[已完工][经典文章翻译]A Crash Course on the Depths of Win32 Structured Exception Handling

原文题目: <<A Crash Course on the Depths of Win32™ Structured Exception Handling>>
原文地址: http://www.microsoft.com/msj/0197/Exception/Exception.aspx
原作者: Matt Pietrek

在Win32的核心, 结构化异常处理(Structured Exception Handling)(SEH) 是操作系统提供的一种服务. 你能找到的所有关于SEH的文档都会描述某一种编译器的运行时库(runtime library)对操作系统实现的某种包装. 我会层层剥析SEH一直到它的最基本的概念.

这篇文章假设你熟悉Win32，C++

文章示例代码: Exception.exe (33KB)
Matt Pietrek 是<Windows 95 System Programming Secrets *(IDG Books, 1995)>的作者. 他在NuMega Technologies Inc.工作, 可以通过 mpietrek@tiac.com联系他.*

在所有由Win32操作系统提供的基础设施中, 可能被最广泛应用却没有文档说明的就是结构化异常处理了. 可能一想到Win32结构化异常处理, 大家就会想到诸如_try, _finally, 和_except这样的术语. 你可以在任何合格的讲Win32的书中找到关于SEH的还不错的描述. 甚至Win32SDK也对也对使用_try, _finally, 和_except来进行结构化异常处理有还不错的概述.

有了这么多的文档, 为什么我还要说SEH没有文档说明呢? 在核心里, 结构化异常处理是一种操作系统提供的服务. 你能找到的所有关于SEH的文档都会描述某一种编译器的运行时库(runtime library)对操作系统实现的某种包装. 关键字_try, _finally, 和_except并没有什么神秘的. 微软的操作系统和编译器团队定义了这些关键字还有这些关键字的行为. 其他的C++编译器供应商就只是简单地顺从这些关键字的语义而已. 当编译器的SEH层驯服了原始操作系统SEH的琐碎混乱之处之后, 编译器就把原始操作系统的关于SEH的细节隐藏起来了.

我收到过很多很多的邮件, 需要实现编译器层的SEH的人根本找不到关于操作系统基础设施提供的关于SEH的细节. 在一个合理的世界中, 我将能够拿出来Visual C++ 或 Borland C++ 的运行库源码来分析他们是如何做到的. 可惜的是, 由于某种不知道的原因, 编译器层次的SEH好像是一个巨大的秘密. 微软和Borlandboundary不愿意拿出来源代码来为最底层的SEH提供支持.

在这篇文章里, 我会剖析异常处理一直到它的最基本的概念. 为了这么做, 我会通过生成代码和运行时库的支持, 把操作系统提供的东西从编译器提供的东西中拆分出来. 当我深入到操作系统的关键例程的代码的时候, 我会使用Intel版本的Windows NT 4.0作为我的基础. 不过, 我所描述的绝大多数东西也适用于其他处理器.

我会避免真实的C++异常处理, 在C++的异常处理中会使用cache()而不是_except. 在幕后, 真实的C++异常处理跟我在这里描述的非常相似. 然而真实的C++异常处理会有一些额外的复杂之处, 我不会涉及他们, 因为他们会混淆我在这片文章中真正想要讲到的概念.

在挖掘组成Win32的结构化处理的晦涩的.H 和 .INC文件片段的时候, 最好的信息来源是IBM OS/2的头文件(尤其是BSEXCPT.H). 如果你在这个行业混过一段时间的话, 那你就不会觉得吃惊了. 这里描述的SEH机制是微软还在OS/2上工作的时候所定义的. 基于这个原因, 你会发现在Win32下的SEH跟OS/2异常类似.

SEH in the Buff
==========
如果要一次性把SEH的细节都照顾到的话, 那么任务量有点太大了, 我会从简单的地方开始, 逐层向上剖析. 如果你从来没有使用过结构化异常处理, 那你的状态还算不错, 不需要什么知识预备. 如果你以前使用过SEH, 你需要从你的脑子里把_try, GetExceptionCode, 和EXCEPTION_EXECUTE_HANDLER这些词汇清理掉. 假设这些概念对你来说是新的. 深呼吸, 准备好了么? 很好.

设想一下我告诉你当一个线程出错的时候, 操作系统会给你一个机会, 让你得到这个错误的通知. 更具体地, 当一个线程出错的时候, 操作系统会调用一个用户定义的callback函数. 这个callback函数能够做它想做的任何事. 比如说, 它可以修复引发错误的地方, 或者播放

一个搞笑的声音文件. 不管这个callback函数做什么, 它最后的动作时返回一个值, 用来告诉系统下一步该做什么的值(严格来说, 不是这样的, 但是对于现在来说已经足够接近了).

当你的代码把事情搞糟的时候让操作系统来调用你的函数, 那么这个callback函数应该像什么样子呢? 换句话说, 关于这个异常, 你想知道什么信息呢? 不必过多操心, Win32已经替你想好了. 一个异常callback函数看起来像这样:

```
EXCEPTION_DISPOSITION
__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

这个原型来自于标准的Win32头文件EXCPT.H, 初次看起来有点吓人. 如果你慢慢来看的话, 其实并不是那么难. 对于初学者来说,应该忽略返回值(EXCEPTION_DISPOSITION). 基本上, 你知道的事实是: 这是一个带有四个参数的叫做_except_handler的函数.

第一个参数是一个指向EXCEPTION_RECORD结构的指针. 这个结构体是在WINNT.H文件中定义的, 如下:

```
typedef struct _EXCEPTION_RECORD
{
  DWORD ExceptionCode;
  DWORD ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID ExceptionAddress;
  DWORD NumberParameters;
  DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
}  EXCEPTION_RECORD;
```

这里的ExceptionCode参数是操作系统赋予这个异常的一个号码. 你可以在WINNT.H中看到很多exception code的列表, 搜索以"STATUS_"开头的#define语句就可以了. 比如说, 熟悉的不能再熟悉的STATUS_ACCESS_VIOLATION的代码是0xC0000005. 一个更详细更全面的exception code的集合可以在Windows NT DDK中的NTSTATUS.H中找到. EXCEPTION_RECORD结构的第四个元素是exception发生的地址. 其他的EXCEPTION_RECORD中的元素目前可以忽略.

_except_handler函数的第二个参数是一个指向establisher frame结构的指针. 这是结构化异常处理中的一个至关重要的参数, 但是现在你暂时可以忽略它.

_except_handler函数的第三个参数是个指向CONTEXT结构的指针. CONTEXT结构是在WINNT.H中定义的, 它代表着某个线程的寄存器的值. Figure1展示了CONTEXT结构的定义. 当在SEH中使用的时候, CONTEXT结构代表着在异常发生时刻寄存器的值. 意外地是, 在GetThreadContext和SetThreadContext这两个API中, 这个结构是一样的.

第四个参数, 也是最后一个参数叫做DispatcherContext. 现在它也可以被忽略.

Figure 1 CONTEXT Structure

```
typedef struct _CONTEXT
{
    DWORD ContextFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
```

```
        DWORD   SegGs;
        DWORD   SegFs;
        DWORD   SegEs;
        DWORD   SegDs;
        DWORD   Edi;
        DWORD   Esi;
        DWORD   Ebx;
        DWORD   Edx;
        DWORD   Ecx;
        DWORD   Eax;
        DWORD   Ebp;
        DWORD   Eip;
        DWORD   SegCs;
        DWORD   EFlags;
        DWORD   Esp;
        DWORD   SegSs;
    } CONTEXT;
```

到目前为止简单地概括一下, 你有一个callback函数, 在异常发生的时候会被调用. 这个callback函数有四个参数, 其中的三个是指向结构的指针. 在这三个结构之中, 有些field很重要, 其他的却不是. 关键点是_except_handler 函数会收到丰富的信息, 比如发生的是什么类型的异常, 在哪里发生的这个异常. 通过这些信息, 异常callback函数可以决定下一步要做些什么.

看来是时候允许我丢出一个简单的小程序来展示_except_handler函数了, 但是还有一点东西需要补充. 特别地, 在异常发生的时候, 操作系统是如何知道到哪里去调用我们的callback函数呢? 答案是另一个叫做EXCEPTION_REGISTRATION的结构体. 在这片文章中你将会看到这个结构体, 别把这一部分跳过了. 唯一一个我能找到的比较正式的对于EXCEPTION_REGISTRATION的定义的地方在EXSUP.INC文件, 它存在于Visual C++运行时库的源文件中:

```
_EXCEPTION_REGISTRATION struct
    prev    dd      ?
    handler dd      ?
_EXCEPTION_REGISTRATION ends
```

你将会看到在WINNT.H中NT_TIB定义中, 这个结构被引用为一个_EXCEPTION_REGISTRATION_RECORD. 再往下, 就没有定义_EXCEPTION_REGISTRATION_RECORD 的地方了, 所以我即将开始的地方是EXSUP.INC里的汇编语言结构定义. 这只是我早先时候提到的SEH没有被文档记录的几个部分的例子之一.

在任何情况下, 都让我们先回过头来处理一下手头的问题. 操作系统是如何得知异常发生的时候到哪里去调用函数呢? EXCEPTION_REGISTRATION 结构由两个fields组成, 其中的第一个你现在可以忽略. 第二个field, 就是handler, 包含一个指向_except_ handler 回调函数的指针. 这让你离答案更近了一步, 但是问题来了, OS到哪里去找EXCEPTION_REGISTRATION结构呢?

为了回答这个问题, 回忆一下结构化异常处理在单线程基础上的工作机制是有帮助的. 每个线程都有自己的exception handler回调函数. 在我1996年的专栏中, 我描述了一个关键的Win32结构, 线程信息块(TEB或TIB). 这个结构体当中的某些field在Windows NT, Windows® 95, Win32s, 和OS/2是一样的. TIB的第一个DWORD是一个指向线程的EXCEPTION_REGISTRATION的指针. 在Intel的Win32平台上, FS寄存器永远指向当前的TIB. 即, 在FS:[0]的位置, 你可以找到一个指向EXCEPTION_REGISTRATION结构的指针.

现在我们已经比较深入了. 当一个exception发生的时候, 系统会查看出错线程的TIB结构, 取回一个指向一个EXCEPTION_REGISTRATION结构的指针. 在这个结构中, 有一个指向_except_handler回调函数的指针. 操作系统现在知道了足够的信息来调用_except_handler回调函数, 如**Figure 2**所示:

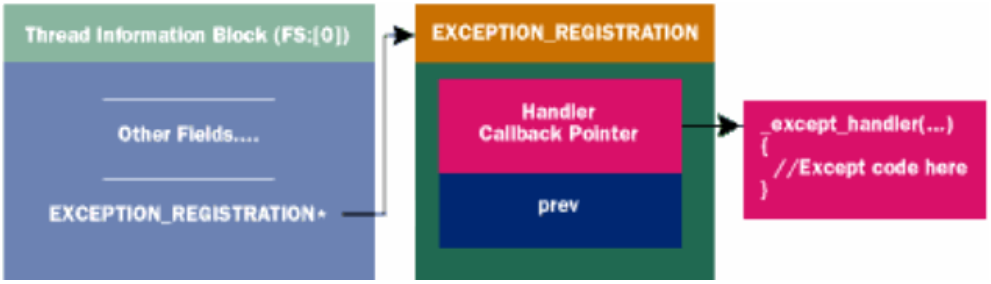


Figure 2 _except_handler_function

小的知识点一块块的拼起来了之后, 我写了一个小程序来示范这个操作系统级的结构化异常处理的描述. **Figure 3** 展现了MYSEH.CPP, 其中仅有两个函数. Main函数使用三个内联的ASM块. 第一块通过两个PUSH指令("PUSH handler" 和"PUSH FS:[0]")在栈上构建了EXCEPTION_REGISTRATION结构. PUSH FS:[0]保存了之前的 FS:[0]的值作为这个结构的一部分, 但是目前来说这还不重要. 重要的是栈上有了一个8-byte大小的EXCEPTION_REGISTRATION 结构. 紧跟着的下一条指令(MOV FS:[0],ESP)使得TIB中的第一个DWORD指向了新的EXCEPTION_REGISTRATION结构.

Figure 3 MYSEH.CPP

```
//==================================================
// MYSEH - Matt Pietrek 1997
// Microsoft Systems Journal, January 1997
// FILE: MYSEH.CPP
// To compile: CL MYSEH.CPP
//==================================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

DWORD  scratch;

EXCEPTION_DISPOSITION
__cdecl
_except_handler(
   struct _EXCEPTION_RECORD *ExceptionRecord,
   void * EstablisherFrame,
   struct _CONTEXT *ContextRecord,
   void * DispatcherContext )
{
   unsigned i;

   // Indicate that we made it to our exception handler
   printf( "Hello from an exception handler\n" );

   // Change EAX in the context record so that it points to someplace
   // where we can successfully write
   ContextRecord->Eax = (DWORD)&scratch;

   // Tell the OS to restart the faulting instruction
   return ExceptionContinueExecution;
}

int main()
{
   DWORD handler = (DWORD)_except_handler;

   __asm
   {                   // Build EXCEPTION_REGISTRATION record:
      push    handler        // Address of handler function
      push    FS:[0]         // Address of previous handler
      mov     FS:[0],ESP     // Install new EXECEPTION_REGISTRATION
   }

   __asm
   {
      mov     eax,0          // Zero out EAX
      mov     [eax], 1       // Write to EAX to deliberately cause a fault
   }

   printf( "After writing!\n" );
```

```
    __asm
    {                   // Remove our EXECEPTION_REGISTRATION record
        mov     eax,[ESP]      // Get pointer to previous record
        mov     FS:[0], EAX     // Install previous record
        add     esp, 8         // Clean our EXECEPTION_REGISTRATION off stack
    }

    return 0;
}
```

如果你好奇为什么我在栈上创建了一个EXCEPTION_REGISTRATION结构而不是使用一个全局变量, 我有一个很好的理由. 当你使用编译器的_try/_except 语法的时候, 编译器也会在栈上创建EXCEPTION_REGISTRATION结构的. 我只是展现给你了编译器用来处理_try/_except的方式的一个简化版本.
回到Main函数, 下一个_asm块的目的是引发一个错误, 先将EAX寄存器清0，然后使用它(EAX)的值作为下一条指令用来写入的内存地址(MOV [EAX],1).
最后的__asm块移除了这个简单的异常处理器: 首先它恢复之前的FS:[0]的内容, 然后它将EXCEPTION_REGISTRATION记录从栈中弹出(ADD ESP,8).

现在, 假设你正在运行MYSEH.EXE, 那么你会看见发生的一切. 当指令MOV [EAX],1执行的时候, 它会引发一个非法访问的异常. 操作系统会查询TIB中的FS:[0], 找到指向EXCEPTION_REGISTRATION结构的指针. 在这个结构中有一个指针指向在MYSEH.CPP中的_except_handler 函数. 系统然后将四个所需的参数压栈, 然后调用_except_handler函数.

在_except_handler中, 代码首先通过一个printf语句说明"嗨, 我搞糟的地方在这里!". 然后_except_handler修复了引发错误的问题. 也就是EAX寄存器指向一个不能写入的内存地址(地址0). 修复的方法是修改CONTEXT结构体中EAX的值, 让它指向一个可写的地址. 在这个简单的程序里, 一个DWORD变量(scratch)就是被设计来完成这个目的的. _except_handler函数的最后的动作时返回值ExceptionContinueExecution, ExceptionContinueExecution是在EXCPT.H文件中定义的.

当操作系统发现返回的值是ExceptionContinueExecution 的时候, 它会理解成这意味着你已经修复了问题, 错误的语句可以被再次执行. 因为我的_except_handler函数修改了EAX寄存器的值, 让它指向了合法的内存地址, MOV EAX, 1指令第二次就成功地执行了, main函数可以正常地继续了. 你看到了, 不是那么复杂的, 对不对?

再深入一点 - Moving In a Little Deeper

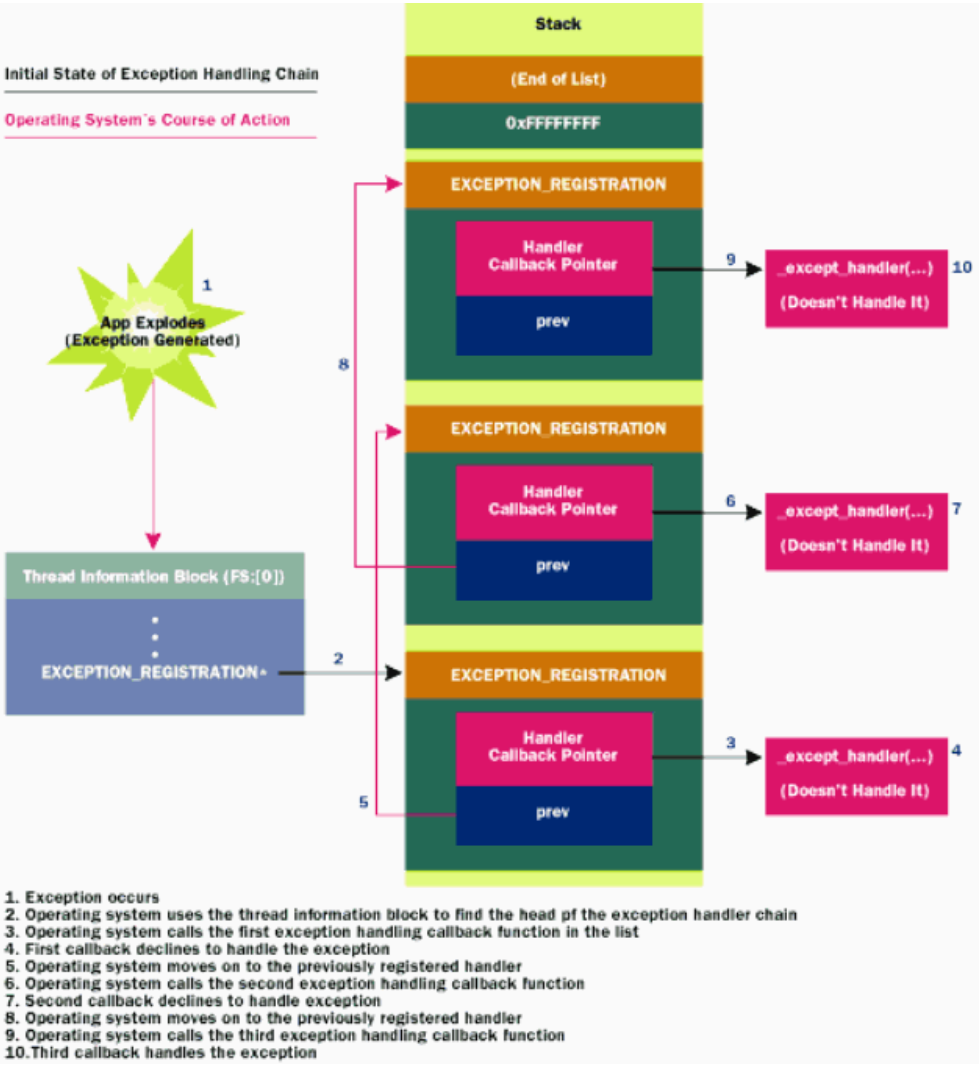
==========

研究了这个最简单的情形后, 让我们回过头来填补一些当时留下的空隙吧. 虽然异常回调完成得很棒, 它却不是一个完美的解决方案. 在任何大小的应用程序中, 书写一个简单的函数来处理程序中任何地方都可能会出现的异常, 会非常麻烦. 一个更加可行的方式是拥有多重处理异常的路径, 每一个都针对应用程序的某部分而特别订制. 难道你不知道么, 操作系统提供的就是这个功能.

还记得操作系统用来查找异常回调函数地址的EXCEPTION_REGISTRATION 结构么? 这个结构的第一个参数, 我们稍早时忽略的那个, 被叫做prev. 它实际上是一个指向另一个EXCEPTION_REGISTRATION 结构的指针. 这第二个EXCEPTION_REGISTRATION 结构能够拥有完全不同的处理函数. 还有, 它的prev域可以指向第三个EXCEPTION_REGISTRATION 结构, 以此类推. 简单点说, 它们形成了一个EXCEPTION_REGISTRATION 的链表. 这个链表的头永远是被线程信息块(TIB)中的第一个DWORD(intel平台机器里的FS:[0])所指向的.

操作系统是如何处理这个EXCEPTION_REGISTRATION 的链表的呢? 当异常发生的时候, 系统会先遍历该结构的链表, 寻找包含愿意处理这个异常的回调函数的EXCEPTION_REGISTRATION结构. 在MYSEH.CPP中, 回调函数通过返回值ExceptionContinueExecution来表示同意处理这个异常. 异常回调函数也可以拒绝处理异常. 在这个情况下, 系统会继续走到链表中的下一个EXCEPTION_REGISTRATION结构上, 询问异常回调函数是否愿意处理这个异常. **Figure 4**展现了这个过程. 一旦操作系统找到了一个能够处理异常的callback函数, 它就停止遍历链表了.

Figure 4 Finding a Structure to Handle the Exception



我展现了一个异常回调函数的例子, 看看Figure 5里的MYSEH2.CPP吧. 为了保持代码的简洁, 我使用编译器层的异常处理玩了个小花样. main函数只是设立了一个_try/_except块. 在__try块中, 有一个对HomeGrownFrame函数的调用. 这个函数与更早的那个MYSEH程序非常类似. 它在栈上创建了一个EXCEPTION_REGISTRATION 结构, 让FS:[0]指向这个结构.在创建了新的处理函数之后, 这个函数通过向NULL指针写数据故意地引发了一个错误:

```
*(PDWORD)0 = 0;
```

Figure 5 MYSEH2.CPP

```
//==================================================
// MYSEH2 - Matt Pietrek 1997
// Microsoft Systems Journal, January 1997
// FILE: MYSEH2.CPP
// To compile: CL MYSEH2.CPP
//==================================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>

EXCEPTION_DISPOSITION
__cdecl
_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext )
{
    printf( "Home Grown handler: Exception Code: %08X Exception Flags %X",
            ExceptionRecord->ExceptionCode, ExceptionRecord->ExceptionFlags );
```

```
    if ( ExceptionRecord->ExceptionFlags & 1 )
        printf( " EH_NONCONTINUABLE" );
    if ( ExceptionRecord->ExceptionFlags & 2 )
        printf( " EH_UNWINDING" );
    if ( ExceptionRecord->ExceptionFlags & 4 )
        printf( " EH_EXIT_UNWIND" );
    if ( ExceptionRecord->ExceptionFlags & 8 )
        printf( " EH_STACK_INVALID" );
    if ( ExceptionRecord->ExceptionFlags & 0x10 )
        printf( " EH_NESTED_CALL" );

    printf( "\n" );

    // Punt... We don't want to handle this... Let somebody else handle it
    return ExceptionContinueSearch;
}

void HomeGrownFrame( void )
{
    DWORD handler = (DWORD)_except_handler;

    __asm
    {                       // Build EXCEPTION_REGISTRATION record:
        push    handler         // Address of handler function
        push    FS:[0]          // Address of previous handler
        mov     FS:[0],ESP      // Install new EXECEPTION_REGISTRATION
    }

    *(PDWORD)0 = 0;             // Write to address 0 to cause a fault

    printf( "I should never get here!\n" );

    __asm
    {                       // Remove our EXECEPTION_REGISTRATION record
        mov     eax,[ESP]       // Get pointer to previous record
        mov     FS:[0], EAX     // Install previous record
        add     esp, 8          // Clean our EXECEPTION_REGISTRATION off stack
    }
}

int main()
{
    _try
    {
        HomeGrownFrame();
    }
    _except( EXCEPTION_EXECUTE_HANDLER )
    {
        printf( "Caught the exception in main()\n" );
    }

    return 0;
}
```

异常回调函数, 又一次被命名为_except_ handler, 这一次跟上个版本有很大不同. 代码首先打印了ExceptionRecord 结构中的异常代码(exception code)和异常标志(exception flag), ExceptionRecord 结构的指针被作为一个参数传递给了我们的_except_ handler函数. 打印出exception flag的原因会在晚些时候变的更清楚. 因为这个_except_ handler函数并没有打算修复违例的代码, 该函数返回了ExceptionContinueSearch. 这会引发操作系统继续搜索链表中的下一个EXCEPTION_REGISTRATION 结构. 现在, 相信我的话, 下一个异常回调函数就是为main函数中的 _try/_except而被设立的了. _except 块简单地打印出了信息"Caught the exception in main()". 在这个例子里, 对异常的处理就跟忽略它的发生一样的简单.

这里需要提及的一个关键点是执行控制. 当一个handler拒绝处理一个exception的时候, 它会有效地拒绝去判断控制将最终在何处被恢复. 接受异常的handler就是那个决定了控制在所有异常处理代码结束之后最终将在哪个地址上继续的那个handler. 这里有一个重要的隐含含义, 它目前还不明显.

当使用结构化异常处理的时候, 如果一个函数的异常处理函数并没有处理掉异常的话, 它也许会以一种不正常的方式退出. 比如说, 在MYSEH2中, HomeGrownFrame 中的最小的handler就没有处理掉异常. 既然链表中的某个部分的处理函数处理了异常(main函数), 出错指令后面的printf就再没有被执行了. 从某种程度上说, 使用结构化异常处理跟使用运行时的setjmp和longjmp函数是一样的.

如果你运行MYSEH2， 你会在输出中发现一些令人吃惊的东西. 看起来对_except_handler 函数的调用有两次! 根据你了解了的知识, 其中的第一次是不难理解的. 但是第二次调用时怎么回事呢?

```
Home Grown handler: Exception Code: C0000005 Exception Flags 0
Home Grown handler: Exception Code: C0000027 Exception Flags 2
                                    EH_UNWINDING
Caught the Exception in main()
```

这里有一处明显的不同: 比较两行以"Home Grown Handler" 开头的输出. 注意, 第一次exception flag是0， 而第二次是2。这把我们带到议题unwinding上来了. 提前一点, 当异常回调拒绝处理一个异常的时候, 它会再被调用一次. 但是这次回调并不会立即发生. 事实比较复杂, 我需要最后细化异常的secnario一下了.

当一个异常发生的时候, 系统会遍历EXCEPTION_REGISTRATION结构的链表, 直到他找到一个能够处理异常的handler为止. 一旦找到了一个handler, 系统会再次遍历列表, 遍历会停在这个能够处理异常的节点上. 在这第二次的遍历中, 系统会第二次的调用每一个异常处理函数. 关键的不同在于, 在第二次调用中, 2这个值会被赋予exception flag. 这个值也就是EH_UNWINDING. (EH_UNWINDING的定义在EXCEPT.INC中, 该文件在Virtual C++运行时库的源代码中, 但是跟Win32SDK中的没啥关系).

那么EH_UNWINDING是什么意思呢? 当一个异常回调函数在第二次被执行的时候(flag的值是EH_UNWINDING), 操作系统会给handler function一个机会来执行一些它需要做的清理工作. 那种清理工作(cleanup)呢? 最好的例子是C++类的析构函数. 当一个函数的exception handler拒绝处理一个异常的时候, 典型地, 控制并不会以正常的方式从函数中离开的. 现在, 假设有一个函数, 其中声明了一个C++的类对象作为一个局部变量. C++标准说, 析构函数是一定会被调用的. 带着EH_UNWINDING标志的exception handler的第二次调用就是个供函数来执行诸如调用析构函数和_finally块的机会.

当一个异常被处理了, 所有前面的exception frame被依次展开了之后, 执行会在任何处理回调函数确定的一个地方继续下去. 记住, 仅仅设置指令指针到需要的代码地址是不够的. 代码继续执行的地方还需要栈顶指针和栈框架指针被设置为合适的值. 所以, 处理异常的handler有责任设置栈顶指针和栈框架指针的值, 设置之后, 栈框架中包含有处理异常的SEH代码.

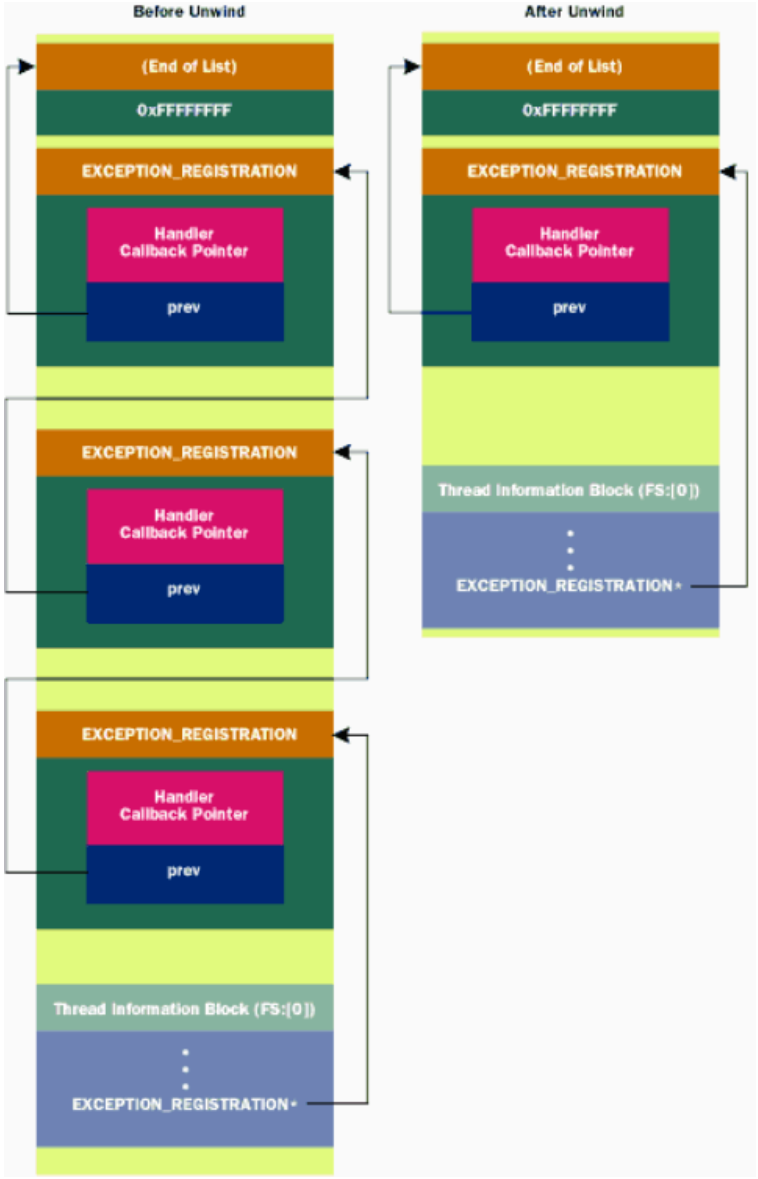


Figure 6 Unwinding from an Exception

用更概括的术语, 从一个exception展开的动作在栈上引发了栈的handling frame之下的部分都被移除了. 这几乎跟那些函数从没被调用过一样. 另一个unwind的效果是处理异常的节点之前的所有EXCEPTION_REGISTRATION都被从列表中移除了. 这是合理的, 因为这些EXCEPTION_REGISTRATION都是建立在栈上的. 在异常被处理了之后, 栈顶和栈框架指针都会比从列表被移出的EXCEPTION_REGISTRATION的地址要高. **Figure 6**说明了我的观点.

救命呀! 没有人处理这个异常! (Help! Nobody Handled It!)

==========

到目前为止, 我一直隐含地假设操作系统总是会在EXCEPTION_REGISTRATION结构的链表的某处找到一个handler. 那么如果没有一个结构愿意站出来处理这个异常怎么办? 事实上, 这中情况从来就不会发生. 原因是操作系统偷偷地为每一个线程配置了一个默认的异常处理的handler. 默认的handler永远是链表的最后一个节点, 并且总是会处理掉异常. 它的行为与一般的异常处理回调函数有某种程度的不同, 我会在稍后展现出来.

让我们看一下操作系统插入默认的, 最终的异常handler的地方吧. 显然, 这个动作会在线程执行的非常早期的时候发生, 所谓早期, 是指在任何用户代码执行之前. **Figure 7** 展现了我为BaseProcessStart方法写的一些伪代码, BaseProcessStart是一个Windows NT的KERNEL32.DLL的内部函数. 它带一个参数, 即线程的入口地址. BaseProcessStart在新进程的context下运行, 并且调用入口地址来开动进程中的首个线程的执行.

Figure 7 BaseProcessStart Pseudocode

```
BaseProcessStart( PVOID lpfnEntryPoint )
{
   DWORD retValue
   DWORD currentESP;
   DWORD exceptionCode;

   currentESP = ESP;

   _try
   {
      NtSetInformationThread( GetCurrentThread(),
                        ThreadQuerySetWin32StartAddress,
                        &lpfnEntryPoint, sizeof(lpfnEntryPoint) );

      retValue = lpfnEntryPoint();

      ExitThread( retValue );
   }
   _except(// filter-expression code
         exceptionCode = GetExceptionInformation(),
         UnhandledExceptionFilter( GetExceptionInformation() ) )
   {
      ESP = currentESP;

      if ( !_BaseRunningInServerProcess )        // Regular process
         ExitProcess( exceptionCode );
      else                                       // Service
         ExitThread( exceptionCode );
   }
}
```

在伪代码中, 注意对lpfnEntryPoint 的调用是包装在一个_try 和_except 的构造中的. 这个_try块就是那个安装默认的, 最终的exception handler的地方. 所有后续注册的异常处理handlers都会被插入到链表中的这个节点的前面. 如果lpfnEntryPoint 函数返回了, 那么线程就成功地运行结束了, 没有引发任何的异常. 如果没有异常, BaseProcessStart 会调用ExitThread 来结束线程.

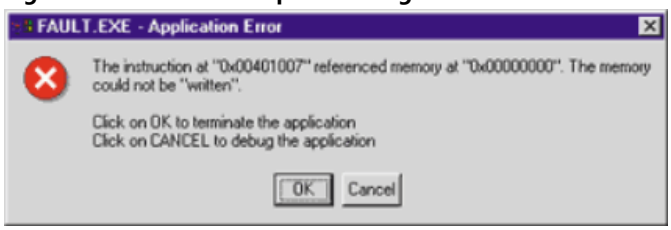
另一方面, 如果线程出错了, 又没有其他的exception handler处理呢? 在这种情况下, 控制会进入到_except关键字后面的括号中. 在BaseProcessStart, 这段代码调用了UnhandledExceptionFilter 这个API函数, 我稍后会介绍这个函数. 现在, 关键点是UnhandledExceptionFilter API包含默认exception handler的重要成分.

如果UnhandledExceptionFilter 返回了EXCEPTION_EXECUTE_HANDLER, 那么BaseProcessStart 中的_except块中的代码会被执行. _except块内的代码所作的工作就是通过调用ExitProcess来结束掉当前的进程. 花一秒中的时间在这里思考一下, 其实这是合理的: 如果一个程序遇到了错误, 并且没有任何人处理这个错误的话, 操作系统应该终止这个进程, 这应该算是常识. 只不过你在伪代码中看到的是这个常识的精确的发生地和发生方式.

对我刚才描述的要点还有一个最后的补充. 如果出错的线程是作为一个服务运行着的, 并且是一个基于线程的服务的话, 那么_except块的代码并不会调用ExitProcess, 取而代之的是调用ExitThread . 你并不需要仅仅因为一个线程出了点毛病, 就把整个进程干掉.

那么, 默认的exception handler中的UnhandledExceptionFilter里的代码都做了些什么呢? 当我在研讨会上问起这个问题的时候, 很少有人能猜到在一个没有被处理的异常发生的时候, 操作系统的默认行为. 如果我们来一个非常简单的对于default handler的行为的demo的话, 事情就简单多了, 大家也更容易理解. 我简单地运行一个程序, 故意地引发一个错误, 并且指出错误的结果(**Figure 8**).

Figure 8 Unhandled Exception Dialog



在较高的层次上看, UnhandledExceptionFilter 会显示一个对话框, 告诉你发生了一个错误. 在那个时间点上, 你被给予一个机会, 要么终止进程, 要么debug出错了的进程. 在幕后还有更多的事情发生, 我会在本文即将结束的时候描述这些事情.

正如我已经演示了的, 当一个异常发生的时候, 用户写的代码能够被执行. 一样地, 在unwind操作的时候, 用户写的代码也能够被执行. 这用户写的代码可能会有bug, 并引发另一个异常. 基于这个原因, exception的回调函数有另外两个值可以返回: **ExceptionNestedException** 和**ExceptionCollidedUnwind.** 很明显这很重要, 而且这很明显是非常高级的话题了, 我并不打算在这里停下来描述它, 因为光是理解基本概念就已经够难了.

编译器水平的结构化异常处理(Compiler-level SEH)

==========

我已经时不时地引用_try 和_except这两个关键字了, 到现在我写的东西还都是由操作系统实现的. 然而, 在我的两个小程序挑逗性地使用着原始的系统结构化异常处理的时候, 编译器所包装的这个功能肯定早为你准备好了. 让我们看看Virtual C++是如何在操作系统级的SEH基础架构上构建自己的对结构化异常处理的支持的吧.

在继续下去之前, 回忆下能够使用操作系统级的SEH基础设施来完成完全不同的事情的另一个编译器是有必要的. 没有人说编译器一定要实现由Win32 SDK文档描述的_try/_except模型. 比如说, 即将发布的Visual Basic 5.0就在它的运行时代码中使用了结构化异常处理, 但是数据结构和算法都与我在这里描述的完全不同.

如果你通读Win32 SDK文档关于结构化异常处理的部分, 你会遇到下面的叫做"frame-based"的语法的exception handler:

```
try {
   // guarded body of code
}
except (filter-expression) {
   // exception-handler block
}
```

简单地说, 所有在try块中的代码都被一个EXCEPTION_REGISTRATION 保护着, 这个EXCEPTION_REGISTRATION 是构建在函数的栈帧上的(stack frame). 在入口处, 这个新的EXCEPTION_REGISTRATION 会被放在exception handler的链表的头部. 在_try块的结尾, 它的EXCEPTION_REGISTRATION 会被从链表的头部移除. 正如我早些时候提到的, exception handler链表的头是存储在FS:[0]当中的. 所以, 如果你步入debugger的汇编语言语句的话, 你会看到如下的指令:

```
    MOV DWORD PTR FS:[00000000],ESP
```

或者

```
    MOV DWORD PTR FS:[00000000],ECX
```

你可以确定, 这就是在配置和拆除一个_try/_except 块了.

现在, 你已经知道一个_try 块跟栈上的EXCEPTION_REGISTRATION 结构的关系, 那在EXCEPTION_ REGISTRATION里的回调函数又是怎么回事儿呢? 用Win32的术语来说, 异常回调函数对应着一个过滤表达式(filter-expression)代码. 清理一下你的记忆, 过滤表达式就是_except关键字后面跟着的括号里的代码. 就是这段过滤表达式能够决定紧随其后的{}里的代码是否会执行.

既然你写了filter-expression代码, 那么就由你来决定是否某个特定的exception应该在你代码的某个特定的位置来处理. 你的filter-experssion代码既可以知识简单地打印一句"我处理了这个异常", 也可以在返回系统, 告诉系统下一步该做什么之前触发一个极其复杂的函数. 你说了算. 重点是, 你的filter-expression代码就是我早先描述的异常回调(exception callback)

我刚刚描述的东西尽管简单的非常合理, 但它确只不过是真实世界的一种乐观抽象. 事实更加复杂这一点无疑是丑陋的现实. 对于初学者来说, 你的filter-expression代码并不是直接由操作系统调用的. 其实, 每一个EXCEPTION_REGISTRATION的exception handler域都指向一个相同的函数. 这个函数存在于Visual C++ runtime library里, 并且被叫做__except_handler3. 实际上是你__except_handler3调用的你的filter-expression code, 晚些时候我会再解释这一点的.

另一个对与简单试图的扭曲之处是: EXCEPTION_REGISTRATION们并不是在每一次进入或离开_try block的时候被构造和拆解的. 取而代之的是, 你可以在一个函数中添加多个_try/_except结构, 但是只能有一个EXCEPTION_REGISTRATION被创建在栈上. 同理, 你或许有一层_try block内嵌在另一个_try block中, 但是, Visual C++只创建一个EXCEPTION_REGISTRATION.

如果一个单独的exception handler(比如__except_handler3)足以处理整个exe或dll, 并且如果一个EXCEPTION_REGISTRATION 处理过个_try block的话, 很显然这里发生的事情会比眼睛看到的多好多. 这些神奇的事情是通过在你一般看不到的表里头的数据来完成的. 然而, 因为这篇文章的目的是解剖异常处理, 看不到这些数据表也不能阻挡我们的, 让我们来一起看一下这些数据结构吧.

扩展了的Exception Handling Frame- (The Extended Exception Handling Frame)

==========

Visual C++ SEH的实现并没有使用原始的EXCEPTION_REGISTRATION. 取而代之的是, 它在这个结构的末尾添加了一些额外的数据域. 这些额外的数据对于允许函数(__except_handler3)处理所有的异常, 还有能够让控制路由到合适的filter-expression和_except块, 这两点都是至关重要的. Visual C++对于这个结构扩展的格式可以在EXSUP.INC中找到, 该文件存在于Visual C++ runtime library的源代码中. 在这个文件中, 你可以找到如下的(已注释的)定义:

```
;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;
;};
```

你已经见过头两个fields了, 一个是prev, 另一个是handler. 它们组成了基本的EXCEPTION_REGISTRATION 结构. 最后的三个field是新加上去的, scopetable, trylevel, 和_ebp. 域scopetable 指向一个元素类型为scopetable_entries的数组, 而域trylevel就是这个数组的索引值. 随后的域_edp, 是在EXCEPTION_REGISTRATION 创建之前的栈框架指针(EBP)的值.

域_ebp称为扩展的EXCEPTION_REGISTRATION结构的一部分并不是巧合. 它通过PUSH EBP指令被放置在结构中, push ebp指令是绝大多数函数开始的指令. 它的效果是使得所有其他的EXCEPTION_REGISTRATION的field都变成可以访问的了, 原因是框架指针的负位移. 比如说, trylevel域在[EBP-04]的位置, 所以scopetable指针的位置就在 [EBP-08], 以此类推.

紧挨着扩展的EXCEPTION_REGISTRATION结构的下面, Visual C++还添加了两个额外的值. 紧接着的一个DWORD里, 它保留了一个指向EXCEPTION_POINTERS结构的指针(标准Win32的结构). 这个指针在你调用GetExceptionInformation API的时候会被返回. SDK文档暗示GetExceptionInformation是一个标准Win32API, 事实上, GetExceptionInformation是一个编译器固有的函数. 当你调用这个函数的时候, Visual C++生成下面的指令:

```
    MOV EAX,DWORD PTR [EBP-14]
```

正如GetExceptionInformation 是一个编译器固有函数一样, 与之相关联的GetExceptionCode函数也是一个编译器固有函数. GetExceptionCode 只是寻找并返回GetExceptionInformation 所返回的结构中的一个数据域(field). 我将会把这个留给读者做一个练习, 练习弄清楚在Visual C++为GetExceptionCode产生如下指令的时候, 究竟都发生了什么:

```
MOV EAX,DWORD PTR [EBP-14]
MOV EAX,DWORD PTR [EAX]
MOV EAX,DWORD PTR [EAX]
```

返回到扩展了的EXCEPTION_REGISTRATION 结构, 在结构开始前的8个字节, Visual C++会保留一个DWORD来保存所有已经执行了的开场代码的最终的栈指针(ESP). 这个DWORD就是函数正常执行时ESP寄存器的一个普通值(除非当参数正在压栈, 并准备调用下一

个函数).

看起来我已经丢给了你一大堆信息, 事实上我的确是这样做的. 在继续下去之前, 让我们稍微暂停并回顾一下Vistal C++为一个使用结构化异常处理而生成的标准的栈内的情况吧.

```
EBP-00 _ebp
EBP-04 trylevel
EBP-08 scopetable pointer
EBP-0C handler function address
EBP-10 previous EXCEPTION_REGISTRATION
EBP-14 GetExceptionPointers
EBP-18 Standard ESP in frame
```

从操作系统的角度来看, 只有两个fields组成了原始的EXCEPTION_REGISTRATION结构: 在[EBP-10]位置上的prev指针, 还有在位置[EBP-0Ch]上的handler函数指针. 其他的东西都是具体针对Visual C++的实现的. 了解了这些之后, 让我们来看看体现了编译器等级的结构化异常处理的Visual C++运行时库的函数__except_handler3吧.

__except_handler3 和 scopetable

==========

我特别希望能够给你看看Visual C++运行时库的源代码, 并且让你自己看一看函数__except_handler3的实现, 但是我不能这样做. 作为替代, 我会让你看看我拼凑出来的伪代码(请看**Figure 9**)

Figure 9 __except_handler3 Pseudocode

```
int __except_handler3(
   struct _EXCEPTION_RECORD * pExceptionRecord,
   struct EXCEPTION_REGISTRATION * pRegistrationFrame,
   struct _CONTEXT *pContextRecord,
   void * pDispatcherContext )
{
   LONG filterFuncRet
   LONG trylevel
   EXCEPTION_POINTERS exceptPtrs
   PSCOPETABLE pScopeTable

   CLD     // Clear the direction flag (make no assumptions!)

   // if neither the EXCEPTION_UNWINDING nor EXCEPTION_EXIT_UNWIND bit
   // is set...  This is true the first time through the handler (the
   // non-unwinding case)

   if ( ! (pExceptionRecord->ExceptionFlags
         & (EXCEPTION_UNWINDING | EXCEPTION_EXIT_UNWIND)) )
   {
      // Build the EXCEPTION_POINTERS structure on the stack
      exceptPtrs.ExceptionRecord = pExceptionRecord;
      exceptPtrs.ContextRecord = pContextRecord;

      // Put the pointer to the EXCEPTION_POINTERS 4 bytes below the
      // establisher frame.  See ASM code for GetExceptionInformation
      *(PDWORD)((PBYTE)pRegistrationFrame - 4) = &exceptPtrs;

      // Get initial "trylevel" value
      trylevel = pRegistrationFrame->trylevel

      // Get a pointer to the scopetable array
      scopeTable = pRegistrationFrame->scopetable;

search_for_handler:

      if ( pRegistrationFrame->trylevel != TRYLEVEL_NONE )
      {
         if ( pRegistrationFrame->scopetable[trylevel].lpfnFilter )
         {
            PUSH EBP                    // Save this frame EBP
```

```
            // !!!Very Important!!!  Switch to original EBP.  This is
            // what allows all locals in the frame to have the same
            // value as before the exception occurred.
            EBP = &pRegistrationFrame->_ebp

            // Call the filter function
            filterFuncRet = scopetable[trylevel].lpfnFilter();

            POP EBP                     // Restore handler frame EBP

            if ( filterFuncRet != EXCEPTION_CONTINUE_SEARCH )
            {
               if ( filterFuncRet < 0 ) // EXCEPTION_CONTINUE_EXECUTION
                  return ExceptionContinueExecution;

               // If we get here, EXCEPTION_EXECUTE_HANDLER was specified
               scopetable == pRegistrationFrame->scopetable

               // Does the actual OS cleanup of registration frames
               // Causes this function to recurse
               __global_unwind2( pRegistrationFrame );

               // Once we get here, everything is all cleaned up, except
               // for the last frame, where we'll continue execution
               EBP = &pRegistrationFrame->_ebp

               __local_unwind2( pRegistrationFrame, trylevel );

               // NLG == "non-local-goto" (setjmp/longjmp stuff)
               __NLG_Notify( 1 );  // EAX == scopetable->lpfnHandler

               // Set the current trylevel to whatever SCOPETABLE entry
               // was being used when a handler was found
               pRegistrationFrame->trylevel = scopetable->previousTryLevel;

               // Call the _except {} block.  Never returns.
               pRegistrationFrame->scopetable[trylevel].lpfnHandler();
            }
         }

         scopeTable = pRegistrationFrame->scopetable;
         trylevel = scopeTable->previousTryLevel

         goto search_for_handler;
      }
      else    // trylevel == TRYLEVEL_NONE
      {
         retvalue == DISPOSITION_CONTINUE_SEARCH;
      }
   }
   else    // EXCEPTION_UNWINDING or EXCEPTION_EXIT_UNWIND flags are set
   {
      PUSH EBP    // Save EBP
      EBP = pRegistrationFrame->_ebp  // Set EBP for __local_unwind2

      __local_unwind2( pRegistrationFrame, TRYLEVEL_NONE )

      POP EBP     // Restore EBP

      retvalue == DISPOSITION_CONTINUE_SEARCH;
   }
}
```

尽管__except_handler3看起来有很多代码, 但请记住它只不过是我在这篇文章开头描述过的一个异常回调函数罢了. 它接受跟我自制的在MYSEH.EXE 和MYSEH2.EXE中的异常回调函数完全相同的4个参数. 在最顶层的等级, __except_handler3被一个IF语句拆分为两个部分. 这是因为这个函数会被两次调到, 一次是普通的调用, 另一次是在unwind展开阶段的调用. 这个函数的很大一部分都是为了非展开(non-unwinding)回调而服务的.

这里的代码的开始部分首先在栈上创建了一个EXCEPTION_POINTERS 结构体, 使用两个__except_handler3的参数来初始化这个结构体. 这个结构体的地址, 也就是我起名为exceptPtrs的, 被放在了[EBP-14]. 这里初始化了GetExceptionInformation 和 GetExceptionCode 两个函数使用的指针.

下一步, __except_handler3从EXCEPTION_REGISTRATION frame (位置在[EBP-04])中取回当前的trylevel变量. 这个trylevel变量的作用就是scopetable数组的一个索引, 通过使用这个索引, 允许了单个的EXCEPTION_REGISTRATION被一个函数中的多个多个_try块所使用, 就跟折叠的_try块一样. 每一个scopetable的条目看起来像这样:

```
typedef struct _SCOPETABLE
{
   DWORD       previousTryLevel;
   DWORD       lpfnFilter
   DWORD       lpfnHandler
} SCOPETABLE, *PSCOPETABLE;
```

在SCOPETABLE 中的第二个和第三个参数比较容易理解. 它们是你的filter-expression和corresponding_except代码块的地址. 前一个tryLevel数据域有点小难. 简单来说, 它是嵌套的try块. 这里的重点是, 在一个函数中对每一个_try块, 都有有一个SCOPETABLE 的入口.

正如我早些时候提到的, 当前的trylevel指定了要被使用的scopeable数组入口. 接下来, 指定filter-expression和_except块的地址. 现在让我们想象一个_try块嵌套在另一个_try块中的场景吧. 如果里面的_try块的filter-expression没有处理掉异常, 那么外面的_try块的filter-expression必须得到消息. 那么__except_handler3如何得知哪个SCOPETABLE 入口关联到外面的_try块呢? 它的索引是通过一个SCOPETABLE 入口里的previousTryLevel来给出的. 使用这个格式, 你可以创建任意嵌套的_try块. previousTryLevel 数据域表现的像链表中的节点一样, 该链表中存储的都是函数中可能的exception handler. 链表的结尾是通过一个trylevel 值为0xFFFFFFFF的节点来标识的.

在__except_handler3获得当前的trylevel的指向相关联的SCOPETABLE 入口的代码点, 调用filter- expression的代码之后, 回到__except_handler3的代码. 如果filter-expression 返回EXCEPTION_CONTINUE_SEARCH, 那么__except_handler3会继续到下一个SCOPETABLE 的入口, 即previousTryLevel 域指定好了的入口. 如果通过遍历链表没有找到任何的handler, __except_handler3 会返回DISPOSITION_CONTINUE_SEARCH, 这会引发系统继续执行到下一个EXCEPTION_REGISTRATION 的frame.

如果filter-expression返回EXCEPTION_EXECUTE_HANDLER, 那这意味着异常应该被当前关联的_except代码块来处理. 这意味着任何前面的EXCEPTION_REGISTRATION真必须被从链表中移除, 并且_except代码块需要被执行. 这些琐事的第一是被名为__global_unwind2的函数处理的, 我会稍后解释它. 在一些其他的清理代码(我现在暂时忽略)执行过后, 代码的执行会离开__except_handler3并继续到_except 块. 奇怪的是控制从来没有回到过_except 块, 即使__except_handler3 函数明确地CALL它也不行.

当前的trylevel是如何设置的呢? 这是由编译器隐式地处理的, 编译器会对"扩展了的EXCEPTION_REGISTRATION结构"的trylevel域进行on-the-fly的修改. 如果你查看为使用SEH的函数生成的汇编代码, , 你会在函数的不同的点发现在[EBP-04]的修改当前trylevel的代码.

__except_handler3 是如何对_except代码进行CALL的动作, 而控制从来不会返回, 这是怎么做到的呢? 因为CALL指令push一个返回值到栈上, 你会觉得CALL某函数却不返回会弄乱掉栈的结构. 如果你查看一个为_except块生成的代码的话, 你会发现它所作的第一件事情就是从EXCEPTION_REGISTRATION结构往下8个字节的地方加载DWORD到ESP寄存器中. 作为这段开场代码的一部分, 函数保存了ESP到其他地方, 从而_except块可以晚些时候获取它.

The ShowSEHFrames Program

==========

如果现在你觉得像EXCEPTION_REGISTRATIONs, scopetables, trylevels, filter-expressions, 和unwinding这样的东西有那么一点难以接受的话, 我会告诉你这很正常. 刚开始的时候我也一样晕. 编译器等级的结构化异常处理的目标就不是让它能被人逐步地学习清楚. 除非你理解全部的细节, 那么它的很多组成部分对你是没有意义的. 当面对一堆理论的时候, 我的自然倾向是写一些能够应用我学到的东西的代码. 如果代码工作正常, 那么说明我的理解是正确的.

Figure 10 is the source code for ShowSEHFrames.EXE. It uses _try/_except blocks to set up a list of several Visual C++

SEH frames. Afterwards, it displays information about each frame, as well as the scopetables that Visual C++ builds for each frame. The program doesn't generate or expect any exceptions. Rather, I included all the _try blocks to force Visual C++ to generate multiple EXCEPTION_ REGISTRATION frames, with multiple scopetable entries per frame.

Figure 10是ShowSEHFrame.exe的源代码. 它使用_try/_except 块来建立一个几个Visual C++ SEH帧的链表. 之后, 它展示了每个帧的信息, 还有Visual C++为每个帧建立的scopetable. 这段程序并没有生成任何的异常. 值得注意的是, 我还让所有的_try块都强制Visual C++来生成多个EXCEPTION_ REGISTRATION帧, 每个帧还是用多个scopetable.

Figure 10 ShowSEHFrames.CPP

```
//==================================================
// ShowSEHFrames - Matt Pietrek 1997
// Microsoft Systems Journal, February 1997
// FILE: ShowSEHFrames.CPP
// To compile: CL ShowSehFrames.CPP
//==================================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#pragma hdrstop

//--------------------------------------------------------------------
// !!! WARNING !!!  This program only works with Visual C++, as the data
// structures being shown are specific to Visual C++.
//--------------------------------------------------------------------

#ifndef _MSC_VER
#error Visual C++ Required (Visual C++ specific information is displayed)
#endif

//--------------------------------------------------------------------
// Structure Definitions
//--------------------------------------------------------------------

// The basic, OS defined exception frame

struct EXCEPTION_REGISTRATION
{
   EXCEPTION_REGISTRATION* prev;
   FARPROC                 handler;
};


// Data structure(s) pointed to by Visual C++ extended exception frame

struct scopetable_entry
{
   DWORD       previousTryLevel;
   FARPROC     lpfnFilter;
   FARPROC     lpfnHandler;
};

// The extended exception frame used by Visual C++

struct VC_EXCEPTION_REGISTRATION : EXCEPTION_REGISTRATION
{
   scopetable_entry *  scopetable;
   int                 trylevel;
   int                 _ebp;
};

//--------------------------------------------------------------------
// Prototypes
//--------------------------------------------------------------------

// __except_handler3 is a Visual C++ RTL function.  We want to refer to
// it in order to print it's address.  However, we need to prototype it since
// it doesn't appear in any header file.

extern "C" int _except_handler3(PEXCEPTION_RECORD, EXCEPTION_REGISTRATION *,
                                PCONTEXT, PEXCEPTION_RECORD);


//--------------------------------------------------------------------
// Code
```

```
//---------------------------------------------------------------------

//
// Display the information in one exception frame, along with its scopetable
//

void ShowSEHFrame( VC_EXCEPTION_REGISTRATION * pVCExcRec )
{
   printf( "Frame: %08X  Handler: %08X  Prev: %08X  Scopetable: %08X\n",
         pVCExcRec, pVCExcRec->handler, pVCExcRec->prev,
         pVCExcRec->scopetable );

   scopetable_entry * pScopeTableEntry = pVCExcRec->scopetable;

   for ( unsigned i = 0; i <= pVCExcRec->trylevel; i++ )
   {
      printf( "    scopetable[%u] PrevTryLevel: %08X  "
            "filter: %08X  __except: %08X\n", i,
            pScopeTableEntry->previousTryLevel,
            pScopeTableEntry->lpfnFilter,
            pScopeTableEntry->lpfnHandler );

      pScopeTableEntry++;
   }

   printf( "\n" );
}

//
// Walk the linked list of frames, displaying each in turn
//

void WalkSEHFrames( void )
{
   VC_EXCEPTION_REGISTRATION * pVCExcRec;

   // Print out the location of the __except_handler3 function
   printf( "_except_handler3 is at address: %08X\n", _except_handler3 );
   printf( "\n" );

   // Get a pointer to the head of the chain at FS:[0]
   __asm   mov eax, FS:[0]
   __asm   mov [pVCExcRec], EAX

   // Walk the linked list of frames.  0xFFFFFFFF indicates the end of list
   while (  0xFFFFFFFF != (unsigned)pVCExcRec )
   {
      ShowSEHFrame( pVCExcRec );
      pVCExcRec = (VC_EXCEPTION_REGISTRATION *)(pVCExcRec->prev);
   }
}

void Function1( void )
{
   // Set up 3 nested _try levels (thereby forcing 3 scopetable entries)
   _try
   {
      _try
      {
         _try
         {
            WalkSEHFrames();    // Now show all the exception frames
         }
         _except( EXCEPTION_CONTINUE_SEARCH )
         {
         }
      }
      _except( EXCEPTION_CONTINUE_SEARCH )
      {
      }
   }
   _except( EXCEPTION_CONTINUE_SEARCH )
   {
   }
}

int main()
{
```

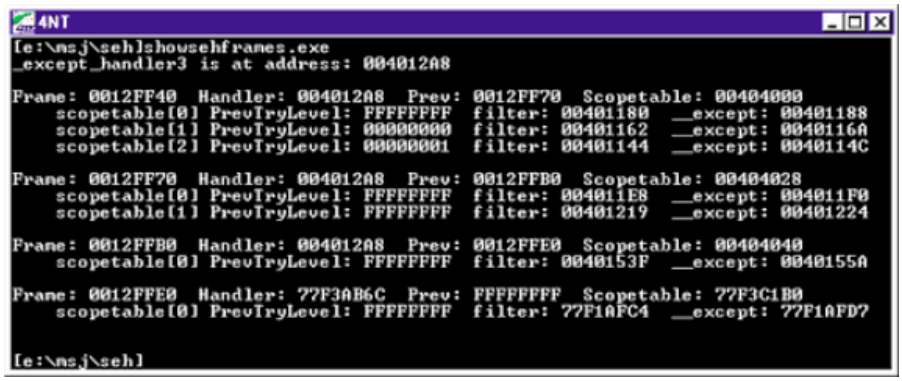
```
    int i;

    // Use two (non-nested) _try blocks.  This causes two scopetable entries
    // to be generated for the function.

    _try
    {
        i = 0x1234;     // Do nothing in particular
    }
    _except( EXCEPTION_CONTINUE_SEARCH )
    {
        i = 0x4321;     // Do nothing (in reverse)
    }

    _try
    {
        Function1();    // Call a function that sets up more exception frames
    }
    _except( EXCEPTION_EXECUTE_HANDLER )
    {
        // Should never get here, since we aren't expecting an exception
        printf( "Caught Exception in main\n" );
    }

    return 0;
}
```

ShowSEHFrames 里的重要函数是WalkSEHFrames 和ShowSEHFrame. WalkSEHFrames 首先打印出__except_handler3的地址, 这么做的原因稍后就会清楚. 下一步, 函数从FS:[0]中获得了一个指针, 保存到了exception list的头节点中. 每个节点的类型都是VC_EXCEPTION_REGISTRATION, 这是我定义的用来描述Visual C++异常处理帧的一个结构. 对于链表中的每个节点, WalkSEHFrames 都会传一个指向节点的指针给ShowSEHFrame 函数.

ShowSEHFrame 函数通过打印exception frame的地址, handler callback的地址, 前一个exception frame的地址, 还有指向scopetable的指针来开始. 接下来, 对每个scopetable入口, 代码都打印出前一个trylevel, 打印出filter-expression 的地址, 还有_except块的地址. 我怎么知道scopetable中有多少条目呢? 我也不知道. 但是我假设当前的VC_EXCEPTION_REGISTRATION 结构中的trylevel的数量少于scopetable条目的总数.

Figure 11展现了运行ShowSEHFrames的结果. 首先, 看看每个用"Frame:"开头的行吧. 注意每个连续的实例如何展现在栈的更高地址的exception frame的. 接下来, 在头三个Frame: 行, 注意Handler的值是相同的(004012A8). 看看输出的开头的部分, 你会看到这个004012A8 不是别的, 正是Visual C++ runtime library里的__except_handler3的地址. 这证明了我早些时候的断言: 所有的exception都被同一个入口点来处理.

Figure 11 Running ShowSEHFrames



你可能在想, 为什么有三个exception frame使用__except_handler3 作为他们的callback, 而ShowSEHFrames 仅仅有两个函数使用SEH. 答案是第三个frame来自Visual C++ runtime library. 在Visual C++ runtime library 源代码CRT0.C的代码中, 调用main或WinMain函数的代码也包装在了_try/_except 块当中了. 针对这个_try块的filter-expression代码可以在WINXFLTR.C文件中找到.

回到ShowSEHFrames, 最后一帧的handler的那一行包含了一个不同的地址, 即77F3AB6C. 到处逛逛, 到处试试, 你会发现这个地址在KERNEL32.DLL中. 这个特别的frame是由KERNEL32.DLL在BaseProcessStart 函数(前面我描述过的)里安装的.

Unwinding

==========

在深入挖掘unwinding的实现代码之前, 让我们简单地回顾一下unwinding是什么意思吧. 之前, 我描述了潜在的exception handler是如何存储在一个链表中的了, 它被一个线程信息块(Thread Information Block)的第一个DWORD(FS:[0])所指向. 因为某个特定异常的handler可能不在链表的头节点, 那么就需要一个秩序来移除列表中的实际处理该异常的handler的前面的所有的exception handler.

正如你在Visual C++ 的__except_handler3 函数中看到的, unwinding是由__global_unwind2 这个RTL函数执行的. 这个函数仅仅是一个对未归档的RtlUnwind这个API的非常简单的包装.

```
__global_unwind2(void * pRegistFrame)
{
   _RtlUnwind( pRegistFrame,
            &__ret_label,
            0, 0 );
   __ret_label:
}
```

虽然RtlUnwind 是实现编译器水平SEH的关键API, 但是它并没有在任何的文档中出现. 技术上来说, RtlUnwind 这个KERNEL32 函数, 即Windows NT KERNEL32 .DLL会把这个Call 发送到NTDLL.DLL, 而NTDLL.DLL也有一个RtlUnwind 函数. 我能做出一些伪代码来说明它, 请看**Figure 12**.

Figure 12 RtlUnwind Pseudocode

```
void _RtlUnwind( PEXCEPTION_REGISTRATION pRegistrationFrame,
             PVOID returnAddr,  // Not used! (At least on i386)
             PEXCEPTION_RECORD pExcptRec,
             DWORD _eax_value )
{
   DWORD   stackUserBase;
   DWORD   stackUserTop;
   PEXCEPTION_RECORD pExcptRec;
   EXCEPTION_RECORD  exceptRec;
   CONTEXT context;

   // Get stack boundaries from FS:[4] and FS:[8]
   RtlpGetStackLimits( &stackUserBase, &stackUserTop );

   if ( 0 == pExcptRec )   // The normal case
   {
      pExcptRec = &exceptRec;

      pExcptRec->ExceptionFlags = 0;
      pExcptRec->ExceptionCode = STATUS_UNWIND;
      pExcptRec->ExceptionRecord = 0;
      // Get return address off the stack
      pExcptRec->ExceptionAddress = RtlpGetReturnAddress();
      pExcptRec->ExceptionInformation[0] = 0;
   }

   if ( pRegistrationFrame )
      pExcptRec->ExceptionFlags |= EXCEPTION_UNWINDING;
   else
      pExcptRec->ExceptionFlags|=(EXCEPTION_UNWINDING|EXCEPTION_EXIT_UNWIND);

   context.ContextFlags =
      (CONTEXT_i486 | CONTEXT_CONTROL | CONTEXT_INTEGER | CONTEXT_SEGMENTS);

   RtlpCaptureContext( &context );

   context.Esp += 0x10;
   context.Eax = _eax_value;

   PEXCEPTION_REGISTRATION pExcptRegHead;

   pExcptRegHead = RtlpGetRegistrationHead();  // Retrieve FS:[0]
```

```
// Begin traversing the list of EXCEPTION_REGISTRATION
while ( -1 != pExcptRegHead )
{
   EXCEPTION_RECORD excptRec2;

   if ( pExcptRegHead == pRegistrationFrame )
   {
      _NtContinue( &context, 0 );
   }
   else
   {
      // If there's an exception frame, but it's lower on the stack
      // then the head of the exception list, something's wrong!
      if ( pRegistrationFrame && (pRegistrationFrame <= pExcptRegHead) )
      {
         // Generate an exception to bail out
         excptRec2.ExceptionRecord = pExcptRec;
         excptRec2.NumberParameters = 0;
         excptRec2.ExceptionCode = STATUS_INVALID_UNWIND_TARGET;
         excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

         _RtlRaiseException( &exceptRec2 );
      }
   }

   PVOID pStack = pExcptRegHead + 8; // 8==sizeof(EXCEPTION_REGISTRATION)

   if (    (stackUserBase <= pExcptRegHead )   // Make sure that
      &&  (stackUserTop >= pStack )          // pExcptRegHead is in
      &&  (0 == (pExcptRegHead & 3)) )       // range, and a multiple
   {                                  // of 4 (i.e., sane)
      DWORD pNewRegistHead;
      DWORD retValue;

      retValue = RtlpExecutehandlerForUnwind(
                  pExcptRec, pExcptRegHead, &context,
                  &pNewRegistHead, pExceptRegHead->handler );

      if ( retValue != DISPOSITION_CONTINUE_SEARCH )
      {
         if ( retValue != DISPOSITION_COLLIDED_UNWIND )
         {
            excptRec2.ExceptionRecord = pExcptRec;
       excptRec2.NumberParameters = 0;
            excptRec2.ExceptionCode = STATUS_INVALID_DISPOSITION;
            excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

            RtlRaiseException( &excptRec2 );
         }
         else
            pExcptRegHead = pNewRegistHead;
      }

      PEXCEPTION_REGISTRATION pCurrExcptReg = pExcptRegHead;
      pExcptRegHead = pExcptRegHead->prev;

      RtlpUnlinkHandler( pCurrExcptReg );
   }
   else    // The stack looks goofy!  Raise an exception to bail out
   {
      excptRec2.ExceptionRecord = pExcptRec;
      excptRec2.NumberParameters = 0;
      excptRec2.ExceptionCode = STATUS_BAD_STACK;
      excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;

      RtlRaiseException( &excptRec2 );
   }
}

// If we get here, we reached the end of the EXCEPTION_REGISTRATION list.
// This shouldn't happen normally.

if ( -1 == pRegistrationFrame )
   NtContinue( &context, 0 );
else
   NtRaiseException( pExcptRec, &context, 0 );
```

```
}

PEXCEPTION_REGISTRATION
RtlpGetRegistrationHead( void )
{
   return FS:[0];
}

_RtlpUnlinkHandler( PEXCEPTION_REGISTRATION pRegistrationFrame )
{
   FS:[0] = pRegistrationFrame->prev;
}

void _RtlpCaptureContext( CONTEXT * pContext )
{
   pContext->Eax = 0;
   pContext->Ecx = 0;
   pContext->Edx = 0;
   pContext->Ebx = 0;
   pContext->Esi = 0;
   pContext->Edi = 0;
   pContext->SegCs = CS;
   pContext->SegDs = DS;
   pContext->SegEs = ES;
   pContext->SegFs = FS;
   pContext->SegGs = GS;
   pContext->SegSs = SS;
   pContext->EFlags = flags; // __asm{ PUSHFD / pop [xxxxxxxx] }
   pContext->Eip = return address of the caller of the caller of this function
   pContext->Ebp = EBP of the caller of the caller of this function
   pContext->Esp = Context.Ebp + 8
}
```

虽然RtlUnwind 看起来挺威风, 但是如果你有条不紊地分开来看, 它还是不难理解的. 这个API通过从FS:[4]和FS:[8]获得当前线程的栈的栈头和栈尾来开始. 这些值很重要, 因为需要用它们来确保所有的exception frame展开后都在栈的范围之内, 这样做是理智的.

RtlUnwind 之后在栈上建立了一个哑的EXCEPTION_RECORD, 并设置它的ExceptionCode为STATUS_UNWIND. 还有, EXCEPTION_UNWINDING 被设置成了EXCEPTION_RECORD结构中的ExceptionFlags 域的值. 一个指向这个结构的指针会在稍后作为一个参数传递给每个exception callback. 之后, 代码调用_RtlpCaptureContext 函数来创建一个哑的CONTEXT 结构, 这个就结构也成为了为了unwind exception callstack的一个参数.

RtlUnwind 的其余部分遍历EXCEPTION_REGISTRATION结构的链表. 对每一帧, 代码都调用RtlpExecuteHandlerForUnwind 函数, 我稍后会解释这个函数. 正是这个函数调用了exception callback, 并且还设置了EXCEPTION_UNWINDING 标志. 每个callback之后, 关联的exception frame都会被RtlpUnlinkHandler的调用所移除.

RtlUnwind 会在它到达作为传给他的第一个参数的位置是停止unwinding frames. 穿插在代码中, 我已经描述了确保万无一失的sanity-checking(理性检查). 如果某个意外问题突然发生, 那么RtlUnwind 会生成一个异常来说明出现了什么问题, 并且这个异常会包含EXCEPTION_NONCONTINUABLE 标志. 一个进程的这个标志位被设置了的话, 那么这个进程是不允许再继续执行的了, 所以该进程必须被杀掉.

Unhandled Exceptions

==========

在这篇文章的前面, 我推迟了对UnhandledExceptionFilter API的完整描述. 一般情况下, 你不会直接调用这个API的(尽管你能够这么做). 多数时候, 它是被KERNEL32的默认异常回调的filter-expression 的代码所激活的. 我在前面的BaseProcessStart的伪代码中已经展现了.

Figure 13展现了我写的UnhandledExceptionFilter的伪代码. 这个API的开始有点奇怪(至少我这么认为). 假设错误是一个EXCEPTION_ACCESS_ VIOLATION, 那么代码会调用_BasepCheckForReadOnlyResource. 虽然我还没提供这个函数的伪代码, 但是我可以总结它一下. 如果异常发生是由于一个EXE或DLL的resource section(.rsrc)被写入了, 那么_BasepCurrentTopLevelFilter会修改错误页面的本来的正常的只读属性, 从而允许写入发生. 如果这个特定的场景发生的话, UnhandledExceptionFilter 会返回EXCEPTION_ CONTINUE_EXECUTION, 并且执行会在错误的指令处重新执行.

Figure 13 UnHandledExceptionFilter Pseudocode

```
UnhandledExceptionFilter( STRUCT _EXCEPTION_POINTERS *pExceptionPtrs )
{
   PEXCEPTION_RECORD pExcptRec;
   DWORD currentESP;
   DWORD retValue;
   DWORD DEBUGPORT;
   DWORD   dwTemp2;
   DWORD   dwUseJustInTimeDebugger;
   CHAR    szDbgCmdFmt[256];   // Template string retrieved from AeDebug key
   CHAR    szDbgCmdLine[256];  // Actual debugger string after filling in
   STARTUPINFO startupinfo;
   PROCESS_INFORMATION pi;
   HARDERR_STRUCT  harderr;    // ???
   BOOL fAeDebugAuto;
   TIB * pTib;               // Thread information block

   pExcptRec = pExceptionPtrs->ExceptionRecord;

   if (   (pExcptRec->ExceptionCode == EXCEPTION_ACCESS_VIOLATION)
      && (pExcptRec->ExceptionInformation[0]) )
   {
      retValue =
         _BasepCheckForReadOnlyResource(pExcptRec->ExceptionInformation[1]);

      if ( EXCEPTION_CONTINUE_EXECUTION == retValue )
         return EXCEPTION_CONTINUE_EXECUTION;
    }

   // See if this process is being run under a debugger...
   retValue = NtQueryInformationProcess(GetCurrentProcess(), ProcessDebugPort,
                                &debugPort, sizeof(debugPort), 0 );

   if ( (retValue >= 0) && debugPort )     // Let debugger have it
      return EXCEPTION_CONTINUE_SEARCH;

   // Did the user call SetUnhandledExceptionFilter?  If so, call their
   // installed proc now.

   if ( _BasepCurrentTopLevelFilter )
   {
      retValue = _BasepCurrentTopLevelFilter( pExceptionPtrs );

      if ( EXCEPTION_EXECUTE_HANDLER == retValue )
         return EXCEPTION_EXECUTE_HANDLER;

      if ( EXCEPTION_CONTINUE_EXECUTION == retValue )
         return EXCEPTION_CONTINUE_EXECUTION;

      // Only EXCEPTION_CONTINUE_SEARCH goes on from here
   }

   // Has SetErrorMode(SEM_NOGPFAULTERRORBOX) been called?
   if ( 0 == (GetErrorMode() & SEM_NOGPFAULTERRORBOX) )
   {
      harderr.elem0 = pExcptRec->ExceptionCode;
      harderr.elem1 = pExcptRec->ExceptionAddress;

      if ( EXCEPTION_IN_PAGE_ERROR == pExcptRec->ExceptionCode )
         harderr.elem2 = pExcptRec->ExceptionInformation[2];
      else
         harderr.elem2 = pExcptRec->ExceptionInformation[0];

      dwTemp2 = 1;
      fAeDebugAuto = FALSE;

      harderr.elem3 = pExcptRec->ExceptionInformation[1];

      pTib = FS:[18h];

      DWORD someVal = pTib->pProcess->0xC;

      if ( pTib->threadID != someVal )
      {
         __try
```

```
        {
            char szDbgCmdFmt[256]
            retValue = _GetProfileStringA( "AeDebug", "Debugger", 0,
                            szDbgCmdFmt, sizeof(szDbgCmdFmt)-1 );

            if ( retValue )
                dwTemp2 = 2;

            char szAuto[8]

            retValue = GetProfileStringA(   "AeDebug", "Auto", "0",
                                    szAuto, sizeof(szAuto)-1 );
            if ( retValue )
                if ( 0 == strcmp( szAuto, "1" ) )
                    if ( 2 == dwTemp2 )
                        fAeDebugAuto = TRUE;
        }
        __except( EXCEPTION_EXECUTE_HANDLER )
        {
            ESP = currentESP;
            dwTemp2 = 1
            fAeDebugAuto = FALSE;
        }
    }

    if ( FALSE == fAeDebugAuto )
    {
        retValue =  NtRaiseHardError(
                        STATUS_UNHANDLED_EXCEPTION | 0x10000000,
                        4, 0, &harderr,
                        _BasepAlreadyHadHardError ? 1 : dwTemp2,
                        &dwUseJustInTimeDebugger );
    }
    else
    {
        dwUseJustInTimeDebugger = 3;
        retValue = 0;
    }

    if (    retValue >= 0
        &&  ( dwUseJustInTimeDebugger == 3)
        &&  ( !_BasepAlreadyHadHardError )
        &&  ( !_BaseRunningInServerProcess ) )
    {
        _BasepAlreadyHadHardError = 1;

        SECURITY_ATTRIBUTES secAttr = { sizeof(secAttr), 0, TRUE };

        HANDLE hEvent = CreateEventA( &secAttr, TRUE, 0, 0 );

        memset( &startupinfo, 0, sizeof(startupinfo) );

        sprintf(szDbgCmdLine, szDbgCmdFmt, GetCurrentProcessId(), hEvent);

        startupinfo.cb = sizeof(startupinfo);
        startupinfo.lpDesktop = "Winsta0\Default"

        CsrIdentifyAlertableThread();   // ???

        retValue = CreateProcessA(
                    0,              // lpApplicationName
                    szDbgCmdLine,   // Command line
                    0, 0,           // process, thread security attrs
                    1,              // bInheritHandles
                    0, 0,           // creation flags, environment
                    0,              // current directory.
                    &statupinfo,    // STARTUPINFO
                    &pi );          // PROCESS_INFORMATION

        if ( retValue && hEvent )
        {
            NtWaitForSingleObject( hEvent, 1, 0 );
            return EXCEPTION_CONTINUE_SEARCH;
        }
    }

    if ( _BasepAlreadyHadHardError )
        NtTerminateProcess(GetCurrentProcess(), pExcptRec->ExceptionCode);
```

```
    }

    return EXCEPTION_EXECUTE_HANDLER;
}

LPTOP_LEVEL_EXCEPTION_FILTER
SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter );
{
    // _BasepCurrentTopLevelFilter is a KERNEL32.DLL global var
    LPTOP_LEVEL_EXCEPTION_FILTER previous= _BasepCurrentTopLevelFilter;

    // Set the new value
    _BasepCurrentTopLevelFilter = lpTopLevelExceptionFilter;

    return previous;    // return the old value
}
```

下一个UnhandledExceptionFilter 的任务是确定进程是否运行在一个Win32 Debugger之下. 如果是的话, 进程的DEBUG_PROCESS或BUG_ONLY_THIS_PROCESS 标志在进程创建时就被标好. UnhandledExceptionFilter 使用NtQueryInformationProcess 函数(该函数我在这个月的Under the Hood专栏里描述过)来判断进程是否正在被debug. 如果是的话, 那么这个API就返回EXCEPTION_CONTINUE_SEARCH, 这就会告诉系统的其他部分来唤醒debugger进程, 并且告诉debugger进程debuggee里发生了一个异常.

接下来, UnhandledExceptionFilter's 会调用user-installed的为unhandled exception filter, 如果有的话. 正常情况下, user-installed的callback函数不存在的, 但是用户可以通过SetUnhandledExceptionFilter 这个API来安装一个. 我也为这个API提供了伪代码. 这个API简单地使用"用户的新callback函数的地址"来替换一个global variable, 并且返回老callback的值.

有了前面基础问题的解决, UnhandledExceptionFilter 能够开始做它的工作了: 使用永久时尚的应用程序错误对话框来通知你你的可耻的编程大错. :) 有两种方法可以避免这个对话框.
第一, 如果进程调用了SetErrorMode 并且指定了SEM_NOGPFAULTERRORBOX 标志.
第二, AeDebug注册表的Auto键设为1. 在这种情况下, UnhandledExceptionFilter 会跳过应用程序错误对话框, 并自动开启在AeDebug 注册表的Debugger键指定的debugger. 如果你熟悉"just in time debugging", 这就是操作系统支持它的地方了. 晚些时候在继续这个部分.

在多数情况下, 这两种避免对话框的情况都不为true, 并且UnhandledExceptionFilter 会调用NTDLL.DLL中的NtRaiseHardError方法. 这个方法会带来Application Error 对话框. 这个对话框等你点击OK来结束进程, 或者点击Cancel来debug它. (或许只是我这样想吧, 点击cancel来开启debugger看起来有点像是退步)

如果你点击Application Error对话框的OK按钮, UnhandledExceptionFilter 会返回EXCEPTION_EXECUTE_HANDLER. 调用UnhandledExceptionFilter 的代码通常会通过结束它自己来响应(正如你在BaseProcessStart 的代码中看到的). 这带来了一个有趣的要点. 大多数人假设系统会使用一个unhandled exception来结束进程. 其实更正确的方法是, 系统配置好了一切, 以便于unhandled exception能够引发进程终结它自己.
UnhandledExceptionFilter 的执行中真正有趣的代码是如果你在Application Error 对话框中选择Cancel, 从而为出错的进程开启一个debugger. 代码首先调用CreateEvent 来制作一个event来通知debugger来attach到出错的进程上. 这个event的句柄, 还有当前进程的ProcessID, 会被传送给sprintf函数, 它会在event中格式化之前创建的NtWaitForSingleObject. 这个调用会阻塞进程直到debugger通知event, 告诉它debugger已经成功地attach到了出错的进程之上. UnhandledExceptionFilter 的代码中还有些小零碎, 但是我这里已经介绍了所有重要的要点了.

地狱深处
==========
如果你已经看到了这么远的地方, 那么如果不完成整个的线路对你就太不公平了. 我已经展现了操作系统如何在异常发生时调用用户定义的函数. 我也展现了那些callback函数的特别之处, 还有编译器如何使用他们来实现_try和_catch的了. 我曾经展现了当没有人处理异常的时候, 系统不得不做一些收尾工作. 所剩下的就是展现一开始exception callback起源于何处了. 是的, 让我们一个猛子扎到系统的肠子里来看看结构化异常处理的顺序吧.

Figure 14展现了一些我为KiUserExceptionDispatcher 写的伪代码, 还有一些相关的函数. KiUserExceptionDispatcher 函数存在于NTDLL.DLL, 它就是异常发生后执行开始的地方. 要百分百准确的话, 我刚才说的也不算正确. 打比方说, 在Intel架构中, 一个异常引发控制向量转移到ring 0(内核态)的handler中. Handler是由interrupt descriptor 表中的关联到某个异常的一行定义的. 我会跳过所有的内核态代码, 并假装CPU直接在exception的情况下调用到KiUserExceptionDispatcher 中.

Figure 14 KiUserExceptionDispatcher Pseudocode

```
KiUserExceptionDispatcher( PEXCEPTION_RECORD pExcptRec, CONTEXT * pContext )
{
   DWORD retValue;

   // Note: If the exception is handled, RtlDispatchException() never returns
   if ( RtlDispatchException( pExceptRec, pContext ) )
      retValue = NtContinue( pContext, 0 );
   else
      retValue = NtRaiseException( pExceptRec, pContext, 0 );

   EXCEPTION_RECORD excptRec2;

   excptRec2.ExceptionCode = retValue;
   excptRec2.ExceptionFlags = EXCEPTION_NONCONTINUABLE;
   excptRec2.ExceptionRecord = pExcptRec;
   excptRec2.NumberParameters = 0;

   RtlRaiseException( &excptRec2 );
}

int RtlDispatchException( PEXCEPTION_RECORD pExcptRec, CONTEXT * pContext )
{
   DWORD   stackUserBase;
   DWORD   stackUserTop;
   PEXCEPTION_REGISTRATION pRegistrationFrame;
   DWORD hLog;

   // Get stack boundaries from FS:[4] and FS:[8]
   RtlpGetStackLimits( &stackUserBase, &stackUserTop );

   pRegistrationFrame = RtlpGetRegistrationHead();

   while ( -1 != pRegistrationFrame )
   {
      PVOID justPastRegistrationFrame = &pRegistrationFrame + 8;
      if ( stackUserBase > justPastRegistrationFrame )
      {
         pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
         return DISPOSITION_DISMISS; // 0
      }

      if ( stackUsertop < justPastRegistrationFrame )
      {
         pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
         return DISPOSITION_DISMISS; // 0
      }

      if ( pRegistrationFrame & 3 )   // Make sure stack is DWORD aligned
      {
         pExcptRec->ExceptionFlags |= EH_STACK_INVALID;
         return DISPOSITION_DISMISS; // 0
      }

      if ( someProcessFlag )
      {
         // Doesn't seem to do a whole heck of a lot.
         hLog = RtlpLogExceptionHandler( pExcptRec, pContext, 0,
                                 pRegistrationFrame, 0x10 );
      }

      DWORD retValue, dispatcherContext;

      retValue= RtlpExecuteHandlerForException(pExcptRec, pRegistrationFrame,
                                    pContext, &dispatcherContext,
                                    pRegistrationFrame->handler );
```

```
      // Doesn't seem to do a whole heck of a lot.
      if ( someProcessFlag )
         RtlpLogLastExceptionDisposition( hLog, retValue );

      if ( 0 == pRegistrationFrame )
      {
         pExcptRec->ExceptionFlags &= ~EH_NESTED_CALL;   // Turn off flag
      }

      EXCEPTION_RECORD excptRec2;

      DWORD yetAnotherValue = 0;

      if ( DISPOSITION_DISMISS == retValue )
      {
         if ( pExcptRec->ExceptionFlags & EH_NONCONTINUABLE )
         {
            excptRec2.ExceptionRecord = pExcptRec;
            excptRec2.ExceptionNumber = STATUS_NONCONTINUABLE_EXCEPTION;
            excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
            excptRec2.NumberParameters = 0
            RtlRaiseException( &excptRec2 );
         }
         else
            return DISPOSITION_CONTINUE_SEARCH;
      }
      else if ( DISPOSITION_CONTINUE_SEARCH == retValue )
      {
      }
      else if ( DISPOSITION_NESTED_EXCEPTION == retValue )
      {
         pExcptRec->ExceptionFlags |= EH_EXIT_UNWIND;
         if ( dispatcherContext > yetAnotherValue )
            yetAnotherValue = dispatcherContext;
      }
      else    // DISPOSITION_COLLIDED_UNWIND
      {
         excptRec2.ExceptionRecord = pExcptRec;
         excptRec2.ExceptionNumber = STATUS_INVALID_DISPOSITION;
         excptRec2.ExceptionFlags = EH_NONCONTINUABLE;
         excptRec2.NumberParameters = 0
         RtlRaiseException( &excptRec2 );
      }

      pRegistrationFrame = pRegistrationFrame->prev;  // Go to previous frame
   }

   return DISPOSITION_DISMISS;
}


_RtlpExecuteHandlerForException:    // Handles exception (first time through)
   MOV     EDX,XXXXXXXX
   JMP     ExecuteHandler


RtlpExecutehandlerForUnwind:        // Handles unwind (second time through)
   MOV     EDX,XXXXXXXX



int ExecuteHandler( PEXCEPTION_RECORD pExcptRec
                    PEXCEPTION_REGISTRATION pExcptReg
                    CONTEXT * pContext
                    PVOID pDispatcherContext,
                    FARPROC handler ) // Really a ptr to an _except_handler()

   // Set up an EXCEPTION_REGISTRATION, where EDX points to the
   // appropriate handler code shown below
   PUSH    EDX
   PUSH    FS:[0]
   MOV     FS:[0],ESP

   // Invoke the exception callback function
   EAX = handler( pExcptRec, pExcptReg, pContext, pDispatcherContext );

   // Remove the minimal EXCEPTION_REGISTRATION frame
   MOV     ESP,DWORD PTR FS:[00000000]
```

```
   POP     DWORD PTR FS:[00000000]

   return EAX;
}

Exception handler used for _RtlpExecuteHandlerForException:
{
   // If unwind flag set, return DISPOSITION_CONTINUE_SEARCH, else
   // assign pDispatcher context and return DISPOSITION_NESTED_EXCEPTION

   return pExcptRec->ExceptionFlags & EXCEPTION_UNWIND_CONTEXT
            ? DISPOSITION_CONTINUE_SEARCH
            : *pDispatcherContext = pRegistrationFrame->scopetable,
              DISPOSITION_NESTED_EXCEPTION;
}

Exception handler used for _RtlpExecuteHandlerForUnwind:
{
   // If unwind flag set, return DISPOSITION_CONTINUE_SEARCH, else
   // assign pDispatcher context and return DISPOSITION_COLLIDED_UNWIND

   return pExcptRec->ExceptionFlags & EXCEPTION_UNWIND_CONTEXT
            ? DISPOSITION_CONTINUE_SEARCH
            : *pDispatcherContext = pRegistrationFrame->scopetable,
              DISPOSITION_COLLIDED_UNWIND;
}
```

在KiUserExceptionDispatcher 的深处是对RtlDispatchException的调用. 这启动了对任何注册了的exception handler的搜索. 如果一个handler处理了异常并且继续执行, 那么对RtlDispatchException 的调用就永远不会返回. 如果RtlDispatchException 返回了, 就有两种可能的路径:要么NtContinue 被调用了, 它使得进程继续执行. 要么就是另一个异常发生了. 这时候, 异常不是可继续的, 并且进程必须终止.

继续说RtlDispatchExceptionCode, 这是你会发现exception frame walking代码的地方, 整篇文章我都在提. 这个函数抓住一个指向EXCEPTION_REGISTRATIONs 链表的指针, 并且遍历所有的节点, 查找handler. 因为有栈崩溃的可能性, 这个过程非常偏执, 让人不爽. 在调用每个EXCEPTION_REGISTRATION指定的handler之前, 代码会确保EXCEPTION_REGISTRATION 是DWORD方式对齐(DWORD-aligned)的, 对齐发生在线程的栈中, 并且下一个EXCEPTION_REGISTRATION的位置要比前一个EXCEPTION_REGISTRATION处于更高的地址.

RtlDispatchException 并不直接地调用EXCEPTION_REGISTRATION 结构中指定的地址. 替代的是, 它会调用RtlpExecuteHandlerForException 来做掉这脏活儿. 取决于RtlpExecuteHandlerForException内部发生了什么, RtlDispatchException 要么继续遍历所有的异常frame, 要么引发另一个异常. 这里的第二个异常指明了在exception callback中有什么不好的事情发生了, 并且执行不能继续下去了.

RtlpExecuteHandlerForException 的代码跟另一个函数关联得很紧, 即RtlpExecutehandlerForUnwind. 你会回想起我早些时候描述unwinding的时候提到过这个函数. 这两个"函数"简单地在把控制传给ExecuteHandler 函数之前使用不同的值来加载到EDX寄存器中. 换种说法, RtlpExecuteHandlerForException 和RtlpExecutehandlerForUnwind 是ExecuteHandler函数的分开的独立的前端而已.

ExecuteHandler 是EXCEPTION_REGISTRATION中抽取出来的handler域, 并且这个域会被调用. 像它看起来一样奇怪, 对exception callback的调用是有一个结构化的exception handler"自我封装"的. 在它自己内部使用SEH看起来有点可笑, 但是如果你思考它多一点的话会发现这是合理的. 如果一个exception callback引发了另一个异常, 那么操作系统需要知道这一点. 取决于异常是否发生在开始的callback中还是发生在unwind callback中, ExecuteHandler 要么返回DISPOSITION_NESTED_ EXCEPTION , 要么返回DISPOSITION_COLLIDED_UNWIND. 这两个基本上意思都是"红色警报! 现在就关闭所有的东西!" 的代码了.

如果你像我一样, 现在已经很难直接把所有的这些函数直接跟SEH联系在一起了. 同样的, 你也很难记住谁调用了谁. 为了帮助我自己, 我搞了一个小图表在下面的**Figure 15**里.

Figure 15 Who Calls Who in SEH

```
KiUserExceptionDispatcher()

   RtlDispatchException()
```

```
    RtlpExecuteHandlerForException()
        ExecuteHandler() // Normally goes to __except_handler3
---------
__except_handler3()
  scopetable filter-expression()
  __global_unwind2()
    RtlUnwind()
      RtlpExecuteHandlerForUnwind()
  scopetable __except block()
```

现在, 看看在到达ExecuteHandler 代码之前设置EDX是什么意思吧. 很简单, 真的. 如果调用user-installed的handler的时候出了什么问题, 那么ExecuteHandler 会使用EDX中的任何东西作为原始的exception handler. 它将EDX寄存器压到栈上, 让它作为一个最小的EXCEPTION_REGISTRATION 的handler数据域. 本质上说, ExecuteHandler 使用原始的structured exception handling, 就跟我在MYSEH 和MYSEH2程序中展示的一样.

结论

==========

结构化异常处理是Win32的一个美妙的特性. 感谢诸如Visual C++一类的编译器放在上面的支持的层次, 一般的程序员都可以花相对较少的学习上的投资从SEH中获益. 然而, 在操作系统层次, 事情就比你在Win32文档想要你相信的东西复杂得多了.

不幸的是, 目前并没有什么系统级别的SEH写出来公布给大家, 因为绝大多数人都认为这一块是个极端困难的话题. 系统等级的细节的文档缺乏不能帮什么忙. 在这篇文章中, 我围绕着一个简单的callback展现了操作系统等级的SEH. 如果你理解了SEH的原理, 还有在其上面建立起来的各个层次, 系统等级的结构化异常处理也不难理解了.

原文地址:

A Crash Course on the Depths of Win32™ Structured Exception Handling

http://www.microsoft.com/msj/0197/Exception/Exception.aspx

相关链接:

CLR 中未处理异常的处置

http://msdn.microsoft.com/zh-cn/magazine/cc793966.aspx