通过 IAT Hook 调试 Windows 自定义未处理异常过滤器①

Debug Windows Custom Filters for Unhandled Exceptions by IAT Hook

吴 标 赵 方 (北京林业大学 信息学院 北京 100083)

摘 要: Windows 自定义未处理异常过滤器在程序崩溃时是查找原因的重要途径，实现了一种使用 IAT(Import Address Table) Hook 改变 Windows 处理未处理异常的流程来调试自定义未处理异常过滤器的代码的方法，简化此部分代码的错误检查工作，此方法兼容性和复用性较高。Hook API 方法采用比较挂钩函数的返回地址来确定其调用函数，改进了普通的 IAT Hook 方法影响整个进程内所有调用的缺点。

关键词: IAT Hook 调试 自定义未处理异常过滤器

1 引言

Windows SEH 为在每个线程启动时都安装了一个最顶层的默认的异常处理程序，这个异常处理程序作为异常帧链表的最后一个节点，专门处理线程的所有内层异常处理程序都未处理的异常，此异常处理程序的异常过滤器部分称为未处理异常过滤器，即 Kernel32.dll 里的 UnhandledExceptionFilter 函数[1]。Windows SEH 可以让用户代码注册一个自定义未处理异常过滤器，然后在 Windows SHE 的未处理异常过滤器调用自定义未处理异常过滤器来让用户代码获得对未处理异常的处理机会，具体通过 SetUnhandledExceptionFilter 函数实现。线程发生了异常却找不到异常处理程序时，程序的控制流转向未处理异常过滤器再转到用户代码自定义未处理异常过滤器，所以自定义未处理异常过滤器部分在一个健壮的程序中必不可少，在程序崩溃时自定义未处理异常过滤器是程序的最后一道关口，在分析程序崩溃原因时自定义未处理异常过滤器部分是重要的技术手段。

而在编写自定义未处理异常过滤器的程序时发现，自定义未处理异常过滤器的代码在程序单独运行的情况下可以执行，而在调试状态下却没有执行，为了解决此问题，本文提出一种使用 IAT Hook 改变 Windows 处理未处理异常的流程的方法，使在调试器下未处理异常程序的控制流能流向自定义未处理异常过滤器。

2 使用自定义未处理异常过滤器的程序框架

写一个简单的程序来追寻此问题的原因，工程名为 DebugCustomFilter，程序代码如下:

```
//自定义未处理异常过滤器函数
LONG WINAPI MyCustomFilter(EXCEPTION_POINTERS* pep){
MessageBox( NULL, "", "MyCustomFilter", MB_OK );
return EXCEPTION_EXECUTE_HANDLER;
}
void main() {
int i, j;
//注册未处理异常过滤器
SetUnhandledExceptionFilter( MyCustomFilter );
//产生除 0 异常
```

① 基金项目:国家科技支撑项目(2006BAD10A03)
收稿时间:2009-03-18

```
j = 0;
i = 1 / j;
}
```

运行程序弹出消息框，但在调试器下在 MyCustomFilter 函数内断点却没有经过，这就是问题所在，自定义未处理异常处理器的代码不能调试会给软件编写带来很大不便。

3 探索原因

使用 Windbg 加载运行此程序，产生 CPU 级的除 0 异常，CPU 在 IDT 中找到对应的异常处理函数入口，异常处理函数调用内核的 KiDispatchException 函数分发异常，KiDispatchException 函数调用 Ntdll.dll 内负责用户态异常分发的 KiUserException Dispatcher 函数[2]，此过程为内核代码 Windbg 用户态调试不能跟踪到，在 Windbg 用户态调试的跟踪从 KiUserxceptionDispatcher 函数内容开始，以后的跟踪流程如下：ntdll!RtlDispatchException==> ntdll!tlpExecuteHandlerForException==>ntdll!ExecuteHandler==>ntdll!ExecuteHandler2==>DebugCustomFilter!_except_handler3(VC 编译器在系统内部封装扩展了了 SEH 机制，每个异常帧的异常处理程序域都指向同一个函数_except_handler3，此函数位于 VC 的运行时库中)==>call eax {DebugCustomFilter!ainCRTtartup+0x167(00402774)}(执行 VC 程序的入口点运行时库函数 mainCRTStartup 保护用户程序代码 Main 函数的异常处理程序的过滤器)==>ntdll!RtlpExecuteHandlerForException==>ntdll!ExecuteHandler==>ntdll!ExecuteHandler2==>DebugCustomFilter!_except_handler3==>call eax {kernel32!BaseProcessStart+0x29(7c843612)}(此为线程启动时最顶层的异常块的过滤器表达式代码地址)==>call kernel32!UnhandledExceptionFilter(7c 862b8a)==>call dword ptr [kernel32!imp__Nt QueryInformationProcess(7c8010ac)]。UnhandledExceptionFilter 函数里调用 NtQueryInformationProcess 函数的部分的反编译代码如图 1 所示。

从图 1 的反编译代码可以看出 UnhandledExceptionFilter 调用 NtQueryInformationProcess 时第二个参数 ProcessInformationClass 的值为 7，即 PROCESSINFOCLASS 枚举类型里的 ProcessDebugPort，此调用的目的是查询进程的调试端口[3]。如果返回值 ProcessInformation 为 0 则表示进程不在调试器下运行。0x7c862c17 和 0x7c862c19 两行的代码在 NtQueryInformationProcess 返回值小于 0(表示出错)时跳转到 0x 7c862cc1，0x7c862c17 和 0x7c862c25 两行代码在 NtQuery Information Process 的第三个参数 Process- Information 所指地址的值(查询到的调试端口)为 0 时(表示进程不在调试器下运行)时跳转到 0x 7c862cc1，观察到整个 UnhandledExceptionFilter 函数代码里只有此处调用了 NtQueryInformation Process，所以 UnhandledExceptionFilter 函数肯定是在此处根据 NtQueryInformationProcess 得到的进程是否在调试器下运行来决定是否调用自定义未处理异常过滤器，所以改变程序流程使自定义未处理异常过滤器在调试器下运行的方法有 3 种：

①修改此处 NtQueryInformationProcess 返回值为负数

②把第三个参数所指地址的值改为 0。

③修改跳转代码(NtQueryInformationProcess 返回值为负数后的跳转代码或者第三个参数所指地址的值为 0 后的跳转代码)。

```
7c862bf9 89bddcfeffff   mov    dword ptr [ebp-124h],edi
7c862bff 57             push   edi
7c862c00 6a04           push   4
7c862c02 8d85dcfeffff   lea    eax,[ebp-124h]
7c862c08 50             push   eax
7c862c09 6a07           push   7
7c862c0b e8fdb3faff     call   kernel32!GetCurrentProcess (7c80e00d)
7c862c10 50             push   eax
7c862c11 ff15ac10807c   call   dword ptr [kernel32!_imp__NtQueryInformationProcess
7c862c17 85c0           test   eax,eax
7c862c19 0f8ca2000000   jl     kernel32!UnhandledExceptionFilter+0x137 (7c862cc1)
7c862c1f 39bddcfeffff   cmp    dword ptr [ebp-124h],edi
7c862c25 0f8496000000   je     kernel32!UnhandledExceptionFilter+0x137 (7c862cc1)
```

图 1 UnhandledExceptionFilter 函数调用 NtQueryInformationProcess 函数部分反编译代码

4 解决问题

以上提出的 3 种方法都可以通过调试器控制和修改进程内存来实现，前两种方法还可以通过 Hook api 实现。

调试器控制的方法较快捷，在调试器下在 UnhandledExceptionFilter 里 NtQueryInformationProcess 返回处断点，程序运行到此处时修改寄存器值或者内存数据代码，但此方法在一些调试器比较难操作，并且每次退出调试器后再进行调试都要操作，虽然一些调试器可以把此部分工作写成脚本增加其可用性和复用性，但是这种脚本只能在对应的调试器下运行，不能兼容其他调试器，有一定的局限性，所以调试器控制的方法不能作为一种通用的方法来使用。

修改进程内存的方法也比较容易实现，该方法在二进制代码级别上处理，使用 ReadProcessMemory 和 WriteProcessMemory 实现，因为各个版本的 Windows操作系统 UnhandledExceptionFilter 的二进制代码都有可能不一样(甚至同一版本不同补丁的都不一样)，所以此方法很难做到兼容各个版本的 Windows 操作系统，除此以外此方法的可理解性也很差。

要想做到兼容各个版本的 Windows 操作系统，可以采用修改 IAT Hook API 方法来修改 UnhandledExceptionFilter 的返回值，但此方法存在一个弊病，Hook API 影响整个进程，被 Hook 的函数在其他地方的调用也会被修改，此缺点可以通过在 NtQueryInformationProcess 的 Hook 函数里判断函数的返回地址是否和 UnhandledExceptionFilter 函数内调用 NtQueryInformationProcess 函数的指令的下一个指令的地址相等来确定，调用 NtQueryInformationProcess 函数的指令的下一个指令的地址可以通过在 UnhandledExceptionFilter 函数地址后搜索 Call dword ptr [NtQueryInformationProcess 的 IAT 项地址]再加上指令长度 6 得到（此方法虽然属于硬编码方法，但是由于 Windows 模块间函数调用都是用 IAT 方式实现，所以此方法并不存在兼容性问题。

Hook 过程分 6 步：

①获取 UnhandledExceptionFilter 函数的地址。

②获取 NtQueryInformationProcess 函数的地址。

③使用 NtQueryInformationProcess 函数的地址来获取其在 Kernel32.dll 输入表里的 IAT 项地址。

④获取 UnhandledExceptionFilter 函数里调用 NtQueryInformationProcess 函数的指令的下一个指令的地址(在 UnhandledExceptionFilter 函数地址后搜索 Call dword ptr [NtQueryInformation Process 的 IAT 项地址]指令的地址再加上指令长度 6)。

⑤修改 Kernel32.dll 里名为 Ntdll.dll 的输入表的 NtQueryInformationProcess 函数对应的 IAT 项里的函数地址为我们的 Hook 函数地址。

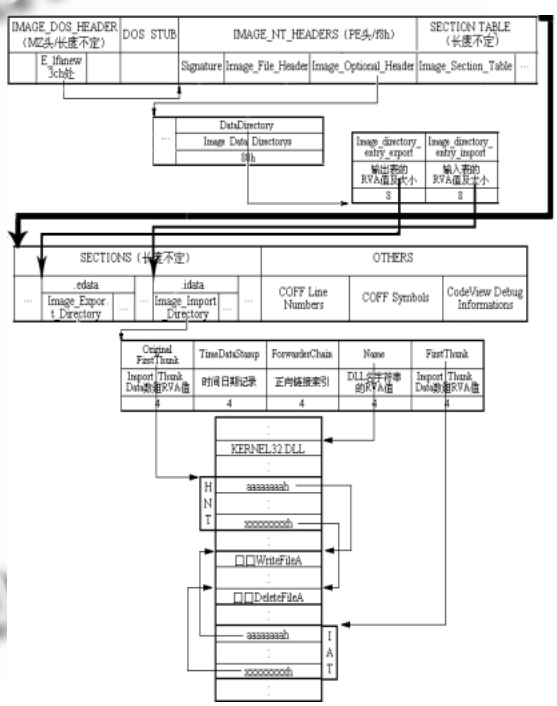

图 2 简略的 PE 文件结构图[4]

⑥Hook 函数里先调用系统原来的 NtQueryInformationProcess 函数并得到其返回值，再判断函数的返回地址是否和 UnhandledExceptionFilter 函数内调用 NtQueryInformationProcess 函数的指令的下一个指令的地址相等来确定本次调用是否来自 UnhandledExceptionFilter 函数，是则修改 NtQueryInformationProcess 的返回值为-1 并复原 Kernel32.dll 的 IAT 表 Unhook NtQueryInforma-

tionProcess 函数，最后返回-1。否则返回 NtQueryInformationProcess 函数的返回值。

本文内容涉及的 PE 文件结构如图 2 所示(略去了与本文无关的部分，最粗黑线箭头表示 SECTION_TABLE 和 SECTIONS 是相连的)。

关键代码分析(部分伪码)：

```
#define CALL_DWORD_PTR_IAT_LEN 6
DWORD
g_dwCall_dword_ptr_IAT_Next_Addr;
void main() {
char *pCall_dword_ptr_IAT = new char [6];
//call dword ptr 的机器码为 Oxff15
pCall_dword_ptr_IAT [0] = 0xff;
pCall_dword_ptr_IAT [1] = 0x15;
//注册未处理异常过滤器
SetUnhandledExceptionFilter( MyCustomFilter );
//获得 UnhandledExceptionFilter 函数地址
GetApiAddr("Kernel32.Dll",
"UnhandledExceptionFilter", &dwUnhandledExceptionFilter_Addr);
//获得 NtQueryInformationProcess 函数地址
GetApiAddr("Ntdll.Dll",
"NtQueryInformationProcess", &g_dwNtQueryInformationProcess_Addr);
//使用 NtQueryInformationProcess 函数的地址来获取其在 Kernel32.dll 输入表里的 IAT 项地址
GetApi_IAT_Addr("Kernel32.dll", "Ntdll.dll",
g_dwNtQueryInformationProcess_Addr,
&dwNtQueryInformationProcess_IAT_Addr);
//构造 call dword ptr [NtQueryInformationProcess 的 IAT 项地址]指令的机器码
memcpy(pCall_dword_ptr_IAT+2, &dwNtQueryInformationProcess_IAT_Addr, 4);
//在 UnhandledExceptionFilter 函数地址后搜索 Call dword ptr [NtQueryInformationProcess 的 IAT 项地址]指令
dwCall_dword_ptr_IAT_Addr=Search_Call_dword_ptr_IAT
(dwUnhandledExceptionFilter_Addr,pCall_dword_ptr_IAT, 6);
//下一个指令的地址，也就是 UnhandledExceptionFilter 调用 NtQueryInformationProcess 的返回地址
g_dwCall_dword_ptr_IAT_Next_Addr=dwCall_dword_ptr_IAT_Addr CALL_DWORD_PTR_ IAT_LEN;
//修改 IAT Hook
HookApi("Kernel32.dll", "Ntdll.dll", g_dwNtQueryInformationProcess_Addr,(DWORD)Hook_NtQueryInformationProcess);
// 产生除 0 异常，促发 SEH 调用 UnhandledExceptionFilter
j = 0;
i = 1 / j;
}
BOOL GetApiAddr( LPCSTR lpszDll, LPCSTR lpszFunc, DWORD *pApiAddr ) {
HINSTANCE hInstance=GetModuleHandle(lpszDll);
//使用 GetProcAddress 函数获得函数地址不但方便快捷，而且不用声明函数原型和引用 lib 文件(Windows 未公开 API 的函数原型和 lib 文件难以获得)
*pApiAddr=(DWORD)GetProcAddress(hInstance, lpszFunc);
}
// Hook 函数
DWORD Hook_NtQueryInformationProcess(...) {
/*取返回地址
调用函数时，先 push 参数再 push 返回地址，然后编译器在每个函数内容前加入 push ebp 和 mov ebp,esp 来建立该函数自己的栈顶，所以该函数的返回地址存放在其内部栈顶往下 4 个字节处*/
DWORD calleraddr;
```

```
_asm mov eax,[ebp+4]
_asm mov calleraddr,eax
//调用 Windows 的 NtQueryInformationProcess
dwResult=func_NtQueryInformationProcess(...);
//判断本次调用是否来自 UnhandledExceptionFilter 函数
if(calleraddr==g_dwCall_dword_ptr_IAT_Next_Addr){
dwResult = -1;
//还原 IAT 表，取消 Hook
UnHookApi(...);
}
return dwResult;
}
```

5 测试结果

本文代码在 Windows 2000 SP4、Windows XPSP2、Windows Server 2003 SP2，3 个版本的操作系统下都能调试自定义未处理异常过滤器。

6 结语

本文实现了一种使用 IAT Hook 改变 Windows 处理未处理异常的流程的方法来调试自定义未处理异常过滤器部分的代码，此方法避免了二进制代码级别处理的硬编码，兼容性较高，并且通过包含代码的方式来使用，易用性和复用性也较高。

参考文献

1 Pietrek M. A Crash Course on the Depths of Win32 Structured Exception Handling.[1997-01]http://www.microsoft. com/msj/0197/Exception/Exception.aspx

2 张银奎.软件调试.北京:电子工业出版社，2008. 284－290.

3 Nowak T. PROCESS_INFORMATION_CLASS.[2001-3-6]http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/PROCESS_INFORMATION_CLASS.html.

4 Rgbsky. PE 格式结构图. http://www.pediy.com/document/PE