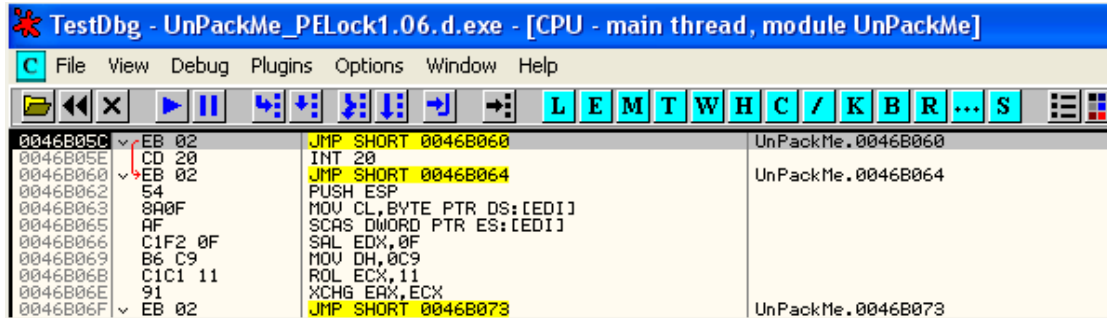


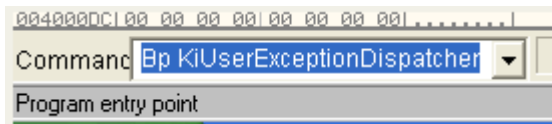
### 第三十九章-神马是 stolen bytes

本章与下一章节将介绍 stolen bytes(PS:壳偷代码)以及 OD 脚本编写方面的内容。我们拿 UnPackMe\_PELock1.06 来讲解。这款壳拿来介绍 stolen bytes 正合适,很多经典的教程都拿它作为范例来讲解。

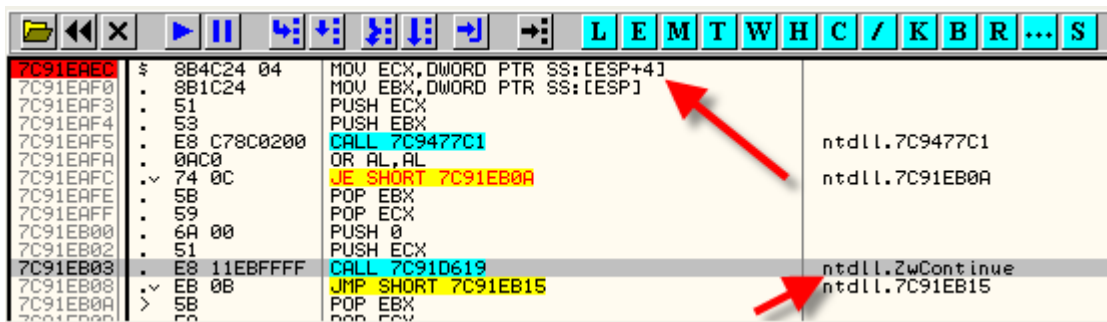
用 OD 加载它,停在了入口点处。



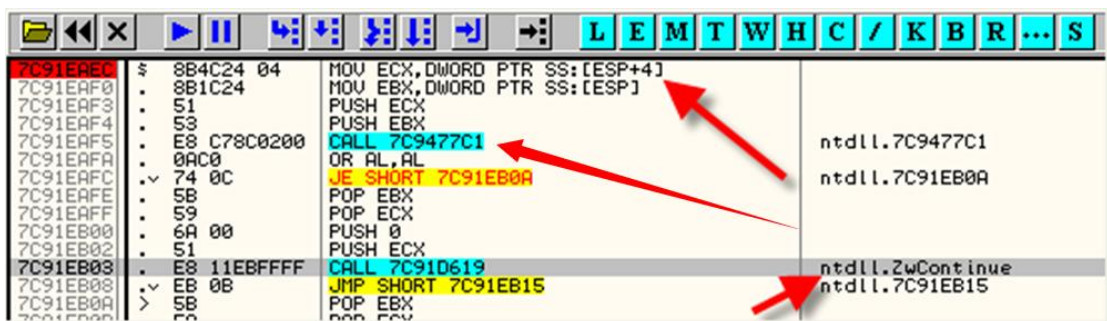
直接按 F9 键运行起来,会发现该程序会异常报错弹个错误框出来,直接 shift + F9 忽略异常继续运行,弹出主程序对话框,这里可以通过日志窗口中最后异常发生处来定位 OEP,大家可以自行尝试,我不再赘述了,下面我们换种方法来定位最后一次异常发生处。



这里我们给 Ring3 异常分发函数 KiUserExceptionDispatcher 设置一个断点,所有的异常都会经过这一个点,我们在 OD 中定位到这个函数。



我们来分析一下这个函数。



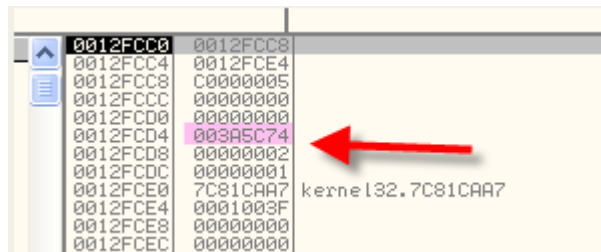
最上面的红色箭头标注的是 NTDLL.DLL 中的 KiUserExceptionDispatcher 的起始地址,紧接着下面有一个 CALL,这里是调用 SEH 链中的异常处理函数,执行完毕以后返回,根据返回的结果来决定是否继续执行程序。

这里我们就没法继续往里面跟了,因为 OD 不能跟进 RING0,这里我们只能对异常处理函数下断点或者对触发异常的区段设置内存访问断点。这样 RING0 的部分执行完了以后,OD 就断在异常产生处所在区段。

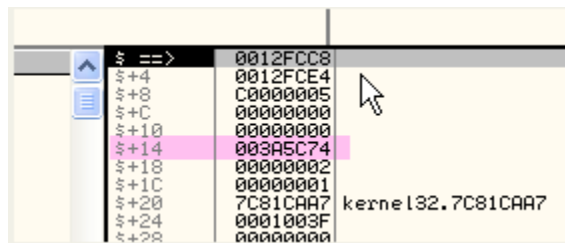
好,现在我们运行起来,第一次异常断了下来。

```
0046B05C main thread with ID 00000000 created
00400000 Module C:\Documents and Settings\Ricardo\Escritorio\UnPackMe_PELock1.06
7C91E000 Code section extended to include self-extractor
7C91E000 Module C:\WINDOWS\system32\kernel32.dll
0046B05C Module C:\WINDOWS\system32\ntdll.dll
003A5C74 Program entry point
7C91EAE0 Access violation when writing to [7C81CAA7]
7C91EAE0 Breakpoint at ntdll.KiUserExceptionDispatcher (KiUserApcDispatcher+2C)
```

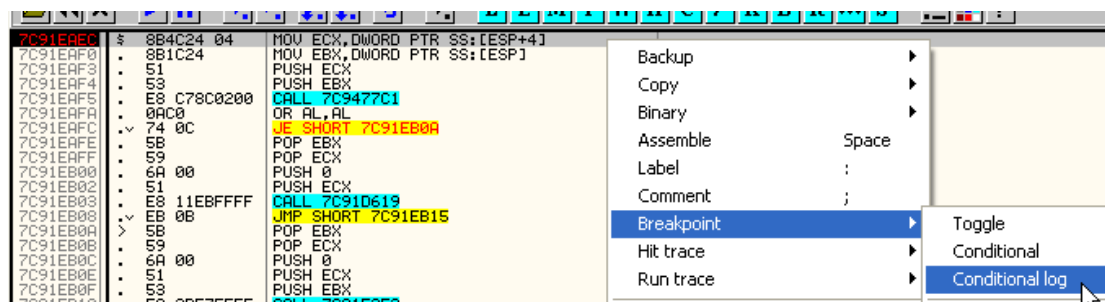
我们在日志窗口中可以看到(在我的机器上)异常发生在 3A5C74 处,好,接下来把忽略异常的调试选项都勾选上,我们给该异常发生处设置一个断点,接着我们看看堆栈窗口。



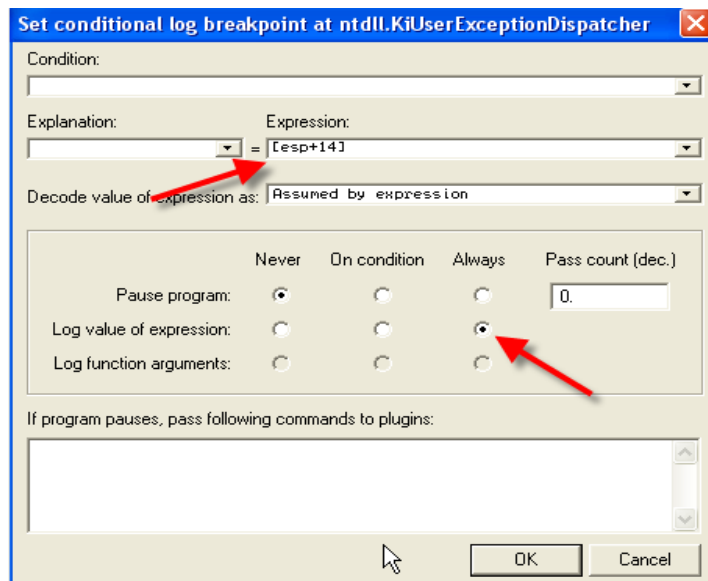
我们可以看到红色箭头标注处,也存放了异常发生的地址,确切的说是[ESP + 14]。



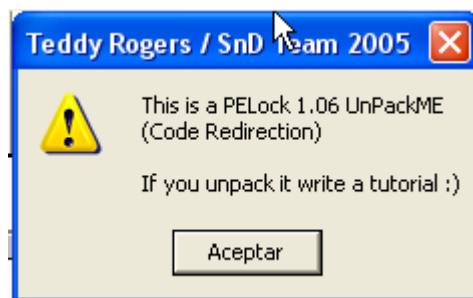
因此,这里我们用条件记录断点代替 INT 3 断点来记录[ESP + 14]的值,即记录异常触发的地址,嘿嘿。



这里我们在 KiUserExceptionDispatcher 的起始地址处单击鼠标右键选择-Breakpoint Conditional log。



这里我们将 Expression 设置为[ESP + 14](异常发生的地址),Pause program 设置为 Never,Log value of expression 设置为 Always,运行起来。



我们可以看到程序运行起来了,奇怪,这个壳居然没有检测条件断点,也没有对 KiUserExceptionDispatcher 下断进行检测。

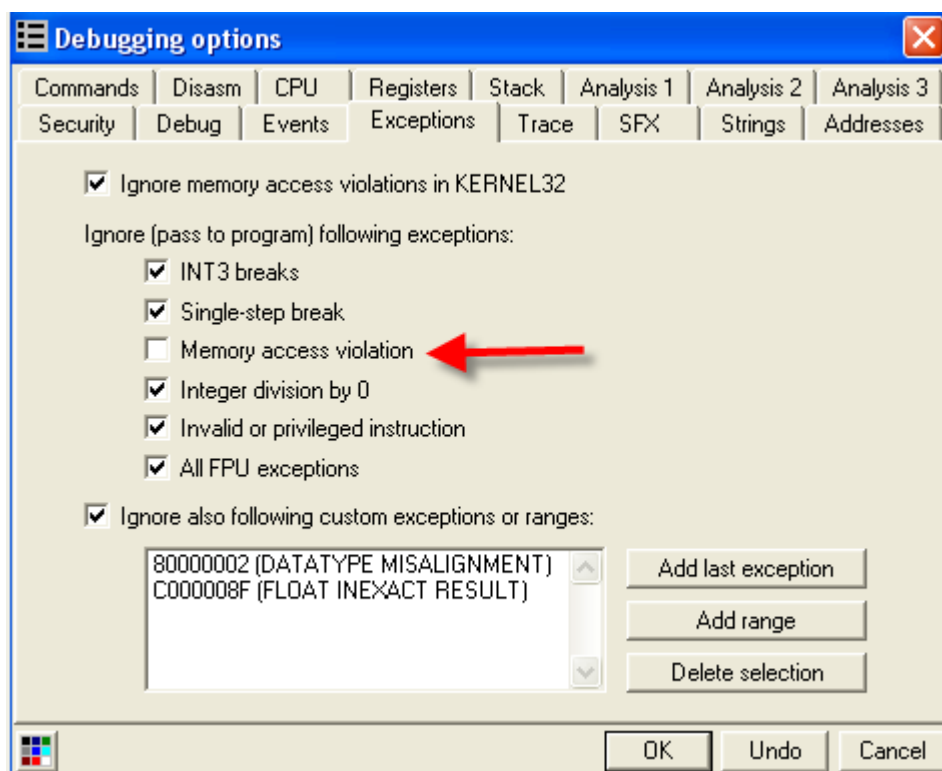
好,下面我们来看看日志窗口中记录的结果。

Address	Message
0046B05C	Program entry point
003A5C74	Access violation when writing to [7C81CAA7]
7C91EAE0	Breakpoint at ntdll.KiUserExceptionDispatcher (KiUserApcDispatcher+2C)
003A1CED	Integer division by zero
7C91EAE0	COND: 003A1CED
003A24EB	Access violation when writing to [00000000]
7C91EAE0	COND: 003A24EB
003A24EE	Integer division by zero
7C91EAE0	COND: 003A24EE
003A2698	Access violation when writing to [00000000]
7C91EAE0	COND: 003A2698
003A269B	Integer division by zero
7C91EAE0	COND: 003A269B
003A2851	Access violation when writing to [00000000]
7C91EAE0	COND: 003A2851
003A2854	Integer division by zero
7C91EAE0	COND: 003A2854
003A3374	Access violation when reading [FFFFFFFF]
7C91EAE0	COND: 003A3374
003A33CB	Illegal instruction
7C91EAE0	COND: 003A33CB
003A3374	Access violation when reading [FFFFFFFF]
7C91EAE0	COND: 003A3374
003A33CB	Illegal instruction
7C91EAE0	COND: 003A33CB
003A3374	Access violation when reading [FFFFFFFF]
7C91EAE0	COND: 003A3374
003A33CB	Illegal instruction
7C91EAE0	COND: 003A33CB
003A3374	Access violation when reading [FFFFFFFF]
7C91EAE0	COND: 003A3374
003A33CB	Illegal instruction
7C91EAE0	COND: 003A33CB
003A3652	Access violation when writing to [00000000]
7C91EAE0	COND: 003A3652
003A3655	Integer division by zero
7C91EAE0	COND: 003A3655
003A37DE	Access violation when writing to [00000000]
7C91EAE0	COND: 003A37DE
003A37E1	Integer division by zero
7C91EAE0	COND: 003A37E1
003A3873	Access violation when writing to [00000000]
7C91EAE0	COND: 003A3873
003A3876	Integer division by zero
7C91EAE0	COND: 003A3876
76B00000	Module C:\WINDOWS\system32\WINMM.dll
77D10000	Module C:\WINDOWS\system32\USER32.dll

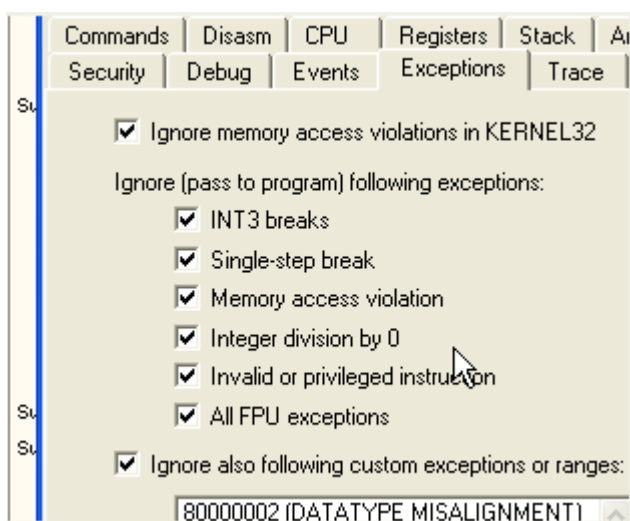
我们可以看到记录了很多异常触发的地址,我们往下看。

774B0000	Module C:\WINDOWS\system32\ole32.dll
7C81084E	Breakpoint at kernel32.7C81084E
003A6744	Access violation when writing to [00000000]
7C91EAE0	COND: 003A6744
774B0000	New thread with ID 00000110 created
770F0000	Module C:\WINDOWS\system32\OLEAUT32.dll
5B150000	Module C:\WINDOWS\system32\uxtheme.dll
746B0000	Module C:\WINDOWS\system32\MSCTF.dll

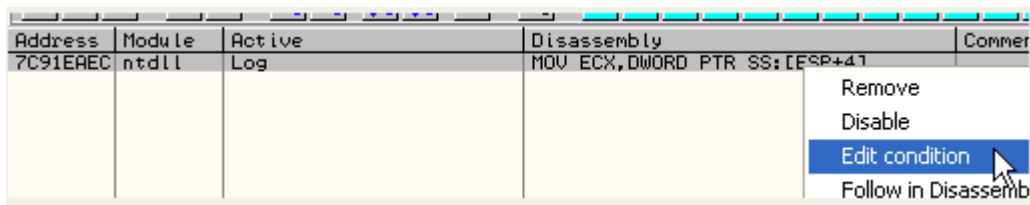
我们可以看到最后一个异常发生在 3A6744 处,是非法访问异常,这里我们不能对该处设置 INT 3 断点,因为该壳会有检测导致程序无法运行。

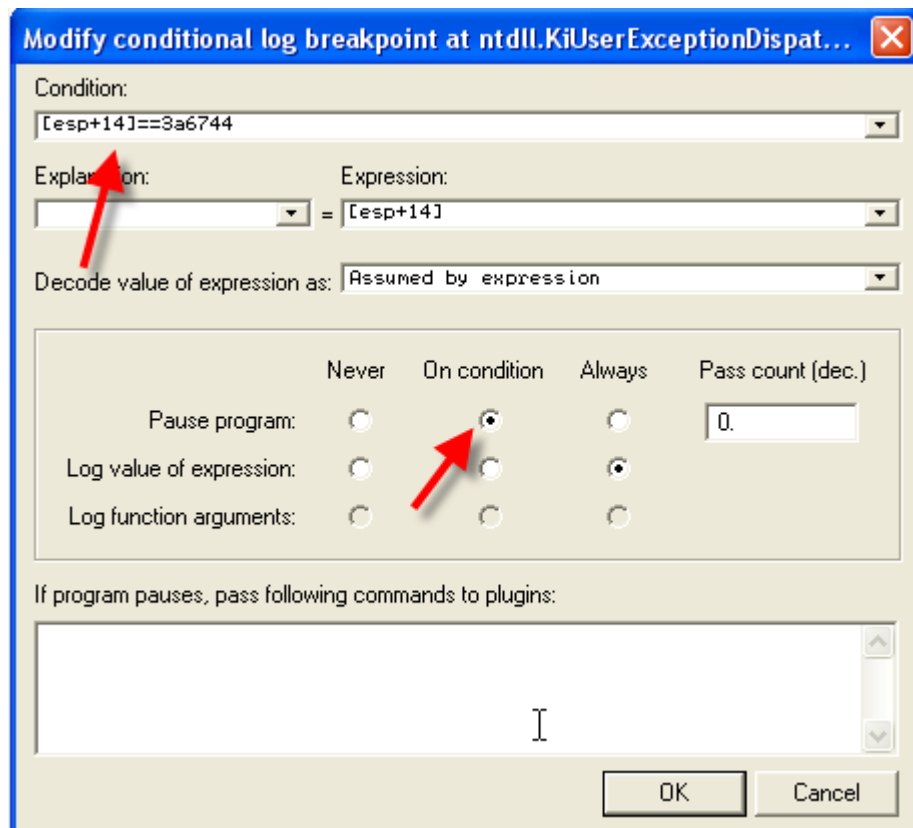


这里我们可以将 Memory access violation 这一项对勾去掉,当产生内存访问异常的时候就会断下来,虽然这种方法可以奏效,都是只要是内存访问异常就会断下来,会断很多次,其实我们还有更快速定位的方法,这里我们还是将 Memory access violation 这一项勾选上。



我们还是来设置条件断点,我们打开断点列表,在断点上单击鼠标右键选择-Edit condition。





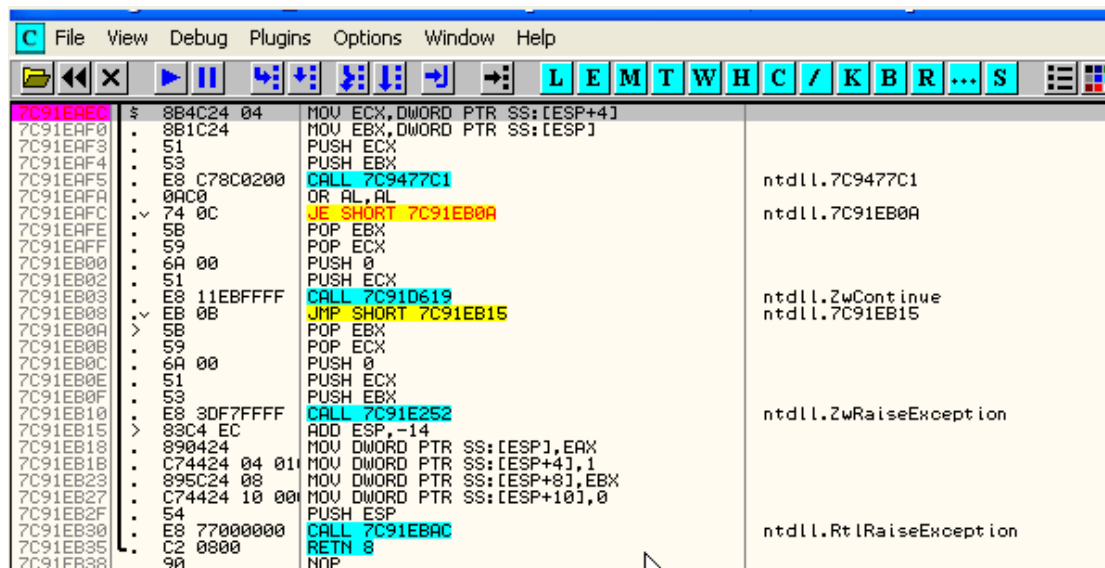
这里根据日志信息中显示的最后一次异常发生在 3A7644 地址处,我们只需添加一个条件就能让其断在最后一次异常处,我们设置条件为[ESP + 14] == 3A7644,这样当最后异常触发时就会断下来。

```

774B0000| Module C:\WINDOWS\system32\ole32.dll
7C81084E| Breakpoint at kernel32.7C81084E
003A6744| Access violation when writing to [00000000]
7C91EAE0| COND: 003A6744
7C810856| New thread with ID 00000110 created
770F0000| Module C:\WINDOWS\system32\OLEAUT32.dll
5B150000| Module C:\WINDOWS\system32\uxtheme.dll
746B0000| Module C:\WINDOWS\system32\MSCTF.dll

```

接着将 Pause program 这一项设置为 On condition,这样当条件满足时就会断下来,我们运行起来。



这里,就断到了最后一次异常处,我们没有必要一个异常一个异常的去定位。

00340000	00010000	UnPackMe	.teddy	PE header	File	rw	rw	
00400000	00001000	UnPackMe	.teddy	code	Image	R	RWE	
00401000	00004000	UnPackMe	.teddy		Image	R	RWE	
0044B000	0000C000	UnPackMe	.teddy					
00457000	00009000	UnPackMe	.teddy					
00460000	00003000	UnPackMe	.teddy					
00463000	00008000	UnPackMe	.teddy	resource:				
0046B000	0000A000	UnPackMe	.teddy	SFX, impo				
00480000	00021000							
00480000	0000C000							
00570000	00002000							
00580000	00103000							
00690000	00001000							
006A0000	001A2000							
009A0000	00001000							
009B0000	00004000							
009C0000	00003000							
009D0000	00001000							
00A50000	00004000							
00A60000	00003000							

Actualize

View in Disassembler Enter

Dump in CPU

Dump

Search Ctrl+B

Set break-on-access F2

Set memory breakpoint on access

Set memory breakpoint on write

Address	Disassembly	Comment
00427106	JMP 00A84865	
00427108	ADD BYTE PTR DS:[EBX],DH	
0042710D	ROR BYTE PTR DS:[EDX+341589D4],CL	I/O command
004271E3	OUT 45,AL	
004271E5	ADD BYTE PTR DS:[EBX+FFE181C8],CL	
004271E8	ADD BYTE PTR DS:[EAX],AL	
004271ED	ADD BYTE PTR DS:[ECX+45E6300D],CL	
004271F3	ADD CL,AL	
004271F5	LOOPDE SHORT 004271FF	UnPackMe.004271FF
004271F7	ADD ECX,EDX	
004271F9	MOV DWORD PTR DS:[45E62C],ECX	
004271FF	SHR EAX,10	
00427202	MOV DWORD PTR DS:[45E628],EAX	
00427207	CALL 004293A0	UnPackMe.004293A0
0042720C	TEST EAX,EAX	
0042720E	JNZ SHORT 0042721A	UnPackMe.0042721A
00427210	PUSH 1C	

stolen bytes 了以后会怎么样呢?很简单,如果我们 dump 以后,修复 IAT 的时候,这里 OEP 就会被指定为 4271D6,但是这样的话程序就无法运行了,因为前面几行代码还没有执行,这几行代码在壳空间中,所以是不会被转读的,因此也得不到执行。

一种方式就是尝试跟踪壳的代码,当所有异常都触发以后我们就会到达错误 OEP 处,我们将跟踪过程中执行过的代码都保存到一个 txt 中,便于我们的分析,这里我们在 JMP 到错误的 OEP 4271D6 之前都执行些什么呢?

				<div>L E M T W H C / K B R ... S</div>
0046B05C	EB 02	JMP SHORT 0046B060	UnPackMe.0046B060	
0046B05E	CD 20	INT 20		
0046B060	EB 02	JMP SHORT 0046B064	UnPackMe.0046B064	
0046B062	54	PUSH ESP		
0046B063	8A0F	MOV CL, BYTE PTR DS:[EDI]		
0046B065	AF	SCAS DWORD PTR ES:[EDI]		
0046B066	C1F2 0F	SAL EDX, 0F		
0046B069	B6 C9	MOV DH, 0C9		
0046B06B	C1C1 11	ROL ECX, 11		
0046B06E	91	XCHG EAX, ECX		
0046B070	FD 83	JMP SHORT 0046B070	UnPackMe.0046B070	

我们看下栈顶:



0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD4000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	81AA7020	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0046B05C	UnPackMe.<ModuleEntryPoint>
0012FFFC	00000000	

我的机器,当前指针为 12FFC4,大家有可能是其他的值,当到达真正的 OEP 处时,一般来说,栈顶指针应该也会指向 12FFC4 或者附近的地址,但是如果是假的 OEP 呢?

我们再次定位到假的 OEP 处。

		0 0 LastErr ERROR_SUCCESS (00000000)
		EFL 00010217 (NO,B,NE,BE,NS,PE,GE,G)
0012FF44	CC6EA31E	
0012FF48	003A6C9B	
0012FF4C	00000000	
0012FF50	746E656D	
0012FF54	6E612073	
0012FF58	65532064	
0012FF5C	6E697474	
0012FF60	525C7367	
0012FF64	72616369	
0012FF68	455C6F64	
0012FF6C	69726373	
0012FF70	69726F74	
0012FF74	6E555C6F	
0012FF78	6B636150	
0012FF7C	505F654D	
0012FF80	636F4C45	
0012FF84	302E3168	
0012FF88	2E642E36	
0012FF8C	22657865	
0012FF90	0056A028	
0012FF94	C0000034	
0012FF98	00000000	
0012FF9C	004A6C04	
0012FFA0	0012FF44	
0012FFA4	00000048	
0012FFA8	0012FFB8	Pointer to next SEH record
0012FFAC	004292C8	SE handler
0012FFB0	00450E60	UnPackMe.00450E60
0012FFB4	FFFFFFFF	
0012FFB8	0012FFE0	Pointer to next SEH record
0012FFBC	7C816D4F	SE handler
0012FFC0	0012FFF0	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD4000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	81AA7020	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0046B05C	UnPackMe.<ModuleEntryPoint>
0012FFFC	00000000	

如果是到达真正的 OEP 处时,栈顶应该是 12FFC4 或者附近的地方,但是该壳把 OEP 处前几行的代码清空掉了,并填入了垃圾数据,并且把这前几行代码放到壳代码的空间中去!接着在壳空间执行原程序的前几条指令,然后再 JMP 到原程序的假 OEP 处。大家应该还记得该 crackme 的 OEP 是 4271B0,我们来看看该地址处是什么。

004271AC	90	NOP	
004271AD	90	NOP	
004271AE	90	NOP	
004271AF	90	NOP	
004271B0	DA6D B6	FISUBR DWORD PTR SS:[EBP-4A]	
004271B3	DB6D B6	FLD TBYTE PTR SS:[EBP-4A]	
004271B6	5B	POP EBX	
004271B7	2D 168BC5E2	SUB EAX,E2C58B16	
004271BC	71 B8	JNO SHORT 00427176	UnPackMe.00427176
004271BE	DC6E 37	FSUBR QWORD PTR DS:[ESI+37]	
004271C1	9B	WAIT	
004271C2	4D	DEC EBP	
004271C3	26:93	XCHG EAX,EBX	Superfluous prefix
004271C5	49	DEC ECX	
004271C6	A4	MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]	
004271C7	52	PUSH EDX	
004271C8	A9 542A95CA	TEST EAX,CA952A54	
004271CD	65:B2 59	MOV DL,59	Superfluous prefix
004271D0	AC	LODS BYTE PTR DS:[ESI]	
004271D1	D6	SALC	
004271D2	EB F5	JMP SHORT 004271C9	UnPackMe.004271C9
004271D4	7A BD	JPE SHORT 00427193	UnPackMe.00427193
004271D6	E9 8AD66500	JMP 00A84865	
004271D8	0033	ADD BYTE PTR DS:[EBX],DH	
004271DD	D28A D4891534	ROR BYTE PTR DS:[EDX+341589D4],CL	

这里我们可以看到,壳将 OEP 处的原始字节都填充为了垃圾指令,我们对该区段设置内存访问断点会断在 4271D6 处,不会断在前面。

有很多方法可能定位 stolen bytes,但最为经典,最为常用的方法还是在最后一次异常产生后单步跟踪,我们现在还是跟到最后一次异常处。

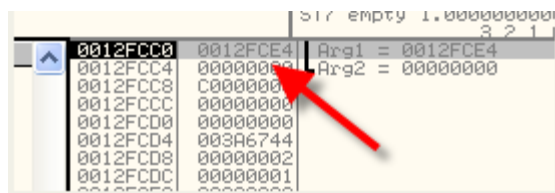
7C91EAE0	\$ 8B4C24 04	MOV ECX,DWORD PTR SS:[ESP+4]	
7C91EAF0	. 8B1C24	MOV EBX,DWORD PTR SS:[ESP]	
7C91EAF3	. 51	PUSH ECX	
7C91EAF4	. 53	PUSH EBX	
7C91EAF5	. E8 C78C0200	CALL 7C9477C1	ntdll.7C9477C1
7C91EAF6	. 0AC0	OR AL,AL	
7C91EAF7	. 74 0C	JE SHORT 7C91EB0A	ntdll.7C91EB0A
7C91EAF8	. 5B	POP EBX	
7C91EAF9	. 59	POP ECX	
7C91EAB0	. 6A 00	PUSH 0	
7C91EAB2	. 51	PUSH ECX	
7C91EAB3	. E8 11EBFFFF	CALL 7C91D619	ntdll.ZwContinue
7C91EAB8	. EB 0B	JMP SHORT 7C91EB15	ntdll.7C91EB15
7C91EAB9	. 5B	POP EBX	
7C91EABD	. 59	POP ECX	
7C91EABE	. 6A 00	PUSH 0	
7C91EABF	. 51	PUSH ECX	
7C91EABF	. 53	PUSH EBX	
7C91EB10	. E8 3DF7FFFF	CALL 7C91E252	ntdll.ZwRaiseException
7C91EB15	. 83C4 EC	ADD ESP,-14	
7C91EB18	. 890424	MOV DWORD PTR SS:[ESP],EAX	
7C91EB1B	. C74424 04 01	MOV DWORD PTR SS:[ESP+4],1	
7C91EB23	. 895C24 08	MOV DWORD PTR SS:[ESP+8],EBX	
7C91EB27	. C74424 10 00	MOV DWORD PTR SS:[ESP+10],0	
7C91EB2F	. 54	PUSH ESP	

我们知道最后一次异常是在 3A6744 地址处产生的,这里我们对异常处理程序并不感兴趣,我们直接在其下面 7C91EB03 地址处下断。

7C91EAE0	\$ 8B4C24 04	MOV ECX,DWORD PTR SS:[ESP+4]	
7C91EAF0	. 8B1C24	MOV EBX,DWORD PTR SS:[ESP]	
7C91EAF3	. 51	PUSH ECX	
7C91EAF4	. 53	PUSH EBX	
7C91EAF5	. E8 C78C0200	CALL 7C9477C1	ntdll.7C9477C1
7C91EAF6	. 0AC0	OR AL,AL	
7C91EAF7	. 74 0C	JE SHORT 7C91EB0A	ntdll.7C91EB0A
7C91EAF8	. 5B	POP EBX	
7C91EAF9	. 59	POP ECX	
7C91EAB0	. 6A 00	PUSH 0	
7C91EAB2	. 51	PUSH ECX	
7C91EAB3	. E8 11EBFFFF	CALL 7C91D619	ntdll.ZwContinue
7C91EAB8	. EB 0B	JMP SHORT 7C91EB15	ntdll.7C91EB15
7C91EAB9	. 5B	POP EBX	
7C91EABD	. 59	POP ECX	
7C91EABE	. 6A 00	PUSH 0	

运行起来,接下来程序将返回到哪里呢?我们要对哪里下断?一起来看看堆栈。





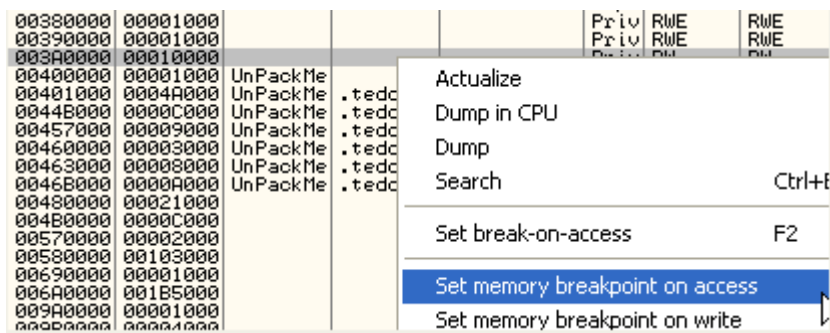
这里第一个参数是 CONTEXT 结构体的指针,我们在数据窗口中定位到该地址。

Address	Hex dump	ASCII
0012FCE4	3F 00 01 00 00 00 00 00	? . 0 . . . . .
0012FCEC	00 00 00 00 00 00 00 00	. . . . .
0012FCF4	00 00 00 00 00 00 00 00	. . . . .
0012FCFC	00 00 00 00 7F 02 FF FF	. . . . Δ 0
0012FD04	20 40 FF FF FF FF FF FF	@ . . . . .
0012FD0C	4D 29 B2 67 1B 00 5E 05	M) 29 g + . ^ 5
0012FD14	28 28 5F 05 23 00 FF FF	(( _ # . . .
0012FD1C	00 00 00 00 04 01 05 01	. . . . 0 # 0
0012FD24	E0 BC 00 00 00 00 00 00	0 . . . . .
0012FD2C	00 00 00 00 00 00 00 00	. . . . .
0012FD34	00 00 00 00 00 00 00 00	. . . . .
0012FD3C	00 00 00 00 00 00 00 00	. . . . .
0012FD44	00 00 00 00 00 00 00 00	. . . . .
0012FD4C	00 00 00 00 00 00 00 00	. . . . .
0012FD54	00 00 00 00 00 00 00 00	. . . . .
0012FD5C	00 00 00 80 FF 3F 00 00	. . . C ? . .
0012FD64	00 00 00 00 00 80 FF 3F	. . . . C ?
0012FD6C	00 00 00 00 00 00 00 00	. . . . .
0012FD74	3B 00 00 00 23 00 00 00	; . . # . . .
0012FD7C	23 00 00 00 1E A3 6E CC	# . . Δ u n f
0012FD84	00 00 00 00 26 54 3A 0D	. . . & T : .
0012FD8C	24 00 00 00 00 00 00 00	\$ . . . . .
0012FD94	00 00 00 00 2D 06 3A 00	. . . - + : .
0012FD9C	46 67 3A 00 1B 00 00 00	. . . . .
0012FDA4	46 02 01 00 B0 12 00 00	F 0 0 . . . 0
0012FDAC	23 00 00 00 7F 02 20 40	# . . Δ 0 @
0012FDB4	00 00 5E 05 4D 29 B2 67	. . ^ # M) 29 g
0012FDBC	00 00 00 00 00 00 00 00	. . . . .
0012FDC4	00 00 FF FF 80 1F 00 00	. . C . .
0012FDCC	00 00 00 00 00 00 00 00	. . . . .
0012FDD4	04 01 05 01 E0 BC 00 00	0 # 0 0 . .
0012FDDC	00 00 00 00 00 00 00 00	. . . . .
0012FDE4	00 00 00 00 00 00 00 00	. . . . .

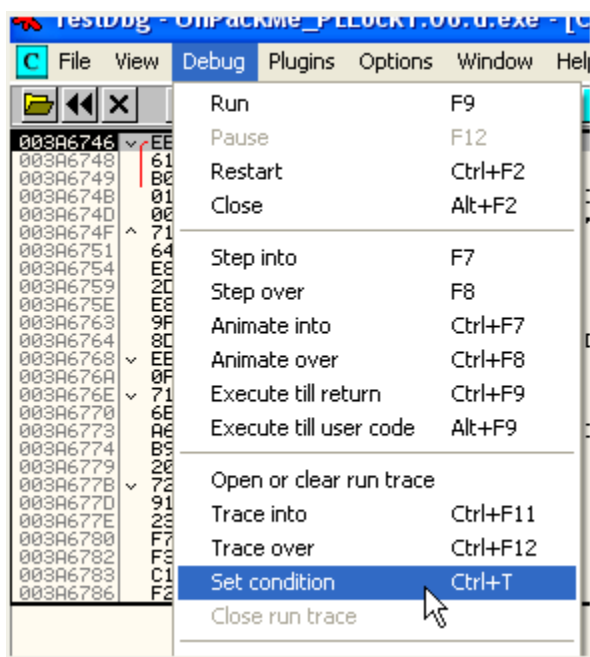
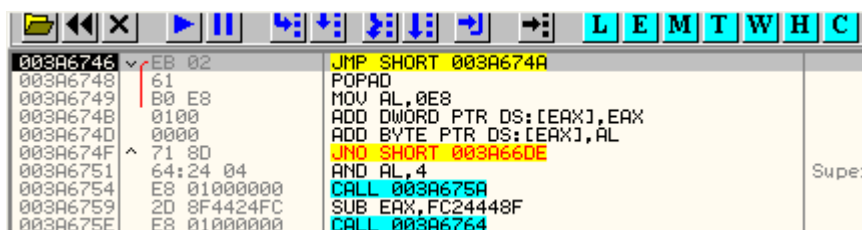
这里如果大家熟悉 CONTEXT 结构的话,就会知道其中有个字段标识的是 EIP 寄存器,其他一些字段标识的是其他一些寄存器,当异常处理完毕以后会根据 CONTEXT 结构中的 EIP 值来决定返回到哪里,后面章节我们会详细这个 CONTEXT 结构,现在我们只需要知道程序返回到哪里,EIP 字段位于偏移 B8 处(PS:可以参考 VC 中 CONTEXT 结构体的定位)。

```
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; // +0xB8
    DWORD SegCs;
    DWORD EFlags;
    DWORD Esp;
    DWORD SegSs;
    BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT;
```

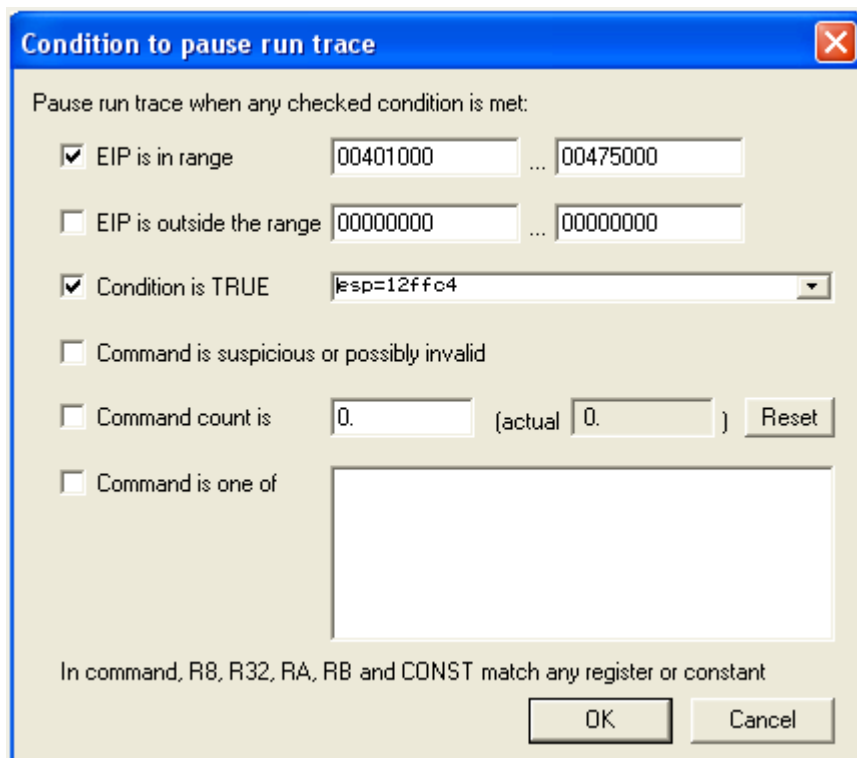
因此,程序将返回到 3A6746 地址处。我们给该地址所在区段设置一个内存访问断点。



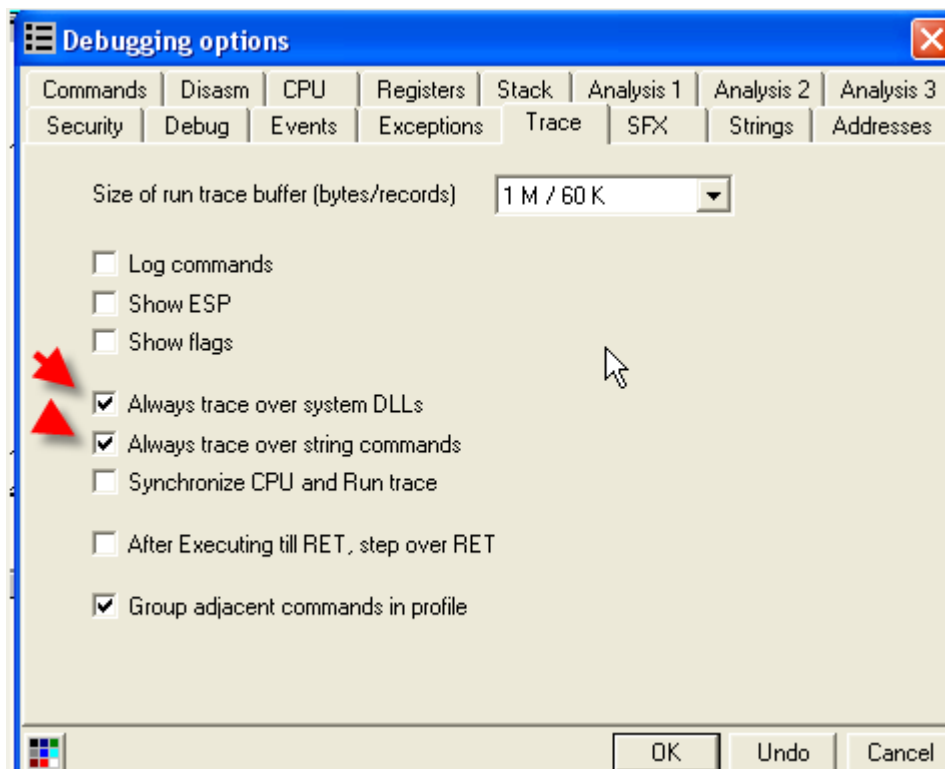
运行起来,我们可以看到断在了这里。



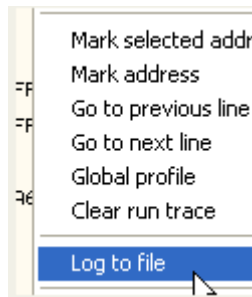
接下来我们来设置自动跟踪的条件,有好几个选项可供设置,比如说我们可以定位恢复寄存器环境指令,当恢复寄存器环境后,接下来就要执行 OEP 处的代码了,当前还可以设置其他条件,比如说,我们设置如下两项。



EIP is Range 这一项我们设置为 401000~475000,即原程序的所有区段,这样我们就能够在 Run Trace 窗口中看到跟踪过程中执行了哪些指令,下面 Condition is TRUE 这一项我们设置 ESP = 12FFC4,即跟踪到栈顶指针跟假的 OEP 一致时,停止跟踪。

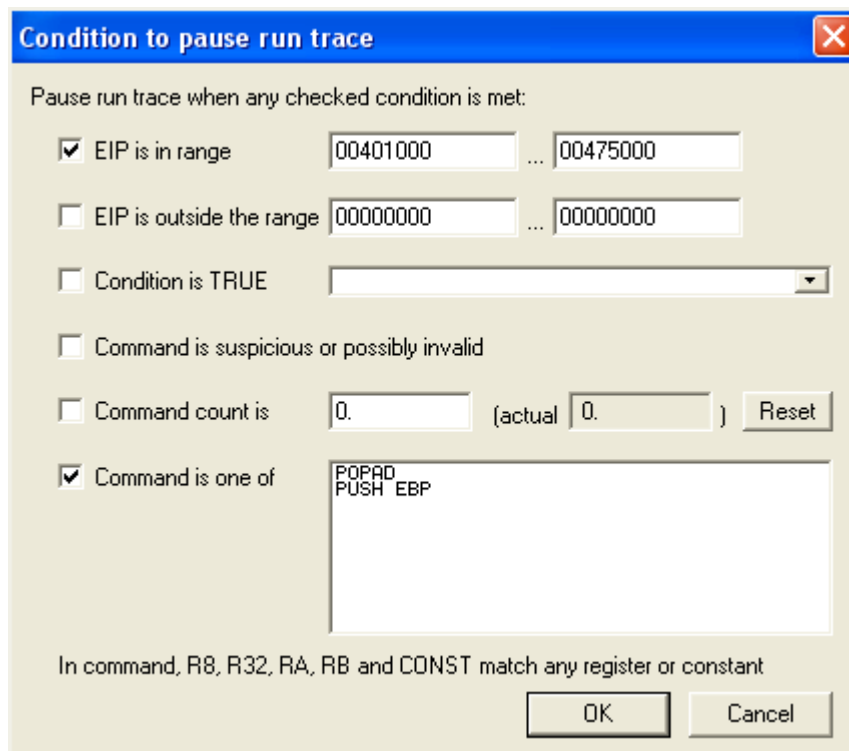


注意,这里调试选项中的 Trace 选项卡中的这两项我们也可以尝试勾选上,看看自动跟踪的效果如何,如果效果不好,我们再来去掉这两个对勾。

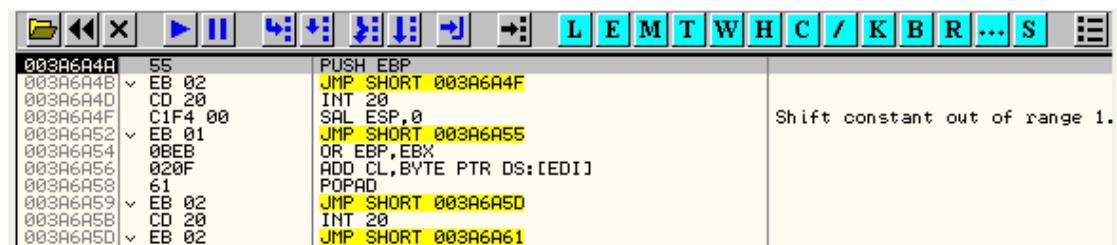


跟踪完毕以后,我们可以单击鼠标右键选择-Log to file(将日志信息保存到文件),再来进一步分析。

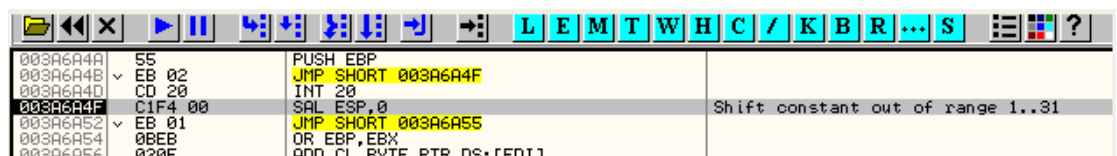
我们选择主菜单中的 Debug-Trace into,断了下来,但是并没有到达假的 OEP 处,我们继续 Trace into 多次以后才会到达假 OEP 处,Run Trace 窗口中记录了自动跟踪所执行的指令,我们依然看出来哪里是真正的 OEP 所在。



既然不奏效,那我们来换个条件试试,我们设置 Command is one of 这一项,当自动跟踪过程中遇到设置的命令序列中的任意一个就会停止跟踪,这里,我们填入两条命令,POPAD 和 PUSH EBP,即当遇到恢复寄存器环境或者 OEP 处常见的第一条指令时停止自动跟踪,我们来看看能不能奏效,单击主菜单中的 Debug-Trace into。



这里有可能是真正的 OEP,我们往下跟。



这里很明显是垃圾指令,紧接着下面是一个无条件跳转,没有实际作用,相当于花指令。继续往下跟。

003A6A6B	EB 02	JMP SHORT 003A6A6F
003A6A6D	65:A7	CMPD DWORD PTR GS:[ESI],DWORD PTR ES:[ESI]
003A6A6F	8BEC	MOV EBP,ESP
003A6A71	EB 01	JMP SHORT 003A6A74
003A6A73	0FC1F2	XADD EDX,ESI
003A6A76	00EB	ADD BL,CH

这里应该是缺失的第二条指令。

003A6A8E	CD 20	INT 20
003A6A90	C1F4 00	SAL ESP,0
003A6A93	6A FF	PUSH -1
003A6A95	C1F4 00	SAL ECX,0
003A6A98	EB 02	JMP SHORT 003A6A9B

我们继续往下跟,这里应该是一条正常的指令。

003A6AB5	EB 02	JMP SHORT 003A6AB9
003A6AB7	CD 20	INT 20
003A6AB9	68 600E4500	PUSH 450E60
003A6ABE	EB 02	JMP SHORT 003A6AC2
003A6AC0	04 72	ADD AL,72
003A6AC2	EB 01	JMP SHORT 003A6AC5
003A6AE0	EB 01	JMP SHORT 003A6AE3
003A6AE2	40	INC EAX
003A6AE3	68 C8924200	PUSH 4292C8
003A6AE8	EB 02	JMP SHORT 003A6AEC
003A6AEA	BB FDEB02CD	MOV EBX,CD02EBFD
003A6AEF	20EB	AND BL,CH

003A6B07	EB 02	JMP SHORT 003A6B0D
003A6B08	CD 20	INT 20
003A6B0D	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
003A6B13	EB 02	JMP SHORT 003A6B17
003A6B15	CD 20	INT 20

003A6B35	50	PUSH EAX
003A6B36	C1F7 00	SAL EDI,0
003A6B39	EB 02	JMP SHORT 003A6B3D
003A6B3B	CD 20	INT 20

003A6B55	EB 02	JMP SHORT 003A6B59
003A6B57	CD 20	INT 20
003A6B59	64:8925 00000000	MOV DWORD PTR FS:[0],ESP
003A6B60	C1F5 00	SAL EBP,0
003A6B63	C1F3 00	SAL EBX,0
003A6B66	EB 02	JMP SHORT 003A6B6A

003A6B7E	EB 02	JMP SHORT 003A6B82
003A6B80	CD 20	INT 20
003A6B82	C1F0 00	SAL EAX,0
003A6B85	83C4 A0	ADD ESP,-58
003A6B88	C1F6 00	SAL ESI,0
003A6B8B	EB 01	JMP SHORT 003A6B8E
003A6B8D	EB 01	POW MM0,QWORD PTR DS:[ECX]
003A6B90	00000000	POW MM0,QWORD PTR DS:[ECX]

003A6BA8	EB 02	JMP SHORT 003A6BAC
003A6BA9	CD 20	INT 20
003A6BAC	53	PUSH EBX
003A6BAD	C1F1 00	SAL ECX,0
003A6BB0	EB 02	JMP SHORT 003A6BB4
003A6BB2	3C 0B	CMP AL,0B
003A6BB4	C1F6 00	SAL ESI,0

003A6BCF	56	PUSH ESI
003A6BD0	EB 02	JMP SHORT 003A6BD4
003A6BD2	CD 20	INT 20
003A6BD4	EB 02	JMP SHORT 003A6BD8
003A6BD6	CD 20	INT 20

003A6BF4	CD 20	INT 20
003A6BF6	57	PUSH EDI
003A6BF7	EB 02	JMP SHORT 003A6BFB
003A6BF9	CD 20	INT 20
003A6BFB	FR 01	JMP SHORT 003A6BFF

003A6C65	CD 20	INT 20
003A6C67	68 0C714200	PUSH 4271D6
003A6C6C	EB 02	JMP SHORT 003A6C70
003A6C6E	C7 02EB	ADD CH, BL
003A6C71	0265 A0	ADD AH, BYTE PTR SS:[EBP-60]
003A6C74	EB 02	JMP SHORT 003A6C70

接下来我们就会到达假的 OEP 4271D6 处,因为这里是 PUSH 4271D6,接着通过 RET 就可以返回到假的 OEP 处。

003A6C93	C3	RETN
003A6C94	EB 02	JMP SHORT 003A6C98
003A6C96	0F50C1	MOVMSKPS EAX, XMM1
003A6C99	F4	HLT

004271D6	E9 8AD66500	JMP 00A84865
004271D8	0033	ADD BYTE PTR DS:[EBX], DH
004271DD	D28A D4891534	ROR BYTE PTR DS:[EDX+341589D4], CL
004271E3	E6 45	OUT 45, AL
004271E5	008B C881E1FF	ADD BYTE PTR DS:[EBX+FFE181C8], CL
004271EB	0000	ADD BYTE PTR DS:[EAX], AL
004271ED	0089 0D30E645	ADD BYTE PTR DS:[ECX+45E6300D], CL
004271F3	00C1	ADD CL, AL

接下来我们可以将 stolen bytes 拷贝出来,我们首先在数据窗口中看看假 OEP 之前的内存单元的情况。

Address	Hex dump	ASCII
004271AE	90 90 DA 6D B6 DB 6D B6	ÉÉrmÄmÄ
004271B6	5B 2D 16 8B C5 E2 71 B8	[_ i+0q0
004271BE	DC 6E 37 9B 4D 26 93 49	h7m%oI
004271C6	A4 52 A9 54 2A 95 CA 65	h0T*o%e
004271CE	B2 59 AC D6 EB F5 7A BD	W%iuSzç
004271D6	E9 8A D6 65 00 00 33 D2	0eie..3E
004271DE	8A D4 89 15 34 E6 45 00	éeéS4pE.
004271E6	8B C8 81 E1 FF 00 00 00	iüß ...
004271EE	89 0D 30 E6 45 00 C1 E1	é..0pE..+ß
004271F6	08 03 CA 89 0D 2C E6 45	0%é..pE
004271FE	00 C1 E8 10 A3 28 E6 45	..+ßü(pE
00427206	00 E8 94 21 00 00 85 C0	.pö!..äL
0042720E	75 0A 6A 1C E8 49 01 00	u..JLßIO.
00427216	00 83 C4 04 F8 D1 2F 00	z..äR0/

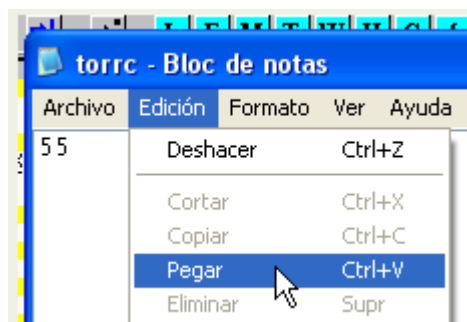
我们需要将 stolen bytes 填充到假 OEP 前面的内存单元中,好,现在我们按减号键返回到刚刚真正 OEP 的第一条指令处。

003A6A4A	55	PUSH EBP
003A6A4B	EB 02	JMP SHORT 003A6A4F
003A6A4D	CD 20	INT 20
003A6A4F	C1F4 00	SAL ESP, 0
003A6A52	EB 01	JMP SHORT 003A6A55
003A6A54	0BE8	OR EBP, EBX
003A6A56	020F	ADD CL, BYTE PTR DS:[EDI]
003A6A58	61	POPAD
003A6A59	EB 02	JMP SHORT 003A6A5D
003A6A5B	CD 20	INT 20
003A6A5D	EB 02	JMP SHORT 003A6A61
003A6A5F	CD 20	INT 20
003A6A61	EB 02	JMP SHORT 003A6A65
003A6A63	CD 20	INT 20
003A6A65	EB 01	JMP SHORT 003A6A68
003A6A67	00FB	FICMP ST(3)

我们依次将缺失的指令所对应的字节码拷贝到记事本中,最后一个 PUSH 指令和 RET 指令不需要拷贝,这两条指令时用来跳转到假 OEP 处的,并不是 stolen bytes。

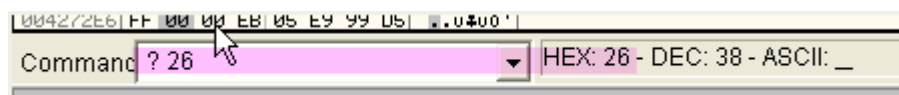
003A6A4A	55	PUSH EBP
003A6A4B	EB 02	JMP SHORT 003A6A4F
003A6A4D	CD 20	INT 20
003A6A4F	C1F4 00	SAL ESP, 0
003A6A52	EB 01	JMP SHORT 003A6A55
003A6A54	0BE8	OR EBP, EBX
003A6A56	020F	ADD CL, BYTE PTR DS:[EDI]
003A6A58	61	POPAD
003A6A59	EB 02	JMP SHORT 003A6A5D
003A6A5B	CD 20	INT 20
003A6A5D	EB 02	JMP SHORT 003A6A61
003A6A5F	CD 20	INT 20
003A6A61	EB 02	JMP SHORT 003A6A65
003A6A63	CD 20	INT 20
003A6A65	EB 01	JMP SHORT 003A6A68
003A6A67	00FB	FICMP ST(3)





55 8B EC 6A FF 68 60 0E 45 00 68 C8 92 42 00 64 A1 00 00 00 00 50 64 89 25 00 00 00 00 83 C4 A8 53 56 57 89 65 E8

以上是 38 个 stolen bytes,也即是 16 进制的 26。



所以应该把假 OEP 往下抬高 26(十六进制)个字节。



即如果壳没有抽取 OEP 处的代码的话,真正的 OEP 应该是 4271B0,我们需要将被抽取的代码填充到 4271B0 处。

Address	Hex dump	ASCII
004271B0	DA 6D B6 DB 6D B6 5B 2D	rmA...mAl-
004271B8	16 8B C5 E2 71 B8 DC 6E	..i+0qB..n
004271C0	37 9B 4D 26 93 49 A4 52	7xM&.oIR
004271C8	A9 54 2A 95 CA 65 B2 59	@T*o"eY
004271D0	AC D6 EB F5 7A B0 E9 8A	%iU&zcuë
004271D8	D6 65 00 00 33 D2 8A D4	ie..3EëE
004271E0	89 15 34 E6 45 00 8B C8	ë&4pE.i <sup>u</sup>
004271E8	81 E1 FF 00 00 00 89 0D	ûB...ë.
004271F0	30 E6 45 00 C1 E1 08 03	0pE.+p00
004271F8	CA 89 0D 2C E6 45 00 C1	"ë...pE.+
00427200	E8 10 A3 28 E6 45 00 E8	p>û(pE.p
00427208	94 21 00 00 85 C0 75 0A	ô†...âLu.

全选记事本中的字节拷贝出来,并在 4271B0 为起始地址,长度为 26(16 进制)的区域上面单击鼠标右键选择 Binary paste(二进制粘贴)。

Address	Hex dump	ASCII
004271B0	55 8B EC 6A FF 68 60 0E	Uij h'A
004271B8	45 00 68 C8 92 42 00 64	E.h <sup>u</sup> EB.d
004271C0	A1 00 00 00 00 50 64 89	i....Pdë
004271C8	25 00 00 00 00 83 C4 A8	%....â-è
004271D0	53 56 57 89 65 E8 E9 8A	SUWëëpüë
004271D8	D6 65 00 00 33 D2 8A D4	ie..3EëE
004271E0	89 15 34 E6 45 00 8B C8	ë&4pE.i <sup>u</sup>
004271E8	81 E1 FF 00 00 00 89 0D	ûB...ë.
004271F0	30 E6 45 00 C1 E1 08 03	0pE.+p00
004271F8	CA 89 0D 2C E6 45 00 C1	"ë...pE.+
00427200	E8 10 A3 28 E6 45 00 E8	p>û(pE.p
00427208	94 21 00 00 85 C0 75 0A	ô†...âLu.
00427210	6A 1C E8 49 01 00 00 83	jLp10...â
00427218	C4 04 E8 D1 2F 00 00 85	-♦p0/..â

我们来看看拷贝的指令。

L E M T W H C / K B F			
004271B0	55	PUSH EBP	
004271B1	8BEC	MOV EBP,ESP	
004271B3	6A FF	PUSH -1	
004271B5	68 600E4500	PUSH 450E60	
004271B8	68 C8924200	PUSH 4292C8	
004271BF	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]	
004271C5	50	PUSH EAX	
004271C6	64:8925 000000	MOV DWORD PTR FS:[0],ESP	
004271CD	83C4 A8	ADD ESP,-58	
004271D0	53	PUSH EBX	
004271D1	56	PUSH ESI	
004271D2	57	PUSH EDI	
004271D3	8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
004271D6	- E9 8AD66500	JMP 00A84865	
004271D8	0033	ADD BYTE PTR DS:[EBX],DH	
004271DD	D28A D4891534	ROR BYTE PTR DS:[EDX+341589D4],CL	
004271E3	E6 45	OUT 45,AL	I/O command
004271E5	008B C881E1FF	ADD BYTE PTR DS:[EBX+FFE181C8],CL	
004271EB	0000	ADD BYTE PTR DS:[EAX],AL	
004271ED	0000 0000E64F	ADD BYTE PTR DS:[ECX+4FE60000],CL	

好了,stolen bytes 被找回来了,下面我们将 OEP 修改为 4271B0,这样我们就解决了 stolen bytes 的问题。

下一章节我们来讨论如何编写脚本修改该程序的 IAT。