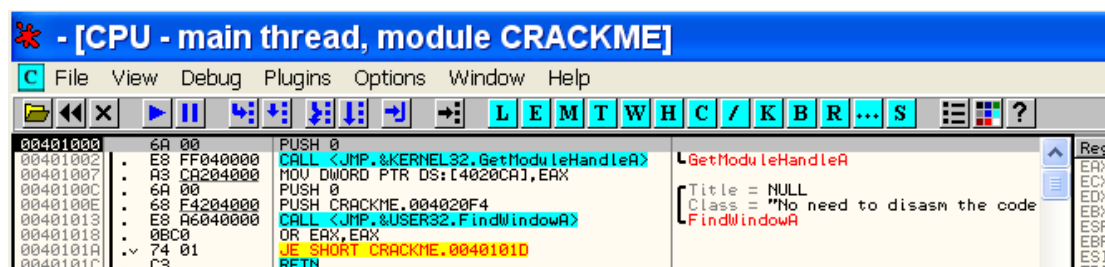


第 25 章-异常处理

本章,我们将介绍异常处理,这一块通常是初学者的绊脚石,但是如果研究深入一点的话,你会发现它并不难。

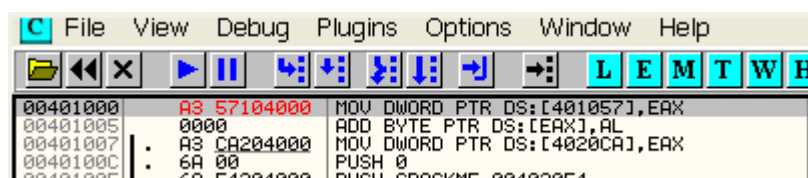
那么异常是如何产生的呢?当处理器执行了一个错误的操作的时候,程序中就会产生异常。好了,我们来看几个异常例子吧,我们用 OD 打开 cruehead'a 的 CrackMe。



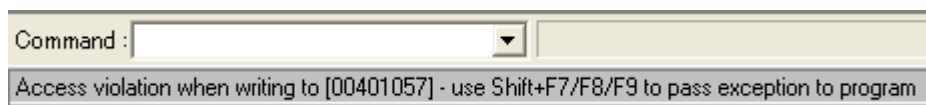
可以看到断在了入口点处,我们在第一行输入会引发异常的指令,这里我们采用 Mr Silver 写的异常例子中指令。

内存访问异常:当线程中尝试访问没有访问权限的内存的时候会发生该类异常。例如:一个线程尝试向只具有读权限的内存写入数据的时候就会产生内存访问异常。

我们在 OD 中输入如下指令:



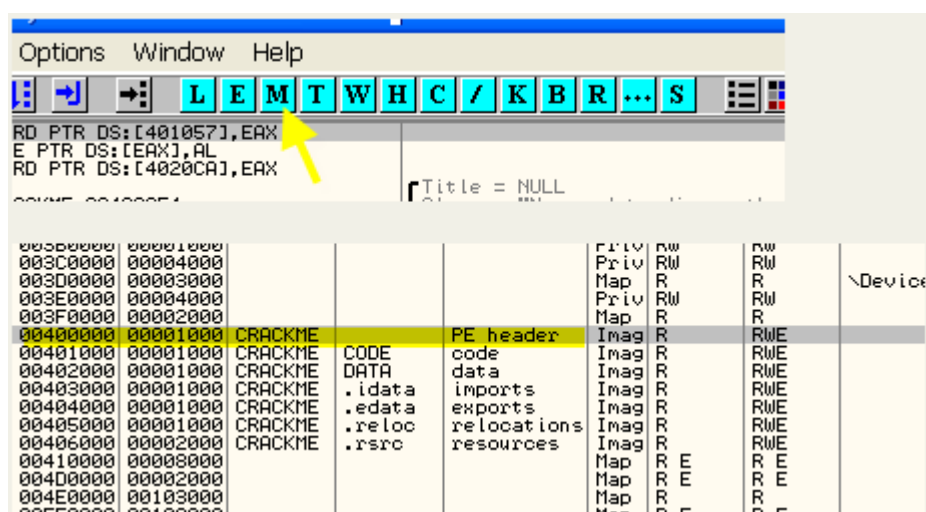
这里我们可以看到 401057 开始的内存单元只具有读取和执行权限,并不具有写入权限。因此如果尝试向该内存单元写入数据的话就会产生异常,我们按 F8 键。



程序会根据 PE 头中的相关信息设置区段的初始权限,当然也可以使用诸如 VirtualProtect 这类 API 函数在运行时修改权限。

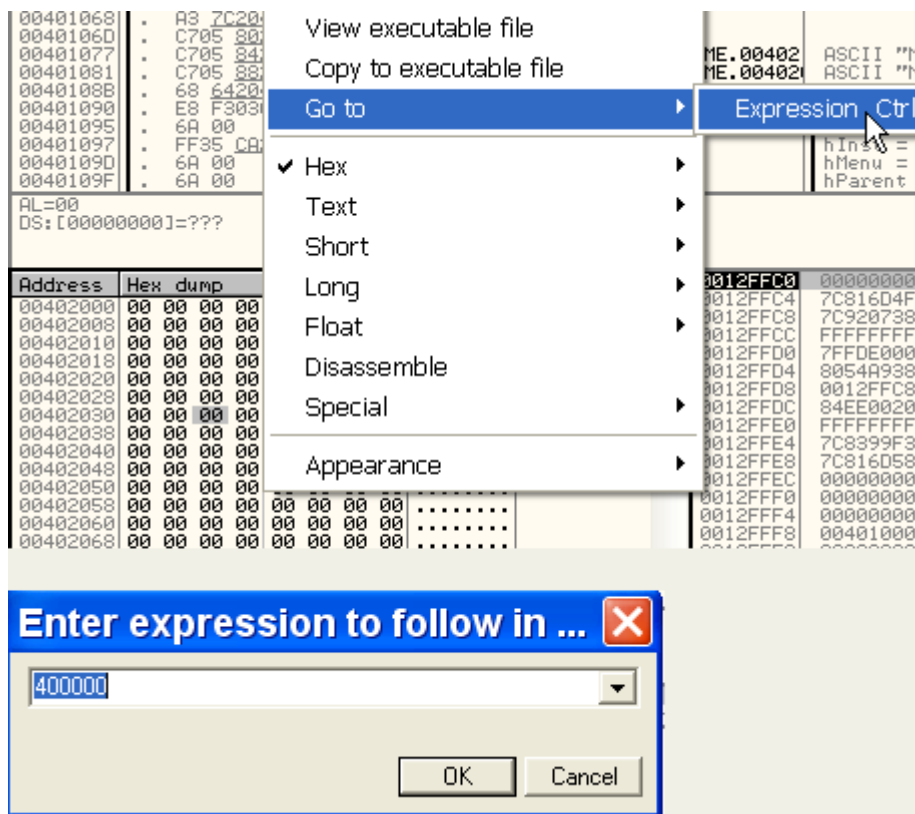
那么在 OllyDbg 中我们如何查看各个区段的初始权限呢,还有就是如何修改这些权限呢?

我们可以通过单击工具栏中的 M 按钮来查看各个区段的情况。



我们可以看到主模块的所在的区段开始于 400000,首先是 PE 头,占 1000 个字节,PE 头中保存了各个区段的名称,长度以及程序运行所必须的一些信息。

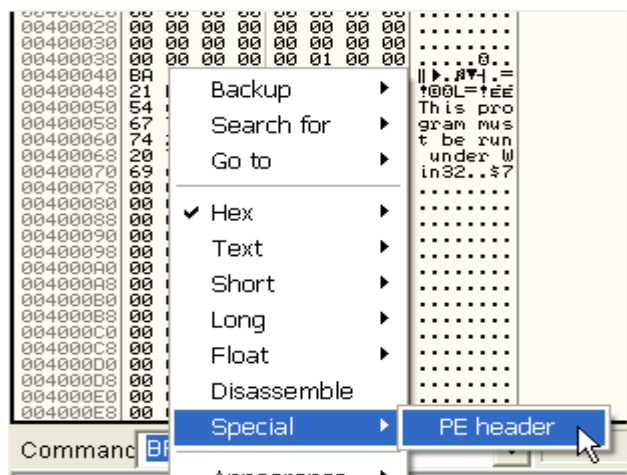
我们在数据窗口中定位到 PE 头。



由于 PE 头开始于 400000,所以我们输入 400000。

Address	Hex dump	ASCII
00400000	4D 5A 50 00 02 00 00 00	MZP.0...
00400008	04 00 0F 00 FF FF 00 00	♦.*. ...
00400010	B8 00 00 00 00 00 00 00	@.....
00400018	40 00 1A 00 00 00 00 00	@.+.....
00400020	00 00 00 00 00 00 00 00
00400028	00 00 00 00 00 00 00 00
00400030	00 00 00 00 00 00 00 00
00400038	00 00 00 00 00 01 00 000..
00400040	BA 10 00 0E 1F B4 09 CD	. .&v . =
00400048	21 B8 01 4C CD 21 90 90	!00L=teÉ
00400050	54 68 69 73 20 70 72 6F	This pro
00400058	67 72 61 6D 20 6D 75 73	gram mus
00400060	74 20 62 65 20 72 75 6E	t be run
00400068	20 75 6E 64 65 72 20 57	under W
00400070	69 6E 33 32 0D 0A 24 37	in32..\$7
00400078	00 00 00 00 00 00 00 00
00400080	00 00 00 00 00 00 00 00
00400088	00 00 00 00 00 00 00 00
00400090	00 00 00 00 00 00 00 00
00400098	00 00 00 00 00 00 00 00
004000A0	00 00 00 00 00 00 00 00

其实 OllyDbg 中有一个可以解析 PE 头的各个字段的选项,我们可以在数据窗口中单击鼠标右键。



DS: [00000000]=???

Address	Hex dump	Data	Comment
00400000	4D 5A	ASCII "MZ"	DOS EXE Signature
00400002	5000	DW 0050	DOS_PartPag = 50 (80.)
00400004	0200	DW 0002	DOS_PageCnt = 2
00400006	0000	DW 0000	DOS_ReloCnt = 0
00400008	0400	DW 0004	DOS_HdrSize = 4
0040000A	0F00	DW 000F	DOS_MinMem = F (15.)
0040000C	FFFF	DW FFFF	DOS_MaxMem = FFFF (65535.)
0040000E	0000	DW 0000	DOS_ReloSS = 0
00400010	0800	DW 0008	DOS_ExeSP = 08
00400012	0000	DW 0000	DOS_ChkSum = 0
00400014	0000	DW 0000	DOS_ExeIP = 0
00400016	0000	DW 0000	DOS_ReloCS = 0
00400018	4000	DW 0040	DOS_Tabloff = 40
0040001A	1A00	DW 001A	DOS_Overlay = 1A
0040001C	00	DB 00	
0040001D	00	DB 00	
0040001E	00	DB 00	
0040001F	00	DB 00	
00400020	00	DB 00	

我们可以看到显示出了 DOS 头的各个字段信息。

00400039	00	DB 00	
0040003A	00	DB 00	
0040003B	00	DB 00	
0040003C	00010000	DD 00000100	Offset to PE signature
00400040	BA	DB BA	
00400041	10	DB 10	
00400042	00	DB 00	
00400043	0E	DB 0E	

如果我们继续往下看,我们首先会看到 Offset to PE signature,这是告诉我们 PE 头的偏移量,我们可以看到偏移量为 100。那么起始地址 400000 加上 100 就是 400100,我们定位到 400100 处。

004000FE	00	DB 00	
004000FF	00	DB 00	
00400100	50 45 00 00	ASCII "PE"	PE signature (PE)
00400104	4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
00400106	0600	DW 0006	NumberOfSections = 6
00400108	2924D90A	DD AD92429	TimeDateStamp = AD92429
0040010C	00000000	DD 00000000	PointerToSymbolTable = 0
00400110	00000000	DD 00000000	NumberOfSymbols = 0
00400114	E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
00400116	8E81	DW 818E	Characteristics = EXECUTABLE_IMAGE 32BIT_MF
00400118	0B01	DW 010B	MagicNumber = PE32
0040011A	02	DB 02	MajorLinkerVersion = 2
0040011B	19	DB 19	MinorLinkerVersion = 19 (25.)
0040011C	00060000	DD 00006000	SizeOfCode = 600 (1536.)
00400120	00220000	DD 00002200	SizeOfInitializedData = 2200 (8704.)
00400124	00000000	DD 00000000	SizeOfUninitializedData = 0
00400128	00100000	DD 00001000	AddressOfEntryPoint = 1000
0040012C	00100000	DD 00001000	BaseOfCode = 1000
00400130	00200000	DD 00002000	BaseOfData = 2000
00400134	00004000	DD 00400000	ImageBase = 400000
00400138	00100000	DD 00001000	SectionAlignment = 1000
0040013C	00020000	DD 00002000	FileAlignment = 200
00400140	0100	DW 0001	MajorOSVersion = 1

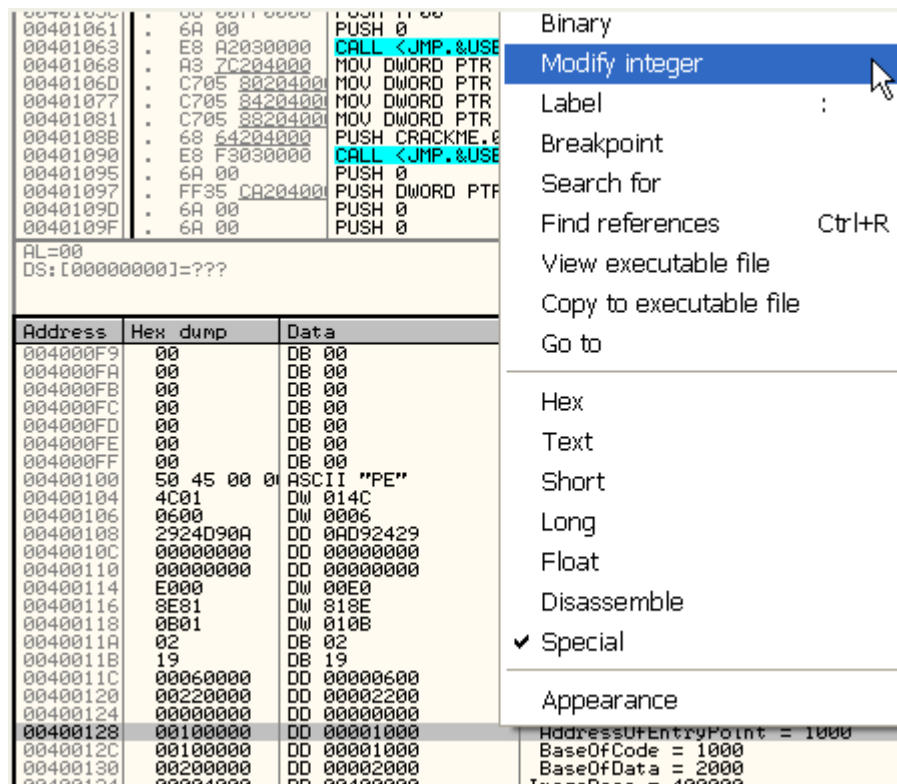
Command: BP_GetDlgItemTextA

这里你所看到的,这里是关于程序的重要信息。

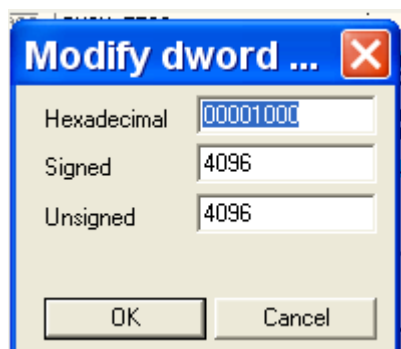
下面让我们来看看部分字节详细的解释。

0040011C	00060000	DD 00006000	SizeOfCode = 600 (1536.)
00400120	00220000	DD 00002200	SizeOfInitializedData = 2200 (8704.)
00400124	00000000	DD 00000000	SizeOfUninitializedData = 0
00400128	00100000	DD 00001000	AddressOfEntryPoint = 1000
0040012C	00100000	DD 00001000	BaseOfCode = 1000
00400130	00200000	DD 00002000	BaseOfData = 2000
00400134	00004000	DD 00400000	ImageBase = 400000

也就是说基地址 400000 加上 1000 就是程序的入口点。如果你想修改入口点的话,例如把入口点修改为 2000,我们可以在入口点这一行上单击鼠标右键。



这里我们可以输入任意数值,例如:如果你想让程序从 402000 处开始执行,我们可以输入 2000.这个 2000 是相对于映像基址 400000 的偏移量。



修改完毕以后,我们可以单击鼠标右键选择-Copy to executable file,然后在弹出的窗口中单击鼠标右键选择-Save file,这样就可以保存到文件了,我们并不修改入口点,只要知道可以这么做就行了。

好了,我们继续往下看。

004001F4	00000000	DD 00000000	Reserved
004001F8	43 4F 44 45	ASCII "CODE"	SECTION
00400200	00100000	DD 00001000	VirtualSize = 1000 (4096.)
00400204	00100000	DD 00001000	VirtualAddress = 1000
00400208	00060000	DD 00006000	SizeOfRawData = 600 (1536.)
0040020C	00060000	DD 00006000	PointerToRawData = 600
00400210	00000000	DD 00000000	PointerToRelocations = 0
00400214	00000000	DD 00000000	PointerToLineNumbers = 0
00400218	0000	DW 0000	NumberOfRelocations = 0
0040021A	0000	DW 0000	NumberOfLineNumbers = 0
0040021C	20000060	DD 60000020	Characteristics = CODE EXECUTE READ

下面各个区段的信息,首先我们看到的这个区段起始虚拟地址为 1000,注意这里是偏移量,实际上内存地址为 401000,并且 Characteristics(特征)为 CODE,EXECUTE,READ(代码段,可执行,可读)。

如果我们想让该区段具有可写权限的话,我们可以将 Characteristics 这个字段的 60000020 修改为 E0000020,这样该区段就具有了所有权限,嘿嘿,我们来验证一下。

Modify dword ...

Hexadecimal

Signed

Unsigned

OK Cancel

004001F0	00000000	DD 00000000	Reserved
004001F4	00000000	DD 00000000	Reserved
004001F8	43 4F 44 41	ASCII "CODE"	SECTION
00400200	00100000	DD 00001000	VirtualSize = 1000 (4096.)
00400204	00100000	DD 00001000	VirtualAddress = 1000
00400208	00060000	DD 00006000	SizeOfRawData = 600 (1536.)
0040020C	00060000	DD 00006000	PointerToRawData = 600
00400210	00000000	DD 00000000	PointerToRelocations = 0
00400214	00000000	DD 00000000	PointerToLineNumbers = 0
00400218	0000	DW 0000	NumberOfRelocations = 0
0040021A	0000	DW 0000	NumberOfLineNumbers = 0
0040021C	200000E0	DD E0000020	Characteristics = CODE EXECUTE READ WRITE
00400220	44 41 54 41	ASCII "DATA"	SECTION
00400228	00100000	DD 00001000	VirtualSize = 1000 (4096.)
0040022C	00200000	DD 00002000	VirtualAddress = 2000
00400230	00020000	DD 00002000	SizeOfRawData = 200 (512.)

好了,我们现在来将修改保存到文件。

00401095 : 6A 00 PUSH

00401097 : FF35 CA204000 PUSH

0040109D : 6A 00 PUSH

0040109F : 6A 00 PUSH

AL=00

DS:[00000000]=???

Address	Hex dump	Data
004001D4	00000000	DD 00000000
004001D8	00000000	DD 00000000
004001DC	00000000	DD 00000000
004001E0	00000000	DD 00000000
004001E4	00000000	DD 00000000
004001E8	00000000	DD 00000000
004001EC	00000000	DD 00000000
004001F0	00000000	DD 00000000
004001F4	00000000	DD 00000000
004001F8	43 4F 44 41	ASCII "CODE"
00400200	00100000	DD 00001000
00400204	00100000	DD 00001000
00400208	00060000	DD 00006000
0040020C	00060000	DD 00006000
00400210	00000000	DD 00000000
00400214	00000000	DD 00000000
00400218	0000	DW 0000
0040021A	0000	DW 0000
0040021C	200000E0	DD E0000020
00400220	44 41 54 41	ASCII "DATA"
00400228	00100000	DD 00001000

Find references Ctrl+R

View executable file

Copy to executable file

Go to

Hex

Text

Short

Long

Float

Disassemble

Special

Appearance

120000	DD 00000200	SizeOfRawData = 200
1C0000	DD 00000C00	PointerToRawData = 0
100000	DD 00000000	PointerToRelocations = 0
100000	DD 00000000	PointerToLineNumbers = 0
10	DW 0000	NumberOfRelocations = 0
10	DW 0000	
1000C0	DD C0000040	
69 64 6	ASCII ".idat"	
000000	DD 00001000	
000000	DD 00003000	
000000	DD 00000300	
0E0000	DD 00000E00	
000000	DD 00000000	
000000	DD 00000000	
10	DW 0000	
10	DW 0000	

Backup

Copy

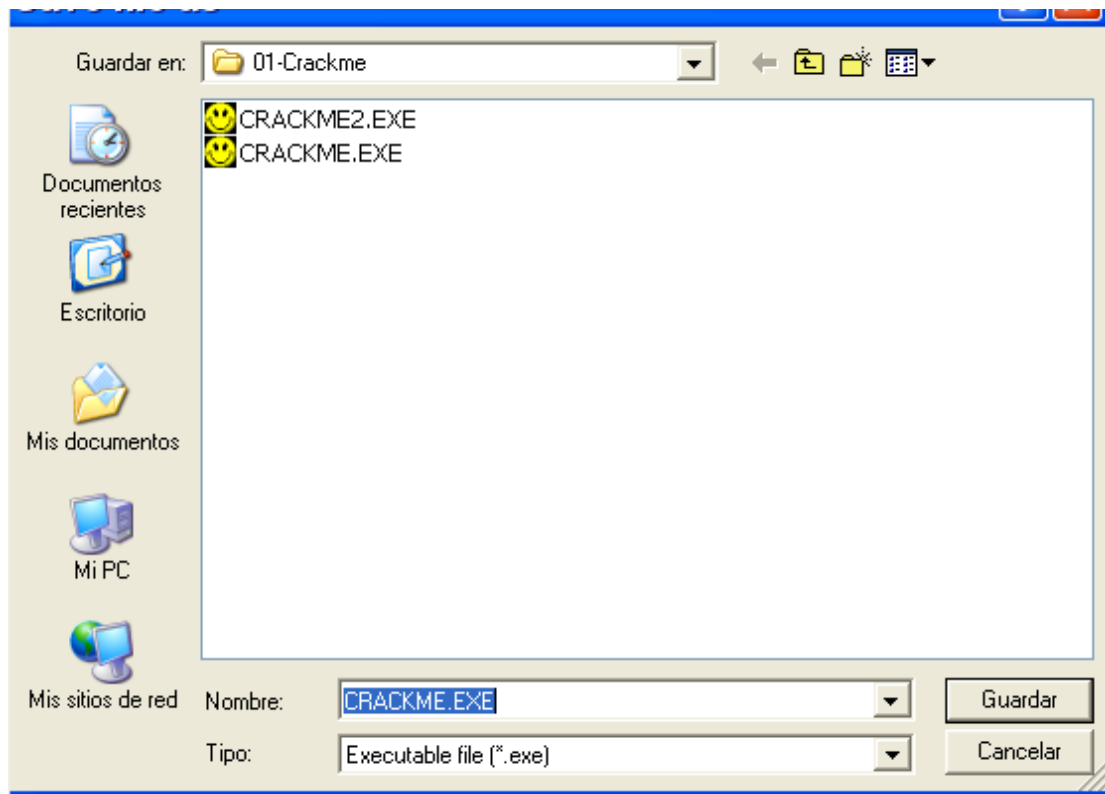
Binary

Modify integer

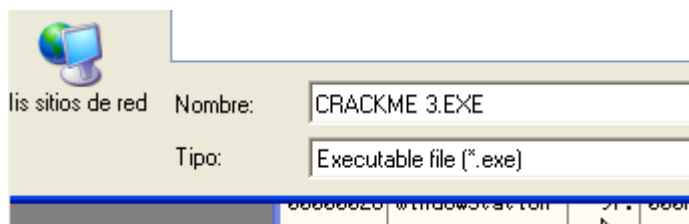
Search for

Save file

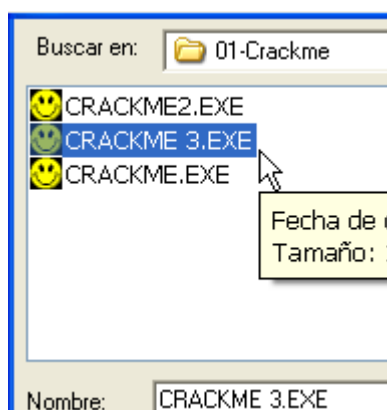
Go to offset Ctrl+G



我们将名称修改为 CRACKME 3,标识这个文件是修改版。



好了,我们现在用 OllyDbg 打开这个 CRACKME3.EXE。



Registers <FPU>	
EAX	00000000
ECX	00000000
EDX	7C92E514 ntdll.KiFastSystemCallRet
EBX	7FFDE000
ESP	0013FFC4
EBP	0013FFF0
ESI	FFFFFFFF
EDI	7C930228 ntdll.7C930228
EIP	00401002 CRACKME.00401002

这里时候除数 ECX 为 0 了。我们继续 F8 单步。

00402078	00 00 00 00	00 00 00 00
Command : <input type="text"/>			
Integer division by zero - use Shift+F7/F8/F9 to pass exception to program			

我们可以看到提示 Integer division by zero(整数除 0)异常。

无效指令,尝试执行特权指令异常:

当 CPU 试图执行越权指令的话就会产生该异常。

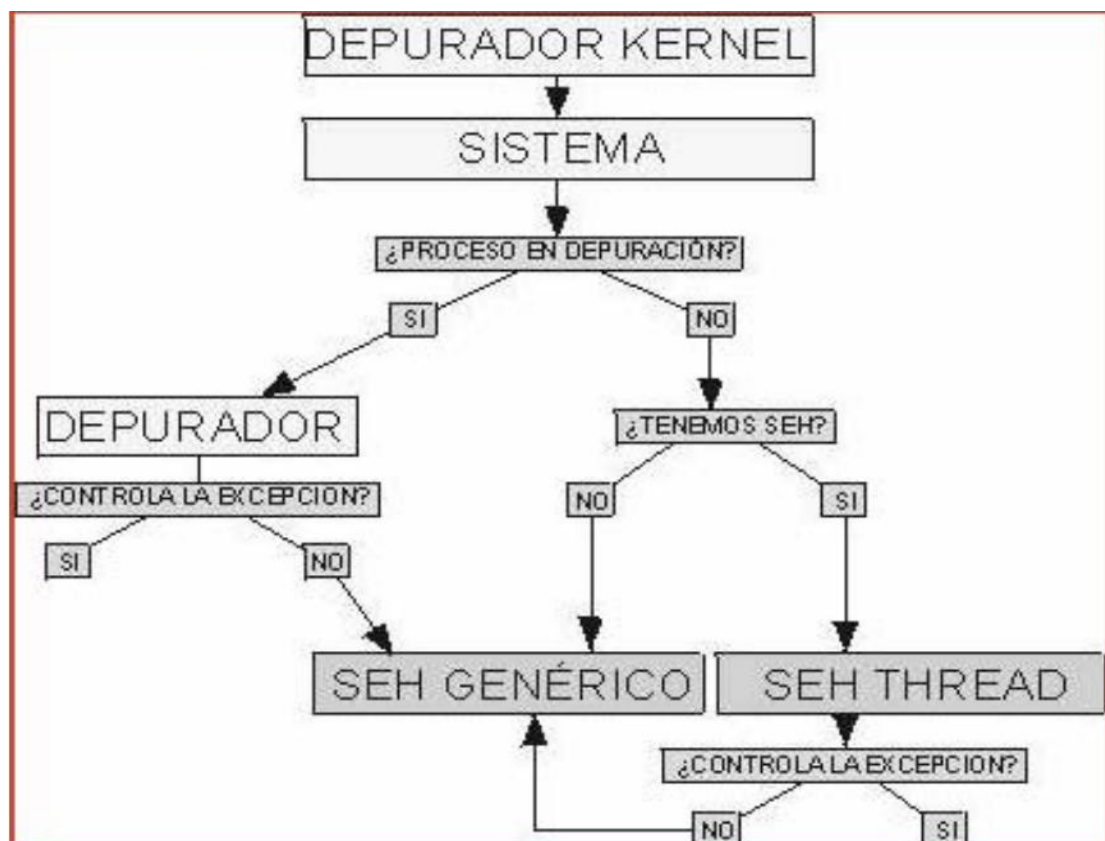
由于 OllyDbg 不允许我们输入 CPU 不可识别的指令,所以我们无法验证。但是程序员可以自己设计一些处理器并不支持的指令,当执行到指令时显示相应的错误即可。

最为典型就是 INT 3 指令,INT 3 指令会产生一个异常,并且该异常会被调试器捕获到,比如,我们可以设置 BPX 断点来让程序中断下来,然后就可以对该程序进行相应的控制了。

另外,有一些程序会直接写入 INT 3 指令,所以说 INT 3 产生的异常是最常见的。

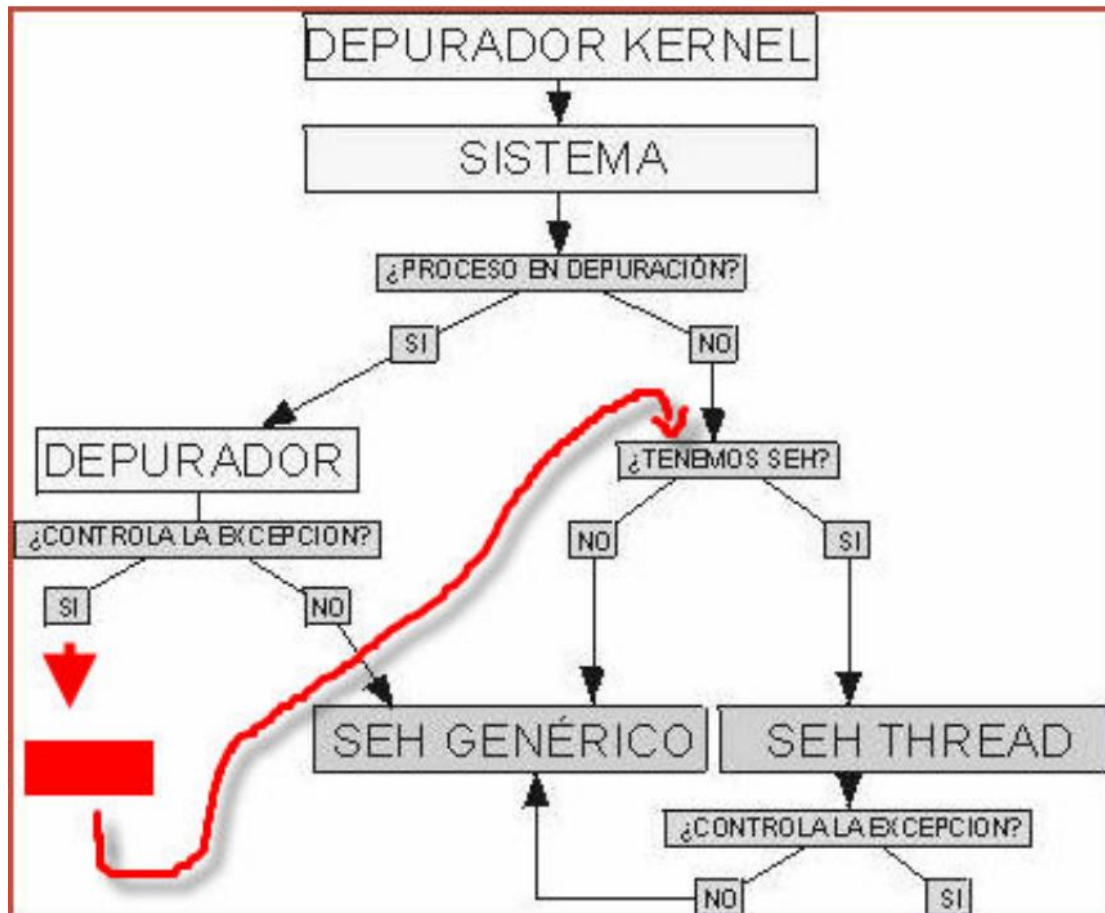
其实还有很多其他的异常,这里我们就不一一介绍了。下面我们看一个简单的例子。

现在我们只知道该程序会产生异常,但是到底会产生哪种异常呢?我们现在先来看看下面这个示意图:



这个图是我从 Mr Silver 的教程中截取出来的,这里我们可以看到一个异常是被处理的流程,首先系统会判断当前进程是否正在被调试。

根据上图来看可以知道异常发生后,如果当前程序正在被调试的话,那么此时控制权就会交予调试器。如果调试器的调试选项勾选了跳过对应的异常类型的话,这个时候控制权又会重新归还给当前程序,如果没有勾选跳过对应的异常的话,那么我们就需要按 Shift + F9 键来跳过该异常并将控制权交予程序了。但是控制权交予程序以后的流程该如何走上图中并没有标注出来。所以我们接着来看下面的流程图。



这里我补全了整个流程图,见上图的红色箭头。

接下来我们可以看到,当前控制权由调试器归还给程序以后,系统会检查当前程序是否安装了 SEH,如果安装了 SEH,就转向 SEH 的异常处理程序执行,如果没有安装 SEH,就会调用系统默认异常处理程序。以上介绍听起来可能有点复杂,其实并不复杂,我们再来详细介绍一下什么是 SEH。

什么是 SEH 呢?

SEH 或者结构化异常处理,它是用来确保该程序可以从错误中恢复,也就是说,如果你没有设置 SEH,那么当程序中有异常发生时,程序就会弹出一个错误信息框,告诉我们程序即将关闭。如果我们设置了 SEH 的话,异常处理程序就能够捕获到程序中发生的异常,进行相应的处理后,就会把控制权重新交予程序继续执行,程序并不会终止也不会弹出那个烦人的错误消息框。

此外,我们需要知道每个线程都可以有自己的异常处理程序,如果当前异常处理程序不予处理的话,可以将异常将于 SEH 链中的其他异常处理程序来处理。

如何定位异常处理程序

好了,我们用 OD 重新加载 cruehead'a 的 CrackMe。

我们来看看堆栈的情况。

0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	8445FA58	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME_.<ModuleEntryPoint>
0012FFFC	00000000	

这是系统默认安装的异常处理程序,无论什么异常交予该默认异常处理程序处理的话,它都会弹出错误消息框。现在我们来定位到该默认异常处理函数。

我们看到,FS:[0]就是指向了当前异常处理程序。

我们在数据窗口中定位到 FS:[0]。

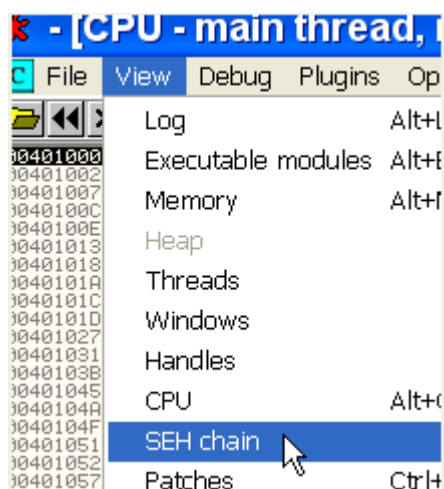
Registers (FPU)			
EAX	00000000		
ECX	0012FFB0		
EDX	7C91EB94	ntdll.KiFastSystemCallRet	
EBX	7FFDE000		
ESP	0012FFC4		
EBP	0012FFF0		
ESI	FFFFFFFF		
EDI	7C920738	ntdll.7C920738	
EIP	00401005	CRACKME_.00401005	
C 1	ES 0023	32bit 0(FFFFFFFF)	
P 0	CS 001B	32bit 0(FFFFFFFF)	
A 0	SS 0023	32bit 0(FFFFFFFF)	
Z 0	DS 0023	32bit 0(FFFFFFFF)	
S 1	FS 003B	32bit 7FFDD000(FFF)	
T 0	GS 0000	NULL	
O 0			
0 0	LastErr	ERROR_MOD_NOT_FOUND (0000007E)	
EFL	00010283	(NO,B,NE,BE,S,PO,L,LE)	
ST0	empty	-UNORM BBB0 01050104 00000000	
ST1	empty	0.0	
ST2	empty	0.0	
ST3	empty	0.0	
ST4	empty	0.0	
ST5	empty	0.0	
ST6	empty	1.000000000000000000000000	
ST7	empty	1.000000000000000000000000	

Address	Hex dump	ASCII
7FFDD000	E0 FF 12 00 00 00 13 00	0 0...!!.
7FFDD008	00 00 12 00 00 00 00 00	.00...00
7FFDD010	00 1E 00 00 00 00 00 00	.00...00
7FFDD018	00 00 FD 7F 00 00 00 00	.00...00
7FFDD020	C8 08 00 00 4C 0A 00 00	00...00
7FFDD028	00 00 00 00 00 00 00 00	00...00
7FFDD030	00 E0 FD 7F 7E 00 00 00	00...00
7FFDD038	00 00 00 00 00 00 00 00	00...00
7FFDD040	38 A2 F5 E1 00 00 00 00	00...00
7FFDD048	00 00 00 00 00 00 00 00	00...00
7FFDD050	00 00 00 00 00 00 00 00	00...00
7FFDD058	00 00 00 00 00 00 00 00	00...00
7FFDD060	00 00 00 00 00 00 00 00	00...00
7FFDD068	00 00 00 00 00 00 00 00	00...00
7FFDD070	00 00 00 00 00 00 00 00	00...00
7FFDD078	00 00 00 00 00 00 00 00	00...00

这里我们可以看到 FS:[0]指向的内存单元中内容是多少,可能不同的机器上这个值会不一样,我这里 FS:[0]指向内存单元中保存的值是 12FFE0。

0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	8445FA58	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME_.<ModuleEntryPoint>
0012FFFC	00000000	

从堆栈中我们可以看到,12FFE0 指向的是 SEH 链的最后一个结点,当前异常被交予该结点的异常处理程序的话,就会弹出一个我们熟悉的错误消息框。



我们来查看一下 SEH 链的情况。

Address	SE handler
0012FFE0	kernel32.7C8399F3

我们可以看到只有系统默认异常处理程序被安装了,当有多个异常处理程序的话,当捕获到异常的话,异常会依次由 SEH 链的顶部向底部传递。

由于 cruehead'a 的 CrackMe 并没有安装自己的异常处理程序,所以这里我们再看另一个例子 smartmouse111。

我们用 OD 加载这个例子。

C File View Debug Plugins Options Window Help			
00404AF1	55	PUSH EBP	
00404AF2	8BEC	MOV EBP,ESP	
00404AF4	6A FF	PUSH -1	
00404AF6	68 60C14000	PUSH smartmou.0040C160	
00404AFB	68 D8664000	PUSH smartmou.004066D8	SE handler installation
00404B00	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
00404B06	50	PUSH EAX	
00404B07	64:8925 0000	MOV DWORD PTR FS:[0],ESP	
00404B0E	83EC 58	SUB ESP,58	
00404B11	53	PUSH EBX	
00404B12	56	PUSH ESI	
00404B13	57	PUSH EDI	
00404B14	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
00404B17	FF15 64C04000	CALL DWORD PTR DS:[<&KERNEL32.GetVersion	kernel32.GetVersion
00404B1D	33D2	XOR EDX,EDX	
00404B1F	8AD4	MOV DL,AH	

你可以看到程序开始处在安装自己的异常处理程序,OllyDbg 中也以注释标注出来了 SE handler installation。

00404AF1	55	PUSH EBP	
00404AF2	8BEC	MOV EBP,ESP	
00404AF4	6A FF	PUSH -1	
00404AF6	68 60C14000	PUSH smartmou.0040C160	
00404AFB	68 D8664000	PUSH smartmou.004066D8	SE handler installation
00404B00	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
00404B06	50	PUSH EAX	
00404B07	64:8925 0000	MOV DWORD PTR FS:[0],ESP	
00404B0E	83EC 58	SUB ESP,58	
00404B11	53	PUSH EBX	
00404B12	56	PUSH ESI	
00404B13	57	PUSH EDI	
00404B14	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
00404B17	FF15 64C04000	CALL DWORD PTR DS:[<&KERNEL32.GetVersion	kernel32.GetVersion
00404B1D	33D2	XOR EDX,EDX	

我们到达 OllyDbg 提示 SE handler installation(安装异常处理程序)指令处,首先是一个 PUSH 4066D8 指令,当程序发生异常时,将会调用 4066D8 地址处的异常处理程序。

0012FFB4	004066D8	Entry address
0012FFB8	0040C160	smartmou.0040C160
0012FFBC	FFFFFFFF	
0012FFC0	0012FFF0	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDF000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84E80918	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00404AF1	smartmou.<ModuleEntryPoint
0012FFFC	00000000	

执行 PUSH 指令后,4066D8 被保存到堆栈中了。

00404AF6	68 60C14000	PUSH smartmou.0040C160	
00404AFB	68 D8664000	PUSH smartmou.004066D8	SE handler installation
00404B00	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
00404B06	50	PUSH EAX	
00404B07	64:8925 0000	MOV DWORD PTR FS:[0],ESP	
00404B0E	83EC 58	SUB ESP,58	
00404B11	53	PUSH EBX	
00404B12	56	PUSH ESI	

接下来一行是将 FS:[0]的值保存到 EAX 中,我们在数据窗口中来看看 FS:[0]指向的内存单元中保存的内容是多少。

Registers (FPU)									
EAX	00000000								
ECX	0012FFB0								
EDX	7C91EB94	ntdll.KiFastSystemCallRet							
EBX	7FFDF000								
ESP	0012FFB4								
EBP	0012FFC0								
ESI	FFFFFFFF								
EDI	7C920738	ntdll.7C920738							
EIP	00404B00	smartmou.00404B00							
C 1	ES 0023	32bit 0(FFFFFFFF)							
P 0	CS 001B	32bit 0(FFFFFFFF)							
A 0	SS 0023	32bit 0(FFFFFFFF)							
Z 0	DS 0023	32bit 0(FFFFFFFF)							
S 1	FS 003B	32bit 7FFDE000(FFF)							
T 0	GS 0000	NULL							
O 0									
O 0	LastErr	ERROR_MOD_NOT_FOUND (00000000)							
EFL	00000283	(NO,B,NE,BE,S,PO,L,LE)							
ST0	empty	-UNORM BCE0 01050104 00000000							
ST1	empty	0.0							
ST2	empty	0.0							
ST3	empty	0.0							
ST4	empty	0.0							
ST5	empty	0.0							
ST6	empty	1.000000000000000000000000							
ST7	empty	1.000000000000000000000000							
3 2 1 0 E S P U O Z									

我们在数据窗口中定位到 7FFDE000 地址处。

smartmou.<ModuleEntryPoint>+0F			
Address	Hex dump	ASCII	
7FFDE000	E0 FF 12 00 00 00 13 00	0 0...!!.	
7FFDE008	00 00 12 00 00 00 00 00	.S0....	
7FFDE010	00 1E 00 00 00 00 00 00	.A.....	
7FFDE018	00 E0 FD 7F 00 00 00 00	.0^0....	
7FFDE020	00 02 00 00 04 0F 00 00	00..0*..	
7FFDE028	00 00 00 00 00 00 00 00	
7FFDE030	00 F0 FD 7F 7E 00 00 00	..^0....	
7FFDE038	00 00 00 00 00 00 00 00	
7FFDE040	00 24 4E E1 00 00 00 00	0\$Np....	
7FFDE048	00 00 00 00 00 00 00 00	
7FFDE050	00 00 00 00 00 00 00 00	
7FFDE058	00 00 00 00 00 00 00 00	
7FFDE060	00 00 00 00 00 00 00 00	
7FFDE068	00 00 00 00 00 00 00 00	
7FFDE070	00 00 00 00 00 00 00 00	
7FFDE078	00 00 00 00 00 00 00 00	

我们可以看到 FS:[0]内存单元中的值为 12FFE0。OD 提示窗口中也显示了。

00404B53	.v 75 08	JNZ SHORT smart
00404B55	. 6A 1C	PUSH 1C
FS:[00000000]=[7FFDE000]=0012FFE0		
EAX=00000000		
smartmou.<ModuleEntryPoint>+0F		
Address	Hex dump	ASCII

我们执行这条指令。

Registers (FPU)									
EAX	0012FFE0								
ECX	0012FFB0								
EDX	7C91EB94	ntdll.K							
EBX	7FFDF000								
ESP	0012FFB4								
EBP	0012FFC0								
ESI	FFFFFFFF								
EDI	7C920738	ntdll.7							
EIP	00404B06	smartmo							
C 1	ES 0023	32bit 0							
P 0	CS 001B	32bit 0							
A 0	SS 0023	32bit 0							
Z 0	DS 0023	32bit 0							
S 1	FS 003B	32bit 7							
T 0	GS 0000	NULL							

EAX 的值变为了 12FFE0,接着这个值被压入堆栈。

0012FFB4	004066D8	Entry address
0012FFB8	0040C160	smartmou.0040C160
0012FFBC	FFFFFFFF	
0012FFC0	0012FFF0	
0012FFC4	7C816D4F	RETURN to kernel32
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDF000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84E80918	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00404AF1	smartmou.<ModuleEn
0012FFFC	00000000	

我们可以看到之所以叫 SEH 链,因为它是一个链表,这里的 12FFE0 就指向了上一个异常处理程序。

00404800	64:00000000	PUSH smartmou.004066D8
00404804	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
00404806	50	PUSH EAX
00404807	64:8925 0000	MOV DWORD PTR FS:[0],ESP
0040480E	83EC 58	SUB ESP,58
00404811	53	PUSH EBX
00404812	56	PUSH ESI
00404813	57	PUSH EDI
00404814	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP

下面一行指令就是将 FS:[0]的内容设置为 ESP 的内容,这样一个异常处理程序就被安装好了。

Registers (FPU)	
EAX	0012FFE0
ECX	0012FFB0
EDX	7C91EB94 ntdll
EBX	7FFDF000
ESP	0012FFB0
ESI	FFFFFFFF
EDI	7C920738 ntdll
EIP	00404807 smartmou
C 1	ES 0023 32b
P 0	CS 001B 32b
A 0	SS 0023 32b

0012FFB4	004066D8	Entry address
0012FFB8	0040C160	smartmou.0040C160
0012FFBC	FFFFFFFF	
0012FFC0	0012FFF0	
0012FFC4	7C816D4F	RETURN to kernel32
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDF000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84E80918	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00404AF1	smartmou.<ModuleEn
0012FFFC	00000000	

我们执行这一行指令。

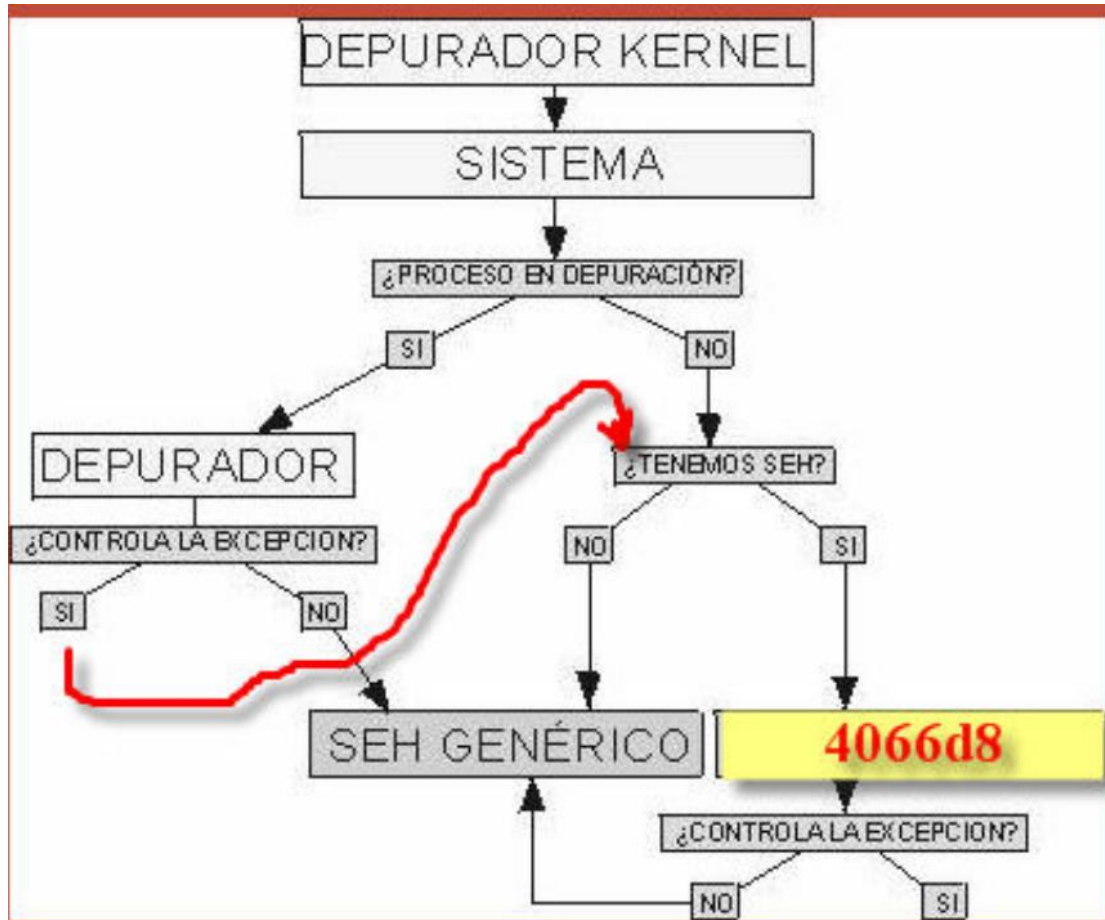
smartmou.<ModuleEntryPoint>+1D				
Address	Hex	dump	ASCII	
7FFDE000	B0 FF 12 00	00 00 13 00	...!!.	
7FFDE008	00 00 12 00	00 00 00 00	.s+....	
7FFDE010	00 1E 00 00	00 00 00 00	.A.....	
7FFDE018	00 E0 FD 7F	00 00 00 00	.0^Δ....	
7FFDE020	D0 02 00 00	04 0F 00 00	.0..é*..	
7FFDE028	00 00 00 00	00 00 00 00	
7FFDE030	00 F0 FD 7F	7E 00 00 00	..-^Δ~..	
7FFDE038	00 00 00 00	00 00 00 00	
7FFDE040	B0 24 4E E1	00 00 00 00	.Np.....	
7FFDE048	00 00 00 00	00 00 00 00	
7FFDE050	00 00 00 00	00 00 00 00	
7FFDE058	00 00 00 00	00 00 00 00	
7FFDE060	00 00 00 00	00 00 00 00	

这样 12FFB0 处就是我们安装的 新的 SEH 结点了,也就是 FS:[0]指向了我们新安装的 SEH 结点。

0012FFB0	0012FFE0	Pointer to next SEH record
0012FFB4	004066D8	SE handler
0012FFB8	0040C160	smartmou.0040C160
0012FFBC	FFFFFFFF	
0012FFC0	0012FFF0	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDF000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84E80918	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00404AF1	smartmou.<ModuleEntryPoint>
0012FFFC	00000000	

OllyDbg 也标注出来了,提示这是一个 SEH 结点,首先的 4 个字节的值指向了老的 SEH 结点,接下来的 4 个字节值即当前的异常处理程序入口地址。

所以当该程序发生异常后,异常被处理流程如下:



判断当前是否被调试,由于当前正在被调试,所以系统将控制权交予调试器,然后如果你勾选了忽略对应异常的选项的话(如果你没有勾选忽略对应异常选项的话,你也可以按 Shift + F9 键来忽略异常),那么控制权将重新交予程序,如图所示,接着判断是否安装了 SEH,这里安装了,所以将会执行 4066D8 处的异常处理程序。

我们来看看 SEH 链的情况:

Address	SE handler
0012FFB0	smartmou.004066D8
0012rFE8	kernel32.7C8099F3

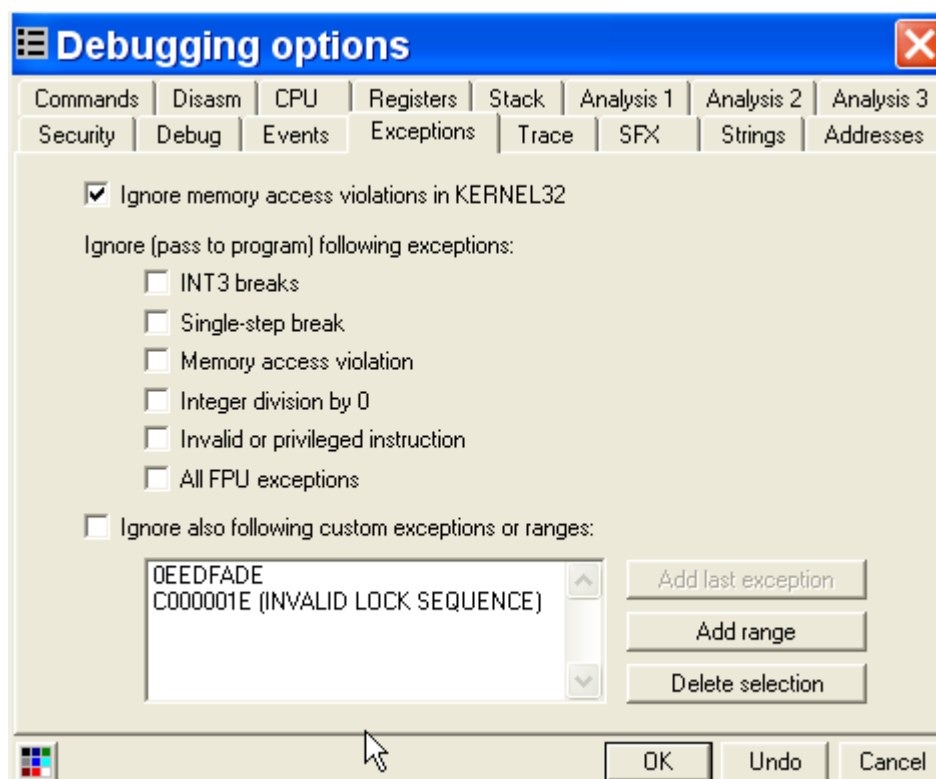
我们可以看到 SEH 链的顶部是程序自己安装的异常处理程序,接下来才是系统默认异常处理程序,如果发生异常,应该是调用 4066D8 处异常处理程序,而不是调用系统默认异常处理程序,我们来手工制造一个异常试试。

Address	Disassembly	Comment
00404AF1	55	PUSH EBP
00404AF2	8BEC	MOV EBP,ESP
00404AF4	6A FF	PUSH -1
00404AF6	68 60C14000	PUSH smartmov.0040C160
00404AF8	68 08664000	PUSH smartmov.004066D8
00404B00	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
00404B06	50	PUSH EAX
00404B07	64:8925 0000	MOV DWORD PTR FS:[0],ESP
00404B0E	83EC 58	SUB ESP,58
00404B11	53	PUSH EBX
00404B12	56	PUSH ESI
00404B13	57	PUSH EDI
00404B14	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
00404B17	FF15 64C04000	CALL DWORD PTR DS:[<&KERNEL32.GetVersion]
00404B1D	33D2	XOR EDX,EDX
00404B1F	8AD4	MOV DL,AH
00404B21	8915 30D54200	MOV DWORD PTR DS:[42D538],EDX
00404B27	8BC8	MOV ECX,EAX
00404B29	81E1 FF000000	AND ECX,0FF
00404B2F	890D 34D54200	MOV DWORD PTR DS:[42D534],ECX
00404B35	C1E1 08	SHL ECX,8
00404B38	03CA	ADD ECX,EDX
00404B3A	890D 30D54200	MOV DWORD PTR DS:[42D530],ECX
00404B40	C1FB 10	SHR ECX,10

我们将该行修改为如下指令:

Address	Disassembly	Comment
00404B14	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP
00404B17	FF15 64C04000	CALL DWORD PTR DS:[<&KERNEL32.GetVersion]
00404B1D	33D2	XOR EDX,EDX
00404B1F	8AD4	MOV DL,AH
00404B21	8915 00000000	MOV DWORD PTR DS:[0],EDX
00404B27	8BC8	MOV ECX,EAX
00404B29	81E1 FF000000	AND ECX,0FF
00404B2F	890D 34D54200	MOV DWORD PTR DS:[42D534],ECX
00404B35	C1E1 08	SHL ECX,8

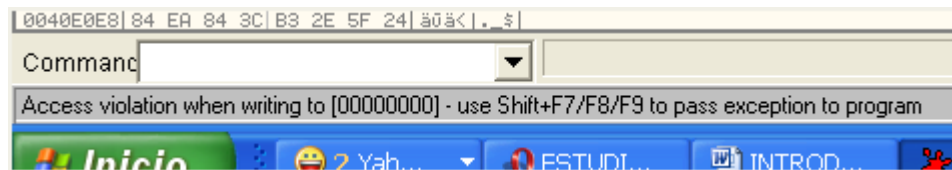
这样会产生一个异常,因为0地址不能写入。我们确保调试选项中忽略各类异常的选项没有被勾选,但是第一个选项还是要勾选的。



我们运行起来。

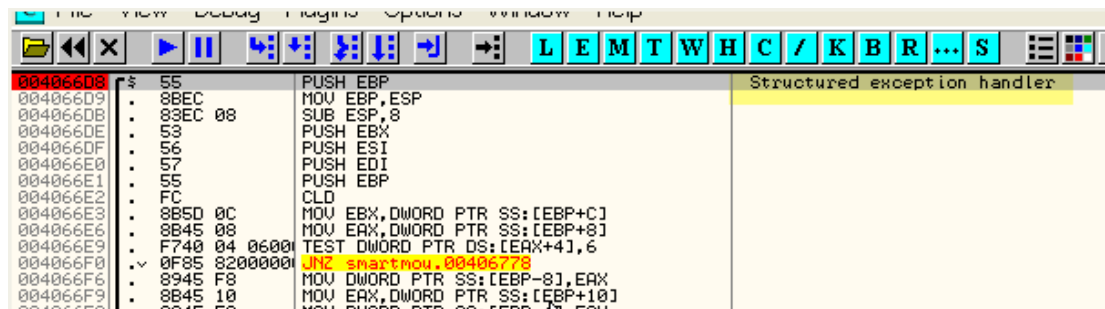
Address	Disassembly	Comment
00404B17	FF15 64C04000	CALL DWORD PTR DS:[<&KERNEL32.GetVersion]
00404B1D	33D2	XOR EDX,EDX
00404B1F	8AD4	MOV DL,AH
00404B21	8915 00000000	MOV DWORD PTR DS:[0],EDX
00404B27	8BC8	MOV ECX,EAX
00404B29	81E1 FF000000	AND ECX,0FF
00404B2F	890D 34D54200	MOV DWORD PTR DS:[42D534],ECX
00404B35	C1E1 08	SHL ECX,8

OD 左下方显示错误,程序将被终止。

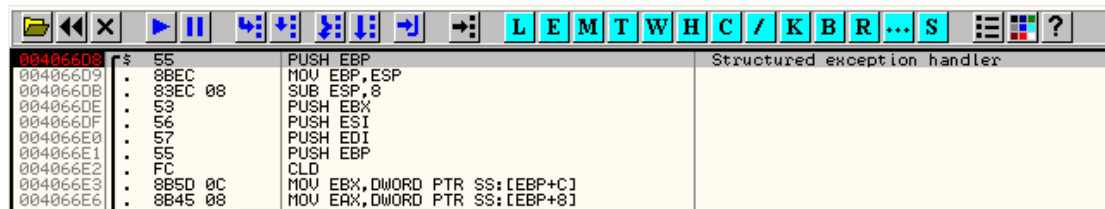


现在程序继续执行的话可以尝试从错误中恢复。

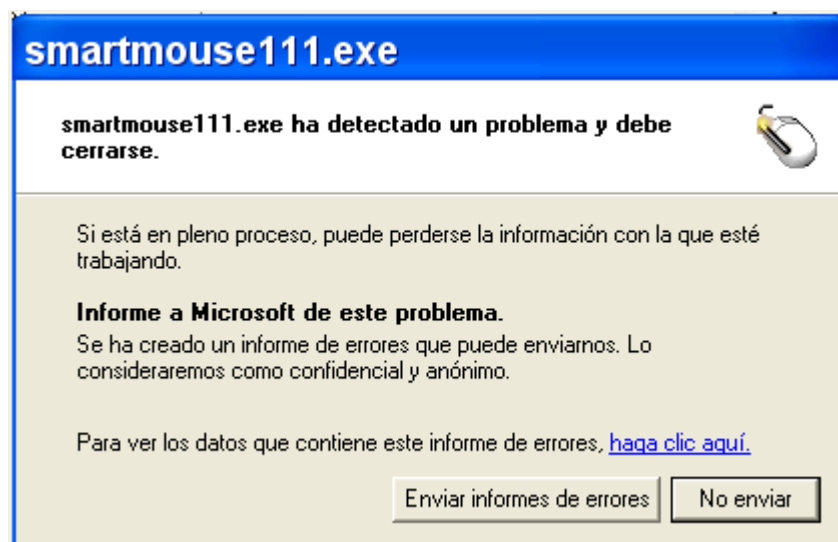
我们在 4066D8 指向的异常处理程序入口处设置一个断点。



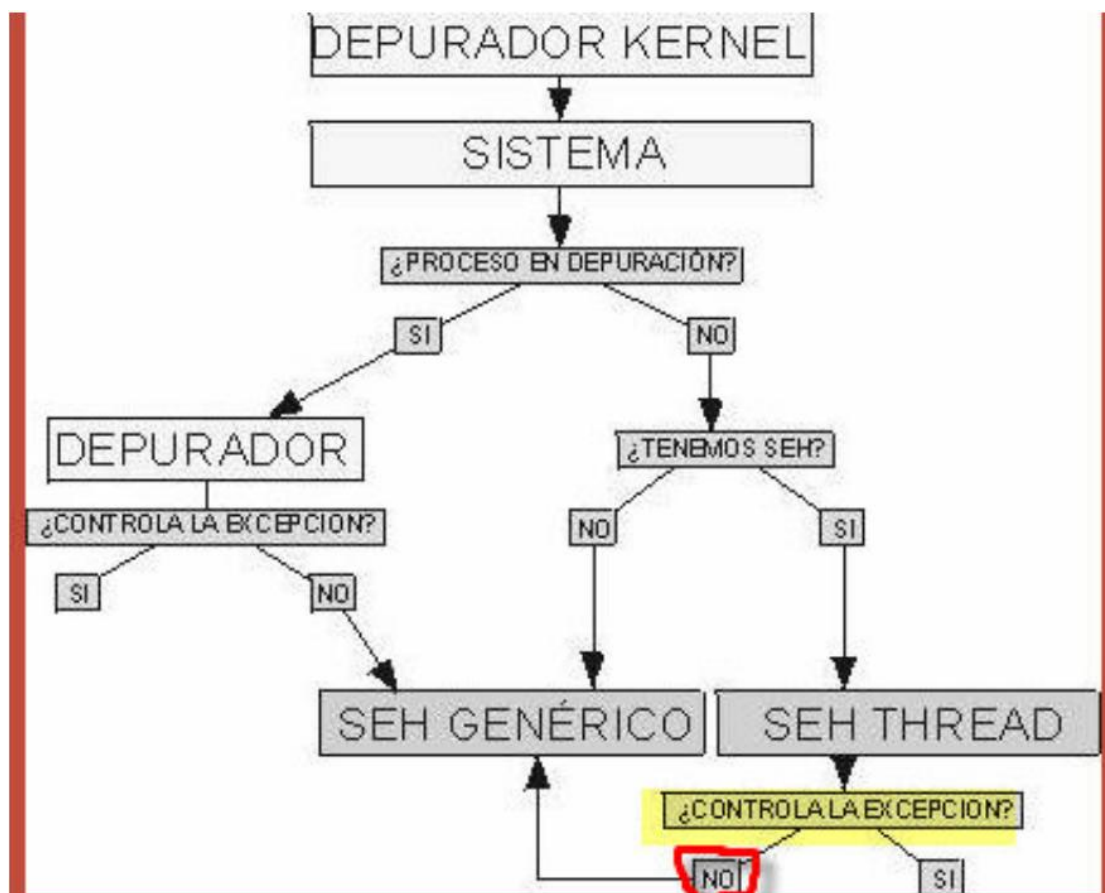
我们可以看到 OD 提示我们,按 Shift + F9 键可以忽略异常,继续执行程序,嘿嘿。



我们可以看到断在了异常处理程序的入口处,我们运行起来看看程序会不会从错误中恢复过来。



我们可以看到程序崩溃,弹出了错误消息框,这该程序表明调用了系统默认异常处理程序。

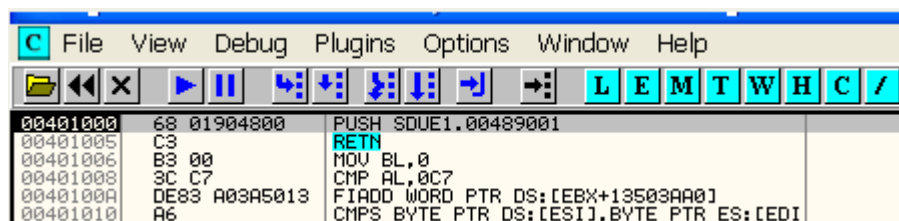


显示这个错误提示框是因为程序自己安装 SEH 异常处理程序中并没有修复刚刚那个异常,所以异常继续传递,最后交予了系统默认的异常处理程序,将弹出了一个错误消息框,程序就终止掉了。

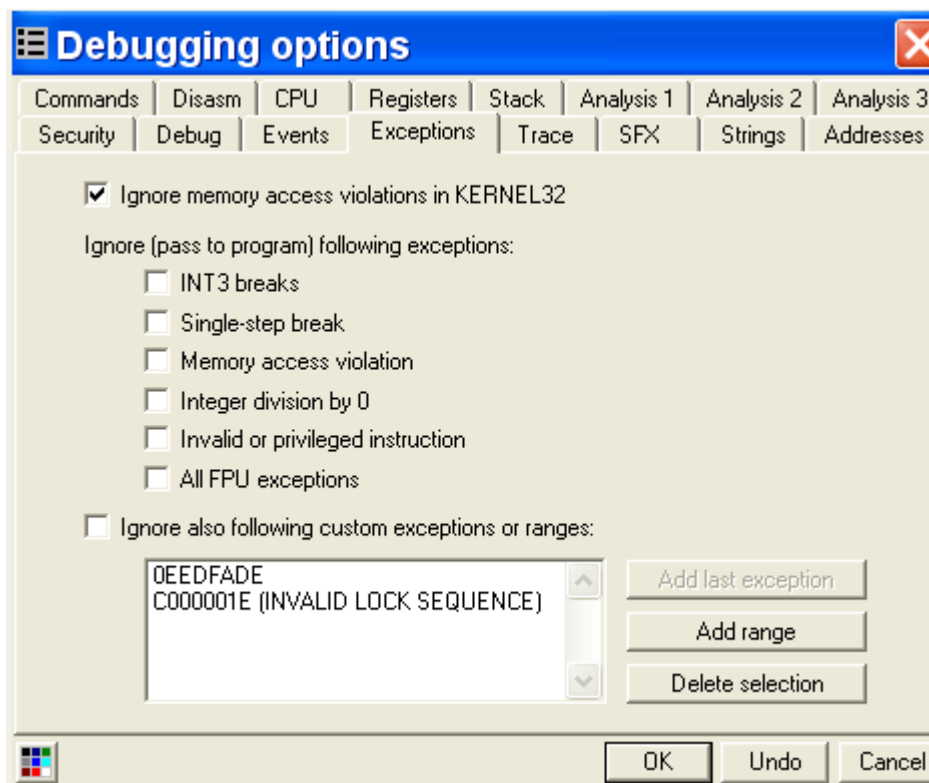
显然,程序自己安装的异常处理程序是用来处理别的类型的异常的,并不能处理向 0 地址处写入导致的异常。

为了能看到异常被成功处理的效果,我们再来看一个例子 SDUE1。

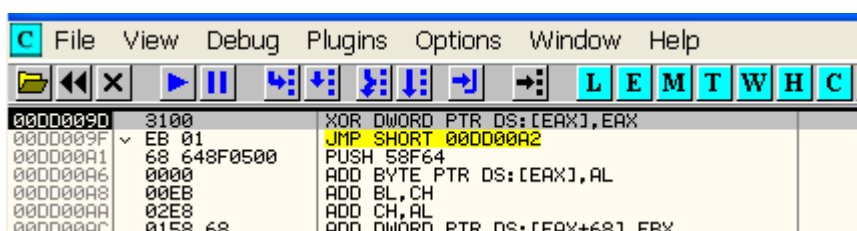
我们用 OD 加载该程序,可以看到 OD 提示说该程序可能被加壳了。



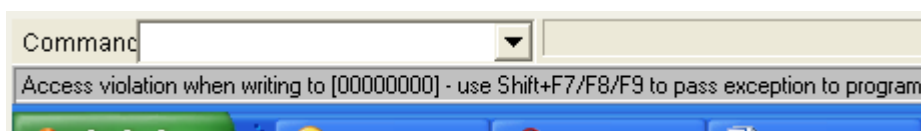
我们依然不勾选调试选项中的忽略各类异常的选项,除了忽略第一个异常以外。



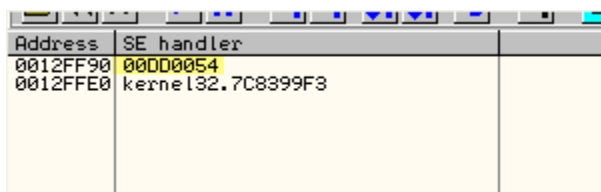
我们运行起来。



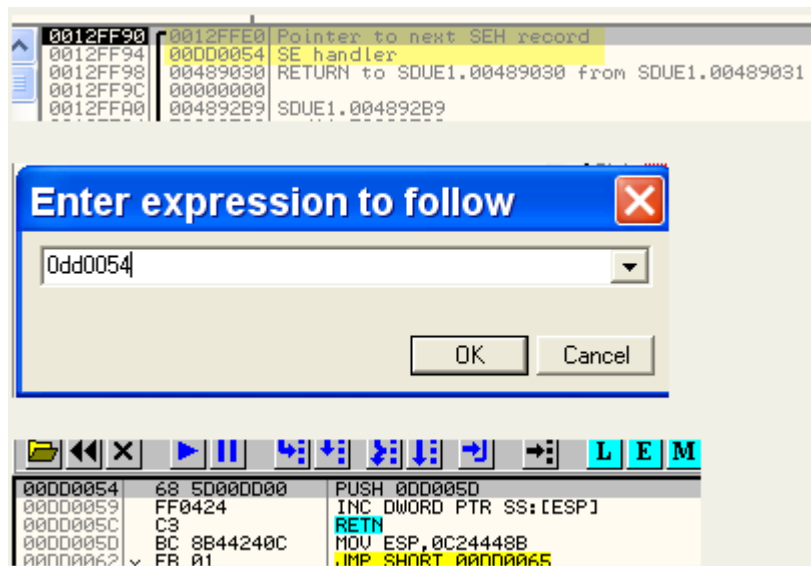
我们可以看到发生了异常,断了下来。



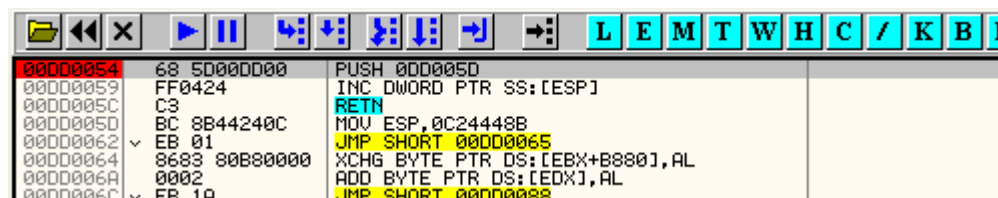
我们来看看异常处理程序在哪里。



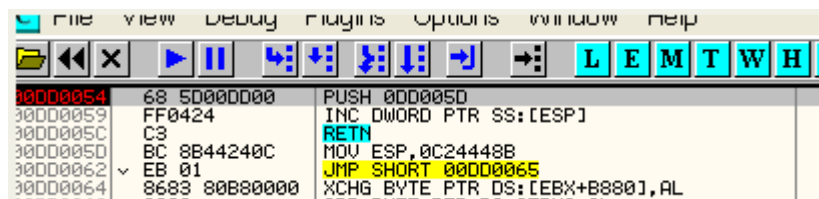
在你的机器上,这个地址可能会不一样,因为该地址属于一个动态创建的区段。我们现在来给该异常处理程序设置一个断点。



我们定位到了该异常处理程序的入口地址,现在我们该它设置一个断点。



我们按 Shift + F9 键运行起来。



我们可以看到断在了异常处理程序的入口处,如果成功从异常中恢复了的话,那么程序将会从刚刚发生异常的指令的下一条指令处继续往下执行。



我们给产生异常的指令的下一条指令设置一个断点,然后运行起来。



我们可以看到程序继续执行起来了,并弹出提示错误消息框,说明异常已经成功被修复了。

其实设置异常处理程序还可以使用 SetUnhandledExceptionFilter 这个 API 函数,可以通过其参数来设置异常处理程序的入口地址。

好了,本章介绍了我们以后破解过程中会用到的一些知识点,下一章开始我们将介绍 VB 相关的内容。

