

2021/01/22_32位汇编_第10课_程序中线程函数自己处理异常、_TEB、_PEB

笔记本: 32位汇编

创建时间: 2021/1/22 星期五 9:52

作者: ileemi

- [程序中线程函数自己处理异常](#)
- [线程函数处理完异常后程序再次出现异常](#)
- [异常展开](#)
- [结构化异常和筛选器异常](#)
- [_TEB](#)
- [_NT_TIB](#)
- [_PEB](#)
 - [BeingDebugged](#)
 - [IsDebuggerPresent](#)
 - [LastErrorValue](#)
 - [Ldr](#)
 - [遍历模块](#)

程序中线程函数自己处理异常

注册SEH时可以多push一个参数作为异常回调函数的第二个参数（回调函数的第二个参数保存当时注册的SEH结构体的首地址），线程函数自己处理异常时方便修改EIP的值，继续执行程序后面的代码。

注意：修改寄存器EIP的值后，需要对堆栈进行平衡（没有平衡堆栈前，栈顶保存的是出错函数的EBP值，可以导致程序不能正常的平衡堆栈以及恢复寄存器环境），防止程序出现递归。

做法：线程函数注册SEH之前将寄存器 EBP 保存到堆栈中，当线程函数出现异常时，在其对应的异常回调函数中恢复对应线程函数的 EBP 即可（平衡堆栈）。

地址	HEX 数据	反汇编	注释	寄存器 (FPU)
004013F0	55	PUSH EBP		EAX 00000001
004013F1	B8EC	MOV EBP,ESP		ECX 7716387A ntdll.7716387A
004013F3	55	PUSH EBP		EDX 0050017A
004013F4	68 1114A000	PUSH SEH.00401411		EBX 7EFD0000
004013F9	68 F412A000	PUSH SEH.004012FA		ESP 0018FF58
004013FE	64:AT 00000000	MOV EAX,DWORD PTR FS:[0]		EIP 00401424 SEH.00401424
00401400	50	PUSH EAX		ESI 00000000
00401405	64:8925 0000	MOV DWORD PTR FS:[0],ESP		EDI 00000000
0040140C	E8 82FEFFFF	CALL SEH.004012C3		EIP 00401424 SEH.00401424
00401411	68 00	PUSH 0	title = NB_OK\NB_APPLMODAL	
00401413	68 5320A000	PUSH SEH.00402053	text = "Hello World!\n"	
00401418	68 4420A000	PUSH SEH.00402044	hOwner = NULL	
0040141D	6A 00	PUSH 0	MessageBox	
00401422	EA 10000000	JMP <JMP.User32.MessageBoxBox>		
0040142A	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	SEH.00401411	
0040142E	04:85 00000000	MOV DWORD PTR FS:[0],EAX		
0040142D	C9	LEAVE		
0040142E	6A 00	PUSH 0	ExitCode = 0	
00401430	E8 19000000	JMP <JMP.Kernel32.ExitProcess>	ExitProcess	
00401435	C3	RET		
00401436	FF25 2C20A000	JMP DWORD PTR DS:[<user32.vsprintf>]	user32.vsprintf	
0040143C	FF25 2820A000	JMP DWORD PTR DS:[<user32.MessageBoxBox>]	user32.MessageBoxBox	
00401442	FF25 0820A000	JMP DWORD PTR DS:[<kernel32.CloseHandle>]	kernel32.CloseHandle	
00401448	FF25 0C20A000	JMP DWORD PTR DS:[<kernel32.CreateFileA>]	kernel32.CreateFileA	
0040144E	FF25 1020A000	JMP DWORD PTR DS:[<kernel32.ExitProcess>]	kernel32.ExitProcess	
00401454	FF25 0020A000	JMP DWORD PTR DS:[<kernel32.GetLastError>]	kernel32.GetLastError	
00401459	FF25 1420A000	JMP DWORD PTR DS:[<kernel32.GetLocalTime>]	kernel32.GetLocalTime	
堆栈 SS:[0018FF58]=00401411 (SEH.00401411)				
EAX=00000001				
地址	HEX 数据	ASCII		
00401400	ES 67 01 00 00 00 17 00 14 00 00 26 00 CE 02	77.16.38.7A		
00401410	00 00 00 00 00 00 00 00 00 00 00 00 00 00		
00401420	00 00 00 00 00 00 00 00 00 00 00 00 00 00		
00401430	00 00 00 00 00 00 00 00 00 00 00 00 00 00		

0018FF58	00401400	返回到 SEH 记录的指令
0018FF60	0018FF70	返回到 SEH 记录的指令
0018FF6A	004012DE	返回到 SEH.004012DE 来自 SEH
0018FF68	0018FF78	返回到 SEH 记录的指令
0018FF6C	00401400	返回到 SEH 记录的指令

线程函数处理完异常后程序再次出现异常

线程函数自己处理完异常后，返回到继续执行的代码位置，执行后面的代码时又出现异常在高级语言中会将这样的异常交操作系统去处理。在32位汇编中，程序会进入死循环，会之一询问哪个线程函数处理异常。出现这种情况原因是：保存异常回调函数的链表头项没有更新，导致程序进入递归。

异常回调函数处理后，需要将保存异常回调函数链表的头项删除（谁处理异常，就将当前注册SEH异常的FS结构体首地址（段寄存器FS地址行保存的就是注册的SEH异常的地址）赋值给链表头项，删除链表前，还需要将资源进行释放（遍历链表）。

程序中线程函数自己处理异常后，需要将EIP修改到继续执行的代码位置上，不修改EIP的值，程序会递归（重复执行异常回调函数）。

```
RadASM - SEH - [C:\Users\lileai\Desktop\SEH\SEH.asm]
文件(F)  编辑(E)  视图(V)  格式(O)  工程(P)  构造(M)  工具(T)  窗口(W)  选项(O)  宏(A)  帮助(H)  收藏夹
Y: SEH.asm

240 ; 删除链表，但是在这里面写，操作系统API会修改FS:[0]处的地址(原因就是API中也注册的SEH)
241 ;mov eax, ContextRecord
242 ;mov fs:[0], eax
243
244 ; 将 EIP 修改为 MAIN 处理的代码位置
245 mov ebx, ContextRecord
246 assume ebx ptr CONTEXT
247 mov eax, EstablisherFrame
248 mov eax, [eax+8] ;offset MAIN_LABEL1
249 mov [ebx].regEip, eax
250
251 ; 返回时，释放堆栈空间，防止后面的代码使用堆栈数据出错（不能正常的平衡堆栈和恢复环境）
252 mov eax, EstablisherFrame
253 mov eax, [eax+12] ; 原线程函数
254 mov [ebx].regEbp, eax
255
256 ; 保存日志到文件
257 invoke GETSYTIME
258 invoke OUTFILE, addr @szBuff
259
260 ; MAIN处理
261 ;mov eax, ExceptionContinueSearch
262 mov eax, ExceptionContinueExecution
263 ret
264 MAIN_except_handler endp
```

```
RadASM - SEH - [C:\Users\lileai\Desktop\SEH\SEH.asm]
文件(F)  编辑(E)  视图(V)  格式(O)  工程(P)  构造(M)  工具(T)  窗口(W)  选项(O)  宏(A)  帮助(H)  收藏夹
Y: SEH.asm

207 ; MAIN异常回调函数
208 MAIN_except_handler proc uses ebx esi ExceptionRecord ptr EXCEPTION_RECORD, EstablisherFrame PVOID, ContextRecord ptr CONTEXT, DispatcherContext PCONTEXT
209 LOCAL @szBuff[256]:byte
210 a
211 mov ebx, ExceptionRecord
212 assume ebx ptr EXCEPTION_RECORD
213 mov ecx, ContextRecord
214 assume ecx ptr CONTEXT
215
216 ; 异常展开，操作系统将异常回调函数的链表所有项删除
217 .if [ebx].ExceptionCode == 0C0000027h
218 ; 完成释放资源，保存文件等操作
219 invoke wsprintf, addr @szBuff, offset MY_MAIN_FMT, [ebx].ExceptionCode, [ebx].ExceptionAddress
220 invoke MessageBox, NULL, addr @szBuff, offset MY_TITLE, MB_OK
221 ret
222 .endif
223
224 ; 如果eip保存的地址超过MAIN_LABEL1，无法处理
225 mov eax, EstablisherFrame
226 mov esi, [eax+8]
227
228 .if [ebx].ExceptionAddress >= esi
229 mov eax, ExceptionContinueSearch ; 不再处理，交给操作系统处理
230 ret
231 .endif
```

异常展开

进行资源清理，更新链表头（保存异常回调函数的链表）。

局部变量不能在注册SEH异常之前抬栈（破坏栈结构），应该等SEH异常注册后抬栈。代码示例：

```

MAIN:
    assume fs:nothing
    push ebp
    mov ebp, esp

    ; 注册SEH异常
    push ebp
    push MAIN_LABEL ; offset
    push offset MAIN_except_handler ; handle
    mov eax, fs:[0]
    push eax ; prev
    mov fs:[0], esp

    sub esp, 50h ; 为局部变量抬栈（申请空间）
    invoke MY_FUN1

; MAIN 处理的代码位置
MAIN_LABEL:
    ; 异常展开, 调用异常函数, 通知清理
    lea edi, [ebp-10h] ; 获取保存异常回调函数链表的首地址
    mov esi, fs:[0] ; 获取链表头Fun2

    ; 遍历保存异常回调函数的链表, 依次进行展开操作
    .while esi != edi
        push 0
        push 0
        push 0
        ; 获取局部变量地址 EXCEPTION_RECORD (16 + 50)
        lea eax, [ebp-66]
        assume eax:ptr EXCEPTION_RECORD
        mov [eax].ExceptionCode, 0c0000027h ; 设置异常原因
        push eax
        mov eax, dword ptr [esi+4]
        ; 调用异常处理回调函数内部的展开代码, 通知其进行展开, 释放资源
        call eax

        ; 更新链表头
        mov esi, dword ptr [esi]
        mov fs:[0], esi
    .endw

    invoke MessageBox, NULL, offset MY_MSG, offset MY_TITLE, MB_OK

    ; 触发异常
    mov eax, 0

```

```
mov dword ptr[ecx], 1

; 卸载SEH异常
mov ecx, dword ptr [ebp - 8]
mov fs:[0], ecx

leave
invoke ExitProcess, 0
ret
end MAIN
```

结构化异常和筛选器异常

结构化异常和筛选器异常在一个线程函数中都进行注册时，结构化异常的优先级要比筛选器异常高。

_TEB

保存线程相关信息，通过 Windbg 输入命令："dt _TEB"

_NT_TIB

保存异常回调函数的链表信息

_PEB

进程环境块，保存进程相关的信息。

BeingDebugged

调试进程标志，操作系统通过这个标志判断程序是否被调试。

IsDebuggerPresent

操作系统提供的API，用于检测进程是否被调试，当该API返回值为真，说明程序被调试，程序被调试可以退出进程或者故意让程序崩溃（防止通过API下断点），防止自己的程序被调试。IsDebuggerPresent 在 "kernel32.dll" 模块中。

代码示例：

```
if (IsDebuggerPresent()) {  
    char* p = NULL;  
    *p = 1;  
}
```

win10 下通过 "ollydbg" 打开一个程序，在 "kernel32.dll" 模块中 "IsDebuggerPresent" 对应的反汇编如下图所示：

76B2924F	CC	INT3
76B29250	64:A1 30000000	MOV EAX,DWORD PTR FS:[30]
76B29256	0FB640 02	MOVZX EAX,BYTE PTR DS:[EAX+2]
76B2925A	C3	RETN
76B2925B	CC	INT3

ntdll32!_TEB

+0x000	NtTib	: _NT_TIB
+0x01c	EnvironmentPointer	: Ptr32 Void
+0x020	ClientId	: _CLIENT_ID
+0x028	ActiveRpcHandle	: Ptr32 Void
+0x02c	ThreadLocalStoragePointer	: Ptr32 Void
+0x030	ProcessEnvironmentBlock	: Ptr32 _PEB
+0x034	LastErrorValue	: UInt4B
+0x038	CountOfOwnedCriticalSections	: UInt4B
+0x03c	CsrClientThread	: Ptr32 Void
+0x040	Win32ThreadInfo	: Ptr32 Void
+0x044	User32Reserved	: [26] UInt4B
+0x0ac	UserReserved	: [5] UInt4B
+0x0c0	Wow32Reserved	: Ptr32 Void

0:000> dt _PEB

ntdll32!_PEB

+0x000	InheritedAddressSpace	: UChar
+0x001	ReadImageFileExecOptions	: UChar
+0x002	BeingDebugged	: UChar
+0x003	BitField	: UChar
+0x003	ImageUsesLargePages	: Pos 0, 1 Bit
+0x003	IsProtectedProcess	: Pos 1, 1 Bit
+0x003	IsLegacyProcess	: Pos 2, 1 Bit
+0x003	IsImageDynamicallyRelocated	: Pos 3, 1 Bit
+0x003	SkipPatchingUser32Forwarders	: Pos 4, 1 Bit
+0x003	SpareBits	: Pos 5, 3 Bits
+0x004	Mutant	: Ptr32 Void

- mov eax, dword ptr fs:[30] -- 首地址加上偏移30的位置也就是访问 _PEB 成员并将结果赋值给寄存器 eax。
- movzx eax, byte ptr ds:[eax+2] -- eax 偏移加2访问的就是 "BeingDebugged" 成员，根据 "BeingDebugged" 的值作为返回值进行返回，操作系统根据返回值进行判断程序是否被调试（API的实现原理就是如此）。

了解原理后，就可以对不可调试的程序进行反反调试：

- 通过寄存器FS获取 _TEB的首地址（通过 mov eax, dword ptr fs:[18]也可以获取）
- 首地址+偏移30的地址并访问地址上的值（地址）

"ollydbg" 调试器不能调试筛选器异常程序，原因就是操作系统派发筛选器异常时发现调试器存在就不进行派发，可通过上面的步骤修改 _PEB，而通过 x64dbg 进行筛选器异常程序调试时，x64dbg会自动修改 "BeingDebugged" 的值。

LastErrorValue

在Windows平台下，使用Windows提供的API，返回值为空时，可以通过 "GetLastError" 获取错误码，而 "GetLastError" 内部访问的就是 _TEB 的成员 "LastErrorValue"。

在 "ollydbg" 以及 "x64dbg" 调试器中寄存器窗口的 "LastErr" 内部也是访问的 "LastErrorValue"，如下图所示：

EIP 001B1023 Test.<模块入口点>

C 0 ES 002B 32位 0(FFFFFFFF)

P 1 CS 0023 32位 0(FFFFFFFF)

A 0 SS 002B 32位 0(FFFFFFFF)

Z 1 DS 002B 32位 0(FFFFFFFF)

S 0 FS 0053 32位 389000(FFF)

T 0 GS 002B 32位 0(FFFFFFFF)

D 0

O 0 LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

Ldr

PEB.Ldr 成员是指向 _PEB_LDR_DATA 结构体的指针。在 Windbg 中输入命令 "dt _PEB_LDR_DATA" 可查看该结构体的成员。

代码47-5 _PEB_LDR_DATA结构体

```
+000 Length : Uint4B
+004 Initialized : UChar
+008 SsHandle : Ptr32 Void
+00c InLoadOrderModuleList : _LIST_ENTRY
+014 InMemoryOrderModuleList : _LIST_ENTRY
+01c InInitializationOrderModuleList : _LIST_ENTRY
+024 EntryInProgress : Ptr32 Void
+028 ShutdownInProgress : UChar
+02c ShutdownThreadId : Ptr32 Void
```

_PEB_LDR_DATA 结构体中有三个双向链表成员，通过 "dt _LIST_ENTRY" 命令查看 _LIST_ENTRY 结构体中的成员，结构体定义如下：

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

使用进程快照遍历进程中加载的所有模块，其内部就是访问的这三个链表，具体如下：

- InLoadOrderModulelist：按加载顺序排列
- InMemoryOrderModuleList：按内存顺序排列
- InInitializationOrderModuleList：按初始化顺序排序

知道上面的三个链表，遍历模块列表就不再需要调用API了。还可以用于 "dll注入"，将注入的dll进行隐藏。注入的dll会保存到上面的双向链表中，注入成功后，防止别人

遍历模块列表发现，可以将注入的dll的前驱和后继进行修改，防止遍历到，这种做法就叫做：隐藏模块（断链）

_LDR_DATA_TABLE_ENTRY 结构体定义如下：

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

遍历模块

使用汇编遍历模块，并隐藏主模块，代码示例如下：

```
.CODE
START:
    assume fs:nothing

    mov eax, fs:[30h]    ;_PEB
    mov eax, [eax+0ch]   ;_PEB_LDR_DATA
    mov edi, [eax+0ch]   ;pev

;断链 -- 将主模块 (edi) 从链表中断掉 (不是删除)
;mov esi, dword ptr [edi] -- 获取链表头的下一项  prev.next = head.next  next.prev = prev
;-----
; NULL data2 (edi) -- 主模块
; |
; prev - data1 - next
; |       |
; prev - data2 - next
; |       |
; prev - data3 - next
; |
; NULL
;-----

mov esi, edi            ; esi -- data2, edi -- data2
mov edi, dword ptr [edi] ; edi -- data3
mov eax, dword ptr [esi + 4] ; eax -- data1
mov dword ptr [edi + 4], eax ; data3 的前驱改为data1

mov edi, esi            ; esi -- data2, edi -- data2
mov edi, dword ptr [edi + 4] ; edi -- data1
mov eax, dword ptr [esi] ; eax -- data3
mov dword ptr [edi], eax ; data1 的后继 改为 data3
```

.386

.model flat, stdcall

option casemap:none

```

include windows.inc
include kernel32.inc
include user32.inc
includelib kernel32.lib
includelib user32.lib

.CODE
START:
    assume fs:nothing

    mov  eax,  fs:[30h]      ;_PEB
    mov  eax,  [eax+0ch]    ;_PEB_LDR_DATA
    mov  edi,  [eax+0ch]    ;pev

    ; 断链 -- 将主模块 (edi) 从链表中断掉 (不是删除)
    ; mov esi, dword ptr [edi] -- 获取链表头的下一项
    ; prev.next = head.next    next.prev = prev

    mov  esi,  edi ; esi -- data2, edi -- data2
    mov  edi,  dword ptr [edi] ; edi -- data3
    mov  eax,  dword ptr [esi + 4] ; eax -- data1
    mov  dword ptr [edi + 4], eax ; data3 的前驱改为data1

    mov  edi,  esi ; esi -- data2, edi -- data2
    mov  edi,  dword ptr [edi + 4] ; edi -- data1
    mov  eax,  dword ptr [esi] ; eax -- data3
    mov  dword ptr [edi], eax ; data1 的后继 改为 data3

    ; 循环遍历模块
    mov  eax,  fs:[30h] ; 获取_PEB首地址
    mov  eax,  [eax+0ch] ; 获取_PEB_LDR_DATA结构体得的首地址
    mov  edi,  [eax+0ch]
    ; 获取链表的前驱pev项, 当做 _LDR_DATA_TABLE_ENTRY 结构体进行解析
    mov  esi,  edi
LOOP1:
    mov  ebx,  [esi+28h] ; FullDllName
    invoke MessageBoxW, NULL, ebx, NULL, MB_OK
    mov  esi,  dword ptr[esi] ; 获取下一个
    cmp  esi,  edi
    jnz  LOOP1

    invoke ExitProcess, 0
    ret
end START

```


