo
);
```

参数hModule用于设定保存资源数据的模块。如果将参数设置为NULL,则表明应用程序会从创建进程的模块中加载资源数据。

参数HRSRC用于设定资源句柄,设置为FindResourceA函数的返回句柄。

如果成功执行LoadResource函数,则会返回一个资源数据句柄,否则返回NULL。使用LoadResource函数返回的资源句柄,调用LockResource函数提取资源数据。这个函数定义在libloaderapi.h头文件中,代码如下:

```
LPVOID LockResource(
  [in] HGLOBAL hResData      //设定资源数据句柄
);
```

如果成功执行 LockResource 函数，则会返回资源数据的第 1 字节的内存地址，否则返回 NULL。

5.4.3　实现嵌入.rsrc 节区 shellcode

首先，使用 Metasploit Framework 渗透测试框架的 msfconsole 命令行接口生成 shellcode 二进制代码，命令如下：

```
use payload/Windows/messagebox
set EXITFUNC thread
generate -f raw -o msg.bin
```

如果成功执行生成 shellcode 二进制代码的命令，则会在当前工作目录生成 msg.bin 文件，如图 5-12 所示。

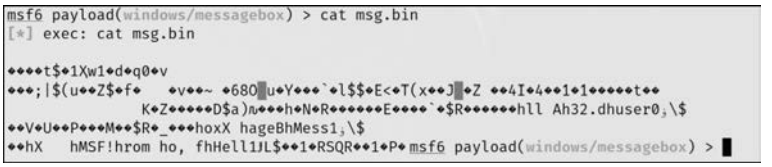

图 5-12　msfconsole 生成原始二进制格式的 shellcode

从结果可以看出，使用 cat 命令无法查看 msg.bin，但是 Hxd 编辑器可以正常查看 msg.bin 的文件内容，如图 5-13 所示。

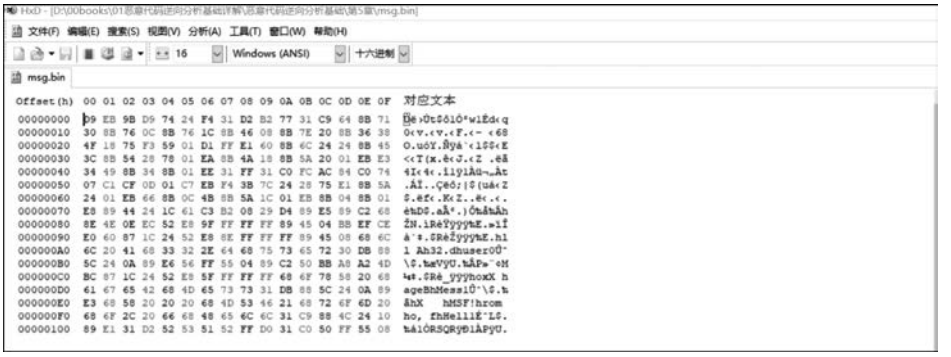

图 5-13　Hxd 查看 msg.bin 文件内容

下一步，在 x64 Natve Tools Command Prompt for VS 2022 命令终端中使用 rc.exe 应用程序生成资源文件 resources.res，命令如下：

```
rc resources.rc
```

资源定义文件 resources.rc 用于规定资源数据，代码如下：

```
#include "resources.h"          //引入头文件
MY_ICON RCDATA msg.bin          //设定 MY_ICON 保存 RCDATA 类型的 msg.bin 数据
```

头文件 resources.h 定义常量 MY_ICON，代码如下：

```
#define MY_ICON 100        //MY_ICON 的值可以设置为任意值
```

如果使用 rc 应用程序能够成功执行 resources.rc 文件，则会在当前目录生成 resources.res 文件，如图 5-14 所示。

```
 D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章\.rsrc_shellcode 的目录

2022/10/26  22:39    <DIR>          .
2022/10/26  22:39    <DIR>          ..
2022/10/26  22:10               272 msg.bin
2021/07/28  16:51                21 resources.h
2022/10/26  22:31                50 resources.rc
2022/10/26  22:39               336 resources.res
               4 个文件            679 字节
               2 个目录 31,930,089,472 可用字节

D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章\.rsrc_shellcode>
```

图 5-14　rc 成功执行 resources.rc 资源定义文件

resources.res 资源文件必须转换为 resources.o 格式文件，这样才能被 cl.exe 识别，因此可以使用 cvtres 工具将 resources.res 转换为 resources.o 文件，代码如下：

```
cvtres /MACHINE:x64 /OUT:resources.o resources.res
```

如果 cvtres 工具执行成功，则会在当前工作目录下生成 resources.o 文件，如图 5-15 所示。

```
 D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章\.rsrc_shellcode 的目录

2022/10/26  22:44    <DIR>          .
2022/10/26  22:44    <DIR>          ..
2022/10/26  22:10               272 msg.bin
2021/07/28  16:51                21 resources.h
2022/10/26  22:44             1,264 resources.o
2022/10/26  22:31                50 resources.rc
2022/10/26  22:39               336 resources.res
               5 个文件          1,943 字节
               2 个目录 31,929,483,264 可用字节

D:\00books\01恶意代码逆向分析基础详解\恶意代码逆向分析基础\第5章\.rsrc_shellcode>
```

图 5-15　cvtres 工具成功将 resources.res 转换为 resources.o

最后，编辑 PErsrc.cpp 源代码文件，实现执行.rsrc 节区的 shellcode 二进制代码的功能，代码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "resources.h"
```

```
int main(void) {

    void * alloc_mem;
    BOOL retval;
    HANDLE threadHandle;
     DWORD oldprotect = 0;
    HGLOBAL resHandle = NULL;
    HRSRC res;

    unsigned char * shellcodePayload;
    unsigned int lengthOfshellcodePayload;

    //从.rsrc 节区中查找并加载 shellcode
    res = FindResource(NULL, MAKEINTRESOURCE(MY_ICON), RT_RCDATA);
    resHandle = LoadResource(NULL, res);
    shellcodePayload = (char *) LockResource(resHandle);
    lengthOfshellcodePayload = SizeofResource(NULL, res);

    //申请内容空间
    alloc_mem = VirtualAlloc(0, lengthOfshellcodePayload, MEM_COMMIT | MEM_RESERVE, PAGE_
READWRITE);


    //将 shellcode 复制到分配的内容空间
    RtlMoveMemory(alloc_mem, shellcodePayload, lengthOfshellcodePayload);

    //将内存空间设置为可执行状态
     retval = VirtualProtect(alloc_mem, lengthOfshellcodePayload, PAGE_EXECUTE_READ,
&oldprotect);

    printf("\nPress Enter to Create Thread!\n");
    getchar();

    //如果成功设置可执行状态,则启动新线程执行 shellcode
    if ( retval != 0 ) {
            threadHandle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) alloc_mem, 0, 0, 0);
            WaitForSingleObject(threadHandle, -1);
    }

    return 0;
}
```

使用 cl.exe 编译链接 PErsrc.cpp 源代码,代码如下:

```
cl.exe /nologo /Ox /MT /W0 /GS- /DNDebug /TcPErsrc.cpp /link /OUT:PErsrc.exe /SUBSYSTEM:
CONSOLE /MACHINE:x64 resources.o
```

如果 cl.exe 成功编译链接 PErsrc.cpp,则会在当前工作目录生成 PErsrc.exe 可执行程

序，如图 5-16 所示。

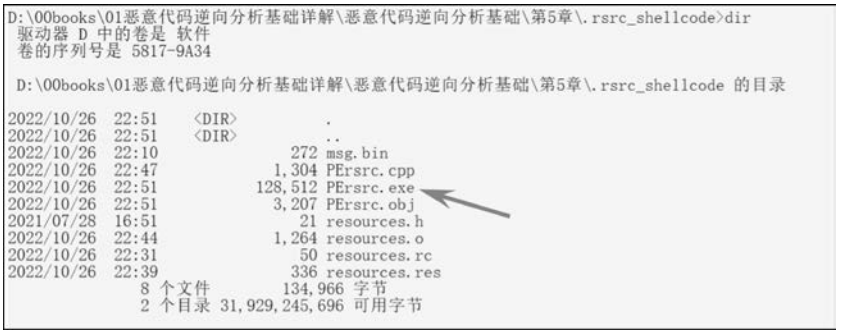

图 5-16　cl.exe 成功编译链接 PErsrc.cpp 源代码文件

在控制台终端执行 PErsrc.exe，弹出提示对话框，如图 5-17 所示。

图 5-17　成功执行 PErsrc.exe 可执行文件 rsrc 节区 shellcode

无论将 shellcode 保存到 PE 文件中的任何节区中，杀毒软件都很容易识别没有经过编码和加密的 shellcode 二进制代码，从而查杀对应可执行文件。恶意代码的分析工作更多集中在提取、解码、解密 shellcode 二进制代码。