

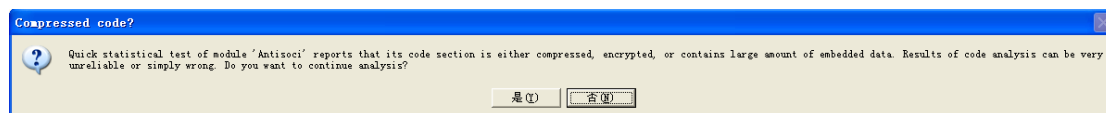
## 第二十四章-OllyDbg 反调试之综合练习

在我们介绍异常处理之前,我们先把上一章留下的 antisocial1 这个反调试的综合练习讲解一下。

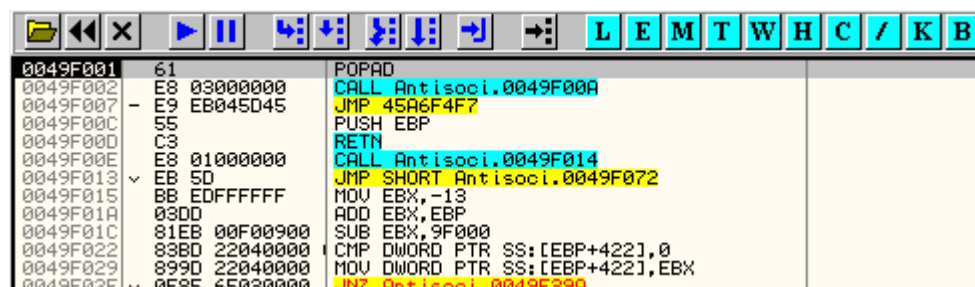
这是一个加过壳的程序,当然,我们还没有介绍壳的相关知识,我们只需要知道该程序加载到内存中以后,壳会解密原区段的各个区段的数据,然后跳转到原入口点(OEP)处执行原程序的代码。当外壳程序解密/解压还原并跳转到 OEP 处,此时的内存映像就是已解压/解密过的程序,这个时候将内存映像抓取并保存为文件即可(该过程称之为 Dump)。

首先我们用之前重命名过的 OD,并且带上之前介绍过的所有插件,运行起来,我们会发现程序会终止,我们来尝试修复它。

好了,现在我们用重命名过的 OD 加载 antisocial,将反反调试插件的选项都勾选上。

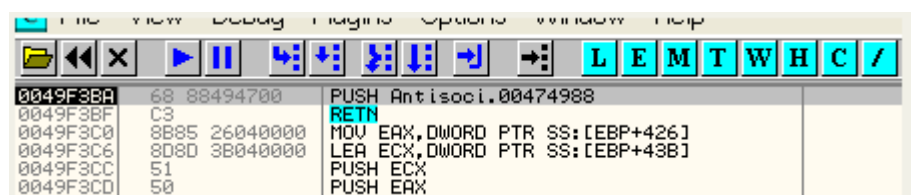


该提示表明该程序可能被加壳了,我们选择“是”按钮,然后就停在了入口点处。

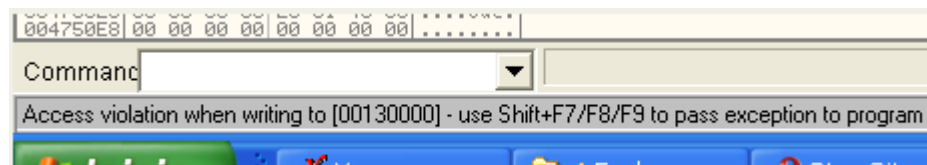


这里我们会看到一些奇怪的东西,POPAD 指令,该指令是从堆栈中恢复各个寄存器的值。正常情况下,应该是首先 PUSHAD 保存各个寄存器的值,而这里并没有执行 PUSHAD 指令,就比较可疑了。

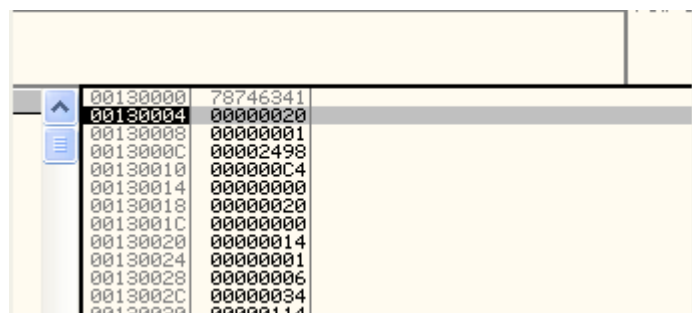
我们运行起来看看会发生什么。



这里



这里提示错误-PUSH 指令尝试压栈,但是这里没有写权限,但是通常来说堆栈应该是具有写权限的。我们来看一看堆栈。



当前栈顶指针指向的是 130000,我们重新启动程序。

Address	Size	Offset	Comment	Type	Access	Attributes	Permissions
00010000	00001000			Priv	RW		RW
00020000	00001000			Priv	RW		RW
0012B000	00001000			Priv	RW	Gua	RW
0012C000	00004000		stack of ma	Priv	RW	Gua	RW
00130000	00003000			Map	R		R
00140000	00003000			Priv	RW		RW
00240000	00006000			Priv	RW		RW
00250000	00003000			Map	RW		RW
00260000	0001C000			Map	R		R

我们可以看到当前各个区段的情况,我的机器上堆栈是从 12C000 开始,到 12FFFF 结束。而现在操作是以 130000 起始的内存,它并不是堆栈,并没有写权限,所以会报错。

我们再次来到发生错误的地方。

0049F3A7	8985 A8030000	OR ECX,ECX
0049F3AF	61	MOV DWORD PTR SS:[EBP+3A8],EAX
0049F3B0	75 08	POPAD
0049F3B2	B8 01000000	JNZ SHORT Antisoci.0049F3BA
0049F3B7	C2 0C00	MOV EAX,1
0049F3BA	68 88494700	RETN 0C
		PUSH Antisoci.00474988

我们可以看到该程序执行了另一个 POPAD 指令,接着就是条件跳转指令 JNZ 跳转到产生异常的 PUSH 的指令处,我们给 POPAD 指令这一行设置一个断点。

0049F3A6	59	POP ECX
0049F3A7	0BC9	OR ECX,ECX
0049F3A9	8985 A8030000	MOV DWORD PTR SS:[EBP+3A8],EAX
0049F3AF	61	POPAD
0049F3B0	75 08	JNZ SHORT Antisoci.0049F3BA
0049F3B2	B8 01000000	MOV EAX,1
0049F3B7	C2 0C00	RETN 0C
0049F3BA	68 88494700	PUSH Antisoci.00474988
0049F3BF	C3	RETN
0049F3C0	8B85 26040000	MOV EAX,DWORD PTR SS:[EBP+426]
0049F3C6	8D8D 3B040000	LEA ECX,DWORD PTR SS:[EBP+43B]

现在,我们重新运行该程序让其断在该断点处。

0049F3AF	61	POPAD
0049F3B0	75 08	JNZ SHORT Antisoci.0049F3BA
0049F3B2	B8 01000000	MOV EAX,1
0049F3B7	C2 0C00	RETN 0C
0049F3BA	68 88494700	PUSH Antisoci.00474988
0049F3BF	C3	RETN
0049F3C0	8B85 26040000	MOV EAX,DWORD PTR SS:[EBP+426]
0049F3C6	8D8D 3B040000	LEA ECX,DWORD PTR SS:[EBP+43B]

我们现在来看下堆栈。

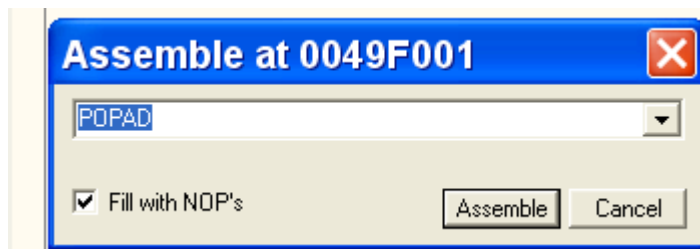
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0049F001	OFFSET Antisoci.<ModuleEntryPoint>
0012FFFC	00000000	

当前栈顶指针还属于堆栈的范围,紧接着执行 POPAD 指令。

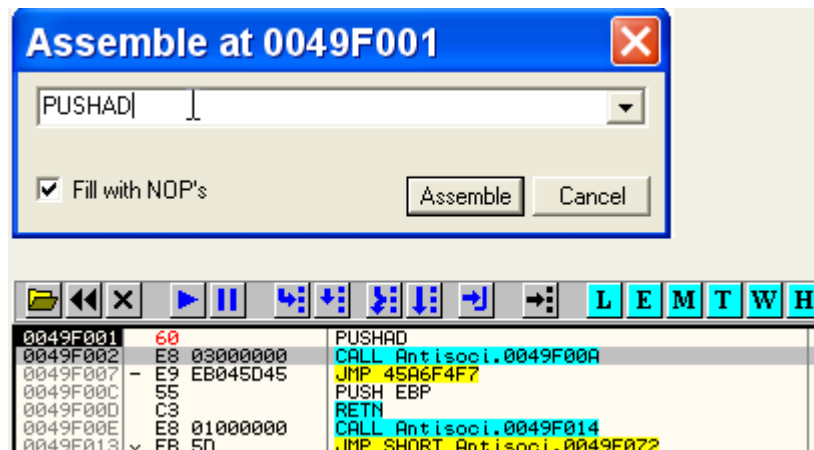
00130004	00000020	
00130008	00000001	
0013000C	00002498	
00130010	000000C4	
00130014	00000000	
00130018	00000020	
0013001C	00000000	
00130020	00000014	

当前栈顶指针已经超出了堆栈的范围,这是由刚刚的 POPAD 指令导致的,正常情况下来说应该是一开始执行 PUSHAD 指令将各个寄存器的值保存堆栈中,然后才是执行 POPAD 指令将堆栈的值恢复到各个寄存器中。好了,我们现在将开始处的 POPAD 指令替换成 PUSHAD 指令看看会发生什么,我们重新启动该程序。

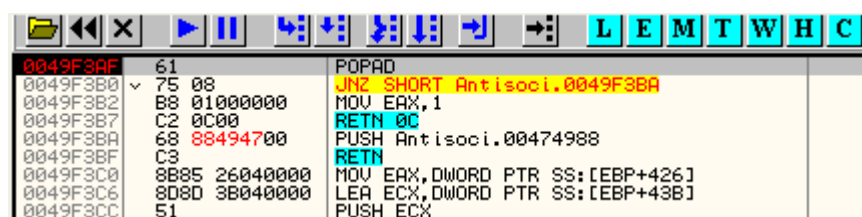
我们按下空格键。



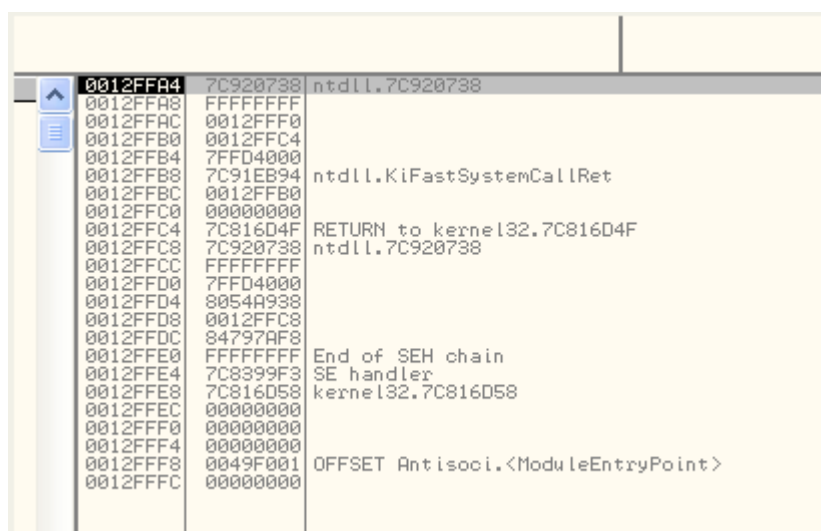
我们输入 PUSHAD。



现在我们运行起来,断在了第二个 POPAD 指令处。



但此时的堆栈情况如下:



此时执行 POPAD 指令堆栈并不会越界,我们按 F8 键单步看看。

0012FFC4	7C816D4F	RETURN to kernel32.7C816D4F
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD4000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	84797AF8	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0049F001	OFFSET Antisoci.<ModuleEntryPoint>
0012FFFC	00000000	

我们可以看到堆栈并没有越界。

0049F3AF	61	POPAD
0049F3B0	75 08	JNZ SHORT Antisoci.0049F3BA
0049F3B2	B8 01000000	MOV EAX,1
0049F3B7	C2 0C00	RETN 0C
0049F3BA	68 88494700	PUSH Antisoci.00474988
0049F3BF	C3	RETN
0049F3C0	8B85 26040000	MOV EAX,DWORD PTR SS:[EBP+426]
0049F3C6	8D8D 3B040000	LEA ECX,DWORD PTR SS:[EBP+43B]

我们到达了 PUSH 指令处,紧接着是 RET,我们按 F8 键单步步过 PUSH 和 RET 指令。

00474988	55	DB 55	CHAR 'U'
00474989	8B	DB 8B	
0047498A	EC	DB EC	
0047498B	83	DB 83	
0047498C	C4	DB C4	
0047498D	F0	DB F0	
0047498E	53	DB 53	CHAR 'S'
0047498F	B8	DB B8	
00474990	E0	DB E0	
00474991	46	DB 46	CHAR 'F'
00474992	47	DB 47	CHAR 'G'
00474993	00	DB 00	
00474994	E8	DB E8	
00474995	73	DB 73	CHAR 's'
00474996	1E	DB 1E	
00474997	F9	DB F9	
00474998	FF	DB FF	
00474999	8B	DB 8B	
0047499A	1D	DB 1D	
0047499B	B8	DB B8	
0047499C	6F	DB 6F	CHAR 'o'
0047499D	47	DB 47	

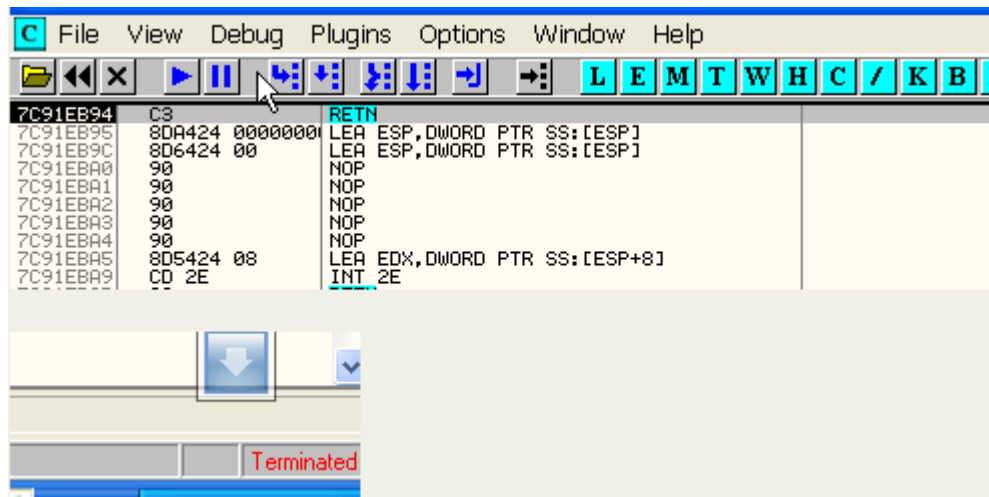
好了,这里的 OD 分析有点问题,OD 将这部分代码当做数据解释了。

Copy to executable	12FFE8	7C816D58	kernel32.7C816D58
Analysis	12FFEC	00000000	
Appearance	12FFFA	00000000	
Analyse code			Ctrl+A
Remove analysis from module			
Scan object files			Ctrl+O

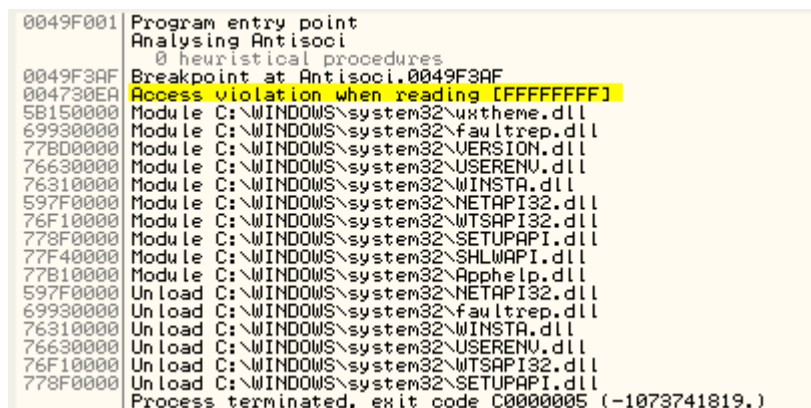
  

474988	55	PUSH EBP	
474989	8BEC	MOV EBP,ESP	
47498B	83C4 F0	ADD ESP,-10	
47498E	53	PUSH EBX	
47498F	B8 E0464700	MOV EAX,Antisoci.004746E0	
474994	E8 731EF9FF	CALL Antisoci.0040680C	
474999	8B1D B86F4700	MOV EBX,DWORD PTR DS:[476FB8]	Antisoci.0047
47499F	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749A1	E8 42F9FFFF	CALL Antisoci.004742E8	
4749A6	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749A8	E8 67F9FFFF	CALL Antisoci.00474314	
4749AD	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749AF	E8 90F9FFFF	CALL Antisoci.00474344	
4749B4	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749B6	E8 8DE4FFFF	CALL Antisoci.00472E48	
4749BB	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749BD	E8 3AE7FFFF	CALL Antisoci.004730FC	
4749C2	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749C4	E8 DFE7FFFF	CALL Antisoci.004731A8	
4749C9	8B03	MOV EAX,DWORD PTR DS:[EBX]	
4749CB	E8 18E7FFFF	CALL Antisoci.004730E8	
4749D0	A1 DC6E4700	MOV EAX,DWORD PTR DS:[476EDC]	
4749D5	8B00	MOV EAX,DWORD PTR DS:[EAX]	
4749D7	E8 6CE7FDFF	CALL Antisoci.00453148	
4749DC	8BCB	MOV ECX,EBX	
4749DE	01 DC6E4700	MOV ECX,DWORD PTR DS:[476EDC]	

这里,我们单击鼠标右键选择 Analysis-Remove analysis from module。现在代码开起来正常了吧,我们运行起来看看会发生什么。

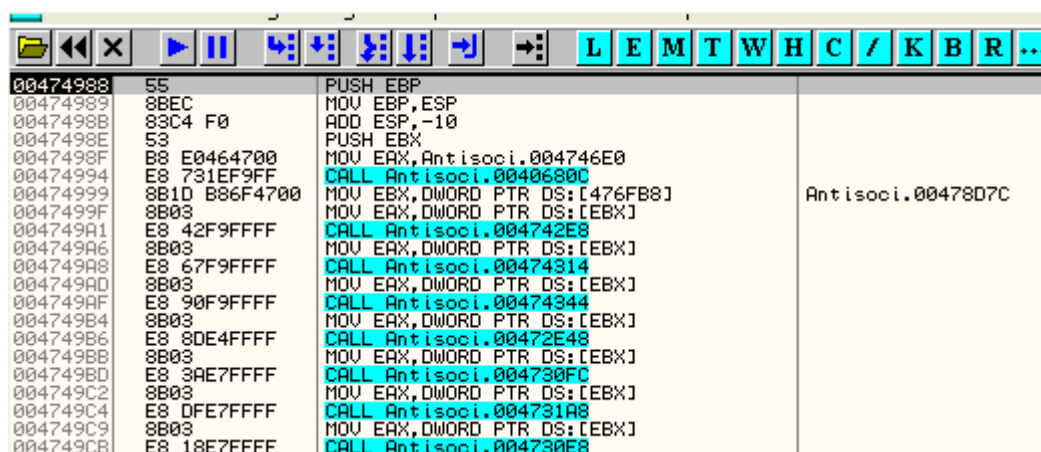


程序终止了。我们查看一下日志信息,可以看到一些有趣的东西。



我们可以看到断在了 POPAD 指令处,然后就发生了异常。

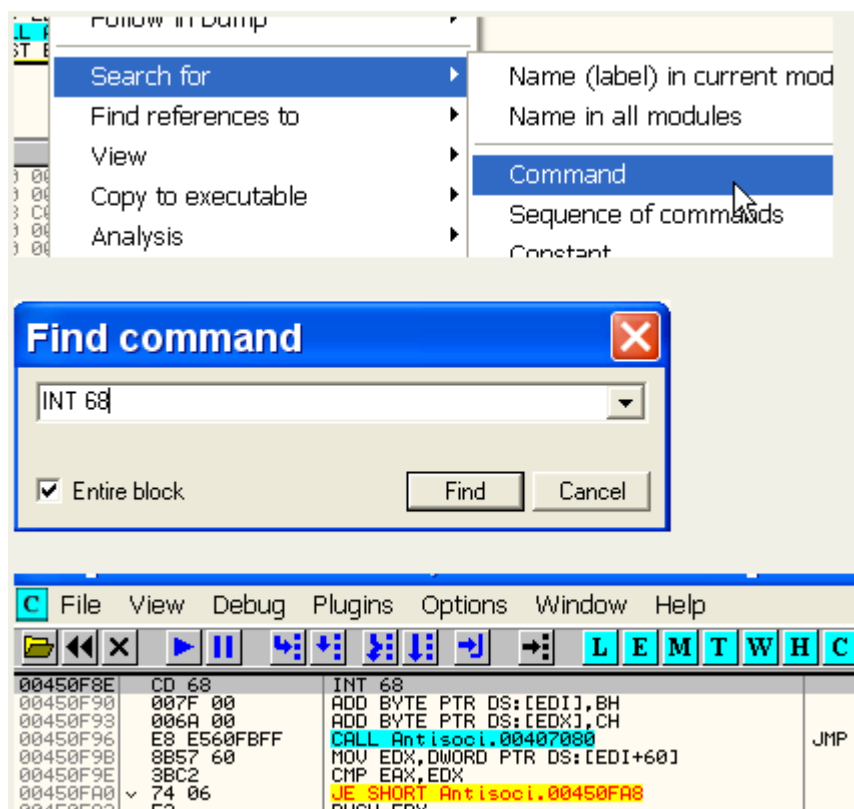
我们重新启动程序,重复前面的步骤再次来到这里。



为了让程序发生异常可以停下来,我们去掉忽略所有异常的选项,第一个选项还是保持勾选。



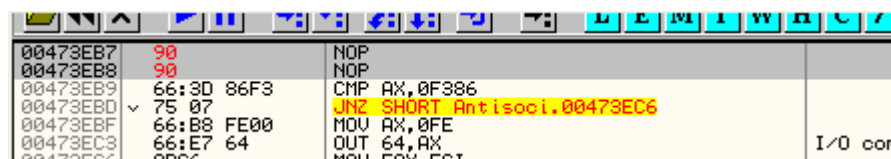
另外,该程序里面可能还存在其他 INT 68 指令会对我们进行干扰,我们直接搜索 INT 68 指令,将其填充为 NOP 即可。



我们可以看到找到了另一个 INT 68 指令,我们直接 NOP 掉。



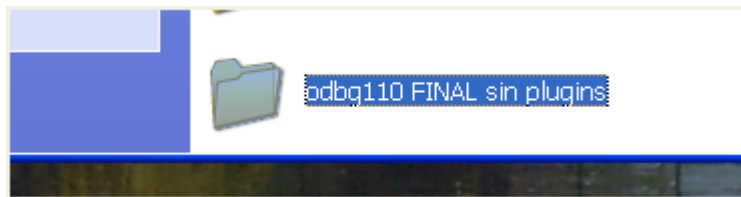
再次搜索,又找到了一个,继续 NOP 掉。



我们继续 CTRL + L 搜索,提示搜索结束,找不到其他的 INT 68 指令了,我们运行起来。

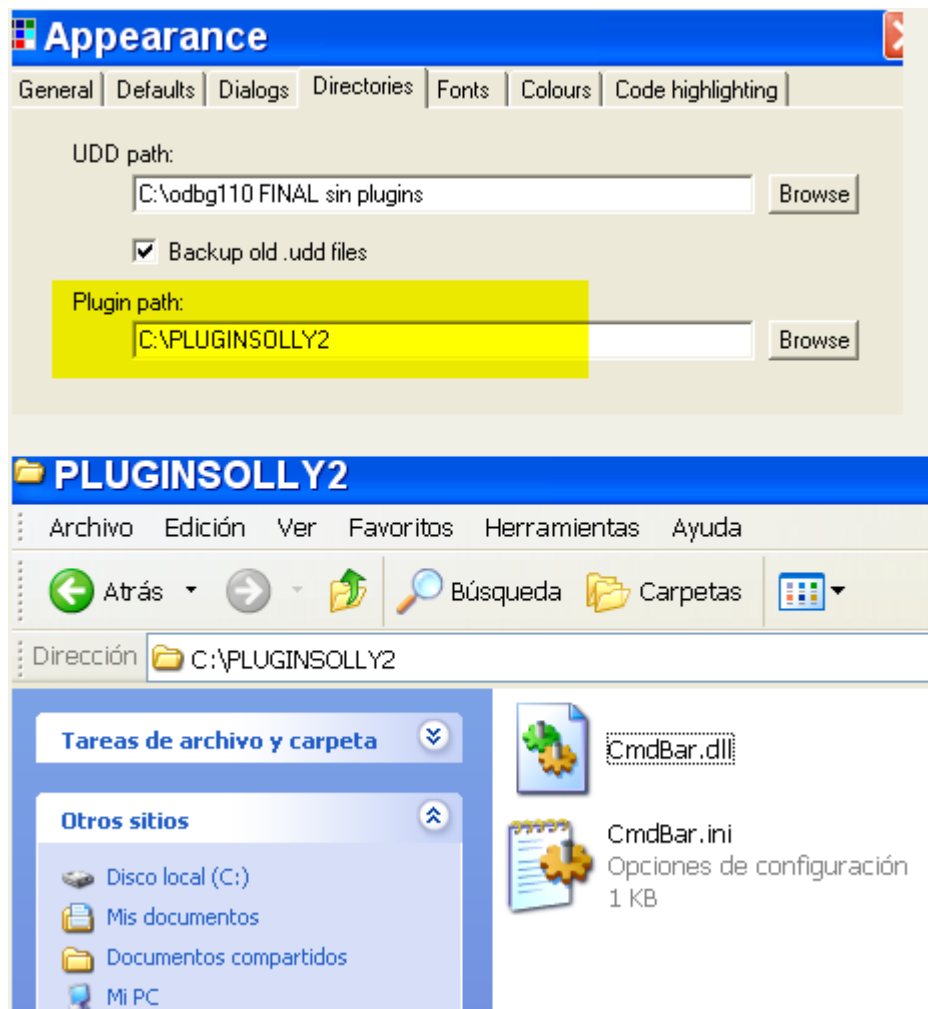


以上是我们使用了带反反调试插件的 OllyDbg 调试的情况,现在我们尝试使用不带插件的原版的 OllyDbg 1.0 来调试,手动来绕过该反调试。

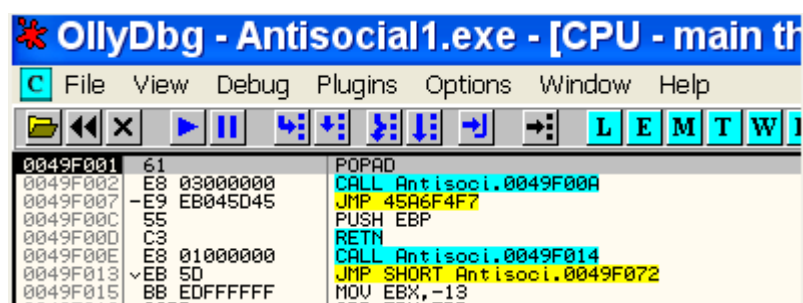


(PS:odbg110 FINAL sin plugins:表示不带插件的 OllyDbg)

我们打开不带反反调试插件的 OllyDbg,由于命令栏插件我们还是需要的,所以我们将插件目录指定为只包含命令栏插件的目录。

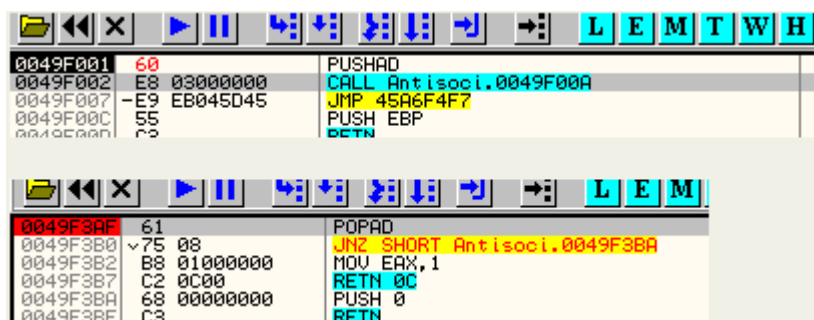


我们可以看到这里面只有一个命令栏插件,我们运行 OD。



还是跟之前一样,将 POPAD 指令替换成 PUSHAD 指令。

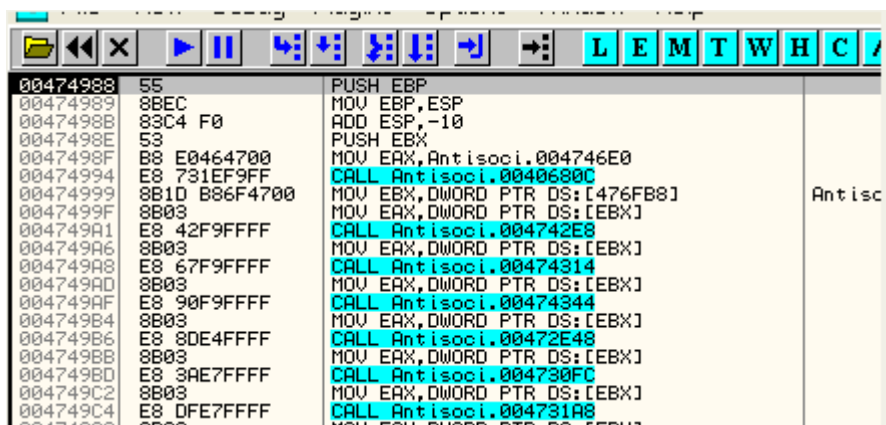




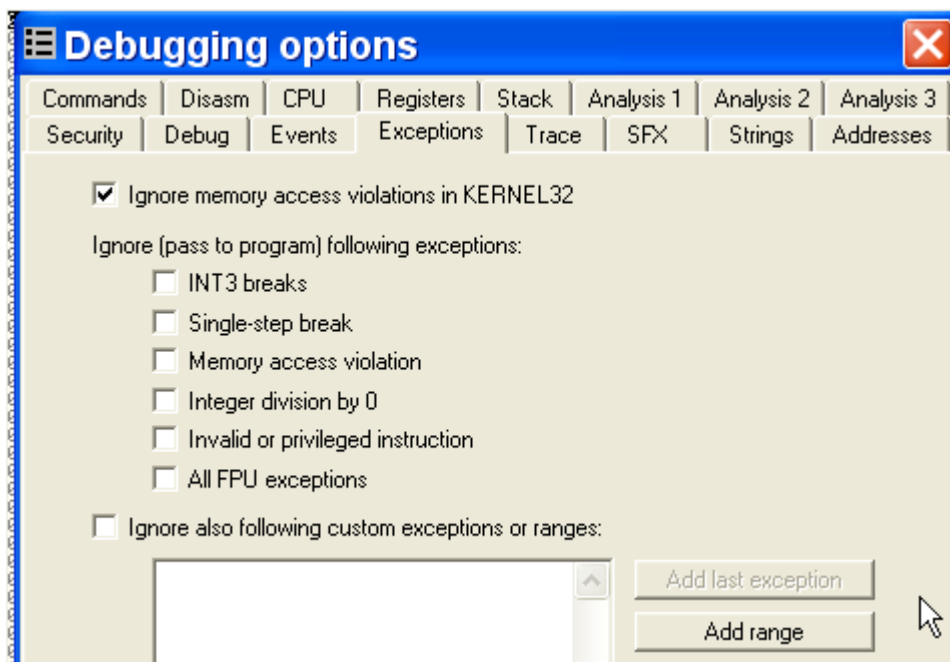
我们还是给后面的 POPAD 指令设置断点,运行起来看看是否会断下来。



我们到了壳解压完毕处。

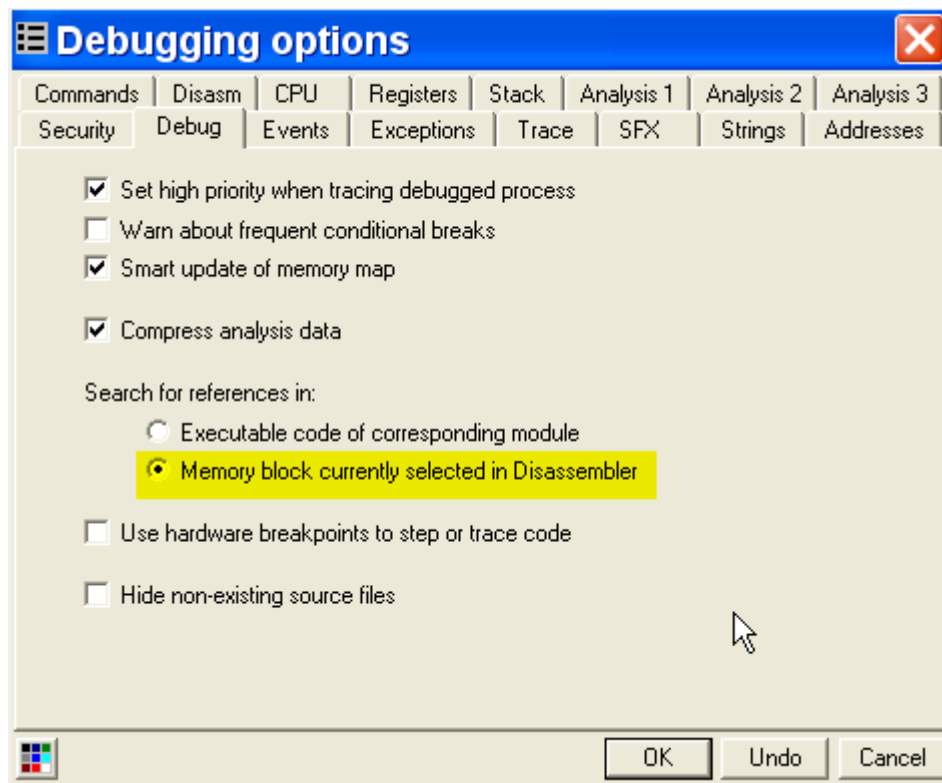


我们清空掉所有的忽略异常的选项,当我们运行起来,遇到 INT 68 指令时,将其 NOP 掉。而并不是使用 SHIFT + F9 忽略异常。

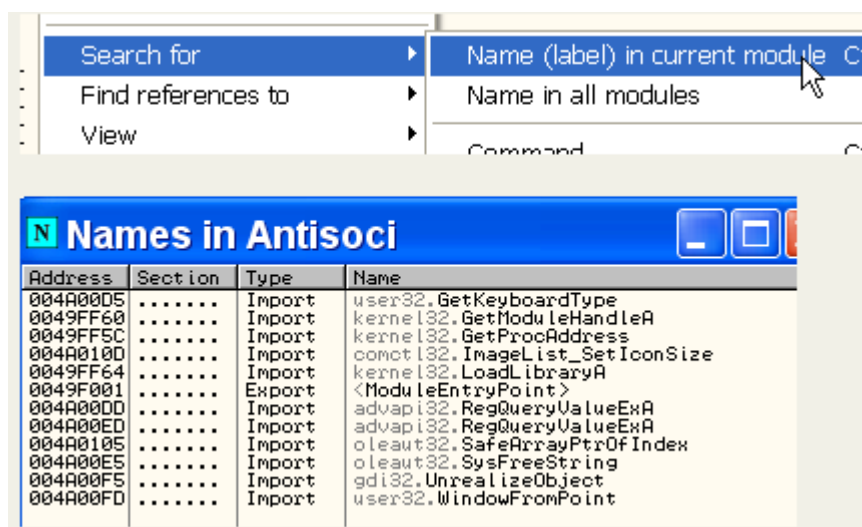


现在来看看程序中使用了哪些 API 函数。

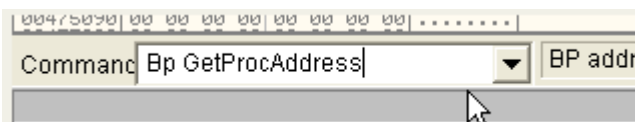
还有一点就是别忘了勾选这个选项:



选择了该选项后,就会显示我们当前所在区段的相关信息。



该列表中显示没有几个 API 函数,并没有看到什么可疑的,但是这里有个 GetProcAddress,该程序可以通过 GetProcAddress 函数来加载其他 API 函数,我们给该函数设置一个断点。



我们运行起来。

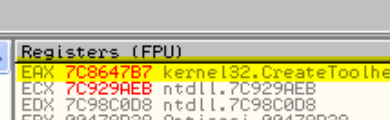
0012FF58	0040B324	CALL to <b>GetProcAddress</b> from <b>Antisoci.0040B31F</b>
0012FF5C	7C800000	hModule = 7C800000 (kernel32)
0012FF60	0040B350	ProcNameOrOrdinal = "GetDiskFreeSpaceExA"
0012FF64	00000000	
0012FF68	0040BFE1	<b>RETURN to Antisoci.0040BFE1 from Antisoci.0040B308</b>
0012FF6C	0012FF80	Pointer to next SEH record
0012FF70	0040BFF4	SE handler
0012FF74	0012FF78	

这个函数没什么可疑的,我们继续运行,直到断在了比较可疑的函数处。

0012FE38	004725E9	CALL to GetProcAddress from Antisoci.004725E4
0012FE2C	7C800000	hModule = 7C800000 (kernel32)
0012FE30	00472724	ProcNameOrOrdinal = "CreateToolhelp32Snapshot"
0012FE34	0000000F	
0012FE38	0047283F	RETURN to Antisoci.0047283F from Antisoci.004725B8
0012FE3C	0012FE78	
0012FE40	00478D7C	Antisoci.00478D7C
0012FE44	00472EA8	RETURN to Antisoci.00472EA8 from Antisoci.00472834

这里我们看到了第一个可疑的 API 函数 `CreateToolhelp32Snapshot`,该函数给当前运行的所有进程创建快照。我们执行到返回,这时,EAX 保存的就是 `CreateToolhelp32Snapshot` 函数的首地址了,我们接着使用 `BPEAX` 命令给该 API 函数设置断点。

7C80AC84	5F	POP EDI
7C80AC85	5B	POP EBX
7C80AC86	C9	LEAVE
7C80AC87	C2 0800	RETN 8
7C80AC8A	83D 10 00	CMP DWORD PTR SS:[EBP+10],0
7C80AC8E	^0F85 81E6FFFF	JNZ kernel!32.7C809815
7C80AC94	33FF	XOR EDI,EDI



The screenshot shows the 'Registers (FPU)' window in Windows Task Manager. The window title is 'Registers (FPU)'. The list of registers and their values is as follows:

Register	Value
EAX	kernel32.CreateToolhelp32Snap
ECX	ntdll.7C929AEB
EDX	ntdll.7C98C0D8
ESP	00478D38
EBP	0012F238
ESI	0012F44
EDI	00000000
EDI	7C920738 ntdll.7C920738
EIP	7C80AC87 kernel32.7C80AC87
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
Q 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7EFDFA00(FFF)

```
00475088|38 39 41 42|43 44 45 46|8YHBLUET|
00475090|00 00 00 00|00 00 40 00|.....@.
```

Command

Address	Disassembly	Comment
7C8647B7	8BFF	MOV EDI,EDI
7C8647B9	55	PUSH EBP
7C8647BA	8BEC	MOV EBP,ESP
7C8647BC	83EC 0C	SUB ESP,0C
7C8647BE	56	PUSH ESI
7C8647C0	8B75 0C	MOV ESI,DWORD PTR SS:[EBP+C]
7C8647C3	85F6	TEST ESI,ESI
7C8647C5	75 07	JNZ SHORT kernel!32.7C8647CE
7C8647C7	F8 8251E9FF	CALL kernel!32.GetCurrentProcessId

别忘了给该 API 函数添加上注释。

Address	Disassembly	Comment
7C8647B7	8BFF	MOV EDI,EDI
7C8647B9	55	PUSH EBP
7C8647BA	8BEC	MOV EBP,ESP
7C8647BC	83EC 0C	SUB ESP,0C
7C8647BF	56	PUSH ESI
7C8647C0	8B75 0C	MOV ESI,DWORD PTR SS:[EBP+C]
7C8647C3	85F6	TEST ESI,ESI
		CREATETOOLHELP32SNAPSHOT

好了,我们继续运行。

0012FE28	00472643	CALL to <b>GetProcAddress</b> from Antisoci.0047263E
0012FE2C	7C800000	hModule = 7C800000 (kernel32)
0012FE30	00472778	ProcNameOrOrdinal = "Toolhelp32ReadProcessMemory"
0012FE34	0000000F	
0012FE38	0047283F	RETURN to Antisoci.0047283F from Antisoci.004725B8
0012FE3C	0012FE78	
0012FE40	00478D7C	Antisoci.00478D7C

这里是 Toolhelp32ReadProcessMemory 这个函数,跟 CreateToolhelp32Snapshot 类似,同上,我们给该函数也设置一个断点。

7C863994	8BFF	MOV EDI,EDI	TOOLHELP
7C863996	55	PUSH EBP	
7C863997	8BEC	MOV EBP,ESP	
7C863999	56	PUSH ESI	
7C86399A	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
7C86399D	6A 00	PUSH 0	

继续运行。

0012FE28	00472655	CALL to <b>GetProcAddress</b> from Antisoci.00472650
0012FE2C	7C800000	hModule = 7C800000 (kernel32)
0012FE30	00472794	ProcNameOrOrdinal = "Process32First"
0012FE34	0000000F	
0012FE38	0047283F	RETURN to Antisoci.0047283F from Antisoci.00472
0012FE3C	0012FE78	
0012FE40	00478D7C	Antisoci.00478D7C

给 Process32First 这个函数也设置一个断点。

7C863A8D	8BFF	MOV EDI,EDI	FIRST
7C863A8F	55	PUSH EBP	
7C863A90	8BEC	MOV EBP,ESP	
7C863A92	81EC 30020000	SUB ESP,230	
7C863A98	A1 CC36887C	MOV EAX,DWORD PTR DS:[7C8836CC]	

0012FE28	00472667	CALL to <b>GetProcAddress</b> from Antisoci.00472662
0012FE2C	7C800000	hModule = 7C800000 (kernel32)
0012FE30	004727A4	ProcNameOrOrdinal = "Process32Next"
0012FE34	0000000F	
0012FE38	0047283F	RETURN to Antisoci.0047283F from Antisoci.004725B8
0012FE3C	0012FE78	

同上。

7C863C00	8BFF	MOV EDI,EDI	NEXT
7C863C02	55	PUSH EBP	
7C863C03	8BEC	MOV EBP,ESP	
7C863C05	81EC 30020000	SUB ESP,230	
7C863C08	A1 CC36887C	MOV EAX,DWORD PTR DS:[7C8836CC]	

0012FE28	00472679	CALL to <b>GetProcAddress</b> from Antisoci.00472674
0012FE2C	7C800000	hModule = 7C800000 (kernel32)
0012FE30	004727B4	ProcNameOrOrdinal = "Process32FirstW"
0012FE34	0000000F	
0012FE38	0047283F	RETURN to Antisoci.0047283F from Antisoci.004725B8
0012FE3C	0012FE78	

0012FE28	0047268B	CALL to <b>GetProcAddress</b> from Antisoci.00472686
0012FE2C	7C800000	hModule = 7C800000 (kernel32)
0012FE30	004727C4	ProcNameOrOrdinal = "Process32NextW"
0012FE34	0000000F	
0012FE38	0047283F	RETURN to Antisoci.0047283F from Antisoci.004725B8

0012FE30	0047284B	CALL to <b>CreateToolhelp32Snapshot</b> from Antisoci.00472844
0012FE34	0000000F	Flags = TH32CS_SNAPALL
0012FE38	00000000	ProcessID = 0
0012FE3C	0012FE78	
0012FE40	00478D7C	Antisoci.00478D7C

现在断在了创建进程快照处,我们知道该处的检测 OD 是基于这个进程快照的,该进程快照包含了进程列表中所有进程的相关信息,我们可以尝试 patch 这个函数,让该函数返回的快照句柄为空,现在我们来到 CreateToolhelp32Snapshot 这个函数的返回处。

7C86482D	83C8 FF	OR EAX,FFFFFFFF
7C864830	EB 03	JMP SHORT kernel32.7C864835
7C864832	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]
7C864835	5E	POP ESI
7C864836	C9	LEAVE
7C864837	C2 0800	RETN 8
7C86483A	90	NOP
7C86483B	90	NOP
7C86483C	90	NOP
7C86483D	90	NOP
7C86483E	90	NOP
7C86483F	8BFF	MOV EDI,EDI
7C864841	55	PUSH EBP

我们可以看到这里有一些空余的空间,我们可以在返回之前将 EAX 赋值为零。

7C86482D	83C8 FF	OR EAX,FFFFFFFF
7C864830	EB 03	JMP SHORT kernel32.7C864835
7C864832	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]
7C864835	5E	POP ESI
7C864836	C9	LEAVE
7C864837	33C0	XOR EAX,EAX
7C864839	C2 0800	RETN 8
7C86483C	90	NOP
7C86483D	90	NOP
7C86483E	90	NOP
7C86483F	8BFF	MOV EDI,EDI

这样 CreateToolhelp32Snapshot 这个函数返回的快照句柄就为空了,该程序就不能通过进程快照来检测 OD 了。当然我们还有另一种方式-修改主程序中的代码,而并不修改 CreateToolhelp32Snapshot 的实现代码。

7C864830	EB 03	JMP SHORT kernel32.7C864835
7C864832	8B45 0C	MOV EAX,DWORD PTR SS:[EBP+C]
7C864835	5E	POP ESI
7C864836	C9	LEAVE
7C864837	C2 0800	RETN 8
7C86483A	90	NOP
7C86483B	90	NOP
7C86483C	90	NOP
7C86483D	90	NOP
7C86483E	90	NOP

按 F8 键单步跟踪,我们可以看到这里有个 JNZ 条件跳转指令会跳转至 TerminateProcess 处结束掉 OD 进程。OD 的进程句柄在之前通过 OpenProcess 获取。我们可以看到这里有五个分支判断。

00472EF7	75 15	JNZ SHORT jejebuen.00472F0E
00472EF9	6A 00	PUSH 0
00472EFB	8B46 08	MOV EAX,DWORD PTR DS:[ESI+8]
00472EFE	50	PUSH EAX
00472EFF	6A FF	PUSH -1
00472F01	6A 01	PUSH 1
00472F03	E8 883BF9FF	CALL <JMP.&kernel32.OpenProcess>
00472F08	50	PUSH EAX
00472F09	E8 EA3BF9FF	CALL <JMP.&kernel32.TerminateProcess>
00472F0E	8D95 C8FEFFFI	LEA EDX,DWORD PTR SS:[EBP-138]
00472F14	B8 84304700	MOV EAX,jejebuen.00473084
00472F19	E8 86FEFFFF	CALL jejebuen.00472DA4
00472F1E	8B95 C8FEFFFI	MOV EDX,DWORD PTR SS:[EBP-138]
00472F24	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
00472F27	E8 C418F9FF	CALL jejebuen.004047F0
00472F2C	75 15	JNZ SHORT jejebuen.00472F43
00472F2E	6A 00	PUSH 0
00472F30	8B46 08	MOV EAX,DWORD PTR DS:[ESI+8]
00472F33	50	PUSH EAX
00472F34	6A FF	PUSH -1
00472F36	6A 01	PUSH 1
00472F38	E8 533BF9FF	CALL <JMP.&kernel32.OpenProcess>
00472F3D	50	PUSH EAX
00472F3E	E8 B53BF9FF	CALL <JMP.&kernel32.TerminateProcess>
00472F43	8D95 C4FEFFFI	LEA EDX,DWORD PTR SS:[EBP-13C]
00472F49	B8 98304700	MOV EAX,jejebuen.00473098
00472F4E	E8 51FEFFFF	CALL jejebuen.00472DA4
00472F53	8B95 C4FEFFFI	MOV EDX,DWORD PTR SS:[EBP-13C]
00472F59	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
00472F5C	E8 8F18F9FF	CALL jejebuen.004047F0
00472F61	75 15	JNZ SHORT jejebuen.00472F78
00472F63	6A 00	PUSH 0
00472F65	8B46 08	MOV EAX,DWORD PTR DS:[ESI+8]
00472F68	50	PUSH EAX
00472F69	6A FF	PUSH -1
00472F6B	6A 01	PUSH 1
00472F6D	E8 1E3BF9FF	CALL <JMP.&kernel32.OpenProcess>
00472F72	50	PUSH EAX
00472F73	E8 803BF9FF	CALL <JMP.&kernel32.TerminateProcess>
00472F78	8D95 C0FEFFFI	LEA EDX,DWORD PTR SS:[EBP-140]
00472F7E	B8 AC304700	MOV EAX,jejebuen.004730AC
00472F83	E8 1CFEFFFF	CALL jejebuen.00472DA4
00472F88	8B95 C0FEFFFI	MOV EDX,DWORD PTR SS:[EBP-140]
00472F8E	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]

这里我们将这几处 JNZ 指令修改为 JMP 指令,这样就可以避免程序执行 TerminateProcess 结束掉 OD 进程。

这样反调试的第一个部分我们就搞定了,现在我们来解决反调试的第二个部分,我们运行起来。

004730EA	CD 68	INT 68
004730EC	66:3D 86F3	CMP AX,0F386
004730F0	75 07	JNZ SHORT Antisoci.004730F9
004730F2	66:B8 FE00	MOV AX,0FE
004730F6	66:E7 64	OUT 64,AX
004730F9	C3	RETN

我们停在了 INT 68 指令处,我们将该 INT 68 指令用 NOP 指令填充掉,然后运行起来会发生程序终止了。

如果我们加载 HideDebugger1.23f 插件,并勾选上 FindWindows/EnumWindows 选项,然后重复之前的步骤会发现运行的很正常。

刚刚导致 OD 终止的地方是这里。

004732EB	. 8B95 DCFEFFFF	MOV EDX,DWORD PTR SS:[EBP-124]
004732F1	. 58	POP EAX
004732F2	. E8 F914F9FF	CALL jejbuen.004047F0
004732F7	. 75 0C	JNZ SHORT jejbuen.00473305
004732F9	. A1 DC6E4700	MOV EAX,DWORD PTR DS:[476EDC]
004732FE	. 8B00	MOV EAX,DWORD PTR DS:[EAX]
00473300	. E8 C7FFDFFF	CALL jejbuen.004532CC
00473305	. 8D85 D4FEFFFF	LEA EAX,DWORD PTR SS:[EBP-12C]
0047330B	. 8B06	MOV EDI,ESI
0047330D	. B9 00010000	MOV ECX,100

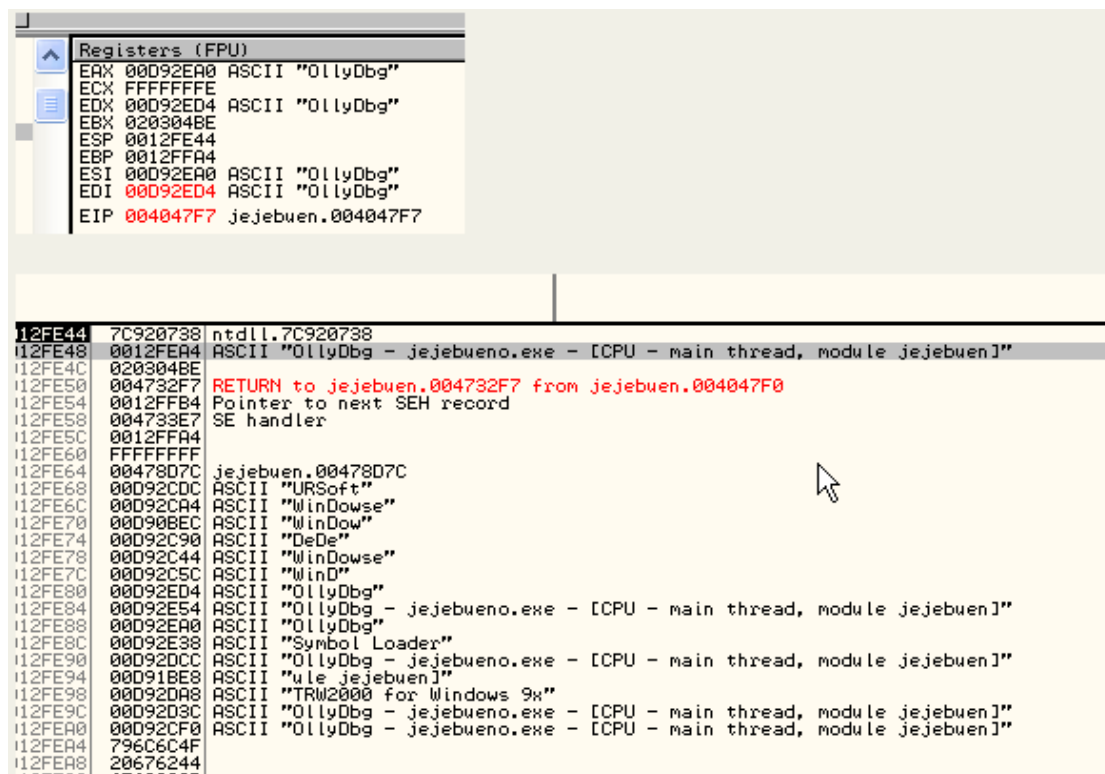
我们跟进都这个 CALL 4047F0 里面。

004047F0	. 53	PUSH EBX
004047F1	. 56	PUSH ESI
004047F2	. 57	PUSH EDI
004047F3	. 89C6	MOV ESI,EAX
004047F5	. 89D7	MOV EDI,EDX
004047F7	. 39D0	CMP EAX,EDX
004047F9	. 0F84 8F000000	JE jejbuen.0040488E
004047FF	. 85F6	TEST ESI,ESI
00404801	. 74 C0	JE SHORT jejbuen.004048C0

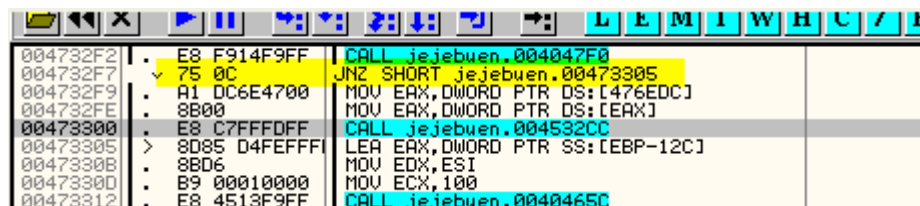
  

Registers (FPU)	
EAX	00D923F8
ECX	FFFFFFFF
EDX	00D9242C ASCII "OlllyDbg"
EBX	01680468
ESP	0012FE44
EBP	0012FFA4
ESI	00D923F8
EDI	00D9242C ASCII "OlllyDbg"
EIP	004047F7 jejbuen.004047F7
C 0	ES 0023 32bit 0(FFFFFFFF)
P 0	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
7 2	DC 0023 32bit 0(FFFFFFFF)

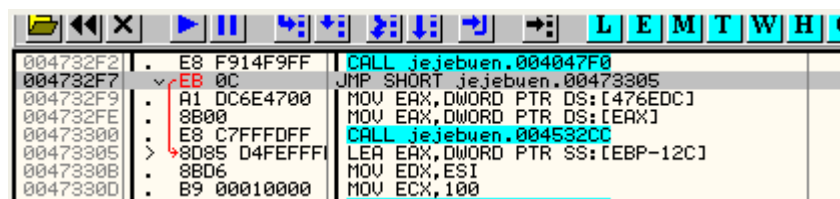
这里,我们可以看到比较指令。



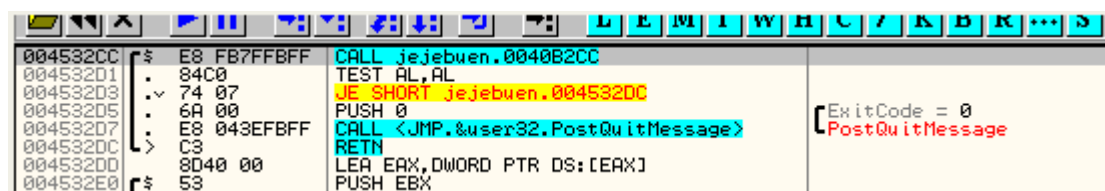
这里当比较出是 OllyDbg 的话,就会终止掉当前进程。这里我们为了避免 JNZ 条件跳转指令不成立进而去调用下面的 CALL 4532CC 结束掉进程,我们将该 JNZ 指令修改为 JMP 指令。



这里将该关键跳转修改为 JMP 指令。



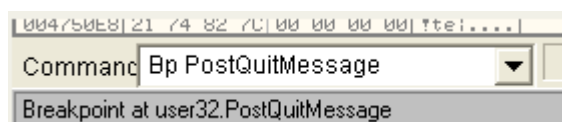
我们再来看看我们绕过的第二个 CALL 指令里面具体是怎么结束进程的。



该 CALL 指令里面是调用 PostQuitMessage 退出主线程。

好了,现在你可能会问了,你是怎么定位上面那个 JNZ 473305 关键跳转的呢?很简单。

我们按照之前的步骤干掉了该反调试的第一部分以后,就可以给 PostQuitMessage 这个 API 函数设置一个断点。





Address	Disassembly	Comment
77021211	8BFF	MOV EDI,EDI
77021213	55	PUSH EBP
77021214	8BEC	MOV EBP,ESP
77021216	6A 33	PUSH 33
77021218	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7702121B	E8 8672FFFF	CALL user32.770184A6
77021220	5D	POP EBP
77021221	C2 0400	RETN 4
77021224	33C0	XOR EAX,EAX
77021226	40	INC EAX
77021227	E9 EA230000	JMP user32.77023616
7702122C	33C0	XOR EAX,EAX
7702122E	E9 7DC5FFFF	JMP user32.7701D7B0
77021233	90	NOP
77021234	90	NOP
77021235	90	NOP

断在了该 API 函数处,我们看下堆栈,看看该调用来至哪里。

Address	Disassembly	Comment
0012FE48	004532DC	CALL to PostQuitMessage from jejebuen.004532D7
0012FE4C	00000000	ExitCode = 0
0012FE50	00473305	RETURN to jejebuen.00473305 from jejebuen.004532CC
0012FE54	0012FFB4	Pointer to next SEH record
0012FE58	004733E7	SE handler
0012FE5C	0012FFA4	
0012FE60	FFFFFFFF	
0012FE64	00478D7C	jejebuen.00478D7C
0012FE68	00090BEC	ASCII "URSoft"
0012FE6C	00092C78	ASCII "WinDose"
0012FE70	00092C98	ASCII "WinDose"

堆栈中信息显示该调用来至于 4532D7 处,我们定位到该地址。

Address	Disassembly	Comment
004532C9	5D	POP EBP
004532CA	C3	RETN
004532CB	90	NOP
004532CC	E8 FB7FFBFF	CALL jejebuen.0040B2CC
004532D1	84C0	TEST AL,AL
004532D3	74 07	JE SHORT jejebuen.004532DC
004532D5	6A 00	PUSH 0
004532D7	E8 043EFBFF	CALL <JMP.&user32.PostQuitMessage>
004532DC	C3	RETN
004532DD	8D40 00	LEA EAX,DWORD PTR DS:[EAX]
004532E0	53	PUSH EBX
004532E1	56	PUSH ESI
004532E2	57	MOV ESI,EBX

这里说明前面的 JE 指令跳转没有发生,进而调用 PostQuitMessage,这里如果我们继续执行 PostQuitMessage 话,程序就退出了,所以这里我们直接在堆栈中查看返回地址是多少。

Address	Disassembly	Comment
0012FE48	004532DC	CALL to PostQuitMessage from jejebuen.004532D7
0012FE4C	00000000	ExitCode = 0
0012FE50	00473305	RETURN to jejebuen.00473305 from jejebuen.004532CC
0012FE54	0012FFB4	Pointer to next SEH record
0012FE58	004733E7	SE handler
0012FE5C	0012FFA4	
0012FE60	FFFFFFFF	
0012FE64	00478D7C	jejebuen.00478D7C
0012FE68	00090BEC	ASCII "URSoft"
0012FE6C	00092C78	ASCII "WinDose"
0012FE70	00092C98	ASCII "WinDose"

这里我们可以看到返回地址为 473305。

Address	Disassembly	Comment
004732F1	58	PUP EAX
004732F2	E8 F914F9FF	CALL jejebuen.004047F0
004732F7	75 0C	JNZ SHORT jejebuen.00473305
004732F9	A1 DC6E4700	MOV EAX,DWORD PTR DS:[476EDC]
004732FE	8B00	MOV EAX,DWORD PTR DS:[EAX]
00473300	E8 C7FFFDFF	CALL jejebuen.004532CC
00473305	8D85 04FEFF	LEA EAX,DWORD PTR SS:[EBP-12C]
00473308	8B06	MOV EDX,ESI
0047330D	B9 00010000	MOV ECX,100
00473312	FA 4513F9FF	CALL jejebuen.0040465C

好了,这里我们就比较熟悉了,我们可以将 4732F7 处的 JNZ 指令修改为 JMP 指令,避免让该程序执行下面的 CALL 中 PostQuitMessage 结束掉进程。这样我们就手工的解决这个反调试,并没有借助反反调试插件。



