第4章-汇编指令

【安于此生译】

之前的章节主要是理论知识,现在我们要在 OllyDbg 中实践一下,为后面打开基础。

OllyDbg 中几乎所有的标志我都有考虑,如果你遇到了我没有给出的指令,你可以查阅更加全面的汇编指南。

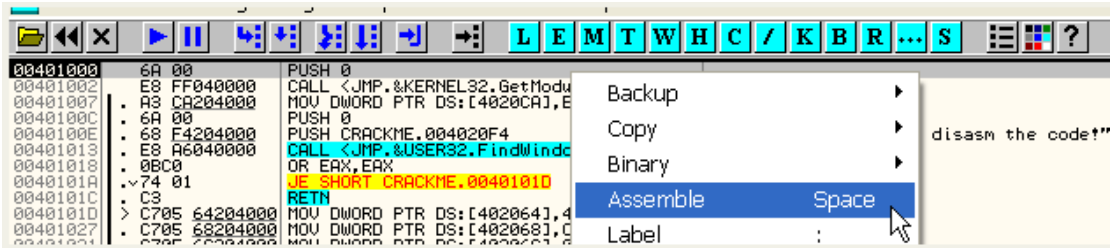
NOP（无操作）

运行这条指令不会对寄存器,内存以及堆栈造成任何影响,英文单词的意思是”无操作”，也就是说,它没有特殊的用途。例如,你用一个短指令来替换一个长指令的话，如果处理器没有错误,多余的空间将会被 NOP 填充。

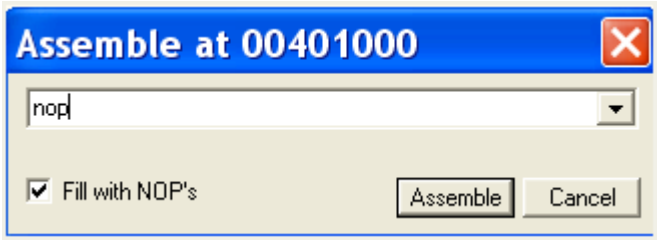
适当数目的 nop 指令可以将其他指令完全替换掉。

下面使用 OllyDbg 重新载入 CrueHead’a（CrackMe 的作者)的 CrackMe。

我们可以看到反汇编的源代码,如上图第一条指令是 PUSH 0,占两个字节,在这条指令上面单击鼠标右键选择 Assemble.



或者直接使用快捷键-空格键,在弹出窗口的编辑框中输入 NOP。



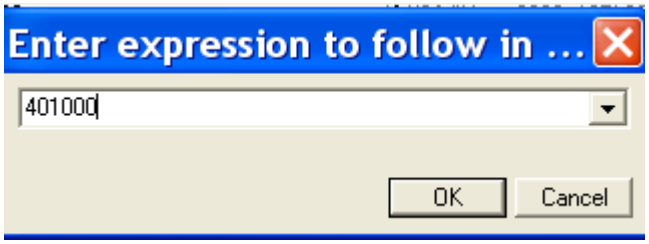
写入 NOP 指令后单击 Assemble 按钮。

这里我们可以看到 OD 设计的非常智能,考虑到 PUSH 指令占两个字节,OD 会使用两条 NOP 指令进行替换,而不是使用一条 NOP 进行替换。

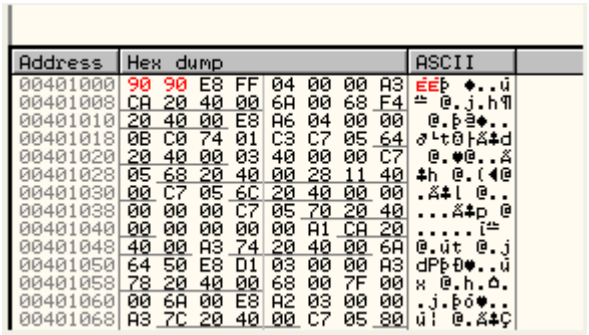
现在,在原来 PUSH 0的地方显示的两条 NOP 指令,单击 F7,指令一条 NOP 指令,可以看到,这里只改变了 EIP(保存了下一条要执行指令的地址)寄存器的值,并没有影响到其他寄存器,堆栈或者标志位。

现在我们需要在数据窗口查看这两个字节,它们的内存地址分别是401000和401001。

在数据窗口中,鼠标右键选择-“Go to”-“Expression”，输入你需要转到的地址。



这里我们需要输入401000。

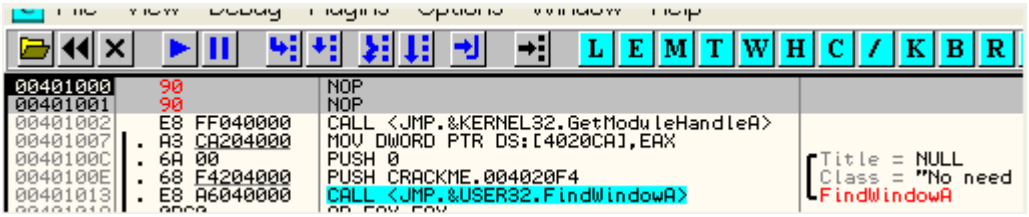


红色突出显示的是刚刚修改过的字节。前两个是90,然后 E8，FF 和04，00,00。这是一个 Call 指令的所有剩余字节。

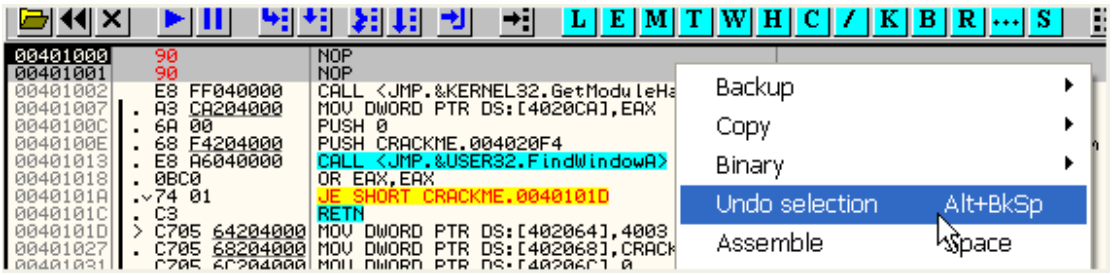
OD 可以撤销我们修改的指令吗?

呵呵,当然啦。

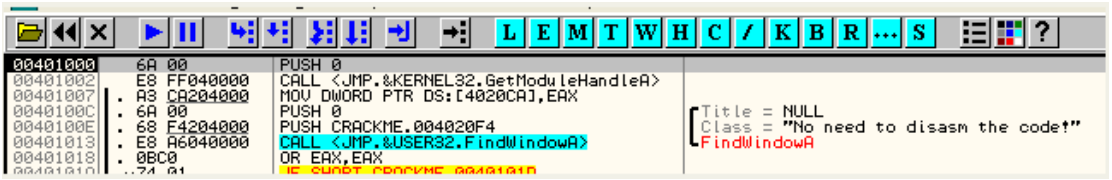
在数据窗口或者反汇编窗口中,鼠标拖选中两个字节。



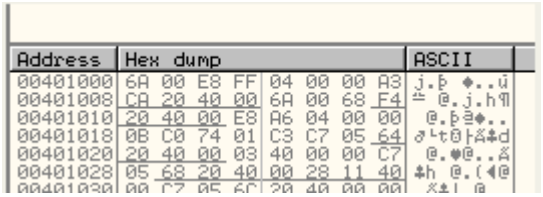
然后单击鼠标右键,选择“UNDO SELECTION”。



这样就恢复了原来的 PUSH 指令。



在数据窗口中的话,你就可以看到它原始的字节了。



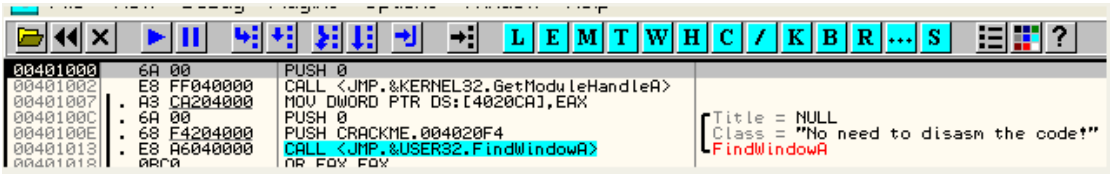
以上是关于 NOP 指令的所有内容。

堆栈相关指令的说明

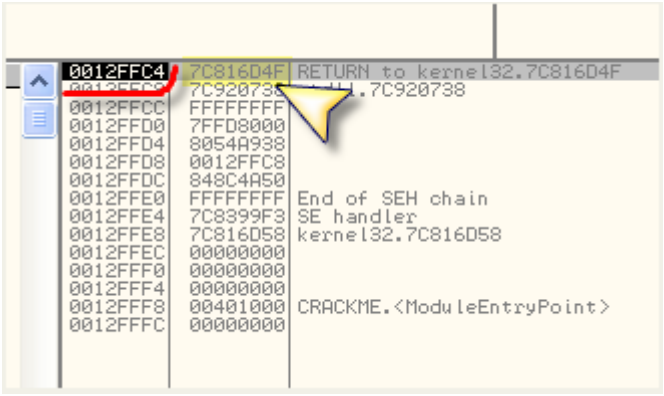
我们之前说过,堆栈就像一个信箱一样,越顶部的信越先被取出来。

PUSH

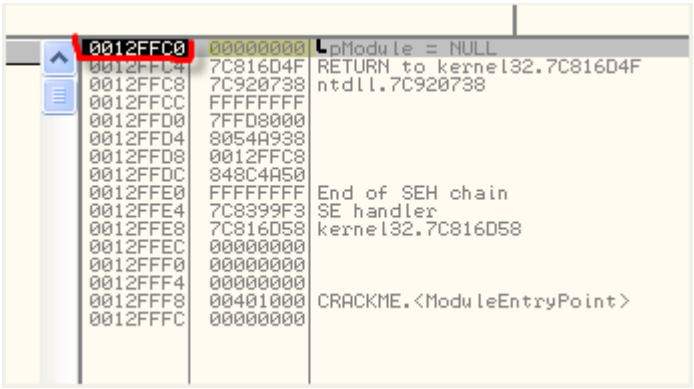
PUSH 指令-将操作数压入堆栈中。我们可以看到,CrueHead’a (CrackMe 的作者)的 CrackMe 的第一条指令就是 PUSH 指令。



“PUSH 0”指令将把0存入到堆栈的顶部,此时并没有压入堆栈,指令执行后,我们看看堆栈如何变化。堆栈的地址在你的机器上可能会有所不同,但效果是一样的。



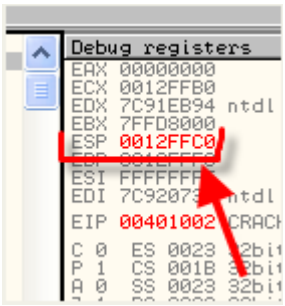
堆栈地址在我的机器上是12FFC4,可能与你机器上的不同，因为堆栈每次可能放置在不同的位置,其初始内容也可能有所不同,即这里的7C816D4F 和你的也可能不同。按 F7,将0压入堆栈,堆栈顶部的就是0了。



按下 F7,堆栈顶部我们可以看到加入了0。下面的12FFC4中仍然是7C816D4F,堆栈中的其他值并没有改变。

主要的变化就是堆栈的顶部变成了12FFC0(这是 PUSH 0指令执行的结果),新压入的数据总是在堆栈的顶部,并不会改变下面的数据。

这里也可以看到,ESP 的值变成了12FFC0。



当然 PUSH 指令不仅仅可以压入数值:

PUSH EAX 的话,那么堆栈的顶部将保存 EAX 的值。

同样的适用于其他的寄存器，你也可以压入特定内存地址中的值。

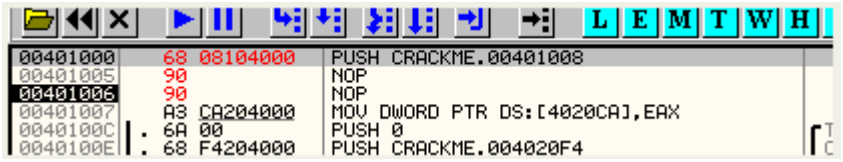
PUSH [401008]

注意,这和下面这条指令的解释是不同的。

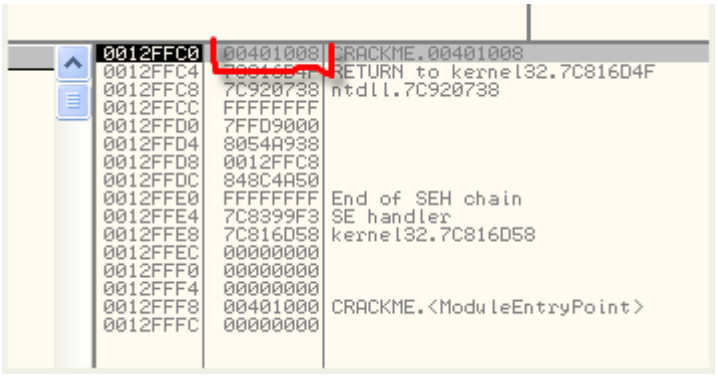
PUSH 401008

(没有方括号)

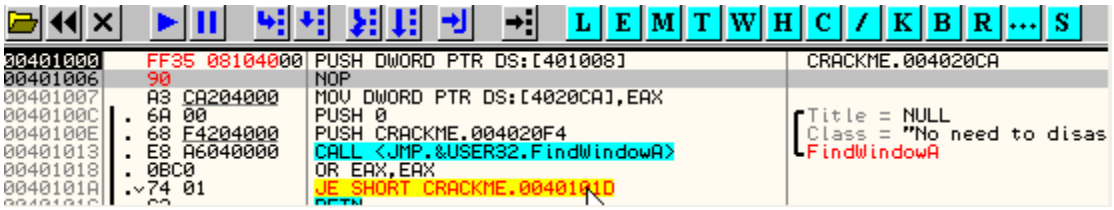
如果你使用的“PUSH 401008”,那么堆栈中将被放置401008。



执行以后,我们可以看到下面的结果:

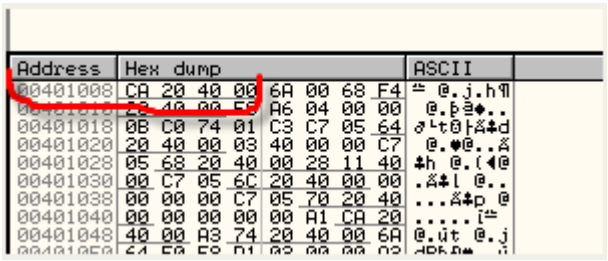


如果换成是“PUSH [401008]”

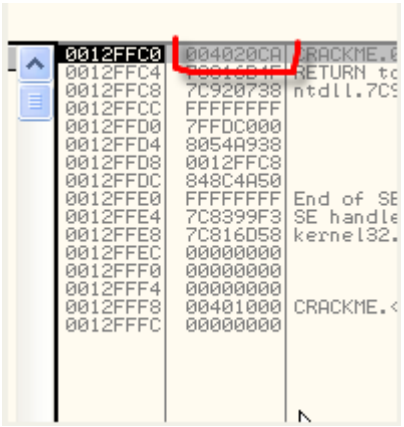


[401008]表示401008这个内存单元中存储的内容,这个时候我们得去数据窗口中查看它的值是多少。

在数据窗口中鼠标右键单击-Go to-Expression,输入401008,可以看到:



这四个字节是 CA 20 40 00。按 F7执行这条 PUSH 执行。

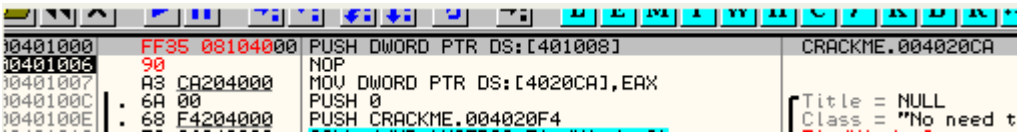


堆栈中我们可以看到在数据窗口中颠倒过来的值,即,它们被倒序放置。

读/写的内容在内存中倒序放置时处理器的特点之一。要怪你只能怪处理器的厂商了，嘿嘿。

通常情况下,没有用方框号括起来的只是一个数字。

现在我们知道,OD 中“PUSH [401000]”的意思了。



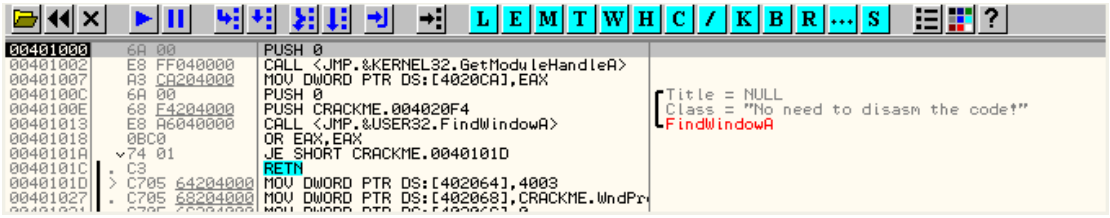
PUSH DWORD PTR DS:[401008]

像这样的,除非有明确规定,否则 OD 都是认为你要操作的是4个字节的内存,也就是 DWORD。其他格式会在其他指令中予以说明。

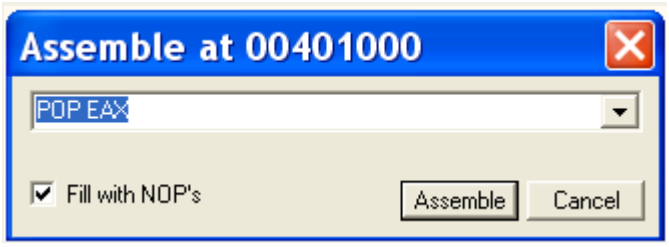
POP

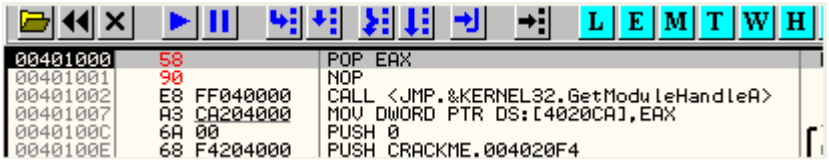
POP 指令是出栈:它会取出堆栈顶部的第一个字母或者第一个值,然后存放到指定的目标地址内存单元中。例如,POP EAX 从栈顶中取出第一个值存放到 EAX 中,随后的一个值随即变成栈顶。

我们还是看到 CrueHead'a 的 CrackMe 的开始的语句:

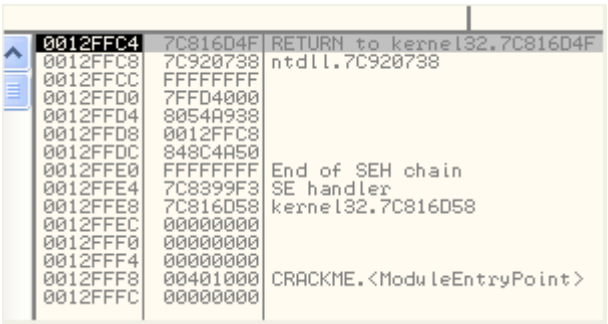


将第一条指令替换成“POP EAX”,注意在第一行,按空格键:

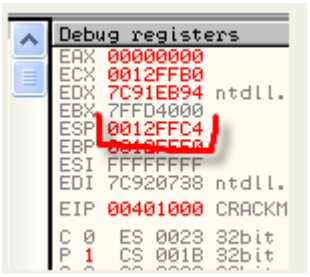




以下是执行此操作,堆栈变化情况的说明:

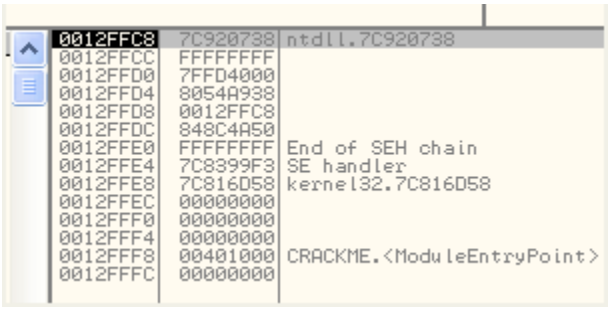


ESP 中存放的是12FFC4,它存放的是堆栈的顶部的内存地址。

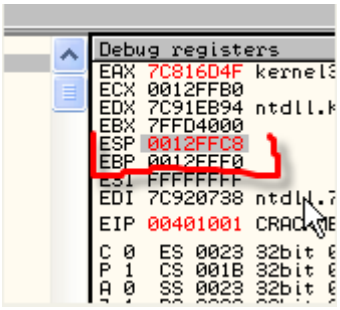


我们可以看到 EAX 的值是0(我这里的情况)。

按 F7键。



我们可以看到,原来堆栈顶部的值消失了,现在 ESP 为12FFC8。



原本在堆栈顶部的7C816D4F 现在到了 EAX 中。

同样地,如果用户是“POP ECX”,那么上面的值将会到 ECX 寄存器中或者你指定其他寄存器中。

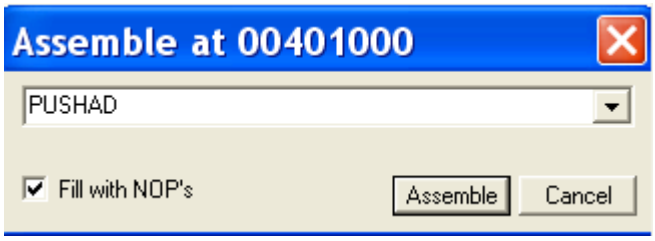
我们这里就研究了入栈和出栈指令。

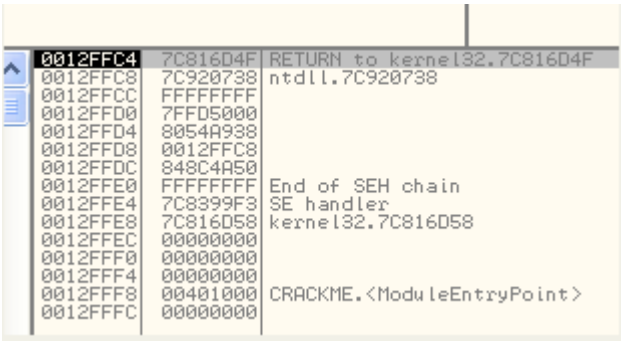
PUSHAD

PUSHAD 指令把所有通用寄存器的内容按一定顺序压入到堆栈中,PUSHAD 也就相当于’PUSH EAX,PUSH ECX,PUSH EDX,PUSH EBX,PUSH ESP,PUSH EBP,PUSH ESI, PUSH EDI’。

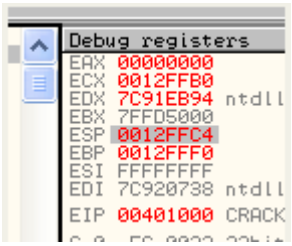
让我们来看看。

再次载入 CrueHead’a 的 CrackMe, 调用右键菜单来修改指令为 PUSHAD。

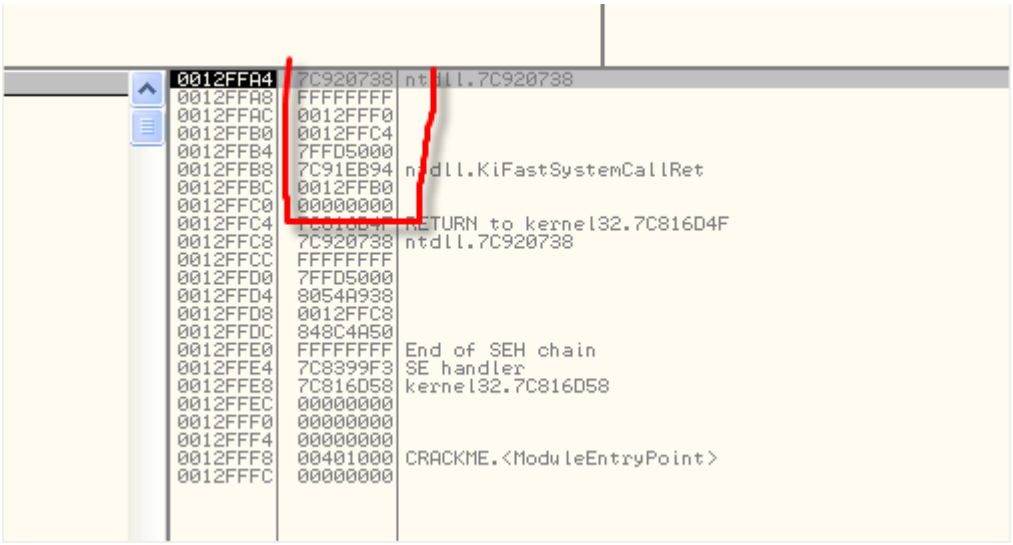




这是我当前寄存器组的情况。



按下 F7键,看看现在堆栈的情况:

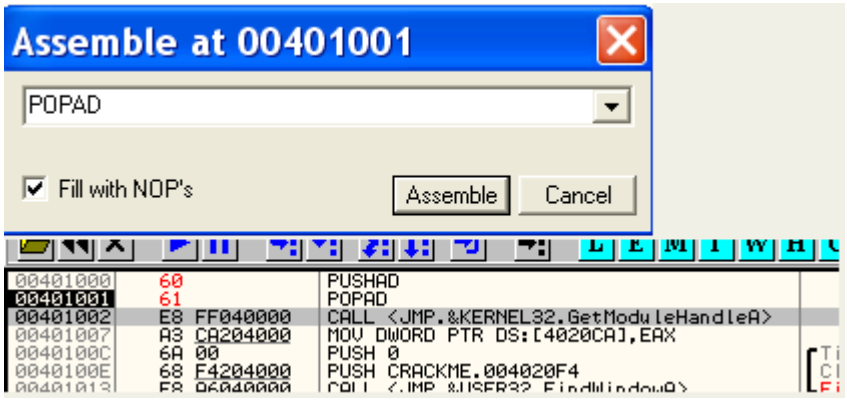


看到所有寄存器的值都被压入堆栈了。12FFC4存放的是之前堆栈顶部的值,然后上面就是0(PUSH EAX),12FFBC 存放的是 ECX 的内容,接下来是 EDX 寄存器的内容,以此类推。

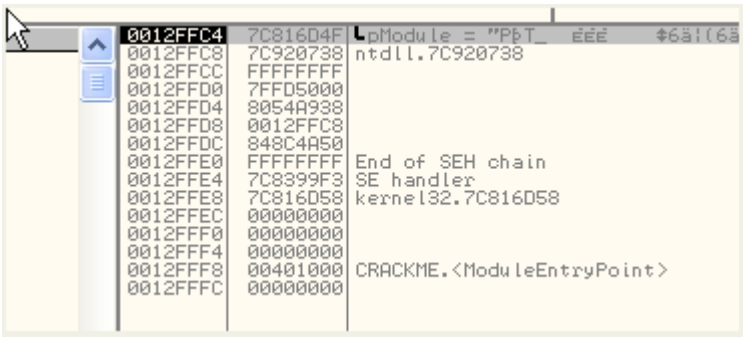
POPAD

该指令与 PUSHAD 正好相反,它从堆栈中取值,并将它们放到相应的寄存器中。POPAD 等价于“POP EDI,POP ESI,POP EBP,POP ESP,POP EBX,POP EDX,POP ECX,POP EAX”。

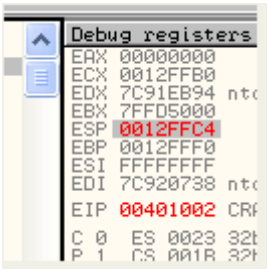
正如前面的例子,我们将第一条指令修改为 POPAD:



由于堆栈中保存的是之前寄存器组的值,执行 POPAD 其返回原来的状态。



因为之前执行了 PUSHAD,所以恢复的寄存器及其值和之前相同。



PUSHAD-POPAD 指令经常被使用,例如:某个时刻你需要保存所有寄存器的内容,然后修改寄存器的值,或者进行堆栈的相关操作,然后使用 POPAD 恢复它们原来的状态。

也有以下的用法:

PUSHA 等价于 'PUSH AX, CX, DX, BX, SP, BP, SI, DI'。

POPA 等价于 'pop DI, SI, BP, SP, BX, DX, CX, AX'。

PUSHA,POPA 和 PUSHAD,POPAD 就像姐妹一样,只不过它们在16位程序中使用,所以我们不感兴趣,OllyDbg 是一个32位程序的调试器。

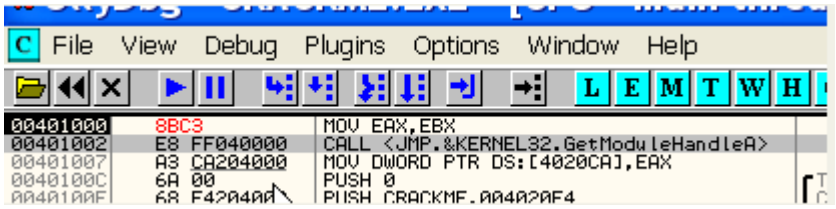
赋值指令的说明

MOV

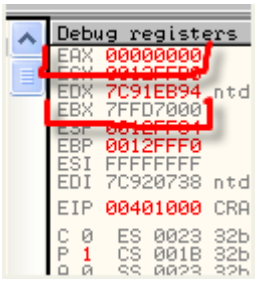
该指令将第二个操作数赋值给第一个操作数,例如:

MOV EAX, EBX

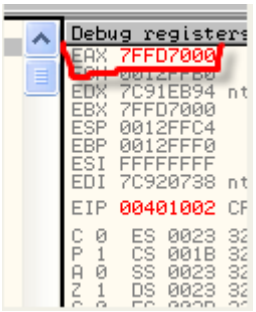
EBX 值赋值给 EAX。继续用 OD 载入 CrueHead'a 的 CrackMe。



两个寄存器的值是不一样的,我们只是看寄存器:



在我的机器上,EAX 为0,EBX 为7FFD7000。这些初始值有可能与你机器上的不一样,我们只是看赋值的过程,按 F7键,EBX 的值就赋值给了 EAX。

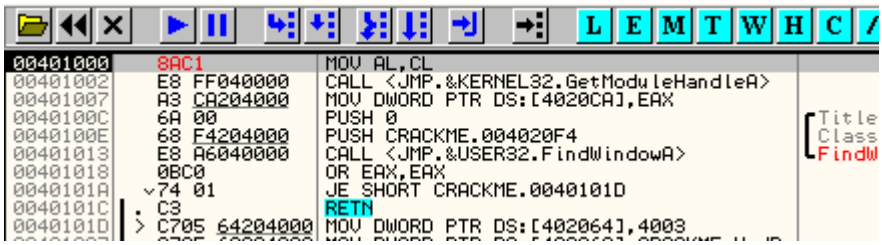


明白了吗?

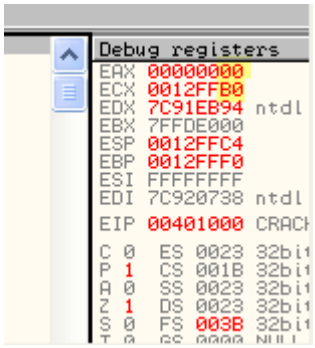
MOV 指令操作数有很多可以选择,例如:

MOV AL, CL

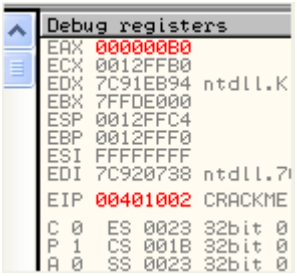
这条指令时将 CL 的值赋值给 AL。在 OD 中写上这句。



寄存器:

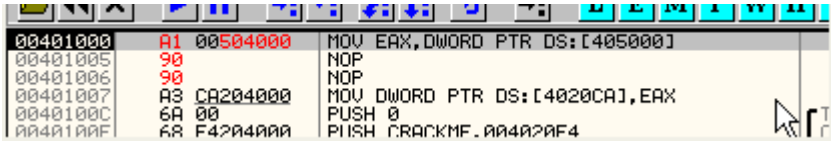


请记住，AL-是 EAX 的最后两位数字以及 CL-是 ECX 的最后两位数字。按 F7



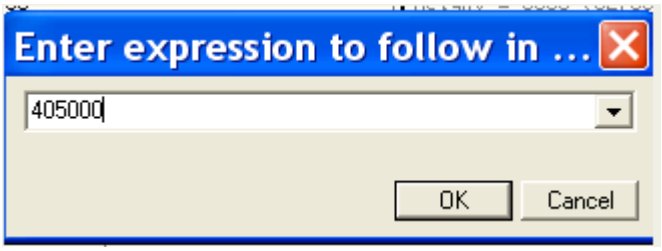
可以看到只是将 B0赋值给 AL 了,并不改变 EAX 和 ECX 的其他内容,即 EAX 的最后两位数字。

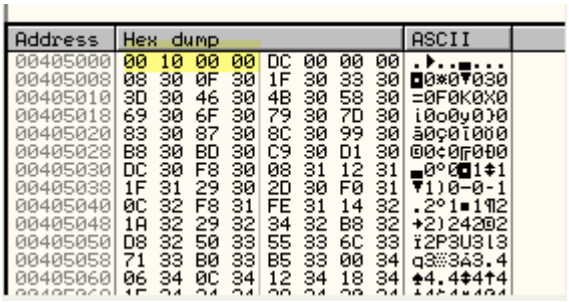
也可以给内存单元赋值,反之亦然一个寄存器的内容。



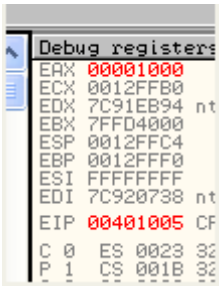
上图这种情况下,将要给 EAX 给的值是405000这个内存单元中的值,正如前面提到的,DWORD 意味着你必须移动4个字节。如果该指令给出了一个不存在的内存单元可能会导致错误。我们可以用 OD 很容易的检查出来。

在数据窗口中右键单击选择 Go to-Expression 转到405000处.





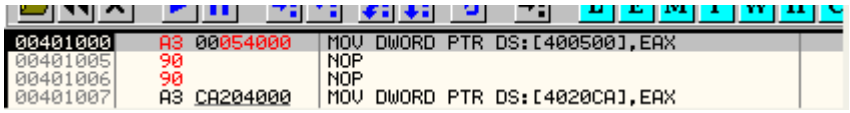
可以看到内存中的内容是-00100000。由于内存中内容是倒序存放的,那么 EAX 中将被装入00001000,按下 F7键,看看会发生什么。



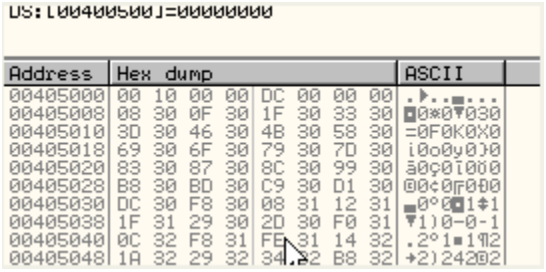
这里是1000，是从内存中读取出来的。现在,如果我们想写一个值到这个地址:

MOV DWORD PTR DS:[400500],EAX

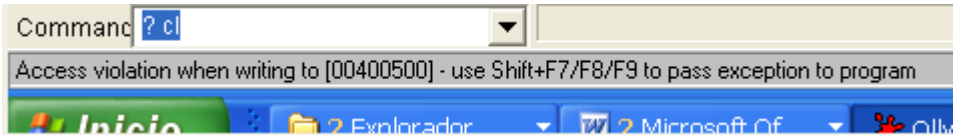
写入该指令。



在数据窗口中查看405000:



然后,按 F7键发生异常:



注意异常的描述,是由于我们要写入400500这个内存地址导致的内存访问异常。

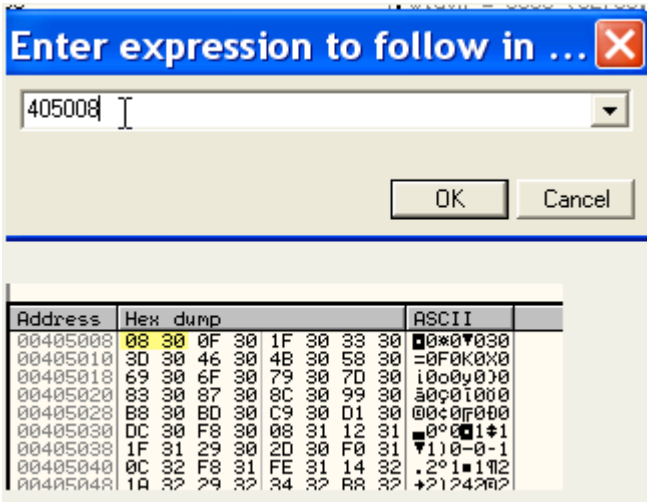
如果移动的是4个字节使用 DWORD,移动两个字节的话使用 WORD。

例如:

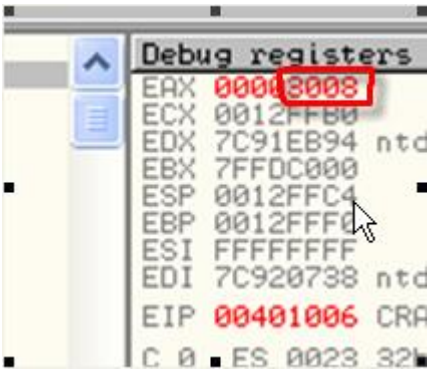
MOV AX,WORD PTR DS:[405008]

将405008内存单元中的两个字节赋值给 AX。这种情况下,我们不能用 EAX,这里移动的只有两个字节,所以你必须使用一个16位的寄存器。

在数据窗口中查看405008.



按 F7赋值给 AX 的只有两个字节。如下:

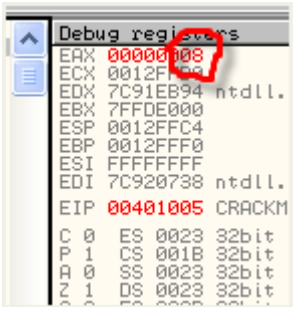


在 AX 中存放的是内存中相反的内容,EAX 的其他部分并没有改变。

同理,一个字节使用 BYTE。

MOV AL, BYTE PTR DS:[405008]

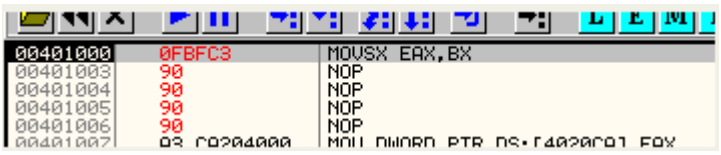
这种情况下,只有最后一个字节赋值给 AL 了，即08。



MOVSX (带符号扩展的传送指令)

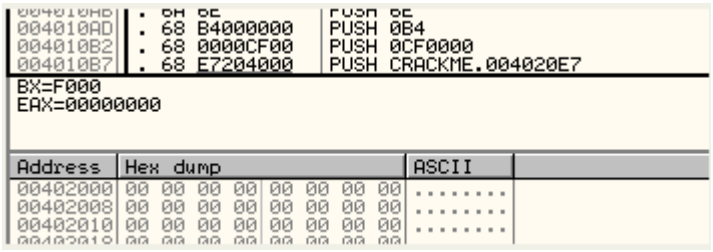
第二个操作数可能一个寄存器也可能是内存单元,第一个操作数的位数比第二个操作数多,第二个操作数的符号位填充第一个操作数剩余部分。 下面是一个例子。

这里我们还是在 OD 中载入 CrueHead'a 的 CrackMe。

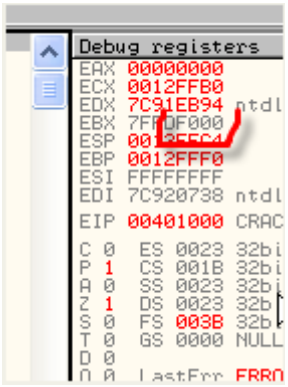


这里我不会说太多，因为我想在座各位自己来计算这个操作数求值,嘿嘿。

在 OD 中，反汇编窗口和数据窗口中间有一个解释窗口。

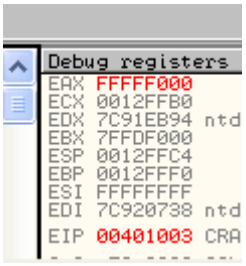


这里我们可以看到,解释窗口向我们展示了我们操作数里面存放的值。我这里,BX 存放的是 F000。



同时可以看到,EAX 的值为0,所以这些东西总是可以帮助我们理解 OD 要执行的指令(我希望你已经掌握了其中的概念和寻址方式,嘿嘿)。

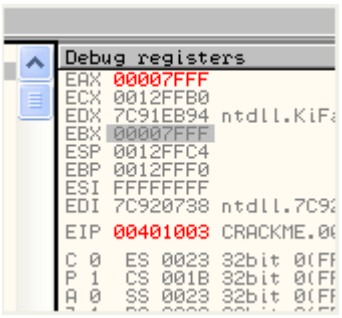
按 F7键。



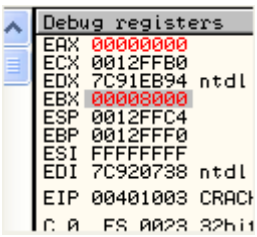
看到 AX,BX 中存放的是 F000，并且 EAX 剩余部分填充为了 FFFF,因为 F000是一个负的16位数字。如果 BX 存放1234,EAX 将等于00001234，即左边的字节将会被0填充,因为1234是一个正的16位数字。

16位数和32位数的正数和负数的概念是一样的,只不过16位数的范围是0000到 FFFF。0000到7FFF 是正数,8000到 FFFF 是负数。

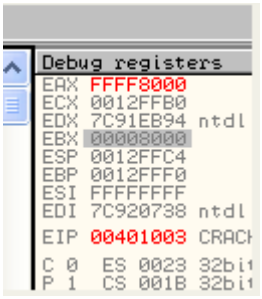
如果我们把 BX 修改为7FFF,把 EAX 修改为0，然后执行这条指令会发生什么呢。



AX 被赋值为了7FFF,其余部分被填充为0了-因为7FFF 是正数。我们还可以把 BX 修改为8000(负)。



按 F7键。



AX 被赋值为了8000,剩余的部分为 FFFF,因为8000是负数。

MOVZX (带0扩展的传送指令)

MOVZX 类似于前面的语句,但是这种情况下,剩余的部分不根据第二个操作数的正负来进行填充。我们这里不提供范例,因为和上面是相似的,剩余的部分总是被填充为0。

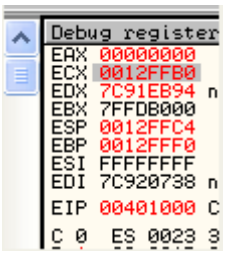
LEA (取地址指令)

类似于 MOV 指令, 但是第一个操作数是一个通用寄存器,并且第二个操作数是一个内存单元。当计算的时候要依赖于之前的结果的话,那么这个指令就非常有用。

我们在 OD 中写入以下指令:

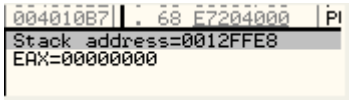
在这种情况下,有括号,但不需要获取 ECX+38指向内存的值,只需要计算 ECX+38的值即可。我这里,ECX 的值为

12FFB0。



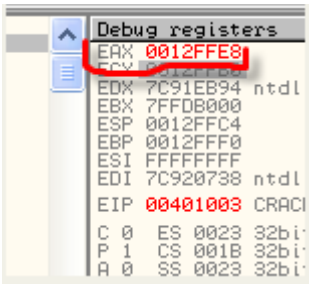
在这个例子中,LEA 指令就计算 ECX + 38的值,然后将计算的结果赋值给 EAX。

解释窗口中显示了两个操作数。



它表示,该操作数是12FFE8,也就是 ECX+38的值,并且 EAX 的值为0。

按 F7键。



指定的地址被存放到了 EAX 中，因为完成的是赋值操作,所以我们会认为操作数是内存单元中的值，但是实际上

操作数仅仅是内存单元的地址,而不是里面的内容。

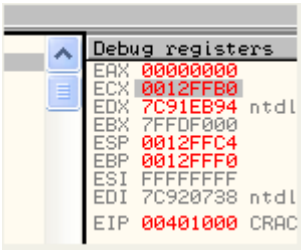
XCHG (交换 寄存器/内存单元 和 寄存器)

该指令交换两个操作数的值,例如:

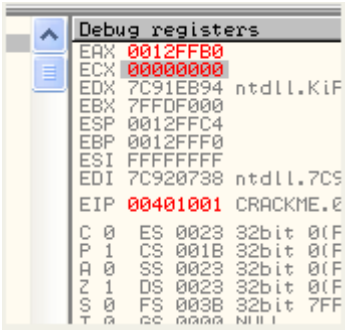
XCHG EAX,ECX

EAX 的值将被存放到 ECX 中。反之亦然。我们在 OD 中来验证这一点。

在我的机器上,EAX 的值为0,ECX 的值为12FFB0。

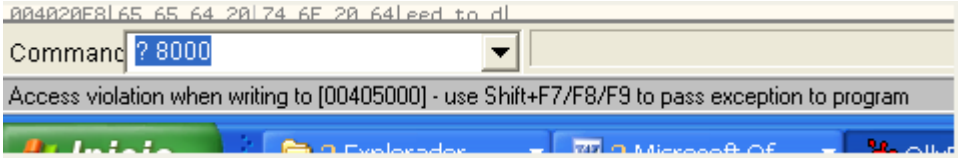


按 F7键,看到他们交换了数值。



你也可以使用这个指令来交换寄存器和内存单元的值,本节之前有提到。

按下 F7键:



这个例子我们在 MOV 指令使用过,这里我们对该内存地址没有写权限。

好了,这就是常用指令的第一部分,是非常有用的以及有趣的,我给出的例子能够体现这一点。在接下来的部分,我们将继续研究指令。