CVE-2010-3333 漏洞分析

一打开 poc 就可以看到万能跳转（jmp　esp）地址的存在。如图 1 所示。

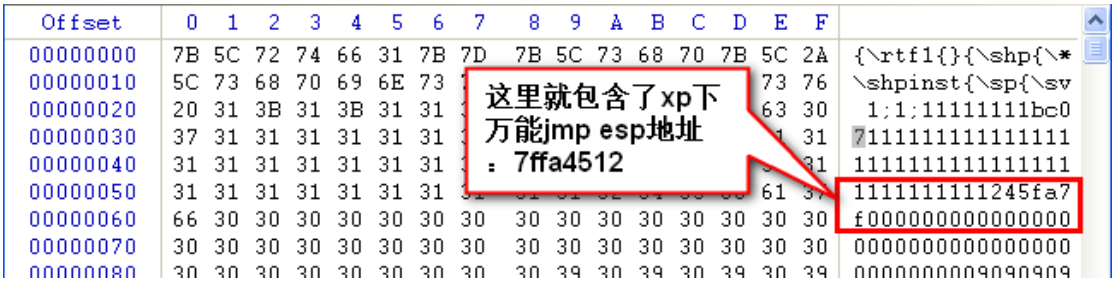

图 1

那么该漏洞的调试就可以从这个地址入手了。

打开 OD 后，载入 winword.exe,ctrl+g 给 7ffa4512 下断点。如图 2 所示

图 2

找到该地址后，由于该地址的特殊性，只能给它下硬件断点。

随后按下 F9 执行该程序。几秒后，打开了 word.exe 程序，然后点击“文件”->“打开”，载入我们的 POC。

果断程序停在了我们 所设置断点的地方，如图 3 所示。

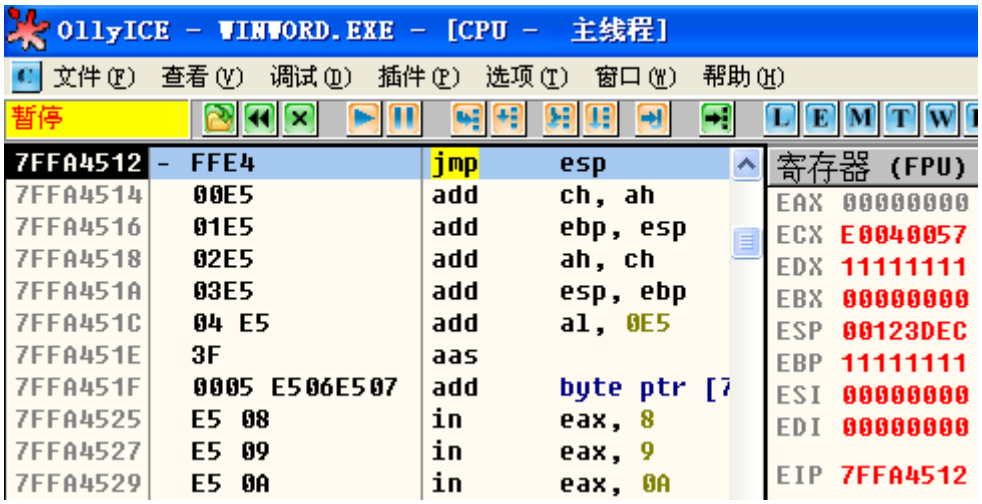

图 3

这是我们可以观察寄存器的值，很明显是发生了溢出才导致了这些不正常值的出现。

按 F8 单步执行下去后，我们就可以看到程序跳进了 shellcode 区域去执行。

这个时候我们观察栈区的数据，或许可以发现程序的调用。

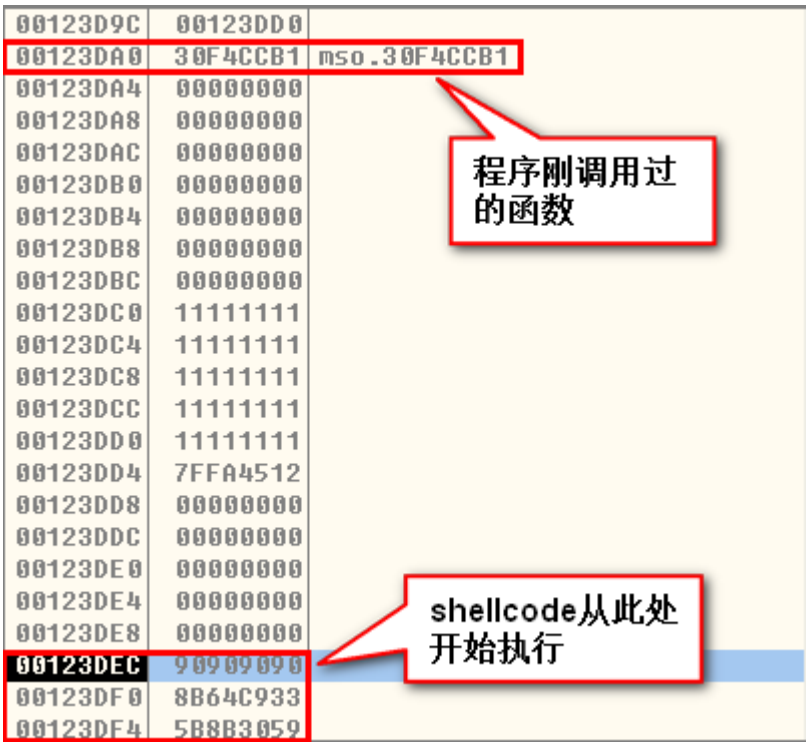

图 4

从图 4 中可以看出，在跳转到 jmp esp 地址之前，word 程序调用了 mso 模块中的地址为：30f4ccb1 的函数，难道就是该函数发生了问题？？我们在此地址下下断点，来看看！如图 5

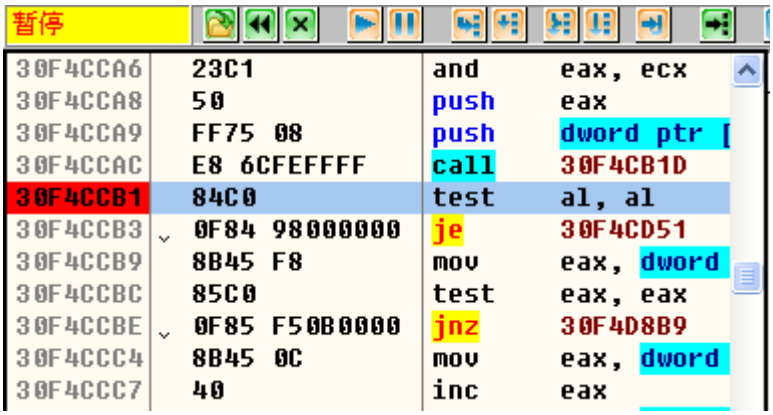

图 5

找到该地址，然后下断点。然后重新载入该程序。

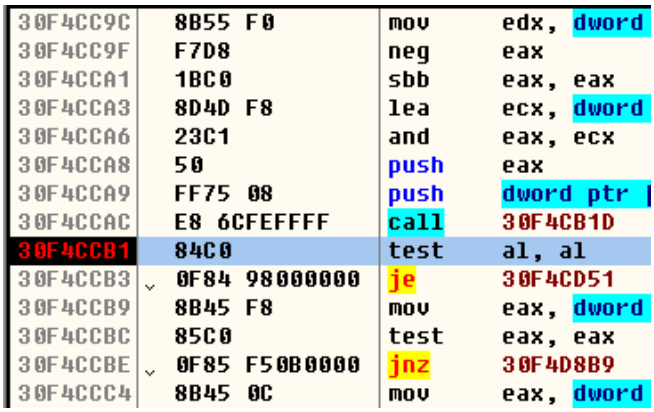

图 6

果断程序妥妥的停在了我们刚设置断点的地址，如图 6。
这个时候，我们继续观察战中的变化，看 shellcode 是否已经被 copy 到栈区。

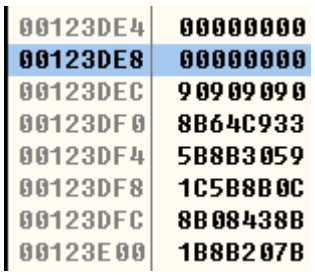

图 7

从图 7 中可以看到，shellcode 已经被拷贝到了栈区，那么也就是说明溢出的过程和这个函数无关。我们只能继续往上寻找，查找在调用这个函数之前是哪个函数出了问题。这个时候我们发现，在它上面有个函数调用，难道是这个函数的问题？

图 8

为了验证图 8 中的问题，我们对该函数调用的任意一行指令下断点。看那时候栈区的情况，如图 9。

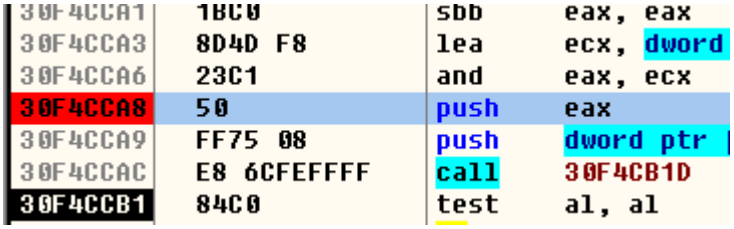

图 9

然后重新载入程序。

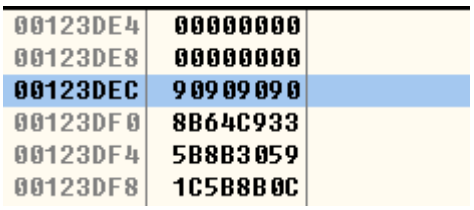

图 10

同样，在此时，栈区数据又已经被 shellcode 所覆盖.如图 10。没办法，我们只能继续往上走。我将断点设置在此处，如图 11。

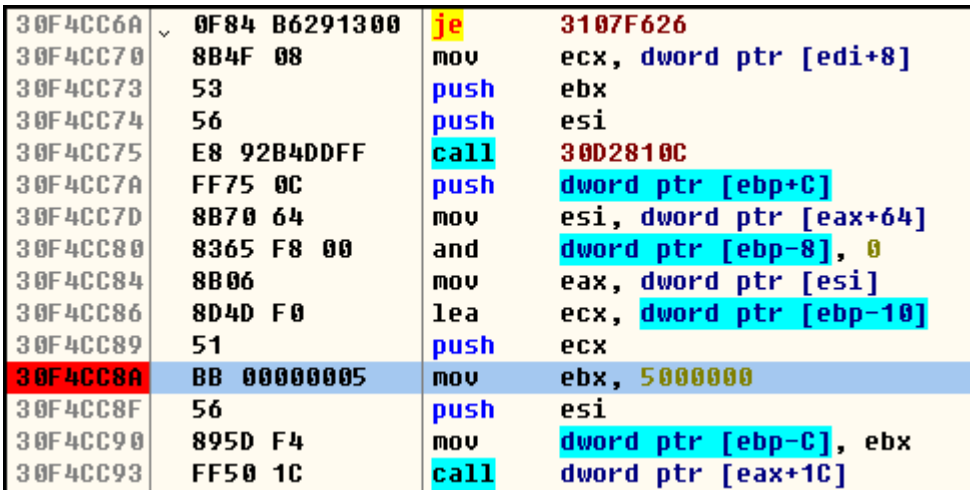

图 11

重新载入程序后运行。

然后我们发现发现这个时候 shellcode 并未覆盖该栈区。如图 12 所示。

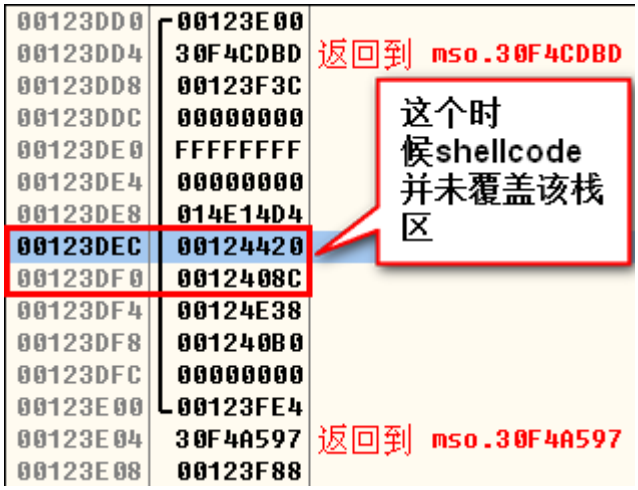

图 12

经过对断点前后代码的简单分析，我们确定了溢出的过程就是图 13 中的那个函数中。

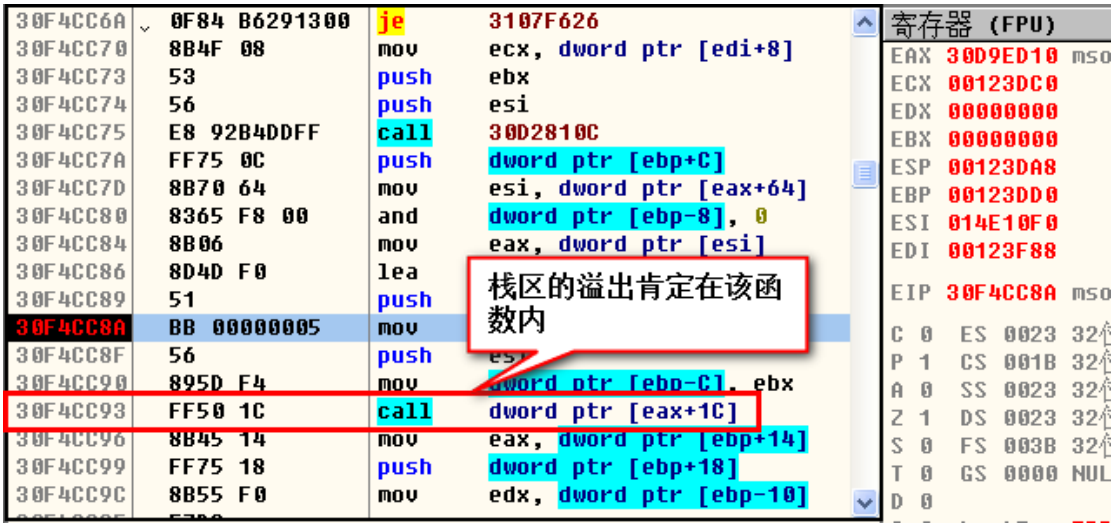

图 13

于是我们按 F8 单步调试，然后 F7 进入该函数的调用。为了时刻观察栈区数据什么时候覆盖，我们可以用锁定堆栈的方法来时刻监控。

根据跟踪的状态，发现执行了这一步（图 14 中所示）之后，栈区的数据就被覆盖了。

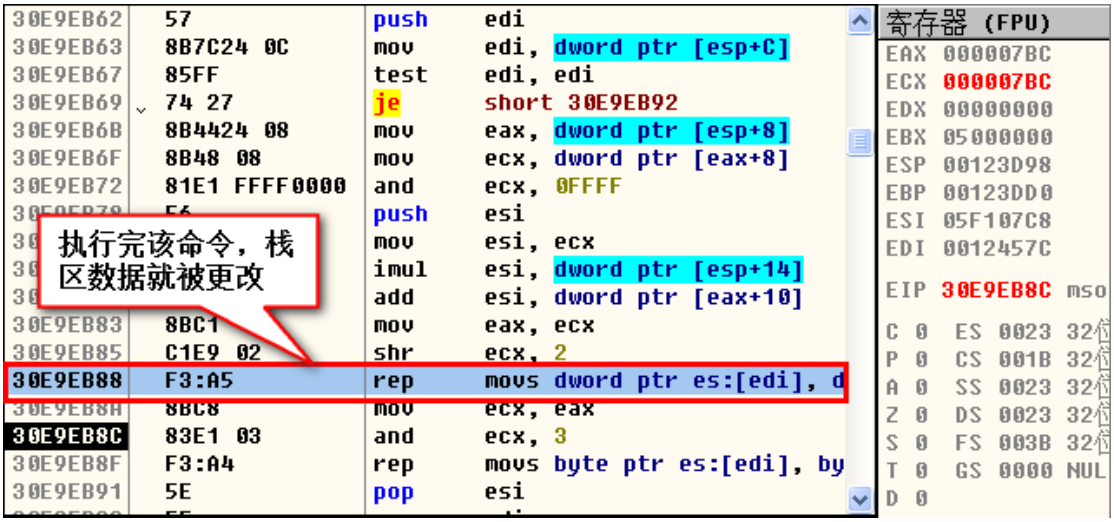

图 14

30E9EB88	F3:A5	rep	movs dword ptr es:[edi], dword ptr [esi]

这一条指令的意思就是说：将 esi 所指向的串移动到 edi 所指向的区域，移动的次数为 ecx 次。那么，很显然，由于移动的次数比预期的多了，所以才造成了这次的栈溢出。那么我们可以确定是 ecx 大小问题，是由于程序没有对 ecx 的大小做检查么？

在程序的上面，我们看到，程序有对 ecx 的长度进行检查，但是明显检查的不够合理。如图 15

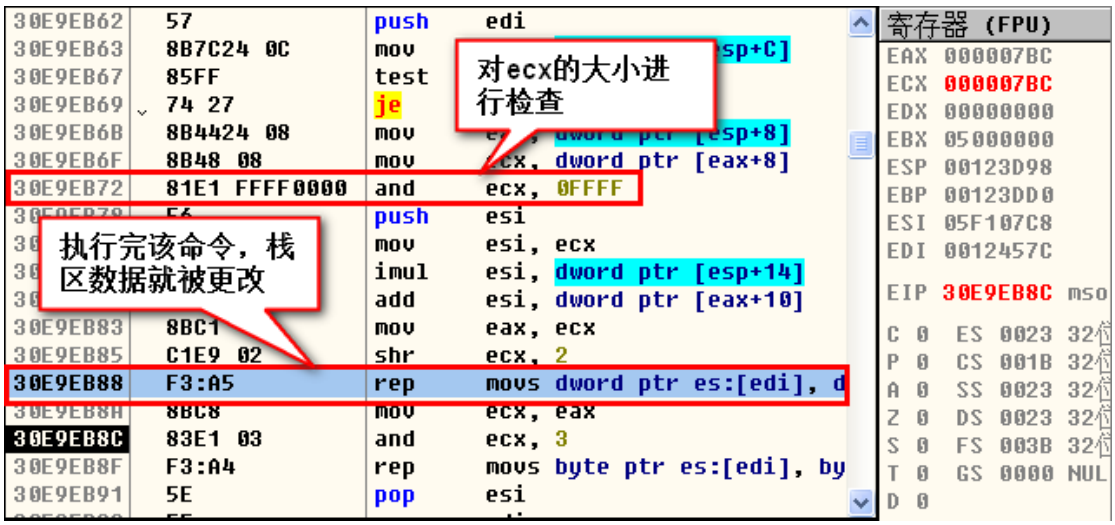

图 15

进行图 15 的检查，ecx 的值的范围可以在 0000h-1111h，明显不太合理，但是分配给他的空间有多少呢？我们继续往上查看。

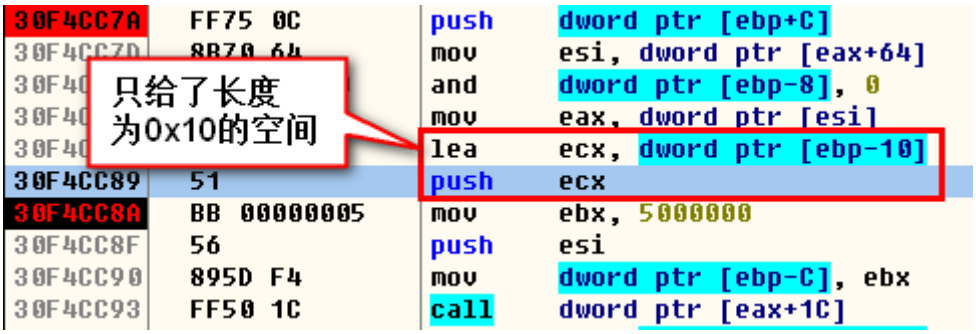

图 16

如图 16，他分配的大小只有 0x10，也就是说 rep 的次数只能是 16 次。但是由于这个 ecx 值的大小未受到合理的检查，所以才导致了该溢出。

新增部分：

在图 15 中可以看出，是 ecx 的值影响了循环次数，才导致了溢出，那么是什么影响了 ecx 的值呢？我们继续来分析。

图 16

从图 16 中，我们可以看出 ecx 值大小是由[eax+8]所指向的内存单元所决定的，具体大小如图 17 所示。

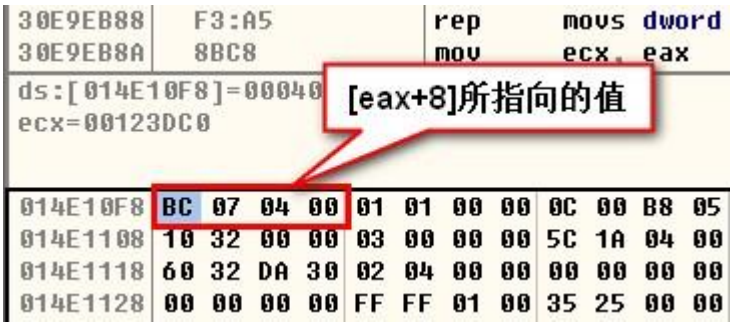

图 17

为了继续深入，我们必须知道是什么命令影响了这个[014E10F8]地址的值。由于该地址处于数据区，我们只能对该地址下硬件写入断点。如图 18 所示。

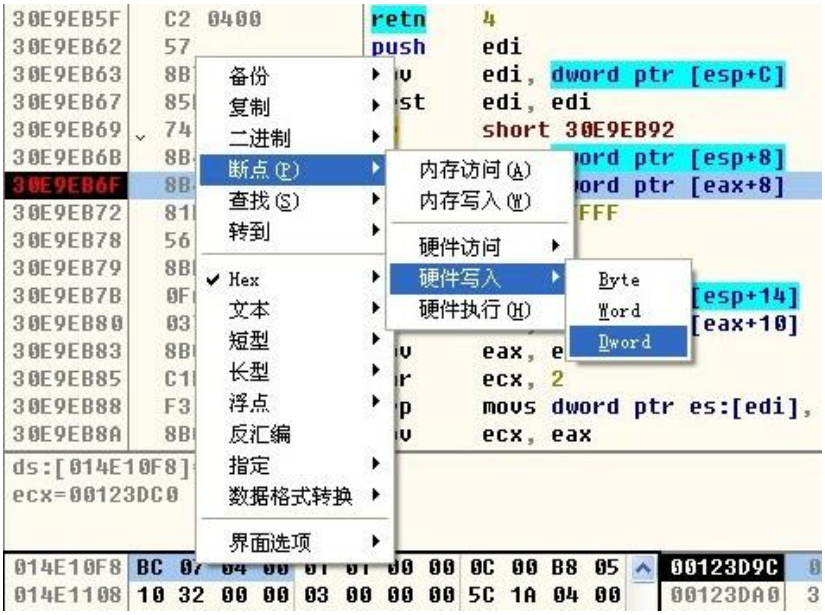

图 18

重新载入，运行程序后，在第三次断于该断点的地方，他的值被改成了图 18 中的值。我们来研究下他执行的代码，如图 19 所示。

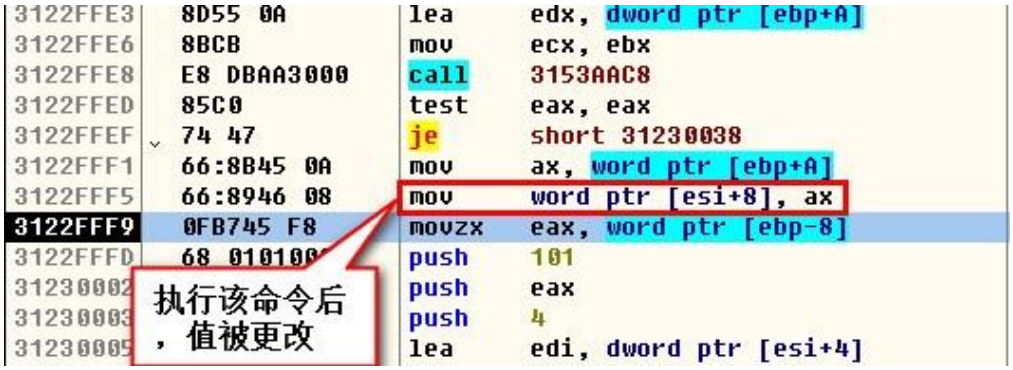

图 19

从图 19 中，可以看到他的值的是因为 ax 寄存器才被影响的，通过上一条命令可得，ax 的值是被[ebp+A]所指向的值决定的。那么我们可以来继续研究[ebp+A]所指向的值是什么时候被更改为现在这个值的。

图 20

图 20 中说明了[ebp+A]所指向的值。那么我们继续来研究影响这个值的因素。和上面一样，我们对该地址下硬件写入断点，然后重新载入程序。

但是，杯具的是，由于该地址一开始就被 word 进程所开辟，对于该地址的写入过程有成千上万次，我们无法定位具体是哪一次将他的值改成了 07bc，才造成了后面的溢出。

于是我想到了条件断点，在每次断点停下来时，让 OD 工具判断下我的附加条件是否满足，如果不满足，则继续跑程序。但是，很可惜，OD 调试工具的条件断点只能用在普通断点上，对于硬件断点无法设置附加条件。

但是，或许我们可以通过插件的方式来实现硬件断点的条件断点。通过搜索，果真有此类插件的存在。

下载完该插件，然后用 OD 载入该插件，设置好硬件断点的条件断点（当该地址[00121cba]的值变为 07bc 时，程序中断，如图 21 所示），于是重新载入。

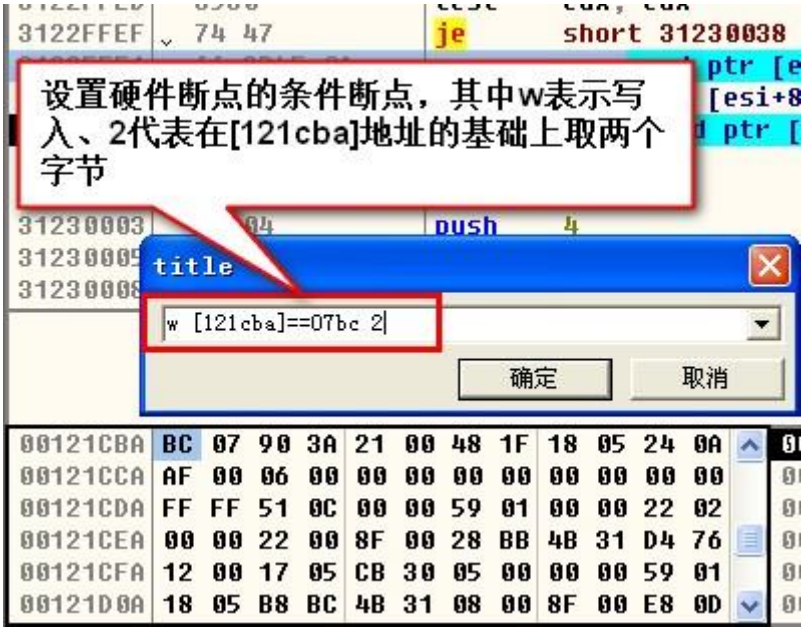

图 21

但是很可惜，程序并没有因此而暂停，反而一下运行到了最后。

难道是插件写的有问题？我设置的有问题？经过多次实验，论证之后才发现原因。程序在写入[00121cba]地址时，并非将 07bc 这个两个字符一下子写入，而是先写入一个 bc，然后在写入一个 07，所以我的插件才没捕获到程序的异常。如图 22 所示。

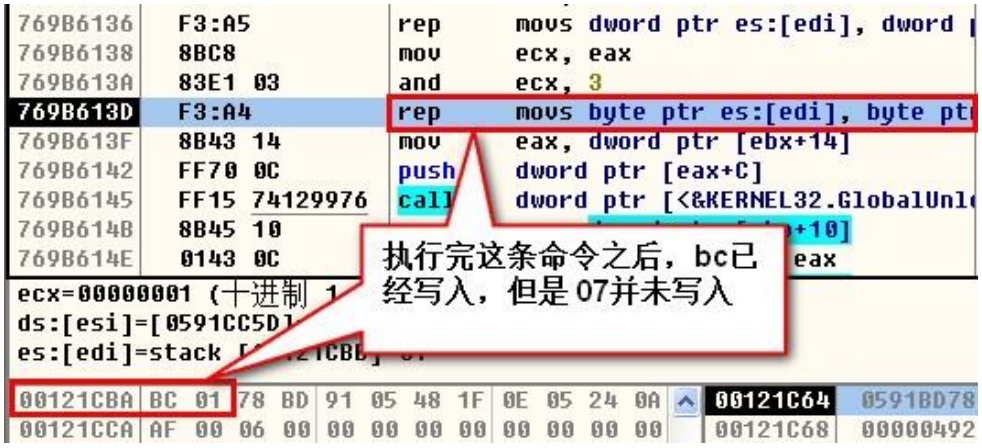

图 22

按下 F8 后 07 才得以写入，也是因为这条 rep 指令的缘故，才把两次写入分开了。

这条指令如下：

769B6136　　F3:A5　rep　　movs dword ptr es:[edi], dword ptr [esi]

我们可以看到，他将[esi]所指向的内存的值拷贝到了我们的[00121cba]处，那么我们就需要去查看影响[esi]的值的情况了。如图 23 所示。

图 23

查看 esi 的所指向的值是，感觉此片区域的值都似曾相识。哦~这里的值就是 poc 中第三个参数的值所拷贝过来的，如图 24 所示。

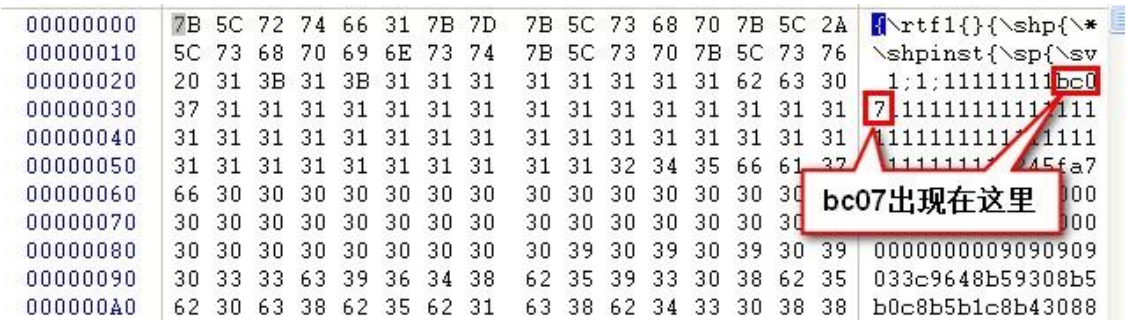

图 24

原来是这里构造好的值才影响了下面的全部。

至此，分析结束。