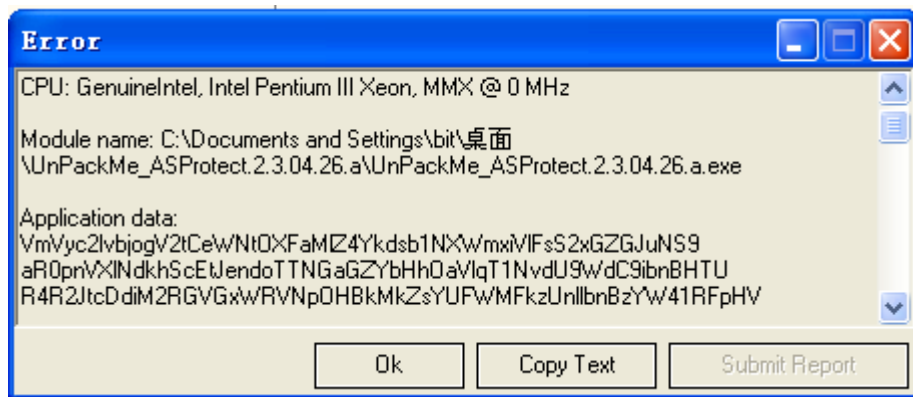


第五十一章-ASProtect v2.3.04.26 脱壳-Part1

从本章开始,我将会换一种讲解的方式,采用互动的方式来讲解。也可以说是引导的方式来讲解。本章我们的脱壳对象是 ASProtect 最新版(PS:作者当年的最新版)。本章就由我解决简单的部分,大家来完成复杂的部分,嘿嘿,给大家充分练手的机会。

这里我们的目标程序 UnPackMe_ASProtect.2.3.04.26.a.exe,该目标程序并没有添加全保护。只是较为简单的一个版本。这里大家需要注意一点,如果 OD 加载目标程序以后,到达 OEP 之前就由于该壳的保护报错的话,大家可以将其复制到别的路径下,然后重启加载试试。譬如说如下错误框:



(PS:我实验的时候,采用 XP SP3 的虚拟机,所有异常都忽略了,但是直接运行起来,还是报这个错误框,后来我换成 XP SP2 的虚拟机,忽略所有异常又是正确的,具体原因还不清楚。所以这里大家遇到这个问题实在搞不定的话,就换 XP SP2 的系统吧)

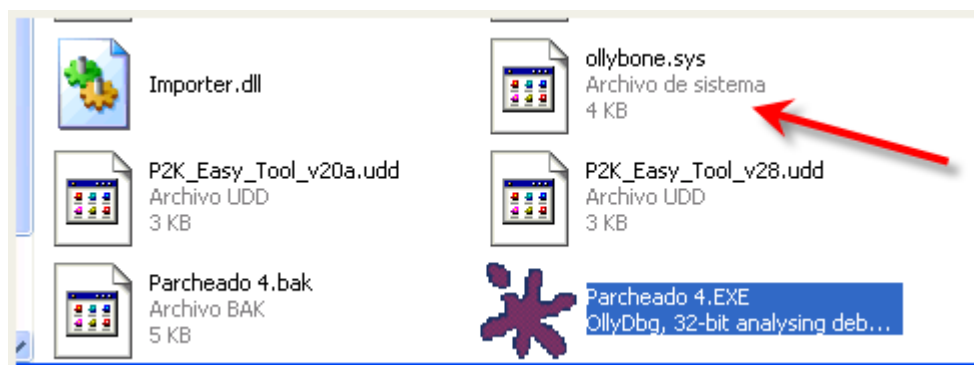
这里给大家介绍两款新的插件 OllyBone 和 Weasle。

OllyBone 这款插件的安装说明如下:

Installation:

Copy ollybone.dll and i386/ollybone.sys to your OllyDbg directory。

安装说明的意思就是说安装 OllyBone 插件的方法就是将 ollybone.dll 放到 OD 目录下的 Plugin 文件夹下,然后将 i386 文件夹下的 ollybone.sys 放到 OD 的目录下。



如上图所示,我们将 ollybone.sys 置于 OD 同目录下了,ollybone.dll 放到了 OD 目录下的 Plugin 文件夹中了。

至于 Weasle 这个插件,我们将 Importer.dll 置于 OD 同目录下,然后将 RL!Weasle.dll 放到 OD 目录下的 Plugin 文件夹下。

大家在脱 ASProtect 这款壳的时候要格外小心,因为它会对 INT3 断点以及硬件断点进行检测,如果检测到的话,就会报错,那我们就只能重新再来了。所以在定位 OEP 之前,我们需要配置 OllyBone 这款插件,首先我来给大家介绍一下 OllyBone 这款插件,首先根据 ollybone.sys 的扩展名来看就知道这是一个驱动程序,这个插件主要是用来模拟执行断点的,协助我们的 OD(我们前面章节介绍的专门用于定位 OEP 的那款 OD)更快的定位到 OEP,但是这个插件有个缺点,就是我们不能对程序进行单步,所以说我们在调试之前先要将异常选项中 Debugging options-Exceptions-Single-step break 这一项的对勾去掉。关于这一点 OllyBone 插件的作者的官网给了详细说明,网址如下:

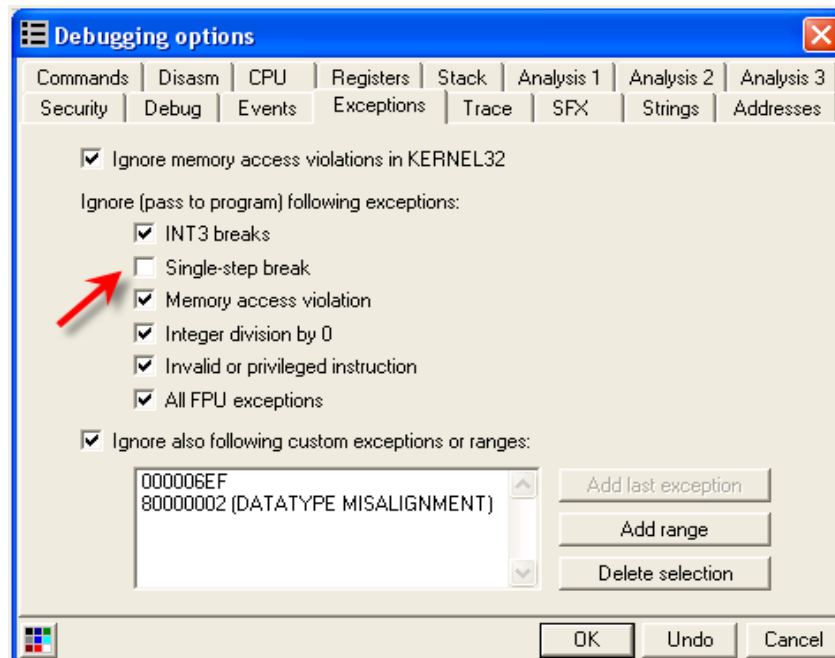
www.joestewart.org/ollybone/tutorial.html

driver load returned status 0x, when the program is run, Ollydbg will 'operatively' break on the first instruction, which would be our original entry point (0x).

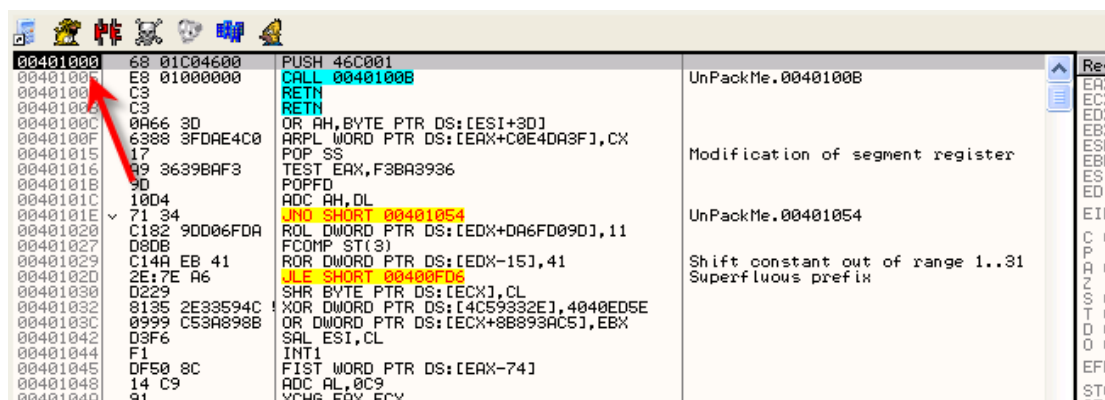
This technique generally only works with packers that append their unpacking code as a stub section to the PE file, then restore the unpacked code into its original sections. In many cases, packers may employ anti-debugging tricks during the execution of the stub code, which must still be worked around. Ollybone isn't going to work in all cases, but it can be a shortcut for quickly unpacking the most common packer code.

Installing OllyBone

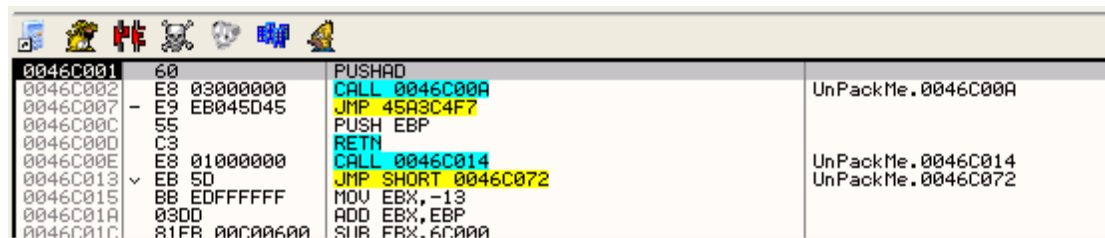
- Download OllyBone from <http://www.joestewart.org/ollybone/ollybone-0.1.zip>
- Unzip and copy ollybone.dll and 1386/ollybone.sys to your OllyDbg directory
- Your memory map right-click menu should now have the option "Set break-on-execute". Ensure that the checkbox in Debugging options->Exceptions for ignoring the Single-step break is "unchecked", otherwise the INT1 handler will not return control to the debugger.



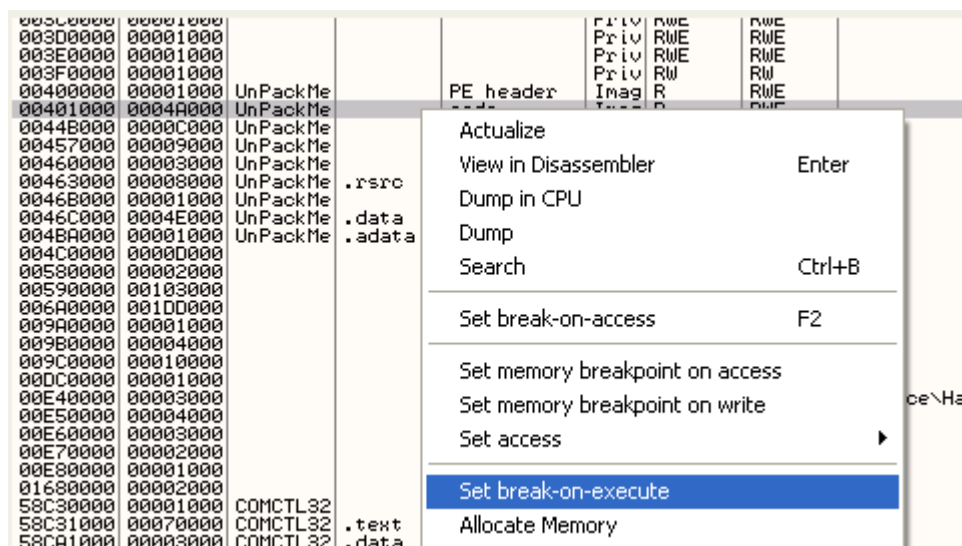
其他的忽略异常选项我们还是勾选上。



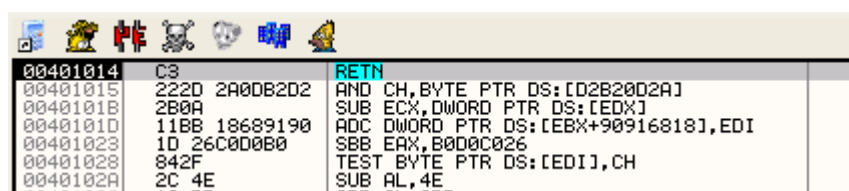
此时我们处于入口点处,即 ASPProtect 壳的入口点位于第一个区段中,我们单步往下跟踪几步就会跳转到别的区段。



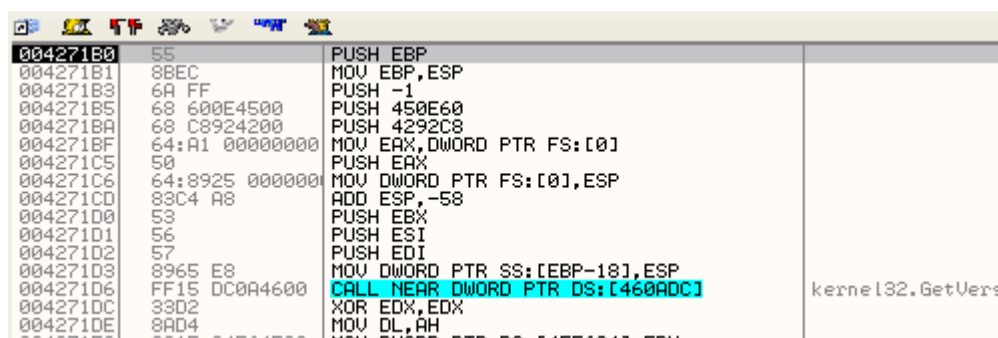
我们大概按 4,5 下 F7 键就可以由第一个区段跳转到起始地址为 460000 的这个区段。下面我们打开区段列表窗口,给第一个区段设置 break-on execute(执行断点,该功能是 OllyBone 插件提供的)。



断在了这里。



这里对于一般的壳来说,我们现在应该就到了 OEP 处了,但是对于 ASProtect 来说,我们断在了这里,这里明显不像 OEP,这里我们单步执行这个 RET 指令,会发现还是返回到了壳创建的区段中,接着我们直接运行起来。



等了一段时间,就断在了 OEP 处。

(PS:依然是作者定位 OEP 的方法,我按照作者的方式选择 Set break-on-execute 这一项,Bingo 蓝屏了)



怎么会这样呢?其实啊,这是 OllyBone 插件要求环境导致的。

我们来看看官方的说明:

Caveats

Since OllyBone fiddles with the memory page permissions, other kernel drivers which might affect memory pages might conflict with OllyBone. One thing known not to get along with OllyBone is DEP, the Microsoft protection built in to XP SP2. DEP can be disabled - if you experience blue-screening when setting a break-on-execute, this is probably what you need to do.

You can change the DEP settings under:

Control Panel->System->Advanced->Performance->Settings->Data Execution Prevention.

If that doesn't work, you may need to add

/noexecute=alwaysOff

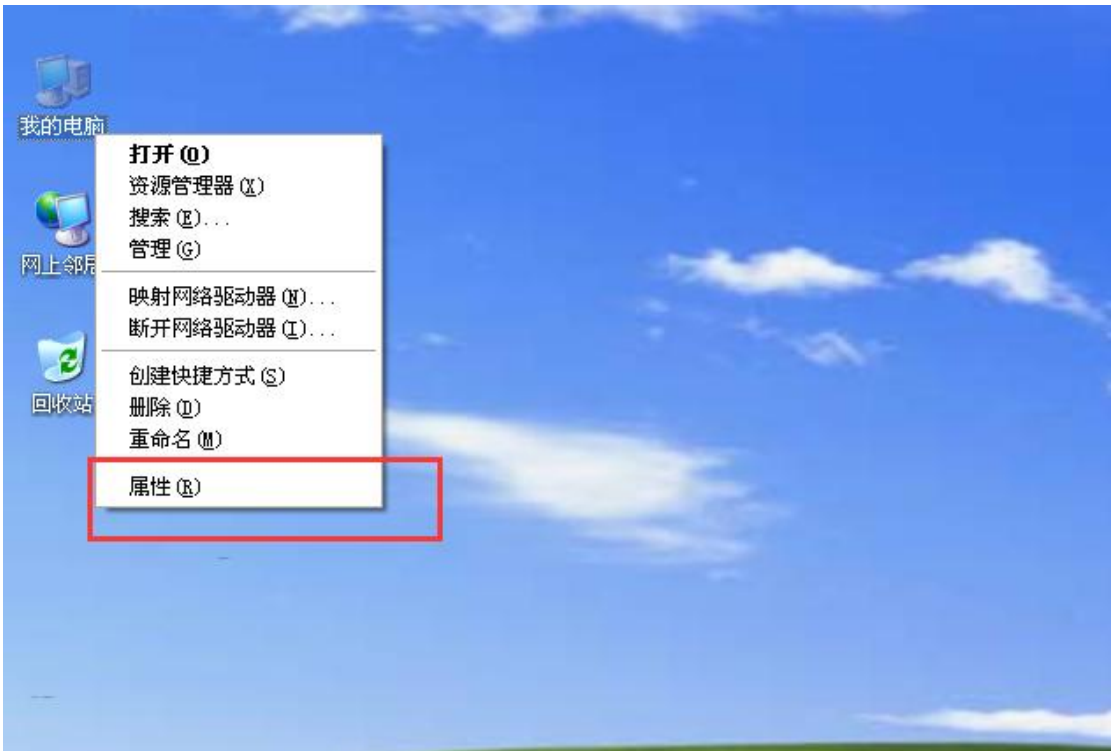
to your Windows partition in boot.ini.

Of course, this makes your system more vulnerable to buffer overflow exploits, so using OllyBone on production systems is not advised. OllyBone will work under VMWare, so this may be a preferred method for its use. However, note that some pointers actively refuse to run under virtual machines, so it may not be the best solution in every case.

OllyBone isn't a magic-bullet unpacker for all cases, and certainly has its flaws. During the time that the program is running before the breakpoint is reached, the unpacking code can be doing anything, including detecting OllyBone and subverting it, using the driver to even crash the system by execute-protecting random pages of memory. For now OllyBone is more of a proof-of-concept than a finished program - keep that in mind as you use it.

可以看到 OllyBone 的作者在其主页上已经说明清楚了,使用 OllyBone 插件的时候不能开启 DEP,不然会蓝屏。

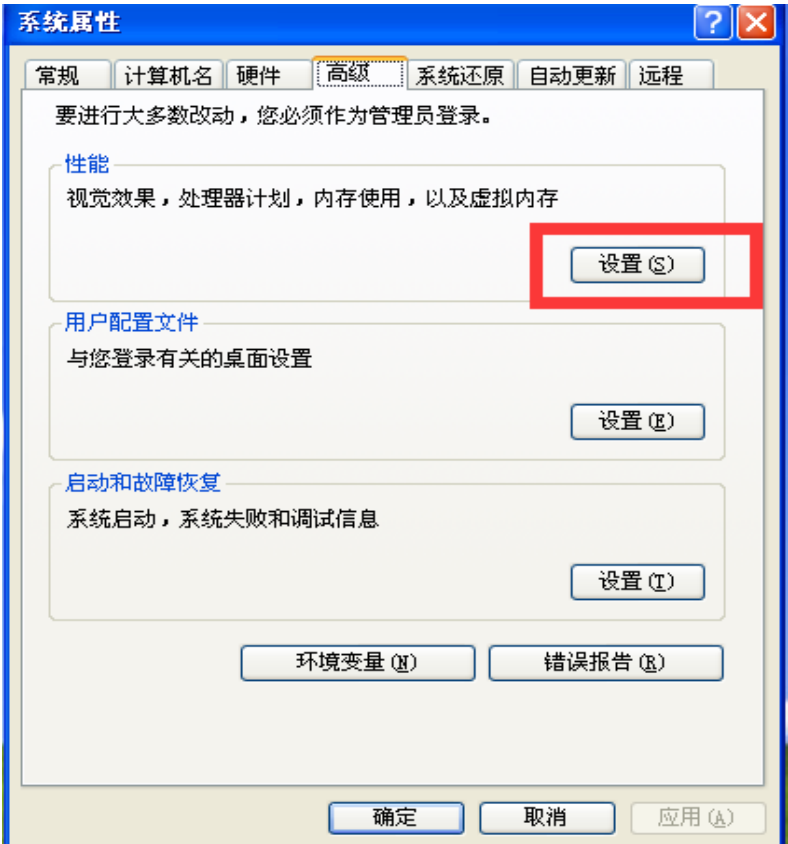
关闭 DEP 的方法如下:



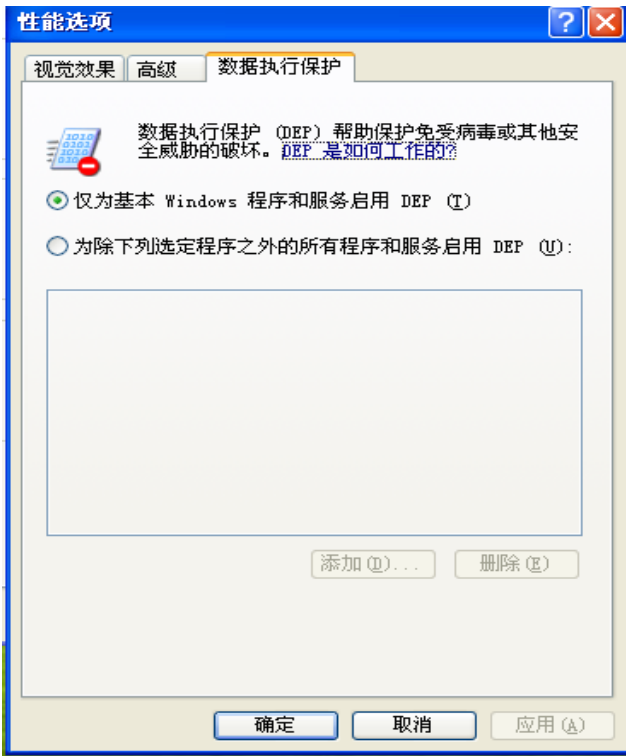
在我的电脑上单击鼠标右键选择属性。



我们先来查看一下 DEP 是不是确实开启了,选择性能的设置按钮。
切换到高级选项卡。

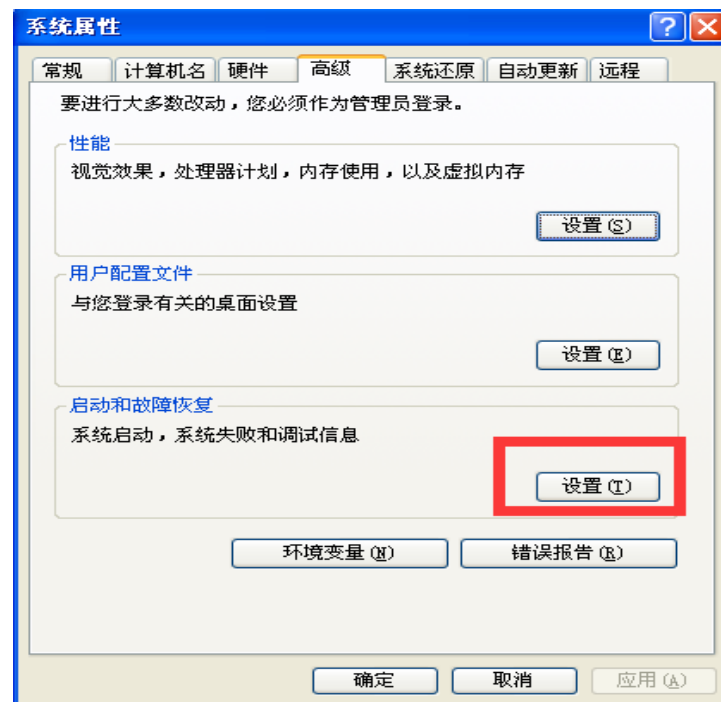


切换到数据执行保护选项卡。

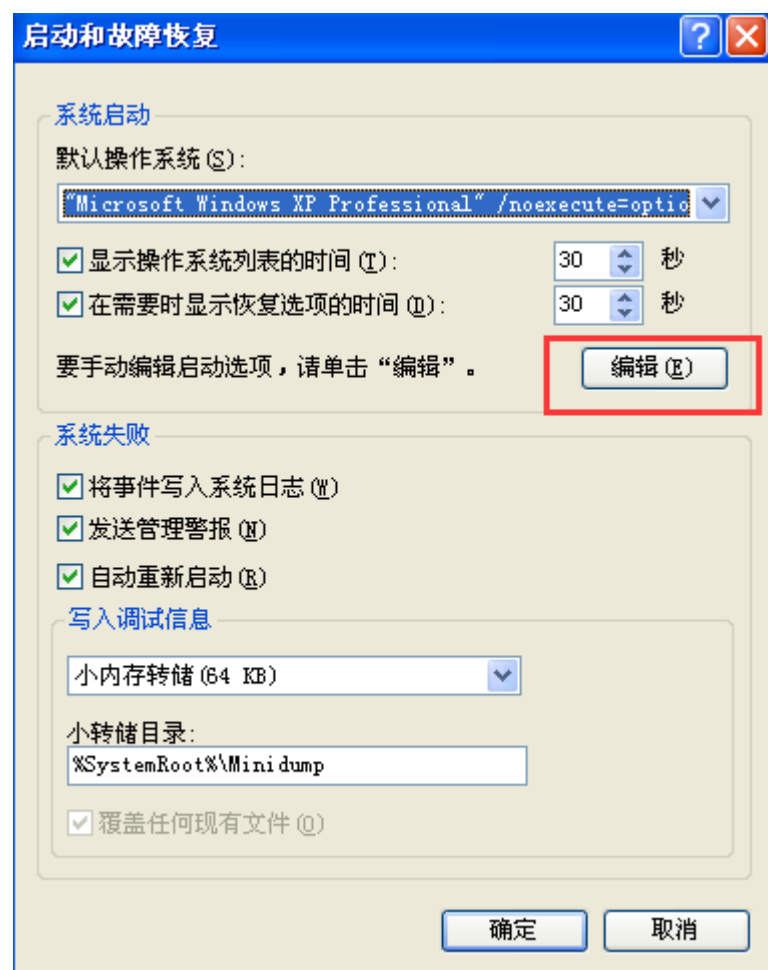


可以看到确实启用了 DEP,未启用 DEP 的时候这个选项卡的对话框是灰下去的,无法选择。

下面我们来关闭 DEP,选择设置启动和故障修复按钮。



选择编辑按钮(即可以编辑 boot.ini 文件)。



接着讲/noexecute 这个选项的值由 option 修改为 AlwaysOff 即可关闭 DEP。

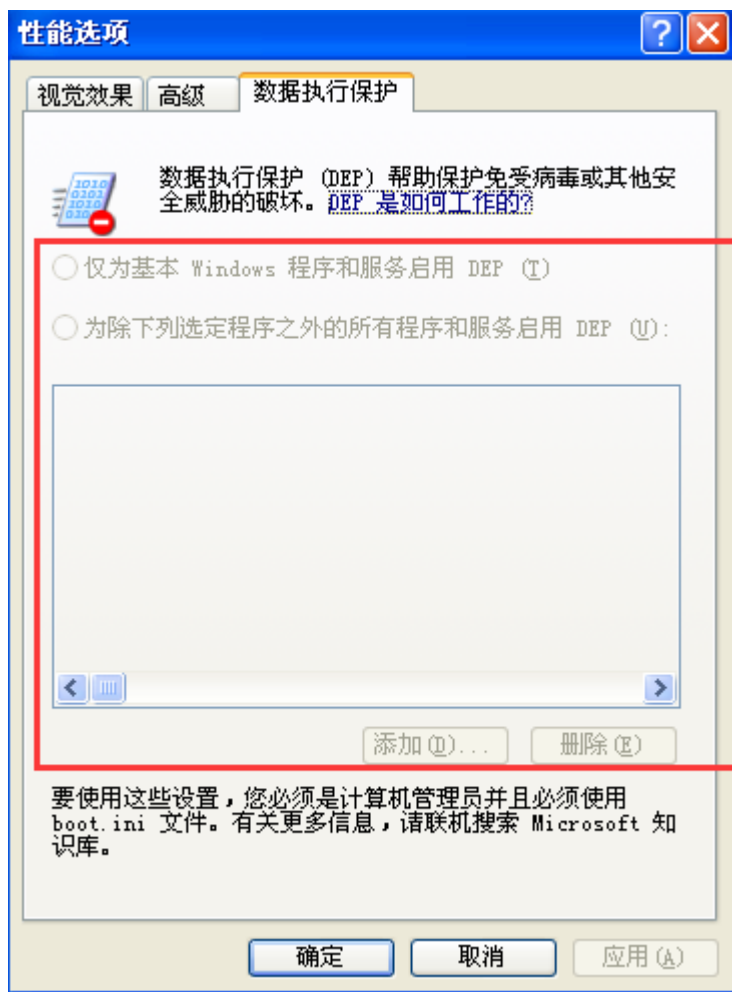
```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=option /fastdetect /PAE
```

将option修改为AlwaysOff即可关闭DEP

修改完毕以后保存文件。然后重启电脑即可关闭 DEP 了。

好了,现在我已经重启电脑了。

我来看看 DEP 是否已经关闭了。



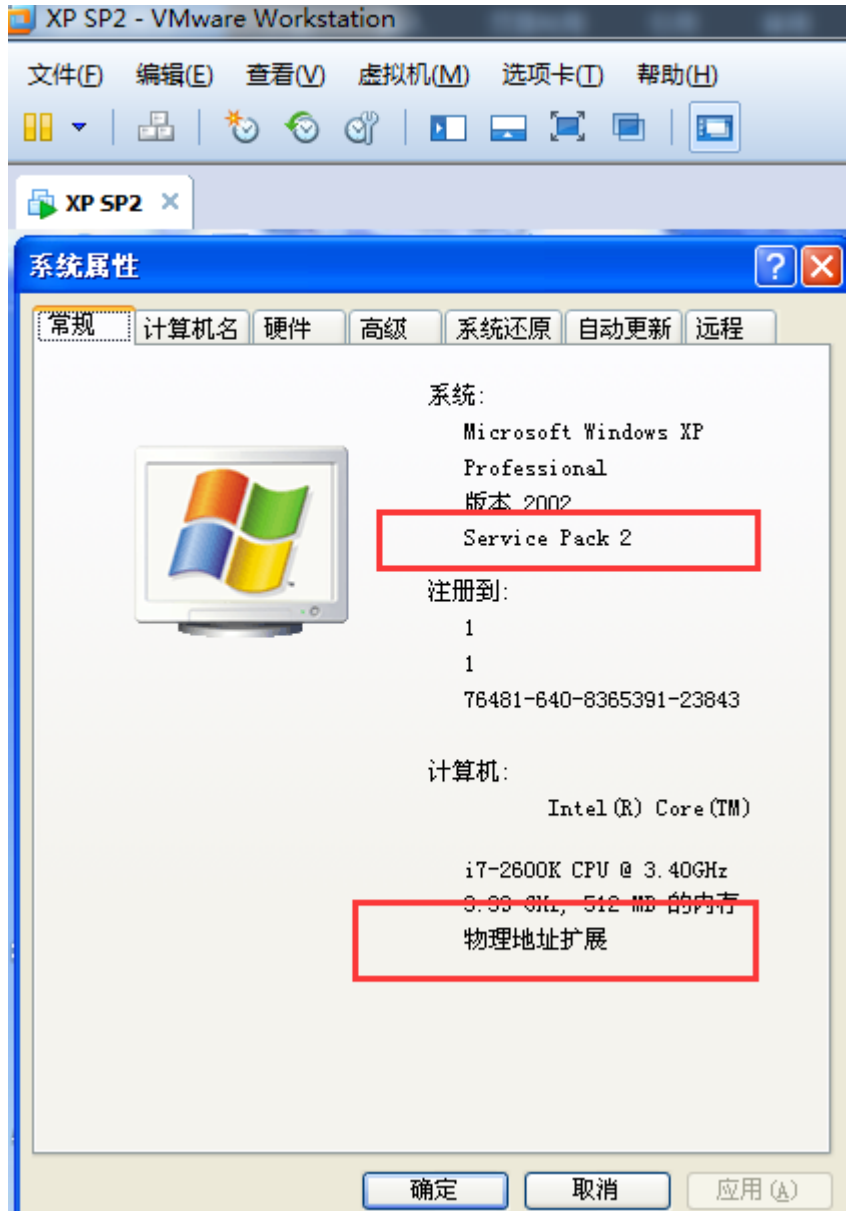
可以看到已经灰下去了,说明 DEP 已经关闭了。

下面我们继续使用 OllyBone 插件来 Set break-on-execute。

(PS:但是我尝试了多遍,OllyBone 压根不起作用,根本断不下来。可能是年代比较久远的原因吧。我查了下资料,softworm 大叔 09 年的时候写了篇帖子提到了 OllyBone,说这个插件不大好用,经常断不下来。可能在 PTE 上做了手脚,softworm 大叔就用 DEP 又实现了一遍 BreakOnExecute 的功能,贴出来实现代码,等我后面有时间,再来尝试下写 BreakOnExecute 这个插件吧。具体什么时候,那就不得而知了,哈哈)

好了,下面说说我是怎么定位 OEP 的吧。

我实验的环境是 XP SP2。



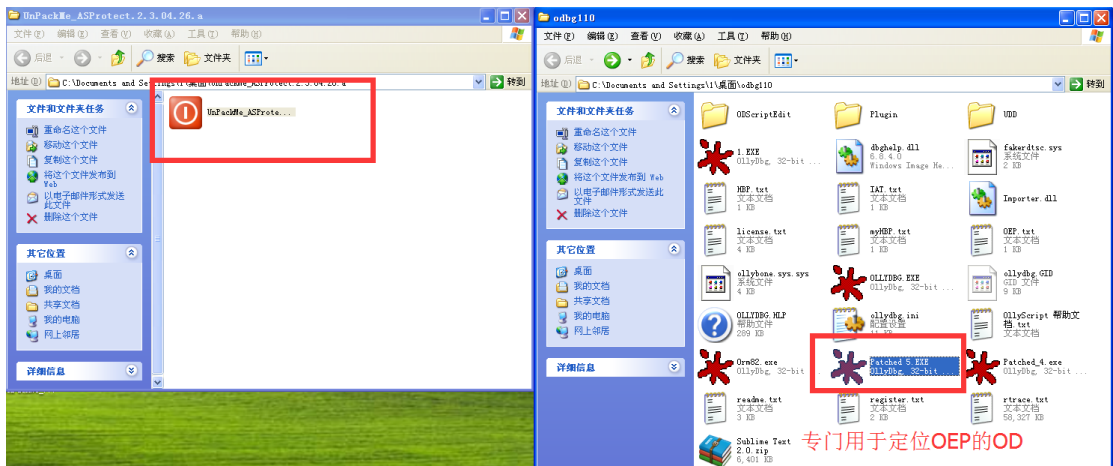
物理地址扩展我是开启的。



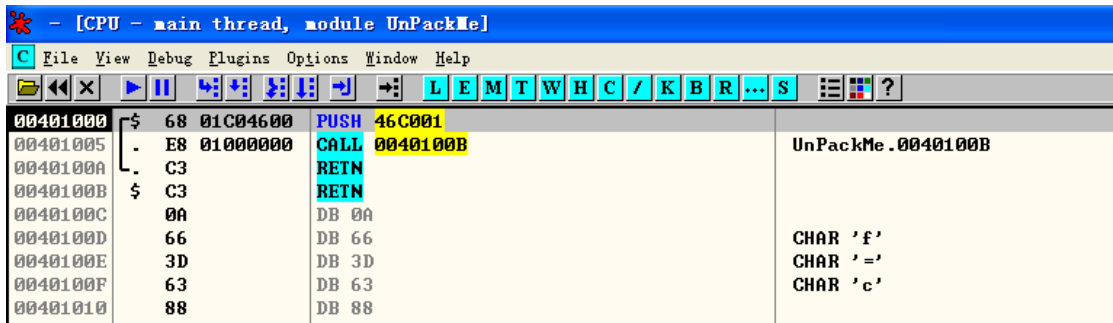
DEP 我也关闭了,这里其实没有什么影响。大家随意。

我采用的是最后一次异常法,大家应该很熟悉了吧。

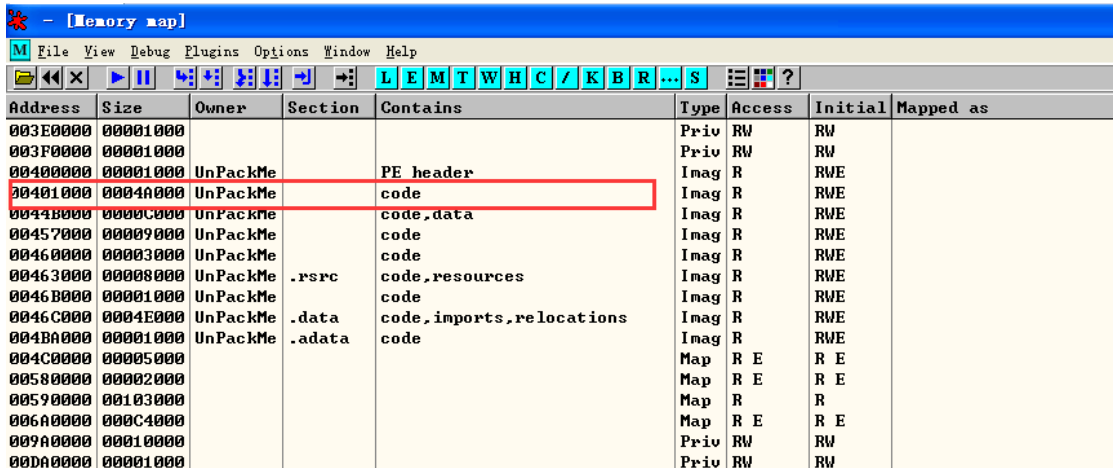
直接用专门定位 OEP 的那款 OD(PS:不用我多说的了,内存单元被 Patch,之前章节介绍过很多遍了)加载目标程序。



停在了壳的入口点处。

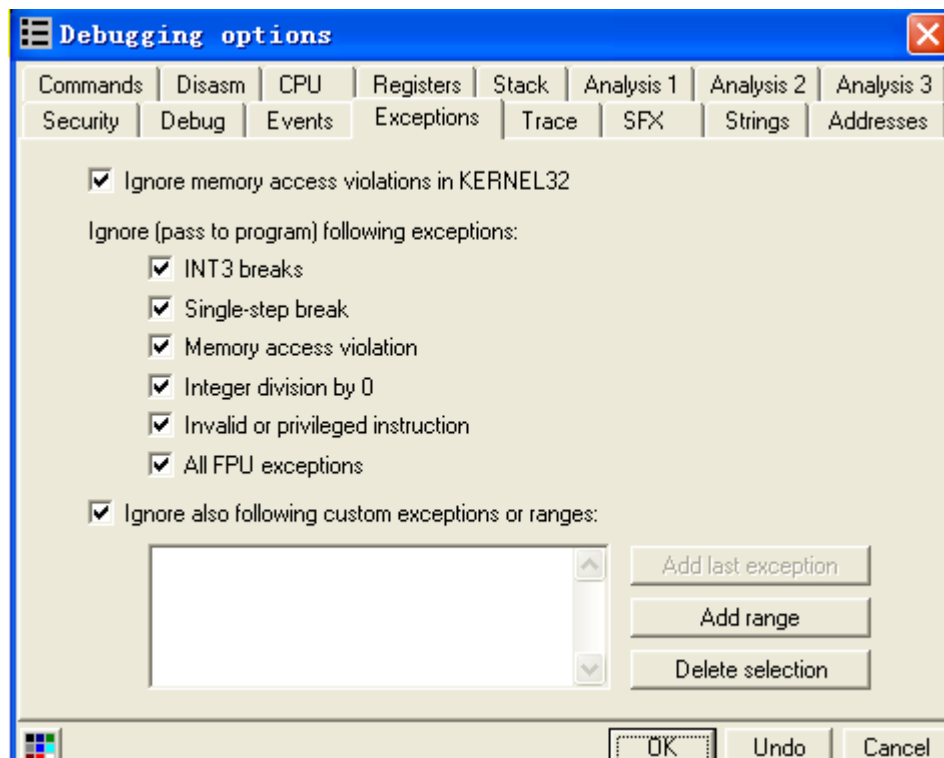


查看区段列表可以得知,壳的入口点也在于代码段(PS:OEP 实际上也位于代码段)。



实际上是 ASProtect 作者玩的伎俩。

接下来我们将忽略异常的选项都勾选上。

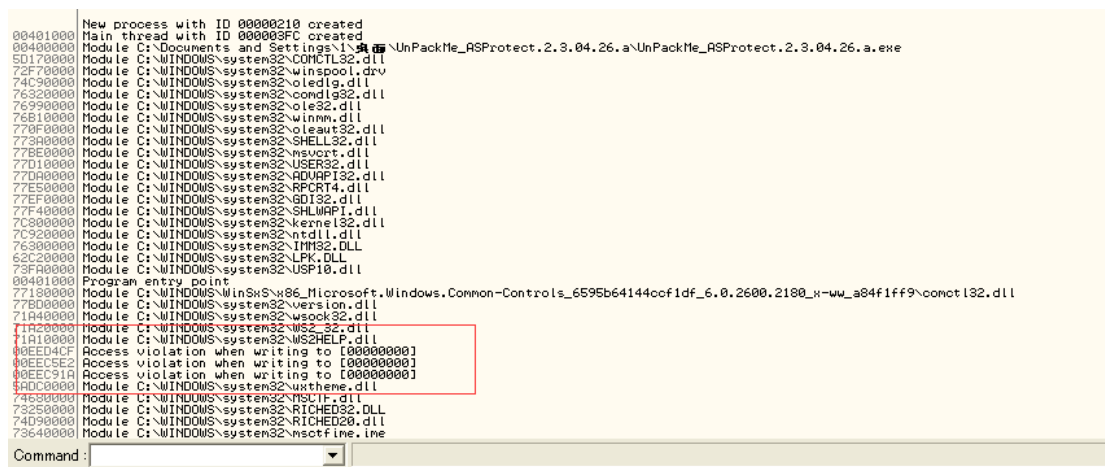


然后直接运行起来。



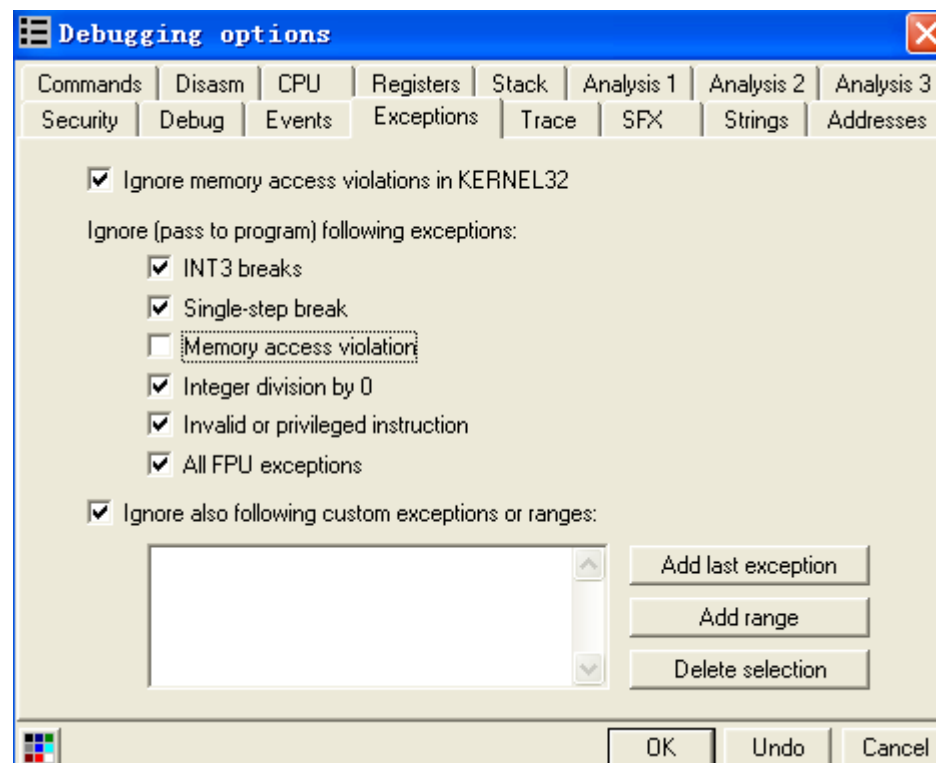
可以看到,程序直接正常运行起来了。

接下来我们打开日志窗口查看下都发生了哪些异常。



可以看到最后一次异常是 EEC91A 处产生的。

接下来,我们将忽略内存访问异常这个选项的对勾去掉。



重启 OD,依然断在了入口点处,然后按 F9 键直接运行起来。

00EED4CF	0000	ADD BYTE PTR DS:[EAX],AL
00EED4D1	E8 33C05A59	CALL 5A499509
00EED4D6	59	POP ECX
00EED4D7	64:8910	MOV DWORD PTR FS:[EAX],EDX
00EED4DA	EB 0F	JMP SHORT 00EED4EB
00EED4DC	E9 8759FCFF	JMP 00EB2E68
00EED4E1	E8 8EFBFFFF	CALL 00EED074
00EED4E6	E8 D95CFCFF	CALL 00EB31C4
00EED4EB	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]
00EED4EE	E8 2157FCFF	CALL 00EB2C14
00EED4F3	E8 A0EAF0FF	CALL 00EEBF98
00EED4F8	5F	POP EDI
00EED4F9	5E	POP ESI
00EED4FA	5B	POP EBX
00EED4FB	0000	MOV ESP,EBP

断在了这里,并不是最后一次异常处。

我们按 shift+F9,忽略掉该异常继续运行。

00EEC5E2	0000	ADD BYTE PTR DS:[EAX],AL
00EEC5E4	E8 33C05A59	CALL 5A49861C
00EEC5E9	59	POP ECX
00EEC5EA	64:8910	MOV DWORD PTR FS:[EAX],EDX
00EEC5ED	EB 0F	JMP SHORT 00EEC5FE
00EEC5EF	E9 7468FCFF	JMP 00EB2E68
00EEC5F4	E8 63FEFFFF	CALL 00EEC45C
00EEC5F9	E8 C66BFCFF	CALL 00EB31C4
00EEC5FE	A1 6CFAEE00	MOV EAX,DWORD PTR DS:[EEFA6C]
00EEC603	C600 CF	MOV BYTE PTR DS:[EAX],0CF
00EEC606	EB 7E	JMP SHORT 00EEC686
00EEC608	E8 3F5FFCFF	CALL 00EB254C
00EEC60D	8BD8	MOV EBX,EAX

接着断在了这里,还不是最后一次异常处,我们继续按 shift+F9,忽略掉该异常继续运行。

00EEC91A	0000	ADD BYTE PTR DS:[EAX],AL	
00EEC91C	E8 33C05A59	CALL 5A498954	
00EEC921	59	POP ECX	
00EEC922	64:8910	MOV DWORD PTR FS:[EAX],EDX	
00EEC925	EB 2B	JMP SHORT 00EEC952	
00EEC927	E9 3C65FCFF	JMP 00EB2E68	
00EEC92C	8B45 F4	MOV EAX,DWORD PTR SS:[EBP-C]	
00EEC92F	50	PUSH EAX	
00EEC930	8B45 F8	MOV EAX,DWORD PTR SS:[EBP-8]	
00EEC933	50	PUSH EAX	
00EEC934	8B45 FC	MOV EAX,DWORD PTR SS:[EBP-4]	

好,这里就断在了最后一次异常处了。

接下来我们给代码段设置内存访问断点(PS:这里该 OD 的内存访问断点实际上只有执行的时候才会断下来。)

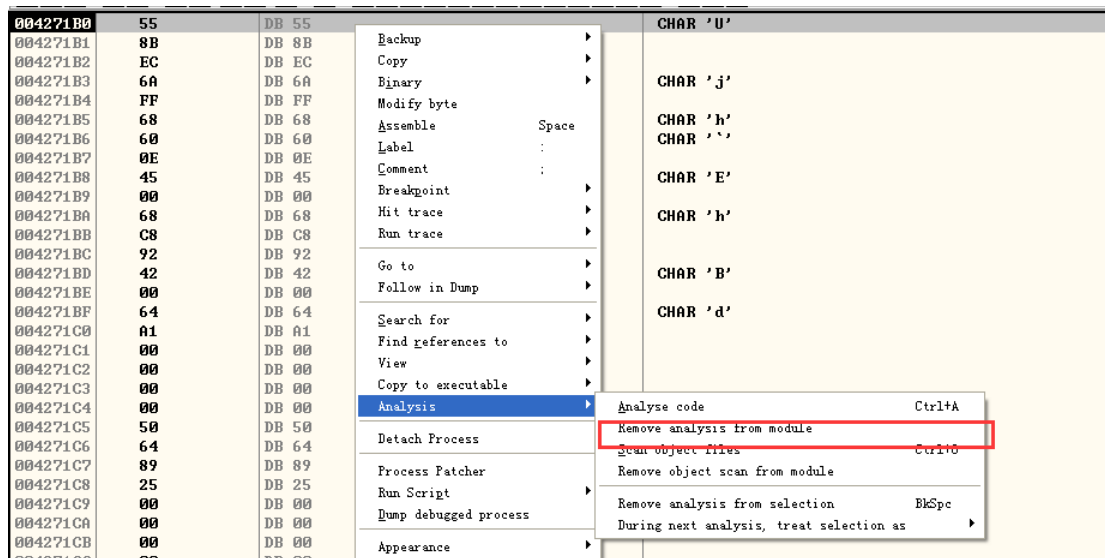
003D0000	00008000				Priv	RW			
003E0000	00001000				Priv	RW			
003F0000	00001000				Priv	RW			
00400000	00001000	UnPackMe	PE header		Imag	R			RWE
00401000	00004000	UnPackMe	code		Imag	R			RWE
0044B000	0000C000	UnPackMe	Actualize			R			RWE
00457000	00009000	UnPackMe	View in Disassembler	Enter		R			RWE
00460000	00003000	UnPackMe	Dump in CPU			R			RWE
00463000	00008000	UnPackMe	.rsrc	Dump		R			RWE
0046B000	00001000	UnPackMe		Search	Ctrl+B	R			RWE
0046C000	00004E000	UnPackMe	.data			R			RWE
004BA000	00001000	UnPackMe	.adata	Set break-on-access	F2	R			RWE
004C0000	00005000			Set memory breakpoint on access		R	E		R E
00580000	00002000			Set memory breakpoint on write		R	E		R E
00590000	00103000			Set access		R	E		R E
009A0000	00010000			Allocate Memory		RW			RW
00DA0000	00001000			Free Memory		RW			RW
00E20000	00003000			Zero Memory		R			R
00E30000	00004000			Dump Memory-Area		RW			RW
00E40000	00003000			Load dumped memory		R			R
00E50000	00002000					R			R
00E60000	00050000					R			R

然后继续按 shift+F9,忽略掉该异常继续运行。

- [CPU - main thread, module UnPackMe]			
File View Debug Plugins Options Window Help			
L E M T W H C / K B R ... S			
004271B0	55	DB 55	CHAR 'U'
004271B1	8B	DB 8B	
004271B2	EC	DB EC	
004271B3	6A	DB 6A	CHAR 'j'
004271B4	FF	DB FF	
004271B5	68	DB 68	CHAR 'h'
004271B6	60	DB 60	CHAR ''
004271B7	0E	DB 0E	
004271B8	45	DB 45	CHAR 'E'
004271B9	00	DB 00	
004271BA	68	DB 68	CHAR 'h'
004271BB	C8	DB C8	
004271BC	92	DB 92	
004271BD	42	DB 42	CHAR 'B'
004271BE	00	DB 00	
004271BF	64	DB 64	CHAR 'd'
004271C0	A1	DB A1	
004271C1	00	DB 00	

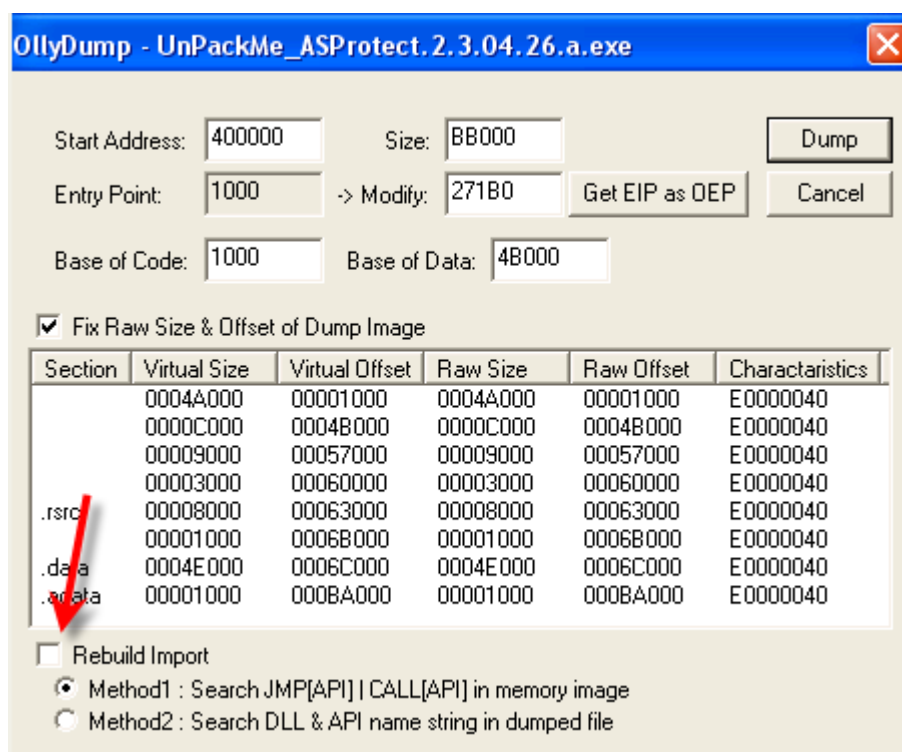
好,断在了这里。这里就是 OEP 了。

这里我们在反汇编窗口中单击鼠标右键选择 Analysis-Remove analysis from module 选项重新分析代码。



好,现在的显示就正常了。啧啧...成功定位到了 OEP。

下面我们来进 dump,我们打开 OllyDump 插件,我个人不太习惯用 OllyDump 来修复 IAT,所以这里我不勾选 Rebuild Import 这个选项。



下面我们来定位 IAT。

004271C0	83C4 H8	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	FF15 DC0A4600	CALL NEAR DWORD PTR DS:[460ADC]	kernel32.dll
004271DC	33D2	XOR EDX,EDX	
004271DE	8AD4	MOV DL,AH	
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX	
004271E6	8BC8	MOV ECX,EAX	
004271E8	81E1 FF000000	AND ECX,0FF	
004271EE	890D 30E64500	MOV DWORD PTR DS:[45E630],ECX	
004271F4	C1E1 08	SHL ECX,8	
004271F7	03CA	ADD ECX,EDX	

DS:[00460ADC]=7C8114B8 (kernel32.GetVersion)

Address	Hex dump	ASCII
00460ADC	AB 14 81 7C F4 97 80 7C	%u!%u!
00460AE4	01 B0 85 7C 19 62 82 7C	%a!%b!
00460AEC	5C E8 81 7C 53 00 83 7C	%u!%s!
00460AF4	19 3C 87 7C CB D8 81 7C	%c!%i!
00460AFC	C1 0F 87 7C 5B B2 81 7C	%c!%u!
00460B04	E9 06 87 7C 4E 99 80 7C	%c!%N!
00460B0C	AC 92 80 7C 11 07 87 7C	%c!%c!
00460B14	42 24 80 7C F3 B8 81 7C	%c!%u!
00460B1C	A9 2C 87 7C F4 2C 87 7C	%c!%c!

这里我们可以看到 OEP 下方有一条指令调用了 GetVersion 这个 API 函数,说明其是 IAT 中的一项,好了,现在我们在数据窗口中定位到这一项,接着我们来定位 IAT 的起始地址。

00440017	CD 0C770001	CALL 01700000	
DS:[00460818]=77DA6BF0 (ADVAPI32.RegCloseKey)			

Address	Hex dump	ASCII
00460808	99 68 5F 1B 00 00 00 00	0h_+....
00460810	60 03 58 FD 00 00 00 00	%s%
00460818	F0 6B DA 77 1B 76 DA 77	-k r w+o r w
00460820	F4 EA DA 77 E7 EB DA 77	%u r w+o r w
00460828	83 78 DA 77 00 00 00 00	%k r w....
00460830	DD 15 C5 58 2E B0 C3 58	%s+X.c %X
00460838	00 00 00 00 D4 6A EF 77	...%j'w
00460840	66 95 EF 77 89 6A EF 77	%o'w%j'w
00460848	F3 AD EF 77 ED D9 EF 77	%a'w%j'w
00460850	99 88 EF 77 C0 B5 EF 77	%i'w%a'w
00460858	2A 7D EF 77 B2 7C EF 77	%j'w%a'w
00460860	77 53 F2 77 1E C9 F1 77	%S=w%a'w
00460868	0C BC EF 77 52 D4 EF 77	%a'w%a'w
00460870	FA 80 EF 77 F1 D0 EF 77	%i'w%a'w
00460878	51 B2 EF 77 26 D5 EF 77	%a'w%a'w
00460880	2A E3 EF 77 5F 39 F2 77	%o'w_9=w
00460888	71 B4 EF 77 2E AD EF 77	%h'w.%a'w
00460890	E1 61 EF 77 B8 85 EF 77	%a'w%a'w
00460898	CC D2 EF 77 43 70 EF 77	%a'w%a'w
004608A0	FB EA F0 77 12 83 EF 77	%o-w%a'w
004608A8	01 72 F0 77 A9 34 F0 77	%o-w%a'w
004608B0	05 03 FF 77 C0 FF FF 77	%s%w%a'w

这里我们可以看到 IAT 的起始地址为 460818。接着我们可以看出 IAT 的结束位置为 460F28。

EBP=0012FF98			
--------------	--	--	--

Address	Hex dump	ASCII
00460EB8	CE 00 37 76 7C 86 37 76	%f.7v!%7v
00460EC0	B0 86 37 76 33 25 36 76	%%7v3%6v
00460EC8	1E 31 36 76 D8 7C 37 76	%!6v!%7v
00460ED0	89 C2 37 76 CD 46 38 76	%t7v%F8v
00460ED8	CE EE 36 76 00 00 00 00	%t'6v....
00460EE0	48 D0 4C 77 9C CB 4D 77	%sLw%a'w
00460EE8	CC 42 4F 77 2C D0 4C 77	%fB0w.%sLw
00460EF0	DA F6 4C 77 73 33 50 77	%r+Lws3Pw
00460EF8	10 64 4D 77 03 0E 52 77	%dMw%a'w
00460F00	33 0F 52 77 40 A6 54 77	%3w%a'w
00460F08	F1 A7 54 77 92 9C 4F 77	%tTw%a'w
00460F10	6F 57 52 77 99 33 4E 77	%oWRw%a'w
00460F18	B2 5D 4E 77 90 C0 5A 77	%INw%a'w
00460F20	00 00 00 00 F3 F0 CC 74	...%f%t
00460F28	00 00 00 00 0B 00 05 47	...%s.%G
00460F30	E8 D5 54 8C E7 86 C4 B9	%b'T!%a'w
00460F38	00 00 9D 19 BA 8C 90 EF	..%a'w!%e
00460F40	B9 B2 8C 00 FE 00 91 7B	%a'w.%a'w
00460F48	FD B2 7E AB 45 6F 22 FD	%a'w%a'w
00460F50	ED 16 C1 93 38 A8 00 00	%Y-68%a'w

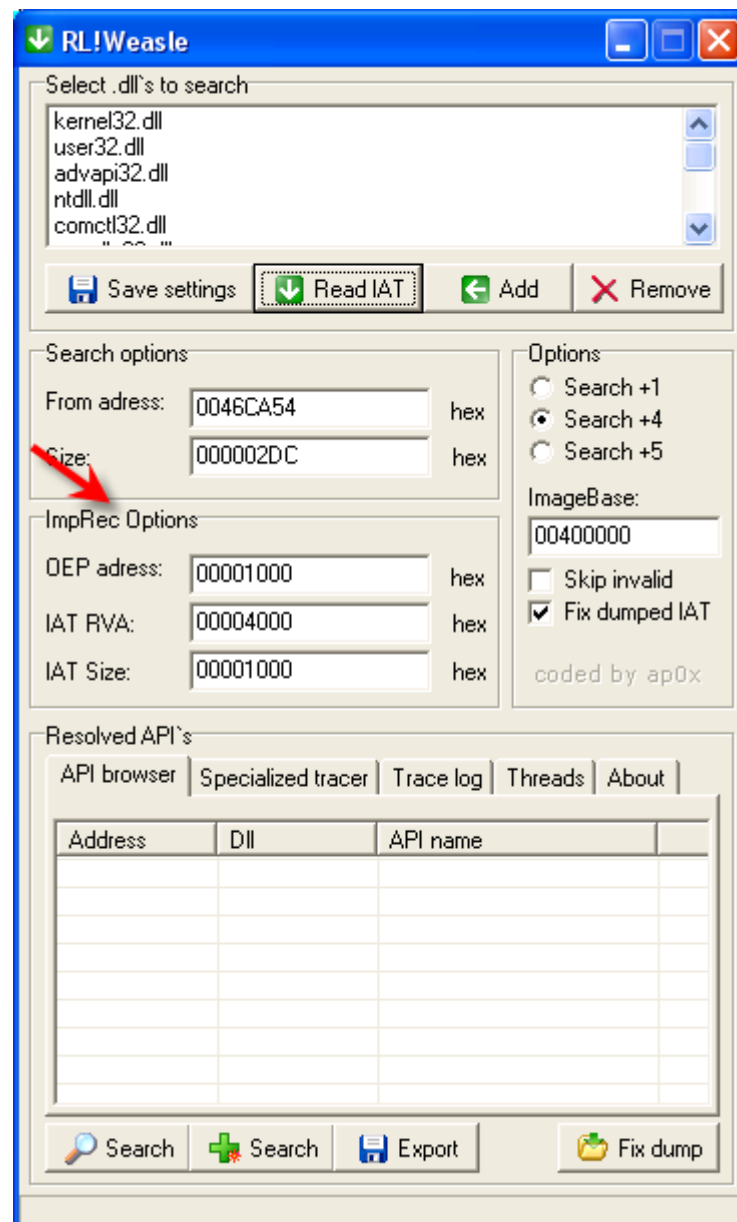
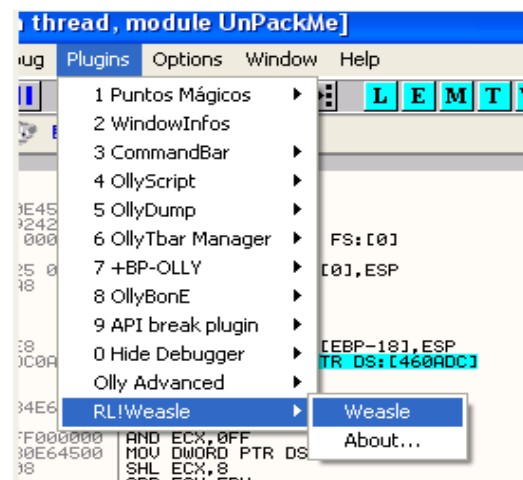
所以我们可以得到:

OEP = 4271B0

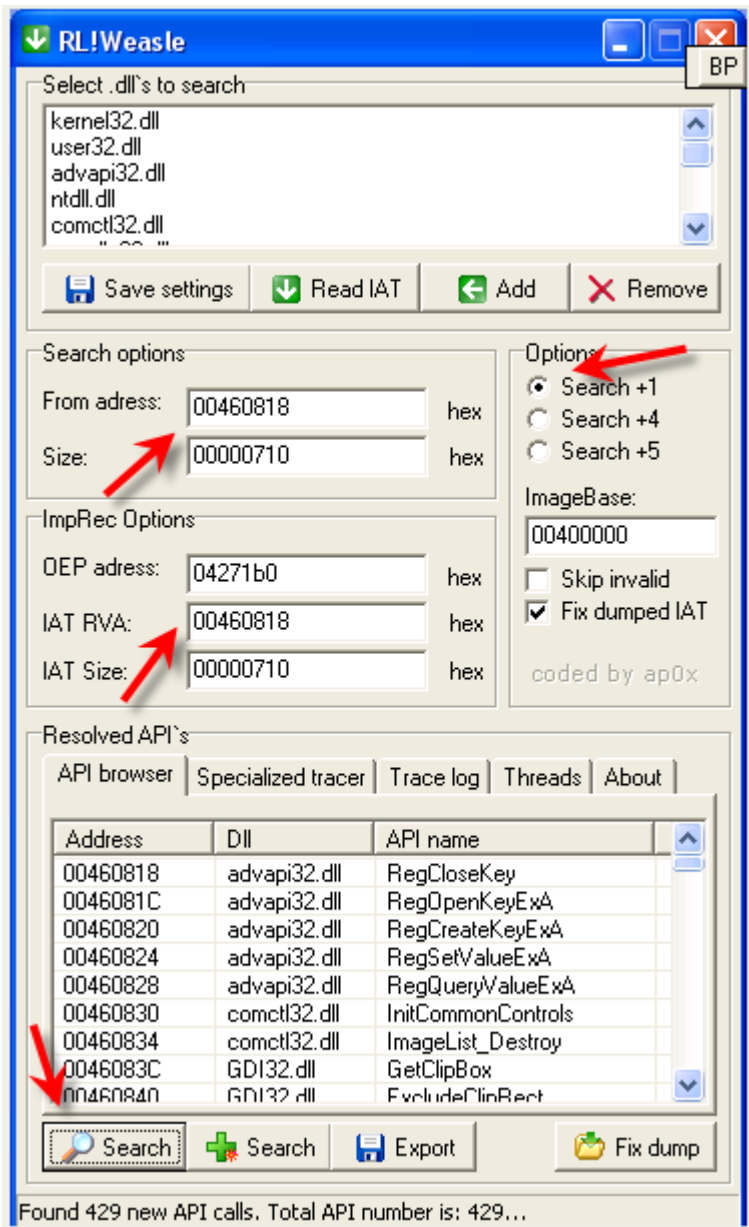
IAT 的起始地址 = 460818

IAT 的长度 = 710

这次我不用 IMP REC 来修复 IAT 了。这里我给大家演示如果用 Weasle 这个插件来修复 IAT。



我们找到 ImpRec Options 这一项,填充 OEP,IAT 起始地址,IAT 大小。

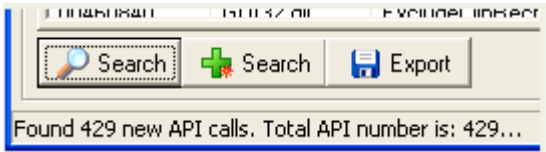


这里我们需要将 OEP,IAT 的起始地址,IAT 的大小,以及从什么地址开始搜索,搜索的范围大小是多少都填充上,接着将搜索模式选为 Search+1。这样理论上就可以开始进行搜索了。但是,这款插件有时候会漏掉某些 DLL,我们就需要手动将 DLL 添加上,这里我们单击 Add 按钮来添加 DLL,譬如这里的 460F24 这个 IAT 项所在的 DLL 就被遗漏了。我们来看看该项的参考引用。

Address	Hex dump	ASCII
00460EFC	03 0E 52 77 33 0F 52 77	03Rw3Rw
00460F04	40 A6 54 77 F1 A7 54 77	03Tw0Tw
00460F0C	92 9C 4F 77 6F 57 52 77	060w0Rw
00460F14	99 33 4E 77 B2 5D 4E 77	03Nw0Nw
00460F1C	90 C0 5A 77 00 00 00 00	012w...
00460F24	F3 F0 CC 74 00 00 00 00	01ft...
00460F2C	0B 00 A5 47 E8 D5 54 8C	01GP'TI
00460F34	E7 86 C4 B9 00 00 9D 19	01-1..0↓
00460F3C	BA 8C 90 EF B9 B2 8C 00	01E'100I.
00460F44	FE 00 91 7B FD B2 7E AB	01.c(200's

Address	Disassembly	Comment
00435D38	JMP NEAR DWORD PTR DS:[460F24]	oledlg.OleUIBusyA

这里明显我们可以看到 oledlg.dll 这个 DLL 被遗漏了。我们添加上这个 DLL 然后再次单击 Search 按钮进行搜索。这样就 OK 了。但是由于这个插件还是一个测试版本,所以可能搜索的结果会有一些错误。我迫不及待的想试试发布版了,嘿嘿。下面我们来看看修复的效果如何。

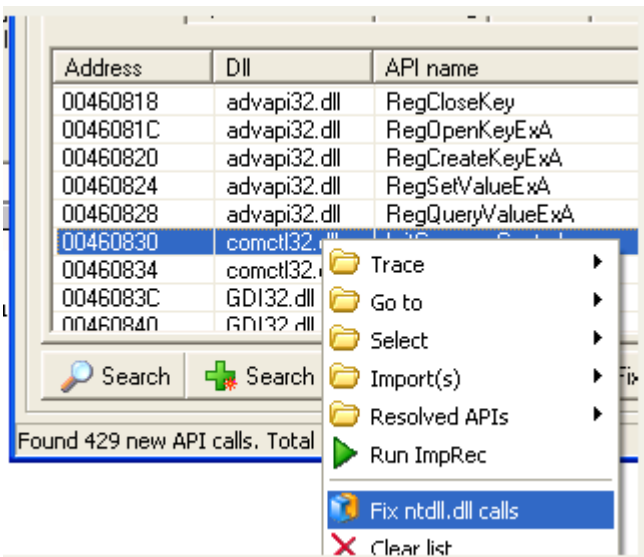


这里我们可以看到提示找到了 429 处 API 函数调用处。总共定位到 API 函数 429 个。

我们直接单击 Fix dump 按钮修改 dump 文件。



我们运行修复后的 dump 文件,直接报错了,可以看到错误提示是无法定位在 kernel32.dll 中定位到 RtlSizeHeap 这个函数。当然无法定位到啦,RtlSizeHeap 这个 API 函数压根就不是 Kernel32.dll 导出的,它是 ntdll.dll 导出的。还好,该插件提供了一个修复 ntdll.dll 中的 API 函数的功能,我们单击鼠标右键选择 Fix ntdll.dll calls 即可。



现在我们将刚刚修复的 exe 后缀的文件删除掉,然后将 bak 后缀的文件重命名为 exe 后缀,然后再次单击 Fix dump 按钮修复 dump 文件。这里 IAT 就被修复了。但是我们直接运行修复后的文件发现还是无法正常运行,说明还存在 AntiDump。我们再次用 OD 加载修复后的程序。

004271B0	55	PUSH EBP	
004271B1	8BEC	MOV EBP,ESP	
004271B3	6A FF	PUSH -1	
004271B5	68 600E4500	PUSH 450E60	
004271BA	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	FF15 DC0A4600	CALL NEAR DWORD PTR DS:[460ADC]	kernel32.GetVersion
004271DC	33D2	XOR EDX,EDX	
004271DE	8AD4	MOV DL,AH	
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX	
004271E6	8BC8	MOV ECX,EAX	

这里我们可以看到 API 函数显示正常了,说明 IAT 已经修复了。我们单击鼠标右键选择 Search for-All intermodulate calls 查看所有 API 函数调用处。

Address	Disassembly	Destination
00423310	CALL 019B0000	
00423A4F	CALL NEAR DWORD PTR DS:[460A00]	ntdll.RtlFreeHeap
00423B1C	CALL NEAR DWORD PTR DS:[4609FC]	ntdll.RtlAllocateHeap
00423C43	CALL NEAR DWORD PTR DS:[4609FC]	ntdll.RtlAllocateHeap
00423CA8	CALL NEAR DWORD PTR DS:[4609F8]	ntdll.RtlReAllocateHeap
00423E5C	CALL 019B0000	
00423E96	CALL NEAR DWORD PTR DS:[460B5C]	ntdll.RtlGetLastWin32Error
004249B6	CALL NEAR DWORD PTR DS:[4609F4]	ntdll.RtlSizeHeap
00425003	CALL NEAR DWORD PTR DS:[460B98]	kernel32.InterlockedIncrement
004251B3	CALL NEAR DWORD PTR DS:[460B98]	kernel32.InterlockedIncrement
004251C7	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
0042520B	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
004252D4	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
004252FB	CALL NEAR DWORD PTR DS:[460978]	kernel32.GetLocalTime
00425306	CALL 019B0000	
0042535E	CALL 019B0000	
004259D1	CALL NEAR DWORD PTR DS:[460A54]	kernel32.InitializeCriticalSection
004259E8	CALL NEAR DWORD PTR DS:[460A3C]	ntdll.RtlEnterCriticalSection
004259F5	CALL NEAR DWORD PTR DS:[460A44]	ntdll.RtlLeaveCriticalSection
00425B2B	CALL 019B0000	
00425B3F	CALL 019B0000	
00425B53	CALL 019B0000	
00425BF1	CALL 019B0000	
00425C9C	CALL 019B0000	
00425CA6	CALL NEAR DWORD PTR DS:[460B5C]	ntdll.RtlGetLastWin32Error
00425D71	CALL NEAR DWORD PTR DS:[460A54]	kernel32.InitializeCriticalSection
00425D8B	CALL NEAR DWORD PTR DS:[460A3C]	ntdll.RtlEnterCriticalSection
00425DBB	CALL NEAR DWORD PTR DS:[460A44]	ntdll.RtlLeaveCriticalSection
00425E13	CALL NEAR DWORD PTR DS:[460B98]	kernel32.InterlockedIncrement
00425E27	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
00425E68	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
00425F34	CALL NEAR DWORD PTR DS:[460B94]	kernel32.InterlockedDecrement
004271B0	PUSH EBP	(Initial CPU selection)
004271D6	CALL NEAR DWORD PTR DS:[460ADC]	kernel32.GetVersion
0042723E	CALL 019B0000	
004272D5	CALL 019B0000	

这里我们可以看到有多处 CALL 的目标地址都是 19B0000(大家的机器上这个地址可能不太一样,以自己机器为准)。下面给大家的任务就是修复 AntiDump。大家不必太担心,因为我会给出一些提示。

给大家的任务就是 15 天之内编写出一个脚本来修复 AntiDump。完成任务的童鞋可以发邮件给我,我会一个一个的看并进行点评。在下一个章节中,我会给出一个在我看来最简单最高效的脚本。

15 天以内完成任务的童鞋可以给我发送邮件,附上脚本以及您的 ID,修复了 AntiDump 的童鞋将在下一章节中被提名以资鼓励。

下面给大家一点提示。

我们用 OD 加载原程序,并且定位到 OEP 处。

004272CA	C745 D0 000000	MOV DWORD PTR SS:[EBP-30],0	
004272D1	8D45 A4	LEA EAX,DWORD PTR SS:[EBP-5C]	
004272D4	50	PUSH EAX	
004272D5	E8 268D5801	CALL 019B0000	
004272DA	D9F6	FDECSTP	
004272DC	45	INC EBP	
004272DD	D001	ROL BYTE PTR DS:[ECX],1	

这里我们比较将未 dump 之前的 OEP 下方的 4272D5 此处 CALL 与 dump 并修复 IAT 以后的进行比较会发现下面还存在其他指令。

004272D5	FF15 80094600	CALL NEAR DWORD PTR DS:[460980]	kernel32.GetStartupInfoA
004272D8	F645 00 01	TEST BYTE PTR SS:[EBP-30],1	
004272DF	74 0A	JE SHORT 004272EB	UnPackMe.004272EB

针对于 4272D5 这一行 dump 并修复 IAT 后,我们可以看到调用了 GetStartupInfoA 这个 API 函数。4272D5 到 4272DB 一共占 6 个字节。而未修复 IAT 之前 ASProtect 将其替换成了一个占 5 个字节的 CALL(很明显第 6 个字节是垃圾指令)。未修复 IAT 的情况下 4272D5 这个 CALL 的返回地址处的指令如下:

004272D5 E8 268D5801 CALL 019B0000

004272DA D9F6 FDECSTP

这里我就不给出原作者的提示了,作者的提示比较隐晦。

这里我给出我的提示。

大家可以给 4272D5 这一行设置一个断点,运行起来,就断在了这一行。然后我们利用 OD 自带的跟踪功能来定位修改 AntiDump 的关键点。我们需要定位处于哪一条指令处时的通用寄存器中保存了 GetStartupInfoA 这个 API 函数的地址。跟踪停止后,如果大家感觉停下来的地方不是很像关键点的话,就继续自动跟踪,直到确认是关键点为止。我们还有另一个切入点可以更加精确的定位关键点。就是执行完 CALL 019B0000 这条指令后,正常情况下会返回到 4272DA 这个地址处,但是 4272DA 处这个字节是垃圾指令。所以 ASProtect 壳在调用完 GetStartupInfoA 这个 API 函数并且返回到之前势必会将 4272DA 这个字节修改掉,并且将返回地址修改为 4272DB,这样接下来才能够正常继续往下执行。所以基于这个切入点,我们可以对 4272D5 开始的多个字节设置内存访问断点,将这两个切入点结合起来,就可以让大家更加精确的定位关键点了。

大家要做的就是尽自己所能在 2006 年 8 月 17 号之前编写能够在各个系统上都能够良好执行的脚本。对于完成任务的童鞋我会在下一章节中予以提名以资鼓励。