

枚举消息钩子是 ARK 的经典功能之一，从 ARK 的鼻祖 IceSword 开始就有了此项功能。在我的 64 位 ARK（WIN64AST）里，也提供了这项功能。接下来我就给大家解密我在 WIN64AST 里是怎么实现枚举消息钩子的。

首先简单说一下什么是“消息钩子”，给完全没有这方面知识的读者解惑。以下内容摘自百度（有这方面知识的人可以忽略不看）：

Windows 系统是建立在事件驱动的机制上的，说穿了就是整个系统都是通过消息的传递来实现的。而消息钩子是 Windows 系统中非常重要的系统接口，用它可以截获并处理送给其他应用程序的消息，来完成普通应用程序难以实现的功能。消息钩子可以监视系统或进程中的各种事件消息，截获发往目标窗口的消息并进行处理。这样，我们就可以在系统中安装自定义的钩子，监视系统中特定事件的发生，完成特定的功能，比如截获键盘、鼠标的输入，屏幕取词，日志监视等等。

钩子的类型：

- （1） 键盘钩子和低级键盘钩子可以监视各种键盘消息。
- （2） 鼠标钩子和低级鼠标钩子可以监视各种鼠标消息。
- （3） 外壳钩子可以监视各种 Shell 事件消息。比如启动和关闭应用程序。
- （4） 日志钩子可以记录从系统消息队列中取出的各种事件消息。
- （5） 窗口过程钩子监视所有从系统消息队列发往目标窗口的消息。

此外，还有一些特定事件的钩子提供给我们使用，不一一列举。

常用的 Hook 类型：

1、WH_CALLWNDPROC 和 WH_CALLWNDPROCRET Hooks

WH_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 或 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 Hook

WH_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，它们不会被注射到任何进程地址空间。（估计按键精灵是用这个 hook 做的）

7、WH_JOURNALRECORD Hook

WH_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，它们不会被注射到任何进程地址空间。

接下来进入正题，说说枚举消息钩子的总体思路。首先获得名为 gSharedInfo 的全局变量的地址（此变量在 user32.dll 里被导出），它的值其实是一个内核结构体 win32k!tagsharedinfo 的地址：

```
lkd> dt win32k!tagsharedinfo
+0x000 psi                : Ptr64 tagSERVERINFO
+0x008 aheList             : Ptr64 _HANDLEENTRY
+0x010 HeEntrySize        : Uint4B
+0x018 pDispInfo          : Ptr64 tagDISPLAYINFO
+0x020 ulSharedDelta       : Uint8B
+0x028 awmControl          : [31] _WNDMSG
+0x218 DefWindowMsgs       : _WNDMSG
+0x228 DefWindowSpecMsgs   : _WNDMSG
```

接下来 tagSERVERINFO.cHandleEntries 的值，这个值记录了当前消息钩子的数目（记为 count）：

```
lkd> dt win32k!tagSERVERINFO
+0x000 dwSRVIFlags        : Uint4B
+0x008 cHandleEntries     : Uint8B
+0x010 mpFnidPfn          : [32] Ptr64      int64
+0x110 aStoCidPfn         : [7] Ptr64      int64
[以下内容太长省略.....]
```

然后读取 gSharedInfo+8 的值，获得 aheList 的值（记为 phe），此值为首个 HANDLEENTRY 结构体的地址：

```
lkd> dt win32k!_HANDLEENTRY
+0x000 phead              : Ptr64 _HEAD
+0x008 pOwner             : Ptr64 Void
+0x010 bType              : UChar
+0x011 bFlags             : UChar
+0x012 wUniq              : Uint2B
```

接下来，从 phe 首地址开始，获得【count-1】个的 HANDLEENTRY 结构体的指

针，HANDLEENTRY.phead 记录的值指向一个 HEAD 结构体，HEAD 结构体才记录了每个消息钩子的具体信息。当然也不是每个 HANDLEENTRY 都是消息钩子，只有当 HANDLEENTRY.bType 为 5 时才是消息钩子。HEAD 结构体的定义如下（记录了不少有用的信息，比如钩子类型、钩子句柄、钩子函数地址等）：

```
lkd> dt win32k!taghook
+0x000 head          : _THRDESKHEAD
+0x028 phkNext       : Ptr64 tagHOOK
+0x030 iHook         : Int4B
+0x038 offPfn        : Uint8B
+0x040 flags         : Uint4B
+0x044 ihmod         : Int4B
+0x048 ptiHooked     : Ptr64 tagTHREADINFO
+0x050 rpdesk        : Ptr64 tagDESKTOP
+0x058 nTimeout      : Pos 0, 7 Bits
+0x058 fLastHookHung : Pos 7, 1 Bit
```

由于结构十分复杂，看得迷糊的人可以边看本文边用 WINDBG 进行内核调试。接下来，给出实现的代码（代码倒十分简短）：

```
void EnumMsgHook()
{
    int i=0;
    UINT64 pgSharedInfo=0;
    pgSharedInfo = (UINT64)GetProcAddress(GetModuleHandleA("user32.dll"), "gSharedInfo");
    UINT64 phe = GetQWORD(pgSharedInfo+8);
    UINT64 count = GetQWORD(GetQWORD(pgSharedInfo)+8);
    HANDLEENTRY heStruct={0};
    HOOK_INFO Hook={0};
    for(i=0;i<count;i++)
    {
        memcpy(&heStruct, (PVOID)(phe + i*sizeof(HANDLEENTRY)), sizeof(HANDLEENTRY));
        if(heStruct.bType==5)
        {
            RKM(heStruct.phead, &Hook, sizeof(HOOK_INFO));
            printf("hHandle:      0x%llx\n", Hook.hHandle);
            printf("iHookFlags:   %s\n", GetHookFlagString(Hook.iHookFlags));
            printf("iHookType:    %s\n", GetHookType(Hook.iHookType));
            printf("OffPfn:      0x%llx\n", Hook.OffPfn);
            printf("ETHREAD:     0x%llx\n", GetQWORD((UINT64)(Hook.Win32Thread)));
            printf("ProcessName: %s\n\n", GetPNbyET(GetQWORD((UINT64)(Hook.Win32Thread))));
        }
    }
}
```

接下来解析子函数。首先是读取内核内存的函数，此函数可用于安全读写内

核内存（看函数名估计是名为 VxK 的网友写的）：

```

BOOLEAN VxkCopyMemory( PVOID pDestination, PVOID pSourceAddress, SIZE_T SizeOfCopy )
{
    PMDL pMdl = NULL;
    PVOID pSafeAddress = NULL;
    pMdl = IoAllocateMdl( pSourceAddress, (ULONG)SizeOfCopy, FALSE, FALSE, NULL );
    if( !pMdl ) return FALSE;
    __try
    {
        MmProbeAndLockPages( pMdl, KernelMode, IoReadAccess );
    }
    __except( EXCEPTION_EXECUTE_HANDLER )
    {
        IoFreeMdl( pMdl );
        return FALSE;
    }
    pSafeAddress = MmGetSystemAddressForMdlSafe( pMdl, NormalPagePriority );
    if( !pSafeAddress ) return FALSE;
    RtlCopyMemory( pDestination, pSafeAddress, SizeOfCopy );
    MmUnlockPages( pMdl );
    IoFreeMdl( pMdl );
    return TRUE;
}

```

通过判断 phead->iHook 可以获得钩子的类型：

```

char *GetHookType(int Id)
{
    char *string;
    string=(char*)malloc(32);
    switch(Id)
    {
        case -1:
        {
            strcpy(string, "WH_MSGFILTER");
            break;
        }
        case 0:
        {
            strcpy(string, "WH_JOURNALRECORD");
            break;
        }
        case 1:
        {
            strcpy(string, "WH_JOURNALPLAYBACK");

```

```
        break;
    }
    case 2:
    {
        strcpy(string, "WH_KEYBOARD");
        break;
    }
    case 3:
    {
        strcpy(string, "WH_GETMESSAGE");
        break;
    }
    case 4:
    {
        strcpy(string, "WH_CALLWNDPROC");
        break;
    }
    case 5:
    {
        strcpy(string, "WH_CBT");
        break;
    }
    case 6:
    {
        strcpy(string, "WH_SYSMSGFILTER");
        break;
    }
    case 7:
    {
        strcpy(string, "WH_MOUSE");
        break;
    }
    case 8:
    {
        strcpy(string, "WH_HARDWARE");
        break;
    }
    case 9:
    {
        strcpy(string, "WH_DEBUG");
        break;
    }
    case 10:
    {
```

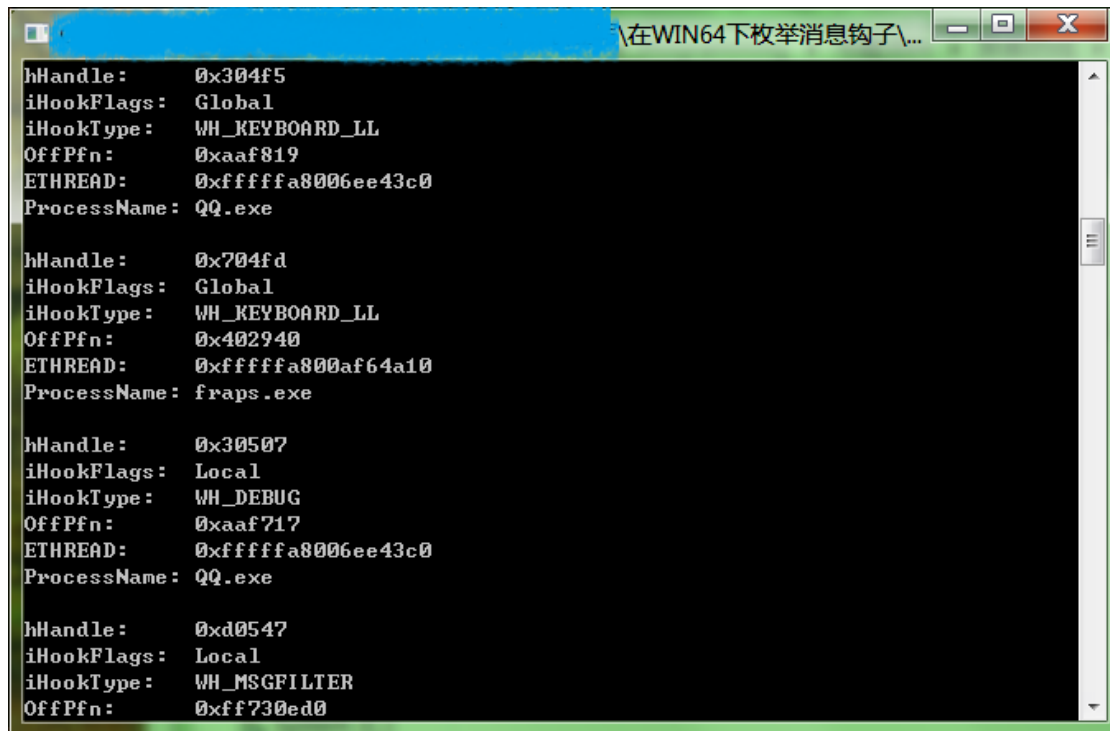
```
        strcpy(string, "WH_SHELL");
        break;
    }
    case 11:
    {
        strcpy(string, "WH_FOREGROUNDIDLE");
        break;
    }
    case 12:
    {
        strcpy(string, "WH_CALLWNDPROCRET");
        break;
    }
    case 13:
    {
        strcpy(string, "WH_KEYBOARD_LL");
        break;
    }
    case 14:
    {
        strcpy(string, "WH_MOUSE_LL");
        break;
    }
    default:
    {
        strcpy(string, "????");
        break;
    }
}
return string;
}
```

通过判断 `phead->flags` 可以获得钩子的标志。钩子标志的意思是，有些钩子只针对当前进程有效，有些钩子针对全局有效。比如 QQ 密码框的消息钩子就是全局钩子：

```
char *GetHookFlagString(int Flag)
{
    char *string;
    string=(char*)malloc(8);
    if(Flag==1 || Flag==3)
        strcpy(string, "Global");
    else
        strcpy(string, "Local");
    return string;
}
```

```
}
```

其它的一些子函数就没有讲解的必要了，最后的效果如下图：



```
hHandle: 0x304f5
iHookFlags: Global
iHookType: WH_KEYBOARD_LL
OffPfn: 0xaaaf819
ETHREAD: 0xfffffa8006ee43c0
ProcessName: QQ.exe

hHandle: 0x704fd
iHookFlags: Global
iHookType: WH_KEYBOARD_LL
OffPfn: 0x402940
ETHREAD: 0xfffffa800af64a10
ProcessName: fraps.exe

hHandle: 0x30507
iHookFlags: Local
iHookType: WH_DEBUG
OffPfn: 0xaaaf717
ETHREAD: 0xfffffa8006ee43c0
ProcessName: QQ.exe

hHandle: 0xd0547
iHookFlags: Local
iHookType: WH_MSGFILTER
OffPfn: 0xff730ed0
```

本文到此结束。示例代码在附件里。