HOOK 基础知识总结

1、基本概念

钩子(Hook)，是 Windows 消息处理机制的一个平台，应用程序可以在上面设置子程以监视指定窗口的某种消息，而且所监视的窗口可以是其他进程所创建的。当消息到达后，在目标窗口处理函数之前处理它。钩子机制允许应用程序截获处理 window 消息或特定事件。钩子实际上是一个处理消息的程序段，通过系统调用，把它挂入系统。每当特定的消息发出，在没有到达目的窗口前，钩子程序就先捕获该消息，亦即钩子函数先得到控制权。这时钩子函数即可以加工处理（改变）该消息，也可以不作处理而继续传递该消息，还可以强制结束消息的传递。Hook API 是指 Windows 开放给程序员的编程接口，使得在用户级别下可以对操作系统进行控制，也就是一般的应用程序都需要调用 API 来完成某些功能，Hook API 的意思就是在这些应用程序调用真正的系统 API 前可以先被截获，从而进行一些处理再调用真正的 API 来完成功能。

HOOK 分为三种：LOCAL HOOK 和 REMOTE HOOK，还有一种是 SYSTEM-WIDE LOCAL HOOK。LOCAL HOOK 就是指程序 HOOK 的就是本程序中的线程。REMOTE HOOK 有两种形式：一种是对其他程序中某个特定的线程；一种是对整个系统。SYSTEM－WIDE LOCAL HOOK 是一种比较特殊的。它具有 REMOTE HOOK 的功能，又可以用 LOCAL HOOK 的表现手法，实际上就是 WH_JOURNALRECORD 和 WH_JOURNALPLAYBACK 两种 HOOK。REMOTE HOOK 必须封装在 DLL 中。这是因为 REMOTE HOOK 是针对整个系统或其他进程的线程，因此 HOOK 必须封装成 DLL，才可以植入到其他进程进行监控。而 SYSTEM-WIDE LOCAL HOOK 采用的是另外一种架构，系统中的线程请求或获得一个硬件消息的话，系统会调用那个安装有 HOOK 的线程，并执行它的 FILTER FUNCTION.然后再返回给请求硬件消息的线程。这种架构有一个缺点就是如果 HOOK FILTER FUNCTION 的处理中进入无限循环的话，那么整个系统将停留再循环中，无法切换到其他线程。为了处理这个缺陷，WINDOW 使用了一个办法来处理：就是 CTRL+ESC 键，如果用户按下 CTRL+ESC 键，则系统将会发送一个 WM_CANCELJOUNAL 消息到有挂上 JOUNAL 系列 HOOK 的线程上面。

2、运行机制

2.1、钩子链表和钩子子程

每一个 Hook 都有一个与之相关联的指针列表，称之为钩子链表，由系统来维护。这个列表的指针指向指定的，应用程序定义的，被 Hook 子程调用的回调函数，也就是该钩子的各个处理子程。当与指定的 Hook 类型关联的消息发生时，系统就把这个消息传递到 Hook 子程。一些 Hook 子程可以只监视消息，或者修改消息，或者停止消息的前进，避免这些消息传递到下一个 Hook 子程或者目的窗口。最近安装的钩子放在链的开始，而最早安装的钩子放在最后，也就是后加入的先获得控制权。Windows 并不要求钩子子程的卸载顺序一定得和安装顺序相反。每当有一个钩子被卸载，Windows 便释放其占用的内存，并更新整个 Hook 链表。如果程序安装了钩子，但是在尚未卸载钩子之前就结束了，那么系统会自动为它做卸载钩子的操作。钩子子程是一个应用程序定义的回调函数(CALLBACK Function),不能定义成某个类的成员函数，只能定义为普通的 C 函数。用以监视系统或某一特定类型的事件，这些事件可以是与某一特定线程关联的，也可以是系统中所有线程的事件。钩子子程必须按照以下的语法：

LRESULT CALLBACK HookProc(int nCode, WPARAM wParam, LPARAM lParam);

HookProc 是应用程序定义的名字。nCode 参数是 Hook 代码，Hook 子程使用这个参数来确定任务。这个参数的值依赖于 Hook 类型，每一种 Hook 都有自己的 Hook 代码特征字符集。wParam 和 lParam 参数的值依赖于 Hook 代码，但是它们的典型值是包含了关于发送或者接收消息的信息。

2.2、钩子的安装与释放

使用 API 函数 SetWindowsHookEx()把一个应用程序定义的钩子子程安装到钩子链表中。SetWindowsHookEx 函数总是在 Hook 链的开头安装 Hook 子程。当指定类型的 Hook 监视的事件发生时，系统就调用与这个 Hook 关联的 Hook 链的开头的 Hook 子程。每一个 Hook 链中的 Hook 子程都决定是否把这个事件传递到下一个 Hook 子程。Hook 子程传递事件到下一个 Hook 子程需要调用 CallNextHookEx 函数。

```
HHOOK SetWindowsHookEx
(

int idHook,                    // 钩子的类型，即它处理的消息类型

HOOKPROC lpfn,          // 钩子子程的地址指针。如果 dwThreadId 参数为 0
  // 或是一个由别的进程创建的线程的标识，

// lpfn 必须指向 DLL 中的钩子子程。

// 除此以外，lpfn 可以指向当前进程的一段钩子子程代码。

// 钩子函数的入口地址，当钩子钩到任何消息后便调用这个函数。

HINSTANCE hMod,

// 应用程序实例的句柄。标识包含 lpfn 所指的子程的 DLL

 // 如果 dwThreadId 标识当前进程创建的一个线程，

 // 而且子程代码位于当前进程，hMod 必须为 NULL。

 // 可以很简单的设定其为本应用程序的实例句柄。

DWORD dwThreadId     // 与安装的钩子子程相关联的线程的标识符。

 // 如果为 0，钩子子程与所有的线程关联，即为全局钩子。
```

);

函数成功则返回钩子子程的句柄，失败返回 NULL。

以上所说的钩子子程与线程相关联是指在一钩子链表中发给该线程的消息同时发送给钩子子程，且被钩子子程先处理。在钩子子程中调用得到控制权的钩子函数在完成对消息的处理后，如果想要该消息继续传递，那么它必须调用另外一个 SDK 中的 API 函数 CallNextHookEx 来传递它，以执行钩子链表所指的下一个钩子子程。这个函数成功时返回钩子链中下一个钩子过程的返回值，返回值的类型依赖于钩子的类型。这个函数的原型如下：

LRESULT CallNextHookEx (HHOOK hhk; int nCode; WPARAM wParam; LPARAM lParam;);

hhk 为当前钩子的句柄，由 SetWindowsHookEx()函数返回。NCode 为传给钩子过程的事件代码。wParam 和 lParam 分别是传给钩子子程的 wParam 值，其具体含义与钩子类型有关。钩子函数也可以通过直接返回 TRUE 来丢弃该消息，并阻止该消息的传递。否则的话，其他安装了钩子的应用程序将不会接收到钩子的通知而且还有可能产生不正确的结果。钩子在使用完之后需要用 UnHookWindowsHookEx()卸载，否则会造成麻烦。释放钩子比较简单，UnHookWindowsHookEx()只有一个参数。函数原型如下：

UnHookWindowsHookEx(HHOOK hhk;);

函数成功返回 TRUE，否则返回 FALSE。

2.3、一些运行机制：在 Win16 环境中，DLL 的全局数据对每个载入它的进程来说都是相同的；而在 Win32 环境中，情况却发生了变化，DLL 函数中的代码所创建的任何对象（包括变量）都归调用它的线程或进程所有。当进程在载入 DLL 时，操作系统自动把 DLL 地址映射到该进程的私有空间，也就是进程的虚拟地址空间，而且也复制该 DLL 的全局数据的一份拷贝到该进程空间。也就是说每个进程所拥有的相同的 DLL 的全局数据，它们的名称相同，但其值却并不一定是相同的，而且是互不干涉的。因此，在 Win32 环境下要想在多个进程中共享数据，就必须进行必要的设置。在访问同一个 Dll 的各进程之间共享存储器是通过存储器映射文件技术实现的。也可以把这些需要共享的数据分离出来，放置在一个独立的数据段里，并把该段的属性设置为共享。必须给这些变量赋初值，否则编译器会把没有赋初始值的变量放在一个叫未被初始化的数据段中。#pragma data_seg 预处理指令用于设置共享数据段。例如：

#pragma data_seg(“SharedDataName”)
HHOOK hHook=NULL;#pragma data_seg()

在# pragma data_seg(“SharedDataName”)和#pragma data_seg()之间的所有变量将被访问该 Dll 的所有进程看到和共享。再加上一条指令
#pragma comment(linker,"/section:.SharedDataName,rws"),
那么这个数据节中的数据可以在所有 DLL 的实例之间共享。所有对这些数据的操作都针对同一个实例的，而不是在每个进程的地址空间中都有一份。当进程隐式或显式调用一个动态库

里的函数时，系统都要把这个动态库映射到这个进程的虚拟地址空间里(以下简称"地址空间")。这使得 DLL 成为进程的一部分，以这个进程的身份执行，使用这个进程的堆栈。

2.4、系统钩子与线程钩子

SetWindowsHookEx()函数的最后一个参数决定了此钩子是系统钩子还是线程钩子。线程勾子用于监视指定线程的事件消息。线程勾子一般在当前线程或者当前线程派生的线程内。系统勾子监视系统中的所有线程的事件消息。因为系统勾子会影响系统中所有的应用程序，所以勾子函数必须放在独立的动态链接库(DLL) 中。系统自动将包含"钩子回调函数"的 DLL 映射到受钩子函数影响的所有进程的地址空间中，即将这个 DLL 注入了那些进程。几点说明:（1）如果对于同一事件（如鼠标消息）既安装了线程勾子又安装了系统勾子，那么系统会自动先调用线程勾子，然后调用系统勾子。 （2）对同一事件消息可安装多个勾子处理过程，这些勾子处理过程形成了勾子链。当前勾子处理结束后应把勾子信息传递给下一个勾子函数。（3）勾子特别是系统勾子会消耗消息处理时间，降低系统性能。只有在必要的时候才安装勾子，在使用完毕后要及时卸载。

3、钩子类型

每一种类型的 Hook 可以使应用程序能够监视不同类型的系统消息处理机制。下面描述所有可以利用的 Hook 类型。

1) 、 WH_CALLWNDPROC 和 WH_CALLWNDPROCRET HooksWH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks 使你可以监视发送到窗口过程的消息。系统在消息发送到接收窗口过程之前调用 WH_CALLWNDPROC Hook 子程，并且在窗口过程处理完消息之后调用 WH_CALLWNDPROCRET Hook 子程。WH_CALLWNDPROCRET Hook 传递指针到 CWPRETSTRUCT 结构，再传递到 Hook 子程。CWPRETSTRUCT 结构包含了来自处理消息的窗口过程的返回值，同样也包括了与这个消息关联的消息参数。

2)、 WH_CBT Hook 在以下事件之前，系统都会调用 WH_CBT Hook 子程，这些事件包括：1). 激活，建立，销毁，最小化，最大化，移动，改变尺寸等窗口事件；2). 完成系统指令；3). 来自系统消息队列中的移动鼠标，键盘事件；4) 设置输入焦点事件；5). 同步系统消息队列事件。Hook 子程的返回值确定系统是否允许或者防止这些操作中的一个。

3)、 WH_DEBUG Hook 在系统调用系统中与其他 Hook 关联的 Hook 子程之前，系统会调用 WH_DEBUG Hook 子程。你可以使用这个 Hook 来决定是否允许系统调用与其他 Hook 关联的 Hook 子程。

4)、 WH_FOREGROUNDIDLE Hook 当应用程序的前台线程处于空闲状态时，可以使用 WH_FOREGROUNDIDLE Hook 执行低优先级的任务。当应用程序的前台线程大概要变成空闲状态时，系统就会调用 WH_FOREGROUNDIDLE Hook 子程。

5)、 WH_GETMESSAGE Hook 应用程序使用 WH_GETMESSAGE Hook 来监视从 GetMessage or PeekMessage 函数返回的消息。你可以使用 WH_GETMESSAGE Hook 去监视鼠标和键盘输入，

以及其他发送到消息队列中的消息。

6)、 WH_JOURNALPLAYBACK HookWH_JOURNALPLAYBACK Hook 使应用程序可以插入消息到系统消息队列。可以使用这个 Hook 回放通过使用 WH_JOURNALRECORD Hook 记录下来的连续的鼠标和键盘事件。只要 WH_JOURNALPLAYBACK Hook 已经安装，正常的鼠标和键盘事件就是无效的。WH_JOURNALPLAYBACK Hook 是全局 Hook，它不能象线程特定 Hook 一样使用。WH_JOURNALPLAYBACK Hook 返回超时值，这个值告诉系统在处理来自回放 Hook 当前消息之前需要等待多长时间（毫秒）。这就使 Hook 可以控制实时事件的回放。WH_JOURNALPLAYBACK 是 system-wide local hooks，它們不會被注射到任何行程位址空間。

7)、 WH_JOURNALRECORD HookWH_JOURNALRECORD Hook 用来监视和记录输入事件。典型的，可以使用这个 Hook 记录连续的鼠标和键盘事件，然后通过使用 WH_JOURNALPLAYBACK Hook 来回放。WH_JOURNALRECORD Hook 是全局 Hook，它不能象线程特定 Hook 一样使用。WH_JOURNALRECORD 是 system-wide local hooks，它們不會被注射到任何行程位址空間。

8)、 WH_KEYBOARD Hook 在应用程序中，WH_KEYBOARD Hook 用来监视 WM_KEYDOWN and WM_KEYUP 消息，这些消息通过 GetMessage or PeekMessage function 返回。可以使用这个 Hook 来监视输入到消息队列中的键盘消息。

9)、 WH_KEYBOARD_LL HookWH_KEYBOARD_LL Hook 监视输入到线程消息队列中的键盘消息。

10)、WH_MOUSE HookWH_MOUSE Hook 监视从 GetMessage 或者 PeekMessage 函数返回的鼠标消息。使用这个 Hook 监视输入到消息队列中的鼠标消息。

11)、WH_MOUSE_LL HookWH_MOUSE_LL Hook 监视输入到线程消息队列中的鼠标消息。

12)、WH_MSGFILTER 和 WH_SYSMSGFILTER HooksWH_MSGFILTER 和 WH_SYSMSGFILTER Hooks 使我们可以监视菜单，滚动条，消息框，对话框消息并且发现用户使用 ALT+TAB or ALT+ESC 组合键切换窗口。WH_MSGFILTER Hook 只能监视传递到菜单，滚动条，消息框的消息，以及传递到通过安装了 Hook 子程的应用程序建立的对话框的消息。WH_SYSMSGFILTER Hook 监视所有应用程序消息。WH_MSGFILTER 和 WH_SYSMSGFILTER Hooks 使我们可以在模式循环期间过滤消息，这等价于在主消息循环中过滤消息。通过调用 CallMsgFilter function 可以直接的调用 WH_MSGFILTER Hook。通过使用这个函数，应用程序能够在模式循环期间使用相同的代码去过滤消息，如同在主消息循环里一样。

13)、WH_SHELL Hook

外壳应用程序可以使用 WH_SHELL Hook 去接收重要的通知。当外壳应用程序是激活的并且当顶层窗口建立或者销毁时，系统调用 WH_SHELL Hook 子程。WH_SHELL 共有 5 钟情况：1). 只要有个 top-level、unowned 窗口被产生、起作用、或是被摧毁；2). 当 Taskbar 需要重画某个按钮；3). 当系统需要显示关于 Taskbar 的一个程序的最小化形式；4). 当目前的键盘布局状态改变；5). 当使用者按 Ctrl+Esc 去执行 Task Manager（或相同级别的程序）。按照惯例，外壳应用程序都不接收 WH_SHELL 消息。所以，在应用程序能够接收 WH_SHELL 消息之

前，应用程序必须调用 SystemParametersInfo function 注册它自己。

4、Hook 消息的一个例子

Hook 全局键盘消息，从而可以知道用户按了哪些键，这种 Hook 消息的功能可以由以下函数来完成，该函数将一个新的 Hook 加入到原来的 Hook 链中，当某一消息到达后会依次经过它的 Hook 链再交给应用程序。

HHOOK SetWindowsHookEx(

int idHook, //Hook 类型，例如 WH_KEYBOARD，WH_MOUSE

HOOKPROC

lpfn, //Hook 处理过程函数的地址

HINSTANCE hMod, //包含 Hook 处理过程函数的 dll 句柄（若在本进程可以为 NULL）

DWORD dwThreadId, //要 Hook 的线程 ID，若为 0，表示全局 Hook 所有

);

这里需要提一下的就是如果是 Hook 全局的而不是某个特定的进程则需要将 Hook 过程编写为一个 DLL，以便让任何程序都可以加载它来获取 Hook 过程函数。
而对于 Hook API 微软并没有提供直接的接口函数，也许它并不想让我们这样做，不过有 2 种方法可以完成该功能。第一种，修改可执行文件的 IAT 表（即输入表），因为在该表中记录了所有调用 API 的函数地址，则只需将这些地址改为自己函数的地址即可，但是这样有一个局限，因为有的程序会加壳，这样会隐藏真实的 IAT 表，从而使该方法失效。第二种方法是直接跳转，改变 API 函数的头几个字节，使程序跳转到自己的函数，然后恢复 API 开头的几个字节，在调用 AP 完成功能后再改回来又能继续 Hook 了，但是这种方法也有一个问题就是同步的问题，当然这是可以克服的，并且该方法不受程序加壳的限制。 下面将以一个 Hook 指定程序 send 函数的例子来详细描述如何 Hook API，以达到监视程序发送的每个封包的目的。采用的是第二种方法，编写为一个 dll。首先是一些全局声明，//本 dll 的 handleHANDLE g_hInstance = NULL;//修改 API 入口为 mov eax, 00400000;jmp eax 是程序能跳转到自己的函数 BYTE g_btNewBytes[8] = { 0xB8, 0x0, 0x0, 0x40, 0x0, 0xFF, 0xE0, 0x0 };//保存原 API 入口的 8 个字节 DWORD g_dwOldBytes[2][2] = { 0x0, 0x0, 0x0, 0x0 };//钩子句柄 HHOOK g_hOldHook = NULL;//API 中 send 函数的地址 DWORD g_pSend = 0;//事务，解决同步问题 HANDLE g_hSendEvent = NULL;//自己的 send 函数地址，参数必须与 API 的 send 函数地址相同 int _stdcall hook_send(SOCKET s, const char *buf, int len, int flags);//要 Hook 的进程和主线程 ID 号 DWORD g_dwProcessID = 0;

DWORD g_dwThreadID = 0;
从声明可以看出，我们会把 API 函数的首 8 个字节改为 mov eax, 00400000;jmp eax ，使程序能够跳转，只需获取我们自己的函数地址填充掉 00400000 即可实现跳转。而 g_dwOldBytes 是用来保存 API 开头原始的 8 个字节，在真正执行 API 函数是需要写回。还有一点，在声明

新的函数时，该例中为 hook_send，除了保正参数与 API 的一致外，还需要声明为__stdcall 类型，表示函数在退出前自己来清理堆栈，因为这里是直接跳转到新函数处，所以必须自己清理堆栈。下面看主函数， BOOL APIENTRY DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved){ If (ul_reason_for_call == DLL_PROCESS_ATTACH) { //获取本 dll 句柄 g_hInstance = hModule; // 创建事务 g_hSendEvent = CreateEvent(NULL, FALSE, TRUE, NULL); //重写 API 开头的 8 字节 HMODULE hWsock = LoadLibrary("wsock32.dll"); g_pSend = (DWORD)GetProcAddress(hWsock, "send"); // 保存原始字节 ReadProcessMemory(INVALID_HANDLE_VALUE, (void *)g_pSend, (void *)g_dwOldBytes[0], sizeof(DWORD)*2, NULL); //将 00400000 改写为我们函数的地址 *(DWORD*)(g_btNewBytes + 1) = (DWORD)hook_send; WriteProcessMemory(INVALID_HANDLE_VALUE, (void *)g_pSend, (void *)g_btNewBytes, sizeof(DWORD)*2, NULL); } return TRUE;

}

以上是 dll 的 main 函数，在被指定的程序加载的时候会自动运行 dll 的 main 函数来完成初始化，这里就是改写 API 的首地址来完成跳转。当然本程序是对于指定程序进行 Hook，如果要进行全局 Hook，可以在 main 函数中用 GetModuleFileName 函数来获取 exe 文件完整路径，判断当前进程是否是想要 Hook 的进程。写函数中使用 INVALID_HANDLE_VALUE，表示写本进程。

```
int _stdcall hook_send( SOCKET s, const char *buf, int len, int flags ){
    int nRet;
    WaitForSingleObject( g_hSendEvent, INFINITE ); // 恢复 API 头 8 个字节 WriteProcessMemory( INVALID_HANDLE_VALUE, ( void* )g_pSend, ( void* )g_dwOldBytes[0], sizeof( DWORD )*2, NULL ); /* 这里可以添加想要进行的处理过程*/ //真正执行 API 函数 nRet = send( s, buf, len, flags ); // 写入跳转语句，继续 Hook WriteProcessMemory( INVALID_HANDLE_VALUE, (void*)g_pSend, (void*)g_btNewBytes, sizeof(DWORD)*2,NULL );

    SetEvent( g_hSendEvent );
    return nRet;}HOOK_API BOOL StartHook(HWND hWnd){ //通过传入的窗口句柄获取线程句柄 g_dwThreadID = GetWindowThreadProcessId( hWnd, &g_dwProcessID ); //WH_CALLWNDPROC 类型的 Hook g_hOldHook = SetWindowsHookEx( WH_CALLWNDPROC, HookProc, ( HINSTANCE ) g_hInstance, g_dwThreadID ); if( g_hOldHook == NULL ) return FALSE; return TRUE;}static LRESULT WINAPI HookProc( int nCode, WPARAM wParam, LPARAM lParam ) {return CallNextHookEx( g_hOldHook, nCode, wParam, lParam ); }HOOK_API void StopHook(void){ if(g_hOldHook != NULL) { WaitForSingleObject( g_hSendEvent, INFINITE ); HANDLE hProcess = NULL; hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, g_dwProcessID); DWORD dwOldProc; DWORD dwNewProc; // 改变页面属性为读写 VirtualProtectEx( hProcess, ( void* )g_pSend, 8, PAGE_READWRITE, &dwOldProc ); //恢复 API 的首 8 个字节 WriteProcessMemory( hProcess, ( void* )g_pSend,
```

```
( void* )g_dwOldBytes[0], sizeof( DWORD )*2, NULL );    //恢复页面文件的属性
VirtualProtectEx( hProcess, ( void* )g_pSend, 8, dwOldProc, &dwNewProc );
CloseHandle(g_hSendEvent);          UnhookWindowsHookEx( g_hOldHook );      }

}
```

可以看出，我们创建的 Hook 类型是 WH_CALLWNDPROC 类型，该类型的 Hook 在进程与系统一通信时就会被加载到进程空间，从而调用 dll 的 main 函数完成真正的 Hook，而在 SetWindowsHookEx 函数中指定的 HookProc 函数将不作任何处理，只是调用 CallNextHookEx 将消息交给 Hook 链中下一个环节处理，因为这里 SetWindowsHookEx 的唯一作用就是让进程加载我们的 dll。　　以上就是一个最简单的 Hook API 的例子，该种技术可以完成许多功能。例如网游外挂制作过程中截取发送的与收到的封包即可使用该方法，或者也可以在 Hook 到 API 后加入木马功能，反向连接指定的主机或者监听某一端口，还有许多加壳也是用该原理来隐藏 IAT 表，填入自己的函数地址。

5、关于 HOOK 效率

使用 HOOK 将会降低系统效率，因为它增加了系统处量消息的工作量。建议在必要时才使用 HOOK，并在消息处理完成后立即移去该 HOOK。建议只在调试时使用全局 HOOK 函数。全局 HOOK 函数将降低系统效率，并且会同其它使用该类 HOOK 的应用程序产生冲突。