

第三十一章-脱壳简介

原定本章是要介绍 P-CODE 的 Part3,然后再介绍脱壳的。但是很多朋友跟我说 P-CODE,WKT 的教程有很多,并且现在 P-CODE 的应用程序已经很少了,没有必要过多的介绍 P-CODE,希望我能够多讲讲壳,提高大家脱壳的能力。所以说本章我们开始讨论壳,首先要给大家明确一点,壳的种类繁多(PS:这里说种类繁多,并不准确,种类也就那么几种,压缩壳,加密壳,虚拟机壳,所以说应该是数量多,之前发过一篇帖子收集了 T4 社区的各种脱壳脚本,充斥着各种各样的壳,大家可以看看),所以接下来的章节,我不会介绍所有的壳,只会给大家介绍壳的基本概念以及原理,还有几个具体壳的实例,大家不要指望看完本教程将能搞定所有壳,本教程的目的在于帮助大家理解壳的原理,锻炼大家解决未知壳的能力,大家要学会举一反三,触类旁通。

本章我们将介绍壳的基本概念,这对我们后面脱壳是大有裨益的,大家不要因为简单,就忽视它,后面章节,我们会介绍一些手工脱壳的实例。

首先大家可能会问为什么要给程序加壳?(PS:其实地球人都知道,嘿嘿)

我们知道一个未加壳或者脱过壳的程序修改起来很方便,但是如果一个加过壳或者自修改的程序,要想修改它就比较困难了,我们在入口点(PS:这里指的入口点是壳的入口点)处修改程序是不起作用的,只有当壳把原程序区段解密完成后修改才能起作用。

加过壳的程序,原程序代码段通常是被加密过的,我们想修改它就不那么容易了。

加壳程序给目标程序加壳的原理通常是加密/压缩原程序各个区段,并且给目标程序添加一个或者多个区段作为原程序的引导代码(壳代码),然后将原程序入口点修改为外壳程序的入口点。

如果我们将加过壳的程序用 OllyDbg 加载的话,OllyDbg 会停在壳的解密例程的入口点处,由此开始执行。

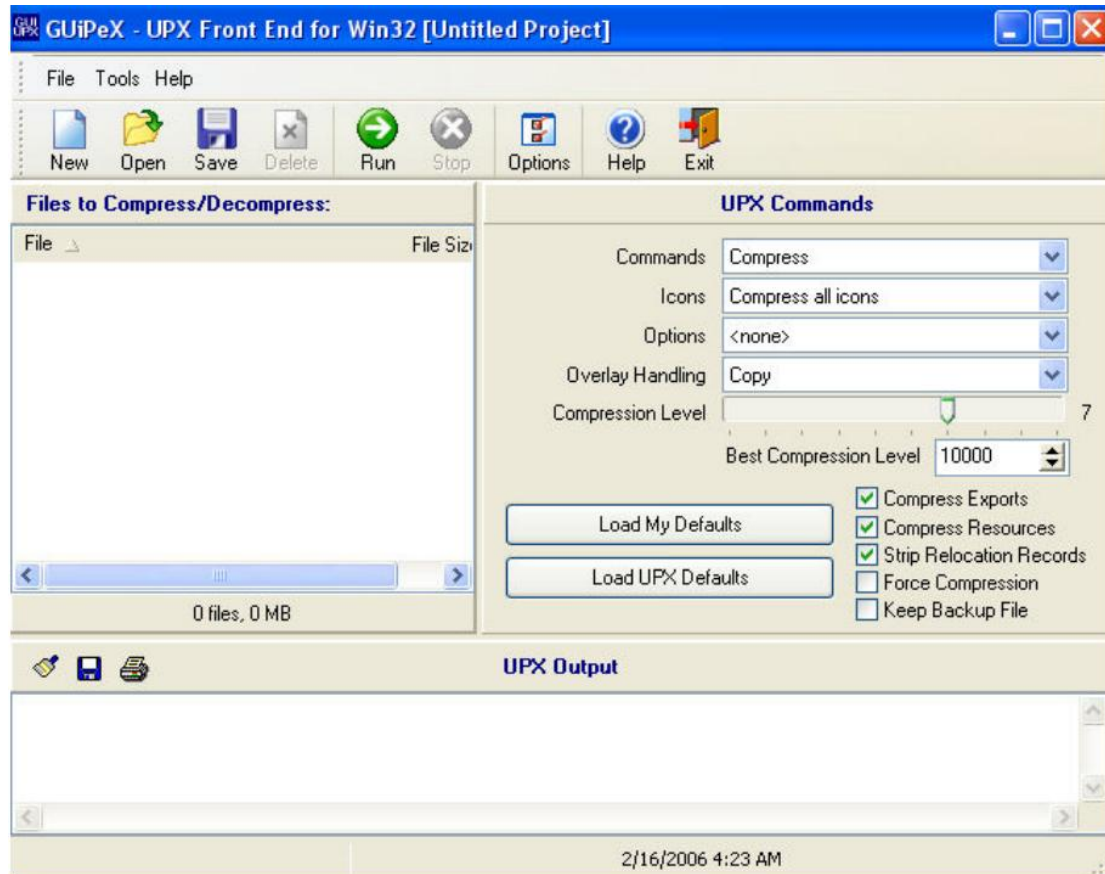
壳的解密例程(外壳程序)首先会定位加密/压缩过的原程序的各个区段,将其解密/解压,然后跳转至 OEP(程序未加壳时的入口点)处开始执行。

现在我们来看一个最简单的壳,UPX 壳,大家可以去下面网址中下载一个 GUI 版,名称为 GUIPeX_Setup。

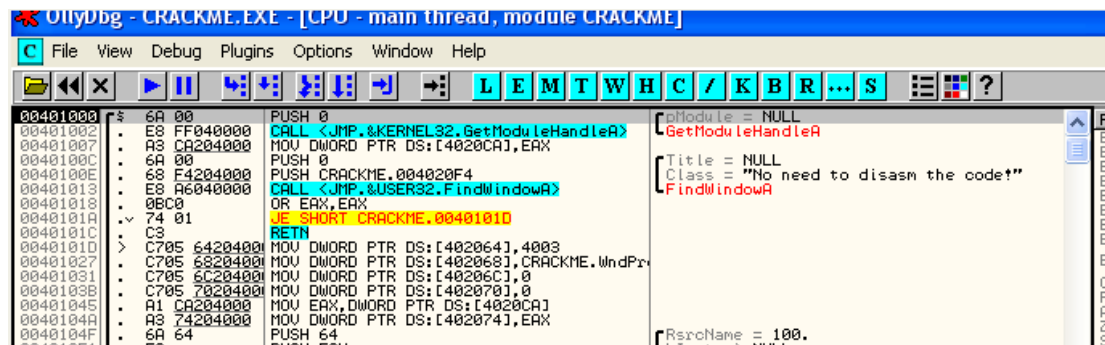
<http://www.blueorbsoft.com/guipex/>



安装好以后运行起来。



我们加壳的对象就选择大家熟悉的 CrueHead 的 CrackMe,首先我们不加壳将其加载到 OllyDbg 中。



我们可以看到其入口点是 401000,也就是说,运行它,将从 401000 地址处开始执行。

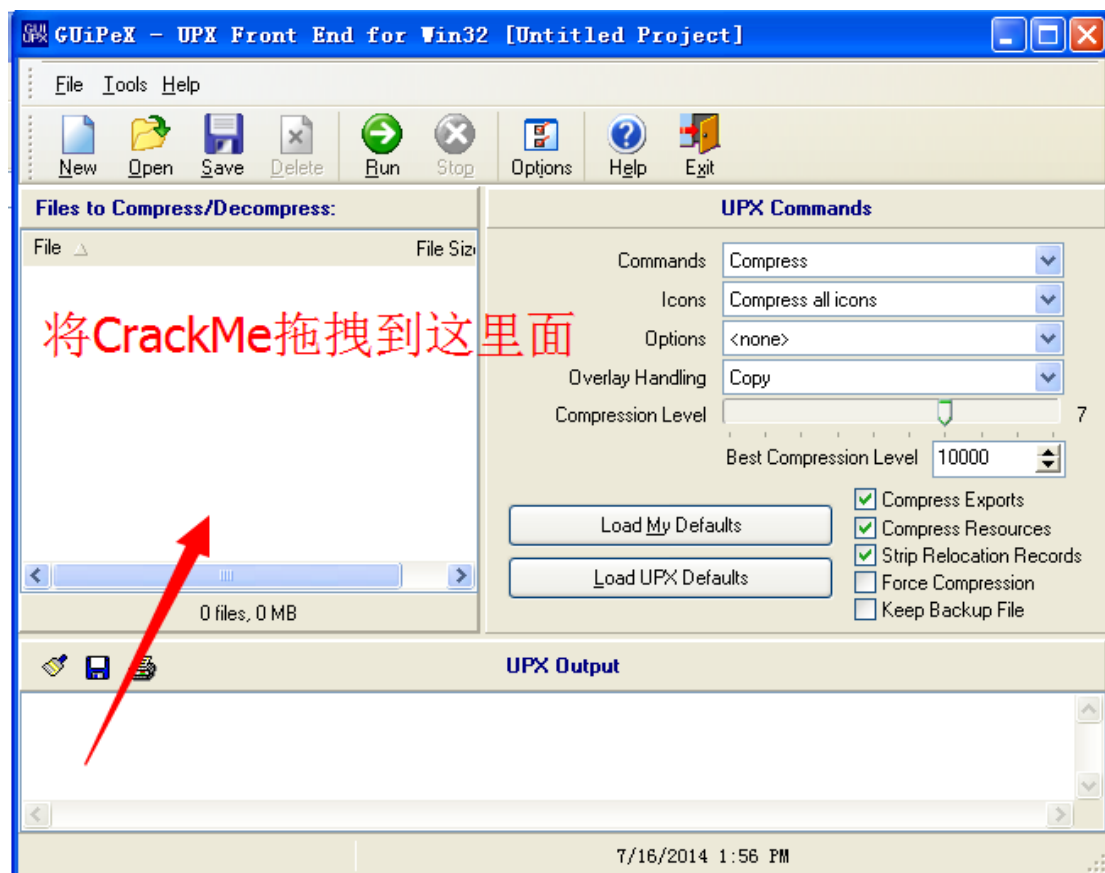
现在我们用 GUIPeX 将其加壳,解密其区段,并且将入口点修改为解密例程(外壳程序)的起始地址。

当我们运行加壳后的 CrackMe,首先会从解密例程开始执行,解密例程会解密原程序各个区段,然后定位到位于原程序代码段中的入口点,也就是大家常说的 OEP(Original Entry Point),即 401000,接着跳往 OEP 处开始执行原程序代码。

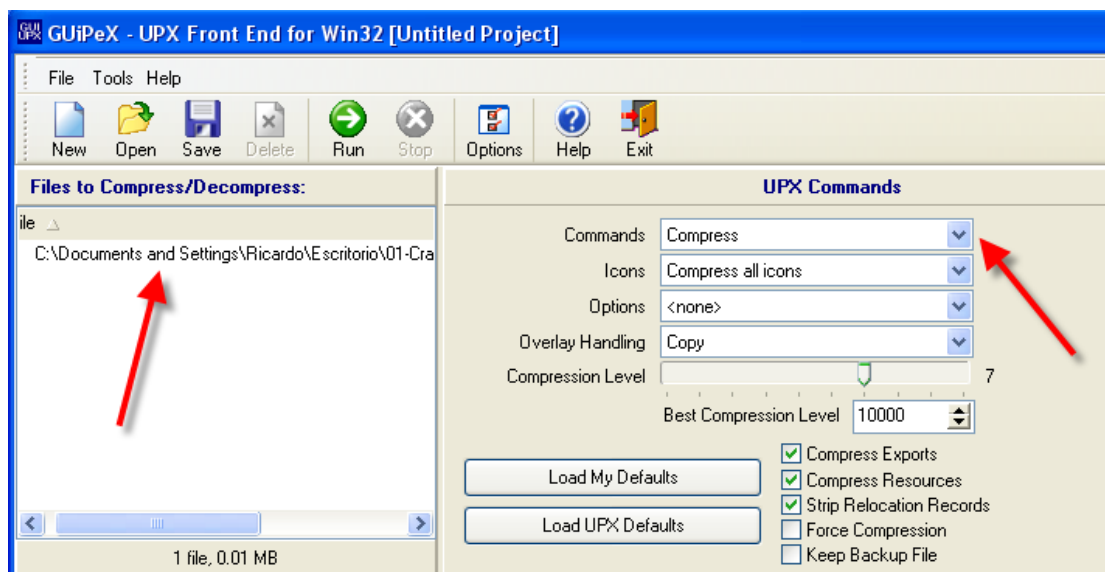
下面我们就来讨论 CrueHead 这个被加过壳的 CrackMe,该 CrackMe 的 OEP 位于何处我们已经清楚了。

我们将该 CrackMe 保存一份备份并将其放到一个安全的地方,比如说桌面,因为接下来我们需要将加过壳的 CrackMe 与原始的 CrackMe 进行对比。

打开 GUIPeX。



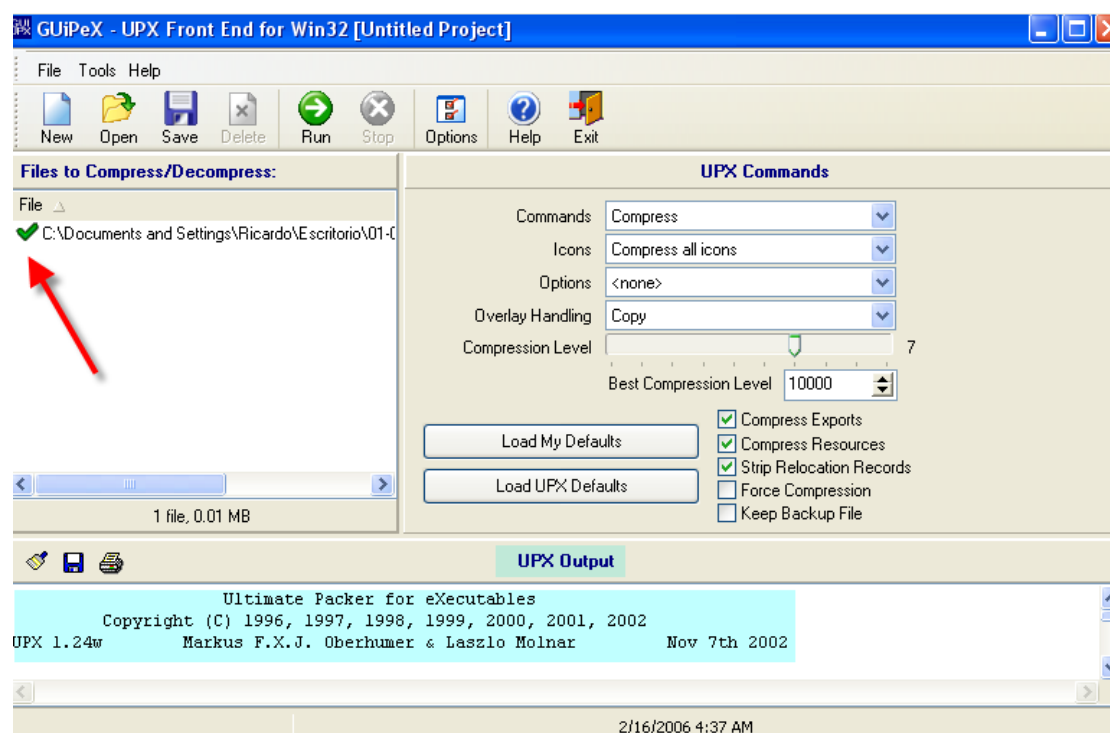
我们将 CrackMe 拖拽到上面那个窗口中。



我们可以看到上面窗口中显示出了带加壳的 CrackMe 的完整路径。后边的 Commands(命令)选项我们选择 Compress(压缩)。

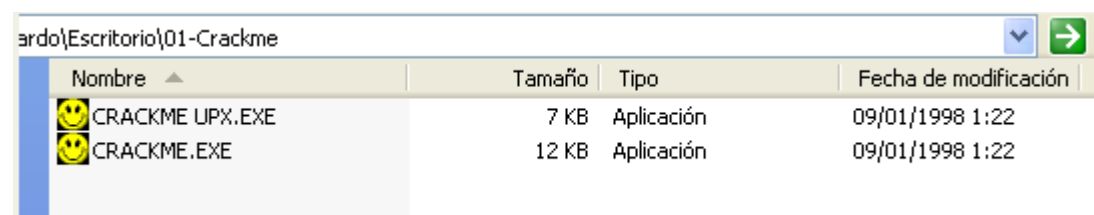
通过这个工具我们可以给目标程序压缩或者解压缩。

然后我们按 Run 按钮就开始给该 CrackMe 加壳了。

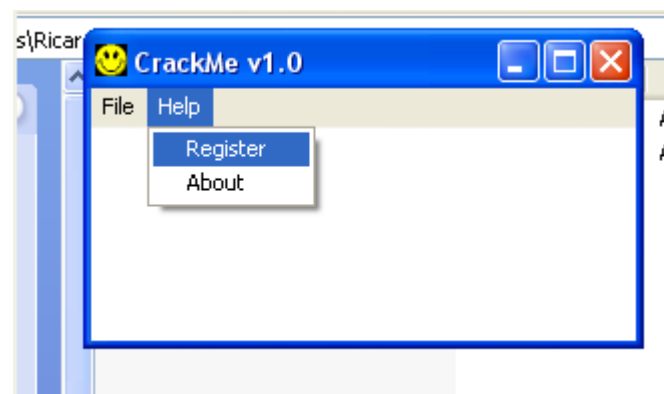


我们可以看到 UPX Output 窗口中显示该 CrackMe 加壳成功了。

我们将加壳后的 CrackMe 重命名为 CRACKME UPX.EXE。



我们会发现加壳后要比原来的小很多,加壳程序给原程序添加了额外的代码来保护原程序,体积反而变小了,嘿嘿。



我们将加壳后的 CrackMe 运行起来,可以看到跟原 CrackMe 运行效果一样。

现在我们用两个 OllyDbg 分别加载 CRACKME UPX.EXE 和 CRACKME.EXE,比较一下两者有什么不同。

CRACKME.EXE 的入口点

```

00401000  6A 00      PUSH 0
00401002  E8 FF040000 CALL <JMP.&KERNEL32.GetModuleHandleA>
00401007  A3 C8204000 MOV DWORD PTR DS:[4020CA],EAX
0040100C  6A 00      PUSH 0
0040100E  68 F4204000 PUSH CRACKME.004020F4
00401013  E8 A6040000 CALL <JMP.&USER32.FindWindowA>
00401018  0BC0      OR EAX,EAX
0040101A  74 01      JE SHORT CRACKME.00401010
0040101C  C3        RETN
0040101D  C705 64204000 MOV DWORD PTR DS:[402064],4003

```

CRACKME UPX.EXE 的入口点

```

00409BF0  60      PUSHAD
00409BF1  BE 00904000 MOV ESI,CRACKME_.00409000
00409BF6  8DBE 0080FFFF LEA EDI,DWORD PTR DS:[ESI+FFFF8000]
00409BFC  57      PUSH EDI
00409BFD  83CD FF  OR EBP,FFFFFFFF
00409C00  EB 10    JMP SHORT CRACKME_.00409C12
00409C02  90      NOP
00409C03  90      NOP
00409C04  90      NOP
00409C05  90      NOP
00409C06  90      NOP

```

正如大家所看到的,CRACKME UPX 的入口点变成了 409BF0,将从这里开始执行解密例程,如果我们定位 401000 地址处,会发现找不到原程序的代码任何踪迹。

```

00401000  0000      ADD BYTE PTR DS:[EAX],AL
00401002  0000      ADD BYTE PTR DS:[EAX],AL
00401004  0000      ADD BYTE PTR DS:[EAX],AL
00401006  0000      ADD BYTE PTR DS:[EAX],AL
00401008  0000      ADD BYTE PTR DS:[EAX],AL
0040100A  0000      ADD BYTE PTR DS:[EAX],AL
0040100C  0000      ADD BYTE PTR DS:[EAX],AL
0040100E  0000      ADD BYTE PTR DS:[EAX],AL
00401010  0000      ADD BYTE PTR DS:[EAX],AL
00401012  0000      ADD BYTE PTR DS:[EAX],AL
00401014  0000      ADD BYTE PTR DS:[EAX],AL
00401016  0000      ADD BYTE PTR DS:[EAX],AL
00401018  0000      ADD BYTE PTR DS:[EAX],AL
0040101A  0000      ADD BYTE PTR DS:[EAX],AL
0040101C  0000      ADD BYTE PTR DS:[EAX],AL

```

正如大家所看到的原程序的代码段是空的,即加壳程序将原程序代码段加密/压缩后保存到其他地方,并且将原程序代码段清空了。通常情况下,大部分加壳程序会在待加壳程序中创建自己的区段,从自己的区段开始执行解压/解密程序,解压/解密程序会将原程序各个区段进行解压/解密。

下面我们一起来看看解压/解密的例程,不运行,直接往下拉。

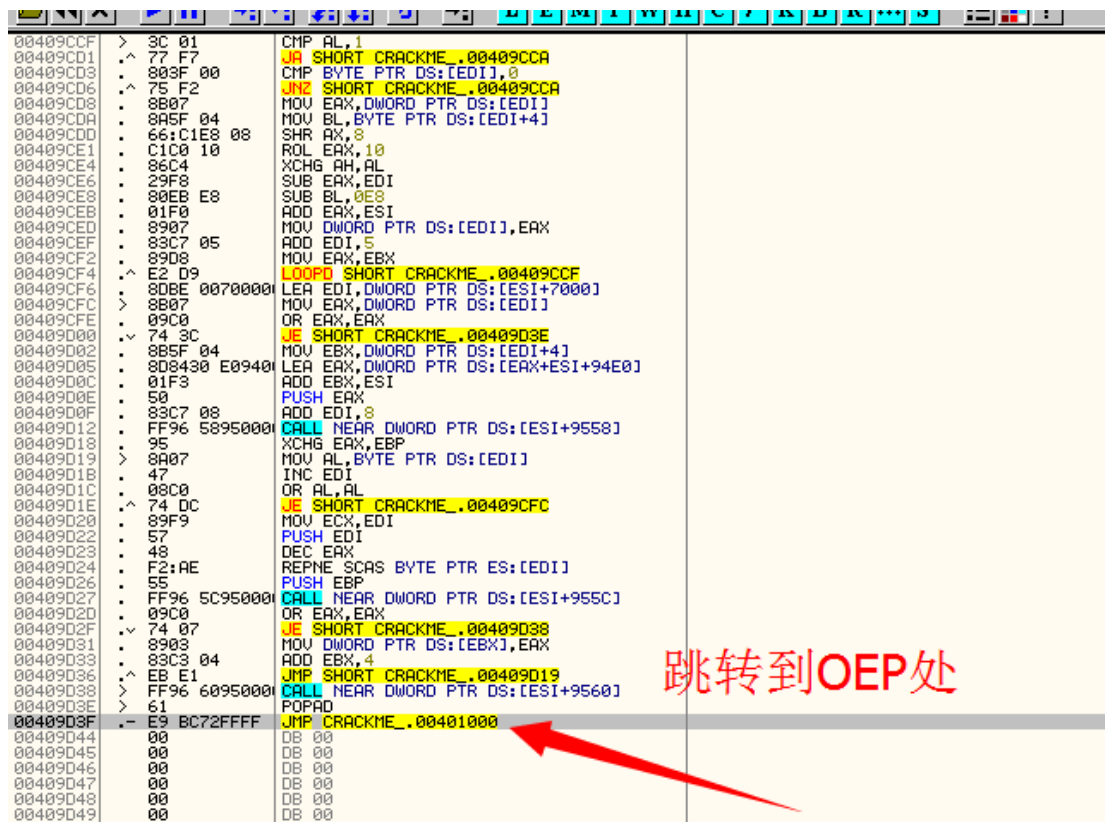


```

00409BF0 $ 60 PUSHAD
00409BF1 > BE 00904000 MOV ESI,CRACKME_.00409000
00409BF6 . 8DBE 0080FFFF LEA EDI,DWORD PTR DS:[ESI+FFFF8000]
00409BFC . 57 PUSH EDI
00409BFD . 83CD FF OR EBP,FFFFFFFF
00409C00 . EB 10 JMP SHORT CRACKME_.00409C12
00409C03 . 90 NOP
00409C04 . 90 NOP
00409C05 . 90 NOP
00409C06 . 90 NOP
00409C07 . 90 NOP
00409C08 > 8A06 MOV AL,BYTE PTR DS:[ESI]
00409C0A . 46 INC ESI
00409C0B . 8807 MOV BYTE PTR DS:[EDI],AL
00409C0D . 47 INC EDI
00409C0E > 01DB ADD EBX,EBX
00409C10 > 75 07 JNZ SHORT CRACKME_.00409C19
00409C12 > 8B1E MOV EBX,DWORD PTR DS:[ESI]
00409C14 . 83EE FC SUB ESI,-4
00409C17 . 11DB ADC EBX,EBX
00409C19 > 72 ED JB SHORT CRACKME_.00409C08
00409C1B . B8 01000000 MOV EAX,1
00409C1D > 01DB ADD EBX,EBX
00409C1F . 75 07 JNZ SHORT CRACKME_.00409C2B
00409C21 . 8B1E MOV EBX,DWORD PTR DS:[ESI]
00409C23 . 83EE FC SUB ESI,-4
00409C25 . 11DB ADC EBX,EBX
00409C27 > 11C0 ADC EAX,EAX
00409C29 > 01DB ADD EBX,EBX
00409C2B . 73 EF JNB SHORT CRACKME_.00409C20
00409C2D . 75 09 JNZ SHORT CRACKME_.00409C3C
00409C2F . 8B1E MOV EBX,DWORD PTR DS:[ESI]
00409C31 . 83EE FC SUB ESI,-4
00409C33 . 11DB ADC EBX,EBX
00409C35 . 73 F4 JNB SHORT CRACKME_.00409C20
00409C37 > 31C9 XOR ECX,ECX
00409C39 . 83E8 03 SUB EAX,3
00409C3B > 72 00 JB SHORT CRACKME_.00409C50
00409C3D . C1E0 08 SHL EAX,8
00409C3F . 8A06 MOV AL,BYTE PTR DS:[ESI]
00409C41 . 46 INC ESI
00409C43 . 83F0 FF XOR EAX,FFFFFFFF
00409C45 > 74 74 JE SHORT CRACKME_.00409CC2
00409C47 . 89C5 MOV EBP,EAX
00409C49 > 01DB ADD EBX,EBX
00409C4B > 75 07 JNZ SHORT CRACKME_.00409C5B
00409C4D . 8B1E MOV EBX,DWORD PTR DS:[ESI]
00409C4F . 83EE FC SUB ESI,-4
00409C51 . 11DB ADC EBX,EBX
00409C53 > 11C9 ADC ECX,ECX
00409C55 > 01DB ADD EBX,EBX
00409C57 > 75 07 JNZ SHORT CRACKME_.00409C68
00409C59 . 8B1E MOV EBX,DWORD PTR DS:[ESI]
00409C5B . 83EE FC SUB ESI,-4
00409C5D . 11DB ADC EBX,EBX
00409C5F > 11C9 ADC ECX,ECX
00409C61 > 75 07 JNZ SHORT CRACKME_.00409C8C

```

继续往下,直到你看到:



```

00409CCF > 3C 01 CMP AL,1
00409CD1 . 77 F7 JA SHORT CRACKME_.00409CCA
00409CD3 . 803F 00 CMP BYTE PTR DS:[EDI],0
00409CD6 . 75 F2 JNZ SHORT CRACKME_.00409CCA
00409CD8 . 8B07 MOV EAX,DWORD PTR DS:[EDI]
00409CDA . 8A5F 04 MOV BL,BYTE PTR DS:[EDI+4]
00409CDD . 66:C1E8 08 SHR AX,8
00409CE1 . C1C0 10 ROL EAX,10
00409CE4 . 86C4 XCHG AH,AL
00409CE6 . 29F8 SUB EAX,EDI
00409CE8 . 80EB E8 SUB BL,0E8
00409CEA . 01F0 ADD EAX,ESI
00409CEC . 8907 MOV DWORD PTR DS:[EDI],EAX
00409CE6 . 83C7 05 ADD EDI,5
00409CF2 . 89D8 MOV EAX,EBX
00409CF4 . E2 D9 LOOPD SHORT CRACKME_.00409CCF
00409CF6 . 8DBE 00700000 LEA EDI,DWORD PTR DS:[ESI+7000]
00409CFC > 8B07 MOV EAX,DWORD PTR DS:[EDI]
00409CFE . 09C0 OR EAX,EAX
00409D00 > 74 3C JE SHORT CRACKME_.00409D3E
00409D02 . 8B5F 04 MOV EBX,DWORD PTR DS:[EDI+4]
00409D05 . 8D8430 E09400 LEA EAX,DWORD PTR DS:[EAX+ESI+94E0]
00409D0C . 01F3 ADD EBX,ESI
00409D0E . 50 PUSH EAX
00409D0F . 83C7 08 ADD EDI,8
00409D12 . FF96 58950000 CALL NEAR DWORD PTR DS:[ESI+9558]
00409D18 . 95 XCHG EAX,EBP
00409D19 > 8A07 MOV AL,BYTE PTR DS:[EDI]
00409D1B . 47 INC EDI
00409D1C . 08C0 OR AL,AL
00409D1E . 74 DC JE SHORT CRACKME_.00409CFC
00409D20 . 89F9 MOV ECX,EDI
00409D22 . 57 PUSH EDI
00409D23 . 48 DEC EAX
00409D24 . F2:AE REPNE SCAS BYTE PTR ES:[EDI]
00409D26 . 55 PUSH EBP
00409D27 . FF96 5C950000 CALL NEAR DWORD PTR DS:[ESI+955C]
00409D2D . 09C0 OR EAX,EAX
00409D2F > 74 07 JE SHORT CRACKME_.00409D38
00409D31 . 8903 MOV DWORD PTR DS:[EBX],EAX
00409D33 . 83C3 04 ADD EBX,4
00409D36 . EB E1 JMP SHORT CRACKME_.00409D19
00409D38 > FF96 60950000 CALL NEAR DWORD PTR DS:[ESI+9560]
00409D3E . 61 POPAD
00409D3F . E9 BC72FFFF JMP 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

```

我们可以看到从解压例程开始执行,解压例程执行完毕以后就会跳往 OEP 处,没有做任何的隐藏工作,是不是很有喜感,不难吧,我们继续。

我们给 JMP OEP 这条指令处设置一个断点。

00409031	8903	MOV DWORD PTR DS:[EBX],EAX	
00409033	83C3 04	ADD EBX,4	
00409036	EB E1	JMP SHORT CRACKME_.00409019	
00409038	FF96 6095000	CALL DWORD PTR DS:[ESI+9560]	
0040903E	61	POPAD	
0040903F	E9 BC72FFFF	JMP CRACKME_.00401000	
00409044	00	DB 00	
00409045	00	DB 00	
00409046	00	DB 00	
00409047	00	DB 00	
00409048	00	DB 00	

运行起来。

00409038	FF96 6095000	CALL DWORD PTR DS:[ESI+9560]	
0040903E	61	POPAD	
0040903F	E9 BC72FFFF	JMP CRACKME_.00401000	
00409044	00	DB 00	
00409045	00	DB 00	
00409046	00	DB 00	
00409047	00	DB 00	
00409048	00	DB 00	

我们可以看到断下来了,这个时候原程序代码段已经解压修复完毕,我们按 F7 键跳转到 OEP 处(原程序代码段的第一行)。

00401000	6A 00	PUSH 0	
00401002	E8 FF040000	CALL CRACKME_.00401506	JMP to kernel32.GetModuleHandleA
00401007	A3 CA204000	MOV DWORD PTR DS:[4020CA],EAX	
0040100C	6A 00	PUSH 0	
0040100E	68 F4204000	PUSH CRACKME_.004020F4	
00401013	E8 A6040000	CALL CRACKME_.004014BE	ASCII "No need to disasm the code!"
00401018	0BC0	OR EAX,EAX	JMP to USER32.FindWindowA
0040101A	74 01	JE SHORT CRACKME_.0040101D	
0040101C	C3	RETN	
0040101D	C705 64204000	MOV DWORD PTR DS:[402064],4003	
00401027	C705 68204000	MOV DWORD PTR DS:[402068],CRACKME_.0040	
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 74204000	MOV DWORD PTR DS:[402074],EAX	
0040104F	6A 64	PUSH 64	
00401051	50	PUSH EAX	
00401052	E8 D1030000	CALL CRACKME_.00401428	JMP to USER32.LoadIconA
00401057	A3 78204000	MOV DWORD PTR DS:[402078],EAX	
0040105C	68 007F0000	PUSH 7F00	
00401061	6A 00	PUSH 0	

这里我们可以看到到了 OEP 处,跟原 CrackMe 的入口点处代码一样。至此,解压例程就完成了其全部任务,整个过程是首先解压恢复了原程序的各个区段,原程序代码段恢复完毕以后,接着跳转到 OEP 处开始执行原程序代码,大家应该还记得刚加载的时候,401000 处是空的吧。

以下是加过壳的程序执行的整个流程(PS:现在的一些强壳的执行流程已经不能简单的这么概括了,以后再讨论):

- 1.执行解压/解密例程
- 2.解压/解密原程序的各个区段的数据
- 3.跳往 OEP 处
- 4.执行原程序代码

经过了这么多年的洗礼,加密解密技术水平的提高,壳的作者们也在使用各种奇技淫巧来扩充这个流程,他们会想尽各种办法来隐藏 OEP,并且还有加入各种各样的反调试保护手段,但是不管怎么变,加过壳的程序执行的常规流程就是上面说的几点(PS:如今的强壳以上几点不足以概括了,按 kanxue 老大的话来说就是“壳里有肉,肉里有壳”,以后我们再讨论)。

我们继续来讨论 CRACKME UPX。

很明显,原代码段是空的,在合适的时间,解压/解密程序会将解压/解密后的代码段数据重新写到这里,所以我们可以对这里设置内存访问断点,当解压/解密例程向这里写入数据的时候就会断下来。

我们一起来看看。

L	E	M	T	W	H	C	/	K	B	R	...	S
409000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
003E0000	00004000	CRACKME_	UPX0	PE header	PRIV	RW						
003F0000	00002000	CRACKME_	UPX1	exports	Map	R						
00400000	00001000	CRACKME_	UPX1	code	Imag	R						
00409000	00001000	CRACKME_	UPX1	code	Imag	R						
0040A000	00001000	CRACKME_	UPX1	code	Imag	R						
00410000	0000A000	CRACKME_	UPX1	data, import	Map	R	E					
004D0000	00002000	CRACKME_	UPX1	data, import	Map	R	E					
004FA000	001A3000	CRACKME_	UPX1	data, import	Map	R						

我们将 CRACKME UPX 的区段与 CRACKME 的区段对比一下就会发现,原先的代码段变大了。

003E0000	00004000				Map	R	R	
003F0000	00002000				Priv	Map	R	
00400000	00001000	CRACKME		PE header	Image	R	RWE	
00401000	00001000	CRACKME	CODE	code	Image	R	RWE	
00402000	00001000	CRACKME	DATA	data	Image	R	RWE	
00403000	00001000	CRACKME	.idata	imports	Image	R	RWE	
00404000	00001000	CRACKME	.edata	exports	Image	R	RWE	
00405000	00001000	CRACKME	.reloc	relocations	Image	R	RWE	
00406000	00002000	CRACKME	.rsrc	resources	Image	R	RWE	
00410000	0000A000				Map	R	R	E
004D0000	00002000				Map	R	R	E

原程序的代码段起始地址为 401000,大小占 1000 字节,而 CRACKME UPX 的代码段起始地址变为了 409000。

这里,我们直接给原程序代码段设置内存访问断点,接着运行起来。

003F0000	00002000				Map	R	R	
00400000	00001000	CRACKME		PE header	Image	R	RWE	
00401000	00008000	CRACKME					RWE	
00409000	00001000	CRACKME	Actualize				RWE	
0040A000	00001000	CRACKME	Dump in CPU				RWE	
00410000	0000A000		Dump				RWE	
004D0000	00002000		Search			Ctrl+B	RWE	
004E0000	00103000						RWE	
005F0000	000EE000						RWE	
00900000	00002000						RWE	
58C30000	00001000	COMC					RWE	
58C31000	00070000	COMC				F2	RWE	
58CA1000	00003000	COMC					RWE	
58CA4000	0001F000	COMC					RWE	
58CC3000	00004000	COMC					RWE	
76360000	00001000	COMC					RWE	
76361000	00030000	COMC					RWE	
76391000	00004000	COMC					RWE	
76395000	00012000	COMC					RWE	

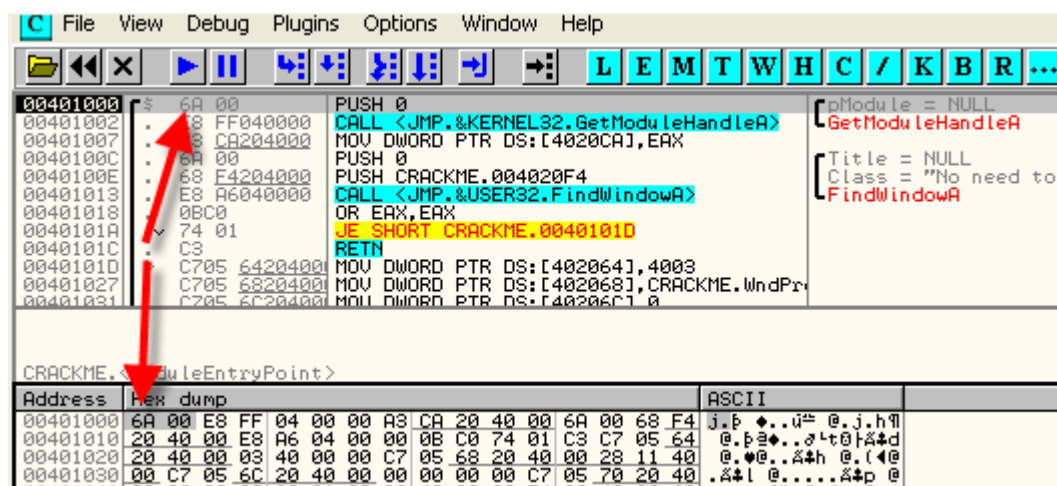
00409C07	90	NOP						
00409C08	> 8A06	MOV AL,BYTE PTR DS:[ESI]						
00409C0A	. 46	INC ESI						
00409C0B	. 8B07	MOV BYTE PTR DS:[EDI],AL						
00409C0D	. 47	INC EDI						
00409C0E	> 010B	ADD EBX,EBX						
00409C10	> 75 07	JNZ SHORT CRACKME_.00409C19						
00409C12	> 8B1E	MOV EBX,DWORD PTR DS:[ESI]						
00409C14	. 83EE FC	SUB ESI,-4						
00409C17	. 110B	ADC EBX,EBX						
00409C19	> 72 ED	JB SHORT CRACKME_.00409C08						
00409C1B	. B8 01000000	MOV EAX,1						
00409C20	> 010B	ADD EBX,EBX						
00409C22	> 75 07	JNZ SHORT CRACKME_.00409C2B						
00409C24	. 8B1E	MOV EBX,DWORD PTR DS:[ESI]						
00409C26	. 83EE FC	SUB ESI,-4						
00409C29	. 110B	ADC EBX,EBX						
00409C2B	> 11C0	ADC EAX,EAX						
00409C2D	. 010B	ADD EBX,EBX						
00409C2F	> 73 EF	JNB SHORT CRACKME_.00409C20						
00409C31	> 75 09	JNZ SHORT CRACKME_.00409C3C						
00409C33	. 8B1E	MOV EBX,DWORD PTR DS:[ESI]						
00409C35	. 83EE FC	SUB ESI,-4						
00409C38	. 110B	ADC EBX,EBX						
00409C3A	> 73 E4	JNB SHORT CRACKME_.00409C20						
00409C3C	> 31C9	XOR ECX,ECX						
00409C3E	. 83E8 03	SUB EAX,3						
00409C41	> 72 0D	JB SHORT CRACKME_.00409C50						
00409C43	. C1E0 08	SHL EAX,8						
00409C46	. 8A06	MOV AL,BYTE PTR DS:[ESI]						

AL=6A ('j')

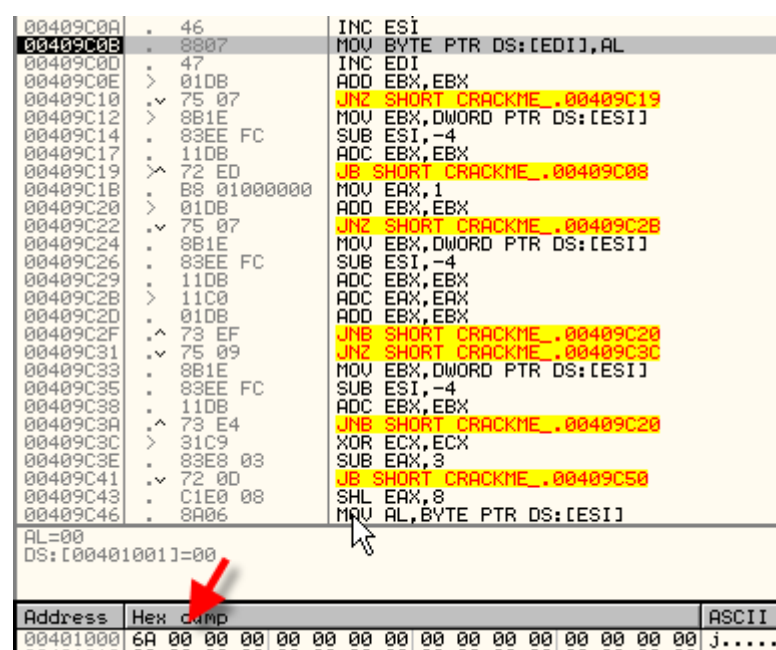
DS:[00401000]=00

Address	Hex	Dump	ASCII
00401000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00401010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00401020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00401030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00401040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00401050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00401060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

断了下来,我们可以看到 401000 开始处的内存单元的第一个字节将被赋值为 AL 的值,此时 AL 值为 6A,我们再来看看原程序 401000 处的第一个字节是什么。



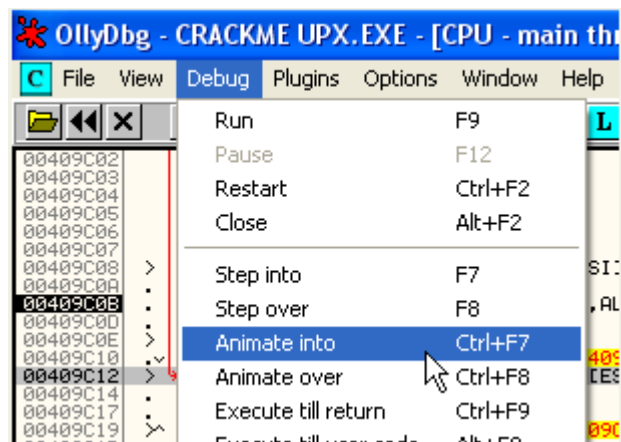
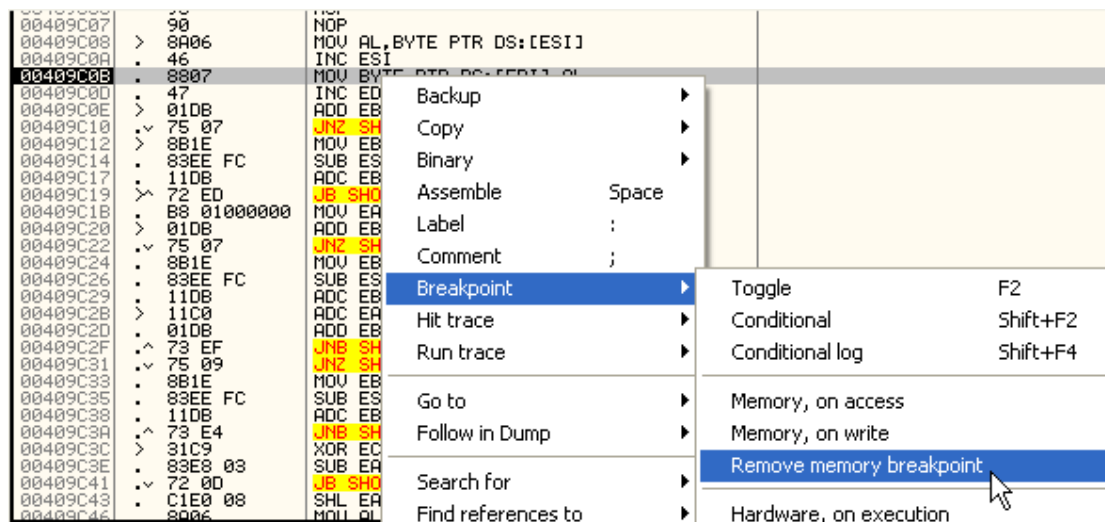
这里我们可以看到原程序入口点 401000 处的第一个字节也是 6A,如果我们按 F9 键,会看到接下来一个字节被填入了 00。



如果我们逐字节的跟踪,可以看到解密/解压例程将原程序代码恢复的整个过程。但是并不是所有的壳都是按照这个顺序来解密/解压原程序区段的,这里我们所看到的是最常规的情况。

如果大家仔细跟踪这个流程的话,就会发现这是一个循环,逐字节读取加密过的字节,接着进行数学运算(例如说:加法,乘法,等等)来解密,运算完毕得到原始字节值,然后将其恢复到原处。

接下来,我们清除掉前面设置的内存访问断点,单击菜单项中 Animate into(自动步入)选项,将能看到原程序代码被逐字节还原的动画过程,原程序各个区段被还原后,就会断在我们前面设置的 JMP OEP 处的断点处。



这里我们可以看到解压例程解压原程序代码段的动画过程。

00409C0E	> 01DB	ADD EBX,EBX	
00409C10	75 07	JNZ SHORT CRACKME_.00409C19	
00409C12	8B1E	MOV EBX,DWORD PTR DS:[ESI]	
00409C14	83EE FC	SUB ESI,-4	
00409C17	110B	ADC EBX,EBX	
00409C19	72 ED	JB SHORT CRACKME_.00409C08	
00409C1B	B8 01000000	MOV EAX,1	
00409C20	01DB	ADD EBX,EBX	
00409C22	75 07	JNZ SHORT CRACKME_.00409C2B	
00409C24	8B1E	MOV EBX,DWORD PTR DS:[ESI]	
00409C26	83EE FC	SUB ESI,-4	
00409C29	110B	ADC EBX,EBX	
00409C2B	11C0	ADC EAX,EAX	
00409C2D	01DB	ADD EBX,EBX	
00409C2F	73 EF	JNB SHORT CRACKME_.00409C20	
00409C31	75 09	JNZ SHORT CRACKME_.00409C3C	
00409C33	8B1E	MOV EBX,DWORD PTR DS:[ESI]	
00409C35	83EE FC	SUB ESI,-4	
00409C38	110B	ADC EBX,EBX	
00409C3A	73 E4	JNB SHORT CRACKME_.00409C20	
00409C3C	31C9	XOR ECX,ECX	
00409C3E	83E8 03	SUB EAX,3	
00409C41	72 0D	JB SHORT CRACKME_.00409C50	
00409C43	C1E0 08	SHL EAX,8	
00409C46	8A06	MOV AL,BYTE PTR DS:[ESI]	
00409C48	46	INC ESI	
00409C49	83F0 FF	XOR EAX,FFFFFFFF	
00409C4C	74 74	JE SHORT CRACKME_.00409CC2	
00409C4E	89C5	MOV EBP,EAX	
00409C50	01DB	ADD EBX,EBX	
00409C52	75 07	JNZ SHORT CRACKME_.00409C5B	
00409C54	8B1E	MOV EBX,DWORD PTR DS:[ESI]	
00409C56	83EE FC	SUB ESI,-4	
00409C59	110B	ADC EBX,EBX	
00409C5B	11C9	ADC ECX,ECX	

Jump is NOT taken
00409C50=CRACKME_.00409C50

Address	Hex dump	ASCII
00401000	6A 00 E8 00 00 05 02 A3 CA 20 40 00 6A 00 68 F4	J..*000 @.j.h
00401010	20 40 00 E8 00 00 04 BA 0B C0 74 01 C3 C7 05 64	@.b..0 8t0t&4d
00401020	20 40 00 03 40 00 00 C7 05 68 20 40 00 28 11 40	@.0@..&4h @.(4@
00401030	00 C7 05 6C 20 40 00 00 00 00 00 00 00 00 00	.&4l @.....
00401040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00401070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

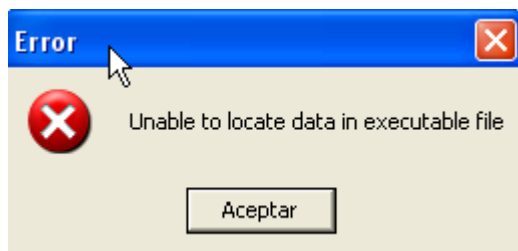
当执行到这里的时候,我们可以看到解压例程在循环读取加密后的原程序区段数据,对其进行解密运算,然后填充到原程序区段所在的原始位置,原程序所有区段被解密还原后就断在了 JMP OEP 处。

这里我们按 F7 键跳转到 OEP 处。

00401000	6A 00	PUSH 0	
00401002	E8 FF040000	CALL CRACKME_.00401506	JMP to kernel32.GetModuleHandleA
00401007	A3 CA204000	MOV DWORD PTR DS:[4020CA],EAX	
0040100C	6A 00	PUSH 0	
0040100E	68 F4204000	PUSH CRACKME_.004020F4	ASCII "No need to disasm the code!"
00401013	E8 A6040000	CALL CRACKME_.004014BE	JMP to USER32.FindWindowA
00401018	0BC0	OR EAX,EAX	
0040101A	74 01	JE SHORT CRACKME_.0040101D	
0040101C	C3	RETN	
0040101D	C705 64204000	MOV DWORD PTR DS:[402064],4003	
00401027	C705 68204000	MOV DWORD PTR DS:[402068],CRACKME_.0040	
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 74204000	MOV DWORD PTR DS:[402074],EAX	
0040104F	6A 64	PUSH 64	
00401051	50	PUSH EAX	
00401052	F8 01030000	CALL CRACKME_.00401428	JMP to USER32.LoadIconB

这里我们就到达了 OEP 处,我们之前说过加壳后的程序使用 OllyDbg 修改代码是不能直接保存存到文件的,下面我们一起来看一看。

例如:我们想将前两个字节 6A 00 修改为 90 90,然后尝试保存存到文件,OllyDbg 会报错:



这里 OD 提示在可执行文件中定位不到该数据。

如果你执意要修改的话,可以使用十六进制编辑器将 401000 对应的数据修改为 90 90,然后用 OllyDbg 加载该修改过的程序,接着来到 401000 处,可以看到是 90 90 开头,然后紧接着全是 00,当解密例程开始解密时,会将解密后的原始的 6A 00 再次填充到 401000 处,当解密区段完成并跳往 OEP 后,依然执行的是 6A 00 而不是 90 90,也就是说我们所做的修改将被解密例程覆盖掉了。

也就是说,如果你想修改这些字节的话,你需要定位到读取加密后字节的地方,在其解密出原始字节时,将其修改掉,这样才能成功修改这些字节,但是如果这样弄的话,需要做的工作太多了,得不偿失,大家把这个过程理解为文件补丁,如果待打补丁的程序被加壳,压缩或者有自校验,则补丁就不会被顺利利用。正因为有了如此不便之处,所以,在文件补丁技术以外,还需要一种更加高阶,隐蔽的方法,也就是内存补丁。

内存补丁的总体思想就是在某个时刻(解压,校验或某种情况发生以后),在目标程序的地址空间中修改数据,因此也被称为 Loader,每次使用时都需要调用程序运行。当然这就不是本教程所要讨论的话题了(PS:如果大家想了解内存补丁的话,可以参考[加密解密 3](#))。

好了,本章就到这里,下一章节我们将讨论该 CrackMe 的脱壳。