

# API Hook 基本原理和实现[图文]

关键字：API Hook,消息,拦截 API,钩子,wskjuf

作者：wskjuf      更新：2007-12-23 08:18:25      浏览：16126

**注：**本文主要为解决论坛上 [http://www.ccrun.com/forum/forum\\_posts.asp?TID=7281](http://www.ccrun.com/forum/forum_posts.asp?TID=7281) 的提问而写的。我搜索了一下互联网，将网络上几篇有代表性的 api hook 文章的精华进行了浓缩和适当简化，写成这篇介绍性文章。另外也希望初学者能够认真思考本文采用的循序渐进的分析思路，如何解决了一个未知的问题。文中借鉴的文献资料列于文末附录一节。

## hook 是什么？

windows 系统下的编程，消息 message 的传递是贯穿其始终的。这个消息我们可以简单理解为一个有特定意义的整数，正如我们看过的老故事片中的“长江长江，我是黄河”一个含义。windows 中定义的消息给初学者的印象似乎是“不计其数”的，常见的一部分消息在 winuser.h 头文件中定义。hook 与消息有着非常密切的联系，它的中文含义是“钩子”，这样理解起来我们不难得出“**hook 是消息处理中的一个环节，用于监控消息在系统中的传递，并在这些消息到达最终的消息处理过程前，处理某些特定的消息**”。这也是 hook 分为不同种类的原因。

hook 的这个本领，使它能够将自己的代码“融入”被 hook 住的程序的进程中，成为目标进程的一个部分。我们也知道，在 windows2000 以后的系统中，普通用户程序的进程空间都是独立的，程序的运行彼此间都不受干扰。这就使我们希望通过一个程序改变其他程序的某些行为的想法不能直接实现，但是 hook 的出现给我们开拓了解决此类问题的道路。

## api hook 是什么？

在 windows 系统下编程，应该会接触到 api 函数的使用，常用的 api 函数大概有 2000 个左右。今天随着控件，stl 等高效编程技术的出现，api 的使用概率在普通的用户程序上就变得越来越小了。当诸如控件这些现成的手段不能实现的功能时，我们还需要借助 api。最初有些人对某些 api 函数的功能不太满意，就产生了如何修改这些 api，使之更好的服务于程序的想法，这样 api hook 就自然而然的出现了。我们可以通过 api hook，改变一个系统 api 的原有功能。基本的方法就是通过 hook“接触”到需要修改的 api 函数入口点，改变它的地址指向新的自定义的函数。api hook 并不属于 msdn 上介绍的 13 类 hook 中的任何一种。**所以**

说，api hook 并不是什么特别不同的 hook，它也需要通过基本的 hook 提高自己的权限，跨越不同进程间访问的限制，达到修改 api 函数地址的目的。对于自身进程空间下使用到的 api 函数地址的修改，是不需要用到 api hook 技术就可以实现的。

## api hook 和 pe 格式的关系

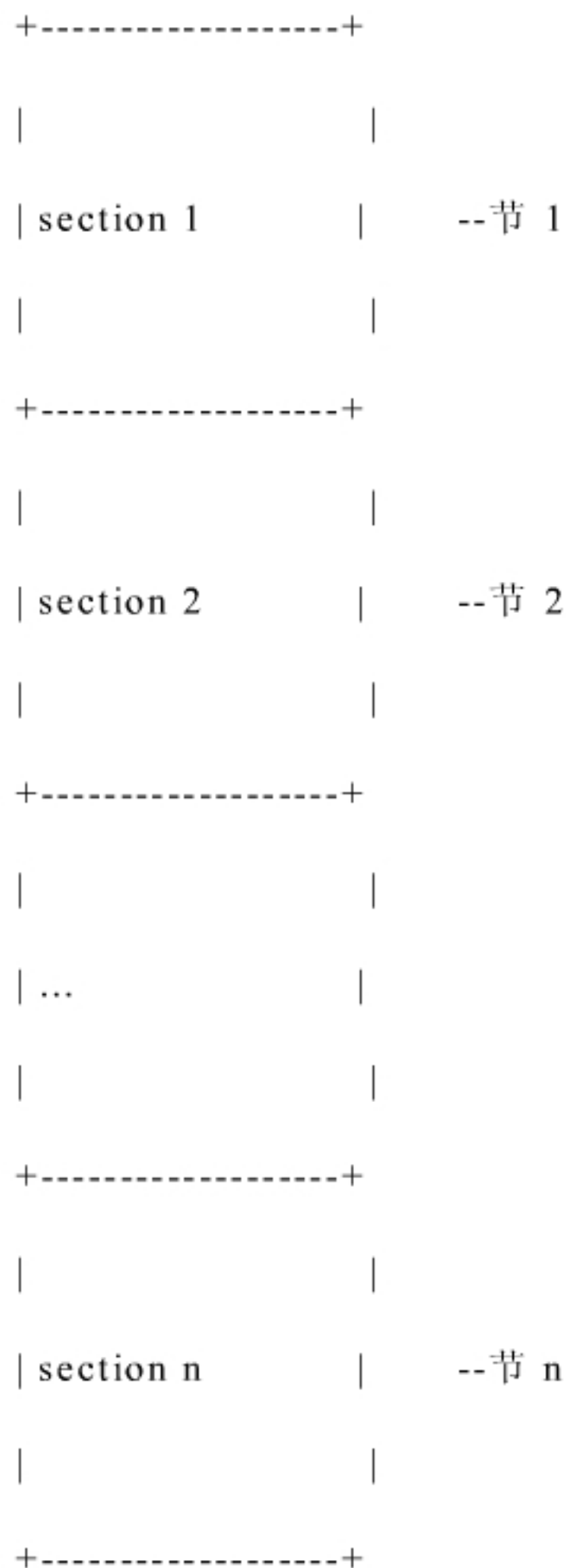
api hook 技术的难点，并不在于 hook 技术，初学者借助于资料“照葫芦画瓢”能够很容易就掌握 hook 的基本使用技术。但是如何修改 api 函数的入口地址？这就需要学习 pe 可执行文件(.exe, .dll 等)如何被系统映射到进程空间中，这就需要学习 pe 格式的基本知识。windows 已经提供了很多数据结构 struct 帮助我们访问 pe 格式，借助它们，我们就不要自己计算格式的具体字节位置这些繁琐的细节。但是从 api hook 的实现来看，pe 格式的访问部分仍然是整个编程实现中最复杂的一部分，对于经常 crack 的朋友不在此列。

假设我们已经了解了 pe 格式，那么我们在哪里修改 api 的函数入口点比较合适呢？这个就是输入符号表 imported symbols table（间接）指向的输入符号地址。

下面对于 pe 格式的介绍这一部分，对于没有接触过 pe 格式学习的朋友应该是看不太明白的，但我已经把精华部分提取出来了，学习了 pe 格式后再看这些就很容易了。

## pe 格式的基本组成

```
+-----+
| DOS-stub      |  --DOS-头
+-----+
| file-header   |  --文件头
+-----+
| optional header |  --可选头
| - - - - - |
|               |
| data directories |  --（可选头尾的）数据目录
|               |
+-----+
|               |
| section headers |  --节头
|               |
```



在上图中，我们需要从“可选头”尾的“数据目录”数组中的第二个元素——输入符号表的位置，它是一个 `IMAGE_DATA_DIRECTORY` 结构，从它中的 `VirtualAddress` 地址，“顺藤摸瓜”找到 `api` 函数的入口地点。

下图的简单说明如下：

`OriginalFirstThunk` 指向 `IMAGE_THUNK_DATA` 结构数组，为方便只画了数组的一个元素，`AddressOfData` 指向 `IMAGE_IMPORT_BY_NAME` 结构。

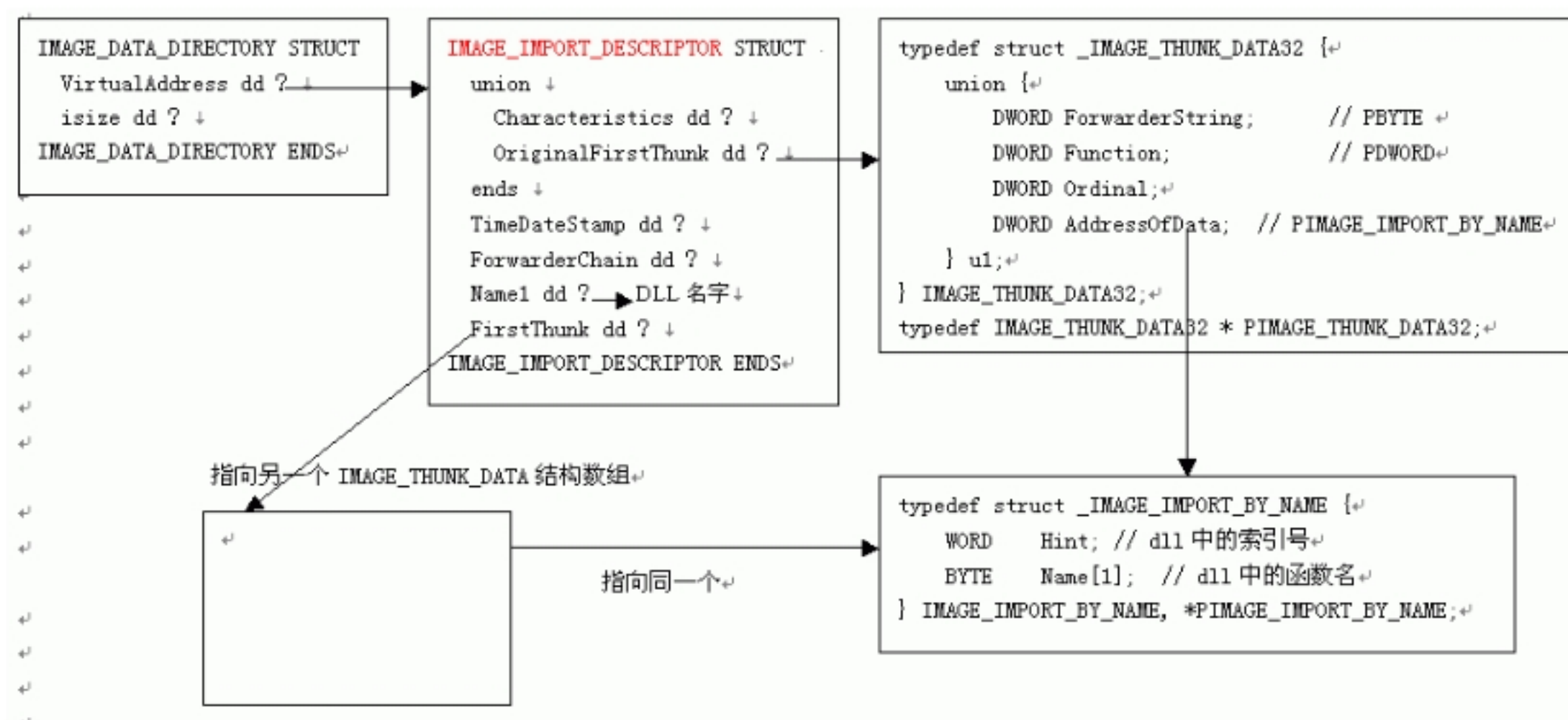
`IMAGE_IMPORT_DESCRIPTOR` 数组：每个引入的 `dll` 文件都对应数组中的一个元素，以全 0 的元素（20 个 bytes 的 0）表示数组的结束

`IMAGE_THUNK_DATA32` 数组：同一组的以全 0 的元素（4 个 bytes 的 0）表示数组的结束，每个元素对应一个 `IMAGE_IMPORT_BY_NAME` 结构

`IMAGE_IMPORT_BY_NAME`：如 `..@Consts@initialization$qqrv` 表示

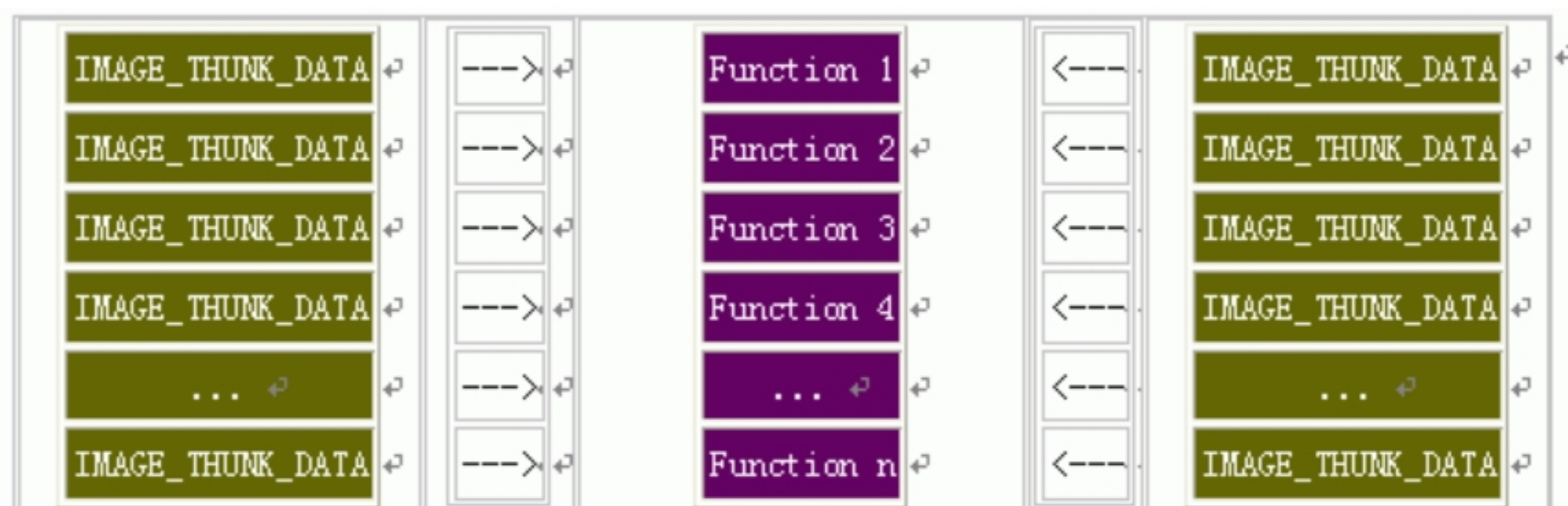
Unmangled Borland C++ Function: qualified function `__fastcall Consts::initialization()`



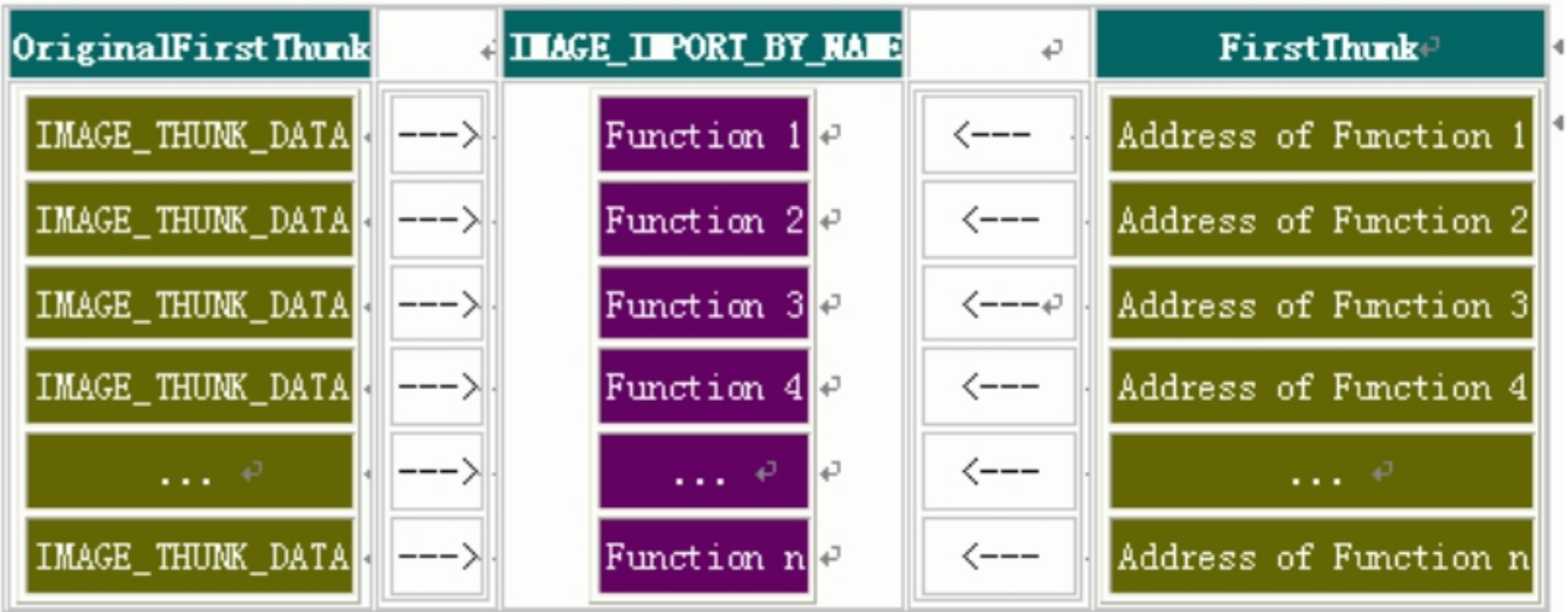


为了减少这个图的大小，不得已将汇编和 c++ 的结构都用上了。这个图是输入符号表初始化的情形，此时两个 **IMAGE\_THUNK\_DATA** 结构数组的对应元素都指向同一个 **IMAGE\_IMPORT\_BY\_NAME** 结构。

程序加载到进程空间后，两个 **IMAGE\_THUNK\_DATA** 结构数组指向有所不同了。看下图：



始化的，“两个结构都指向同一个 **IMAGE\_IMPORT\_BY\_NAME**”，此时还没有 **api** 函数地址



当 PE 文件准备执行时，前图已转换成上图。一个结构指向不变，另一个出现 api 函数地址

如果 PE 文件从 kernel32.dll 中引入 10 个函数，那么 IMAGE\_IMPORT\_DESCRIPTOR 结构的 Name1 域包含指向字符串 "kernel32.dll" 的 RVA，同时每个 IMAGE\_THUNK\_DATA 数组有 10 个元素。(RVA 是指相对地址，每一个可执行文件在加载到内存空间前，都以一个基址作为起点，其他地址以基址为准，均以相对地址表示。这样系统加载程序到不同的内存空间时，都可以方便的算出地址)

上述这些结构可以在 [winnt.h](#) 头文件里查到。

### 具体编程实现

我将手上的 vc 示例代码进行了适当修正，修改了一些资源泄漏的小问题，移植到 c++builder6 & update4 上，经过测试已经可以完成基本的 api hook 功能。有几个知识点说明一下：

#### 1、 dll 中共享内存变量的实现

正常编译下的 dll，它的变量使用到的内存是独立的。比如你同时运行两个调用了某个 dll 的用户程序，试图对某一个在 dll 中定义的全局变量修改赋值的时候，两个程序里的变量值仍然是不同的。

共享的方法为：在 .cpp 文件（.h 文件里如此设置会提示编译错误）的头部写上如上两行：

```
#pragma option -zRSHSEG          // 改变缺省数据段名
#pragma option -zTSHCLASS         // 改变缺省数据类名
```

```
HINSTANCE hdll = NULL;           // 用来保存该动态连接库的句柄
```

```
HHOOK hApiHook = NULL;          // 钩子句柄
```



```
HHOOK hWndProc = NULL; // 窗口过程钩子用来拦截 SendMessage

int threadId = 0;
```

另外建立一个与 dll 同名，不同后缀的 def 文件，如 HookDll.def 文件，写上：

```
LIBRARY HookDll.dll

EXPORTS









;...

SEGMENTS

    SHSEG CLASS 'SHCLASS' SHARED

;end
```

这样设置后在 .cpp 文件中定义的变量，如果进行了初始化，将进入“SHCLASS”共享内存段（如果不初始化，将不改变其默认段属性）。

Name	Virtual Size	Virtual Address	Size of Raw Data	Pointer to Raw Data	Characteristics	Pointing Directories
<input checked="" type="checkbox"/>  .text	00001000h	00401000h	00000A00h	00000600h	60000020h	
<input checked="" type="checkbox"/>  .data	00001000h	00402000h	00000400h	00001000h	C0000040h	
<input checked="" type="checkbox"/>  .tls	00001000h	00403000h	00000200h	00001400h	C0000040h	
<input checked="" type="checkbox"/>  SHSEG	00001000h	00404000h	00000400h	00001600h	D0000040h	
<input checked="" type="checkbox"/>  .idata	00001000h	00405000h	00000600h	00001A00h	40000040h	Import Table
<input checked="" type="checkbox"/>  .edata	00001000h	00406000h	00000200h	00002000h	40000040h	Export Table
<input checked="" type="checkbox"/>  .rsrc	00001000h	00407000h	00000600h	00002200h	40000040h	Resource Table
<input checked="" type="checkbox"/>  .reloc	00001000h	00408000h	00000200h	00002800h	50000040h	Relocation Table

上述的共享对于本示例代码并不是必须的，只是稍微演示了一下。

2、 api hook 修改 api 函数入口点地址的时机

很显然，我们必须通过 hook 进入目标进程的地址空间后，再在位于该地址空间里的 hook 消息处理过程里修改输入符号表“指向”的 api 函数入口点地址，退出 hook 前也必须在这个消息处理过程里恢复原来的地址。只要我们牢记修改的过程发生在目标进程的地址空间中，就不会发生访问违例的错误了。

示例代码使用了 WH\_GETMESSAGE、WH\_CALLWNDPROC 两中 hook 来演示如何 hook api，但 WH\_GETMESSAGE 实际上并没有完成具体的功能。

为了让初学者尽快的掌握重点，我将代码进行了简化，是一个不健壮、不灵活的演示示例。

3、 函数的内外部表现形式

例如 api 函数 `MessageBox`，这个形式是我们通常用到的，但到了 `dll` 里，它的名字很可能出现了两个形式，一个是 `MessageBoxA`，另一个是 `MessageBoxW`，这是因为系统需要适应 `Ansi` 和 `Unicode` 编码的两种形式，我们不在函数尾端添加“A”或“W”，是不能 hook 到需要的函数的。

4、 辅助 pe 格式查看工具

`PE Explorer` 是一个非常好的查看 `pe` 资源的工具，通过它可以验证自己手工计算的 `pe` 地址，可以更快的掌握 `pe` 格式。

调试器 `ollydbg` 也是非常好的辅助工具，例如查看输入符号表中的 `api` 函数。

5、 程序文件列表

`dll` 基本文件：`Hook.h`，`Hook.cpp`，`HookDll.def`

`client` 验证方基本文件：`HookTest.h`，`HookTest.cpp`，`ApiHookTest.cpp`

名称	大小
client	
ApiHook.bpf	1 KB
ApiHook.bpr	4 KB
ApiHookProjectGroup.bpg	1 KB
Hook.h	4 KB
Hook.cpp	18 KB
HookDll.def	1 KB
ApiHook.res	1 KB
Hook.obj	24 KB
ApiHook.lib	12 KB
ApiHook.tds	960 KB
ApiHookTest.exe	26 KB
ApiHook.dll	11 KB

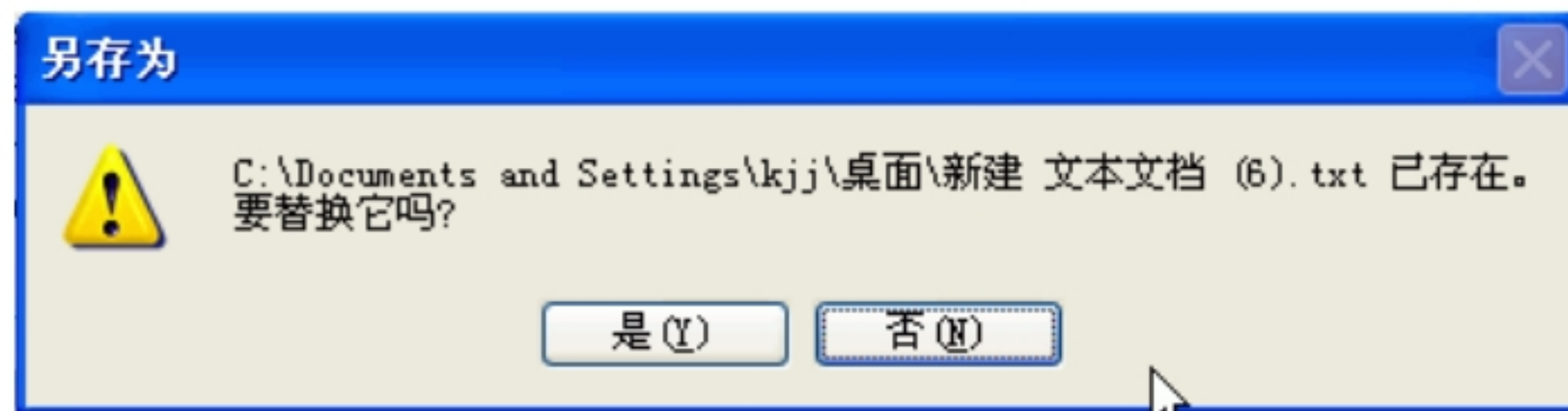
名称	大小
ApiHookTest.bpr	4 KB
HookTest.h	1 KB
HookTest.dfm	1 KB
ApiHookTest.cpp	1 KB
HookTest.cpp	2 KB
HookTest.ddp	1 KB
ApiHookTest.obj	16 KB
ApiHookTest.res	1 KB
HookTest.obj	37 KB
ApiHookTest.tds	1,984 KB
ApiHookTest.exe	26 KB

## 6、 实现的功能

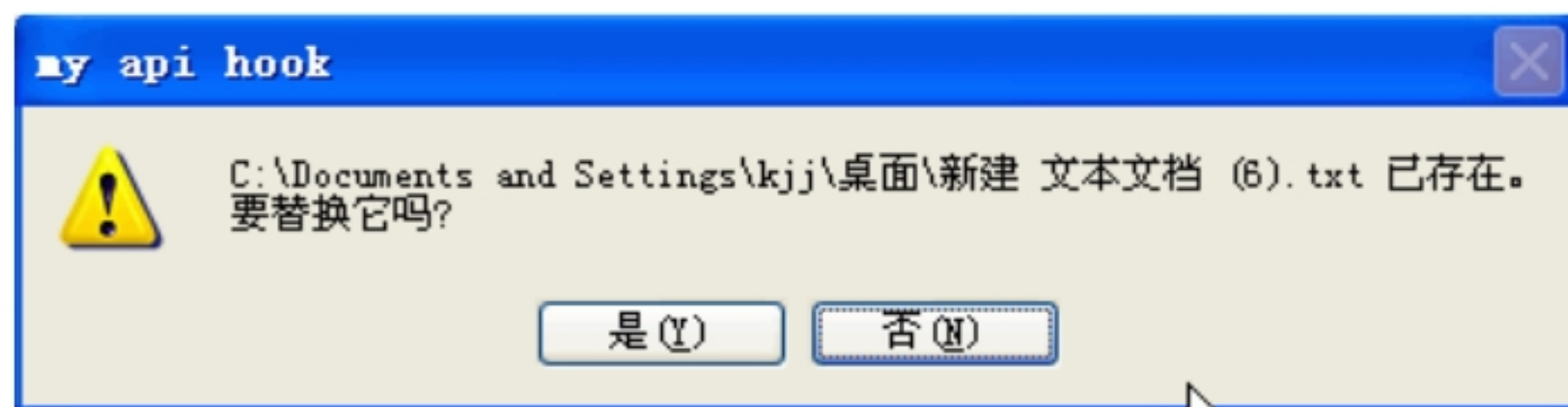
对记事本的 MessageBoxW 函数进行了 hook，先执行自定义的

```
int WINAPI MyMessageBoxW(HWND hWnd, LPCWSTR M1, LPCWSTR M2, UINT M3)
{
    return oldMessageBoxW(hWnd, M1, L"my api hook", M3);
}
```

从这里可以看到，由于目标进程空间中的执行线程并不知道你已经改变了 api 函数的实际入口地址，它在调用时仍旧将参数一成不变的压入堆栈（这个说法是汇编代码时看到的等价情形），事实上你已经提前接收到了函数调用的所有参数。这里就是篇首帖子的回复了。



hook 之前



hook 以后

## 示例代码

1、client 验证方的代码非常简单。建立一个 Application 工程，在窗体上放一个 memo（提示信息），两个 button（一个 SetHook，另一个 RemoveHook）。

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    DWORD dwProcessId, dwThreadId;

    HWND hWnd = FindWindow("Notepad", NULL);
```



```

        if (!hWnd)
        {
            Memo1->Lines->Add("Notepad is not found");
        }
        else
        {
            dwThreadId = GetWindowThreadProcessId(hWnd, &dwProcessId);

            Memo1->Lines->Add(dwThreadId);

            SetHook(dwThreadId);
        }
    }

//-----

void __fastcall TForm1::Button2Click(TObject *Sender)
{
    RemoveHook();
}

//-----

```

2、api hook dll 稍微复杂些，建立一个 dll 工程之后，修改之。代码中有一些函数并未用上，

**ReplaceApiAddress** 是核心函数，完整代码提供下载：

[ApiHook.rar](#)

## 参考文献

- 1、《iczelion 汇编程序设计教程》pe 专题部分
- 2、《WINDOWS 核心编程》第 22 章
- 3、《PE 文件格式 1.9 版》汉译版，原著 B. Luevelsmeyer
- 4、《跨进程 API Hook》，出自 <http://blog.csdn.net/detrox/archive/2004/01/29/17511.aspx>，作者 detrox
- 5、《DLL 木马注入程序》，出自 <http://www.mydown.com/code/245/245731.html>
- 6、另有两 vc6 下的源代码包，APIHOOK 与 pw，因时间久远，出处不明。在此对原作者的辛勤工作表示真挚的谢意。

本文转自：[http://www.ccrun.com/forum/forum\\_posts.asp?TID=7328&PID=25446#25446](http://www.ccrun.com/forum/forum_posts.asp?TID=7328&PID=25446#25446)