**CVE-2013-1347漏洞利用分析**

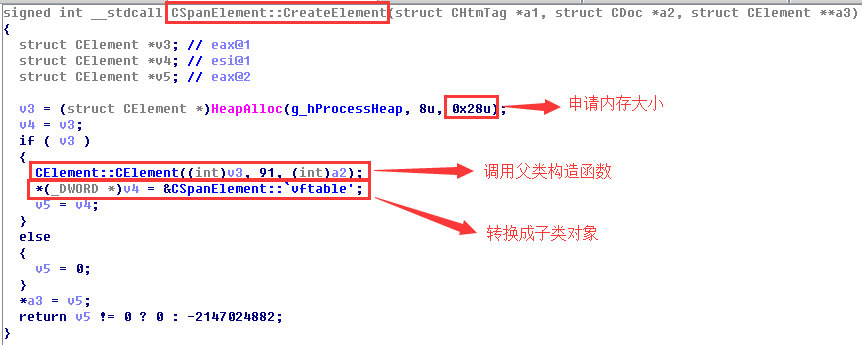
**0x00 调试环境**

|  |  |
| --- | --- |
| OS: | win7 x86 |
| IE | 8.0.7601.17514 |

**0x01 背景知识**

1. **HTML元素CElement**

IE中所有html元素继承自CElement，一般命名为CXXXElement，以<span>标签为例其创建过程：



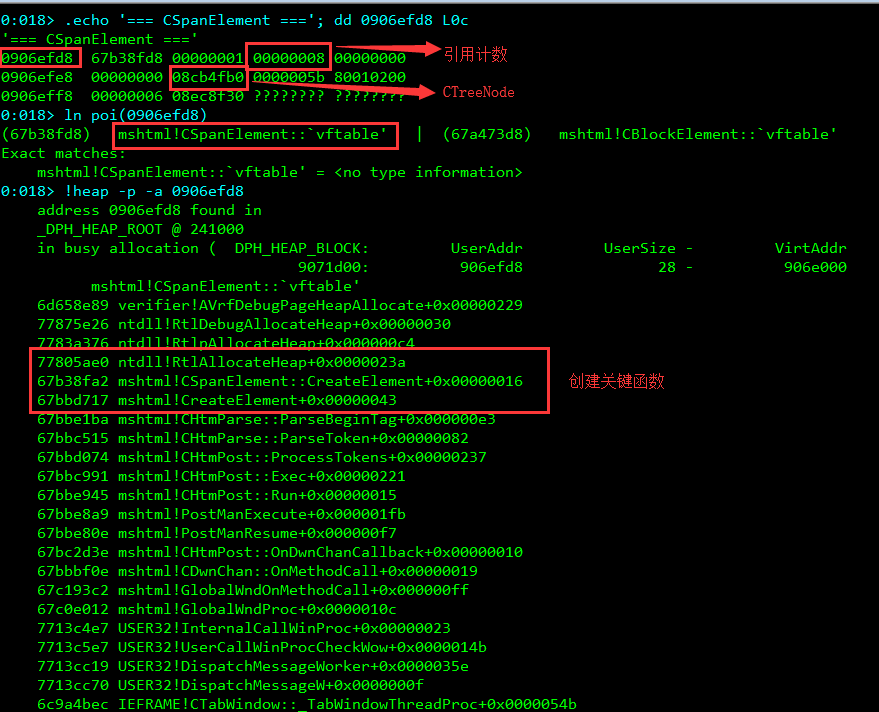
**已知数据结构：**

**CElement:**

**+0x00 vtable**

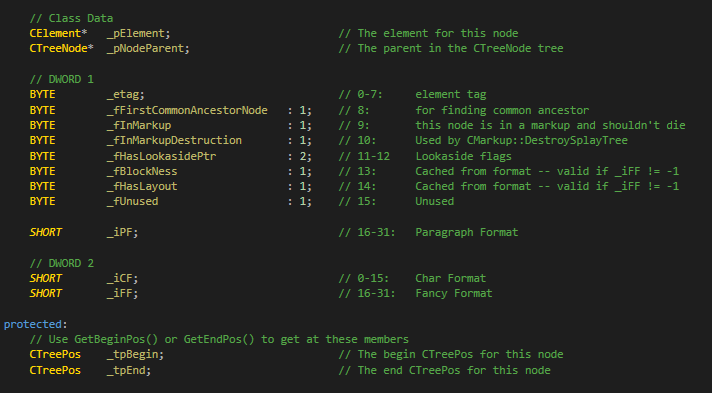
**+0x08 引用计数**

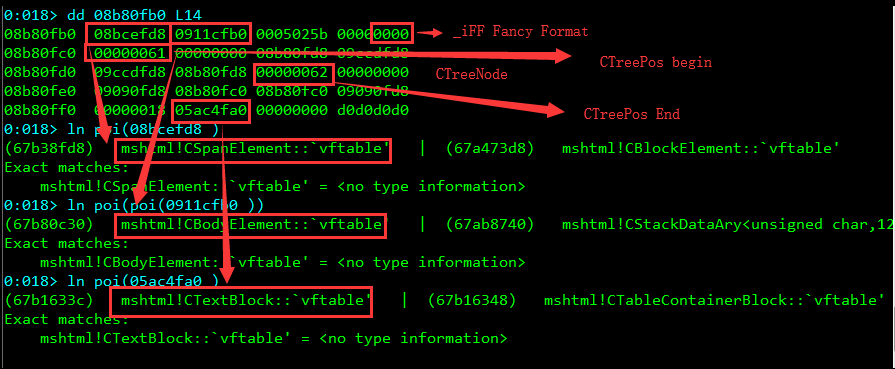
**+0x14 CTreeNode**



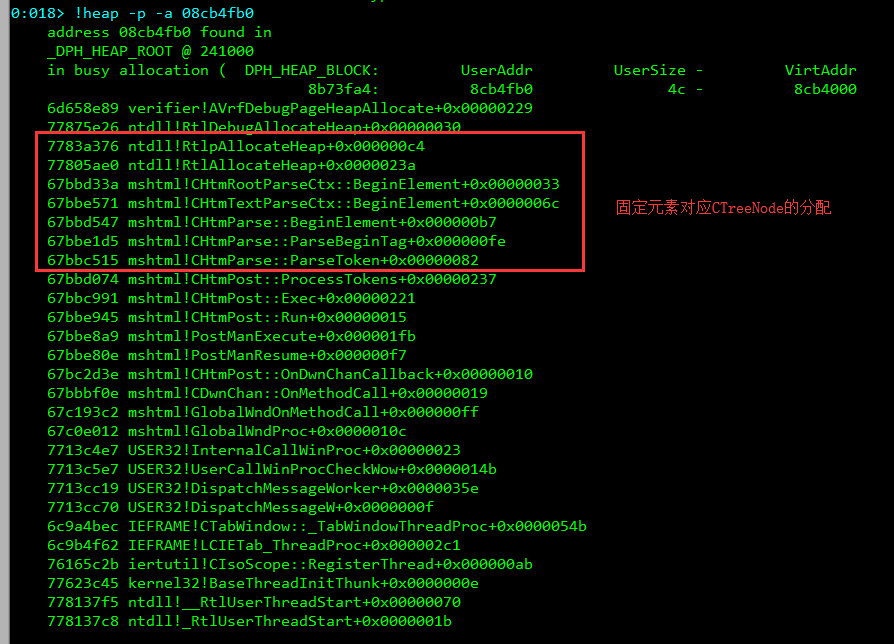
1. **DOM元素CTreeNode**

mshtml使用CTreeNode记录DOM树的数据结构，IE5源码中CTreeNode的定义：

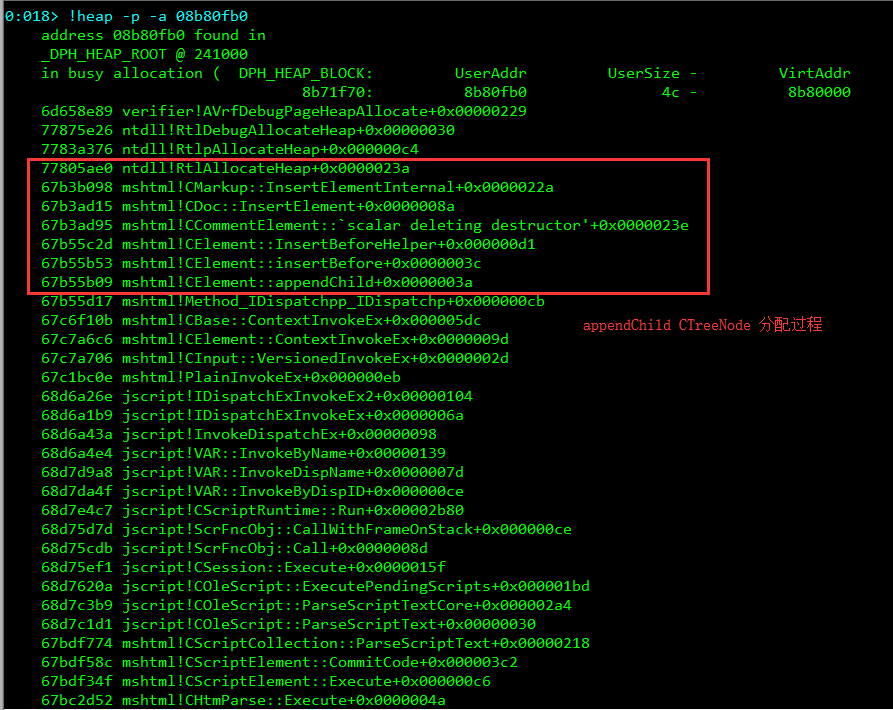




固定元素对应CTreeNode的分配过程：



动态生成元素（如appendChild）对应CTreeNode的分配过程：



1. **DOM位置元素CTreePos**

CTreeNode中包含了标记CTreeNode在DOM树位置的起始CTreePos（CTreeNode+0x10）和终止CTreePos(CTreeNode+0x28)。

已知数据结构：

**CTreePos:**

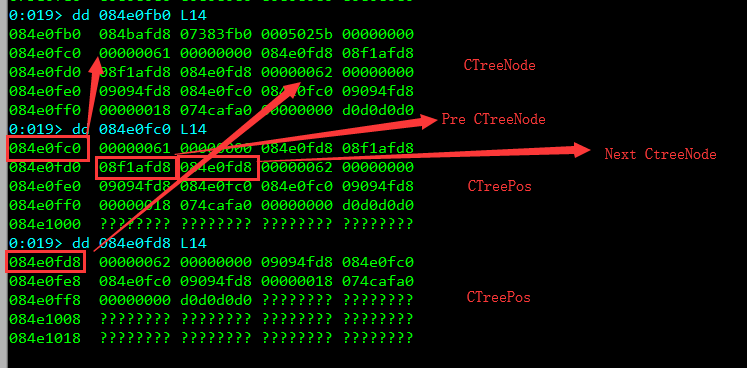
**+0x00 key**

**+0x10 上一个CTreePos对象**

**+0x14 下一个CTreePos对象**

**+0x18 引用计数**

**+0x20 对应的CTreeNode**



1. **渲染布局CLayoutBlock**

mshtml对DOM树进行渲染布局时使用CLayoutBlock来记录布局信息，对于内联元素会使用CLayoutBlock的继承类CTextBlock来记录，CTextBlock+0x58指向一个数据array\_element，用来记录各个内联元素在布局中的信息。

已知数据结构：

**CLayoutBlock:**

**+0x00 key**

**+0x10 上一个CTreePos对象**

**+0x14 下一个CTreePos对象**

**+0x18 引用计数**

**+0x20 对应的CTreeNode**

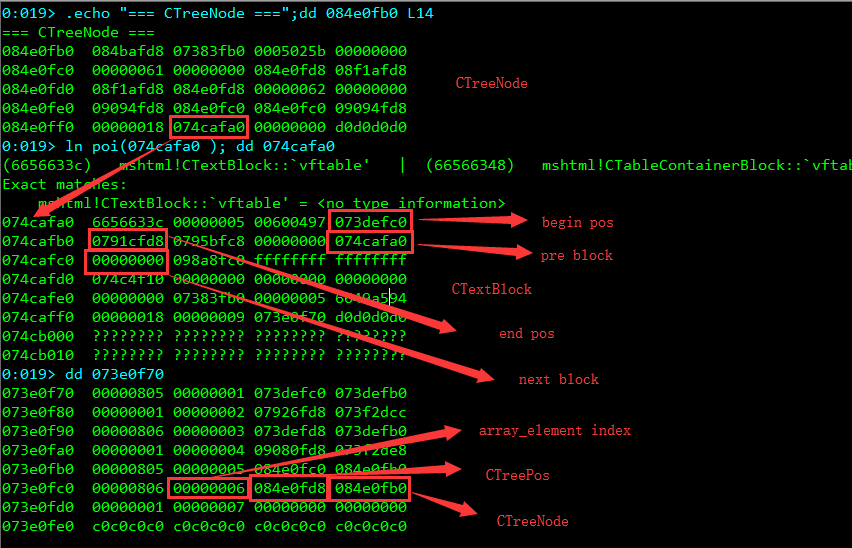
**array\_element**

**+0x00 flag**

**+0x04 数组中的索引**

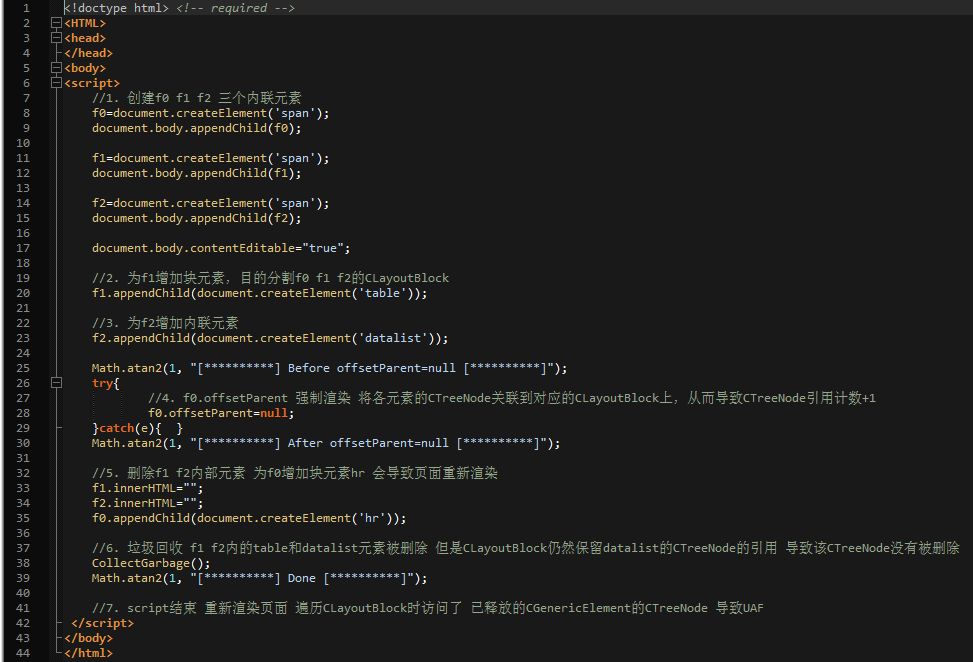
**+0x08 CTreePos**

**+0x0C CTreeNode**

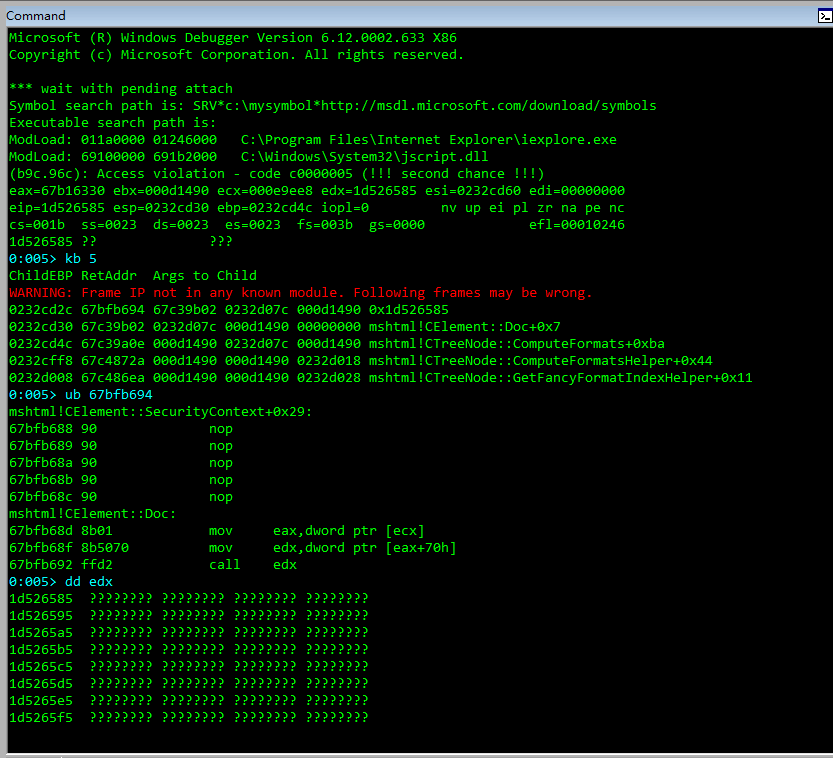


**0x02漏洞原理**

触发漏洞的POC如下：

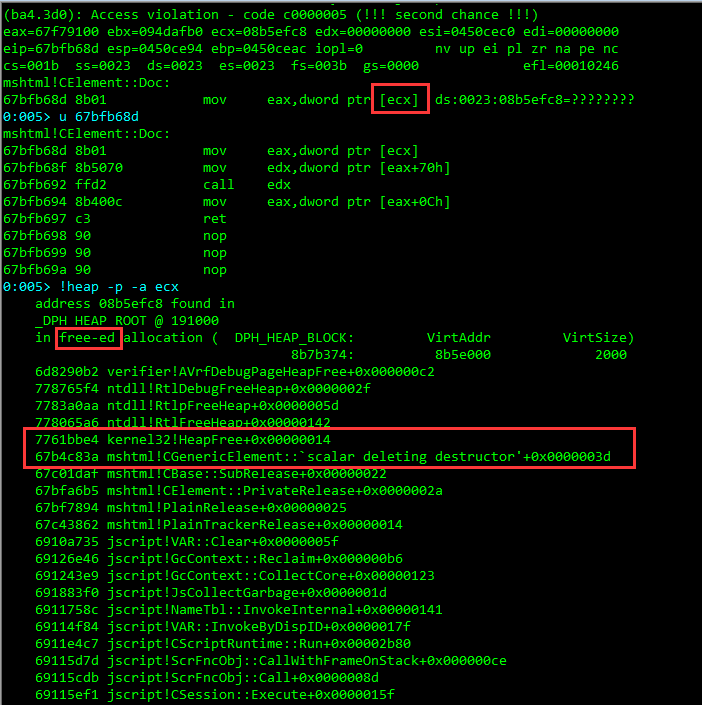


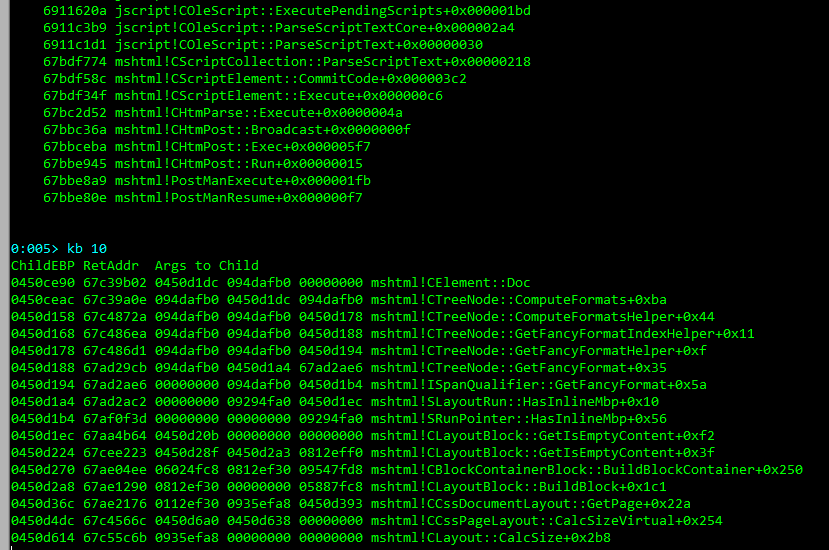
运行，发现IE Crash:



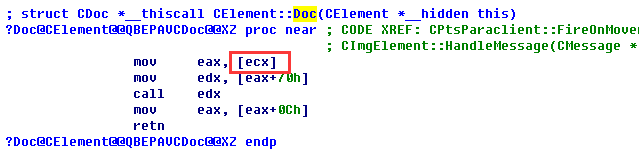
**1. 漏洞触发条件初步分析**

这里显然是一处虚函数调用，为了找到漏洞现场，开启hpa和ust，再次触发漏洞：





可以发现问题出现在CElement：：Doc的ecx中，ecx指向的内存已经在CGenericElement被释放了，用IDA查下CElement：：Doc的ecx：

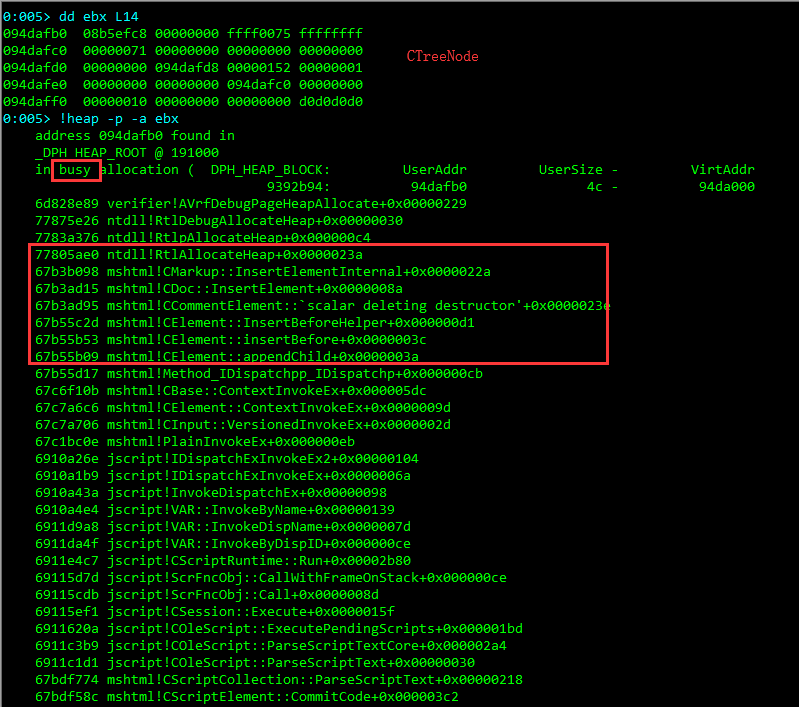


这里ecx就是CElement的this指针，并且通过ust可以猜到就是CGenericElement对象。下面根据调用栈找出是哪里引用了这个已经释放的CGenericElement对象：

**调用栈1：CTreeNode::ComputeFormats**

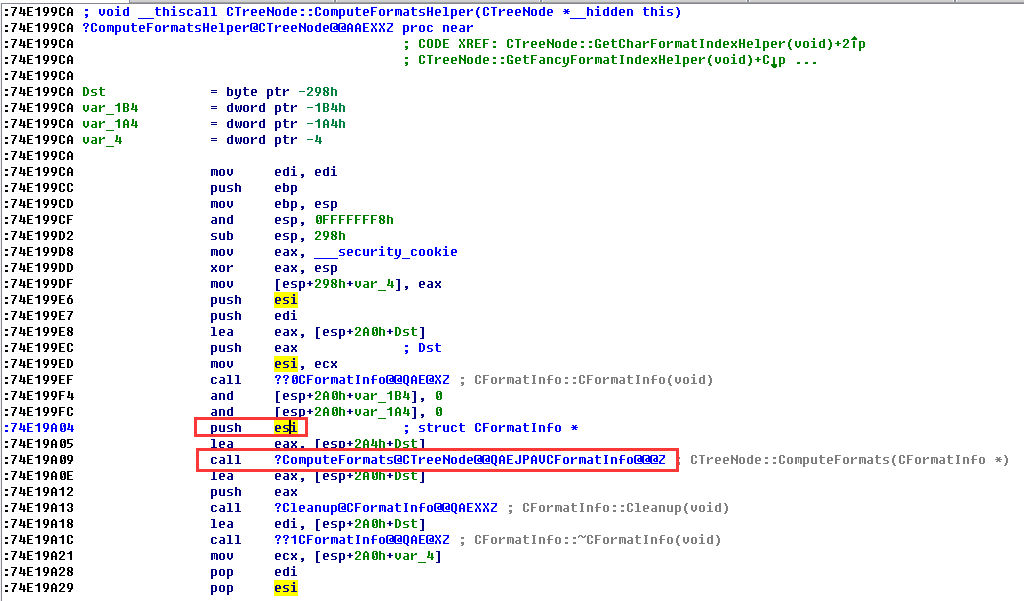


ecx=[ebx]，而ebx=[ebp+8]=this->CTreeNode，这和CTreeNode+0x00->CElement是一致的，先看下ebx的情况：



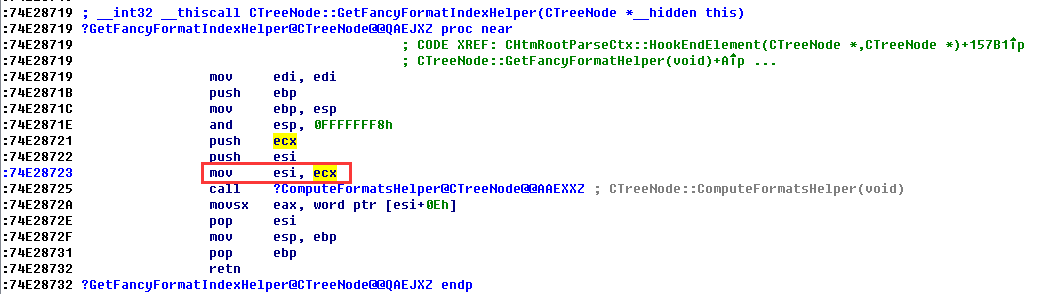
可以看到ebx指向的正是通过appednChild生成的CTreeNode，并且这块内存没有被释放，所以漏洞触发的原因是引用了CTreeNode但是CTreeNode对应的CGenericElement对象已经被释放了。继续跟踪观察是哪里引用CTreeNode：

**调用栈2：CTreeNode::ComputeFormatsHelper**



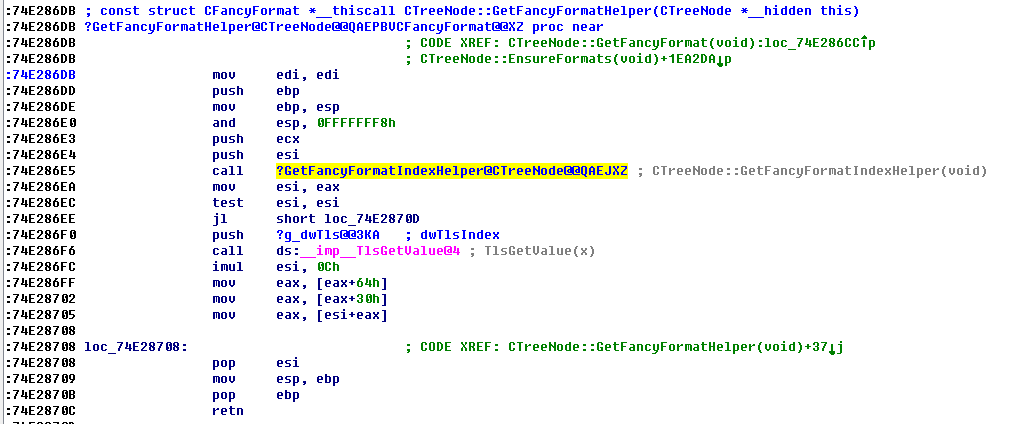
ebx=esi

**调用栈3：CTreeNode::GetFancyFormatIndexHelper**



esi=ecx

**调用栈4：CTreeNode::GetFancyFormatHelper**



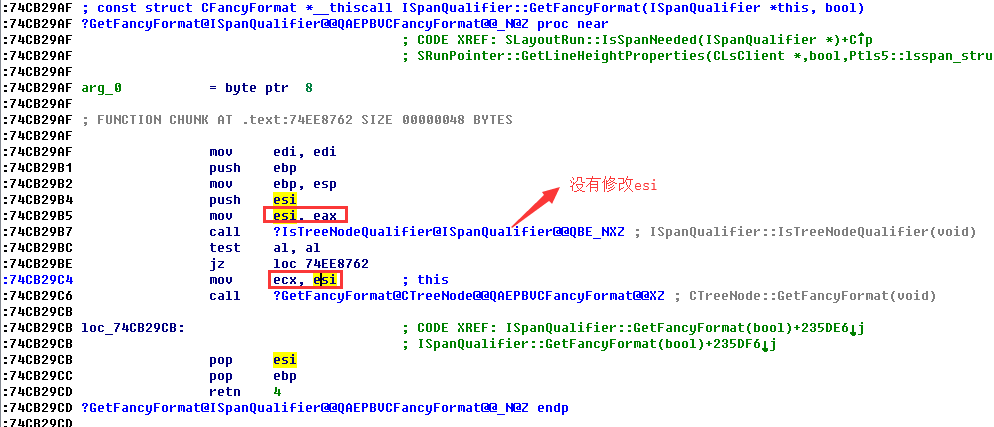
没有修改ecx

**调用栈5：CTreeNode::** **GetFancyFormat**



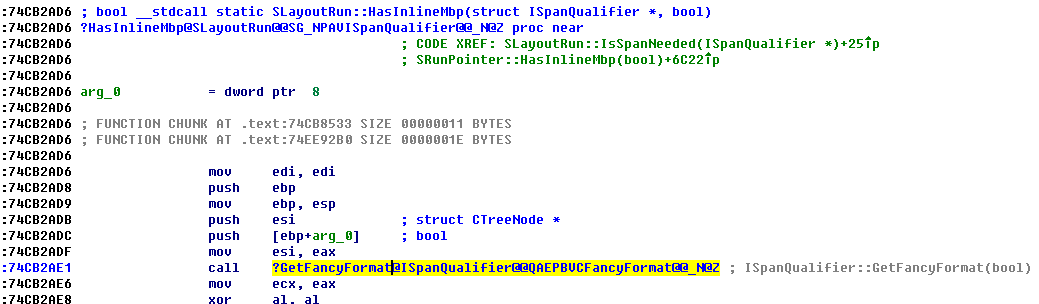
没有修改ecx

**调用栈6：ISpanQualifier::GetFancyFormat**



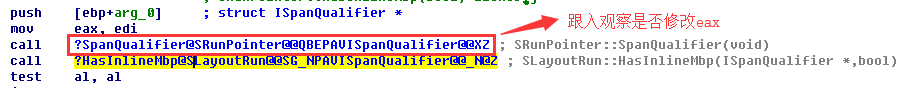
ecx=eax

**调用栈7：SLayoutRun::HasInlineMbps**

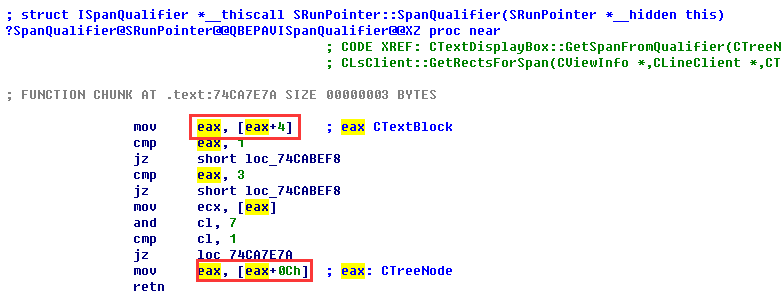


没有修改eax

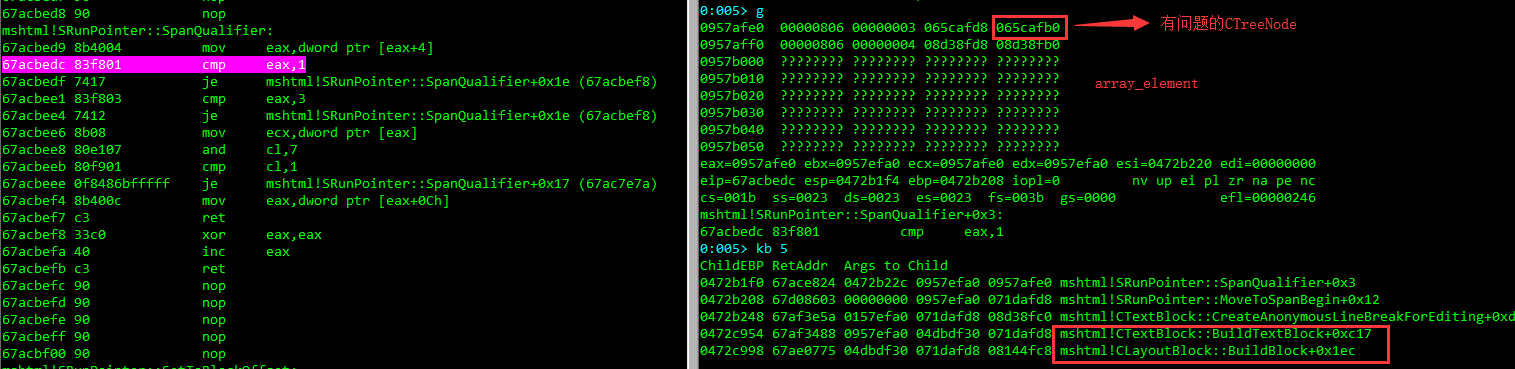
**调用栈8：SRunPointer::HasInlineMbp**

eax=edi，但是之前还有SpanQualifier@SRunPointer需要跟入观察是否对eax修改：

**SRunPointer::SpanQualifier**



这里eax=[[eax+0x4]+0xc]，可以在CollectGarbage()后对这个函数下条件断点（0x065cafb0是跟踪的CTreeNode），观察eax的来源，条件断点bp mshtml!SRunPointer::SpanQualifier+3 ".if(poi(eax+c)== 0x065cafb0){dd eax} .else{gc}"

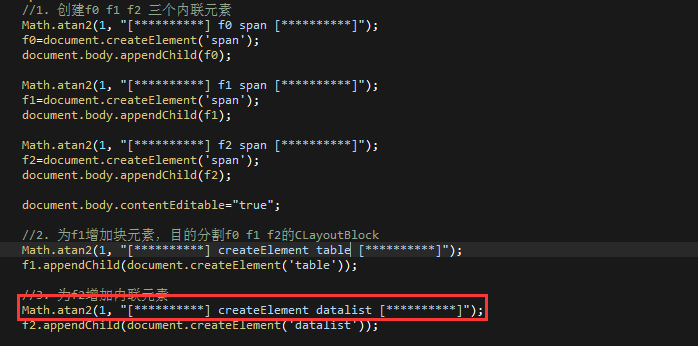


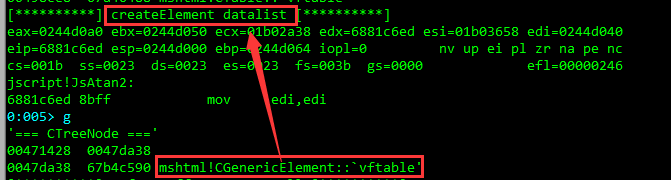
这里eax指向的就是CTextBlock+0x58处的array\_element，array\_element+0x0c则保存了CTreeNode，而通过调用栈可以猜测这里正是由于在mshtml在渲染DOM树生成CLayoutBlock对象时，引用了包含已释放CGenericElement元素的CTreeNode对象引起的UAF漏洞。

**2. UAF原因分析**

了解了漏洞触发的条件，接下来需要分析的是poc中哪里会造成UAF。

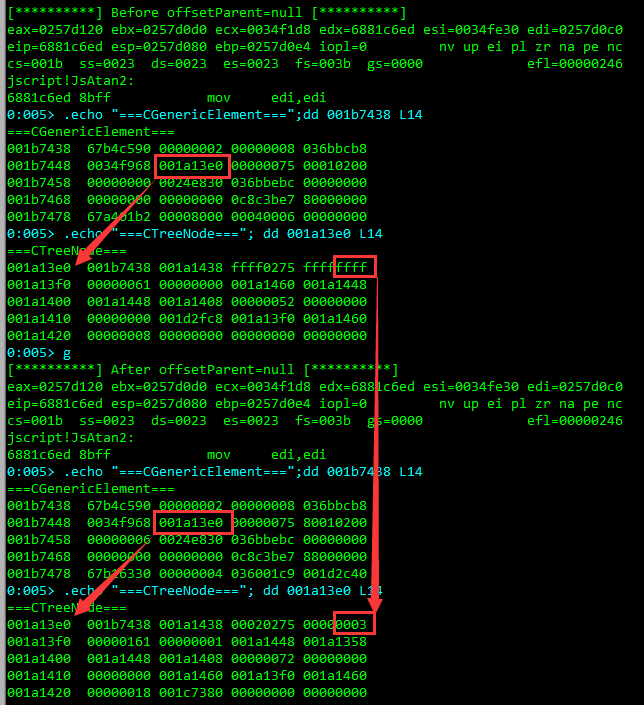
首先需要知道CGenericElement是哪个HTML元素创建的，对poc中每个html创建前加调试信息：



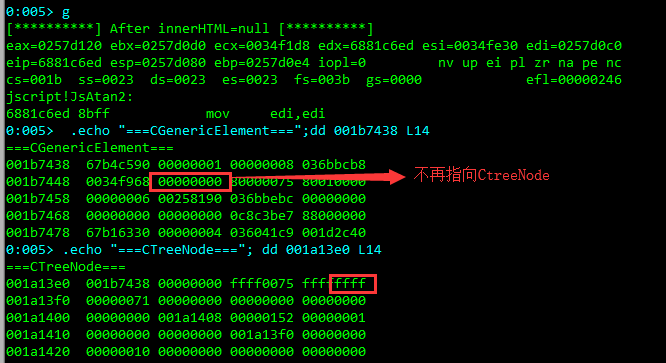


可以发现CGenericElement是datalist元素对应的HTML对象。接下来主要关心CGenericElement和其对应的CTreeNode的生成和释放情况。

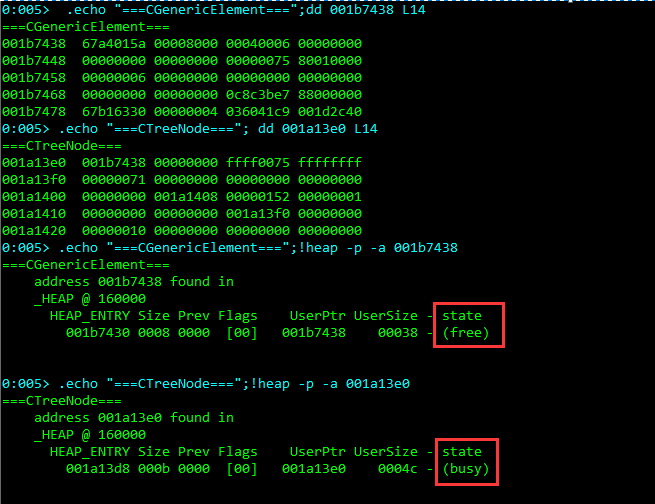
首先在**f0.offsetParent=null**会完成目前创建元素（f0-span f1-span-table f2-span-datalist）DOM树的构建（但是CLayoutBlock并没有生成，因为如果DOM树变化Layout立刻变化的话效率会比较低），观察**f0.offsetParent=null**前后CGenericElement和其对应的CTreeNode内存情况：



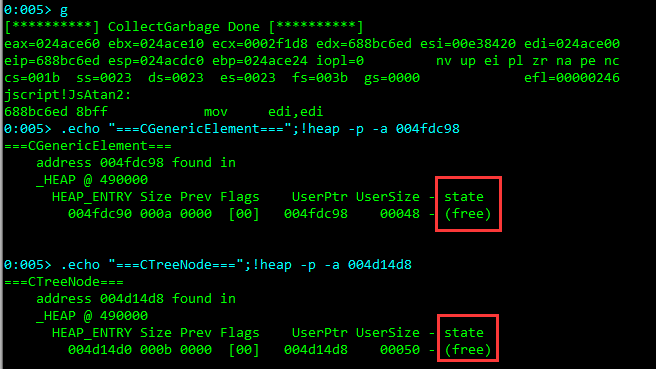
可以看到**f0.offsetParent=null**后CTreeNode+0xC (\_iCF)被修改成了03，继续执行到innerHTML=“”后：



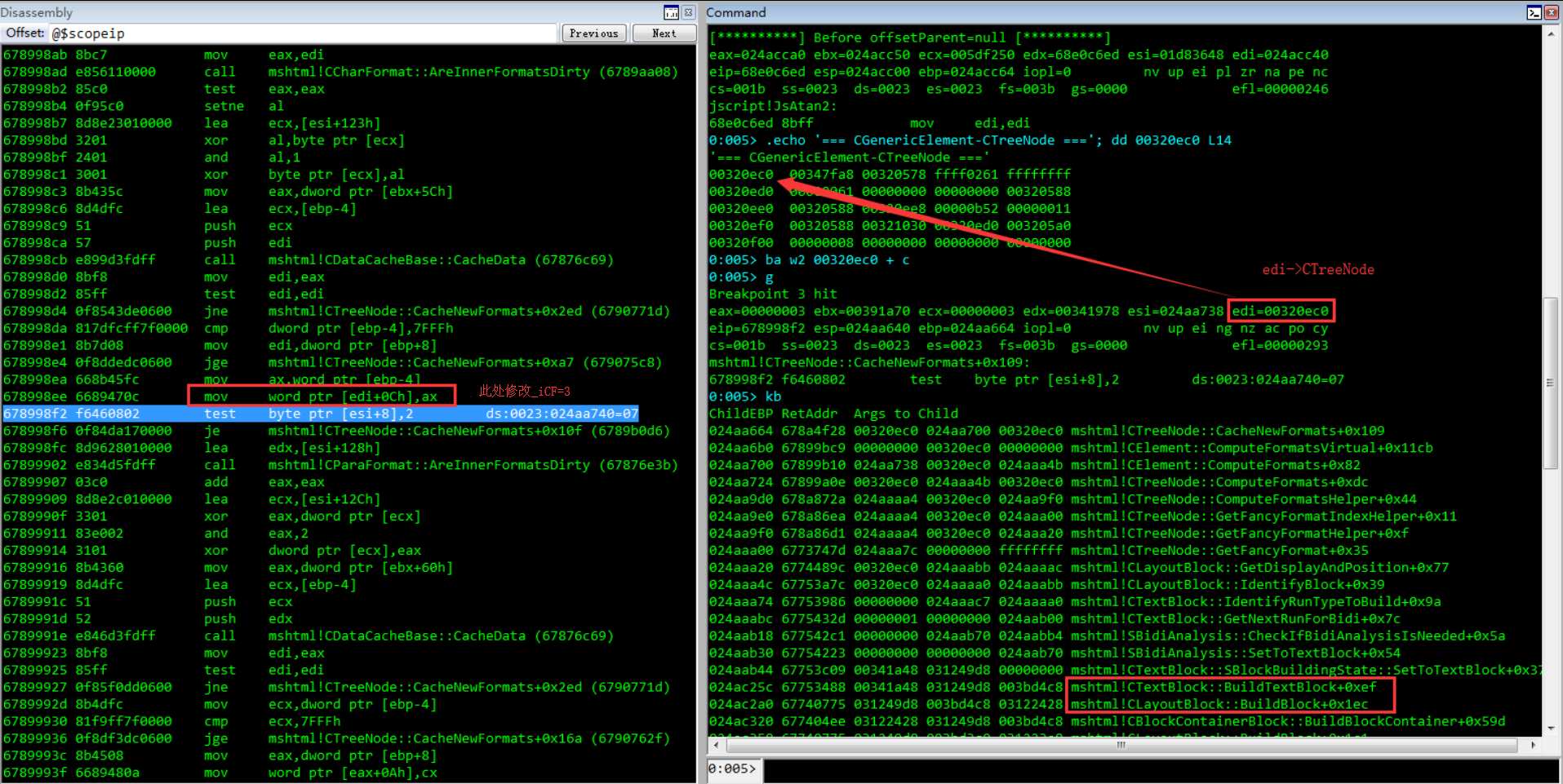
这时候已经将CGenericElement里的CTreeNode引用删除，也就是将CGenericElement从DOM树卸下来，同时CTreeNode在CGenericElement的引用计数也会减一，最终CollectGarbage后CGenericElement以及CTreeNode都应该被释放，CollectGarbage后的情况：



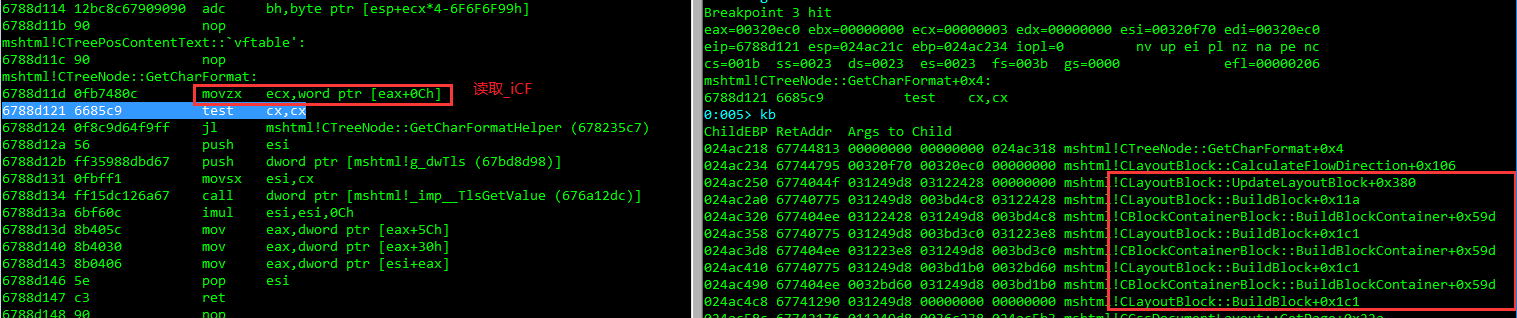
CGenericElement已经被释放，但是CTreeNode却没有按照预期被释放，所以造成UAF的原因可能是**f0.offsetParent=null，**将POC里这句话注释，再次观察CollectGarbage后的情况：



CGenericElement对象本身的释放是正常的，所以看似是CGenericElement的UAF漏洞，其根本原因是CGenericElement对应的CTreeNode对象在**f0.offsetParent=null**被某处引用导致没有被释放造成，前面观察到**f0.offsetParent=null**前后CTreeNode+0xC (\_iCF)被修改成了03，对此处下写断点，观察修改此处的函数：



可以看到**f0.offsetParent=null**会调用CLayoutBlock::BuildBlock，也就是会对DOM进行渲染，继续对CTreeNode+0xC下读断点：

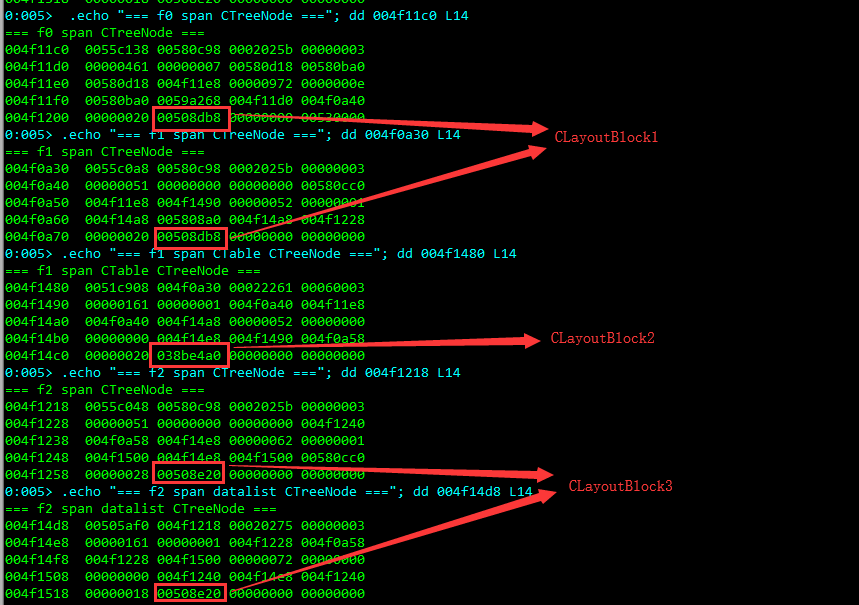


这里对CTreeNode的\_iCF进行判断，如果\_iCF<0则进入CTreeNode::GetCharFormatHelper，这里\_iCF=3>0则跳过该分支。跟入CTreeNode::GetCharFormatHelper会发现这里最终会调用CTreeNode::ComputeFormats重新计算节点，因此由于**f0.offsetParent=null**使得CTreeNode被误认为已经被渲染从而没有重新计算节点，最终导致在innerHTML=””后没有将该CTreeNode从CLayoutBlock卸载下来。

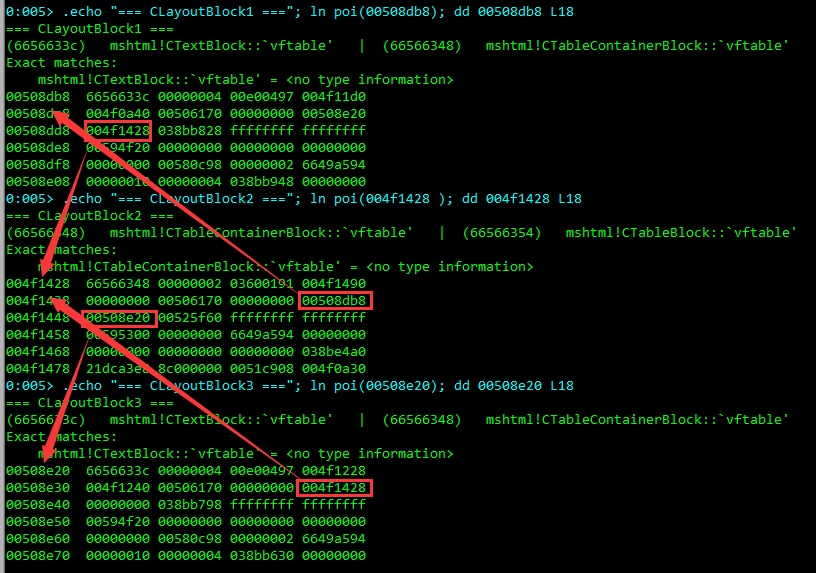
**3.漏洞本质**

通过前面的分析可以知道这个漏洞是由于在mshtml在渲染DOM树生成CLayoutBlock对象时，引用了包含已释放CGenericElement元素的CTreeNode对象引起的，而问题CTreeNode是由于**f0.offsetParent=null**后错误的渲染了该CTreeNode并且没有重新计算节点造成的，最后需要分析的是这个本来该释放的CTreeNode被错误的引用在了什么位置。

由于**f0.offsetParent=null**会导致页面渲染，从而生成CLayoutBlock，观察**f0.offsetParent=null**后的CLayoutBlock：



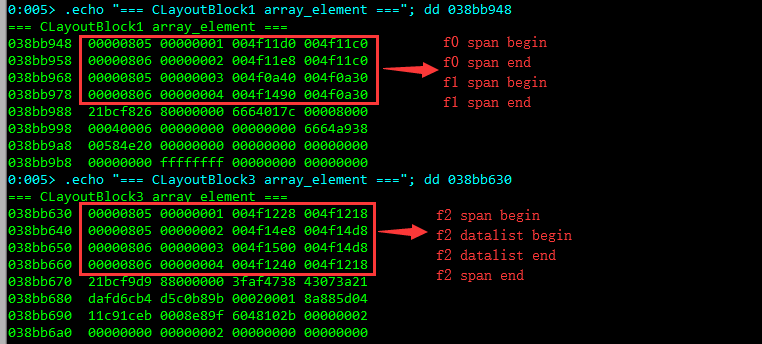
因为f1里增加了块元素Table导致本来都是内联元素的f1 f2 f3被分割成了3块：



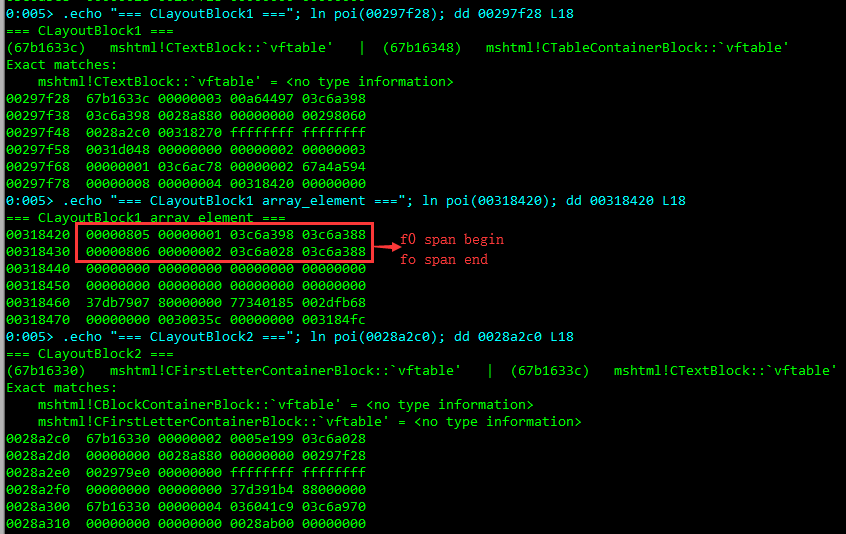
可以看到CLayoutBlock以双向链表的形式串联起来，渲染的时候会遍历这个链表。现在的CLayoutBlock链表是：

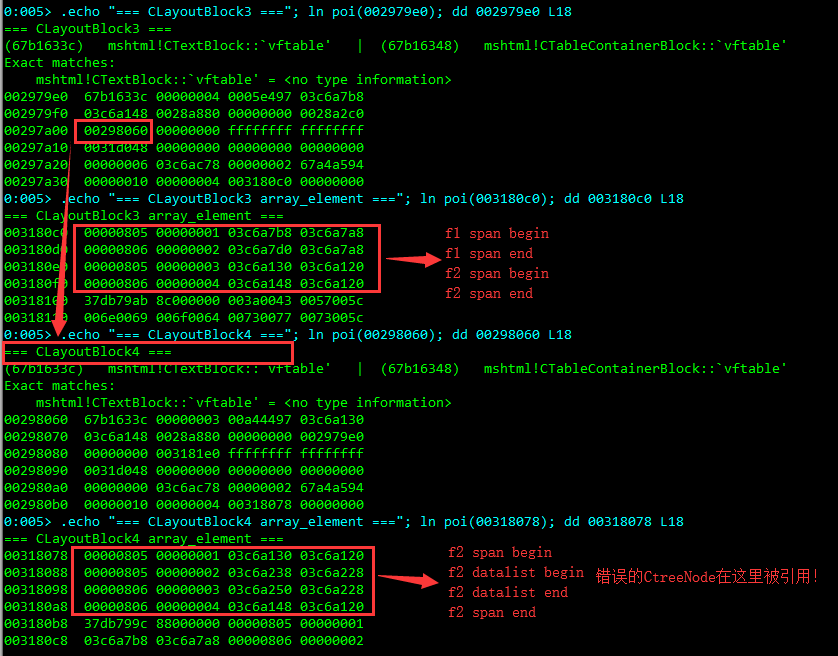
**f0 span + f1 span <=> f1 child table <=> f2 span + f2 child datalist**

因为CLayoutBlock1和CLayoutBlock3都是CTextBlock可以顺便看下array\_element是否符合预期：



CollectGarbage后，页面重新渲染，由于此时f1的table和f2的datalist已经被删除，f1和f2都只剩下内联元素span，而f0新增加了块元素hr，最终仍会被分成3个CLayoutBlock：





可以看到f1 f2合并后的CLayoutBlock3后面还链接着一个CLayoutBlock4这个block就包含了应该被释放的CTreeNode!从而最后页面渲染遍历CLayoutBlock链表的时候引用了指向已释放的CGenericElement的CTreeNode最终导致UAF漏洞。