

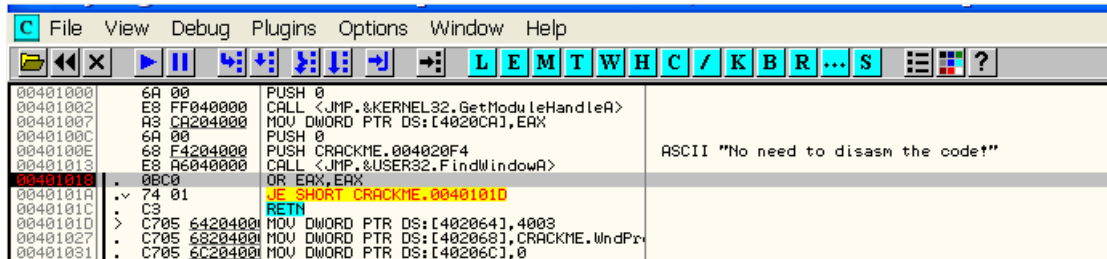
第十章-断点

本章将介绍各种类型的断点。断点可以让你在程序代码执行到合适的时候暂停下来。这次我们的实验的对象还是 CrueHead'a 的 CrackMe。

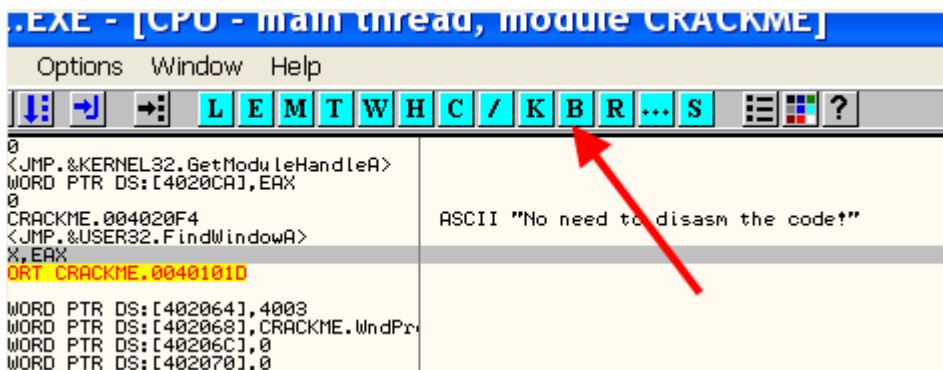
普通断点

这是一个很普通的断点,我们前面章节已经使用了。在 SoftICE 中我们可以使用 BPX 命令来设置断点。OD 中可以使用 BP 命令或者 F2 快捷键来设置断点。也可以再按一次 F2 快捷键来取消断点。

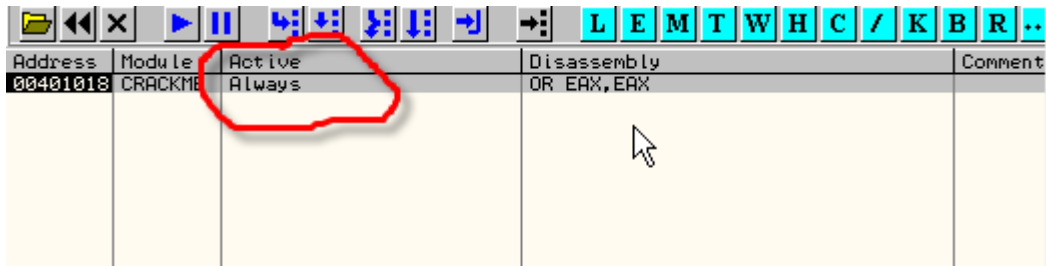
我们来到 CrueHead'a 的 CrackMe 的入口点处。



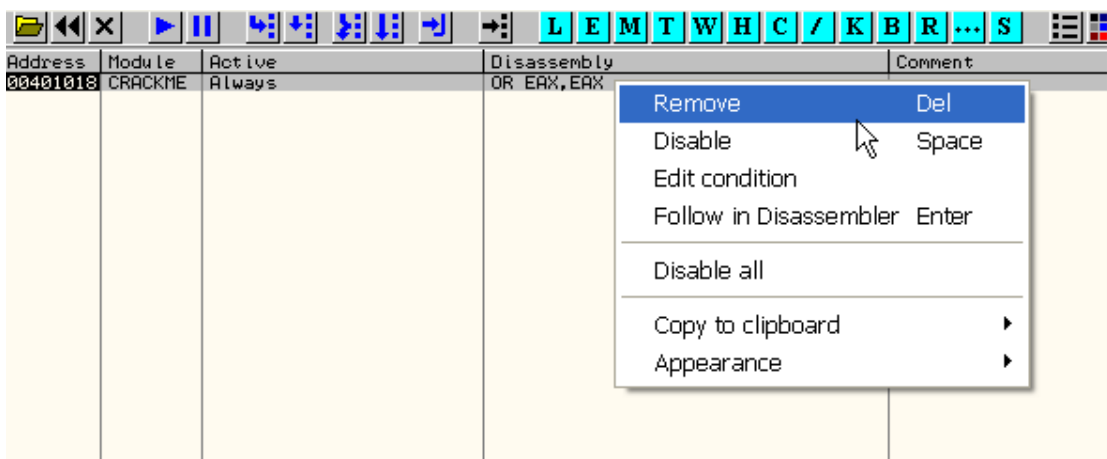
举个例子,拿 401018 这行来说吧,按 F2 键-这行就会以红色突出显示,在该地址处设置的断点就会加入了断点列表中。



我们来看看断点列表刚刚设置的断点,此时该断点是激活状态。



断点列表中 Active 栏显示的是 Always。在该行上单击鼠标右键会弹出一些操作断点的菜单项。



Remove:从列表中删除断点。

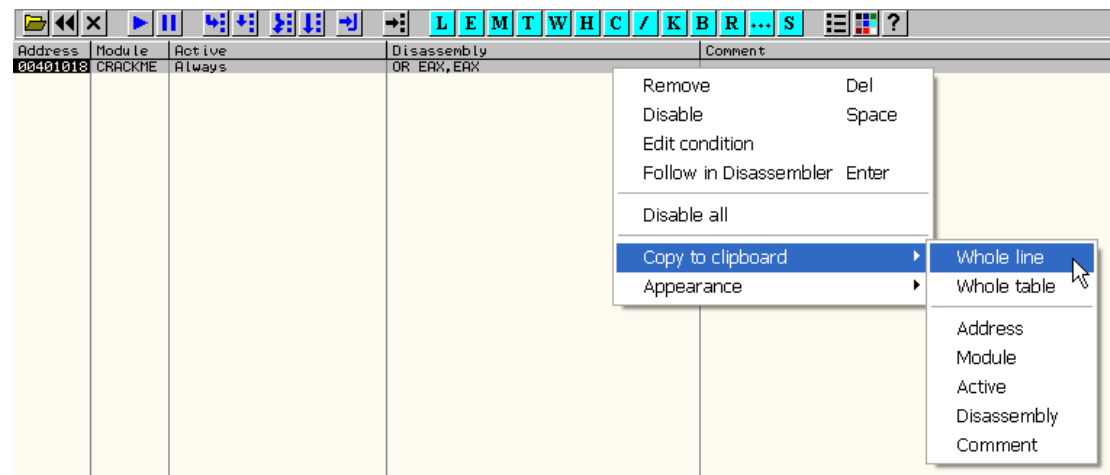
Disable:禁用断点但并不将断点从列表中删除。禁用时,断点并不会触发。

Edit condition:给断点设置触发条件,我们后面再来讨论。

Follow in disassembler:在反汇编窗口中显示断点。

Disable all or enable all:禁用/启用列表中的全部断点。这里没有启用的选项,因为列表中唯一的断点没有被禁用。

Copy to Clipboard:把选中断点的信息复制到剪贴板。我们来实验一下。



我们选择 Whole line 拷贝整行,Whole Table 可以拷贝整个列表的断点信息。

Breakpoints, item 0

Address=00401018

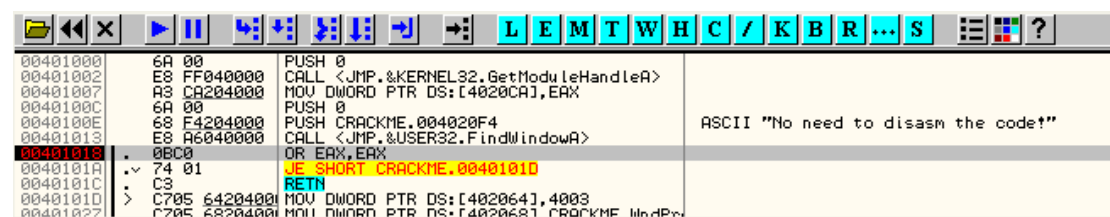
Module=CRACKME

Active=Always

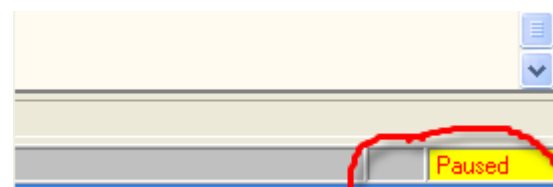
Disassembly=OR EAX,EAX

拷贝下来的信息显示了断点的地址,对应的指令以及激活状态。

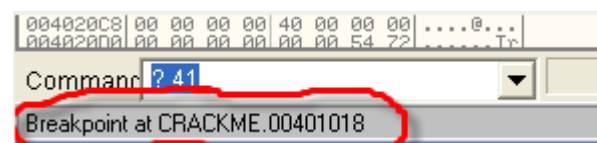
按下 F9 键-CrackMe 运行起来了。然后正如猜想的一样中断了下来。



在状态栏显示暂停状态。

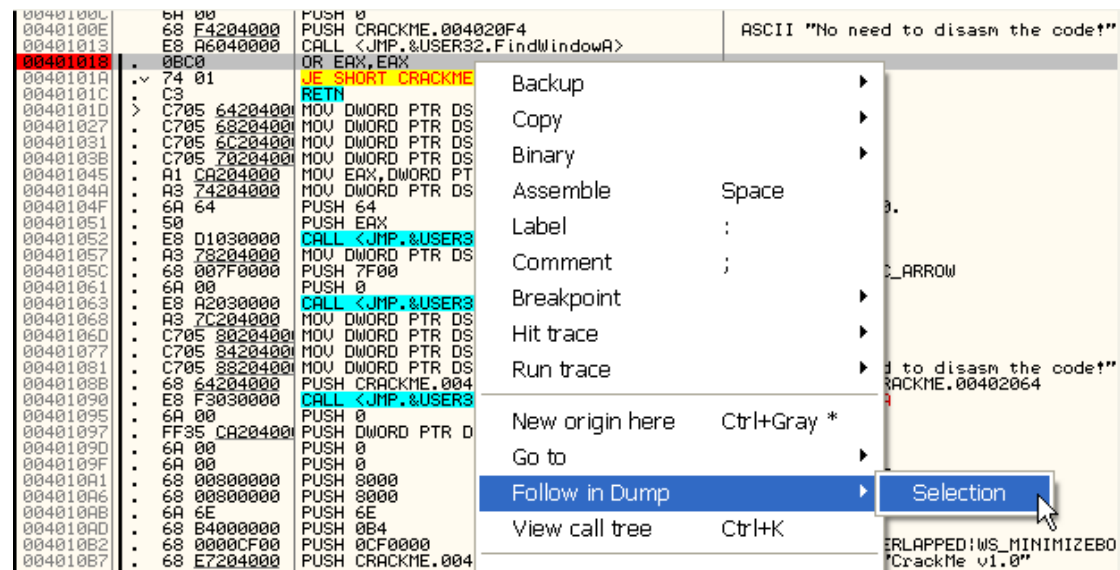


暂停的原因如下:



我们来了解一下当设置一个断点以后,二进制代码会发生什么变化。

单击鼠标右键选择-Follow in Dump-Selection



我们看看数据窗口中 401018 地址处的内容:

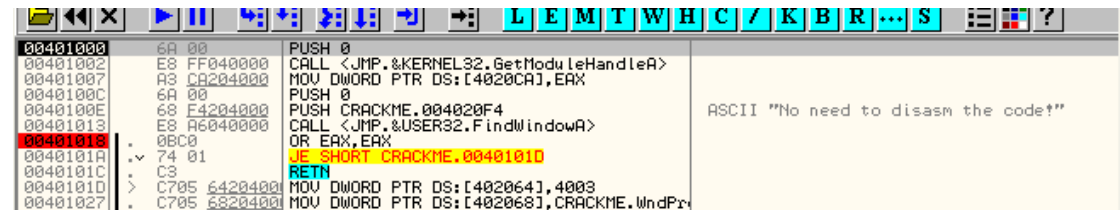
Address	Hex dump	ASCII
00401018	0B C0 74 01 C3 C7 05 64	0t0tA&d
00401020	20 40 00 03 40 00 00 C7	@.00..A
00401028	05 68 20 40 00 28 11 40	#h @.(0
00401030	00 C7 05 6C 20 40 00 00	.A!l @.
00401038	00 00 00 C7 05 70 20 40	...A+p @
00401040	00 00 00 00 00 A1 CA 20i
00401048	40 00 A3 74 20 40 00 6A	@.0t @.j
00401050	64 50 E8 D1 03 00 00 A3	dPb00..u

我们初看一下数据窗口中的内容和反汇编代码中代码是一样的:

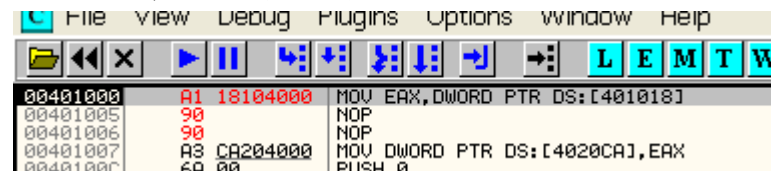


数据窗口和反汇编代码中我们看到的都是 0B C0,对应的是 OR EAX,EAX。似乎代码没有什么变化,但是真的没有变化吗?

我们保留 401018 处的断点,重新加载 CrackMe。



我们将 OR EAX,EAX 对应的机器代码读取出来然后写到别处。



该指令将 401018 地址处的双字值保存到 EAX 中,我们看看 OD 的提示框中提示的信息。

004010CF	FF35 04204001	PUSH DWORD PTR DS:[402004]
004010D5	FF35 00030000	CALL <JMP.&USER32.ShowWindow
DS:[00401018] 0174C00B		
EAX=0174C0CC		
Address	Hex dump	ASCII
00401018	0B C0 74 01 C3 C7 05 64	0174C00B
00401020	00 00 00 00 00 00 00 00	

在数据窗口中和提示框中显示的都是相同的内容:0B C0。但是,我们按下 F7 键看看 EAX 显示的内容。

Registers (FPU)	
EAX	0174C0CC
ECX	0012FFB0
EDX	7C91EB94 ntdll.
EBX	7FFD0000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll.
EIP	004010D5 CRACKM
C 0	ES 0023 32bit
P 1	CS 001B 32bit

读出来的 401018 处的内容并不是 OD 刚刚显示的 0B C0 74 01(小端存储),按双字节取出来是 0174C00B,而现在显示的是 0174C0CC,因此 401018 处字节值是 CC。当我们设置断点后,OD 会将对应指令处第一个字节指令替换成 CC。但是为了不影响界面显示效果,OD 会将 CC 显示为原字节。但是,我们可以在内存单元中读取出其真实的内容,并且可以在反调试中用此方法来检测断点。所以,我们设置的断点有时候莫名其妙的消失了不要感到奇怪,或许说这是调试器的本身的弱点吧。

除了 F2 设置断点以外,我们还可以通过命令栏来设置断点,如下:

BP 401018

Command	BP 401018
---------	-----------

在 NT(2000,XP 和 2003)系统中我们也可以很容易的给 API 函数设置断点-我们前面章节中已经介绍过了。要给 MessageBoxA 设置断点,请输入:

Command	BP MessageBoxA
---------	----------------

并且你必须指定 API 函数的确切名称,而且大小写敏感。

Command	BPX MessageBoxA
---------	-----------------

还有一个比 BP 更加强大的命令 BPX 可以给引用或者调用了指定 API 函数的指令都下断点。

下面是 BPX 给 MessageBoxA 设置的断点列表。正如你所看到的,OD 找到了 3 处地方调用 MessageBoxA,并设置了 3 个断点。

Address	Module	Active	Disassembly	Comme
00401018	CRACKME	Always	OR EAX,EAX	
0040135C	CRACKME	Always	CALL <JMP.&USER32.MessageBoxA>	
00401378	CRACKME	Always	CALL <JMP.&USER32.MessageBoxA>	
004013BC	CRACKME	Always	CALL <JMP.&USER32.MessageBoxA>	
77D504EA	USER32	Always	MOV EDI,EDI	

还有一种设置断点的方法:在反汇编窗口中你想设置断点的那一行双击机器码即可。如果想删除的话,再双击一次即可。

00401000	6A 00	PUSH 0	
00401002	E8 FF040000	CALL <JMP.&KERNEL32.GetModuleHandleA>	
00401007	A3 C0204000	MOV DWORD PTR DS:[4020CA],EAX	
0040100C	6A 00	PUSH 0	
0040100E	68 F4204000	PUSH CRACKME.004020F4	ASCII "No need to disasm the code!"
00401013	E8 A6040000	CALL <JMP.&USER32.FindWindowA>	
00401018	0BC0 74 01	JE SHORT CRACKME.0040101D	
0040101A	C3	RETN	
0040101C	>		
0040101D	C705 64204000	MOV DWORD PTR DS:[402064],4003	
00401027	C705 68204000	MOV DWORD PTR DS:[402068],CRACKME.WndProc	
00401031	C705 6C204000	MOV DWORD PTR DS:[40206C],0	
0040103B	C705 70204000	MOV DWORD PTR DS:[402070],0	
00401045	A1 C0204000	MOV EAX,DWORD PTR DS:[4020CA]	
0040104A	A3 74204000	MOV DWORD PTR DS:[402074],EAX	
0040104F	6A 64	PUSH 64	
00401051	50	PUSH EAX	

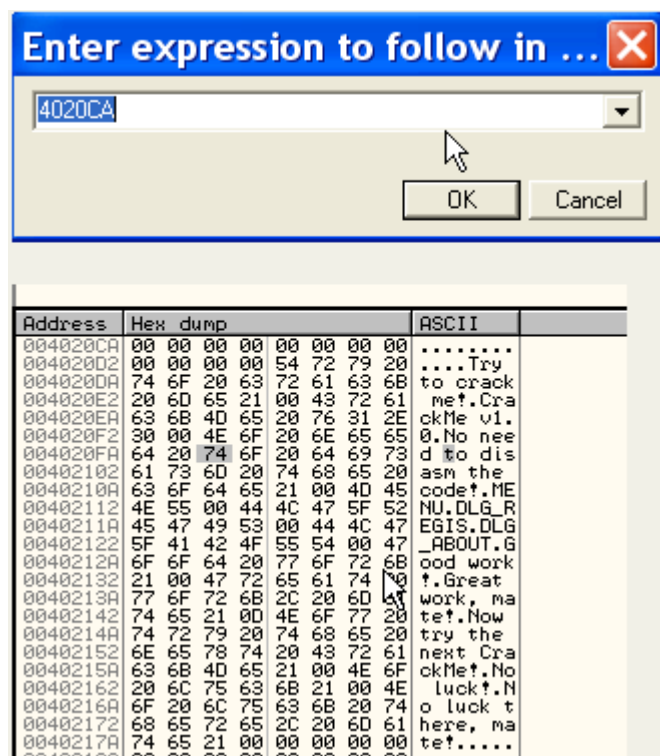
内存断点

内存访问断点有时候也称之为 BPM,但是不要与 SoftICE 中的 BPM 弄混淆了,这二者是完全不同的。

这种类型的断点修改内存页的访问属性。当前我们设置了内存断点。任何代码访问(读,写或者执行代码)了该处代码的话,都会触发异常。我们来看一个例子:

00401000	6A 00	PUSH 0	
00401002	E8 FF040000	CALL <JMP.&KERNEL32.GetModuleHandleA>	
00401007	A3 C0204000	MOV DWORD PTR DS:[4020CA],EAX	
0040100C	6A 00	PUSH 0	
0040100E	68 F4204000	PUSH CRACKME.004020F4	ASCII "No need to disasm the code!"
00401013	E8 A6040000	CALL <JMP.&USER32.FindWindowA>	
00401018	0BC0 74 01	JE SHORT CRACKME.0040101D	
0040101A	C3	RETN	
0040101C	>		
0040101D	C705 64204000	MOV DWORD PTR DS:[402064],4003	

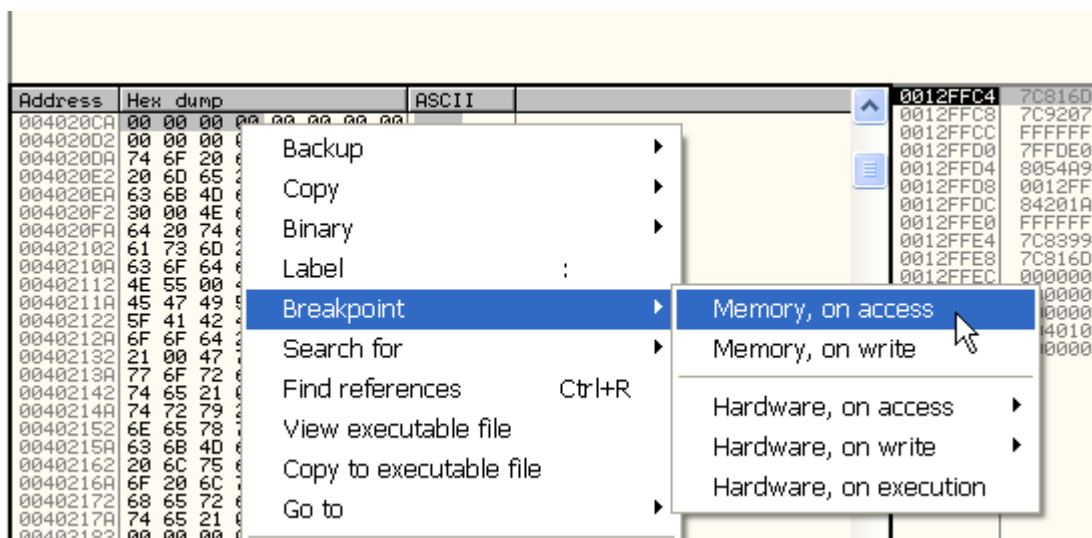
现在我们在 CrackMe 的入口点处,我们尝试设置一个内存断点。单击鼠标右键选择-Goto-Expression 输入 4020CA,转到这个地址。



我们在 4020CA 处设置 4 个字节的内存断点。当前有指令尝试读取这几个字节的时候,就会中断下来。

Address	Hex dump	ASCII
004020CA	00 00 00 00 00 00 00 00
004020D2	00 00 00 00 54 72 79 20Try
004020DA	74 6F 20 63 72 61 63 6B	to crack
004020E2	20 6D 65 21 00 43 72 61	me!.Cra
004020EA	63 68 4D 65 20 76 31 2E	ckMe v1.
004020F2	30 00 4E 6F 20 6E 65 65	0.No nee
004020FA	64 20 74 6F 20 64 69 73	d to dis
00402102	61 73 6D 20 74 68 65 20	sem the

我们这 4 字节上单击鼠标右键选择-Breakpoint-Memory,on access,这里不一定要设置的 4 个字节,你也可以设置长一点,也可以设置短一点。

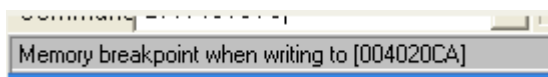


内存访问断点有两个缺点:1.它们不会出现[B]断点列表中和其他的地址。所以,你必须记得设置在什么地址处。2.不能同时设置多个内存断点。如果你设置了一个那么你之前设置的就会被自动删除。

当运行到 401007 地址处的时候,该地址处指令试图写入内容到 4020CA 内存单元中。

00401000	6A 00	PUSH 0	
00401002	E8 FF040000	CALL <JMP.&KERNEL32.GetModuleHandleA>	
00401007	A3 C8204000	MOV DWORD PTR DS:[4020CA],EAX	CRACKME.00400000
0040100C	6A 00	PUSH 0	
0040100E	68 F4204000	PUSH CRACKME.004020F4	
00401013	E8 A6040000	CALL <JMP.&USER32.FindWindowA>	ASCII "No need to disasm the code!"

下面状态栏清楚了描述了暂停的原因:

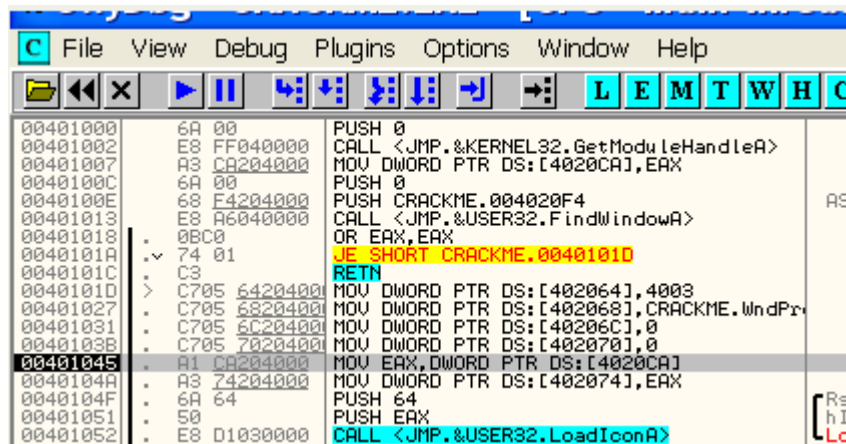


当指令尝试将 EAX 的值写入 4020CA 内存单元的时候,OD 会断下来。记住,当我们对指定内存单元没有写权限,尝试写入的时候会触发异常,OD 会拦截到这个异常,并中断下来,我们看到断下来的时候,OD 已经将内存页的访问属性设置正常了。

如果此时我们按 F7 键,EAX 的值会被写入到 4020CA 内存单元中,此处的异常不会再次发生。

Address	Hex dump	ASCII
004020CA	00 00 40 00 00 00 00 00	..@....
004020D2	00 00 00 00 54 72 79 20Try
004020DA	74 6F 20 63 72 61 63 6B	to crack
004020E2	20 6D 65 21 00 43 72 61	me!.Cra

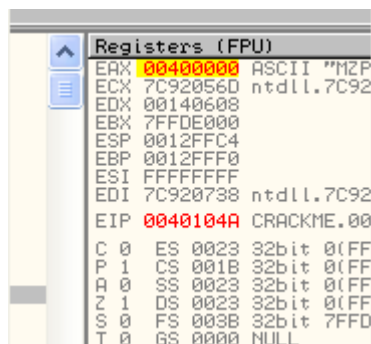
400000h 的值成功写入。我们运行起来,由于内存访问断点仍然存在,如果程序尝试访问 4020CA 内存单元的话,会再次触发异常。



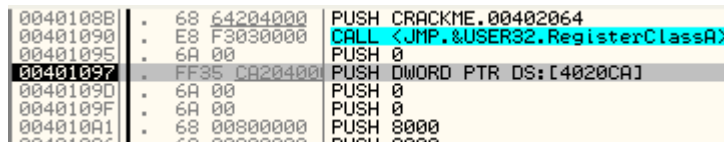
正如你看到的,该指令尝试读取 4020CA 内存单元的内容,证明内存访问断点又触发了。

Memory breakpoint when reading [004020CA]

再次按 F7 键运行一步,将 4020CA 内存单元内容读取出来并保存到 EAX 中。



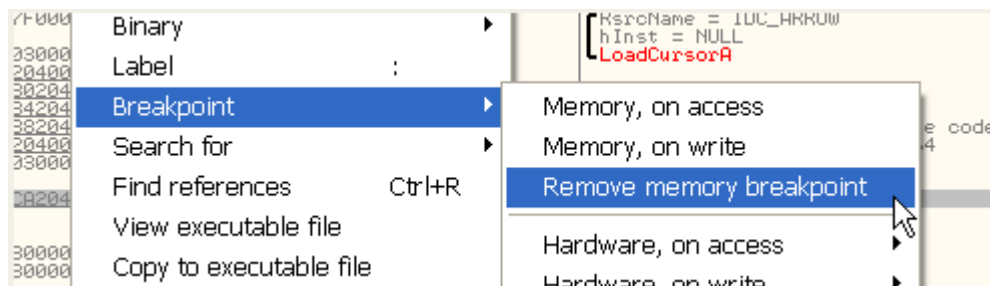
再次 F9 键运行起来,还会不会出发内存断点呢?对,依然会。



OD 会再次中断在尝试读取 4020CA 内存单元的指令处,这是一个 push 指令,该指令会尝试将 4020CA 内存单元的内容压入堆栈。

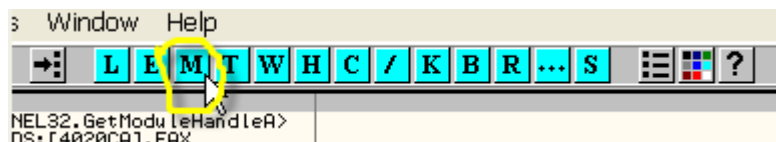
Memory breakpoint when reading [004020CA]

要删除内存断点的话,可以数据窗口中单击鼠标右键选择-Breakpoint-Remove memory breakpoint。你还可以设置一个新的内存断点,旧的内存断点会自动被删除。



“Memory,on access”是内存访问断点(读或者写),“Memory,on write”是内存写断点。

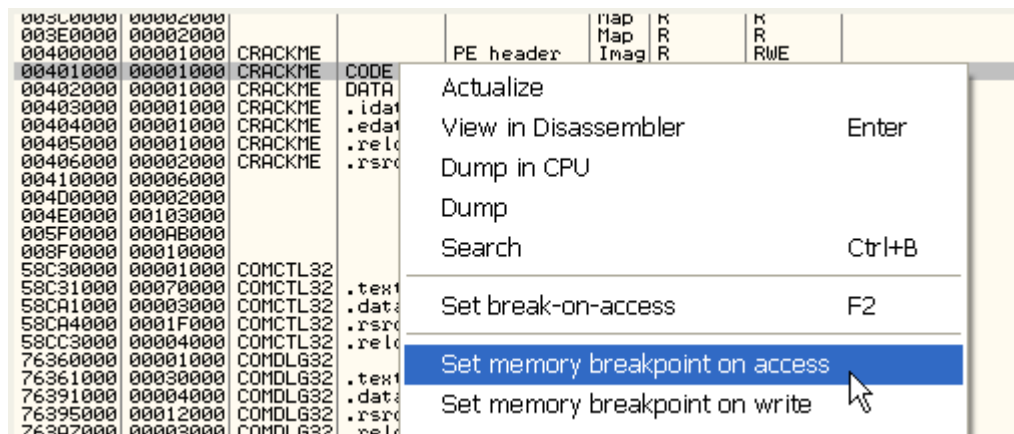
OD 也可以对区段设置内存断点,我们选择菜单项 View-Memory,也可以按工具栏中的[M]按钮打开内存窗口。



这个列表包含了 CrackMe 加载的一些区段以及其加载的一些 DLL 的区段,如图我们选中了以 401000 开头的区段。

00350000	00001000				Priv	RW	RW	
00390000	00004000				Priv	RW	RW	
003A0000	00004000				Priv	RW	RW	
003B0000	00003000				Map	R	R	\Device\HarddiskVolume1\W
003C0000	00002000				Map	R	R	
003E0000	00002000				Map	R	R	
00400000	00001000	CRACKME	PE header		Imag	R	RWE	
00401000	00001000	CRACKME	code		Imag	R	RWE	
00402000	00001000	CRACKME	DATA		Imag	R	RWE	
00403000	00001000	CRACKME	.idata	imports	Imag	R	RWE	
00404000	00001000	CRACKME	.edata	exports	Imag	R	RWE	
00405000	00001000	CRACKME	.reloc	relocations	Imag	R	RWE	
00406000	00002000	CRACKME	.rsrc	resources	Imag	R	RWE	
00410000	00006000				Map	R E	R E	
004D0000	00002000				Map	R E	R E	
004E0000	00103000				Map	R	R	
005F0000	000AB000				Map	R E	R E	
00600000	00010000				Map	R	R	

在选中部分上单击鼠标右键选择-Set Memory breakpoint on access。



下面还有 Set memory breakpoint on write 的选项,但是这里我们选择 Set memory breakpoint on access 并且运行起来。



中断在了下一行指令上。

Memory breakpoint when executing [00401002]

因为试图执行 401002 处的代码,而当前代码段设置了内存访问断点,尝试执行当前代码段的任何一条指令都会触发内存访问断点。

77FAE000	00002000	SHLWAPI	.rsrc	resources	Imag	R	RWE	
77FB0000	00006000	SHLWAPI	.reloc	relocations	Imag	R	RWE	
7C800000	00001000	kernel32		PE header	Imag	R	RWE	
7C801000	00002000	kernel32	.text	code,import	Imag	R	RWE	
7C883000	00005000	kernel32	.data	data	Imag	R	RWE	
7C888000	00073000	kernel32	.rsrc	resources	Imag	R	RWE	
7C8FB000	00006000	kernel32	.reloc	relocations	Imag	R	RWE	
7C910000	00001000	ntdll		PE header	Imag	R	RWE	
7C911000	0007B000	ntdll	.text	code,export	Imag	R	RWE	

我们对 kernel32 的代码段设置内存访问断点,列表的下面可以找到。

77E50000	00001000	RPCRT4							
77E51000	00002000	RPCRT4	.text						
77E53000	00007000	RPCRT4	.orpc						
77EDA000	00001000	RPCRT4	.data						
77EDB000	00001000	RPCRT4	.rsrc						
77EDC000	00005000	RPCRT4	.reloc						
77EF0000	00001000	GDI32							
77EF1000	000042000	GDI32	.text						
77F33000	00001000	GDI32	.data						
77F34000	00001000	GDI32	.rsrc						
77F35000	00002000	GDI32	.reloc						
77F40000	00001000	SHLWAPI							
77F41000	00006C000	SHLWAPI	.text						
77FAD000	00001000	SHLWAPI	.data						
77FAE000	00002000	SHLWAPI	.rsrc						
77FB0000	00006000	SHLWAPI	.reloc						
7C800000	00001000	kernel32							
7C801000	00002000	kernel32	.text						
7C803000	00005000	kernel32	.data	data	resources	Image R	RWE		
7C808000	00007000	kernel32	.rsrc			Image R	RWE		
7C80FA000	00006000	kernel32	.reloc	relocations		Image R	RWF		

Set memory breakpoint on access
Set memory breakpoint on write
Remove memory breakpoint
Set access
Copy to clipboard
Sort by
Appearance

我们选择-Set memory breakpoint on access,当试图读取/写入/执行 Kernel32 代码段的时候就会中断下来，我们运行起来。

7C80B529	8BFF	MOV EDI,EDI	ntdll.7C920738
7C80B52B	55	PUSH EBP	
7C80B52C	8BEC	MOV EBP,ESP	
7C80B52E	837D 08 00	CMP DWORD PTR SS:[EBP+8],0	
7C80B532	74 18	JE SHORT kernel32.7C80B540	
7C80B534	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
7C80B537	E8 682D0000	CALL kernel32.7C80E2A4	
7C80B53C	85C0	TEST EAX,EAX	
7C80B53E	74 08	JE SHORT kernel32.7C80B548	
7C80B540	FF70 04	PUSH DWORD PTR DS:[EAX+4]	
7C80B543	E8 F4300000	CALL kernel32.GetModuleHandleW	
7C80B548	5D	POP EBP	
7C80B549	C2 0400	RETN 4	
7C80B54C	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
7C80B552	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]	
7C80B555	8B40 08	MOV EAX,DWORD PTR DS:[EAX+8]	
7C80B558	EB EE	JMP SHORT kernel32.7C80B548	
7C80B55A	90	NOP	
7C80B55B	90	NOP	
7C80B55C	90	NOP	

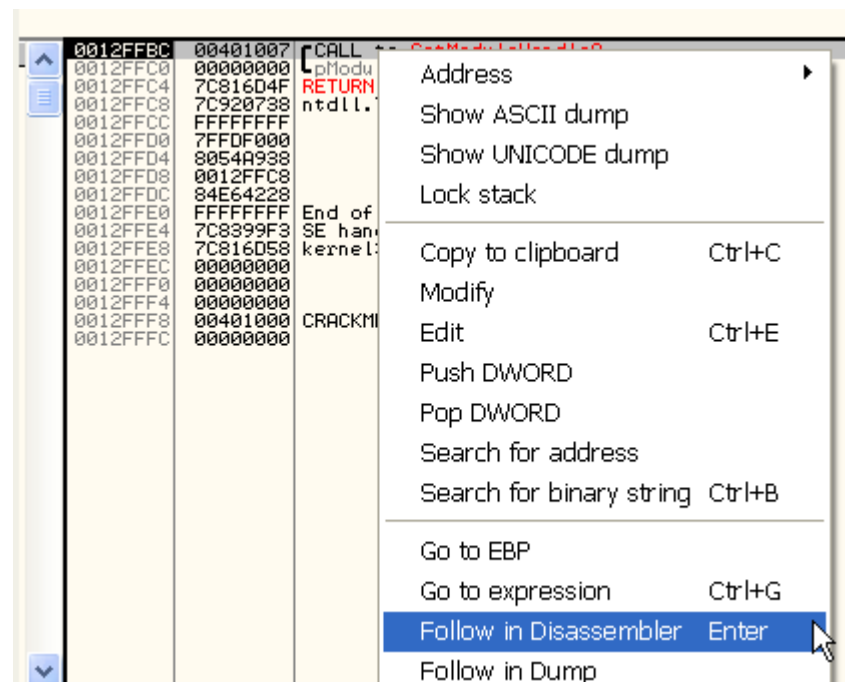
从堆栈的顶部来看,调用 API 函数的时候中断下来了。

0012FFBC	00401007	CALL to GetModuleHandleA	
0012FFC0	00000000	pModule = NULL	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F	
0012FFC8	7C920738	ntdll.7C920738	
0012FFCC	FFFFFFFF		
0012FFD0	7FFDF000		
0012FFD4	8054A938		
0012FFD8	0012FFC8		
0012FFDC	84E64228		
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		

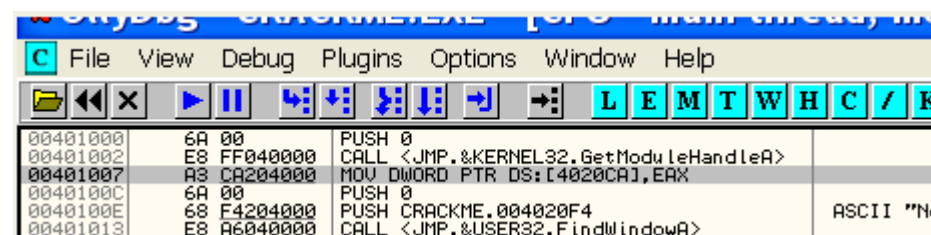
当前执行 kernel32.dll 中的第一个 API 函数的时候中断下来了,从堆栈中我们可以看到返回地址:

0012FFBC	00401007	CALL to GetModuleHandleA	
0012FFC0	00000000	pModule = NULL	
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F	
0012FFC8	7C920738	ntdll.7C920738	
0012FFCC	FFFFFFFF		
0012FFD0	7FFDF000		
0012FFD4	8054A938		
0012FFD8	0012FFC8		
0012FFDC	84E64228		
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		

在返回地址上单击鼠标右键选择-Follow in Disassembler。



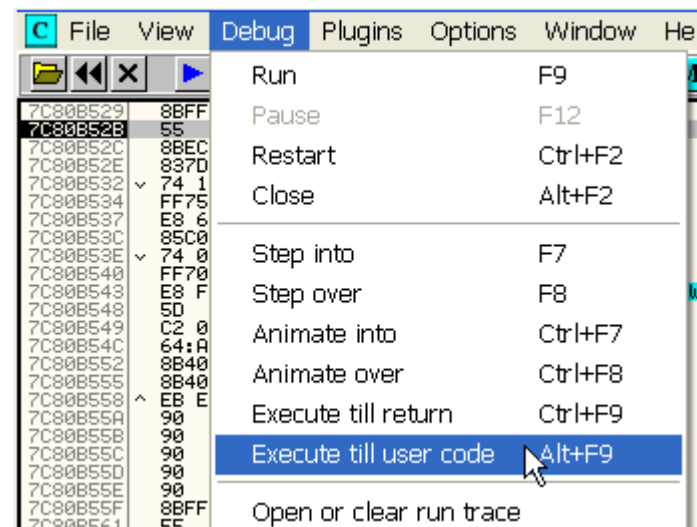
返回地址是 401007,是调用 GetModuleHandleA 后面的一行。



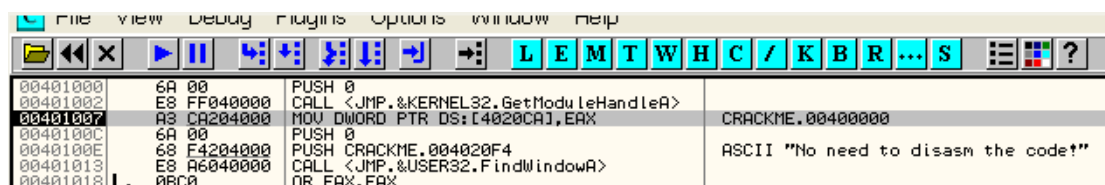
正下方是调用 `FindWindowA`,调用这个函数的时候不会触发内存访问断点,应该这个函数是属于另一个 DLL-`user32.dll` 的。如果你再次运行的话,会中断在下一条指令处,因为下一条指令也属于 `kernel32` 的代码段,

所以执行 `GetModuleHandleA` 的每条指令的时候内存访问断点都会触发,所以这个时候我们可以选择 `-Remove memory breakpoint` 删除内存断点。

如果你想要返回到主程序模块中,可以选择主菜单项-Debug-Execute till user code。有的时候,这个方法不起作用。我们可以在函数返回指令外设置断点,然后运行起来,等中断下来以后,再 F7 或者 F8 单步到主程序的代码中。

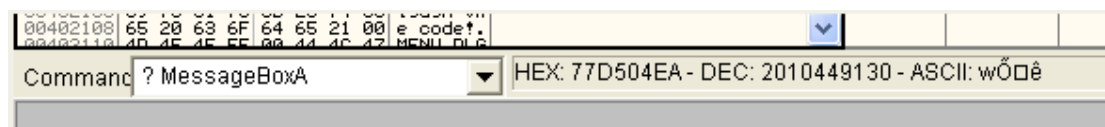


这里 Execute till user code 起作用了,我们回到了主程序模块的代码中。后面我们会介绍哪些情况下这种方式不奏效。

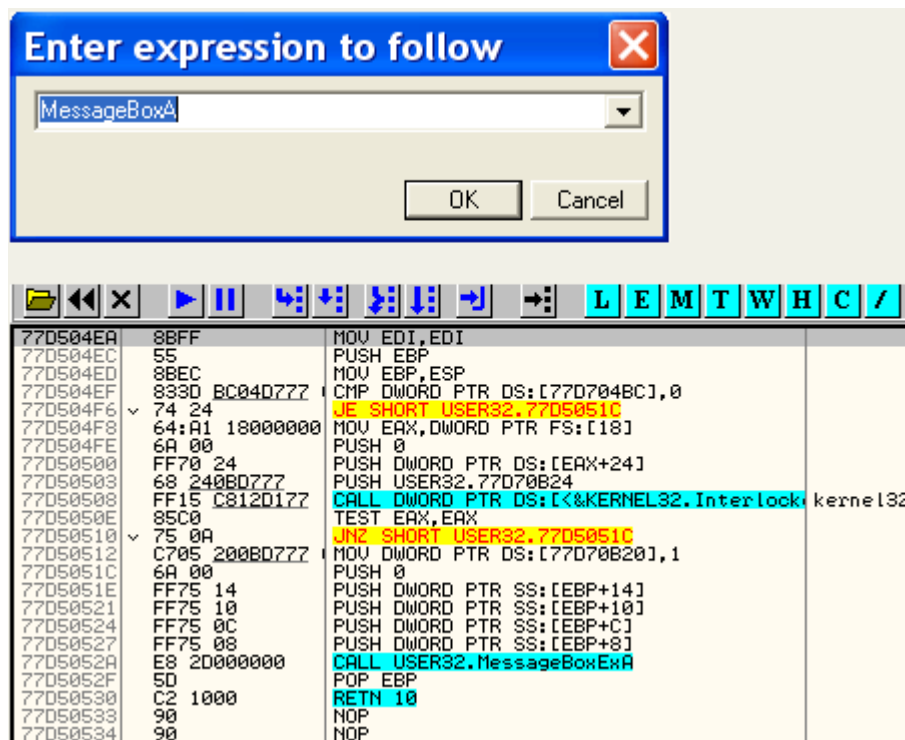


现在,你可以再次对 kernel32.dll 的代码段设置内存访问断点了,让程序调用 Kernel32.dll 中的 API 函数的时候再次中断下来。

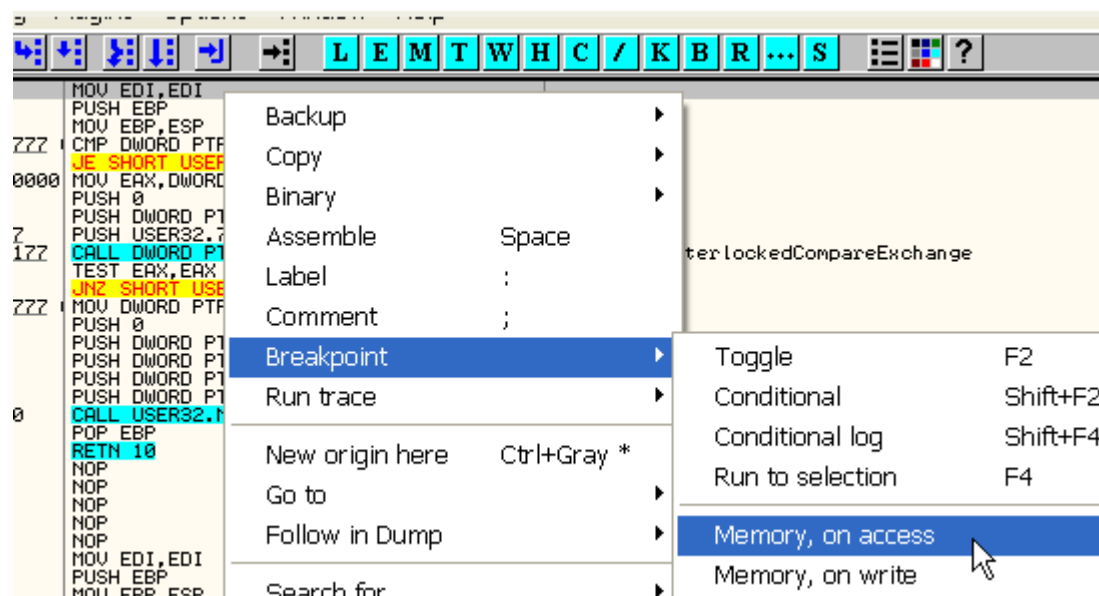
此外,如果程序会检测函数首字节是否为 0xCC 的话,这个时候我们使用 bp MessageBoxA 命令下断点就无效了,这个时候我们可以尝试一下内存访问断点。我们来看个例子:



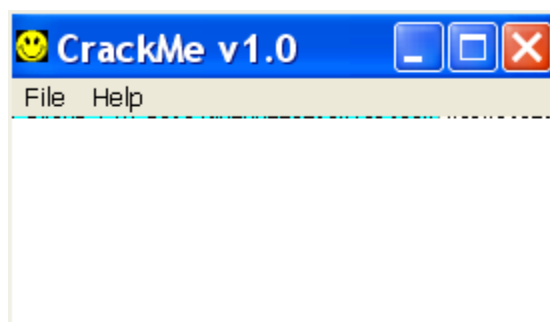
MessageBoxA 在我的机器上的对应的地址是 77D504EA。通过单击鼠标右键选择-Goto-Expression 输入该函数的名称:



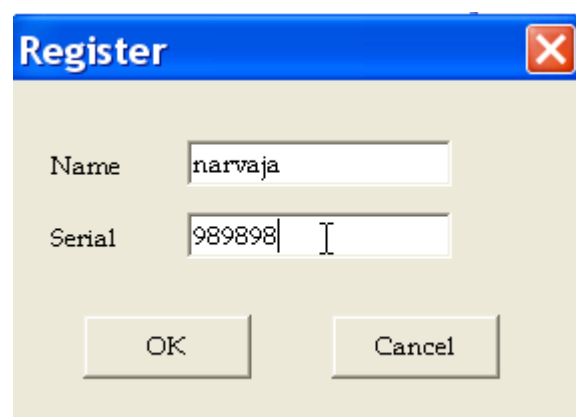
第一条指令被高亮显示:



这里我们单击鼠标右键选择-Breakpoint-Memory,on access 或者 memory,on write,这里我选择 Memory,on access 然后运行起来。



选择 help-Register,然后输入任意用户名和序列号:



单击 OK。

77D504EA	8BFF	MOV EDI,EDI	
77D504EC	55	PUSH EBP	
77D504ED	8BEC	MOV EBP,ESP	
77D504EF	833D BC040777	CMP DWORD PTR DS:[77D704BC],0	
77D504F6	74 24	JE SHORT USER32.77D50510	
77D504F8	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
77D504FE	6A 00	PUSH 0	
77D50500	FF70 24	PUSH DWORD PTR DS:[EAX+24]	
77D50503	68 24080777	PUSH USER32.77D70824	
77D50508	FF15 C812D177	CALL DWORD PTR DS:[<&KERNEL32.Interlock	kernel32.Interl
77D5050E	85C0	TEST EAX,EAX	
77D50510	75 0A	JNZ SHORT USER32.77D50510	
77D50512	C705 20080777	MOV DWORD PTR DS:[77D70820],1	
77D5051C	6A 00	PUSH 0	
77D5051E	FF75 14	PUSH DWORD PTR SS:[EBP+14]	
77D50521	FF75 10	PUSH DWORD PTR SS:[EBP+10]	
77D50524	FF75 0C	PUSH DWORD PTR SS:[EBP+C]	
77D50527	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
77D5052A	E8 2D000000	CALL USER32.MessageBoxExA	
77D5052F	5D	POP EBP	
77D50530	C2 1000	RETN 10	
77D50533	90	NOP	
77D50534	90	NOP	

在同一个 API 函数中,如果通过 bp 命令设置断点会被程序检测而导致断点失效的话,也许设置内存访问断点可以绕过这个检测。

设置内存访问断点这个方法也可以通过检测内存页的属性并恢复内存页的属性来进行保护,但是这在反调试技巧中并不常见。

下一章,我们将介绍硬件断点和消息断点,条件断点我们会在后面介绍。