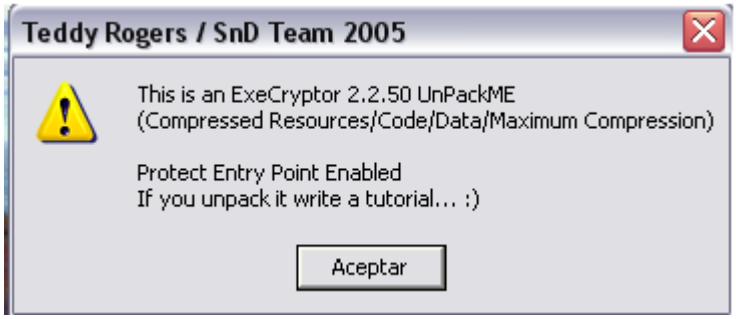
第五十八章-ExeCryptor v2.2.50.h

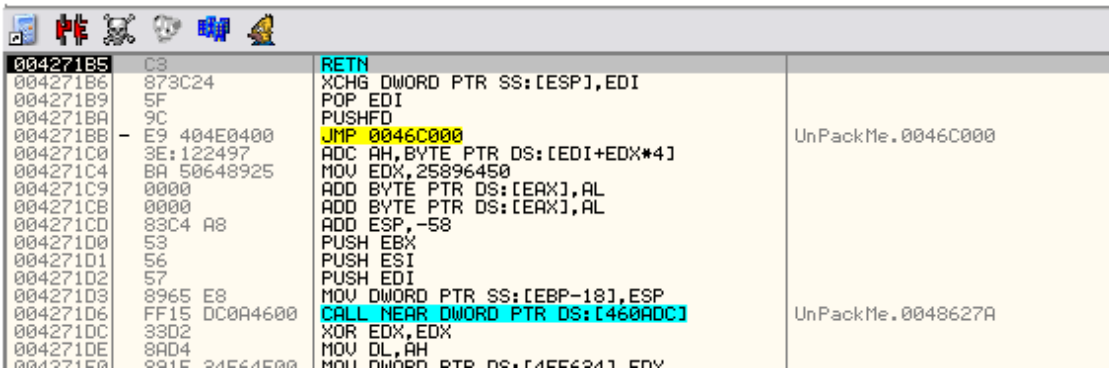
好,本章还剩下 ExeCryptor 的最后三个 UnPackMe,这三个 UnPackMe 是本系列教程中最难脱的。我们来尝试对它们进行脱壳,如果没有脱壳成功的话,至少也会记录下脱壳思路,供大家学习。

我们双击运行 UnPackMe H,可以看到它新增了如下保护:

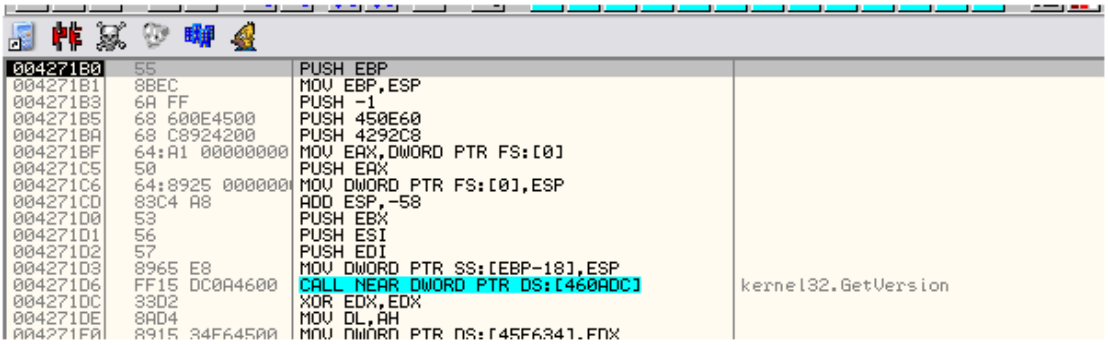


我们可以看到入口点保护开启了。也就是说,入口点可能被隐藏起来了。

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。

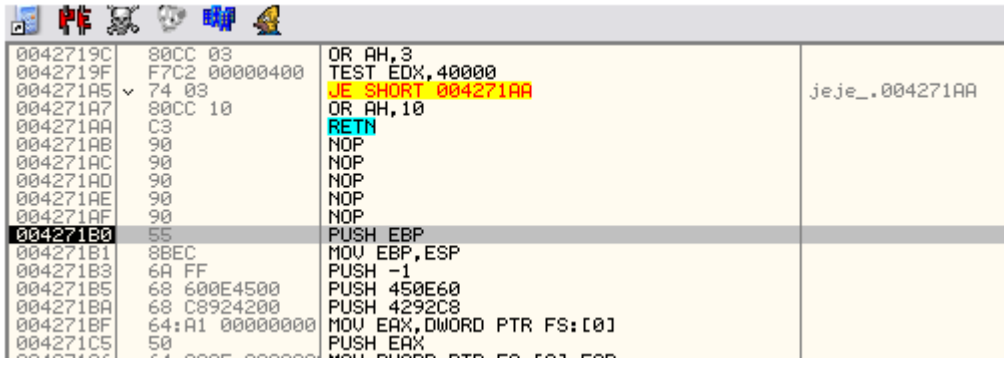


貌似有点不对劲,我们看看之前脱壳过的版本。

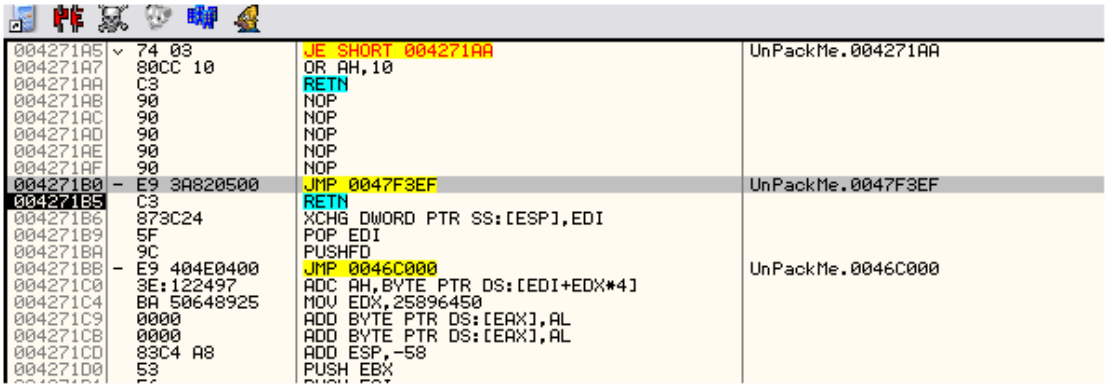


我们可以看到入口点处的代码明显被隐藏了。

这是之前脱壳版本的截图:

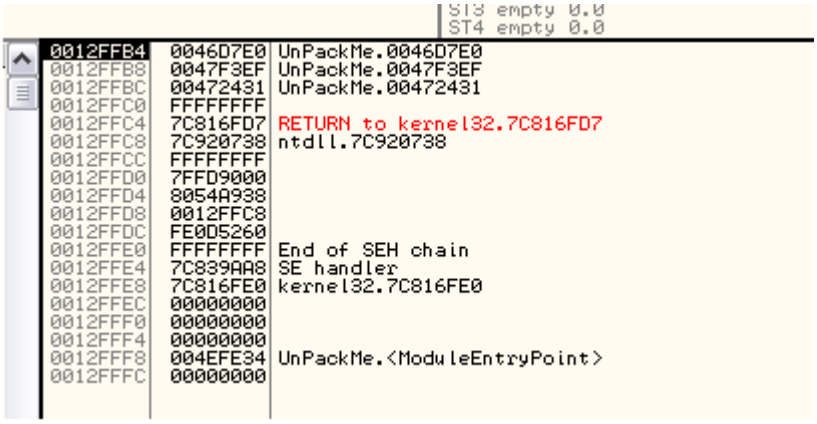


这是现在 UnPackMe H 的截图:



对比着看,明显有差别。

我们来看看 UnPackMe H 此时堆栈的情况:



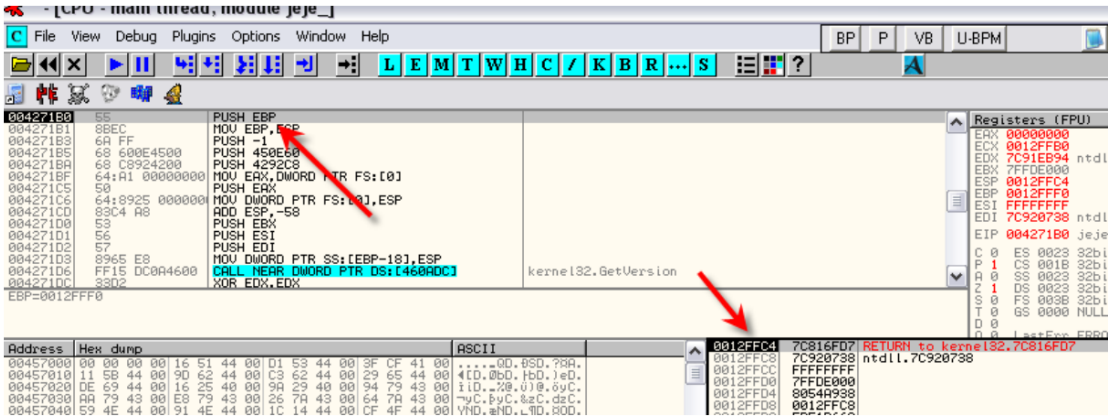
我们再来看看之前脱过壳的版本:

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,譬如说: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。

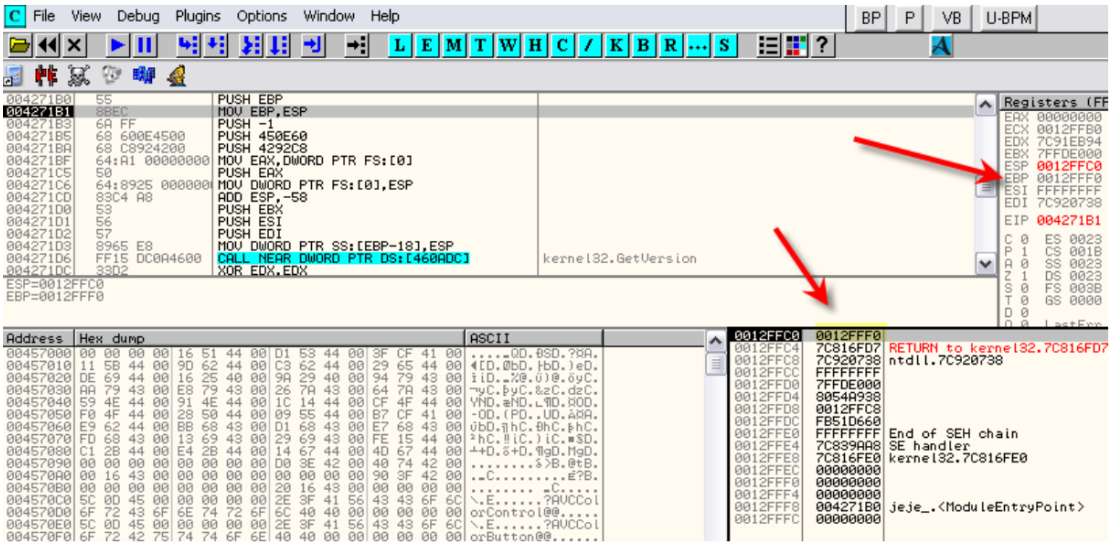
就我们当前这个例子来说,入口点处时栈顶指针 ESP 指向的地址是 12FFC4。大家在平时脱壳的时候也要多多留意入口点处的栈顶指针指向了哪里。

成功脱壳后,OEP 处的第一条指令应该是 PUSH EBP。

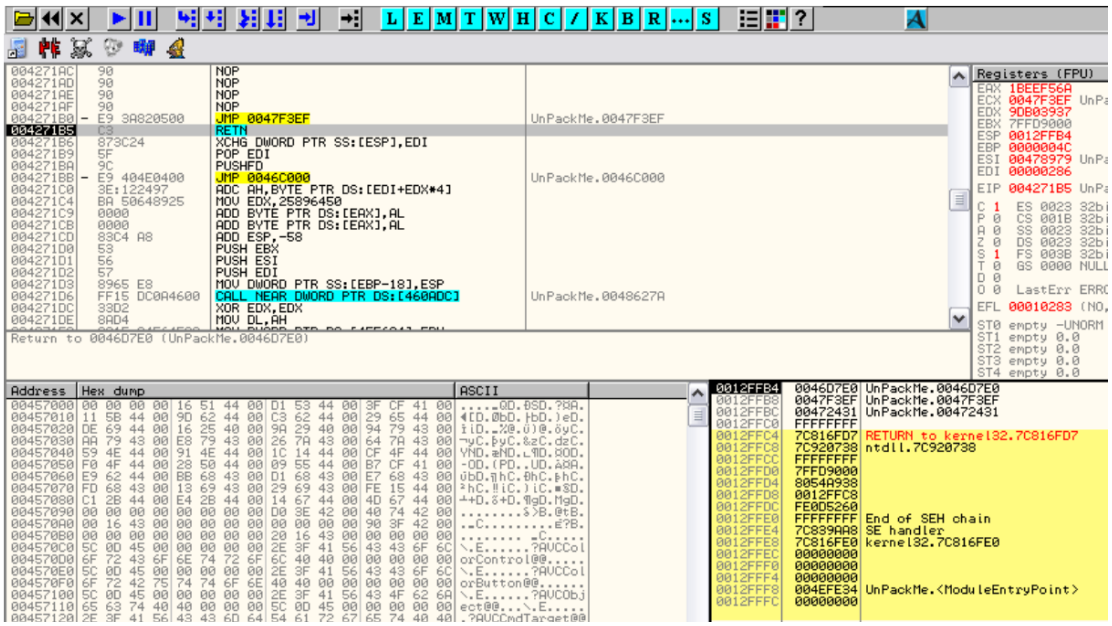
下面我们来执行 PUSH EBP 这条指令。



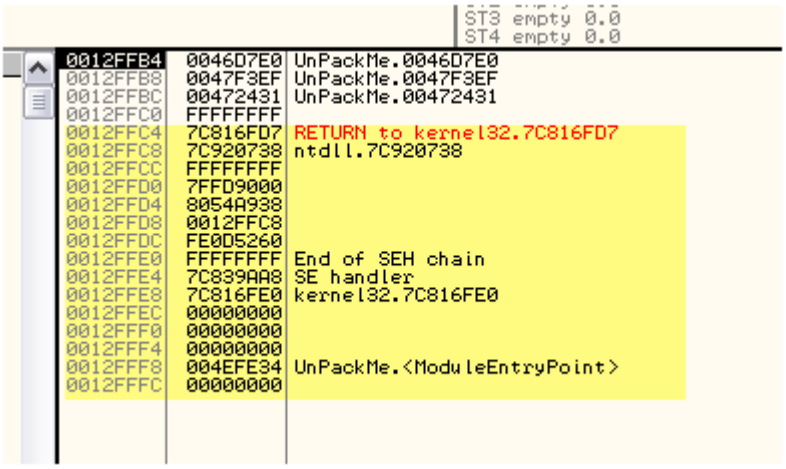
我们可以看到 EBP 的值被压入到堆栈中了:



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。



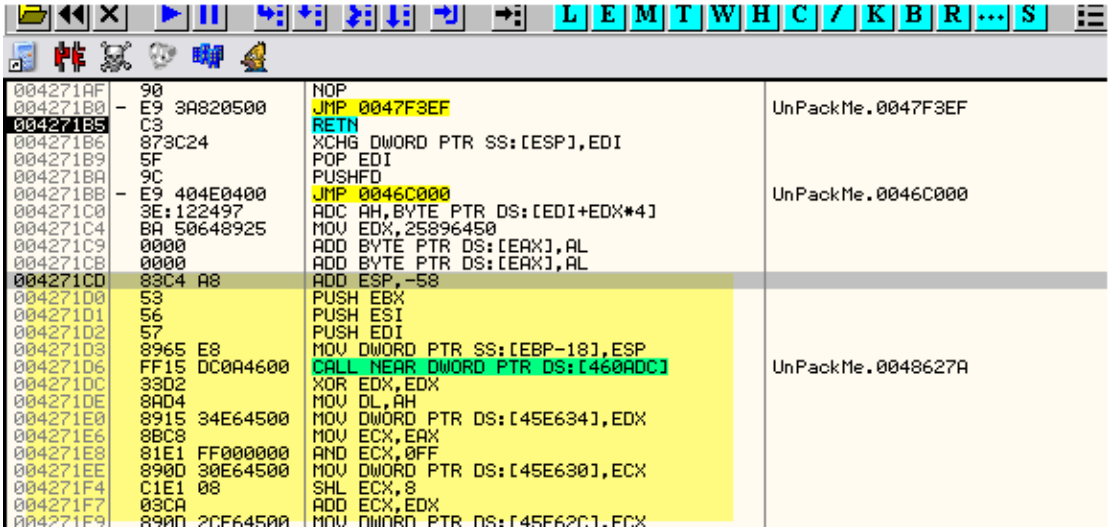
我们继续观察堆栈:



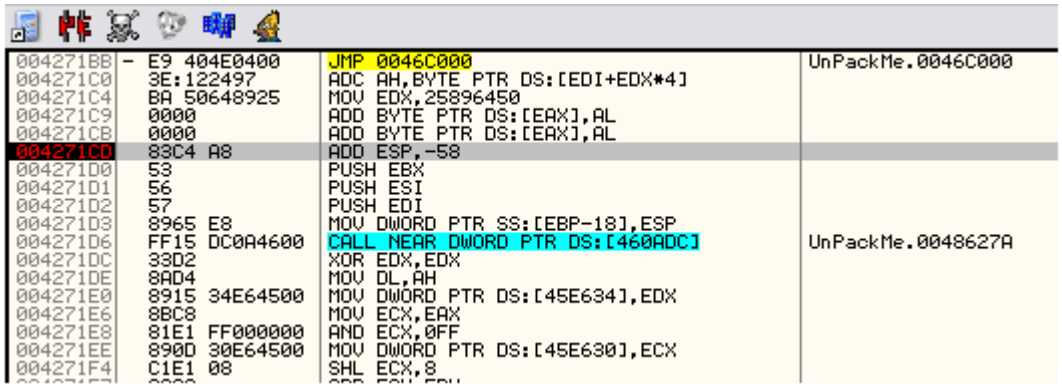
正常来说,到达 OEP 处时,栈顶指针指向 12FFC4 才对,这里我们姑且算它执行了 PUSH EBP,那么栈顶指针也应该指向的是 12FFC0 才对。

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。

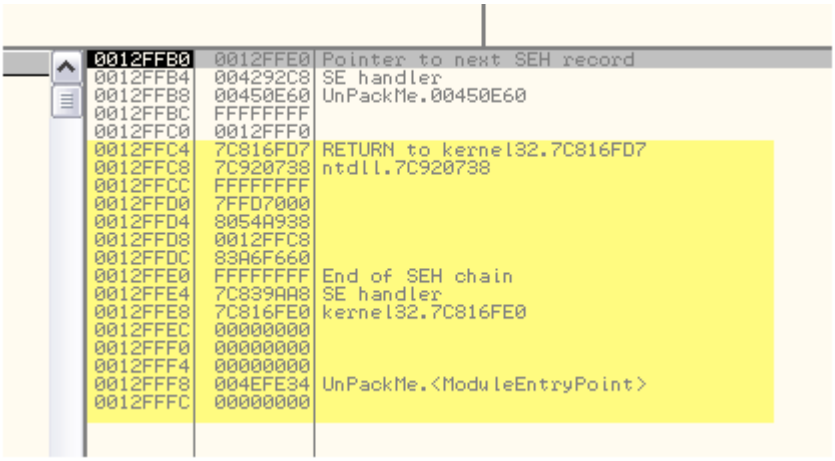
这里我们对黄色标注出来的区域设置硬件执行断点或者 BP 断点。



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:

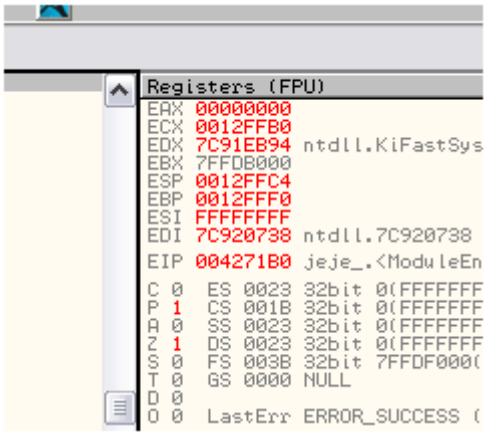


断在了这里,我们来看下堆栈:



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。



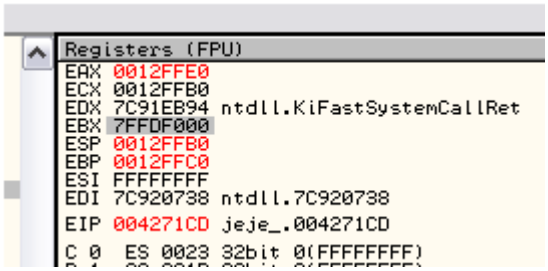
我们会发现在剥离了 TLS 以后,每次断在 OEP 处时,唯一会变化的就是 EBX 寄存器的值,但是无论怎么变,总是在 7FFDB000~7FFDF000 这个范围之内,反观其他寄存器的值都是固定不变的。

好,我们重启 OD,再次断到 ADD EBP,-58 指令处。

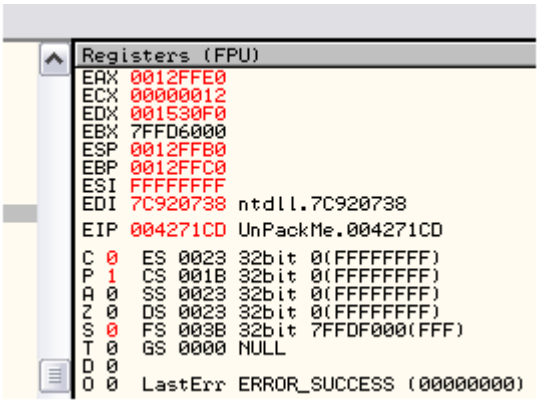
004271CD 83C4 A8 ADD ESP,-58

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:

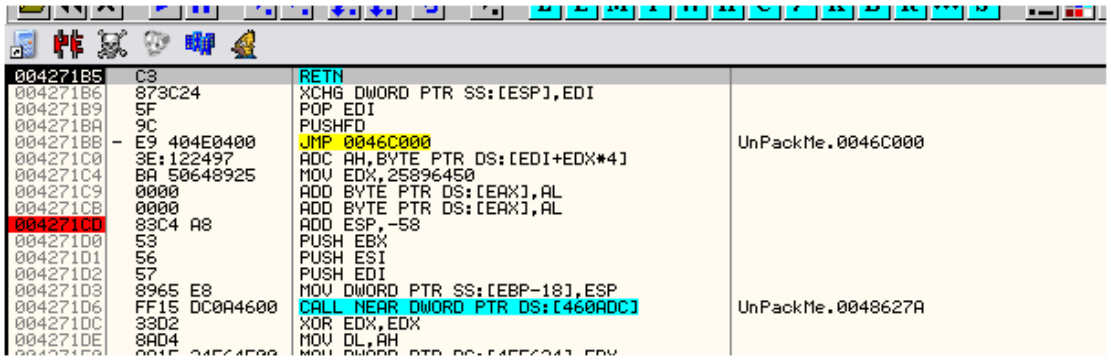


未脱壳:

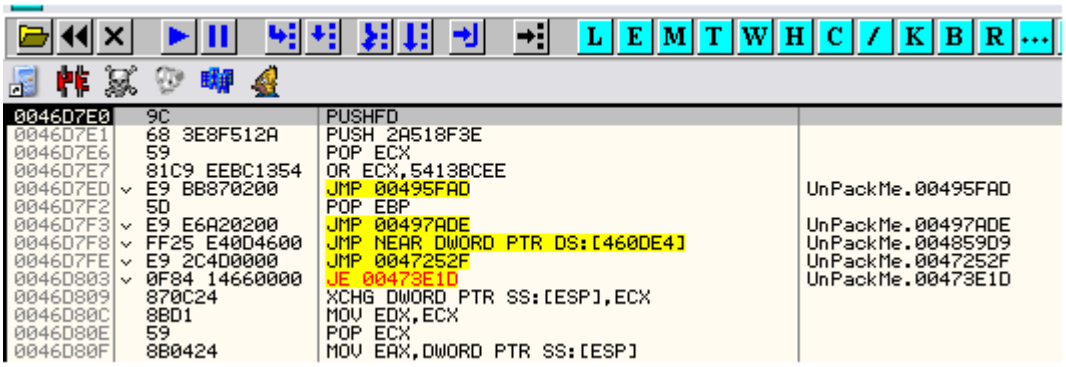


我们可以看到在 UnPackMe 模拟执行指令的过程中连同堆栈以及除 ECX,EDX 以外的其他通用寄存器也都模拟了。也就是当已脱壳和未脱壳的版本处于同一个位置时,我们要留意 EAX,EBX,ESP,EBP,ESI,EDI 这几个寄存器的情况,ECX,EDX 的话,我们并不关心。

好,现在我们回到该 UnPackMe 模拟执行指令的起始地址处。



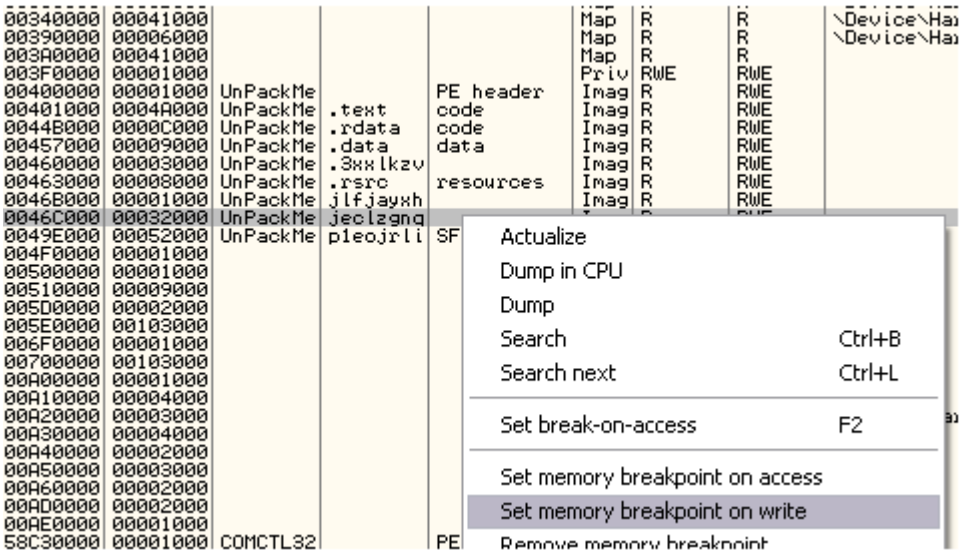
我们可以看到在 4271B5~4271CD 之间掺杂了大量的垃圾指令-起到混淆的作用。实际上它们是要完成 5 条指令的功能。



我们按 F7 单步执行 RET 指令,到了这里。

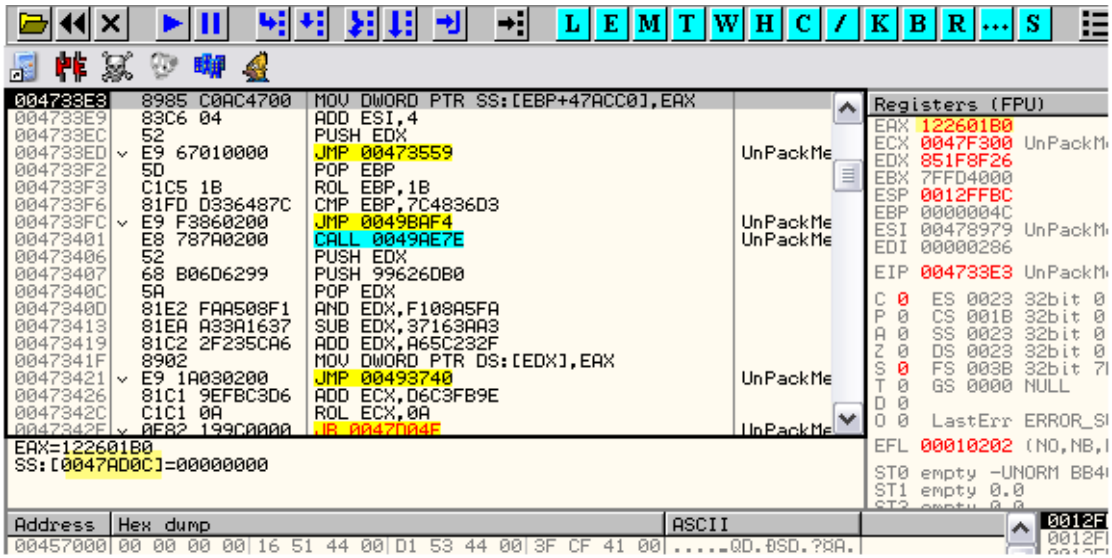
首先是利用 PUSHFD 保存 EFLAGS(标志寄存器)的值。

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。



设置完内存写入断点以后,我们运行起来,看看会发生什么。

断在了这里。



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 = 122601B0 保存到 47AD0C 中

EAX = 5A731601 保存到 4815E0 中

EBX = 5A731601 保存到 4815E0 中

EAX = 0046C5DE 保存到 47A90C 中

00496CBB 01 变成了 00

00496C9C 01 变成了 02

00496CAB 00 变成了 01

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。

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

这里是利用 POP 指令将 12FFF0 保存到 47A0CC 中

EDI = 7C920738 保存到 47A4CC 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。

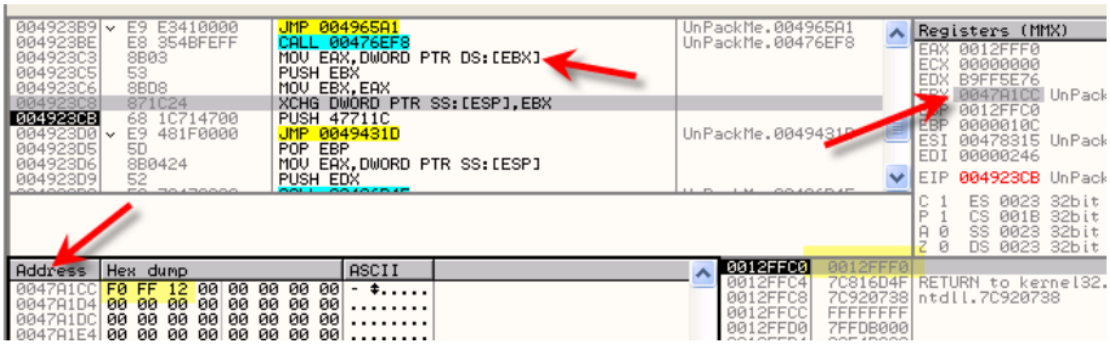
EBX = 7FFDB000 保存到 00481298 中

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



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。

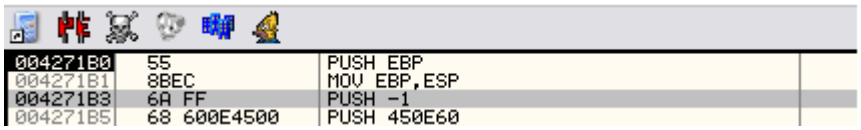
0047711C 83AD CCB04700 0> SUB DWORD PTR SS: [EBP+47B0CC], 4

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。



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

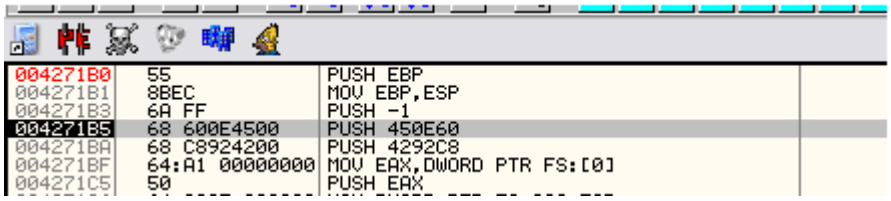
明显是垃圾指令,我们不理会它。

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。



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

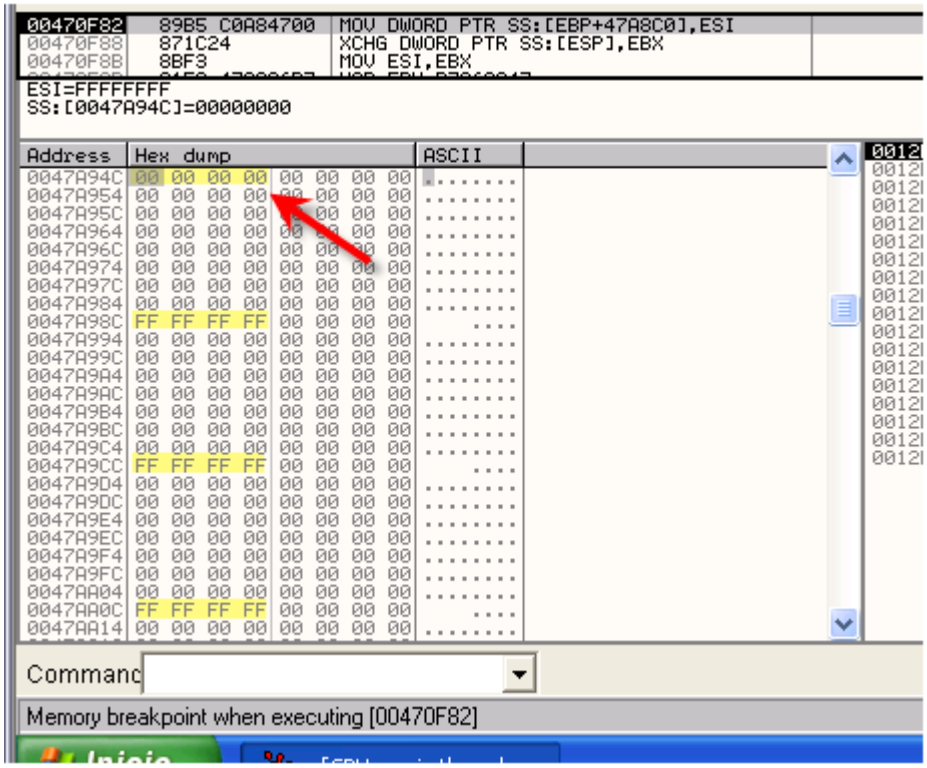
每次循环都会保存 EDI 的值,现在也会保存 ESI 的值。

ESI=FFFFFFFF 保存到 0047A94C 中

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

这里面有是大量的跳转,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。

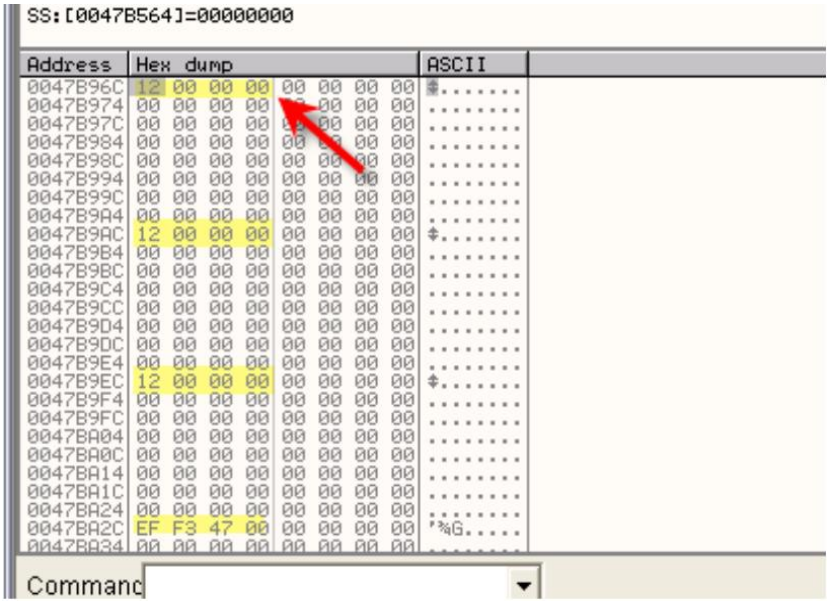


这里又是将 ESI 的值保存到空闲区域,存放的从上图来看非常的对称。

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。



我晕,已经写了这么多页了。

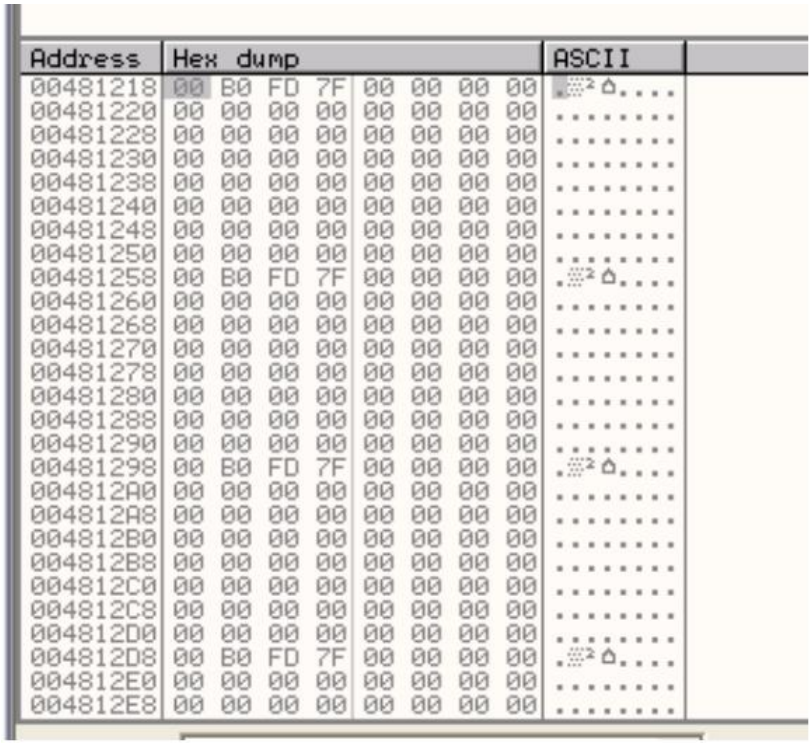
EAX=00472B00 保存到 0047B564 中

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

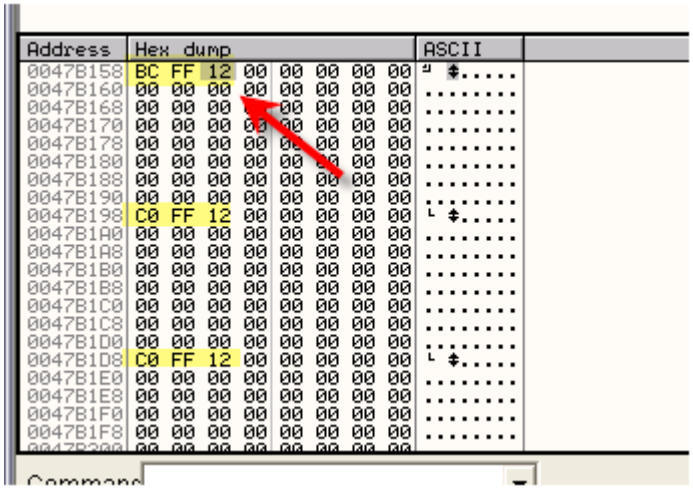


EAX=12FFBC 保存到 0047B158 中

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。



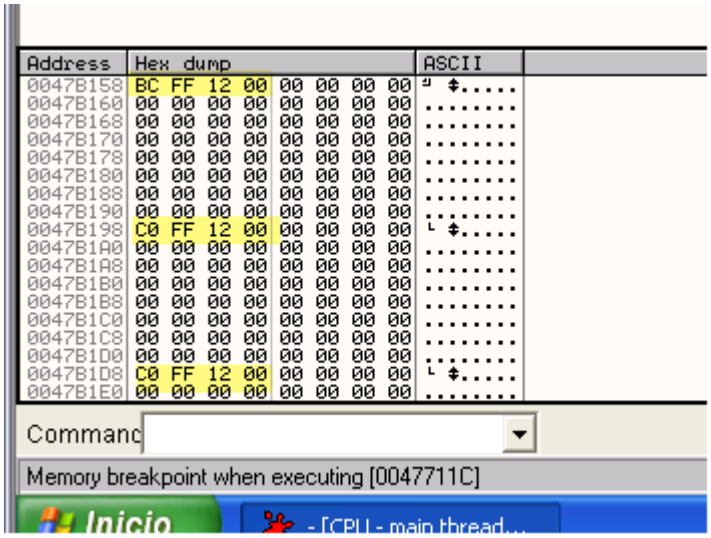
这里我们可以看到当前的值 12FFBC 以及之前的值 12FFC0。

我们继续,又是一轮循环。

EAX=1EF00200 保存到 0047AD4C 中

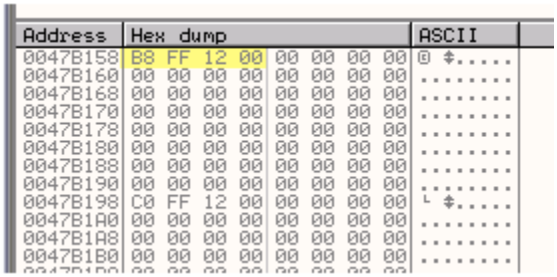
现在我们又看到了 SUB 指令了,减去 4,接下来应该是保存 ESP 的值,然后 PUSH,嘿嘿。

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



这里减去 4。

变成了



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

貌似又一轮循环开始了

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

又一轮循环

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

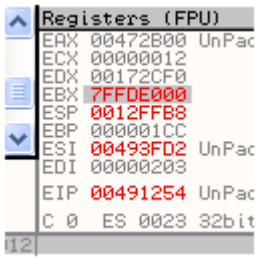
这里将 EDI,ESI 恢复为正确的值。

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



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

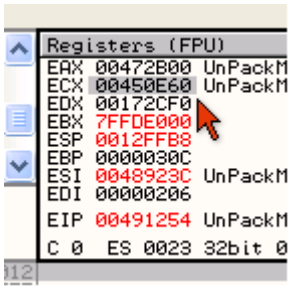
0047B418 B4 FF 12 00

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

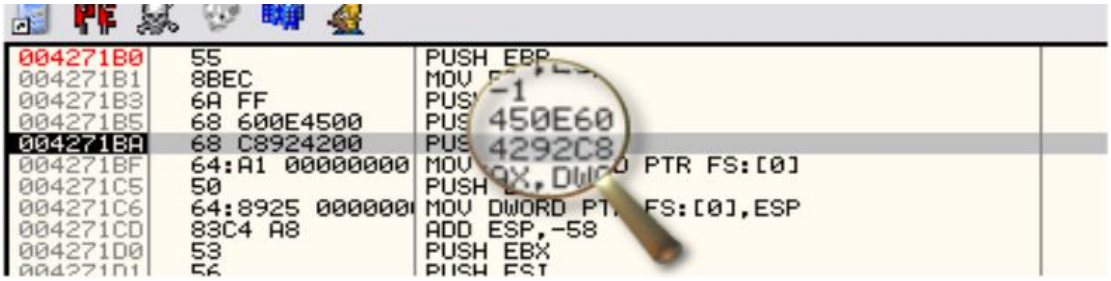


保存到了这里

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

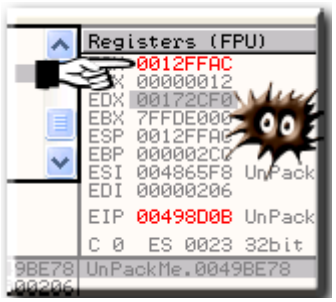
通过 XCHG 指令实现了 PUSH 450E60 的效果,在大量的垃圾中定位到它真心困难。

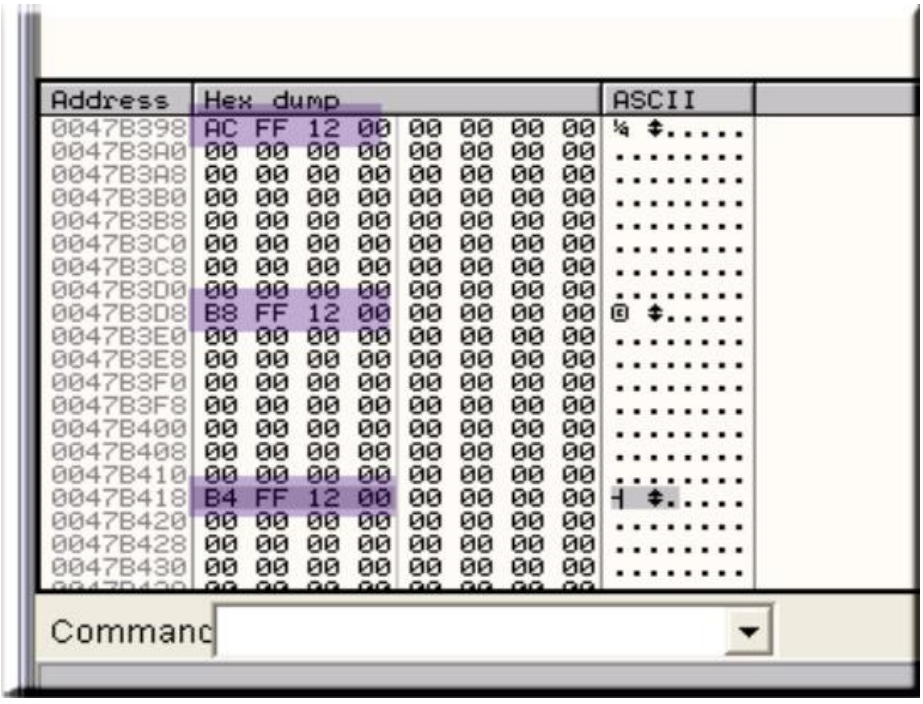
我们继续,关注重点的地方,其他的略过就行了。



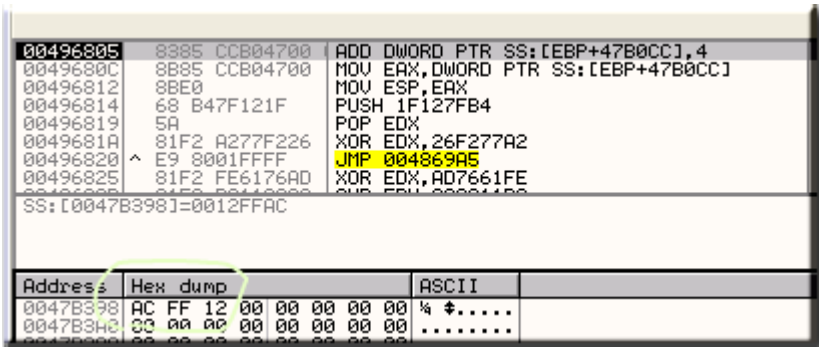
接下来的原始指令应该是 PUSH 4292C8,我们继续:

我们可以看到 ESP 的值保存在 EAX 中了。





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。



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

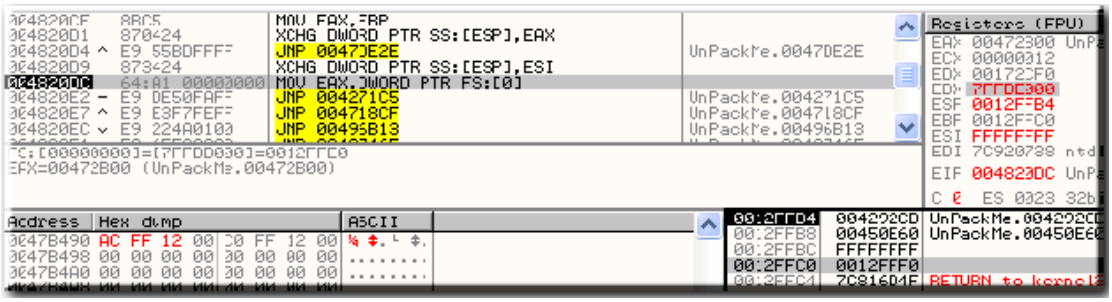
00498D0B 8985 CCB04700 MOV DWORD PTR SS: [EBP+47B0CC], EAX

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=F2AFFFFC

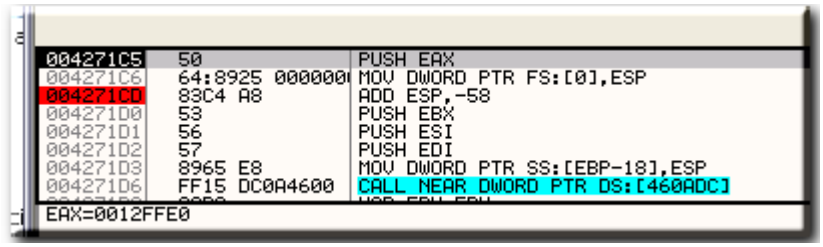


这里我们可以看到不仅仅使用 XCHG 指令模拟 PUSH 了,下面还夹杂着原始指令

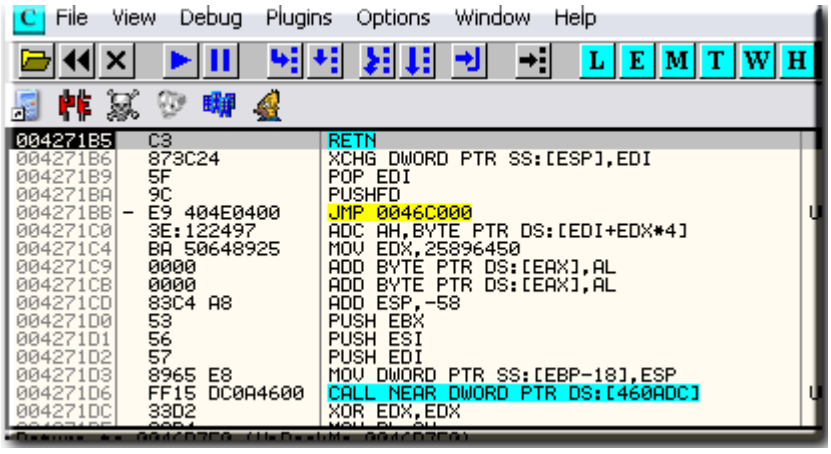
004820D9 873424 XCHG DWORD PTR SS: [ESP], ESI

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

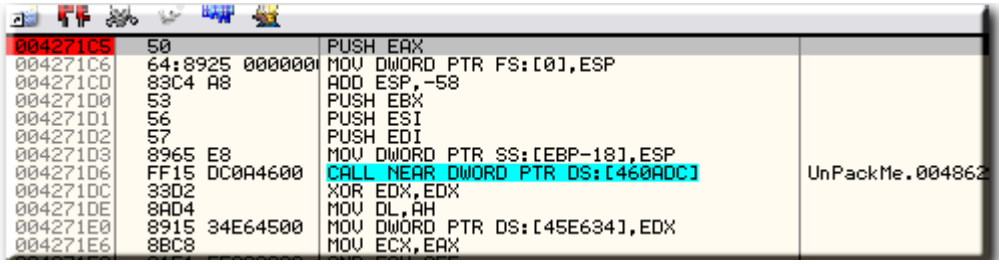
我们继续跟踪,会发现来到了代码段。



我们继续



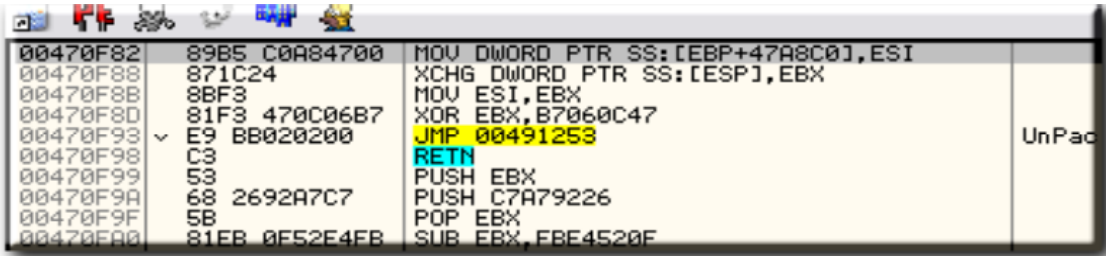
我们可以看到已经到了 OEP 附近了。



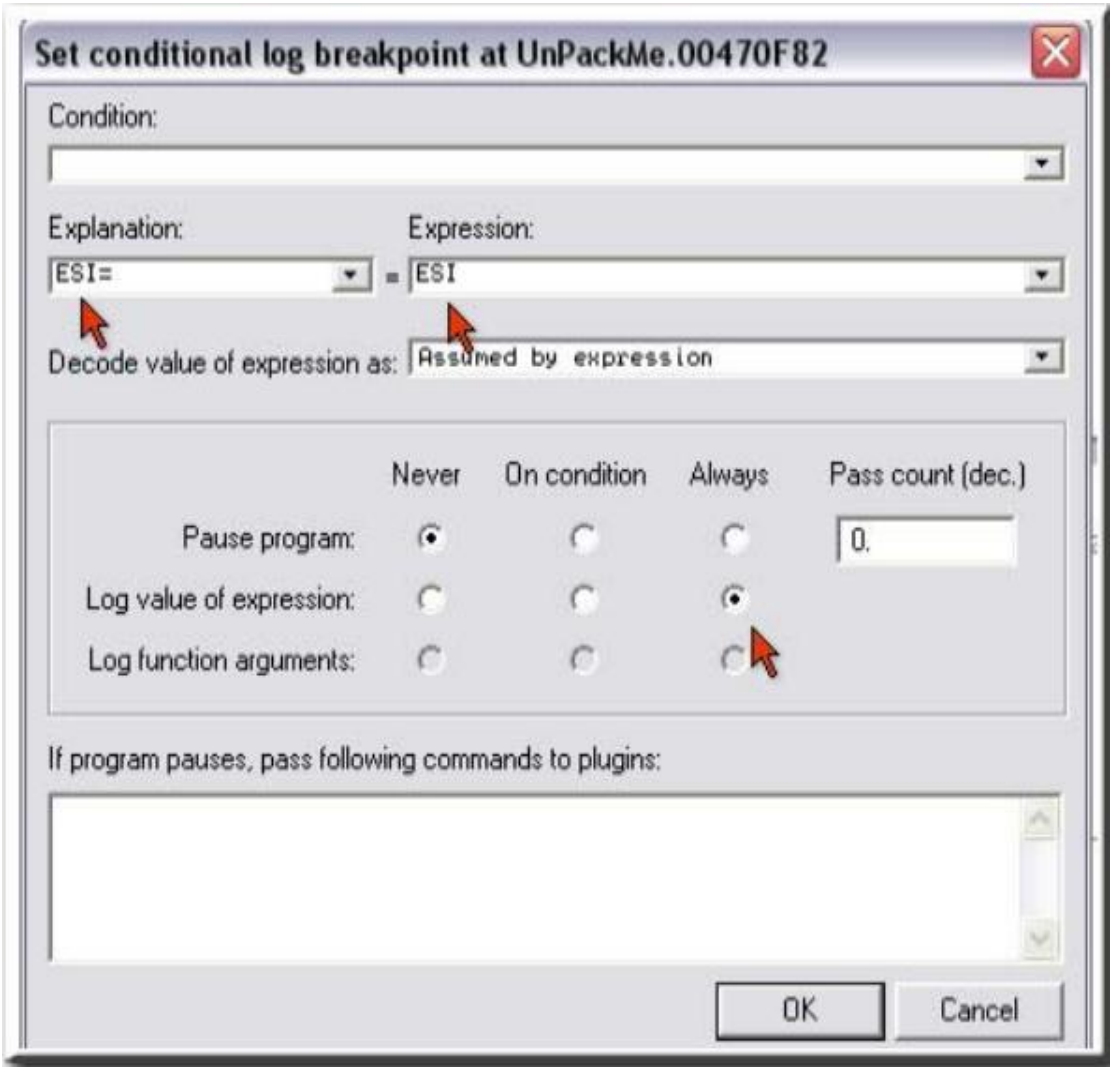
下面我们来尝试设置条件断点,观察各个寄存器的值是如何保存的。

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

这里保存的是 ESI 的值



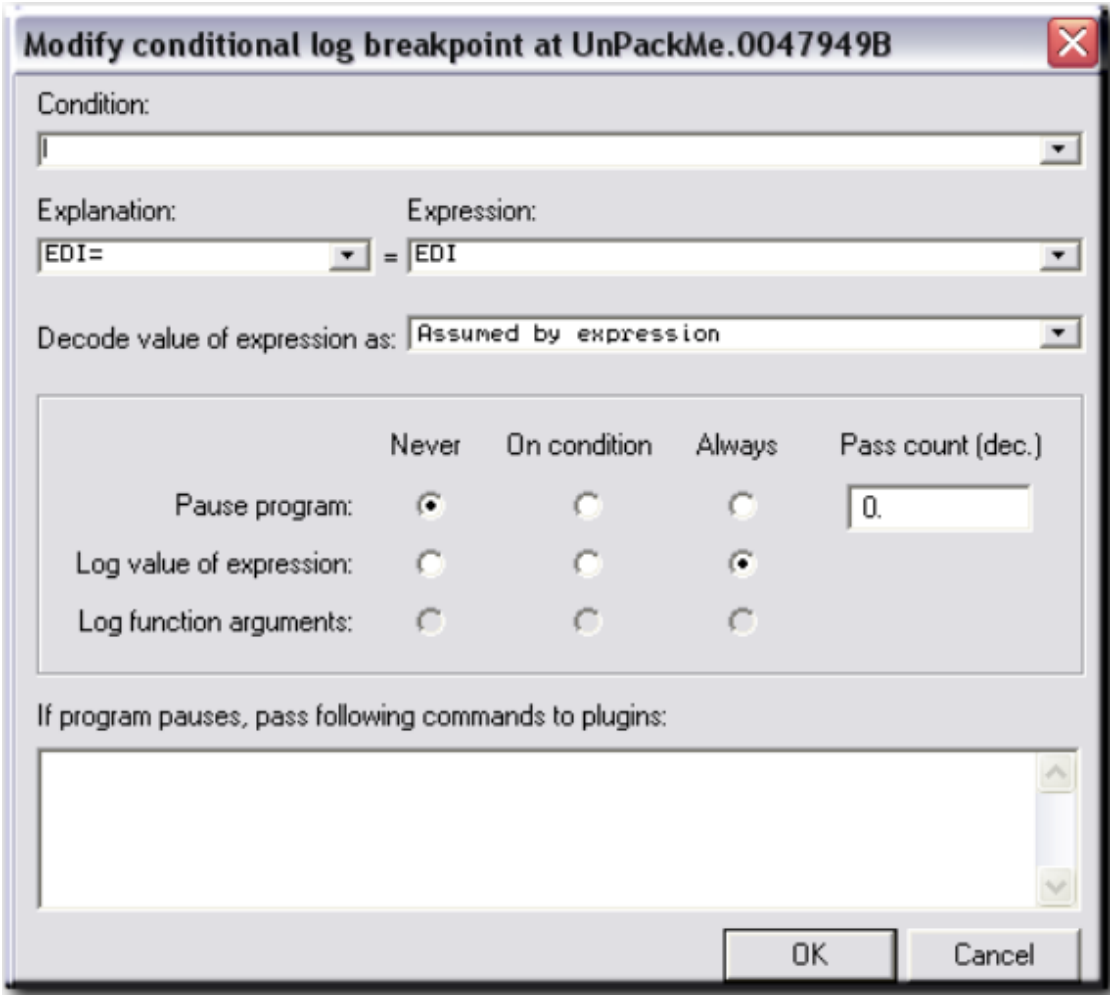
接着的条件断点。



接下来是看看哪里保存 EDI 的值。

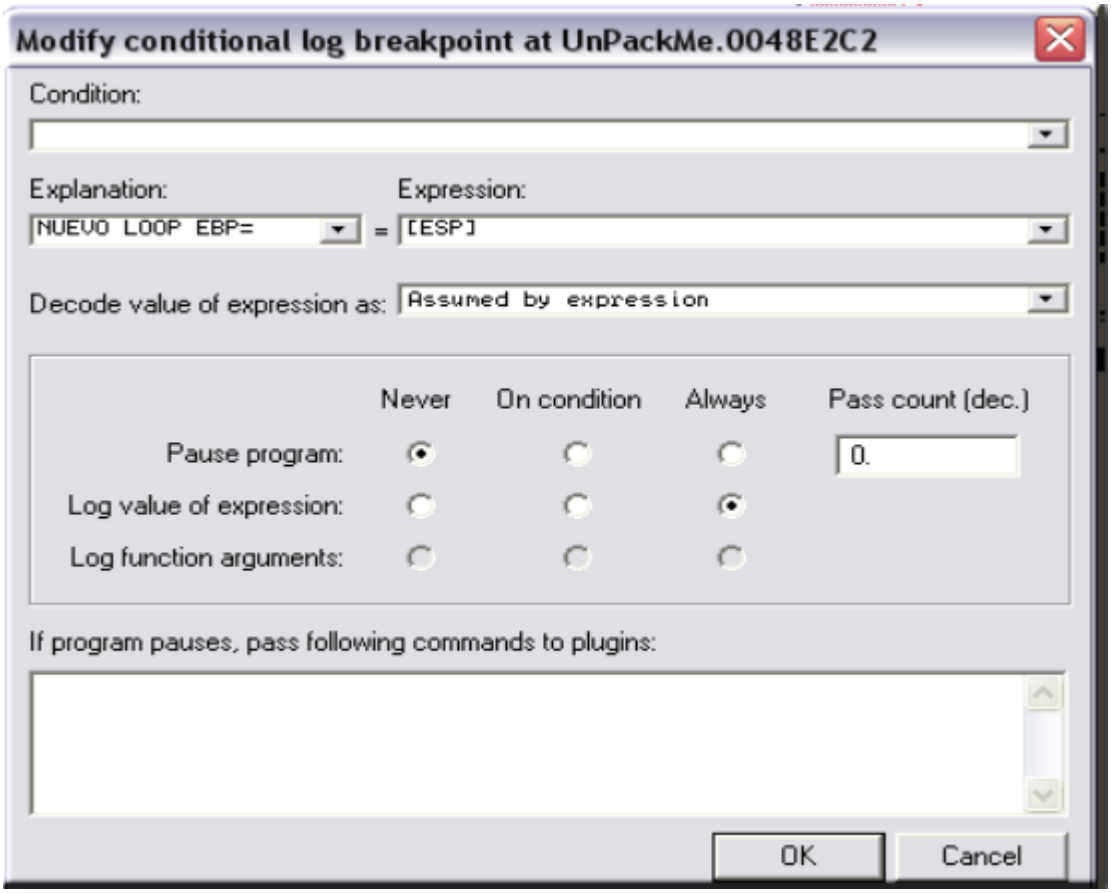
0047949B 89BD C0A44700 MOV DWORD PTR SS: [EBP+47A4C0], EDI

照例



接着是 EBP

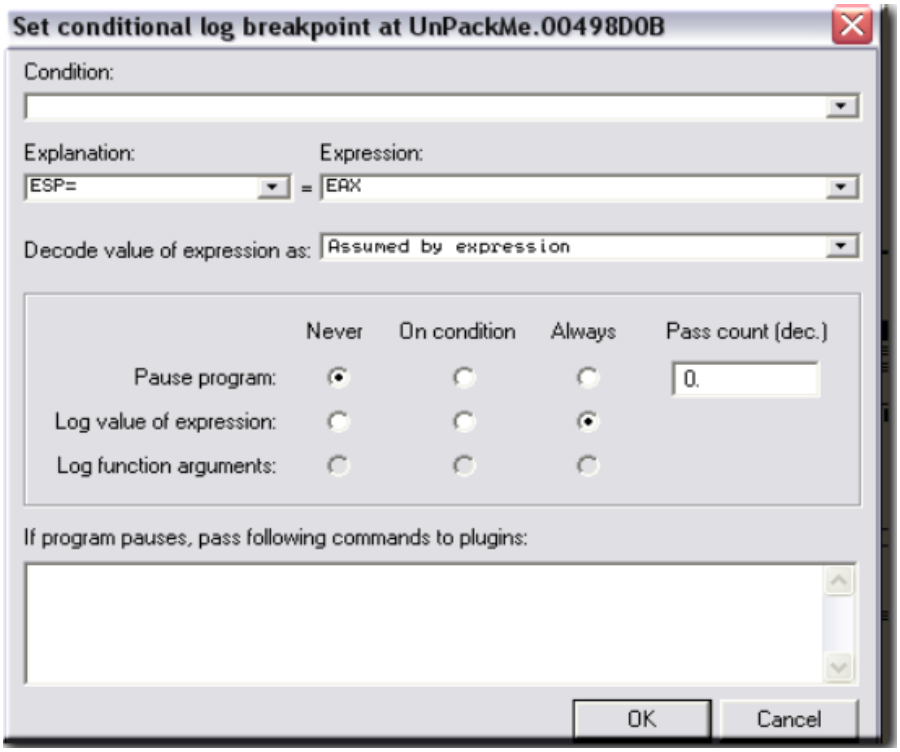
0048E2C2 8F85 C0A04700 POP DWORD PTR SS: [EBP+47A0C0]



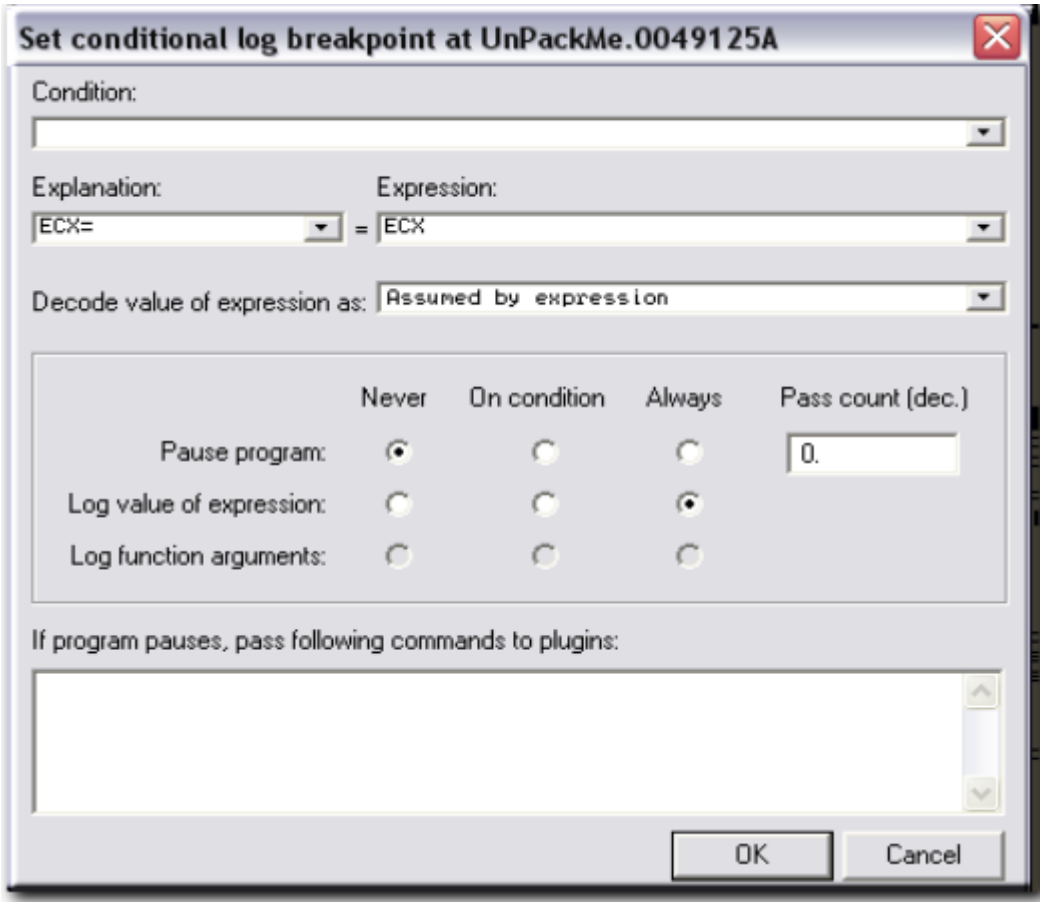
接下来是 ESP

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

继续调整条件断点



0049125A 这里更新 ECX。

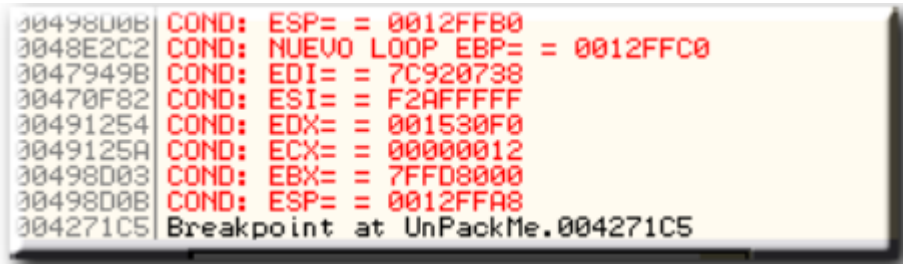


这里虽然我们不关心 EDX 的值,还是给它设置条件断点。

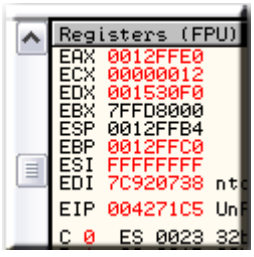
00491254 8995 ECBC4700 MOV DWORD PTR SS: [EBP+47BCEC], EDX

接着 EBX

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

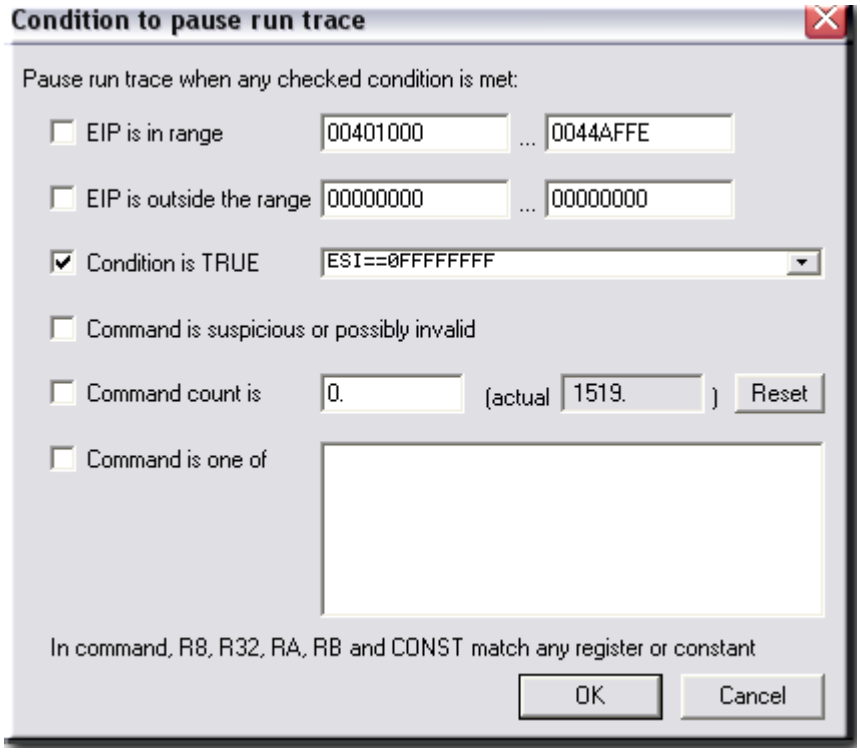


我们可以看到

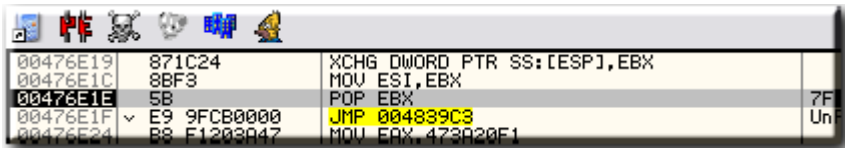


这里我们可以看到 EBP,ECX,EDX,EBX,EDI 这些值是相同的,ESP,EAX,ESI 的值有些差别。

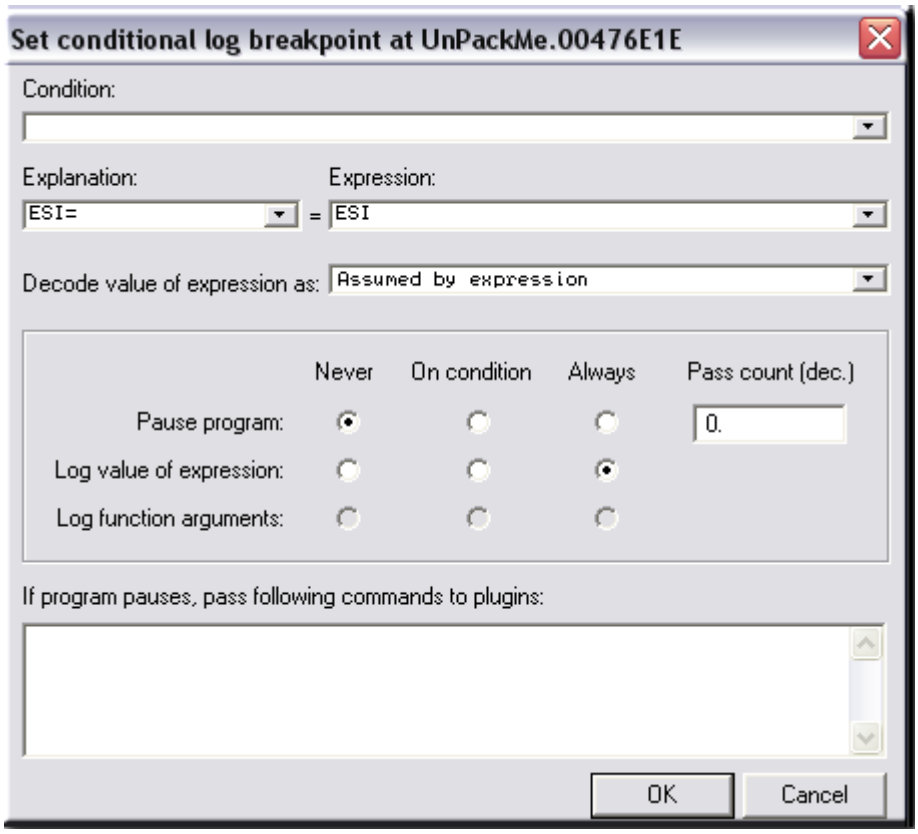
好,那么我们将条件断点的条件换一换,换成 ESI==FFFFFFFF 试试看。

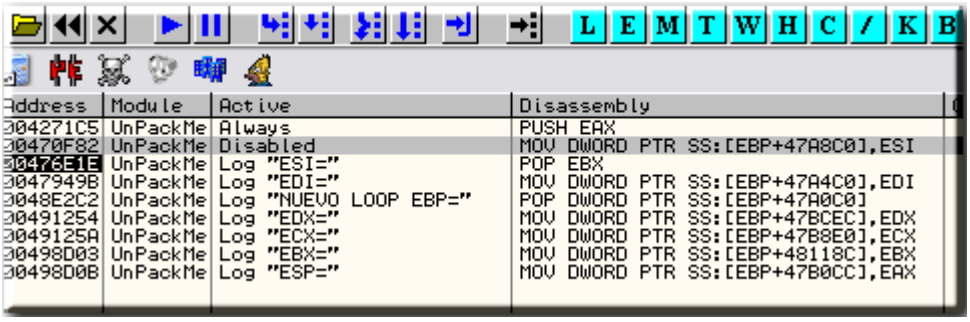


到了这里

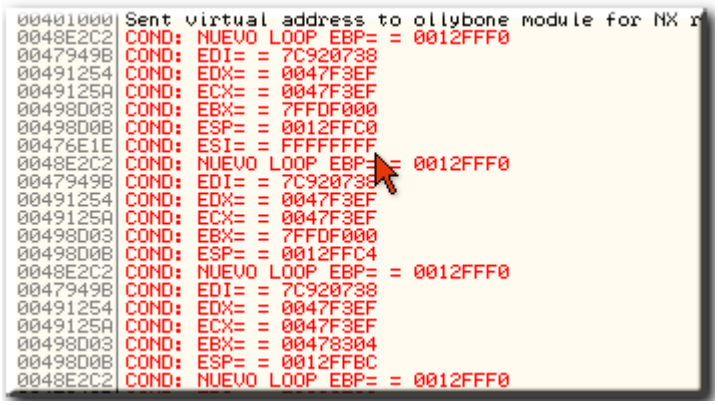


此时 ESI 的值为 FFFFFFFF。

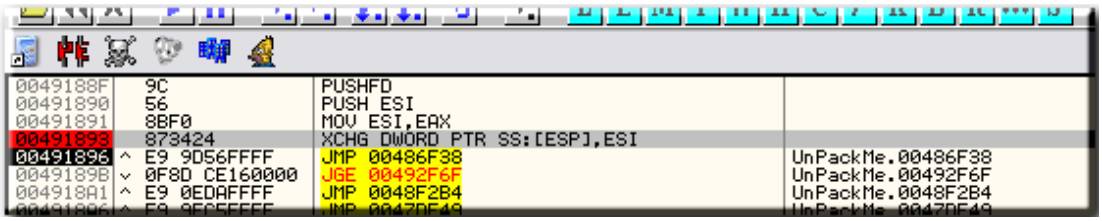




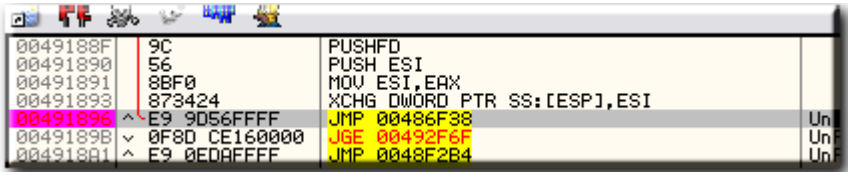
将针对 ESI 的断点禁用掉。



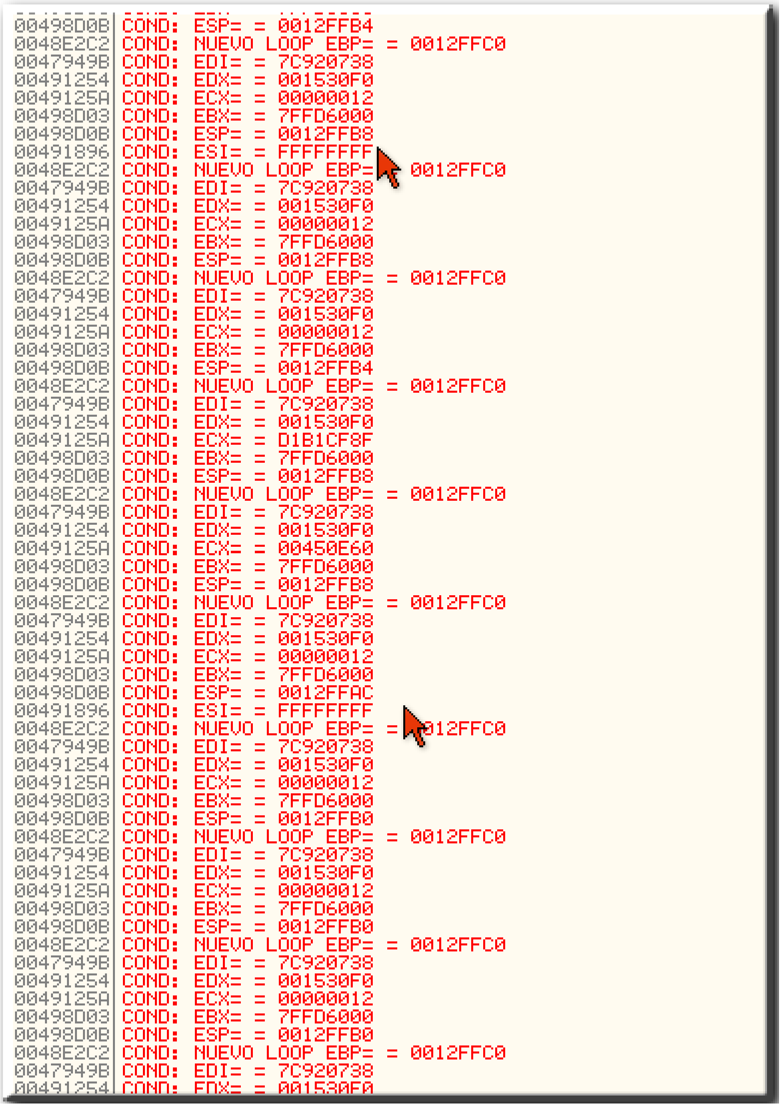
我们继续跟踪到了 47CE1E 这里。



恢复针对 ESI 的断点,条件依然设置为 ESI == FFFFFFFF。



这里已经获取到了 ESI 正确的值。



好了,本章结束。

这个系列也结束了。