实验三 Windows实验1

一、实验概述

Windows 系统是目前被广泛使用的操作系统，也是众多恶意程序主要感染和破坏的重点操作系统。本实验主要介绍 Windows 下汇编指令级调试器 x64dbg 的使用，以及 Windows 下可执行程序所遵循的 PE 文件结构，并在此基础上完成病毒入口点感染实验。

二、实验目的

熟练掌握 Windows 下 x64dbg 汇编指令级调试器的基本操作和使用，熟悉 Windows 下 PE 可执行程序的基本结构，理解病毒感染 PE 文件的原理。同时为实验四做准备。

三、实验任务

1）修改入口点地址：首先解析 PE 文件头部，然后找到存储入口点地址的地方，随后修改入口点地址；

2）手动增加代码寄生到 PE 文件尾区段后的空洞区域，并保持文件大小不变，首先解析 PE 文件头部，找到段表，从中找到最后一个段的位置。定位到最后一个段的末尾，加入少量代码。修改文件入口点并指向添加的代码。

3）寄生到非代码区段的末尾，修改入口点指向该位置，并跳回原入口点。

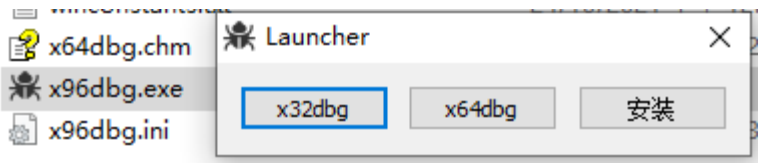
4)模拟寄生代码长度大于末尾区段空洞区域长度的情况，增加末尾区段的长度。

四、实验原理

4-1. x64dbg 调试器的使用

x64dbg 是一款免费开源的 x86/x64 汇编指令级动态调试器，软件原生支持中文界面和插件，其界面及操作方法与 OllyDbg 调试工具类似，支持类似 C 的表达式解析器、全功能的 DLL 和 EXE 文件调试、IDA 般的侧边栏与跳跃箭头、动态识别模块和串、快反汇编、可调试的脚本语言自动化等多项实用分析功能。

X64dbg 中包含有针对 x86 和 x64 应用程序的两个不同的调试器，在调试不同类型程序的使用需要使用对应的调试器。可以通过默认的 x96dbg 启动对应的调试器：



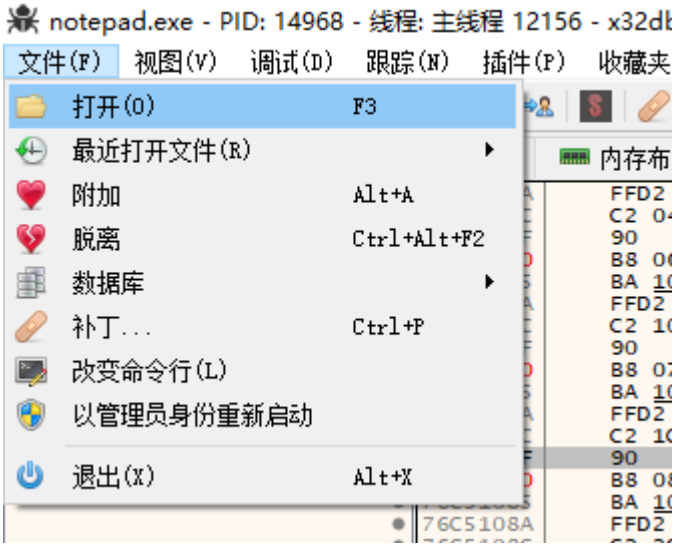
本次实验所使用到的 x64dbg 基本功能如下：

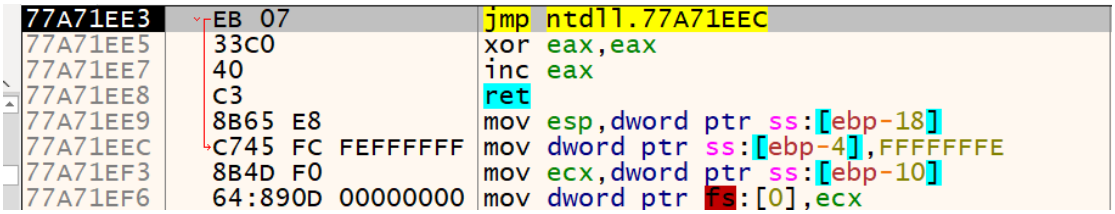
■ 启动一个程序调试

- Attach 到一个已经运行的程序调试
- 单步，step into and step over
- 断点
- 继续运行
- 查看内存
- 修改内存
- 查看寄存器
- 修改寄存器
- 代码窗体跳到指定地址
- 修改指令
- 查看一个进程加载的 dll
- 查看 dll 中有哪些函数

启动被调试进程

启动 x64dbg 后，选择“文件-打开”，选择被调试程序，该程序将以调试状态被启动，在 x64dbg 中，将显示程序停在第一条指令。

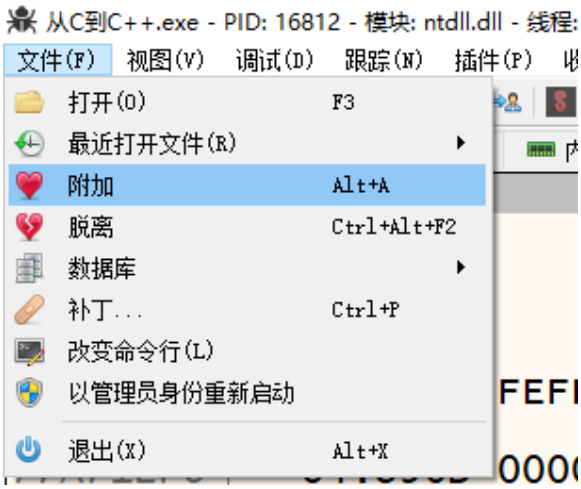




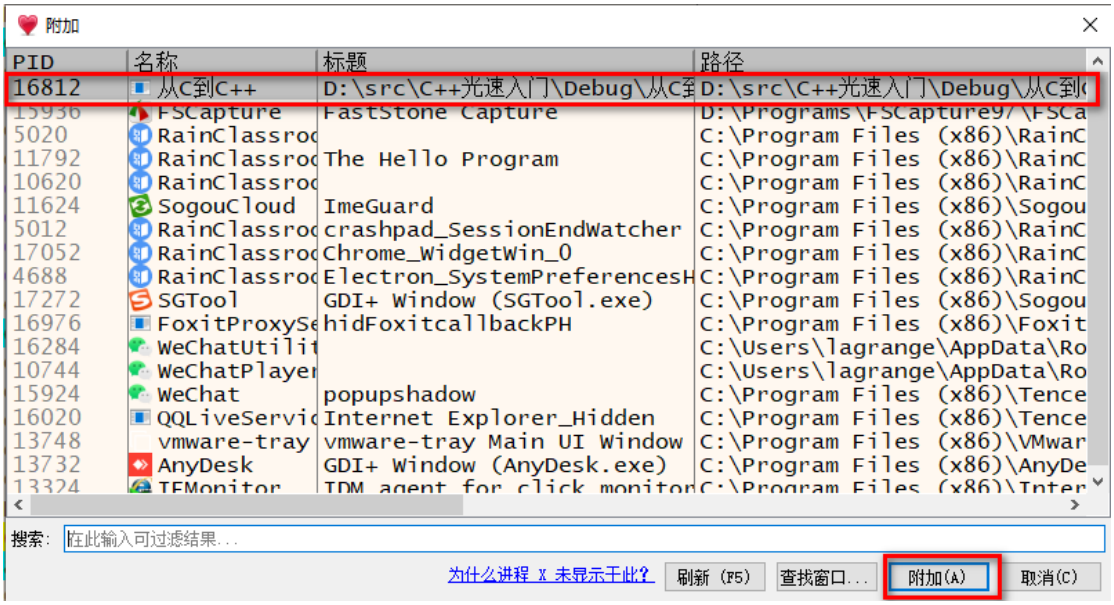
附加到被调试进程

针对已经启动的目标进程，或目标进程存在“强壳”保护的情况下，也可以使用 x64dbg 在中途“附加”到该进程上，进行调试。

启动 x64dbg 后，文件-附加，在进程列表选中目标进程，点击 attach 按钮，出现当前运行的进程列表（注意目标进程类型，从而使用 x64dbg 的 32 位和 64 位版本！）



在进程列表中，选中待调试进程，点击下方附加即可。



此时点击“名称”列头，可按字母排序。点击附加按钮后，调试器就正式与目标进程建立起连接，出现调试界面。此时，被调试进程暂停了，我们可以做各种操作。

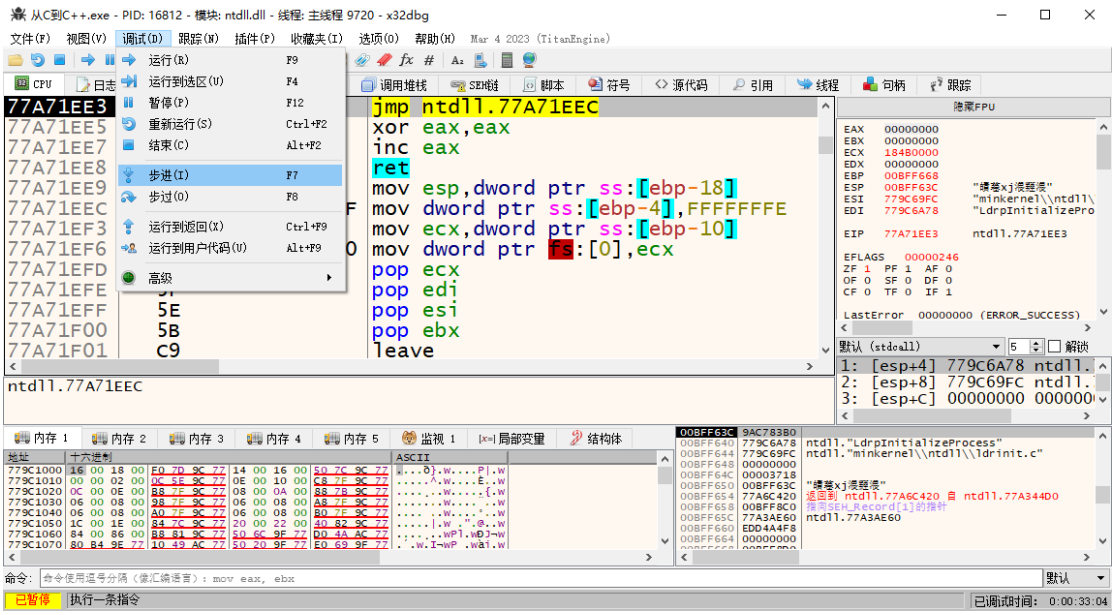
基本调试操作之单步

使用调试器打开或附加到调试进程后，就可以在汇编指令一级开始对目标程序进行调试了。

最为基本的调试方式是单步运行。可以根据情况选择以下两种单步运行的方式：

单步步入：遇到 call 指令会跟踪进入函数，快捷键 F7

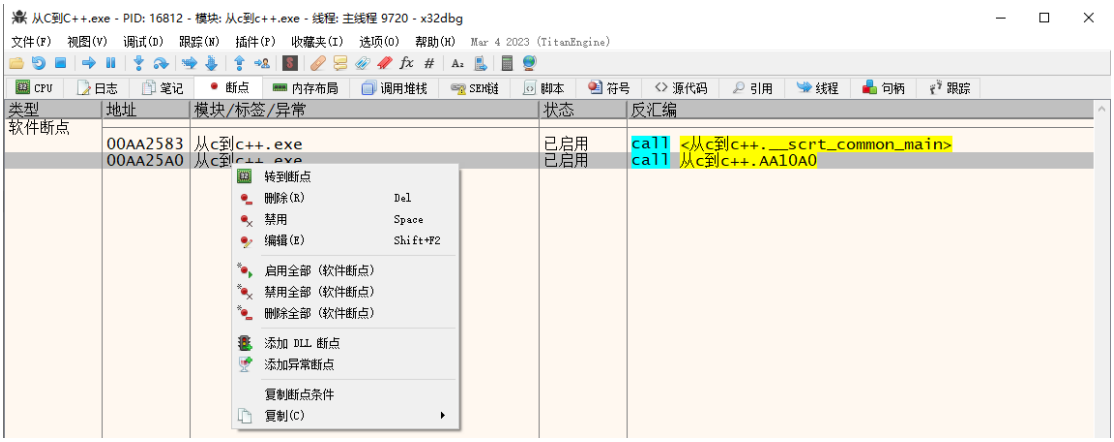
单步步过：遇到 call 指令会运行完整个函数，不进入函数，快捷键 F8



基本调试操作之断点运行

软件断点：使用快捷键 F2 可在对应地址处下软件断点（int 3 断点），此时，对应行的地址变为红色。

可以在断点选项卡中看到本程序地址空间中所有下过的断点：



也可以选中断点后，右键控制其启用或禁用。完成断点设置后，可以继续运行程序。

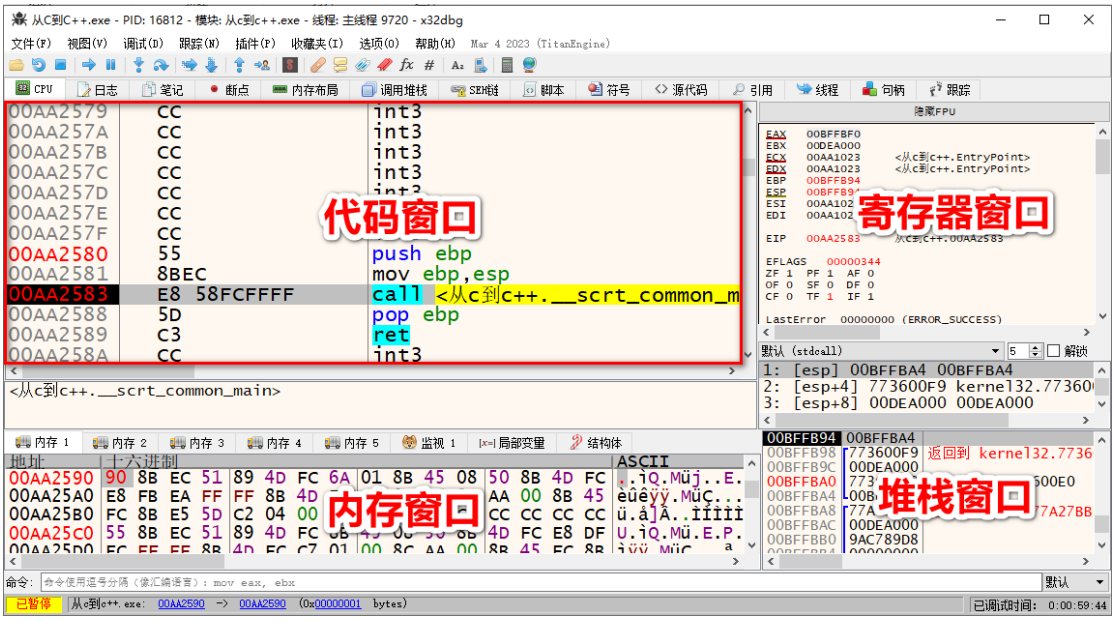
继续运行：按 F9 可继续运行程序

继续运行程序后，当软件断点命中后，调试器会中断在目标行，此时地址会高亮显示：

```
00AA2580   55              push ebp                          exe_main.cpp:15
00AA2581   8BEC            mov ebp,esp
00AA2583   E8 58FCFFFF    call <从c到c++.__scrt_common_main>  exe_main.cpp:16
00AA2588   5D              pop ebp                           exe_main.cpp:17
00AA2589   C3              ret
00AA258A   CC              int3
00AA258B   CC              int3
00AA258C   CC              int3
```

基本调试操作之程序状态查看与修改

X64dbg 的整体布局如下图所示：

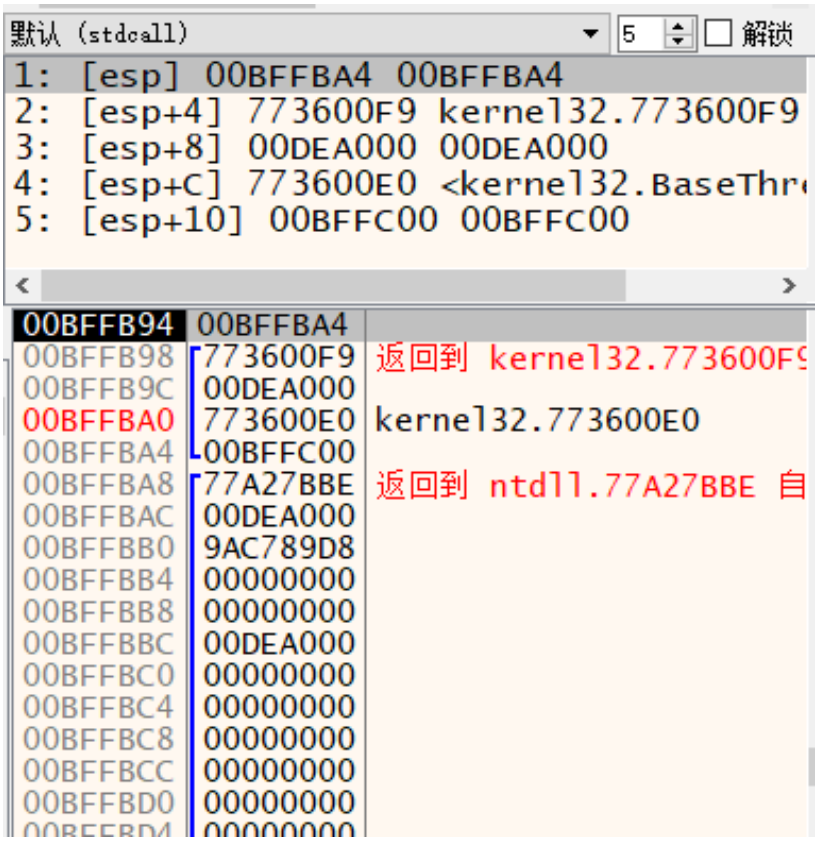


断点命中后，程序暂停在对应地址，此时可通过右侧的寄存器窗口查看寄存器的值：

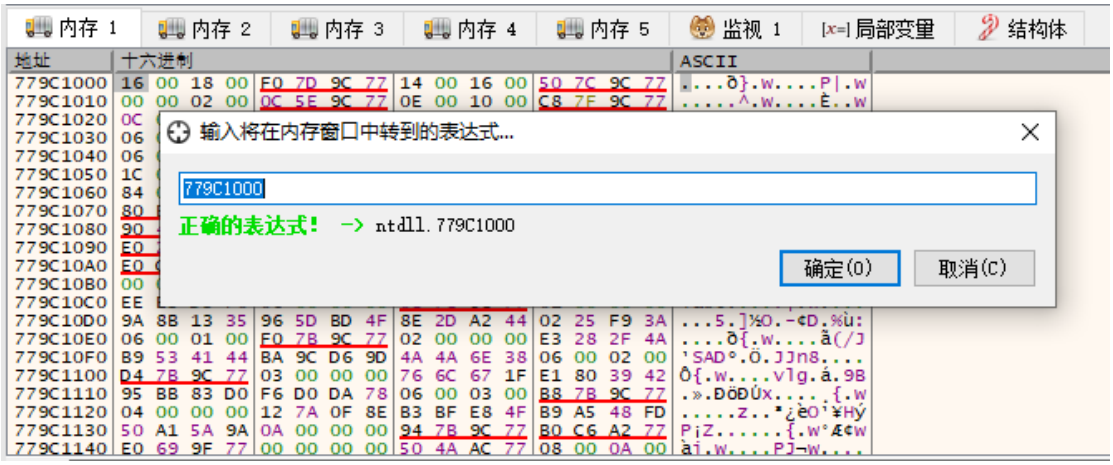
```
EAX   00BFFBF0
EBX   00DEA000
ECX   00AA1023     <从c到c++.EntryPoint>
EDX   00AA1023     <从c到c++.EntryPoint>
EBP   00BFFB94
ESP   00BFFB94
ESI   00AA1023     <从c到c++.EntryPoint>
EDI   00AA1023     <从c到c++.EntryPoint>

EIP   00AA2583     从c到c++.00AA2583

EFLAGS   00000344
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C0000034 (STATUS_OBJECT_NAME_NOT_FOUND)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B
```

如果需要对寄存器的值进行修改，可鼠标在目标寄存器值位置双击，并在弹出的修改窗口中填入想要修改的值，然后确定即可。

也可通过右下方的堆栈窗口查看当前栈的情况：

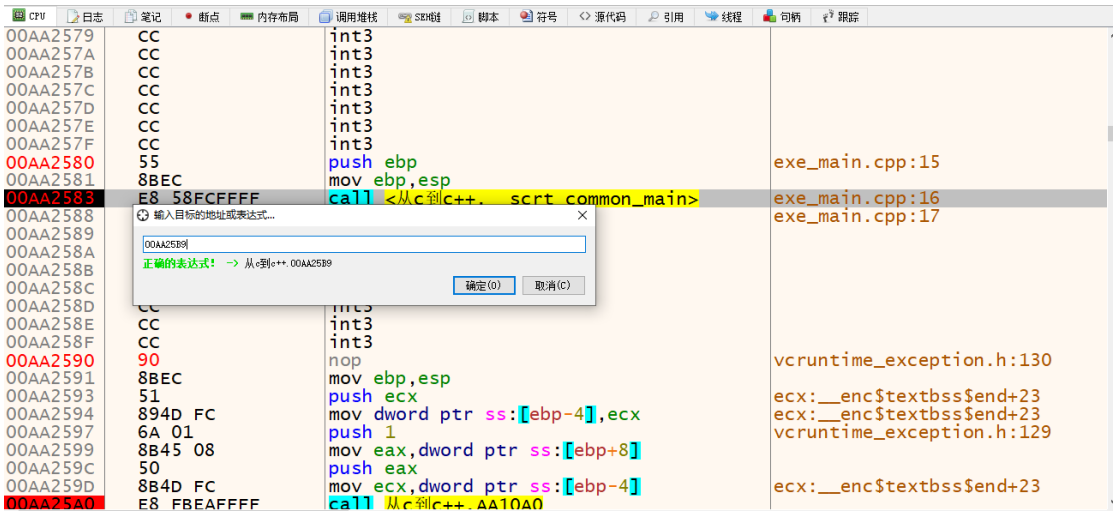


在左下方的内存窗口中，则可以观察感兴趣的内存区域的值情况。鼠标选中内存窗口后，按键盘快捷键 Ctrl+G 可以弹出地址输入框，在其中输入想要查看的内存地址的位置，按确定按钮后，内存窗口将转到对应位置，此时可以对内存中的值进行观察。



如果需要对内存中的值进行修改，选中对应位置后，按空格会跳出如图的窗体，输入要输入的值，完成修改：

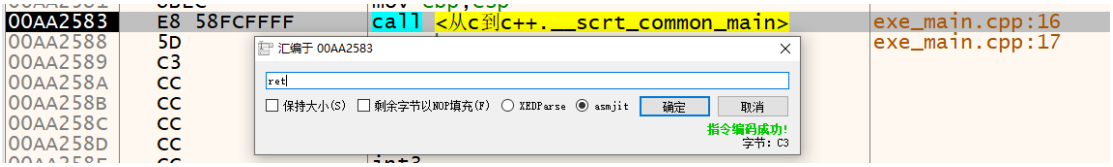
如果需要在指令窗口查看对应地址的反汇编代码（并不是要执行对应位置的代码，只是查看其汇编指令！），可定位到指令窗口后，按 Ctrl+G，输入对应的地址后回车即可。



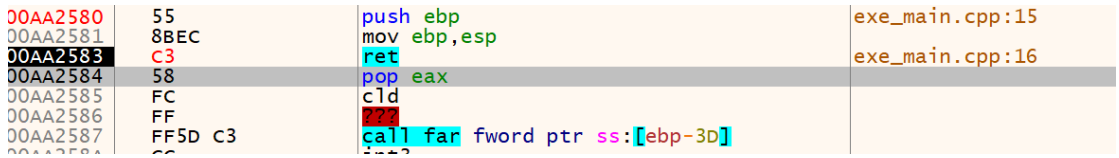
修改指令

- 方法 1：知道机器码，在内存窗体的相关内存地址中相关处输入新的机器码值即可（参见上述修改内存的操作）

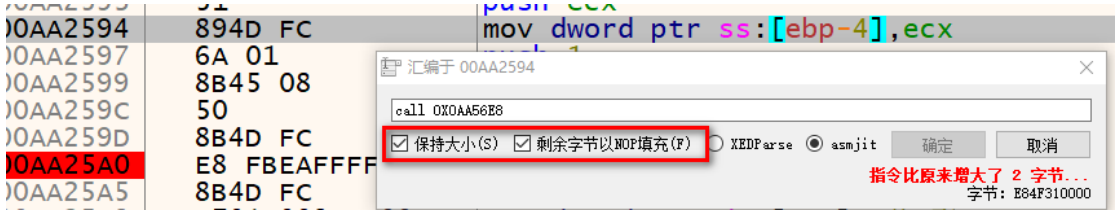
■ 方法 2：选择代码窗体中某条指令，按空格键，将弹出修改窗。下图选择了地址 0x00AA2583，按空格键，在弹出的汇编窗口中输入新的汇编指令，即可自动填入机器码



需要注意的是，如果修改后的指令与原有指令的长度不一致，那么修改之后可能造成原有指令的部分字节成为新的指令，或占用到后续指令的内存空间，导致后续指令无法正确被解析。因此，修改完指令之后，需要仔细核对检查。

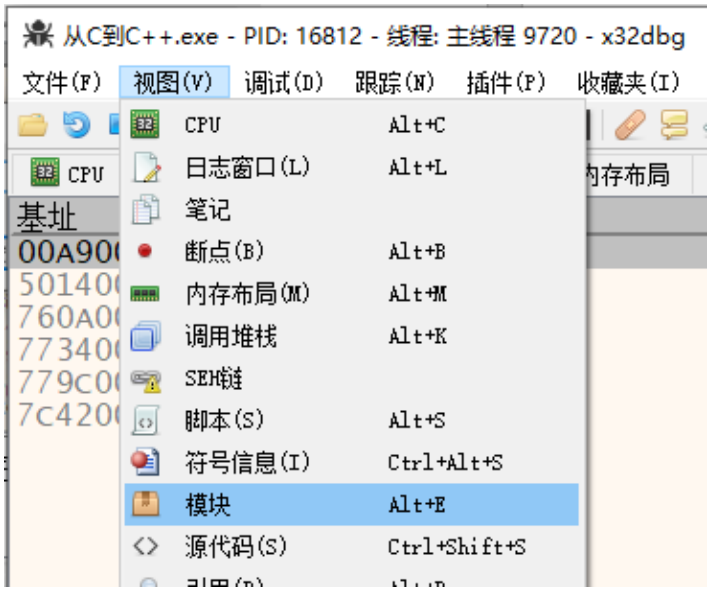


在上述案例中，将 call 指令改为 ret 指令后，call 指令后续的 58 FC FF FF 就变成了我们不需要的机器指令，此时，建议将其填入 NOP 进行无害化处理。为了保险起见，可以勾选“剩余字节以 NOP 填充”和“保持大小”。

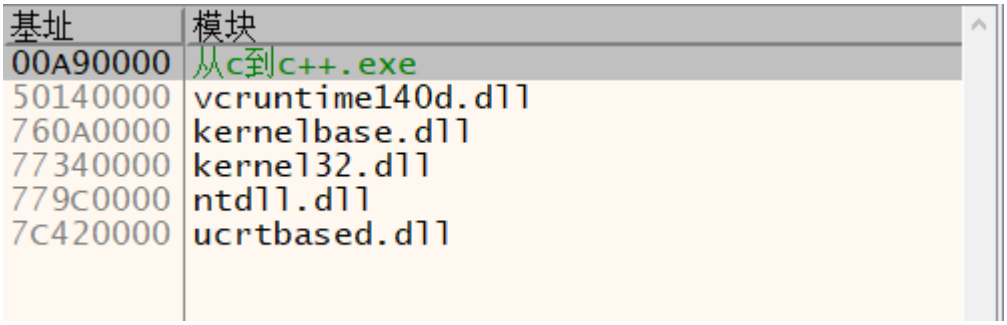


查看进程加载了哪些 Dll

在菜单栏中，选择“视图-模块”，或直接按快捷键 Alt+E，即可打开模块视图，其中显示了当前可执行程序加载的所有 Dll。



值得注意的是，如果在 64 位系统上使用的是 32 位的 x32dbg，由于 Windows 的 Wow64 机制存在，x32dbg 将只能看到 32 位的 Dll 模块，这时候显示的内容就比较少。



基址	模块
00A90000	从c到c++.exe
50140000	vcruntime140d.dll
760A0000	kernelbase.dll
77340000	kernel32.dll
779C0000	ntdll.dll
7C420000	ucrtbased.dll

查看 dll 中的导出函数

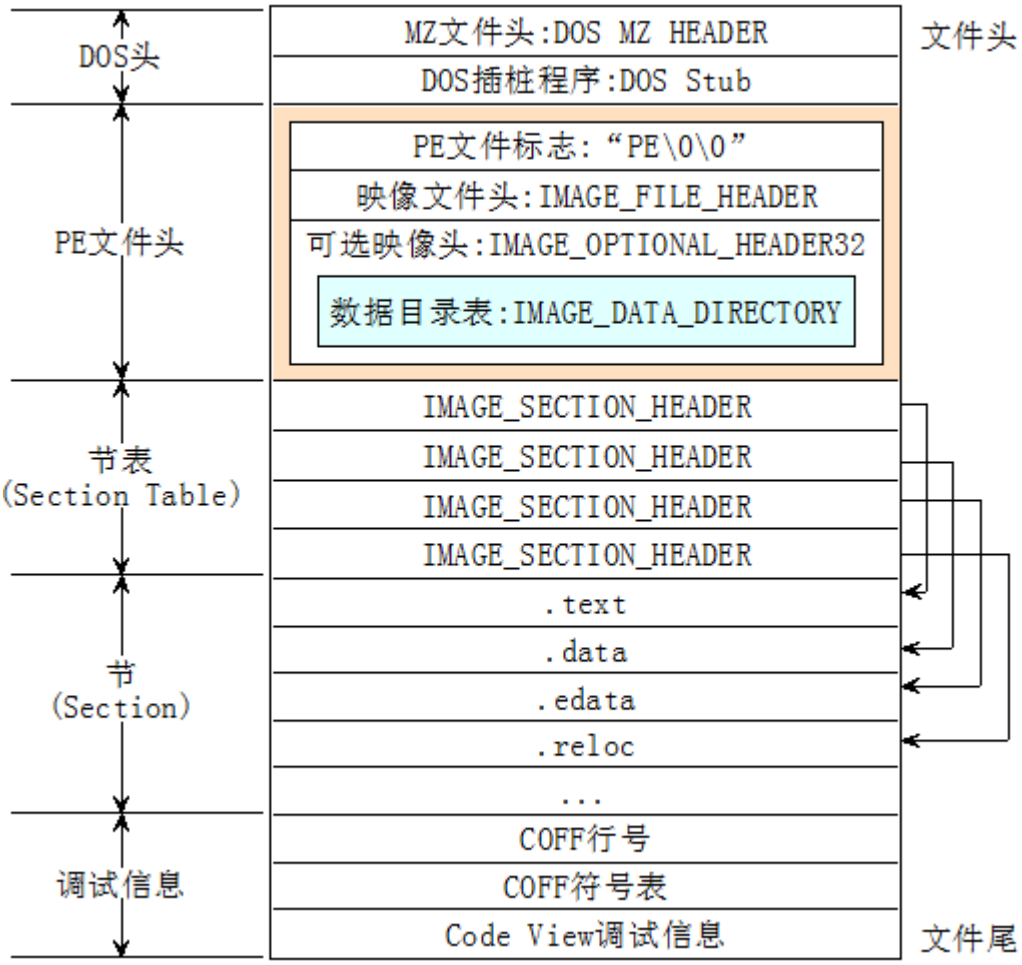
在 Dll 列表窗口，选中对应的 Dll，x64dbg 会自动在右侧展示该 Dll 的所有导入与导出函数：

地址	类型	序号	符号
00007FFA49741100	导出	1	_Aligned_get_default_resource
00007FFA49741120	导出	2	_Aligned_new_delete_resource
00007FFA49741130	导出	3	_Aligned_set_default_resource
00007FFA49741150	导出	4	_Unaligned_get_default_resource
00007FFA49741170	导出	5	_Unaligned_new_delete_resource
00007FFA49741180	导出	6	_Unaligned_set_default_resource
00007FFA497411A0	导出	7	null_memory_resource
00007FFA497415C0	导出	0	OptionalHeader.AddressOfEntryPoint
00007FFA49743070	导入		msvcp140.?_Xbad_alloc@std@@YAXXZ
00007FFA49743080	导入		vcruntime140.__std_type_info_destroy_list
00007FFA49743088	导入		vcruntime140._CxxThrowException
00007FFA49743090	导入		vcruntime140.memset
00007FFA49743098	导入		vcruntime140.__std_exception_copy
00007FFA497430A0	导入		vcruntime140.memcpy
00007FFA497430A8	导入		vcruntime140.__C_specific_handler
00007FFA497430B0	导入		vcruntime140.__std_exception_destroy
00007FFA497430C0	导入		ucrtbase._aligned_malloc
00007FFA497430C8	导入		ucrtbase._callnewh
00007FFA497430D0	导入		ucrtbase.free
00007FFA497430D8	导入		ucrtbase._aligned_free
00007FFA497430E0	导入		ucrtbase.malloc
00007FFA497430F0	导入		ucrtbase._initialize_onexit_table
00007FFA497430F8	导入		ucrtbase._configure_narrow_argv
00007FFA49743100	导入		ucrtbase._seh_filter_dll
00007FFA49743108	导入		ucrtbase._cexit
00007FFA49743110	导入		ucrtbase._initterm_e
00007FFA49743118	导入		ucrtbase._initterm
00007FFA49743120	导入		ucrtbase._initialize_narrow_environment
00007FFA49743128	导入		ucrtbase._execute_onexit_table
00007FFA49743000	导入		kernel32.QueryPerformanceCounter
00007FFA49743008	导入		kernel32.GetCurrentProcessId
00007FFA49743010	导入		kernel32.GetCurrentThreadId
00007FFA49743018	导入		kernel32.GetSystemTimeAsFileTime
00007FFA49743020	导入		ntdll.InitializeSListHead
00007FFA49743028	导入		kernel32.RtlCaptureContext
00007FFA49743030	导入		kernel32.RtlLookupFunctionEntry
00007FFA49743038	导入		kernel32.RtlVirtualUnwind
00007FFA49743040	导入		kernel32.IsDebuggerPresent
00007FFA49743048	导入		kernel32.UnhandledExceptionFilter

单击表头会按相关顺序排列，如点击符号，就会按名字排列。双击某一项，会转到对应的函数入口。

PE 格式之 RVA 与 FOA

Windows 下执行程序的格式叫 PE 格式。下面我们需要学习其格式。如此才能完成寄生。我们可以用 LordPE，PEView 之类的 PE 查看工具来学习 PE 格式。一个 PE 文件总体的结构如下：



在 PE 文件中，微软引入了两种定位方式，一种是 RVA，另一种是 FOA。

FOA：在 PE 文件中，某个地址相对于文件头部的偏移。

RVA：由于 PE 文件被加载到内存中后，需要按照一定的粒度对齐，在 PE 文件中引入了相对虚拟地址 RVA 的概念。RVA 就是加载到内存后，相对于整个 PE 文件首部的偏移量（不是硬盘上，因为两者对齐粒度不同）。

PEView 显示某个位置的偏移，可在 RVA，文件偏移间切换。手动进行 RVA 和 FOA 的转换时，可遵循如下算法：

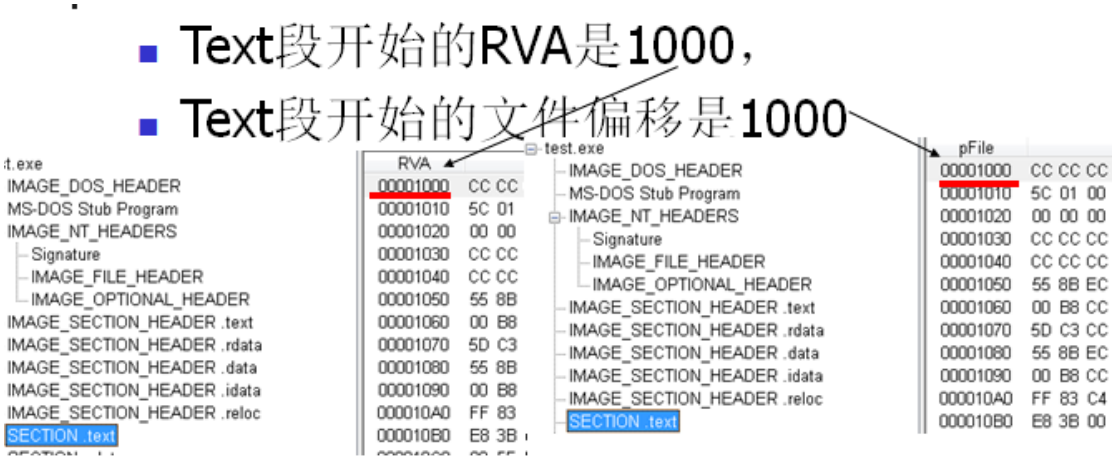
FOA 转 RVA：

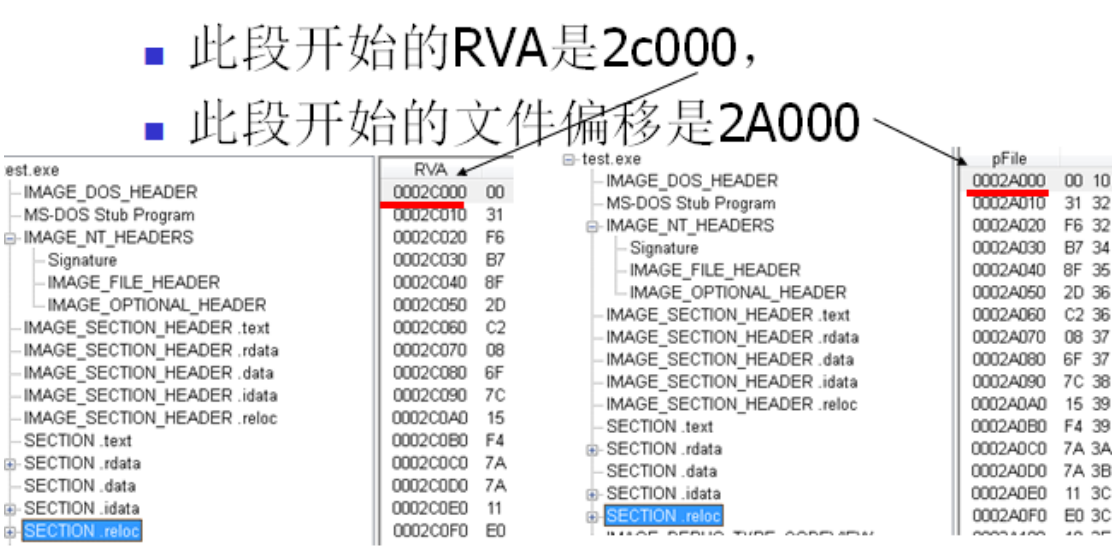
- 循环遍历各区段头，找出其文件偏移起始位置 PointerToRawData 以及结束位置 PointerToRawData+SizeOfRawData

- 若 FOA 落在某个区段的区间范围内，则用 FOA - PointerToRawData + 区段起始位置 RVA 得到其 RVA

RVA 转 FOA：

- 循环遍历各区段头，找出该区段起始位置 RVA 以及结束位置 RVA+SizeOfRawData
- 若 RVA 落在某个区段的区间范围内，则用 RVA - 该区段起始位置 RVA + 区段起始文件偏移位置 PointerToRawData 得到其 FOA





- 在本例中，导致前面段 RVA 和文件偏移不同的原因是，data 段，本来只有 4000h 大小，但因为对齐原因，下一个 idata 段的 RVA 并没有起始于 data 首址（RVA 25000h）+4000h=29000h.而更往后移动了（2B000h）。但 idata 段的 Pointer to Raw Data 就是文件偏移，依然是 29000h

PE 格式之三个头

在 PE 格式中，重要的结构信息包含 DOS 头、NT 头文件头和 NT 可选头。其中 DOS 头主要为了兼容 DOS 时代的旧文件，在 Windows 时代，只有其最后一个

字段 e_lfanew 越过 DOS_STUB 指向 NT 头的起始位置。

DOS 头的结构定义如下：

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
      } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

值得注意的是，由于其开头的魔数为 MZ，因此有时也将其称为“MZ”头。由 e_lfanew 字段标识的是 Windows 时代最为重要的 NT 头。NT 头由三部分构成：

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

其中，第一部分为魔法数字 PE\0\0,其对应的 DWORD 值为 0x00005045，这也是 PE 文件的一个典型标记。第二部分和第三部分分别是文件头和可选头。

文件头定义如下：

```
typedef struct _IMAGE_FILE_HEADER {
    WORD    Machine;
    WORD    NumberOfSections;
    DWORD   TimeDateStamp;
    DWORD   PointerToSymbolTable;
    DWORD   NumberOfSymbols;
    WORD    SizeOfOptionalHeader;
    WORD    Characteristics;
```

```
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

其中 NumberOfSections 标识了本 PE 文件中区段（有时又被译为节或节区）的数量，而 SizeOfOptionalHeader 则标识了可选头的大小，可根据此大小越过可选头直接定位到区段表（有些恶意代码为了对抗分析，会人为改变此字段的大小，如果在撰写 PE 分析工具时没有注意到可选头大小是可变的，则会解析出错）。

可选头虽然名称中有“可选”二字，但它并不是可有可无的，而是必不可少的，其定义如下：

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;                  // 又一个魔法数字
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;    // 入口点RVA
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    DWORD   ImageBase;              // 建议的加载地址
    DWORD   SectionAlignment;       // 区段内存对齐粒度
    DWORD   FileAlignment;          // 区段文件对齐粒度
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;            // PE映像大小
    DWORD   SizeOfHeaders;          // 三个头加区段表的大小
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
```

```
DWORD   SizeOfStackReserve;
DWORD   SizeOfStackCommit;
DWORD   SizeOfHeapReserve;
DWORD   SizeOfHeapCommit;
DWORD   LoaderFlags;
DWORD   NumberOfRvaAndSizes;            // 数据目录表的数量
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

在可选表中，重要的字段很多，在上面已用注释的方式进行标识。**回到病毒加载执行的问题：**

- 如何让病毒被加载？
- 如何让病毒被执行？

带着这两个问题来学习 PE 格式才能有效。