

## 第七章

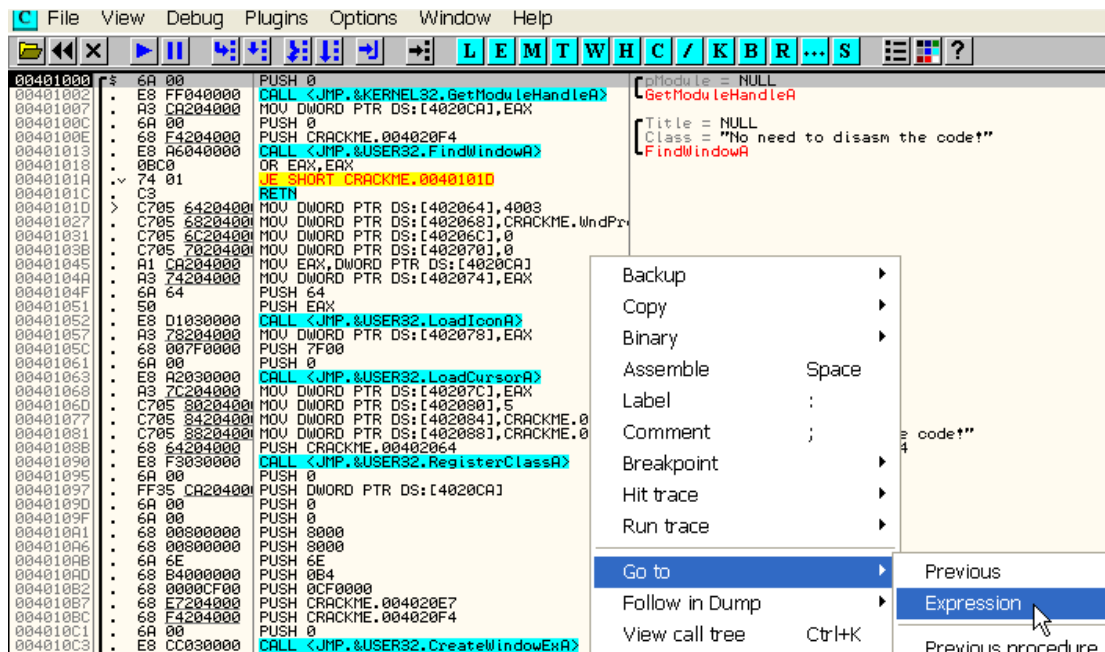
call 和 ret

我故意把 call 和 ret 指令留到这个部分来讲,因为你们必须得弄明白前面的基础知识,有了前面的基础,我们现在来介绍 call 和 ret。

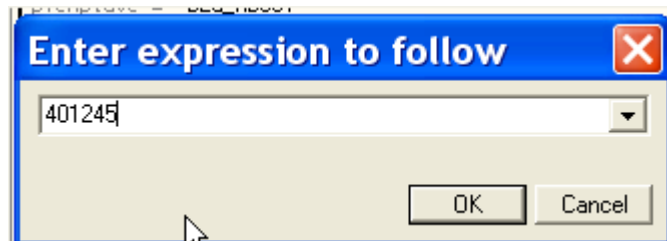
尽管这两条指令看起来很简单,但是很多初学者并不能完全领会这两条指令的本质。所以我把它们作为一个单独的章节来介绍,我认为这样做是可取的。

让我们继续用 OD 载入 CrueHead'a 的 CrackMe。

在任意一行点击鼠标右键选择-“Go to”-“Expression”。



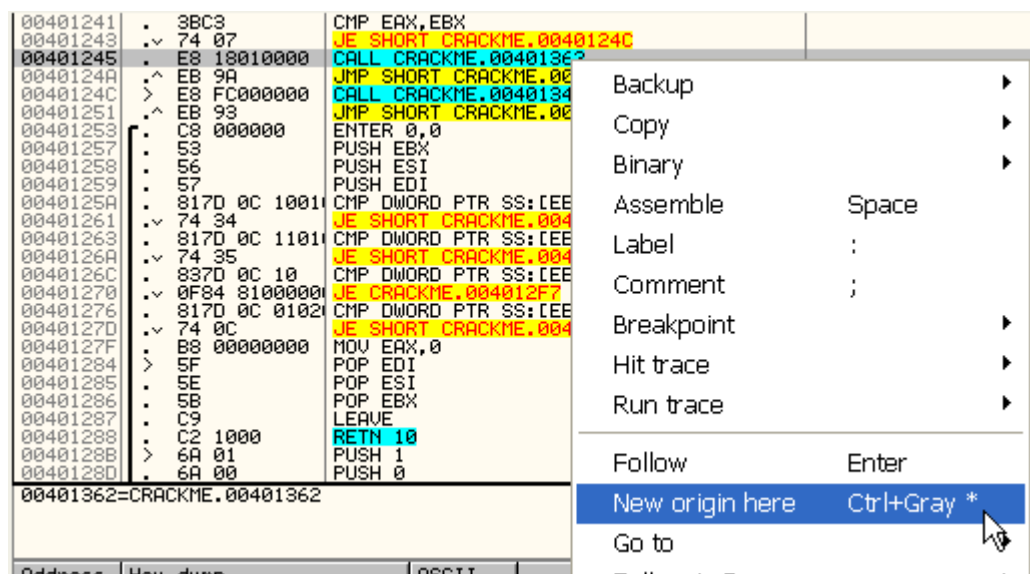
在弹出的对话框中输入 401245。



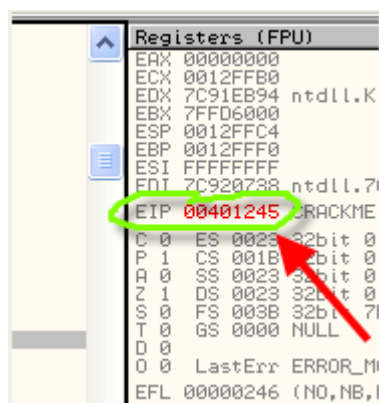
直接跳转到一个反汇编代码的地址处,这里有一个 Call 指令,我们拿它来练习。

00401240	. 58	POP EAX	
00401241	. 3BC3	CMP EAX,EBX	
00401243	. 74 07	JE SHORT CRACKME.0040124C	
00401245	. E8 18010000	CALL CRACKME.00401362	
0040124A	. EB 9A	JMP SHORT CRACKME.004011E6	
0040124C	. E8 FC000000	CALL CRACKME.0040134D	
00401251	. EB 93	JMP SHORT CRACKME.004011E6	
00401253	. C8 000000	ENTER 0,0	
00401257	. 53	PUSH EBX	
00401258	. 56	PUSH ESI	
00401259	. 57	PUSH EDI	
0040125A	. 817D 0C 1001	CMP DWORD PTR SS:[EBP+C],110	

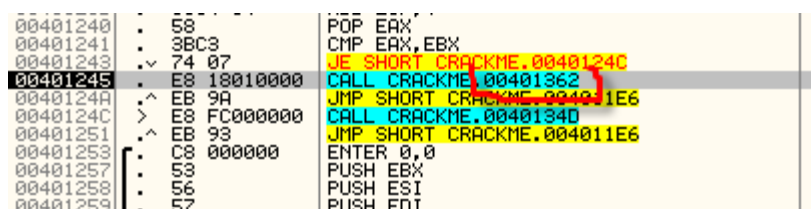
我们在练习的这个 Call 指令上面单击鼠标右键选择-“New origin here”。现在,EIP 的值就变成了 401245,这就意味着下一条要执行的指令就是我们的这个 Call 指令。



我们将看到 EIP 被修改了。



我们回到我们 Call 指令这里。



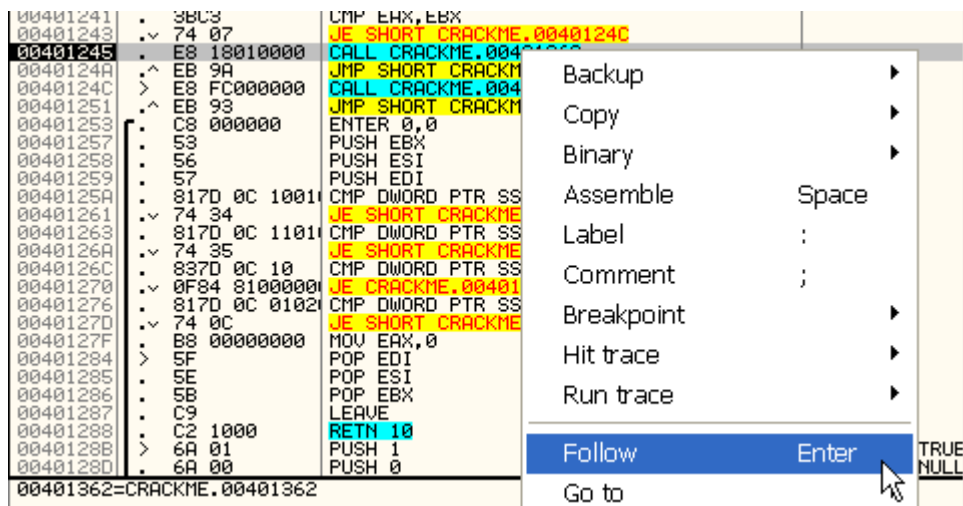
Call 指令是将转移到指定的子程序处,它的操作数就是给定的地址。例如:

Call 401362 表示将转移到地址 401362 处,将调用 401362 处的子程序,一旦子程序调用完毕就返回到 Call 指令的下一条语句处。

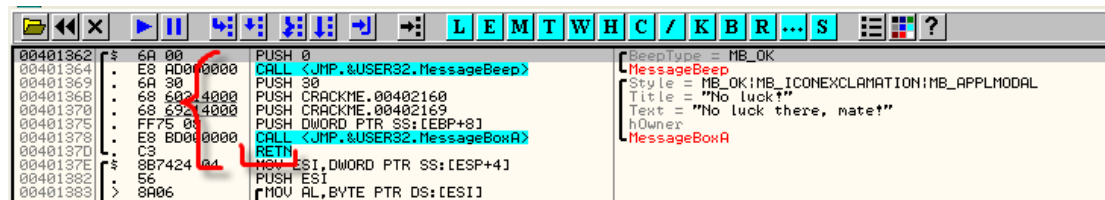
在这种情况下,完成 401362 的子程序调用以后,则会返回到 40124A 地址处。

针对于 call 指令,OD 提供了一些有用的跟踪机制。如果想继续跟进子程序内部,你可以按 F7 键跟进。如果只是想先看看子程序里面的内容再决定要不要跟进的话,可以单击鼠标右键-“Follow”。最后,如果我们不想继续跟踪该子程序了,我们可以按下 F8 键,继续执行 call 指令的下一条语句。

OD 允许我们看我们感兴趣的子程序里面的内容,但是不执行,只需要在 call 指令上面单击鼠标右键选择“Follow”。



正如你所看到的,EIP 的值仍然是 401245。“Follow”按钮只是简单的将指定地址处代码反汇编,但是并不会执行代码,并且等待我们进一步的操作。



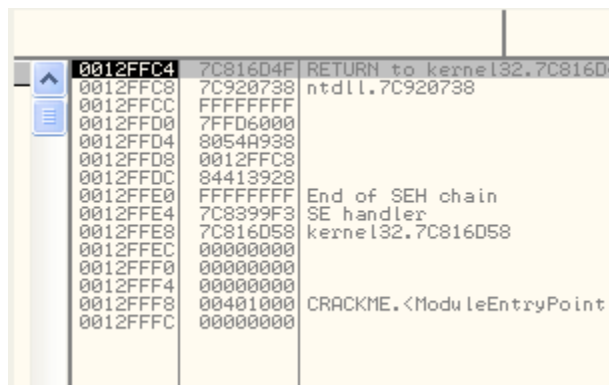
我们这个子程序起始地址是 401362,那么哪里是子程序的结束呢?这里我们下面第一个出现的 ret 指令就是子程序的结束。OD 中还可以写成 retm。执行完 ret 指令以后,程序就会返回到 call 指令的下一条指令 40124A 处。

现在我们知道了怎么查看子程序里面代码了,如果我们想回到之前的那一行的话,我们可以按数字键减号。这个键可以让我们回到按“Follow”的那一行。

在这里,我们可以看到原先的 call 指令:

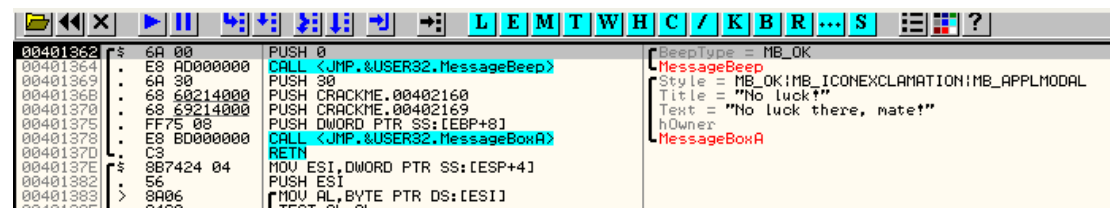


现在我们将按 F7 键跟进这个子程序,首先我们来看看堆栈的情况。这一步很重要,为了让子程序执行完 ret 指令后,知道要返回到哪里,返回地址将要存到堆栈当中。



上一张图片显示是我机器上的栈中的内容。或许跟你机器上的不一样,这无关紧要。

按 F7 键:

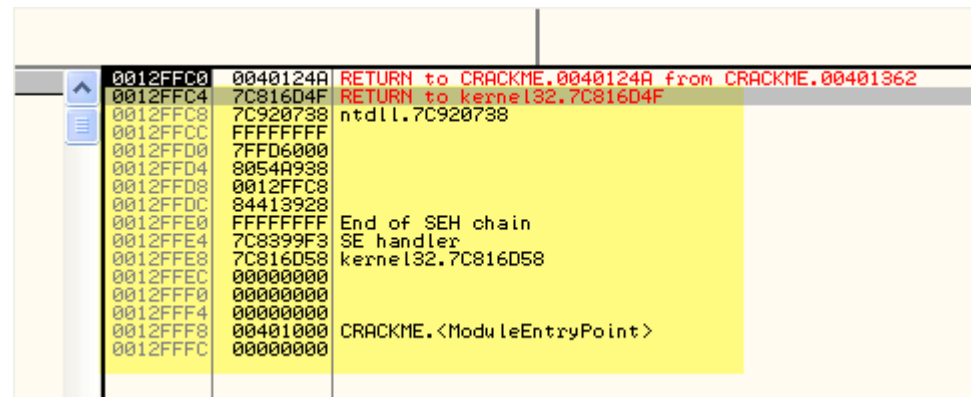


Address	Disassembly	Comment
00401362	PUSH 0	
00401364	CALL <JMP.<USER32.MessageBeep>	
00401369	PUSH 30	
0040136B	PUSH CRACKME.00402160	
00401370	PUSH CRACKME.00402169	
00401375	PUSH DWORD PTR SS:[EBP+8]	
00401378	CALL <JMP.<USER32.MessageBoxA>	
0040137D	RETN	
0040137E	MOV ESI,DWORD PTR SS:[ESP+4]	
00401382	PUSH ESI	
00401383	MOV AL,BYTE PTR DS:[ESI]	

BeepType = MB\_OK  
MessageBeep  
Style = MB\_OK!MB\_ICONEXCLAMATION!MB\_APPLMODAL  
Title = "No luck!"  
hOwner  
MessageBoxA

现在跟进子程序里面了,可以看到不像上面的“Follow”操作,现在 EIP 的值已经改变了-现在它等于 401362,这就说明我们真正开始执行代码了。

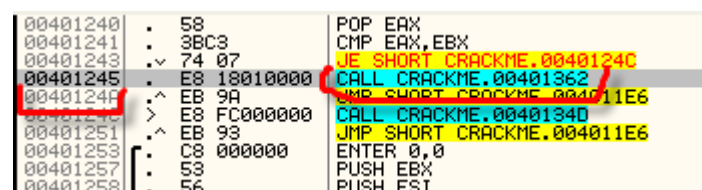
让我们继续把注意力转移到堆栈中。



Address	Disassembly	Comment
0012FFC0	0040124A	RETURN to CRACKME.0040124A from CRACKME.00401362
0012FFC4	7C81604F	RETURN to kernel32.7C81604F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD6000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84413928	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816058	kernel32.7C816058
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME.<ModuleEntryPoint>
0012FFFC	00000000	

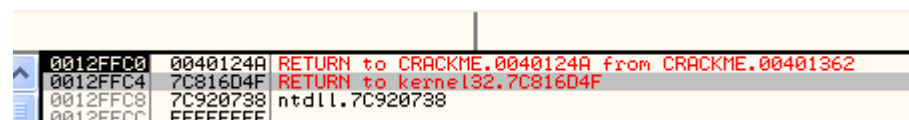
这张图片高亮显示的部分是我们跟进子程序之前的情况,现在我们有了一个新的元素,即 call 指令的下一条指令的地址被压入了堆栈中作为返回地址。

40124A 这个地址就是 call 指令的下一条指令的地址,如果你不记得了,我们来 call 指令处看看。



Address	Disassembly	Comment
00401240	POP EAX	
00401241	CMP EAX,EBX	
00401243	JE SHORT CRACKME.0040124C	
00401245	CALL CRACKME.00401362	
0040124F	JMP SHORT CRACKME.004011E6	
00401251	CALL CRACKME.0040134D	
00401253	JMP SHORT CRACKME.004011E6	
00401257	ENTER 0,0	
00401258	PUSH EBX	
00401259	PUSH ESI	

OD 还为我们提供了一些额外的信息。

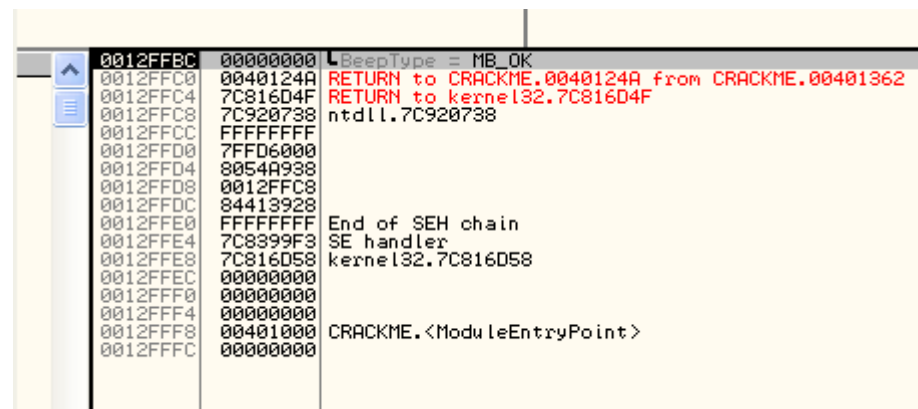


Address	Disassembly	Comment
0012FFC0	0040124A	RETURN to CRACKME.0040124A from CRACKME.00401362
0012FFC4	7C81604F	RETURN to kernel32.7C81604F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	

红色的提示信息告诉我们,40124A 就是以 401262 为起始地址的子程序的返回地址。

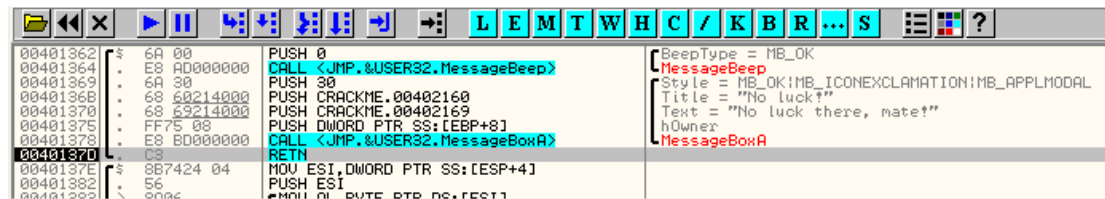
但是 OD 仍然不知道 ret 指令在哪里,但是知道 401362 是子程序的起始地址,当执行完 ret 指令后就会返回到 40124A 地址处。

我们继续按 F7 键,执行 push 0 指令,看看这个 0 是不是压入堆栈中并且存放返回地址的上面了。

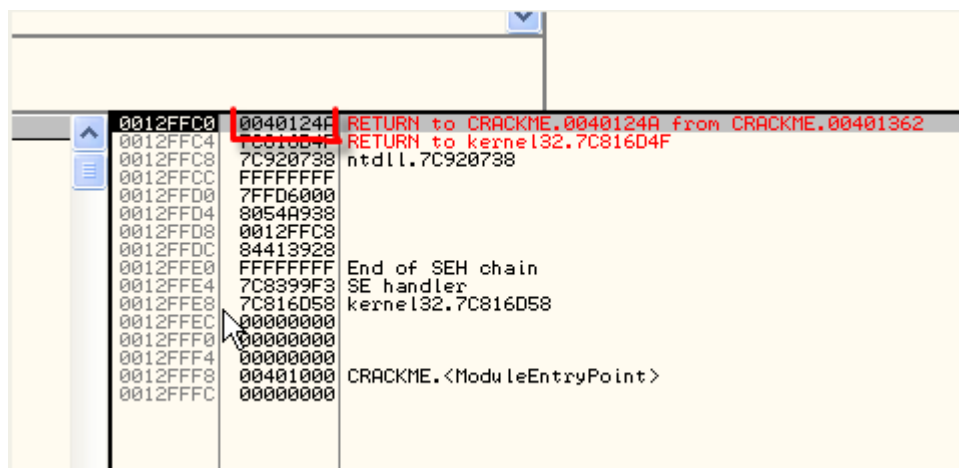


Address	Disassembly	Comment
0012FFC0	0040124A	RETURN to CRACKME.0040124A from CRACKME.00401362
0012FFC4	7C81604F	RETURN to kernel32.7C81604F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD6000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84413928	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816058	kernel32.7C816058
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	CRACKME.<ModuleEntryPoint>
0012FFFC	00000000	

这个程序可能包含了成千上万个堆栈操作(push,pop 等),在堆栈中添加或者删除了各种各样的值,但是当我们执行到 ret 指令的时候,栈顶存放的一般是子程序的返回地址。我们一直按 F8 不跟进 call 指令里面,直到遇到了 ret 指令停止。



现在我们到了子程序末尾的 ret 指令了,这个时候栈顶存放的是子程序的返回地址。

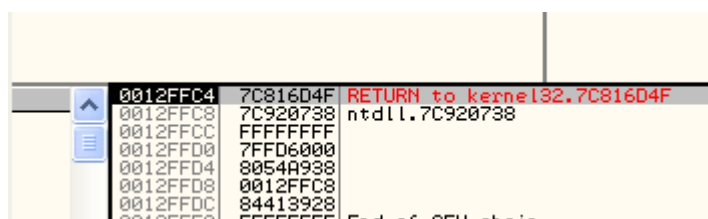


因此,我们可以知道 ret 指令是子程序的结束,也就是说,如果我们 call 跟进的话,那么 ret 就能返回到 call 指令的下一条语句处。

按 F7 键:



现在返回到了 40124A 处了,堆栈也恢复到了调用 call 指令之前的状态。



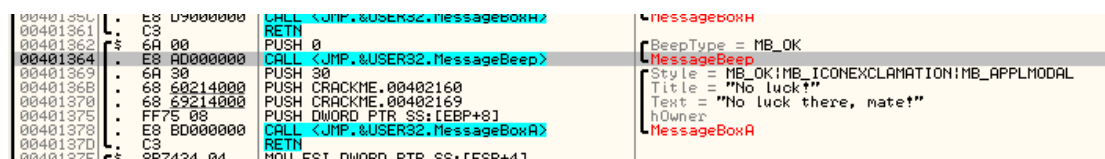
我这里补充一点,ret 指令可不仅仅用于子程序的返回,例如:

PUSH 401256

RET

这里将 401256 压入堆栈。下面的 ret 指令会将 401256 当做子程序的返回地址,其实它并不是返回地址,但是执行 ret 指令后我们依然可以转移到 401256 地址处。这段代码和 JMP 401256 指令的功能是一样的。

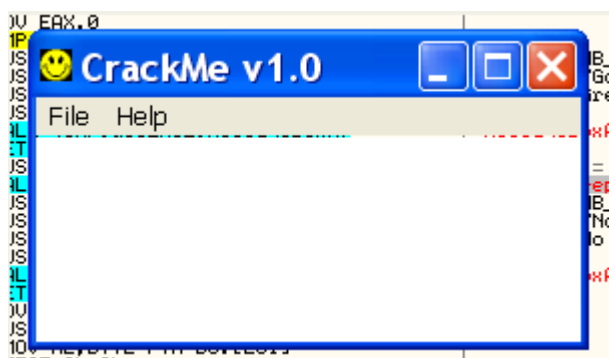
接下来,我们来看看 CALL/RET 的另外一个例子。重新载入 CrueHead'a 的 CrackMe。单击鼠标右键选择“Go to” - “Expression”来到 401364 地址处。



在这个地址上按 F2 键,这样就设置了一个断点(我们后面会详细介绍)。当 OD 指令执行到这条语句的时候就会中断下来。

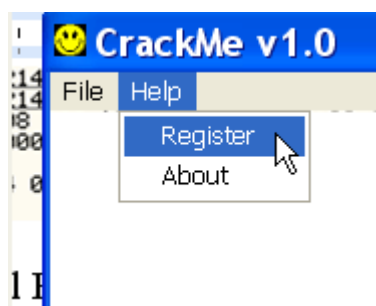
00401359	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
0040135C	E8 D9000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
00401361	C3	RETN	
00401362	6A 00	PUSH 0	BeepType = MB_OK
00401364	E8 A0000000	CALL <JMP.&USER32.MessageBeep>	MessageBeep
00401369	6A 30	PUSH 30	Style = MB_OK MB_ICONEXCLAMATION MB_APPLMODAL
0040136B	68 60214000	PUSH CRACKME.00402160	Title = "No luck!"
00401370	68 63214000	PUSH CRACKME.00402169	Text = "No luck there, mate!"
00401375	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401378	E8 BD000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
0040137D	C3	RETN	
0040137E	8B7424 04	MOV ESI,DWORD PTR SS:[ESP+4]	
00401382	56	PUSH ESI	

该地址处红色突出显示了,这说明这里条语句设置了断点。按 F9,让程序运行起来。

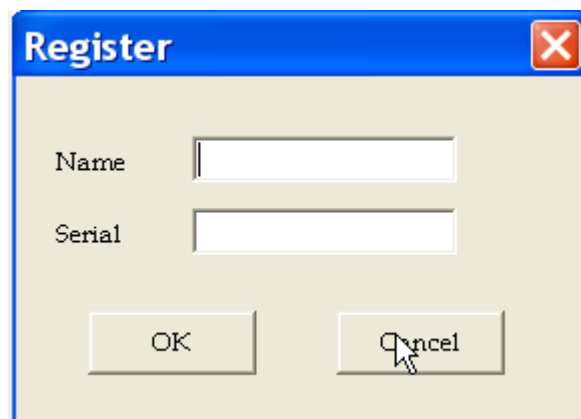


出现 CrackMe 的窗口。如果你没有看到它,通过 Alt+Tab 键切换窗口试试。

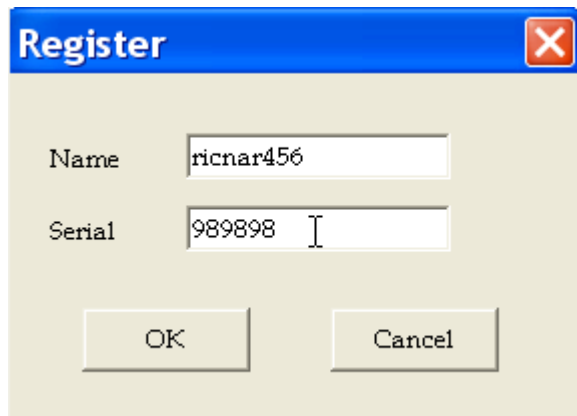
该程序还没有执行到我们设置了断点的代码处。我们找到 CrackMe 的菜单项,选择“Help” - “Register”。



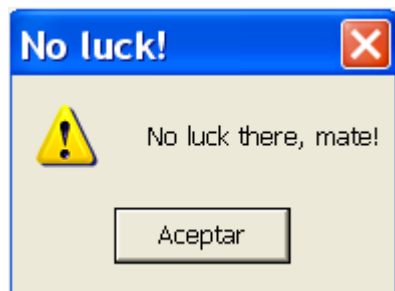
出现了下面的窗口,你可以输入名字和序列号。



随便填:



单击 OK。



弹出一个消息框,说我们不走运,提示名字和序列号不正确,当你关闭这个消息框,程序就会中断下来。

00401359	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
0040135C	E8 D9000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
00401361	C3	RETN	
00401362	6A 00	PUSH 0	BeepType = MB_OK
00401364	E8 AD000000	CALL <JMP.&USER32.MessageBeep>	MessageBeep
00401369	6A 30	PUSH 30	Style = MB_OK MB_ICONEXCLAMATION MB_APPLMODAL
0040136B	68 60214000	PUSH CRACKME.00402160	Title = "No luck!"
00401370	68 69214000	PUSH CRACKME.00402169	Text = "No luck there, mate!"
00401375	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401378	E8 BD000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
0040137D	C3	RETN	
0040137E	8B7424 04	MOV ESI,DWORD PTR DS:[ESP+4]	
00401382	56	PUSH ESI	
00401383	0000	CALL 00401383	

我们处于程序的代码中间,但是我们可以从堆栈当中得到一些基本的信息。

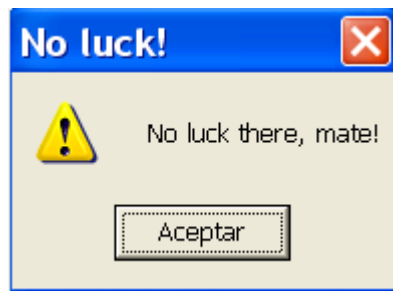
0012FE9C	00000000	BeepType = MB_OK
0012FEA0	0040124A	RETURN to CRACKME.0040124A from CRACKME.00401362
0012FEA4	68621800	ASCII "RICNAR456"
0012FEA8	00000000	
0012FEAC	0012FF1C	
0012FEB0	00401128	RETURN to CRACKME.WndProc from <JMP.&KERNEL32.ExitProcess>
0012FEB4	0012FF48	
0012FEB8	77D18734	RETURN to USER32.77D18734
0012FEBD	091E08A6	
0012FEC0	00000111	
0012FEC4	00000066	
0012FEC8	00000000	
0012FECC	00401128	RETURN to CRACKME.WndProc from <JMP.&KERNEL32.ExitProcess>
0012FED0	DCBAABCD	
0012FED4	00000000	
0012FED8	0012FF1C	
0012FEDC	00401128	RETURN to CRACKME.WndProc from <JMP.&KERNEL32.ExitProcess>
0012FEE0	0012FF48	
0012FEE4	77D18816	RETURN to USER32.77D18816 from USER32.77D1870C
0012FEE8	00401128	RETURN to CRACKME.WndProc from <JMP.&KERNEL32.ExitProcess>
0012FEEC	091E08A6	
0012FEF0	00000111	
0012FEF4	00000066	
0012FEF8	00000000	
0012FEFC	00402050	CRACKME.00402050
0012FF00	00402048	CRACKME.00402048
0012FF04	006DBFC0	
0012FF08	00000014	
0012FF0C	00000001	
0012FF10	00000000	

这里我们看到几个 RETURN TO 指令。显然,这些是我们在程序中保存的返回地址。我们现在在 40124A 这行按下回车键。

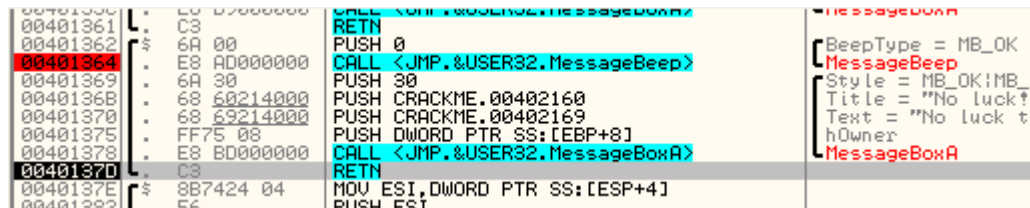
你会发现,我们来到 EIP 改变之前分析的子程序处。但是这次我们程序已经执行起来了。



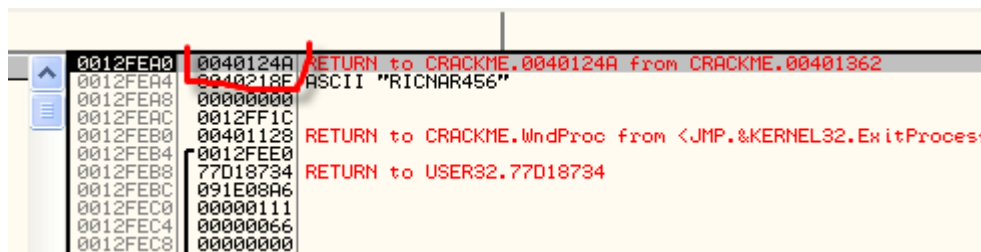
想之前一样一直单击 F8 键,直到遇到了 ret 指令。但是,在执行到 ret 指令之前,会弹出一个消息框,我们关闭消息框继续单步。



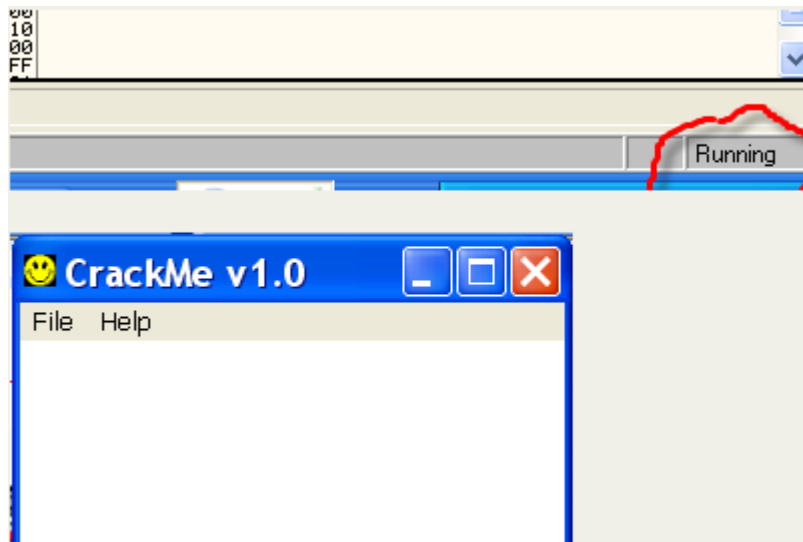
关闭它。



现在我们执行到了 ret 指令,和我们想的一样,栈顶现在保存的是返回地址。



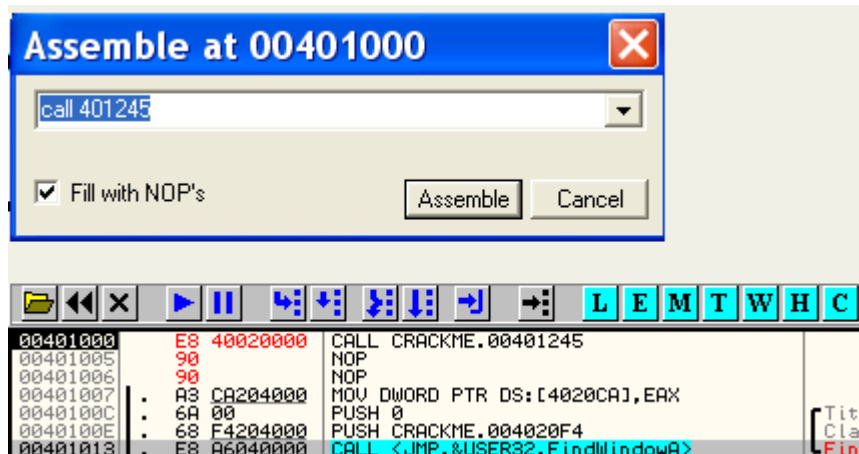
这次我们直接 F9 运行起来。



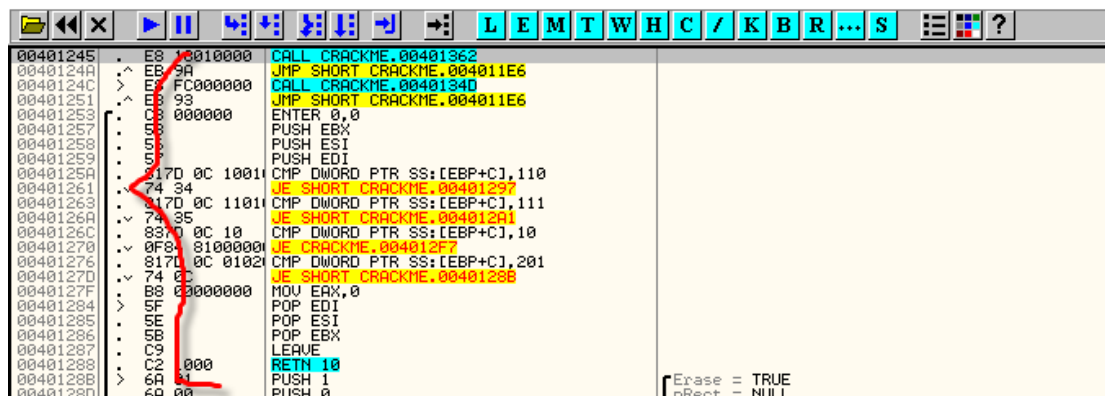
总结一下,我想说明的是,当程序中断下来的时候,我们可以从堆栈中得到很多有用的信息。我们只要粗略的看一眼堆栈中内容,就基本上可以知道哪些 call 指令被调用了,以及将要返回地址是多少。很明显,当一个子程序调用了另一个子程序的时候,通过这个子程序也可以去调用别的子程序,这就叫做嵌套调用。

现在来看看另外一个例子。重新载入 CrackMe,按下空格键,输入一条指令:Call 401245。



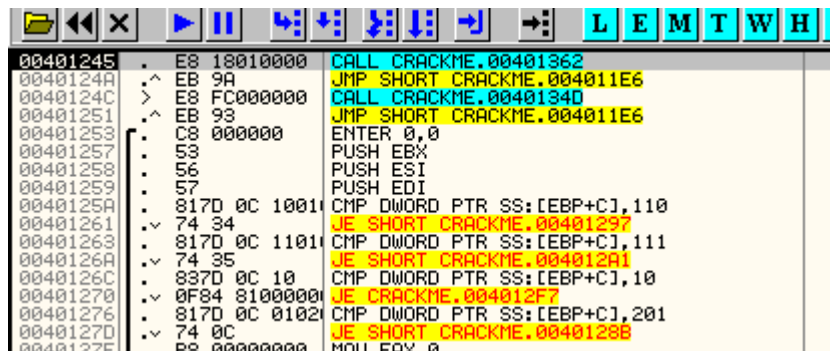


完了以后，我们单击鼠标右键选择“Follow”查看这个子程序。

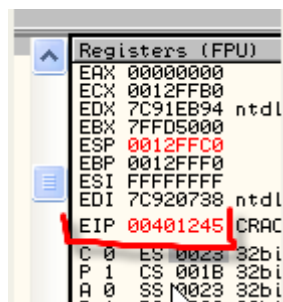


这个子程序起始于 401245,到 401288 地址处结束(其中 retm 10 指令跟我们熟悉 ret 指令略有不同,稍后详细说明)。注意嵌套调用(这个子程序的第一条指令就是调用另一个子程序)。

按减号键返回到上一层。然后按 F7 键,跟进到子程序里面。



这里我们就到了子程序里面,EIP 的值为 401245。



栈顶保存了 call 指令的返回地址。

0012FFC0	00401005	CRACKME.00401005
0012FFC4	7C920738	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD8000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	8412BAF0	
0012FFE0	FFFFFFFF	End of SEH chain

这确实是一个返回地址,但是 OD 没有以红色显示 RETURN TO 401005。要弄明白为什么这一次 OD 没有以红色显示返回信息,我们回车键跟随到 call 里面,然后让 OD 重新分析代码。

因为我们已经让 OD 混淆了。

为了纠正这种问题,我们可以在反汇编窗口中单击鼠标右键选择-“Analysis”-“Analyse code”。

00401259	57	PUSH EDI	
0040125A	817D 0C 1001	CMP DWORD PTR SS:[EBP+C],110	
00401261	74 34	JE SHORT CRACKME.00401297	
00401263	817D 0C 1101	CMP DWORD PTR SS:[EBP+C],111	
0040126A	74 35	JE SHORT CRACKME.004012A1	
0040126C	837D 0C 10	CMP DWORD PTR SS:[EBP+C],10	
00401270	0FB4 81000001	JE CRACKME.004012E7	
00401276	817D 0C 0102	CMP DWORD PTR SS:[EBP+C],102	
0040127D	74 0C	JE SHORT CRACKME.00401297	
0040127F	B8 00000000	MOV EAX,0	
00401284	5F	POP EDI	
00401285	5E	POP ESI	
00401286	5B	POP EBX	
00401287	C9	LEAVE EBX	
00401288	C2 1000	RETN 4	
0040128B	6A 01	PUSH 1	
0040128D	6A 00	PUSH 0	
0040128F	FF75 08	PUSH 8	
00401292	E8 B5010000	CALL CRACKME.00401297	
00401297	FF75 08	PUSH 8	
0040129A	E8 95010000	CALL CRACKME.00401297	
0040129F	EB E3	JMP SHORT CRACKME.00401297	
004012A1	33C0	XOR EAX,EAX	
004012A3	817D 10 EB03	CMP DWORD PTR SS:[EBP+3],EBX	
004012AA	74 4B	JE SHORT CRACKME.00401297	
004012AC	817D 10 EA03	CMP DWORD PTR SS:[EBP+3],EBX	
004012B3	75 3B	JNZ SHORT CRACKME.00401297	
004012B5	6A 0B	PUSH 11	
004012B7	68 8E214000	PUSH CRACKME.004012B7	
004012BC	68 E8030000	PUSH CRACKME.004012BC	
004012C1	FF75 08	PUSH 8	
004012C4	E8 07020000	CALL CRACKME.004012C4	
004012C9	83F8 01	CMP AL,1	
004012CC	C745 10 EB03	MOV DWORD PTR SS:[EBP+3],EAX	

004012F7=CRACKME.004012F7	
Address	Hex dump
00402000	00 00 00 00 00 00 00 00
00402008	00 00 00 00 00 00 00 00
00402010	00 00 00 00 00 00 00 00
00402018	00 00 00 00 00 00 00 00
00402020	00 00 00 00 00 00 00 00
00402028	00 00 00 00 00 00 00 00
00402030	00 00 00 00 00 00 00 00
00402038	00 00 00 00 00 00 00 00
00402040	00 00 00 00 00 00 00 00
00402048	00 00 00 00 00 00 00 00
00402050	00 00 00 00 00 00 00 00
00402058	00 00 00 00 00 00 00 00
00402060	00 00 00 00 00 00 00 00

Analysis	Analyse code
Remove analysis from module	

我们来看看重新分析后的代码。

0012FFC0	00401005	RETURN to CRACKME.<ModuleEntryPoint>+5 from CRACKME.00401245
0012FFC4	7C920738	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD8000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	8412BAF0	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C920738	ntdll.7C920738

现在返回地址被表示成了:程序入口点+5 = 401000 + 5 = 401005。

一般情况下,在改变程序代码后不要忘了重新分析代码。有的时候,代码分析可能是错的,这个时候应该选择“Analysis”-“Remove analysis from module”。

继续回到我们的例子。

00401245	\$ E8 18010000	CALL CRACKME.00401362
0040124A	> ^ EB 9A	JMP SHORT CRACKME.004011E6
0040124C	> E8 FC000000	CALL CRACKME.00401340
00401251	> ^ EB 93	JMP SHORT CRACKME.004011E6
00401253	· C8 000000	ENTER 0,0
00401257	· 53	PUSH EBX

F7 键跟进这个子程序。

0012FFBC	0040124A	RETURN to CRACKME.0040124A from CRACKME.00401362
0012FFC0	00401005	RETURN to CRACKME.<ModuleEntryPoint>+5 from CRACKME.00401245
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F from CRACKME.00401245
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD8000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	8412BAF0	
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	

现在的返回地址变成了另一个。一般情况下,返回地址之间的内容是子程序嵌套调用中 PUSH 指令或者其他操作压入堆栈的值。

如果你对程序没有什么头绪,不妨设置一个断点,然后将程序中断下来观察一下堆栈里面的情况以获取一些有用的信息。

继续看代码。

00401362	\$ 6A 00	PUSH 0
00401364	· E8 AD000000	CALL <JMP.&USER32.MessageBeep>
00401369	· 6A 30	PUSH 30
0040136B	· 68 60214000	PUSH CRACKME.00402160
00401370	· 68 60214000	PUSH CRACKME.00402169
00401375	· FF75 08	PUSH DWORD PTR SS:[EBP+8]
00401378	· E8 BD000000	CALL <JMP.&USER32.MessageBoxA>
0040137D	· C3	RETN
0040137E	\$ 8B7424 04	MOV ESI,DWORD PTR SS:[ESP+4]
00401382	· 56	PUSH ESI
00401383	> 8A06	MOV AL,BYTE PTR DS:[ESI]
00401385	· 84C0	TEST AL,AL
00401387	· 74 13	JE SHORT CRACKME.0040139C

BeepType = MB\_OK  
 MessageBeep  
 Style = MB\_OK|MB\_ICONEXC  
 Title = "No luck!"  
 Text = "No luck there, m  
 hOwner  
 MessageBoxA

查看一下堆栈:

0012FFBC	0040124A	RETURN to CRACKME.0040124A from CRACKME.00401362
0012FFC0	00401005	RETURN to CRACKME.<ModuleEntryPoint>+5 from CRACKME.00401245
0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F from CRACKME.00401245
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD8000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	8412BAF0	
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	


最上面的一个返回地址是当前子程序的返回地址。这里是 40124A。

```
00401241 3BC3 CMP EAX,EBX
00401243 74 07 JE SHORT CRACKME.0040124C
00401245 E8 18010000 CALL CRACKME.00401362
00401248 EB 9A JMP SHORT CRACKME.004011E6
0040124A E8 FC000000 CALL CRACKME.0040134D
0040124C EB 93 JMP SHORT CRACKME.004011E6
00401253 C8 000000 ENTER 0,0
00401257 53 PUSH EBX
00401258 56 PUSH ESI
00401259 57 PUSH EDI
00401261 817D 0C 1001 CMP DWORD PTR SS:[EBP+C],110
00401264 74 34 JE SHORT CRACKME.00401292
```

此外下面一个返回地址可能表示该子程序被另个子程序调用。

0012FFBC	00401240	RETURN to CRACKME.00401240 from CRACKME.00401362
0012FFC0	00401005	RETURN to CRACKME.<ModuleEntryPoint>+5 from CRACKME.00401240
0012FFC4	7C916D4F	RETURN to kernel32.7C916D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	

另外还告诉我们,程序最终会返回到 401005 处,我们直接在这一行按回车键,就能看到是由哪里调用的。



File View Debug Plugins Options Window Help

File View Debug Plugins Options Window Help

00401000 E8 40020000 CALL CRACKME.00401245

00401005 90 NOP

00401006 90 NOP

00401007 A3 C0204000 MOV DWORD PTR DS:[4020C0],EAX

00401008 6A 00 PUSH 0

00401009 68 F4204000 PUSH CRACKME.004020F4

00401013 E8 A6040000 CALL <JMP.>USER32.FindWindowA

我们要习惯子程序的嵌套调用,因为有的情况下,不是 2 层的子程序嵌套调用,有的时候可能是 30 层子程序的嵌套调用让我们来跟踪,所以我们要有足够的耐性。

如果你要什么不明白的地方就提出来。只有很好的理解了本章的内容才会更好的学习下一章。