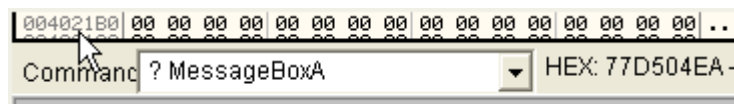


第三十三章-神马是 IAT,如何修复

在介绍如何修复 IAT 之前,我们首先来介绍一下 IAT 的相关基本概念,本章的实验对象依然是 Cruehead 的 CrackMe。首先我们来定位该程序的 IAT 位于何处,然后再来看看对其加了 UPX 的壳后,IAT 又位于何处。

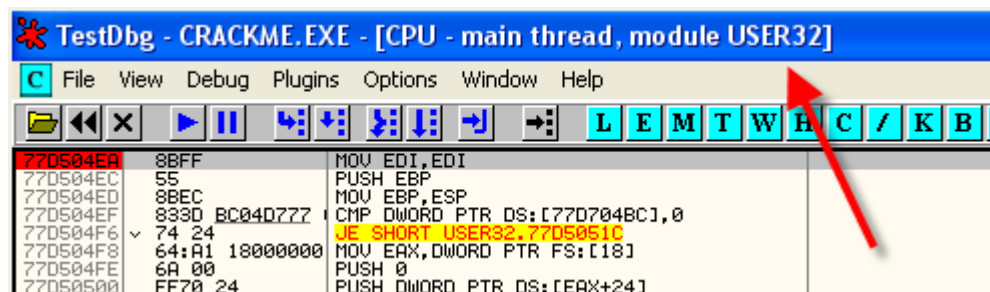
什么是 IAT:

我们知道每个 API 函数在对应的进程空间中都有其相应的入口地址,例如:我们用 OD 加载 Cruehead 的 CrackMe,在命令栏中输入? MessageBoxA



大家可以看到在我的机器上,MessageBoxA 这个 API 函数的地址为 77D504EA,如果大家在自己的机器上面定位到这个地址的话,可能有部分人的机器上该地址对应的还是 MessageBoxA 的入口地址,而另外一部分人的机器上该地址对应的就不是 MessageBoxA 的入口地址了,这取决于大家机器的操作系统版本,以及打补丁的情况。众所周知,操作系统动态库版本的更新,其包含的 API 函数入口地址通常也会改变。

比如 User32.dll

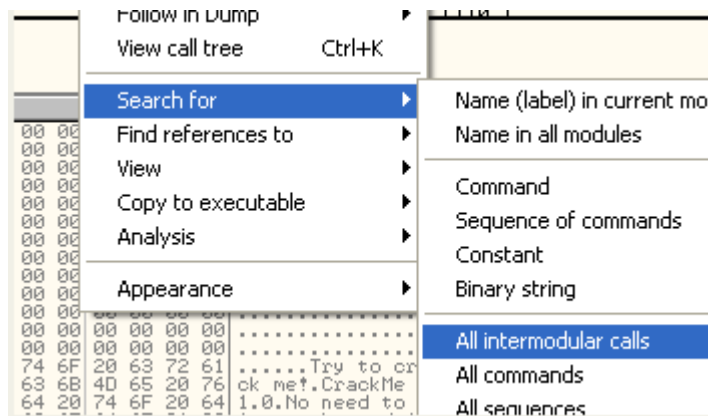


我们就拿 Cruehead 的 CrackMe 中的 MessageBoxA 这个 API 函数来说吧,其入口地址为 77D504EA,在我的机器上运行的很好,那些跟我操作系统版本以及 User32.dll 版本相同的童鞋的机器上该程序运行可能也很正常,但是如果在操作系统版本或者 User32.dll 的版本跟我的不同童鞋的机器上运行,可能就会出错。

为了解决以上兼容问题,操作系统就必须提供一些措施来确保该 CrackMe 可以在其他版本的 Windows 操作系统,以及 DLL 版本下也能正常运行。

这时 IAT(Import Address Table:输入函数地址表)就应运而生了。

大家不要觉得其名字很霸气,就会问是不是很难?其实不然。接下来我们一起来探讨一下如何在脱壳过程中定位 IAT。



我们现在通过在反汇编窗口中单击鼠标右键选择-Search for-All intermodular calls 来看看主模块中调用了哪些模块以及 API 函数。

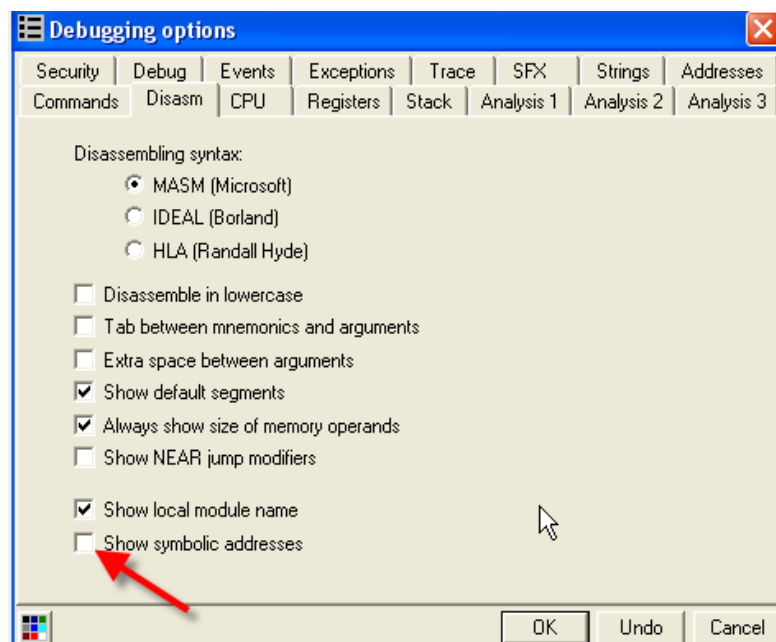
Address	Disassembly	Destination
00401000	PUSH 0	(Initial CPU selection)
00401002	CALL <JMP.&KERNEL32.GetModuleHandleA>	kernel32.GetModuleHandleA
00401013	CALL <JMP.&USER32.FindWindowA>	USER32.FindWindowA
00401052	CALL <JMP.&USER32.LoadIconA>	USER32.LoadIconA
00401063	CALL <JMP.&USER32.LoadCursorA>	USER32.LoadCursorA
00401090	CALL <JMP.&USER32.RegisterClassA>	USER32.RegisterClassA
004010C3	CALL <JMP.&USER32.CreateWindowExA>	USER32.CreateWindowExA
004010D5	CALL <JMP.&USER32.ShowWindow>	USER32.ShowWindow
004010E0	CALL <JMP.&USER32.UpdateWindow>	USER32.UpdateWindow
004010EC	CALL <JMP.&USER32.InvalidateRect>	USER32.InvalidateRect
004010FC	CALL <JMP.&USER32.GetMessageA>	USER32.GetMessageA
0040110C	CALL <JMP.&USER32.TranslateMessage>	USER32.TranslateMessage
00401116	CALL <JMP.&USER32.DispatchMessageA>	USER32.DispatchMessageA
00401123	CALL <JMP.&KERNEL32.ExitProcess>	kernel32.ExitProcess
0040118C	CALL <JMP.&USER32.DefWindowProcA>	USER32.DefWindowProcA
00401195	CALL <JMP.&USER32.PostQuitMessage>	USER32.PostQuitMessage
00401202	CALL <JMP.&USER32.ShowDialogBoxParamA>	USER32.ShowDialogBoxParamA
0040121E	CALL <JMP.&USER32.ShowDialogBoxParamA>	USER32.ShowDialogBoxParamA
00401292	CALL <JMP.&USER32.InvalidateRect>	USER32.InvalidateRect
0040129A	CALL <JMP.&USER32.SetFocus>	USER32.SetFocus
004012C4	CALL <JMP.&USER32.GetDlgItemTextA>	USER32.GetDlgItemTextA
004012E4	CALL <JMP.&USER32.GetDlgItemTextA>	USER32.GetDlgItemTextA
004012FB	CALL <JMP.&USER32.EndDialog>	USER32.EndDialog
0040133A	CALL <JMP.&USER32.EndDialog>	USER32.EndDialog
0040135C	CALL <JMP.&USER32.MessageBoxA>	USER32.MessageBoxA
00401364	CALL <JMP.&USER32.MessageBeep>	USER32.MessageBeep
00401378	CALL <JMP.&USER32.MessageBoxA>	USER32.MessageBoxA
004013BC	CALL <JMP.&USER32.MessageBoxA>	USER32.MessageBoxA

这里我们可以看到有几处调用了 MessageBoxA，我们在第一个 MessageBoxA 调用处双击鼠标左键。

00401346	> B8 00000000	MOV EAX,0	
00401348	EB 08	JMP SHORT CRACKME.00401325	
0040134D	6A 30	PUSH 30	
0040134F	68 29214000	PUSH CRACKME.00402129	
00401354	68 34214000	PUSH CRACKME.00402134	
00401359	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
0040135C	E8 09000000	CALL <JMP.&USER32.MessageBoxA>	MessageBoxA
00401361	C3	RETN	
00401362	6A 00	PUSH 0	
00401364	E8 AD000000	CALL <JMP.&USER32.MessageBeep>	MessageBeep
00401369	6A 30	PUSH 30	
0040136B	68 60214000	PUSH CRACKME.00402160	
00401370	68 63214000	PUSH CRACKME.00402169	
00401375	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
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	
00401383	8A06	MOV AL,BYTE PTR DS:[ESI]	
00401385	84C0	TEST AL,AL	
00401387	74 13	JE SHORT CRACKME.0040139C	
0040143A	<JMP.&USER32.MessageBoxA>		

反汇编窗口就会马上定位到该调用处,OD 提示窗口中显示其实际调用的是 40143A 处的 JMP.&USER32.MessageBoxA,这里用尖括号括起来了,说明这里是直接调用,而非间接调用。

这里其实就是 CALL 40143A,显示为 CALL <JMP.&USER32.MessageBoxA>大家可能会觉得不太直观。这里我们打开 Debugging options 菜单项:



Disasm 标签页中的 Show symbolic addresses 选项被勾选上了,如果我们去掉该对勾,将不会显示函数地址。

00401335	> 5A 00	PUSH 0	Result = 0
00401337	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
0040133A	E8 73010000	CALL 004014B2	EndDialog
0040133F	B8 01000000	MOV EAX,1	
00401344	E8 DF	JMP SHORT 00401325	CRACKME.00401325
00401346	B8 00000000	MOV EAX,0	
00401348	E8 D8	JMP SHORT 00401325	CRACKME.00401325
0040134D	6A 30	PUSH 30	Style = MB_OK MB_ICONEXCLAMATI
0040134F	68 29214000	PUSH 402129	Title = "Good work!"
00401354	68 34214000	PUSH 402134	Text = "Great work, mate! Now
00401359	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
0040135C	E8 D9000000	CALL 0040143A	MessageBoxA
00401361	C3	RETN	
00401362	6A 00	PUSH 0	BeepType = MB_OK
00401364	E8 AD000000	CALL 00401416	MessageBeep
00401369	6A 30	PUSH 30	Style = MB_OK MB_ICONEXCLAMATI
0040136B	68 60214000	PUSH 402160	Title = "No luck!"
00401370	68 63214000	PUSH 402169	Text = "No luck there, mate!"
00401375	FF75 08	PUSH DWORD PTR SS:[EBP+8]	hOwner
00401378	E8 BD000000	CALL 0040143A	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 0040139C	CRACKME.0040139C

0040143A=<JMP.&USER32.MessageBoxA>

我们可以看到右边的注释窗口中同样显示了 API 函数的参数以及函数名称,比刚刚显示符号地址看起来更直观,一眼就可以看出是一个直接调用。

CALL 40143A

在 Search for All intermodular calls 窗口中显示如下:

Address	Disassembly	Description
00401000	PUSH 0	(Initial CPU selection)
00401002	CALL 00401506	kernel32.GetModuleHandleA
00401013	CALL 0040148E	USER32.FindWindowA
00401052	CALL 00401428	USER32.LoadIconA
00401063	CALL 0040140A	USER32.LoadCursorA
00401090	CALL 00401488	USER32.RegisterClassA
004010C3	CALL 00401494	USER32.CreateWindowExA
004010D5	CALL 0040146A	USER32.ShowWindow
004010E0	CALL 00401482	USER32.UpdateWindow
004010EC	CALL 0040144C	USER32.InvalidateRect
004010FC	CALL 004014D6	USER32.GetMessageA
0040110C	CALL 0040145E	USER32.TranslateMessage
00401116	CALL 004014A6	USER32.DispatchMessageA
00401123	CALL 00401512	kernel32.ExitProcess
0040118C	CALL 0040149A	USER32.DefWindowProcA
00401195	CALL 00401440	USER32.PostQuitMessage
00401202	CALL 004014A0	USER32.ShowDialogBoxParamA
0040121E	CALL 004014A0	USER32.ShowDialogBoxParamA
00401292	CALL 0040144C	USER32.InvalidateRect
0040129A	CALL 00401434	USER32.SetFocus
004012C4	CALL 004014D0	USER32.GetDlgItemTextA
004012E4	CALL 004014D0	USER32.GetDlgItemTextA
004012FB	CALL 004014B2	USER32.EndDialog
0040133A	CALL 004014B2	USER32.EndDialog
0040135C	CALL 0040143A	USER32.MessageBoxA
00401364	CALL 00401416	USER32.MessageBeep
00401378	CALL 0040143A	USER32.MessageBoxA
004013BC	CALL 0040143A	USER32.MessageBoxA

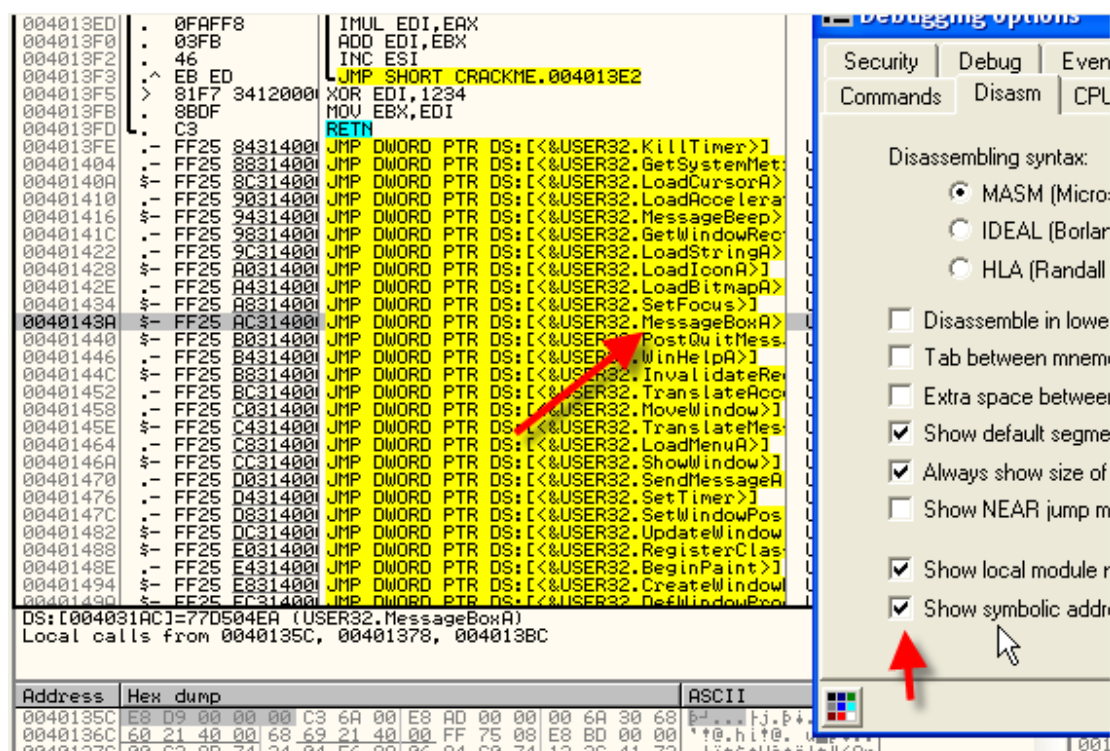
我们可以看到这里有三处通过 CALL 40143A 调用 MessageBoxA,我们定位到 40143A 处看看是什么。

00401428	FF25 A0314000	JMP DWORD PTR DS:[4031A0]	USER32.LoadIconA
0040142E	FF25 A4314000	JMP DWORD PTR DS:[4031A4]	USER32.LoadBitmapA
00401434	FF25 A8314000	JMP DWORD PTR DS:[4031A8]	USER32.SetFocus
0040143A	FF25 AC314000	JMP DWORD PTR DS:[4031AC]	USER32.MessageBoxA
00401440	FF25 B0314000	JMP DWORD PTR DS:[4031B0]	USER32.PostQuitMessage
00401448	FF25 B4314000	JMP DWORD PTR DS:[4031B4]	USER32.WinHelpA
00401450	FF25 B8314000	JMP DWORD PTR DS:[4031B8]	USER32.InvalidateRect
00401458	FF25 BC314000	JMP DWORD PTR DS:[4031BC]	USER32.TranslateAcceleratorA
0040145E	FF25 C0314000	JMP DWORD PTR DS:[4031C0]	USER32.MoveWindow
00401464	FF25 C4314000	JMP DWORD PTR DS:[4031C4]	USER32.TranslateMessage
0040146A	FF25 C8314000	JMP DWORD PTR DS:[4031C8]	USER32.LoadMenuA
00401470	FF25 CC314000	JMP DWORD PTR DS:[4031CC]	USER32.ShowWindow
00401476	FF25 D0314000	JMP DWORD PTR DS:[4031D0]	USER32.SendMessageA
0040147C	FF25 D4314000	JMP DWORD PTR DS:[4031D4]	USER32.SetTimer
00401482	FF25 D8314000	JMP DWORD PTR DS:[4031D8]	USER32.SetWindowPos
00401488	FF25 DC314000	JMP DWORD PTR DS:[4031DC]	USER32.UpdateWindow
0040148E	FF25 E0314000	JMP DWORD PTR DS:[4031E0]	USER32.RegisterClassA
00401494	FF25 E4314000	JMP DWORD PTR DS:[4031E4]	USER32.BeginPaint
0040149A	FF25 E8314000	JMP DWORD PTR DS:[4031E8]	USER32.CreateWindowExA
004014A0	FF25 EC314000	JMP DWORD PTR DS:[4031EC]	USER32.DefWindowProcA

DS:[004031AC]=77D504EA (USER32.MessageBoxA)
Local calls from 0040135C, 00401378, 004013BC

这里我们可以看到是一个间接跳转。即

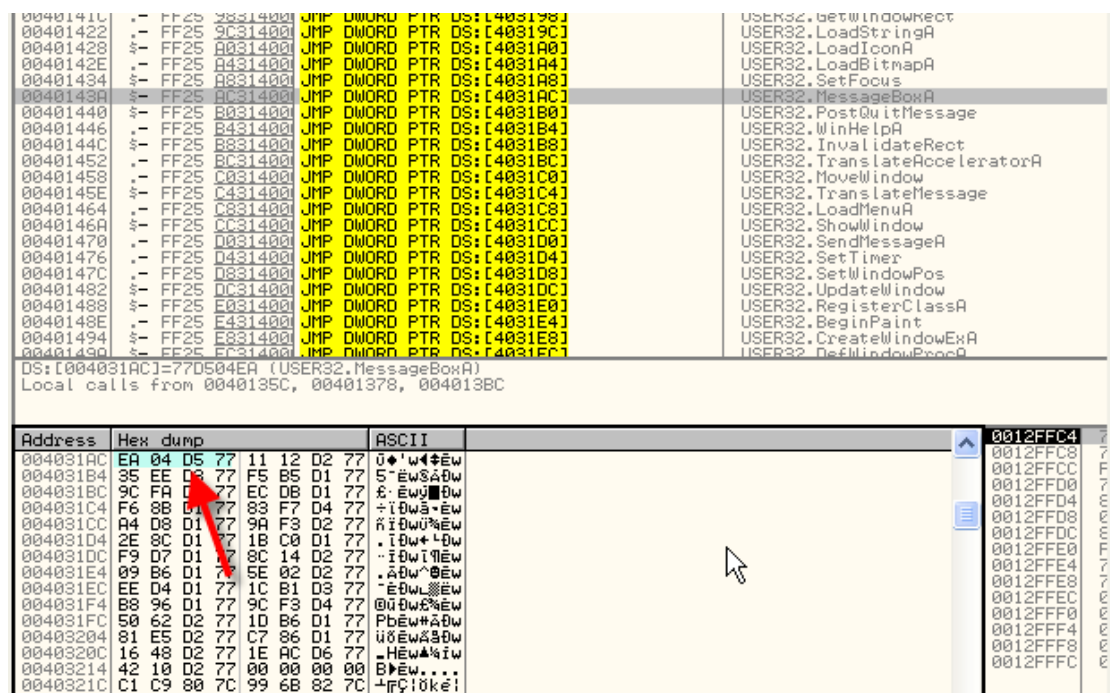
JMP [4031AC]



这里我们再次勾选上显示符号地址的选项,可以更加直观的看出其调用的 API 函数。

这里有意思的地方就来了,我们看到 JMP [4031AC](4031AC 这个内存单元中保存的数值才是 MessageBoxA 真正的入口地址)。我们还可以看到很多类似的间接 JMP。

这就是为了解决各操作系统之间的兼容问题而设计的,当程序需要调用某个 API 函数的时候,都是通过一个间接跳转来调用的,读取某个地址中保存的 API 函数地址,然后调用之。我们现在在数据窗口中定位到 4031AC 地址处,看看该内存单元中存放的是什么。

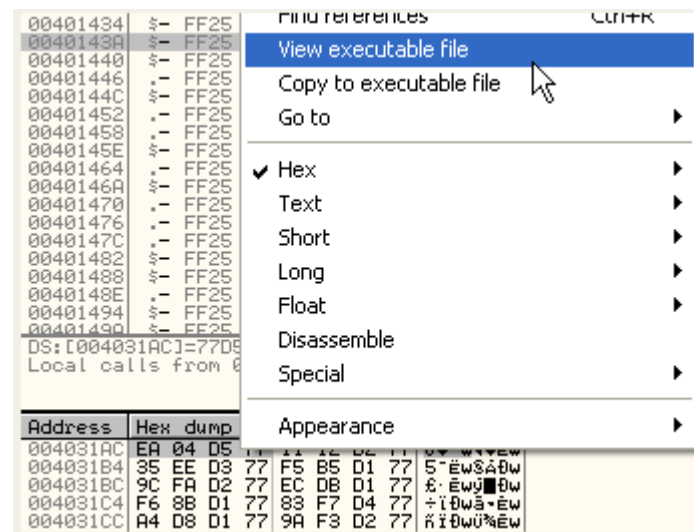


这里我们可以看到,4031AC 中保存的是 77D504EA,这一片区域包含了该程序调用的所有 API 函数的入口地址,这块区域我们称之为 IAT(导入函数地址表),这里就是解决不同版本操作系统间调用 API 兼容问题的关键所在,该程序在不同版本操作系统上都是调用间接跳转到 IAT 表中,在 IAT 中读取到真正的 API 函数入口地址,然后调用之,所以说只需要将不同系统中的 API 函数地址填充

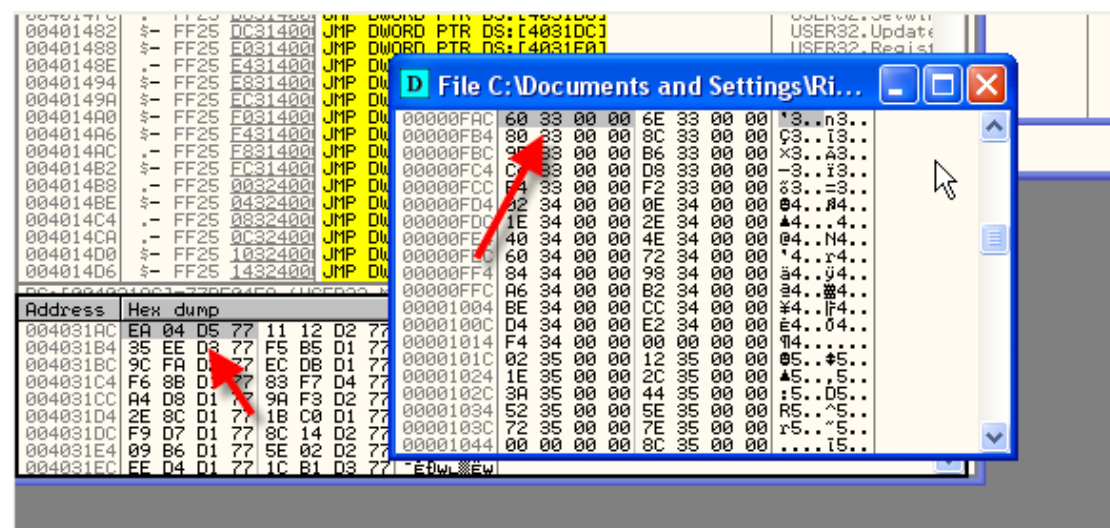
到 IAT 中,这样就可以确保不同版本系统调用的都是正确的 API 函数。

有些人可能会问,4031AC 这个地址在不同机器上也可能会不同的吧?

呵呵,这个问题提的非常好,我们一起来看看操作系统将正确的 API 函数入口地址填充到 IAT 中的具体原理,大家就会明白了。



这里我们选中 4031AC 中保存的内容,单击鼠标右键选择-View executable file,就能看到 4031AC 这个虚拟地址对应于可执行文件中的文件偏移是多少了。



我们可以看到在可执行文件对应文件偏移处的内容为 60 33 00 00,当程序运行起来的时候,0FAC 这个文件偏移对应的虚拟地址处就会被填充为 EA 04 D5 77,也就是说该 CrackMe 进程空间中的 4031AC 地址处会被填入正确的 API 函数地址。

有这么神奇?

Windows 操作系统当可执行文件被加载到进程所在内存空间中时,会将正确的 API 函数地址填充到 IAT 中,这里就是 4031AC 中被填入了 MessageBoxA 的入口地址,其他 IAT 项也会被填入对应的 API 函数地址。

其实操作系统并没有大家想象得的那么神奇,我们看到 0FAC 文件偏移处的值 3360,该数值其实是 RVA(相对虚拟地址),其指向对应的 API 函数名称。

这里 3360 加上映像基址即 403360,我们定位到 403360 处,看看是什么。

Address	Hex dump	ASCII
00403360	00 00 40 65 73 73 61 67	..Messag
00403368	65 42 6F 78 41 00 00 00	eBoxA..
00403370	50 6F 73 74 51 75 69 74	PostQuit
00403378	40 65 73 73 61 67 65 00	Message.
00403380	00 00 57 69 6E 48 65 6C	..WinHel
00403388	70 41 00 00 00 00 49 6E	pA....In
00403390	76 61 6C 69 64 61 74 65	validate
00403398	52 65 63 74 00 00 00 00	Rect....
004033A0	54 72 61 6E 73 6C 61 74	Translat
004033A8	65 41 63 63 65 6C 65 72	eAcceler
004033B0	61 74 6F 72 41 00 00 00	atorA...
004033B8	40 6F 76 65 57 69 6E 64	MoveWind
004033C0	6F 77 00 00 00 00 54 72	ow....Tr
004033C8	61 6E 73 6C 61 74 65 4D	anslateM
004033D0	65 73 73 61 67 65 00 00	essage..
004033D8	00 00 4C 6F 61 64 4D 65	..LoadMe
004033E0	6E 75 41 00 00 00 53 68	nuA...Sh
004033E8	6F 77 57 69 6E 64 6F 77	owWindow
004033F0	00 00 00 00 53 65 6E 64Send
004033F8	4D 65 73 73 61 67 65 41	MessageA
00403400	00 00 00 00 53 65 74 54SetT
00403408	69 6D 65 72 00 00 00 00	imer....

这里我们可以看到指向的是 MessageBoxA 这个字符串,也就是说操作系统可以根据这个指针,定位到相应的 API 函数名称,然后通过调用 GetProcAddress 获取对应 API 函数的地址,然后将该地址填充到 IAT 中,覆盖原来的 3360。这样就能保证在程序执行前,IAT 中被填充了正确的 API 函数地址。如果我们换一台机器,定位到 4031AC 处,可能会看到里面存放着不同的地址。

JMP [4031AC]

这样就能够调用 MessageBoxA 了,大家可能会觉得这个过程很复杂,其实填充 IAT 的过程都是操作系统帮我们完成的,在程序开始执行前,IAT 已经被填入了正确的 API 函数地址。

也就是说,为了确保操作系统将正确的 API 函数地址填充到 IAT 中,应该满足一下几点要求:

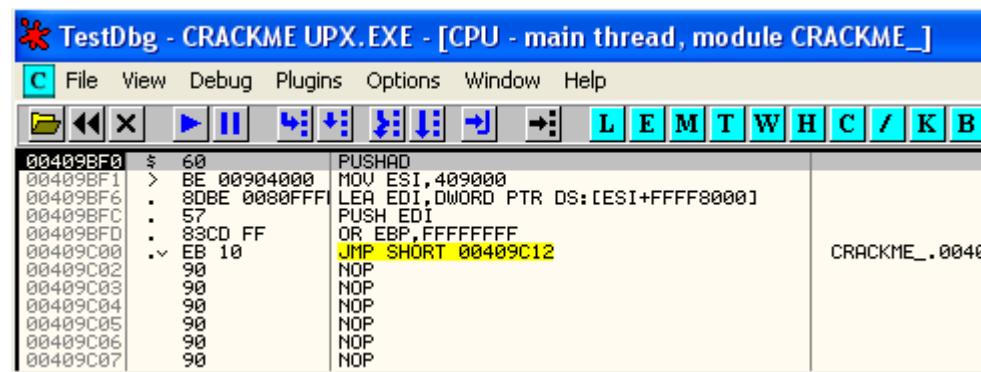
- 1:可执行文件各 IAT 项所在的文件偏移处必须是一个指针,指向一个字符串。
- 2:该字符串为 API 函数的名称。

如果这两项满足,就可以确保程序在启动时,操作系统会将正确的 API 函数地址填充到 IAT 中(后面会详细介绍操作系统是如何填充 IAT 的)。

假如,我们当前位于被加壳程序的 OEP 处,我们接下来可以将程序 dump 出来,但是在 dump 之前我们必须修复 IAT,为什么要修复 IAT 呢?难道壳将 IAT 破坏了吗?对,的确是这样,壳压根不需要原程序的 IAT,因为被加壳程序首先会执行解密例程,读取 IAT 中所需要的 API 的名称指针,然后定位到 API 函数地址,将其填入到 IAT 中,这个时候,IAT 中已经被填充了正确的 API 函数地址,对应的 API 函数名称的字符串已经不需要了,可以清除掉。

大部分的壳会将 API 函数名称对应的字符串以密文的形式保存到某个地址处,让 Cracker 们不能那么容易找到它们。

下面我们来看看 CrackMe UPX 这个程序,在 dump 之前我们需要修复 IAT。



我们定位到 4031AC 处-原程序 MessageBoxA 入口地址的存放处。

Address	Hex dump	ASCII
004031AC	00 00 00 00 00 00 00 00
004031B4	00 00 00 00 00 00 00 00
004031BC	00 00 00 00 00 00 00 00
004031C4	00 00 00 00 00 00 00 00
004031CC	00 00 00 00 00 00 00 00
004031D4	00 00 00 00 00 00 00 00
004031DC	00 00 00 00 00 00 00 00
004031E4	00 00 00 00 00 00 00 00
004031EC	00 00 00 00 00 00 00 00
004031F4	00 00 00 00 00 00 00 00
004031FC	00 00 00 00 00 00 00 00
00403204	00 00 00 00 00 00 00 00
0040320C	00 00 00 00 00 00 00 00
00403214	00 00 00 00 00 00 00 00
0040321C	00 00 00 00 00 00 00 00
00403224	00 00 00 00 00 00 00 00
0040322C	00 00 00 00 00 00 00 00
00403234	00 00 00 00 00 00 00 00
0040323C	00 00 00 00 00 00 00 00
00403244	00 00 00 00 00 00 00 00
0040324C	00 00 00 00 00 00 00 00
00403254	00 00 00 00 00 00 00 00
0040325C	00 00 00 00 00 00 00 00
00403264	00 00 00 00 00 00 00 00
0040326C	00 00 00 00 00 00 00 00
00403274	00 00 00 00 00 00 00 00
0040327C	00 00 00 00 00 00 00 00
00403284	00 00 00 00 00 00 00 00
0040328C	00 00 00 00 00 00 00 00

是空的,那么 403360 指向的字符串呢?

Address	Hex dump	ASCII
00403360	00 00 00 00 00 00 00 00
00403368	00 00 00 00 00 00 00 00
00403370	00 00 00 00 00 00 00 00
00403378	00 00 00 00 00 00 00 00
00403380	00 00 00 00 00 00 00 00
00403388	00 00 00 00 00 00 00 00
00403390	00 00 00 00 00 00 00 00
00403398	00 00 00 00 00 00 00 00
004033A0	00 00 00 00 00 00 00 00
004033A8	00 00 00 00 00 00 00 00
004033B0	00 00 00 00 00 00 00 00
004033B8	00 00 00 00 00 00 00 00
004033C0	00 00 00 00 00 00 00 00
004033C8	00 00 00 00 00 00 00 00
004033D0	00 00 00 00 00 00 00 00
004033D8	00 00 00 00 00 00 00 00
004033E0	00 00 00 00 00 00 00 00
004033E8	00 00 00 00 00 00 00 00
004033F0	00 00 00 00 00 00 00 00
004033F8	00 00 00 00 00 00 00 00
00403400	00 00 00 00 00 00 00 00

也是空的,我们跟到 OEP 处,再看看这几个地址处有没有内容,我们知道原程序在运行之前,IAT 必须被填充上正确的 API 函数地址。

JMP [4031AC]

如果此时 IAT 还是空的话,那么程序运行起来就会出错,我们现在定位到 OEP。

00409D2D	. 09C0	OR EAX,EAX	
00409D2F	. 74 07	JE SHORT 00409D38	CRACKME_.00409D38
00409D31	. 8903	MOV DWORD PTR DS:[EBX],EAX	
00409D33	. 83C3 04	ADD EBX,4	
00409D36	. EB E1	JMP SHORT 00409D19	CRACKME_.00409D19
00409D38	> FF96 6095000	CALL DWORD PTR DS:[ESI+9560]	
00409D3E	> 61	POPAD	
00409D3F	.- E9 BC72FFFF	JMP 00401000	CRACKME_.00401000
00409D44	00	DB 00	
00409D45	00	DB 00	
00409D46	00	DB 00	
00409D47	00	DB 00	
00409D48	00	DB 00	
00409D49	00	DB 00	
00409D4A	00	DB 00	

我们在这个 JMP OEP 指令处设置一个断点,运行起来,接着来看看 IAT:

0040142E	- FF25 44314000	JMP DWORD PTR DS:[4031A4]	USER32.LoadBitmapH
00401434	- FF25 A8314000	JMP DWORD PTR DS:[4031A8]	USER32.SetFocus
0040143A	- FF25 AC314000	JMP DWORD PTR DS:[4031AC]	USER32.MessageBoxA
00401440	- FF25 B0314000	JMP DWORD PTR DS:[4031B0]	USER32.PostQuitMessage
00401446	- FF25 B4314000	JMP DWORD PTR DS:[4031B4]	USER32.WinHelpA
0040144C	- FF25 B8314000	JMP DWORD PTR DS:[4031B8]	USER32.InvalidRect
00401452	- FF25 BC314000	JMP DWORD PTR DS:[4031BC]	USER32.TranslateAccelerator
00401458	- FF25 C0314000	JMP DWORD PTR DS:[4031C0]	USER32.MoveWindow
0040145E	- FF25 C4314000	JMP DWORD PTR DS:[4031C4]	USER32.TranslateMessage
00401464	- FF25 C8314000	JMP DWORD PTR DS:[4031C8]	USER32.LoadMenuA
0040146A	- FF25 CC314000	JMP DWORD PTR DS:[4031CC]	USER32.ShowWindow
00401470	- FF25 D0314000	JMP DWORD PTR DS:[4031D0]	USER32.SendMessageA
00401476	- FF25 D4314000	JMP DWORD PTR DS:[4031D4]	USER32.SetTimer
0040147C	- FF25 D8314000	JMP DWORD PTR DS:[4031D8]	USER32.SetWindowPos
00401482	- FF25 DC314000	JMP DWORD PTR DS:[4031DC]	USER32.UpdateWindow
00401488	- FF25 E0314000	JMP DWORD PTR DS:[4031E0]	USER32.RegisterClassA
0040148E	- FF25 E4314000	JMP DWORD PTR DS:[4031E4]	USER32.BeginPaint
00401494	- FF25 E8314000	JMP DWORD PTR DS:[4031E8]	USER32.CreateWindowExA
0040149A	- FF25 EC314000	JMP DWORD PTR DS:[4031EC]	USER32.DefWindowProcA
004014A0	- FF25 F0314000	JMP DWORD PTR DS:[4031F0]	USER32.DialogBoxParamA
004014A6	- FF25 F4314000	JMP DWORD PTR DS:[4031F4]	USER32.DispatchMessageA
004014AC	- FF25 F8314000	JMP DWORD PTR DS:[4031F8]	USER32.DrawMenuBar
004014B2	- FF25 FC314000	JMP DWORD PTR DS:[4031FC]	USER32.EndDialog
004014B8	- FF25 00324000	JMP DWORD PTR DS:[403200]	USER32.EndPaint
004014BE	- FF25 04324000	JMP DWORD PTR DS:[403204]	USER32.FindWindowA

DS:[004031AC]=77D504EA (USER32.MessageBoxA)

Address	Hex dump	ASCII
004031AC	EA 04 05 77 11 12 02 77	04'w4577
004031B4	35 EE 03 77 F5 B5 01 77	5-ew35B5w
004031BC	9C FA 02 77 EC 0B 01 77	9C-ew9CEBw
004031C4	F6 8B 01 77 83 F7 04 77	+iBw83F7w
004031CC	A4 08 01 77 9A F3 02 77	A4Bw9AF3w
004031D4	2E 8C 01 77 1B C0 01 77	.iBw+1Bw
004031DC	F9 07 01 77 8C 14 02 77	-iBw14w

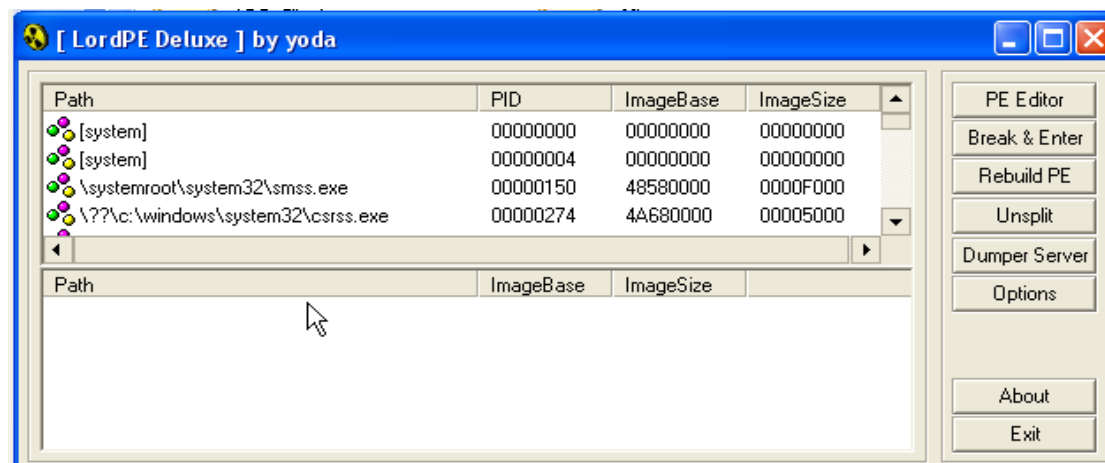
我们可以看到壳的解密例程已经将正确的 API 函数地址填充到原程序的 IAT 中,如果这个时候我们将程序 dump 出来的话,运行会出错,因为 dump 出来的程序启动所必须的数据是不完整的。

我们现在来看看各个 API 函数名称,定位到 403360 处,会发现是空的。

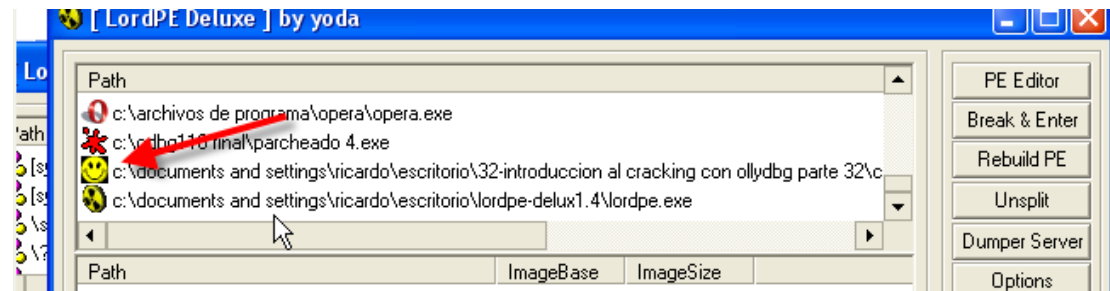
Address	Hex dump	ASCII
00403360	00 00 00 00 00 00 00 00
00403368	00 00 00 00 00 00 00 00
00403370	00 00 00 00 00 00 00 00
00403378	00 00 00 00 00 00 00 00
00403380	00 00 00 00 00 00 00 00
00403388	00 00 00 00 00 00 00 00
00403390	00 00 00 00 00 00 00 00
00403398	00 00 00 00 00 00 00 00
004033A0	00 00 00 00 00 00 00 00
004033A8	00 00 00 00 00 00 00 00
004033B0	00 00 00 00 00 00 00 00
004033B8	00 00 00 00 00 00 00 00
004033C0	00 00 00 00 00 00 00 00
004033C8	00 00 00 00 00 00 00 00
004033D0	00 00 00 00 00 00 00 00
004033D8	00 00 00 00 00 00 00 00
004033E0	00 00 00 00 00 00 00 00
004033E8	00 00 00 00 00 00 00 00
004033F0	00 00 00 00 00 00 00 00

现在我们 dump 出来看看,dump 出来的原程序代码肯定是正确的,但是程序仍然无法正常运行,因为缺少数据,操作系统无法填充 IAT。

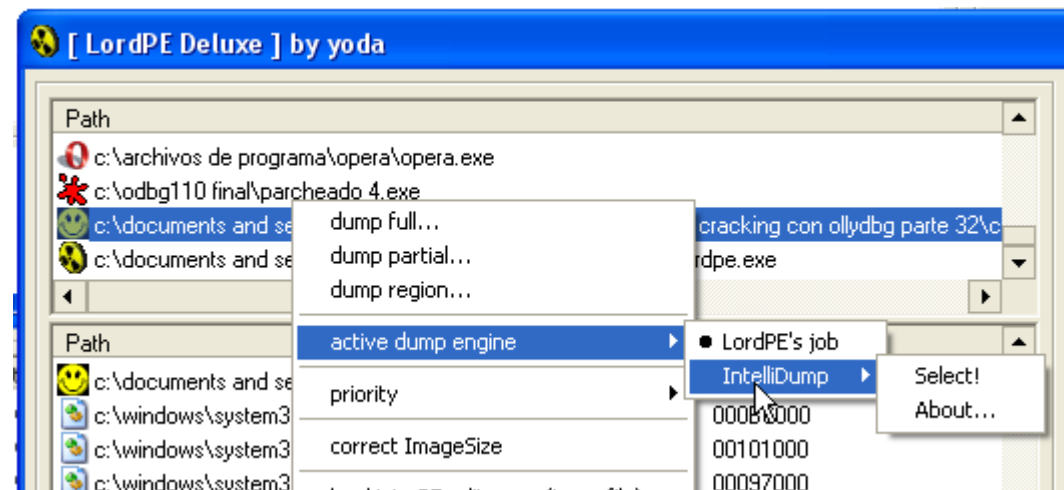
Dump 的话我们需要用到一个工具,名字叫做 LordPE(PS:大家应该用的很多吧)。



我们运行 LordPE,定位到需要 dump 的 CRACKME UPX 所在的进程,当前该进程处于 OEP 处。



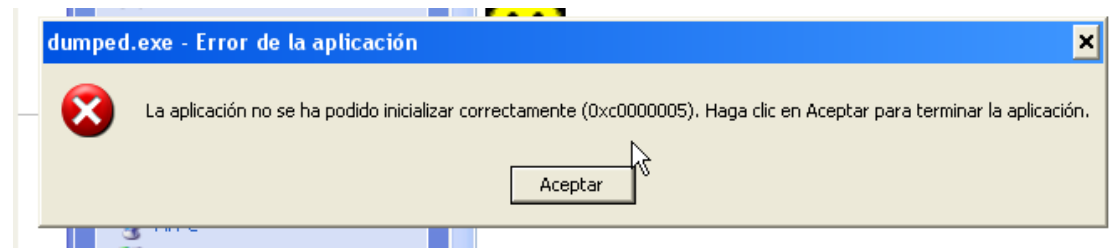
选中 CRACKME UPX 所在的进程。



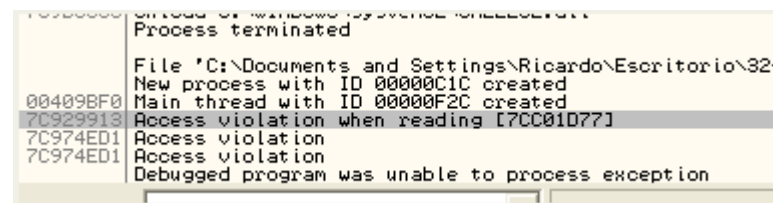
我们单击鼠标右键选择-active dump engine-IntelliDump-Select!。接着选择 dump full。



我们将 dump 出来的程序命名为 dumped.exe。



如果我们直接运行 dumped.exe 的话会发现无法启动,尝试用 OD 加载 dumped.exe,OD 会报错,我们来看看日志窗口中的错误信息。



这里我们机器上提示错误发现在 7C929913 地址处,我们定位到该地址(大家可以根据自己机器上显示的错误地址自行定位)。

7C929908	03C8	ADD ECX,EAX	
7C92990A	8B46 20	MOV EAX,DWORD PTR DS:[ESI+20]	
7C92990D	0345 08	ADD EAX,DWORD PTR SS:[EBP+8]	
7C929910	8D59 02	LEA EBX,DWORD PTR DS:[ECX+2]	
7C929913	0FB709	MOVZX ECX,WORD PTR DS:[ECX]	
7C929916	3B4E 18	CMP ECX,DWORD PTR DS:[ESI+18]	
7C929919	895D 0C	MOV DWORD PTR SS:[EBP+C],EBX	
7C92991C	894D FC	MOV DWORD PTR SS:[EBP-4],ECX	
7C92991F	0F83 A0020000	JNB 7C929BC5	
7C929925	8B0C88	MOV ECX,DWORD PTR DS:[EAX+ECX*4]	
7C929928	034D 08	ADD ECX,DWORD PTR SS:[EBP+8]	
7C92992B	895D 18	MOV DWORD PTR SS:[EBP+18],EBX	
7C92992E	8B55 18	MOV EDX,DWORD PTR SS:[EBP+18]	
7C929931	8A1A	MOV BL,BYTE PTR DS:[EDX]	
7C929933	8AD3	MOV DL,BL	

这里我们可以给这一行设置一个硬件执行断点或者 INT 3 断点,即当断在这一行时看看错误发生之前是什么状况。

我们运行起来,会发现没有断在这一行,这是因为勾选了忽略异常选项的缘故,这里我们去掉忽略异常选项的对勾,重新运行起来。

C File View Debug Plugins Options Window Help			
[Icons] [Buttons] L E M T W H C / K E			
7C929908	03C8	ADD ECX,EAX	
7C92990A	8B46 20	MOV EAX,DWORD PTR DS:[ESI+20]	
7C92990D	0345 08	ADD EAX,DWORD PTR SS:[EBP+8]	
7C929910	8D59 02	LEA EBX,DWORD PTR DS:[ECX+2]	
7C929913	0FB709	MOVZX ECX,WORD PTR DS:[ECX]	
7C929916	3B4E 18	CMP ECX,DWORD PTR DS:[ESI+18]	
7C929919	895D 0C	MOV DWORD PTR SS:[EBP+C],EBX	
7C92991C	894D FC	MOV DWORD PTR SS:[EBP-4],ECX	
7C92991F	0F83 A0020000	JNB 7C929BC5	
7C929925	8B0C88	MOV ECX,DWORD PTR DS:[EAX+ECX*4]	
7C929928	034D 08	ADD ECX,DWORD PTR SS:[EBP+8]	
7C92992B	895D 18	MOV DWORD PTR SS:[EBP+18],EBX	
7C92992E	8B55 18	MOV EDX,DWORD PTR SS:[EBP+18]	
7C929931	8A1A	MOV BL,BYTE PTR DS:[EDX]	
7C929933	8AD3	MOV DL,BL	

断了下来,我们可以看到该错误是在到达入口点之前产生的,所以 dumped.exe 无法正常运行,我们现在来看看 IAT 的情况。

Address	Hex dump	ASCII
004031AC	EA 04 05 77 11 12 D2 77	04'w4&Ew
004031B4	35 EE D3 77 F5 B5 D1 77	5'EwS4Bw
004031BC	9C FA D2 77 EC D8 D1 77	8:EwW4Bw
004031C4	F6 8B D1 77 83 F7 D4 77	+iBw3-Ew
004031CC	A4 D8 D1 77 9A F3 D2 77	8iBwU%EW
004031D4	2E 8C D1 77 1B C0 D1 77	-iBw+&Bw
004031DC	F9 D7 D1 77 8C 14 D2 77	-iBwI%EW
004031E4	09 B6 D1 77 5E 02 D2 77	.ABw^EW
004031EC	EE D4 D1 77 1C B1 D3 77	EBwL%EW
004031F4	B8 96 D1 77 9C F3 D4 77	00BwE%EW
004031FC	50 62 D2 77 1D B6 D1 77	PbEW#ABw
00403204	81 E5 D2 77 C7 86 D1 77	u8EW\$Bw
0040320C	16 48 D2 77 1E AC D6 77	-HEwA%iW
00403214	42 10 D2 77 00 00 00 00	BtEW....
0040321C	C1 C9 80 7C 99 6B 82 7C	4fC!0kE!
00403224	2F FE 80 7C 2D FF 80 7C	/wC!-C!
0040322C	E0 C6 80 7C 77 9B 80 7C	08C!wC!
00403234	9F AF 81 7C 29 A5 8A 7C	*wi!AC!

我们可以看到当前虽然在我的机器上各个 API 函数的地址被填充到 IAT 中,但是想要正常运行在其他机器上的话,必须要指向各个 API 函数名称字符串的指针,这样才是确保操作系统能够通过 GetProcAddress 获取到正确的 API 函数地址并填充到 IAT 中。

这里该 dumped.exe 缺少这些指向 API 函数名称字符串的指针,所以运行的时候会发生错误。

这里大家不要尝试先 dump 出来,然后再恢复各个 API 函数的名称字符串以及其指针,如果这样手工修复的话,是一件极其困难的工作,你需要将 4031AC 地址处的内容修改为 MessageBoxA 这个字符串的指针,IAT 中的其他项也要进行相应的处理。

比较明智的做法是,dump 出来之前就将 IAT 修复了。

我们知道 dump 出来的代码肯定是正确的,我们定位到 401000 处看一看。

00401000	6A 00	PUSH 0	
00401002	E8 FF040000	CALL 00401506	
00401007	A3 CA204000	MOV DWORD PTR DS:[4020CA],EAX	
0040100C	6A 00	PUSH 0	
0040100E	68 F4204000	PUSH 4020F4	
00401013	E8 A6040000	CALL 004014BE	
00401018	0BC0	OR EAX,EAX	
0040101A	74 01	JE SHORT 0040101D	
0040101C	C3	RETN	
0040101D	C705 64204000	MOV DWORD PTR DS:[402064],4003	
00401027	C705 68204000	MOV DWORD PTR DS:[402068],401128	
00401031	C705 6C204000	MOV DWORD PTR DS:[40206C],0	
0040103B	C705 70204000	MOV DWORD PTR DS:[402070],0	
00401045	A1 CA204000	MOV EAX,DWORD PTR DS:[4020CA]	
0040104A	A3 F4204000	MOV DWORD PTR DS:[402074],EAX	
0040104F	6A 64	PUSH 64	
00401051	50	PUSH EAX	
00401052	E8 D1030000	CALL 00401428	
00401057	A3 78204000	MOV DWORD PTR DS:[402078],EAX	
0040105C	68 007F0000	PUSH 7F00	
00401061	6A 00	PUSH 0	
00401063	E8 A2030000	CALL 00401408	
00401068	A3 7C204000	MOV DWORD PTR DS:[40207C],EAX	
0040106D	C705 80204000	MOV DWORD PTR DS:[402080],5	
00401077	C705 84204000	MOV DWORD PTR DS:[402084],402110	
00401081	C705 88204000	MOV DWORD PTR DS:[402088],4020F4	
0040108B	68 64204000	PUSH 402064	
00401090	E8 F3030000	CALL 00401488	
00401095	6A 00	PUSH 0	

ASCII "No need to disasm the code!"

ASCII "MENU"

ASCII "No need to disasm the code!"

我们看到 API 函数的调用处,40135C 地址处应该是调用的 MessageBoxA。

00401337	FF75 00	PUSH DWORD PTR SS:[EBP+0]	
00401339	E8 73010000	CALL 004014B2	
0040133F	E8 01000000	MOV EAX,1	
00401344	EB DF	JMP SHORT 00401325	
00401346	B8 00000000	MOV EAX,0	
0040134B	EB D8	JMP SHORT 00401325	
0040134D	6A 30	PUSH 30	
0040134F	68 29214000	PUSH 402129	
00401354	68 34214000	PUSH 402134	
00401359	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
0040135C	E8 D9000000	CALL 0040143A	
00401361	C3	RETN	
00401362	6A 00	PUSH 0	
00401364	E8 AD000000	CALL 00401416	
00401369	6A 30	PUSH 30	
0040136B	68 60214000	PUSH 402160	
00401370	68 69214000	PUSH 402169	
00401375	FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00401378	E8 BD000000	CALL 0040143A	
0040137D	C3	RETN	
0040137F	8B7424 04	MOV ESI,DWORD PTR SS:[ESP+4]	

ASCII "Good work!"

ASCII "Great work, mate! Now try the next CrackMe!"

ASCII "No luck!"

ASCII "No luck there, mate!"

我们定位到 40143A 处,这里依然是通过一个间接跳转。

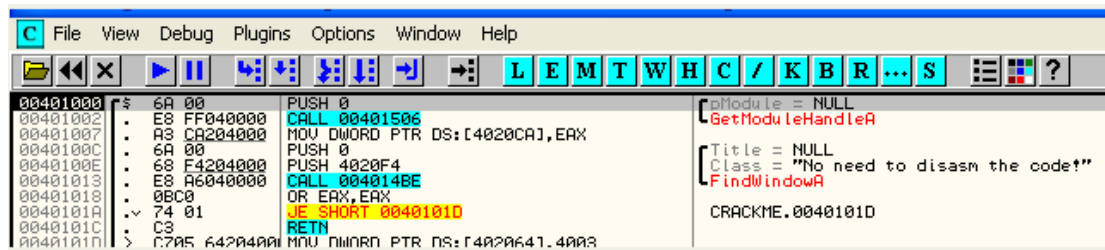
004013FB	8BDF	MOV EBX,EDI	
004013FD	C3	RETN	
004013FE	FF25 84314000	JMP DWORD PTR DS:[403184]	
00401404	FF25 88314000	JMP DWORD PTR DS:[403188]	
0040140A	FF25 8C314000	JMP DWORD PTR DS:[40318C]	
00401410	FF25 90314000	JMP DWORD PTR DS:[403190]	
00401416	FF25 94314000	JMP DWORD PTR DS:[403194]	
0040141C	FF25 98314000	JMP DWORD PTR DS:[403198]	
00401422	FF25 9C314000	JMP DWORD PTR DS:[40319C]	
00401428	FF25 A0314000	JMP DWORD PTR DS:[4031A0]	
0040142E	FF25 A4314000	JMP DWORD PTR DS:[4031A4]	
00401434	FF25 A8314000	JMP DWORD PTR DS:[4031A8]	
0040143A	FF25 AC314000	JMP DWORD PTR DS:[4031AC]	
00401440	FF25 B0314000	JMP DWORD PTR DS:[4031B0]	
00401446	FF25 B4314000	JMP DWORD PTR DS:[4031B4]	
0040144C	FF25 B8314000	JMP DWORD PTR DS:[4031B8]	
00401452	FF25 BC314000	JMP DWORD PTR DS:[4031BC]	
00401458	FF25 C0314000	JMP DWORD PTR DS:[4031C0]	
0040145E	FF25 C4314000	JMP DWORD PTR DS:[4031C4]	
00401464	FF25 C8314000	JMP DWORD PTR DS:[4031C8]	
0040146A	FF25 CC314000	JMP DWORD PTR DS:[4031CC]	
00401470	FF25 D0314000	JMP DWORD PTR DS:[4031D0]	
00401476	FF25 D4314000	JMP DWORD PTR DS:[4031D4]	
0040147C	FF25 D8314000	JMP DWORD PTR DS:[4031D8]	
00401482	FF25 DC314000	JMP DWORD PTR DS:[4031DC]	
00401488	FF25 E0314000	JMP DWORD PTR DS:[4031E0]	
0040148E	FF25 E4314000	JMP DWORD PTR DS:[4031E4]	
00401494	FF25 E8314000	JMP DWORD PTR DS:[4031E8]	
0040149A	FF25 EC314000	JMP DWORD PTR DS:[4031EC]	
004014A0	FF25 F0314000	JMP DWORD PTR DS:[4031F0]	
004014A6	FF25 F4314000	JMP DWORD PTR DS:[4031F4]	
004014AC	FF25 F8314000	JMP DWORD PTR DS:[4031F8]	
004014B2	FE25 FC314000	JMP DWORD PTR DS:[4031FC]	

DS:[004031AC]=77D504EA

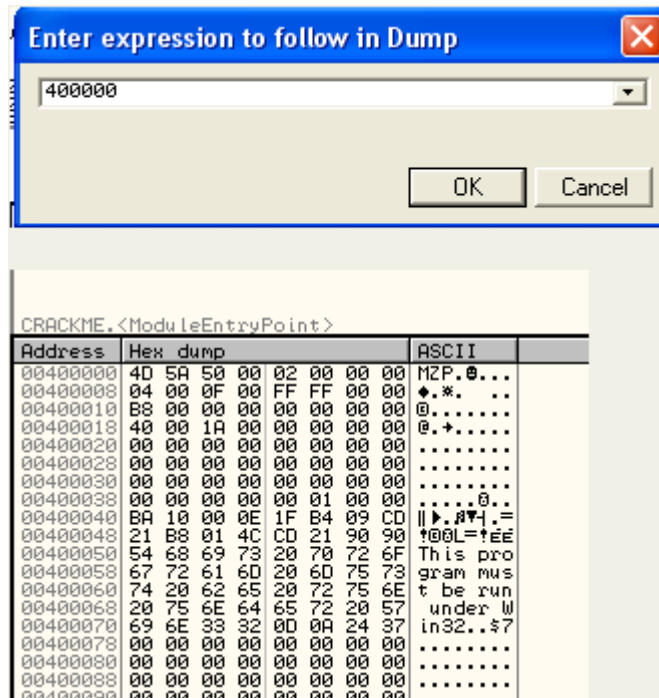
这些间接跳转是无法正常运行的,因为在正常情况,操作系统必须知道指向各个 API 函数名称字符串的指针,然后通过 GetProcAddress 定位到各个 API 函数正确的入口地址并填充到 IAT 中,这样这些间接跳转才能起作用。

下面我们来看看未加壳程序的 IAT。

我们用 OD 加载 Cruehead 的 CrackMe。



我们来定位该 CrackMe PE 结构中一些重要字段。首先在数据窗口中定位 400000 地址处。



单击鼠标右键选择-Special-PE header 切换到 PE 头的显示模式。

Address	Hex dump	Data	Comment
00400000	4D 5A	ASCII "MZ"	DOS EXE Signature
00400002	5000	DW 0050	DOS_PartPag = 50 (80.)
00400004	0200	DW 0002	DOS_PageCnt = 2
00400006	0000	DW 0000	DOS_ReloCnt = 0
00400008	0400	DW 0004	DOS_HdrSize = 4
0040000A	0F00	DW 000F	DOS_MinMem = F (15.)
0040000C	FFFF	DW FFFF	DOS_MaxMem = FFFF (65535.)
0040000E	0000	DW 0000	DOS_ReloSS = 0
00400010	B800	DW 00B8	DOS_ExeSP = B8
00400012	0000	DW 0000	DOS_ChkSum = 0
00400014	0000	DW 0000	DOS_ExeIP = 0
00400016	0000	DW 0000	DOS_ReloCS = 0
00400018	4000	DW 0040	DOS_TableOff = 40
0040001A	1A00	DW 001A	DOS_Overlay = 1A
0040001C	00	DB 00	
0040001D	00	DB 00	
0040001E	00	DB 00	
0040001F	0A	DB 0A	

往下拉。

Address	Hex dump	Data	Comment
00400035	00	DB 00	
00400036	00	DB 00	
00400037	00	DB 00	
00400038	00	DB 00	
00400039	00	DB 00	
0040003A	00	DB 00	
0040003B	00	DB 00	
0040003C	00010000	DD 00000100	Offset to PE signature
00400040	BA	DB BA	
00400041	10	DB 10	
00400042	00	DB 00	
00400043	0E	DB 0E	
00400044	1F	DB 1F	
00400045	B4	DB B4	
00400046	09	DB 09	
00400047	00	DB 00	

我们可以看到 PE 头的偏移为 100。

Address	Hex dump	Data	Comment
004000F6	00	0B 00	
004000F7	00	0B 00	
004000F8	00	0B 00	
004000F9	00	0B 00	
004000FA	00	0B 00	
004000FB	00	0B 00	
004000FC	00	0B 00	
004000FD	00	0B 00	
004000FE	00	0B 00	
004000FF	00	0B 00	
00400100	50 45 00 00	ASCII "PE"	PE signature (PE)
00400104	4C01	DW 014C	Machine = IMAGE_FILE_MACHINE_I386
00400106	0600	DW 0006	NumberOfSections = 6
00400108	2924D90A	DD 0AD92429	TimeDateStamp = AD92429
0040010C	00000000	DD 00000000	PointerToSymbolTable = 0
00400110	00000000	DD 00000000	NumberOfSymbols = 0
00400114	E000	DW 00E0	SizeOfOptionalHeader = E0 (224.)
00400116	8E81	DW 818E	Characteristics = EXECUTABLE_IMAGE 32BIT
00400118	0B01	DW 010B	MagicNumber = PE32
0040011A	02	DB 02	MajorLinkerVersion = 2
0040011B	19	DB 19	MinorLinkerVersion = 19 (25.)
0040011C	00060000	DD 00000600	SizeOfCode = 600 (1536.)
00400120	00220000	DD 00002200	SizeOfInitializedData = 2200 (8704.)
00400124	00000000	DD 00000000	SizeOfUninitializedData = 0
00400128	00100000	DD 00001000	AddressOfEntryPoint = 1000
0040012C	00100000	DD 00001000	BaseOfCode = 1000
00400130	00200000	DD 00002000	BaseOfData = 2000
00400134	00040000	DD 00040000	ImageBase = 400000

即 PE 头位于 400100 地址处。

0040017C	46000000	DD 00000046	Export Table size = 46 (70.)
00400180	00300000	DD 00003000	Import Table address = 3000
00400184	70060000	DD 00000670	Import Table size = 670 (1648.)
00400188	00600000	DD 00006000	Resource Table address = 6000
0040019C	00140000	DD 00001400	Resource Table size = 1400 (5120.)

继续往下拉,我们可以看到 IT(导入表)的指针,这里大家不要将其跟 IAT 搞混淆了。

IT = 导入表

IAT = 输入函数地址表

我们知道当程序启动之前操作系统会将各个 API 函数的地址填充到 IAT 中,那么 IT(导入表)又是怎么回事呢?首先我们定位到导入表,该导入表偏移值为 3000(即虚拟地址为 403000),长度为 670(十六进制),即 403670 为导入表的结尾。我们一起来看看。

我们将数据窗口的显示模式切换为正常状态。

Address	Hex dump	ASCII
00403000	78 30 00 00 00 00 00 00 00 00 90 32 00 00	x0.....e2..
00403010	84 31 00 00 10 31 00 00 00 00 00 00 00 00	a1..1.....
00403020	98 32 00 00 1C 32 00 00 3C 31 00 00 00 00 00	s2...L2.<1....
00403030	00 00 00 00 A8 32 00 00 48 32 00 00 4C 31 00 002..H2..L1..
00403040	00 00 00 00 00 00 00 00 53 32 00 00 53 32 00 00A2..X2..
00403050	74 31 00 00 00 00 00 00 00 00 00 00 BF 32 00 00	t1.....X2..
00403060	00 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00	c2.....X2..
00403070	00 00 00 00 00 00 00 00 CC 32 00 00 D8 32 00 00f2..i2..
00403080	EC 32 00 00 FA 32 00 00 0E 33 00 00 1C 33 00 00	y2...2..#3..L3..
00403090	30 33 00 00 30 33 00 00 46 33 00 00 54 33 00 00	3...3...F3..T3..
004030A0	60 33 00 00 6E 33 00 00 80 33 00 00 8C 33 00 00	3...3...C3..T3..
004030B0	9E 33 00 00 B6 33 00 00 C4 33 00 00 D8 33 00 00	x3..A3..-3..i3..
004030C0	E4 33 00 00 F2 33 00 00 02 34 00 00 0E 34 00 00	s3..-3..04..#4..
004030D0	1E 34 00 00 2E 34 00 00 40 34 00 00 4E 34 00 00	A4...4..04..N4..
004030E0	60 34 00 00 72 34 00 00 84 34 00 00 98 34 00 00	4...r4..#4..y4..
004030F0	0E 34 00 00 82 34 00 00 9C 34 00 00 C0 34 00 00	3...04...f4...0..
00403100	04 34 00 00 E2 34 00 00 F4 34 00 00 00 00 00 00	E4..04..04....
00403110	02 35 00 00 12 35 00 00 1E 35 00 00 2C 35 00 00	05..+5...A5...5..
00403120	3A 35 00 00 44 35 00 00 52 35 00 00 5E 35 00 00	5..D5..R5...A5..
00403130	72 35 00 00 7E 35 00 00 00 00 00 00 8C 35 00 00	r5...5.....15..
00403140	A2 35 00 00 84 35 00 00 00 00 00 00 C4 35 00 00	05...5.....-5..
00403150	00 35 00 00 DC 35 00 00 E8 35 00 00 FA 35 00 00	\$5...5...p5...5..
00403160	0C 36 00 00 16 36 00 00 20 36 00 00 30 36 00 00	6...6...6...06..
00403170	00 36 00 00 3C 36 00 00 50 36 00 00 64 36 00 00	...<6..P6..d6..
00403180	00 00 00 00 42 8C D1 77 90 8F D1 77 3E 0B D2 77B10w0A0w>0ew
00403190	24 15 D3 77 4C 1F D3 77 D4 B6 D1 77 E8 0F D2 77	\$0ewL0ewE0w0ew
004031A0	24 13 D2 77 0A 5E D2 77 60 DA D1 77 EA 04 D5 77	\$0ewr0ewr00w0ew
004031B0	11 12 D2 77 35 EE D3 77 F5 B5 D1 77 9C FA D2 77	40ew5ewS0ew6ew
004031C0	EC D8 D1 77 F6 8B D1 77 83 F7 D4 77 A4 D8 D1 77	0ew0w+00w+0ew0w
004031D0	9A F3 D2 77 2E 8C D1 77 1B C0 D1 77 F9 D7 D1 77	0ew0w+00w+0ew
004031E0	8C 14 D2 77 09 B6 D1 77 5E 02 D2 77 EE D4 D1 77	0ew0ew0ew0ew0ew
004031F0	1C B1 D3 77 08 96 D1 77 9C F3 D4 77 50 62 D2 77	0ew000ew0ew0ew
00403200	10 B6 D1 77 81 E5 D2 77 C7 86 D1 77 16 48 D5 77	*0ew08ew0ew0ew
00403210	1E AC D6 77 42 10 D2 77 00 00 00 C1 C9 80 7C	A0ewB0ew....+0C!
00403220	99 68 82 7C 2F FE 80 7C 2D FF 80 7C E0 C6 80 7C	0ke!/*0C!- 0105C!
00403230	77 9B 80 7C 9F 0F 81 7C 29 B5 80 7C 0E 18 80 7C	w0C!/*0C!0C!0C!
00403240	A2 CB 81 7C 00 00 00 00 15 C5 53 21 96 C4 53	00w0....!0+0ew-X
00403250	36 8E C6 58 00 00 00 00 0C BF F7 26 F1 F6 77	0ew0ew0ew0ew0ew
00403260	E9 49 F2 77 68 E0 EF 77 E1 61 EF 77 C9 D0 F0 77	0ew0ew0ew0ew0ew
00403270	51 E0 F0 77 20 6C EF 77 98 6E EF 77 00 00 00 00	00-w-l'w0w'w....
00403280	D8 7C 37 76 1E 31 36 76 CD 46 38 76 00 00 00 00	0ew0ew0ew0ew0ew
00403290	55 53 45 52 33 32 2E 64 6C 6C 00 48 45 52 4E 45	USER32.dll.KERNEL
004032A0	61 70 41 00 00 00 53 64 6F 72 41 00 00 4C 6F 61	L32.dll.COMCTL32
004032B0	2E 44 4C 40 00 47 44 49 33 32 2E 64 6C 6C 00 43	...MessageBoA...
004032C0	4F 40 44 4C 47 33 32 2E 64 6C 6C 00 00 00 4E 4B	...MessageBoA...
004032D0	6C 64 54 69 60 65 72 00 00 00 47 65 74 53 79 73	...GetSys...
004032E0	74 65 6D 40 65 74 72 69 63 73 00 00 00 00 4C 6F	...Load...
004032F0	61 64 43 75 72 73 6F 72 41 00 00 00 4C 6F 61	...Load...
00403300	41 63 01 65 6C 65 72 61 74 6F 72 41 00 00 00 00	...Load...
00403310	40 65 73 73 61 67 65 42 65 65 70 00 00 00 47 65	...Load...
00403320	74 57 69 6E 64 6F 77 52 65 63 74 00 00 00 4C 6F	...Load...
00403330	61 64 53 74 72 69 6E 67 41 00 00 00 4C 6F 61	...Load...
00403340	49 63 6F 6E 41 00 00 00 4C 6F 61 64 42 69 74 6D	...Load...
00403350	61 70 41 00 00 53 64 6F 72 41 00 00 4C 6F 61	...Load...
00403360	00 00 40 65 73 73 61 67 65 42 6F 78 41 00 00 00	...Load...
00403370	50 6F 73 74 51 75 69 74 40 65 73 73 61 67 65 00	...Load...
00403380	00 00 57 69 6F 48 65 6C 70 41 00 00 00 4C 6F	...Load...

这就是导入表了,我们来介绍一下导入表的结构吧。

我们选中的这 20 个字节是导入表的描述符结构。官方的叫法为 IMAGE_IMPORT_DESCRIPTOR。每组为 20 个字节,IMAGE_IMPORT_DESCRIPTOR 包含了一个的字符串指针,该指针指向了某个的动态链接库名称字符串。

我们来看个例子:

CRACKME.<ModuleEntryPoint>											
Address	Hex dump										ASCII
00403000	78	30	00	00	00	00	00	00	90	32 00 00	x0.....E2..
00403010	84	31	00	00	10	31	00	00	00	00 00 00	ä1..1.....
00403020	98	32	00	00	1C	32	00	00	3C	31 00 00	ø2..2..<1....
00403030	00	00	00	00	A8	32	00	00	48	32 00 002..H2..L1..
00403040	00	00	00	00	00	00	00	00	B5	32 00 00A2..X2..
00403050	74	31	00	00	00	00	00	00	00	BF 32 00 00	t1.....12..
00403060	80	32	00	00	00	00	00	00	00	00 00 00 00	Ç2.....
00403070	00	00	00	00	00	00	00	CC	32	00 00 00 00f2..i2..
00403080	EC	32	00	00	FA	32	00	00	0E	33 00 00 00	y2..2..#3..L3..
00403090	2C	33	00	00	3A	33	00	00	46	33 00 00 00	3..3..F3..T3..
004030A0	60	33	00	00	6E	33	00	00	80	33 00 00 00	3..n3..Ç3..i3..
004030B0	9E	33	00	00	B6	33	00	00	C4	33 00 00 00	x3..A3..-3..i3..
004030C0	E4	33	00	00	F2	33	00	00	02	34 00 00 00	š3..=3..04..#4..

这里我们将 IMAGE_IMPORT_DESCRIPTOR 简称为 IID。这里选中的部分为导入表中的第一个 IID。其中 5 个 DWORD 字段的含义如下:

OriginalFirstThunk

TimeDateStamp

时间戳

ForwarderChain

链表的前一个结构

Name1

指向 DLL 名称的指针

FirstThunk

指向的链表定义了针对 Name1 这个动态链接库引入的所有导入函数

前三个字段不是很重要,对于我们 Cracker 来说,我们只对第 4,5 字段感兴趣。

Address	Hex dump										ASCII	
00403000	78	30	00	00	00	00	00	00	90	32	00 00 00	x0.....
00403010	84	31	00	00	10	31	00	00	00	00	00 00 00	ä1..1.....
00403020	98	32	00	00	1C	32	00	00	3C	31	00 00 00	ø2..2..<1....
00403030	00	00	00	00	A8	32	00	00	48	32	00 00 002..H2..L1..
00403040	00	00	00	00	00	00	00	00	B5	32	00 00 00A2..X2..
00403050	74	31	00	00	00	00	00	00	00	00	00 00 00	t1.....

正如大家所看到的,第 4 个字段为指向 DLL 名称字符串的指针,我们来看看 403290 处是哪个 DLL 的名称。

00403250	3E	88	C6	58	00	00	00	00	0C	BC	EF	77	26	F1	F0	77	!t%X....'w&:-
00403260	E9	49	F2	77	68	0E	EF	77	E1	61	EF	77	C9	DD	F0	77	Üi=wh0'wpa'wfi-w
00403270	51	E0	F0	77	2D	6C	EF	77	98	6E	EF	77	00	00	00	00	Q0-w-l'wyn'w...
00403280	08	7C	37	76	1E	31	36	76	CD	46	38	76	00	00	00	00	i17vΔ16v=F8v....
00403290	55	53	45	52	33	32	2E	64	6C	6C	00	48	45	52	4E	45	USER32.dll.KERNE
004032A0	4C	33	32	2E	64	6C	6C	00	43	4F	4D	43	54	4C	33	32	L32.dll.COMCTL32
004032B0	2E	44	4C	4C	00	47	44	49	33	32	2E	64	6C	6C	00	43	.DLL,GDI32.dll,C
004032C0	4F	4D	44	4C	47	33	32	2E	64	6C	6C	00	00	00	48	69	OMDLG32.dll...Ki
004032D0	6C	6C	54	69	6D	65	72	00	00	00	47	65	74	53	79	73	llTimer...GetSys
004032E0	74	65	6D	4D	65	74	72	69	63	73	00	00	00	00	4C	6F	temMetrics...Lo
004032F0	61	64	43	75	72	73	6F	72	41	00	00	00	4C	6F	61	64	adCursorA...Load
00403300	41	63	63	65	6C	65	72	61	74	6F	72	73	41	00	00	00	AcceleratorsA...

这里我们可以看到是 USER32.DLL,第 5 个字段指向了 USER32.DLL 对应 IAT 项的起始地址,即 403184。

CRACKME.<ModuleEntryPoint>																	ASCII
Address	Hex dump																
00403184	42	8C	D1	77	9D	8F	D1	77	3E	0B	D2	77	24	15	D3	77	B10w0A0w>0ew\$8ew
00403194	4C	1F	D3	77	04	B6	D1	77	E8	0F	D2	77	24	13	D2	77	L7ewEÄ0w0*ew\$!8ew
004031A4	0A	5E	D2	77	60	DA	D1	77	EA	04	D5	77	11	12	D2	77	r^ew' r0w0* w40ew
004031B4	35	EE	D3	77	F5	B5	D1	77	9C	FA	D2	77	EC	0B	D1	77	5^ew\$Ä0w0* ew000ew
004031C4	F6	88	D1	77	83	F7	D4	77	A4	08	D1	77	9A	F3	D2	77	+i0wä-ew0i0w0\$ew
004031D4	2E	8C	D1	77	1B	C0	D1	77	F9	07	D1	77	8C	14	D2	77	.i0w+^0w--i0wi0ew
004031E4	09	B6	D1	77	5E	02	D2	77	EE	04	D1	77	1C	B1	D3	77	.Ä0w^0ew-ë0wL88ew
004031F4	B8	96	D1	77	9C	F3	D4	77	50	62	D2	77	1D	B6	D1	77	0ü0w0\$ewPbëw#Ä0w
00403204	81	E5	D2	77	C7	86	D1	77	16	48	D2	77	1E	AC	D6	77	ü0ewÄ\$0w..HEwÄ\$ew
00403214	42	10	D2	77	00	00	00	00	C1	C9	80	7C	99	68	82	7C	B0ew...+fC!0ke!
00403224	2F	FE	80	7C	2D	FF	80	7C	E0	C6	80	7C	77	9B	80	7C	/wC!- C!0\$C!w0C!
00403234	9F	0F	81	7C	29	B5	80	7C	0E	18	80	7C	A2	CA	81	7C	f*ü!))ÄC!Ä+C!0^ü!
00403244	00	00	00	00	15	C5	58	7C	21	9B	C4	58	38	8B	C6	58!S+X!0-X;!t%X
00403254	00	00	00	00	0C	BC	EF	77	26	F1	F0	77	E9	49	F2	77'w&:-w0I=w
00403264	68	E0	EF	77	E1	61	EF	77	C9	DD	F0	77	51	E0	F0	77	h0'wpa'wfi-w00-w

这里就是 IAT 了,导入表的结束地址为 403670。导入表中的每个 IID 项指明了 DLL 的名称以及其对应 IAT 项的起始地址。紧凑的排列在一起,供操作系统使用。

大量实验表明,IAT 并不一定位于在导入表中。IAT 可以位于程序中任何具有写权限的地方,只要当可执行程序运行起来时,操作系

统可以定位到这些 IID 项,然后根据 IAT 中标明的 API 函数名称获取到函数地址即可。下面我们来总结一下操作系统填充 IAT 的具体步骤:

- 1:定位导入表
- 2:解析第一个 IID 项,根据 IID 中的第 4 个字段定位 DLL 的名称
- 3:根据 IID 项的第 5 个字段 DLL 对应的 IAT 项的起始地址
- 4:根据 IAT 中的指针定位到相应 API 函数名称字符串
- 5:通过 GetProcAddress 获取 API 函数的地址并填充到 IAT 中
- 6:当定位到的 IAT 项为零的时候表示该 DLL 的 API 函数地址获取完毕了,接着继续解析第二个 IID,重复上面的步骤。

下面我们来手工的体验一下这个步骤:

- 1)定位导入表

0040017C	46000000	DD 00000046	Export Table size = 46 (70.)
00400180	00300000	DD 00003000	Import Table address = 3000
00400184	70060000	DD 00000670	Import Table size = 670 (1648.)
00400188	00600000	DD 00006000	Resource Table address = 6000

- 2)定位到导入表的起始地址

CRACKME.<ModuleEntryPoint>			
Address	Hex dump	ASCII	
00403000	78 30 00 00 00 00 00 00 00 00 00 00 90 32 00 00	x0.....ē2..	
00403010	84 31 00 00 10 31 00 00 00 00 00 00 00 00 00 00	ā1..!.....	
00403020	9B 32 00 00 1C 32 00 00 3C 31 00 00 00 00 00 00	ø2..L2..<1.....	
00403030	00 00 00 00 A8 32 00 00 48 32 00 00 4C 31 00 00ē2..H2..L1..	
00403040	00 00 00 00 00 00 00 00 B5 32 00 00 58 32 00 00A2..X2..	
00403050	74 31 00 00 00 00 00 00 00 00 00 00 BF 32 00 00	t1.....r2..	
00403060	80 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00	ç2.....	
00403070	00 00 00 00 00 00 00 00 CC 32 00 00 08 32 00 00f2..i2..	
00403080	EC 32 00 00 FA 32 00 00 0E 33 00 00 1C 33 00 00	y2..2..#3..L3..	
00403090	2C 33 00 00 3A 33 00 00 46 33 00 00 54 33 00 00	,3...3..F3..T3..	
004030A0	60 33 00 00 6E 33 00 00 80 33 00 00 8C 33 00 00	*3..n3..ç3..l3..	
004030B0	0F 33 00 00 0F 33 00 00 04 33 00 00 00 33 00 00	v3..r3..	

- 3)根据第一个 IID 项中的第四个字段得到 DLL 名称字符串的指针,这里指向的是 USER32.DLL

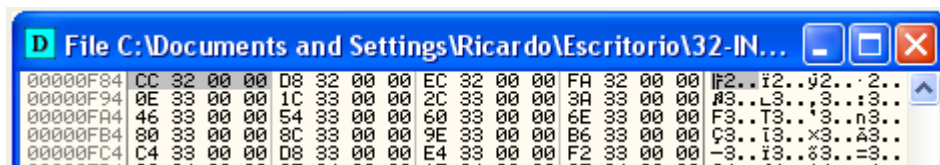
CRACKME.<ModuleEntryPoint>			
Address	Hex dump	ASCII	
00403290	55 53 45 52 33 32 2E 64 6C 6C 00 4B 45 52 4E 45	USER32.dll.KERNE	
004032A0	4C 33 32 2E 64 6C 6C 00 43 4F 4D 43 54 4C 33 32	L32.dll.COMCTL32	
004032B0	2E 44 4C 4C 00 47 44 49 33 32 2E 64 6C 6C 00 43	.DLL.GDI32.dll.C	
004032C0	4F 4D 44 4C 47 47 33 32 2E 64 6C 6C 00 00 4B 69	OMDLG32.dll...Ki	
004032D0	6C 6C 54 69 6D 65 72 00 00 00 47 65 74 53 79 73	llTimer...GetSys	
004032E0	74 65 6D 4D 65 74 72 69 63 73 00 00 00 00 4C 6F	temMetrics....Lo	
004032F0	61 64 43 75 72 73 6F 72 41 00 00 00 4C 6F 61 64	adCursorA...Load	
00403300	41 63 63 65 6C 65 72 61 74 6F 72 73 41 00 00 00	AcceleratorsA...	
00403310	4D 65 73 73 61 67 65 42 65 65 70 00 00 00 47 65	MessageBeep...Ge	
00403320	74 57 69 6E 64 6F 77 52 65 63 74 00 00 00 4C 6F	twindowRect...Lo	
00403330	61 64 53 74 72 69 6E 67 41 00 00 00 4C 6F 61 64	adStringA...Load	
00403340	49 63 6F 6E 41 00 00 00 4C 6F 61 64 42 69 74 6D	IconA...LoadBitm	
00403350	61 70 41 00 00 00 53 65 74 46 6F 63 75 73 00 00	apA...SetFocus...	
00403360	00 00 4D 65 73 73 61 67 65 42 6F 78 41 00 00 00	..MessageBoxA...	
00403370	50 6F 73 74 51 75 69 74 4D 65 73 73 61 67 65 00	PostQuitMessage.	

根据第五个字段的内容定位到 IAT 项的起始地址,这里是 403184,我们定位到该地址处。

CRACKME.<ModuleEntryPoint>			
Address	Hex dump	ASCII	
00403000	78 30 00 00 00 00 00 00 00 00 00 00 90 32 00 00	x0.....ē2..	
00403010	84 31 00 00 10 31 00 00 00 00 00 00 00 00 00 00	ā1..!.....	
00403020	9B 32 00 00 1C 32 00 00 3C 31 00 00 00 00 00 00	ø2..L2..<1.....	

Address	Hex dump	ASCII	
00403184	42 8C D1 77 9D 8F D1 77 3E 0B D2 77 24 15 D3 77	B!0w0A0w>0ēw\$Sēw	
00403194	4C 1F D3 77 04 B6 D1 77 E8 0F D2 77 24 13 D2 77	L!ēwēA0w0*ēw\$!!ēw	
004031A4	DA 5E D2 77 60 DA D1 77 EA 04 D5 77 11 12 D2 77	r!ēw' r0w0♦'w!0ēw	
004031B4	35 EE D3 77 F5 B5 D1 77 9C FA D2 77 EC D8 D1 77	S!ēwS0w0ē!ēw00w	
004031C4	F6 8B D1 77 83 F7 D4 77 A4 D8 D1 77 9A F3 D2 77	÷!0wā!ēw0i0w0!ēw	
004031D4	2E 8C D1 77 1B C0 D1 77 F9 D7 D1 77 8C 14 D2 77	.!0w+!0w!i0wi!ēw	
004031E4	09 B6 D1 77 5E 02 D2 77 EE D4 D1 77 1C B1 D3 77	.A0w+0ēw!ēw0L0ēw	
004031F4	B8 96 D1 77 9C F3 D4 77 50 62 D2 77 1D B6 D1 77	@00w0ēw0Pbēw#A0w	
00403204	81 E5 D2 77 C7 86 D1 77 16 48 D2 77 1E AC D6 77	u0ēwA0w0..HEw0!ēw	
00403214	42 10 D2 77 00 00 00 00 C1 C9 80 7C 99 6B 82 7C	B!ēw....!r0!0kē!	

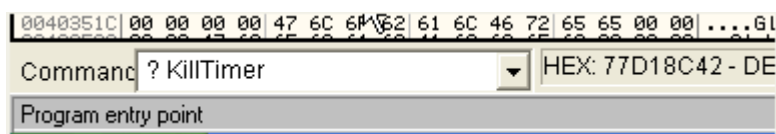
这里我们可以看到已经被填充了正确的 API 函数的入口地址,跟我们 dump 出来的结果一样,我们再来看看相应的可执行文件偏移处的内容是什么。



这里我们可以看到第一个 API 函数的名称位于 4032CC 地址处,我们定位到该地址处。

CRACKME.<ModuleEntryPoint>		
Address	Hex dump	ASCII
004032CC	00 00 4B 69 6C 6C 54 69 6D 65 72 00 00 00 47 65	..KillTimer...Ge
004032DC	74 53 79 73 74 65 6D 4D 65 74 72 69 63 73 00 00	tSystemMetrics...
004032EC	00 00 4C 6F 61 64 43 75 72 73 6F 72 41 00 00 00	..LoadCursorA...
004032FC	4C 6F 61 64 41 63 63 65 6C 65 72 61 74 6F 72 73	LoadAccelerators
0040330C	41 00 00 00 4D 65 73 73 61 67 65 42 65 65 70 00	A...MessageBeep.
0040331C	00 00 47 65 74 57 69 6E 64 6F 77 52 65 63 74 00	..GetWindowRect.
0040332C	00 00 4C 6F 61 64 53 74 72 69 6E 67 41 00 00 00	..LoadStringA...
0040333C	4C 6F 61 64 49 63 6F 6E 41 00 00 00 4C 6F 61 64	LoadIconA...Load
0040334C	42 69 74 6D 61 70 41 00 00 00 53 65 74 46 6F 63	BitmapA...SetFoc
0040335C	75 73 00 00 00 00 4D 65 73 73 61 67 65 42 6F 78	us...MessageBox

第一个 API 函数是 KillTimer,我们在 OD 中看到的 KillTimer 的入口地址是操作系统调用 GetProcAddress 获取到的。



这里我们可以看到 KillTimer 的入口地址为 77D18C42。该地址将被填充到 IAT 相应单元中去覆盖原来的值。

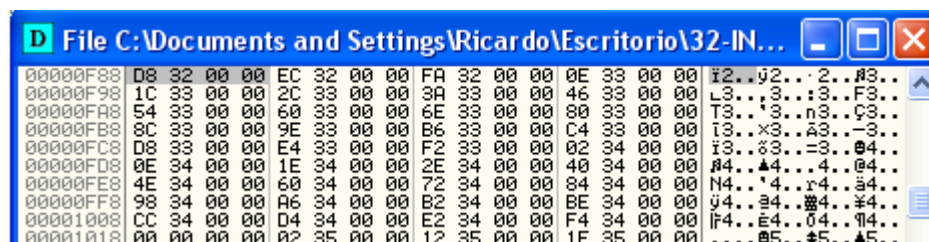
CRACKME.<ModuleEntryPoint>		
Address	Hex dump	ASCII
00403184	42 8C D1 77 9D 8F D1 77 3E 0B D2 77 24 15 D3 77	B!0w@A0w>0ew\$Sew
00403194	4C 1F D3 77 04 B6 D1 77 E8 0F D2 77 24 13 D2 77	L!ew@A0w0ew\$!!ew
004031A4	DA 5E D2 77 60 DA D1 77 EA 04 D5 77 11 12 D2 77	r^ew' r0w0'w!0ew
004031B4	35 EE D3 77 F5 B5 D1 77 9C FA D2 77 EC D8 D1 77	S^ew\$A0w0ew\$0ew
004031C4	F6 8B D1 77 83 F7 D4 77 A4 D8 D1 77 9A F3 D2 77	+i0w@ew0i0w0ew
004031D4	2E 8C D1 77 1B C0 D1 77 F9 D7 D1 77 8C 14 D2 77	.i0w^0w~i0wi0ew
004031E4	09 B6 D1 77 5E 02 D2 77 EE D4 D1 77 1C B1 D3 77	.A0w^0ew^0wL0ew

这里是 IAT 中的第一元素。

我们再来看下一个元素,向后偏移 4 就是,来看一看该 API 函数名称字符串的指针是多少。

CRACKME.<ModuleEntryPoint>		
Address	Hex dump	ASCII
00403184	42 8C D1 77 9D 8F D1 77 3E 0B D2 77 24 15 D3 77	B!0w@A0w>0ew\$Sew
00403194	4C 1F D3 77 04 B6 D1 77 E8 0F D2 77 24 13 D2 77	L!ew@A0w0ew\$!!ew
004031A4	DA 5E D2 77 60 DA D1 77 EA 04 D5 77 11 12 D2 77	r^ew' r0w0'w!0ew
004031B4	35 EE D3 77 F5 B5 D1 77 9C FA D2 77 EC D8 D1 77	S^ew\$A0w0ew\$0ew
004031C4	F6 8B D1 77 83 F7 D4 77 A4 D8 D1 77 9A F3 D2 77	+i0w@ew0i0w0ew
004031D4	2E 8C D1 77 1B C0 D1 77 F9 D7 D1 77 8C 14 D2 77	.i0w^0w~i0wi0ew
004031E4	09 B6 D1 77 5E 02 D2 77 EE D4 D1 77 1C B1 D3 77	.A0w^0ew^0wL0ew

定位到可执行文件的相应偏移处:

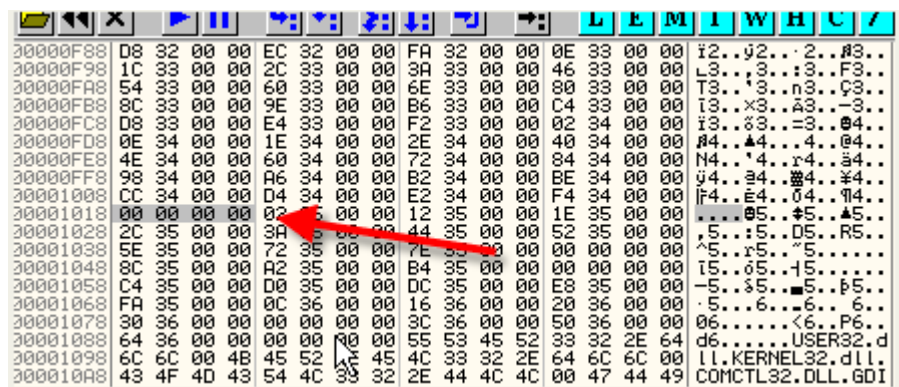


32D8 即 4032D8,来看看该 API 函数的名称是什么,这里由于该指针不为零,说明该 API 函数还是位于 USER32.DLL 中的。

CRACKME.<ModuleEntryPoint>		
Address	Hex dump	ASCII
004032D8	00 00 47 65 74 53 79 73 74 65 6D 4D 65 74 72 69	..GetSystemMetri
004032E8	63 73 00 00 00 00 4C 6F 61 64 43 75 72 73 6F 72	cs....LoadCursor
004032F8	41 00 00 00 4C 6F 61 64 41 63 63 65 6C 65 72 61	A...LoadAccelera
00403308	74 6F 72 73 41 00 00 00 4D 65 73 73 61 67 65 42	torsA...MessageB
00403318	65 65 70 00 00 00 47 65 74 57 69 6E 64 6F 77 52	eeep...GetWindowR
00403328	65 63 74 00 00 00 4C 6F 61 64 53 74 72 69 6E 67	ect...LoadString
00403338	41 00 00 00 4C 6F 61 64 49 63 6F 6E 41 00 00 00	A...LoadIconA...

这里我们可以看到第二个 API 函数是 GetSystemMetrics,通过该函数名称可以通过 GetProcAddress 获取到其函数地址然后填充到 IAT 中。接下来按照以上步骤依次获取 USER32.DLL 中的其他的函数地址,直到遇到的 IAT 项为零为止。我们来看一看可执行文

件中结束项位于哪里。



Address	Hex dump	ASCII
00000F88	D8 32 00 00 EC 32 00 00 FA 32 00 00 0E 33 00 00 i2..92..2..#3..	
00000F98	1C 33 00 00 2C 33 00 00 3A 33 00 00 46 33 00 00 L3...3...3...F3..	
00000FA8	54 33 00 00 60 33 00 00 6E 33 00 00 80 33 00 00 T3...3...n3..Q3..	
00000FB8	8C 33 00 00 9E 33 00 00 B6 33 00 00 C4 33 00 00 i3...x3..A3...-3..	
00000FC8	D8 33 00 00 E4 33 00 00 F2 33 00 00 02 34 00 00 i3...3...-3...04..	
00000FD8	0E 34 00 00 1E 34 00 00 2E 34 00 00 40 34 00 00 #4...4...4...04..	
00000FE8	4E 34 00 00 60 34 00 00 72 34 00 00 84 34 00 00 N4...4...r4...34..	
00000FF8	98 34 00 00 A6 34 00 00 B2 34 00 00 BE 34 00 00 y4...34...#4...¥4..	
00001008	CC 34 00 00 D4 34 00 00 E2 34 00 00 F4 34 00 00 f4...E4...04...¶4..	
00001018	00 00 00 00 00 00 00 00 12 35 00 00 1E 35 00 00 ...05...#5...▲5..	
00001028	2C 35 00 00 3A 35 00 00 44 35 00 00 52 35 00 00 ,5...5...05...R5..	
00001038	5E 35 00 00 72 35 00 00 7E 35 00 00 00 00 00 00 ^5...r5...~5.....	
00001048	8C 35 00 00 A2 35 00 00 B4 35 00 00 00 00 00 00 15...05...+5.....	
00001058	C4 35 00 00 D0 35 00 00 DC 35 00 00 E8 35 00 00 -5...5...5...p5..	
00001068	FA 35 00 00 0C 36 00 00 16 36 00 00 20 36 00 00 ,5...6...6...6...	
00001078	30 36 00 00 00 00 00 00 3C 36 00 00 50 36 00 00 06.....<6...P6..	
00001088	64 36 00 00 00 00 00 00 55 53 45 52 33 32 2E 64 d6.....USER32.d	
00001098	6C 6C 00 48 45 52 45 4C 33 32 2E 64 6C 6C 00 ll.KERNEL32.dll.	
000010A8	43 4F 4D 43 54 4C 33 32 2E 44 4C 4C 00 47 44 49 COMCTL32.DLL.GDI	

我们可以看到当 IAT 中元素为零的时候表明 USER32.DLL 就搜索完毕了,我们接着来看下一个 IID。

Address	Hex dump	ASCII
00403000	78 30 00 00 00 00 00 00 00 00 00 90 32 00 00 x0.....ē2..	
00403010	84 31 00 00 10 31 00 00 00 00 00 00 00 00 00 00 ā1...1.....	
00403020	9B 32 00 00 1C 32 00 00 3C 31 00 00 00 00 00 00 ſ2...L2...<1.....	
00403030	00 00 00 00 A8 37 00 00 48 32 00 00 4C 31 00 0022...H2...L1..	
00403040	00 00 00 00 00 00 00 00 B5 32 00 00 58 32 00 00A2...X2.....	
00403050	74 37 00 00 00 00 00 00 00 00 00 00 BF 32 00 00 t1.....72.....	
00403060	80 32 00 00 00 00 00 00 00 00 00 00 00 00 00 00 Ć2.....	
00403070	00 00 00 00 00 00 00 00 C3 32 00 00 00 32 00 00f2...i2..	

这里我们根据第 4,5 字段分别可以知道第二个 DLL 的名称,以及对应 IAT 项的起始地址。

DLL 的名称字符串位于 40329B 地址处。

Address	Hex dump	ASCII
0040329B	48 45 52 4E 45 4C 33 32 2E 64 6C 6C 00 43 4F 4D KERNEL32.dll.COM	
004032AB	43 54 4C 33 32 2E 44 4C 4C 00 47 44 49 33 32 2E CTL32.DLL.GDI32.	
004032BB	64 6C 6C 00 43 4F 4D 44 4C 47 33 32 2E 64 6C 6C dll.COMDLG32.dll	
004032CB	00 00 00 4B 69 6C 6C 54 69 6D 65 72 00 00 00 47 ...KillTimer...G	
004032DB	65 74 53 79 73 74 65 6D 4D 65 74 72 69 63 73 00 etSystemMetrics.	

我们可以看到第二个 DLL 为 KERNEL32.DLL,该 DLL 对应的 IAT 项起始地址为 40321C。

Address	Hex dump	ASCII
0040320C	16 48 D2 77 1E AC D6 77 42 10 D2 77 00 00 00 00 ..HEW▲%iwB!ew...	
0040321C	C1 C9 80 7C 99 68 82 7C 2F FE 80 7C 2D FF 80 7C +fC!0ke!/\C!-C!	
0040322C	E0 C6 80 7C 77 9B 80 7C 9F 0F 81 7C 29 B5 80 7C 03C!wC!f#u!)AC!	
0040323C	0E 18 80 7C A2 CA 81 7C 00 00 00 00 DD 15 C5 58 A+C!0u!....!S+X	
0040324C	21 9B C4 58 3B 8B C6 58 00 00 00 00 0C BC EF 77 t0-X;isX.....a'w	
0040325C	26 F1 F0 77 E9 49 F2 77 68 E0 EF 77 E1 61 EF 77 &t-wU!-wh0'wB'a'w	
0040326C	C9 DD F0 77 51 E0 F0 77 2D 6C EF 77 98 6E EF 77 f!-w00-w-l'w9n'w	
0040327C	00 00 00 00 08 7C 37 76 1E 31 36 76 CD 46 38 76i!7v▲16v=F8v	
0040328C	00 00 00 00 55 53 45 52 33 32 2E 64 6C 6C 00 4BUSER32.dll.K	

这里我们可以看到前一个 DWORD 是零,表示 USER32.DLL 的 API 函数的结尾。40321C 表示 KERNEL32.DLL 的 API 函数地址项的开始。

根据这些指针我们就可以定位到 kernel32.dll 中的各个 API 函数名称字符串,进而获取到其函数入口地址,接着填充到对应的 IAT 项中覆盖原来的内容。

本章这里就结束了,我给大家描述了 IAT 被填充的整个过程,了解这个过程对大家来说是很必要的,这部分内容是重建 IAT 必备的基础知识,大家只有理解了其基本原理,然后配上适当的工具,就可以方便进行 IAT 的修复工作了。

好了,下一章我们将介绍具体如何修复 IAT。