第35卷 第19期 计 算 机 工 程 2009年10月
Vol.35 No.19 Computer Engineering October 2009
·安全技术· 文章编号：1000—3428(2009)19—0157—04 文献标识码：A 中图分类号：TP311.5

Windows系统异常处理机制的研究及应用

张 明，徐万里
(装甲兵工程学院信息工程系，北京 100072)

摘 要：从分析梳理Windows平台下异常处理机制的种类和特点入手，探讨它们的内部机理，特别是反汇编的实现和特征，研究它们在软件安全领域中的重要应用，从而说明异常处理机制除了可以帮助处理软件中出现的错误和异常外，还在软件安全领域大有作为，为进一步研究异常处理机制和扩展其应用范围打下基础。

关键词：异常处理；反调试；堆栈溢出；软件漏洞

Research and Application of Windows System Exception Handling Mechanism

ZHANG Ming, XU Wan-li
(Department of Information Engineering, Academy of Armed Force Engineering, Beijing 100072)

【Abstract】 The paper classifies the different kinds of exception handling mechanism according to their features and discusses their internals, especially their disassembling realization and characteristics. It further focuses on their important application in software security field. Through the research, the paper explains that it can use exception handling mechanism in more fields besides handling different kinds of errors and exceptions, which can help to further study and expand application's fields.

【Key words】 exception handling; anti-debugging; stack buffer overflow; software holes

Windows系统的异常处理机制[1]除了能够帮助软件开发人员发现和解决软件中的错误外，还被广泛应用于软件保护技术[2]、软件漏洞利用[3-4]等方面。因此，深入研究异常处理机制的原理和实现以扩展其应用的范围是有必要的。

1 异常处理机制的分类和工作原理

从大的范围来讲，目前Windows平台下实现和使用的异常处理机制主要有3种：结构化异常处理(Structure Exception Handler, SEH)，向量化异常处理(Vectored Exception Handler, VEH)，C++异常处理(C++ Exception Handler, C++EH)。前2种是由Windows操作系统实现的异常处理机制，C++EH是由C++编译器实现的异常处理机制，但三者在其内部实现机理和调用底层函数方面是十分相似的，而且它们的底层实现机理是未公开的，需要通过逆向分析等手段来了解它们的内部机理。

1.1 结构化异常处理(SEH)

SEH是Windows平台下使用最早和最广泛的异常处理机制[4]。SEH最基本的特点是在堆栈中实现并且分为进程相关和线程相关2种类型[2]。下面分别对用户模式下的原始型SEH和封装型SEH进行讨论。

1.1.1 原始型SEH

SEH的进程相关类型是整个进程作用范围的异常处理函数，通过WIN32 API函数SetUnhandledExceptionFilter进行注册，而操作系统内部使用一个全局变量来记录这个顶层的处理函数，因此只能有一个全局性的异常处理函数。而线程相关类型的作用范围是本线程内，并且可注册多个，甚至可以嵌套注册。两者相比线程相关类型在实际应用中使用较为广泛，因此此处重点对此类型进行研究。

当线程初始化时，会自动向栈中安装一个异常处理结构，作为线程默认的异常处理。SEH最基本的数据结构是保存在堆栈中的称为EXCEPTION_REGISTRATION的结构体[2,5]，结构体包括2个元素：第1个元素是指向下一个EXCEPTION_REGISTRATION结构的指针(prev)，第2个元素是指向异常处理程序的指针(handler)。这样一来，基于堆栈的异常处理程序就相互连接成一个链表。异常处理结构在堆栈中的典型分布如图1所示。最顶端的异常处理结构通过线程控制块(TEB)0 Byte偏移处指针标识，即FS：[0]处地址。

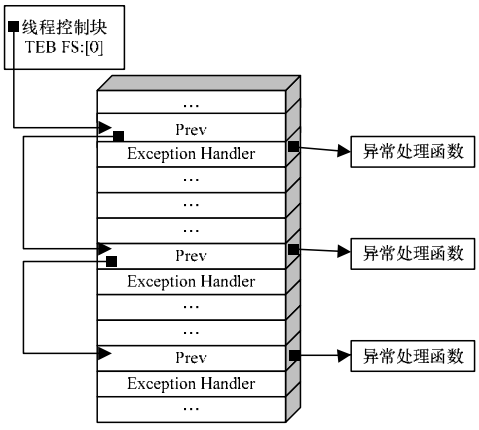

图1 异常处理结构在堆栈中的分布

用于进行实际异常处理的函数原型[5]可表示如下：

作者简介：张 明(1974－)，男，工程师、硕士研究生，主研方向：软件加密技术，软件反逆向技术；徐万里，教授、硕士

收稿日期：2009-04-20 **E-mail：**lovejazzandblues@hotmail.com

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

该函数的最重要的 2 个参数是指向_EXCEPTION_RECORD 结构的 ExceptionRecord 参数和指向_CONTEXT 结构的 ContextRecord 参数[2,4,5]，前者主要包括异常类别编码、异常发生地址等重要信息；后者主要包括异常发生时的通用寄存器、调试寄存器和指令寄存器的值等重要的线程执行环境。而用于注册异常处理函数的典型汇编代码可表示如下：

```
PUSH handler       ; handler 是新的异常处理函数地址
PUSH FS:[0]        ;指向原来的处理函数的地址压入栈内
MOV FS:[0],ESP     ;注册新的异常处理结构
```

当异常发生时，操作系统的异常分发函数在进行初始处理后，如果异常没有被处理就会开始在如图 1 所示的线程堆栈上遍历异常处理链，直到异常被处理，如果仍没有注册函数处理异常，则将异常交给缺省处理函数或直接结束产生异常的进程[2]。

1.1.2 封装型 SEH

通过使用_try{}/_except(){}/_finally{}等关键字，使开发人员更方便地在软件中使用 SEH 是封装型 SEH 的主要特点。该机制的异常处理数据结构定义如下[2,5]：

```
struct VC_EXCEPTION_REGISTRATION
{VC_EXCEPTION_REGISTRATION* prev;
                    FARPROC    handler;
    scopetable_entry* scopetable;    //指向 scopetable 数组指针
                    int    _index; //在 scopetable_entry 中索引
                    DWORD    _ebp; //当前 EBP 值
}
```

显而易见，结构体中后 3 个成员是新增的。而 scopetable_entry 的结构如下所示：

```
struct scopetable_entry
{DWORD     prev_entryindex; //前一 scopetable_entry 的索引
  FARPROC lpfnFilter;       //过滤函数地址
  FARPROC lpfnHandler;    //处理异常代码地址
}
```

封装型 SEH 的基本思想是为每个函数内的_try{}块建立一个 scopetable 表，每个_try{}块对应于 scopetable 中的一项，该项指向_try{}块对应的 scopetable_entry 结构，该结构含有与_except(){}/_finally{}对应的过滤函数和处理函数。若有_try{}块嵌套，则在 scopetable_entry 结构的 prev_entryindex 成员中指明，多层嵌套形成单向链表。而每个函数只注册一个 VC_EXCEPTION_REGISTRATION 结构，该结构中的 handler 成员是一个重要的运行时库函数_except_handler3[2,5]。该异常处理回调函数负责对结构中的成员进行设置，查找处理函数并根据处理结果决定是继续执行还是让系统继续遍历外层 SEH 链。

为了弄清看似复杂的封装型 SEH 原理，此处通过分析一个简单的使用封装型 SEH 的函数的反汇编实现，从而深入地了解封装型 SEH 的实现过程。该函数的 C 语言实现如下：

```
void A()
{ __try       // 0 号 try 块
        {__try // 1 号 try 块
            {*(PDWORD)0 = 0;
            }
            __except(EXCEPTION_CONTINUE_SEARCH)
            {printf("Exception Handler!");
            }              }
        __finally
        { puts("in finally");
        }   }
```

对应该函数的序言部分反汇编代码如下：

```
push      ebp
mov       ebp, esp
push      -1
push      offset _A_scopetable
push      offset _except_handler3
mov       eax, large fs:0
push      eax
mov       large fs:0, esp
…
```

显而易见，压入堆栈的结构与 VC_EXCEPTION_REGISTRATION 结构是一致的。查找 scopetable 的地址为 0x00422048，在调试器中查找该地址起始的内容如下：

```
FFFFFFFF     ;scopetable_entry0 的 prev_entryindex 值
00000000     ;lpfnFilter 地址值，为 0，对应_finally{}
004010EE     ;lpfnHandler 地址值
00000000     ;scopetable_entry1 的 prev_entryindex 值
004010C6     ;lpfnFilter 地址值
004010C9     ;lpfnHandler 地址值
…
```

第 1 组值对应 0 号 try 块，而该块对应_finally{}块，所以过滤函数地址为 0；第 2 组值对应 1 号 try 块，而该块对应_except(){}块，所以有过滤函数和处理函数的地址。进一步查看 0x004010EE 地址处的反汇编代码如下：

```
PUSH   OFFSET   ??_C@0L@PEFD@in?5finally?$AA@;   ”in finally”
CALL puts
ADD ESP,4
RETN
```

显然上述语句与_finally{}块中的 C 语言语句是对应的，而其他的地址经过查找也是分别对应的。

在进入 0 号 try 块时的反汇编语句如下：

```
MOV DWORD PTR SS:[EBP-4],0     ;对应第 1 个_try 语句
MOV DWORD PTR SS:[EBP-4],1     ;对应第 2 个_try 语句
…
```

而在退出_try 块时对应的反汇编语句如下：

```
…
MOV DWORD PTR SS:[EBP-4],0     ;退出第 2 个_try 块
MOV DWORD PTR SS:[EBP-4],-1    ;退出第 1 个_try 块
…
```

根据异常处理的堆栈结构可知，[EBP-4]处值就是 VC_EXCEPTION_REGISTRATION 结构中_index 的值。异常处理机制在进入和退出每个_try 块前设置相应的_index 值，这样就可正确处理封装型 SEH 内发生的各种异常。

以上通过实例进一步验证和明确了封装型 SHE 的内部机理，它只是扩展了原始型 SEH 的功能，简化了软件开发人员的工作。

1.2 向量化异常处理(VEH)

向量化异常处理(VEH)是在 Windows XP 以后版本中新增的一种异常处理机制。通过使用 Win32 API 函数 Add

VectoredExceptionHandler 就可以注册新的异常处理函数[2]，函数的参数就是指向 EXCEPTION_POINTERS 结构的指针。而 VEH 用到的数据结构可以构成一个双向链表。通过对用于注册 VEH 的函数进行反汇编分析研究可知，VEH 数据结构的相关信息存储在堆中。在用户模式下发生异常时，异常处理分发函数在内部会先调用遍历 VEH 记录链表的函数，如果没有找到可以处理异常的注册函数，再开始遍历 SEH 注册链表。

通过对 VEH 的原理和实现的研究，可以总结出 VEH 具有如下的特点：

(1)VEH 是进程相关的，同时可以注册多个，甚至嵌套注册，而且注册的位置可以指定。

(2)VEH 保存在堆中，而不像 SEH 保存在堆栈中。

(3)VEH 只能用在用户模式程序中，而 SEH 还可以用在内核模式中。

(4)VEH 处理先于 SEH 处理执行。

正因为 VEH 具有以上特点，所以在实际应用中它的使用更加灵活，甚至可以利用它来实现一个微型调试器。

1.3 C++异常处理(C++EH)

与 SHE, VEH 相比，C++异常处理的内部实现更加复杂，因为它牵扯到类、对象等相关处理，但 C++EH 也是建立在基本异常处理机制基础之上的。另外，编译器提供了类似封装型 SEH 的关键字 try{}/catch(){}/throw 帮助开发人员方便地处理 C++程序中出现的各种异常。C++EH 的独有特点有：

首先，C++EH 扩展了 EXCEPTION_REGISTRATION 结构，新添加的成员主要作用是当异常发生时用于确定对应的 try{}块，其功能类似于封装型 SEH 结构中的_index 成员。

当编译器对带有 C++EH 的函数进行编译时，要添加 2 个重要的数据结构：一个是异常处理函数_CxxThrowException()；另一个是 funcinfo 结构，里面包含对应 catch{}块的地址、catch{}块所关心的异常类型等重要的信息。_CxxThrowException()是 C++EH 的统一的异常处理函数，相当于封装型 SEH 中的_except_handler3()。_CxxThrowException()函数内部还是调用了 Win32API 函数 RaiseException()来产生异常，并且异常码是固定的 0xE06D7363。

当 C++异常发生时，发生异常函数的 funcinfo 结构传给_CxxThrowException()，开始寻找是否有对该异常感兴趣的 catch{}块，如果有就调用 catch{}块处理异常；如果没有则沿异常处理链表继续寻找，直到异常被处理或最后抛出错误对话框为止。C++EH 除了增加了 funcinfo 结构外，还增加了 tryblock, catchblock 等数据结构，主要的目的就是应对复杂的 try{}/catch(){}/throw 结构，在异常发生时能正确找到对应的异常处理块，在发生堆栈展开时能够调用对象的析构函数，释放资源等。

以上分析了在用户模式下常见的 3 种异常处理类型，这 3 种类型适用于不同的场合，具有各自的优缺点。但有一点需要注意，即异常处理在帮助处理各种异常和错误的同时也是有代价的，它增加了软件在空间和时间上的开销，增加了程序运行在内核模式和用户模式之间的切换次数，因此，应该避免滥用异常处理。

2 异常处理机制的应用

通过第 1 节的分析可知，异常处理机制的实现是非常复杂的，但同时也提供了很多重要信息，因此，异常处理机制除了用于软件中处理各种可能发生的异常和错误外，还在软件安全领域应用广泛。这里从正反两方面来研究它的应用，即软件的反调试技术和缓冲区溢出利用技术。

2.1 软件反调试应用

软件安全现在已经逐渐被人们重视起来，防止核心软件、商业软件被破解、被逆向分析是软件安全中一项重要内容。而破解和逆向分析的一种重要手段就是调试软件，分析软件重要部分的功能、算法实现等。

现在很多软件在代码中嵌入了反调试代码，防止被恶意调试或增加逆向分析的难度。而反调试的重要手段之一就是利用异常处理机制[2]。

下面给出的一段代码就是利用 SEH 来检测是否正在被调试，如果满足条件，则再次引发一个故意制造的异常，在异常处理函数中清除可能存在的硬件断点并更改指令指针的值，最终导致程序可能崩溃。由于异常处理涉及很多底层操作，因此代码的很多部分以汇编语言形式给出。其基本思想是先读取状态寄存器(EFLAGS)的值并设置其陷阱标志位(TF)为 1。如果程序正常运行会抛出一个单步执行异常；而如果程序在被调试，由于调试器接管了异常，因此异常不会抛出，进而可以判断出程序正在被跟踪调试并采取一定的反应措施。具体实现(代码的主要部分)如下：

主函数部分：

```
…
BOOL bIFDebugged=FALSE;
 __try
 {
      __asm
       {
            pushfd                          //读状态寄存器的值
            or dword ptr [esp],0x100     //设置陷阱标志位
            popfd                  //将新值装入状态寄存器
            nop
       }
 }
 __except(EXCEPTION_EXECUTE_HANDLER)
 {
 bIFDebugged=TRUE; //该语句如果被执行，证明被调试
 }
if(FALSE== bIFDebugged)
{
      __asm
     {
            push OFFSET my_handler
            push FS:[0]
            mov FS:[0],esp           //登记异常处理结构
            xor edx,edx
            mov eax,10
            xor ecx,ecx
            idiv ecx
            mov eax,[esp]
            mov FS:[0],eax
            add esp,8
     }
…
```

异常处理函数部分：

```
…
   if(ExceptionRecord->ExceptionCode==
        STATUS_INTEGER_DIVIDE_BY_ZERO)
```

```
3       {
4               ContextRecord->Dr0=0;
5               ContextRecord->Dr1=0;
6               ContextRecord->Dr2=0;
7               ContextRecord->Dr3=0;
8               ContextRecord->Dr6=0xFFFF0FF0;
9               ContextRecord->Dr7=0x2400;
10              ContextRecord->Eip+=20;
11              return ExceptionContinueExecution;
12      }
…
```

在以上代码段中主函数的第 6 行~第 8 行用于设置状态寄存器的陷阱标志位，第 12 行的异常处理标志表明如果异常被抛出就执行__except{}块中的语句。而如果程序在被调试则不会抛出这个单步执行异常，因此 bIFDebugged 的值仍是 FALSE，证明在被调试。发现被调试时，通过第 20 行~第 22 行注册一个异常处理结构，第 23 行~第 26 行故意制造一个除零错误进而执行异常处理函数，在异常处理函数中第 4 行~第 9 行的作用主要就是清除可能存在的硬件断点。因为目前 Intel 系列的 CPU 共有 8 个调试寄存器，其中的 DR0~DR3 用于存储硬件断点值，而 DR6, DR7 是调试控制寄存器，通过对 4 个调试寄存器清零，可以达到清除硬件断点、反调试的目的。异常处理函数的第 10 行随意修改了 EIP 的值，导致异常处理返回后程序会崩溃，即发现被调试后作出反应。

2.2 堆栈溢出利用技术

堆栈溢出技术的主要思想就是向程序的缓冲区中输入精心设计的指令编码并覆盖原来堆栈上的数据和指令，再通过一定的手段来执行这段输入指令(称作 shellcode)从而达到预先的目的。而通过 2.1 节的研究知道线程级的异常处理链是保存在程序运行的堆栈上，因此可以把它作为溢出攻击的一种方便手段[3]。

利用 SEH 执行 shellcode 的基本原理就是，首先分析破解程序，找到其中的堆栈溢出漏洞(例如字符串拷贝函数 strcpy()的使用)及存在的 SEH 处理链，设计一段 shellcode 执行指令保存在字符串变量中并确定该变量的地址。该段指令能将原来保存在堆栈中的异常处理函数地址覆盖并替换为 shellcode 的地址，然后故意制造异常，由于异常处理函数已经被替换，因此系统会执行新的异常处理函数，也就是那段 shellcode 代码，从而达到堆栈溢出的目的。

下面的一段代码演示了这一过程的具体实现：

```
1   #include <windows.h>
2   #include "stdio.h"
3   char shellcode[]="\x90\x90\x90\x90...\x98\xFE\x12\x00";
4   DWORD myhandler(void)
5   {
6       printf("press Enter to kill process!\n");
7       getchar();
8       ExitProcess(1);
9       return 1;
10  }
11  void function(char *input)
12  {
13      char buf[200];
14      int divide;
15      divide=0;
16      __try
17      {
18          strcpy(buf,input);
19          divide=10/divide;
20      }
21      __except(myhandler()){}
22  }
23  void main()
24  {
25      function(shellcode);
26  }
```

从上面的代码中可以看出，第 18 行语句 strcpy(buf,input)是一个安全漏洞，通过对程序进行分析可以找到离栈顶最近的异常处理函数地址保存在堆栈地址 0x0012ff68 处，buf 变量的地址在 0x0012fe98。因此设计 shellcode 字符串长度要能覆盖到异常处理函数地址处，并将地址替换为 0x0012fe98。注意此处 shellcode 的中间部分为了节省篇幅被省去，在字符串的最后就是新地址的值。

因此，当程序运行完第 18 行语句后，已经完成替换，第 19 行语句会抛出一个除零异常，而根据异常处理设计要执行异常处理代码，此时会执行 shellcode 中的语句，达到了堆栈溢出目的。

为了防止异常处理机制被恶意利用，一种改进的方法是使用 SAFESEH(Secure Exception Handling)技术。其基本思想是将模块中合法的异常处理函数登记在一个专用的称作 SAFESEH 的表中。当有异常发生时，异常分发函数会根据异常处理函数的地址到它所对应的模块中查询这个函数是否在表中，如果在就执行，否则不执行。可见其核心思想是一种验证机制。而且 SAFESEH 表存储在只读内存区域，由操作系统加载器在加载模块时写入。因此，一般情况下很难对其进行非法修改。

3 结束语

Windows 平台下异常处理机制是 Windows 操作系统的重要组成部分，有着广泛的应用领域。本文深入挖掘分析了 3 种异常处理机制的内部机理，总结各自的特点和使用条件，进而研究它们在软件反调试技术和软件漏洞利用技术中的应用。下一步还应在利用异常处理机制辅助进行软件调试和防止利用异常处理机制其他弱点进行攻击等方向继续深入研究。

参考文献

[1] Russinovich M E, Solomon D A. 深入解析 Windows 操作系统[M]. 4 版. 潘爱民, 译. 北京: 电子工业出版社, 2007.

[2] 看雪学院. 软件加密技术内幕[M]. 北京: 电子工业出版社, 2004.

[3] Anley C, Heasman J. The Shellcoder's Handbook——Discovering and Exploiting Security Holes[M]. 2nd ed. Indiana, USA: Wiley Publishing Inc., 2007.

[4] 齐 雷, 谢余强, 程东年, 等. Win32 SEH 异常处理机制分析[J]. 信息工程大学学报, 2004, 5(6): 49-52.

[5] Pietrek M. A Crash Course on the Depths of Win32 Structured Exception Handling[EB/OL]. (1997-01-02). http://www.microsoft.com/msj/0197/exception/exception.aspx.

编辑 任吉慧