实验四 Windows 实验 2 指导书

目录

一、实验概述

PE 结构中的导入表与导出表部分是 PE 结构中最为重要的两个部分，涉及到不同 PE 映像之间的协同配合，以及 API 的查找定位和调用相关机制，是恶意代码攻防对抗中的核心要点。

本实验主要围绕 Windows 下 PE 结构中的导入表和导出表部分进行展开，首先通过查找 Dll 导出表中的导出函数完成对 PE 结构导出表相关结构的理解和掌握，然后通过对比硬盘中以及内存中可执行程序导入表中 IAT 表的差异，帮助理解 PE 结构中导入表相关结构，最后通过修改可执行程序中 API 调用相关指令，使得执行流程转到模拟恶意代码位置从而模拟恶意代码对抗中更为隐蔽的执行权限获取方式。

二、实验目的

建立起对于 Windows 下 PE 文件结构的整体印象，能灵活对硬盘上的 PE 文件和内存中的 PE 映像进行修改，从而理解恶意代码对 PE 文件的感染流程以及相应的清除方式。

熟练掌握 PE 文件所涉及的导入表机制及导出表机制，理解恶意代码攻防中常用的 API 定位机制，以及 PE 文件加载阶段对于所需调用的系统 API 的解析和定位过程。

三、实验任务

1）理解 PE 导出表结构，完成对指定导出函数的定位，并在加载到内存中的 PE 文件映像中定位到对应的函数代码。

2）理解 PE 导入表结构，通过对 PE 文件和 PE 内存映像中导入表结构中 IAT 表内存的对比，深刻理解 PE 导入表机制。

3）在理解 PE 导入表和导出表的基础上，以在 PE 文件代码段尾部寄生的方式完成病毒寄生，并通过 Patch API 调用指令的方式获取执行权。

四、实验原理

4-1. PE 结构概述

PE（Portable Executable）文件格式是微软 Win32 环境可移植可执行文件的标准文件格式即链接器生成的可执行文件。需要说明的是，PE 格式是由 COFF 文件格式演变而来的，因此二者在结构方面具有相似性。

PE 文件结构的主要组成部分如图 4-1 所示：

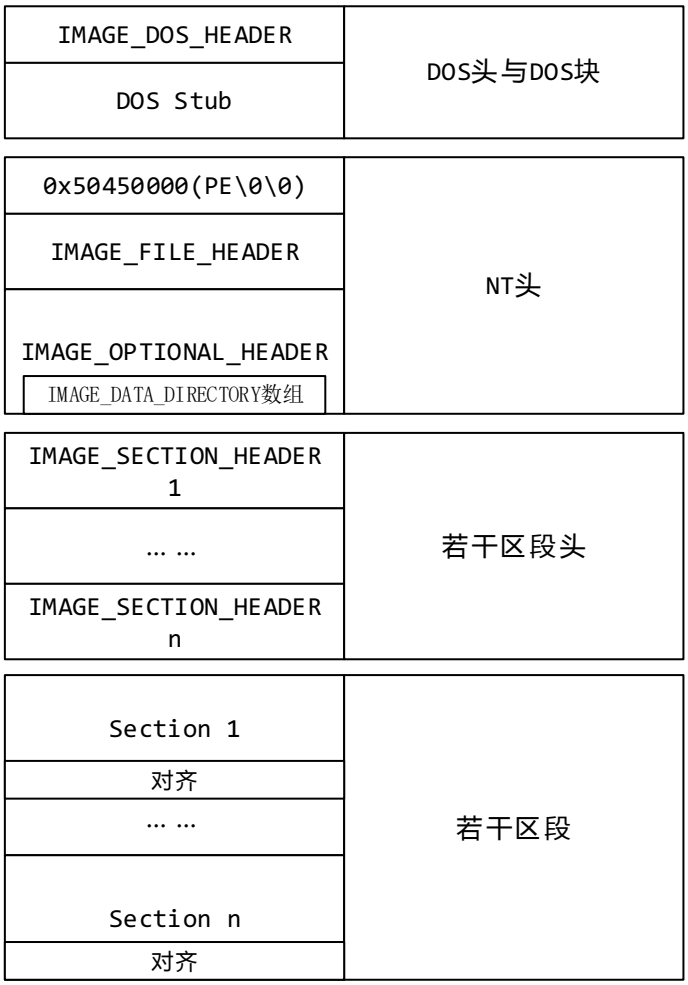

图 4-1 PE 文件的基本结构

4-2. 导入表与导出表的定位

在 NT 头的可选头之后，即为 DataDirectory 字段。由于 PE 结构中的数据是按照属性存入对应的区段中的，这就意味着不同用途但相同属性的数据可以在合并后归入同一个区段中。为此，在可选头中定义了一个被称为数据目录表的数组以指明这些数据的位置。默认的数据目录表结构数组中包含 16 个元素，可以更改可选头中 NumberOfRvaAndSizes 扩充元素个数。DataDirectory 结构数组中的结构成员定义如表 4-1 所示。

表 4-1 IMAGE_DATA_DIRECTORY 结构定义

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;          //起始 RVA
    DWORD   Size;                    //大小
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

可以看出，表 4-1 所示的是一个包含起始 RVA 和大小的结构。

DataDirectory 结构数组的第一个元素指定了导出表的位置和大小；第二个元素指明了导入表的位置和大小。

4-3. 导出表原理

导出表主要提供本 PE 文件导出的符号供其他 PE 文件使用。一个函数或变量既可以按名称导出也可以按序号导出，下面将结合如图 4-2 所示的导出表结构示意图介绍如何用这两种方法导出一个函数或变量。

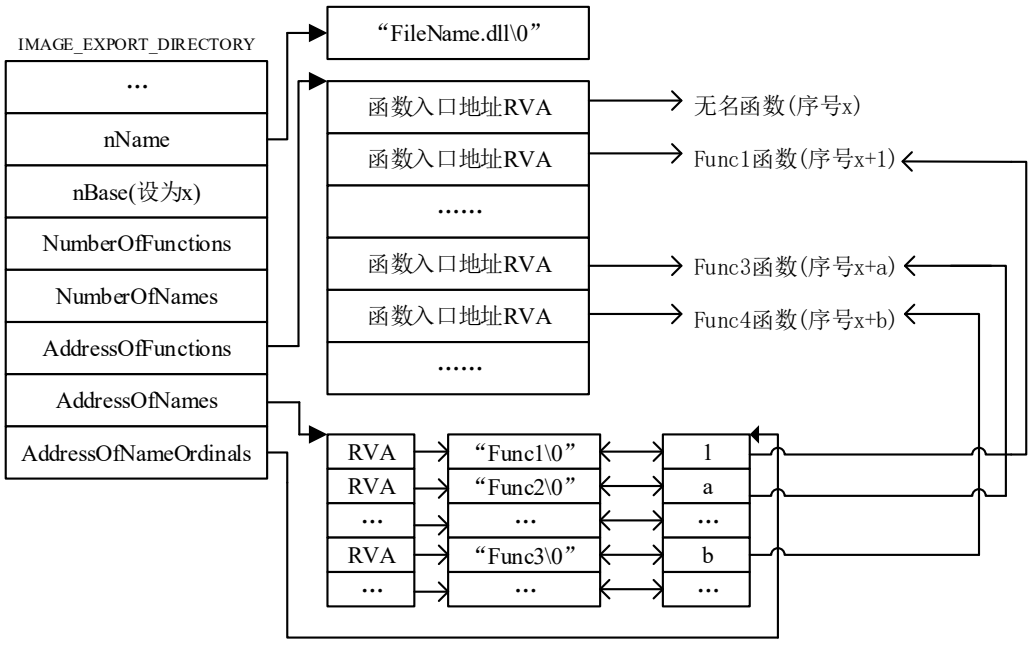

图 4-2　导出表结构示意图

第一种方法是按函数名导出。此方法要求使用该导出函数的 PE 文件知晓本模块导出函数的名称，并在其导入表的 IMAGE_THUNK_DATA 中指明是按函数名称导入（最高位清 0）。为实现此种导出方法，既可以使用 Visual C++中定义的关键字__declspec(dllexport)修饰对应函数，也可以采用传统方法，在模块定义文件中指明要导出的函数名。

第二种方法是按序号导出。此方法要求使用该导出函数的 PE 文件知晓本模块导出函数的序号，并在其导入表的 IMAGE_THUNK_DATA 中指明是按序号导入（最高位置 1）。要按照序号导出，必须采用传统方法，在模块定义文件中指明要导出函数名所对应的序号。

PE 文件的导出表位置和大小由可选头中数据目录表结构数组首个元素中的信息来确定。其中 VirtualAddress 字段指示导出表目录结构的 RVA，其定义如表 4-2 所示。

表 4-2 IMAGE_EXPORT_DIRECTORY 结构定义

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;            //未使用，总是定义为 0
    DWORD   TimeDateStamp;              //文件生成时的时间戳
    WORD    MajorVersion;               //未使用，总是定义为 0
    WORD    MinorVersion;               //未使用，总是定义为 0
    DWORD   Name;                       //模块的真实名称的 RVA
    DWORD   Base;                       //基数，加上序数就是函数地址数组的索引值

    DWORD   NumberOfFunctions;          //导出函数的总数
    DWORD   NumberOfNames               //以名称方式导出的函数的总数
    DWORD   AddressOfFunctions;         //指向导出函数地址的 RVA
    DWORD   AddressOfNames;             //指向导出函数名字的 RVA
    DWORD   AddressOfNameOrdinals;      //指向导出函数序号的 RVA
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

下面依次介绍其中的重要字段。

- Name 字段：RVA 值，指向本模块名称的字符串。
- NumberOfFunctions 字段：本模块总共导出的符号数量。
- AddressOfFunctions 字段：RVA 指，指向一个元素个数等于 NumberOfFunctions 字段的 DWORD 数组，该数组被称为符号地址表（简称地址表），其中每个元素都是一个导出符号所在地址的 RVA。
- NumberOfNames 字段：按符号名导出的函数和变量的总数。
- AddressOfNames 和 AddressOfNameOrdinals 字段：二者共同确定按名称导出函数的序号。AddressOfNames 字段存储了指向一个 DWORD 类型数组的 RVA，该数组被称为导出符号名表（简称符号表），其中元素个数等于 NumberOfFunctions 字段的值，每项都存储一个导出函数名称的 RVA。而 AddressOfNameOrdinals 字段则存储了指向一个 WORD 类型数组的 RVA，该数组是一张导出符号名索引到导出函数入口的映射表（简称映射表）。首先在导出符号名表中匹配到对应函数名，再检索映射表中相应位置的值，再以该值作为地址表的索引，既可得到目标函数的 RVA。
- Base 字段：标识地址表中第一个导出符号的序号值，其后的符号序号值依次递增。

从函数名称查找入口地址的步骤是：先由包含导出表的模块的基地址算出导出表的地址，从 NumberOfNames 字段得到已命名函数的总数作为循环次数，紧接着从 AddressOfNames 字段指向的符号名表中循环判定其与待查找函数名是否相同，直到完成查找。若成功匹配，则由此时符号名数组的下标，从

AddressOfNamesOrdinals 对应位置得到其在地址表中的索引，最后从 AddressOf Functions 字段指向的导出符号入口地址表中取得该索引的 RVA 并与模块的基地址想加就得到了该符号的地址。

4-4. 导出函数的定位过程

1. 以实验工具中的 user32.dll 为例，首先使用 PEView 打开映像文件，根据 4-1 和 4-2 的介绍，定位到该文件的导出表。

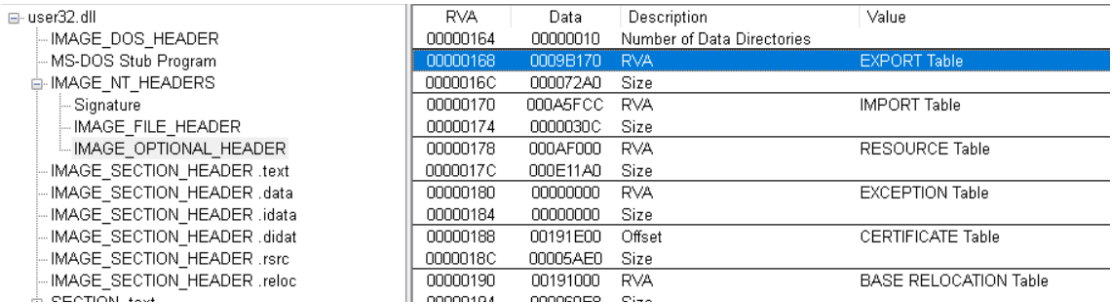

图 4-3　定位导出表结构

2. 使用 C32ASM 或 UltraEdit 等 16 进制编辑器，打开对应映像文件，定位到导出表位置，验证导出表结构体信息。
3. 定位到导出表中的**函数名指针表**，**按学号末位数**找到对应函数名称的 RVA，再根据该 RVA 定位到对应的函数名。
4. 定位到导出表中的**函数序号表**，查找目标函数在**函数地址表**中的索引值。
5. 定位到**函数地址表**，按找到的索引值获取函数的入口 RVA。
6. 验证找到的函数地址，使用 x64dbg 或 ollydbg 载入目标 Dll，查看对应 RVA 位置的代码（x64dbg 注意先 F9 运行到该 Dll 入口，然后通过“符号”选项卡选中对应 Dll，再在左侧符号窗口找到对应符号名（函数名），双击即可定位。

4-5. 导入表原理

导入表是动态链接机制的重要组成部分。所谓导入函数，即该函数被某模块调用，但其并不在该模块中定义，仅将诸如函数名、其所在的 DLL 名等信保存在该模块的导入表中。PE 文件的导入表的位置和大小可由可选头中数据目录表结构数组第二个元素中的信息来确定。其中的 VirtualAddress 字段即指示了导入表描述符结构数组的 RVA，该结构数组的每一个成员都代表一个 DLL，表明本 PE 文件需要用到这些 DLL 中的某些导出函数，该结构数组的最后一个成员

是一个空描述符。导入表描述符结构定义如表 4-3 所示。

表 4-3 IMAGE_IMPORT_DESCRIPTOR 结构定义

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR
{
    union
    {
        DWORD    Characteristics;
        DWORD    OriginalFirstThunk;
    } DUMMYUNIONNAME;
    DWORD    TimeDateStamp;
    DWORD    ForwarderChain;
    DWORD    Name;
    DWORD    FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED
*PIMAGE_IMPORT_DESCRIPTOR;
```

下面对其中的重要字段进行讲解。

Name 字段：存放保存 DLL 名字的以结尾的字符串的 RVA。

OriginalFirstThunk 和 FirstThunk 字段：存放 IMAGE_THUNK_DATA 结构数组的 RVA。该结构数组定义如表 4-4 所示。

表 4-4　IMAGE_THUNK_DATA 结构定义

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        DWORD ForwarderString;          // PBYTE
        DWORD Function;                  // PDWORD
        DWORD Ordinal;
        DWORD AddressOfData;             // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA32;
```

这个看似复杂的结构其实只是一个 DWORD 类型的联合。在 PE 文件中，若该双字的最高位为 1，表示此处存储的是该函数在对应 DLL 中的序号，此时称为按序号导出；若该双字的最高位为 0，则意味着此处为对应函数名称的 RVA，此时称为按函数名导入。

其中 OriginalFirstThunk 对应的 IMAGE_THUNK_DATA 结构数组被称为 IAT（Import Address Table 导入地址表），而 FirstThunk 对应的结构数组则被称为 INT（Import Name Table 导入名称表）。当 PE 文件被存储在磁盘中时，INT 表和 IAT 表的内容完全一样，而一旦 PE 被执行，载入内存后 IAT 表所对应的

结构数组就会被 PE 加载器更改为对应函数的入口地址。

需要指出的是，只要 PE 文件使用导入表机制，IAT 表就必须存在，而 INT 表则是可选的。但如果 PE 文件中使用了绑定导入技术，即为加快 PE 文件加载速度而将 IAT 表设置为对应 API 函数在目标主机上的虚拟地址时，则 INT 表必须存在，详细信息请参考本文文末翻译部分的第 3 部分相关内容。虽然大多数链接器在链接过程中会使用 IAT 与 INT 构成双表桥式结构，但也有少部分链接器只生成 IAT 表。

图 4-4 VC++链接器生成的导入表结构

Microsoft 公司的 VC++链接器生成的导入表结构如图 4-4 所示。通常导入表所在区段的名称为“.idata”，该区段属性可读可写不可执行。导入表所在区段的开始部分存放的是若干 DLL 的 IAT 表，紧随其后的是对应 DLL 的导入描述符表，然后是与 IAT 表相同的一份 INT 表，最后是上述各表中用到的字符串表。

4-6. 导入表实验

1. 观察文件的导入表情况，先按 4-1、4-2 的讲解，在 PE 文件的可选头的数据目录中找到导入表的 RVA。根据该 RVA 定位到导入表所在的区段（节）。

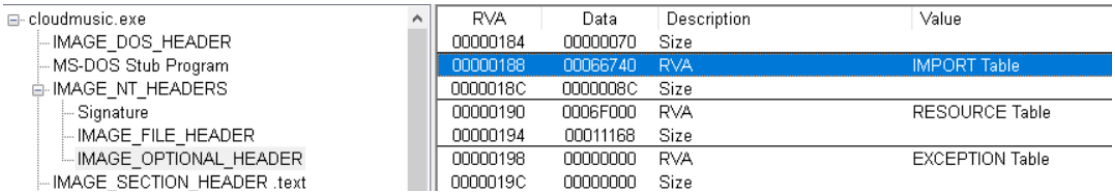

图 4-5 定位导入表结构

2. 按 4-5 的讲解，解析对应的导入表结构，获取任意一个（可以是第一个，第二个，第三个…）导入项对应的 INT 和 IAT 表的 RVA 地址。
3. 在对应导入项的 INT 和 IAT 表中，定位到学号末位数项（0 对应 1,1 对应 2，…，以此类推）位置，分别查看其 RVA 对应位置的内容。
4. 使用 x64dbg/OD 将目标 exe 载入内存，再查看步骤 3 中对应 RVA 位置的内存内容，并描述对应变化。

4-7. 使用 VS 生成内联汇编

在第三个实验中，我们使用一段汇编指令模拟恶意代码的行为，可使用 Visual Studio 获得其机器码。

需要注意的是，由于 x64 版本的 VC++编译器（即 cl.exe）不支持内联汇编的语法格式，因此应以 x86 调试（Debug）模式编译对应的内联汇编代码！

在 VS 中，输入如下代码，注意将 mov ebx, 0x63030023 中的**源操作数 0x63030023 改为学号的后八位**。

表 4-5 IMAGE_THUNK_DATA 结构定义

```
int main()
{
    _asm
    {
        call code_start
        nop
        nop
        nop
        nop
    code_start:
        pop eax
        mov ebx, 0x63030023            ; 这里改为你学号后八位
        mov [eax], ebx
        jmp code_start
```

```
    }
}
```

完成之后，调试运行程序，即可利用 VS 的反汇编功能配合内存查看功能查看到上述代码对应的机器码。

4-8. Patch API 调用指令获取执行时机

1. 使用 VS 编译生成 4-7 指令对应的代码，获得其机器码，**注意学号后八位的修改**！
2. 将上述机器码寄生到目标可执行文件.text 区段（即代码区段），在对应区段头添加可写属性。
3. 调整 PE 头及区段头相关属性（参照实验三），使得寄生的模拟病毒代码可被顺利加载到内存中。
4. 查找该可执行程序中一定会被调用的 API（**鼓励选用非课件列出的 API，为可选加分项，如进行了此步骤，请在实验报告中列出你是如何找到该 API 的**），并在目标可执行文件中找到需要 Patch 的指令
5. 将需要 Patch 的指令修改为跳转到模拟寄生代码处
6. 在寄生代码尾部通过计算，修改回跳 JMP 指令到原 IAT 表位置（间接跳转）。
7. 在调试器中验证上述过程是否顺利完成（寄生代码被执行且对应内存位置数据发生变化，原可执行程序正常运行，不发生崩溃）。