

# 对于结构化异常处理（SEH）的进一步探索



本文关键字: SEH, \_except\_handler3, Windows, VisualC



写本文的起因

Matt Pietrek 关于结构化异常处理的文章 [A Crash Course on the Depths of Win32 Structured Exception Handling](#) 是一篇很棒的文章（在本文末的“相关文章”中有我整理过的中文版文章链接），对于我了解 Win32 下的这种异常处理机制帮助很大。在仔细阅读完整篇文章、跟踪过相关代码的执行流程后，仍有意犹未尽的感觉。本文就是在读过这篇文章之后写的，具有一定“实验记录”的性质。所以强烈建议在阅读本文之前看一下 Matt Pietrek 的文章。

在所有被编译器封装的异常处理行为中，最重要的恐怕就要数 \_except\_handler3 和 \_\_local\_unwind2 函数了。在这两者之间，又数 \_except\_handler3 尤为重要。因为这个函数是操作系统与编译器之间的接口，操作系统只能按照栈帧结构一个一个地调用 handler，而真正的 filter 调用，以及同一个函数内的嵌套 \_\_try 的处理都要依赖 \_except\_handler3 来完成。因此，认真研究一下这个函数的实现，对于完全理解 VC 下的 SEH 机制就显得十分必要。



对 \_except\_handler3 伪码的勘误

Matt Pietrek 在他的文章中已经给出了 \_except\_handler3() 的伪代码，但不知细心的人是否发现了伪码中的错误，当然，我并不是在计较那两个把“=”写成“==”的无聊笔误，而是另外一个小逻辑错误。让我们再仔细看一下文中给出的伪码：

```
001: int __except_handler3(
002:     struct _EXCEPTION_RECORD * pExceptionRecord,
003:     struct _EXCEPTION_REGISTRATION * pRegistrationFrame,
004:     struct _CONTEXT * pContextRecord,
005:     void * pDispatcherContext )
006: {
007:     LONG filterFuncRet
008:     LONG trylevel
009:     EXCEPTION_POINTERS exceptPtrs
010:     PSCOPETABLE pScopeTable
011:
012:     CLD    // Clear the direction flag (make no assumptions!)
013:
014:     // if neither the EXCEPTION_UNWINDING nor
    EXCEPTION_EXIT_UNWIND bit
015:     // is set... This is true the first time through the handler (the
016:     // non-unwinding case)
017:
018:     if ( ! (pExceptionRecord->ExceptionFlags
019:             & (EXCEPTION_UNWINDING |
020:                EXCEPTION_EXIT_UNWIND)) )
021:     {
022:         // Build the EXCEPTION_POINTERS structure on the stack
023:         exceptPtrs.ExceptionRecord = pExceptionRecord;
024:         exceptPtrs.ContextRecord = pContextRecord;
025:
026:         // Put the pointer to the EXCEPTION_POINTERS 4 bytes below
    the
```

```

026:    // establisher frame. See ASM code for GetExceptionInformation
027:    *(PDWORD)((PBYTE)pRegistrationFrame - 4) = &exceptPtrs;
028:
029:    // Get initial "trylevel" value
030:    trylevel = pRegistrationFrame->trylevel
031:
032:    // Get a pointer to the scopetable array
033:    scopeTable = pRegistrationFrame->scopetable;
034:
035: search_for_handler:
036:
037:    if ( pRegistrationFrame->trylevel != TRYLEVEL_NONE )
038:    {
039:        if ( pRegistrationFrame->scopetable[trylevel].lpfnFilter )
040:        {
041:            PUSH EBP                // Save this frame EBP
042:
043:            // !!!Very Important!!! Switch to original EBP. This is
044:            // what allows all locals in the frame to have the same
045:            // value as before the exception occurred.
046:            EBP = &pRegistrationFrame->_ebp
047:
048:            // Call the filter function
049:            filterFuncRet = scopetable[trylevel].lpfnFilter();
050:
051:            POP EBP                // Restore handler frame EBP
052:
053:            if ( filterFuncRet != EXCEPTION_CONTINUE_SEARCH )
054:            {
055:                if ( filterFuncRet < 0 ) //
EXCEPTION_CONTINUE_EXECUTION
056:                    return ExceptionContinueExecution;
057:
058:                // If we get here, EXCEPTION_EXECUTE_HANDLER was
specified
059:                scopetable == pRegistrationFrame->scopetable
060:
061:                // Does the actual OS cleanup of registration frames
062:                // Causes this function to recurse
063:                __global_unwind2( pRegistrationFrame );
064:
065:                // Once we get here, everything is all cleaned up, except
066:                // for the last frame, where we'll continue execution
067:                EBP = &pRegistrationFrame->_ebp
068:
069:                __local_unwind2( pRegistrationFrame, trylevel );
070:
071:                // NLG == "non-local-goto" (setjmp/longjmp stuff)
072:                __NLG_Notify( 1 ); // EAX == scopetable->lpfnHandler

```

```

073:
074:         // Set the current trylevel to whatever SCOPETABLE entry
075:         // was being used when a handler was found
076:         pRegistrationFrame->trylevel =
077:         scopetable->previousTryLevel;
078:
079:         // Call the _except {} block. Never returns.
080:         pRegistrationFrame->scopetable[trylevel].lpfnHandler();
081:     }
082:
083:     scopeTable = pRegistrationFrame->scopetable;
084:     trylevel = scopeTable->previousTryLevel
085:
086:     goto search_for_handler;
087: }
088: else // trylevel == TRYLEVEL_NONE
089: {
090:     retvalue == DISPOSITION_CONTINUE_SEARCH;
091: }
092: }
093: else // EXCEPTION_UNWINDING or EXCEPTION_EXIT_UNWIND
094: flags are set
095: {
096:     PUSH EBP // Save EBP
097:     EBP = pRegistrationFrame->_ebp // Set EBP for __local_unwind2
098:     __local_unwind2( pRegistrationFrame, TRYLEVEL_NONE )
099:
100:     POP EBP // Restore EBP
101:
102:     retvalue == DISPOSITION_CONTINUE_SEARCH;
103: }
104: }

```

注意第 37 行的 if 语句，对当前 pRegistrationFrame 中的 trylevel 进行了判定：

```
if ( pRegistrationFrame->trylevel != TRYLEVEL_NONE )
```

也就是说，如果已经没有 try 块了，就直接返回 DISPOSITION\_CONTINUE\_SEARCH，然后操作系统会调用下一个栈帧的 handler。但这里的判定写错了，显然应该是：

```
if ( trylevel != TRYLEVEL_NONE )
```

因为在没有 filter 或者 filter 不处理异常的情况下，第 84 行的赋值将会回溯到外层 \_\_try 块的 scopetable\_entry 结构中：

```
trylevel = scopeTable->previousTryLevel
```

如果不进行这样的修改，就不难想象这个 handler 会导致怎样的后果：异常发生后，

pRegistrationFrame->trylevel 指示了异常发生的地方所处的 \_\_try 块在 scopetable 中的索引。如果这一层的 filter 没有处理这个异常，那么在这个 handler 中将会执行到第 86 行，也就是说准备开始执行外层 \_\_try 块的 filter。此时 pRegistrationFrame->trylevel 的值并没有任何改变，也就是说，无论外层是否有 \_\_try 块，第 37 行的判定一定可以再次通过，但由于 84 行的赋值操作已经在上一个循环中更新了 trylevel 变量的值，trylevel 的值就有可能是 -1 了（也就是说该帧内没有人处理这个异常），在这种情况下，第 39 行的判定一定会引发一个 Access Violation 异常，因为 trylevel 作为数组下标是一个非法值。而且不难预见的是：除非该帧内没有一个 \_\_try，或者某个 filter 处理了异常，否则这个 handler 肯定是次要在这里摔跟头的。并且，这个跟头摔的不算轻：这属于在一个异常 handler 中引发了另一个异常（传说中的 double fault?），这个异常会被系统函数 RtlpExecuteHandlerForException 安装的简易 handler 处理（Matt Pietrek 在他的文章中提到过这一点，参看“Into the Inferno”一节），处理结果就是返回 DISPOSITION\_NESTED\_EXCEPTION，然后给这个异常打上一个“异常嵌套了!!!”的标志（Matt Pietrek 提供的伪码中写的是 EH\_EXIT\_UNWIND，虽然乍看上去他的伪码似乎更合理一些，但是却与实际情况不符，我将在后面提到这一点）。

另外，伪码中第 96 行似乎少了一个非常关键的取地址符“&”，但我相信这是另一个笔误罢了，因为前面第 46 行的赋值表达式是正确的。但是，我刚刚看到这里的时候却没有现在这么清楚，我曾经为判断这两种写法哪个是正确的、哪个是错误的而迷惑了一段时间。之后，在跟踪了 VC 构造异常帧的代码后我终于意识到：46 行的那一句才是正确的。再后来，当我在跟踪 \_except\_handler3 的代码时，无意间发现了 VC 内部真正的 \_EH3\_EXCEPTION\_REGISTRATION 结构的定义才知道：CRT 源码文件 EXSUP.INC 中的那个 \_EXCEPTION\_REGISTRATION 结构的定义实在是太迷惑人了，尤其是那个“\_ebp”成员；而另一个 EXCEPT.INC 文件中用汇编语言给出的 \_\_EXCEPTIONREGISTRATIONRECORD 定义更是胡扯。当然，这些问题一会儿再说，现在先回到 \_except\_handler3。

我是个好奇心很强的人，发现了伪码中的错误以后不禁觉得有些兴奋（众人语：这什么人嘛！），并且想到了另一个问题：Matt Pietrek 是怎么写出这些伪代码的？如果说这个错误是他不小心犯的，那么他有没有犯别的错误——他犯错误我管不着，但是如果我跟着学坏了，那岂不是很冤……所以，我决定自己去看看 \_except\_handler3 的代码究竟是什么样子的。

### 激动人心的旅程

我知道作出这个决定后肯定要经历一个痛苦的过程，但我仍然义无反顾地开了 VC，然后一头扎进机器语言的茫茫大雾中……

想看到 \_except\_handler3，就要先抓住它；想抓住它，就要先引发一个异常。这个好办，几行程序就可以把它引出来：

```
1: __try {
2:     int *p = 0;
3:     *p = 0;
4: } __except(EXCEPTION_EXECUTE_HANDLER) {
5: }
```

在第 3 行设断点，然后切换到反汇编，就看到了这样的景象：

```
01: __try {
02: 00411A4B mov     dword ptr [ebp-4],0
03:     int *p = 0;
04: 00411A52 mov     dword ptr [p],0
05:     *p = 0;
06: 00411A59 mov     eax,dword ptr [p]
07: 00411A5C mov     dword ptr [eax],0
08: 00411A62 mov     dword ptr [ebp-4],0FFFFFFFFh
09: 00411A69 jmp     $L28580+0Ah (411A7Bh)
```

```

10: } _except(EXCEPTION_EXECUTE_HANDLER) {
11: 00411A6B mov     eax,1
12: $L28581:
13: 00411A70 ret
14: $L28580:
15: 00411A71 mov     esp,dword ptr [ebp-18h]
16: 00411A74 mov     dword ptr [ebp-4],0FFFFFFFFh
17: }

```

啊，在明白了大部分事情之后，一切显得都是那么的自然：第 2 行的指令不就是在设置那个“传说中的”trylevel 么？呵呵，基址后的第一个 DWORD 就是，果然不错。AV 异常显然应该在第 7 行发生，step into 那一行，却发现：VC 在输出窗口中显示有异常发生，然后直接停在了 15 行，也就是 handler 代码开始的地方。这不是我想要的结果，因为据我所知，异常发生后，会产生一大堆系统调用，最后由 \_except\_handler3 把控制权交回我写的 handler。换句话说，当进入我的 handler 代码时，这一切都已经结束了……

既然 VC 不愿意让我这么容易地看到 \_except\_handler3 的代码，那么我也就不得不耍点手段了，于是我盯上了 11、13 行的 filter 指令。是的，这应该就是 filter 的代码，如果有人 CALL 到 11 行，那么这行指令会将 eax 置为 1，然后在第 13 行返回，也就是返回 1，根据 EXCPT.H 中的宏定义，1 就是 EXCEPTION\_EXECUTE\_HANDLER 的值，所以这正是我的 filter-expression 的行为，这就是我的 filter 代码。那么，如果是 \_except\_handler3 调用了 filter，那么我在 filter 返回之前中断，是不是就能跟回到我梦寐以求的 \_except\_handler3 中去了呢？是的，当我在第 13 行设断点、step over 之后，VC 终于老老实实在把我带回了 \_except\_handler3 家。

好在 \_except\_handler3 的代码不多，更何况我之前已经看过了伪码，所以想弄懂这些指令在做什么并不是件很难的事。首先我意识到必须先弄到它的定义，否则看那一大堆相对于 ebp 寄存器的偏移肯定不是件多么舒服的事。好在 Matt Pietrek 已经在他的文章中提到了，EXCPT.H 中包含了这个函数定义：

```

1: EXCEPTION_DISPOSITION
2: __cdecl _except_handler(
3:   struct _EXCEPTION_RECORD *ExceptionRecord,
4:   void *EstablisherFrame,
5:   struct _CONTEXT *ContextRecord,
6:   void *DispatcherContext
7: );

```

虽然这个定义中的函数名是 \_except\_handler 而非 \_except\_handler3，但估计也就是一个 Place Holder。因为我已经尝试过直接在代码中显示调用这个函数名了，但是 Link 不上，所以名字不一样也无所谓了。根据这个定义，可以得出结论：这是一个 \_\_cdecl 调用约定的函数，4 个参数从右至左入栈，调用者负责清理堆栈。因此：指令中出现的 [ebp+8] 引用的是 ExceptionRecord、[ebp+0Ch] 引用的是 EstablisherFrame、[ebp+10h] 引用的是 ContextRecord、[ebp+14h] 引用的是 DispatcherContext，函数返回使用 ret 而非 \_\_stdcall 的函数常用的“ret N”。好了，有了这些信息，分析起来就容易多了：

```

001: _except_handler3:
002: 004141A0 push    ebp
003: 004141A1 mov     ebp,esp
004:   ; // EXCEPTION_POINTERS exceptPtrs;
005: 004141A3 sub     esp,8
006: 004141A6 push    ebx
007: 004141A7 push    esi
008: 004141A8 push    edi

```

```

009: 004141A9 push    ebp
010: 004141AA cld
011:    ; // EstablisherFrame => ebx
012: 004141AB mov     ebx,dword ptr [ebp+0Ch]
013:    ; // ExceptionRecord => eax
014: 004141AE mov     eax,dword ptr [ebp+8]
015:    ; // if (ExceptionRecord->ExceptionFlags &
    EXCEPTION_UNWIND_CONTEXT)
016:    ; //    goto _lh_unwinding;
017: 004141B1 test    dword ptr [eax+4],6
018: 004141B8 jne     _lh_unwinding (414269h)
019:    ; // exceptPtrs.ExceptionRecord = ExceptionRecord
020: 004141BE mov     dword ptr [ebp-8],eax
021:    ; // exceptPtrs.ContextRecord = ContextRecord;
022: 004141C1 mov     eax,dword ptr [ebp+10h]
023: 004141C4 mov     dword ptr [ebp-4],eax
024:    ; // *(PDWORD)((PBYTE)EstablisherFrame - 4) = &exceptPtrs
025: 004141C7 lea     eax,[ebp-8]
026: 004141CA mov     dword ptr [ebx-4],eax
027:    ; // EstablisherFrame->trylevel => esi
028: 004141CD mov     esi,dword ptr [ebx+0Ch]
029:    ; // EstablisherFrame->scopetable => edi
030: 004141D0 mov     edi,dword ptr [ebx+8]
031:    ; // if (_ValidateEH3RN(EablisherFrame) == 0)
032:    ; //    goto _lh_abort;
033: 004141D3 push    ebx
034: 004141D4 call    @ILT+775(__ValidateEH3RN) (41130Ch)
035: 004141D9 add     esp,4
036: 004141DC or     eax,eax
037: 004141DE je     _lh_abort (41425Bh)
038: _lh_top:
039:    ; // if (trylevel == TRYLEVEL_NONE)
040:    ; //    goto _lh_bagit;
041: 004141E0 cmp     esi,0FFFFFFFFh
042: 004141E3 je     _lh_bagit (414262h)
043:    ; // EstablisherFrame->scopetable[trylevel].lpfnFilter => eax
044: 004141E5 lea     ecx,[esi+esi*2]
045: 004141E8 mov     eax,dword ptr [edi+ecx*4+4]
046:    ; // if (EstablisherFrame->scopetable[trylevel].lpfnFilter == NULL)
047:    ; //    goto _lh_continue;
048: 004141EC or     eax,eax
049: 004141EE je     _lh_continue (414249h)
050:    ; // PUSH EBP
051: 004141F0 push    esi
052: 004141F1 push    ebp
053:    ; // EBP = &EstablisherFrame->_ebp
054: 004141F2 lea     ebp,[ebx+10h]
055:    ; // ret = EstablisherFrame->scopetable[trylevel].lpfnFilter();
056: 004141F5 xor     ebx,ebx

```

```

057: 004141F7 xor     ecx,ecx
058: 004141F9 xor     edx,edx
059: 004141FB xor     esi,esi
060: 004141FD xor     edi,edi
061: 004141FF call    eax
062:    ; // POP EBP
063: 00414201 pop     ebp
064: 00414202 pop     esi
065:    ; // EstabliherFrame => ebx
066: 00414203 mov     ebx,dword ptr [ebp+0Ch]
067:    ; // if (ret == EXCEPTION_CONTINUE_SEARCH)
068:    ; //     goto _lh_continue;
069:    ; // else if (ret < 0)
070:    ; //     goto _lh_dismiss;
071: 00414206 or      eax,eax
072: 00414208 je      _lh_continue (414249h)
073: 0041420A js      _lh_dismiss (414254h)
074:    ; // __global_unwind2(EstabliherFrame);
075: 0041420C mov     edi,dword ptr [ebx+8]
076: 0041420F push    ebx
077: 00414210 call    @ILT+700(__global_unwind2) (4112C1h)
078: 00414215 add     esp,4
079:    ; // EBP = &EstabliherFrame->_ebp
080: 00414218 lea     ebp,[ebx+10h]
081:    ; // __local_unwind2(EstabliherFrame, trylevel);
082: 0041421B push    esi
083: 0041421C push    ebx
084: 0041421D call    @ILT+385(__local_unwind2) (411186h)
085: 00414222 add     esp,8
086:    ; // __NLG_Notify(1);
087: 00414225 lea     ecx,[esi+esi*2]
088: 00414228 push    1
089: 0041422A mov     eax,dword ptr [edi+ecx*4+8]
090: 0041422E call    @ILT+1045(__NLG_Notify) (41141Ah)
091:    ; // EstabliherFrame->trylevel =
092:    ; //     EstabliherFrame->scopetable[trylevel].previousTryLevel
093: 00414233 mov     eax,dword ptr [edi+ecx*4]
094: 00414236 mov     dword ptr [ebx+0Ch],eax
095:    ; // EstabliherFrame->scopetable[trylevel].lpfnHandler();
096: 00414239 mov     eax,dword ptr [edi+ecx*4+8]
097: 0041423D xor     ebx,ebx
098: 0041423F xor     ecx,ecx
099: 00414241 xor     edx,edx
100: 00414243 xor     esi,esi
101: 00414245 xor     edi,edi
102: 00414247 call    eax
103: _lh_continue:
104:    ; // EstabliherFrame->scopetable[trylevel].previousTryLevel => esi
105: 00414249 mov     edi,dword ptr [ebx+8]

```



```

106: 0041424C lea      ecx,[esi+esi*2]
107: 0041424F mov      esi,dword ptr [edi+ecx*4]
108: 00414252 jmp      _lh_top (4141E0h)
109: _lh_dismiss:
110:      ; // return ExceptionContinueExecution;
111: 00414254 mov      eax,0
112: 00414259 jmp      _lh_return (41427Eh)
113: _lh_abort:
114:      ; // ExceptionRecord->ExceptionFlags |=
EXCEPTION_STACK_INVALID;
115: 0041425B mov      eax,dword ptr [ebp+8]
116: 0041425E or       dword ptr [eax+4],8
117: _lh_bagit:
118:      ; // return ExceptionContinueSearch;
119: 00414262 mov      eax,1
120: 00414267 jmp      _lh_return (41427Eh)
121: _lh_unwinding:
122:      ; // PUSH EBP
123: 00414269 push     ebp
124:      ; // EBP = &EstablisherFrame->_ebp
125: 0041426A lea      ebp,[ebx+10h]
126:      ; // __local_unwind2(EstablisherFrame, TRYLEVEL_NONE);
127: 0041426D push     0FFFFFFFh
128: 0041426F push     ebx
129: 00414270 call     @ILT+385(__local_unwind2) (411186h)
130: 00414275 add      esp,8
131:      ; // POP EBP
132: 00414278 pop      ebp
133:      ; // return ExceptionContinueSearch;
134: 00414279 mov      eax,1
135: _lh_return:
136: 0041427E pop      ebp
137: 0041427F pop      edi
138: 00414280 pop      esi
139: 00414281 pop      ebx
140: 00414282 mov      esp,ebp
141: 00414284 pop      ebp
142: 00414285 ret

```

好了，我已经在指令前插入了 C 语句，现在 `_except_handler3` 对于我来说已经没有任何神秘之处了。说点题外话：我发现如果把这些语句提取出来、组成伪码的话，与 Matt Pietrek 的伪码将会非常的像，如果说代码结构方面有相似性也就罢了——毕竟牛人写出来的东西一般都很靠谱的，但是像变量的赋值顺序、指令流的走向、甚至 `CLD` 指令这样的小地方都一样。不知道他是不是也是用跟踪反汇编的方法写出的那些伪代码？真想问问他本人……

不难发现，Matt Pietrek 没有在他的文章中提到第 31、32 行的代码（也就是反汇编第 33 至 37 行间的指令），这段代码调用了另一个函数并检查返回值，如果返回 0，handler 的指令流就会跳转到 `_lh_abort` 处：给异常打上一个“`EXCEPTION_STACK_INVALID`”的标志位（or 上了一个 8，也就是 `EXSUP_INC` 中定义的 `EXCEPTION_STACK_INVALID` 的值）然后立即返回。根据这个函数符号名中

“Validate”的含义、以及 `_except_handler3` 发现其返回 0 后神经质般的举动可以判断——这个函数执行的是对栈帧指针的合法性检查。这种检查可以说在整个异常处理过程中并不鲜见，`Rtl` 函数里经常



进行这样的检查，什么是否上下越界、是否 DWORD 对齐什么的……在这里出现也并不稀奇。我也没有对这个函数做深入研究，只是跟进去看了一眼，但是却有了意外的发现。

### \_EH3\_EXCEPTION\_REGISTRATION 结构的本来面目

到目前为止，VC 中的 EXCEPTION\_REGISTRATION 出现了两个版本。一个是 EXSUP.INC 中的定义，也就是 Matt Pietrek 使用的那个版本；另一个是我自己找到的 EXCEPT.INC 中的版本，是这样定义的：

```
__EXCEPTIONREGISTRATIONRECORD struc
    prev_structure      dd    ?
    ExceptionHandler     dd    ?
    ExceptionFilter      dd    ?
    FilterFrame         dd    ?
    PExceptionInfoPtrs  dd    ?
__EXCEPTIONREGISTRATIONRECORD ends
```

可是我在前面说过，这个定义简直就是胡扯。因为，可以肯定的是：这个结构中的 ExceptionFilter 就是 scopetable 指针，FilterFrame 就是当前的 trylevel。那么 PExceptionInfoPtrs 是什么？从名字上判断，这个就是指向 EXCEPTION\_POINTERS 结构的指针。这个指针应该在这个位置出现吗？NO，这明明就是 \_ebp 的位置嘛……所以我不知道这是一个在什么地方用到的结构。那么，在 EXSUP.INC 的注释中定义的 \_EXCEPTION\_REGISTRATION 就没有问题吗？答案仍然是否定的：

```
; struct _EXCEPTION_REGISTRATION {
;     struct _EXCEPTION_REGISTRATION *prev;
;     void (*handler)(PEXCEPTION_RECORD,
;                     PEXCEPTION_REGISTRATION,
;                     PCONTEXT,
;                     PEXCEPTION_RECORD);
;     struct scopetable_entry *scopetable;
;     int trylevel;
;     int _ebp;
;     PEXCEPTION_POINTERS xpointers;
; };
```

如果说这个结构中的 \_ebp 成员还勉强说得过去的话，那么 xpointers 成员简直就是匪夷所思。因为据我所知，在堆栈中，\_ebp 下存放的是 CALL 指令压入的返回地址，而不是什么 PEXCEPTION\_POINTERS。一下子怀疑这么多问题，即怀疑 CRT 的汇编定义、又怀疑牛人的教导？是不是有点儿过分了……是的，我也觉得挺过分，但是我仍然坚持我的观点，因为我有事实替我说话。

在前面我提到过，我曾经跟踪了 VC 构造异常帧的代码，也就是在函数起始处由编译器自动生成的准备代码（Matt Pietrek 所说的 prologue code），现在就回过头来仔细看看编译器到底在堆栈上干了些什么：

```
01: 00411A10 push    ebp
02: 00411A11 mov     ebp,esp
03: 00411A13 push    0FFFFFFFh
04: 00411A15 push    424020h
05: 00411A1A push    offset @ILT+365(__except_handler3) (411172h)
06: 00411A1F mov     eax,dword ptr fs:[00000000h]
07: 00411A25 push    eax
08: 00411A26 mov     dword ptr fs:[0],esp
09: 00411A2D add     esp,0FFFFFF2Ch
```

```
10: 00411A33 push    ebx
11: 00411A34 push    esi
12: 00411A35 push    edi
13: 00411A36 lea     edi,[ebp-0E4h]
14: 00411A3C mov     ecx,33h
15: 00411A41 mov     eax,0CCCCCCCCh
16: 00411A46 rep stos  dword ptr [edi]
17: 00411A48 mov     dword ptr [ebp-18h],esp
```

那么，当这段指令执行完毕后，堆栈应该是这个样子的：

00000000 →		低地址，栈顶
		.....
esp →	edi	12: push edi
	esi	11: push esi
	ebx	10: push ebx
ebp-0E4h →	.....	
	204 个字节全部填充为 0CCCCCCCCh	
ebp-01Ch →	.....	09: add esp,0FFFFFFF2Ch
ebp-18h →	prologue code 执行完成后的 esp	
	?	
异常帧 →	之前的异常帧 FS:[0]	07: push fs:[0]
	__except_handler3 的地址	05: push __except_handler3
	424020h	04: push 424020h
ebp-4 →	0FFFFFFFFh	03: push 0FFFFFFFFh
ebp →	调用者的基址 ebp	01: push ebp
	CALL 指令压入的返回地址	
		.....
FFFFFFFF →		高地址，栈底

表格的第一列是 DWORD 数据单元的地址，第二列是堆栈中的内容，第三列是影响到 esp 的指令。

根据先前的理解，第 7 条指令执行完成后，异常帧结构就已经在堆栈上构造完成了，并且当前的栈顶指针 esp 所指的地址正是这个结构的首址，第 8 条指令就是把这个地址装入 FS:[0]，做为新的异常 handler 链表的表头。那么，这个异常帧的结构此时就可以确定下来了。这时候再把上面提出的那两个异常帧结构套上去看看，怎么就觉得都不太对劲呢？第一个结构的 PExceptionInfoPtrs 成员对应到了保存的 ebp 的位置上，而第二个结构的 xpointers 成员所对应的数据就更离谱了——居然是返回地址？！

说一下表格中那个问号：为什么是问号呢？因为那个 DWORD 没有经过初始化。那么，为什么不初始化它呢？因为目前不知道该用什么值初始化它，也没有必要初始化它。是什么东西这么邪乎？其实，这个 DWORD 就是 \_\_except\_handler3 中的表达式 ((PBYTE)EstablisherFrame - 4) 引用到的那个 DWORD，也就是 EXCEPTION\_POINTERS 结构的地址。回想一下 \_\_except\_handler3 的代码就可以意识到：EXCEPTION\_POINTERS 结构是建立在 \_\_except\_handler3 堆栈上的临时变量，换句话说，这个结构的地址也只有在 \_\_except\_handler3 执行期间、也就是说有异常发生的时候才有意义。那么，目前我们显然拿

这个 DWORD 没有办法，由它去吧。

至此，可以得出结论：PEXCEPTION\_POINTERS 存放在异常帧地址前的那个 DWORD 中，如果硬要把它“塞”到结构中，那也要放在 prev 的前面，怎么也不可能到最后去。所以这两个结构定义一个都不对！挺疯狂的结论，不是吗？而且有一个值得注意的现象：Matt Pietrek 在他的讲解中完整地引用了 EXSUP.INC 中的异常帧定义，却在他自己的 ShowSEHFrames 演示程序中也把这个成员从他的 VC\_EXCEPTION\_REGISTRATION 结构中“省略掉”了……我不是把自己的快乐建立在别人的痛苦之上的那一类人，所以与“找碴儿”相比，弄清问题的实质会带给我更多的快感。那么，VC 内部真正的异常帧究竟是什么样儿的？如果可能的话，我甚至连结构中的变量名都想知道。我很幸运，我最终真的知道了——这就是我在跟踪 \_ValidateEH3RN 时的意外收获。

\_ValidateEH3RN 在上面研究 \_except\_handler3 的时候提到过，它是用来对异常帧进行合法性验证的，它需要且仅需要用一个参数调用，就是一个 VC 的异常帧指针。Matt Pietrek 说的没错，CRT 中关于 SEH 的函数没有源代码可供参考。但幸运的是，Symbol 文件中的符号信息很充足，只要从 \_except\_handler3 函数中 step into 到 \_ValidateEH3RN，就可以发现调式环境的“局部变量”窗口有了反应！首先出来的是一个 pRN 变量，有四个成员：

1. Next: 展开之后发现还是一个 \*pRN 结构
2. ExceptionHandler: 值域中写着“\_\_except\_handler3”
3. ScopeTable: 指向一个结构，展开之后有三个成员
  - EnclosingLevel: 值为 -1
  - FilterFunc: 把值敲入反汇编的“地址”窗口，可以定位到 filter 入口
  - HandlerFunc: 把值敲入反汇编的“地址”窗口，可以定位到 handler 入口
4. TryLevel: 值为 0

呵呵，没错了，这个就是 VC 内部的异常帧结构了！再看看调用栈窗口，借了 \_ValidateEH3RN 的光，连结构名都看到了：\_EH3\_EXCEPTION\_REGISTRATION！而且 ScopeTable 的结构也可以看到了。不难发现这个结构中并没有那个“\_ebp”成员：最后一个成员是 TryLevel。再回头看看 \_except\_handler3 的反汇编，就会发现一个规律：所有对“\_ebp”的引用（也就是 [ebx+10h]）全部都出现在 lea 指令中，这说明什么呢？这说明，\_ebp 成员存在的意义只是为了取它的地址！那么 \_ebp 成员的值是什么呢？如果把带有 \_ebp 成员的 \_EXCEPTION\_REGISTRATION 结构套到上面的堆栈结构上就可以看出来：\_ebp 成员正好处于“调用者的基址 ebp”那个 DWORD 上。也就是说，\_ebp 成员确实是 ebp 寄存器的值，但却是上一个函数的 ebp，不是当前函数的。当前函数的 ebp 应该是这个 DWORD 的地址，而不是它的值！所以我前面说过，这是一个很迷惑人的成员，伪码中的第二个笔误必须改正，否则就会在上层函数的 ebp 上下文中执行当前函数的 Unwind 过程，那将是一个什么结果啊……

所以，目前 VC 中的异常帧结构中没有这个“\_ebp”成员——显然没有必要，SEH 中大量的递归调用、“non-local-goto”和堆栈 Unwind、已经够让人头昏脑胀的了，这个成员只能把事情搞得更离谱。想要的 ebp 值紧接着当前 \_EH3\_EXCEPTION\_REGISTRATION 结构的地址，只要 &pRN[1] 就可以取到了，实在没有必要为了取这个地址而强加上一个“\_ebp”成员。

## 暂时告一段落

写到这里，我似乎可以松口气了：Matt Pietrek 的文章已经吃透了，\_EH3\_EXCEPTION\_REGISTRATION 真正的结构也已经大白于天下了，VC 中的 SEH 处理似乎已经没有什么神秘的了，唯一剩下还没有研究过的就是 Unwind 过程。但这个过程完全封装在各个编译器厂商的内部实现中，与系统几乎没有关系，系统只负责发起 Unwind 调用，至于怎么 Unwind，系统也不知道。所以，虽然现在还不了解 Unwind，但它也已经是囊中之物了，只是目前还没有必要关心它。本着“师傅领进门、修行在个人”的精神，我又跟踪到 NTDLL.DLL 中的 Rtl 函数中转了一圈，不仅看到了 NT 中异常帧的具体结构，而且又发现了 Matt Pietrek 的伪码中与事实不符的地方——看上去这个地方涉及到嵌套异常处理甚至堆栈耗尽的问题……所以我打算单独写一篇文章好好分析一下这一部分。那么现在，我应该做的就是去洗个澡，然后舒舒服服地睡上一觉了。

## 相关文章

