汇编语言与逆向工程

北京邮电大学
付俊松

第九章 Hook入门

- 一. Hook概述
- 二. API Hook

（1）Hook概述

- 1 Hook概念及其种类
- 2 Message Hook
- 3 IAT Hook

（1）Hook概述 — Hook概念及其种类

- Hook概念及其种类
  - 中文名叫做“钩子”或者“钩取”
  - hook技术能在程序实现正常功能的情况下，获取到程序的目的参数、信息等，或者实现嵌入的代码功能
  - 应用程序的补丁，程序的破解，插件的开发等等，都可能需要通过hook来实现
  - 可以进行钩取的方法有很多
    - 在Windows中，hook的方法主要有Message Hook，IAT Hook，API Hook等等，本章将重点讲解API Hook。

(1) Hook概述 — Message Hook

Message Hook

- Message Hook也叫做消息钩子，针对的是GUI（图形用户界面）程序
- 使用的关键Windows API函数SetWindowsHookEx，MSDN(Microsoft Developer Network)定义如下

```
HHOOK WINAPI SetWindowsHookEx(
  _In_ int       idHook,
  _In_ HOOKPROC  lpfn,
  _In_ HINSTANCE hMod,
  _In_ DWORD     dwThreadId
);
```

(1) Hook概述 — Message Hook

- 第一个参数是windows 消息值
  - 比如WH_KEYBOARD可以控制WM_KEYUP和WM_KEYDOWN两个消息值（是描述键盘虚拟键码的，它对应的是键盘物理按键Pgup和Pgdn）
- 第二个参数是我们对应消息的回调函数
  - 比如当触发按键消息的时候，设置的对应按键回调函数就会调用。
- 第三个参数是包含hook代码的动态链接库DLL句柄
- 第四个参数是所钩取的线程id，如果设置为0，则为全局钩取，即所有的GUI程序都将被钩取。

```
HHOOK WINAPI SetWindowsHookEx(
  _In_ int       idHook,
  _In_ HOOKPROC  lpfn,
  _In_ HINSTANCE hMod,
  _In_ DWORD     dwThreadId
);
```

(1) Hook概述 — Message Hook

- 在使用SetWindowsHookEx函数以后，消息钩子会先于应用程序看到相应的消息，并对消息进行拦截，处理，调用回调函数里的代码。

(1) Hook概述 — Message Hook

- 自己写的回调函数代码应该是在我们自己写的应用程序里，怎么会被其他应用程序调用的呢，进程和进程之间不应该是相互隔离的么？
- 在SetWindowsHookEx函数成功产生作用时，对应监控的应用程序里会被注入一个动态链接库DLL，我们的关键代码就写在被加载的这个DLL当中
- 下面简单介绍一下进程注入

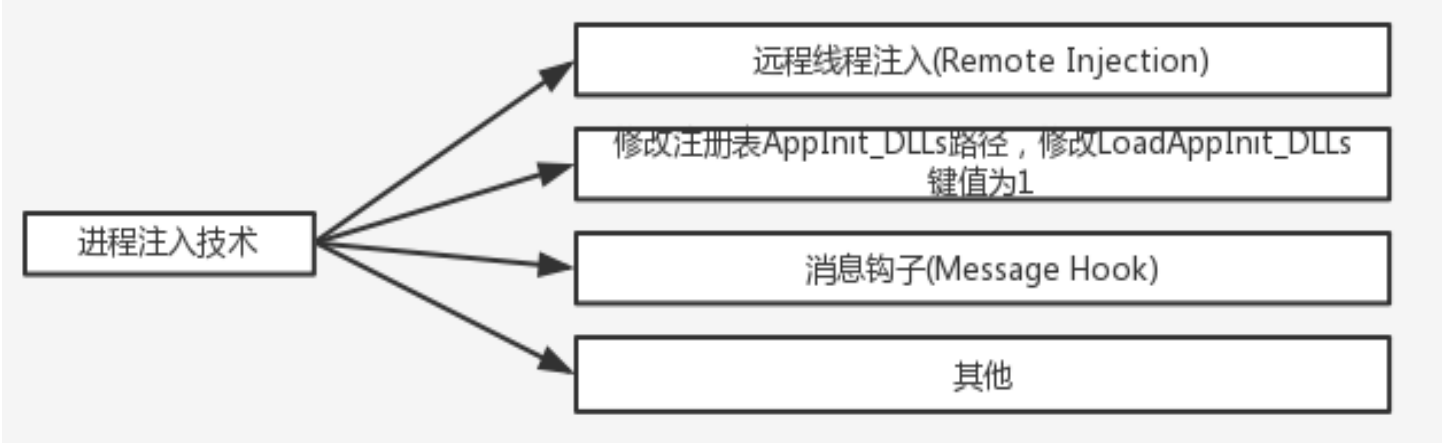
(1) Hook概述 — Message Hook

– 进程注入需求的产生

- 应用程序之间是相互独立的，各自享有自己的内存空间
- 应用程序需要跨越进程边界来访问另一个进程的地址空间时，就会使用进程注入。
- 为了对内存中的某个进程进行操作，并且获得该进程地址空间里的数据，或者修改进程的私有数据，修改程序执行流，就需要把代码放入到目的进程当中，这时就避免不了使用进程注入方法了。

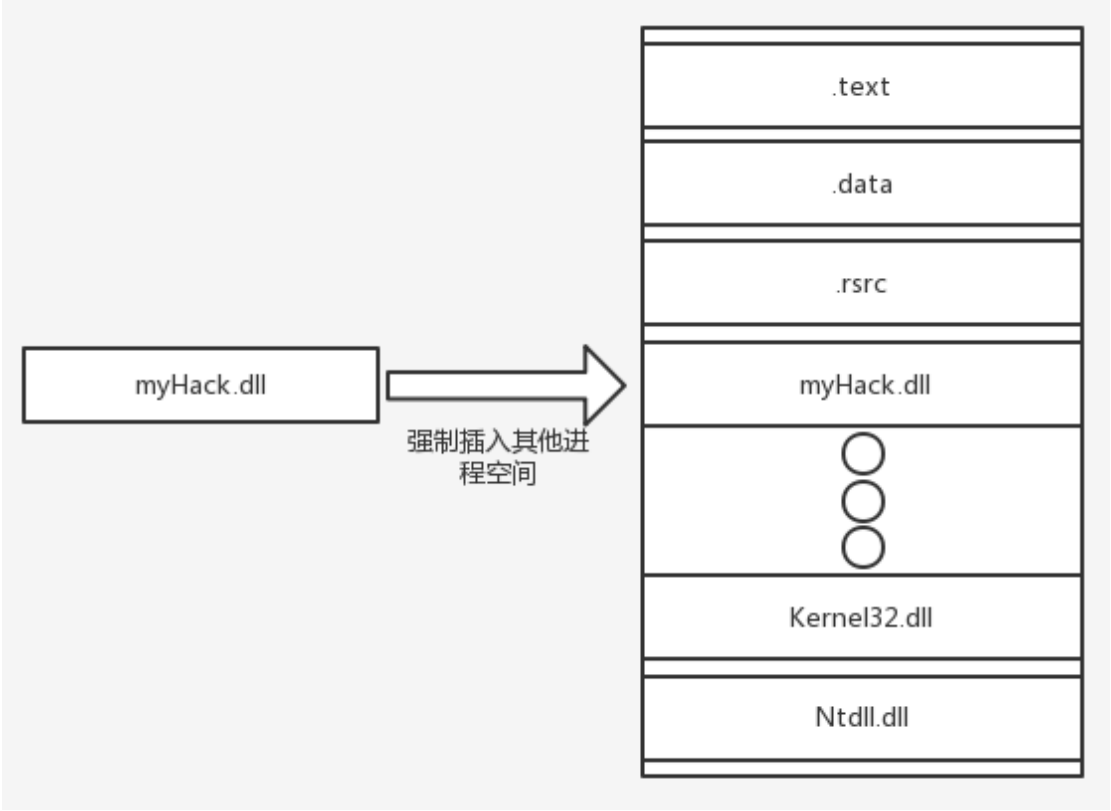
(1) Hook概述 — Message Hook

– 进程注入的方法有很多



(1) Hook概述 — Message Hook

– 进程注入图示



(1) Hook概述 — Message Hook

- Message Hook方式虽然简单，但是也存在不足，它只能捕获Windows操作系统向用户提供的消息事件，诸如借助键盘，鼠标操作的事件，能够控制的函数有限
  - 具体的消息范围都可以在MSDN的介绍中找到

(1) Hook概述 — IAT Hook

- IAT Hook
  - 对于32位的PE程序来讲，结构中有一个IMPORT Table，包含两个元素
    - 一个是第一个IMAGE_IMPORT_DESCRIPTOR的起始地址，另一个元素则是整个IMPORT Table的大小。
  - IMPORT Table中又有多个IMAGE_IMPORT_DESCRIPTOR（IDT）
    - 其中一个IDT的大小是20字节，一个空的20字节作为末尾
  - 用IMPORT Table的大小除以20，再减去1，就可以得到IDT的数量。

(1) Hook概述 — IAT Hook

– IDT包含的结构如图

➢观察PE结构所使用的工具是PEView

VA	Data	Description	Value
00403394	000033D0	Import Name Table RVA	
00403398	00000000	Time Date Stamp	
0040339C	00000000	Forwarder Chain	
004033A0	00003560	Name RVA	KERNEL32.dll
004033A4	00003000	Import Address Table RVA	
004033A8	0000341C	Import Name Table RVA	
004033AC	00000000	Time Date Stamp	
004033B0	00000000	Forwarder Chain	
004033B4	000035C8	Name RVA	MSVCR110.dll
004033B8	0000304C	Import Address Table RVA	
004033BC	00000000		
004033C0	00000000		
004033C4	00000000		
004033C8	00000000		
004033CC	00000000		

(1) Hook概述 — IAT Hook

- 在程序没有加载的时候，Import Address Table中的值和Import Name Table 的值一样，都是指向导入函数的名称。
- 当程序动态加载以后，装载器在每一个动态链接库加载过程中，都会把实际的函数地址填入Import Address Table当中，这样我们的程序在运行的过程中就可以直接调用了。

(1) Hook概述 — IAT Hook

- IAT Hook原理
  - 在程序动态装载以后，先获得原始目标API的函数地址，然后保存在一个变量里。
  - 通过查找IAT的形式，找到IAT表。
  - 将IAT中的每一项的值和之前保存目标API函数地址的变量值进行比较，如果相同，则替换成我们自己函数的地址。

(1) Hook概述 — IAT Hook

- IAT Hook方式也存在局限性
  - IAT Hook需要在程序动态装载以后，才能实施将hook程序装载进的API函数地址
  - 如果需要hook的某个函数并没有动态加载，那么这种思路行不通

第九章 Hook入门

- 一. Hook概述
- 二. API Hook

(2) API Hook

- 1 API Hook原理
- 2 动态调试API Hook

(2) API Hook — API Hook原理

- API
  - Application Programming Interface，应用程序编程接口
  - Windows API是一个实现某种功能的函数，一个内部已经封装好代码的函数，用户在使用时，无需知道其内部实现，或是理解它与系统的软硬件是如何交互的，只需要明白该API函数如何使用，通过这个接口来实现某种特定功能，这就是API。

(2) API Hook — API Hook原理

- 标准的API Hook的实现步骤
  - 第一步，获取API的函数地址
  - 第二步，修改API起始地址字节，使之跳转到我们自己定义的函数
  - 第三步，在自定义的函数中实现unhook（脱钩），恢复原API起始地址处字节
  - 第四步，执行我们自定义的代码以及原API函数，再次hook该API函数，便于下次拦截，最后返回

(2) API Hook — API Hook原理

– 第一步，获取API函数的地址

➢例子：获取WriteFile函数地址。

```
#include<stdio.h>
#include<Windows.h>
Int main()
{
        HMODULE hKernel32;
        FARPROC pWriteFile;
        hKernel32 =GetModuleHandle(L"kernel32.dll");
        pWriteFile=GetProcAddress(hKernel32, "WriteFile");
        printf("0x%p\n", pWriteFile);
Return 0;
}
```

➢相关Windows API使用可搜索msdn

(2) API Hook — API Hook原理

- 第二步，修改API函数起始地址处字节
  - 以x86为例
  - hook的原理是相同的，但是处理细节确是不同的
    - 以Ke，Nt，Zw开头的API，以及32位和64位系统下的API函数地址，入口处可修改的字节码数量可能是不同的，在处理时的细节上可能有所差异，
    - 没有一成不变的hook模板。需要用户在hook之前，自行调试，确认修改的字节无误。

```
#include<stdio.h>
#include<windows.h>
BYTE pOrgByte[5] = { 0, };
Typedef BOOL(WINAPI *PFWriteFile)(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD
lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);
Void unhook(){   }
BOOL MyWriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten,
LPOVERLAPPED lpOverlapped)
{
        FARPROC pFunc;
        unhook();
        pFunc=GetProcAddress(GetModuleHandleA("kernel32.dll"), "WriteFile");
        ((PFWriteFile)pFunc)(hFile, lpBuffer, nNumberOfBytesToWrite, lpNumberOfBytesWritten, lpOverlapped);
        return TRUE;
}
Int main()
{
        HMODULE hKernel32;
        FARPROC pWriteFile;
        PBYTE pEditFunc;
        byte pJmpCode[6] = { 0xE9,0, };
        DWORD dwOldProtect,pOffset;
        hKernel32 =GetModuleHandle(L"kernel32.dll");
        pWriteFile=GetProcAddress(hKernel32, "WriteFile");
        pEditFunc= (PBYTE)pWriteFile;
        if (VirtualProtect(pEditFunc, 5, PAGE_EXECUTE_READWRITE, &dwOldProtect))
        {
                memcpy(pOrgByte, pEditFunc, 5);
                pOffset= (ULONGLONG)MyWriteFile- (ULONGLONG)pWriteFile-5;
                memcpy(&pJmpCode[1], &pOffset, 4);
                memcpy(pWriteFile, &pJmpCode[0], 5);
                VirtualProtect(pWriteFile, 5, dwOldProtect, &dwOldProtect);
        }
return0;
}
```

(2) API Hook — API Hook原理

– 重点是main函数中的VirtualProtect函数

➢ VirtualProtect能够改变指定地址处，指定字节数量的可读可写可执行属性。

① 获取WriteFile的地址

② 对WriteFile地址处的5个字节修改了属性，使该5个字节可读、可写、可执行

③ 保存原始WriteFile函数的5个字节

④ 获得我们的自定义函数MyWriteFile和WriteFile函数间的偏移，保存于变量pOffset当中

⑤ 之后两次memcpy函数的调用修改了WriteFile起始地址处的5个字节

⑥ 最后，恢复WriteFile函数处的代码属性

(2) API Hook — API Hook原理

- 0xE9是Jmp指令的字节码，其后跟随4个将会被解析为跳转偏移的字节码，这样一来，当我们调用WriteFile函数的时候，就会跳转到MyWriteFile执行。

(2) API Hook — API Hook原理

- 第三步，在自定义的函数中实现unhook（脱钩），恢复原API起始地址处字节。
  - 第二步中，MyWriteFile函数中的unhook即是我们的脱钩函数
  - 脱钩函数在自定义函数中并不是必须存在的，如果不调用原来系统本身的API，可以选择不对函数进行脱钩，直接执行我们自己的代码。
  - 注意：我们自定义的函数，函数的返回值，参数类型，参数个数要与被hook的API保持一致，目的是保证程序在运行过程中的堆栈平衡。

(2) API Hook — API Hook原理

```
#include<stdio.h>
#include<Windows.h>
BYTE pOrgByte[6] = { 0, };
Void unhook()
{
        DWORD dwOldProtect;
        PBYTE pWriteFile;
        FARPROC pFunc;
        pFunc=GetProcAddress(GetModuleHandleA("kernel32.dll"), "WriteFile");
        pWriteFile= (PBYTE)pFunc;
        VirtualProtect(pWriteFile, 5, PAGE_EXECUTE_READWRITE, &dwOldProtect);
        memcpy(pWriteFile, pOrgByte, 5);
        VirtualProtect(pWriteFile, 5, dwOldProtect, &dwOldProtect);
}
Int main()
{
        unhook();
        return0;
}
```

(2) API Hook — API Hook原理

- 上述代码unhook函数用于脱钩
  - 使用VirtualProtect函数，先将已经修改掉的WriteFile函数地址处5字节修改代码属性
  - 然后将原WriteFile的5个字节copy回去，其中pOrgByte是之前hook代码时，保存的原WriteFile函数地址处5个字节的变量。

(2) API Hook — API Hook原理

- 第四步，执行我们自定义的代码以及原API函数，再次hook该API函数，便于下次拦截，最后返回。
- 执行了我们自己的代码以后，需不需要再次hook住该API函数，是由我们自己决定的。如果我们只选择进行一次hook，那么在unhook以及执行完我们自己的代码以后，就可以直接返回了。

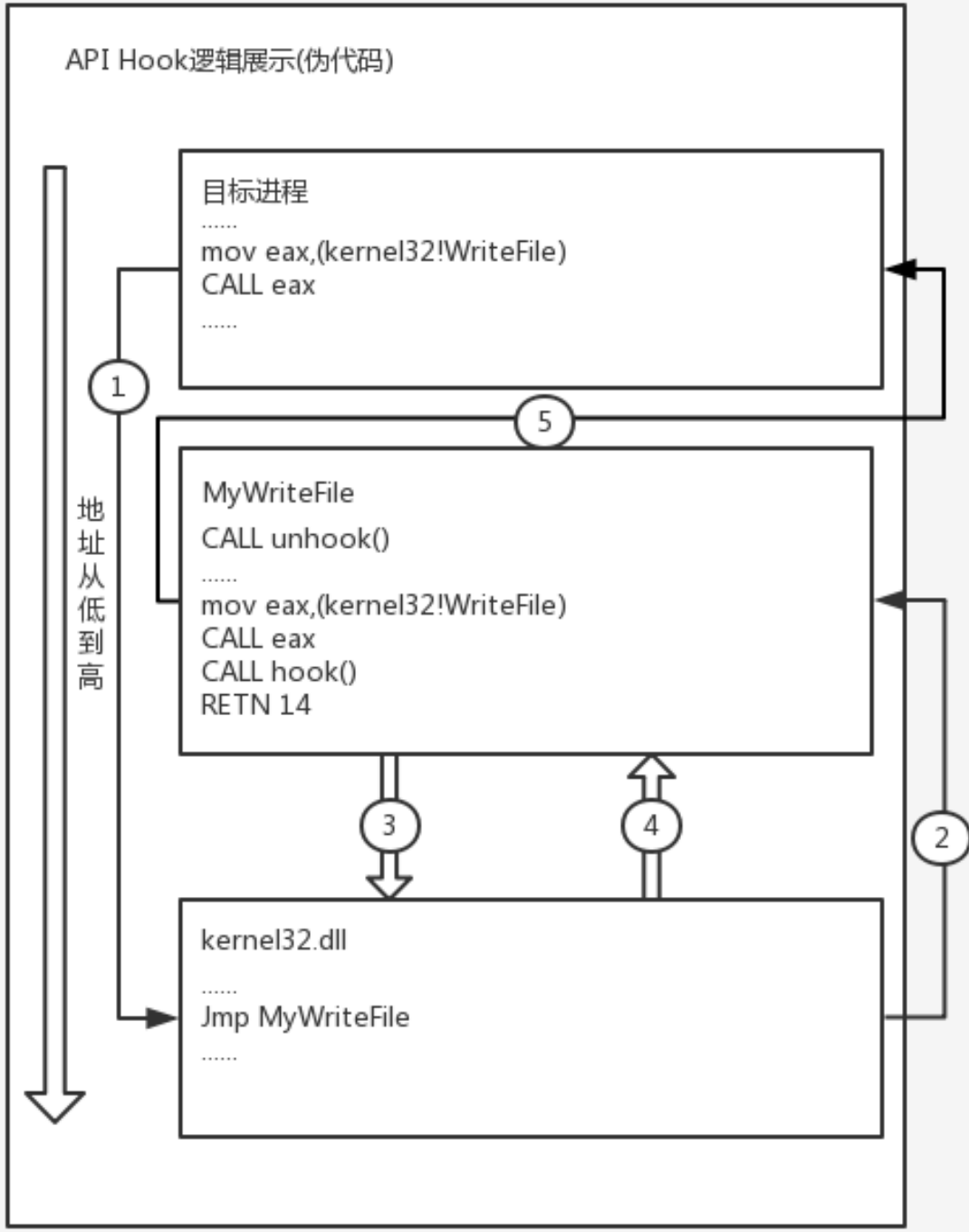
(2) API Hook — API Hook原理

– 自定义函数MyWriteFile

```
BOOL MyWriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped)
{
        FARPROC pFunc;
        unhook();
        pFunc=GetProcAddress(GetModuleHandleA("kernel32.dll"), "WriteFile");
        ((PFWriteFile)pFunc)(hFile, lpBuffer, nNumberOfBytesToWrite,
lpNumberOfBytesWritten, lpOverlapped);
        /*
        执行我们自己的代码
        */
        hook(); //代码逻辑与第二个步骤中main函数的代码相似
        return TRUE;
}
```

(2) API Hook — API Hoo

– API Hook逻辑梳理图



(2) API Hook — API Hook原理

1. 程序从目标进程位置出发，首先是调用了WriteFile函数，然后跳转到了kernel32.dll。
2. 由于此时WriteFile已经被hook，开头的5个字节对应的汇编代码已经变成了JmpMyWriteFile，程序从kernel32.dll跳转到了MyWriteFile当中。
3. 在MyWriteFile中，脱钩了WriteFile函数，执行了我们自己的代码以及WriteFile函数，程序从MyWriteFile位置又再次跳转到了kernel32.dll。
4. 在kernel32.dll中执行完WriteFile函数以后，返回到了MyWriteFile函数当中。
5. 最后再次hook了WriteFile函数，返回到我们自己一开始目标进程的位置。

(2) API Hook — 动态调试API Hook

- 2 动态调试API Hook
  - 通过IDA来动态调试一下API Hook，用调试器来感受一下API Hook
  - 首先看一下整体的C代码，对程序逻辑有一个初步的印象
  - 程序替换了原本要写入hook文件的“WriteFile”字符串，修改为了“The magic of API Hook”

```
#include<stdio.h>
#include<windows.h>
#include<string.h>
BYTE pOrgByte[5] = { 0, };
typedefBOOL(WINAPI *PFWriteFile)(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);
Void unhook()
{
        DWORD dwOldProtect;
        PBYTE pWriteFile;
        FARPROC pFunc;
        pFunc=GetProcAddress(GetModuleHandleA("kernel32.dll"), "WriteFile");
        pWriteFile= (PBYTE)pFunc;
        VirtualProtect(pWriteFile, 5, PAGE_EXECUTE_READWRITE, &dwOldProtect);
        memcpy(pWriteFile, pOrgByte, 5);
        VirtualProtect(pWriteFile, 5, dwOldProtect, &dwOldProtect);
}
BOOL __stdcallMyWriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD
lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped)
{
        FARPROC pFunc;
        charbuf[] = { "The magic of API Hook!" };
        unhook();
        pFunc=GetProcAddress(GetModuleHandleA("kernel32.dll"), "WriteFile");

        ((PFWriteFile)pFunc)(hFile, buf, strlen(buf), lpNumberOfBytesWritten, lpOverlapped);
        return TRUE;
}
```

```
Int main()
{
        HANDLE hFile;
        HMODULE hKernel32;
        FARPROC pWriteFile;
        PBYTE pEditFunc;
        byte pJmpCode[6] = { 0xE9,0, };
        DWORD dwOldProtect, pOffset,dwWritenSize;
        charbuf[] ="WriteFile";
        hKernel32 =GetModuleHandle(L"kernel32.dll");
        pWriteFile=GetProcAddress(hKernel32, "WriteFile");
        pEditFunc= (PBYTE)pWriteFile;
        if (VirtualProtect(pEditFunc, 5, PAGE_EXECUTE_READWRITE, &dwOldProtect))
        {
                memcpy(pOrgByte, pEditFunc, 5);
                pOffset= (ULONGLONG)MyWriteFile- (ULONGLONG)pWriteFile-5;
                memcpy(&pJmpCode[1], &pOffset, 4);
                memcpy(pWriteFile, &pJmpCode[0], 5);
                VirtualProtect(pWriteFile, 5, dwOldProtect, &dwOldProtect);
        }
        hFile=CreateFile(L"hook", GENERIC_WRITE | GENERIC_READ, NULL, NULL,
CREATE_ALWAYS, 0x80, NULL);
        WriteFile(hFile, buf, strlen(buf), &dwWritenSize, NULL);
        return0;
}
```

(2) API Hook — 动态调试API Hook

– 配套的程序使用的是vc6.0编译的release版本，用ida6.8打开生成的二进制程序，对程序进行分析

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  HMODULE v3; // eax@1
  FARPROC v4; // esi@1
  HANDLE v5; // eax@3
  DWORD flOldProtect; // [sp+8h] [bp-1Ch]@1
  DWORD NumberOfBytesWritten; // [sp+Ch] [bp-18h]@3
  int v9; // [sp+10h] [bp-14h]@1
  int Buffer; // [sp+18h] [bp-Ch]@1
  int v11; // [sp+1Ch] [bp-8h]@1
  __int16 v12; // [sp+20h] [bp-4h]@1

  LOWORD(v9) = 233;
  *(int *)((char *)&v9 + 2) = 0;
  Buffer = *(_DWORD *)ProcName;
  v11 = dword_405044;
  v12 = word_405048;
  v3 = GetModuleHandleA(ModuleName);
  v4 = GetProcAddress(v3, ProcName);
  if ( VirtualProtect(v4, 5u, 0x40u, &flOldProtect) )
  {
    dword_4052C0 = *(_DWORD *)v4;
    byte_4052C4 = *((_BYTE *)v4 + 4);
    *(int *)((char *)&v9 + 1) = (char *)sub_401060 - (char *)v4 - 5;
    *(_DWORD *)v4 = v9;
    *((_BYTE *)v4 + 4) = (unsigned int)((char *)sub_401060 - (char *)v4 - 5) >> 24;
    VirtualProtect(v4, 5u, flOldProtect, &flOldProtect);
  }
  v5 = CreateFileA(FileName, 0xC0000000, 0, 0, 2u, 0x80u, 0);
  WriteFile(v5, &Buffer, strlen((const char *)&Buffer), &NumberOfBytesWritten, 0);
  return 0;
}
```

(2) API Hook — 动态调试API Hook

- main函数中代码的主要功能即实现hook，先获得WriteFile的函数地址，对其进行修改，创建一个名为hook的文件，然后写入Buffer里面的内容，Buffer里面的原本内容是WriteFile

```
.data:00405030 ; CHAR ModuleName[]
.data:00405030 ModuleName      db 'kernel32.dll',0      ; DATA XREF: sub_401000+8↑o
.data:00405030                                          ; sub_401060+22↑o ...
.data:0040503D                 align 10h
.data:00405040 ; CHAR aWritefile[]
.data:00405040 aWritefile      db 'WriteFile',0         ; DATA XREF: sub_401000+3↑o
.data:00405040                                          ; sub_401060+1D↑o ...
.data:0040504A                 db    0
.data:0040504B                 db    0
.data:0040504C aTheMagicOfApiH db 'The magic of API Hook!',0 ; DATA XREF: sub_401060+A↑o
.data:00405063                 db    0
.data:00405064 ; CHAR FileName[]
.data:00405064 FileName        db 'hook',0              ; DATA XREF: _main+B5↑o
```

(2) API Hook — 动态调试API Hook

- 动态调试hook的过程
  - ida菜单栏中选择Local Win32 debugger，通过F2设置断点，在0x40115F，0x4011AE，0x401060地址处下断。然后ida菜单栏中选择Local Win32 debugger，点击左侧的开始按钮，进行调试。
  - 程序成功在0x40115F处断下，这个位置就是即将要修改WriteFile函数地址开头5个字节的地方，edx中保存的就是kernel32.dll中WriteFile的起始地址
  - 此时可看看还没有修改时，入口地址处代码，右键寄存器窗口中的edx寄存器，选择Jump，反汇编窗口就自动跳到了WriteFile的入口地址

(2) API Hook — 动态调试API Hook

```
kernel32.dll:7462FC30
kernel32.dll:7462FC30                   ; Attributes: thunk
kernel32.dll:7462FC30
kernel32.dll:7462FC30                   kernel32_WriteFile proc near
kernel32.dll:7462FC30 FF 25 78 0F 64 74 jmp     off_74640F78
kernel32.dll:7462FC30                   kernel32_WriteFile endp
kernel32.dll:7462FC30
```

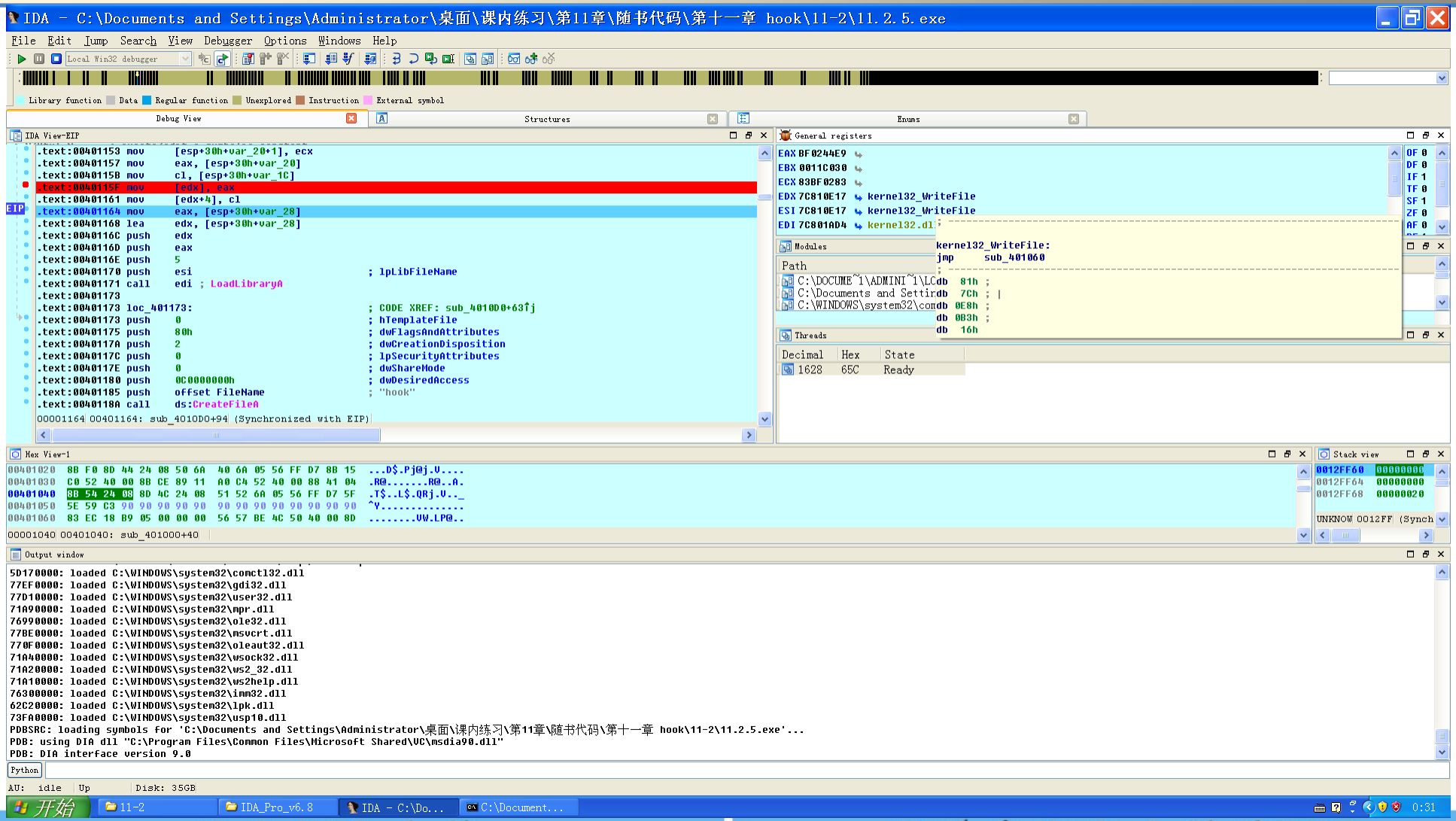
– F8单步执行两次，再次观察同一个位置，此时WriteFile入口地址变成了下图

```
kernel32.dll:7462FC30                   ; ---------------------------------------
kernel32.dll:7462FC30
kernel32.dll:7462FC30                   kernel32_WriteFile:
kernel32.dll:7462FC30 E9 2B 14 DD 8B    jmp     sub_401060
kernel32.dll:7462FC30                   ; ---------------------------------------
```

– 可以看到，修改之后跳转的位置不再位于kernel32.dll当中，而是我们的MyWriteFile函数，地址0x401060的位置。

第九章 Hook入门

(2) API Hook — 动态调试API Hook



(2) API Hook — 动态调试API Hook

– 摁下F9，到达下一处断点位置，也就是调用WriteFile之前

```
.text:00401185 68 64 50 40 00    push    offset FileName                 ; "hook"
.text:0040118A FF 15 10 40 40 00 call    ds:CreateFileA
.text:00401190 8D 4C 24 0C       lea     ecx, [esp+24h+NumberOfBytesWritten]
.text:00401194 6A 00             push    0                               ; lpOverlapped
.text:00401196 8B D0             mov     edx, eax
.text:00401198 51                push    ecx                             ; lpNumberOfBytesWritte
.text:00401199 8D 7C 24 20       lea     edi, [esp+2Ch+Buffer]
.text:0040119D 83 C9 FF          or      ecx, 0FFFFFFFFh
.text:004011A0 33 C0             xor     eax, eax
.text:004011A2 F2 AE             repne scasb
.text:004011A4 F7 D1             not     ecx
.text:004011A6 49                dec     ecx
.text:004011A7 8D 44 24 20       lea     eax, [esp+2Ch+Buffer]
.text:004011AB 51                push    ecx                             ; nNumberOfBytesToWrite
.text:004011AC 50                push    eax                             ; lpBuffer
.text:004011AD 52                push    edx                             ; hFile
.text:004011AE FF 15 0C 40 40 00 call    ds:WriteFile
.text:004011B4 5F                pop     edi
.text:004011B5 33 C0             xor     eax, eax
.text:004011B7 5E                pop     esi
.text:004011B8 83 C4 1C          add     esp, 1Ch
.text:004011BB C3                retn
.text:004011BB                   _main endp
```

(2) API Hook — 动态调试API Hook

- F7单步步入WriteFile函数，这时程序会跳到下图的位置，而后到达设置的第三个断点0x401060的位置。
- 进入之后我们反编译一下代码，可以看到这就是C源码中的MyWriteFile函数，而v6则是我们再次调用的WriteFile函数，0x401000处代码对WriteFile函数进行脱钩操作

MyWriteFile函数

```
signed int __stdcall sub_401060(int a1, int a2, int a3, int a4, int a5)
{
  HMODULE v5; // eax@1
  FARPROC v6; // edx@1
  char v8; // [sp+8h] [bp-18h]@1

  qmemcpy(&v8, aTheMagicOfApiH, 0x17u);
  sub_401000();
  v5 = GetModuleHandleA(ModuleName);
  v6 = GetProcAddress(v5, aWritefile);
  ((void (__stdcall *)(int, char *, unsigned int, int, int))v6)(a1, &v8, strlen(&v8), a4, a5);
  return 1;
}
```

(2) API Hook — 动态调试API Hook

– 以上就是hook代码的所有重点，我们直接f9结束程序，看看生成的hook文件里的内容是什么

```
1 The magic of API Hook!
```