编号：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 实验 | 一 | 二 | 三 | 四 | 五 | 六 | 七 | 八 | 总评 | 教师签名 |
| 成绩 |  |  |  |  |  |  |  |  |  |  |

武汉大学国家网络安全学院

课程实验(设计)报告

课程名称： 计算机病毒

实验名称： PE病毒实验

专业(班) ： 网络空间安全

学 号 ： 2020302181165

姓 名 ： 张逸凡

学 号 ： 2020302181265

姓 名 ： 姚栋宇

学 号 ： 2020302181054

姓 名 ： 张培妍

任课教师 ： 陈泽茂

2023 年 4 月 30 日

目 录

[1. 实验目的 - 1 -](#_Toc133770525)

[2. 实验内容 - 1 -](#_Toc133770526)

[3. 实验关键过程、数据及其分析 - 1 -](#_Toc133770527)

[4. 实验思考和建议 - 13 -](#_Toc133770528)

[5. 实验总结 - 13 -](#_Toc133770529)

实验二 实验环境搭建

# 1. 实验目的

加深理解PE文件格式，掌握基本PE病毒的编写技术。

# 2. 实验内容

编写一个PE文件传染程序infect.exe，功能要求如下：

1. infect.exe运行后，向同目录下的某个Windows可执行程序（下称**T程序**，建议找一个免安装的绿色程序，以方便测试。），植入“病毒载荷”代码。
2. infect.exe不能重复传染**T程序**。
3. **T程序**被植入“病毒载荷”后，一旦执行，具备如下行为：
   1. 在其同目录下创建一个新文件，该文件名为本组组长的学号。
   2. 在其同目录下查找是否有PE格式的.exe文件，如果有，则传染之。这些被**T程序**传染的.exe文件，运行后具有与**T程序**相同的行为。

# 3. 实验关键过程、数据及其分析

**3.1 infect功能说明**

我们组编写了PE文件传染程序infect-final.exe，可以感染同目录下的Test-x86.exe程序（我们选取的T程序），实现实验要求的两个行为：**创建新文件**和**传染同目录下其它PE格式的.exe文件**，被传染后的文件运行后具有与T程序相同的行为和传染能力。

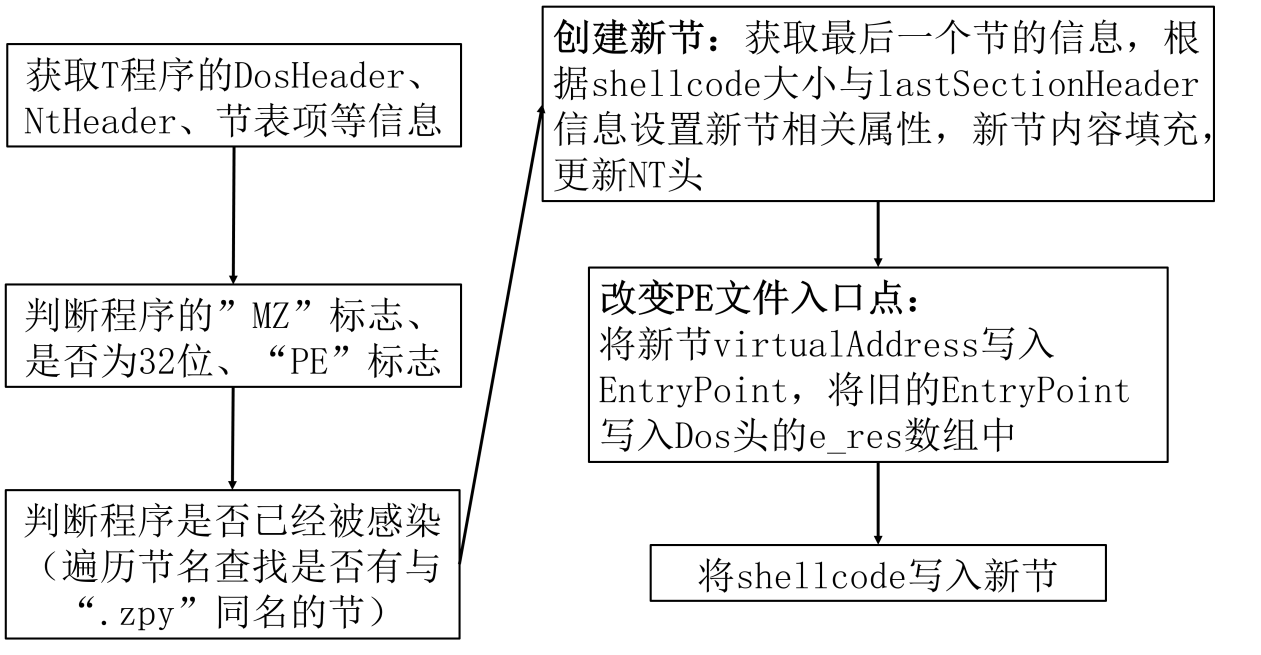
**3.2 Infect程序传染功能的设计与实现**

我们的代码包括inject.cpp和shellcode.cpp两部分，前者通过将shellcode写入在T程序中新增的节实现感染，其中用到的shellcode是由shellcode.cpp生成的。

接下来首先介绍inject.cpp的编写逻辑，然后依次介绍实现行为一和行为二的shellcode.cpp的设计、实现与提取。

**1. Inject.cpp编写逻辑**

Inject.cpp实现的流程图如下图所示：



文件设置了部分全局变量如下：

IMAGE\_DOS\_HEADER dosHeader;

IMAGE\_NT\_HEADERS ntHeader;

IMAGE\_SECTION\_HEADER newSectionHeader; // 新增节的节表项

IMAGE\_SECTION\_HEADER lastSectionHeader; // 旧的最后一个节表项

int numSections = 0;

函数调用如下所示，与上述流程图相同：

int main() {

char\* pFile = (char\*)"Test-x86.exe";

if (getHeader(pFile)) {

printf("Failed getting Header.\n");

return 1;

}

if (isPE(pFile) || isInfected(pFile))

return 1;

else printf("This is a 32bit PE file.\n");

addSection(pFile);

changeEntry(pFile);

paddingShellcode(pFile);

return 0;

}

接下来分别对各函数进行介绍（**函数具体代码及相关注释请在inject.cpp中查看**）：

**getHeader(const char\* file)**，获取文件DosHeader、NtHeader、节表项数等信息。

首先打开文件（所有打开、读写文件均有相应的错误处理）->读取Dos头->移动文件指针到Nt头位置->读取Nt头，获取当前节表项数->关闭文件并返回。

**isPE(const char\* file)，**判断文件的“MZ”标志、32bit标志、“PE”标志。

首先打开文件->判断dosHeader.e\_magic是否为0x5A4D(即MZ)->判断ntHeader.OptionalHeader.Magic是否为0x10b(即32bit文件)->判断ntHeader.Signature是否为0x4550(即PE)->关闭文件并返回。

**isInfected(const char\* file)，**判断文件是否已经被感染。

首先打开文件->文件指针依次移动到各节开头处，判断Section.Name与.zpy是否相同，相同则表明文件已经被感染->遍历结束，关闭文件并返回。

**addSection(const char\* file)，**新增病毒载荷节。

首先打开文件->读取最后一个节的大小、位置等信息->设置新节名为.zpy，同时根据最后一个节的大小和病毒载荷大小，设置新节的大小、地址等属性，注意对齐操作->将新节项信息写入->文件指针移动到新节位置，将新节内容填充为0->更新NT头信息，节数加一，同时更新SizeOfImage->关闭文件并返回

**changeEntry(const char\* File)，**改变程序入口点。

首先打开文件->获取旧的EntryPoint->将oldEntryPoint存在Dos头的e\_res数组中，其中地址高16位存在e\_res[1]，低16位存在e\_res[0]中->更新Dos头->设置新的EntryPoint为新增节的virtualAddress->更新NT头->关闭文件并返回。

**paddingShellcode(const char\* file)，**将病毒载荷写入新节中。

首先打开文件->文件指针移动到新节，将shellcode依次写入->关闭文件并返回。

**2. 行为一：同目录下创建新文件**

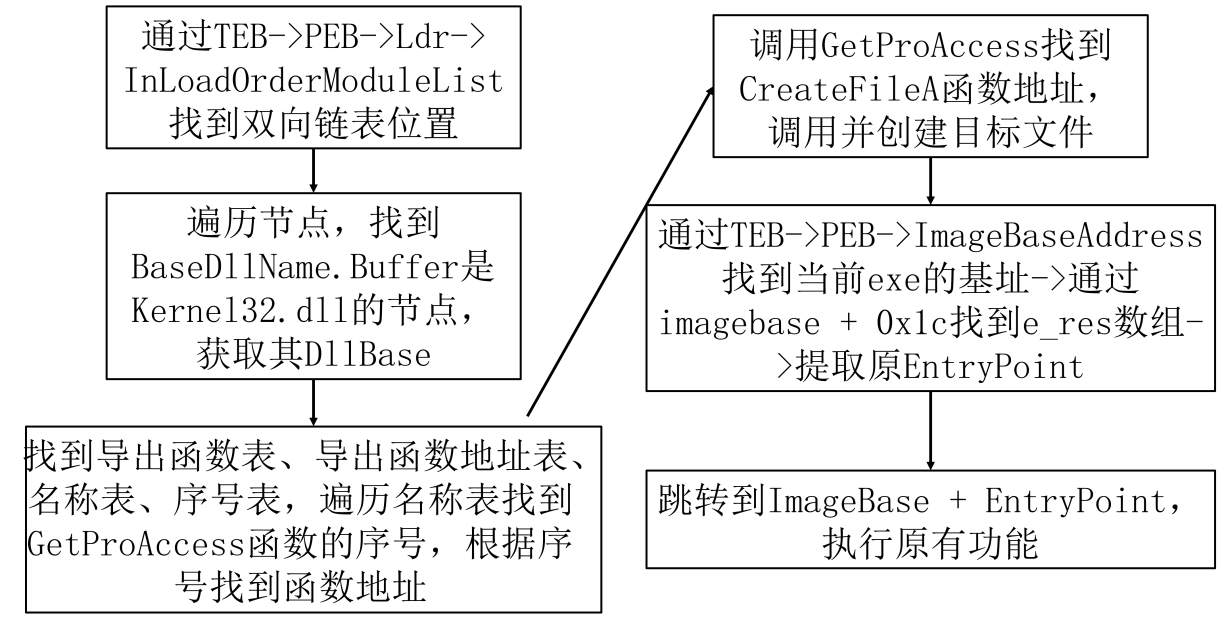
**1) 行为逻辑分析**

基础任务要求在同目录下创建文件，由于此时并不知道CreateFile函数地址，所以需要先找到Kernel32.dll地址、再找到GetProAccess函数地址，利用GetProAccess函数找到CreateFile函数地址，实现对应功能。

在实现所有功能后，还需要寻找当前实际的ImageBase与原先EntryPoint的值，将二者相加后获取原先程序入口点的绝对地址进行跳转，使被感染的T程序仍能执行原有的功能。本实验中，我们选取的T程序执行的功能是打印“This is in x86.”语句。

**2) 设计思路介绍**

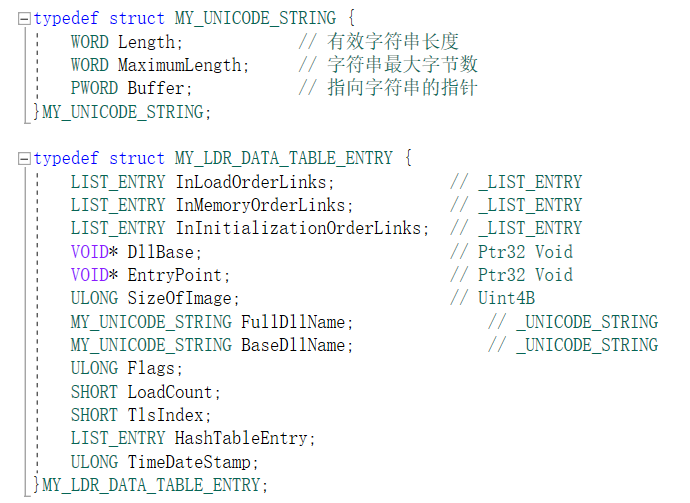
基础版shellcode的代码逻辑如下流程图所示，也可以理解为设计思路：



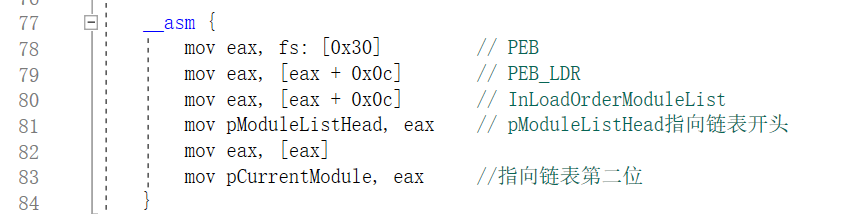
**3) 设计具体实现介绍**

接下来对代码的具体实现进行讲解（也可以直接查看shellcode.cpp，注释也非常详细）。

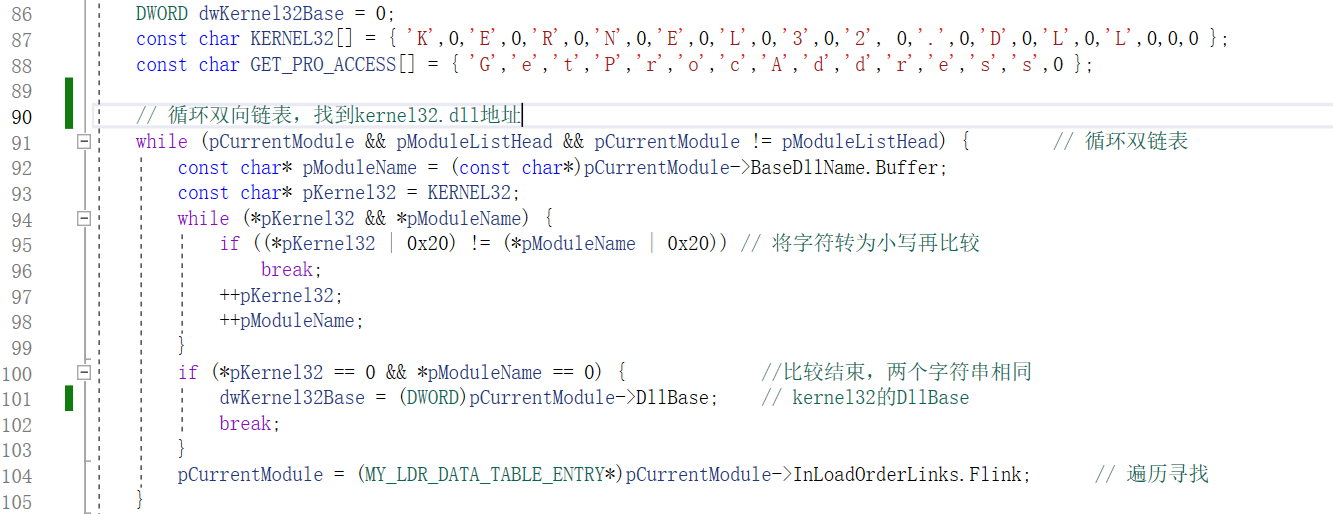
由于Winnt.h中对部分变量的定义不完整，因此需要实现进行完善的定义，如：



通过汇编指令找到双向链表开头位置。



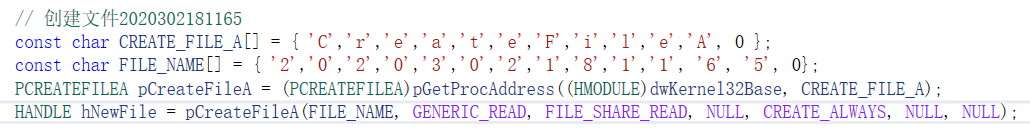
然后遍历链表，通过指针操作比对名字为“kernel32.dll”的节点，获取其DllBase地址作为dwKernel32Base。



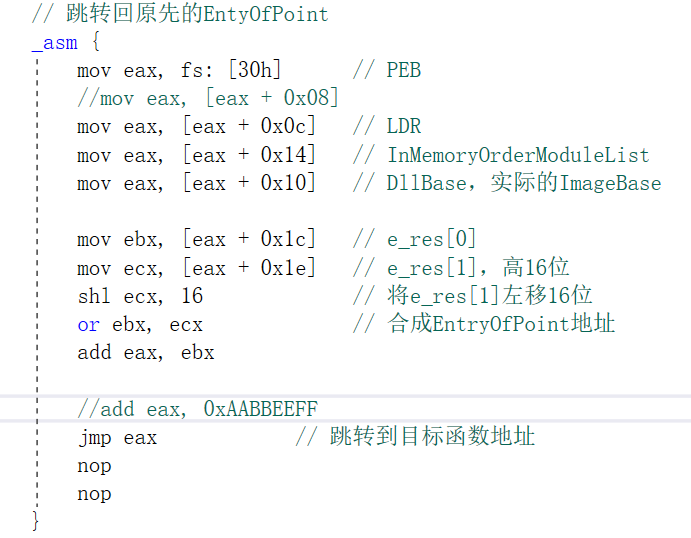
获取导出函数表、函数地址表、函数名称表、函数序号表；遍历函数名称表，通过指针操作对比名称找到“GetProAccess”函数在函数名称表中的序号A，通过A和函数序号表找到该函数的实际序号B，通过B和函数地址表找到函数的地址。



利用GetProAccess函数创建CreateFileA函数，并进行创建文件操作。

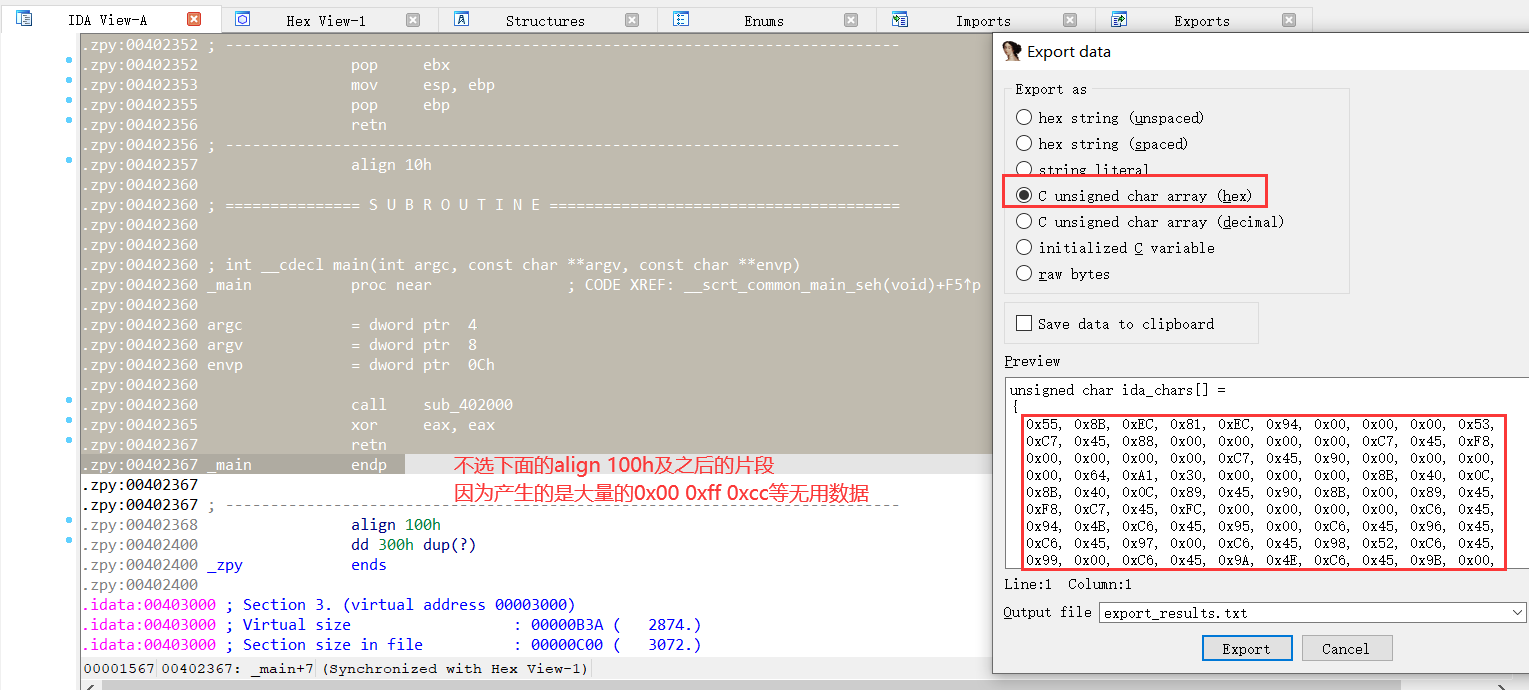


通过PEB->LDR->InMemoryOrderModuleList->DllBase找到实际的ImageBase，再通过ImageBase + 0x1c找到Dos头的e\_res数组，获取在里面存储的原先的EntryPoint，然后将基址与偏移相加获取入口点的绝对地址，跳转回去。



**4) 病毒载荷提取**

利用ida.exe打开上述shellcode.cpp生成的shellcode.exe，找到.zpy节部分，选取后导出数据，如下图：



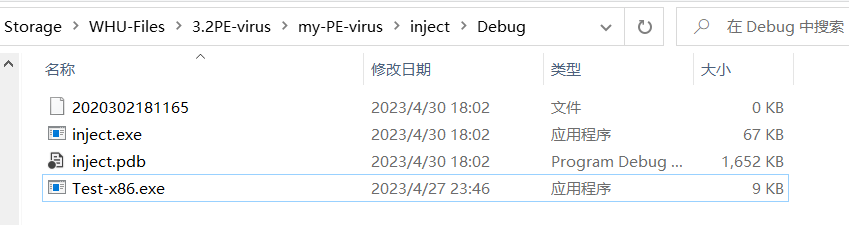
还可以查找左侧的shellcode(void)函数，双击后点击上方的Hex view-1，选中这个函数的所有十六进制部分进行导出，效果相同。

将导出的shellcode放入inkect.cpp中的const char shellcode[]数组中即可。

注：shellcode.cpp项目一定要**禁用安全检查**，否则会产生很多杂乱的数据，导致感染失败。

**5)实验结果**

运行inject.exe后，运行Test-x86.exe，可以看到同目录下创建了名为“2020302181165”的文件。



**3. 行为二：查找其他.exe文件并传染**

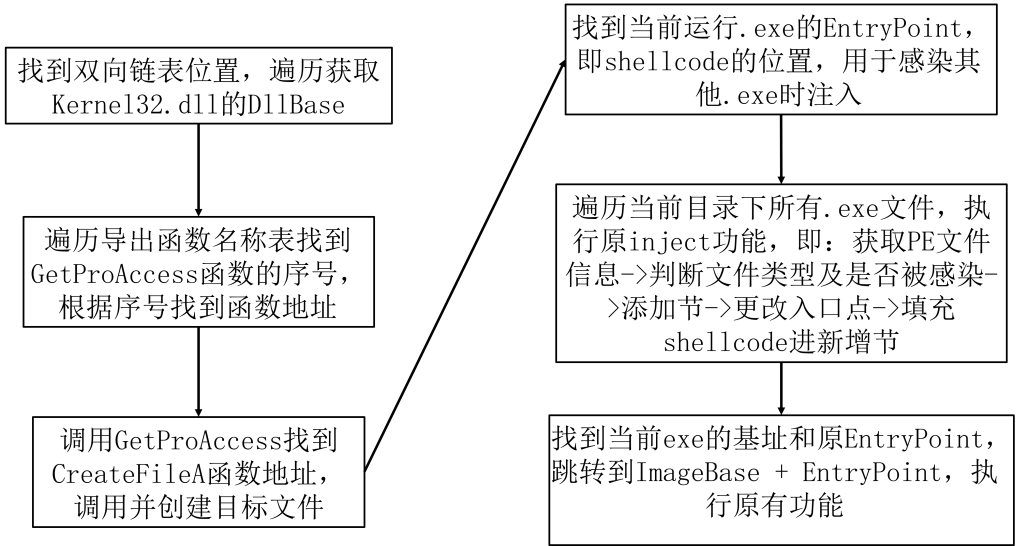
**1). 行为逻辑分析**

由于行为二的要求是被植入病毒载荷的T程序继续传染其他.exe程序，其他.exe程序被传染后运行具有与T程序相同的行为，因此首先考虑将编写的inject文件代码部分植入shellcode文件中，使shellcode具有传染能力，能够传染其他程序。

而行为二的要求中说明，同目录下的所有PE格式的.exe文件应当均能被T程序传染，所以在参考行为一中编写CreateFileA函数通过GetProcAddress函数找到地址的操作，编写FindFirstFileA和FindNextFileA函数来进行循环调用，直到访问目录下所有PE格式的.exe文件为止。

依照行为二的逻辑，我们将inject.cpp文件中的函数代码按照主函数调用顺序，去除参数并将返回值赋值新变量，植入shellcode.cpp文件中，并采用先查找文件目录下第一个文件，然后采用do-while循环调用查找下一个文件是否为PE格式的.exe文件，是则植入病毒载荷，不是或已经植入病毒载荷的文件跳过。我们设想的所有被传染文件应当有两种行为：创建一个名称为2020302181165的文件，传染同目录下所有PE格式的.exe文件。

进阶版shellcode流程图如下：



其中感染的关键点在于注入的shellcode存放位置是当前执行的、被感染的.exe文件的节，通过ImageBase + EntryPoint定位，并进行复制感染。

**2). 使用win32函数移植替换c库函数**

在尝试直接植入inject代码进入shellcode之后，我们发现：由于shellcode在被当作病毒载荷植入目标程序时是作为16进制码被inject植入的，而16进制码无法调用c库函数，如fread、fwrite、fopen等，在思考解决方案后，我们采用与读取和创建文件相同的方式：手动编写功能与win32库内函数相同的函数，并查找函数的地址调用。

最终，我们通过使用移植替换的方式，将原来功能为向其他程序植入病毒载荷的代码中的c库函数全部移植为win32库函数，并保证它们功能一致。具体功能实现详见下文。

**3).更改shellcode文件中的shellCode变量**

在行为一中，我们并不需要在shellcode中再定义shellCode变量，因为我们只需要将shellcode文件转为16进制码移入inject中，注入进T程序即可。但行为二的要求让我们必须在shellcode文件中设置shellCode变量，保证T程序还有感染其他程序的能力。

在编写初期，我们采用数组的形式，将添加inject代码段的shellcode文件转化为16进制码，然后赋值给shellcode文件内新建的shellCode变量，最后再将赋值后的shellcode文件再次转化为16进制码移入inject中，注入进T程序。

我们发现，这样编程的inject感染了T程序后，T程序并不具备传染的能力，仅仅具备创建学号文件的能力。通过分析得知，当我们使用数组类型shellCode变量时，是无法做到无限嵌套循环的。原理如下图所示：



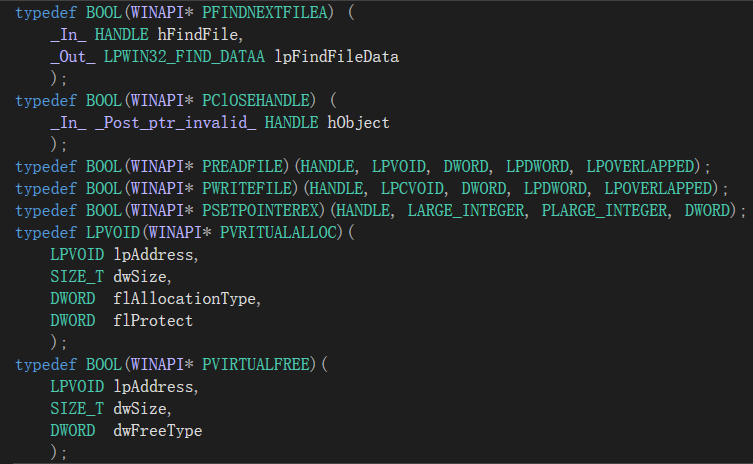
上图已经表明，在使用数组shellCode类型时，由于嵌套不可以无限循环，当第二次传播完成的时候，此时被传染的程序仅能生成一个学号文件。与我们希望的无限传染构想不符。

基于此原理，我们想要更换shellCode变量的类型，将其定义为指针变量。这样做的原理在于：我们提取的shellcode本质是可执行的二进制机器码，字符数组的本质是内存中的二进制数据。而程序的可执行代码一定会出现在内存中，只要我们能找到这些代码的位置，我们就可以直接从内存中读取可执行代码，那么我们将shellcode.cpp文件中的入口点从main函数修改为shellcode函数，此时shellcode函数作为处在shellcode.cpp文件入口点的连续二进制代码，只要知道其起始内存地址和大小，就能够将这段二进制代码看成字符数组，复制传染至新程序内部，这样就能做到避开有限嵌套的麻烦，真正做到无限传染。

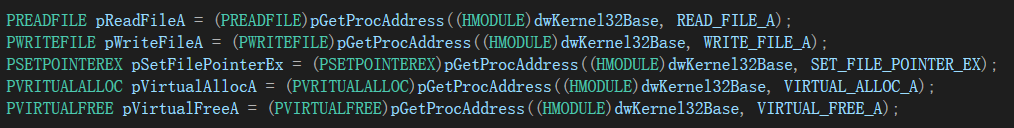
**4). 编写shellcode.cpp，实现上述构想**

（1）库函数定义及替代：

我们首先根据需求，增设了几个新函数的定义如下：



并通过GetProcAddress函数获取win32库函数地址，构建新函数如下：

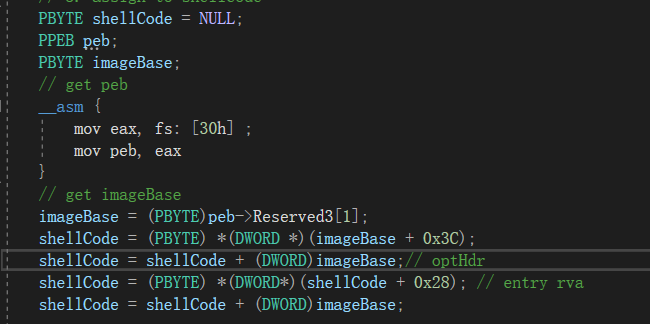


在这里对移植替代做一些简单说明：

* fopen：由pCreateFileA函数进行替代，pCreateFileA函数可以通过设置第五个参数为OPEN\_EXISTING来将功能设定为打开已有文件，函数返回的句柄即可作为后面ReadFileA等函数的输入使用。
* fseek：由pSetFilePointerEx函数代替，通过预先给设定的64位整数变量offset赋值，确定函数指针指向的位置。
* fread：由pReadFileA函数代替，配合pSetFilePointerEx使用，设定指针后将fread的读取大小计算出来当作参数输入即可
* fwrite：由pWriteFileA函数代替，配合pSetFilePointerEx使用，设定指针后将fwrite的读取大小计算出来当作参数输入即可
* malloc：由pVirtualAllocA函数代替，配合pSetFilePointerEx使用，设定指针后将想分配的数据大小当作参数输入即可
* free：由pVirtualFreeA函数代替。
* 另外，我们宏定义了一个sizeInt来代替c库函数中向上取整函数ceil进行操作。

（2）shellCode变量设定及入口点查找位置：

定义PBYTE类型变量shellCode，通过引入winternl.h库来使用PPEB类型变量peb，查询peb类型定义可知，imagebase就在peb变量的Reserved3[1] 变量处，最后通过查询入口点RVA得到入口点VA。



得知了函数入口点并将其存在shellCode中后，我们开始查看函数代码段大小，经过使用PE-bear、IDA、OD工具的反复debug后，确认了将函数大小CODE\_SIZE设置为0x1000是没有问题的。

最后在shellcode函数中，使用pSetFilePointerEx和pWriteFileA将从shellCode地址开始的大小为CODE\_SIZE的数据写入传染目标程序中，即可实现目标行为。

（3）其他修改：

由于判断文件是否为PE文件或是否已经植入病毒载荷的代码段处于do-while循环中，所以判断条件和判断结果都有所更改。

**5). 实验结果：**

将修改后的shellcode.cpp编译出的exe文件转化为16进制码，写入inject.cpp中，inject注入的目标T程序命名为“Test-x86.exe”，目标功能为T程序及T程序传染的程序都能传染同目录下未传染的程序，并创建一个文件名为“2020302181165”的文件。

运行inject.exe，观察目录下变化：

运行前：

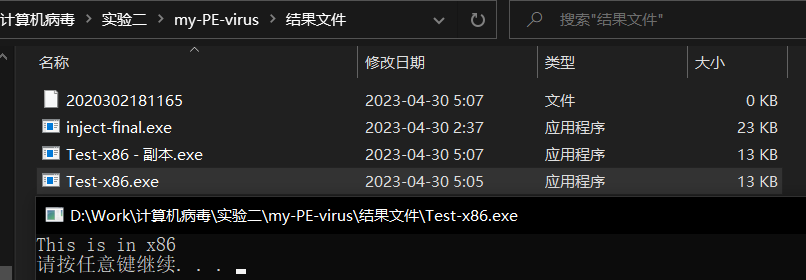


运行后：



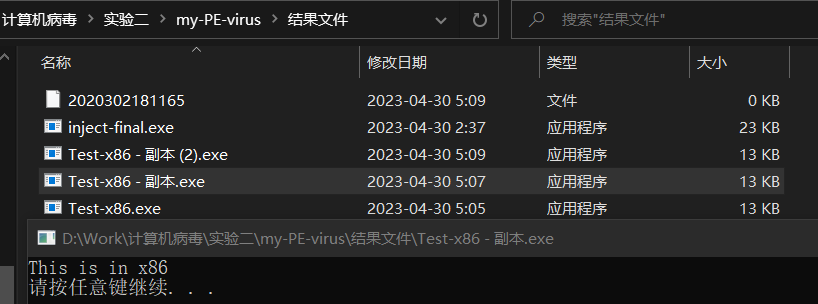
将其他文件复制进目录，运行Test-x86.exe文件：

运行结果如下：



Test-x86.exe文件成功返回本文件功能，观察修改时间，Test-x86-副本.exe文件也已经被传染，同时生成2020302181165文件。

运行Test-x86-副本.exe文件观察传染结果如下：



同样观察到Test-x86-副本(2).exe文件已经被传染，2020302181165文件时间也证明Test-x86-副本(2).exe文件同样拥有创建文件的功能。T程序行为二完成。

# 4. 实验思考和建议

**（1）实验问题**

a)基础版提取shellcode时中间有大段的0xcc、0xff无用字段，后来老师点明是因为没有禁用安全检查，禁用安全检查后恢复正常。

b)在最初开始写进阶版shellcode的时候，我们采取将导出的shellcode作为数组，在shellcode进行调用，这样不但导致最终的shellcode特别长，而且还不能正确运行。

后来我们商讨过后，决定shellcode中感染的思路不按照原先inject里向新节写入数组中存储的病毒载荷走，而是找到当前exe文件的EntryPoint地方（也就是当前文件被植入shellcode的地方），自调用这一块，将其写入其余.exe文件中实现感染。这样既解决了感染过程中传播性变弱的特点，也使导出的shellcode不那么长。

c)在对程序进行不断debug中，shellcode.exe出现了一个非常奇怪的错误：它在给目录下其他文件植入代码后，其他程序无法传染，且无法正常执行原程序。在使用OD单步跳跃反复查看和PE-bear查看对比内存二进制码后，发现shellcode函数代码段在判断文件是否为PE文件或是否已经植入病毒载荷的代码段后面的代码全部复制错乱，查看指令显示此时内存相应区域的指令功能不明。此bug原因不明，但在将sectionAlignment、fileAlignment两个变量从float改为int并将原来引用两者处的int强制转变去除后，shellcode.exe功能恢复正常，其感染的程序也可以正常对其他程序进行感染。

猜测上述bug产生原因是c++语言变量类型强制转变也调用了c++库函数，在转成二进制代码后无法调用导致出错。

d)在打包提交文件的时候，又运行了一遍inject-final.exe，发现运行inject-final.exe后创建了“2020302181165”文件，经过检查后才发现，因为按照运行inject-final.exe->Test-x86.exe->inject-final.exe的顺序，运行第二遍inject-final.exe时，该程序已经被感染，因此会出现上述问题。

**(2)实验思考**

其实在实验中还是有一些可以优化的地方的，比如：

a)我们判断是否被感染是通过节名，诚然“.zpy”节名不容易和文件原有节名冲突，但因为这是一个不常见的节名，被攻击者很容易察觉到该文件被攻击了。

b)我们在shellcode中是通过GetProAccess函数调用所有WINAPI函数的，在一些病毒检测软件中，会检测软件对GetProAccess函数和LoadLibrary函数的调用，所以我们的病毒是有可能被察觉的。同时包括跳回原有EntryPoint的地方，也是很容易被察觉的，因为这是一个不正常的跳转。

同时通过函数名遍历寻找也可以优化成通过定长的hash值去寻找函数地址，这可能是更优雅的方案。

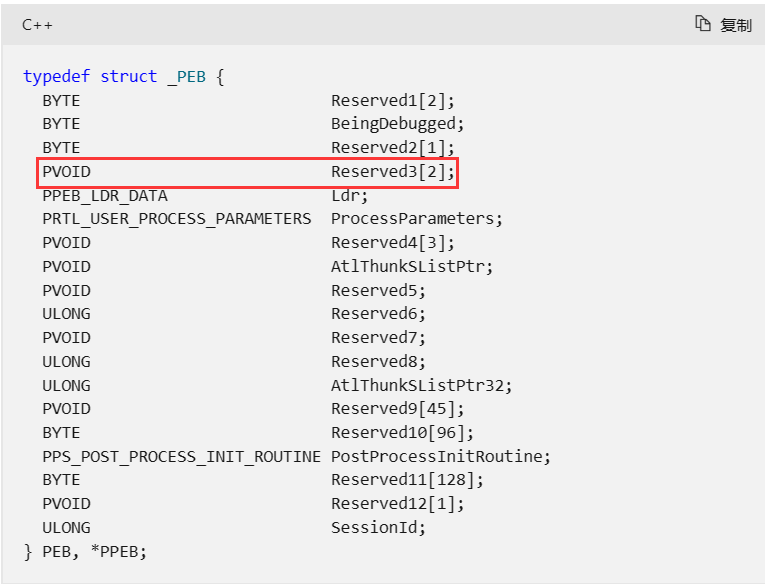
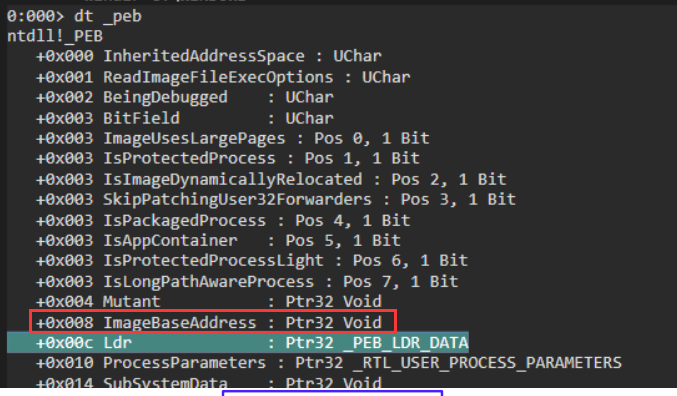
c)病毒的感染是很有趣的过程，有机会的话可以探索一下病毒的清除，应该也会是一个很有趣的过程。

d)想起来在shellcode调试的时候，一直提醒我jmp的地址不是有效的地址，实际在编写shellcode的时候确实这个地址不一定有效，但它并不影响，因为此时还没有将原有的EntryPoint写入调取的位置。

e)关于如何找到程序加载到内存后的实际Imagebase，在实验过程中尝试了以下几种方法，都是可以的：

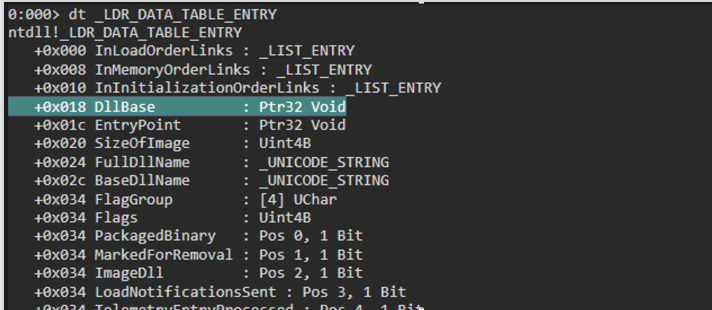
Ⅰ.TEB->PEB->ImageBaseAddress，是实际的ImageBase值

Ⅱ.TEB->PEB->Reserved3[1]，与Ⅰ思路相同，不过由于winternl.h对PEB部分结构有保留，所以Reserved3[1]实际就是ImageBaseAddress



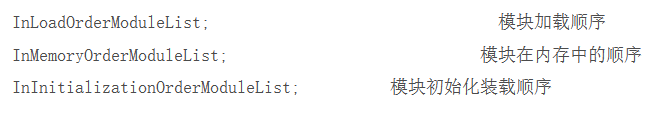
Ⅲ.TEB->PEB->LDR->InMemoryOrderModuleList->DllBase，第一个模块的DllBase即exe模块自身的信息，也就是实际的ImageBase

Ⅳ. 同样通过TEB->PEB->LDR->InMemoryOrderModuleList 找到存放模块的链表节点信息后，第一个节点的InInitializationOrderLinks值也很巧的是ImageBase，这个纯粹是因为意外试验出来的，在跳回时我们用ImMemoryOrderModuleList + 0x10，也获得了实际的ImageBase值



Ⅴ.通过TEB->PEB->LDR->InInitializationOrderLinks找到当前程序模块信息，然后调用DllBase。

后来通过查询资料找到以下说明：



三者都是双向链表，并且指向的第一个模块都是当前exe程序的模块，这就意味着有很多种找到ImageBase的方法，我并没有完整的验证，只验证了一部分，不过这个发现也是让我觉得非常有趣的。

**(3)实验建议**

建议老师在介绍工具时介绍一下ida、od等工具的使用，在debug的过程中觉得动态调试器是非常重要的，因为编写shellcode放入inject后，是不清楚执行到哪条命令出问题了的。

# 5. 实验总结

本次实验进行了对PE文件的感染操作，同时让PE文件也具有了感染能力，这是一次非常棒的动手体验，在动手过程中我们加深了对PE结构和如何进行感染操作的理解，也熟悉了PEbear、ida、od等工具的使用。

实验过程中比较困难的是shellcode的编写，首先是找到函数地址的部分，我们小组经过讨论、翻阅PPT、上网查资料后确定了采用遍历双向链表找Kernel32.dll，再结合导出函数地址表、名称表、序号表来找函数地址，其中序号表上课似乎没提过，是我们通过查阅资料找到的。

此外便是如何跳回原入口点的问题，起初我们想要把跳回的代码写到inject.cpp中，但这样不方便实现进阶版的感染，在和老师讨论之后我们决定将原入口点写到Dos头中的e\_res数组中，解决了上述的问题。

最后令我感受最深的是感染的shellcode存放位置的问题，我们最初采用了嵌套 的方式，即shellcode里有一个数组存放shellcode，但这样感染是逐渐减弱的过程。后来我们还讨论了在感染文件中创建两个除了名称都相同的节，如“.1”和“.2”节，程序新的入口点是跳转到“.1”节，在搜寻到的别的PE文件中创建两个新节，将“.2”节写进去，这样进行多次感染。后来我们又觉得这样太过麻烦，其实定位当前的节，把自己这个节作为shellcode写入别的PE文件是最简单的方式，这种自调用的方式让我大开眼界。

总的来说，这是一次既痛苦又快乐的实验过程，痛苦的debug，激烈的讨论，实现对应功能的时候真的非常的快乐，感谢同组的同学们互相讨论互相debug，感谢老师对于问题的解答！

我们的项目代码已上传至GitHub，待课程结束后，我们会将我们的仓库改为public进行开源。