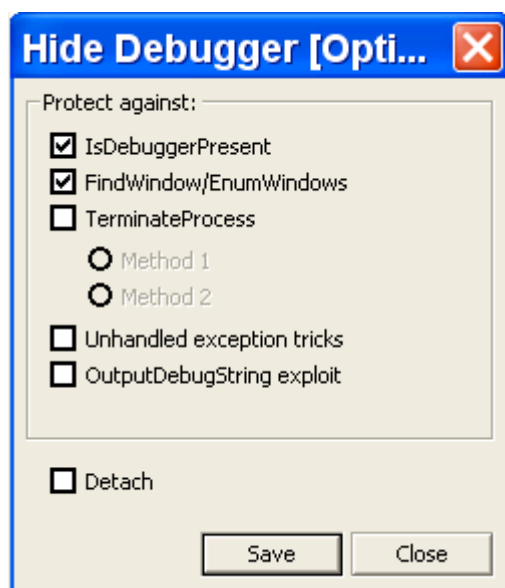


第二十二章-OllyDbg 反调试之 UnhandledExceptionFilter,ZwQueryInformationProcess

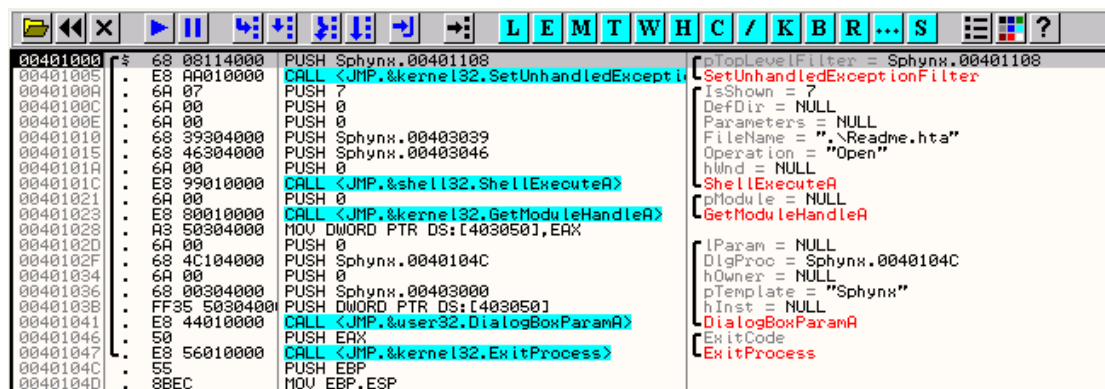
本章我们继续讨论反调试技术,我们将介绍反调试的另外两个小技巧,由于其中一个可以配合另一个来使用,所以我们两个一起介绍。本章我们使用上一章打过补丁的 OllyDbg,也就是 Nvp11,其中 HideDebugger 插件的配置如下:



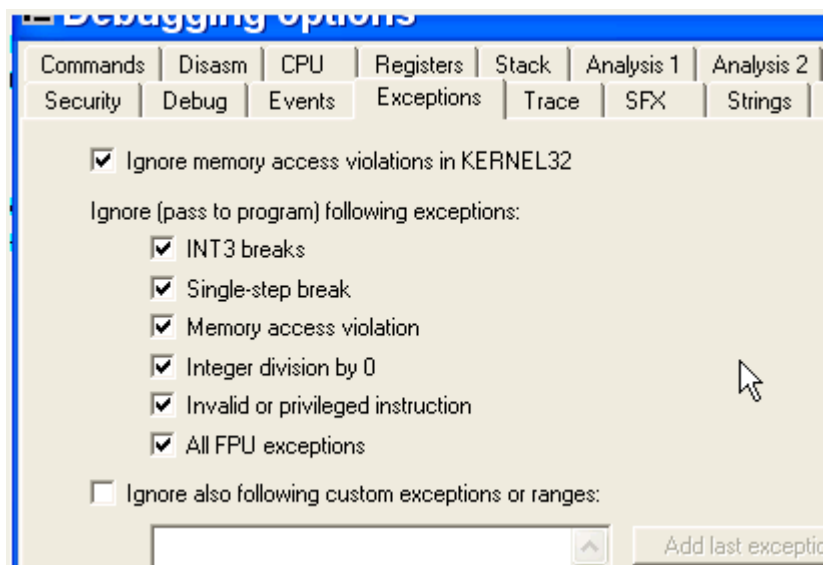
这里我们可以看到其中有一个 Unhandled exception tricks 的选项。接下来我们将学习 Unhandled exception 和另一个 API 函数 ZwQueryInformationProcess 检测调试器的工作原理。

这里我们实验的对象叫做 sphynx。如果你勾选上了 HideDebugger 插件的 Unhandled exception tricks 就可以正常运行起来了,但是使用插件之前我们还是来介绍一下它的实现原理。

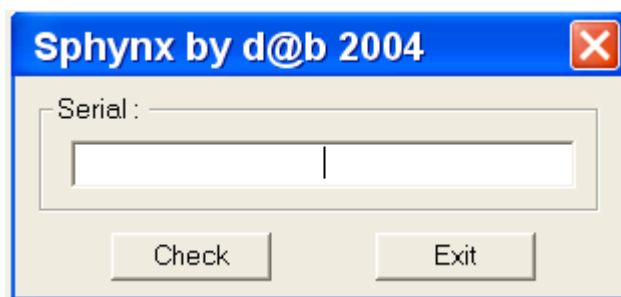
另外说一点,我们现在暂时不解决这个 CrackMe,我们只是来看看该 CrackMe 是如何检测 OD 的。



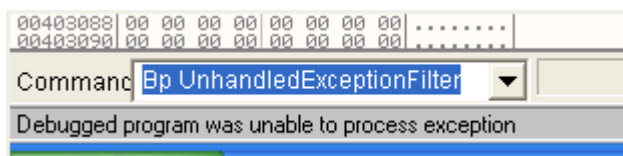
我们使用 Nvp11(打过补丁的 OllyDbg)加载该 CrackMe,并且确保 HideDebugger1.23 插件的配置如第一幅图所示,接着将 Debugging options-Exceptions 选项中忽略的异常选项全部勾选上。



运行起来。



出现了 CrackMe 的主窗口,那么反调试体现在哪里呢?我们随便输入一个错误的序列号然后点击 Check 按钮。

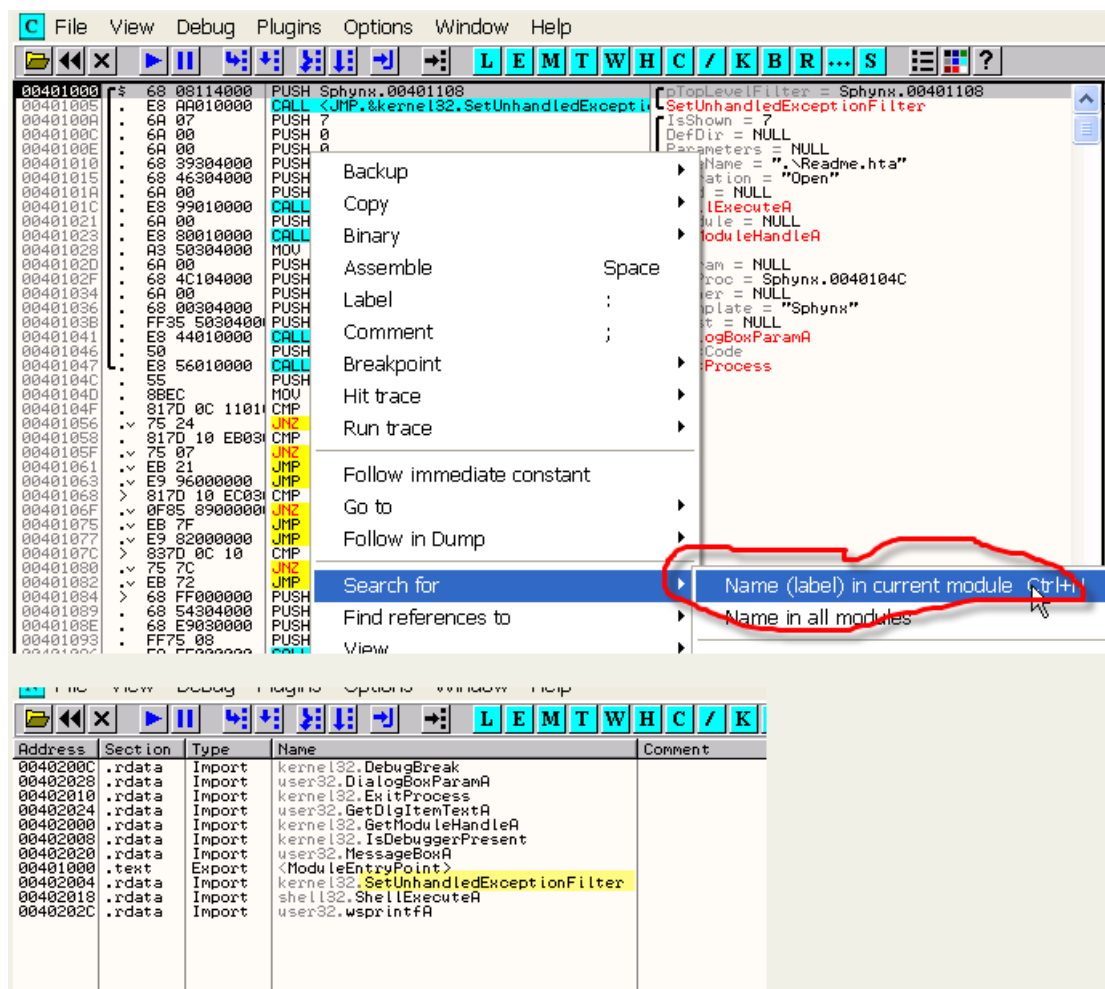


OD 的左下方提示存在不可处理的异常,程序将关闭,我们继续运行。

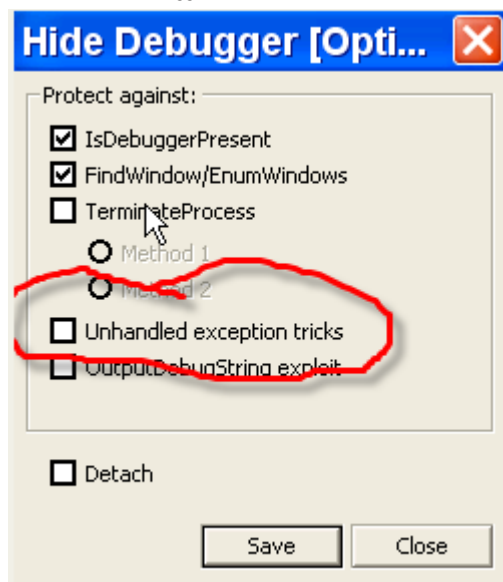


好了,我们不用 OLLYDBG 加载该 CrackMe,它是不会关闭的,我们可以尝试输入不同的序列号,同样也不会关闭。

好了,现在我们重新启动该 CrackMe,看看其使用了哪些 API 函数。



我们记得 HideDebugger 插件中有绕过该反调试的选项。



我们设置该选项前先来学习一下如何手工绕过该反调试以及其原理。

我们先来看看 MSDN 中关于 SetUnhandledExceptionFilter 的说明。

SetUnhandledExceptionFilter Quick Info

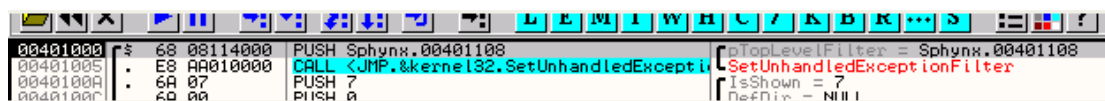
The **SetUnhandledExceptionFilter** function lets an application supersede the top-level exception handler that Win32 places at the top of each thread and process.

After calling this function, if an exception occurs in a process that is not being debugged, and the exception makes it to the Win32 unhandled exception filter, that filter will call the exception filter function specified by the *lpTopLevelExceptionFilter* parameter.

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(  
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter    // exception filter function  
);
```

该函数的唯一一个参数为异常处理函数指针。当程序发生异常是,且程序不处于调试模式(在 VS 或者其他调试器里运行)则首先调用该异常处理函数。因此,程序可以主动抛出一个异常来判断当前程序是否正在被调试,嘿嘿,这里我们并不需要使用 ZwQueryInformationProcess。

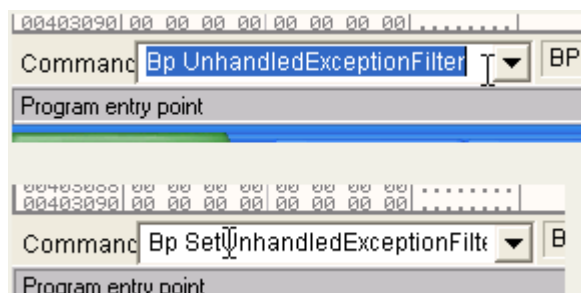
我们回到 OD 中,看到 SetUnhandledExceptionFilter 的调用处。



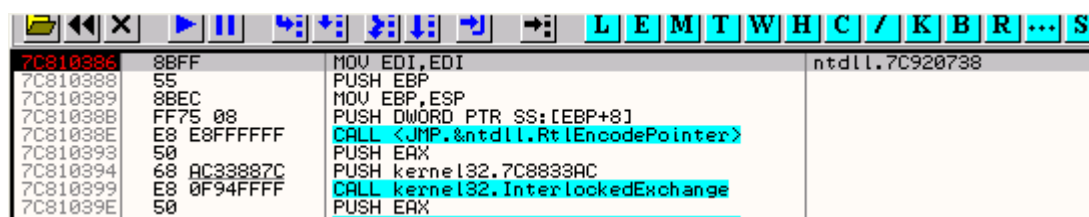
正如你所看到的,只有一个参数,在程序执行过程中会抛出一个异常,如果当前程序没有被调试,那么就会调用该参数指定的异常处理函数,嘿嘿。这里该异常处理函数入口地址为 401108,如果当前程序正在被调试的话,程序最终将终止运行。

这是我们要看到的该程序安装的其中一个异常处理函数,当有异常发生并且当前程序没有被调试的情况下,该异常处理函数将得以执行。

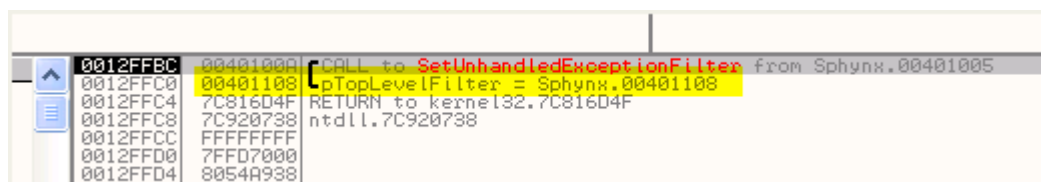
好了,我们现在给 SetUnhandledExceptionFilter,UnhandledExceptionFilter 这两个函数设置断点。



我们运行起来。



断在了 SetUnhandledExceptionFilter 的入口处。我们看下堆栈的情况。



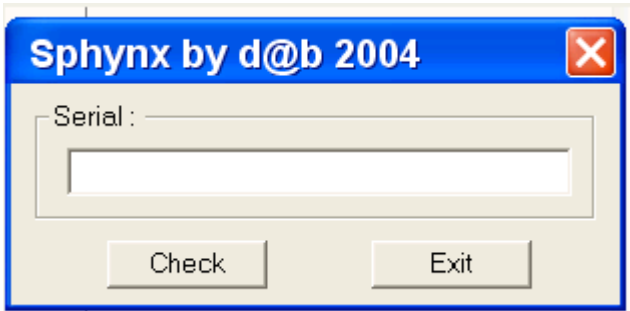
正如你所看到的异常处理函数入口地址为 401108,我们在命令栏中输入 BP 401108 给该函数设置断点。



我们运行起来,可以看到又断在 SetUnhandledExceptionFilter 的入口处,这个调用来至一个 shellext.dll。

```
0012D2AC 00E7D4B7 CALL to SetUnhandledExceptionFilter from shellext.00E7D4B1
0012D2B0 00E7D454 pTopLevelFilter = shellext.00E7D454
0012D2B4 00E7B539 RETURN to shellext.00E7B539
0012D2B8 7C80AA49 kernel32.GetProcessHeap
0012D2BC 00150680
0012D2C0 00E7D295 RETURN to shellext.00E7D295 from shellext.00E7B50F
```

我们对这处调用不感兴趣,异常处理函数前面已经设置过了,所以我们将对 SetUnhandledExceptionFilter 设置的断点删除掉。



现在,我们随便输入一个错误的序列号,然后单击 Check 按钮,将会断在系统默认异常处理函数入口处,因为程序有异常发生,并且当前程序正在被调试,所以并不会首先调用程序之前设置的入口为 401108 的异常处理函数,而异常转交给调试器处理了,而调试器也无法处理该异常,所以最终调用系统默认的异常处理函数 UnhandledExceptionFilter 来处理,嘿嘿。

```
7C862B8A 68 48020000 PUSH 248
7C862B8F 68 E035867C PUSH kernel32.7C8635E0
7C862B94 E8 32F9F9FF CALL kernel32.7C8024CB
7C862B99 A1 CC36887C MOV EAX,DWORD PTR DS:[7C8836CC]
7C862BAE 8945 E4 MOV DWORD PTR SS:[EBP-1C],EAX
7C862BA1 8B5D 08 MOV EBX,DWORD PTR SS:[EBP+8]
7C862BA1 8B5D 08 MOV EBX,DWORD PTR SS:[EBP+8]

0012F70C 7C843622 CALL to UnhandledExceptionFilter from kernel32.7C84361D
0012F710 0012F730 pExceptionInfo = 0012F730
0012F714 7C839A54 RETURN to kernel32.7C839A54
0012F718 0012F738
0012F71C 00000000
```

这个 API 函数用来检测当前程序是否正在被调试,我们 F8 键单步看看该函数是如何实现检测调试器的。

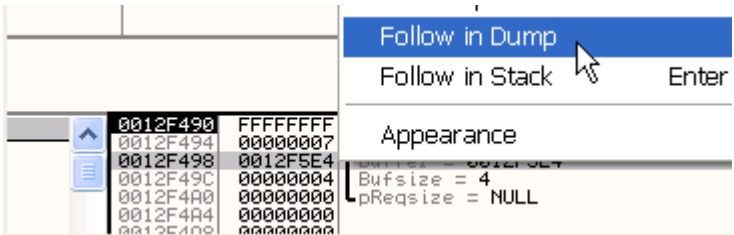
```
7C862C09 6A 07 PUSH 7
7C862C0B E8 FDB3FAFF CALL kernel32.GetCurrentProcess
7C862C10 50 PUSH EAX
7C862C11 FF15 AC10807C CALL DWORD PTR DS:[<&ntdll.NtQueryInformationProcess
7C862C17 85C0 TEST EAX,EAX
7C862C19 0F8C A2000000 JL kernel32.7C862CC1
7C862C1F 39BD DCFEFFFF CMP DWORD PTR SS:[EBP-124],EDI
```

这里是我们今天要介绍的第二个反调试知识点,这个函数也可以单独用来检测调试器只需要把 InfoClass 设置为 7。

```
0012F490 FFFFFFFF hProcess = FFFFFFFF
0012F494 00000007 InfoClass = 7
0012F498 0012F5E4 Buffer = 0012F5E4
0012F49C 00000004 Bufsize = 4
0012F4A0 00000000 pReqsize = NULL
0012F4A4 00000000
```

该函数通过将 InfoClass 参数设置为 7,将可以获取到当前进程是否被调试的信息,该信息将保存在 Buffer 参数指向的缓冲区中。

我们在数据窗口中定位到给缓冲区。



Address	Hex dump	ASCII
0012F5E4	00 00 00 00 18 9D 95 7C	...↑00!
0012F5EC	00 00 39 00 61 01 00 50	..9.a0.P
0012F5F4	86 B6 92 7C 00 00 00 00	3A!....
0012F5FC	10 02 00 00 00 00 39 00	!0....9.
0012F604	2A 0A 1C 00 7F 00 00 00	*.L.Δ...
0012F60C	01 00 00 00 00 00 00 00	0.....
0012F614	50 F6 12 00 B3 40 D2 77	P÷\$. 0Ew
0012F61C	68 80 71 00 00 00 00 00	hCq.....
0012F624	25 6C EF 77 0F 6C EF 77	%l'w*!l'w
0012F62C	1C FC 12 00 0D 00 00 00	L?÷.....
0012F634	B4 FA 12 00 1C FC 12 00	+÷.L?÷.
0012F63C	00 00 00 00 1C 00 00 00L....
0012F644	01 00 00 00 0D 00 00 00	0.....

我们可以看到该缓冲区的大小为4个字节,如果该缓冲区返回的是 FFFFFFFF 的话表示当前程序正在被调试,如果返回的是 0 的话,表示当前程序没有被调试,我们按 F8 键单步执行该函数,看看缓冲区中返回的是什么。

Address	Hex dump	ASCII
0012F5E4	FF FF FF FF 18 9D 95 7C	↑00!
0012F5EC	00 00 39 00 61 01 00 50	..9.a0.P
0012F5F4	86 B6 92 7C 00 00 00 00	3A!....
0012F5FC	10 02 00 00 00 00 39 00	!0....9.
0012F604	2A 0A 1C 00 7F 00 00 00	*.L.Δ...
0012F60C	01 00 00 00 00 00 00 00	0.....

我们可以看到返回值为 FFFFFFFF,表示当前程序正在被调试,嘿嘿。

7C862C17	85C0	TEST EAX,EAX
7C862C19	0F8C A2000000	JL kernel32.7C862CC1
7C862C1F	39BD DCFEFFFF	CMP DWORD PTR SS:[EBP-124],EDI
7C862C25	0F84 96000000	JE kernel32.7C862CC1
7C862C2B	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]
7C862C31	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]

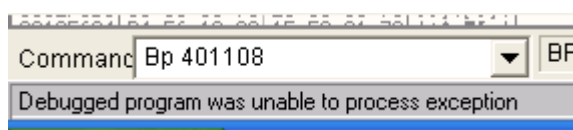
可以看到现在判断该值是否为零,当前 EDI 为零。

7C862C79	68 02000020	PUSH
7C862C7E	F8 F8C80100	CALL
EDI=00000000		
Stack SS:[0012F5E4]=FFFFFFFF		

这里,JE 条件跳转将不会发生。

7C862C17	85C0	TEST EAX,EAX	
7C862C19	0F8C A2000000	JL kernel32.7C862CC1	
7C862C1F	39BD DCFEFFFF	CMP DWORD PTR SS:[EBP-124],EDI	
7C862C25	0F84 96000000	JE kernel32.7C862CC1	
7C862C2B	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
7C862C31	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]	
7C862C34	F640 69 01	TEST BYTE PTR DS:[EAX+69],1	
7C862C38	0F84 F1070000	JE kernel32.7C86342F	
7C862C3E	8B03	MOV EAX,DWORD PTR DS:[EBX]	
7C862C40	3930	CMP DWORD PTR DS:[EAX],ESI	
7C862C42	75 3F	JNZ SHORT kernel32.7C862C83	
7C862C44	6A 01	PUSH 1	
7C862C46	68 0854887C	PUSH kernel32.7C885408	
7C862C48	E8 5D6BFAFF	CALL kernel32.InterlockedExchange	
7C862C52	85C0	TEST EAX,EAX	
7C862C54	75 2F	JNZ SHORT kernel32.7C862C83	
7C862C56	8B03	MOV EAX,DWORD PTR DS:[EBX]	
7C862C58	68 C435867C	PUSH kernel32.7C8635C4	ASCII ".cwr (cont
7C862C5A	FF73 04	PUSH DWORD PTR DS:[EBX+4]	
7C862C5C	68 AC35867C	PUSH kernel32.7C8635AC	ASCII ".cwr (excep
7C862C5E	50	PUSH EAX	
7C862C60	68 8C35867C	PUSH kernel32.7C86358C	ASCII "Code perfor
7C862C62	FF70 0C	PUSH DWORD PTR DS:[EAX+C]	
7C862C64	68 6C35867C	PUSH kernel32.7C86356C	ASCII "Invalid ad
7C862C66	FF70 18	PUSH DWORD PTR DS:[EAX+18]	
7C862C68	68 3835867C	PUSH kernel32.7C863538	ASCII "access vio
7C862C6A	68 02000020	PUSH 20000020	
7C862C6C	F8 F8C80100	CALL <JMP.>ntdll.RtlApplicationVerifier	
Jump is NOT taken			
7C862CC1=kernel32.7C862CC1			

所以说该缓冲区中的值不为零的话,JE 条件跳转将不会执行,程序最终将终止执行。



我们可以看到 OD 下方的提示:调试器遇到不可处理的异常,程序将终止。

现在重启 OD,重复上面的步骤,直到调用完 ZwQueryInformationProcess,然后我们来修改其返回的结果。

我们来到这里

7C862C08	50	PUSH EAX	
7C862C09	6A 07	PUSH 7	
7C862C0B	E8 FDB3FAFF	CALL kernel32.GetCurrentProcess	
7C862C10	50	PUSH EAX	
7C862C11	FF15 AC10807C	CALL DWORD PTR DS:[<ntdll.NtQueryInformationProcess	ntdll.ZwQueryInformationProcess
7C862C17	85C0	TEST EAX,EAX	
7C862C19	0F8C A2000000	JL kernel32.7C862CC1	
7C862C1F	39BD DCFEFFFF	CMP DWORD PTR SS:[EBP-124],EDI	

我们在数据窗口中定位缓冲区。

Address	Hex dump	ASCII
0012F5E4	00 00 00 00 28 E7 97 7C(ëü!
0012F5EC	FF FF FF FF 07 E7 97 7C	..ëü!
0012F5F4	6B 97 95 7C 00 00 E6 00	küð!..p.
0012F5FC	61 00 00 50 78 24 E6 00	a..Px\$p.
0012F604	00 00 00 00 54 F6 12 00	...T+\$.
0012F60C	D6 20 34 7C 00 00 E6 00	i 4!..p.
0012F614	00 00 00 00 0F 20 34 7C	.. 4!

我们按 F8 键单步执行该 API 函数。

Address	Hex dump	ASCII
0012F5E4	FF FF FF FF 28 E7 97 7C	(ëü!
0012F5EC	FF FF FF FF 07 E7 97 7C	..ëü!
0012F5F4	6B 97 95 7C 00 00 E6 00	küð!..p.
0012F5FC	61 00 00 50 78 24 E6 00	a..Px\$p.
0012F604	00 00 00 00 54 F6 12 00	T+\$.

缓冲区中返回的值跟上次一样,也是 FFFFFFFF,我们将其修改为零。

Address	Hex dump	ASCII
0012F5E4	FF FF FF FF 28 E7 97 7C	(ëü!
0012F5EC	FF FF FF FF 07 E7 97 7C	..ëü!
0012F5F4	6B 97 95 7C 00 00 E6 00	küð!..p.
0012F5FC	61 00 00 50 78 24 E6 00	a..Px\$p.
0012F604	00 00 00 00 54 F6 12 00	T+\$.

将其修改为零。

Edit data at 0012F5E4

ASCII

....

UNICODE

..

HEX +04

00 00 00 00

☒ Keep size

OK

Cancel

Address	Hex dump	ASCII
0012F5E4	00 00 00 00 28 E7 97 7C(ëü!
0012F5EC	FF FF FF FF 07 E7 97 7C	..ëü!
0012F5F4	6B 97 95 7C 00 00 E6 00	küð!..p.
0012F5FC	61 00 00 50 78 24 E6 00	a..Px\$p.

接着还是进行比较。

7C862C17	85C0	TEST EAX,EAX	
7C862C19	0F8C A2000000	JL kernel32.7C862CC1	
7C862C1F	39BD DCFEFFFF	CMP DWORD PTR SS:[EBP-124],EDI	
7C862C25	0F84 96000000	JE kernel32.7C862CC1	
7C862C2B	64:A1 18000000	MOV EAX,DWORD PTR FS:[18]	
7C862C31	8B40 30	MOV EAX,DWORD PTR DS:[EAX+30]	

现在两者都为零。


```
EDI=00000000
Stack SS:[0012F5E4]=00000000
```

好了,现在 JE 条件跳转将实现。

7C862C11	FF15	HC10807C	CALL DWORD PTR DS:[K&ntdll.NtQueryInfo
7C862C17	85C0		TEST EAX,EAX
7C862C19	0F8C	A2000000	JL kernel!32.7C862CC1
7C862C1F	39BD	DCFEFFFF	CMP DWORD PTR SS:[EBP-124],EDI
7C862C25	0F84	96000000	JE kernel!32.7C862CC1
7C862C2B	64A1	18000000	MOV EAX,DWORD PTR FS:[18]
7C862C31	8B40	30	MOV EAX,DWORD PTR DS:[EAX+30]
7C862C34	5F40	60 04	TEST BYTE PTR DS:[FSI+60],4

我们运行起来。

00401108	8B75 08	MOV ESI,DWORD PTR SS:[EBP+8]
0040110B	8B46 04	MOV EAX,DWORD PTR DS:[ESI+4]
0040110E	05 B8000000	ADD EAX,0B8
00401113	8BF0	MOV ESI,EAX
00401115	8B00	MOV EAX,DWORD PTR DS:[EAX]
00401117	83C0 0E	ADD EAX,0E
0040111A	8906	MOV DWORD PTR DS:[ESI],EAX
0040111C	A1 53314000	MOV EAX,DWORD PTR DS:[403153]
00401121	33C9	XOR ECX,ECX
00401123	33D2	XOR EDX,EDX

断在了 401108 地址处,我们看到下面的关键部分。

0040111C	A1 53314000	MOV EAX,DWORD PTR DS:[403153]
00401121	33C9	XOR ECX,ECX
00401123	33D2	XOR EDX,EDX
00401125	33FF	XOR EDI,EDI
00401127	0FBEB9 543041	MOVSX EDI,BYTE PTR DS:[ECX+403054]
0040112E	81C2 78563411	ADD EDX,12345678
00401134	D1D2	RCL EDX,1
00401136	13D7	ADC EDX,EDI
00401138	81C2 2143658	ADD EDX,87654321
0040113E	41	INC ECX
0040113F	3BC8	CMP ECX,EAX
00401141	75 E4	JNZ SHORT Sphynx.00401127
00401143	81FA 382AB4C	CMP EDX,C3B42A38
00401149	75 32	JNZ SHORT Sphynx.0040117D
0040114B	33C9	XOR ECX,ECX
0040114D	33D2	XOR EDX,EDX
0040114F	BE 07304000	MOV ESI,Sphynx.00403007
00401154	BF 63314000	MOV EDI,Sphynx.00403163
00401159	8A56 12	MOV DL,BYTE PTR DS:[ESI+12]
0040115C	8A0431	MOV AL,BYTE PTR DS:[ECX+ESI]
0040115F	32C2	XOR AL,DL
00401161	8B0439	MOV BYTE PTR DS:[ECX+EDI],AL
00401164	41	INC ECX
00401165	83F9 31	CMP ECX,31
00401168	75 F2	JNZ SHORT Sphynx.0040115C
0040116A	6A 00	PUSH 0
0040116C	68 63314000	PUSH Sphynx.00403163
00401171	68 76314000	PUSH Sphynx.00403176
00401176	6A 00	PUSH 0
00401178	E8 19000000	CALL <JMP.&user32.MessageBoxA>
0040117D	33C9	XOR ECX,ECX

Style = MB_OK!MB_APPLMODAL
Title = ""
Text = ""
hOwner = NULL
MessageBoxA

这里验证序列号是否正确以决定是否弹出正确的消息框。

00401138	81C2 2143658	ADD EDX,87654321
0040113E	41	INC ECX
0040113F	3BC8	CMP ECX,EAX
00401141	75 E4	JNZ SHORT Sphynx.00401127
00401143	81FA 382AB4C	CMP EDX,C3B42A38
00401149	75 32	JNZ SHORT Sphynx.0040117D
0040114B	33C9	XOR ECX,ECX
0040114D	33D2	XOR EDX,EDX
0040114F	BE 07304000	MOV ESI,Sphynx.00403007
00401154	BF 63314000	MOV EDI,Sphynx.00403163
00401159	8A56 12	MOV DL,BYTE PTR DS:[ESI+12]
0040115C	8A0431	MOV AL,BYTE PTR DS:[ECX+ESI]
0040115F	32C2	XOR AL,DL
00401161	8B0439	MOV BYTE PTR DS:[ECX+EDI],AL
00401164	41	INC ECX
00401165	83F9 31	CMP ECX,31
00401168	75 F2	JNZ SHORT Sphynx.0040115C
0040116A	6A 00	PUSH 0
0040116C	68 63314000	PUSH Sphynx.00403163
00401171	68 76314000	PUSH Sphynx.00403176
00401176	6A 00	PUSH 0
00401178	E8 19000000	CALL <JMP.&user32.MessageBoxA>
0040117D	33C0	XOR EAX,EAX
0040117F	48	DEC EAX
00401180	C3	RETN 4

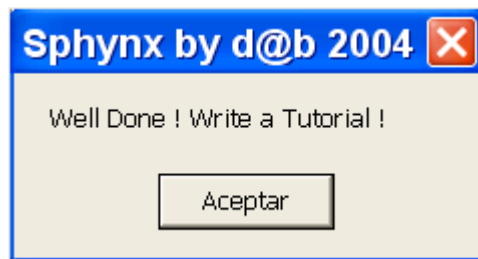
Style = MB_OK!MB_APPLMODAL
Title = ""
Text = ""
hOwner = NULL
MessageBoxA

如果我们修改跳转,让其跳转不实现,将会弹出序列号正确的消息框。

00401141	. ^ 75 E4	JNZ SHORT Sphynx.00401127	
00401143	. 81FA 382AB4C	CMP EDX,C3B42A38	
00401149	. ^ 75 32	JNZ SHORT Sphynx.00401170	
0040114B	. 33C9	XOR ECX,ECX	
0040114D	. 33D2	XOR EDX,EDX	
0040114F	. BE 07304000	MOV ESI,Sphynx.00403007	
00401154	. BF 63314000	MOV EDI,Sphynx.00403163	
00401159	. 8A56 12	MOV DL,BYTE PTR DS:[ESI+12]	
0040115C	. 8A0431	MOV AL,BYTE PTR DS:[ECX+ESI]	
0040115F	. 32C2	XOR AL,DL	
00401161	. 880439	MOV BYTE PTR DS:[ECX+EDI],AL	
00401164	. 41	INC ECX	
00401165	. 83F9 31	CMP ECX,31	
00401168	. ^ 75 F2	JNZ SHORT Sphynx.0040115C	
0040116A	. 6A 00	PUSH 0	
0040116C	. 68 63314000	PUSH Sphynx.00403163	
00401171	. 68 76314000	PUSH Sphynx.00403176	
00401176	. 6A 00	PUSH 0	
00401178	. E8 19000000	CALL <JMP.&user32.MessageBoxA>	
0040117D	. 33C0	XOR EAX,EAX	
0040117F	. 48	DEC EAX	

Style = MB_OK!MB_APPLMODAL
Title = ""
Text = ""
hOwner = NULL
MessageBoxA

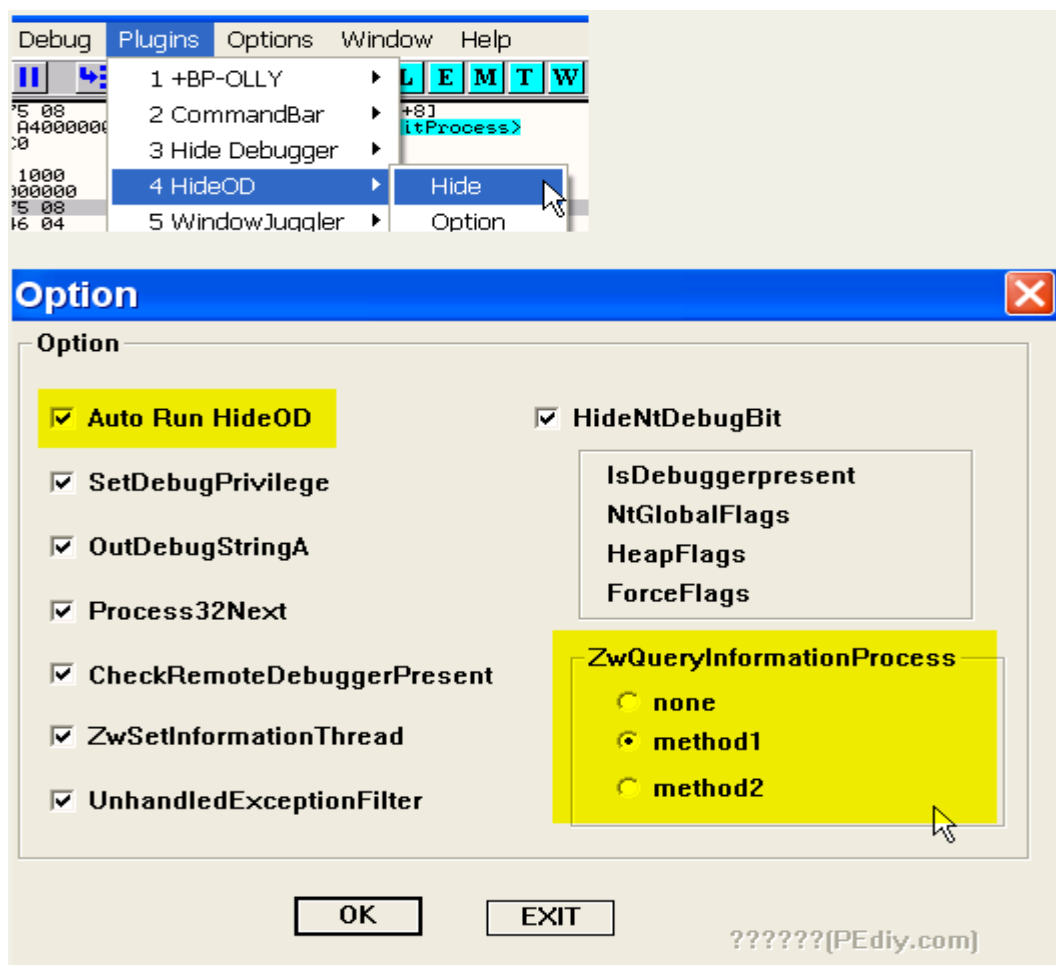
运行起来。



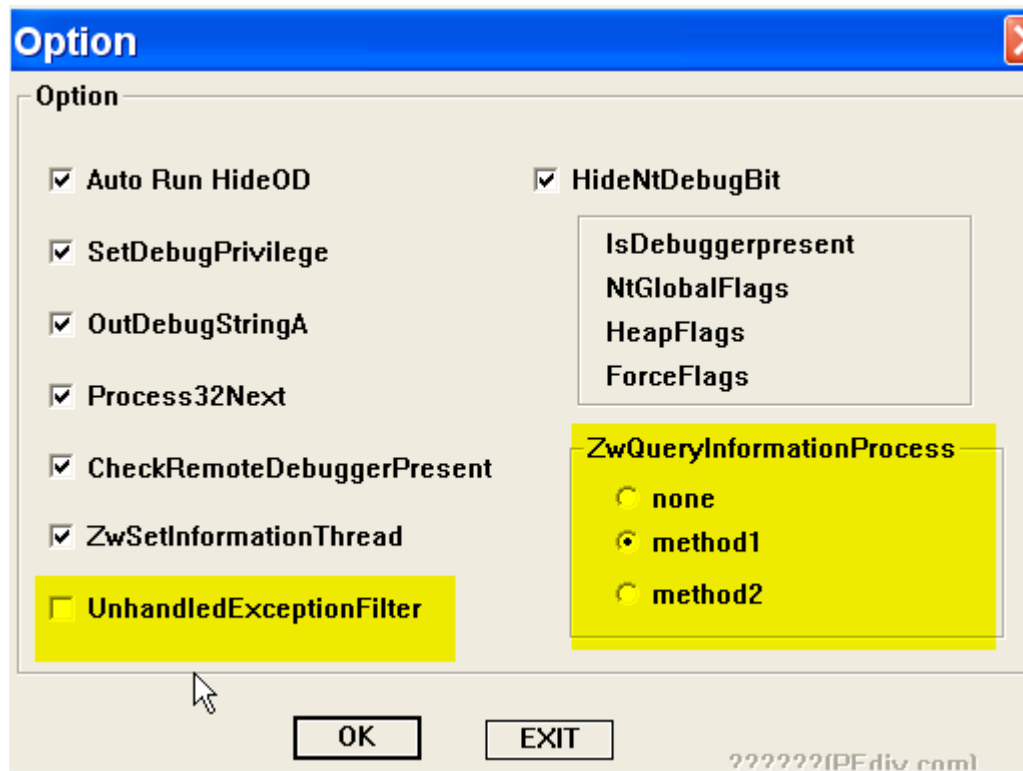
好了,我们前面已经提到过,通过 HideDebugger 插件就可以绕过 UnhandledExceptionTricks 的反调试。我们勾选上 UnhandledExceptionTricks 选项以后,重启 OD,可以看到运行的很正常。

好了,那么不考虑 UnhandledExceptionFilter,如何单独绕过 ZwQueryInformationProcess 这个函数的检测呢。

当然,我们可以手工将其返回值修改为零,那么有自动绕过的插件吗?当然有,那就是 HideOD 这款插件。



我们可以看到很多 HideDebugger 插件中没有的选项,我们和 HideDebugger 配合起来用,有一点很重要,就是别忘了勾选上 Auto Run HideOD 这个选项,这样我们就不必每次启动 OllyDbg 的时候配置该插件了。



这里我们不勾选上 UnhandledExceptionFilter,因为该选项最后也会绕过 ZwQueryInformationProcess 的反调试。我们现在只使用右边的单独绕过 ZwQueryInformationProcess 检测调试器的选项。

好了,本章我们学会了如何绕过 UnhandledExceptionFilter 以及 ZwQueryInformationProcess 的反调试原理,我们配合使用 HideOD 和 HideDebugger 两款插件让 OD 更加健壮了,嘿嘿。