

第五十八章-ExeCryptor v2.2.50.h

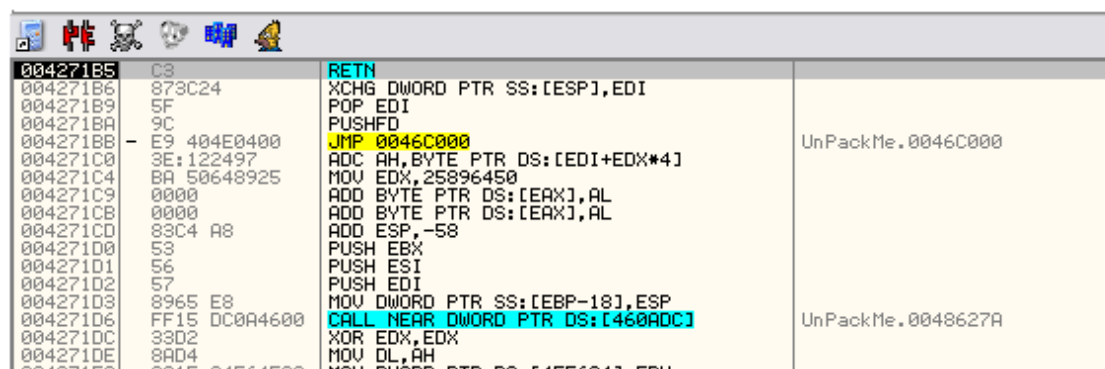
好,本章还剩下 ExeCryptor 的最后三个 UnPackMe,这三个 UnPackMe 是本系列教程中最难脱的。我们来尝试对它们进行脱壳,如果没有脱壳成功的话,至少也会记录下脱壳思路,供大家学习。

我们双击运行 UnPackMe H,可以看到它新增了如下保护:

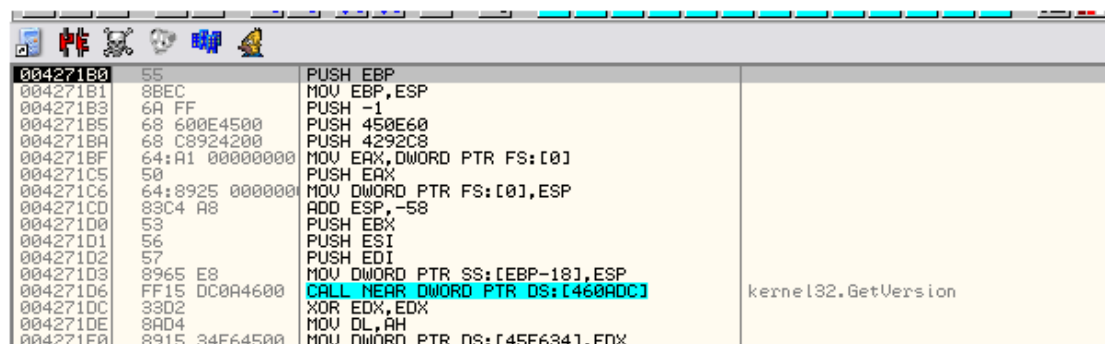


我们可以看到入口点保护开启了。也就是说,入口点可能被隐藏起来了。

好,我们还是跟之前一样对代码段设置 break-on-execute 断点,运行起来,断到了这里。

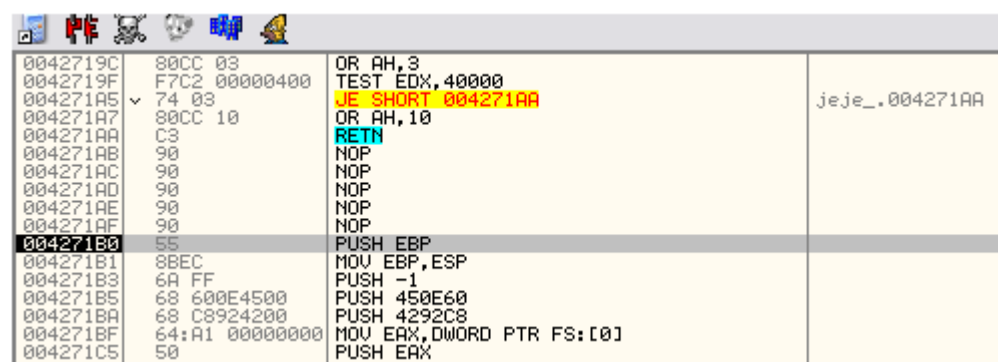


貌似有点不对劲,我们看看之前脱壳过的版本。



我们可以看到入口点处的代码明显被隐藏了。

这是之前脱壳版本的截图:



这是现在 UnPackMe H 的截图:

004271A5	74 03	JE SHORT 004271AA	UnPackMe.004271AA
004271A7	80CC 10	OR AH,10	
004271AA	C3	RETN	
004271AB	90	NOP	
004271AC	90	NOP	
004271AD	90	NOP	
004271AE	90	NOP	
004271AF	90	NOP	
004271B0	E9 3A820500	JMP 0047F3EF	UnPackMe.0047F3EF
004271B5	C3	RETN	
004271B6	873C24	XCHG DWORD PTR SS:[ESP],EDI	
004271B9	5F	POP EDI	
004271BA	9C	PUSHFD	
004271BB	E9 404E0400	JMP 0046C000	UnPackMe.0046C000
004271C0	3E:122497	ADC AH,BYTE PTR DS:[EDI+EDX*4]	
004271C4	BA 50648925	MOV EDX,25896450	
004271C9	0000	ADD BYTE PTR DS:[EAX],AL	
004271CB	0000	ADD BYTE PTR DS:[EAX],AL	
004271CD	83C4 A8	ADD ESP,-58	
004271D0	53	PUSH EBX	

对比着看,明显有差别。

我们来看看 UnPackMe H 此时堆栈的情况:

		SI3 empty 0.0	
		ST4 empty 0.0	
0012FFB4	0046D7E0	UnPackMe.0046D7E0	
0012FFB8	0047F3EF	UnPackMe.0047F3EF	
0012FFBC	00472431	UnPackMe.00472431	
0012FFC0	FFFFFFFF		
0012FFC4	7C816FD7	RETURN to kernel32.7C816FD7	
0012FFC8	7C920738	ntdll.7C920738	
0012FFCC	FFFFFFFF		
0012FFD0	7FFD9000		
0012FFD4	8054A938		
0012FFD8	0012FFC8		
0012FFDC	FE0D5260		
0012FFE0	FFFFFFFF	End of SEH chain	
0012FFE4	7C839AA8	SE handler	
0012FFE8	7C816FE0	kernel32.7C816FE0	
0012FFEC	00000000		
0012FFF0	00000000		
0012FFF4	00000000		
0012FFF8	004EFE34	UnPackMe.<ModuleEntryPoint>	
0012FFFC	00000000		

我们再来看看之前脱过壳的版本:

0012FFC4	7C816FD7	RETURN to kernel32.7C816FD7	
0012FFC8	7C920738	ntdll.7C920738	
0012FFCC	FFFFFFFF		
0012FFD0	7FFD9000		
0012FFD4	8054A938		
0012FFD8	0012FFC8		
0012FFDC	819A0020		
0012FFE0	FFFFFFFF	End of SEH chain	
0012FFE4	7C839AA8	SE handler	
0012FFE8	7C816FE0	kernel32.7C816FE0	
0012FFEC	00000000		
0012FFF0	00000000		
0012FFF4	00000000		
0012FFF8	004271B0	jeje_.<ModuleEntryPoint>	
0012FFFC	00000000		

我们可以看到之前脱壳修复后的版本断在 OEP 处时,栈顶指针指向的地址是 12FFC4(不同的机器这个地址可能会不同)。根据堆栈平衡的原理,对于大部分壳(PS:有少数壳可能会玩一些把戏,比如说:ExeCryptor,它利用了 TLS 在入口点之前执行代码,所以此时的栈顶指针可能与 OEP 处时的栈顶指针不一致)来说,用 OD 加载断在入口点处时的栈顶指针应该和断在 OEP 处时的栈顶指针是保持一致的。

就我们当前这个例子来说,入口点处时栈顶指针 ESP 指向的地址是 12FFC4。大家在平时脱壳的时候也要多多留意入口点处的栈顶指针指向了哪里。

成功脱壳后,OEP 处的第一条指令应该是 PUSH EBP。

下面我们来执行 PUSH EBP 这条指令。

我们可以看到 EBP 的值被压入到堆栈中了:

我们可以看到 EBP 的值被保存到 12FFC0 中了,这是原程序执行的第一条指令。下面我们来看看 UnPackMe H,此时的栈顶指针指向的地址明显高于 12FFC4,也不等于 12FFC0。

我们继续观察堆栈:

Address	Disassembly	Comment
0012FFB4	004607E0	UnPackMe.004607E0
0012FFB8	0047F3EF	UnPackMe.0047F3EF
0012FFBC	00472431	UnPackMe.00472431
0012FFC0	FFFFFFFF	
0012FFC4	7C816FD7	RETURN to kernel32.7C816FD7
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD9000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	FE005260	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8390A8	SE handler
0012FFE8	7C816FE0	kernel32.7C816FE0
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	004EFE34	UnPackMe.<ModuleEntryPoint>
0012FFFF	00000000	

正常来说,到达 OEP 处时,栈顶指针指向 12FFC4 才对,这里我们姑且算它执行了 PUSH EBP,那么栈顶指针也应该指向的是 12FFC0 才对。

也就是说该 UnPackMe 模拟执行了 PUSH EBP,并且中间掺杂了大量的垃圾指令。

这里我们对黄色标注出来的区域设置硬件执行断点或者 BP 断点。

Address	Disassembly	Comment
004271AF	NOP	
004271B0	- E9 3A820500	UnPackMe.0047F3EF
004271B5	C3	
004271B6	873C24	XCHG DWORD PTR SS:[ESP],EDI
004271B9	5F	POP EDI
004271BA	9C	PUSHFD
004271BB	- E9 404E0400	UnPackMe.0046C000
004271C0	3E:122497	ADC AH, BYTE PTR DS:[EDI+EDX*4]
004271C4	BA 50648925	MOV EDX, 25896450
004271C9	0000	ADD BYTE PTR DS:[EAX], AL
004271CB	0000	ADD BYTE PTR DS:[EAX], AL
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]
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	89D0 30E64500	MOV DWORD PTR DS:[45E630], ECX
004271F4	C1E1 08	SHL ECX, 8
004271F7	03CA	ADD ECX, EDX
004271F9	89D0 2CF64500	MOV DWORD PTR DS:[45F62C], ECX

从这里开始该 UnPackMe 就没有继续模拟执行指令了,跟原程序是一样的。我们来看看断在这里时的堆栈情况:

0042718B	- E9 404E0400	JMP 0046C000	UnPackMe.0046C000
004271C0	3E:122497	ADC AH, BYTE PTR DS:[EDI+EDX*4]	
004271C4	BA 50648925	MOV EDX, 25896450	
004271C9	0000	ADD BYTE PTR DS:[EAX], AL	
004271CB	0000	ADD BYTE PTR DS:[EAX], AL	
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]	UnPackMe.0048627A
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	

断在了这里,我们来看下堆栈:

0012FFB0	0012FFE0	Pointer to next SEH record
0012FFB4	004292C8	SE handler
0012FFB8	00450E60	UnPackMe.00450E60
0012FFBC	FFFFFFFF	
0012FFC0	0012FFF0	
0012FFC4	7C816FD7	RETURN to kernel32.7C816FD7
0012FFC8	7C920738	ntdll.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFD7000	
0012FFD4	8054A938	
0012FFD8	0012FFC8	
0012FFDC	83A6F660	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C839AA8	SE handler
0012FFE8	7C816FE0	kernel32.7C816FE0
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	004EFE34	UnPackMe.<ModuleEntryPoint>
0012FFFC	00000000	

我们可以看到此时 12FFC0 中已经存放了 12FFF0(EBP 寄存器的初始值),也就是说该 UnPackMe 已经成功模拟执行了 PUSH EBP 指令,接着 12FFBC 中存放了 FFFFFFFF,相当于模拟执行了 PUSH -1 指令。我们知道 PUSH EBP 与 PUSH -1 这两条指令之间应该有一条 MOV EBP,ESP 指令才对。

我们耐心看的话,就会发现 4271CD 前面的几条指令都被成功模拟执行了。

Registers (FPU)	
EAX	00000000
ECX	0012FFB0
EDX	7C91EB94 ntdll.KiFastSys
EBX	7FFDB000
ESP	0012FFC4
EBP	0012FFF0
ESI	FFFFFFFF
EDI	7C920738 ntdll.7C920738
EIP	004271B0 jeje_.<ModuleEn
C 0	ES 0023 32bit 0(FFFFFFFF)
P 1	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 1	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFDF000(
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_SUCCESS (

我们会发现在剥离了 TLS 以后,每次断在 OEP 处时,唯一会变化的就是 EBX 寄存器的值,但是无论怎么变,总是在 7FFDB000~7FFDF000 这个范围之内,反观其他寄存器的值都是固定不变的。

好,我们重启 OD,再次断到 ADD EBP,-58 指令处。

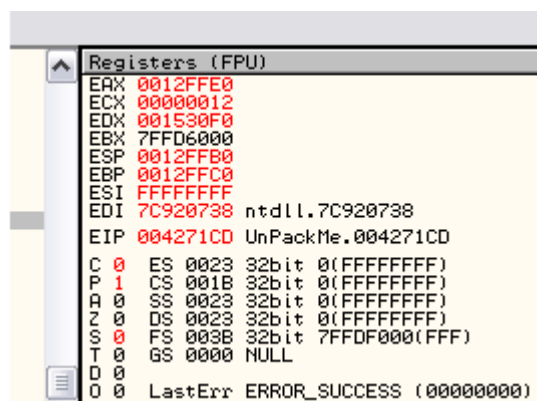
004271CD 83C4 A8 ADD ESP,-58

我们将当前未脱壳和之前已脱壳的寄存器组情况进行对比:

已脱壳:

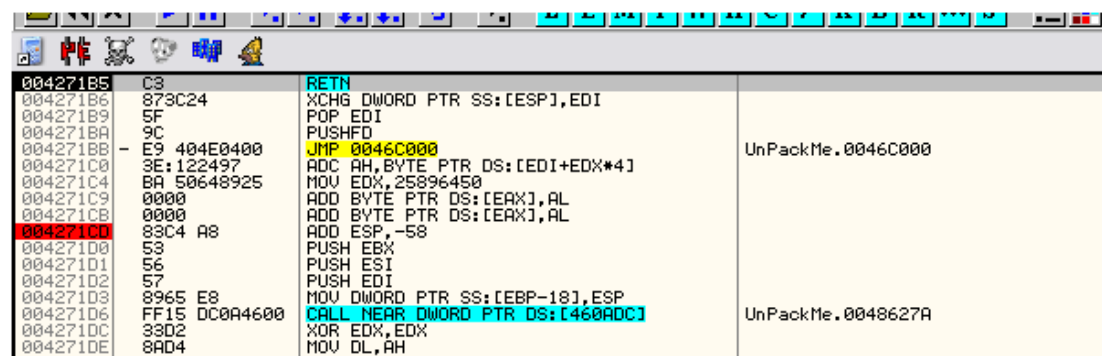
Registers (FPU)	
EAX	0012FFE0
ECX	0012FFB0
EDX	7C91EB94 ntdll.KiFastSystemCallRet
EBX	7FFDF000
ESP	0012FFB0
EBP	0012FFC0
ESI	FFFFFFFF
EDI	7C920738 ntdll.7C920738
EIP	004271CD jeje_.004271CD
C 0	ES 0023 32bit 0(FFFFFFFF)

未脱壳:

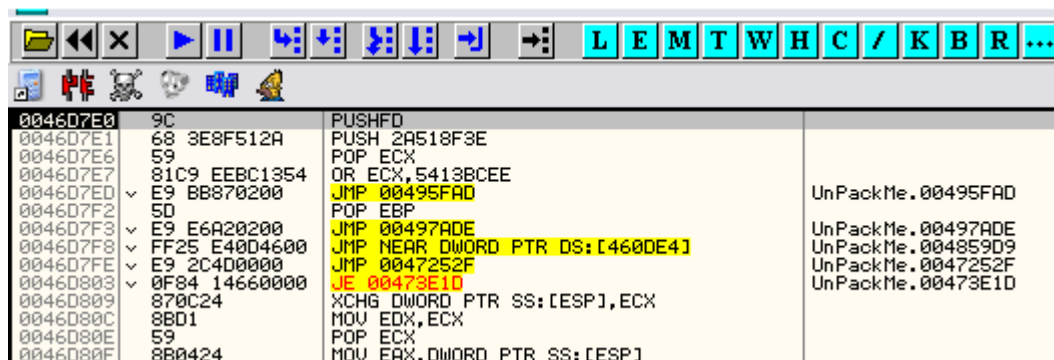


我们可以看到在 UnPackMe 模拟执行指令的过程中连同堆栈以及除 ECX,EDX 以外的其他通用寄存器也都模拟了。也就是当已脱壳和未脱壳的版本处于同一个位置时,我们要留意 EAX,EBX,ESP,EBP,ESI,EDI 这几个寄存器的情况,ECX,EDX 的话,我们并不关心。

好,现在我们回到该 UnPackMe 模拟执行指令的起始地址处。



我们可以看到在 4271B5~4271CD 之间掺杂了大量的垃圾指令-起到混淆的作用。实际上它们是要完成 5 条指令的功能。



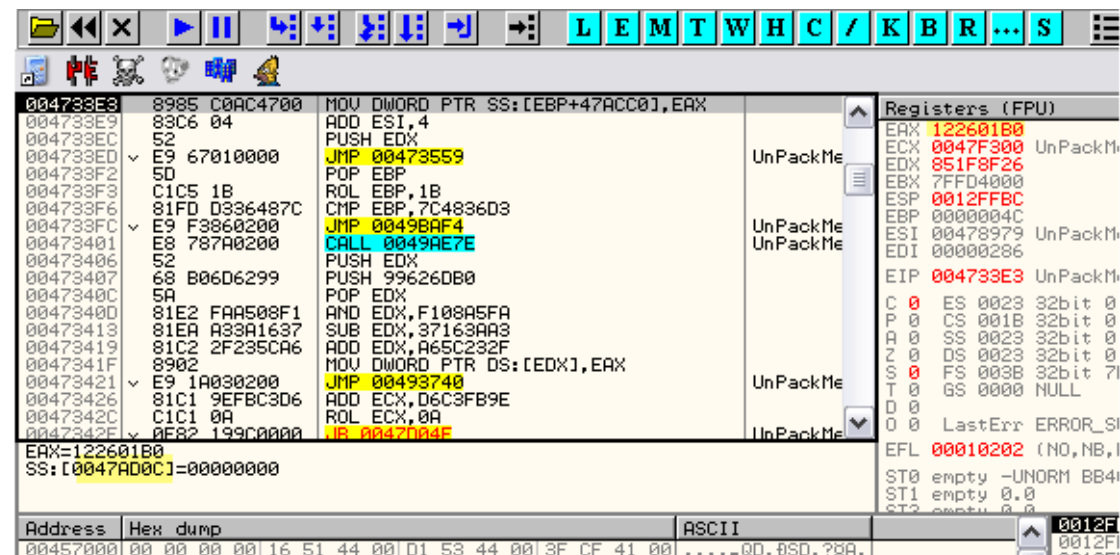
我们按 F7 单步执行 RET 指令,到了这里。

首先是利用 PUSHFD 保存 EFLAGS(标志寄存器)的值。

接下来我们对 ExeCryptor 壳所在区段设置内存写入断点,看看其在执行的过程中会不会保存什么值之类的东西,这里 ExeCryptor 有好几个区段,大家可以逐一尝试。

00340000	00041000			Map	R	R	Device\Hax
00390000	00006000			Map	R	R	Device\Hax
003A0000	00041000			Map	R	R	
003F0000	00001000			Priv	RWE	RWE	
00400000	00001000	UnPackMe	PE header	Imag	R	RWE	
00401000	0004A000	UnPackMe	code	Imag	R	RWE	
0044B000	0000C000	UnPackMe	code	Imag	R	RWE	
00457000	00009000	UnPackMe	data	Imag	R	RWE	
00460000	00003000	UnPackMe		Imag	R	RWE	
00463000	00008000	UnPackMe	resources	Imag	R	RWE	
0046B000	00001000	UnPackMe		Imag	R	RWE	
0046C000	00032000	UnPackMe					
0049E000	00052000	UnPackMe	SF				
004F0000	00001000						
00500000	00001000						
00510000	00009000						
005D0000	00002000						
005E0000	00103000						
006F0000	00001000						Ctrl+B
00700000	00103000						
00A00000	00001000						Ctrl+L
00A10000	00004000						
00A20000	00003000						
00A30000	00004000						
00A40000	00002000						
00A50000	00003000						
00A60000	00002000						
00AD0000	00002000						
00AE0000	00001000						
58C30000	00001000	COMCTL32	PE				

设置完内存写入断点以后,我们运行起来,看看会发生什么。



这里我们可以看到将 EAX 的值保存到 47AD0C 这个内存单元中,但是此时 EAX 的值并不是某个寄存器的初始值,之前并没有见过这个值。虽然没弄明白这条指令实际的作用,但我们还是将什么数值被保存到哪里简单的记录一下吧,方便下面的分析。

EAX = 5A731601 保存到 4815E0 中

EBX = 5A731601 保存到 4815E0 中

EAX = 0046C5DE 保存到 47A90C 中

00496C9C 01 变成了 02

我们要时刻留意堆栈的情况:看看有没有出现 12FFF0 这个值,到目前为止还没有出现这个值,我们继续记录。

这里是利用 POP 指令将 12FFF0 保存到 47A0CC 中

这里

0047949B 89BD C0A44700 MOV DWORD PTR SS: [EBP+47A4C0], EDI

这是保存 EDI 寄存器的值,我们要稍微留意一下,它是要模拟的寄存器之一。

ESI=0046C5DE 保存到 0047A8CC 中

EDX=0047F3EF 保存到 0047BCF8 中

ECX=0047F3EF 保存到 0047B8EC 中

EAX=0047E97E 保存到 0047B4E4 中

EBX=7FFDB000 保存到 00481198 中

这里

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

EAX = 0012FFC0 保存到 0047B0D8 中

这里我们可以看到是 12FFC0,我们继续。

EAX=14F43E15 保存到 0047ACCC 中

00496CAB 01 变成了 00

00496C9C 02 变成了 03

00496D9A 00 变成了 01

跟之前一样 POP

利用 POP 指令将 12FFF0 保存到 47A488 中

EDI = 7C920738 保存到 0047A888 中

这里我们应该欣喜才对,因为出现了 12FFF0,因为要模拟执行 PUSH EBP 的话,EBP 的值应该为 12FFF0 才对,但是 12FFC0 这个值还没有出现。

ESI = FFFFFFFF 保存到 0047AC88 中

这里

00470F82 89B5 C0A84700 MOV DWORD PTR SS: [EBP+47A8C0], ESI

EDX=0047F3EF 保存到 0047C0B4 中

ECX=0047F3EF 保存到 0047BCA8 中

EAX=0047E97E 保存到 0047B8A0 中

EBX=7FFDB000 保存到 00481554 中

这里跟之前某条指令是一样的,保存 EBX 的值

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

虽然我们已经执行了大量的垃圾指令,但是还是没有遇到第一条该程序真正要执行的指令,我们只能耐心的继续往下跟。

EAX = 0012FFC4 保存到 0047B494 中

这里

00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

0012FFC4 是 ESP 的初始值,我们要重点关注。

我们继续。

EAX=1E500000 保存到 0047B088 中

0047B494 C4 FF 12 00

这里我们可以看到 12FFC4-ESP 寄存器的初始值,现在要将其减去 4。

0047711C 83AD CCB04700 0> SUB DWORD PTR SS: [EBP+47B0CC], 4

减去 4 以后

0047B494 C0 FF 12 00

现在变成了 12FFC0。

00496D9A 01 变成了 00

00496C9C 03 变成了 04

00496D8A 00 变成了 01

又是 POP 指令

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFF0

再次将 12FFF0 保存到 0047A448 中

大家应该还记得吧-还没有执行该程序真正要执行的指令呢,嘿嘿

EDI=7C920738 保存到 0047A848 中

再次

0047949B 89BD C0A44700 MOV DWORD PTR SS: [EBP+47A4C0], EDI; ntdll.7C920738

ESI=FFFFFFFF 保存到 0047AC48 中

这里又是跟之前一样保存 ESI 的值

00470F82 89B5 C0A84700 MOV DWORD PTR SS: [EBP+47A8C0], ESI

继续耐心往下跟踪

EDX=0047F3EF 保存到 0047C074 中

ECX=0047F3EF 保存到 0047BC68 中

EAX=0047E97E 保存到 0047B860 中

EBX = 00478304 保存到 00481514 中

这里

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX; UnPackMe.0047830

EAX=0012FFBC 保存到 0047B454 中

这里我们可以看到是 12FFBC-第二个值要被压入到这个地址中

EAX=192082C0 保存到 0047B048 中

我们明显的看出这里实际上是一个循环,继续。

EAX=0048F082 保存到 0048191C 中

EBX=0048F082 保存到 0048191C 中

00496D8A 01 变成了 00

00496C9C 04 变成了 05

00496CFB 00 变成了 01

又是 POP 指令

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFF0

12FFF0 保存到 0047A20C 中

EDI=7C920738 保存到 0047A60C 中

这里又是跟之前一样保存 EDI 的值

ESI = FFFFFFFF 保存到 0047AA0C 中

哎呀妈呀,仍然没有看到第一条真正要执行的指令。

EDX=0047F3EF 保存到 0047BE38 中

ECX=0047F3EF 保存到 0047BA2C 中

EAX=0047E97E 保存到 0047B624 中

诶,我们看到了一点曙光

EBX=7FFDB000 保存到 004812D8 中

再次到了这里

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

EAX=0012FFC0 保存到 0047B218 中

大家留心点的话会发现这是最后一次循环,之前 EAX 的值为 12FFC4,现在为 12FFC0。

EAX=1B700602 保存到 0047AE0C 中

EAX=FFFFFFFF 保存到 0047B624 7E E9 47 00 中

这个 FFFFFFFF 也是值得我们关注的,它是要被压入到堆栈中的一个值。

00496CFB 01 变成了 00

00472B9C 00 变成了 01

00472B9C 01 变成了 00

00496C9C 05 变成了 06

00496CEB 00 变成了 01

又是 POP 指令

12FFF0 保存到 0047A1CC 中

然后

EDI=7C920738 保存到 0047A5CC 中

ESI=FFFFFFFF 保存到 0047A9CC 中

EDX=00172CF0 保存到 0047BDF8 中

ECX=00000012 保存到 0047B9EC 中

EAX=00472B00 保存到 0047B5E4 中

还是没有看到真正要执行的第一条指令,我的天。

EBX = 7FFDB000 保存到 00481298 中

EAX = 0012FFC4 保存到 0047B1D8 中

EAX = 18000500 保存到 0047ADCC 中

我们对 12FFC0 设置一个硬件写入断点,看看哪里会调用 PUSH EBP 向 12FFC0 进行写入。

004923C8 871C24 XCHG DWORD PTR SS: [ESP], EBX

The screenshot shows a debugger window with the following components:

- Assembly List:** Displays instructions at various addresses. A red arrow points to the instruction at address 004923C8: `XCHG DWORD PTR SS:[ESP], EBX`.
- Registers (MMX):** A table on the right showing the current values of registers. A red arrow points to the EBP register, which contains the value 0012FFC0.
- Memory Dump:** At the bottom, a table shows memory addresses, hex dumps, and ASCII values. A red arrow points to the address 0047A1CC, which contains the hex value F0 FF 12 00 00 00 00 00.

我们可以看到这一行,从 47A1CC 中读取 12FFF0 的值。

如果往前翻的话,会发现是通过 POP 指令将 12FFF0 保存到 0047A1CC 中的。

12FFF0 这个值是我们重点关注的,它是 EBP 的初始值,这里又通过 XCHG DWORD PTR SS:[EBP],EBX 模拟 PUSH EBP 执行的结果。

接下来,我们会逐步发现真正要执行的指令,嘿嘿

下一条真正要执行的指令是:

004271B1 8BEC MOV EBP, ESP

此时 ESP 的值为 12FFC0,因此执行了 MOV EBP,ESP 以后,EBP 的值也会变成 12FFC0。

这个 12FFC0 是通过减 4 得来的。

0047711C 83AD CCB04700 0> SUB DWORD PTR SS: [EBP+47B0CC], 4

0047B1D8 C4 FF 12 00

我们可以看到 47B1D8 中的值变成了 12FFC0,不出意外的话,应该是要保存到 EBP 中的,我猜测接下来可能要执行 MOV EBP,ESP。

0047B1D8 C0 FF 12 00

下面我们对 12FFBC 设置硬件写入断点,来看看哪里执行了 PUSH -1 将 FFFFFFFF 压入到 12FFBC 中。

004271B0	55	PUSH EBP
004271B1	8BEC	MOV EBP,ESP
004271B3	6A FF	PUSH -1
004271B5	68 600E4500	PUSH 450E60

这里是脱壳修复后的情形。

好,我们继续,看看真正要执行的第二条指令在哪里。

00496CEB 01 变成了 00

00496C9C 06 变成了 07

00496CDB 00 变成了 01

又是 POP 指令

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFF0

将 12FFF0 保存到 0047A18C 中

我们继续

EDI=7C920738 保存到 0047A58C 中

ESI=FFFFFFFF 保存到 0047A98C 中

EDX=00172CF0 保存到 0047BDB8 中

ECX=00000012 保存到 0047B9AC 中

EAX=00472B00 保存到 0047B5A4 中

继续

EBX=7FFDB000 保存到 00481258 中

EAX=12FFC0 保存到 0047B198 中

循环再次开始

EAX=1590F683 保存到 0047AD8C 中

EAX=004820F6 保存到 00481660 中

EBX=004820F6 保存到 00481660 中

我们跟之前一样对堆栈设置硬件写入断点,到了这里

00478520 68 7F354700 PUSH 47357F

明显是垃圾指令,我们不理它。

00496CDB 01 变成了 00

跳过位操作指令

00495F2D 6A FF PUSH-1

嘿嘿,这里是将 FFFFFFFF 保存到 12FFBC 中,这是真正要执行的第二条指令。

004271B0	55	PUSH EBP
004271B1	8BEC	MOV EBP,ESP
004271B3	6A FF	PUSH -1
004271B5	68 600E4500	PUSH 450E60
004271B9	68 C8924200	PUSH 4292C8
004271BF	64:A1 00000000	MOV EAX,DWORD PTR FS:[0]
004271C5	50	PUSH EAX

好,按照这个指令执行的顺序,接下来将是 450E60 保存到 12FFB8 中,4292C8 保存到 12FFB4 中,所以我们接着对 12FFB8 设置硬件写入断点。

00496C9C 07 变成了 08

00496CCB 00 变成了 01

又是 POP 指令

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFC0

这里我每次 POP 都用蓝色标注出来,方便大家观察每次循环的周期。

EDI=7C920738 保存到 0047A54C 中

这里又是保存 EDI 的值。

0047949B 89BD C0A44700 MOV DWORD PTR SS:[EBP+47A4C0], EDI; ntdll.7C920738

每次循环都会保存 EDI 的值,现在也会保存 ESI 的值。

ESI=FFFFFFFF 保存到 0047A94C 中

00470F82 89B5 C0A84700 MOV DWORD PTR SS:[EBP+47A8C0], ESI

这里面有大量的跳转,入栈操作,还有混淆。

其实这些步骤我们完全可能编写脚本来完成,记录每次更新寄存器的指令,有助于我们理解程序的意图。

00470F82 89B5 C0A84700 MOV DWORD PTR SS:[EBP+47A8C0], ESI
00470F88 871C24 XCHG DWORD PTR SS:[ESP], EBX
00470F8B 8BF3 MOV ESI, EBX

ESI=FFFFFFFF
SS:[0047A94C]=00000000

Address	Hex dump	ASCII
0047A94C	00 00 00 00
0047A954	00 00 00 00
0047A95C	00 00 00 00
0047A964	00 00 00 00
0047A96C	00 00 00 00
0047A974	00 00 00 00
0047A97C	00 00 00 00
0047A984	00 00 00 00
0047A98C	FF FF FF FF
0047A994	00 00 00 00
0047A99C	00 00 00 00
0047A9A4	00 00 00 00
0047A9AC	00 00 00 00
0047A9B4	00 00 00 00
0047A9BC	00 00 00 00
0047A9C4	00 00 00 00
0047A9CC	FF FF FF FF
0047A9D4	00 00 00 00
0047A9DC	00 00 00 00
0047A9E4	00 00 00 00
0047A9EC	00 00 00 00
0047A9F4	00 00 00 00
0047A9FC	00 00 00 00
0047AA04	00 00 00 00
0047AA0C	FF FF FF FF
0047AA14	00 00 00 00

Command
Memory breakpoint when executing [00470F82]

这里又是将 ESI 的值保存到空闲区域,存放的从上图来看非常的对称。

EDX=00172CF0 保存到 0047BD78 中

ECX=00000012 保存到 0047B96C 中

这里又是将寄存器的值保存到空白区域,这里保存的是 12。

SS:[0047B564]=00000000

Address	Hex dump	ASCII
0047B96C	12 00 00 00
0047B974	00 00 00 00
0047B97C	00 00 00 00
0047B984	00 00 00 00
0047B98C	00 00 00 00
0047B994	00 00 00 00
0047B99C	00 00 00 00
0047B9A4	00 00 00 00
0047B9AC	12 00 00 00
0047B9B4	00 00 00 00
0047B9BC	00 00 00 00
0047B9C4	00 00 00 00
0047B9CC	00 00 00 00
0047B9D4	00 00 00 00
0047B9DC	00 00 00 00
0047B9E4	00 00 00 00
0047B9EC	12 00 00 00
0047B9F4	00 00 00 00
0047B9FC	00 00 00 00
0047BA04	00 00 00 00
0047BA0C	00 00 00 00
0047BA14	00 00 00 00
0047BA1C	00 00 00 00
0047BA24	00 00 00 00
0047BA2C	EF F3 47 00	*%G....
0047BA34	00 00 00 00

Command

我晕,已经写了这么多页了。

EAX=00472B00 保存到 0047B564 中

EBX=7FFDB000 保存到 00481218 中

这次是保存 EBX 的值。

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

排列的依然很整齐

Address	Hex dump	ASCII
00481218	00 B0 FD 7F 00 00 00 00
00481220	00 00 00 00 00 00 00 00
00481228	00 00 00 00 00 00 00 00
00481230	00 00 00 00 00 00 00 00
00481238	00 00 00 00 00 00 00 00
00481240	00 00 00 00 00 00 00 00
00481248	00 00 00 00 00 00 00 00
00481250	00 00 00 00 00 00 00 00
00481258	00 B0 FD 7F 00 00 00 00
00481260	00 00 00 00 00 00 00 00
00481268	00 00 00 00 00 00 00 00
00481270	00 00 00 00 00 00 00 00
00481278	00 00 00 00 00 00 00 00
00481280	00 00 00 00 00 00 00 00
00481288	00 00 00 00 00 00 00 00
00481290	00 00 00 00 00 00 00 00
00481298	00 B0 FD 7F 00 00 00 00
004812A0	00 00 00 00 00 00 00 00
004812A8	00 00 00 00 00 00 00 00
004812B0	00 00 00 00 00 00 00 00
004812B8	00 00 00 00 00 00 00 00
004812C0	00 00 00 00 00 00 00 00
004812C8	00 00 00 00 00 00 00 00
004812D0	00 00 00 00 00 00 00 00
004812D8	00 B0 FD 7F 00 00 00 00
004812E0	00 00 00 00 00 00 00 00
004812E8	00 00 00 00 00 00 00 00

EAX=12FFBC 保存到 0047B158 中

这里

00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

这些指令执行过程中,ESP 的值一直没有变过,我们来看看它什么时候会变化。

Address	Hex dump	ASCII
0047B158	BC FF 12 00 00 00 00 00
0047B160	00 00 00 00 00 00 00 00
0047B168	00 00 00 00 00 00 00 00
0047B170	00 00 00 00 00 00 00 00
0047B178	00 00 00 00 00 00 00 00
0047B180	00 00 00 00 00 00 00 00
0047B188	00 00 00 00 00 00 00 00
0047B190	00 00 00 00 00 00 00 00
0047B198	C0 FF 12 00 00 00 00 00
0047B1A0	00 00 00 00 00 00 00 00
0047B1A8	00 00 00 00 00 00 00 00
0047B1B0	00 00 00 00 00 00 00 00
0047B1B8	00 00 00 00 00 00 00 00
0047B1C0	00 00 00 00 00 00 00 00
0047B1C8	00 00 00 00 00 00 00 00
0047B1D0	00 00 00 00 00 00 00 00
0047B1D8	C0 FF 12 00 00 00 00 00
0047B1E0	00 00 00 00 00 00 00 00
0047B1E8	00 00 00 00 00 00 00 00
0047B1F0	00 00 00 00 00 00 00 00
0047B1F8	00 00 00 00 00 00 00 00
0047B200	00 00 00 00 00 00 00 00

这里我们可以看到当前的值 12FFBC 以及之前的值 12FFC0。

我们继续,又是一轮循环。

EAX=1EF00200 保存到 0047AD4C 中

现在我们又看到了 SUB 指令了,减去 4,接下来应该是保存 ESP 的值,然后 PUSH,嘿嘿。

0047711C 83AD CCB04700 0-> SUB DWORD PTR SS: [EBP+47B0CC], 4

Address	Hex dump	ASCII
0047B158	BC FF 12 00 00 00 00 00
0047B160	00 00 00 00 00 00 00 00
0047B168	00 00 00 00 00 00 00 00
0047B170	00 00 00 00 00 00 00 00
0047B178	00 00 00 00 00 00 00 00
0047B180	00 00 00 00 00 00 00 00
0047B188	00 00 00 00 00 00 00 00
0047B190	00 00 00 00 00 00 00 00
0047B198	C0 FF 12 00 00 00 00 00
0047B1A0	00 00 00 00 00 00 00 00
0047B1A8	00 00 00 00 00 00 00 00
0047B1B0	00 00 00 00 00 00 00 00
0047B1B8	00 00 00 00 00 00 00 00
0047B1C0	00 00 00 00 00 00 00 00
0047B1C8	00 00 00 00 00 00 00 00
0047B1D0	00 00 00 00 00 00 00 00
0047B1D8	C0 FF 12 00 00 00 00 00
0047B1E0	00 00 00 00 00 00 00 00

Command

Memory breakpoint when executing [0047711C]

这里减去 4。

变成了

Address	Hex dump	ASCII
0047B158	B8 FF 12 00 00 00 00 00
0047B160	00 00 00 00 00 00 00 00
0047B168	00 00 00 00 00 00 00 00
0047B170	00 00 00 00 00 00 00 00
0047B178	00 00 00 00 00 00 00 00
0047B180	00 00 00 00 00 00 00 00
0047B188	00 00 00 00 00 00 00 00
0047B190	00 00 00 00 00 00 00 00
0047B198	C0 FF 12 00 00 00 00 00
0047B1A0	00 00 00 00 00 00 00 00
0047B1A8	00 00 00 00 00 00 00 00
0047B1B0	00 00 00 00 00 00 00 00
0047B1C0	00 00 00 00 00 00 00 00

这里 ESP 的值更新了,那么接下来肯定是模拟 PUSH 指令向堆栈中压入相应的值。

00496CCB 01 变成了 00

00496C9C 08 变成了 09

00496D3B 00 变成了 01

又是 POP 指令

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFC0

12FFC0 保存到 0047A30C 中

我们继续

0047949B 89BD C0A44700 MOV DWORD PTR SS: [EBP+47A4C0], EDI; UnPackMe.0049B8C7

EDI=0049B8C7 保存到 0047A70C 中

继续

ESI=FFFFFFFF 保存到 0047AB0C 中

EDX=00172CF0 保存到 0047BF38 中

ECX=00000012 变成了 00000012

EAX=00472B00 变成了 0047B724

保存 EBX 的值

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

EBX=7FFDB000 保存到 004813D8 中

这里我们可以看到 ESP 的值。

00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

EAX=12ffb4 保存到 0047B318 中

继续

EAX=12A43F15 保存到 0047AF0C 中

貌似新一轮循环开始了

00496D3B 01 变成了 00

00496C9C 09 变成了 0a

00496D2B 00 变成了 01

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFC0

又是 POP 指令

0047A2CC C0 FF 12 00

继续变成了 12FFC0

跟之前一样保存 EDI 的值。

0047949B 89BD C0A44700 MOV DWORD PTR SS: [EBP+47A4C0], EDI; ntdll.7C920738

EDI=7C920738 保存到 0047A6CC 中

ESI=FFFFFFFF 保存到 0047AACC 中

00470F82 89B5 C0A84700 MOV DWORD PTR SS: [EBP+47A8C0], ESI

EDX=00172CF0 保存到 0047BEF8 中

ECX=0047BEF8 保存到 0047BAEC 中

这里有可能连 EDX,ECX 也模拟了。

00491254 8995 ECBC4700 MOV DWORD PTR SS: [EBP+47BCEC], EDX

0049125A 898D E0B84700 MOV DWORD PTR SS: [EBP+47B8E0], ECX

其实这两个寄存器的值无关紧要,既然我们发现了,还是来看看吧。

EAX=00472B00 保存到 0047B6E4 中

保存 EBX 的值

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

EBX=7FFDB000 保存到 00481398 中

保存 ESP 的值

00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

EAX=0012FFB8 保存到 0047B2D8 中

新一轮循环

EAX=09AFCDC0 保存到 0047AECC 中

EAX=00493FCD 保存到 004817A0 中

EBX=00493FCD 保存 004817A0 中

004965A4 8BB5 C0A84700 MOV ESI, DWORD PTR SS: [EBP+47A8C0]

这里向 ESI 中写入内容

0047AACC FF FF FF FF

00496D2B 01 变成了 00

00496C9C 0a 变成了 0b

00496D1B 00 变成了 01

再次利用 POP 指令更新 EBP 的值

0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]; 0012FFC0

这里将 EDI,ESI 恢复为正确的值。

00491254 8995 ECBC4700 MOV DWORD PTR SS: [EBP+47BCEC], EDX

0049125A 898D E0B84700 MOV DWORD PTR SS: [EBP+47B8E0], ECX

00491260 8985 D8B44700 MOV DWORD PTR SS: [EBP+47B4D8], EAX

更新 EDX,ECX,EAX

Registers (FPU)		
EAX	00472B00	UnPac
ECX	00000012	
EDX	00172CF0	
EBX	7FFDE000	
ESP	0012FFB8	
EBP	000001CC	
ESI	00493FD2	UnPac
EDI	00000203	
EIP	00491254	UnPac
C 0	ES 0023	32bit

接着更新 EBX = 7FFDE000

00498D03 899D 8C114800 MOV DWORD PTR SS: [EBP+48118C], EBX

ESP 变成了 0012FFB8

00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

ESP 减去 4

0047711C 83AD CCB04700 0> SUB DWORD PTR SS: [EBP+47B0CC], 4

0047B418 B4 FF 12 00

接下来将是 PUSH 450E60

00491254 8995 ECBC4700 MOV DWORD PTR SS: [EBP+47BCEC], EDX

0049125A 898D E0B84700 MOV DWORD PTR SS: [EBP+47B8E0], ECX

00491260 8985 D8B44700 MOV DWORD PTR SS: [EBP+47B4D8], EAX

Registers (FPU)		
EAX	00472B00	UnPackM
ECX	00450E60	UnPackM
EDX	00172CF0	
EBX	7FFDE000	
ESP	0012FFB8	
EBP	0000030C	
ESI	0048923C	UnPackM
EDI	00000206	
EIP	00491254	UnPackM
C 0	ES 0023	32bit 0

保存到了这里

0048D299 870C24 XCHG DWORD PTR SS: [ESP], ECX

通过 XCHG 指令实现了 PUSH 450E60 的效果,在大量的垃圾中定位到它真心困难。

		EIP 0048D299 UnPackMe.
		C 0 ES 0023 32bit 0(F
0012FFB8	00450E60	UnPackMe.00450E60
0012FFBC	FFFFFFFF	
0012FFC0	0012FFFF	
0012FFC4	7C816D4F	RETURN to kernel32.7C8
0012FFC8	7C920738	nt!L.7C920738
0012FFCC	FFFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	8054B038	
0012FFD8	0012FFC8	
0012FFDC	80E4E020	
0012FFE0	FFFFFFFF	End of SEH chain
0012FFE4	7C8399F3	SE handler
0012FFE8	7C816D58	kernel32.7C816D58
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	004EFE34	UnPackMe.<ModuleEntryF
0012FFFC	00000000	

我们继续,关注重点的地方,其他的略过就行了。

004271B0	55	PUSH EBP
004271B1	8BEC	MOV ESI, EBP
004271B3	6A FF	PUSH EBX
004271B5	68 600E4500	PUSH 450E60
004271B8	68 C8924200	PUSH 4292C8
004271BF	64:A1 00000000	MOV 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	54	PUSH ESI

接下来的原始指令应该是 PUSH 4292C8,我们继续:

我们可以看到 ESP 的值保存在 EAX 中了。

Registers (FPU)	
EAX	0012FFAC
ECX	00000012
EDX	00172CF0
EBX	7FFDE000
ESP	0012FFAC
EBP	000002C0
ESI	004865F8 UnPack
EDI	00000206
EIP	0049800B UnPack
C 0	ES 0023 32bit
9BE78	UnPackMe.0049BE78
00000	

Address	Hex dump	ASCII
0047B398	AC FF 12 00 00 00 00 00	% +.....
0047B3A0	00 00 00 00 00 00 00 00
0047B3A8	00 00 00 00 00 00 00 00
0047B3B0	00 00 00 00 00 00 00 00
0047B3B8	00 00 00 00 00 00 00 00
0047B3C0	00 00 00 00 00 00 00 00
0047B3C8	00 00 00 00 00 00 00 00
0047B3D0	00 00 00 00 00 00 00 00
0047B3D8	B8 FF 12 00 00 00 00 00	@ +.....
0047B3E0	00 00 00 00 00 00 00 00
0047B3E8	00 00 00 00 00 00 00 00
0047B3F0	00 00 00 00 00 00 00 00
0047B3F8	00 00 00 00 00 00 00 00
0047B400	00 00 00 00 00 00 00 00
0047B408	00 00 00 00 00 00 00 00
0047B410	00 00 00 00 00 00 00 00
0047B418	B4 FF 12 00 00 00 00 00	+ +.....
0047B420	00 00 00 00 00 00 00 00
0047B428	00 00 00 00 00 00 00 00
0047B430	00 00 00 00 00 00 00 00
0047B438	00 00 00 00 00 00 00 00

从上图中我们可以看到 ESP 的值在变化,但是变化的很小。之前是 12FFB8,现在变为了 12FFAC。

00496805	8385 CCB04700	ADD DWORD PTR SS:[EBP+47B0CC], 4
0049680C	8B85 CCB04700	MOV EAX, DWORD PTR SS:[EBP+47B0CC]
00496812	8BE0	MOV ESP, EAX
00496814	68 B47F121F	PUSH 1F127FB4
00496819	5A	POP EDX
0049681A	81F2 A277F226	XOR EDX, 26F277A2
00496820	E9 8001FFFF	JMP 004869A5
00496825	81F2 FE6176AD	XOR EDX, AD7661FE
00496828	81F2 00410000	XOR EDX, 00004100
SS:[0047B398]=0012FFAC		
Address	Hex dump	ASCII
0047B398	AC FF 12 00 00 00 00 00	% +.....
0047B3A0	00 00 00 00 00 00 00 00
0047B3B0	00 00 00 00 00 00 00 00

这里 ESP 将加 4。

然后会再减去 4。

00471718 83AD CCB04700 0> SUB DWORD PTR SS:[EBP+47B0CC], 4

这里保存的 ESI 的值不正确,然后是 EDI,接着又会恢复。

00470F82 89B5 C0A84700 MOV DWORD PTR SS:[EBP+47A8C0], ESI

ESI=CB4A9B05

好,这里

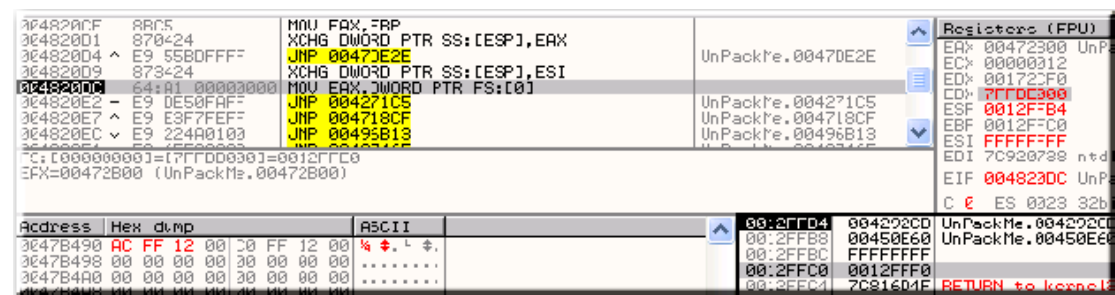
00498D0B 8985 CCB04700 MOV DWORD PTR SS:[EBP+47B0CC], EAX

EAX 实际上就是 ESP 的值为 12FFB0。

ESI 的值继续变化,而且都是不正确的值,很显然是被混淆过的。

00470F82 89B5 C0A84700 MOV DWORD PTR SS:[EBP+47A8C0], ESI

ESI=F2AFFFC

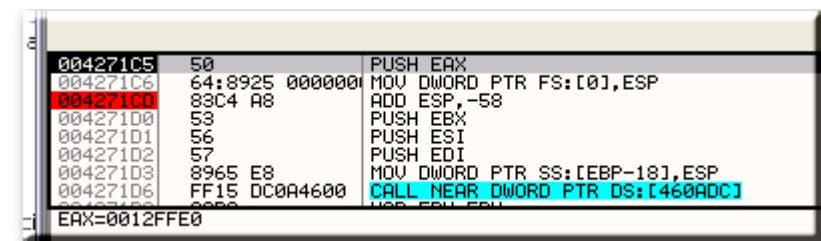


这里我们可以看到不仅仅使用 XCHG 指令模拟 PUSH 了,下面还夹杂着原始指令

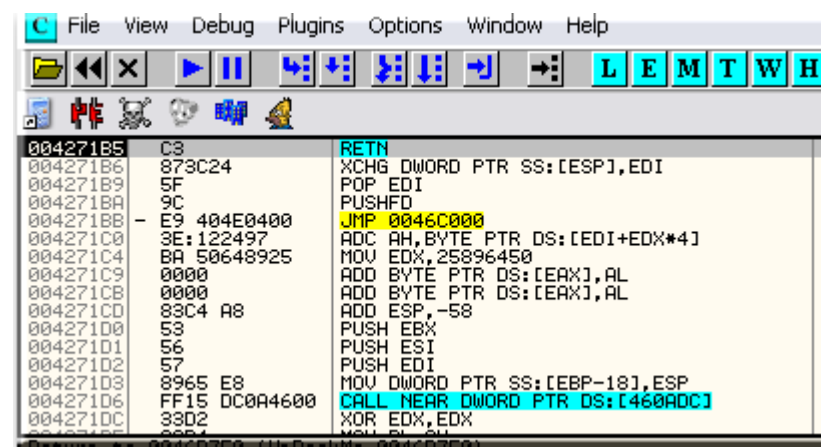
004820D9 873424 XCHG DWORD PTR SS:[ESP], ESI

004820DC 64:A1 00000000 MOV EAX, DWORD PTR FS:[0]

我们继续跟踪,会发现来到了代码段。



我们继续



我们可以看到已经到了 OEP 附近了。

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]	UnPackMe.004862
004271DC	33D2	XOR EDX,EDX	
004271DE	8AD4	MOV DL,AH	
004271E0	8915 34E64500	MOV DWORD PTR DS:[45E634],EDX	
004271E6	8BC8	MOV ECX,EAX	

下面我们来尝试设置条件断点,观察各个寄存器的值是如何保存的。

00470F82 89B5 C0A84700 MOV DWORD PTR SS:[EBP+47A8C0],ESI

这里保存的是 ESI 的值

00470F82	89B5 C0A84700	MOV DWORD PTR SS:[EBP+47A8C0],ESI	
00470F88	871C24	XCHG DWORD PTR SS:[ESP],EBX	
00470F8B	8BF3	MOV ESI,EBX	
00470F8D	81F3 470C06B7	XOR EBX,B7060C47	
00470F93	E9 BB020200	JMP 00491253	UnPackMe.00491253
00470F98	C3	RETN	
00470F99	53	PUSH EBX	
00470F9A	68 2692A7C7	PUSH C7A79226	
00470F9F	5B	POP EBX	
00470FA0	81EB 0F52E4EB	SUB EBX,FBE4520F	

接着的条件断点。

Set conditional log breakpoint at UnPackMe.00470F82

Condition:

Explanation:

Expression:

ESI = ESI

Decode value of expression as:

Assumed by expression

	Never	On condition	Always	Pass count (dec.)
Pause program:	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	0.
Log value of expression:	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	
Log function arguments:	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	

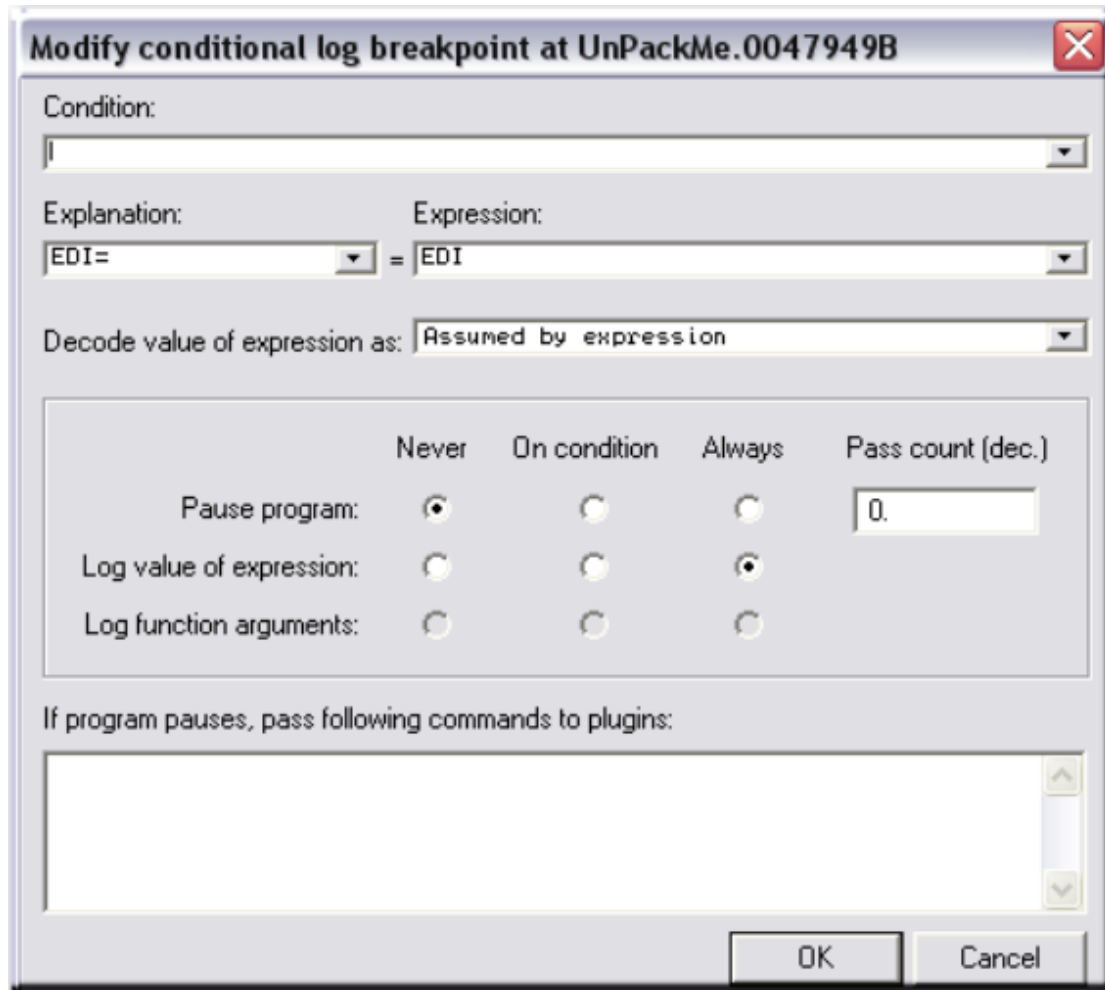
If program pauses, pass following commands to plugins:

OK Cancel

接下来是看看哪里保存 EDI 的值。

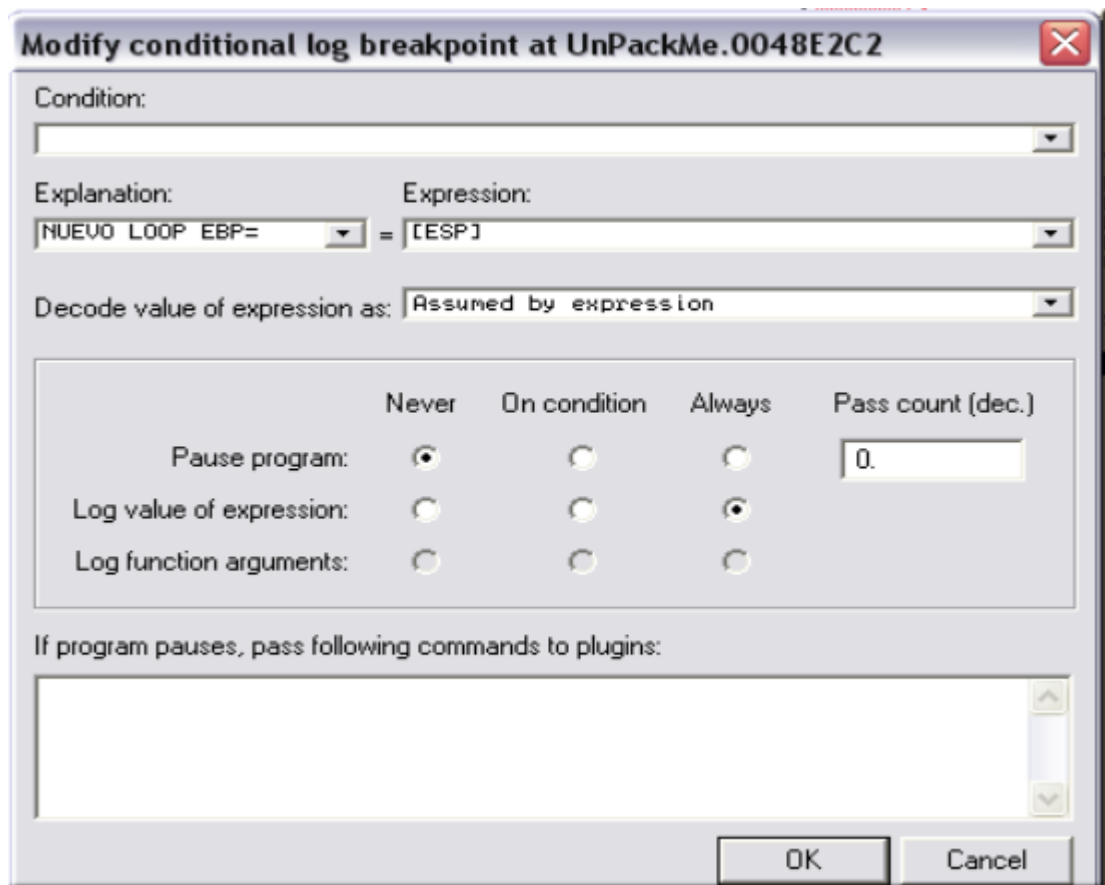
0047949B 89BD C0A44700 MOV DWORD PTR SS: [EBP+47A4C0], EDI

照例



接着是 EBP

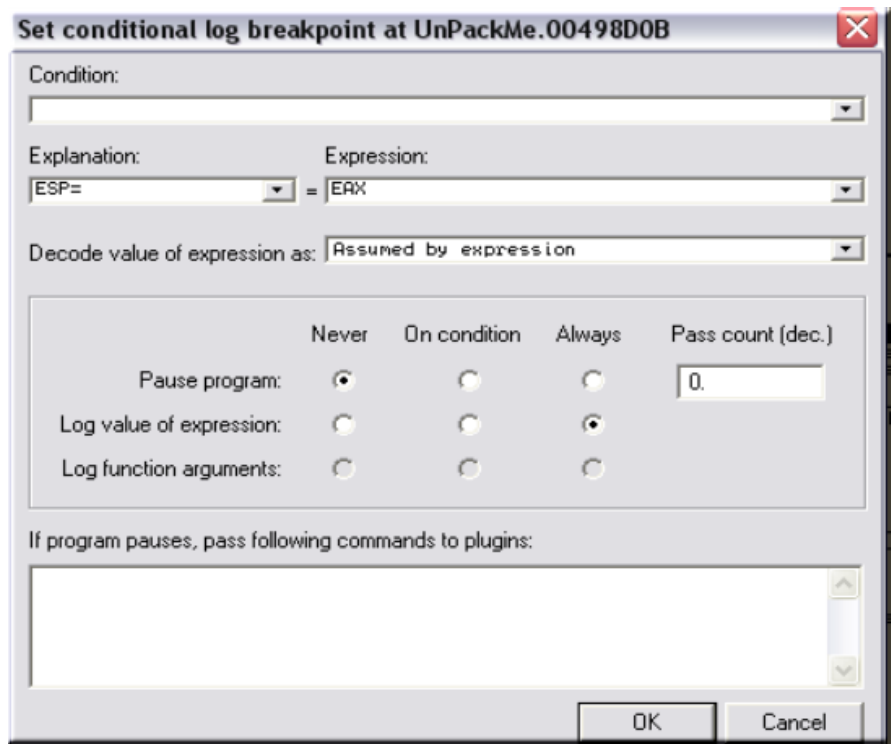
0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]



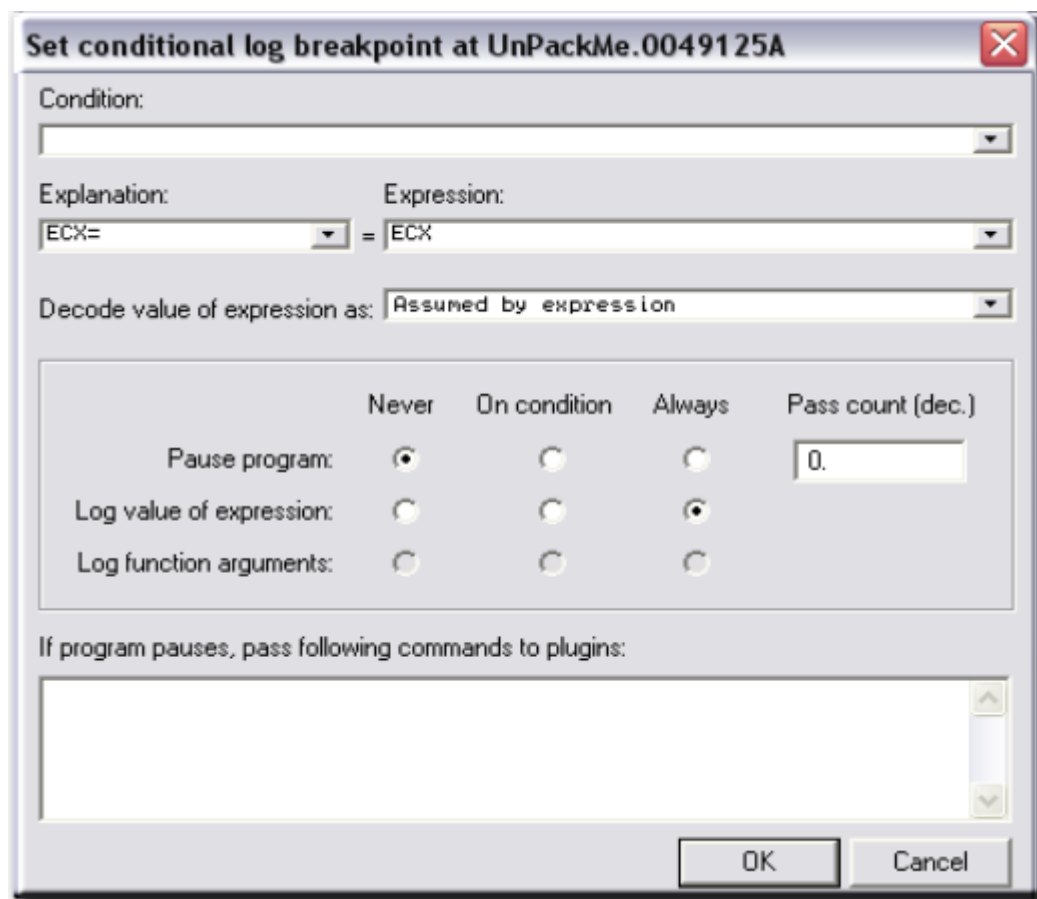
接下来是 ESP

00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

继续调整条件断点



0049125A 这里更新 ECX。

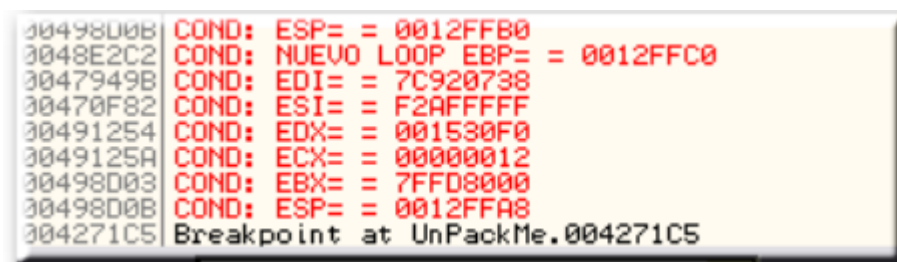


这里虽然我们不关心 EDX 的值,还是给它设置条件断点。

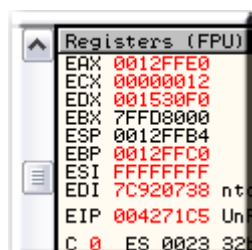
00491254 8995 ECBC4700 MOV DWORD PTR SS: [EBP+47BCEC], EDX

接着 EBX

00498D03 899D 8C114800 MOV DWORD PTR SS:[EBP+48118C],EBX



我们可以看到



这里我们可以看到 EBP,ECX,EDX,EBX,EDI 这些值是相同的,ESP,EAX,ESI 的值有些差别。

好,那么我们将条件断点的条件换一换,换成 ESI==FFFFFFFF 试试看。

Condition to pause run trace

Pause run trace when any checked condition is met:

☐ EIP is in range 00401000 ... 0044AFFE

☐ EIP is outside the range 00000000 ... 00000000

☒ Condition is TRUE ESI==0FFFFFFFF

☐ Command is suspicious or possibly invalid

☐ Command count is 0. (actual 1519)

☐ Command is one of

In command, R8, R32, RA, RB and CONST match any register or constant

到了这里

00476E19	871C24	XCHG DWORD PTR SS:[ESP],EBX	
00476E1C	8BF3	MOV ESI,EBX	
00476E1E	5B	POP EBX	7F
00476E1F	E9 9FCB0000	JMP 004839C3	Un
00476E24	B8 F1203047	MOV EBX,473020F1	

此时 ESI 的值为 FFFFFFFF。

Set conditional log breakpoint at UnPackMe.00476E1E

Condition:

Explanation: ESI= Expression: = ESI

Decode value of expression as: Assumed by expression

	Never	On condition	Always	Pass count (dec.)
Pause program:	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	0.
Log value of expression:	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	
Log function arguments:	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	

If program pauses, pass following commands to plugins:

Address	Module	Active	Disassembly
004271C5	UnPackMe	Always	PUSH EAX
00470F82	UnPackMe	Disabled	MOV DWORD PTR SS:[EBP+47A8C0],ESI
00476E1E	UnPackMe	Log "ESI="	POP EBX
0047949B	UnPackMe	Log "EDI="	MOV DWORD PTR SS:[EBP+47A4C0],EDI
0048E2C2	UnPackMe	Log "NUEVO LOOP EBP="	POP DWORD PTR SS:[EBP+47A0C0]
00491254	UnPackMe	Log "EDX="	MOV DWORD PTR SS:[EBP+47BCEC],EDX
0049125A	UnPackMe	Log "ECX="	MOV DWORD PTR SS:[EBP+47B8E0],ECX
00498D03	UnPackMe	Log "EBX="	MOV DWORD PTR SS:[EBP+48118C],EBX
00498D0B	UnPackMe	Log "ESP="	MOV DWORD PTR SS:[EBP+47B0CC],EAX

将针对 ESI 的断点禁用掉。

```

00401000 Sent virtual address to ollybone module for NX r
0048E2C2 COND: NUEVO LOOP EBP= 0012FFF0
0047949B COND: EDI= 7C920738
00491254 COND: EDX= 0047F3EF
0049125A COND: ECX= 0047F3EF
00498D03 COND: EBX= 7FFDF000
00498D0B COND: ESP= 0012FFC0
00476E1E COND: ESI= FFFFFFFF
0048E2C2 COND: NUEVO LOOP EBP= 0012FFF0
0047949B COND: EDI= 7C920738
00491254 COND: EDX= 0047F3EF
0049125A COND: ECX= 0047F3EF
00498D03 COND: EBX= 7FFDF000
00498D0B COND: ESP= 0012FFC4
0048E2C2 COND: NUEVO LOOP EBP= 0012FFF0
0047949B COND: EDI= 7C920738
00491254 COND: EDX= 0047F3EF
0049125A COND: ECX= 0047F3EF
00498D03 COND: EBX= 00478304
00498D0B COND: ESP= 0012FFBC
0048E2C2 COND: NUEVO LOOP EBP= 0012FFF0

```

我们继续跟踪到了 47CE1E 这里。

Address	Module	Active	Disassembly
0049188F	9C		PUSHFD
00491890	56		PUSH ESI
00491891	8BF0		MOV ESI,EAX
00491893	873424		XCHG DWORD PTR SS:[ESP],ESI
00491896	^ E9 9D56FFFF		JMP 00486F38
0049189B	^ 0F80 CE160000		JGE 00492F6F
004918A1	^ E9 0EDAFFFF		JMP 0048F2B4
00491896	^ E9 9D56FFFF		JMP 00486F38

恢复针对 ESI 的断点,条件依然设置为 ESI == FFFFFFFF。

Address	Module	Active	Disassembly
0049188F	9C		PUSHFD
00491890	56		PUSH ESI
00491891	8BF0		MOV ESI,EAX
00491893	873424		XCHG DWORD PTR SS:[ESP],ESI
00491896	^ E9 9D56FFFF		JMP 00486F38
0049189B	^ 0F80 CE160000		JGE 00492F6F
004918A1	^ E9 0EDAFFFF		JMP 0048F2B4

这里已经获取到了 ESI 正确的值。

```

00498D08 COND: ESP= = 0012FFB4
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB8
00491896 COND: ESI= = FFFFFFFF
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB8
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB4
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = D1B1CF8F
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB8
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00450E60
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB8
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFAC
00491896 COND: ESI= = FFFFFFFF
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB0
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB0
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0
0049125A COND: ECX= = 00000012
00498D03 COND: EBX= = 7FFD6000
00498D08 COND: ESP= = 0012FFB0
0048E2C2 COND: NUEVO LOOP EBP= = 0012FFC0
00479498 COND: EDI= = 7C920738
00491254 COND: EDX= = 001530F0

```

好了,本章结束。

这个系列也结束了。